

[illegible]

A vertical strip of a book cover featuring a repeating pattern of stylized, rounded, light-colored shapes (possibly representing letters or abstract forms) arranged in a grid-like fashion against a dark background. The pattern is consistent across the entire strip.

Компьютеры

**THE
McGRAW-HILL
COMPUTER
HANDBOOK**

**Editor in Chief
Harry Helms**

**Overview by
Adam Osborne**

**Foreword by
Thomas C. Bartee**

**McGraw-Hill Book Company
New York St. Louis San Francisco Auckland
Bogotá Hamburg Johannesburg London Madrid
Mexico Montreal New Delhi Panama Paris
São Paulo Singapore Sydney Tokyo Toronto
1983**

КОМПЬЮТЕРЫ

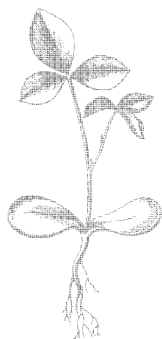
СПРАВОЧНОЕ РУКОВОДСТВО

В ТРЕХ ТОМАХ

2

Под редакцией Г. Хелмса

Перевод с английского
канд. физ.-мат. наук М. Г. Фуругяна
под редакцией
канд. техн. наук В. В. Василькова



МОСКВА «МИР»
1986

ББК 32.97
К 63
УДК 681.3

Гиа С., Таккер А. (мл.), Уидерхолд Д., Хелмс Г.,
Эрикссон Д.

Компьютеры: Справочное руководство. В 3-х т. Т. 2. Пер.
К 63 с англ./Под ред. Г. Хелмса — М.: Мир, 1986.— 440 с., ил.

Во втором томе рассматриваются принципы организации программного обеспечения на базе языка ассемблера. Описаны алгоритмические языки Бейсик, Кобол, Фортран, Паскаль и ПЛ/1. Излагаются принципы построения баз данных и организации файловых систем.

Для инженеров, имеющих дело с вычислительной техникой, и студентов соответствующих специальностей вузов.

К $\frac{2405000000-122}{041(01)-86}$ 173-86, ч. 1

ББК 32.97
6Ф7

Редакция литературы по информатике и электронике

Copyright © 1983 McGraw-Hill, Inc.
© перевод на русский язык, «Мир»,
1986.

ПРЕДИСЛОВИЕ РЕДАКТОРА ПЕРЕВОДА

Второй том справочного руководства по компьютерам содержит сведения, необходимые современному разработчику программного обеспечения.

Основное внимание уделено языкам программирования, причем приводятся описания не «вообще» языков, а их стандартных версий. Если же стандарт на некоторый язык отсутствует или не имеет смысла (например, для ассемблера), то при описании языка авторы стараются привести наиболее характерные общие решения и одновременно акцентируют внимание на специфике реализации для распространенных ЭВМ.

Особенности языков программирования и принципы составления программ с их использованием излагаются таким образом, что читатель, незнакомый с ними, быстро усваивает основные идеи языка и технику применения. Прилагаемые примеры программ облегчают понимание языка.

Ограниченный объем книги не позволил дать исчерпывающего описания всех приводимых языков: ассемблеров и таких языков высокого уровня, как Бейсик, Фортран-77, ПЛ/1, Паскаль, Кобол. Однако содержащиеся в ней сведения достаточны при обучении программированию. Книга может быть использована и как справочное пособие для практиков.

Особый интерес представляет глава, описывающая базы данных. В отличие от ранее изданных на русском языке монографий, посвященных в основном вопросам концептуального и логического проектирования баз данных, в настоящей книге подробно и достаточно представительно рассматриваются вопросы организации данных на внешних устройствах, методы организации доступа к файлам баз данных и вопросы определения характеристик функционирования соответствующих файловых систем.

При подготовке этого тома к изданию на русском языке мы испытывали определенные трудности — его девять глав написаны пятью различными специалистами. Мы надеемся, что нам

удалось добиться единообразия в представлении материала и стиле изложения. При переводе был устранен ряд замеченных опечаток.

Принимая во внимание широту, конкретность и практическую направленность материала, можно надеяться, что данный том будет полезен широкой аудитории читателей: от студентов, изучающих технологию программирования и технику конструирования систем управления базами данных, до специалистов в этих вопросах.

В. В. Васильков

ЯЗЫК АССЕМБЛЕРА И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

С. Гуа

11.1. ВВЕДЕНИЕ

Задание, которое нормально выполняется на сложной системе, может быть выполнено и на простейшей вычислительной машине с минимальными системными средствами при условии, что имеется требуемый объем памяти и достаточное количество машинного времени. Однако значительные затраты времени на подготовку программ, предназначенных для простейших систем, а также трудности проверки таких программ делают эту задачу чрезмерно дорогостоящей. Цель системы заключается не только в том, чтобы сделать возможным быстрое выполнение программ, но и в том, чтобы сделать возможным их быстрое написание. Часто отмечают, что ЭВМ не может сделать того, чего нельзя выполнить вручную, и что поэтому она не может предоставить человеку возможностей, которых ранее у него не было. Заметим, что и на автомобиле нельзя доехать до места, до которого нельзя дойти пешком, но он позволяет человеку посетить гораздо больше мест и намного быстрее. Автомобиль лишь в 10 раз быстрее человека, ЭВМ приблизительно в 10^9 раз быстрее ручного калькулятора. Достижения в области автоматического программирования и операционных систем за последние 25 лет позволили увеличить скорость написания программ в 10—100 раз. Это не только дает возможность пользователю писать программы быстрее и с меньшими затратами, но и позволяет решать их за приемлемое время.

Назначение программного обеспечения вычислительной системы двояко: упростить процедуру подготовки пользователями задания для вычислительной машины, а также облегчить процесс его прогона и отладки. На машинном уровне ЭВМ выполняет команды, составленные из цифр. Язык ассемблера позволяет не только упростить чтение и написание программ, но и создает также дополнительные языковые средства, которые

улучшают пользование вычислительной машиной. Эти дополнительные средства обеспечиваются операционной системой. Для обычного пользователя не имеет значения, являются ли эти дополнительные команды частью аппаратных средств или они относятся к программному обеспечению. Важно предоставить ему наиболее мощную систему программирования высокого уровня. В этой главе будут описаны дополнительные средства, необходимые для ввода и прогона программ.

11.2. НЕЗАГРУЖЕННАЯ МАШИНА. НАЧАЛЬНАЯ ЗАГРУЗКА ПРОГРАММЫ

Основные аппаратные средства обеспечивают выполнение последовательности команд, находящихся в памяти ЭВМ. Задача пользователя и системы заключается в том, чтобы ввести программу в память и начать ее выполнение с нужной команды. Устройства ввода-вывода обеспечивают считывание данных в память, поэтому именно аппаратные средства должны использоваться для загрузки программы. Внутренним представлением программы, числовых данных и другой информации являются «слова» (или группы слов), составленные из двоичных цифр. Команды ввода обеспечивают считывание внешнего представления этих битов в память. Поэтому если программа, которая должна быть выполнена, представлена в двоичном виде в точности так, как она должна находиться в памяти ЭВМ, то для ее загрузки могут быть использованы команды ввода. Однако сами команды ввода перед своим выполнением тоже должны быть загружены в память. Для осуществления этого обычно используется **последовательность команд начальной загрузки**, которую называют также командами **начальной загрузки программы** (НЗП), или **последовательностью команд первичной загрузки**.

Последовательность команд начальной загрузки приводится к выполнению нажатием кнопки на главном пульте управления ЭВМ. Нажатие кнопки инициирует выполнение одного небольшого сегмента команд, который запаян в вычислительной машине и содержит программу старта процесса загрузки. Немногие программисты сталкиваются с необходимостью изучать эту программу, поскольку она запаивается изготовителем и обычно не меняется. (Исключение составляют некоторые мини- и микро-ЭВМ, поскольку в них программа начальной загрузки не запаяна, а должна быть введена программистом с помощью консоли, с тем чтобы записать необходимую комбинацию двоичных разрядов в определенные ячейки памяти.) Последовательность команд начальной загрузки является началом сложного процесса загрузки, служащего для вызова системных программ, которые в свою очередь загружают программы пользователя и

принимают участие в их выполнении. Большую часть времени эти дополнительные системные программы находятся в оперативной памяти; начальная загрузка программ используется только при остановках, вызванных такими нечастыми событиями, как отключение электроэнергии, сбой основных аппаратных средств или ошибка в программном обеспечении. Перейдем к краткому рассмотрению событий, возникающих после нажатия кнопки начальной загрузки программы. Каждый из описанных ниже процессов раскрывает важную сторону операционных систем и показывает их историческое развитие.

Поскольку последовательность команд НЗП постоянно хранится в памяти (или, что еще хуже, вводится вручную), она должна быть очень короткой. Для некоторых вычислительных машин она состоит из одного слова, но обычно из нескольких. Предположим, что команда ввода, имеющаяся в ЭВМ, считывает одну запись (например, образ перфокарты с 80 колонками) в последовательный набор ячеек памяти. Нажатие кнопки НЗП переведет вычислительную машину в состояние, в котором она находилась бы, если бы такая команда ввода была только что выбрана из памяти и готова к выполнению. Предположим также, что команда, инициируемая нажатием кнопки НЗП, предписывает считать содержимое записи в ячейки с 0 по 19 (предположим, что для заполнения одного слова достаточно четырех знаков). Если нажатие кнопки НЗП также обнуляет счетчик управления, то вычислительная машина будет выполнять команду ввода, заполняя ячейки с 0 по 19, и затем выполнять команду, содержащуюся в ячейке 0. Поскольку ячейка 0 была только что заполнена командой ввода, управление вычислительной машиной будет осуществлять только что считанная программа. Хотя количество слов в этой программе ограничено двадцатью, их вполне достаточно для того, чтобы прочитать дополнительную программу. (Например, если считанная программа содержит 20 команд ввода, то с ее помощью можно заполнить еще 400 ячеек. Этот процесс называется **первичной загрузкой**, поскольку программа вводит сама себя посредством «своих команд начальной загрузки».) Если начальная запись считывается с диска или ленты, то нет необходимости ограничивать ее 20 словами; вся программа может быть считана одной командой чтения, находящейся в последовательности команд НЗП.

После ввода дополнительных записей в память может быть считана дополнительная программа. Некоторые ранние ЭВМ, а также простейшие современные микроЭВМ используют этот процесс для непосредственной загрузки программ пользователя, но при этом требуется, чтобы программа была уже преобразована в окончательную двоичную форму. Однако такой процесс, даже если он и не содержит ошибок, будет иметь беспорядочный

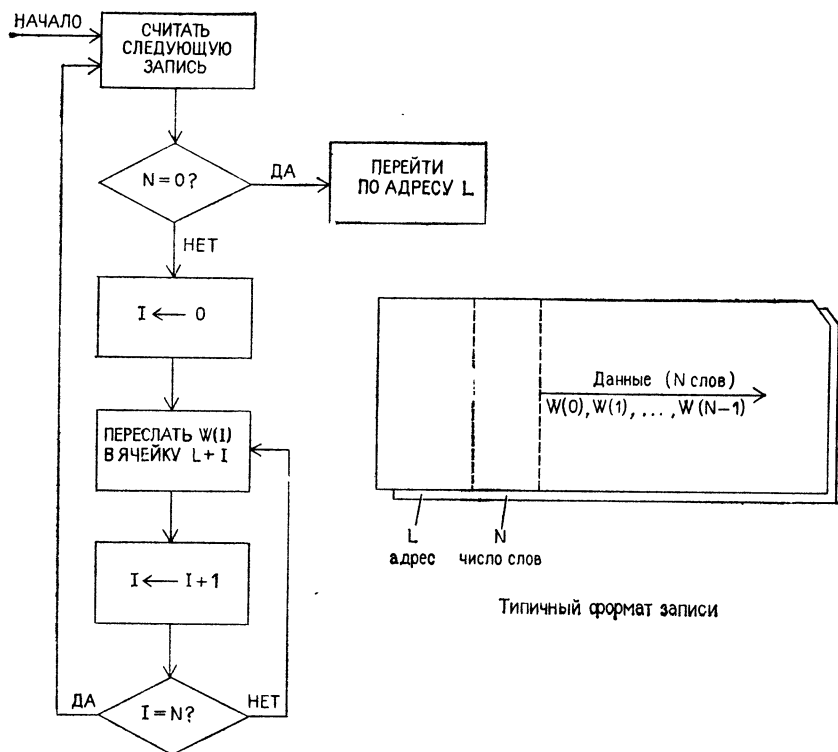


Рис. 11.1. Абсолютный загрузчик и формат записи.

характер. Чтобы избежать этого, для ввода **абсолютного двоичного кода** удобно использовать стандартный формат абсолютного двоичного кода программы. Обычно каждая запись программы в абсолютной двоичной форме содержит несколько последовательных слов, адрес первого слова и число слов. Если записи находятся на образцах перфокарт, то формат может быть таким, как показано на рис. 11.1. Поскольку длина программы заранее не известна, последняя запись должна содержать соответствующий указатель. Для этого счетчик слов N во второй группе четырех знаков полагается равным нулю. Ячейка L в первой группе четырех знаков содержит адрес начала загружаемой программы.

Программа, считывающая данные в таком формате, называется **абсолютным загрузчиком**. Обычно она считывается последовательностью команд НЗП. Если эта программа небольшая (что, конечно, имеет место для простых ЭВМ), то она, ве-

роятно, может быть считана одной командой ввода из последовательности команд НЗП. Программа 11.1 описывает работу абсолютного загрузчика для вычислительной машины с командой ввода, загружающей 20 слов.

Программа 11.1. Абсолютный загрузчик

```
do forever
ВВОД следующей перфокарты в ячейки с A по A+19
  X1←C(A+1)
  X2←C(A)
  if X1=0 then go to Адрес в X2 endif
  do for J=0 to X1-1
    C(X2+J)←C(A+J)
  enddo
enddo
```

Формат входных данных предполагается таким, как показано на рис. 11.1, где первые два набора четырех знаков содержат адрес размещения и счетчик слов соответственно, а остальные 18 слов принадлежат последующим 18 группам из четырех знаков. **X1** и **X2** — индексные регистры. **C(A)** означает содержимое ячейки **A**, а **C(X2 + J)** — содержимое ячейки, адрес которой равен содержимому **X2** плюс значение **J**. Поскольку значение **J** наверняка будет содержаться в другом индексном регистре, то вероятно, что для некоторого выполнения этой программы потребуется, чтобы возрастание **X1** было согласованным с **J**, что позволит избежать двойной индексации для **C(X1 + J)**.

11.3. АССЕМБЛЕР

При использовании абсолютного загрузчика требуется, чтобы пользователь составлял законченную программу, определяя адреса всех команд и данных и записывая все команды и данные, которые должны быть введены, в двоичной форме. Указание таких подробностей не только утомительно, но и не нужно при написании программы. Для пользователя не имеет значения, где размещается переменная, которая обозначена через **X**, — в ячейке 1026 или 10260. На выполнение программы не влияет конкретное значение адреса переменной, так как все ссылки на эту переменную используют один и тот же адрес. Поэтому желательно позволить пользователю называть переменную **X** своим именем, а не требовать введения абсолютной адресации. Другими словами, мы хотим работать с символическими адресами. Аналогично при написании команд желательно пользоваться их mnemonic именами, как, например, **ADD**.

Аппаратные средства «способны понимать» только двоичный код. Они не воспринимают ни символических адресов, ни мнемонических имен команд. (Нельзя сказать, что аппаратные средства вообще нельзя спроектировать таким образом, чтобы они воспринимали такие символы, но существующие устройства их не воспринимают.) Для преодоления этих трудностей мы должны перейти от простого абсолютного загрузчика, описанного в предыдущем разделе, к программе, которая переводит исходный язык мнемонических имен и символов (язык ассемблера) в объектный язык двоичных команд и адресов. Такая транслирующая программа называется ассемблером. Отметим, что процесс загрузки становится более сложным, однако работа пользователя упрощается. Для ввода программы на языке ассемблера в ЭВМ надо вначале выполнить последовательность команд начальной загрузки программы, с тем чтобы ввести абсолютный загрузчик. С помощью абсолютного загрузчика можно ввести ассемблер, который затем используется для трансляции программы пользователя.

Двухпроходный ассемблер

Для трансляции программы, написанной на языке ассемблера, ассемблер должен заменить мнемоническое имя каждой операции эквивалентным двоичным кодом, а каждый символический адрес — его числовым эквивалентом. Например, если ассемблер читает последовательность символов

	LOAD	A
LOOP	STORE	B
	ADD	C
	BMI	LOOP
	STOP	
A	
B	
C	

то он должен заменить мнемонические имена операций **LOAD**, **STORE**, **ADD**, **BMI** и **STOP** некоторым двоичным эквивалентом, имеющимся у них на вычислительной машине, на которой ассемблируется программа. Кроме того, ассемблер должен заменить символы **A**, **B**, **C** и **LOOP** их эквивалентными адресами. Если программа должна размещаться, начиная с ячейки 100, а операции **LOAD**, **STORE**, **ADD**, **BMI** и **STOP** имеют коды 52, 53, 48, 32 и 03 соответственно, то ассемблер должен протранслировать эту программу в последовательность чисел, указанную в табл. 11.1 (Для простоты используются десятичные числа.)

Таблица 11.1

Ячейка	Содержимое	Входной текст
100	52 105	LOAD A
101	53 106	STORE B
102	48 107	ADD C
103	32 101	BMI LOOP
104	03 000	STOP
105	
106	
107	

Замену мнемонических имен операций можно легко осуществить, если хранить таблицу всех мнемонических обозначений команд. При чтении мнемонического имени с помощью этой таблицы может быть найден эквивалентный двоичный код. Для выполнения аналогичной процедуры с символическими адресами также необходимо построить таблицу эквивалентности. Коды операций известны заранее, однако связь между символами, используемыми программистом, и адресами не известна до того, как программа будет закодирована и прочитана вычислительной машиной. В рассмотренном примере ассемблер до чтения программы не может определить, что символ **LOOP** соответствует ячейке 101. Таким образом, с каждым символом, определенным программистом, необходимо выполнить две операции. Первая заключается в построении таблицы всех символов и назначении им числовых значений, вторая — в выборе этих значений и замене ими символов. Эти две операции обычно напоминают два сканирования, или **прохода**, по тексту исходной программы. При первом проходе в памяти машины строится **таблица символов**, задающая для каждого символа эквивалентный ему адрес. Ниже дается таблица символов, полученная для рассмотренного нами примера после первого прохода.

Символ	Значение
LOOP	101
A	105
B	106
C	107

При втором проходе производится замена, с помощью которой исходный код переводится в двоичный. Отметим, что указанные

два прохода ассемблера выполняются до того, как программа пользователя становится готовой к загрузке в память и выполнению. То есть трансляция выполняется во **время ассемблирования** (во время работы ассемблера). После завершения трансляции программы пользователя можно перейти к ее выполнению, или **прогону**.

При первом проходе следует использовать правила назначения адресов различным меткам. Основное правило заключается в том, что команды должны нумероваться в той последовательности, в которой они считываются. Если каждую команду записать в ячейку, следующую за ячейкой с предыдущей командой, то такой порядок записи определит адреса всех команд. Вычисление адресов производится ассемблером с помощью **счетчика адресов**.

Сначала мы рассмотрим общую форму ассемблера, а затем кратко опишем некоторые его характерные особенности в четырех различных системах. Читатель должен понимать, что различных версий ассемблера существует столько, сколько имеется различных вычислительных систем, и даже еще больше, так как некоторые системы располагают более чем одним ассемблером. В ассемблере должны быть учтены и использованы особенности конкретной вычислительной машины. Однако, несмотря на это, многие черты различных версий ассемблера очень похожи. Поэтому при изучении ассемблера следует сконцентрировать внимание на основных его элементах, а не на деталях, присущих какой-то одной системе.

Внешнее представление команд

Программист должен иметь возможность указывать имя команды и адреса ее операндов и определять, соответствует ли ячейке, содержащей эту команду, некоторый символический адрес. Под каждый из этих элементов отводится **поле** во входной записи. (Предполагается, что входная запись является строкой или образом перфокарты. Она может также поступать с файла на диске или с ленты.) Во многих ранних версиях ассемблера использовался **фиксированный формат** входных данных, в котором каждое поле занимает фиксированное место во входной строке. Это было удобно при использовании перфокарт (что характерно для многих ранних ЭВМ), но менее удобно при вводе информации с терминала. Кроме того, при фиксированном формате исключается возможность смещения полей при написании программы с целью показать ее структуру. Поэтому в большинстве современных языков ассемблера используется **переменный формат** входных данных. В переменном формате поля отделяются друг от друга одним или более разделителями. В действи-

тельности во всех языках ассемблера используется формат, в котором четыре элементарных данных — метка, операция, адрес (адреса), комментарий — задаются в указанном здесь порядке. Одно или более полей могут быть пустыми. Для типичного формата требуется, чтобы метка располагалась, начиная с крайней колонки (1-й в IBM 370, 2-й в Cyber 170), а после нее следовали один или более пробелов. Таким образом, отсутствие метки указывается с помощью некоторого числа пробелов в начале строки. Далее располагается мнемоническое имя операции, после которого также располагаются один или более пробелов. Затем следует адресное поле. Если команда содержит несколько адресов, то они обычно разделяются запятыми, хотя в этом различные ассемблеры могут отличаться друг от друга. За адресом также следует один или более пробелов, а все, что записано после пробелов, является комментарием. В языке ассемблера для небольших ЭВМ для разделения полей часто используются специальные знаки. Например, в языках ассемблера в системах PDP-11 и Intel 8080 после метки ставится двоеточие, после мнемонического имени операции — пробел, а после поля адреса — точка с запятой. (Любое из полей может также заканчиваться знаком конца строки.) Преимущество таких форматов заключается в том, что в любом месте можно использовать пробелы и допускаются строки, состоящие из одних комментариев (первым знаком, отличным от пробела, должна быть точка с запятой). Строки, состоящие из одних комментариев, допускаются и в других ассемблерах. Для обозначения таких строк в первой колонке записывается специальный знак (обычно *) строки-комментария.

При работе с образами перфокарт для идентификации строки нередко используются последние восемь колонок. В этом случае в колонках 73—76 набивается состоящее из четырех знаков кодовое слово для сегмента программы, а в колонках 77—80 — четырехзначный порядковый номер. Эти обозначения составляют часть поля комментария, и они широко используются в программах, хранимых на перфокартах. Однако в программах, записанных на диске, они обычно не применяются, поскольку во многих системах хранится не полный образ перфокарты, а лишь та его часть, которая расположена до последнего отличного от пробела знака включительно. (Как правило, это экономит 50 % объема памяти на диске.)

Поскольку для разделения полей требуется только одна колонка или один разделитель, программу можно записывать «сжатой влево». Например,

```
GCD LOAD FF1
SUB X2
BZ XT
```



```

BPL Y
SMI
STORE X2
BR YNOT
Y STORE F1
YNOT BR GCD
XT LOAD F1
RET

```

Этот фрагмент программы внушает ужас по нескольким причинам, не последней из которых является то, что в ней ничего не разделено и поэтому почти невозможно определить начало и конец различных полей. Для ассемблера не важно, сколько пробелов имеется во входной записи или какие имена используются. Это должен определить программист, с тем чтобы сделать программу наиболее удобной для чтения человеком. То, что может оказаться непонятным, следует пояснить в комментариях (которые совсем отсутствуют в приведенном выше фрагменте программы). Имена переменных и метки для указания размещения команд следует выбирать не произвольным образом, а так, чтобы они несли содержательную информацию, например являлись сокращениями определенных слов. Таким образом, данный выше фрагмент программы можно и нужно переписать так, как показано в табл. 11.2

Таблица 11.2

* В ДАННОЙ СЕКЦИИ ПРОГРАММЫ ВЫЧИСЛЯЕТСЯ НАИБОЛЬ-			
* ШИЙ ОБЩИЙ ДЕЛИТЕЛЬ ЧИСЕЛ FACT1 И FACT2. ОНА НАЗЫ-			
* ВАЕТСЯ ПОДПРОГРАММОЙ. РЕЗУЛЬТАТ ОСТАЕТСЯ В СУММА-			
* ТОРЕ. FACT1 И FACT2 — ПЕРЕМЕННЫЕ. ИСПОЛЬЗУЕТСЯ			
* АЛГОРИТМ ЭЙЛЕРА, ОПИСАНИЕ КОТОРОГО СОДЕРЖИТСЯ			
* В ...			

GCD	LOAD	FACT1	Начало цикла
	SUB	FACT2	Вычисление разности
	BZ	DONE	Если FACT1 = FACT2, то останов
	BPL	F1LARGE	Если FACT1 < FACT2, то ...
*			FACT2 больше
*	SMI		Изменить знак, чтобы получить
			FACT2 — FACT1
	STORE	FACT2	Положить FACT2 = FACT2 — FACT1
	BR	IFEND	
*			иначе (FACT1 больше)
F1LARGE	STORE	FACT1	Положить FACT1 = FACT1 — FACT2
IFEND	BR	GCD	Повторить цикл
DONE	LOAD	FACT1	Заслать результат в сумматор
	RET		Возвратиться в вызывающую про-
			грамму

Отметим, что никто кроме автора, по-видимому, не смог бы понять смысла исходной программы и привести ее к такому виду. (Однако и это не лучший вариант программы, так как алгоритм работает только со строго положительными целыми числами. Поэтому необходимо проверять, удовлетворяют ли исходные данные этому требованию. Приведенная же выше программа может просто заиклиться, если исходные данные заданы некорректно.)

В языках ассемблера на используемые символические имена (метки) накладываются ограничения, аналогичные тем, которые существуют для идентификаторов в языках высокого уровня. Как правило, метки должны начинаться с буквы и содержать только алфавитно-цифровые знаки (т. е. буквы и десятичные цифры, хотя в некоторых системах в число алфавитно-цифровых вводятся и дополнительные знаки). Длина меток обычно ограничивается пятью, максимум восемью, знаками. В большинстве языков ассемблера допускается использование выражений в тех местах, где должны стоять адреса. Например, если метка **BLK1** соответствует адресу **105**, то выражение **BLK1 + 2** — адресу **107**. Выражения могут содержать обычные арифметические и, быть может, другие операторы. Однако допустимая их форма зависит от реализации ассемблера.

Определение данных и резервирование памяти с помощью псевдокоманд

При записи имени в поле метки определяется символический адрес, используемый для обращения к соответствующей команде. Помимо этих символических адресов, которые обычно используются в командах перехода, необходимо обеспечить возможность определения адресов ячеек, содержащих константы и переменные. В некоторых языках ассемблера можно не определять символический адрес ячейки, содержащей одну переменную, например **X**. При отсутствии такого определения ассемблер сам выделяет в конце программы одну ячейку для этой переменной. Подобные действия аналогичны тем, которые выполняет компилятор с Фортрана при выделении области памяти всем неописанным переменным. В современной практике программирования отсутствие определений переменных не одобряется, поскольку из-за этого в программе могут возникнуть серьезные ошибки. Поэтому в большинстве языков ассемблера требуется, чтобы все символические адреса были определены в некотором месте программы, а отсутствие таких определений воспринимается как ошибка. Для этой цели используются **псевдокоманды**. Псевдокоманда (или, коротко, **псевдо**) является не командой, которую следует транслировать и выполнять вычислительной

машине, а инструкцией, или директивой, ассемблеру, сообщаящей, что он должен сделать. Таким образом, псевдо «выполняется» во время ассемблирования. Существуют два основных типа псевдокоманд: псевдо, с помощью которых данные загружаются как константы или начальные значения переменных, и псевдо, которые просто сообщают ассемблеру некоторую информацию о программе, например что необходимо выделить область для переменных. Последний тип псевдокоманд будем называть **директивами**.

Псевдокоманды загрузки данных. Типичными псевдокомандами, загружающими данные в память, являются псевдо, позволяющие определять, или описывать, числовые значения и символьные строки. Существуют два подхода к составлению псевдо этой группы. При первом из них для различных типов данных (десятичные числа, восьмеричные числа, символьные строки и т. д.) используются различные псевдо. При втором подходе используется одна-единственная псевдокоманда, а тип данных определяется из их формата. В ассемблере IBM 370 используется второй метод, а в ассемблерах PDP-11, Cyber 170 и Intel 8080 — комбинация двух указанных методов. В следующем примере показаны основные формы этих псевдокоманд:

```
NUMB  WORD  10B,27.3,FFFF
PIECE  BYTE  101B,23,A1H,—1
NAMES  CHAR  'ABCDEFGH1JK'
```

Первая псевдокоманда **WORD** загружает группу слов в последовательные ячейки. Символический адрес **NUMB** в поле метки определяет адрес первого загружаемого слова. В данном примере загружаются три слова, каждое из которых специфицируется элементарным данным в адресном поле. Элементарные данные отделяются друг от друга запятыми. Осуществляемый перевод данных определяется их форматом. Используемые здесь конечные символы **В** и **Н** служат для указания двоичных и шестнадцатеричных чисел соответственно. Для десятичных чисел конечный символ не используется. В большинстве языков ассемблера предполагается, что если число не специфицировано, оно десятичное. В некоторых языках основание чисел должно устанавливаться пользователем. В первом из приведенных выше примеров показана также спецификация констант с плавающей точкой. Таким числом является второе элементарное данное в **WORD**, поскольку в нем указана десятичная точка. Правила перевода для целых чисел и чисел с плавающей точкой обычно такие же, как и во многих языках высокого уровня, например в Фортране или Паскале. Вторая псевдокоманда **BYTE** встречается в некоторых вычислительных машинах, имеющих побайтовую адресацию, и позволяет загружать 8-битовые числа в от-

дельные байты. В остальном она аналогична псевдо **WORD**. В этом примере указанный адрес **PIECE** назначается первому загружаемому байту. Псевдо **CHAR** загружает символьные строки в последовательные байты. В рассмотренном выше примере 11 знаков загружаются в последовательные байты памяти (или в другие единицы памяти, в которых можно хранить 11 знаков). Символом **NAMES** обозначен адрес первого знака. Адресным полем в псевдо **CHAR** является строка, заключенная в кавычки. Загружаются те знаки, которые заключены между кавычками. Если требуется, чтобы строка содержала кавычку как знак, то в соответствующем месте следует записать две кавычки. Так,

QUOTE CHAR ""

определяет строку, содержащую одну кавычку. Помимо того что пользователь может определять константы с помощью псевдо, во многих языках ассемблера разрешено использование **литералов**. Литералы — это константы, записываемые в адресном поле команды, которая к ним обращается. В большинстве языков ассемблера им предшествует знак **=**. Ассемблер помещает константу, расположенную за знаком литерала **=**, в ячейку памяти, находящуюся в конце программы, и формирует символический адрес для обращения к ней. Например, предложение **LOAD = 6** после трансляции превращается в

```
LOAD D$001
.....
D$001 WORD 6
```

Отметим, что ассемблер формирует такую метку для литерала, которая вряд ли будет использована программистом. В действительности, во многих языках ассемблера формируемая метка будет выбрана из числа тех, которые не могут быть использованы программистом. Ассемблер хранит список всех специфицируемых литералов и строит из них таблицу в ячейках, расположенных вслед за программой. Всякий раз, когда должна встретиться константа, предпочтительнее воспользоваться литералами, поскольку, во-первых, программа в этом случае станет более удобной для чтения, а во-вторых, если один и тот же литерал встретится дважды, то ассемблер, как правило, будет хранить только одну его копию. Излишне говорить, что такие команды, как **STORE = 6**, не следует писать даже в тех случаях, когда ассемблер позволяет это! Пояснить сказанное можно на следующем примере:

```
LOAD    =3
STORE   =6
LOAD    =6
STORE   RESULT
```

В результате выполнения этих команд в ячейке **RESULT** будет храниться число 3, что не очевидно, если прочитать только две последние строки!

Директивы ассемблера. Эти псевдо используются для занесения информации в ячейки памяти, расположенные не в естественной последовательности, с целью зарезервировать область памяти для данных, состоящих из нескольких слов, а также для определения символических адресов и других имен. Типичными из этой группы псевдо являются **BSS**, **EQU**, **SET** и **ORG**; они встречаются почти во всех вариантах языка ассемблера. Например, псевдо

QARRAY BSS 100

резервирует следующую область, состоящую из 100 слов. Первое из этих слов имеет адрес **QARRAY**. (**BSS** расшифровывается как блок, начинающийся с символа.) Адресным полем команды **BSS** обычно является выражение, зависящее от других, ранее определенных символов. Псевдо **BSS** резервирует область памяти и тем самым увеличивает счетчик адресов.

Псевдо **ORG** (начальный) устанавливает определенное значение счетчика адресов. Например, псевдо

ORG 200+A

устанавливает значение счетчика адресов равным **200 + A**.

Псевдо **EQU** (эквивалентно) используется для непосредственного присвоения значения символу. Например, в результате выполнения команды

N EQU 100

значение символа **N** принимается равным 100, т. е. **N** становится символическим именем для 100. Псевдо **EQU** используется главным образом для задания значений «параметрам» программы, которые известны до ассемблирования. Например, если пишется программа сортировки блоков данных, то для выполнения некоторых действий над этими данными понадобятся области памяти. Если будут нужны, например, две области, размер одной из которых вдвое больше размера другой, то надо написать программу с параметром, скажем, **N**, который задает размер одной области. Программа может начинаться предложениями

N EQU 100
REG1 BSS N
REG2 BSS 2*N

которые резервируют эти две области. Если потребуется изменить размеры этих областей, то достаточно будет исправить только первое из этих предложений.

Псевдо **SET** похожа на псевдо **EQU** в том плане, что обе они задают значение символа. Однако первая из них выполняется в смысле присвоения. Символу может быть присвоено некоторое значение в одном **SET**-предложении, и это значение можно изменить в другом **SET**-предложении. Например, в результате выполнения команд

```
MEMSZ SET 300
MEMSZ SET MEMSZ/3
REG1 BSS MEMSZ
MEMSZ SET MEMSZ*2
REG2 BSS MEMSZ
```

будет получен тот же результат, что и в предыдущем примере. Вначале символу **MEMSZ** присваивается значение 300. Затем это значение заменяется на 100 и в **REG1** резервируется блок из 100 ячеек. Наконец, значение **MEMSZ** заменяется на 200 и в **REG2** резервируется 200 ячеек. Важно понимать, что **SET** является псевдокомандой и обрабатывается ассемблером во время ассемблирования. Она не генерирует никакого кода и не влияет на время выполнения программы. Например, при выполнении фрагмента программы

```
XX SET 20
    код позволяющий выполнить следующий цикл 20 раз
    (часть тела цикла)
XX SET XX+1
    (остальная часть тела цикла)
    конец цикла
YY SET XX
```

символу **YY** присваивается значение 21, поскольку ассемблер читает программу сверху вниз даже при ассемблировании команд, которые входят в тело цикла и должны повторяться во время выполнения программы.

Ассемблер вырабатывает объектный код, который загружается в последовательные ячейки памяти, начиная, как правило, с нулевой. Вначале значение счетчика адресов полагается равным 0. Оно изменяется каждой командой или псевдокомандой, резервирующей область памяти или меняющей место, в которое должны быть загружены последующий код и данные. Многие ассемблеры имеют несколько счетчиков адресов. Псевдокоманды позволяют программисту переключаться с одного счетчика адресов на другой. Программист может интерпретировать это как работу с различными частями бланка. Каждая часть имеет свой собственный счетчик адресов. При переключении на другой счетчик адресов программа выполняется так, как если бы она была записана на соответствующей части бланка.

Ассемблер собирает тексты каждой части (называемой **управляющей секцией** в одних языках ассемблера и **блоком** в других) таким образом, что код одной части занимает последовательные ячейки памяти, следуя за кодом другой.

Последняя строка программы на языке ассемблера должна быть указана. Обычно это делается с помощью псевдо **END**, которая сообщает ассемблеру, что надо начать следующий проход или завершить работу.

Второй проход

На втором этапе сканирования входа ассемблер вырабатывает двоичный объектный код, состоящий из машинного эквивалента входного текста и данных в двоичной форме. Куда размещает ассемблер эту выходную информацию? Одна возможность — разместить ее непосредственно в память, в ячейки, указанные пользователем. В этом случае выполнение программы можно начать сразу же после окончания трансляции. Такая схема носит название **загрузить и выполнить**. Она имеет много недостатков, хотя по сравнению с большинством других схем, которые будут рассмотрены, она имеет несомненные преимущества в скорости. Среди недостатков схемы «загрузить и выполнить» можно отметить следующие.

1. Программа не может быть загружена в ячейки, занимаемые ассемблером. На небольших вычислительных машинах программа ассемблера нередко занимает большую часть оперативной памяти, исключая тем самым возможность прямой загрузки.

2. Каждый раз, когда нужно выполнить программу, ее необходимо ассемблировать заново. (Этот недостаток является серьезным только в том случае, если ассемблер работает медленно.)

Первая проблема может быть решена, если во время второго прохода ассемблера объектный код размещать на внешнем запоминающем устройстве. Такую схему нельзя строго назвать схемой «загрузить и выполнить», однако она все же может сохранять некоторые преимущества в скорости, если внешнее запоминающее устройство достаточно быстрое. Если копия объектного кода хранится на диске или перфокартах, то для последующего выполнения программы повторного ассемблирования не требуется — нужна будет лишь перезагрузка. Следовательно, форма выхода ассемблера должна быть приемлемой для загрузчика (например, для абсолютного загрузчика, который был описан выше).

- Ниже приводится пример программы на языке ассемблера для типичной одноадресной машины.

```

START  LOAD    A
        ADD     B
        STORE   C
        BRANCH D
A       WORD    13,15
B       EQU     A+1
C       BSS     2
D       ADD     =10
        STORE   C+1
        HALT
        END     START

```

Адресное поле предложения **END**, написанного в последней строке, сообщает ассемблеру, что выполнение программы следует начать со строки, помеченной символом **START**. При трансляции этой программы ассемблер вырабатывает двоичный эквивалент кода, данного в табл. 11.3. Для человека удобнее, чтобы коды операции оставались в мнемонической форме.

Таблица 11.3

Ячейка	Команда	Адрес или содержимое ячейки в десятичной форме
0	LOAD	4
1	ADD	5
2	STORE	6
3	BRANCH	8
4	WORD	13
5	WORD	15
8	ADD	11
9	STORF	7
10	HALT	
11	WORD	10

Отметим, что ассемблер разместил литеральную константу 10 в конце программы, в ячейке 11. Программа завершает работу на команде **HALT**, содержащейся в ячейке 10. Во многих вычислительных машинах в этом месте управление передается операционной системе. Если бы команда **HALT** была пропущена, то вычислительная машина попыталась бы «выполнить» числовое данное, содержащееся в ячейке, следующей за последней командой **STORE**. Это было бы немедленно воспринято как ошибка, если числовое данное нельзя было интерпретировать как допустимую команду. Если же оно воспринялось бы как допустимая команда, то задача стала бы более сложной, поскольку эта ошибка была бы обнаружена не сразу, а только при появлении первой ячейки, содержимое которой эквивалентно

запрещенной команде, нарушающей требования защиты. Программисты должны проявлять особое внимание и быть уверенными в том, что написанные ими команды и данные разделены и ошибок такого рода у них не будет. Поэтому данный выше пример не демонстрирует хороший стиль программирования. Команды и данные лучше полностью отделять друг от друга.

Ассемблер в системе IBM 370

Язык ассемблера для вычислительных машин IBM 370 отличается от только что рассмотренного по нескольким пунктам. Некоторые из отличий возникают из-за дополнительных задач, возложенных на ассемблер и вызванных сложностью машины; другие являются следствием попытки сделать более единообразным использование многих псевдокоманд, связанных с загрузкой данных. В настоящем разделе будет дано краткое описание ассемблера для IBM 370 и показаны его характерные особенности.

Входная информация поступает в виде образа перфокарт с полями переменного формата. Поля отделяются друг от друга пробелами. Колонки 73—80 игнорируются. Символические имена могут содержать до восьми знаков. Многие программисты обычно записывают мнемонические имена с 10-й колонки, а адресное поле с 16-й колонки. Однако если для описания структуры программы используется смещение предложений при их записи, то лучше, чтобы адресное поле начиналось не раньше чем с 20-й колонки. В системе IBM 370 мнемонические имена операций задаются таким образом, что основные функции обозначаются определенными буквами, к которым могут быть приписаны другие буквы, уточняющие эти операции. Так, **A**, **S**, **M** и **D** обозначают четыре арифметические операции, а **L** и **ST** — загрузку и запись. Добавление буквы **E** указывает на плавающую точку, буквы **D** — на удвоенную точность и плавающую точку и т. д. Команды типа **RR** обозначаются добавлением буквы **R**, команды с непосредственной адресацией — добавлением буквы **I** и т. д. Так, **ADR** означает сложение с удвоенной точностью содержимого регистров. Простота такой схемы может привести к ошибкам. Например, глядя на команду **LTR** (загрузить и проверить регистры), можно попытаться использовать команду **LT**, которой не существует.

Адреса в многоадресных командах разделяются запятыми, а индексы указываются в круглых скобках. Например,

A R3, B (R7)

означает прибавление к **R3** адреса **B**, индексированного с помощью регистра 7. Символ **R** в записи **R7** не является необхо-

димым. Однако во многих версиях языка ассемблера имена от **R0** до **R15** полагаются эквивалентными именам от **0** до **15** соответственно и поэтому можно использовать эту более наглядную запись. При этом можно считать, что в начале каждой программы содержатся предложения

```
R0 EQU 0
R1 EQU 1
.....
R15 EQU 15
```

За исключением нескольких случаев, команды системы **IBM 370** содержат не более двух адресов. Исключение составляют команды **LM** и **STM** (групповая загрузка и групповая запись в память соответственно). Первая из этих команд позволяет загрузить несколько регистров общего назначения словами из памяти, а вторая — записать содержимое регистров в память. Длины символьных и десятичных операндов также указывают в круглых скобках. Например, при выполнении команды

```
AP B(6), C(4)
```

4-байтовое упакованное десятичное число, начинающееся с адреса **C**, прибавляется к 6-байтовому упакованному десятичному числу, начинающемуся с адреса **B**.

Конечно, программисту удобнее, когда длину определять не нужно. В таком случае это должен сделать ассемблер. Как ассемблер определит длину? Он исследует определение операнда. Когда определяются символические адреса **B** и **C**, задается объем области памяти. Это и есть длина, используемая ассемблером. Например, если область для данных резервируется с помощью псевдо **DS** (определить память), то программист может указать длину несколькими способами. Предложения

```
B DS PL6
C DS PL4
```

указывают на то, что **B** и **C** являются областями памяти для хранения упакованных десятичных строк длиной 6 и 4 байт соответственно. (Символ **P** означает «упакованное», **L** — «длина».) Общий формат псевдо **DS** следующий:

```
символ DS элементарное, элементарное, ..., элементарное
           данное 1      данное 2      данное n
```

Каждое из этих элементарных данных является спецификацией набора областей памяти, которые описываются буквами и числами. Например, символ **"F"** описывает число с фиксированной точкой и одинарной точностью, т. е. 32-битовое слово, а символ **"D"** — число с плавающей точкой и удвоенной точностью, т. е.

64-битовое слово. Каждому такому символу может предшествовать целочисленный коэффициент повторения. Например, предложение

XVAR DS 3F,2D

описывает набор из трех 32-битовых слов, за которым следует набор из двух 64-битовых слов. Поскольку данные установленной длины должны находиться в соответствующих границах памяти, в системе IBM 370 и других, более ранних ее версиях с помощью дескрипторов полуслова, слова и двойного слова происходит выравнивание области по границам соответствующего размера. Например, если записаны предложения

MSG DS CL7
YVAR DS F,2D

то вначале резервируется область для символьной строки длиной 7 байт, адрес первого из которых есть **MSG**. Затем резервируется слово с одинарной точностью, за которым следуют два слова с удвоенной точностью. Предположим, что адрес начала области **MSG** равен 64. Ее длина составляет 7 байт, и поэтому ближайшая свободная область начинается с адреса 71. А поскольку дескриптор **F** является 32-битовым, т. е. выравнивающим по границе слова, то область **YVAR** будет размещена, начиная с ближайшего слова, следующего за 71-м байтом, т. е. с 72-го по 75-й байт. Отметим, что адрес области **YVAR** (т. е. адрес первого элементарного данного, зарезервированного командой **DS**) равен 72. И наконец, резервируются два слова с удвоенной точностью (состоящие из 64 бит каждая). Поскольку слова с удвоенной точностью должны начинаться с границы двойного слова (т. е. их адреса кратны 8), то они разместятся в байтах 80—87 и 88—95 соответственно.

Основное отличие псевдо **DS** от рассмотренной выше общей псевдо **BSS** заключается в том, что для команд системы IBM 370 надо задавать длины различных полей данных. Когда в программе на языке ассемблера IBM 370 определяется некоторое имя, то определяются также его длина и адрес. Длина задается дескриптором (**F**, **D** и т. д.), а адрес — местом, в котором находится этот символ.

В языке ассемблера IBM 370 используется одна псевдокоманда (**DC**), определяющая константы из любого класса данных. Тип данных указывается в адресном поле. Дескрипторы в псевдо **DC** (определить константу) аналогичны тем, которые используются в псевдо **DS**, однако за ними должны следовать значения данных, заключенные в кавычки. Например, предложение

```
XARRAY DC F'12', F' — 24', F'36'
```

определяет набор из трех слов с фиксированной точкой, значения которых равны 12, —24 и 36. Эти слова выравниваются точно так же, как и при выполнении псевдо **DS**. Данное предложение можно записать в более короткой форме:

```
XARRAY DC F'12, — 24, 36'
```

Другие примеры показаны в следующих секциях определения данных:

```
D1      DC 2PL3'45'
XSTR    DC C'1234 A'
SHORT   DC H'34'
YDEC    DC P'13', C'AB, CD'
HEX     DC X'5555FFFF'
        DS 0F
HEX1    DC X'5555FFFF'
FPDATA  DC E'43.5', D'—56.2'
```

Первое предложение определяет две строки длиной 3 байт, в каждом из которых содержится упакованное десятичное число **45**. Следующее предложение определяет строку из шести знаков **1234—A**, где — знак пробела. Отметим, что нет необходимости задавать длину в тех случаях, когда она равна той, которая подразумевается самим элементарным данным. Третье предложение определяет полуслово (16 бит), содержащее целое число **34**. В область **YDEC** входит упакованная десятичная строка длиной 2 байт, содержащая число **13**, за которой следует строка, состоящая из пяти символов, включая запятую. Область **HEX** содержит группу из 32 бит, шестнадцатеричное значение которой равно **5555FFFF**. Отметим, что она может не лежать на границе слова. Если необходимо произвести выравнивание слова, можно использовать псевдокоманду **DS**. Она осуществляет выравнивание по границе слова, но память не резервирует. Следовательно, **HEX1** будет выровненным словом. Последнее предложение определяет два числа (**43.5** и **—56.2**) с плавающей точкой, с одинарной и удвоенной точностью соответственно.

Существует немало других дескрипторов данных, которые можно найти в руководстве по языку ассемблера IBM. Одним из наиболее важных является дескриптор **A**, который описывает адрес. Например, предложение

```
YADDR DC A(Y)
```

определяет 32-битовое значение, равное адресу переменной **Y** и выровненное по границе слова.

Литералы. После рассмотренных выше вопросов следует перейти к рассмотрению литералов. Литерал образуется с

помощью знака $=$, за которым следует допустимое для псевдо **ДС** элементарное данное адресного поля. Ниже приводится несколько примеров.

A	R4,=F'13'	Прибавить число 23 с фиксированной точкой
AE	FPR2,=E'13.5'	Прибавить число 13.5 с плавающей точкой
АН	R1,=H'-9'	Прибавить полуслово -9
AP	A,=P'-235'	Прибавить упакованное двухбайтовое число -235
N	R6,=X'0000FFFF'	И шестнадцатеричное значение 0000FFFF

При ассемблировании этих команд в конце программы определяются соответствующие константы, а их адреса подставляются вместо литералов.

Назначение базового регистра. Напомним, что в системе IBM 370 адрес должен быть представлен с помощью 12-битового смещения и адреса, содержащегося в базовом регистре. Это означает, что в такую же форму ассемблер должен перевести адрес **X**, если программист пишет команду, например **A R4, X**. Поэтому необходимо знать, какой регистр может быть использован в качестве базового и что в нем будет содержаться во время выполнения программы. Такую информацию ассемблеру должен дать программист и делается это с помощью псевдо **USING**. Предположим, например, что программист решает использовать регистр **15** в качестве базового. С помощью последовательности команд

```

L          R15,=A(NEXT)
USING     NEXT,R15
NEXT      .....    следующая команда

```

во время выполнения в регистр **15** вначале засылается адрес размещения команды **NEXT**. Затем ассемблеру (во время ассемблирования) сообщается, что регистр **15** используется для хранения адреса **NEXT**. Аналогичная последовательность операций может быть выполнена несколькими способами. Команду загрузки можно заменить командой **BAL R15,NEXT**. Символ **BAL** означает «переход с возвратом». Эта команда осуществляет переход по указанному адресу (**NEXT**) и засылает в регистр **15** адрес следующего за ней байта. Этот адрес равен адресу, соответствующему **NEXT**. Еще лучше использовать команду **BALR**. Рассмотрим последовательность команд

```

BALR      R15,0
USING     *,R15

```

Первой выполняется команда «переход по регистру с возвратом» (**BALR**), которая вначале засылает в регистр **15** адрес следующего за ней байта, а затем осуществляет переход по адресу, содержащемуся во втором из указанных регистров. В данном

случае вторым из указанных регистров является регистр **R0**, который не может быть использован для индексирования, и поэтому появляется команда перехода на ячейку 0. Однако в системе **IBM 370** в этом особом случае переход не осуществляется, а производится лишь засылка адреса следующей команды в регистр **15**. Псевдо **USING** сообщает ассемблеру, что **R15** используется для хранения текущего значения счетчика адресов, которое обозначается символом '*'.

После выполнения команды **USING** ассемблеру сообщается, что для вычисления некоторого адреса им может быть использовано содержимое определенных регистров. Например, при ассемблировании последовательности команд

```
BALR    R15,0
USING   *,R15
T  SR    R5,R6
U  BP     T
    AR    R5,R7
    BM    U
```

ассемблер должен определить адрес **T** в команде **BP**. Во время своей работы ассемблер располагает информацией о том, что **R15** содержит адрес, соответствующий **T**. Поэтому этот адрес представляется в виде **0(15)**, т. е. команда **BP T** аналогична команде **BP 0(15)**. Команда **BM** после ассемблирования будет выглядеть как **BM 2(15)**, так как адрес **U** превосходит значение базового регистра **15** на величину смещения, равную 2 байт.

Очевидно, что программист не должен изменять содержимое базовых регистров, не информируя об этом ассемблер. Для передачи такой информации ассемблеру используется псевдо **DROP**. Псевдокоманда **DROP R15,R12** сообщает ассемблеру, что регистры **15** и **12** больше не могут быть использованы как базовые. Одновременно программист должен заботиться о том, чтобы базовые регистры находились в распоряжении ассемблера и чтобы тем самым ассемблер смог определить любой указанный адрес. Обычно требуется по меньшей мере два базовых регистра: один для адресов программы и один для данных. Единственное ограничение, накладываемое ассемблером на регистры, которые могут быть использованы для индексирования в машине, заключается в том, что их число не должно превосходить 15. Однако программист должен разумно выбрать их количество, с тем чтобы для выполнения других арифметических операций и процедур индексирования в его распоряжении также имелись бы регистры.

Ассемблер МАКРО-11 в системе PDP-11

В языке ассемблера PDP-11 для строк операторов ассемблера используются четыре стандартных поля, которые разделяются специальными знаками и (не обязательно) пробелами. За символическим адресом, если он имеется, следует двоеточие. Отсутствие двоеточия после первой непустой символьной строки обычно указывает на то, что эта строка является мнемоническим обозначением операции. За полем операции следует по меньшей мере один пробел, а адресное поле отделяется от комментария точкой с запятой. Несколько адресов могут отделяться друг от друга пробелами или запятыми. Указанные конструкции иллюстрируются в следующем фрагменте программы:

```
START: MOV  A,RO      ;Загрузить индекс
        MOV  ZERO R1  ;Заслать начальное значение в R1
        ADD  D(RO) R1
```

Символические адреса в МАКРО-11 являются алфавитно-цифровыми строками, начинающимися с алфавитного знака. В число алфавитных входят также точка и знак доллара, однако обычно они резервируются для использования системой и макро. Допускается произвольное число знаков, однако обрабатываются только первые шесть из них. Поэтому лучше ограничиться шестью знаками.

Для указания требуемого типа структуры адреса используются специальные знаки, полный список которых приводится в табл. 11.4.

Из таблицы видно, что `%3` означает регистр 3, так что `MOV %3,3` означает засылку содержимого регистра 3 в память по адресу 3. Выше регистр 3 мы обозначали через `R3`. Во многих версиях МАКРО-11 символы от `R0` до `R7` эквивалентны символам от `%0` до `%7` соответственно. Если это не так, то пользователь может написать в начале программы следующие строки:

```
R0 = %0
R1 = %1
.....
R6 = %6
R7 = %7
PC = %7
```

Смысл знака «`=`» такой же, как и псевдо `SET`. Например, предложение `R3 = %3` означает, что значение имени `R3` полагается равным значению `%3`. Символические имена имеют адресное значение и другие квалификаторы. В рассматриваемом примере тот факт, что `R3` представляет регистр, является одной из указательных частей информации, записанной для `R3`. Этот факт

становится известен ассемблеру при чтении им предложения $R3 = \%3$.

Другими примерами, иллюстрирующими действия с адресами в языке МАКРО-11, являются следующие предложения:

```
MOV  #10,%2      ;Заслать целое число 10 в регистр 2
ADD  A(2),(4)     ;Заслать A, индексированное с помощью R2,
                  ;по адресу, содержащемуся в R4
SUB   (R0)+,-(R2) ;Заслать (R0) в -(R2)
                  ;Содержимое R0 и R2 увеличивается и умень-
                  ;шается на 2 соответственно
```

Таблица 11.4. Указатели структуры адреса
в МАКРО-11

#	Непосредственный (прямой) адрес
@	Косвенный адрес
()	Косвенный (только с использованием регистра) или индексирование
—	Автодекремент
+	Автоинкремент
%	Регистровый адрес

К псевдокомандам загрузки данных относятся следующие:

```
.WORD
.BYTE
.ASCII
```

Адресным полем псевдокоманд **.WORD** и **.BYTE** является совокупность элементов, отделенных друг от друга запятыми. Этими элементами могут быть произвольные выражения, содержащие символические адреса. Выражения вычисляются и хранятся в 16 или 8 бит соответственно. Для псевдо **.WORD** эта информация располагается, начиная с границы слова. Символ в поле размещения приписывается адресу первого загружаемого байта.

Необходимо учитывать особенности вычисления выражений в МАКРО-11. Во-первых, вся числовая информация записывается в восьмеричном виде. Во-вторых, вычисления производятся слева направо, если не используются скобки. В-третьих, могут использоваться угловые скобки. Так, в языке МАКРО-11 выражение $13 + 3 * 2$ равно 28 в десятичной системе, поскольку число 13 записано в восьмеричной системе и равно 11 в десятичной, а сложение выполняется до умножения. Для получения «обычного» результата следует записать это выражение в виде $13. + \langle 3 * 2 \rangle$. В этом случае с помощью точки, поставленной после числа 13, указано, что данное число является десятичным, а произведение записывается в угловых скобках.

Псевдо **.ASCII** позволяет ассемблировать символьные строки в код **ASCII**. Так, по команде

STR: .ASCII /ABCDEFGHJKLM/

12 байт загружаются содержимым указанной строки. Адресу первого байта присваивается имя **STR**. Директивами ассемблеру являются следующие:

.ODD
.EVEN
.BLKW
.BLKB
.RADIX

Псевдо **.ODD** и **.EVEN** засылают в счетчик команд адрес ближайшего нечетного и четного байта соответственно. Для этого, если необходимо, значение счетчика увеличивается на единицу. Псевдо **.BLKB** и **.BLKW** резервируют в памяти байты и слова соответственно. Например, по директиве

X: .BLKW 40

в памяти будет зарезервировано 40 восьмеричных слов, начиная с границы слова. **X** — это адрес первого слова. Псевдо **.RADIX** позволяет пользователю задавать основание системы счисления. По умолчанию основание равно 8, но пользователь может сделать его равным 2, 4, 8 или 10, записав, например,

.RADIX 10

После того как основание объявлено, оно остается неизменным до появления нового объявления. Основание может быть изменено временно. Для этого применяется одна из следующих конструкций:

.WORD 54, ↑D231, ↑B1101, F3.6, ↑O7777

Первое слово ассемблируется с использованием текущего основания, заданного псевдо **.RADIX**. Последующие слова переводятся как десятичное целое (**D**), двоичное целое (**B**), число с плавающей точкой в 16-битовом формате (**F**) и восьмеричное целое (**O**).

Именем для счетчика адресов является точка. Например, значение счетчика адресов может быть увеличено на 7, если записать

.=.+7

Ассемблер КОМПАСС в системе Сугер 170

В языке ассемблера КОМПАСС формат команд несколько отличается от того, который используется в большинстве языков ассемблера, однако в основном он такой же. Псевдокоман-

ды имеют обычный формат. Исходные операторы имеют, как обычно, четыре поля. Первое поле, поле метки, может начинаться с колонок 1 или 2. Если и в первой, и во второй колонках стоят пробелы, то команде (или псевдокоманде) не присписывается никакого символического адреса. Если же метка указана, то она должна содержать от одного до семи алфавитно-цифровых знаков и начинаться с алфавитного знака. За меткой должны следовать один или более пробелов. Поле операции может начинаться в любой из колонок с 3-й по 29-ю включительно. За полем операции, отделяясь от него одним или более пробелами, следует адресное поле, называемое в документациях по языку КОМПАСС **переменным полем**. Оно должно начинаться с 30-й колонки. В конце строки находится поле комментария, которое может располагаться в любых колонках, начиная с 30-й. Требование, согласно которому комментарий не может начинаться до 30-й колонки, преследует несколько целей. Одна из них связана с тем, что в адресном поле пробелы могут появиться до 30-й колонки (хотя это и не рекомендуется). Отметим, однако, что все знаки, которые следуют за первым пробелом, стоящим в 29-й колонке или правее, являются комментарием. Так, строка с пробелами во всех колонках с 1-й по 29-ю является исключительно строкой-комментарием. Строка-комментарий может быть также указана с помощью символа * в 1-й колонке (что свойственно для многих языков ассемблера). Пользователь должен также постоянно помнить, что адресное поле начинается до 30-й колонки, иначе оно рассматривается как комментарий.

Обычно в программе, написанной на языке ассемблера КОМПАСС, первая колонка резервируется для звездочки, являющейся знаком комментария, или для запятой, которая указывает, что данная строка является продолжением предыдущей, колонки 2—9 используются для поля метки, 11—16 — для поля операции, а 18—29 — для адреса. Некоторые программисты предпочитают отступать от этого правила, смещая команды при их записи таким образом, чтобы показать структуру программы. (Колонка, с которой начинаются комментарии, может быть изменена с помощью директивы ассемблеру **COL**.)

Вообще говоря, в системе Cyber 170 существуют команды двух типов: выполняющие арифметические и логические операции и все остальные, которыми являются главным образом команды проверки. В командах первого типа используется формат, в котором часть приказа записывается в адресном поле, а часть адресов — в поле операции. Например, в команде

RX3 X4+X5

один из трех адресов расположен в поле операции, а знак плюс в адресном поле сообщает нам, что это команда сложения. В командах с непосредственной адресацией некоторые приказы записываются в адресном поле. Например, в команде

SB3 A5—B4

знак минус является частью кода операции. В других командах, как, например,

SB3 A5—L

знак минус просто сообщает ассемблеру, что надо вычислить отрицательное значение адреса L (обратный код) и затем выполнить команду

SB3 A5+(—L)

Команды второго типа записываются в более традиционной форме. Например, по команде

NE B3,B6,T

будет выполнен переход на T, если содержимое регистра B3 не совпадает с содержимым регистра B6. Язык КОМПАСС позволяет записывать многие приказы различными способами. Например, две команды

GT B5,B6,T

LT B6,B5,T

идентичны: при выполнении любой из них будет осуществлен переход на T, если B5 больше, чем B6 (или B6 меньше, чем B5).

Имена регистров с X0 по X7, с B0 по B7 и с A0 по A7 зафиксированы в ассемблере, и поэтому программист не может их переопределить. Если для некоторого регистра требуется использовать символическое имя, то оно должно быть записано в форме A. имя, X. имя или B. имя. Например, если имя REGSUM было определено как соответствующее 5 (возможно, с помощью псевдо EQU), то вместо X5 может быть использовано имя X.REGSUM.

Псевдокоманды. В каждой программе, написанной на языке КОМПАСС, должны присутствовать две псевдокоманды. Первая — псевдо IDENT. Она должна быть записана в первой строке программы. Эта псевдокоманда задает имя программы. Например, предложение

IDENT TEST1

определяет TEST1 в качестве имени программы. В последней строке программы должна находиться псевдо END, записанная в форме

метка **END** **старт**

где значение метки в поле метки, если оно указано, полагается равным общей длине программы, а адресное поле «старт», если оно указано, является меткой, с которой должно начаться выполнение.

В языке КОМПАСС имеется много других псевдокоманд. Для получения полной документации читателю следует обратиться к руководству по КОМПАССу. С помощью псевдокоманд **EQU**, **SET** и **BSS** могут быть определены символы и зарезервирована область памяти. В адресном поле этих псевдо могут содержаться символы и выражения, содержащие символы. Однако, вообще говоря, все такие символы должны быть заранее определены. Иными словами, необходимо, чтобы при первом проходе ассемблера можно было вычислить все адреса.

Основными псевдокомандами генерации данных являются **BSSZ** (которая генерирует слова нулевых данных, а в остальном она аналогична псевдо **BSS**), **DATA** (которая генерирует слова данных в большом числе форматов), **DIS** (которая генерирует символьные строки данных), **VFD** (которая переводит ряд выражений в набор полей заданной длины) и **CON** (генерирующая константы, значение каждой из которых задается выражением и которые имеют размер слова). Например, последовательность псевдокоманд

```
A BSSZ 10
B DATA —56,1.5E3,B7700,O777.D4.5,3HABC,A.WXYZ.
C DIS 2,ABCDE
D DIS ,*0123456789ABCDEF*
X VFD 10/D—C,40/Z,25/—1
Y CON D—A,B,2•B—A
```

генерирует 26 слов данных. Первая из них генерирует 10 слов нулевых данных. Вторая генерирует семь слов данных: целое число —56, число с плавающей точкой 1500., восьмеричное целое 7700, восьмеричное целое 777, десятичное число с плавающей точкой 4.5 (обычно числа являются десятичными, но можно использовать опцию для замены основания), за которым следуют два слова, содержащие символьные строки. Первое содержит строку **ABC**, дополненную справа семью пробелами, второе — строку **WXYZ**, дополненную слева шестью пробелами. Начальный знак определяет тип используемого преобразования. В этом примере были использованы знаки **B** и **O** для восьмеричных чисел, **D** для десятичных, **H** для символьных строк, выровненных по левому краю, и **A** для символьных строк, выровненных по правому краю. Перечень остальных знаков можно найти в руководстве по языку КОМПАСС.

Псевдо **DIS** имеет две формы, каждая из которых используется в рассматриваемом примере. Число слов, которое требуется сгенерировать, может быть указано до запятой. В этом случае считаются знаки, стоящие после запятой, а их количество равно указанному числу, домноженному на 10. (Напомним, что в системе Cyber 170 слова 60-битовые, а знаки занимают по 6 бит.) В качестве указателя числа слов может также стоять нуль (или пробел). В этом случае первый знак после запятой используется как ограничитель строки. В следующем предложении ограничителем строки является знак *. Строка состоит из всех знаков, расположенных между двумя звездочками.

Адресное поле псевдо **VFD** в рассматриваемом примере состоит из последовательности подполей, отделенных друг от друга запятыми. Каждое подполе содержит указатель длины (в битах), за ним следует выражение, которое переводится в целое число и размещается в указанном числе битов. Так, подполю **10/D—C** будет соответствовать 10-битовое число, равное $19(=D-C)$. Подполю **25/—1** будет соответствовать битовая строка **11 ... 110** (отметим, что здесь применяется арифметика в обратных кодах). Таким образом, в результате выполнения приведенной выше псевдокоманды **VFD** будет записано 75 бит, или $1\frac{1}{4}$ слова. Если за ней следует команда или псевдокоманда, при выполнении которой осуществляется выравнивание по границе слова, то оставшиеся $\frac{3}{4}$ слова заполняются нулями.

Псевдо **CON** аналогична псевдо **VFD**, за исключением того, что она генерирует 60-битовые слова данных. Ее адресное поле состоит из нескольких выражений, значения которых вычисляются ассемблером и размещаются в последовательные слова.

Счетчики адресов и начального адреса. Для указания ячейки, в которую должен быть загружен код, в языке КОМПАСС используется счетчик начального адреса. При ассемблировании каждого слова счетчик начального адреса увеличивается на единицу. С помощью псевдо **ORG** можно производить засылку значений в счетчик начального адреса. Например, псевдокоманды

```
X DATA 1
   ORG   X+10
```

устанавливают такое значение счетчика начального адреса, что после слова **X** остается интервал в 9 слов. В языке КОМПАСС используется также счетчик адресов, который обычно имеет то же значение, что и счетчик начального адреса. Когда в поле метки некоторой команды указывается символический адрес, для присвоения ему значения используется счетчик адресов. С помощью псевдо **LOC** в счетчик адресов можно заслать зна-

чение, отличное от значения счетчика начального адреса. Если в него засылается значение, на 100 большее, чем значение счетчика начального адреса, то эта разность будет оставаться неизменной до тех пор, пока счетчик адресов вновь не изменится. Такой прием может использоваться в тех случаях, когда некоторые команды, размещаемые при ассемблировании в одну область памяти, необходимо переслать с помощью программы пользователя в другую область памяти до того, как эти команды будут выполняться. Псевдо **LOC** может быть использована для засылки окончательного адреса кода в счетчик адресов. Такой прием не рекомендуют использовать, поскольку он вносит путаницу в программу. Однако в системе Cyber 170 I/O он может оказаться полезным.

Ассемблер в системе Intel 8080

Данный язык ассемблера предназначен для очень небольших ЭВМ. Поэтому он имеет меньше характерных особенностей, чем языки ассемблера, рассмотренные в предыдущих трех разделах. Ассемблер в системе Intel 8080, так же как и в PDP-11, ориентирован на входную информацию, поступающую с перфоленты или терминала. Следующие четыре поля операторов являются стандартными: метка (символический адрес) — после нее ставится точка с запятой (если метка записана); операция — после нее ставится пробел; адрес — после него ставится точка с запятой или знак конца файла; комментарий — после него ставится знак конца файла.

Вместе с указанными знаками конца полей могут использоваться дополнительные пробелы. Символические адреса (имена) должны состоять не более чем из пяти знаков и не могут быть именами регистров (**A**, **B**, **C**, **D**, **E**, **F**, **H**, **L** или **SP**) и кодами операций. Остальные требования являются стандартными. В адресном поле разрешается использовать обычные выражения, значения которых могут быть представлены допустимым числом битов. Например, предложение

```
MVI  B,X*(Y+Z)
```

допустимо, если значения **X**, **Y** и **Z** таковы, что величина $X*(Y+Z)$ заключена между 0 и 255 включительно, поскольку в команде **MVI** (пересылка непосредственная) для записи прямого адреса требуется один байт (8 бит). В выражениях могут быть также использованы следующие операции.

AND

OR

NOT

XOR Исключающее или

- MOD** Остаток от деления левого операнда на правый
SHR Сдвиг первого операнда вправо на *N* позиций, где *N* — второй операнд
SHL Аналогично **SHR**, но сдвиг осуществляется влево

При выполнении операции сдвига биты нулей «вдвигаются» с соответствующего конца. Например, пересылку в сумматор одного байта с содержимым старших 8 бит адреса **X** можно выполнить с помощью команды

MVI A, X SHR 8

Восемь нулевых битов, появившихся при сдвиге слева, гарантируют, что адрес содержится в 8 бит справа. С другой стороны, пересылку последних 8 бит адреса **X** в регистр **C** можно выполнить так:

MVI C, X AND OFF

Константа **OFF** является шестнадцатеричной и содержит восемь двоичных единиц. С помощью операции **AND** из **X** извлекаются 8 младших значащих битов, так что адрес является допустимым. Вычисление адресов производится в 16-битовой арифметике в дополнительных кодах. Счетчик адресов представляется знаком \$. Так, по команде

JMP \$+10

будет осуществлено смещение на 10 байт от адреса команды **JMP**. В качестве операнда в адресном поле может быть использована команда, взятая в круглые скобки. Например, загрузка пары регистров **B, C** двоичным кодом команды **MVI E, 'X'** может быть выполнена так:

LXI B, (MVI E, 'X')

Псевдокомандами размещения данных являются следующие:

- DB** ; определить байты
DW ; определить слова

Ниже показано, как псевдо **DW** может быть использована для спецификации последовательности 16-битовых слов:

DW элементарное, элементарное, ..., элементарное
 данное 1 данное 2 данное *n*

Если это предложение помечено, то адресу первого загружаемого байта присваивается соответствующее имя. Каждое элементарное значение транслируется в 16-битовую величину и загружается в два последовательных байта; младший значащий байт элементарного данного хранится в ячейке с меньшим адресом. Каждое элементарное значение может быть одним из следующих типов:

- десятичная константа (например, 373, —27 или 31D);
- восьмеричная константа (например, +277O, 135O или —1111O);
- двоичная константа (например, 101B, или —111B);
- шестнадцатеричная константа (например, 375H или OFFF);
- символический адрес, который где-то определяется;
- литер, заключенная в кавычки;
- выражение, содержащее любое из перечисленных выше данных, при условии, что значение выражения может быть записано в 16 бит.

Отметим, что если за числом не указан знак, **O**, **B** или **H**, и если в нем не используются шестнадцатеричные цифры, то оно является десятичным. Шестнадцатеричные числа должны начинаться с цифр, иначе они будут приняты за имена.

Псевдо **DB** аналогична псевдо **DW**, за исключением того, что при выполнении первой из них загружаются 8-битовые величины, и поэтому элементарные данные должны быть 8-битовыми. Адресным полем псевдо **DB** может быть символьная строка, заключенная в кавычки. Например, при выполнении псевдокоманды

MSG: DB 'END OF INPUT'

данная символьная строка в коде ASCII записывается в группу последовательных байтов, адресу первого из которых присваивается имя **MSG**. (Кавычка внутри строки представляется, как обычно, двумя кавычками.)

Область памяти резервируется с помощью псевдо **DS** (определить память). Например, псевдокоманда

W: DS 4 * KW

резервирует блок из $4 * KW$ байт. Все символы в адресном поле должны быть определены ранее, так что ассемблер располагает информацией о числе байтов, которые надо зарезервировать.

Кроме указанных псевдокоманд, в языке ассемблера Intel 8080 имеются и другие псевдокоманды. Это **ORG**, **EQU**, **SET** и **END**. Единственная их особенность заключается в том, что после метки (которая должна присутствовать в псевдо **EQU** и **SET**, но которой не должно быть в псевдо **ORG** и **END**) в псевдокомандах **EQU** и **SET** не ставится двоеточия.

11.4. ПЕРЕМЕЩАЮЩИЕ ЗАГРУЗЧИКИ И РЕДАКТОРЫ СВЯЗЕЙ

Как было отмечено выше, ассемблер составляет программу в кодах в предположении, что первая команда должна быть загружена по адресу 0. Во многих случаях возникает необходимость

задавать начальную точку загрузки. Если несколько программных секций должны быть размещены в память одновременно, то очевидно, что они не должны накладываться друг на друга. Такая ситуация может иметь место в тех случаях, когда один пользователь составляет программу из нескольких секций или когда несколько программ с различных кодирующих устройств размещаются в память одновременно. Желательно иметь возможность загружать любой сегмент программы в любое место памяти и выбирать точку, начиная с которой программа будет загружена после ассемблирования. В этом случае программу можно ассемблировать один раз и повторное ассемблирование не потребует до тех пор, пока не надо будет изменить функцию программы или поправить в ней ошибки. Наличие указанных возможностей у программиста является необходимым условием для создания библиотек, содержащих протранслированные программы.

Желательно также оставить определение точки загрузки на последний момент. В этом случае система сможет учитывать текущую обстановку при загрузке другой программы. (В программировании во многих случаях поощряют медлительность. Основное правило заключается в том, что для сохранения максимально возможной гибкости системы лучше откладывать на последний момент решение как можно большего числа вопросов.)

Сейчас мы находимся в кажущемся тупике. Ассемблер составляет программы в предположении, что она должна быть загружена, начиная с адреса 0, а мы хотим иметь возможность загружать ее в любое место. Решение проблемы заключается в том, что ассемблер должен сообщать загрузчику об изменениях, которые следует внести в программу при ее перемещении. Рассмотрим следующий сегмент программы на языке ассемблера для одноадресной машины, в которой одна команда занимает слово:

	LOAD	A
	ADD	B
	BRANCH	C
A	BSS	1
B	EQU	9900
C	STORE	A

Если этот фрагмент программы загружается, начиная с адреса 0, то значения используемых меток будут такими: **A = 3**, **B = 9900**, **C = 4**. С другой стороны, если он загружается, начиная с адреса 2300, то значения используемых меток будут следующими: **A = 2303**, **B = 9900**, **C = 2304**. Таким образом, некоторые из адресов операндов в программе должны быть изме-

нены путем сложения их с начальным адресом 2300. Если ассемблеру известно, какие метки изменяются, он может указать адреса, которые должны быть изменены. О метках, которые изменяются, можно сообщить, определяя их соответствующим образом. Те из них, которые определяются абсолютно в таких псевдокомандах, как **EQU**, не изменяются. Изменяются же метки, которые определены таким образом, что их значение зависит от способа пересчета, используемого для обозначения последовательности команд, блоков памяти или данных, т. е. метки, значения которых зависят от счетчика адресов. Метки и адреса, которые изменяются при перемещении программы, называются **перемещаемыми**. Ассемблер отслеживает все метки, используемые в программе. При определении каждой новой метки ассемблер должен не только записать её значение, но и отметить, является ли она перемещаемой или нет.

Абсолютный загрузчик, описанный в разд. 11.2, не вносит изменений в программу при ее загрузке. Если требуется перемещать программу в памяти, то надо использовать **перемещающий загрузчик**. Загрузчик этого типа модифицирует адреса операндов при загрузке программы. Для того чтобы загрузчику было известно, адреса каких операндов следует изменять, ассемблер должен сообщать об адресе каждого операнда в каждой команде, является ли он перемещаемым или нет. Это может быть сделано, если во вводимых данных загрузчика выделять дополнительный бит для каждого адресного операнда в каждом загружаемом слове. Например, в одноадресной машине, в которой каждая команда занимает одно слово, от ассемблера к загрузчику передается по одному биту на каждое слово. Этот бит сообщает загрузчику, надо или нет прибавлять начальный адрес к данному слову. Предложения из приведенного выше примера ассемблируются в следующий эквивалентный им текст:

0	LOAD	3R
1	ADD	9900
2	BRANCH	4R
4	STORE	3R

где перемещаемые адреса обозначены конечной меткой **R**. Если программа загружается, начиная с адреса **4108**, то получаем

4108	LOAD	4111
4109	ADD	9900
4110	BRANCH	4112
4112	STORE	4111

Программист должен понимать, что использование перемещаемых адресов может ограничить использование арифметических выражений в адресных полях. Описанная выше схема

допускает только положительное перемещение, т. е. величина смещения может либо один раз прибавляться к адресу, либо не прибавляться к нему вообще. Поэтому нельзя было бы записать адрес, такой, как **100 — XVAR** (где **XVAR** — перемещаемый адрес), поскольку его использование требует отрицательного перемещения. Аналогично если **A** и **B** — перемещаемые, то недопустим такой адрес, как **A + B**, поскольку для него требуется двойное перемещение. Адреса вида **A — B**, где **A** и **B** — перемещаемые, допустимы, поскольку они не требуют перемещения.

При выполнении трансляции ассемблер может записывать **перемещаемый объектный код** во внешнюю память. После завершения ассемблирования объектный код либо загружается сразу, либо остается на некоторое время на внешнем запоминающем устройстве для последующего использования. В первом случае программисту часто требуется загрузить объектные коды нескольких ассемблирований или ассемблирований и компиляций. Поэтому обычно всеми трансляторами системы генерируется объектный код в допустимом для перемещаемого загрузчика формате.

Такой прием позволяет беспрепятственно загружать в оперативную память несколько программ. Загрузчик загружает программы по очереди, запоминая для каждой из них начало размещения и длину. Обычно загрузчик загружает каждую программу в последовательные ячейки памяти, начиная с нулевой. Одна из секций кода должна быть основной программой, т. е. выполняемой в первую очередь, поскольку выполнение должно начаться с определенного места. Другие секции — это подпрограммы, вызываемые основной программой или ранее вызванными подпрограммами. Обычно, если не указан специальный знак, предполагается, что первая загружаемая программа является основной.

При написании одной из программ, скажем основной, программист не имеет сведений о расположении в памяти других подпрограмм, которые могут быть использованы. Например, при вызове из библиотеки подпрограммы, вычисляющей квадратный корень, ее расположение станет известно только после того, как будут выполнены ассемблирование и загрузка основной программы. Следовательно, должен существовать такой механизм перехода на другие подпрограммы, при котором во время ассемблирования информация об их адресах не потребуется. Существует несколько способов организации такого перехода. Ниже будут рассмотрены некоторые из них.

Когда пользователь пишет на языке ассемблера текст для одной секции, содержащей переход к другой (которая ассемблируется отдельно от первой), желательно для второй секции использовать символическое имя. Наиболее общим приемом здесь

является использование псевдо, например **CALL**, которая аналогична оператору **CALL** в некоторых языках высокого уровня. Желательно иметь возможность писать предложение

CALL SQRT

указывающее на то, что мы хотим перейти к программе с именем **SQRT**, загружаемой вместе с нашей программой. Ассемблеру не известно, какой адрес следует поместить в команде перехода, так как не известно, куда будет загружена подпрограмма **SQRT**.

Один из наиболее старых методов, используемых при решении этой проблемы, называется **методом вектора передач**. Теперь было бы правильнее называть его **методом вектора переходов**, поскольку слово передача здесь понимается в смысле перехода. Согласно методу вектора передач, ассемблер строит таблицу всех имен используемых подпрограмм и каждому из них последовательно приписывает уникальный номер, начиная с 0. Это делается при первом проходе. На втором проходе генерируется объективный код, который начинается с ячейки **N**, где **N** — число различных имен подпрограмм, найденных при первом проходе. Каждая псевдо **CALL** заменяется командой перехода по перемещаемому адресу, соответствующему имени, указанному в этом **CALL**-предложении, т. е. по адресу, расположенному между 0 и **N** — 1. Если, например, имени **SQRT** был приписан номер 3, то предложение **CALL SQRT** заменяется командой перехода по адресу 3. Это означает, что при выполнении оператора **CALL** будет осуществлен переход на одну из **N** первых ячеек данной программой секции. Необходимо по аналогии с работой загрузчика заполнить эти ячейки командами перехода по адресу, начиная с которого загружается требуемая программа. Набор таких команд перехода называется **вектором передач**. Для того чтобы загрузчик смог построить вектор передач, ассемблер сообщает ему имена требуемых программ, передавая список из **N** различных имен программ. Это показано в среднем столбце рис. 11.2. Загрузчик должен также определить, какое имя имеет каждая секция программы, с тем чтобы он смог заполнить вектор передач командами перехода. Эта информация должна быть подготовлена программистом. Для этой цели в большинстве языков ассемблера имеется псевдо **ENTRY**. Предложение

ENTRY A, B

сообщает ассемблеру, что метки **A** и **B** должны быть переданы загрузчику как имена точек входа в текущую программу. Символы **A** и **B** должны быть определены как адреса в некотором месте текущей программы. Например, фрагмент программы

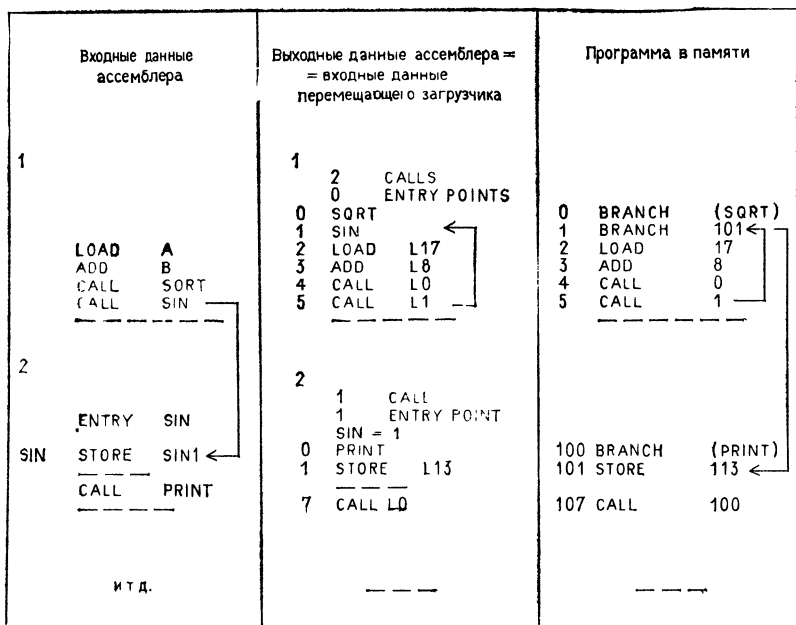


Рис. 11.2. Перемещаемый двоичный код с векторами передач.

SQRT может быть таким:

```

ENTRY SQRT
SQRT LOAD .....

```

Как правило, псевдо **ENTRY** может располагаться в любом месте программной секции, содержащей определения всех меток, используемых в этой псевдокоманде. Однако лучше помещать ее как можно ближе к началу, поскольку она несет полезную документальную информацию. Когда ассемблер обрабатывает псевдо **ENTRY**, он просто записывает имена (при первом проходе) и соответствующие адреса (при втором проходе). В конце ассемблирования он может составить для загрузчика список всех точек входа и соответствующих адресов. Это показано на рис. 11.2.

Второй подход к объединению воедино нескольких подпрограмм основан на **методе связи**. Соединитель, называемый также **редактором связей**, по существу является другим ассемблером, который получает на вход частично протранслированную программу и завершает ассемблирование. Как и при использовании метода вектора передач, во время ассемблирования некоторые символические адреса являются неопределенными, поскольку они находятся в других секциях, ассемблируемых от-

дельно. С помощью псевдо, например **CALL**, строится таблица таких меток. Так же как и при использовании метода вектора передач, эти метки остаются неопределенными в ассемблируемом сегменте. Однако в методе связи не осуществляется запись соответствующего числа из таблицы в адресное поле. Вместо этого адрес команды, содержащей неопределенную метку, сохраняется вместе с этой меткой. Такая таблица передается редактору связей, с тем чтобы он мог поместить адрес метки в команду, использующую эту метку.

Одно из преимуществ метода связи заключается в том, что при его использовании выполняется на одну команду перехода меньше. Более важным его достоинством является то, что он может быть использован не только при организации перехода на другие программы, но и при обращении к данным из других секций. Необходимо лишь, чтобы программист сообщил ассемблеру о том, что некоторая метка будет определена в другой программе, и тогда ассемблер сможет поместить эту метку в свою таблицу внешних меток. Соответствующее предложение может быть написано с помощью псевдо, такой, как **EXTERNAL**. Например, предложением

EXTERNAL SQRT, TRPDAT, XYZ

сообщается, что три указанные метки должны быть переданы загрузчику для осуществления связей. В программе, которая содержит такое предложение, можно ссылаться на адреса, используя эти метки. Например,

LOAD XYZ+6
BRANCH SQRT

и т. д. При ассемблировании адресов ассемблер заменит меткой **O** каждый внешний символ. Однако он также передаст загрузчику имена внешних меток и укажет ему все места, в которых каждая внешняя метка используется. Эти метки должны быть определены в некоторой секции программы и описаны в ней как точки входа (с помощью псевдо **ENTRY**). (В некоторых языках ассемблера для обеих целей используется описание **EXTERNAL**. Каждое имя, объявленное внешним, но, кроме того, определенное в программе, является, очевидно, точкой входа.) Во многих языках ассемблера допускаются выражения, содержащие внешние метки. Однако при этом требуется, чтобы для вычисления результата не требовалось более одного перемещения по любой заданной внешней метке. Основное правило написания программ на любом из языков ассемблера заключается в том, что в каждом выражении не должно содержаться более одного перемещаемого или внешнего адреса, за исключением случая, когда используется разность двух адресов, определенных

ВХОДНЫЕ ДАННЫЕ АСЕМБЛЕРА			ВЫХОДНЫЕ ДАННЫЕ АСЕМБЛЕРА			ПРОГРАММА В ПАМЯТИ		
Основная программа								
ENTRY X, Y EXTERNAL DSUB			ENTRY POINTS: X = 21 Y = 25 EXTERNAL REFS DSUB IN 17			...		
...				
LOAD	X		11	LOAD	L 21	511	LOAD	521
STORE	B		12	STORE	L 22	512	STORE	522
ADD	X		13	ADD	L 21	513	ADD	523
STORE	B+1		14	STORE	L 23	514	STORE	524
STORE	B+2		15	STORE	L 24	515	STORE	525
LOAD	Y		16	LOAD	L 25	516	LOAD	723
CALL	DSUB		17	CALL	0	517	CALL	525
STORE	Y		18	STORE	L 25	518	STORE	
...				
X	DEC	5	21	5		521	5	
B	BSS	3		-				
Y	DEC	4	25	4		525	4	
Подпрограмма			ENTRY POINTS DSUB = 0 EXTERNAL REFS					
EXTERNAL X, B			X IN 2, ...					
ENTRY DSUB			B IN 1, ...					
DSUB	STORE	Z	0	STORE	L 13	723	STORE	730
D1	ADD	B+2	1	ADD	2	724	ADD	524
	MPY	X	2	MPY	0	725	MPY	521
...			...					
	BRANCH	D1	12	BRANCH	L 1	735	BRANCH	724
	BSS	1		-				

Рис. 11.3. Внешние ссылки на данные и подпрограммы.

в текущем сегменте программы. Пример использования внешних переменных показан на рис. 11.3.

Результат работы (выход) редактора связей может оставаться в перемещаемой двоичной форме для последующей загрузки. Действительно, он по-прежнему может содержать **неопределенные внешние ссылки**, т. е. ссылки на еще неопределенные внешние метки, которые будут определены впоследствии при объединении с дополнительной программой. В определенный момент выход редактора связей будет преобразован в абсолютный двоичный код, готовый к выполнению. Если связь и загрузка выполняются одной программой на одном шаге, то эта программа называется **связующим загрузчиком**.

Если программист использует внешние описания для обращения к другим программным секциям, то обычно все символы

определяются где-то на входе. С другой стороны, при определении имени подпрограммы, например **SQRT**, программист может подразумевать, что она должна быть получена из библиотеки. Поэтому редактор связей при обработке входа вначале определяет, все ли внешние символы определены в каком-нибудь месте. Все, что после этого осталось неопределенным, ищется затем в библиотеке. (В больших системах, как правило, имеется много различных библиотек и пользователь имеет возможность давать запрос на поиск по нескольким библиотекам в определенном порядке.)

Редактор связей и загрузчик в системе IBM 370

В языке ассемблера системы IBM 370 для передачи информации редактору связей используются псевдо **ENTRY** и **EXTRN**. Функции загрузчика очень незначительны, поскольку система адресации с использованием базовых регистров исключает необходимость выполнения большей части работы по перемещению, которая требуется в других системах. Во время ассемблирования адресное поле команды преобразуется в форму «база + + смещение», где смещение — это положительное целое число, не превосходящее 4095. Поскольку базовые регистры загружаются динамически во время выполнения, то они содержат адрес, определяемый местом загружаемой программы. Информация ассемблеру, необходимая для вычисления перемещаемых адресов, задается с помощью поля **A** в псевдо **DC**. Если **LOOP** — это адрес программы, то псевдокоманда

```
DC A(LOOP)
```

резервирует 32-битовое слово, значением которого является адрес **LOOP** и которое поэтому является перемещаемым. Такая конструкция часто используется для внешних адресов. Если адрес **X** внешний, то команда **L R4, X** является недопустимой, так как адрес **X** при ассемблировании не может быть преобразован в форму «база плюс смещение». Вместо нее должны использоваться команды

```
EXTRN X
L      R4,AX
L      R4,0(R4)
.....
AX DC   A(X)
```

(Первая строка может быть записана в виде **L R4, = A(X)**, где используется литеральный адрес. В этом случае последняя строка может быть опущена.) При этом загрузчик будет иметь 32 бит, к которым во время загрузки он может прибавить адрес **X**.

Эта группа команд может быть также записана в виде

```
L   R4,=V(X)
L   R4,0(R4)
```

Здесь используется литерал $=V(X)$. Конструкция $V(X)$ аналогична $A(X)$, однако в ней объявляется также, что X — внешний символ.

Адресные выражения допускаются в полях A , но не в полях V . Выражение в поле A не может быть перемещаемым более чем один раз по любому адресу в сегменте программы или один раз по каждому внешнему адресу.

Загрузчик LINK-11 в системе PDP-11

Программист, работающий на языке ассемблера, определяет внешние переменные и точки входа с помощью псевдо **.GLOBL**. Так, предложением

```
.GLOBL X1, AAA, DEF
```

три символа **X1**, **AAA** и **DEF** объявляются внешними, если они не определены в этом же сегменте программы, и точками входа, если они определены. В программе могут быть использованы выражения, содержащие глобальные переменные, но результат не должен требовать более одного перемещения по внешней переменной или началу текущей программы. Так, предложения

```
          .GLOBL A,B
C=4
D:  MOV    R1,R2
E:  ADD    R1,R3
     MOV    #A,D-E(R2)
     MOV    D+C,A+C
```

являются допустимыми, поскольку каждый адрес является либо абсолютным, либо однократно перемещаемым. Согласно данным выше определениям символических адресов, команда **MOV D + E, A * B** содержит две ошибки. Адрес **D + E** является дважды перемещаемым относительно начала текущей программной секции, а выражение **A * B** является произведением перемещаемых адресов.

Загрузка в системе Cyber 170

Внешние символы и точки входа описываются в псевдо-командах **EXT** и **ENTRY**. В адресных выражениях могут использоваться внешние или внутренние перемещаемые символы

при условии, что после вычислений будет получено либо абсолютное значение, либо адрес, положительно перемещаемый по одному внешнему символу, либо адрес, положительно или отрицательно перемещаемый по адресу одного внутреннего блока. Например, выражение

3*ALPHA—2*BETA

является допустимым, если адреса **ALPHA** и **BETA** либо оба абсолютные, либо оба перемещаемые по одной и той же величине (в последнем случае это выражение является однократно положительно перемещаемым).

Загрузка в системе Intel 8080

В системе Intel 8080 ассемблер основан на принципе «загрузить и выполнить». Таким образом, в простейших системах не существует отдельного загрузчика.

11.5. БИБЛИОТЕКИ

Для того чтобы избежать загрузки многих программ с устройств ввода, в вычислительных системах обычно имеются библиотеки программ общего пользования. Как следует из названия, библиотека — это не более чем набор программ, которые могут быть использованы в других программах. Чтобы сделать этот набор полезным и доступным, его размещают на быстром внешнем запоминающем устройстве, например на диске, и пользователь для загрузки может получить копию любого раздела. Загрузчик вначале объединяет секции программы пользователя, связывая их с помощью одного из описанных выше методов. Если некоторые из внешних имен не определяются в программе пользователя, то загрузчик пытается найти библиотечную программу с тем же именем. Он просматривает каталог, или указатель, имен всех программ в библиотечном файле. Когда программа найдена, она считывается загрузчиком и связывается с программой пользователя. Это означает, что программы в библиотеке могут храниться в оттранслированном виде и их не надо ассемблировать заново каждый раз при обращении к ним. Во многих системах пользователь может специфицировать набор библиотек. Например, группа пользователей химического института организует свою групповую библиотеку программ химического анализа, с которой она работает до обращения к библиотеке общего пользования. Химики дают директиву машине проводить вначале поиск в их библиотеке. Некоторые пользователи могут также иметь свою личную библиотеку и сообщают машине, чтобы поиск велся вначале в ней,

а затем в групповой библиотеке. Программы, общие для всех, например **SQRT**, хранятся в библиотеке общего пользования. Если более чем в одной библиотеке имеются программы с одним и тем же именем, то загрузчик выбирает программу из первой библиотеки, в которой она была найдена. Это указывает на очень важный принцип: пользователь может не знать всех имен программ в существующих библиотеках. Если некоторое задание содержит внешнюю метку, которая определяется в другой части этого же задания, то загрузчик вообще не ищет его в библиотеке. Если в задании есть обращение к программе из личной библиотеки пользователя, то загрузчик будет искать программу в этой библиотеке и не будет проводить дальнейшего поиска. Таким образом, программист может использовать некоторые имена библиотечных программ, хотя надобности в этих программах нет. Определение в программе пользователя будет учитываться в первую очередь. Это также позволяет пользователю завести копии библиотечных программ, которые будут загружаться вместо системных.

11.6. ДРУГИЕ ТРАНСЛЯТОРЫ

В состав математического обеспечения ЭВМ обычно входит несколько трансляторов с языков высокого уровня. Все они выполняют одну и ту же функцию — переводят исходную (входную) программу в объектную (выходную) программу в двоичном коде, которая понятна для ЭВМ. Некоторые трансляторы переводят исходную программу на язык ассемблера, оставляя полученный текст ассемблеру для перевода его в двоичную форму. Другие переводят в двоичный код непосредственно.

Крайне важно, чтобы все трансляторы, являющиеся частью одной системы, вырабатывали совместимый объектный код. Тогда во всех случаях может использоваться один загрузчик и программы, написанные на одном языке, могут вызывать программы, написанные на другом языке. Иногда пишутся компиляторы, работающие по принципу «загрузить и выполнить». Проектировщики компиляторов стоят перед выбором между быстрой трансляцией и быстрым выполнением (в некоторых случаях, по-видимому, ни одна из этих целей не достижима). Компиляторы, работающие по принципу «загрузить и выполнить», обычно бывают очень быстрыми в плане компиляции. Если имеется достаточный объем памяти, то компилятор может храниться в ней и использоваться для обработки нескольких программ. В этом случае компилятор не надо считывать в оперативную память перед каждой компиляцией. Загрузка компилятора из внешней памяти занимает значительную часть времени работы коротких программ, так что система, работающая

по принципу «загрузить и выполнить», может быть очень эффективной для небольших заданий, например для отладки программ.

11.7. УПРАВЛЕНИЕ ВЫПОЛНЕНИЕМ ПРОГРАММ

Даже очень аккуратно написанная программа содержит ошибки различного рода. Программист может составить неверный алгоритм, например написать $A \rightarrow B$ вместо $B \rightarrow A$. Ошибки такого рода могут быть найдены путем сравнения полученных ответов с результатами вычислений по альтернативному методу или путем проверки **инвариантов** задачи. Инвариант — это выражение или факт, которые не меняются. Например, в физике полная энергия системы плюс поступающая энергия минус потерянная энергия есть инвариант. В задачах сортировки число данных есть инвариант. Можно показать, что эти величины не изменяются (хотя в числовых задачах ошибки округления могут вызвать небольшие изменения). Ошибки числового характера не могут быть обнаружены системой программирования, хотя некоторая помощь программисту оказывается, например обнаруживается переполнение и выдается соответствующее сообщение.

Существуют ошибки, вызванные неверной записью алгоритма на исходном языке или неправильным использованием библиотечных программ. Часто это приводит к непредсказуемым результатам. Например, программа может попытаться интерпретировать содержимое некоторой ячейки как команду, хотя в нее команда никогда не загружалась, а записаны числовые данные. Программа может также заикнуться. Всегда, когда возникает одна из перечисленных ситуаций, программист должен получить помощь. Как только программа пытается выполнить запрещенную команду, пользователь должен быть оповещен об этом. Если прошло слишком много времени с момента запуска программы, то ее выполнение должно быть прекращено, а пользователю надо выдать результаты работы программы, полученные до останова.

Для того чтобы следить за всем этим, выполнение программы контролируется другой программой, а также аппаратными средствами с помощью захватываний и прерываний. Эта другая программа носит название **монитора**, что отражает ее функцию. Ее часто называют **супервизором** или **исполнительной программой**. Механизмы захватывания и прерывания позволяют при возникновении различных исключительных ситуаций передавать управление в определенные участки памяти. Когда программа-монитор (размещаемая вблизи области, в которую передается управление при захватывании и прерывании)

обнаруживает запрещенную команду, она обращается к программам, позволяющим определить расположение этой команды, сообщить его пользователю, а также распечатать участки оперативной памяти, которые могут представлять интерес для пользователя. Перед запуском новой программы монитор засылает в таймер количество времени, заказанное пользователем для ее выполнения. При обработке прерывания по таймеру, которое указывает на истечение запрошенного времени, монитор печатает соответствующее сообщение пользователю, завершает выполнение программы и, если это было указано, выдает дамп.

Средства защиты памяти позволяют объявлять некоторые участки памяти запрещенными для программы пользователя. Любая попытка обратиться к этой области памяти приведет к захватыванию и передаче управления монитору. Таким образом, программе пользователя можно запретить запись в область памяти, к которой эта программа не имеет отношения, и чтение из этой области. До начала выполнения программы монитор устанавливает для нее границы памяти. В некоторых системах средства защиты памяти реализованы отдельно для записи и чтения. Это позволяет хранить в некоторых областях памяти системные данные или программы, доступные для всех пользователей, но которые могут быть изменены только авторизованными программами. Наиболее часто такой подход используется в системах с разделением времени.

Основное назначение защиты памяти двояко:

1. Она помогает при обнаружении потенциальных ошибок в программе пользователя.

2. Она способствует тому, чтобы программам пользователя память выделялась только в случае необходимости.

Защита памяти как при чтении, так и при записи используется для обнаружения ошибок. Очевидно, что попытка записать в область памяти, расположенную за пределами зарезервированного для программы диапазона адресов, является ошибкой. Таким образом, защита памяти при записи может быть использована для обнаружения всех адресов, используемых в командах записи и выходящих за допустимый диапазон. Нет необходимости использовать защиту памяти при чтении как ограничитель, так как по вполне понятным причинам программе пользователя можно разрешить считывать информацию из некоторых ячеек, например содержащих время суток или дату. Однако поскольку всегда существует необходимость надежного сохранения данных пользователей системы, то следует запретить считывание данных, принадлежащих другим пользователям. Средства защиты памяти при считывании и записи также применимы для предотвращения доступа программ к областям памяти, использующимся в текущий момент для ввода и вывода

данных. Поскольку при вводе и выводе осуществляется обмен между оперативной памятью и другими устройствами, то очевидно, что могут появиться ошибки, если пользователь будет считывать или изменять содержимое памяти, используемой для выполнения операций ввода-вывода.

Далее возникает проблема выделения памяти. Для современных операционных систем характерно, что в оперативной памяти одновременно находится несколько программ пользователей. Хотя каждый пользователь ничего не знает о других, в любой момент времени для одной из программ может не хватить требуемого объема оперативной памяти. Однако каждая программа пользователя запрашивает необходимый объем памяти, когда это ей требуется. Если достаточного объема оперативной памяти нет, то возникает прерывание как следствие использования недействительных адресов. Это прерывание указывает монитору на то, что прерывающей программе необходимо выделить больше памяти, как только появится свободная память.

Монитор хранится в защищенной области памяти. Чтобы монитор мог обращаться к любым участкам памяти, определяются различные уровни защиты, называемые **режимами** в одних системах и **кольцевыми уровнями** в других. В **режиме супервизора** программа может обращаться к любому участку памяти. Следовательно, монитор работает в режиме супервизора. В **режиме пользователя** применяется защита. Таким образом, программы пользователя работают в режиме пользователя. Прерывания и захватывания вызывают переход на другой режим, обычно на режим супервизора. (Между двумя указанными уровнями защиты могут существовать и другие. Они могут использоваться при выполнении некоторых системных программ, не нуждающихся в доступе ко всей памяти. Это способствует обнаружению ошибок в системных программах, которые возникают редко и довольно неожиданно.)

Поскольку данные и программы многих пользователей хранятся на внешних запоминающих устройствах и в любой момент могут вводиться и выводиться, то операции ввода-вывода также должны быть защищены от неавторизованного пользователя. В большинстве систем команды ввода-вывода являются **привилегированными** и могут выполняться только в режиме супервизора.

Многие функции монитора и связанные с ним средства защиты носят негативный характер и предназначены для того, чтобы ввести некоторые ограничения на работу пользователя, например запретить использовать слишком много времени. Однако монитор предоставляет также пользователю и ряд услуг. Основные из них связаны с управлением устройствами ввода-

вывода. Ниже мы остановимся на трудностях работы с этими устройствами. Монитор же обычно предоставляет набор подпрограмм, которые упрощают их использование. Другие программы монитора используются для того, чтобы при возникновении исключительных ситуаций, которые пользователь пожелает контролировать, выполнять специальные действия. Например, пользователь по желанию может обратно получить управление после переполнения, с тем чтобы внести изменения в программу.

Монитор часто предоставляет дополнительные средства пользователю при захватывании, возникающем вследствие обнаружения ошибки. Он позволяет обращаться к процедурам, которые после определенного типа остановов программы выдадут содержимое занятой пользователем области оперативной памяти. Другие средства отладки обычно обеспечиваются системными программами, которые функционируют как подпрограммы в области пользователя. Среди них имеются программы трассировки, которые печатают значения указанных переменных каждый раз, когда выполняются определенные команды. При использовании программы трассировки предполагается, что во время ассемблирования или компиляции программе присвоили свойство «трассируемая». При этом выдается некоторая информация, получаемая во время выполнения. Однако существуют системы трассировки более низкого уровня. Они позволяют пользователю вставлять в загрузочный модуль команды, осуществляющие перехват управления. Эти системы трассировки используют копии таблицы имен, построенной транслятором, и копии **плана памяти**, представляющего собой таблицу распределения памяти, построенную связующим загрузчиком. С помощью этой информации программа трассировки преобразует имя в адрес памяти, который будет присвоен ему во время выполнения, и поэтому пользователь может запрашивать перехват управления в различных местах программы. Эта символьная информация может быть также использована для выдачи дампа памяти, содержащего значение каждой используемой переменной вместе с ее именем. Иногда удобно периодически получать несколько коротких дампов указанных областей памяти (они называются **снимками**). Полезным также бывает сравнительный «**посмертный**» дамп. С его помощью можно сравнить состояние памяти после выполнения с начальным состоянием и определить изменившиеся ячейки.

Интерпретация

Интерпретация может быть использована для увеличения возможностей ЭВМ. Одним из путей достижения этого является

совместное ее использование с механизмом перехвата, что дает пользователю возможность использовать дополнительные машинные команды. Например, в большинстве систем ввод и вывод управляются монитором. Пользователь обращается к этим процедурам посредством «выполнения» определенных «команд», которых на самом деле не существует. Эти команды вызывают прерывание, и управление передается монитору, который выполняет операции ввода-вывода. Для определения причины возникновения прерывания монитор исследует данные в области памяти, принадлежащей пользователю. Например, ассемблер может обрабатывать команду вида

READ A,10,IN3

означающую, что нужно считать 10 слов в ячейки с **A** по **A+9** из входного файла **IN3**. Ассемблер протранслировал бы ее в некоторую несуществующую команду, в которой адрес операнда задает расположение блока из трех слов, содержащих данные **A**, **10** и **IN3**. Во время выполнения эта несуществующая команда вызовет прерывание и будет исследована монитором. Монитор распознаёт, что данная несуществующая команда соответствует операции чтения, и выполнит эту операцию с помощью данных **A**, **10** и **IN3**. С точки зрения машины операция **READ** является интерпретируемой. С точки зрения пользователя команду **READ** выполняет система. Таким образом, пользователь видит машину более высокого уровня, называемую **виртуальной машиной**.

Интерпретация также очень полезна при поиске ошибок в программе. В этом случае интерпретатор интерпретирует каждую машинную команду в программе. Интерпретатор машинного языка по сути тот же, что и интерпретатор языка высокого уровня. Интерпретирующая программа работает точно так же, как вычислительная машина, извлекая из памяти последовательные команды интерпретируемой программы, исследуя их и выполняя указанные действия. Например, если извлекается команда **STORE**, интерпретатор засылает в ячейку памяти с указанным адресом содержимое ячейки, используемой для того, чтобы сохранить для интерпретируемой программы содержимое сумматора. Однако интерпретатор может выполнять и другие действия. Если ячейка не принадлежит области данных программы, то печатается сообщение об ошибке. (Если интерпретатор располагает таблицей областей данных, определенной программой, то он имеет информацию для выполнения такой проверки.) Как правило, это позволяет выявить большее число ошибок по сравнению со случаем, когда используются средства простой защиты памяти. Интерпретатор может также хранить список ячеек, за которыми программист хотел бы

следить. Пользователю может быть также выдано сообщение о любой операции записи или чтения с использованием этих ячеек.

11.8. КОМАНДЫ УПРАВЛЕНИЯ ЗАДАЧАМИ И ЗАДАНИЯМИ

Как было показано выше, системные программы предназначены для трансляции и загрузки программ пользователя, а также для управления ими. В некоторых старых системах пользователь или оператор должны были последовательно выполнять отдельные задачи, которые составляли единое задание для ЭВМ. Например, если задание состояло из секции, написанной на Фортране, и секции, написанной на языке ассемблера, и требовало три библиотечные подпрограммы, то оператор старой ЭВМ выполнял следующую последовательность действий:

1. Загрузка компилятора с Фортрана. Это делалось с помощью абсолютного загрузчика, который либо хранился в памяти, либо загружался последовательностью команд начальной загрузки.

2. Компиляция секции, написанной на Фортране. Выход хранился для последующей загрузки.

3. Загрузка ассемблера (так же как загружался компилятор).

4. Ассемблирование следующей секции программы. Сохранение выхода для загрузки.

5. Загрузка перемещаемого загрузчика с помощью абсолютного загрузчика.

6. Засылка выхода компилятора и ассемблера в загрузчик. Поскольку потребовались три подпрограммы, загрузчик сообщил оператору их имена. Необходимо было получить копии этих программ и передать их перемещаемому загрузчику.

7. Инициация выполнения программы. В случае ее останова должны были выполняться действия, указанные пользователем. Например, если требовался дамп, то необходимо было загрузить программу выдачи.

Такой процесс выполнялся на старых системах медленно по двум причинам. Во-первых, промежуточная память находилась на носителях твердой копии, например на перфокартах или перфолентах, обмен с которыми осуществлялся очень медленно. Во-вторых, инициировать каждый шаг должен был оператор. При появлении внешних запоминающих устройств, которые в настоящее время имеются почти во всех системах, отпала необходимость в использовании носителей твердых копий для промежуточной памяти и хранения системных программ и появилась возможность автоматически выполнять последовательность всех действий на ЭВМ. Вместо того чтобы команды,

написанные пользователем и указывающие требуемые шаги, передавать оператору, их следует давать вычислительной машине. Поэтому необходимо иметь еще одну системную программу, которая должна обрабатывать эти команды. И не удивительно, что она называется **интерпретатором команд**, а предложения, которые формирует пользователь, пишутся на языке, называемом **командным языком**. (По терминологии операционной системы 370 этот язык называется **языком управления заданиями**, а предложения — **предложениями управления заданиями**. К сожалению, язык управления заданиями является одним из самых плохих среди существующих языков, предназначенных для использования вычислительной машины и управления ею. Поэтому мы не будем здесь его рассматривать.) Когда пользователь составляет задание, он должен добавить к программам и данным соответствующие команды, которые сообщают системе все то, что она должна выполнить. Установки ЭВМ обычно содержат описания наборов простых команд, необходимых для выполнения стандартных операций, и пользователю рекомендуется познакомиться с этими командами. Часто они так или иначе зависят от конкретной машины. Мы рассмотрим простой пример командного языка, который похож на язык, используемый в некоторых вычислительных машинах Cyber. Однако он отражает характерные черты всех современных командных языков.

Командный язык аналогичен любому другому языку. Для того чтобы сообщить системе, какую выполнять работу, в нем используются операторы и операнды. Операторами являются запросы на компиляцию, ассемблирование, загрузку и т. д., а операндами — программы и данные. Программы и данные хранятся на внешних запоминающих устройствах как поименованные файлы, а их имена используются в этих командах. Например, в команде

FORTRAN,PROG1,OBJECTA

указано, что надо выполнить компилятор **FORTRAN**, на вход которого должен поступить набор данных **PROG1**. Выход (перемещаемый модуль в двоичной форме) следует поместить в файл с именем **OBJECTA**. (Мы опустили рассмотрение листинга, выдаваемого большинством компиляторов, так что в команде могло потребоваться имя дополнительного файла.) Если следующей указана команда

ASSEMBLE,PROG3,OBJECTA

то вычислительной машине сообщается, что надо выполнить ассемблирование программы из файла с именем **PROG3** и перемещаемый двоичный модуль также поместить в файл

ОБЪЕКТА. Если этот модуль размещается вслед за ранее записанными данными, то файл **ОБЪЕКТА** будет содержать перемещаемые двоичные модули **PROG1** и **PROG3**. Далее мы могли бы написать команду

LOAD,ОБЪЕКТА,МУJOB

по которой выполнялась бы загрузка данных из файла **ОБЪЕКТА** и в файл с именем **МУJOB** был бы записан абсолютный двоичный код, готовый к выполнению. Наконец, команда

МУJOB,INPUT,OUTPUT

инициировала бы выполнение только что подготовленной программы, входом для которой является файл **INPUT**, а выходом — файл **OUTPUT**. Если бы нам понадобилось сохранить скомпилированную программу для последующего использования, мы могли бы воспользоваться командой

SAVE,МУJOB

При выполнении этой команды на диске была бы создана постоянная копия файла **МУJOB**, которую можно использовать в дальнейшем.

Формат рассмотренных выше команд следующий:

Оператор входной файл, выходной файл

Было бы хорошо, если бы все выполнялось так просто. На практике возникает необходимость указывать имена других файлов и некоторую дополнительную информацию, однако мы не будем здесь усложнять картину. Поскольку необходим большой объем информации, в системе предусмотрен стандартный выбор по умолчанию. Например, в системе Cyber предполагается, что если имена входного и выходного файлов не указаны, то входной файл поступает со стандартного устройства ввода, а выходной файл записывается на стандартное устройство вывода. Если программист использует терминал, то такими стандартными устройствами являются клавиатура и экран диалогового дистанционного терминала или устройство печати, а если он пропускает задание в пакетном режиме, то устройства ввода и печати.

Пользователь может вводить эти команды с терминала, работающего в интерактивном режиме, в тот момент, когда они должны быть выполнены. Однако удобно также иметь возможность создать файл команд и выполнять их без дальнейшего участия человека. Это бывает полезно не только при запуске задания в пакетном режиме, когда все задание оставляется для дальнейшей обработки, но и при работе в интерактивном режиме, когда пользователю надо неоднократно использовать

один и тот же набор команд для различных наборов программ и данных.

Вначале рассмотрим ввод пакетного задания. Первые строки в пакетном задании обычно связаны с идентификацией и учетом. Последующие строки содержат команды, а далее следуют программы и данные. В приведенном выше примере требовалось скомпилировать первую часть задания, написанную на Фортране, выполнить ассемблирование следующей части, загрузить два объектных модуля вместе с библиотечными командами и, наконец, полученный в результате модуль выполнить с использованием данных, следующих за программой. В некоторых системах перед каждым командным оператором ставится специальный знак (в языке управления заданиями используются два знака //), в других системах все команды просто группируются вместе перед программой и данными. Мы остановимся на рассмотрении второго случая, хотя первый аналогичен ему. Типичная структура задания для приведенного выше примера выглядит следующим образом:

Информация идентификации (имя пользователя, учетный номер, пароль и т. д.)

FORTRAN,OBJECTA

ASSEMBLE,OBJECTA

LOAD,OBJECTA,MYJOB

MYJOB,,

знак конца записи

Исходная Фортран-программа

знак конца записи

Исходная программа на языке ассемблера

знак конца записи

Данные для прогона

знак конца файла

В первых двух командах после идентификации отсутствует имя входного файла. Поэтому предполагается, что ввод должен осуществляться со стандартного устройства ввода, которым является устройство, считывающее этот поток данных. Следовательно, Фортран-программа является первой записью, следующей за записью, содержащей команды. (Здесь используется терминология системы Cyber. Запись — это набор данных, заканчивающийся указателем конца записи. Она может состоять из нескольких образов карт.) Программа, которая должна ассемблироваться, является записью, следующей за Фортран-программой. Наконец, данные, которые используются на шаге выполнения, являются последней записью. Выход, полученный на шаге выполнения, поступает на стандартное устройство вывода, так как файл не указан. В самом конце стоит знак конца файла.

Мы видим, что набор команд — это просто другой тип программы. Это программа, которая выполняется интерпретатором команд. Ее операторы выполняются последовательно, точно так же как команды программы, написанной на машинном языке. Нет причины, по которой в этих управляющих программах нельзя было бы иметь управляющих операторов, и во многих системах такие операторы существуют. Основными из них являются условный оператор

```
IF, выражение
.....
ELSE
.....
ENDIF
```

который исследует значение логического выражения и определяет, какую группу команд надо выполнить, и оператор цикла

```
WHILE, выражение
.....
ENDWHILE
```

обеспечивающий повторение выполнения группы команд до тех пор, пока значение логического выражения истинно. В следующем примере показано, как могут быть использованы эти операторы. Рассмотрим процесс обработки файла **STUDJOBS**, который состоит из нескольких записей. Каждая запись представляет собой Фортран-программу, написанную студентом в аудитории. Задача заключается в том, чтобы скомпилировать каждую из них, выполнить каждое задание, прошедшее компиляцию, с использованием данных из файла **TEST** и затем проверить выход с помощью программы **GRADE**, которая записывает рассортированную информацию в файл с именем **CLASSGRADE**. Если программа не проходит компиляцию, для указания ошибки должна быть использована программа **DIAGNOSE**. Команда **REWIND** располагает файл, начиная с первого данного. Логическое выражение **FILE (имя, NO_EOF)** позволяет для файла с указанным именем проверять условие конца файла. Выражение **ERRORFLAG** истинно, если на предыдущем шаге (Фортран) была обнаружена ошибка.

```
REWIND,STUDJOBS
WHILE,FILE(STUDJOBS,NO_EOF)
  FORTRAN,STUDJOBS,OBJECT
  IF,ERRORFLAG
    DIAGNOSE,OBJECT,CLASSGRADE
  ELSE
    REWIND,TEST
  REWIND,OUT
```

```
LOAD,OBJECT,RUNFILE  
RUNFILE,TEST,OUT  
REWIND,OUT  
GRADE,OUT,CLASSGRADE  
ENDIF  
ENDWHILE
```

(Многие интерпретаторы не допускают смещения команд при их написании, как это было сделано выше. Мы воспользовались этим приемом для того, чтобы показать читателю структуру программы.)

Командная программа, как, например, приведенная выше, может быть записана в задании или храниться в файле. В последнем случае она может содержать параметры, так же как подпрограмма в любом языке. Например, имена файлов **STUDJOBS**, **TEST** и **GLASSGRADE** могли быть переданы как параметры. Тогда этот командный файл мог бы быть использован для нескольких классов. Командные языки позволяют также программисту посылать оператору сообщения с требованиями смонтировать ленту или другие запоминающие устройства.

Система обрабатывает последовательности заданий. При завершении каждого задания система определяет, какое задание выполнять следующим. В простейшей системе пакетной обработки выбирается очередное задание, находящееся в ожидании на устройстве ввода с перфокарт или диске. В сложных системах мультипрограммирования программа-планировщик исследует задания, находящиеся в ожидании, и выбирает из них одно в соответствии с некоторой комбинацией приоритетов, длиной задания, а также в зависимости от того, насколько легко оно может выполняться при текущем состоянии системы.

ОБЗОР ЯЗЫКОВ ПРОГРАММИРОВАНИЯ ВЫСОКОГО УРОВНЯ

Г. Хелмс

12.1. ВВЕДЕНИЕ

Языки высокого уровня являются средством, с помощью которого можно вычислить заданную функцию или решить некоторую задачу, преобразуя их в множество машинных инструкций. Команды, написанные на языке высокого уровня, должны быть преобразованы в машинные команды с помощью вычислительной системы, на которой будет выполняться программа. В этом случае «транслятор» называется **компилятором**. Программа, написанная на языке высокого уровня, называется **исходной программой**, а полученная в результате компиляции программа на машинном языке — **объектной программой**. Сам компилятор является программой. В некоторых вычислительных системах (особенно в микроЭВМ) вместо компилятора используется **интерпретатор**. Интерпретаторы транслируют команды исходной программы по одной. В результате интерпретаторы генерируют объектные программы медленнее, чем компиляторы. Однако интерпретаторы требуют меньше памяти, чем компиляторы.

Языки высокого уровня иногда описываются как **машинно-независимые**. Это означает, что программа, написанная на языке высокого уровня, например на Коболе, должна выполняться на любой вычислительной системе, имеющей компилятор с Кобола. На практике, однако, существует несколько вариантов каждого языка высокого уровня, так что ни один из них не является полностью «переносимым» на все вычислительные системы, имеющие компилятор для данного языка. В последние годы много внимания уделяется вопросу стандартизации языков высокого уровня.

По сравнению с машинным языком или языком ассемблера языки высокого уровня значительно упрощают программирование и позволяют уменьшить количество ошибок в программах. Программа, написанная на языке высокого уровня для одной машины, с небольшими изменениями или вообще без изменений

может быть использована на других машинах. Однако программы на языках высокого уровня выполняются медленнее и занимают больше памяти, чем программы на машинном языке или языке ассемблера.

12.2. РАЗВИТИЕ ЯЗЫКОВ ВЫСОКОГО УРОВНЯ

Первым языком высокого уровня, получившим широкое распространение, был Фортран (сокращенное название от Formula TRANslator, т. е. переводчик формул). Первое описание этого языка было опубликовано в 1954 г. Впервые он был использован в 1956 г. на вычислительной машине IBM. Руководителем группы, разработавшей Фортран, был Джон Бэкус, который работал в то время в фирме IBM. Позднее он участвовал в разработке формального метода для определения синтаксиса языков программирования, который получил название «нотации Бэкуса — Наура» (BNF).

В 1958 г. состоялась встреча представителей Ассоциации по вычислительным методам и Европейских обществ по вычислительной технике. В результате был разработан язык Алгол (ALGOrithmic Language, т. е. алгоритмический язык). Алгол, так же как и Фортран, был эффективным средством для решения широкого круга математических задач. В отличие от Фортрана Алгол не был поддержан фирмами — производителями ЭВМ в Соединенных Штатах. Поэтому Алгол получил гораздо большее распространение в Европе, чем в Соединенных Штатах.

В 1959 г. министерство обороны США при содействии других правительственных учреждений начало разработку общего языка для коммерческих задач. В результате был разработан язык Кобол (COmmon Business Oriented Language, т. е. общий язык, ориентированный на решение коммерческих задач). В отличие от Фортрана и Алгола Кобол дает возможность составлять более удобочитаемые программы, которые часто бывают понятными непрограммисту. В программах на Коболе особенно проявляется самодокументируемость, что облегчает их исправление и усовершенствование.

В 1965 г. появились два новых важных языка. Профессорами Университета в г. Дартмуте Томасом Куртцем и Джоном Кемени был разработан язык для обучения программированию, который явился упрощенной версией Фортрана и получил название языка Бейсик (Beginner's All-purpose Symbolic Instruction Code, т. е. многоцелевой код символических команд для начинающих). Бейсик предоставляет разнообразные средства для диалога: пользователь имеет возможность «общаться» с Бейсик-программой во время ее выполнения. Программа может

«попросить» пользователя ввести данные, проверить их, осуществить выбор и т. д. Наибольшее признание Бейсик получил с появлением микроЭВМ начиная с 1975 г. Сейчас Бейсик вполне можно назвать наиболее известным и широко распространенным в мире языком высокого уровня.

Вторым языком, появившимся в 1965 г., был ПЛ/1 (Programming Language 1, т. е. язык программирования 1). ПЛ/1 был разработан при содействии фирмы IBM. При этом преследовалась цель создать язык, сочетающий в себе лучшие стороны Алгола, Кобола и Фортрана. ПЛ/1 действительно подходит для решения широкого круга задач и является более гибким по сравнению с Алголом, Коболом и Фортраном. При появлении языка ПЛ/1 некоторые предсказывали, что он станет основным языком и в конечном счете заменит Кобол и Фортран. Конечно, этого не случилось. Главная причина, по-видимому, связана с тем, что некоторые пользователи Кобола и Фортрана не ощутили тех преимуществ языка ПЛ/1, которые оправдывали бы переход к нему. Другая причина связана с «размерами» языка. Большое количество средств и разнообразие операторов ПЛ/1 приводят к тому, что для его изучения требуется больше времени по сравнению с Коболом и Фортраном.

В 1971 г. профессор Никлаус Вирт из Технического университета в Цюрихе, Швейцария, разработал новый важный язык, известный под названием «Паскаль» (в честь хорошо известного математика XVII века Блеза Паскаля). Язык Паскаль основан на Алголе, но содержит ряд усовершенствований. Средства манипулирования нечисловыми данными в языке Паскаль намного совершеннее, чем в Алголе. Язык Паскаль предоставляет также более удобные средства ввода-вывода, позволяет легко обрабатывать структуры данных (такие, как списки и таблицы), и, кроме того, пользователь может определить свои собственные типы данных. Так же как и Алгол, язык Паскаль является **языком с блочной структурой**. Это означает, что для определенных целей программы разбиваются на секции, или блоки, и что можно вносить изменения в один блок, не касаясь других.

Новым языком, который обещает получить широкое распространение, является язык Ада. Он был разработан в 1979 г. по инициативе министерства обороны США с целью заменить многочисленные языки, используемые американскими вооруженными силами и их поставщиками. Основанный на языке Паскаль язык Ада является его расширением и включает в себя новые элементы. Особенно он подходит для разработки систем реального времени.

12.3. ОПИСАНИЕ ЯЗЫКОВ ВЫСОКОГО УРОВНЯ

В настоящее время существует большое число языков высокого уровня, и каждый год появляются новые языки (хотя немногие из них действительно реализованы и в какой-то степени используются). Конечно, здесь невозможно перечислить все языки высокого уровня. Ниже будет дано краткое описание наиболее важных из них.

Ада. Ада — сравнительно новый язык, основанный на языке Паскаль. Так же как и язык Паскаль, он является существенно структурированным языком. Основное преимущество перед языком Паскаль заключается в возможности его использования в системах реального времени. Программы на языке Ада состоят из модулей; модули — из более мелких единиц, называемых пакетами и процедурами. Ада — достаточно «широкий» язык. В плане изучения он относится к языкам средней трудности.

Алгол. Алгол был первым языком с блочной структурой. Программа на Алголе представляет собой один-единственный оператор (который называется блоком), описывающий шаги, необходимые для выполнения требуемого действия. Начало и конец каждого блока указываются явно. Блоки могут быть вложенными друг в друга. Однако Алгол располагает бедными средствами ввода-вывода.

АПЛ. АПЛ (A Programming Language, т. е. язык программирования) предоставляет большие возможности для работы в диалоге и несложен для изучения. Он отличается простым синтаксисом, большим числом операторов и усовершенствованными структурами данных. АПЛ широко используется в образовании, но редко при решении коммерческих задач.

Бейсик. В настоящее время Бейсик широко используется в микроЭВМ. Первоначально Бейсик был разработан как язык для обучения программированию. Он является упрощенной версией Фортрана и несложен для изучения. По сравнению с большинством других языков высокого уровня он обладает ограниченными возможностями, однако в настоящее время используется много различных, несовместимых друг с другом реализаций Бейсика.

Си. Язык Си первоначально был разработан для мини-ЭВМ, использующих операционную систему UNIX. Он является относительно простым языком; например, в нем нет операций над символьными строками и списками. Язык Си находит широкое применение при написании программ вычислительного характера и программ операционных систем. Он является высокоэф-

фективным языком в плане скорости выполнения программы и необходимой памяти.

Си-Бейсик. Си-Бейсик является модификацией Бейсика и был разработан с целью способствовать более широкому использованию структурного подхода к программированию. Наиболее заметное отличие между этими языками заключается в том, что в Си-Бейсике операторы можно не нумеровать, если только на них не ссылаются другие операторы. Си-Бейсик используется главным образом при решении коммерческих задач.

Кобол. В Коболе особое место отведено таким понятиям, как запись, файл и описание поля, и предусмотрены широкие возможности для манипулирования данными. Кобол-программа состоит из различных разделов (например, раздела оборудования и раздела данных), каждый из которых выполняет определенную функцию.

Форт. Форт позволяет пользователю определять свои собственные команды, функции и процедуры. Форт успешно применяется в микропроцессорных системах для решения разнообразных задач, например при управлении электродвигателями и в телеиграх.

Фортран. Фортран широко применяется при решении математических и научных задач. Он предоставляет пользователю большие возможности для обработки числовых данных (особенно комплексных чисел), но располагает бедными средствами для работы с символьными строками. В плане изучения и использования он относится к языкам средней трудности.

Лисп. Лисп (LISt Processing, т. е. обработка списков) предназначен для обработки строк и рекурсивных данных. Лисп располагает также средствами для выполнения арифметических и логических операций. Он находит широкое применение в исследованиях по созданию искусственного интеллекта.

Паскаль. Паскаль — это широко используемый язык, основанный на Алголе. Он является языком с блочной структурой и способствует структурному подходу к программированию. Располагает большим набором управляющих операторов и позволяет пользователю определять свои собственные типы данных.

ПЛ/1. ПЛ/1 сочетает в себе многие черты Алгола, Кобола и Фортрана. Он применим к решению большого числа задач. Однако ПЛ/1 располагает большим числом средств, что делает его весьма сложным для изучения и использования.

РПГ. РПГ (Report Program Generator, т. е. генератор отчетов) включает многие понятия и выражения, связанные с ма-

шинными методами составления отчетов. Он имеет ограниченную область применения и используется главным образом для печати отчетов, записанных в одном или нескольких входных файлах. Многие системы, располагающие языком РПГ, имеют также другой язык высокого уровня, применимый к решению задач, для которых РПГ не подходит.

Снобол. Снобол был разработан в 1962 г. Bell Laboratories. Он располагает мощными средствами манипулирования строками и сравнения с образцом. Снобол используется главным образом при обработке текстов.

12.4. РЕЗЮМЕ

Нетрудно заметить, что не существует языка высокого уровня, который был бы идеальным для всех случаев. Наиболее важная задача, по-видимому, заключается в том, чтобы определить, какой язык является «наилучшим» в каждой конкретной ситуации.

В последующих главах будут более подробно описаны некоторые широко используемые языки высокого уровня.

13

БЕЙСИК ¹⁾

Г. Хелмс

13.1. ВВЕДЕНИЕ

Бейсик — это сокращенное название от Beginner's All-purpose Symbolic Instruction Code (многоцелевой язык символических команд для начинающих). Бейсик был разработан в начале 60-х годов профессорами Университета в г. Дартмуте Джоном Кемени и Томасом Куртцем как язык для обучения программированию. Он оставался главным образом учебным языком до появления в середине 70-х годов микроЭВМ. Бейсик был выбран в качестве языка высокого уровня для таких систем благодаря его простоте и минимальным требованиям к памяти в сравнении с другими языками высокого уровня. Большой прогресс в области микроЭВМ, наблюдающийся с конца 70-х годов, способствовал тому, что в настоящее время Бейсик является, по всей вероятности, наиболее широко используемым в мире машинным языком.

Такое широкое признание Бейсик получил за счет унификации. ANSI ²⁾ был разработан стандартный Бейсик, однако в большинстве микроЭВМ используются свои собственные реализации этого языка. Это объясняется главным образом тем, что аппаратная часть микроЭВМ чрезвычайно быстро совершенствуется, а с помощью языка стремятся максимально использовать возможности аппаратных средств. В результате некоторые люди могут программировать на Бейсике на одной вычислительной системе (например, на Apple) и не могут использовать его на другой системе (например, на IBM).

Широкое разнообразие различных реализаций Бейсика делает невозможным рассмотрение всех существующих его вариантов. Поэтому мы ограничимся изучением следующих широко известных реализаций: Apple II Applesoft, Atari 400/800, Commodore PET, IBM Personal Computer Advanced, Radio Shack Level II, Radio Shack Color Computer Extended, Texas Instru-

¹⁾ Adapted from The BASIC Book, by Harry L. Helms. Copyright © 1973. Used by permission of McGraw-Hill, Inc. All rights reserved.

²⁾ ANSI — Американский национальный институт стандартов. — *Прим. ред.*

ments 99/4(A)¹⁾. Большинство других реализаций Бейсика подобны перечисленным выше. Для изучения не вошедших в этот список реализаций языка читателю рекомендуется обратиться к руководству по программированию для той системы, которую он собирается использовать.

13.2. СИСТЕМНЫЕ КОМАНДЫ

Команды — это директивы вычислительной системе. Они не зависят от программы, находящейся в памяти системы, и выполняются немедленно. Ниже приводится список команд и указывается, в каких реализациях они имеются. Если после команды в скобках не указана ни одна из систем, то эта команда имеется во всех семи реализациях, перечисленных в предыдущем разделе.

AUDIO Соединяет или разъединяет кассетное устройство вывода с громкоговорителем телевизора (Radio Shack Extended Color).

AUTO Автоматически нумерует строки программы при их вводе с клавиатуры (Atari, IBM Advanced, Radio Shack Level II).

BLOAD Загружает в память двоичные данные или программы на машинном языке (IBM Advanced).

BREAK Устанавливает точку прерывания для останова программы на строке с заданным номером (Texas Instruments 99/4).

BSAVE Сохраняет двоичные данные на дискете (IBM Advanced).

BYE Переход в режим калькулятора из Бейсика (Atari, Texas Instruments 99/4).

CALL-151 Переводит систему в режим монитора для выполнения программы на машинном языке (Apple II).

CALL CLEAR Очищает экран видеомонитора (Texas Instruments 99/4).

CLEAR Обнуляет все числовые переменные и делает все строковые переменные пустыми (Apple II и Atari).

Резервирует заданное число байтов памяти для хранения строки; кроме того, обнуляет числовые переменные и делает все строковые переменные пустыми (Radio Shack Level II и Extended Color).

Очищает все переменные программы и дополнительно устанавливает размер области памяти (IBM Advanced).

¹⁾ В настоящее время отечественная промышленность выпускает микроЭВМ, аналогичные по архитектуре Apple II (Agat-2), IBM PC (микроЭВМ серии ЕС). — *Прим. ред.*

CLOAD Загружает Бейсик-программу с кассеты (Atari, Radio Shack Level II, Extended Color).

CLOADM Загружает программу на машинном языке с кассеты (Radio Shack Extended Color).

CLOAD? Сравнивает программу, находящуюся в памяти, с программой на кассете. Если они отличаются, то на видеотерминал выдается сообщение BAD (Radio Shack Level II).

CLR Такая же функция, как **CLEAR** (Apple II, Commodore PET).

CONT Возобновляет выполнение программы после ее останова (отсутствует в Texas Instruments 99/4).

CONTINUE Такая же функция, как **CONT** (Texas Instruments 99/4).

CSAVE Пишет программу из памяти на кассету (Atari, Radio Shack Level II, Extended Color).

CSAVEM Выводит файл с машинным кодом (Radio Shack Extended Color).

DEL Исключает указанные строки из программы. Форма:

DEL номера строк(и)

(Apple II, Radio Shack Extended Color).

DELETE Такая же функция, как **DEL** (IBM Advanced, Radio Shack Level II).

Исключает файлы программ или данных из файловой системы (Texas Instruments 99/4).

DLOADM Загружают программы на машинном языке с указанной скоростью: 0 для 300 бод, 1 для 1200 бод (Radio Shack Extended Color).

EDIT Допускает редактирование строк с указанными номерами (IBM Advanced, Radio Shack Level II, Extended Color).

FILES Для файлов с указанным именем печатает соответствующее содержимое справочника дискеты (IBM Advanced).

HIMEN Определяет максимальный адрес, доступный при выполнении программы (Apple II).

HOME Передвигает курсор в левый верхний угол дисплея (Apple II).

KILL Стирает файл на дискете (IBM Advanced).

LIST Выдает на экран указанные строки программы. Если строки не указаны, то выдается вся программа. Форма:

LIST номер первой — номер последней
 выводимой строки выводимой строки

LOAD Такая же функция, как **CLOAD** (Apple II, Commodore PET, IBM Advanced).

LOMEN Определяет минимальный адрес, доступный в программе (Apple II).

MERGE Объединяет программу, записанную на внешнем устройстве, с программой, находящейся в памяти (IBM Advanced).
MOTOR Включает или выключает кассетный магнитофон (Radio Shack Extended Color).

NAME ... AS Переименовывает файл на дискете. Форма:

NAME старое имя **AS** новое имя

(IBM Advanced).

NEW Удаляет всю программу из памяти и очищает все переменные.

NOTRACE Выключает режим **TRACE** (Apple II).

NUM Такая же функция, как **AUTO**, но нумерация строк производится, начиная с 100, и с шагом 10 (Texas Instruments 99/4).

OLD Такая же функция, как **CLOAD** (Texas Instruments 99/4).

RENUM Перенумеровывает строки программы с заданным шагом. Форма:

RENUM новый, старт, шаг

где новый — номер первой строки, старт — номер строки в исходной программе, начиная с которой надо производить перенумерацию, и шаг — шаг, с которым производится перенумерация. Если шаг не указан, то он полагается равным 10 (IBM Advanced, Radio Shack Extended Color).

RESEQUENCE Перенумеровывает строки программы с заданным шагом, начиная с указанного номера строки. Форма:

RESEQUENCE начальная строка, шаг

(Texas Instruments 99/4).

RESET Повторная инициализация всей информации на дискете (IBM Advanced).

RUN Иницирует выполнение программы. Если указан номер строки, то программа выполняется, начиная с этой строки.

SAVE Такая же функция, как **CSAVE** (Apple II, Commodore PET, IBM Advanced, Texas Instruments 99/4).

SKIPF Переход к следующей программе на кассете или к концу указанной программы (Radio Shack Extended Color).

SYS Такая же функция, как **CALL-151** (Commodore PET).

SYSTEM Такая же функция, как **CALL-151** (IBM Advanced, Radio Shack Level II).

TRACE Указывает номер выполняемой строки программы (Apple II, Texas Instruments 99/4).

TROFF Такая же функция, как **NOTRACE** (IBM Advanced, Radio Shack Level II и Extended Color).

TRON Такая же функция, как **TRACE** (IBM Advanced, Radio Shack Level II и Extended Color).

UNBREAK Ликвидирует точку прерывания, установленную командой **BREAK** (Texas Instruments 99/4)

UNTRACE Такая же функция, как **NOTRACE** (Texas Instruments 99/4).

VERIFY Такая же функция, как **CLOAD?** (Commodore PET).

13.3. СТРУКТУРА ПРОГРАММЫ

Все строки в Бейсик-программе должны быть пронумерованы. Программа начинает выполняться со строки с минимальным номером. Далее строки обрабатываются в порядке возрастания их номеров. Обычно при программировании на Бейсике строки нумеруют с 0 до 999 с шагом 10. Использование такого шага позволяет впоследствии при необходимости вставлять дополнительные операторы. Принято также использовать номера с 0 по 999 для строк основной программы, а номера свыше 1000 — для подпрограмм.

Одна строка может содержать несколько операторов; при этом операторы отделяются друг от друга двоеточием (:). (Такая возможность отсутствует в реализации Texas Instruments 99/4 (A).)

В практике программирования принято заканчивать программу оператором **END**, хотя это условие и не является обязательным. Однако если в программе используются подпрограммы, то в основной программе оператор **END** должен присутствовать обязательно. Это необходимо для того, чтобы не допустить выполнения всех процедур после завершения работы основной программы.

С помощью операторов **REM** в программу могут быть включены комментарии. Оператор **REM** не вызывает никаких действий при работе программы, хотя и занимает место в памяти. Он имеет следующую форму:

100 REM КОММЕНТАРИЙ

Операторы **REM** должны включаться в программу для пояснения ее содержания, если листинг программы будет читаться другими людьми. Они также могут оказаться полезными при составлении и отладке программы.

В Бейсике не предусмотрены средства определения разделов или секций основной программы, выполняющих определенные функции, как, например, ввод-вывод, арифметические операции и т. д. Однако в практике программирования принято группировать операторы, выполняющие определенную функцию.

13.4. ПЕРЕМЕННЫЕ И КОНСТАНТЫ

Имена всех переменных и констант должны начинаться с буквы алфавита (A—Z). Остальные знаки, составляющие имя,

могут быть либо буквами, либо цифрами. Никаких других знаков в именах (!, % и т. д.) использовать не разрешается.

В большинстве реализаций Бейсика разрешается использовать имена переменных, содержащие до 255 букв или цифр. Однако только первые две из них являются значащими, т. е. используются вычислительной машиной для того, чтобы различать имена переменных. Например, имена **DOLLARS** и **DOWNTIME** соответствуют одной и той же переменной. Исключения составляют реализации Texas Instruments 99/4(A) и IBM Advanced, в которых число значащих знаков в именах переменных составляет 15 и 40 соответственно.

Переменные могут принимать целые или вещественные значения. Тип переменной может быть определен с помощью соответствующего знака (см. табл. 13.1), который указывается после имени переменной.

Таблица 13.1

Знак	Тип	Определение
\$	Строка	Переменная, содержащая до 255 знаков
%	Целое	Целочисленная переменная, принимающая значения от -32767 до 32767
! или E	Обычная точность	Переменная, принимающая значение с шестью значащими цифрами
#	Удвоенная точность	Переменная, принимающая значение с шестнадцатью значащими цифрами
D	Удвоенная точность, используемая при решении научных задач	Используется для констант или выходных данных, имеющих очень большое (или очень маленькое) значение

Предполагается, что переменные без знаков описания имеют обычную точность.

Значения переменным могут быть присвоены с помощью оператора **LET**:

$$\text{LET } X = 10$$

Однако переменным можно присвоить значения, не пользуясь оператором **LET**:

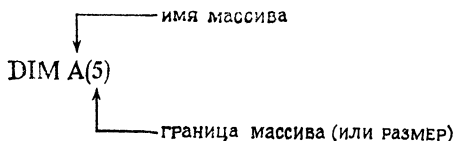
$$X = 10$$

Значения переменным могут быть присвоены в результате выполнения операций:

$$X = A/B$$

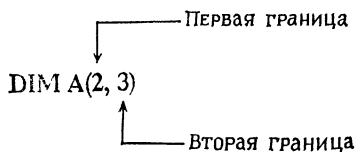
Массив — это упорядоченный набор данных, хранимый с использованием имени одной-единственной переменной. Отдельные части массива называются **элементами**. Элементы могут быть числами или строками. Каждый элемент определяется именем массива и следующим за ним целым числом (называемым индексом). Для имен массивов существуют те же правила, что и для имен переменных.

Число элементов в массиве задается оператором DIM. Оператор



определяет **одномерный** массив, содержащий элементы **A(0)**, **A(1)**, **A(2)**, **A(3)**, **A(4)** и **A(5)**. Имя переменной с индексом 0, хотя и имеется в распоряжении программиста, обычно не используется.

Число измерений массива может быть больше 1. Оператор



определяет **двумерный** массив с элементами **A(0, 0)**, **A(1, 1)**, **A(1, 2)**, **A(2, 1)** и т. д.

Границы массива задаются числами или выражениями. Операторы **DIM** могут быть размещены в любых местах Бейсик-программы.

В реализациях IBM Advanced и Texas Instruments 99/4(A) имеется оператор **OPTION BASE**. Он позволяет задавать нижнюю границу индекса массива. Например, операторы

```
100 OPTION BASE = 5
200 DIM X(10)
```

определяют массив, содержащий элементы от **X(5)** до **X(10)**.

В Бейсике IBM Advanced имеются также операторы **ERASE** и **SWAP**. Оператор **ERASE** позволяет исключать специфицированные переменные из программы. Он имеет следующую форму:

ERASE список переменных

Оператор **SWAP** позволяет менять местами значения двух переменных. Он имеет следующую форму:

SWAP первая переменная, вторая переменная

В реализациях IBM Advanced и Radio Shack Level II имеются операторы, позволяющие описывать список переменных как переменные определенного типа, не добавляя знака описания типа к каждому имени переменной:

DEFDBL Переменные, имена которых начинаются с любой буквы из указанного списка, хранятся и обрабатываются как переменные с удвоенной точностью. Форма:

DEFDBL буквы

DEFINT Переменные, имена которых начинаются с любой буквы из указанного списка, хранятся и обрабатываются как целочисленные переменные. Форма такая же, как у оператора **DEFDBL**.

DEFSNG Переменные, имена которых начинаются с любой буквы из указанного списка, хранятся и обрабатываются как переменные с одинарной точностью. Форма такая же, как у оператора **DEFDBL**.

DEFSTR Переменные, имена которых начинаются с любой буквы из указанного списка, хранятся и обрабатываются как строковые переменные. Форма такая же, как у оператора **DEFDBL**.

13.5. АРИФМЕТИЧЕСКИЕ ОПЕРАТОРЫ

Арифметические операторы в Бейсике обозначаются следующими символами:

+	Сложение
-	Вычитание
*	Умножение
/	Деление
\	Целочисленное деление (IBM Advanced)
^ или ↑	Возведение в степень
MOD	Получение целого остатка от целочисленного деления (Apple, IBM Advanced)

13.6. ОПЕРАТОРЫ ОТНОШЕНИЯ

Бейсик содержит следующие знаки для обозначения операций отношения:

<	Меньше
>	Больше
=	Равно
<>	Не равно
<=	Меньше или равно
>=	Больше или равно

13.7. ЛОГИЧЕСКИЕ ОПЕРАТОРЫ

Бейсик содержит следующие логические операторы:

AND	Выражение истинно, если обе части истинны; в противном случае выражение ложно.
OR	Выражение истинно, если одна из частей истинна; в противном случае выражение ложно.
NOT	Выражение истинно, если значение аргумента ложно; в противном случае выражение ложно.
XOR	Выражение ложно, если обе части ложны или обе части истинны; выражение истинно, если одна часть истинна, а другая ложна (IBM Advanced).
IMP	Выражение ложно, если первая часть истинна, а вторая ложна; в противном случае выражение истинно.
EQV	Выражение истинно, если обе части истинны или обе части ложны; в противном случае выражение ложно (IBM Advanced).

Для вычисления значения, противоположного по отношению к значению выражения, перед последним ставится знак «—».

13.8. ПОРЯДОК ВЫПОЛНЕНИЯ ОПЕРАЦИЙ

Арифметические и логические операции и операции отношения выполняются в следующем порядке:

1. Возведение в степень
2. Отрицание
3. Умножение и деление слева направо
4. Сложение и вычитание слева направо
5. Операторы отношения слева направо
6. **NOT**
7. **AND**
8. **OR**
9. **XOR**
10. **IMP**
11. **EQV**

Порядок выполнения операций может изменяться в зависимости от расположения выражений и наличия скобок. Если имеется несколько уровней вложенных скобок, то вначале выполняются операции в самых внутренних скобках, затем операции в скобках следующего уровня и т. д.

13.9. УПРАВЛЯЮЩИЕ ОПЕРАТОРЫ

Бейсик-программа обычно начинает выполняться со строки с наименьшим номером, а дальнейшее выполнение осуществляется в порядке возрастания номеров строк. Такой порядок выполнения программы можно изменить, используя управляющие операторы и операторы передачи. Операторы передачи можно

разбить на **безусловные**, т. е. те, которые всегда изменяют ход выполнения программы, и **условные**, т. е. изменяющие порядок выполнения операторов в программе только при определенных условиях.

Бейсик содержит следующие управляющие операторы.

END Завершает выполнение программы.

RETURN Завершает выполнение подпрограммы и возвращает управление оператору, непосредственно следующему за последним выполненным оператором **GOSUB**.

STOP Прерывает выполнение программы.

WAIT Приостанавливает выполнение программы. Программа не начнет выполняться до тех пор, пока не будут выполнены условия, указанные после **WAIT** (Apple II, Commodore Pet и IBM Advanced).

Имеются два безусловных оператора передачи:

GOSUB Передает управление подпрограмме, начинающейся со строки с номером, равным значению выражения, записанного после **GOSUB**.

GOTO Передает управление строке с номером, равным значению выражения, записанного после **GOTO**.

Условных операторов передачи значительно больше:

ELSE Используется вместе с оператором **IF** для задания альтернативного действия, если выражение, записанное после **IF**, ложно.



(Имеется в IBM Advanced, Radio Shack Level II и Extended Color).

ERROR Используется вместе с **IF ... THEN** для печати сообщения об ошибке при выполнении указанных условий (Radio Shack Level II).

Печатает сообщение об ошибке и позволяет определить код ошибки (IBM Advanced).

FOR ... TO Задаёт последовательность операторов, выполнение которых должно повторяться указанное число раз.

Цикл **FOR ... TO** заканчивается словом **NEXT**:

```

10 FOR I = 1 TO 10
20 PRINT I;
30 NEXT
  
```

Переменная **I** называется **индексной**. После каждого выполнения цикла значение индексной переменной увеличивается на 1.

Когда значение индексной переменной становится больше верхней границы диапазона ее изменения (в данном случае 10), цикл заканчивается и программа продолжает выполняться обычным образом. Для задания шага, с которым должно увеличиваться значение **I**, может быть использовано слово **STEP**. Если в программе содержится строка

10 FOR I=1 TO 50 STEP 5

то значение **I** будет возрастать от 1 до 50 с шагом 5, и цикл завершится, когда значение **I** превысит 50. Если слово **STEP** опущено, то величина шага полагается равной 1. Величина шага, а также начальное и конечное значения переменной **I** могут быть отрицательными.

IF ... GOSUB Вычисляет следующее за **IF** логическое выражение. Если оно истинно, выполняется подпрограмма, начинающаяся со строки, номер которой указан после **GOSUB**. Если выражение ложно, выполняется следующая строка программы. (Отсутствует в Atari, IBM Advanced и Texas Instruments 99/4.)

IF ... THEN Вычисляет следующее за **IF** логическое выражение. Если оно истинно, выполняется оператор, стоящий после **THEN**. Если выражение ложно, выполняется следующая строка программы. Действие, которое выполнится в случае, когда выражение ложно, может быть также указано с помощью **ELSE**:

IF A = B THEN PRINT "A = B" ELSE STOP

(В Бейсике Texas Instruments 99/4 после слов **THEN** и **ELSE** можно указывать только номер строки.)

ON COM(n) GOSUB Передает управление подпрограмме, начинающейся со строки, номер которой указан после **GOSUB**, в случае, если в буфер связи поступает информация через адаптер связи (1 или 2), указанный значением **n** (имеется в IBM Advanced).

ON ERROR ... GOTO Передает управление строке с номером, указанным после **GOTO**, в случае, если при выполнении программы будет обнаружена ошибка. Для этого оператор

ON ERROR ... GOTO должен быть выполнен до обнаружения ошибки. (Имеется в IBM Advanced и Radio Shack Level II.)

ONERR ... GOTO Такая же функция, как **ON ERROR ... GOTO** (Apple II).

ON ... GOTO Передает управление одной из строк, номера которых указаны после **GOTO**, в зависимости от значения выражения, стоящего после **ON**:

100 ON I GOTO 300, 400, 500

$$\begin{array}{ccc} \uparrow & \uparrow & \uparrow \\ \mathbf{I} = & 1 & 2 & 3 \end{array}$$

I — это выражение, значение которого сравнивается с целыми числами. Если значение **I** превосходит число элементов, сле-

дующих за **GOTO**, то выполняется следующая строка программы.

ON ... GOSUB Аналогично **ON ... GOTO**, но управление передается не строкам программы, а подпрограммам.

ON KEY(n) GOSUB Осуществляет переход к программе, связанной с ключом, выбираемым с помощью **n**, где **n** — выражение, принимающее значения от 1 до 14 (IBM Advanced).

ON PEN GOSUB Передает управление подпрограмме, начинающейся со строки с номером, указанным после **GOSUB**, в случае, если активируется световое перо (IBM Advanced).

ON STRIG(n) GOSUB Осуществляет переход к специальной программе в случае, когда нажат один из джойстиков. Если **n = 0**, управляет первый джойстик, если **n = 2**, управляет второй джойстик (IBM Advanced).

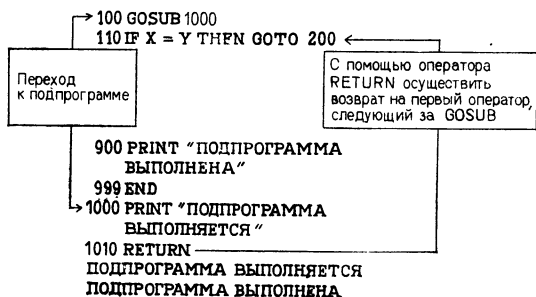
WHILE ... WEND Задаёт цикл, который выполняется до тех пор, пока истинно указанное выражение. Обычно этот оператор имеет следующую форму:

```
WHILE выражение
Последовательность операторов
WEND
```

Выражение истинно до тех пор, пока оно не равно нулю. После каждого выполнения последовательности операторов проверяется выражение, стоящее после **WHILE**. Если это выражение не истинно, то выполнение программы продолжается с первого оператора, следующего за **WEND** (IBM Advanced).

13.10. ПОДПРОГРАММЫ

Подпрограмма — это последовательность операторов, выполняющих определенное действие, и оформленная определенным образом. Подпрограмма может использоваться в программе столько раз, сколько это потребуется. Передача управления подпрограмме осуществляется с помощью оператора **GOSUB** или его модификаций. При возврате из подпрограммы (с помощью оператора **RETURN**) в основную программу, осуществляется переход на первый оператор, следующий за словом **GOSUB**.



Подпрограммы размещаются за основной программой. В практике хорошего программирования принято нумеровать строки основной программы от 0 до 999, а строки подпрограмм — от 1000 до 9999.

При использовании подпрограмм последним оператором основной программы должен быть оператор **END**. Он необходим для того, чтобы не допустить перехода к подпрограммам сразу же после окончания основной программы.

13.11. ЧИСЛОВЫЕ ФУНКЦИИ

Различные реализации Бейсика содержат несколько числовых функций. Общая форма числовых функций следующая:

числовая функция (число или выражение)

ABS Возвращает абсолютное значение выражения.

ATN Возвращает арктангенс выражения.

CDBL Возвращает число или значение выражения в форме с удвоенной точностью (IBM Advanced, Radio Shack Level II).

CINT Возвращает максимальное целое, не превосходящее заданное число или значение выражения (IBM Advanced, Radio Shack Level II).

CLOG Возвращает десятичный логарифм выражения (Apple II, Atari).

COS Возвращает косинус выражения.

CSNG Возвращает число или значение в форме с обычной точностью (IBM Advanced, Radio Shack Level II).

DEF Определяет новую числовую функцию (Texas Instruments 99/4).

DEF FN Такая же функция, как **DEF** (Apple II, Commodore PET, IBM Advanced, Radio Shack Extended Color).

ERL Возвращает номер строки, в которой обнаружена ошибка (IBM Advanced, Radio Shack Level II).

ERR Возвращает код ошибки (IBM Advanced, Radio Shack Level II).

EXP Возвращает результат возведения числа e в степень, величина которой задается выражением.

FIX Возвращает аргумент в усеченной форме (IBM Advanced, Radio Shack Level II).

FRE Выдает число неиспользованных байтов в памяти. Если за этим оператором следует строковая переменная, то выдает величину объема неиспользованной памяти для строк (Atari, Commodore PET, IBM Advanced, Radio Shack Level II).

HEX\$ Возвращает шестнадцатеричное значение числа (IBM Advanced, Radio Shack Extended Color).

INT Возвращает целую часть значения выражения.

LOG Возвращает натуральный логарифм аргумента.

MEM Возвращает величину объема свободной памяти (Radio Shack Level II и Extended Color).

MKDS Преобразует число с обычной точностью в 8-байтовую строку (IBM Advanced).

MKI\$ Преобразует целое число в 2-байтовую строку (IBM Advanced).

MKSS Преобразует число с обычной точностью в 4-байтовую строку (IBM Advanced).

NULL Выводит заданное число пробелов при печати (Atari).

OCT\$ Возвращает восьмеричное значение числа (IBM Advanced).

POS Возвращает число от 0 до 63, указывающее позицию курсора на видеотерминале (Apple II, Commodore PET, IBM Advanced, Radio Shack Level II и Extended Color).

PPOINT Возвращает код цвета указанной графической точки (Radio Shack Extended Color).

RANDOM Обращается к генератору случайных чисел (Commodore PET, Radio Shack Level II).

RANDOMIZE Такая же функция, как **RANDOM** (IBM Advanced, Texas Instruments 99/4).

RND Генерирует псевдослучайное число (отсутствует в Radio Shack Extended Color).

SGN Возвращает -1 , если выражение отрицательно; возвращает 0, если выражение равно 0; возвращает 1, если выражение положительно.

SIN Возвращает синус выражения, вычисленного в радианах.

SPC Возвращает указанное число пробелов (Commodore PET и IBM Advanced).

SQR Возвращает квадратный корень выражения (отсутствует в Atari).

TAN Возвращает тангенс выражения (отсутствует в Atari).

TI Засылает заданное значение в часы реального времени (Commodore PET).

TIMER Возвращает содержимое таймера или позволяет осуществить его установку (Radio Shack Extended Color).

TIMES Устанавливает или выдает текущее время (IBM Advanced).

13.12. СТРОКОВЫЕ ФУНКЦИИ

Бейсик содержит также несколько строковых функций. Общая форма строковых функций следующая:

строковая функция (строковая переменная или аргумент)

ADR Возвращает адрес, по которому в памяти размещается имя, значение или указатель переменной (Atari).

ASC Возвращает значение первого знака строки в коде ASCII.

CALL KEY Опрашивает клавиатуру и возвращает литеру, вызывающую нажатую клавишу, или пустую строку, если ни одна из клавиш не нажата (Texas Instruments 99/4).

CHR\$ Возвращает строку, состоящую из одного знака, который имеет графический или управляющий код *ASCII*, заданный числом или выражением, принимающим значение от 0 до 255.

CVD Преобразует 8-байтовую строку в число с удвоенной точностью (IBM Advanced).

CVI Преобразует 2-байтовую строку в целое число (IBM Advanced).

CVS Преобразует 2-байтовую строку в число с одинарной точностью (IBM Advanced).

FRE Возвращает величину объема свободной памяти для хранения строковых переменных (Atari, Commodore PET, IBM Advanced, Radio Shack Level II).

GET Такая же функция, как **CALL KEY** (Apple II, Commodore PET).

Считывает запись из произвольного файла в произвольный буфер (IBM Advanced).

INKEY\$ Такая же функция, как **CALL KEY** (IBM Advanced, Radio Shack Level II и Extended Color).

INSTR Ищет заданную строку, начиная с указанной позиции другой строки и возвращает номер позиции, в которой найдена искомая строка (IBM Advanced, Radio Shack Extended Color).

LEFT\$ Возвращает указанное число знаков (n) строки, начиная слева. Форма:

LEFT\$ (строка, n)

(Отсутствует в Atari и Texas Instruments 99/4.)

LEN Возвращает длину указанной строки или число 0, если строка пустая.

MID\$ Возвращает указанное число знаков (n) строки, начиная с позиции p. Форма:

MID \$ (строка,n,p)

(Отсутствует в Atari и Texas Instruments 99/4.)

POS Возвращает подстроку, которая начинается с позиции p заданной строки. Форма:

POS (строка, подстрока, n)

(Имеется в IBM Advanced, Radio Shack Level II, Radio Shack Extended Color, Texas Instruments 99/4.)

RIGHT\$ Такая же функция, как **LEFT\$**, но возвращает заданное число знаков строки, начиная справа (отсутствует в Atari и Texas Instruments 99/4).

SEG\$ Возвращает указанное число знаков (n) строки, начиная с позиции p (нумерация знаков в строке производится слева направо). Форма:

SEG\$ (строка, p, n)

(Texas Instruments 99/4).

STR\$ Преобразует числовое выражение в строку.

STRING\$ Возвращает строку длиной n, составленную из знака с. Форма:

STRING\$ (n, c)

(Имеется в IBM Advanced, Radio Shack Level II, Radio Shack Extended Color.)

VAL Преобразует строку в число.

VARPTR Такая же функция, как **ADR** (IBM Advanced, Radio Shack Level II, Radio Shack Extended Color).

13.13. ОПЕРАТОРЫ И ПРОГРАММЫ ЯЗЫКА АССЕМБЛЕРА

Благодаря широкому использованию Бейсика в микропроцессорных вычислительных системах в большинство его реализаций были включены операторы и программы, которые позволяют осуществлять прямой доступ к микропроцессору и связанной с ним полупроводниковой памяти.

Для правильного использования операторов и программ языка ассемблера необходимо иметь распределение памяти системы и знать набор команд для используемого в системе микропроцессора. Применение программ языка ассемблера способствует более быстрому выполнению задания и более эффективному использованию имеющейся памяти. Однако программирование на языке ассемблера обычно трудоемко и требует больших затрат времени.

Прямой доступ к памяти вычислительной системы позволяют осуществлять следующие операторы.

PEEK Возвращает значение, хранимое по указанному адресу (в Atari можно указывать только адрес памяти экрана; отсутствует в Texas Instruments 99/4).

GO GCHAR Такая же функция, как **PEEK** (Texas Instruments 99/4).

POKE Помещает указанное значение по заданному адресу. Форма:

POKE адрес памяти, значение

(отсутствует в Texas Instruments 99/4).

Для обращения к подпрограммам, написанным в кодах операций микропроцессора системы, используются следующие операторы:

CALL Передает управление от основной программы подпрограмме на языке ассемблера, размещенной по указанному адресу. Форма:

CALL адрес памяти

Команды возврата на основную программу содержатся в подпрограмме на языке ассемблера (Apple II, IBM Advanced).

DEFUSR Определяет начальный адрес подпрограммы на машинном языке (IBM Advanced, Radio Shack Extended Color).

EXEC Передает управление программе на языке ассемблера, размещенной по указанному адресу (Radio Shack Extended Color).

POP Удаляет последний занесенный элемент из регистрового стека (Apple II, Atari).

USR Такая же функция, как **CALL** (отсутствует в Atari и Texas Instruments 99/4).

13.14. ГРАФИЧЕСКИЕ ОПЕРАТОРЫ

Наибольшие отличия между отдельными реализациями Бейсика проявляются в графических операторах. Больше всего вычислительные системы различаются в средствах графического отображения, и эти различия отражаются в реализациях Бейсика, используемых в этих системах. Графические операторы являются наиболее «непостоянной» частью Бейсика. По мере того как производители микроЭВМ постоянно создают более усовершенствованные средства графического отображения в своих системах, появляются новые графические операторы.

CALL CHAR Определяет новый знак для видеодисплея (Texas Instruments 99/4).

CALL CLEAR Очищает видеодисплей, но не влияет на программу в памяти (Texas Instruments 99/4).

CALL COLOR Определяет цветовой фон отдельных знаков (Texas Instruments 99/4).

CALL HCAR Проводит горизонтальную линию в строке с указанным номером (Texas Instruments 99/4).

CALL SCREEN Определяет цветовой фон для видеодисплея (Texas Instruments 99/4).

CALL VCHAR Проводит вертикальную линию в указанном столбце (Texas Instruments 99/4).

CIRCLE Проводит окружность на видеодисплее (IBM Advanced, Radio Shack Extended Color).

CLS Такая же функция, как **CALL CLEAR** (Apple II, IBM

Advanced, Radio Shack Level II, Radio Shack Extended Color).
COLOR Устанавливает цвет следующей точки (Apple II). Определяет фоновый цвет для отдельных знаков (Atari).

Устанавливает основной и фоновый цвета (Radio Shack Extended Color).

Устанавливает основной, фоновый и граничный цвета (IBM Advanced).

DRAW Проводит линию, имеющую указанную длину и цвет, которая начинается с заданной точки (Radio Shack Extended Color).

Чертит объект, определенный знаками строки, следующей за словом **DRAW** (IBM Advanced).

DRAWTO Проводит линию из последней нанесенной точки к новой указанной позиции (Atari).

GET Считывает графическое содержание прямоугольника в память (Radio Shack Extended Color).

В текстовом режиме считывает запись из произвольного файла в произвольный буфер; в графическом режиме считывает точки из области экрана (IBM Advanced).

GR Переключает на графику с низким разрешением (Apple II).

GRAPHICS Такая же функция, как **CALL HCAR** (Atari).

HCOLOR Выбирает цветовой фон экрана видеодисплея (Apple II).

HLIN ... AT Такая же функция, как **CALL HCAR** (Apple II).

HPLOT Такая же функция, как **DRAWTO** (Apple II).

LINE Проводит линию из одной заданной точки в другую (IBM Advanced, Radio Shack Extended Color).

PAINT «Раскрашивает» видеодисплей между двумя заданными точками. (IBM Advanced, Radio Shack Extended Color).

PCLEAR Резервирует указанный объем графической памяти (Radio Shack Extended Color).

PCLS Очищает дисплей, используя указанный фоновый цвет (Radio Shack Extended Color).

PCOPY Копирует графику из исходной страницы в назначенную страницу (Radio Shack Extended Color).

PLOT Переключается на указанный графический блок (Apple II, Atari).

PMODE Выбирает разрешающую способность графики и первую страницу памяти (Radio Shack Extended Color).

POINT Проверяет указанную точку экрана и возвращает значение 1, если она включена, и 0, если выключена (Radio Shack Level II).

Возвращает цвет указанной точки на экране (IBM Advanced).

PRESET Перекрашивает точки в заданный фоновый цвет (IBM Advanced, Radio Shack Extended Color).

PSET Раскрашивает точку в указанный цвет (IBM Advanced, Radio Shack Extended Color).

PUT Помещает графическое изображение из исходного в начальный/конечный прямоугольник (Radio Shack Extended Color). В текстовом режиме пишет запись из произвольного буфера в произвольный файл. В графическом режиме раскрашивает заданную область экрана (IBM Advanced).

RESET Восстанавливает графическую точку (Radio Shack Extended Color, Radio Shack Level II).

SCREEN Выбирает графический или текстовый режим и цвет экрана (Radio Shack Extended Color).

Возвращает код ASCII знака, расположенного в заданной строке и заданном столбце экрана (IBM Advanced).

SET Такая же функция, как **PLOT** (Radio Shack Level II, Radio Shack Extended Color).

SETCOLOR Такая же функция, как **CALL SCREEN** (Atari).

TEXT Переключает с графического на текстовый режим (Apple II).

VLIN ... AT Такая же функция, как **CALL VCHAR** (Apple II).

VTAB Передвигает курсор вниз на заданное число строк (Apple II).

13.15. ОПЕРАТОРЫ ВВОДА И ВЫВОДА

Большинство реализаций Бейсика содержат одинаковые группы операторов ввода и вывода, такие, как **PRINT**, **INPUT** и **READ**. Однако во многих реализациях имеются также некоторые модификации этих операторов.

Операторы вывода

PRINT Выводит строковые и числовые переменные, числа, а также текст, заключенный в кавычки:

```

100 X = 10
200 PRINT X
      10
100 A$ = "OUTPUT"
200 PRINT A$
      OUTPUT
100 PRINT "OUTPUT"
      OUTPUT

```

После оператора **PRINT** может быть указано несколько элементов данных. Если они отделяются друг от друга запятыми, то каждый элемент печатается на видеодисплее микроЭВМ в

отдельной зоне печати.

```
100 PRINT "OUTPUT", "OUTPUT"
      OUTPUT OUTPUT
```

Если элементы данных отделяются друг от друга точкой с запятой, то на дисплее между ними не оставляется пробелов:

```
100 PRINT "OUTPUT"; "OUTPUT"
      OUTPUTOUTPUT
```

Оператор **PRINT** может быть также использован для выполнения вычислений:

```
100 PRINT 5+2
      7
```

PRINT@ Указывает позицию, начиная с которой надо начать печатать. Обычная форма:

PRINT@ n, выход

где *n* — целое число, заключенное между 0 и 1023, а выход — данные, которые должны быть напечатаны (Radio Shack Level II, Radio Shack Extended Color).

POSITION Такая же функция, как **PRINT@** (Atari).

PRINT USING Печатает строковые и числовые значения в соответствии с заданным форматом. Форма:

PRINT USING спецификатор формата; значение

В качестве спецификаторов формата в операторе **PRINT USING** используются следующие символы:

#	Определяет позицию цифры Определяет десятичную точку в числе Указывает, что после каждой третьей цифры должна быть вставлена запятая
**	Указывает, что все неиспользованные позиции слева от десятичного числа будут заполнены звездочками
\$\$	Указывает, что в первой позиции, предшествующей числу, будет располагаться знак доллара
**\$	Указывает, что в первой позиции, предшествующей числу, будет располагаться знак доллара, а во всех оставшихся неиспользованных позициях — звездочки
Λ или ↑	Указывает, что значение должно быть напечатано в показательной форме
+	Определяет знак "+" для положительных чисел и "—" для отрицательных (помещается в начале спецификатора формата)
/n/	Указывает, что должны быть напечатаны <i>n</i> + 2 знаков, содержащихся в строке (IBM Advanced).
%n%	Определяет строковое поле, состоящее более чем из одного знака; длиной поля будет число позиций, равное <i>n</i> + 2 (Radio Shack Level II, Radio Shack Extended Color).
!	Указывает, что первый знак текущего значения строки будет возвращен

(Оператор **PRINT USING** имеется в IBM Advanced, Radio Shack Extended Color).

TAB Используется вместе с **PRINT** для указания колонки, с которой следует начать печать. Форма:

PRINT TAB(n)

где **n** — целое число или выражение, принимающее целочисленные значения (отсутствует в Atari).

PRINT# Записывает указанные данные в файл или на кассетную ленту (отсутствует в Atari).

DISPLAY Такая же функция, как **PRINT** (Texas Instruments 99/).

WRITE Аналогично **PRINT**, но запятые вставляются между данными в тех случаях, когда последние являются выходными (IBM Advanced).

WIDTH Устанавливает ширину выходной строки, выраженную в знаках (IBM Advanced).

Операторы ввода

INPUT Останавливает выполнение программы и переводит ее в состояние ожидания ввода данных с клавиатуры. В кавычках может быть добавлена «подсказка» (или наводящее сообщение), которая будет отображена на дисплее. Форма:

INPUT «подсказка»; переменные

INPUT# Вводит данные с кассеты и присваивает их переменным (отсутствует в Apple II и Atari).

RECALL Такая же функция, как **INPUT#** (Apple II).

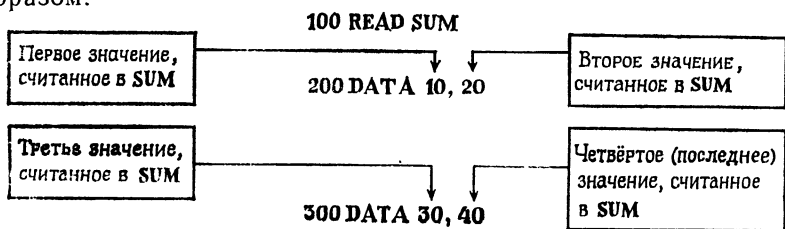
READ Считывает значения, указанные в операторе **DATA**, и присваивает их заданным переменным. Форма:

READ список переменных

DATA Задает список данных в программе, обращение к которым может быть осуществлено с помощью оператора **READ**. Форма:

DATA список данных

Операторы **READ** и **DATA** используются совместно следующим образом:



RESTORE При выполнении следующего оператора **READ** ввод данных будет осуществляться, начиная с первого элемента в первом операторе **DATA**.

13.16. СПЕЦИАЛЬНЫЕ ОПЕРАТОРЫ ВВОДА И ВЫВОДА

Быстрое усовершенствование аппаратных средств микрокомпьютерных систем привело к тому, что в них кроме видеотерминалов и обычной клавиатуры можно использовать дополнительно большое число периферийных устройств. Для этого в язык Бейсик, используемый на этих системах, были добавлены новые операторы.

Операторы вывода

BEEP Производит звук «бип», который поступает от динамика (IBM Advanced).

CALL SOUND Назначает звуковой выходной канал системы (Texas Instruments 99/4).

CLOSE Закрывает файл на периферийном устройстве (Commodore PET, IBM Advanced, Radio Shack Extended Color, Texas Instruments 99/4).

LLIST Печатает программу или указанную строку (IBM Advanced, Radio Shack Level II, Radio Shack Extended Color).

LPRINT Аналогично **PRINT**, но выходные данные передаются на периферийное печатающее устройство (Atari, IBM Advanced, Radio Shack Level II).

LPRINT USING Аналогично **PRINT USING**, но используется периферийное печатающее устройство (IBM Advanced).

OPEN Открывает периферийное устройство для ввода и вывода файла (Commodore PET, IBM Advanced, Texas Instruments 99/4).

DSP Отображает номер строки, в которой изменяются значения переменных (Apple II).

OPEN COM ... AS Открывает файл для обмена (IBM Advanced).

OUT Засылает указанное значение в заданный порт (IBM Advanced, Radio Shack Level II).

PLAY Воспроизводит музыку по заданным нотам, октаве, громкости и продолжительности (IBM Advanced, Radio Shack Extended Color).

PR# Аналогично **OUT** (Apple II).

SOUND Воспроизводит звучание определенного тона в течение указанного промежутка времени (Atari, IBM Advanced, Radio Shack Extended Color).

SPEED Выбирает скорость передачи знаков на устройство вывода (Apple II).

STORE Пишет содержимое числового массива на кассету (Apple II).

UPDATE Считывает и пишет открытый файл, находящийся на кассете (Texas Instruments 99/4).

Операторы ввода

APPEND Добавляет данные в конец файла (Texas Instruments 99/4).

CALL JOYSTK Проверяет и принимает входные данные из джойстика (Texas Instruments 99/4).

IN Запрашивает значение из входного порта (Radio Shack Level II).

IN# Аналогично **IN** (Apple II).

JOYSTK Возвращает горизонтальную или вертикальную координату джойстика (Radio Shack Extended Color).

LINE INPUT Задает значение строковой переменной, вводимое с клавиатуры (IBM Advanced, Radio Shack Extended Color).

STICK Такая же функция как **JOYSTK** (IBM Advanced).

13.17. ЗАРЕЗЕРВИРОВАННЫЕ СЛОВА

В каждой реализации Бейсика имеются свои зарезервированные слова. Общее правило заключается в том, что в качестве имен переменных нельзя выбирать слова, используемые в данной реализации для обозначения команд, функций и операторов.

14

КОБОЛ¹⁾

А. Таккер-мл.

14.1. ВВЕДЕНИЕ

В настоящее время Кобол (Common Business Oriented Language, т. е. общий язык, ориентированный на решение коммерческих задач) является наиболее широко используемым языком, предназначенным для обработки данных. Он был разработан и совершенствовался в качестве языка общего назначения, который позволяет легко переносить программы и технику программирования с одной машины на другую. Кроме того, Кобол похож на английский язык; его синтаксис был разработан так, чтобы человек, не имеющий специального образования, смог бы, так же как и программист, умело прочитать программу и понять ее. В этой главе мы изучим язык Кобол и увидим, насколько он удовлетворяет этим требованиям.

Краткая история создания Кобола

В конце 50-х годов возникла потребность в создании языка обработки данных общего назначения. В мае 1959 г. представители фирм, производивших ЭВМ, совместно с представителями промышленных предприятий и правительственных учреждений, использовавших ЭВМ, собрались в Вашингтоне, округ Колумбия, для обсуждения вопроса о возможности и целесообразности разработки такого языка.

Так был образован комитет CODASYL (Conference on Data System Language, т. е. Ассоциация по языкам систем обработки данных), который вскоре разработал проект такого языка. Исправленная версия этой разработки была опубликована в апреле 1960 г. правительственным издательством и получила название «Кобол-60».

Вторая версия, известная под названием «Кобол-61», была опубликована в следующем году и получила широкое распространение. В 1963 г. была опубликована расширенная версия

¹⁾ Adapted from Programming Languages, by Allen B. Tucker, Jr. Copyright © 1977. Used by permission of McGraw-Hill, All rights reserved.

Кобола, получившая название «Расширенный Кобол-61». Дальнейшая работа по расширению и усовершенствованию языка была проделана в 1965 г., в результате чего была создана новая версия, получившая название «Кобол, редакция 1965». В 1968 г. эта версия была окончательно утверждена в качестве американского национального стандарта. Тем временем работы по усовершенствованию Кобола продолжались. И в 1968, 1969, 1970 и 1973 гг. в COBOL Journal of Development были опубликованы его новые версии. Измененный американский национальный стандарт Кобола был утвержден в 1974 г. В нем отражены особенности всех модификаций Кобола, опубликованных с момента принятия стандартного Кобола в 1968 г.

Реализации и модификации Кобола

Версия Кобола, которая будет описана в этой главе, является американским национальным стандартом Кобола, принятым в 1974 г. Поскольку характерные особенности, которые отличают ее от стандарта 1968 г., не унифицированы в данном описании, мы будем особо отмечать их каждый раз, когда они встретятся в этой главе.

Кобол реализован на большинстве средних и больших ЭВМ (табл. 14.1).

Таблица 14.1

Изготовитель	Реализация
Burroughs	B1700, B6700
CDC	3000, 6000, Cyber Series
DEC	PDP-10
Honeywell	Multics, 600, 6000 Series
IBM	360, 370 Series
Univac	1100 Series

Некоторые изготовители предоставляют несколько реализаций Кобола, которые немного отличаются друг от друга. Однако все перечисленные в табл. 14.1 реализации Кобола соответствуют американскому национальному стандарту Кобола 1968 г. Кроме того, большинство из них является расширением языка (в зависимости от реализации).

Помимо реализаций Кобола, предоставляемых изготовителями ЭВМ, различными фирмами, специализирующимися на производстве программного обеспечения, и университетскими группами, были разработаны другие его варианты. Наиболее

известным из них, по-видимому, является Ватбол — быстро-компилирующая реализация Кобола, разработанная в Университете г. Ватерлоо.

Основные применения Кобола

Кобол был разработан для решения задач обработки данных. Хотя Кобол может быть использован и для решения научных задач, а также для обработки текстов, результат при этом обычно оказывается не столь удовлетворительным, как при использовании языка, разработанного специально для этих целей.

Последние версии Кобола предназначены для обработки данных, так как эти версии содержат такие мощные встроенные функциональные элементы, как генератор отчетов, функции поиска в таблицах и средства сортировки. Эти элементы были разработаны с целью помочь программисту организовывать данные в файле, осуществлять к ним доступ, корректировать, переупорядочивать и выдавать их.

Однако написание Кобол-программ является несколько более сложной задачей, чем написание программ на других языках. Это следует из того, что Кобол-программы состоят из английских фраз (вместо более сжатых формул), предложений, параграфов и секций как основных синтаксических блоков. Кроме того, программист при написании Кобол-программы должен размещать операторы и метки операторов в специально отведенных для них позициях.

Учитывая эти трудности, последние версии Кобола были разработаны таким образом, что программист может использовать некоторые сокращения и имеет возможность выбора форматов. Кроме того, средства автоматического ввода текста исходной программы из библиотеки также облегчают труд программиста.

14.2. НАПИСАНИЕ КОБОЛ-ПРОГРАММ

Как отмечалось в предыдущем разделе, Кобол-программа представляет собой совокупность строк, содержащих английский текст. Вышим уровнем структуры программы является **раздел**, и каждая Кобол-программа состоит из четырех разделов. В табл. 14.2 приводятся имена этих разделов и поясняется их назначение.

Эти четыре раздела должны быть включены в Кобол-программу в указанном порядке. Местоположение каждого из них определяется именем соответствующего раздела, помещенным в его начало. Имя раздела должно быть записано в точности так, как указано в табл. 14.2, и заканчиваться точкой (.).

Таблица 14.2

Имя раздела	Назначение
IDENTIFICATION DIVISION (Раздел идентификаций)	Определяет имя программы, имя ее автора, назначение и другие основные операционные характеристики
ENVIRONMENT DIVISION (Раздел оборудования)	Определяет некоторые характеристики оборудования ЭВМ, на которой будет выполняться программа, в особенности характеристики устройств ввода-вывода, используемых программой
DATA DIVISION (Раздел данных)	Описывает тип и структуру данных, используемых в программе. Это описание включает в себя главным образом указание разбивки записей внешних файлов, а также определение типов внутренних переменных и таблиц, которое является локальным в программе
PROCEDURE DIVISION (Раздел процедур)	Описывает алгоритм, который должен быть реализован при выполнении программы

Следующим уровнем структуры Кобол-программы является **секция**. Каждый раздел, кроме раздела идентификаций, может содержать более одной секции. Каждая секция может содержать более одного **параграфа**. Параграфы в свою очередь состоят из **предложений**, а предложения — из **слов**. Как мы увидим, каждый из этих терминов имеет вполне определенный смысл в Коболе.

Предлагаемое здесь описание Кобол-программы не отражает всех особенностей этого языка. Рассматриваются только те его элементы, которые, на наш взгляд, являются наиболее полезными при программировании. Однако мы не стремились к тому, чтобы дать упрощенное описание языка. Мы выделили его наиболее важные особенности для того, чтобы сосредоточить внимание на основных элементах Кобол-программы.

Типы данных и константы

Элементарные данные в Коболе могут быть представлены в виде **литералов**. Существуют два класса литералов: **числовые литералы** и **нечисловые литералы**. Числовые литералы записываются как обычные десятичные числа. Они содержат от 1 до 18 десятичных цифр, которым может предшествовать знак (+ или —), и могут содержать десятичную точку (.).

Ниже даются примеры числовых литералов в Коболе:

1.73 0 -17 250

Нечисловой литерал является строкой, заключенной в кавычки (") и содержащей один или более знаков из набора, определяемого конкретной реализацией языка. Этот набор должен содержать по крайней мере следующий 51 знак, образующий набор знаков Кобола:

_(пробел) . < (+ \$ *) ; - / , > = " A B ... Z 01 ... 9

Кроме того, если литерал сам содержит кавычки, то они должны быть записаны два раза подряд для того, чтобы их можно было отличить от внешних кавычек. Значение нечислового литерала в точности совпадает с последовательностью знаков, расположенных между внешними кавычками, за исключением того, что внутренние кавычки (") записываются два раза подряд (" "). В следующем примере приводятся нечисловые литералы Кобола:

"ABC"
"A_B_C"
"WHAT" "S_THIS?"

Читатель должен был заметить, что знаки пробела существенны в литералах, так что первые два литерала не эквивалентны.

Таблица 14.3

Фигуральная константа	Представляемый ею класс литералов
ZERO ZEROS ZEROES	Числовой литерал 0 или нечисловой литерал, целиком состоящий из нулей, как, например, "0", "00" и "000"
SPACE SPACES	Нечисловой литерал, целиком состоящий из пробелов (), как, например, " ", " " и " "
HIGH-VALUE HIGH-VALUES	Нечисловой литерал, целиком состоящий из максимального значения в сортирующей последовательности для набора знаков конкретной реализации, как, например, "9", "99" и "999"
LOW-VALUE LOW-VALUES	Нечисловой литерал, целиком состоящий из минимального значения в сортирующей последовательности для набора знаков конкретной реализации, как, например, " ", " " и " "
ALL литерал	Нечисловой литерал, состоящий из повторений указанного литерала. Например, 'ALL "14"' означает "14" или "1414" или "141414" и т. д.

Некоторые константы в Коболе могут быть представлены в виде **фигуральных констант**. Фигуральная константа — это слово Кобола, обозначающее некоторый литерал (или класс литералов).

Как видно из табл. 14.3, каждая фигуральная константа, кроме последней, может быть записана более чем одним способом. Конкретное значение фигуральной константы в программе зависит от контекста, в котором она используется.

Имена, переменные и структуры данных

Переменная в Коболе называется **элементарной единицей данных**. Она является **именем данных** и соответствует одной или более величинам, которые могут изменяться при выполнении программы. Имя данных может быть либо допустимым в Коболе словом, определенным пользователем, либо заполнителем и должно отличаться от всех других имен данных и имен процедур в программе, а также не должно быть зарезервированным словом Кобола.

Словом, определенным пользователем, в Коболе является последовательность знаков, состоящая не более чем из 30 букв (A—Z), цифр (0—9) или знаков черты (-), причем черта не может быть первым или последним знаком последовательности.

Кроме того, имя данных в Коболе должно содержать по крайней мере один алфавитный знак (A—Z). Следующие имена данных являются допустимыми в Коболе:

GROSS-PAY X N19 19N

Имена процедур используются для обозначения секций и параграфов в программе. В отличие от имен данных они могут не содержать ни одного алфавитного знака.

Зарезервированные слова Кобола не могут использоваться в качестве имен данных или процедур. В табл. 14.4 дается полный список зарезервированных слов Кобола. Черта под зарезервированным словом указывает его допустимое сокращение. Например, слово **CORRESPONDING** может быть кратко записано как **CORR** и при этом его смысл не меняется.

Большинство реализаций Кобола содержат дополнительные зарезервированные слова, соответствующие элементам расширения языка. Читателю рекомендуется запомнить все эти зарезервированные слова, поскольку имена переменных не должны совпадать ни с одним из них. Это приводит к определенным неудобствам, так как некоторые из зарезервированных слов (например, **PAGE**) являются естественными кандидатами на имена переменных.

Таблица 14.4

ACCEPT	74	COMMUNICATION	74	EGI	IF
ACCESS		<u>COMPUTATIONAL</u>		ELSE	IN
68 ACTUAL		COMPUTE	74	EMI	INDEX
ADD		CONFIGURATION	74	ENABLE	INDEXED
68 ADDRESS		CONTAINS		END	INDICATE
ADVANCING		CONTROL		END-OF-PAGE	74 INITIAL
AFTER		CONTROLS		ENTER	INITIATE
ALL		COPY		ENVIRONMENT	INPUT
ALPHABETIC		<u>CORRESPONDING</u>		EOP	INPUT-OUTPUT
74 ALSO	74	COUNT	74	EQUAL	74 INSPECT
ALTER		CURRENCY		ERROR	INSTALLATION
ALTERNATE					INTO
AND		DATA	74	ESI	INVALID
ARE	74	DATE		EVERY	IS
<u>AREAS</u>		DATE-COMPILED	68	EXAMINE	
ASCENDING		DATE-WRITTEN	74	EXCEPTION	<u>JUSTIFIED</u>
ASSIGN	74	DAY		EXIT	
AT	74	DEBUG-CONTENTS	74	EXTEND	KEY
AUTHOR	74	DEBUG-ITEM		FD	LABEL
BEFORE	74	DEBUG-LINE		FILE	LAST
68 BEGINNING	74	DEBUG-NAME		FILE-CONTROL	LEADING
BLANK	74	DEBUG-SUB-1	68	FILE-LIMIT	LEFT
BLOCK	74	DEBUG-SUB-2	68	FILE-LIMITS	74 LENGTH
74 BOTTOM	74	DEBUG-SUB-3		FILLER	LESS
BY	74	DEBUGGING		FINAL	LIMIT
		DECIMAL-POINT		FIRST	LIMITS
74 CALL		DECLARATIVES		FOOTING	74 LINAGE
74 CANCEL	74	DELETE		FOR	74 LINAGE-COUNTER
74 CD	74	DELIMITED		FROM	LINE
CF	74	DELIMITER			LINE-COUNTER
CH		DEPENDING		GENERATE	LINES
74 CHARACTER		DESCENDING		GIVING	74 LINKAGE
CHARACTERS	74	DESTINATION		GO	LOCK
CLOCK-UNITS		DETAIL		GREATER	<u>LOW-VALUES</u>
CLOSE	74	DISABLE		GROUP	
COBOL		DISPLAY			MEMORY
CODE		DIVIDE		HEADING	74 MERGE
74 CODE-SET		DIVISION		<u>HIGH-VALUES</u>	74 MESSAGE
74 COLLATING		DOWN		I-O	MODE
COLUMN	74	DUPLICATES		I-O-CONTROL	MODULES
COMMA	74	DYNAMIC		IDENTIFICATION	

MOVE	PROGRAM-ID	SECURITY	THAN
MULTIPLE	74 QUEUE	68 SEEK	<u>THROUGH</u>
MULTIPLY	<u>QUOTES</u>	74 SEGMENT	74 TIME
74 NATIVE	RANDOM	SEGMENT-LIMIT	TIMES
NEGATIVE	RD	SELECT	TO
NEXT	READ	74 SEND	74 TOP
NO	74 RECEIVE	SENTENCE	74 TRAILING
NOT	RECORD	74 SEPARATE	TYPE
68 NOTE	RECORDS	74 SEQUENCE	UNIT
NUMBER	REDEFINES	SEQUENTIAL	74 UNSTRING
NUMERIC	REEL	SET	UNTIL
OBJECT-COMPUTER	74 REFERENCES	SIGN	UP
OCCURS	74 RELATIVE	SIZE	UPON
OF	RELEASE	SORT	USAGE
OFF	REMAINDER	74 SORT-MERGE	USE
OMITTED	68 REMARKS	SOURCE	USING
ON	74 REMOVAL	SOURCE-COMPUTER	
OPEN	RENAMES	<u>SPACES</u>	VALUE
OPTIONAL	REPLACING	SPECIAL-NAMES	VALUES
OR	REPORT	STANDARD	VARYING
74 ORGANIZATION	REPORTING	74 STANDARD-1	
OUTPUT	REPORTS	74 START	WHEN
74 OVERFLOW	RERUN	STATUS	WITH
	RESERVE	STOP	WORDS
PAGE	74 REWRITE	74 STRING	WORKING-STORAGE
PAGE-COUNTER	RF	74 SUB-QUEUE-1	WRITE
PERFORM	RH	74 SUB-QUEUE-2	
PF	RIGHT	74 SUB-QUEUE-3	ZERO
PH	ROUNDED	SUBTRACT	<u>ZEROES</u>
<u>PICTURE</u>	RUN	SUM	+
PLUS	SAME	74 SUPPRESS	-
74 POINTER	SD	74 SYMBOLIC	.
POSITION	SEARCH	<u>SYNCHRONIZED</u>	/
POSITIVE	SECTION		**
74 PRINTING		74 TABLE	>
PROCEDURE		68 TALLY	<
74 PROCEDURES		TALLYING	=
PROCEED		TAPE	
68 PROCESSING		74 TERMINAL	
74 PROGRAM		TERMINATE	
		74 TEXT	

Все переменные, используемые в Кобол-программе, должны быть описаны в Разделе данных. Переменная может появиться в программе самостоятельно как элемент таблицы (т. е. массива) или структуры данных (т. е. поле записи). В соответ-

ствии с этим переменная описывается одним из следующих двух способов:

1. 77 описание имени-данных
2. уровень описание имени-данных

Первая форма используется в тех случаях, когда переменная появляется как независимый элемент, вторая — когда переменная появляется либо как элемент таблицы, либо как элемент структуры данных. Здесь имя-данных — это имя переменной. Описание — ряд так называемых **статей**, которые задают область и тип значений переменной. Уровень — двузначное число, определяющее иерархический уровень переменной в таблице или структуре данных.

Существует несколько различных статей, которые могут появиться в описании переменной. Наиболее важные из них указаны в табл. 14.5.

Таблица 14.5

Статья	Назначение
REDEFINES	Указанная переменная разделяет один и тот же участок памяти с другой переменной (см. разд. 14.7)
JUSTIFIED	Выравнивание нечисловых переменных не по левому, а по правому краю поля (с добавлением пробелов слева). (См. разд. 14.7)
PICTURE	Определяет, какие значения, числовые или нечисловые, будет принимать переменная, а также указывает область ее значений. Описывает также любые знаки редактирования (например, "\$"), которые могут использоваться при печати значения переменной
USAGE	Определяет, будет ли числовая переменная использоваться при вычислениях, и задает, таким образом, внутреннее представление ее значений
VALUE	Определяет начальные значения, которые должны быть присвоены переменной в начале выполнения программы

Среди них самой важной является статья **PICTURE**, поскольку она описывает особенности переменной. Статья **PICTURE** имеет следующую общую форму:

PICTURE [IS] строка

Здесь строка описывает класс переменной (алфавитный, цифровой или алфавитно-цифровой), область значений и некоторые другие характеристики редактирования. Она записывается в виде последовательности знаков из списка, данного в табл. 14.6.

В соответствии со статьей **PICTURE** переменная может быть алфавитной, алфавитно-цифровой, числовой, алфавитно-

Таблица 14.6

Знак в статье	Смысл
A	Соответствует одному алфавитному знаку (A — Z) или знаку пробела ()
B	Соответствует одному знаку пробела
S	Соответствует знаку числа, который не выдается при печати значения
V	Соответствует десятичной точке в числе, которая не выдается при его печати
X	Соответствует одному алфавитно-цифровому знаку
Z	Соответствует ведущему нулю числа, который не выдается при печати
9	Соответствует одной десятичной цифре в числе
.	Соответствует десятичной точке в числе, которая выдается при его печати
\$	Соответствует ведущему знаку доллара, который должен быть отпечатан вместе с числом
—	Соответствует знаку числа, который печатается как “—”, если выводимое число отрицательное, и “_” в противном случае

цифровой редактируемой или числовой редактируемой. Ниже даются определения этих пяти классов переменных и приводятся соответствующие примеры.

Алфавитные переменные. Строка **PICTURE** содержит последовательность литер «A», указывающих на то, что значениями переменной могут быть любые нечисленные значения, содержащие только алфавитные литеры (A—Z) и литеры пробела (). Число литер «A» определяет длину этих значений. Например, тот факт, что переменная с именем **TITLE** содержит некоторые значения, состоящие из 15 алфавитных литер, описывается одним из следующих двух эквивалентных способов (в предположении, что **TITLE** не является частью таблицы или другой структуры):

77 TITLE PIC AAAAAAAAAAAAAAAAAA.

77 TITLE PIC A(15).

Согласно этим описаниям, переменная **TITLE** может содержать 15-литерные значения, состоящие только из пробелов () и букв (A—Z). Например, **TITLE** может принять значение **“ANNUAL_REPORT_”**. Запись **A(15)** во втором варианте описания является удобным сокращением для обозначения последовательности, состоящей из 15 литер “A”.

Алфавитно-цифровые переменные. Строка **PICTURE** содержит последовательность знаков X, указывающих на то, что переменная может принимать любые нечисловые значения, содержащие алфавитные (A—Z), цифровые (0—9) и специальные (., +, * и т. д.) литеры. Число знаков X определяет длину всех значений, которые может принимать переменная. Например, тот факт, что переменная **TITLE** теперь может принимать любые 15-литерные значения, указывается одним из следующих двух способов:

77 **TITLE PIC XXXXXXXXXXXXXXXX.**

77 **TITLE PIC X(15).**

Например, переменная **TITLE** может принять значение **"ANNUAL_REPORT_"** или **"1975_REPORT_"** или любое другое 15-литерное значение.

Числовые переменные. Строка **PICTURE** содержит последовательность цифр 9, которая может начинаться знаком S и содержать знак V. Количество цифр 9 указывает число десятичных цифр в значении переменной. Наличие знака S указывает, что переменная может принимать как положительные, так и отрицательные значения. Позиция знака V среди цифр 9 соответствует (фиксированной) позиции десятичной точки в значении переменной. Например, тот факт, что переменная **GROSS-PAY** будет принимать числовые значения в диапазоне от 0 до 99999.99, переменная **I** будет принимать числовые значения в диапазоне от 0 до 999, а переменная **NET-INCOME** — в диапазоне от —99999.99 до 99999.99, указывается следующим образом (в предположении, что эти переменные не являются частью таблицы или другой структуры):

77 **GROSS-PAY PIC 99999V9.**

77 **I PIC 999.**

77 **NET-INCOME PIC S99999V99.**

Алфавитно-цифровые редактируемые переменные. Строка **PICTURE** содержит последовательность, состоящую из знаков A, X, 9 и B. Область значений переменной описывается последовательностью, состоящей из знаков A, X и 9 в строке **PICTURE**, а каждый знак B соответствует пробелу в указанной позиции значения переменной. Предположим, например, что переменная **TITLE** была описана следующим образом:

77 **TITLE PIC 9999BA(10).**

Тогда **TITLE** может содержать любую 15-знаковую строку, первые четыре знака которой являются цифровыми (0—9), пятый знак — пробелом (), а остальные знаки — алфавитными (A—Z) или пробелом (). Таким образом, переменная **TITLE** вновь может принять значение **"1975_REPORT_"**.

Числовые редактируемые переменные. Строка **PICTURE** содержит последовательность, состоящую из знаков **B**, **V**, **Z**, **9**, **.**, **—**, **\$**. Смысл знаков **B**, **V** и **9** такой же, как и раньше. Кроме того, один или более головных знаков **9** в строке **PICTURE** могут быть заменены знаками **Z**, что указывает на подавление незначащих нулей при печати. Аналогично знак **V** в строке **PICTURE** может быть заменен знаком точки (**.**), указывающим позицию десятичной точки, которая будет добавлена при печати значений. Знак **\$**, располагаемый левее крайнего слева знака **9** или **Z**, будет выдан при печати числа. Знак **\$** может быть записан вместо головных знаков **9** или **Z**. В этом случае знак **\$** будет плавающим, т. е. при печати значения переменной он будет располагаться в ближайшей слева позиции от крайней слева (значащей) цифры. Наконец, в крайней левой позиции строки **PICTURE** может быть записан знак минус (**—**). При печати значения переменной ему соответствуют знаки пробела (**_**) (если значение неотрицательно) или минус (**—**) (если значение отрицательно), располагаемые слева от числа. Знак минус аналогичен знаку **\$** для числовых переменных в том смысле, что описываемые им переменные могут принимать как отрицательные, так и неотрицательные значения. Предположим, например, что области значений переменных **GROSS-PAY** и **NET-INCOME** такие же, какие были раньше, но при печати требуется указывать десятичную точку. Предположим также, что при печати значения переменной **NET-INCOME** головные нули должны быть подавлены (заменены знаком пробела), а при печати значения переменной **GROSS-PAY** в ближайшей слева позиции от крайней слева значащей цифры должен быть указан «плавающий» знак доллара. Тогда эти переменные должны быть описаны следующим образом:

77 GROSS-PAY PIC \$\$\$\$\$.99.

77 NET-INCOME PIC —ZZZZZ.99.

Например, если значения переменных **GROSS-PAY** и **NET-INCOME** были равны 03571.52 и —00025.53 соответственно, они были бы напечатаны как **_ \$3571.52** и **— 25.53**. Следует отметить, что знак минус также может быть определен «плавающим» с помощью тех же соглашений, которые относились к знаку **\$**.

Статья **USAGE** используется для того, чтобы указать, каким образом следует представлять в памяти значения переменной. Она имеет следующую форму:

$$[\text{USAGE} [\text{IS}]] \left\{ \begin{array}{l} \text{DISPLAY} \\ \text{COMPUTATIONAL} \\ \text{INDEX} \end{array} \right\}$$

Если выбирается опция **DISPLAY**, значение переменной будет храниться в таком же виде, в каком оно будет печататься. Если выбирается опция **COMPUTATIONAL**, переменная должна быть числовой (как указано в статье **PICTURE**). В этом случае ее значение будет храниться в двоичной форме. Если выбирается опция **INDEX**, переменная должна быть числовой и может быть использована только в ограниченном числе случаев. Если статья **USAGE** опущена, предполагается использование опции **DISPLAY**.

Статья **VALUE** используется в тех случаях, когда в начале выполнения программы переменной должно быть присвоено значение некоторой константы. Это, конечно, не препятствует тому, чтобы значение переменной изменить в дальнейшем. Эта статья имеет следующую форму:

VALUE [IS] литерал

Здесь литерал — либо числовой литерал (если в статье **PICTURE** переменная описана как числовая или числовая редактируемая), либо нечисловой литерал (если в статье **PICTURE** переменная описана как-то иначе), либо фигуральная константа.

Предположим, например, что в начале выполнения программы определенным выше переменным **TITLE**, **I** и **GROSS-PAY** должны быть присвоены начальные значения “**ANNUAL REPORT—**”, 0 и 3571.52 соответственно. Предположим также, что переменная **I** должна использоваться в основном в арифметических выражениях. Тогда эти переменные следует описать следующим образом:

```
77 TITLE      PIC X(15) VALUE "ANNUAL-REPORT—".
77 I          PIC 999  VALUE 0 COMP.
77 GROSS-PAY PIC $$$$.99 VALUE "—$3571.52".
```

Читатель должен был заметить, что во всех трех описаниях статьи **PICTURE** и **VALUE** записаны в сокращенной форме. Статья **USAGE** во втором описании также сокращена.

Существует два пути представления более чем одного значения одним-единственным именем данных. Один путь связан с использованием так называемой записи описания входа.

Таблица — это одномерный, двумерный или трехмерный набор значений, имеющих одинаковые характеристики. На

A	B			
	006	17	42	00 -10
	035	18	03	00 -11
	217	19	47	19 -12
	001	20	48	49 22
	023	00	00	-2 -3

Рис. 14.1.

рис. 14.1 изображены одномерная таблица (или, короче, список) **A** из пяти элементов, каждый из которых является трехзначным неотрицательным числом, и двумерная таблица **B**, состоящая из пяти строк, каждая из которых содержит по четыре элемента, являющихся двузначными целыми числами. Конкретные значения элементов **A** и **B** носят иллюстративный характер и не существенны для дальнейшего изложения.

Одномерная таблица описывается следующим образом:

01 имя-таблицы

02 описание-имени-входа статья **OCCURS**.

Двумерная таблица описывается следующим образом:

01 имя-таблицы.

02 имя-строки статья **OCCURS**.

03 описание-имени-входа статья **OCCURS**.

Трехмерная таблица описывается следующим образом:

01 имя-таблицы.

02 имя-строки статья **OCCURS**.

03 имя-столбца статья **OCCURS**.

04 описание-имени-входа статья **OCCURS**

В каждом из этих случаев имя-таблицы — это уникальное слово, определенное пользователем для обозначения таблицы, а описание — отдельные статьи (например, статья **PICTURE**), которые описывают характеристики типичного элемента таблицы. Эти статьи записываются точно так же, как для обычной (скалярной) переменной 77-го уровня. Имя-строки, имя-столбца и имя-входа — это слова, определенные пользователем для обозначения типичной строки, столбца и одного входа соответственно.

Статья **OCCURS** используется для определения числа элементов в каждом измерении таблицы. Она имеет следующие две формы:

1 **OCCURS** целое [TIMES] [INDEXED] [BY] индексы]

2. **OCCURS** целое-1 TO целое-2 [TIMES]
[DEPENDING [ON] имя-данных]
[INDEXED [BY] индексы]

Здесь *целое* — число элементов в измерении, соответствующем данной статье **OCCURS**. Целое-1 и целое-2 — нижняя и верхняя границы числа элементов в данном измерении (в конечном счете это число изменяется в ходе выполнения программы). Если во второй форме используется опция **DEPENDING ON**, то значение имени-данных определяет текущее число элементов в данном измерении в ходе выполнения программы. Имя-дан-

ных должно быть отдельно определено как числовая переменная. Опция **INDEXED BY** используется в тех случаях, когда к элементу таблицы обращаются с помощью одной или более переменных типа **INDEX**. Имена этих переменных указываются в данной опции как индексы.

Возвращаясь к приведенным выше таблицам, можно описать **A** и **B** (но не указанные значения их элементов) следующим образом:

01 TABLE-A.

02 A PIC 999 OCCURS 5.

01 TABLE-B.

02 ROW-B OCCURS 5.

03 B OCCURS 4 PIC S99.

Из описания таблицы **B** видно, что статья **OCCURS** может как предшествовать статье **PICTURE**, так и следовать за ней.

Начальные значения элементов таблиц **A** и **B**, данные на рис. 14.1, не могут быть присвоены в описании **A** и **B**. Для задания начального значения массива может быть использована статья **VALUE**, однако при этом все элементы должны иметь одно и то же значение и, кроме того, в статье **USAGE** для них должна быть указана опция **DISPLAY** (а не **COMP** или **INDEX**). Поэтому присвоение начальных значений массивам обычно переносится в Раздел процедур.

Обращение к обычной переменной может быть выполнено в Разделе процедур простым указанием имени этой переменной. Однако обращение к входу таблицы осуществляется в Разделе процедур одним из следующих двух способов в зависимости от того, является ли таблица индексированной или нет (т. е. содержится ли опция **INDEXED BY** в ее описании или нет).

Если таблица не является индексированной, обращение к отдельному входу осуществляется с помощью указателей следующим образом:

имя-входа_(указатель[,_указатель][,_указатель])

Здесь *имя-входа* — это имя отдельного входа в описании таблицы (например, **A** и **B** являются именами входов в наших описаниях). Число задаваемых указателей должно соответствовать числу измерений (1, 2 или 3) в описании таблицы. Знак **_** указывает, что в соответствующей позиции должен стоять в точности один пробел. Каждый указатель должен быть либо числовым литералом, либо именем числовой переменной. Значение указателя должно находиться в области, указанной в статье **OCCURS** для данного измерения,

Например, при обращении к третьему элементу таблицы **A** следует записать

A_(3)

При обращении к **I**-му элементу таблицы **A** следует записать

A_(I)

Здесь **I** — числовая переменная, принимающая значения 1, 2, 3, 4 или 5 (так как **A** содержит 5 элементов).

Аналогично, при обращении к элементу таблицы **B**, расположенному в третьей строке и втором столбце, следует записать

B_(3,-2)

Обращение к элементу **I**-й строки и **J**-го столбца таблицы **B** осуществляется следующим образом:

B_(I,-J)

где **I** и **J** — числовые переменные, принимающие значения 1, 2, 3, 4 или 5 и 1, 2, 3 или 4 соответственно.

С другой стороны, если таблица является индексированной, то ссылка на ее отдельный элемент может, кроме того, осуществляться с помощью аналогичного выражения, имеющего следующую форму:

```
имя-входа_(индексное-имя[_{±}_целое]
            [,_индексное-имя[_{±}_целое]]
            [,_индексное-имя[_{±}_целое]]
```

Здесь *имя-входа* вновь означает отдельный вход в описании таблицы, а число указанных *индексных-имен* равно числу (1, 2 или 3) измерений таблицы.

Предположим например, что таблица **B** была заново описана следующим образом:

01 TABLE-B.

02 ROW-B OCCURS 5 INDEXED BY I.

03 B OCCURS 4 PIC S99 INDEXED BY J.

Как и в предыдущем описании, здесь **B** — это таблица тех же размеров (5×4) и тип ее элементов тот же самый. Однако переменные **I** и **J** определяются как индексы для обращения соответственно к строкам и столбцам таблицы **B**. В рассматриваемом случае при использовании переменных **I** и **J** в качестве индексов для обращения к одному-единственному элементу таблицы **B** их значения должны лежать в диапазонах от 1 до 5 и от 1 до 4 соответственно. Для ссылки на элемент из **I**-й стро-

```

[ALLEN,B,TUCKER,.....]
[275407437]
[25400.00]
[1800,BULL,RUN,ALEXANDRIA,VA.,22200,.....]

```

Рис. 14.2.

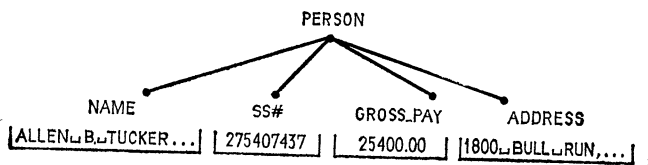


Рис. 14.3.

ки и **J**-го столбца таблицы **B** вновь может быть использовано выражение

B_(I,_J)

Использование индексов вместо указателей дает несколько большие возможности для ссылки на элемент таблицы. Например, для обращения к элементу, расположенному в $(I+1)$ -й строке и $(J-2)$ -м столбце таблицы **B**, может быть использовано выражение

B_(I+_1,_J_-_2)

которое по-прежнему действительно только в том случае, когда $I+1$ находится в пределах от 1 до 5, а $J-2$ — в пределах от 1 до 4.

Статья описания записи — это описание структурированного набора отдельных переменных, которые, как правило, не являются одинаковыми по типу. Например, можно описать запись с именем **PERSON** (человек) как набор следующих четырех переменных: алфавитно-цифровой переменной **NAME** (имя), принимающей 25-значковые значения, девятизначной числовой переменной **SS-NO** (номер страхового полиса), числовой переменной **GROSS-PAY** (годовой оклад), значения которой могут изменяться от 0 до 99999.99, и алфавитно-цифровой переменной **ADDRESS** (адрес), принимающей 40-значковые значения. (Напомним, что более подходящее имя **ADDRESS** является зарезервированным словом и поэтому в данном случае не может быть использовано.) На рис. 14.2 дан пример записи **PERSON**. В этом примере видны два уровня структуры. На элементарном уровне располагаются четыре переменные, а на высшем уровне — переменная **PERSON**, которая включает в себя все четыре переменные как подчиненные. Статью описания записи можно также изобразить в виде дерева (рис. 14.3). В этом слу-

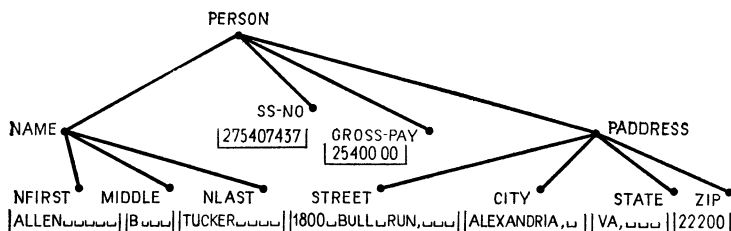


Рис. 14.4.

чае имя (например, **PERSON**) статьи описания записи является корнем дерева, а отдельные элементы — его листьями.

Для определения статьи описания записи в Коболе различным именам в структуре приписываются **номера уровней**. Имени всей структуры (например, **PERSON**) приписывается номер уровня 01, именам всех данных следующего уровня приписывается номер 02 и т. д. Внутри одного уровня имена перечисляются слева направо. Таким образом, приведенная выше структура может быть определена следующим образом:

```

01 PERSON.
  02 NAME      PIC X(25).
  02 SS-NO     PIC 9(9).
  02 GROSS-PAY PIC 9(5)V99.
  02 PADDRESS  PIC X(40).
  
```

Отметим также, что каждое отдельное значение в статье описания записи должно заканчиваться точкой, а статья **PICTURE** описывает только элементарные данные.

Некоторые переменные уровня 02 в этом примере сами могут быть структурированными. С помощью дерева, изображенного на рис. 14.4, показано, как переменная **NAME** может разбиваться на три элемента (фамилия, имя, отчество), а переменная **PADDRESS** — на четыре (улица, город, штат, почтовый индекс). В этом случае имеем следующую статью описания записи.

```

01 PERSON.
  02 NAME.
    03 NFIRST  PIC X(10).
    03 MIDDLE  PIC X(5).
    03 NLASt   PIC X(10).
  02 SS-NO     PIC 9(9).
  02 GROSS-PAY PIC 9(5)V99.
  02 PADDRESS.
    03 STREET  PIC X(17).
  
```

03 CITY	PIC X(12).
03 STATE	PIC X(6).
03 ZIP	PIC 9(5).

Мы вновь обращаем внимание читателя на то, что статья **PICTURE** описывает только те переменные, которые сами не разбиваются на элементы более низкого уровня. Они называются **элементарными данными** и соответствуют листьям в древообразном описании. Остальные данные в статье описания записи называются **групповыми**. В рассматриваемом примере **PERSON**, **NAME** и **PADDRESS** являются групповыми данными.

Для того чтобы из Раздела процедур Кобол-программы обратиться ко всей статье описания записи, следует просто указать имя, которому приписано число уровня 01. Например, для обращения ко всему набору значений, определенному в приведенном выше примере, используется имя **PERSON**. Поэтому все статьи описания записи в одной Кобол-программе должны иметь различные имена (т. е. уникальные имена на уровне 01).

Однако имена на подчиненных уровнях (например, 02, 03 и т. д.) одной статьи описания записи могут совпадать с подчиненными именами другой статьи описания записи. Предположим, например, что требуется одновременно хранить значения переменных **NAME**, **SS-NO**, **GROSS-PAY** и **PADDRESS**, соответствующих двум людям, скажем **PERSON-A** и **PERSON-B**. В этом случае нужно написать следующие статьи описания записи:

01 PERSON-A.	
02 NAME	PIC X(25).
02 SS-NO	PIC 9(9).
02 GROSS-PAY	PIC 9(5)V99.
02 PADDRESS	PIC X(40).
01 PERSON-B.	
02 NAME	PIC X(25)
02 SS-NO	PIC 9(9).
02 GROSS-PAY	PIC 9(5)V99.
02 PADDRESS	PIC X(40).

Обращение к переменной **SS-NO**, соответствующей **PERSON-A**, осуществляется с помощью уточненного имени

SS-NO OF PERSON-A

С другой стороны, если требуется обратиться к переменной **SS-NO**, соответствующей **PERSON-B**, то следует записать

SS-NO OF PERSON-B

Очевидно, что уточнение необходимо только в том случае, когда указанное имя (в данном случае **SS-NO**) не является уникальным. Если же оно уникально, то указания самого имени достаточно для однозначного обращения к данным.

Отметим также, что в статье описания записи может содержаться таблица. Предположим, что для каждого человека (**PERSON**) требуется хранить не текущий адрес, а последние пять адресов. Тогда статью описания записи следует переписать следующим образом:

```

01 PERSON.
    02 NAME      PIC X(25).
    02 SS-NO     PIC 9(9).
    02 GROSS-PAY PIC 9(5)V99.
    02 PADDRES   PIC X (40) OCCURS 5.
  
```

Указатели, индексы и уточненные имена для таблиц, содержащихся в статье описания записи, используются точно так же, как было описано в предыдущих параграфах. Наконец, ссылка на любой уникальный элемент данных, будь то отдельная переменная, элемент таблицы или элементарного уровня статьи описания записи, называется в Коболе **идентификатором**.

14.3. ОПЕРАТОРЫ КОБОЛА

В этой главе мы не будем останавливаться на рассмотрении всех операторов Кобола, поскольку важно сосредоточить внимание на тех из них, которые используются наиболее часто. В табл. 14.7 перечислены операторы, которые будут нами рассмотрены, и дано краткое описание их назначения в Кобол-программе.

Таблица 14.7

Оператор	Назначение
ADD	Выполнение одной или нескольких арифметических операций (сложение, вычитание и т. д.) и назначение результата в качестве нового значения переменной или элемента таблицы
SUBTRACT	
MULTIPLY	
DIVIDE	
COMPUTE	

Продолжение табл. 14.7

Оператор	Назначение
CALL EXIT PROGRAM	Обращение к подпрограмме с последующей передачей управления от подпрограммы вызывающей программе. Рассматривается в разд. 14.5
CLOSE DELETE OPEN READ REWRITE WRITE	Выполнение различных операций ввода-вывода. Рассматривается в разд. 14.4
COPY	Включение исходного текста из библиотеки в Кобол-программу во время компиляции. Рассматривается в разд. 14.7
GENERATE INITIATE TERMINATE	Используются при составлении отчетов. Рассматриваются в разд. 14.7
GO TO IF	Изменение последовательности выполнения операторов программы
INSPECT SEARCH SET	Исследование символьной строки или таблицы
MERGE RELEASE RETURN SORT	Сортировка файла или объединение двух файлов. Рассматриваются в разд. 14.7
MOVE STRING UNSTRING	Пересылка, сцепление или расщепление значений данных соответственно
PERFORM STOP	Повторное выполнение группы операторов. Завершение выполнения программы

В оставшейся части этого раздела будут рассмотрены все перечисленные выше операторы, изучение которых не намечено в других разделах. Рассмотрим вначале простую Кобол-программу, приведенную на рис. 14.5. Несмотря на то что эта программа не имеет никакого практического значения, она иллюстрирует применение многих основных типов операторов Кобол.

IDENTIFICATION DIVISION.
 PROGRAM-ID. SIMPLE.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. IBM-370-145.
 OBJECT-COMPUTER. IBM-360-145.

DATA DIVISION.
 WORKING-STORAGE SECTION.
 77 H PIC 9 (5) COMP.
 77 I PIC 9 (5) COMP.
 77 J PIC 9 (5) COMP.
 01 ARRAY-A.
 02 A PIC 999 OCCURS 5.
 01 ARRAY-B.
 02 ROW-B OCCURS 5.
 03 B PIC S99 OCCURS 4.
 01 PERSON.
 02 NAME PIC X (25).
 02 SS-NO PIC 9 (9).
 02 GROSS-PAY PIC 9 (5) V99.

PROCEDURE DIVISION.

- * ЭТА ПРОГРАММА ЛИШЬ ИЛЛЮСТРИРУЕТ ПРИМЕНЕНИЕ
- * НЕКОТОРЫХ ОСНОВНЫХ ОПЕРАТОРОВ КОБОЛА

PERFORM S1 VARYING I FORM 1 BY 1 UNTIL I = 5.
 S1. COMPUTE A (I) = 6 - I.
 S2. MOVE 1 TO H.
 LOOP. IF H < 6 THEN
 MOVE 1 TO B (H, 1)
 COMPUTE B (H, 2) = 5 + 3 * (H - A (H))
 ADD 1 TO H
 GO TO LOOP.
 PERFORM S3 VARYING I FROM 1 BY 1 UNTIL I = 5.
 S3. MOVE A (I) TO B (I, 3).
 COMPUTE B (I, 4) = B (I, 1) + B (I, 2) + B (I, 3).
 S4. MOVE 'ALLEN B. TUCKER' TO NAME.
 MOVE 25400.00 TO GROSS-PAY.
 STOP RUN.

Рис. 14.5.

Каждая Кобол-программа содержит четыре раздела, назначение которых было пояснено выше. Эти четыре раздела должны располагаться в том порядке, который указан в приводимой программе. Каждый раздел начинается с заголовка раздела (например, **IDENTIFICATION DIVISION**), который указывает его начало.

Читатель должен вспомнить описания переменных (**H**, **I**

и **J**) и таблиц (**A** и **B**), а также определение статьи описания записи (**PERSON**) в Разделе данных. Операторы, которые действительно управляют выполнением программы, расположены в Разделе процедур. Читатель должен также заметить довольно строгую синтаксическую структуру данной программы.

Обратимся теперь к рассмотрению Раздела процедур в этой программе. В общем случае он может состоять из одной или более секций. Каждая секция состоит из одного или более параграфов, а каждый параграф состоит из одного или более предложений.

Секция всегда начинается с заголовка секции, который имеет следующую форму:

имя-секции **SECTION.**

Здесь *имя-секции* дает заголовок секции и может быть любым уникальным незарезервированным словом, определенным пользователем. В тех случаях, когда Раздел процедур содержит только одну секцию, заголовок секции может быть опущен. Этот случай имеет место для рассматриваемой программы.

Параграф всегда имеет следующую форму:

имя-параграфа. последовательность-предложений

Здесь *имя-параграфа* дает название параграфу и может быть любым уникальным незарезервированным словом, определенным пользователем. В *последовательности-предложений* записаны предложения, входящие в данный параграф. Конец параграфа граничит с началом следующего параграфа (или является концом программы, если больше параграфов нет). В рассматриваемой программе Раздел процедур содержит пять параграфов с именами **S1**, **S2**, **LOOP**, **S3** и **S4**.

Предложение — это последовательность, состоящая из одного или более операторов, которая всегда заканчивается точкой (.) и пробелом (—). Первые два параграфа в рассматриваемой программе содержат по одному предложению, каждое из которых состоит из одного оператора. Третий параграф содержит два предложения, первое из которых состоит из пяти операторов (**IF**, **MOVE**, **COMPUTE**, **ADD** и **GO TO**), а второе — из одного. Четвертый параграф содержит два предложения и пятый — три.

Имена секций и параграфов, а также сам заголовок Раздела процедур должны начинаться с позиции (называемой областью **A**) на бланке, расположенной слева от позиции (называемой областью **B**), отведенной для предложений, перед которыми не стоит имя параграфа. Кроме того, заголовки секций и заголовок Раздела процедур должны появляться в строке самостоятельно.

Оператор GO TO

Операторы в Кобол-программе выполняются в том порядке, в котором они записаны, если не появляется оператор, который может изменить этот порядок. Одним из таких операторов является оператор **GO TO**, имеющий одну из следующих форм:

1. **GO TO** имя-процедуры
2. **GO TO** список-имен-процедур **DEPENDING [ON]** идентификатор

Здесь *имя-процедуры* — это имя параграфа или секции, содержащееся в Разделе процедур. *Список-имен-процедур* — это список, состоящий из одного или более (скажем, *n*) имен процедур, которые отделяются друг от друга запятой и пробелом (, —). *Идентификатор* — имя элементарной числовой переменной, принимающей только целочисленные значения.

При выполнении первой формы оператора **GO TO** осуществляется непосредственная передача управления первому оператору, расположенному в первом предложении параграфа или секции с указанным именем-процедуры. В рассматриваемой программе выполнение оператора

GO TO LOOP

приведет к передаче управления оператору, начинающемуся выражением **“IF N < 6 THEN ...”**, так как он является первым оператором в параграфе с именем **LOOP**.

При выполнении второй формы оператора **GO TO** осуществляется непосредственная передача управления на начало оператора (или секции) с первым, вторым, ...или *n*-м именем в списке-имен-процедур в зависимости от того, с какой из величин 1, 2, ... или *n* совпадает текущее значение идентификатора. Если текущее значение идентификатора не совпадает ни с одной из этих величин, то выполняется оператор, следующий за оператором **GO TO**. Вторая форма оператора **GO TO** в рассматриваемой программе не используется.

Оператор STOP

Оператор **STOP** завершает выполнение программы. Он имеет следующую форму:

STOP RUN

Не требуется, чтобы оператор **STOP** был последним в программе. Он располагается в тех местах Раздела процедур, в которых выполнение программы должно завершиться.

Арифметические выражения

Арифметические выражения в Кобол-программе обозначают некоторую последовательность арифметических действий (например, сложение, вычитание и т. д.). Они используются в операторах **COMPUTE** и **IF**. Арифметическое выражение может быть одним из следующих:

1. Имя элементарной числовой переменной.
2. Числовой литерал.
3. Два арифметических выражения, разделенные знаком арифметической операции.
4. Арифметическое выражение, заключенное в скобки. Ниже приводятся знаки арифметических операций и указывается их смысл.

Знак арифметической операции	Смысл
+	Сложение
—	Вычитание
*	Умножение
/	Деление
**	Возведение в степень

Каждая из этих операций выполняется с двумя значениями, одно из которых пишется слева от знака, а другое — справа. Например, арифметическое выражение

$$N + 2$$

означает сложение значения переменной **N** с числовым литералом 2.

Две или более арифметических операций выполняются последовательно. Например, если требуется к сумме **N + 2** прибавить значение переменной **I**, то следует записать выражение (опуская при этом занумерованные стрелки)

$$N + 2 + I$$

$\uparrow \quad \uparrow$
 ① ②

Здесь вначале выполняется сложение **N + 2**, а затем к полученной сумме прибавляется значение переменной **I**, как показано занумерованными стрелками.

Две или более операций в арифметическом выражении выполняются слева направо, за исключением двух случаев. Во-первых, среди операций существует следующая иерархия. Вначале выполняются все операции возведения в степень (**) (слева направо), затем — все операции умножения (*) и деления (/) (слева направо) и наконец — все операции сложения (+) и вычитания (—) (слева направо). Это означает, что значение выражения

$$\begin{array}{c} \mathbf{H + 2 * I} \\ \uparrow \quad \uparrow \\ \textcircled{2} \quad \textcircled{1} \end{array}$$

равно сумме \mathbf{H} и $\mathbf{2 * I}$, а не произведению $\mathbf{H + 2}$ и \mathbf{I} .

Во-вторых, когда указанный порядок выполнения операций не устраивает программиста, он может изменить его, указывая в скобках те операции, которые должны быть выполнены первыми. Например, для вычисления произведения величин $\mathbf{H + 2}$ и \mathbf{I} программист использует выражение

$$\begin{array}{c} \mathbf{(H + 2) * I} \\ \uparrow \quad \uparrow \\ \textcircled{1} \quad \textcircled{2} \end{array}$$

Следующие три арифметических выражения вычисляются в Разделе процедур приведенной выше программы.

$$\begin{array}{c} \mathbf{6 - I} \\ \uparrow \\ \textcircled{1} \\ \\ \mathbf{5 + 3 * (H - A(H))} \\ \uparrow \quad \uparrow \quad \uparrow \\ \textcircled{3} \quad \textcircled{2} \quad \textcircled{1} \\ \\ \mathbf{B(1, 1) + B(1, 2) + B(1, 3)} \\ \uparrow \quad \uparrow \\ \textcircled{1} \quad \textcircled{2} \end{array}$$

Читателю следует обратить внимание на то, с какой тщательностью при написании этих выражений используются пробелы. И это не случайно. В Коболе требуется, чтобы слева и справа от каждого знака арифметической операции в арифметическом выражении был по крайней мере один пробел. Поэтому выражения

$$\mathbf{6 - I} \quad \text{и} \quad \mathbf{6 - I}$$

не эквивалентны и допустимым является только последнее.

Арифметические операторы

В Коболе имеются следующие пять арифметических операторов: **ADD**, **SUBTRACT**, **MULTIPLY**, **DIVIDE** и **COMPUTE**. Эти операторы имеют опции **ROUNDED** и **SIZE ERROR**, которые будут рассмотрены в конце этого раздела.

Оператор **ADD** может быть записан в одной из следующих форм:

1. **ADD** список-значений **TO** список-идентификаторов [**ROUNDED**]
[опция **SIZE ERROR**]
2. **ADD** значение, список-значений **GIVING**
список-идентификаторов [**ROUNDED**]
[опция **SIZE ERROR**]

Здесь *список-значений* — это список, состоящий из одного или более числовых идентификаторов и (или) числовых литералов, отделенных друг от друга запятой и пробелом (, _). *Список-идентификаторов* — список, состоящий из одного или более числовых идентификаторов, отделенных друг от друга запятой и пробелом (, _). *Значение* — один-единственный числовой идентификатор или литерал.

В результате выполнения первой формы оператора **ADD** каждому идентификатору из *списка-идентификаторов* присваивается значение, равное сумме его текущего значения и всех значений из *списка-значений*. Первая форма оператора **ADD** используется в рассматриваемой программе:

ADD 1 TO N

Здесь *список-значений* состоит всего из одной величины, которая прибавляется к текущему значению переменной **N** (единственному элементу *списка-переменных*).

В результате выполнения второй формы оператора **ADD** каждому идентификатору из *списка-идентификаторов* присваивается результат суммирования значения идентификатора или литерала со всеми значениями идентификаторов и литералов из *списка-значений*. Поэтому при выполнении данного вычисления предыдущее значение каждого идентификатора из *списка-идентификаторов* не используется. Хотя вторая форма оператора **ADD** в рассматриваемой программе не используется, приведем оператор, эквивалентный тому, который был дан выше с использованием первой формы:

ADD 1, N GIVING N

Оператор **SUBTRACT** может быть записан в одной из следующих форм:

1. **SUBTRACT** список-значений **FROM** список-идентификаторов **[ROUNDED]**
[опция **SIZE ERROR**]
2. **SUBTRACT** список-значений **FROM** идентификатор **[ROUNDED]**
GIVING список-идентификаторов
[опция **SIZE ERROR**]

Здесь *список-значений* и *список-идентификаторов* имеют такой же смысл, какой они имели в применении к оператору **ADD**.

В результате выполнения первой формы оператора **SUBTRACT** каждому идентификатору из *списка-идентификаторов* присваивается разность его текущего значения и суммы всех значений из *списка-значений*. Например, при выполнении оператора

SUBTRACT 1 FROM I

переменной **I** присваивается новое значение, равное разности ее текущего значения и 1.

В результате выполнения второй формы оператора **SUBTRACT** каждому идентификатору из *списка-идентификаторов* присваивается разность текущего значения *идентификатора* и суммы всех значений *списка-значений*. Однако при выполнении этого оператора текущее значение *идентификатора* не изменяется, если только он не входит в *список-идентификаторов*. Например, для присвоения переменной **H** значения, равного разности значения переменной **I** и 1, можно воспользоваться оператором

SUBTRACT 1 FROM I GIVING H

Текущее значение **I** при этом не изменяется.

Операторы **MULTIPLY** и **DIVIDE** имеют следующие формы:

- MULTIPLY** значение-1 **BY** значение-2 **GIVING** идентификатор-3 **[ROUNDED]**
[опция **SIZE ERROR**]
- DIVIDE** значение-1 **BY** значение-2 **GIVING** идентификатор-3 **[ROUNDED]**
[REMAINDER идентификатор-4]
[опция **SIZE ERROR**]

Здесь *значение-1* и *значение-2* — числовые идентификаторы или числовые литералы, а *идентификатор-3* и *идентификатор-4* — числовые идентификаторы.

В результате выполнения оператора **MULTIPLY** идентификатору-3 присваивается значение, равное произведению значения-1 и значения-2. Если необязательная часть “**GIVING** идентификатор-3” опущена, то значение-2 должно быть идентификатором и результат умножения становится новым значением этого идентификатора. Например, при выполнении оператора

MULTIPLY 3 BY N GIVING I

переменной **I** присваивается величина, равная произведению 3 и текущего значения **N**. Однако при выполнении оператора

MULTIPLY 3 BY N

та же величина становится новым значением переменной **N**.

В результате выполнения оператора **DIVIDE** идентификатору-3 присваивается значение, равное частному от деления значения-1 на значение-2. Если указана необязательная часть "**REMAINDER** идентификатор-4", то из значения-1 вычитается произведение значения-2 и частного, а полученная величина, определяемая как *остаток* (remainder), присваивается идентификатору-4. Например, при выполнении оператора

DIVIDE 5 BY 2 GIVING I REMAINDER J

переменным **I** и **J** присваиваются значения 2 и 1 соответственно.

Оператор **COMPUTE** позволяет указывать несколько арифметических операций, результат выполнения которых присваивается некоторому идентификатору. Он имеет следующую форму:

COMPUTE идентификатор [**ROUNDED**] = арифметическое-выражение
[опция **SIZE ERROR**]

Здесь *идентификатор* — это числовой идентификатор. В результате выполнения оператора **COMPUTE** идентификатору присваивается новая величина, равная значению арифметического выражения. Например, в рассматриваемой программе записан оператор

COMPUTE A (I) = 6 — I

При выполнении этого оператора **I**-му элементу таблицы **A** присваивается значение, равное разности 6 и **I**. Поэтому этот оператор эквивалентен следующему оператору **SUBTRACT**:

SUBTRACT I FROM 6 GIVING A (I)

Опция **ROUNDED** позволяет округлять значение, полученное в результате выполнения арифметических операций, до того, как оно будет присвоено идентификатору, являющемуся целью этих вычислений. Эта опция может быть использована для любого из пяти описанных выше арифметических операторов. Она специфицируется зарезервированным словом **ROUNDED**, помещаемым непосредственно после тех целевых идентификаторов, значения которых должны быть округлены. Проиллюстрируем ее применение на примере оператора

MULTIPLY, предполагая, что используемые переменные описаны следующим образом:

77 HOURS PIC 9(2)V9.

77 RATE PIC 9V99.

77 GROSS-PAY PIC 9(5)V99.

Предположим, что требуется вычислить зарплату **GROSS-PAY** как произведение отработанного времени **HOURS** (например, 37.5) и почасовой расценки **RATE** (например, 5.25 долл.), округляя результат до ближайшего цента. Для этого может быть использован следующий оператор:

MULTIPLY HOURS BY RATE GIVING GROSS-PAY ROUNDED

Переменной **GROSS-PAY** будет присвоено значение 196.88 как результат округления величины 196.875 до ближайшего числа, имеющего два знака после запятой. Если бы опция **ROUNDED** была опущена, то переменной **GROSS-PAY** было бы присвоено значение 196.87. Позиция в десятичной дроби, в которой проводится округление, является всегда самой правой десятичной цифрой в целевом значении.

Состояние *переполнение* возникает в тех случаях, когда число, полученное в результате выполнения арифметической операции, содержит значащих цифр левее запятой больше, чем допустимо для данной целевой переменной (что определяется в статье **PICTURE**). Кроме того, переполнение возникает при попытке деления на нуль. Например, если бы переменная **GROSS-PAY** была описана как **PICTURE 9(2)V99** (а не **9(5)V99**), то в результате выполнения записанного выше оператора **MULTIPLY** возникло бы переполнение.

Если арифметический оператор, при выполнении которого возникает переполнение, не содержит опции **SIZE ERROR**, значение целевого идентификатора становится неопределенным. С другой стороны, арифметический оператор может содержать опцию **SIZE ERROR**, которая имеет следующую форму:

[ON] SIZE ERROR оператор

Здесь *оператор* — это предписываемый оператор, как, например, **GO TO**, который при возникновении переполнения передает управление параграфу, осуществляющему обработку ошибки. Если арифметический оператор содержит опцию **SIZE ERROR**, то при возникновении переполнения текущее значение определяемого идентификатора не изменяется и, кроме того, выполняется указанный *оператор*. Если же это состояние не возникает, выполнение арифметического оператора осуществля-

ется обычным образом. Вернемся к примеру, в котором переменная **GROSS-PAY** описана как **PICTURE 9(2)V99**, и рассмотрим следующий оператор:

**MULTIPLY RATE BY HOURS GIVING GROSS-PAY
ON SIZE ERROR GO TO BAD-GROSS**

В результате выполнения этого оператора переменной **GROSS-PAY** присвоится значение произведения величин **RATE** и **HOURS** при условии, что оно не превосходит 99.99. Однако, если это произведение будет превосходить 99.99, возникнет переполнение и осуществится передача управления параграфу с именем **BAD-GROSS**.

Условия и операторы **IF**

Оператор **IF** используется в тех случаях, когда требуется проверить так называемое *условие* и в зависимости от результата проверки выполнить соответствующую последовательность операторов.

Условия нужны для задания отношений между переменными и значениями. В Коболе существуют различные средства задания условий. Можно выделить следующие группы условий: отношения, условия принадлежности классу, проверки условных переменных и проверки знака.

Отношение состоит из двух произвольных арифметических выражений (а и b), отделенных друг от друга знаком отношения, слева и справа от которого должен быть пробел (—). В табл. 14.8 приводятся операторы отношения (OO) и поясняется смысл выражения “а OO b”.

Таблица 14.8

Оператор отношения (OO)	Смысл выражения “а OO b”
=	Значения а и b равны
<	Значение а меньше, чем значение b
>	Значение а больше, чем значение b
NOT=	Либо $a < b$, либо $a > b$ (т. е. $a \neq b$)
NOT<	Либо $a = b$, либо $a > b$ (т. е. $a \geq b$)
NOT>	Либо $a < b$, либо $a = b$ (т. е. $a \leq b$)

В зависимости от того, выполнено ли условие “а OO b” или нет, вырабатывается значение “истина” или “ложь” соответственно. То есть вначале вычисляются арифметические выражения а и b, а затем проверяется указанное отношение между ними (в обычном арифметическом смысле). В качестве при-

мера рассмотрим следующее отношение, содержащееся в рассматриваемой программе:

$$N < 6$$

При проверке этого отношения вырабатывается значение “истина”, если текущее значение N строго меньше 6.

При записи отношения вместо арифметических выражений могут использоваться нечисловые литералы и нечисловые переменные. Например, для того чтобы проверить, является ли строка “**ALLEN B. TUCKER**” текущим значением переменной **NAME** (которая определена в рассматриваемой программе), можно использовать следующее отношение:

$$NAME = \text{“ALLEN_B_TUCKER_”}$$

При проверке этого отношения вырабатывается значение “истина” или “ложь” в зависимости от того, совпадает посимвольно или нет текущее значение переменной **NAME** с литералом ‘**ALLEN_B_TUCKER_**’.

Для нечисловых значений отношения $<$ и $>$ определяются на основе сортирующей последовательности для набора знаков Кобола. Например, “**A**” $<$ “**B**” есть истина, “**B**” $<$ “**C**” есть истина и т. д. Итак, для двух нечисловых литералов, например **a** и **b**, отношение “**a**” $<$ “**b**” истинно, если выполнено одно из следующих двух условий:

1. Литералы **a** и **b** содержат одинаковое число знаков, первые k (для некоторого $k \geq 0$) знаков в **a** совпадают с соответствующими первыми k знаками в **b**, а $(k + 1)$ -й знак в **a** меньше, чем $(k + 1)$ -й знак в **b**.

2. Литералы **a** и **b** содержат разное число знаков, однако если k более короткому из них приписать справа такое число пробелов, что длины литералов станут равными, то выполняется условие 1.

Аналогично определяется отношение “**a** = **b**”, когда длины **a** и **b** не равны. Поэтому следующее отношение эквивалентно приведенному выше:

$$NAME = \text{“ALLEN_B_TUCKER”}$$

Отношение “**a** $>$ **b**” для нечисловых значений истинно в том случае, если ни одно из отношений “**a** $<$ **b**” и “**a** = **b**” не является истинным. Отношения “**a** NOT $<$ **b**”, “**a** NOT = **b**”, и “**a** NOT $>$ **b**” для нечисловых значений определяются так же, как они были определены для арифметических выражений.

Условие принадлежности классу используется в тех случаях, когда требуется определить, числовым (целиком состоящим из цифр и, возможно, знака) или алфавитным (целиком состоящим из букв (A — Z) и (или) пробелов) является данное зна-

чение. Это условие может быть записано следующим образом:

идентификатор IS [NOT] { **NUMERIC**
 ALPHABETIC }

Здесь *идентификатор* — это любой идентификатор, для которого в статье **USAGE** указана опция **DISPLAY** (а не **INDEX** или **COMPUTATIONAL**). При проверке условия принадлежности классу вырабатывается значение “истина” или “ложь” в зависимости от того, удовлетворяет или нет текущее значение идентификатора указанному критерию. В качестве примера вновь рассмотрим переменную **NAME**, которая использовалась выше. Предположим, что она имеет значение “**ALLEN_B._TUCKER**”. Тогда при проверке условий

NAME IS NUMERIC
NAME IS ALPHABETIC

вырабатывается значение “ложь”, так как значение переменной **NAME**(1) не является числом и (2) содержит знак (а именно “.”), который не является буквой или пробелом.

Проверка условной переменной используется для того, чтобы определить, удовлетворяет ли текущее значение элементарной переменной некоторому условию, задаваемому при описании *условной-переменной*. Для этого сразу же после описания переменной описывается условная-переменная с уровнем 88 следующим образом:

88 условная-переменная список-статей-**VALUE**

Здесь *условная-переменная* — это уникальное незарезервированное слово, определенное пользователем, а *список-статей-VALUE* — последовательность из одной или более статей **VALUE**, отделенных друг от друга запятой и пробелом (, _). Статья **VALUE** имеет одну из следующих форм:

1. **VALUE [IS]** литерал-1
2. **VALUES [ARE]** литерал-1 **THROUGH** литерал-2

Здесь *литерал-1* и *литерал-2* — это числовые и нечисловые литералы, согласующиеся со статьей **PICTURE** для переменной.

Предположим, например, что переменная **GROSS-PAY** из рассматриваемой программы заново описана следующим образом:

02 **GROSS-PAY PIC 9(5)V99.**
88 **LOW-PAY VALUES 0 THRU 6000.**
88 **AVERAGE-PAY VALUE 10000.**
88 **HIGH-PAY VALUES 20000 THRU 99999.**

Здесь мы определили три условные переменные, **LOW-PAY** (низкий заработок), **AVERAGE-PAY** (средний заработок) и

HIGH-PAY (высокий заработок), которые связаны с переменной **GROSS-PAY**. Каждая из этих условных переменных определяет конкретное значение или область значений для переменной **GROSS-PAY**.

Проверка любой условной переменной (которая была определена для некоторой переменной) осуществляется простым ее указанием. При этом вырабатывается значение “истина” или “ложь” в зависимости от того, принадлежит или нет текущее значение переменной, связанной с этой условной переменной, области значений, определенной той же условной переменной. Например, если записана условная переменная

LOW-PAY

а текущее значение переменной **GROSS-PAY** равно 25400.00, будет выработано значение “ложь”. Условные переменные **AVERAGE-PAY** и **HIGH-PAY** выработают значения “ложь” и “истина” соответственно, поскольку значение 25400 принадлежит только области, определенной для **HIGH-PAY**.

Проверка знака используется для того, чтобы определить, положительным, отрицательным или нулевым является текущее значение арифметического выражения. Проверка осуществляется в следующей форме:

арифметическое-выражение IS [NOT] $\left\{ \begin{array}{l} \text{POSITIVE} \\ \text{NEGATIVE} \\ \text{ZERO} \end{array} \right\}$

При проверке знака вырабатывается значение “истина” или “ложь” в зависимости от того, удовлетворяет значение выражения указанному условию или нет.

Несколько перечисленных выше условий могут быть объединены с помощью логических операторов **AND** и **OR**. При проверке условия

C₁ AND C₂

(где **C₁** и **C₂** — условия любого из четырех описанных выше типов) вырабатывается значение “истина” или “ложь” в зависимости от того, выполнены ли оба условия **C₁** и **C₂** или нет. Аналогично “**C₁ OR C₂**” “истинно” или “ложно” в зависимости от того, выполнено ли хотя бы одно из условий **C₁** и **C₂** или нет. Оператор **AND** имеет более высокий приоритет, чем **OR**, так что “**C₁ OR C₂ AND C₃**” означает “**C₁ OR (C₂ AND C₃)**”, а не “**(C₁ OR C₂) AND C₃**”. Как уже отмечалось выше, приоритет может быть подавлен с помощью соответствующим образом расставленных скобок.

Оператор **IF** может быть записан в одной из следующих форм:

1. IF условие оператор-1
2. IF условие оператор-1 ELSE оператор-2

Здесь *условие* — это любое из рассмотренных выше условий, а *оператор-1* и *оператор-2* — любая последовательность операторов, содержащая не более одного условного оператора.

Выполнение первой формы оператора IF осуществляется за два шага. Вначале проверяется указанное *условие*. Затем, если результат есть истина, управление передается первому оператору в последовательности, обозначенной как *оператор-1*, в противном случае управление передается оператору, непосредственно следующему за оператором IF.

Выполнение второй формы оператора IF также осуществляется за два шага. Вначале проверяется указанное *условие*. Затем осуществляется передача управления на начало *оператора-1* или *оператора-2* в зависимости от того, какой результат был получен при проверке условия — “истина” или “ложь” соответственно.

В качестве примера первой формы рассмотрим следующий оператор, содержащийся в программе:

	$\overbrace{\text{Условие}}$	
	LOOP. IF $H < 6$	
Оператор-1	{	MOVE 1 TO B (H, 1)
		COMPUTE B (H, 2) = $5 + 3 * (H - A(H))$
		ADD 1 TO H
		GO TO LOOP.

Отметим, что здесь *оператор-1* состоит из нескольких операторов. Поэтому если $H < 6$ есть “истина”, то управление передается первому из них (MOVE...). В противном случае все эти операторы обходятся и управление передается следующему за ними оператору.

Оператор PERFORM

В программировании часто возникает необходимость в написании **управляемых циклов**. Кобол предоставляет возможность написания таких циклов с помощью оператора **PERFORM**.

Управляемый цикл — это последовательность операторов, выполнение которой повторяется до тех пор, пока не будет выполнено определенное условие. Многие такие циклы являются **управляемыми с помощью счетчика**. Это означает, что некоторой управляющей переменной присваивается начальное значение и затем каждый раз после выполнения последовательности операторов значение этой переменной увеличивается и проверяется. Две различные формы этого процесса показаны на

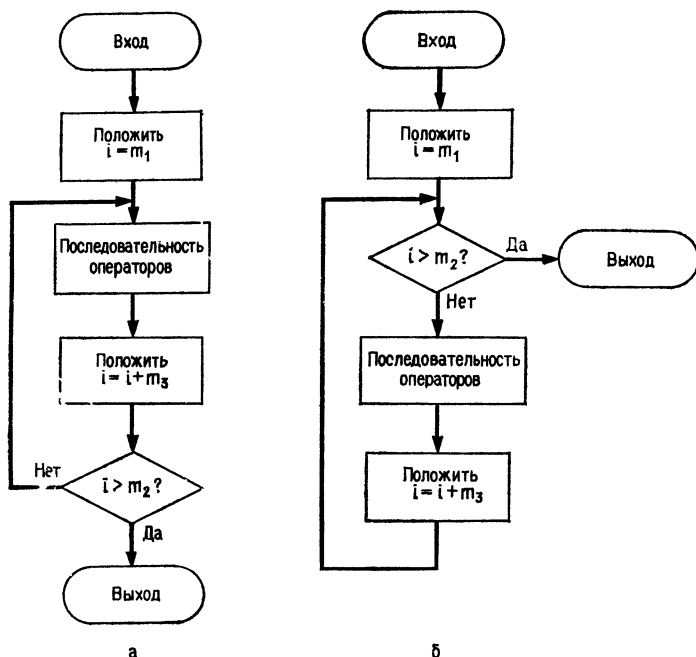


Рис. 14.6.

рис. 14.6. Здесь i — это управляющая переменная, а m_1 , m_2 и m_3 — числовые переменные или числовые литералы, которые являются соответственно начальным значением, конечным значением и шагом для управляющей переменной. Пример блок-схемы (б) содержится в рассматриваемой программе. Здесь H — управляющая переменная, а ее начальное значение, конечное значение и шаг равны 1, 5 и 1 соответственно.

Цикл, изображенный блок-схемой (б), может быть записан с помощью оператора **PERFORM** следующим образом:

PERFORM имя-процедуры
VARYING i **FROM** m_1 **BY** m_3 **UNTIL** $i > m_2$

имя-процедуры ...

Последовательность операторов

Например, параграфы с именами **S2** и **LOOP** из рассматриваемой программы могут быть переписаны с помощью оператора **PERFORM** следующим образом:

S2. **PERFORM LOOP VARYING H FROM 1 BY 1 UNTIL H > 5.**

GO TO LOOP-EXIT.

LOOP. MOVE 1 TO B (H, 1)

COMPUTE B (H, 2) = 5 + 3 * (H - A (H)).

LOOP-EXIT. EXIT.

Оператор **PERFORM** имеет несколько различных форм. Ниже приводятся некоторые из них.

1. **PERFORM** ип-1 [**THROUGH** ип-2] [число **TIMES**]
2. **PERFORM** ип-1 [**THROUGH** ип-2] **UNTIL** условие-1
3. **PERFORM** ип-1 [**THROUGH** ип-2]
VARYING i **FROM** m_1 **BY** m_3 **UNTIL** условие-1
[AFTER j **FROM** n_1 **BY** n_3 **UNTIL** условие-2
[AFTER k **FROM** p_1 **BY** p_3 **UNTIL** условие-3]]

Здесь *ип-1* и *ип-2* — имена процедур, а *число* — целочисленная переменная или целая константа. *Условие-1*, *условие-2* и *условие-3* — это произвольные условия, i , j и k — числовые переменные, а m_1 , m_3 , n_1 , n_3 , p_1 и p_3 — числовые переменные или числовые литералы.

При использовании формы 1 без обеих указанных опций выполняются все операторы параграфа или секции с именем *ип-1*, после чего управление передается оператору, следующему за оператором **PERFORM**.

В результате выполнения любой из форм оператора **PERFORM** управление в конечном итоге передается следующему за ним оператору. Если в любой из этих форм указана

При использовании формы 2 выполнение указанных параграфов от *ип-1* до *ип-2* включительно. Если в форме 1 указана опция *число TIMES*, то указанные параграфы или секции выполняются заданное *число* раз.

При использовании формы 2 выполнение указанных параграфов или секций повторяется до тех пор, пока не будет выполнено *условие-1*. Проверка условия осуществляется перед каждым выполнением параграфов или секций, так что последние могут вообще не выполняться.

Оператор **PERFORM** в форме 3 без указанных опций выполняется так, как описано на блок-схеме (б) рис. 14.6, где вместо проверки, указанной на рис. 14.7, следует осуществлять проверку, указанную на рис. 14.8. Если в форме 3 указаны одна или обе опции, то это соответствует двум или трем *вложенным* циклам. В этом случае переменная i будет изменяться реже, а переменная k — чаще остальных переменных. Например, при выполнении оператора

PERFORM LOOP VARYING H FROM 1 BY 1 UNTIL H > 5
AFTER I FROM 1 BY 1 UNTIL I > 2
AFTER J FROM 1 BY 1 UNTIL J > 3

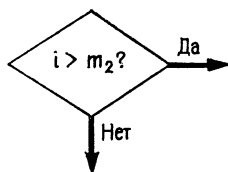


Рис. 14.7.

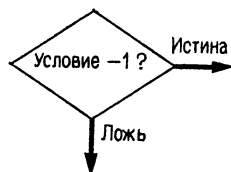


Рис. 14.8.

параграф с именем **LOOP** будет выполняться $5 \times 2 \times 3 = 30$ раз. При повторных выполнениях параграфа **LOOP** значения переменных **H**, **I** и **J** будут следующими:

Шаг	1	2	3	4	5	6	7	...	30
H	1	1	1	1	1	1	2	...	5
I	1	1	1	2	2	2	1	...	2
J	1	2	3	1	2	3	1	...	3

Непосредственно до 31-го выполнения параграфа **LOOP** значения переменных **J**, **I** и **H** будут увеличены (в указанном порядке) и превзойдут верхние границы. В результате этого управление будет передано оператору, следующему за оператором **PERFORM**.

Отметим, что если оператор, которому передается управление по окончании цикла, сам является началом параграфа, выполняемого оператором **PERFORM**, то этот параграф выполнится на один раз больше, чем указано в операторе **PERFORM**. По этой причине в предыдущий пример были включены оператор **GO TO LOOP-EXIT** и параграф **LOOP-EXIT**. Этот оператор **GO TO** можно исключить, немного изменив условие выхода из цикла:

```
S2.  PERFORM LOOP VARYING H FROM 1 BY 1 UNTIL H = 5.
      LOOP. MOVE ...
            COMPUTE ...
      LOOP-EXIT. EXIT.
```

Необходимость в использовании метки **LOOP-EXIT** по-прежнему остается, однако в данном случае она нужна для указания конца параграфа.

Оператор **MOVE**

При выполнении оператора **MOVE** одной или более переменным присваиваются новые значения. Он имеет следующую основную форму:

MOVE значение TO список-идентификаторов

Здесь *значение* — это идентификатор или литерал, а *список-идентификаторов* — список, состоящий из одного или более идентификаторов, отделенных друг от друга запятой и пробелом (, —). *Идентификатор-1* и *идентификатор-2* — это идентификаторы.

При выполнении оператора **MOVE** *значение* становится новым значением каждого идентификатора из *списка-идентификаторов*. Например, при выполнении оператора

MOVE 1 TO N

из рассматриваемой программы переменная **N** принимает значение 1.

С другой стороны, *значение* и *список-идентификаторов* могут представлять не элементарные, а групповые данные. В этом случае все данные должны иметь идентичную структуру, так что результат выполнения оператора **MOVE** аналогичен результату выполнения последовательности операторов **MOVE**, записанных для каждой пары соответствующих элементарных данных из двух групп.

В том случае, когда *значение* и *список-идентификаторов* представляют элементарные данные, их типы должны совпадать. Иными словами, они должны быть либо оба числовыми, либо оба нечисловыми (т. е. алфавитными, алфавитно-цифровыми или алфавитно-цифровыми редактируемыми). Например, в операторе **MOVE 1 TO N** обе величины 1 и **N** являются числовыми.

Кроме того, если статьи **PICTURE** для *значения* и некоторого идентификатора из *списка-идентификаторов* не совпадают, то до пересылки *значения* в переменную осуществляется его преобразование. Преобразование одного числового значения в другое осуществляется в соответствии со статьей **PICTURE** для идентификатора. При этом может осуществляться выравнивание полей относительно десятичной точки путем добавления ведущих нулей к *значению* или отбрасывания от него ведущих цифр, а также добавления нулей к дробной части *значения* или отбрасывания от нее десятичных цифр. При преобразовании одного нечислового значения в другое требуется только, чтобы пересылаемое значение было выровнено по длине переменной, в которой оно будет храниться. Это осуществляется либо путем отбрасывания крайних правых знаков *значения*, либо путем добавления к нему справа пробелов (—). В табл. 14.9 показаны результаты выполнения операций пересылки с помощью оператора **MOVE**.

Таблица 14.9

Пересылаемое значение	Статья для результата	Значение после выполнения оператора MOVE
3.19	9V9	3.1
	V99	.19
	99V999	03.190
'TEXT'	X(3)	'TEX'
	X(4)	'TEXT'
	X(5)	'TEXT—'

Следует отметить, что преобразование данных осуществляется только в тех случаях, когда и *значение*, и *идентификатор* представляют элементарные данные. Если это не так, то операция пересылки выполняется без учета типа значения и статьи **PICTURE** для результата. В этом случае оператор **MOVE** выполняется так же, как и при пересылке элементарного нечислового значения в элементарную нечисловую переменную с добавлением или отбрасыванием, если это требуется, крайних правых знаков. Например, при выполнении оператора

MOVE 'TEXT' TO C, D

где **C** и **D** определены как

```
01 C.
02 C1 PIC X.
02 C2 PIC X.
01 D.
02 D1 PIC X(3).
02 D2 PIC X(3).
```

элементам переменных **C** и **D** будут присвоены следующие значения:

```
C1 T D1 TEX
C2 E D2 T—
```

Поэтому мы рекомендуем строго придерживаться следующего правила. При пересылке данных из одной группы в другую с помощью оператора **MOVE** необходимо быть уверенным, что данные в этих двух группах имеют идентичную структуру и что соответствующие элементы имеют идентичные статьи **PICTURE**.

Операторы **STRING** и **UNSTRING**

Операторы Кобола **STRING** и **UNSTRING** предоставляют простейшие средства символьной обработки. Оператор **STRING**

позволяет сцеплять (соединять) несколько нечисловых значений (строк) в одно, а оператор **UNSTRING** — разбивать одну строку на несколько. Они имеют следующую форму:

```

STRING список-значений-1 [ DELIMITED BY { значение-1 } ]
      [ , список-значений-2 [ DELIMITED BY { значение-2 } ] ]
      .
      .
      .
INTO идентификатор [[WITH] POINTER указатель]
[ ; [ON] OVERFLOW оператор]
UNSTRING идентификатор [DELIMITED BY список-ограничителей]
INTO список-индификаторов
[ ; [ON] OVERFLOW оператор]

```

Здесь *список-значений-1* и *список-значений-2* — это списки элементарных нечисловых данных или литералов, отделенных друг от друга запятой и пробелом (,). *Значение-1* и *значение-2* — отдельные нечисловые элементарные данные или литералы. *Идентификатор* — идентификатор нечисловой переменной, *список-идентификаторов* — список идентификаторов, отделенных друг от друга запятой и пробелом (,), *указатель* — идентификатор числовой переменной. *Оператор* — это любой безусловный оператор. *Список-ограничителей* — это список, состоящий из одного или более идентификаторов нечисловых переменных и (или) литералов, отделенных друг от друга зарезервированным словом **OR**.

При выполнении оператора **STRING** осуществляется сцепление слева направо значений из списка-значений-1, списка-значений-2, ... в том порядке, в котором они записаны, и полученное значение присваивается *идентификатору*. Если полученная строка содержит больше знаков, чем указано в статье **PICTURE** для идентификатора, то возникает состояние **OVERFLOW** и выполняется *оператор* (если указана опция **ON OVERFLOW**). В противном случае (т. е. если состояние **OVERFLOW** не возникает или отсутствует опция **ON OVERFLOW**) управление передается оператору, следующему за оператором **STRING**. С другой стороны, если полученная строка содержит меньше знаков, чем указано в статье **PICTURE** для идентификатора, то оставшиеся (крайние правые) знаки в *идентификаторе* остаются неизменными. (Следует отметить, что в этом плане оператор **STRING** отличается от оператора **MOVE**.)

Если используется опция **DELIMITED BY SISE**, то каждое значение из соответствующего списка-значений берется цели-

ком. С другой стороны, если используется опция **DELIMITED BY значение-1**, первое (начиная слева) появление *значения-1* внутри каждого из значений из соответствующего списка значений определяет конец этого значения. (Если *значение-1* не содержится внутри одного или более значений, то последние используются целиком, так же как и в варианте **SIZE**).

При использовании опции **WITH POINTER** *указатель* текущее целое значение указателя определяет, в какой позиции значения *идентификатора* будет храниться крайний слева знак *списка-значений-1*.

Предположим, например, что имеются идентификаторы и их значения, указанные в табл. 14.10. При выполнении оператора

Таблица 14.10

Идентификатор	PICTURE	Текущее значение
STREET	X(17)	"1800-BULL-RUN,___"
CITY	X(12)	"ALEXANDRIA,-"
STATE	X(6)	"VA.____"
ZIP	9(5)	22200
ADDR	X(40)	пробелы

**STRING STREET, CITY, STATE, ZIP DELIMITED BY SIZE
INTO ADDR**

переменной **ADDR** будет присвоено значение

"1800-BULL-RUN,___ALEXANDRIA,-VA.____22200"

Однако если требуется исключить дополнительные пробелы, запятые и точки, расположенные в конце значений идентификаторов, то этот оператор надо переписать следующим образом:

**STRING STREET DELIMITED BY " , ",
"_" DELIMITED BY SIZE,
CITY DELIMITED BY " , ",
"_" DELIMITED BY SIZE,
STATE DELIMITED BY " . ",
"_" DELIMITED BY SIZE,
ZIP DELIMITED BY SIZE,
INTO ADDR**

В этом случае переменной **ADDR** будет присвоено значение

"1800—BULL—RUN—ALEXANDRIA—VA—22200—"

Отметим, что пробелы, расположенные в конце этой строки, присутствовали в ней еще до выполнения оператора **STRING**; они не были добавлены в процессе его выполнения.

При выполнении оператора **UNSTRING** значение *идентификатора* расщепляется и засылается в идентификаторы из *списка-идентификаторов* в том порядке, в котором они записаны. При использовании опции **DELIMITED BY** *список-ограничителей* встречаются внутри расщепленного значения, пересылка последнего завершается преждевременно. Обратимся вновь к рассмотрению предыдущего примера и предположим, что значения переменных **STREET**, **CITY**, **STATE** и **ZIP** целиком состоят из пробелов, а переменная **ADDR** имеет следующее значение:

"1800—BULL—RUN,—ALEXANDRIA,—VA.—22200—"

В результате выполнения оператора

**UNSTRING ADDR DELIMITED BY “,—” OR “.—”
INTO STREET, CITY, STATE, ZIP**

переменным **STREET**, **CITY**, **STATE** и **ZIP** вновь будут присвоены их начальные значения, за исключением того, что запятые и точка не передадутся. Ограничителями здесь являются символы “,—” и “.—”. Строка **ADDR** рассматривается слева направо, и ее знаки последовательно пересылаются в строку **STREET** до тех пор, пока последняя не будет заполнена или пока не появится один из символов “,—” или “.—”. Сами символы “,—” и “.—” не пересылаются, так как они являются ограничителями. Затем заполняется строка **CITY**, в которую пересылаются знаки строки **ADDR**, начиная со следующего после крайнего левого ограничителя “,—”. Аналогично заполняются строки **STATE** и **ZIP**.

Оператор **INSPECT**

Возможности символьной обработки в Коболе существенно расширяются с помощью оператора **INSPECT**. Он позволяет исследовать символьную строку (т. е. значение типа **DISPLAY**) на предмет вхождения в нее другой символьной строки и, если необходимо, заменять последнюю третьей строкой. Оператор **INSPECT** имеет следующую форму:

INSPECT идентификатор-1

$$\left[\text{TALLYING ид-2 FOR } \left\{ \begin{array}{l} \text{ALL значение-1} \\ \text{LEADING значение-1} \\ \text{CHARACTERS} \end{array} \right\} \right. \\ \left. \left[\left\{ \begin{array}{l} \text{BEFORE} \\ \text{AFTER} \end{array} \right\} \text{INITIAL значение-2} \right] \right. \\ \left. \text{CHARACTERS} \right. \\ \left. \left[\text{REPLACING } \left\{ \begin{array}{l} \text{ALL значение-3} \\ \text{LEADING значение-3} \\ \text{FIRST значение-3} \end{array} \right\} \text{BY значение-4} \right. \right. \\ \left. \left. \left[\left\{ \begin{array}{l} \text{BEFORE} \\ \text{AFTER} \end{array} \right\} \text{INITIAL значение-5} \right] \right] \right]$$

Здесь *идентификатор-1* — любое значение типа **DISPLAY**. *Ид-2* — любое элементарное числовое значение, используемое для подсчета числа вхождений указанной строки *значение-1* в *идентификатор-1*. *Значение-1*, *значение-2*, *значение-3*, *значение-4* и *значение-5* — нечисловые элементарные идентификаторы или литералы.

Опция **BEFORE** (или **AFTER**) определяет конечную (или соответственно начальную) точку поиска как позицию в строке *идентификатор-1*, расположенную непосредственно перед крайним слева символом *значение-2* или *значение-5* (после него). Когда эта опция опущена, конечная и начальная точки определяются соответственно как правый и левый концы строки *идентификатор-1*.

Таким образом, оператор **INSPECT** выполняется следующим образом. Если используется опция **TALLYING**, то переменной *ид-2* будет присвоено значение, равное общему числу насчитанных знаков. Подсчет выполняется в зависимости от параметров, указанных после слова **FOR**. **ALL значение-1** указывает, что должно быть подсчитано общее число вхождений *значения-1* в отрезок строки от начальной до конечной точки. **LEADING значение-1** указывает, что должно быть подсчитано количество символов *значение-1*, расположенных до первого отличного от *значения-1* символа. **CHARACTERS** указывает, что должно быть подсчитано общее число знаков в отрезке строки от начальной до конечной точки. Во всех этих случаях в результате выполнения оператора **INSPECT** переменной *ид-2* начальное значение не присваивается, и поэтому это должно быть сделано где-то в другом месте программы.

Если используется опция **REPLACING**, то начальная и конечная точки для замены знаков в строке *идентификатор-1* определяются аналогично в зависимости от того, указано или нет *значение-5*. Здесь *значение-3* определяет знаки, которые следует заменить (параметр **CHARACTERS** указывает, что

должны быть заменены все знаки), а *значение-4* указывает значение, которым заменяются эти знаки.

В операторе **INSPECT** должна быть указана либо опция **TALLYING**, либо опция **REPLACING**. Рассмотрим примеры (табл. 14.11), в которых в качестве *идентификатора-1* используется 17-знаковая переменная **STREET**, имеющая в каждом случае значение

“__1800_BULL_RUN,‑”

Таблица 14.11

Пример	Результирующее значение переменной STREET	Результирующее значение переменной I
1 INSPECT STREET TALLYING I FOR ALL “‑”	Не меняется	5
2 INSPECT STREET TALLYING I FOR ALL “‑” AFTER INITIAL “‑”	Не меняется	4
3 INSPECT STREET REPLACING LEADING “‑” BY “*,”	“**1800_BULL_RUN,‑”	0 (не меняется)
4 INSPECT STREET REPLACING ALL “‑” BY “*,” AFTER INITIAL “,”	“__1800_BULL_RUN,*”	0 (не меняется)

Предполагается также, что для подсчета числа знаков используется переменная **I**, описанная как **PICTURE 99**, текущее значение которой в каждом случае равно 0.

Операторы SEARCH и SET

В то время как оператор **INSPECT** позволяет исследовать значения символьных строк, оператор **SEARCH** позволяет исследовать значения элементов таблиц. Оператор **SET** используется для присвоения значений переменной типа **INDEX**, которая связана с таблицей, исследуемой с помощью оператора **SEARCH**. Эти операторы имеют следующие формы:

1. **SEARCH** идентификатор-1 [**VARYING** индекс]
[**AT END** оператор-1]
 WHEN условие оператор-2
2. **SET** список-индексов $\left\{ \begin{array}{l} \text{TO} \\ \text{UP BY} \\ \text{DOWN BY} \end{array} \right\}$ значение

A

43
19
-5
0
-19

Рис. 14.9.

Здесь *идентификатор-1* определяет исследуемую таблицу. Он должен быть именем групповой переменной, при описании которой используется статья **OCCURS** с опцией **INDEXED BY**. *Индекс* — это индексная переменная, которая определяется в части **INDEXED BY** *переменная* статьи **OCCURS** описания таблицы. *Список-индексов* — это список таких переменных, отделенных друг от друга запятой и пробелом (,—). *Оператор-1* и *оператор-2* — предписываемые операторы. *Значение* — индексная переменная или целая константа.

Оператор **SEARCH** выполняется следующим образом. До тех пор, пока не будет выполнено указанное условие или не будет достигнут конец таблицы, исследуются ее элементы и изменяется собственный индекс таблицы (и одновременно *индекс*, если указана статья **VARYING**). Соответственно выполняется либо *оператор-2*, либо *оператор-1*. Если *оператор-1* не указан и при исследовании таблицы ее конец достигается раньше, чем выполняется *условие*, то управление передается следующему предложению.

При выполнении оператора **SET** значение каждой переменной из *списка-индексов* либо устанавливается равным (**TO**) целому *значению*, либо увеличивается (**UP BY**), либо уменьшается (**DOWN BY**) на эту величину.

Рассмотрим пример, в котором в таблице с именем **TABLE-A** ищется первый нулевой элемент. Таблица **TABLE-A** и индекс **I** описаны следующим образом:

01 **TABLE-A.**

02 **A PIC S99 OCCURS 5 INDEXED BY I.**

Отметим, что определение переменной **I** как индекса при описании таблицы **TABLE-A** является неявным описанием **I**. Предположим теперь, что элементы таблицы **A** имеют значения, заданные на рис. 14.9. Тогда для исследования таблицы могут быть использованы следующие операторы:

SET I TO 1.

SEARCH TABLE-A

AT END GO TO ZERO-NOT-FOUND.

WHEN A (I)=0 GO TO ZERO-FOUND.

В результате выполнения этих операторов управление будет передано **ZERO-FOUND**, а переменной **I** будет присвоено значение 4.

14.4. СОГЛАШЕНИЯ О ВВОДЕ-ВЫВОДЕ

Кобол предоставляет пользователю широкие возможности для чтения и записи данных в ходе выполнения программы. **Файлом** в Коболе называется набор записей, каждая из которых имеет такой же формат, как и следующая. Существуют файлы с последовательным доступом, предназначенные для ввода или вывода, а также файлы с прямым доступом, предназначенные либо для ввода, либо для вывода, либо для ввода-вывода (т. е. записи могут как считываться, так и записываться).

При последовательной организации файла записи обрабатываются в том порядке, в котором хранятся, по одной, начиная с первой. В файлах с прямым доступом записи могут обрабатываться в произвольном порядке, указанном программистом. Поэтому за один прогон программы любая запись файла с прямым доступом может быть прочитана (записана) произвольное число раз.

Для каждого файла, к которому осуществляется обращение в Кобол-программе, необходимо следующее:

1. В Разделе оборудования указать класс физических устройств (например, устройство считывания с перфокарт, магнитная лента), где размещается этот файл.

2. В Секции файлов Раздела данных определить атрибуты файла и выделить область (с помощью статьи описания записи), в которую (из которой) записи будут считываться (записываться) из файла (в файл).

3. В Раздел процедур поместить операторы, которые будут считывать (писать) записи из файла (в файл), а также операторы, которые будут “открывать” и “закрывать” файл.

Назначение устройств в Разделе оборудования

В тех случаях, когда в программе используется один или более файлов, раздел оборудования должен содержать **секцию ввода-вывода** и **параграф управления файлами**. Параграф управления файлами содержит для каждого файла **статью управления файлом**, которая назначает ему конкретное (в зависимости от реализации) физическое устройство ввода-вывода. Секция ввода-вывода озаглавливается следующим образом:

INPUT-OUTPUT SECTION.

FILE-CONTROL. статья-управления-файлом ...

Каждая *статья-управления-файлом* имеет следующую форму:

```
SELECT имя-файла ASSIGN TO имя-устройства
[RESERVE целое AREAS]
[ ORGANIZATION IS { SEQUENTIAL } ]
[ ACCESS MODE IS { SEQUENTIAL } ]
[RECORD KEY IS имя-данных].
```

Прежде всего следует обратить внимание на то, что эта статья всегда заканчивается точкой (.). Здесь *имя-файла* — любое незарезервированное определенное пользователем слово, содержащее по крайней мере один алфавитный знак (A — Z). *Имя-устройства* зависит от реализации. В табл. 14.12 даются допустимые имена-устройств для Кобола ANS OS IBM. Здесь файл, имеющий индексированную организацию, допускает либо последовательный, либо прямой доступ, но должен храниться на устройстве прямого доступа. Кроме того, *ddимя* — это имя, содержащее от одного до восьми знаков, указанное при определении данных для этого файла в карте управления заданиями (*ddкарта*).

Таблица 14.12

Имя-устройств	Смысл
DA-S-ddимя	Некоторое устройство прямого доступа (DA) и последовательный (S) файл
DA-I-ddимя	То же, за исключением того что файл индексированный (I)
UT-S-ddимя	Некоторое устройство с магнитной лентой (UT) и последовательный файл
UR-S-ddимя	Некоторое устройство для работы с единичными записями (UR) (устройство считывания с перфокарт, перфоратор или линейное печатающее устройство) и последовательный файл

Статья **RESERVE** является необязательной и используется для указания числа буферов ввода-вывода (**AREAS**), предназначенных для данного файла. Если она опущена, то число буферов зависит от реализации.

Статьи **ORGANIZATION** (организация) и **ACCESS** (доступ) также не обязательны. Если какая-либо из них опущена, то предполагается, что в данном случае указан параметр **SEQUENTIAL** (последовательный). Если файл имеет индексированную организацию (**INDEXED**), то доступ (**ACCESS**

MODE) может быть либо последовательным (**SEQUENTIAL**), либо прямым (**RANDOM**).

При использовании индексированного (**INDEXED**) файла и прямого (**RANDOM**) доступа (**ACCESS MODE**) поиск отдельной записи осуществляется по ключу. Ключ записи — это значение, однозначно определяющее данную запись в файле. Например, для основного файла учета служащих ключом может быть номер страхового полиса, так что никакие два служащих не будут иметь один и тот же ключ. Таким образом, при прямом доступе к индексированному (**INDEXED**) файлу статья **RECORD KEY** определяет *имя-данных* (или поле) внутри статьи описания записи файла, которое используется в качестве ключа для файла. Эта статья, конечно, не используется при последовательном (**SEQUENTIAL**) доступе.

Предположим, например, что имеются входной файл **CARDS** на перфокартах, выходной файл печати **PAPER** и выходной файл **MAG-TAPE** на магнитной ленте. Тогда в Разделе оборудования программы, использующей эти три файла, должна содержаться следующая секция ввода-вывода:

INPUT-OUTPUT SECTION.

FILE-CONTROL.

SELECT-CARDS	ASSIGN TO UR-S-SYSIN.
SELECT PAPER	ASSIGN TO UR-S-SYSPRINT.
SELECT MAG-TAPE	ASSIGN TO UT-S-TAPEDD.

Описание файлов и записей в Разделе данных

Каждый файл, используемый в программе, должен быть описан в секции файлов Раздела данных. Сразу же после описания файла должно следовать одно или более описаний записей. Общий вид Раздела данных следующий:

DATA DIVISION.

FILE SECTION.

{описание-файла
описание-записи ...}

WORKING-STORAGE SECTION.

{описание переменных, таблиц и записей}

Описание-файла имеет следующую форму:

FD имя-файла

[BLOCK [CONTAINS] целое-1 [CHARACTERS]]

[RECORD [CONTAINS] целое-2 [CHARACTERS]]

LABEL RECORDS ARE { **STANDARD** }
 { **OMITTED** }

[LINAGE IS значение LINES].

Прежде всего следует отметить, что описание каждого файла должно заканчиваться точкой (.). *Имя-файла* должно совпадать с именем файла, данным в соответствующей статье **SELECT** Раздела оборудования. Статья **BLOCK CONTAINS** используется в том случае, когда каждая физическая запись в файле содержит более чем одну логическую запись (т. е. записи являются **сблокированными**). В этом случае *целое-1* означает общее число знаков в физической записи. Если записи в файле являются *несблокированными* (как в случае файлов с единичными записями), то статья **BLOCK CONTAINS** не требуется.

Статья **RECORD CONTAINS** определяет число знаков в логической записи, обозначенное как *целое-2*. Эта статья может быть опущена в тех случаях, когда длина логической записи фиксирована для данного класса устройств, который зависит от реализации.

Статья **LABEL RECORDS** обязательна во всех случаях. Она определяет, имеет (**STANDARD**) файл стандартные метки или нет (**OMITTED**). Формат меток файла зависит от реализации.

Статья **LINAGE** может быть использована только для файлов печати. Она определяет число строк (которое равно *значению*) в логической странице. Само *значение* представляется целочисленной переменной или константой. Если статья **LINAGE** присутствует, то в программе можно воспользоваться системным счетчиком числа строк в странице (**LINAGE-COUNTER**). Он содержит целое значение, равное номеру следующей строки, которая должна быть записана в текущей странице. Это значение вычисляется системой, и к нему можно обращаться в программе.

За *описанием-файла* следуют одно или более *описаний-записи*, которые определяют структуру и разбивку логической записи в файле. Иногда файл содержит записи двух, трех или более различных структур. В этом случае требуется два, три или более различных описаний-записи. Кроме того, число знаковых позиций, определенных в объединенных статьях **PICTURE** элементарных данных описания записи, должно совпадать с размером логической записи файла.

Обратимся вновь к рассмотрению трех файлов **CARDS**, **PAPER** и **MAG-TAPE**, описанных выше. Предположим также, что входной файл **CARDS** располагается на стандартных 80-знаковых перфокартах, файл **PAPER** является выходной формой, содержащей 132 знака в строке, а файл **MAG-TAPE** является последовательностью 80-знаковых логических записей, объединенных в 800-знаковые блоки (физические записи). Кроме того, ленточный файл имеет стандартные метки (другие файлы их

Позиция	1-10	11-15	16-30	31-32	33	34-35	36-38	39-47	48-80
	Имя сотрудника			Возраст	Рост		Вес	Номер страхового полиса	...
	Фамилия	Имя	Отчество		Футы	Дюймы			

Рис. 14.10.

не содержат), а разбивка записи показана на рис. 14.10. Тогда для этих файлов в Разделе данных могут быть записаны следующие описания файлов и записей:

```

FD CARDS
RECORD CONTAINS 80 CHARACTERS
LABEL RECORDS ARE OMITTED.
01 CARDS-RECORD.
02 CARD PIC X(80).

FD PAPER
RECORD CONTAINS 133 CHARACTERS
LABEL RECORDS ARE OMITTED
LINAGE IS 60 LINES.
01 PAPER-RECORD.
02 FILLER PIC X.
02 DATA-LINE PIC X(132).

FD MAG-TAPE
BLOCK CONTAINS 800 CHARACTERS
RECORD CONTAINS 80 CHARACTERS
LABEL RECORDS ARE STANDARD.
01 MAG-TAPE-RECORD.
02 NAME.
03 NFIRST PIC X(10).
03 MIDDLE PIC X(5).
03 NLAST PIC X(15).
02 AGE PIC 99.
02 HEIGHT.
03 FEET PIC 9.
03 INCHES PIC 99.
02 WEIGHT PIC 999.
02 SS-NO PIC 9(9).
02 FILLER PIC X(33).

```

Следует отметить, что зарезервированное слово **FILLER** используется для обозначения любой части записи, обращения к которой отсутствуют в Разделе процедур. Например, неиспользуемая часть записи **MAG-TAPE-RECORD** располагается в позициях 48—80

Отметим также, что запись в файле **PAPER** содержит 133 (а не 132) знака, поскольку начальная позиция зарезервирована. Эта дополнительная позиция используется системой для управления пропуском строк в соответствии с тем, как указано в операторе **WRITE** для этого файла.

Доступ к файлам в Разделе процедур

При выполнении Кобол-программы доступ к файлу осуществляется при каждом выполнении операторов **OPEN**, **CLOSE**, **READ**, **WRITE**, **REWRITE** и **DELETE**.

Оператор **OPEN** иницирует доступ к файлу. Пока файл не открыт, никакие записи не могут быть считаны (записаны) из (в) него. Оператор **OPEN** имеет следующую форму:

$$\text{OPEN } \left\{ \begin{array}{l} \text{INPUT} \\ \text{OUTPUT} \\ \text{I-O} \end{array} \right\} \text{ список-имен-файлов}$$

Здесь *список-имен-файлов* — это список имен файлов, которые должны быть открыты. Имена отделяются друг от друга запятой и пробелом (,—). Выбор параметра **INPUT**, **OUTPUT** или **I-O** зависит от характера работы с файлом. Когда файл открывается для ввода (**INPUT**), записи могут только считываться из файла (последовательно или произвольно) с помощью оператора **READ**. Когда файл открывается для вывода (**OUTPUT**), записи могут записываться в файл (последовательно или произвольно) с помощью оператора **WRITE**. Когда файл открывается для ввода-вывода (**I-O**), записи могут считываться из файла (последовательно или произвольно) с помощью оператора **READ**, произвольно записываться (переписываться) в файл с помощью оператора **WRITE** (**REWRITE**), последовательно переписываться в файл с помощью оператора **REWRITE** или удаляться из файла (последовательно или произвольно) с помощью оператора **DELETE**. Однако оператор **DELETE** может быть использован только для индексированных файлов.

Оператор **CLOSE** завершает доступ к файлу. Он должен появляться только после того, как вся остальная обработка (операторы **READ**, **WRITE** и т. д.) файла закончится. Его форма следующая:

CLOSE список-имен-файлов

В результате выполнения оператора **CLOSE** закрывается доступ из программы к перечисленным файлам. Однако к любому из этих файлов впоследствии вновь может быть осуществлен доступ после выполнения другого оператора **OPEN** для этого файла.

Оператор **READ** используется для передачи одной логической записи из файла в программу. Он имеет следующую форму:

READ имя-файла [**RECORD**]
[**AT END** оператор-1]
[**INVALID KEY** оператор-2]

Здесь *имя-файла* указывает файл, из которого запись должна быть считана. Сама запись хранится в области, определенной в статье (или статьях) описания записи для этого файла. Никаких преобразований не осуществляется. *Оператор-1* — предписываемый оператор, который должен быть выполнен при достижении конца файла, т. е. при попытке считать запись из файла с последовательным доступом, когда последняя запись уже была считана. *Оператор-2* — предписываемый оператор, который должен быть выполнен при неудачной попытке считывания записи из файла с прямым доступом, т. е. при отсутствии записи в файле, ключ которой соответствовал бы текущему значению переменной, чье имя-данных указано в статье **RECORD-KEY** оператора **SELECT** (для этого файла), расположенного в секции ввода-вывода Раздела оборудования. Присвоение значения этой переменной, конечно, должно осуществляться в программе.

Оператор **WRITE** используется для передачи одной записи из программы в файл. Он имеет следующую форму:

WRITE имя-записи
[**AFTER** [**ADVANCING**] { значение LINES }]
[**AT END-OF-PAGE** оператор-1]
[**INVALID KEY** оператор-2]

Имя-записи — это групповое имя некоторой статьи описания записи для этого файла (отметим, что *имя-файла* здесь не используется). Статья **AFTER** используется только для файлов печати, а *значение* в этом случае — это целочисленная переменная или константа, указывающая количество пропускаемых строк до печати указанной строки. Параметр **PAGE** указывает переход на начало новой страницы. Статья **AT END-OF-PAGE** также используется только для файлов печати, а *оператор-1* — предписываемый оператор, который должен выполняться каждый раз, когда при записи происходит выход за последнюю строку текущей страницы.

Статья **INVALID KEY** используется только для индексированных файлов. Если осуществляется последовательный доступ к файлу, то при попытке записать запись, ключ которой не превосходит значения ключа последней записанной записи, возни-

кает состояние **INVALID KEY**. Если осуществляется прямой доступ к файлу, то состояние **INVALID KEY** возникает при попытке записать запись, ключ которой такой же, как у некоторой другой записи файла. В каждом случае *оператор-2* — это предписываемый оператор, который должен выполняться каждый раз, когда при выполнении оператора **WRITE** возникает состояние **INVALID KEY**.

В результате выполнения оператора **REWRITE** осуществляется замена записи в файле, расположенном на устройстве прямого доступа (DA). Файл может быть либо последовательным, либо индексированным. Оператор **REWRITE** имеет следующую форму:

REWRITE имя-записи [**FROM** идентификатор]
[**INVALID KEY** оператор-2]

Оператор **DELETE** имеет следующую форму:

DELETE имя-файла [**RECORD**]
[**INVALID KEY** оператор-2]

При использовании оператора **DELETE** файл должен быть открыт для ввода-вывода, а доступ к файлу может быть либо последовательным, либо прямым. Если доступ последовательный, то стирается последняя считанная запись. Если доступ прямой, то из файла стирается запись, ключ которой совпадает с текущим значением переменной, содержащей ключ файла. Условия возникновения состояния **INVALID KEY** для оператора **DELETE** те же, что и для оператора **REWRITE**.

Для рассмотрения примера использования некоторых из этих операторов вновь обратимся к файлам **CARDS**, **PAPER** и **MAG-TAPE**, которые были определены выше. В результате выполнения следующих операторов из Раздела процедур файл **CARDS** будет считан, отпечатан (через два интервала) и скопирован на магнитную ленту.

```

OPEN INPUT CARDS, OUTPUT PAPER, OUTPUT
MAG-TAPE.
L1.    MOVE SPACES TO DATA-LINE.
      WRITE PAPER-RECORD AFTER PAGE.
LOOP.  READ CARDS AT END GO TO ALL-DONE.
      MOVE CARD TO DATA-LINE, MAG-TAPE-RECORD.
      WRITE MAG-TAPE-RECORD.
      WRITE PAPER-RECORD AFTER ADVANCING 2 LINES
        AT END-OF-PAGE GO TO L1.
      GO TO LOOP.
ALL-DONE. CLOSE CARDS, PAPER, MAG-TAPE.
          STOP RUN.
```

14.5. ПОДПРОГРАММЫ

На практике в большинстве случаев программа достигает таких размеров, что становится оправданным ее разбиение на несколько функциональных блоков. Блоки объединяются с помощью основной программы, которая управляет последовательностью их выполнения. Преимуществом подпрограмм является то, что они могут использоваться многократно и при этом не требуется каждый раз их переписывать и отлаживать заново.

Кобол располагает средствами, позволяющими работать с подпрограммами. Подпрограмма в Коболе — это завершенная программа, которая может вызываться при выполнении другой Кобол-программы или подпрограммы. В действительности подпрограмма может выполняться только при ее вызове из другой программы. В этом разделе мы остановимся на вопросах написания и вызова подпрограмм в Коболе и приведем соответствующие примеры.

Будет рассмотрена задача вычисления факториала (f) целого числа n , который определяется следующим образом:

Если $n < 2$, то факториал $f = 1$.

В противном случае факториал $f = 2 \times 3 \times \dots \times n$.

Например, факториал 5 равен $2 \times 3 \times 4 \times 5 = 120$.

Написание подпрограмм

Сама подпрограмма фактически является определением задачи (например, задачи вычисления факториала), которая должна быть выполнена. Подпрограммы в Коболе должны удовлетворять следующим условиям:

1. Раздел данных должен содержать секцию связей, в которой определены тип и структура всех переменных, *передаваемых* подпрограмме из вызывающей программы.

2. Заголовок Раздела процедур должен быть расширенным, с тем чтобы определить эти переменные и порядок, в котором они должны быть перечислены при обращении к подпрограмме из вызывающей программы.

3. Раздел процедур должен содержать оператор **EXIT** в тех местах, где управление должно возвращаться вызывающей программе.

Секция связей располагается в Разделе данных сразу же после секции рабочей памяти. Она содержит описание каждого параметра, который будет передан подпрограмме, в соответствии с соглашениями, существующими для определения обычных переменных и статей описания записи в секции рабочей памяти. Она начинается с заголовка **LINKAGE SECTION**.

Например, для подпрограммы вычисления факториала требуются два параметра, n и f . Будем предполагать, что в вызы-

вающей программе оба они описаны как **PICTURE S9(9)** и **USAGE COMPUTATIONAL**. Таким образом, секция связей может быть написана следующим образом:

LINKAGE SECTION.

77 N PIC S9(9) COMP.

77 F PIC S9(9) COMP.

Заголовок Раздела процедур для подпрограмм имеет следующую общую форму:

PROCEDURE DIVISION USING список-параметров.

Здесь *список-параметров* — это список параметров, определенных в секции связей. Параметры отделяются друг от друга запятой и пробелом (,—). В рассматриваемом примере заголовок Раздела процедур должен быть записан следующим образом:

PROCEDURE DIVISION USING N, F.

Сам Раздел процедур записывается так же, как и в обычной программе, выполняющей определенную задачу. Однако после завершения решения этой задачи необходимо возвратить управление вызывающей программе с помощью следующего оператора:

имя-п. **EXIT PROGRAM.**

Как указывалось, оператор **EXIT PROGRAM** должен отдельно содержаться в некотором параграфе (с именем *имя-п*).

Теперь можно написать всю программу вычисления факториала, которая будет называться **FACT**.

IDENTIFICATION DIVISION.

PROGRAM-ID. FACT.

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER. IBM-370-145.

OBJECT-COMPUTER. IBM-370-145.

DATA DIVISION.

WORKING-STORAGE SECTION.

77 I PIC S9(9) COMP.

LINKAGE SECTION.

77 N PIC S9(9) COMP.

77 F PIC S9(9) COMP.

PROCEDURE DIVISION USING N, F.

MOVE 1 TO F.

PERFORM S1 VARYING I FROM 1 BY 1 UNTIL I = N OR I > N.

S1. COMPUTE F = F * I.

S2. EXIT PROGRAM.

Следует отметить, что **N** и **F** — это не переменные, как, например, **I**, а параметры, *резервирующие место* для переменных, которым будут присвоены значения вызывающей программой в тот момент, когда подпрограмма **ФАКТ** начнет выполняться.

Вызов подпрограмм

Подпрограмма вызывается из другой подпрограммы или обычной программы при помощи оператора **CALL**. В операторе **CALL** указываются также переменные, значения которых будут присвоены параметрам подпрограммы при выполнении Раздела процедур подпрограммы. Оператор **CALL** имеет следующую форму:

CALL "имя" **USING** список-переменных

Здесь *имя* — это имя подпрограммы, определенное в параграфе **PROGRAM-ID** Раздела идентификаций. *Список-переменных* — список тех переменных из Раздела данных вызывающей программы, которые должны быть переданы подпрограмме при данном вызове.

Между переменными из *списка-переменных* и параметрами из *списка-параметров* подпрограммы существует взаимно однозначное соответствие (слева направо). Предположим, например, что требуется написать программу, которая для любого положительного целого двузначного числа n с помощью подпрограммы **ФАКТ** вычисляет и печатает биномиальные коэффициенты

$$a_i = \frac{n!}{i!(n-i)!} \text{ для всех } i = 0, 1, \dots, n$$

Эта программа может быть написана следующим образом:

IDENTIFICATION DIVISION.

PROGRAM-ID. BINOMIAL.

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER. IBM-370-145.

OBJECT-COMPUTER. IBM-370-145.

INPUT-OUTPUT SECTION.

SELECT CARDS ASSIGN TO UR-S-SYSIN.

SELECT PAPER ASSIGN TO UR-S-SYSPRINT.

DATA DIVISION.

FILE SECTION.

FD CARDS RECORD CONTAINS 80, LABEL RECORDS ARE OMITTED.

01 CARD.

02 N PIC 99.

02 FILLER PIC X(78).

FD PAPER RECORD CONTAINS 133 LABEL RECORDS ARE OMITTED.

01 PRINT-LINE.

02 FILLER PIC X.

02 I PIC 99.

02 FILLER PIC X(2).

02 AI PIC 9(9).

02 FILLER PIC X(119).

WORKING-STORAGE SECTION.

77 NC PIC 9(9) COMP.

77 IC PIC 9(9) COMP.

77 NMIC PIC 9(9) COMP.

77 NFACT PIC 9(9) COMP.

77 IFACT PIC 9(9) COMP.

77 NMIFACT PIC 9(9) COMP.

PROCEDURE DIVISION.

OPEN INPUT CARDS, OUTPUT PAPER.

MOVE SPACES TO PRINT-LINE.

READ CARDS AT END GO TO P2.

MOVE N TO NC.

CALL FACT USING NC, NFACT.

PERFORM P1 VARYING IC FROM 0 BY 1 UNTIL IC = NC.

P1. CALL FACT USING IC, IFACT.

SUBTRACT IC FROM NC GIVING NMIC.

CALL FACT USING NMIC, NMIFACT.

COMPUTE AI = NFACT / (IFACT * NMIFACT).

MOVE IC TO I.

WRITE PRINT-LINE AFTER ADVANCING 1.

P2. CLOSE CARDS, PAPER.

STOP RUN.

Рассмотрим первый оператор **CALL** (который подчеркнут) в этой программе. Между переменными, указанными здесь, и параметрами подпрограммы **FACT** существует следующее соответствие:

переменная	NC	NFACT
↑	↑	↑
↓	↓	↓
параметр	N	F

В действительности при выполнении вызываемой программы каждый параметр в Разделе процедур заменяется соответствующей переменной, указанной в операторе **CALL**. Таким образом, при выполнении этого оператора **CALL** вычисляется факториал текущего значения переменной с именем **NC** и полученный результат присваивается переменной с именем **NFACT**.

Если значение **N** равно 05, то будут напечатаны следующие значения:

0	1
1	5
2	10
3	10
4	5
5	1

Следует отметить, что в этом случае значение **NFACT** будет равно 120, **NC** будет изменяться от 0 до 5, а параграф **P1** будет выполняться шесть раз.

14.6. ЗАКОНЧЕННЫЕ ПРОГРАММЫ

В предыдущих разделах читатель, вероятно, обратил внимание на использование довольно строгих правил записи операторов. В этом разделе будет дано более точное описание требований, которым должны удовлетворять Кобол-программы.

Раздел идентификаций в Кобол-программе служит главным образом в качестве документации. Хотя обязательными в нем являются только заголовки раздела и параграф **PROGRAM-ID**, для более полного документирования программы могут быть написаны дополнительные параграфы. Общую структуру Раздела идентификаций можно описать следующим образом:

IDENTIFICATION DIVISION.

PROGRAM-ID. имя-программы.

[**AUTHOR.** комментарий.]

[**INSTALLATION.** комментарий.]

[**DATE-WRITTEN.** комментарий.]

[**DATE-COMPILED.** комментарий.]

[**SECURITY.** комментарий.]

Здесь *имя-программы* — это имя программы, которое может быть любым незарезервированным словом (могут накладываться и другие ограничения в зависимости от реализации). *Комментарий* включает в себя некоторые сведения об авторе, установке и т. п.

Порядок расположения четырех разделов, а также порядок расположения секций внутри Раздела оборудования и Раздела данных нельзя изменять. Кроме того, все данные 77-го уровня в секции рабочей памяти и секции связей должны предшествовать всем статьям описания записи в этой секции. Однако порядок расположения отдельных данных 77-го уровня относительно друг друга не существует. То же относится и к отдельным статьям описания записи.

На разбивку отдельной строки внутри Кобол-программы также накладываются некоторые ограничения. Поле А строки располагается в позициях 8—11, а поле В — с 12 позиции по конец строки. (Длина строки зависит от реализации.)

Позиции 1—6 строки обычно оставляют пустыми. Позиция 7 используется в следующих двух случаях. Если в ней содержится звездочка (*), то оставшаяся часть строки принимается за комментарий. Если в ней содержится черта (—), то оставшаяся часть строки является продолжением литеральной константы, которая не умещается в предыдущую строку. В остальных случаях позиция 7 обычно пустая.

Поле А строки отводится под заголовки разделов и секций, названия параграфов, данные уровня 01 в статьях описания записи, данные уровня 77 и статьи описания файлов (FD). Все они должны начинаться с какой-нибудь позиции поля А, хотя и не обязательно с позиции 8. Все другие элементы программы (слова, операторы, предложения и данные уровней, отличных от 01, в статьях описания записи) должны полностью располагаться в поле В, но не обязательно начиная в точности с позиции 12. Кроме того, любой элемент может быть продолжен с одной строки программы на другую при условии, что его продолжение располагается в поле В.

Предложение может занимать одну или несколько строк, и строка может содержать одно или несколько предложений. Если предложение переносится на следующую строку, то оно может обрываться между двумя соседними словами, где имеется пробел. Кроме того, везде, где имеется один пробел, могут быть два или более пробелов. Наконец, запятая и пробел (,—), используемые для записи различных списков в операторах некоторых типов, могут быть заменены одним или более пробелами.

Не выходя за рамки указанных ограничений, программист должен записать программу так, чтобы ее логика легко прослеживалась.

14.7. ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ

Мы не можем подробно остановиться на рассмотрении всех дополнительных возможностей Кобол. В этом разделе будут рассмотрены следующие его элементы: статьи **VALUE**, **REDIFINES** и **JUSTIFIED**, средства отладки программ и организации библиотек, средства сортировки-объединения, средства написания отчетов. Здесь будут рассмотрены только наиболее важные стороны этих элементов. Более подробные сведения о них читатель может получить из справочного пособия по Коболу.

Статьи **VALUE**, **REDIFINES** и **JUSTIFIED**

Один из способов присвоения начального значения переменной или элементарному данному в статье описания записи в начале выполнения программы заключается в задании этого значения в статье **VALUE**, которая имеет следующую форму:

VALUE [IS] литерал

Здесь *литерал* — это числовая или нечисловая константа, которая согласуется со статьей **PICTURE** для переменной. Например, если требуется присвоить начальное значение 0 числовой переменной **I**, то можно воспользоваться одним из следующих описаний:

77 I PIC S9(5) VALUE 0.

77 I PIC S9(5) VALUE ZERO.

Следует отметить, что такой способ присвоения начального значения переменной имеет весьма ограниченное использование, поскольку он не может применяться для переменных, имеющих статью **OCCURS** или **REDEFINES**. Кроме того, такое присвоение может выполняться только в секции рабочей памяти Раздела данных.

Статья **REDEFINES** позволяет программисту переименовывать область памяти двумя или более различными способами. Она может использоваться, например, в тех случаях, когда одно 80-знаковое значение иногда должно обрабатываться как 80 смежных однознаковых элементов. Она имеет следующую форму:

REDEFINES имя-данных

Здесь *имя-данных* — это имя переопределяемой области. Имя, которое переопределяет эту область, должно быть описано сразу же после его описания. Например, рассмотрим следующие описания:

01 CARD-AREA.

02 CARD PIC X(80).

02 CARD-POS REDEFINES CARD PIC X OCCURS 80.

Теперь если требуется обратиться к переменной **CARD-AREA** как к одной величине, то следует записать "**CARD**". С другой стороны, если требуется обратиться к **I**-му элементу (где **I** — целочисленная переменная) той же переменной **CARD-AREA**, то следует записать "**CARD-POS (I)**".

Статья **JUSTIFIED** позволяет при пересылке нечисловых данных выравнивать их по правому полю, а не по левому (что

имеет место при отсутствии статьи **JUSTIFIED**). Она имеет следующую форму:

JUSTIFIED [RIGHT]

Эта статья может быть использована только на элементарном уровне. Например, предположим, что область **CARD** заново описана следующим образом:

01 CARD-AREA.

02 CARD PIC X(80) JUSTIFIED RIGHT.

Тогда при выполнении оператора

MOVE "ALLEN" TO CARD

в пять крайних правых (а не крайних левых) позиций области **CARD** засылается значение **"ALLEN"**, а оставшаяся часть слева (а не справа) заполняется пробелами.

Средства отладки программ и организации библиотек

Кобол располагает средствами, предназначенными для упрощения утомительного процесса модификации и отладки программ. Эти средства позволяют программисту (1) трассировать выполнение программы и выдавать промежуточные результаты при ее отладке и (2) постоянно изменять и расширять существующие программы.

Средства отладки позволяют включать **отладочный алгоритм** в программу при проверке ее работы. С помощью этого алгоритма можно задавать условия, при выполнении которых будут выданы значения некоторых переменных, начата трассировка программы и т. д. К сожалению, в большинстве последних реализаций Кобола отсутствуют средства отладки, описанные в стандартном Коболе 1974 г. Как правило, они либо содержат другие средства отладки, либо вообще их не имеют. В стандартном Коболе 1968 г. такие средства отсутствуют.

Средства организации библиотек позволяют программисту записывать в библиотеку некоторые часто используемые сегменты Кобол-программ, а затем во время компиляции динамически вставлять один или более таких сегментов в программу. Обычно эти средства используются в тех случаях, когда длинное описание записи (например, описание записи основного файла учета служащих) совместно используется несколькими программами, обрабатывающими соответствующий файл. Это позволяет составлять и редактировать описание такой записи только один раз, а затем при необходимости автоматически получать его копию, избегая повторного выполнения одной и той же работы и повторения ошибок, которые при этом возникают.

Средства сортировки-объединения

Поскольку в задачах обработки данных часто возникает необходимость в сортировке и объединении файлов, в Коболе было предусмотрено мощное средство, позволяющее непосредственно выполнять эти действия в программе. Для сортировки и объединения файлов необходимо выполнить следующие действия:

1. В параграфе управления файлами Раздела оборудования определить файл, который должен быть упорядочен (объединен).

2. В статьях описания сортировки (SD) секции файлов в Разделе данных описать файл, который должен быть упорядочен (объединен), и разбивку его записей.

3. С помощью операторов **SORT** или **MERGE**, записываемых в Разделе процедур, указать, что должна быть выполнена процедура сортировки или объединения соответственно. До начала, а также после окончания процесса сортировки (**SORT**) или объединения (**MERGE**) может выполняться дополнительная обработка записей с помощью операторов **RELEASE** и **RETURN** соответственно.

Файл, который должен быть упорядочен (объединен), определяется и связывается с устройством ввода-вывода с помощью статей **SELECT** и **ASSIGN** (в параграфе управления файлами Раздела оборудования) точно так же, как и обычные файлы, используемые в программе. Однако процедуру сортировки или объединения можно выполнять не с индексированными, а с последовательными файлами.

Для файла, который должен быть упорядочен (объединен), в секцию файлов Раздела данных необходимо включить статью описания сортировки (**SD**). Она имеет следующую форму:

SD имя-файла

RECORD CONTAINS целое [**CHARACTERS**].

Здесь *целое* — это длина логической записи файла. Далее должны следовать одна или более статей описания записи для этого файла.

Оператор **SORT** имеет следующую форму:

SORT имя-файла-1

[ON] { **ASCENDING**
DESCENDING } **KEY** список-имен-данных
{ **INPUT PROCEDURE** [IS] имя-секции-1 **[THROUGH]** имя-секции-2
USING имя-файла-2
{ **OUTPUT PROCEDURE** [IS] имя-секции-3 **[THROUGH]** имя-секции-4
GIVING имя-файла-3

Здесь *имя-файла-1* — это имя так называемого **файла сортировки**, и оно совпадает с *именем-файла*, определенным в статье **SD**. *Имя-файла-2* и *имя-файла-3* — входной и выходной файлы сортировки соответственно. Они должны быть определены в обычных статьях **FD** секции файлов Раздела данных. *Список-имен-данных* — список, состоящий из одного или более имен-данных, которые определяются в описании файла сортировки. Они вместе составляют управляющий ключ сортировки. *Имя-секции-1, -2, -3 и -4* — это имена секций Раздела процедур.

В операторе **SORT** должна быть указана одна из статей **INPUT PROCEDURE** или **USING**, но не обе. Если указывается статья **INPUT PROCEDURE** (процедура ввода), то для создания файла сортировки *имя-файла-1* выполняется секция *имя-секции-1* (или выполняются секции с *имя-секции-1* по *имя-секции-2*). При каждой передаче записи в файл сортировки должен выполняться следующий оператор в процедуре ввода:

RELEASE имя-записи

Здесь *имя-записи* — это статья описания записи, определенная в соответствии со статьей описания сортировки (**SD**) для файла сортировки. Далее, процедура ввода должна быть независимой, так что управление не может передаваться из нее за ее пределы или из другого места программы в эту процедуру. С другой стороны, если указывается статья **USING**, то записи автоматически передаются в файл сортировки непосредственно из файла *имя-файла-2*. Однако до выполнения оператора **SORT** должен быть открыт (с помощью оператора **OPEN**) файл *имя-файла-2*.

Аналогично если указывается статья **OUTPUT PROCEDURE** (процедура вывода), то файл сортировки, полученный в результате выполнения операции сортировки, становится доступным для секции *имя-секции-3* (или для секций с *имя-секции-3* по *имя-секции-4*). Для того чтобы получить отдельную запись из файла сортировки, процедура вывода должна содержать оператор **RETURN**, который имеет следующую форму:

RETURN имя-файла [**RECORD**]

AT END оператор

Здесь *имя-файла* — это файл сортировки, из которого запись будет считана и помещена в статью описания записи файла сортировки. *Оператор* — предписываемый оператор, который должен быть выполнен при достижении конца упорядоченного файла. С другой стороны, если указывается статья **GIVING**, упорядоченный файл будет записан непосредственно в файл с именем *имя-файла-4*, который также должен быть открыт (с помощью оператора **OPEN**) до выполнения оператора **SORT**.

Средства написания отчетов

При решении коммерческих задач обычно возникает необходимость в составлении программным способом одного или более тщательно оформленных отчетов. Средства написания отчетов, существующие в Коболе, позволяют упростить эту задачу, предоставляя программисту возможность лишь указывать структуру отчета, а не писать программу, которая потребовалась бы для его печати. В этом разделе будут рассмотрены основные из этих средств. Более подробные сведения о них читатель может получить из других справочников.

Для того чтобы воспользоваться средствами написания отчетов, необходимо поместить в Кобол-программу следующие элементы:

Раздел данных

1. Статья описания файла (**FD**) в секции файлов. Эта статья определяет выходной файл, в который должен быть записан каждый отчет.

2. Секция отчетов, содержащая описание каждого отчета, который должен быть сгенерирован.

Раздел процедур

1. Оператор **INITIATE**, инициирующий генерацию отчета.

2. Оператор **GENERATE**, выполняющий генерацию отчета.

3. Оператор **TERMINATE**, завершающий генерацию отчета.

Программа должна также устанавливать доступ к последовательному входному файлу, отдельные записи которого содержат основные данные для отчета.

Этот файл должен быть упорядочен в соответствии с некоторым заданным множеством полей, называемых управляющими. Всякий раз, когда считывается запись, управляющие поля которой отличаются от управляющих полей предыдущей записи, возникает прерывание управления.

Всякий отчет делится на последовательность групп, каждая из которых содержит три различных вида информации:

- Заголовки (идентификация отчета и управляющие данные для текущей группы).
- Строки сообщений (данные отдельных записей отчета).
- Концовки (итоговая информация, получаемая из строк сообщений для текущей группы).

Например, краткий отчет о продаже товара делится на последовательность групп, по одной на каждый штат (**STATE**) и каждый округ (**DISTRICT**), указанные в файле. Заголовок каждой группы определяет штат и округ, а также содержит другую буквенную информацию. В строках сообщений содер-

жаты итоговые цифры для каждого продавца, а в концовках — итоговые цифры для всех продавцов штата и округа.

Для файлов, в которых должны быть сгенерированы отчеты, в секции файлов Раздела данных следует записать обычную **FD**-статью. Она должна содержать следующую специальную статью:

REPORT IS название-отчета

Здесь *название-отчета* — это любое уникальное незарезервированное слово, определенное пользователем, которое идентифицирует отчет.

Для каждого отчета, который должен быть сгенерирован, в секции отчетов содержится одна *статья описания отчета (RD)* и набор *статей описания групп отчета*. Секция отчетов всегда должна быть последней секцией Раздела данных, а ее заголовок начинается в поле A и записывается следующим образом:

REPORT SECTION.

В статье описания отчета определяются общие характеристики отчета, главным образом его управляющие поля, название, а также число строк в странице. Эта статья начинается в поле A и имеет следующую форму:

RD название-отчета **CONTROLS ARE** список-имен
PAGE LIMIT IS целое **LINES.**

Здесь *название-отчета* — название отчета, а *список-имен* — список имен управляющих полей отчета (например, **STATE-DIST** и **SALESPSN-ID**), отделенных друг от друга запятой и пробелом (, _). Порядок, в котором эти имена перечислены, указывает иерархию управляющих полей; первое имя в списке соответствует главному управляющему полю и т. д.

За **RD** — статьей для отчета следует набор статей описания группы отчета, который определяет разбивку и содержание типичных заголовка группы, строк сообщений и концовки. Здесь также определяются прерывания управления. Каждая из этих трех типов *групп отчета* описывается с помощью иерархической структуры, аналогичной той, которая существует в статье описания записи.

Группа отчета, описывающая заголовок, имеет следующую форму:

01 TYPE CONTROL HEADING имя **LINE NEXT PAGE.**
{описание-заголовка}

Здесь *имя* определяет прерывание управления, при возникновении которого будет сгенерирован заголовок для новой группы. *Описание-заголовка* определяет фактические значения, ко-

торые составляют заголовок. Оно является иерархией статей описания записи, имеющих следующую форму:

```

номер-уровня [имя]
[ LINE { целое-1
      NEXT PAGE } ]
[ COLUMN целое-2 ]
[ PICTURE шаблон ]
[ { SOURCE идентификатор }
  { VALUE литерал } ]

```

Здесь *номер-уровня* — это двузначное целое число, заключенное в пределах от 02 до 49, которое определяет иерархический уровень статьи с именем *имя*.

Статья **LINE** указывает на переход к строке с номером *целое-1* или к первой строке следующей страницы (параметр **NEXT PAGE**). Если эта статья опущена, то печать осуществляется с текущей строки. Статья **COLUMN** указывает на переход к позиции с номером *целое-2* в данной строке.

Статьи **PICTURE** и **SOURCE (VALUE)** либо обе используются, либо обе не используются. В статье **PICTURE** обычно описывается формат, в котором должен быть напечатан элемент заголовка, а статья **SOURCE** или **VALUE** используется для задания фактического значения, которое будет напечатано. Статья **VALUE** используется при печати литеральной константы, а статья **SOURCE** — при печати значения *идентификатора*.

Элементы описания **CONTROL HEADING** (управляемый заголовок) состоят из четырех различных строк, которые располагаются в возрастающем порядке с верхней части страницы. В первой строке располагаются два элемента: первый — заглавие в форме литеральной константы, как, например, **“SALES SUMMARY REPORT”** (краткий отчет о продаже товара), и второй — переменная, указанная в статье **SOURCE**, например **“THIS-MONTH”** (в этом месяце).

Строки сообщений определяются аналогично, за исключением того что их тип не **TYPE CONTROL HEADING**, а **TYPE DETAIL**. Кроме того, вертикальное расположение строк сообщений может быть описано следующим образом:

LINE PLUS целое

что означает пропуск числа строк, равного *целому*, после предыдущей строки. С другой стороны, отчет может быть определен таким образом, что отдельные строки сообщений не будут печататься. Это достигается путем исключения обеих статей **LINE** и **COLUMN** из описания строки.

Управляемые концовки определяются аналогично, однако следует учесть некоторые дополнительные особенности. Во-первых, их статьи уровня 01 имеют следующую форму:

01 TYPE CONTROL FOOTING имя.

Здесь *имя* указывает прерывание управления, при возникновении которого будет сгенерирована управляемая концовка. Во-вторых, значение, печатаемое в управляемой концовке, может быть указано в статьях **SOURCE**, **VALUE** или **SUM**. Последняя статья используется в тех случаях, когда требуется вычислить сумму значений всех вхождений *идентификатора* до возникновения прерывания управления. Статья **SUM** имеет следующую форму:

SUM идентификатор

Это краткое описание средств составления отчетов позволит читателю усвоить основные понятия о них. Для более полного изучения данного вопроса читатель должен обратиться к справочному пособию по Коболу.

14.8. СРАВНЕНИЕ СТАНДАРТНЫХ ВЕРСИЙ 1968 И 1974 ГГ.

Ниже приводятся основные различия между стандартными версиями Кобола 1968 г. и 1974 г.

1. Вместо параграфа **REMARKS** и оператора **NOTE** из стандарта 1968 г. в 7 позиции указывается звездочка (*) для обозначения комментария, который может быть вставлен в любое место программы.

2. Оператор **EXAMINE** из стандарта 1968 г. был заменен более общим оператором **INSPECT**.

3. Некоторые требования стандарта 1968 г. были ослаблены: данные 77-го уровня могут не предшествовать всем данным уровня 01; запятой, точке и точке с запятой может предшествовать пробел; между переменной и ее индексом можно не оставлять пробела.

4. Средства обработки строк были усовершенствованы путем добавления операторов **STRING** и **UNSTRING**. Кроме того, программист может динамически переопределять сортирующую последовательность, которая используется как базис для сравнения строк.

5. К средствам ввода-вывода с прямым доступом, имеющимся в стандарте 1968 г., были добавлены средства ввода-вывода, основанные на использовании индексированных файлов. Индексированные файлы допускают как прямой, так и последовательный доступ, а файлы с прямым доступом только прямой доступ.

6. Средства сортировки стандарта 1968 г. были названы средствами сортировки-объединения и усовершенствованы таким образом, что появилась возможность объединять два файла с помощью одного оператора (**MERGE**).

7. В стандарте 1974 г. средства написания отчетов были усовершенствованы и определены более понятно.

8. В стандарте 1974 г. средства отладки являются совершенно новыми.

9. В стандарте 1974 г. средства связи с подпрограммами (операторы **CALL** и **EXIT PROGRAM**, секция связей) являются совершенно новыми.

10. В стандарт 1974 г. включены новые средства написания программ, предназначенные для решения задач связи.

15

ФОРТРАН

Г. Хелмс

15.1. ВВЕДЕНИЕ

Фортран — это сокращенное название от FORMula TRANslator (переводчик формул). Фортран является одним из самых старых и в то же время одним из наиболее широко распространенных языков высокого уровня.

Первая реализация Фортрана, Фортран I, была разработана в 1954 г. и впервые использована на вычислительной машине IBM 704. В 1958 г. появилась усовершенствованная версия, известная под названием Фортран II. Спустя несколько лет после появления Фортрана II был разработан язык Фортран III. Однако широкого распространения он не получил.

Наиболее удачной оказалась версия Фортрана, известная под названием Фортран IV. Она была разработана в 1962 г. и вскоре благодаря успешной реализации на системах IBM получила широкое распространение. В 1962 г. Американская ассоциация по стандартам организовала комитет по разработке единой версии Фортрана. В 1966 г. комитет завершил свою работу, результатом которой явился язык Фортран ANSI, или Фортран-66.

После 1966 г. было разработано несколько версий Фортрана, предназначенных для обучения. В частности, наиболее известными из них являются Ватфор и Ватфив, разработанные в Университете г. Ватерлоо (Онтарио). Было также разработано и несколько других версий Фортрана. В результате этого в 1977 г. появилась пересмотренная версия Фортрана, получившая название Фортран-77. Она заменила Фортран-66 в качестве стандартной (ANSI) версии Фортрана. Фортран-77 включает в себя некоторые наиболее употребительные элементы Ватфора и Ватфива. В этой главе будет дано описание Фортрана-77.

Наиболее широко Фортран используется при решении задач, которые могут быть сформулированы в математической форме. Благодаря способности обрабатывать комплексные числа, его

часто используют при решении научных задач прикладного характера. Однако этим не ограничивается область применения Фортрана.

15.2. ФОРМАТ ПРОГРАММЫ

Фортран-программы состоят из строк, называемых **операторами**. Операторы обычно выполняются последовательно, начиная с первого и кончая последним оператором программы. Однако такой порядок выполнения может быть изменен с помощью управляющих операторов и операторов передачи. Обычно Фортран-программы оканчиваются операторами **STOP** и **END**. Оператор **STOP** завершает выполнение программы, а оператор **END** сообщает компилятору о достижении конца исходной Фортран-программы. Оператор **END** должен быть последним в каждой Фортран-программе.

В случае необходимости к Фортран-программе можно добавить комментарии, помещая в первую позицию строки звездочку (*) или букву **C**. Комментарии никак не влияют на выполнение программы.

Если требуется, операторы Фортрана можно пометить с помощью номеров. Оператор можно не нумеровать в том случае, когда на него не ссылается другой оператор (как, например, управляющий оператор или оператор передачи). Номера могут быть любыми целыми числами, состоящими из пяти или менее цифр, и помещаются в первых пяти позициях строк. Не требуется, чтобы они были каким-либо образом упорядочены; больший номер может предшествовать меньшему. Однако в практике хорошего программирования принято выбирать номера не произвольным образом, а так, чтобы при этом была отражена логика работы программы.

Если оператор Фортрана не помещается в одной строке, его можно продолжить на следующие строки, записывая в шестой слева позиции каждой из строк продолжения любой знак, отличный от нуля. В первой строке оператора, который должен быть продолжен, этот знак может быть опущен.

15.3. ВВОД С ПЕРФОКАРТ И ТЕРМИНАЛА

В течение многих лет Фортран-программы вводились в вычислительную машину с перфокарт. В последнее время для ввода Фортран-программ стали использоваться терминалы. Требования, предъявляемые к подготовке Фортран-программы для ввода, зависят от того, как он осуществляется — с помощью перфокарт или терминала.

Перфокарта делится на 80 колонок, которые нумеруются слева направо. Первая колонка используется для пометки строки, содержащей комментарий. В первых пяти колонках указываются номера строк. Шестая колонка предназначена для знака, указывающего, что оператор является продолжением предыдущей строки. Колонки 7—72 используются для записи операторов Фортрана. Информация, содержащаяся в оставшихся колонках (73—80), вычислительной системой не считывается. Они могут содержать пробелы, однако обычно их используют для последовательной нумерации перфокарт, с тем чтобы упорядочить операторы Фортран-программы.

Основное отличие описанного выше формата от того, который установлен для случая, когда программа вводится с терминала, заключается в том, что при использовании терминала каждой строке программы должен быть приписан номер следования. В качестве номеров следования строк обычно используются пятизначные номера от 00001 до 99999. Операторы будут выполняться в порядке возрастания номеров строк.

Для того чтобы показать, что строка содержит продолжение оператора, используется знак плюс (+), который помещается в колонке, непосредственно следующей за номером следования строки. Если в указанную колонку помещается любой другой знак, то строка обрабатывается как комментарий.

Номера следования строк не заменяют необязательные номера операторов. Номера операторов могут располагаться между номерами следования строк и самими операторами, как показано в следующем примере:

```
00010* ЭТА СТРОКА ЯВЛЯЕТСЯ КОММЕНТАРИЕМ  
00020 100 A = B + C
```

15.4. КОНСТАНТЫ И ПЕРЕМЕННЫЕ

К именам констант и переменных в Фортране предъявляются одни и те же требования. Первым знаком должна быть буква алфавита. Остальные знаки могут быть буквами алфавита или цифрами (0—9). Максимальная длина имени константы или переменной составляет шесть знаков.

Первая буква имени константы или переменной определяет ее тип (вещественный или целый). Имена целых констант и переменных начинаются с букв I, J, K, L, M или N. Имена вещественных констант и переменных начинаются с остальных букв алфавита (A—H и O—Z). Однако существуют операторы Фортрана, позволяющие обходить это соглашение. Они будут рассмотрены позже в этой главе.

Фортран позволяет использовать шесть типов констант и переменных: целые, вещественные, с удвоенной точностью,

комплексные, логические и символьные. Эти типы характеризуются следующим образом:

Целое. Значение без десятичной точки, буквы **E** или буквы **D**. Оно может содержать знак. Если знак опущен, то значение обрабатывается как положительное.

Вещественное. Значение с десятичной точкой или выраженное в показательной форме с помощью буквы **E**. Значение этого типа имеет семь значащих цифр.

Удвоенная точность. Вещественное значение с шестнадцатью значащими цифрами вместо семи. Оно записывается в показательной форме с помощью буквы **D** вместо **E**.

Комплексное. Значение, содержащее вещественную и мнимую числовые компоненты. Комплексное значение представляется двумя вещественными значениями. Например, выражение

(1.7, 7.98)

представляет комплексное число

$1.7 + 7.98i$

Логическое. Константа или переменная, которые могут принимать одно из двух значений, **.TRUE.** или **.FALSE.**

Символьное. Значение, являющееся строкой символов, заключенных в кавычки. Например,

'CHARACTER'

Если кавычка необходима внутри строки, то нужно использовать две следующие друг за другом кавычки. Так, строка **VARIABLE'S NAME** представляет в виде **'VARIABLE'S NAME'**.

15.5. ОПИСАНИЯ ТИПОВ

Как отмечалось выше, Фортран предоставляет возможность определять целые и вещественные константы и переменные с помощью первой буквы имени. Однако могут возникнуть ситуации, когда требуется определить целочисленную переменную, имя которой начинается с буквы **A**, или вещественную константу с таким именем, как, например, **IHT**. Кроме того, программист должен иметь возможность определять имена, представляющие комплексные константы, логические переменные и т. д. Для этих случаев в Фортране предусмотрены описания **TYPE** и **IMPLICIT**.

Описание **TYPE** определяет тип данных, которые представляются именами переменных. Оно имеет следующую форму:

тип имени переменных

где *тип* — **INTEGER**, **REAL**, **CHARACTER**, **DOUBLE PRECISION**, **COMPLEX** или **LOGICAL**. Предположим, что переменную **VECTOR** требуется описать как комплексную. Тогда описание выглядит следующим образом:

COMPLEX VECTOR

Предположим также, что **TEST1** и **TEST2** требуется описать как логические константы. Тогда описание выглядит следующим образом:

LOGICAL TEST1, TEST2

Описание **IMPLICIT** позволяет устанавливать определенный тип для всех переменных и констант, имена которых начинаются с заданной буквы. Оно имеет следующую форму:

IMPLICIT тип (буквы)

где *тип* — **REAL**, **COMPLEX** и т. д., а *буквы* — это первые буквы имен переменных, тип которых должен быть таким, который указан в этом описании. Буквы в скобках могут отделяться друг от друга запятыми (A,B,D,F) или, если должна быть описана область букв, черточкой. Например, (A,B,C,D,E,F) эквивалентно (A—F).

Действие описания **IMPLICIT** по отношению к некоторым константам и переменным можно отменить с помощью следующего за ним описания **TYPE**. Например, описания

**IMPLICIT DOUBLE PRECISION (A—Z)
COMPLEX VECTOR, FUNC**

задают удвоенную точность для всех констант и переменных, за исключением **FUNC** и **VECTOR**, которые описаны как комплексные.

15.6. МАССИВЫ

Массив в Фортране — это группа данных, обращение к которой осуществляется с помощью имени одной переменной. Обращение к отдельному элементу этой группы осуществляется с помощью имени переменной, за которым следуют одно или более чисел, заключенных в скобки. Эти числа называются индексами. Массив может иметь до семи индексов. Индексы отделяются друг от друга запятыми.

Массивы определяются с помощью оператора **DIMENSION**, который имеет следующую форму:

DIMENSION имя переменной (первый индекс, второй индекс и т. д.)

Например, оператор **DIMENSION ABLE(2,3)** определяет массив с элементами **ABLE(1,1)**, **ABLE(1,2)**, **ABLE(2,1)**, **ABLE(2,2)** и **ABLE(2,3)**.

Все индексы должны быть либо целыми числами, либо целочисленными выражениями, либо целочисленными неиндексированными переменными.

Операторы **DIMENSION** должны располагаться в начале программы, до всех выполняемых операторов.

Индексы в операторе **DIMENSION** можно также указать с помощью верхних и нижних границ. Например, оператор **DIMENSION YEAR(1952:1982)** определяет массив с элементами **YEAR(1952)**, **YEAR(1953)**, **YEAR(1954)**, ..., **YEAR(1982)**.

15.7. ПРИСВОЕНИЕ ЗНАЧЕНИЙ

Часто требуется присвоить переменной начальное значение. Это делается с помощью оператора **DATA**. Он имеет следующие две формы:

DATA имена переменных/значения

DATA имена переменных/значения, имена переменных/значения и т. д.

Если нескольким переменным должно быть присвоено одно и то же значение, то для удобства можно использовать символ повторения. Символ повторения — это целое число, определяющее число повторений, и звездочка, стоящие перед значением, которое должно быть присвоено. Например, оператор

DATA A,B,C,D,E/5*1.5/

присваивает переменным **A**, **B**, **C**, **D** и **E** значение 1.5.

Имена переменных должны согласовываться с типом данных, присваиваемых им с помощью оператора **DATA**. Например, если переменной, имя которой начинается с буквы **I**, будет присваиваться вещественное значение, то возникнет ошибка.

Операторы **DATA** должны располагаться после операторов спецификации, как, например, **DIMENSION**, но до любого выполняемого оператора.

15.8. АРИФМЕТИЧЕСКИЕ ОПЕРАТОРЫ

Для обозначения арифметических операций в Фортране используются следующие символы:

+	Сложение
-	Вычитание
/	Деление
*	Умножение
**	Возведение в степень

Арифметические операторы в каждой строке программы выполняются слева направо в следующем порядке:

1. Возведение в степень.
2. Умножение и деление.
3. Сложение, вычитание и арифметическое отрицание.

Однако указанный выше порядок может быть изменен с помощью скобок. Первыми всегда выполняются операторы в скобках.

В Фортране допускаются арифметические операции, выполняемые одновременно над целыми и вещественными значениями (операции над числами со смешанными характеристиками). Однако результат будет вещественным значением.

Если при целочисленном делении образуется дробная часть, то она отбрасывается. Например, результатом выполнения операции $5/2$ является 2, а не 2.5.

15.9. ОПЕРАТОРЫ ОТНОШЕНИЯ

Два арифметических выражения или переменные могут сравниваться с помощью следующих операторов отношения:

.LT.	Меньше
.LE.	Меньше или равно
.EQ.	Равно
.NE.	Не равно
.GT.	Больше
.GE.	Больше или равно

15.10. СИМВОЛ РАВЕНСТВА

Символ `=` может быть использован в арифметических выражениях для указания результата выполнения операции и присвоения его переменной. Например, с помощью выражения $5/2 = A$ переменной `A` присваивается значение 2.5. Символ `=` может быть использован вместо оператора `DATA` для присвоения переменным начальных значений (например, `A = 2.5`).

15.11. УПРАВЛЯЮЩИЕ ОПЕРАТОРЫ И ОПЕРАТОРЫ ПЕРЕДАЧИ

Как отмечалось выше, операторы Фортран-программы обычно выполняются последовательно, начиная с первого и кончая последним оператором программы. Однако в Фортране существует несколько операторов, которые изменяют этот порядок выполнения. Ниже дается краткое описание каждого такого оператора.

GO TO Существуют три типа оператора **GO TO**: безусловный, условный и назначаемый.

Безусловный. Безусловный оператор **GO TO** осуществляет передачу управления оператору, номер строки которого указан после **GO TO**. При выполнении оператора

GO TO 100

управление сразу же будет передано оператору с номером строки 100. Затем будет выполнен оператор, непосредственно следующий за оператором с номером строки 100. После этого операторы будут выполняться обычным образом.

Условный. Безусловный оператор **GO TO** выполняется независимо от каких-либо условий. Условный оператор **GO TO**, напротив, передает управление различным операторам в зависимости от значения индексной переменной. Он имеет следующую форму:

GO TO (номера строк), индексная переменная

где *номера строк*, указанные в скобках, отделяются друг от друга запятыми, а *индексная переменная* — это целочисленная переменная.

Если значение индексной переменной равно 1, то условный оператор **GO TO** осуществляет передачу управления оператору, номер строки которого указан в скобках первым. Если значение индексной переменной равно 2, то управление передается оператору, номер строки которого указан вторым. Если значение индексной переменной равно 3, то управление передается оператору, номер строки которого указан третьим, и т. д. Если значение индексной переменной больше числа номеров, указанных в скобках, то выполняется оператор, непосредственно следующий за оператором **GO TO**. Ниже приводится пример условного оператора **GO TO**.

GO TO (50, 20, 75) N

IF N = 1 2 3 4 ↓

Выполняется оператор, следующий непосредственно за данным операт.

Назначаеый. Для применения назначаемого оператора **GO TO** требуются два оператора: **GO TO** и **ASSIGN**. Назначаеый оператор **GO TO** имеет следующую форму:

ASSIGN первая переменная **TO** вторая переменная
GO TO вторая переменная (номера строк)

где *первая* и *вторая переменные* — целочисленные переменные. *Первая переменная* может также быть целой константой. Оператор **ASSIGN** должен быть выполнен до выполнения назначаемого оператора **GO TO**.

При выполнении оператора **ASSIGN** значение первой переменной присваивается второй переменной, которая затем используется в назначаемом операторе **GO TO**. Назначаемый оператор **GO TO** осуществляет передачу управления одному из операторов, номера строк которых указаны в скобках, в зависимости от значения второй переменной аналогично тому, как это делается для вычисляемого оператора **GO TO**. Однако значение второй переменной должно совпадать с одним из номеров строк, указанных в скобках. Кроме того, имя второй переменной не должно использоваться в программе для других целей.

IF ... THEN ... ELSE Операторная конструкция **IF ... THEN ... ELSE** позволяет осуществлять выбор между взаимоисключающими действиями в зависимости от заданного условия. Она имеет следующую форму:

```
IF (условие) THEN первая последовательность
                ELSE вторая последовательность
END IF
```

где *условие* — это отношение или логическое выражение, а *первая* и *вторая последовательности* — один или более операторов. Здесь вначале проверяется условие, записанное в скобках. Если оно выполнено, то выполняется первая последовательность операторов (следующая за словом **THEN**). В противном случае выполняется вторая последовательность операторов (следующая за словом **ELSE**). Эта конструкция может быть также использована с оператором **ELSE IF** следующим образом:

```
IF (первое условие) THEN первая последовательность
    ELSE IF (второе условие) THEN вторая последовательность
    END IF
END IF
```

В приведенном выше примере вначале проверяется первое условие. Если оно выполнено, то выполняется первая последовательность операторов. В противном случае проверяется второе условие. Если второе условие выполнено, то выполняется вторая последовательность операторов; в противном случае выполняется третья последовательность операторов¹⁾.

Вычисляемое IF. Этот оператор осуществляет передачу управления одному из указанных операторов в зависимости от

¹⁾ Следующая за операторной конструкцией. — Прим. ред.

значения арифметического выражения. Он имеет следующую форму:

IF (выражение) p_1, p_2, p_3

где *выражение* в скобках — это арифметическое выражение, а p_1, p_2, p_3 — номера строк трех операторов. Если значение выражения меньше нуля, то управление передается на p_1 . Если значение выражения равно нулю, то управление передается на p_2 . Если значение выражения больше нуля, то управление передается на p_3 .

Логическое IF. При выполнении этого оператора вычисляется логическое выражение, записанное в скобках. Если оно истинно, то выполняется оператор, расположенный в той же строке. Если значение логического выражения есть ложь, то выполняется оператор, непосредственно следующий за оператором **IF**. Например, если **A** меньше **B**, то оператор

IF (A .LT. B) STOP

завершает выполнение программы; в противном случае он игнорируется. Оператором в строке логического **IF** может быть любой оператор Фортрана, за исключением другого логического **IF** или оператора **DO** (который будет описан ниже).

В логических выражениях помимо описанных выше операторов отношения могут использоваться логические операторы **.NOT.**, **.AND.** и **.OR.** Их действие заключается в следующем:

.NOT. Возвращает значение **.TRUE.**, если значение выражения есть **.FALSE.**. Возвращает значение **.FALSE.**, если значение выражения есть **.TRUE.**.

.AND. Используется для объединения двух выражений. Возвращает значение **.TRUE.**, если значения обоих выражений суть **.TRUE.**. В противном случае возвращает значение **.FALSE.**.

.OR. Используется для объединения двух выражений. Возвращает значение **.TRUE.**, если значение по крайней мере одного выражения есть **.TRUE.**. Возвращает значение **.FALSE.**, если значения обоих выражений суть **.FALSE.**.

Если в одном выражении содержатся операторы отношения, а также логические и арифметические операторы, то все эти операторы выполняются слева направо в следующем порядке:

1. Возведение в степень
2. Умножение и деление
3. Сложение и вычитание
4. Операторы отношения
5. **.NOT.**
6. **.AND.**
7. **.OR.**

DO Оператор **DO** позволяет повторять требуемое число раз выполнение одного или нескольких операторов. Обычная форма оператора **DO** следующая:

```
DO метка I = p1, p2, p3
      .....
      операторы
      .....
метка CONTINUE
```

где *метка* — это номер строки оператора, *I* — целочисленная переменная (называемая **индексом** или **управляющей переменной**), *p₁* — начальное значение индексной переменной, *p₂* — конечное значение индексной переменной, *p₃* — шаг возрастания индексной переменной. Значения *p₁*, *p₂* и *p₃* могут быть отрицательными. В Фортране не требуется, чтобы повторяющиеся циклы, определенные с помощью оператора **DO**, заканчивались словом **CONTINUE**. Однако это принято в практике хорошего программирования.

Типичный оператор **DO** может выглядеть, например, так:

```
DO 100 I = 5, 1000, 10
      N = N + 1
100 CONTINUE
```

При выполнении этого оператора **DO** к значению *N* постоянно прибавляется 1. Начальное значение индексной переменной равно 5. После первого выполнения цикла к индексной переменной прибавляется 10, и ее значение становится равным 15. Этот процесс продолжается, пока значение индекса меньше или равно 1000. Когда значение индексной переменной превысит 1000, будет выполняться первый оператор, следующий за словом **CONTINUE**.

PAUSE Оператор **PAUSE** временно останавливает выполнение программы и выдает слово **PAUSE** на выходное устройство вычислительной системы. Этот оператор имеет следующую форму:

PAUSE целая константа

где *целая константа* содержит пять или менее цифр. Оператор **PAUSE** используется для того, чтобы позволить оператору ЭВМ помещать ленты или диски, проверить результаты, полученные в определенной точке программы, и т. д.

STOP Этот оператор завершает выполнение программы. Как отмечалось выше, он располагается в конце программы, перед оператором **END**. Он также используется в качестве следствия в операторах **IF** и **IF...THEN...ELSE**.

END Этот оператор сообщает компилятору о достижении конца программы. Он должен быть последним оператором программы и не должен иметь номера.

15.12. ПОДПРОГРАММЫ

Фортран предоставляет возможность писать некоторые программы отдельно от основной программы и затем использовать их в основной программе. Это позволяет выполнять нужное действие, не записывая каждый раз одну и ту же группу операторов.

Фортран допускает использование двух типов подпрограмм: функций и процедур. Функции возвращают только одно значение, в то время как процедуры могут возвращать более одного значения.

Функции определяются с помощью оператора **FUNCTION**, который имеет следующую форму:

```
FUNCTION имя (фиктивные аргументы)
.....
операторы Фортрана (за исключением операторов
FUNCTION, SUBROUTINE и BLOCK DATA)
.....
RETURN
.....
END
```

Здесь *имя* удовлетворяет тем же правилам, которым удовлетворяют имена констант и переменных. *Фиктивные аргументы* — это имена переменных, используемые внутри функции для представления значений, *передаваемых* функции из основной программы. Тип результата (вещественный, комплексный и т. д.), возвращаемого функцией, должен согласовываться с именем, следующим за словом **FUNCTION**. В противном случае слову **FUNCTION** должно предшествовать соответствующее описание типа (**REAL**, **COMPLEX** и т. д.). Кроме того, внутри функции должен быть один оператор, присваивающий значение имени функции.

Обращение к функции из основной программы осуществляется посредством указания имени функции. Предположим, что функция **RATIO** определена следующим образом:

```
FUNCTION RATIO (A, B)
RATIO = A/B
RETURN
END
```

Предположим, что в основной программе нужно использовать функцию **RATIO** со значениями 15 и 2, а результат присвоить

переменной **RESULT**. Это может быть сделано с помощью оператора

RESULT=**RATIO**(15,2)

При этом в функции **RATIO** переменным **A** и **B** будут присвоены значения 15 и 2 соответственно. После выполнения этой функции имени **RATIO** в основной программе будет присвоено значение 7.5. Затем значение **RATIO** будет присвоено переменной **RESULT**.

Процедуры определяются аналогично функциям, но они могут возвращать более одного значения. Процедура обычно имеет следующую форму:

```

SUBROUTINE имя (фиктивные аргументы)
.....
операторы Фортрана (за исключением операторов
FUNCTION, SUBROUTINE и BLOCK DATA)
.....
RETURN
.....
END

```

Так же как и имя функции, имя процедуры должно удовлетворять тем же правилам, которым удовлетворяют имена констант и переменных.

Обращение к процедурам из основной программы осуществляется с помощью оператора **CALL**. Он имеет следующую форму:

CALL имя (фактические аргументы)

где *имя* — это имя процедуры. Между фактическими и фиктивными аргументами должно существовать взаимно однозначное соответствие, и, кроме того, их характеристики также должны соответствовать друг другу. При выполнении оператора **CALL** значения фактических аргументов присваиваются фиктивным аргументам. При завершении выполнения процедуры значения фиктивных аргументов обратно передаются фактическим аргументам. Фиктивные аргументы могут быть неиндексированными переменными или именами массивов. Фактические аргументы могут быть константами, неиндексированными переменными, элементами массивов, выражениями, именами массивов или обращениями к другой подпрограмме.

Для данных основной программы и подпрограмм можно вводить одну и ту же область памяти с помощью оператора **COMMON**. Он имеет следующую форму:

COMMON имена

где *имена* — это список данных, используемых как основной программой, так и подпрограммами. Предположим, что в основной программе имеется оператор

COMMON X, Y, Z

Для того чтобы в подпрограмме обращаться к данным, содержащимся в той же области памяти, может быть использован оператор

COMMON X, Y, Z

или

COMMON A, B, C

хотя обычно в операторах **COMMON** основной программы и подпрограмм используются одни и те же имена переменных. Типы соответствующих переменных должны согласовываться друг с другом.

Кроме того, можно резервировать *блоки* общей памяти, помечая такие блоки. Для этого используется оператор **COMMON**, имеющий следующую форму:

COMMON метка/имена/метка/имена

где *метки* и *имена* должны удовлетворять тем же требованиям, которым удовлетворяют имена констант и переменных. Помеченный оператор **COMMON** используется в тех случаях, когда в подпрограмме используется только часть данных, являющихся общими для основной программы и подпрограмм. В этом случае оператор **COMMON** подпрограммы должен содержать метки только тех областей данных, которые являются общими для данной подпрограммы.

В помеченный оператор **COMMON** данные могут быть введены посредством подпрограммы **BLOCK DATA**, имеющей следующую форму:

BLOCK DATA

.....

операторы **DATA, DIMENSION, IMPLICIT, TYPE,**
SAVE, EQUIVALENCE, COMMON, PARAMETER

.....

END

Оператор **BLOCK DATA** определит начальные значения для основной программы и подпрограмм.

Для того чтобы указать, что две переменные разделяют одну и ту же область памяти, может быть также использован оператор **EQUIVALENCE**. Оператор

EQUIVALENCE(A,B,C)

резервирует одну и ту же область памяти для переменных **A**, **B** и **C**.

Значения переменных подпрограммы, не содержащихся в непомеченном операторе **COMMON** и списке аргументов, при возврате управления основной программе “теряются”. Они могут быть сохранены с помощью оператора **SAVE**, имеющего форму

SAVE имя подпрограммы

или

SAVE имена переменных

При использовании первой формы сохраняются значения всех переменных процедуры. Во втором случае сохраняются значения только указанных переменных.

В Фортране имеется ряд *встроенных* функций, которые называют также внутренними. Они будут рассмотрены позже. Однако при вызове подпрограммы иногда возникает необходимость использовать в списке аргументов имя внутренней функции, функции, определенной пользователем, или процедуры. Поскольку имена подпрограмм и имена переменных удовлетворяют одним и тем же требованиям, то компилятор не сможет различить их, если не будет использован оператор **INTRINSIC** или **EXTERNAL**. Форма обоих этих операторов такова, что за словом **INTRINSIC** или **EXTERNAL** следует имя, которое должно быть описано. **INTRINSIC** используется с внутренними подпрограммами Фортрана, а **EXTERNAL** — с подпрограммами, определенными пользователем.

Вход в подпрограмму может быть осуществлен через точку, отличную от нормальной точки входа. Для этого используется оператор **ENTRY**, который помещается в нужные точки входа в процедуру. Он имеет следующую форму:

ENTRY имя (фиктивные аргументы)

Обращение к нему осуществляется с помощью оператора **CALL** так же, как и к обычной процедуре.

15.13. ВНУТРЕННИЕ ФУНКЦИИ

Фортран содержит ряд внутренних (заранее определенных и описанных) функций. Многие из них могут быть использованы только с определенными типами выражений, как, например, вещественными, комплексными и т. д. Обычная форма функции следующая:

функция (выражение)

Ниже дается перечень функций по типам.

Комплексные

CABS Возвращает абсолютное значение выражения.

CCOS Возвращает косинус угла, выраженного в радианах.

CEXP Возвращает результат возведения числа e в степень, равную значению выражения.

CLOG Возвращает натуральный логарифм выражения.

CMPLX Преобразует выражение в комплексное число.

CONJ Возвращает сопряженное комплексное число.

CSQRT Возвращает квадратный корень выражения.

CSIN Возвращает синус угла, выраженного в радианах.

Целочисленные

IABS Возвращает абсолютное значение выражения.

IDIM Возвращает положительную разность между двумя значениями.

IDINT Преобразует значение с удвоенной точностью в целое.

IFIX Преобразует вещественное значение в целое.

INT Отбрасывает дробную часть значения.

ISIGN Передает знак одного целого значения другому.

MAXO Выбирает наибольшее из нескольких значений.

MINO Выбирает наименьшее из нескольких значений.

MIN1 Выбирает наименьшее из нескольких значений, но преобразует каждый вещественный результат в целый.

С удвоенной точностью

DABS Возвращает абсолютное значение выражения.

DACOS Возвращает арккосинус выражения.

DASIN Возвращает арксинус выражения.

DATAN Возвращает арктангенс одного аргумента.

DANTAN2 Возвращает арктангенс двух аргументов.

DBLE Преобразует выражение в значение с удвоенной точностью.

DCOS Возвращает косинус аргумента, выраженного в радианах.

DCOSH Возвращает гиперболический косинус аргумента.

DDIM Возвращает положительную разность между двумя аргументами.

DEXP Возвращает результат возведения числа e в степень, равную значению выражения.

DINT Отбрасывает дробную часть значения выражения.

DLOG Возвращает натуральный логарифм выражения.

DLOG10 Возвращает десятичный логарифм выражения.

DMAX1 Выбирает наибольшее из нескольких значений.

DMIN1 Выбирает наименьшее из нескольких значений.

DMOD Возвращает остаток от деления двух чисел.

DNINT Возвращает ближайшее к значению выражения целое число.

DPROD Преобразует вещественное значение в значение с удвоенной точностью.

DSIGN Передает знак одного значения другому.

DSIN Возвращает синус выражения, вычисленного в радианах.

DSINH Возвращает гиперболический синус аргумента.

DSQRT Возвращает квадратный корень выражения.

DTAN Возвращает тангенс аргумента, выраженного в радианах.

DTANH Возвращает гиперболический тангенс выражения.

IDINT Преобразует выражение в ближайшее целое значение.

Вещественные

ABS Возвращает абсолютное значение выражения.

ACOS Возвращает арккосинус выражения.

AIMAG Возвращает мнимую часть комплексного числа.

AINT Отбрасывает дробную часть от значения выражения.

ALOG Возвращает натуральный логарифм выражения.

ALOG10 Возвращает десятичный логарифм выражения.

AMAX1 Выбирает наибольший из нескольких аргументов.

AMIN1 Выбирает наименьший из нескольких аргументов.

AMOD Возвращает остаток от деления двух чисел.

ANINT Возвращает ближайшее к значению аргумента целое число.

ASIN Возвращает арксинус аргумента.

ATAN Возвращает арктангенс выражения в радианах.

ATAN2 Возвращает арктангенс двух аргументов в радианах.

COS Возвращает косинус аргумента, выраженного в радианах.

COSH Возвращает гиперболический косинус аргумента.

DIM Возвращает положительную разность между двумя аргументами.

EXP Возвращает результат возведения числа e в степень, равную значению выражения.

FLOAT Преобразует значение выражения в вещественное число.

NINT Возвращает ближайшее целое значение.

REAL Преобразует комплексное значение в вещественное.

SIGN Передает знак одного значения другому.

SIN Возвращает синус аргумента, выраженного в радианах.

SINH Возвращает гиперболический синус аргумента.

SQRT Возвращает квадратный корень аргумента.

SNGL Преобразует значение с удвоенной точностью в значение с одинарной точностью.

TAN Возвращает тангенс аргумента, выраженного в радианах.

TANH Возвращает гиперболический тангенс аргумента.

15.14. ПАРАМЕТРЫ

Константе может быть присвоено имя с помощью оператора **PARAMETER**. Он имеет следующую форму:

PARAMETER имя

где *имя* должно удовлетворять тем же требованиям, которым удовлетворяют имена переменных. Если константе присвоено имя с помощью оператора **PARAMETER**, то ее значение не может быть изменено в программе.

15.15. ЗАДАНИЕ ИМЕНИ ПРОГРАММЫ

Фортран позволяет при необходимости задавать имя программы. Это делается с помощью оператора **PROGRAM**, имеющего следующую форму:

PROGRAM имя

Оператор **PROGRAM**, если он используется, должен быть первым оператором в программе.

15.16. МАНИПУЛИРОВАНИЕ СИМВОЛЬНЫМИ ДАННЫМИ

Данные типа **CHARACTER** выше были определены как значения, являющиеся строками символов, заключенными в кавычки. При программировании на Фортране программисты долгое время испытывали трудности, связанные с обработкой символьных данных. Однако Фортран-77 содержит некоторые новые средства манипулирования символьными (или строковыми) данными.

Две символьные строки могут быть соединены с помощью оператора сцепления **//**. Предположим, что

A='ABC' B='CBA'

Тогда в результате выполнения оператора

C=A//B

будет получено значение

C='ABCCBA'

Можно также получить подстроку символьной строки. Для этого используется оператор, имеющий следующую форму:

имя подстроки = имя строки (целое 1 : целое 2)

где имя подстроки — это имя, определяющее подстроку, имя строки определяет строку, из которой должна быть получена подстрока, целое 1 и целое 2 определяют соответственно начальную и конечную позиции подстроки. Предположим, что имеется строка **DIGITS** = '12345678'. Тогда в результате выполнения оператора

INTS = **DIGITS**(2 : 7)

будет получена подстрока **INTS** = "234567"

Для сравнения двух символьных строк имеются четыре внутренние функции. Они имеют следующую форму:

функция (строка-1, строка 2)

Каждая функция возвращает значение **.TRUE.**, если выполняется описанное условие. Эти функции следующие:

LGE Первая строка равна второй или следует за ней в лексикографическом порядке.

LGT Первая строка следует за второй в лексикографическом порядке.

LLE Первая строка равна второй или предшествует ей в лексикографическом порядке.

LLT Первая строка предшествует второй в лексикографическом порядке.

Имеются также следующие четыре внутренние функции для манипулирования символьными данными:

CHAR Преобразует одну цифру в символ.

ICAR Преобразует один знак в строку.

INDEX Возвращает целое значение, указывающее начальную позицию подстроки в более длинной строке. Форма:

INDEX (строка, подстрока)

где строка и подстрока представляются именами переменных.

LEN Возвращает длину символьной строки.

15.17. ОПЕРАТОРЫ ЭКВИВАЛЕНТНОСТИ

Фортран содержит также два оператора, позволяющие определить, эквивалентны ли друг другу два выражения. Эти операторы следующие:

.EQV. Результат есть **.TRUE.**, если выражения эквивалентны.

.NEQV. Результат есть **.TRUE.**, если выражения не эквивалентны.

Среди всех операционных символов **.EQV.** и **.NEQV.** имеют самый низкий приоритет, причем **.NEQV.** имеет более низкий приоритет, чем **.EQV.**

15.18. ОРГАНИЗАЦИЯ ФАЙЛОВ

Фортран располагает средствами обработки последовательных, прямых и потокоориентированных файлов, хотя эти средства не так универсальны, как в Коболе.

Для различных компиляторов и вычислительных систем методы создания и обработки файлов на диске и ленте различны. Поэтому операторы обработки файлов будут описаны в общих чертах. Более точное описание средств обработки файлов читатель найдет в справочном руководстве для используемой им системы.

OPEN Этот оператор связывает существующий файл с устройством ввода-вывода или генерирует новый файл. За словом **OPEN** следует заключенный в скобки список спецификаторов, описывающих файл. Существуют следующие спецификаторы:

UNIT = номер устройства для хранения файла

IOSTAT целочисленная переменная, описывающая состояние устройства ввода-вывода

FILE имя файла

ERR метка оператора, которому следует передать управление в случае возникновения ошибки

STATUS описывает состояние файла с помощью дескрипторов

OLD (старый), **NEW** (новый),

SCRATCH (случайный) или

UNKNOWN (неизвестный)

ACCESS SEQUENTIAL (последовательный) или

DIRECT (прямой)

FORM FORMATTED (форматизованный) или

UNFORMATTED (неформатизованный)

RECL длина записи, если файл с прямым доступом

BLANK определяет обработку пробелов; либо **NULL**, либо **ZERO**

CLOSE Этот оператор ликвидирует связь между периферийным устройством и файлом. За словом **CLOSE** следует заключенный в скобки список следующих спецификаторов:

UNIT = номер устройства для хранения файла

IOSTAT целочисленная переменная, описывающая состояние устройства ввода-вывода

ERR метка оператора, которому следует передать управление в случае возникновения ошибки

STATUS (статус) **KEEP** (сохранить) или

DELETE (удалить)

INQUIRE Этот оператор возвращает информацию об атрибутах файла. Информация о файле возвращается в виде **"YES"** (да), **"NO"** (нет) или **"UNKNOWN"** (неизвестный); для других спецификаторов информация представляется в виде **.TRUE.**

(истина) или **.FALSE.** (ложь). Помимо спецификаторов **UNIT=**, **IOSTAT** и **ERR** в операторе **INQUIRE** используются следующие:

EXIST .TRUE., если файл существует
OPENED .TRUE., если открыт
NUMBER число периферийных устройств, связанных с файлом
NAMED .TRUE., если файл имеет имя
NAME возвращает имя файла
ACCESS возвращает **"SEQUENTIAL"** или **"DIRECT"**
FORM возвращает информацию о том, является ли файл форматизованным или нет
FORMATTED возвращает „да” или „нет”
UNFORMATTED возвращает „да” или „нет”
RECL возвращает длину записи в файле
NEXTREC возвращает номер следующей записи в файле с прямым доступом
BLANK возвращает информацию о том, какие знаки используются — пробелы или нули

REWIND Перемотка ленты к началу файла

BACKSPACE Обратное перемещение на одну запись по файлу на ленте

ENDFILE Запись на ленту признака конца файла

15.19. ВВОД-ВЫВОД, УПРАВЛЯЕМЫЙ СПИСКОМ (ПОТОКООРИЕНТИРОВАННЫЙ)

Данные в Фортране могут вводиться и выводиться в заранее установленном формате. Такой ввод-вывод называется управляемым списком, или потокоориентированным. Он осуществляется с помощью операторов **READ**, **PRINT** и **WRITE**, в которых на месте идентификатора формата ставится звездочка (*). Операторы **READ**, **PRINT** и **WRITE** выполняют следующие функции:

READ Вводит данные и присваивает их переменным

PRINT Печатает значения переменных, указанных в списке

WRITE Аналогично **PRINT**, но может также выводить данные в память и на внешние устройства

Операторы ввода-вывода, управляемого списком, имеют следующую форму:

READ* переменная 1, переменная 2, ...

PRINT* переменная 1, переменная 2, ...

WRITE* переменная 1, переменная 2, ...

Для каждого компилятора существуют свои собственные правила, определяющие способ разделения данных, максимальное число цифр и т. д. Эту информацию читатель может найти в руководстве по эксплуатации используемой им системы.

15.20. ФОРМАТИЗОВАННЫЙ ВВОД-ВЫВОД

С помощью оператора **FORMAT** можно задать форму, в которой данные будут считываться в вычислительную систему или выводиться из нее. Оператор **FORMAT** должен использоваться вместе с оператором **READ**, **WRITE** или **PRINT**. Операторы форматизованного ввода-вывода записываются следующим образом:

READ (n_1, n_2) список переменных
WRITE (n_1, n_2) список переменных
PRINT n_2 список переменных

где n_1 — номер устройства ввода-вывода, которое должно быть использовано, n_2 — номер строки оператора **FORMAT**, а *список переменных* задает те переменные, значения которых должны быть введены или выведены в нужном формате. В некоторых вычислительных системах при использовании оператора **READ** можно не указывать номер устройства ввода-вывода. Например, в результате выполнения оператора

WRITE (1,700) A,B,C

значения переменных **A**, **B** и **C** будут выведены на устройство, обозначенное знаком 1, в формате, указанном в операторе **FORMAT**, имеющем номер строки 700. Оператор формата имеет следующую форму:

номер строки **FORMAT** (спецификации)

где *номер строки* удовлетворяет обычным требованиям Фортрана, а *спецификации*, указанные в скобках, отделяются друг от друга запятыми. Для спецификаций используются следующие символы:

A символьные значения
B значения с удвоенной точностью
E показательная форма вещественных значений
F вещественные значения
G общая форма
H символьные константы
I целые значения
L логические значения

P масштабный коэффициент, используемый со спецификациями **B**, **E**, **F** и **G**, для того чтобы сдвинуть десятичную точку и изменить порядок при выводе значения

S восстанавливает режим печати знака "+", принятый в компиляторе по умолчанию

SP печать знака "+" со всеми последующими положительными данными

SS подавление знака "+" для всех последующих положительных данных

TL следующий знак указывает число пропускаемых позиций слева от текущей позиции при вводе или выводе

TR следующий знак указывает число пропускаемых позиций справа от текущей позиции при вводе или выводе

X пропуск указанного числа позиций

: завершение управления форматом, если в списке больше нет данных

/ пропуск записи

(Следующие спецификации используются только для ввода)

BN определяет, что знаки пробела должны игнорироваться

BZ определяет, что все пробелы должны обрабатываться как нули

Ниже описаны формы спецификаций для четырех наиболее общих типов данных — целых, вещественных, символьных и показательных.

Целое Iw

Вещественное Fw.d

Символьное Aw

Показательное Ew.d

где **w** указывает число позиций, отводимых для значения при вводе или выводе, **.** указывает десятичную точку, а **d** — число позиций справа от десятичной точки. Спецификации формата для других типов значений могут быть определены с помощью символов формата аналогично тому, как это делалось при описании спецификаций для целых, вещественных, символьных и показательных значений.

Как отмечалось выше, операторы **FORMAT** должны использоваться вместе с операторами **READ**, **WRITE** или **PRINT**. Предположим, что при вводе потребовалось выделить пять позиций для целого числа и десять позиций для вещественного числа с семью знаками справа от десятичной точки. Это может быть сделано с помощью следующей пары операторов:

READ (1, 750) INT, REALVR

750 FORMAT (I5, F10.7)

Знак **1** в операторе **READ** указывает устройство ввода (например, терминал или устройство ввода с перфокарт); **750** — это номер оператора **FORMAT**, а **INT** и **REALVR** — имена переменных, которым должны быть присвоены вводимые значения. Спецификация **I5** в операторе **FORMAT** указывает, что для переменной **INT** будет выделено пять позиций (называемых *полями*), а спецификация **F10.7** указывает, что для переменной **REALVR** будет выделено десять позиций, семь из которых — справа от десятичной точки.

В этом примере для десятичной точки позиция не отводится, т. е. три цифры могут быть слева от десятичной точки и семь цифр — справа от нее. Это относится только к вводу данных. Если бы такой же оператор **FORMAT** использовался с оператором **WRITE** или **PRINT**, то для значений переменной **REALVR** были бы отведены только две позиции слева от десятичной точки и семь — справа от нее.

Если при выводе целая часть числа занимает не все отведенные для нее позиции, то все неиспользованные позиции слева заполняются пробелами. Если дробная часть числа занимает не все отведенные для нее позиции, то все неиспользованные позиции справа заполняются нулями. Если целая часть занимает больше позиций, чем для нее отведено, то возникает ошибка и выводится строка, состоящая из звездочек. Если дробная часть числа занимает больше позиций, чем для нее отведено, то она усекается и округляется в соответствии с отводимым для нее числом позиций.

При вводе нет необходимости оставлять интервалы между данными, расположенными в одной строке. Оператор **FORMAT** “разделит” строку в соответствии с указанными в нем спецификациями. В предыдущем примере первые пять знаков строки считались бы как целое число, а следующие десять — как вещественное. Если строка входных данных выглядела бы, например, как **1 2 3 4 5 6 7 8 9 0 1 2 3 4 5**, то переменным **INT** и **REALVR** присвоились бы значения **1 2 3 4 5** и **6 7 8 9 0 1 2 3 4 5** соответственно. Однако если спецификация поля не будет согласована с типом данного, которое должно быть введено или выведено, то возникнет ошибка.

Спецификация поля может быть повторена с помощью помещаемого перед ней целого числа, указывающего число повторений. Например, спецификация **3I5** определяет три поля по пять знаков каждое для целочисленных данных.

Заголовки и описательные пометки добавляются с помощью оператора **FORMAT** путем записи соответствующих символов, заключенных в кавычки, как части спецификации. Если при выводе требуется выделить пять позиций для целого числа и добавить описательную пометку, то это может быть сделано сле-

дующим образом:

```
WRITE (1, 725) INT  
725 FORMAT ('ОТВЕТ:', I5)
```

В Фортране имеются символы спецификации, предназначенные для управления кареткой печатающего устройства и других периферийных устройств вывода, однако эти символы меняются от системы к системе.

15.21. ЗАРЕЗЕРВИРОВАННЫЕ СЛОВА

В Фортране ни одно слово, используемое как оператор, имя функции или команда, не должно использоваться в качестве имени переменной или константы.

16

ПАСКАЛЬ

Г. Хелмс

16.1. ВВЕДЕНИЕ

Язык Паскаль назван в честь математика XVII века Блеза Паскаля. Язык разработан профессором Технического университета в Цюрихе (Швейцария) Никлаусом Виртом. Профессор Вирт опубликовал первое сообщение о языке Паскаль в 1971 г.

Язык Паскаль основан на языке Алгол. Как преподаватель профессор Вирт хорошо понимал, что первый изучаемый студентом язык программирования оказывает большое влияние на формирование его стиля программирования. Поэтому при разработке языка Паскаль большое внимание уделялось вопросу хорошего стиля программирования. В результате язык Паскаль способствует написанию хорошо структурированных программ и дает возможность повысить производительность труда программиста.

Так же как и Алгол, язык Паскаль является языком с блочной структурой. Программа на языке Паскаль состоит из *блоков*, каждый из которых выполняет определенную функцию. Кроме того, для каждого блока четко указаны его начало и конец. Это позволяет легко модифицировать программу, поскольку изменения в одном блоке не влекут за собой изменений в других блоках.

Кроме стандартного языка Паскаль, существует много его диалектов, два из которых используются наиболее часто. Один из них, получивший название *UCSD-Паскаль*, был разработан сотрудником Калифорнийского университета в Сан-Диего Кеннетом Боулезом. UCSD-Паскаль является подмножеством стандартного языка Паскаль¹⁾ и предназначен главным образом для мини- и микроЭВМ. Другим широко используемым диалектом языка Паскаль является *Apple-Паскаль для персонального компьютера Apple*, разработанного фирмой Apple Computer Company. По существу, Apple-Паскаль является модификацией

¹⁾ UCSD-Паскаль является на самом деле расширением стандартного языка Паскаль, так как, имея все средства стандартного языка, он содержит и ряд расширений. — *Прим. ред.*

UCSD-Паскаля, однако он предоставляет пользователю дополнительные средства, связанные с особенностью аппаратной части микроЭВМ Apple, например средства графического отображения. Здесь мы ограничимся рассмотрением стандартного языка Паскаль.

16.2. СТРУКТУРА ПРОГРАММЫ

Программа на языке Паскаль имеет следующую структуру:

```
PROGRAM имя (INPUT,OUTPUT);  
BEGIN  
    .....  
    Операторы, каждый из которых должен  
    заканчиваться точкой с запятой (;)  
    .....  
END.
```

Заголовок программы, начинающийся словом **PROGRAM**, задает *имя*, или *идентификатор*, программы. (**INPUT,OUTPUT**) являются параметрами программы. Вопрос о параметрах программы будет рассмотрен позже. Сейчас достаточно лишь отметить, что любая программа, получающая входные данные или вырабатывающая выходные, должна иметь параметры. В конце заголовка программы **PROGRAM** ставится точка с запятой.

После заголовка программы **PROGRAM** следуют описания констант и переменных, используемых в программе (более подробно эти описания будут рассмотрены позже). Операторы программы заключены между словами **BEGIN** и **END**. После **BEGIN** не ставится никаких знаков; после **END** ставится точка (.) вместо точки с запятой.

Операторы программы, каждый из которых заканчивается точкой с запятой, располагаются между словами **BEGIN** и **END**. Они выполняются в порядке их следования.

В программу могут быть включены невыполняемые комментарии, которые помещаются между символами (* и *). Например,

(*Пример написания комментария*)

Комментарии не влияют на выполнение программы и игнорируются компилятором. Они используются для указания основных секций программы и пояснения выполняемых действий. Комментарии особенно полезны в тех случаях, когда написанная программа будет использоваться другими программистами, а также когда программу необходимо исправить или модифицировать,

Пробелы в программе не влияют на ее выполнение. Поэтому, для того чтобы сделать программу удобочитаемой, опытные программисты оставляют в нужных местах пробелы и смещают операторы относительно друг друга.

16.3. ИДЕНТИФИКАТОРЫ

Примером идентификатора является имя, следующее за словом **PROGRAM**. Идентификаторы используются также для обозначения констант, переменных и различных частей программы (например, процедур). Идентификаторы должны начинаться с буквы, за которой должны следовать буквы или цифры. Пробелы или другие символы не допускаются. Идентификаторы могут содержать в себе зарезервированные слова языка Паскаль, но не должны совпадать с ними. Идентификаторы могут иметь произвольную длину.

16.4. ТИПЫ ДАННЫХ

Одно из основных достоинств языка Паскаль заключается в возможности определения типов данных. Кроме типов данных, имеющихся в языке Паскаль, можно также определять и использовать другие типы данных, которые нужны программисту.

В языке Паскаль имеются следующие типы данных:

BOOLEAN: Тип данных, принимающих логические значения **TRUE** и **FALSE**.

CHAR: Тип данных, принимающих символьные значения.

INTEGER: Тип данных, принимающих целочисленные значения. Данные этого типа в мини- и микроЭВМ, как правило, могут принимать значения от -32767 до $+32767$, а в больших ЭВМ — от -2147483647 до $+2147483647$.

REAL Тип данных, принимающих вещественные значения. Диапазон их изменения и точность определяются используемой вычислительной системой.

Вопрос определения других типов данных будет рассмотрен позже.

16.5. ОПРЕДЕЛЕНИЯ И ОПИСАНИЯ

Константы и переменные обозначаются с помощью идентификаторов. Однако, до того как константы и переменные и соответствующие им идентификаторы будут использованы в Паскаль-программе, они должны быть определены или описаны.

Определения констант и описания переменных расположены непосредственно за заголовком программы **PROGRAM**, но рань-

ше любого выполняемого оператора. В этом случае они называются глобальными. Глобальные константы и переменные могут использоваться в любом месте программы. Определения констант и описания переменных могут также находиться внутри различных блоков Паскаль-программы. Такие определения и описания называются локальными и используются внутри соответствующего блока. В различных блоках для локальных определений и описаний могут использоваться одни и те же идентификаторы, хотя это не отвечает требованиям хорошего стиля программирования.

Определения констант всегда должны предшествовать всем описаниям переменных. Определяемые константы отделяются друг от друга точкой с запятой. Определениям констант должно предшествовать слово **CONST**, например:

```
CONST
    MILE = 5280;
    FOOT = 12;
    METER = 39.36;
```

Здесь определены две целочисленные константы, **MILE** и **FOOT**, и одна вещественная константа, **METER**. После того как константа определена, ни ее тип, ни значение изменяться не могут.

Язык Паскаль содержит идентификатор **MAXINT** встроенной константы, соответствующей наибольшему целому числу, которое может обрабатываться на данной вычислительной системе. Наибольшее по абсолютной величине отрицательное целое число обозначается как **-MAXINT**, а наибольшее положительное целое число — как **+MAXINT**.

Каждая переменная, используемая в Паскаль-программе, должна быть описана в разделе описания переменных, начинающемся словом **VAR**, которое связывает идентификаторы с переменными и определяет их тип. Например:

```
VAR
    COUNT, TOTAL: INTEGER;
    TEMPERATURE: REAL;
    RESULTS: BOOLEAN;
```

Этот оператор показывает, что идентификаторы **COUNT** и **TOTAL** служат для обозначения целочисленных переменных, идентификатор **TEMPERATURE** — для обозначения вещественной переменной и **RESULTS** — для переменной, принимающей булевы значения. После того как переменная описана, ее тип не может изменяться.

Язык Паскаль допускает определение и использование типов данных, отличных от типов **BOOLEAN**, **CHAR**, **INTEGER** и **REAL** (в действительности **BOOLEAN**, **CHAR**, **INTEGER** и

REAL можно рассматривать как встроенные типы данных). Для определения новых типов данных используется раздел определения типов, начинающийся словом **TYPE** и имеющий следующую форму:

TYPE

COLORS = (RED, WHITE, YELLOW, BLUE)

Описания **TYPE** располагаются между описаниями **CONST** и **VAR**. Все переменные определяемого типа должны быть указаны в разделе описания переменных.

В каждом блоке программы должно содержаться только по одному описанию **CONST**, **TYPE** и **VAR**.

16.6. МАССИВЫ

Массив в языке Паскаль определяется как структура данных с фиксированным числом элементов одного и того же типа, для обращения к которым используется один и тот же идентификатор. Массив описывается с помощью зарезервированного слова **ARRAY** и имеет следующую форму:

идентификатор: **ARRAY** [интервал] **OF** тип

где *идентификатор* удовлетворяет перечисленным ранее правилам, интервал задает границы массива, а *тип* — это тип элементов массива. Интервал содержит нижнюю и верхнюю границы массива, разделенные двумя точками. Описание

SUMTOTAL:ARRAY[1..10] OF INTEGER

задает массив с элементами **SUMTOTAL[1],SUMTOTAL[2],...,SUMTOTAL[10]**. Число в квадратных скобках называется индексом. При обращении к элементу, выходящему за границы, определенные интервалом, выполнение программы аварийно завершается.

Массивы могут иметь более одного индекса, или измерения. Форма:

идентификатор **ARRAY** [интервал] **OF** **ARRAY** [интервал] **OF** тип

16.7. ОПЕРАТОРЫ ПРИСВАИВАНИЯ

Операторы присваивания вычисляют значения и присваивают их переменной. Значения переменным присваиваются в соответствии с описанием этих переменных. Операторы присваивания задают новые значения переменным и записываются с помощью следующих знаков:

:=	Знак присваивания, похожий на знак “=” в алгебре
+	Сложение
—	Вычитание и унарный минус
*	Умножение
/	Деление, результат которого — вещественное число; операнды могут быть вещественными или целыми
DIV	Деление, результат которого есть целое число; оба операнда должны быть целыми
MOD	Вычисление остатка от целочисленного деления

Операторы присваивания выполняются в порядке их следования слева направо. Если в одном операторе присваивания содержится несколько знаков операций, то они выполняются в следующем порядке:

1. Операции в скобках
2. Унарный минус
3. Умножение и деление
4. Сложение и вычитание

16.8. ОПЕРАТОРЫ ОТНОШЕНИЯ

Отношения между выражениями в языке Паскаль записываются с помощью следующих операторов отношения:

=	Равно
<>	Не равно
>	Больше
>=	Больше или равно
<	Меньше
<=	Меньше или равно

В языке Паскаль имеются также три оператора отношения¹⁾, у которых тип операндов может быть только булевым. Эти операторы следующие:

AND	Предложение истинно, если оба выражения истинны
OR	Предложение истинно, если хотя бы одно выражение истинно
NOT	Предложение истинно, если значение операнда есть ложь

16.9. УПРАВЛЯЮЩИЕ ОПЕРАТОРЫ

Язык Паскаль содержит несколько операторов, предназначенных для управления выполнением программы. Эти операторы можно разбить на две большие группы. В первую группу входят операторы, позволяющие повторять некоторое действие нужное число раз. Операторы второй группы проверяют, нуж-

¹⁾ Обычно операторы **AND**, **OR** и **NOT** относят не к классу операторов отношения, а к классу логических операторов. — *Прим. ред.*

но ли выполнять указанное действие, или определяют, какое из действий следует выполнить.

Одним из операторов, позволяющих повторять требуемое действие, является оператор **WHILE**, который используется вместе с **DO**. Он имеет следующую форму:

WHILE логическое выражение **DO**
оператор

При каждом повторении действия вычисляется логическое выражение. До тех пор пока его значение является **TRUE**, выполняется оператор, следующий за словом **DO**. Если значением выражения является **FALSE**, то этот оператор не выполняется; вместо него выполняется следующий за ним оператор.

Часто *оператор*, выполнение которого должно повторяться, в действительности состоит из нескольких строк программы. В этом случае оператор **WHILE** имеет следующую форму:

WHILE логическое выражение
BEGIN
операторы (разделенные точкой с запятой)
END

Отметим, что в этом случае после **BEGIN** и **END** нет никаких знаков. Кроме того, нет знака и после строки, непосредственно предшествующей слову **END**.

Другим оператором, позволяющим повторять действие, является оператор **REPEAT**, который используется вместе с **UNTIL**. Он имеет следующую форму:

REPEAT
оператор
UNTIL логическое выражение

В этом случае выполняется оператор, а затем вычисляется логическое выражение, следующее за словом **UNTIL**. Если значением выражения является **FALSE**, оператор выполняется вновь. Оператор будет выполняться до тех пор, пока значение выражения не станет равным **TRUE**.

Так же как и в операторе **WHILE**, *оператор*, расположенный между словами **REPEAT** и **UNTIL**, может состоять из нескольких строк программы, разделенных точкой с запятой. Однако в отличие от оператора **WHILE** эти строки не обязательно помещать между словами **BEGIN** и **END**. Последняя строка перед словом **UNTIL** не должна содержать точки с запятой.

Язык Паскаль содержит также условную конструкцию **IF...THEN...ELSE**. Одна из ее форм следующая:

IF логическое выражение
THEN оператор

В этом примере вначале вычисляется логическое выражение. Если его значением является **TRUE**, выполняется оператор, следующий за словом **THEN**. Если значение выражения есть **FALSE**, то оператор, следующий за словом **THEN**, не выполняется и управление передается следующей строке программы.

Другая форма этой конструкции следующая:

```
IF логическое выражение
  THEN первый оператор
  ELSE второй оператор
```

Вначале вычисляется логическое выражение. Если его значением является **TRUE**, выполняется оператор, следующий за словом **THEN**. Если значение есть **FALSE**, выполняется оператор, следующий за словом **ELSE**.

С помощью оператора **CASE** осуществляется выбор среди нескольких различных действий. Он имеет следующую форму:

```
CASE селекторное выражение OF
  метка: оператор;
  метка: оператор;
  метка: оператор;
  .....
  метка: оператор;
END
```

При выполнении оператора **CASE** вычисляется селекторное выражение, значением которого является целое число, или номер. Метки, которые предшествуют каждому из операторов, следующих за **CASE...OF**, представляют собой целое число, или номер. При совпадении значения селекторного выражения со значением некоторой метки выполняется оператор, следующий за этой меткой.

В языке Паскаль имеется также оператор **GOTO**. Это единственный оператор языка Паскаль, в котором требуется указание номера строки. Он имеет следующую форму:

```
GOTO номер строки
номер строки оператора
```

При выполнении оператора **GOTO** осуществляется непосредственная передача управления оператору, расположенному в начале строки с номером, указанным после слова **GOTO**. (Оператор **GOTO** часто используют вместе с оператором **IF...THEN**.)

Все номера строк, используемые в операторах **GOTO**, должны быть указаны в разделе описания меток, начинающемся словом **LABEL** и имеющем следующую форму:

```
LABEL номер строки
```

Описания **LABEL** следуют за заголовком **PROGRAM** и предшествуют описаниям **CONST** и **VAR**.

При программировании на языке Паскаль следует избегать применения оператора **GOTO**. Он должен использоваться только в тех случаях, когда нет другого способа составления программы¹⁾.

16.10. ФУНКЦИИ И ПРОЦЕДУРЫ

Функцией называется совокупность действий, которая в случае необходимости может быть вызвана при выполнении программы. Функция получает одно или более заданных значений и возвращает одно-единственное значение. Процедура аналогична функции, но она может возвращать несколько значений или вообще не возвращать никаких значений. Процедура может *располагаться обособленно* как оператор; функция может быть использована почти во всех случаях, когда могут использоваться константы или переменные. Короче говоря, процедуры можно рассматривать как операторы, а функции — как выражения. И функции, и процедуры должны быть описаны.

Описание функции имеет следующую форму:

```
FUNCTION идентификатор (список параметров): тип;  
  BEGIN  
    операторы  
  END;
```

где *идентификатор* — это обычный идентификатор языка Паскаль, *список параметров* состоит из параметров, используемых в функции, а *тип* определяет тип значения, вычисляемого функцией.

Параметры — это переменные, значения которых не задаются операторами присваивания, а поступают из оператора, вызывающего функцию или процедуру. В одних и тех же скобках вместе со списком параметров обычно содержится другое описание типа, которое задает типы параметров.

В действительности каждая функция или процедура является независимой Паскаль-программой. Переменные и константы в функции или процедуре могут быть описаны и определены точно так же, как и в основной программе. Поскольку каждая функция или процедура не зависит от остальной части программы и других функций и процедур, то идентификаторы используются в каждой функции или процедуре независимо (хотя это не отвечает требованиям хорошего стиля программирования).

¹⁾ Существует строгое доказательство факта, что любая управляющая конструкция программы может быть выражена без **GOTO**. — *Прим. ред.*

Процедуры описываются так же, как и функции, но слово **FUNCTION** заменяется на **PROCEDURE**. Поскольку процедура сама не передает значения, то в описании **PROCEDURE** нет описания типа.

16.11. ВСТРОЕННЫЕ ФУНКЦИИ

Язык Паскаль содержит следующие встроенные функции:

ABS Результат — абсолютное значение выражения.

ARCTAN Результат — вещественное значение, равное арктангенсу (в радианах) выражения.

CHR Результат — литера (тип **CHAR**), имеющая в используемом коде порядковый номер, заданный аргументом.

COS Результат — косинус выражения, заданного в радианах.

EOF Результат — **TRUE**, если достигнут конец файла, и **FALSE** в противном случае.

EXP Результат — число e , возведенное в указанную степень, где e — основание натурального логарифма.

LN Результат — натуральный логарифм выражения.

ODD Результат — **TRUE**, если целое число нечетное, и **FALSE** в противном случае.

ORD Результат — порядковый номер аргумента во множестве значений, элементом которого является аргумент.

PRED Результат — значение, предшествующее значению аргумента.

RESET Инициализирует входной файл для чтения данных.

REWRITE Инициализирует выходной файл для вывода данных.

ROUND Округляет значение вещественного типа в значение целого типа.

SIN Результат — синус выражения, заданного в радианах.

SQR Результат — квадрат выражения.

SQRT Результат — квадратный корень выражения.

SUCC Результат — значение, следующее за значением аргумента.

TRUNC Результат — целое число, полученное отбрасыванием дробной части вещественного числа.

16.12. ВВОД И ВЫВОД

Оператор **READ** в языке Паскаль считывает элемент данных и сохраняет его в том месте памяти, которое соответствует указанной переменной. Оператор имеет следующую форму:

READ (список переменных);

Данные, которые должны быть считаны, размещаются после программы и отделяются друг от друга по крайней мере одним пробелом. Значения данных присваиваются переменным в том порядке, в котором они вводятся. Например, если первым элементом данных является 1, а первым идентификатором в списке переменных — **A**, то переменной **A** присваивается значение, равное 1. Типы вводимых данных должны соответствовать типам переменных в списке. Если для переменных типа **CHAR** считываются числовые данные, то последние будут вводиться и обрабатываться как символьные данные. Однако считывание символьных данных для числовых переменных приведет к возникновению ошибки.

В некоторых случаях желательно считывать данные по одной строке за один раз. Для этой цели используется оператор **READLN**. Оператор **READLN** передает управление вводом следующей строке входных данных при достижении конца списка переменных или конца строки входных данных.

Вывод данных может быть осуществлен с помощью оператора **WRITE**. Его общая форма следующая:

WRITE (список переменных)

При выводе символьной строки с помощью оператора **WRITE** она заключается в апострофы и помещается внутри скобок, которые следуют за словом **WRITE**. Если требуется вывести апостроф, в символьную строку должна быть помещена пара апострофов.

При выводе данных можно указать требуемое число позиций (или поле), отводимых под каждый элемент данных. Для этого после идентификатора элементарного значения указывается целое число, равное требуемому количеству позиций в поле. Это число отделяется от идентификатора двоеточием. Например, для того чтобы элементарному значению, представленному переменной **SUM**, отвести поле длиной 5, надо записать **SUM:5**.

Управление выводом осуществляется также с помощью оператора **WRITELN**. Он аналогичен оператору **WRITE**, за исключением того, что управление выводом передается следующей строке всякий раз, когда выходной список, следующий за словом **WRITELN**, исчерпан.

16.13. УПАКОВАННЫЕ МАССИВЫ

Более эффективное использование памяти для хранения данных достигается с помощью упакованных массивов. Упакован-

ный массив определяется с помощью описания следующего вида:

идентификатор: **PACKED ARRAY** [интервал] **OF** тип

Использование упакованных массивов не влияет на результаты работы программы. Однако выполнение программы замедляется, поскольку увеличивается время доступа к отдельным элементам упакованного массива.

16.14. МНОЖЕСТВА

Средства языка Паскаль позволят использовать в программировании математическое понятие множества. Множество задается с помощью определения

SET OF тип

где *тип* должен быть порядковым. Таким образом, множества вещественных чисел или строк не допускаются. Кроме того, множества должны обрабатываться как единое целое, поскольку в языке Паскаль нет операций разбиения множества на отдельные элементы.

Язык Паскаль содержит следующие операторы для работы с множествами:

*	Пересечение
+	Объединение
—	Вычитание

В языке Паскаль имеется также оператор сравнения **IN**. Его типичная форма следующая:

IF переменная **IN** [множество] **THEN** оператор

Оператор **IN** может использоваться вместе с другими условными операторами и операторами отношения.

16.15. ФАЙЛЫ И ЗАПИСИ

Файл — это структура данных, состоящая из элементов одного типа. Файлы в языке Паскаль являются последовательными. Для добавления, удаления и исследования элементов необходимо пройти по всему файлу. Файлы описываются следующим образом:

идентификатор: **FILE OF** тип

При генерации файла запись компонент в него осуществляется поочередно, по одной компоненте. Подготовка файла к записи осуществляется с помощью встроенной процедуры

REWRITE (файл)

которая стирает все компоненты файла. Для добавления элементов используется процедура **WRITE**, вместе с именем которой в скобках указываются идентификатор файла и требуемая компонента. Идентификатор файла и компонента должны отделяться друг от друга запятой.

Файл может быть подготовлен к чтению с использованием встроенной процедуры **RESET**, за именем которой указывается имя файла, заключенное в скобки. Эта процедура подготавливает файл к чтению. Затем файл может быть прочитан с помощью процедуры **READ**, за именем которой следует имя файла и имя переменной, взятые в скобки.

Описание (**INPUT**, **OUTPUT**), следующее за заголовком **PROGRAM**, определяет файлы для ввода и вывода. Эти файлы называются внешними. За заголовком **PROGRAM** могут быть также указаны дополнительные внешние файлы, но они должны быть описаны в программе как переменные. Файлы, используемые не для ввода-вывода, а для других целей, называются внутренними.

Записью в языке Паскаль называется структура данных, которая может иметь элементы различных типов и допускает прямой доступ к ним. Общая форма описания записи следующая:

RECORD

Идентификатор (ы): типы

END

Идентификаторы и типы называются **полями записи**.

Доступ к отдельным компонентам осуществляется с помощью оператора **WITH** точно так же, как если бы они были обычными переменными. Форма:

WITH переменная записи **DO** оператор

Необходимость в операторе **WITH** отпадает, если ввести требование, согласно которому идентификаторы полей переменных записи должны быть составными. Это сокращает время программирования, поскольку обычно элементы записи обрабатываются несколькими операторами.

16.16. ЗАРЕЗЕРВИРОВАННЫЕ СЛОВА

AND	NIL
ARRAY	NOT
BEGIN	OF
CASE	OR
CONST	PACKED
DIV	PROCEDURE
DO	PROGRAM
DOWNTO	RECORD
ELSE	REPEAT
END	SET
FILE	THEN
FOR	TO
FUNCTION	TYPE
GOTO	UNTIL
IF	VAR
IN	WHILE
LABEL	WITH
MOD	

А. Таккер-мл.

17.1. ВВЕДЕНИЕ

В последние годы ПЛ/1 привлек внимание многих программистов и получил широкое распространение. Этому послужил ряд причин. Во-первых, он является более современным по сравнению с большинством существующих языков и поэтому тесно связан с современными ЭВМ и решаемыми на них задачами. Во-вторых, ПЛ/1 является языком широкого профиля и может применяться при решении научных задач, задач обработки данных и текстов, а также в системном программировании. В-третьих, компиляторы с ПЛ/1 имеются на многих вычислительных машинах. Они эффективны и надежны. В этой главе мы дадим описание языка ПЛ/1 и покажем, насколько он эффективен при решении научных задач и задач обработки данных и текстов.

Краткая история создания языка ПЛ/1

Первая версия ПЛ/1 была названа языком НПЛ (New Programming Language, т. е. новый язык программирования). Впервые он был реализован в 1965 г. на вычислительной машине IBM и получил название ПЛ/1.

В 60-х годах ПЛ/1 не получил широкого признания. Первые компиляторы с ПЛ/1 были неэффективными и ненадежными. Хотя он и предоставлял множество разнообразных средств программирования, его не считали практически приемлемым ни в одной области. Однако реализация на IBM сделала ПЛ/1 реально используемым языком.

В феврале 1975 г. техническим комитетом X3J1-PL/1 американского национального комитета по стандартам совместно с техническим комитетом TC10-PL/1 европейской ассоциации изготовителей ЭВМ был опубликован проект стандартного языка

ПЛ/1. В настоящей главе рассматриваются основные элементы этой версии языка.

Касаясь характеристик ПЛ/1, следует сравнить его с более ранними языками. Основные особенности ПЛ/1 присущи его предшественникам — Алголу, Фортрану и Коболу. Действительно, если представить себе язык, сочетающий синтаксическую структуру и динамическое распределение памяти Алгола, структуры записей и средства ввода-вывода Коболу и арифметические операции Фортрана, а также располагающий некоторыми дополнительными средствами обработки строк и списков и средствами прерывания-захватывания, этот язык будет очень напоминать ПЛ/1.

Реализации и модификации ПЛ/1

Поскольку ПЛ/1 впервые был реализован еще в 1965 г., его компиляторы разработаны для ряда вычислительных машин и операционных систем, например: Burroughs — B6700 System, CDC — Cyber Series, Honeywell — Multics System, IBM — 360, 370 Series.

ПЛ/1 используется не столь широко, как Фортран или Кобол, по следующим причинам.

1. ПЛ/1 пока еще нестандартизован.
2. Затраты на проектирование и эксплуатацию компилятора с ПЛ/1 значительно превосходят соответствующие затраты для специальных языков.

Помимо компиляторов, поставляемых изготовителями ЭВМ, имеется ряд независимо спроектированных компиляторов с ПЛ/1. Они отличаются друг от друга по степени реализации средств программирования ПЛ/1. Наиболее известным из них, по-видимому, является компилятор PL/C, который был спроектирован в Университете г. Корнелл.

Основные области применения ПЛ/1

В отличие от других языков, описанных в этой книге, ПЛ/1 является языком широкого профиля. При его разработке ставилась цель создать такой язык, который с одинаковым успехом применялся бы при решении научных задач, при обработке данных и текстов, а также в системном программировании. И в каждой из этих областей он действительно нашел широкое применение. Однако ни в одной из этих областей ПЛ/1 не заменил основной язык (Фортран, Кобол, Снобол и язык ассемблера соответственно).

17.2. НАПИСАНИЕ ПЛ/1-ПРОГРАММ

ПЛ/1-программа — это последовательность операторов, которые описывают алгоритм. Благодаря своей широте ПЛ/1 предоставляет большое разнообразие типов операторов и функциональных средств. Мы рассмотрим только те из них, которые, на наш взгляд, являются наиболее широко используемыми при решении научных задач и задач обработки данных и текстов. В частности, мы не будем останавливаться на особенностях компиляции и многозадачном режиме в ПЛ/1. Не будет также рассмотрен и ряд встроенных функций, условий прерывания и опций операторов. Предлагаемое здесь описание ПЛ/1 включает наиболее важные, на наш взгляд, элементы языка, но не является упрощенным его вариантом.

Типы данных и константы

Строго говоря, в ПЛ/1 существуют два типа данных: **проблемные данные** и **данные, управляющие задачей**. К первому типу относятся объекты, которые программист определяет как данные (символьные строки, числа и другие элементы, используемые в программе как входные, выходные и рабочие значения). Данные, управляющие программой, включают такие элементы, как метки операторов и указатели (которые соединяют узлы связанного списка).

Проблемные данные в ПЛ/1 разбиваются на два класса: **арифметические** и **строковые**. Арифметические данные — это числа, и как таковые они имеют следующие атрибуты:

Атрибут	Опции
основание	<u>DECIMAL</u> или <u>BINARY</u>
тип	<u>REAL</u> или <u>COMPLEX</u>
формат	<u>FIXED</u> или <u>FLOAT</u>
точность	(p, q) или (p)

Перечисленные здесь **опции** написаны заглавными буквами, с тем чтобы показать, что они являются элементами языка ПЛ/1. Кроме того, многие из этих **ключевых слов** ПЛ/1 могут быть сокращены. Сокращение ключевого слова указывается подчеркиванием соответствующих букв. Например, слово **COMPLEX** может быть также записано как **CPLX**, **DECIMAL** — как **DEC** и т. д.

Арифметические данные типа **REAL FIXED DECIMAL** — это не что иное, как обычные десятичные числа с дробной частью или без нее. Например,

1.73 0 —17 250

Данные типа **REAL FIXED BINARY** — это двоичные числа с дробной частью или без нее, справа от которых пишется буква "В". Например,

1.01В 0В —11В 111110В

Точность арифметических данных типа **REAL FIXED (DECIMAL или BINARY)** записывается в форме (p, q) , где p и q — положительные целые числа, указывающие общее число цифр в числе и число цифр в его дробной части соответственно. Например, константа 1.73 типа **REAL FIXED DECIMAL** имеет точность $(3, 2)$, как и константа 1.01В типа **REAL FIXED BINARY**. Максимальная точность для арифметических данных типа **FIXED** зависит от реализации.

Арифметическое значение типа **REAL FLOAT DECIMAL** — это обычное десятичное число, за которым следует показательная часть, означающая умножение на степень числа 10. Например, десятичное число 105 000 может быть записано в форме **REAL FLOAT DECIMAL** как 1.05Е5. Такая запись соответствует числу 1.05×10^5 , или 105 000. Аналогично число типа **REAL FLOAT BINARY** записывается как двоичное число, за которым следует показательная часть, означающая умножение на степень числа 2. Например, двоичное число 101 000 может быть записано в форме **REAL FLOAT BINARY** как 1.01Е5В, что означает 1.01×2^5 (здесь точка — это не десятичная, а двоичная точка). Символ В указывает, что значение является не десятичным, а двоичным.

Точность для арифметических данных типа **REAL FLOAT (DECIMAL или BINARY)** всегда задается в форме (p) , где p — положительное целое число, указывающее общее количество значащих цифр (десятичных или двоичных). Например, константа 1.05Е5 типа **REAL FLOAT DECIMAL** имеет точность (3) , так как она содержит 3 значащие цифры (предшествующие показательной части). Аналогично число 1.01Е5В имеет точность (3) . Точность, с которой значение типа **FLOAT** действительно хранится в памяти, зависит от реализации.

Арифметическое значение типа **COMPLEX** состоит из двух частей — вещественной и следующей за ней мнимой частью. Мнимая часть обозначается с помощью буквы I, которая приписывается к ней справа. Хотя в ПЛ/1 определены все действия с комплексными числами, в этой главе мы не будем их рассматривать.

Данные с атрибутом **FLOAT** обычно используются при решении научных задач вычислительного характера, так как эти данные имеют широкий диапазон значений. В задачах обработки данных обычно предпочитают использовать данные типа **FIXED DECIMAL**, поскольку они точно представляют десятичные дроби. Например, число 10.53 будет переведено в двоичную форму, если оно хранится как **FLOAT**, и в результате этого произойдет потеря точности (т. е. оно могло бы быть приблизительно представлено как 10.5299). Однако если это число хранится как **FIXED DECIMAL**, то оно не переводится в двоичную форму и потери точности не происходит. Это имеет важное значение в тех случаях, когда данное число обозначает денежную сумму — 10.53 долл. Наконец, данные типа **FIXED BINARY** обычно предпочитают использовать в целочисленной арифметике, например в качестве счетчика целых чисел, переменной, управляющей циклом и т. д.

Перейдем к рассмотрению другого класса проблемных данных языка ПЛ/1 — строковых данных. Строка в ПЛ/1 может быть либо символьной (обозначенной атрибутом **CHARACTER**), либо битовой (обозначенной атрибутом **BIT**).

Символьная строка — это либо пустая строка, либо последовательность некоторого числа знаков алфавита (который зависит от реализации). В этой главе мы будем предполагать, что алфавит содержит следующие знаки:

_(+ &\$*); — /,%?:#@' = “

ABCDEFGHIJKLMNOPQRSTUVWXYZ

0123456789

Кроме того, будем предполагать, что они упорядочены точно так же, как записаны здесь, т. е. знак “_” меньше, чем “.”, который в свою очередь меньше, чем знак “(”, и т. д. Это означает, что знаки алфавита (A — Z) упорядочены обычным образом, что числовые знаки (0—9) также упорядочены обычным образом и что специальные знаки предшествуют алфавитным, которые в свою очередь предшествуют числовым.

Символьная строка при ее написании в ПЛ/1-программе должна быть заключена в кавычки ('). Ниже приводятся пять примеров символьных строк:

'ABC'

'A—B—C'

“

'WHAT“S—THIS?'

(5) 'ABC'

Первые две строки не эквивалентны, поскольку пробел () является значащим знаком в символьной строке. Третья строка пустая, т. е. она не содержит символов. Четвертый пример иллюстрирует тот случай, когда сама строка содержит кавычку. В этом случае кавычку следует записать два раза подряд для того, чтобы она была синтаксически различимой с открывающей и закрывающей кавычками. Открывающая и закрывающая кавычки не являются частью строки; они просто определяют ее начало и конец. Пятый пример показывает, как можно коротко записать строку, которая является простым повторением более короткой строки. Здесь представлен другой способ записи строки 'АВСАВСАВСАВСАВС'.

Помимо значения символьные строки всегда имеют атрибут длины: целое число знаков, содержащихся в ней (исключая открывающую и закрывающую кавычки). Длины пяти записанных выше символьных строк составляют 3, 5, 0, 12 и 15 соответственно. Отметим, что вхождение кавычки (') в строку увеличивает ее длину только на 1, хотя она и записывается дважды ("). Максимально допустимая длина символьной строки в ПЛ/1 зависит от реализации.

Битовая строка — это либо пустая строка, либо последовательность некоторого числа двоичных цифр (нулей и единиц), которые часто называют битами. При использовании в ПЛ/1-программе битовая строка должна быть заключена в кавычки и сразу же после нее должен быть указан символ В. Ниже приводятся пять примеров битовых строк.

'0'В
'1'В
"В
'01001'В
(3) '01'В

Так же как и символьные строки, битовые строки имеют атрибут длины. Длины записанных выше битовых строк составляют 1, 1, 0, 5 и 6 соответственно. Отметим, что, так же как и для символьных строк, для битовых строк справедливо соглашение о сокращении записи.

При программировании на ПЛ/1 битовые строки широко используются для обработки логических значений (значений истинности). В частности, битовые строки '0'В и '1'В интерпретируются как *ложь* и *истина* соответственно.

Имена, переменные и структуры данных

Переменная в ПЛ/1 — это имя, связанное с одним или более значениями, которые могут изменяться при выполнении программы. Именем переменной может быть любая последователь-

ность алфавитных (A — Z, @, #, \$), числовых (0—9) знаков и знака (-) разбивки, первым из которых должен быть алфавитный знак. Следующие имена переменных являются допустимыми:

X
GROSS-PAY
19

В некоторых реализациях накладывается ограничение на максимальную длину имен переменных.

Если переменная связывается в точности с одной величиной, то она называется **элементарной переменной**. Если она связывается более чем с одной величиной, то эта переменная является **массивом** или **структурой**.

Каждая элементарная переменная в ПЛ/1-программе имеет свой набор атрибутов. Эти атрибуты описывают тип значений, которые данная переменная может принимать при выполнении программы. Предположим, что **X**, **GROSS-PAY** и **TITLE** — это переменные, используемые в программе. Пусть **X** имеет атрибуты **REAL FLOAT DECIMAL** и точность (6). Пусть **GROSS-PAY** имеет атрибуты **REAL FIXED DECIMAL** и точность (7,2). Наконец, пусть **TITLE** имеет атрибут **CHARACTER** и длину (25). Это означает следующее.

1. **X** может принимать любые арифметические значения типа **REAL FLOAT DECIMAL** и иметь 6 или менее значащих десятичных цифр. Например, значением **X** может быть 3.59847E-5.

2. **GROSS-PAY** может принимать любые арифметические значения типа **REAL FIXED DECIMAL**, лежащие в пределах от —99 999.99 до 99 999.99. Например, значением **GROSS-PAY** может быть 23 400.00.

3. **TITLE** может принимать любые символьно-строчные значения и иметь длину 25 знаков. Например, значениями **TITLE** могут быть строки **'1975-ANNUAL-BUDGET-REPORT'** или **'1975-ANNUAL-BUDGET-_____'**.

Атрибуты элементарной переменной могут быть заданы либо явным описанием, либо по умолчанию. Если атрибуты переменной не описываются явно, система назначает их в соответствии с четко определенными правилами.

Для того чтобы явно задать атрибуты переменной, программист использует оператор описания, который может иметь следующую форму:

DECLARE имя атрибута

Здесь *имя* определяет имя переменной, а *атрибут* — это последовательность атрибутов, соответствующих данному имени. Воз-

вращаясь к предыдущему примеру, можно описать переменные с именами **X**, **GROSS-PAY** и **TITLE** следующим образом:

```
DECLARE X FLOAT DECIMAL (6);  
DECLARE GROSS_PAY FIXED DECIMAL (7,2);  
DECLARE TITLE CHARACTER (25);
```

Отметим, что различные части оператора описания отделяются друг от друга по крайней мере одним пробелом. Если эти описания присутствуют в одной и той же программе, они могут быть записаны в более короткой форме: (1) с использованием допустимых сокращений для ключевых слов и (2) с помощью объединения описаний нескольких переменных в одно описание:

```
DCL X FLOAT DEC (6),  
    GROSS_PAY FIXED DEC (7,2),  
    TITLE CHAR (25);
```

В тех случаях, когда несколько переменных имеют одни и те же атрибуты, могут быть использованы сокращения другого вида. Например, если **I**, **J** и **K** имеют атрибуты **FIXED BINARY** и точность (31,0), они могут быть описаны следующим образом:

```
DCL (I,J,K) FIXED BIN (31,0);
```

То есть атрибуты **FIXED BINARY (31,0)** могут быть указаны за скобками, в которые заключены переменные, имеющие все эти атрибуты. Наконец, для переменных с атрибутом **FIXED** точность (p,q), где q = 0, может быть указана в более короткой форме, как (p). Таким образом, (31,0) в предыдущем описании можно заменить на (31).

Как было показано в предыдущем примере, при описании символьно-строчной переменной должны быть указаны атрибут **CHARACTER** и длина **I**. Аналогично при описании битово-строчной переменной должны быть указаны атрибут **BIT** и длина **I**. В каждом из этих случаев значениями переменной могут быть символьные (битовые) строки, число знаков (битов) в которых задается величиной **I**. Например, если описать, что **TITLE** — это переменная с атрибутами **CHARACTER(25)**, то каждое значение, хранимое под именем **TITLE**, должно быть 25-символьной строкой.

Существует другой атрибут, **VARYING**, который позволяет при выполнении программы изменять длину символьно-строчной или битово-строчной переменной, так же как и ее значение. Например, если описать переменную **TITLE** как

```
DCL TITLE CHAR (25) VARYING;
```

то ее значениями могут быть любые символьные строки длиной от 0 до 25 включительно. Так, следующие две строки могут быть значениями переменной **TITLE** в различные моменты вы-

полнения программы:

'1975_ANNUAL_BUDGET_REPORT'

'1975_ANNUAL_BUDGET'

Переменную в программе не всегда нужно описывать явно. Переменная существует в программе с момента использования ее имени в некотором выполняемом операторе независимо от того, была она описана или нет. Кроме того, не все атрибуты описываемой переменной должны быть указаны явно. Если один или более атрибутов описываемой переменной опущены или если переменная вообще не описана, атрибуты назначаются системой (по умолчанию) следующим образом:

1. Переменная не описана или в описании не указан ни один из ее атрибутов. Если ее имя начинается с одной из букв I, J, K, L, M или N, предполагаются арифметические атрибуты

REAL FIXED BIN (15,0).

В противном случае предполагаются арифметические атрибуты

REAL FLOAT DEC (6).

2. Если переменная описана и в описании отсутствует по крайней мере один из ее атрибутов, как, например, арифметический тип, формат и основание, то отсутствующий атрибут назначается из следующего списка:

Отсутствующий атрибут	Назначаемый атрибут
тип	REAL
формат	FLOAT
основание	DECIMAL

3. Если арифметическая переменная описывается без атрибута точности, он назначается следующим образом. (Никакая переменная не может быть описана только лишь заданием атрибута точности. По крайней мере один дополнительный атрибут должен в описании текстуально предшествовать атрибуту точности переменной.)

Формат и основание	Назначаемая точность
FIXED DEC	(5,0)
FIXED BIN	(15,0)
FLOAT DEC	(6)
FLOAT BIN	(21)

4. Если строковая переменная описывается без атрибута длины, ее длина полагается равной 1.

Поэтому в предыдущем примере переменную **X** можно было не описывать. Описание переменных **GROSS_PAY** и **TITLE** было необходимо, поскольку в противном случае они были бы описаны по умолчанию как **REAL FLOAT DEC (6)**. Однако наше описание может быть сокращено следующим образом:

DCL GROSS_PAY FIXED (7,2);

При этом в качестве атрибута типа (**REAL** или **COMPLEX**) по умолчанию выбирается **REAL**, а в качестве атрибута основания (**DEC** или **BIN**) по умолчанию выбирается **DEC**. Это в точности то, что нам нужно.

Решение вопроса о том, описывать ли переменную в программе явно или нет, зависит от стиля программиста, от логической структуры программы, а также от самих правил умолчания.

Существует еще один способ описания арифметических и строковых переменных, а именно с помощью спецификации **PICTURE**. Этот тип описания имеет следующую форму:

DCL имя PICTURE 'спецификация';

Здесь спецификация может принимать разнообразные формы. Мы рассмотрим только две из них.

Символьную переменную **TITLE** из предыдущего примера мы могли бы также описать одним из следующих двух способов:

DCL TITLE PIC 'XXXXXXXXXXXXXXXXXXXXXXXXXXXX';

DCL TITLE PIC '(25)X';

Здесь знак **X** является символьной спецификацией, а число знаков **X** указывает длину строки. Второй способ описания демонстрирует, как можно сократить длинную последовательность знаков **X**.

Рассмотренную в предыдущем примере переменную **GROSS_PAY** с атрибутами **REAL FIXED DECIMAL (7,2)** мы могли бы также описать следующим образом:

DCL GROSS_PAY PIC '99999V99';

Здесь знак **9** — это спецификация десятичной цифры, а число знаков **9** указывает количество цифр, отводимых под данную переменную. Кроме того, расположение знака **V** соответствует расположению десятичной точки. Если знак **V** опущен, то предполагается, что десятичная точка располагается правее крайнего справа знака **9**. Таким образом, это описание эквивалентно

тому, что переменная **GROSS_PAY** имеет атрибуты **REAL FIXED DECIMAL (7,2)**.

Однако эта так называемая спецификация изображения (**PICTURE**) числовых данных предоставляет программисту несколько большие возможности, чем описание в форме **REAL FIXED DECIMAL (7,2)**, поскольку в нее могут быть включены знаки редактирования как, например, . и \$, позволяющие управлять печатью значения переменной. Например, если бы переменная **GROSS_PAY** была описана в форме

DCL GROSS_PAY PIC '\$99999V.99';

имела значение 23 400.00 и затем была выведена на печать, то результат был бы \$23 400.00. Для более подробного рассмотрения вопроса о редактировании числовых данных читателю предлагается обратиться к справочному руководству по ПЛ/1.

В ПЛ/1 имеется два вида переменных, представляющих групповые значения. Первые называются **массивами**, вторые — **структурами**. Массив — это *n*-мерный набор элементов, имеющих одинаковые атрибуты. Например, массив **A** может быть одномерным списком из пяти элементов, каждый из которых имеет атрибуты **REAL FLOAT DEC (6)**. Аналогично **B** может обозначать двумерный массив с пятью строками и четырьмя столбцами элементов, имеющих атрибуты **REAL FIXED DEC (2,0)**. Массивы **A** и **B** изображены на рис. 17.1. Как показано на рисунке, хотя все элементы массива **A** (или массива **B**) должны иметь одни и те же атрибуты, значения элементов, как правило, отличаются друг от друга и изменяются при выполнении программы. Максимально допустимое число измерений для массива зависит от реализации.

Для того чтобы определить массив, число измерений в нем и число элементов в каждом измерении, массив надо описать следующим образом:

DCL имя (размер) атрибуты;

Здесь *имя* — это имя массива, *размер* определяет его размерность и число элементов в каждом измерении, а *атрибуты* — атрибуты каждого элемента массива. Если один или более атрибутов опущены, они будут назначены (по умолчанию) в соответствии с правилами для элементарных переменных.

Размер может быть указан в виде последовательности, состоящей из одной или более положительных целых констант,

A		B			
1.50000E0		17	42	00	-10
2.70000E-3		18	03	00	-11
1.49999E1		19	47	19	-12
0.00000E0		20	48	49	22
1.11001E17		00	00	-2	-3

Рис. 17.1.

отделенных друг от друга запятыми. Число констант определяет число измерений в массиве, а значение каждой константы определяет число элементов в соответствующем измерении. Например, описание

DCL A(5) FLOAT DEC (6);

определяет одномерный массив **A** с пятью элементами (т. е. обычный список из пяти элементов). Отметим, что атрибуты **FLOAT DEC (6)** могли бы быть опущены и при этом смысл описания не изменился бы. Аналогично описание

DCL B(5,4) FIXED DEC (2,0);

определяет двумерный массив **B** с пятью элементами в первом измерении (строками) и четырьмя элементами во втором измерении (столбцами). Таким образом, эти описания определяют массивы **A** и **B** так, как было показано выше (но не присваивают значения отдельным элементам).

Размер массива в ПЛ/1 может быть не постоянным. Вообще говоря, может быть описан массив с переменным числом элементов в каждом измерении, как это сделано с помощью следующего описания:

DCL A(N) FLOAT DEC (6);

Однако этот важный случай будет рассмотрен после того, как мы достигнем более глубокого понимания структуры ПЛ/1-программы.

Второй вид переменной в ПЛ/1, представляющей группу элементов,— это структура. В отличие от массива атрибуты отдельных элементов структуры могут не совпадать. Например, структура **PERSON** (человек) может иметь четыре различных элемента: **NAME** (имя), который является 25-символьной строкой, **\$\$#** (номер страхового полиса), который является девятизначным символьно-цифровым значением, **GROSS_PAY** (основной оклад), имеющий атрибуты **REAL FIXED DECIMAL (7,2)**, и **ADDRESS** (адрес), который является 40-символьной строкой. Пример набора значений для переменной **PERSON** дан на рис. 17.2.

В этом примере показаны два уровня переменных. На элементарном уровне находятся четыре переменные, а на более высоком уровне — переменная **PERSON**, которая включает в себя все четыре подчиненные переменные. Другой способ изображения структуры заключается в представлении ее в виде дерева, в котором имя структуры (в данном случае **PERSON**) является *корнем*, а отдельные элементы — *листьями*. Такое дерево для предыдущего примера изображено на рис. 17.3.

```
[ALLEN,B,TUCKER.....]
[275407437]
[25400.00]
[1800,BULL,RUN,ALEXANDRIA,VA,22200....]
```

Рис. 17.2.

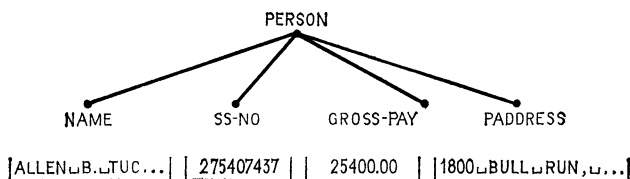


Рис. 17.3.

Для описания структуры в ПЛ/1 используется оператор **DECLARE**. Отдельные уровни переменных внутри структуры определяются следующим образом: к имени структуры (в данном случае **PERSON**) слева приписывается число 1, ко всем переменным следующего уровня — число 2 и т. д. Внутри одного уровня переменные перечисляются слева направо. Так, предыдущая структура может быть описана следующим образом:

```

DCL 1 PERSON
    2 NAME          CHAR (25),
    2 SS #          PIC '(9)9',
    2 GROSS-PAY     FIXED DEC (7,2),
    2 ADDRESS       CHAR (40);

```

Отметим, что отдельные элементы в структуре отделяются друг от друга запятыми, а все описание заканчивается точкой с запятой (;).

Некоторые переменные второго уровня в этом примере сами могут быть структурами. Например, переменная **NAME** может сама состоять из элементов **FIRST** (фамилия), **MIDDLE** (имя) и **LAST** (отчество), а переменная **ADDRESS** — из элементов **STREET** (улица), **CITY** (город), **STATE** (штат) и **ZIP** (почтовый индекс), как показано на дереве, изображенном на рис. 17.4. В этом случае описание структуры выглядит следующим образом:

```

DCL 1 PERSON,
    2 NAME,
        3 FIRST CHAR (10),
        3 MIDDLE CHAR (5),
        3 LAST CHAR (10),

```



```

2 SS # PIC '(9)9',
2 GROSS-PAY FIXED DEC (7,2),
2 ADDRESS,
  3 STREET CHAR (17),
  3 CITY CHAR (12),
  3 STATE CHAR (6),
  3 ZIP PIC '99999';

```

Следует отметить, что атрибуты имеют только те переменные в структуре, которые сами не делятся на более мелкие элементы. Они являются элементарными переменными в структуре и изображаются *листьями* на дереве, представляющем структуру. Следует также учесть, что значения, присвоенные этой структуре, были выбраны только в целях иллюстрации, и они никак не указываются и не подразумеваются в приведенных

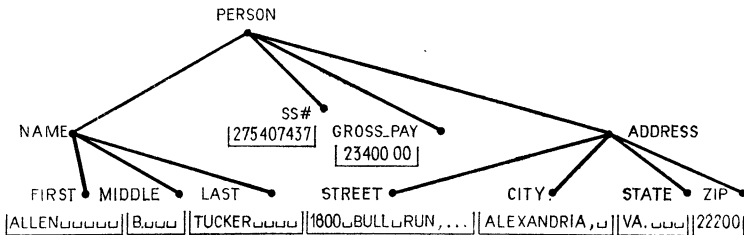


Рис. 17.4.

выше описаниях. Позже мы остановимся на рассмотрении различных способов присвоения значений элементарным переменным, массивам и структурам.

Значения элементарной переменной, элементов массива и элементов структуры могут быть изменены (или назначены) с помощью различных операторов. Для обращения к значению элементарной переменной в операторе нужно просто указать имя этой переменной.

Для того чтобы обратиться ко всему набору элементов массива, необходимо указать имя этого массива. Однако для обращения к отдельному элементу массива необходимо указать имя этого массива, за которым следует список индексов, заключенный в скобки. Число индексов должно совпадать с числом измерений массива, указанным в описании. Вернемся к рассмотрению массивов **A** и **B**, описанных выше. Для обращения к четвертому элементу массива **A** следует записать **A(4)**. Аналогично, для того чтобы обратиться к элементу, расположенному в третьей строке и втором столбце массива **B**, следует записать **B(3,2)**. Хотя в этих двух примерах индексы — целые константы, это не является общим правилом. Например, если **I** — цело-

численная переменная (скажем, переменная с атрибутами **REAL FIXED BIN (15,0)**), принимающая значения 1, 2, 3, 4 или 5, то выражение **A(I)** определяет соответственно первый, второй, третий, четвертый или пятый элементы массива **A**. То есть при различных значениях **I** выражение **A(I)** определяет различные элементы массива **A**. Аналогично, если **I** и **J** — целочисленные переменные, выражение **B(I,J)** определяет элемент, расположенный в **I**-й строке и **J**-м столбце массива **B**.

Другой способ обращения к выбранным элементам массива основан на использовании так называемого сечения. Рассмотрим вновь определенный выше массив **B** размером 5×4 . Для того чтобы обратиться ко всем элементам одной строки, напри-

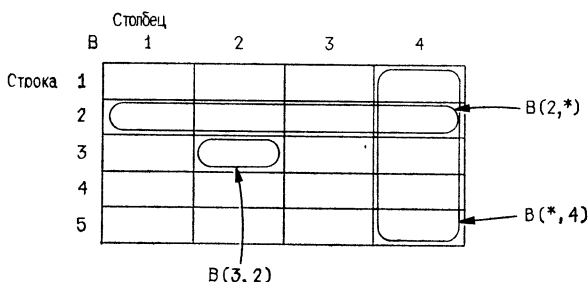


Рис. 17.5.

мер второй строки массива **B**, следует записать **B(2,*)**. Аналогично с помощью выражения **B(*,4)** можно обращаться ко всему четвертому столбцу массива **B**. Эти два сечения массива **B** показаны на рис. 17.5.

Таким образом, сечения используются для уменьшения размерности массива; результатом является массив с меньшим числом измерений. В рассмотренном примере были взяты одномерные сечения **B(2,*)** и **B(*,4)** двумерного массива **B**. Так же как и для случая отдельных элементов, при определении сечений вместо констант могут быть использованы переменные. Например, **B(I,*)** означает **I**-ю строку массива **B**, а **B(*,J)** — **J**-й столбец. Как мы увидим в дальнейшем, использование сечений особенно удобно при решении задач матричной алгебры.

Для того чтобы обратиться ко всей структуре, нужно просто указать ее имя, например **PERSON** в предыдущем примере. Для обращения к подструктуре указывается имя этой подструктуры. Для обращения к отдельному элементу структуры необходимо указать только имя этого элемента. Кроме того, могут существовать два различных элемента (или две подструктуры) с

одним и тем же именем, как показано в следующем примере:

```

DCL 1 PERSON-A,
    2 NAME CHAR (25),
    2 SS # PIC '(9)9',
    2 GROSS-PAY FIXED DEC (7,2),
    2 ADDRESS CHAR (40),
1 PERSON-B,
    2 NAME CHAR (25),
    2 SS # PIC '(9)9',
    2 GROSS-PAY FIXED DEC (7,2),
    2 ADDRESS CHAR (40);

```

Эти две структуры, **PERSON_A** и **PERSON_B**, содержат элементы с одинаковыми именами. (Сами структуры должны иметь уникальные имена.) Для того чтобы отличить элемент, например **SS#**, структуры **PERSON_A** от того же элемента структуры **PERSON_B**, необходимо указать составное имя. При обращении к элементу **SS#** структуры **PERSON_B** следует записать составное имя **PERSON_B.SS#**, а к элементу **SS#** структуры **PERSON_A** следует обращаться по имени **PERSON_A.SS#**. Составное имя должно использоваться в тех случаях, когда указание простого имени приводит к неоднозначности.

Наконец, следует отметить, что ПЛ/1 позволяет использовать массивы в качестве элементов структуры, а структуры — в качестве элементов массива. Однако на практике в большинстве случаев в качестве элементов структуры используются только одномерные массивы.

Присвоение начальных значений переменным, массивам и структурам

Часто бывает удобно присваивать начальные значения переменным, массивам или структурам изначально (в момент, когда начинает выполняться сегмент программы, в котором они описаны). В ПЛ/1 это достигается с помощью атрибута **INITIAL**, который указывается в описании переменной, массива, структуры. Он имеет следующую форму:

INITIAL (список-значений)

Здесь *список-значений* — это список, состоящий из одного или более значений, которые должны быть присвоены описанной переменной, массиву или структуре. Каждое значение может быть константой или коэффициентом повторения, который сокращает список констант.

Например, рассмотрим вновь переменные **X** и **TITLE** и массив **B**, которые были описаны в предыдущем разделе. Значения, которые там были указаны, могли быть присвоены следующим образом:

```
DCL X FLOAT DEC (6) INIT(3.59847E-5),  
    TITLE CHAR (25) INIT('1975—ANNUAL—BUDGET'),  
    B (5,4) FIXED DEC (2,0) INIT (17,42,0,—10,18,3,0,  
    —11,19,47,19,—12,20,48,49,22, 0,0,—2,—3);
```

Элементарным переменным в структуре начальные значения присваиваются аналогично.

При использовании атрибута **INITIAL** следует учитывать два обстоятельства. Во-первых, присвоение начальных значений элементам массива осуществляется по строкам. Во-вторых, если нескольким соседним (или всем) элементам массива должно быть присвоено одно и то же значение, то перед этим значением может быть указано число его повторений, заключенное в скобки. Например, если всем элементам массива **B** требуется присвоить нулевое начальное значение, то можно записать:

```
DCL B(5,4) FIXED DEC (2,0) INIT ((20)0);
```

Следует обратить особое внимание на то, что если здесь воспользоваться записью "**INIT(0)**", то нулевое значение будет присвоено только элементу **B(1,1)**, а значения всех остальных элементов массива **B** останутся неопределенными.

Наконец, следует отметить, что, если какие-либо атрибуты переменной и присваиваемого ей начального значения не совпадают, последнее *преобразуется* в эквивалентное значение, атрибуты которого согласуются с атрибутами переменной.

17.3. ОСНОВНЫЕ ОПЕРАТОРЫ

В этой главе мы не будем рассматривать все операторы ПЛ/1, поскольку важно сосредоточить наше внимание на наиболее широко используемых. В табл. 17.1 приводится список операторов, которые будут рассмотрены, а также кратко описывается их назначение.

В оставшейся части этого раздела будут рассмотрены операторы присваивания, **DECLARE**, **DO**, **END**, **GO TO**, **IF**, **PROCEDURE** и **STOP**. Остальные операторы из приведенного списка будут рассмотрены в последующих разделах. Изучение этих операторов начнем с рассмотрения следующей простой ПЛ/1-программы.

Таблица 17.1

Оператор	Назначение
ALLOCATE	Осуществляют динамическое распределение памяти
FREE	
Оператор присваивания	Выполняет ряд арифметических операций (сложение, вычитание и т. д.) или операций над строками (сцепление) и затем присваивает полученное значение переменной, массиву (элементу) или структуре (элементу)
BEGIN	Задаёт логическое начало сегмента программы
CALL	Осуществляет вызов подпрограммы
RETURN	Осуществляет возврат из подпрограммы
CLOSE	Выполняют различные операции ввода-вывода
OPEN	
GET	Выполняют различные операции ввода-вывода
PUT	
FORMAT	
READ	
WRITE	
DECLARE	Определяет переменные и их атрибуты
DO	Задаёт начало группы операторов, выполнение которых должно повториться некоторое число раз или они должны быть выполнены по условию
END	Задаёт (1) конец программы или подпрограммы, (2) конец сегмента программы, который начинается оператором BEGIN , или (3) конец группы операторов, которая начинается оператором DO
GO TO	Изменяют последовательность выполнения операторов программы
IF	
ON	Описывает действие, которое должно быть выполнено в том случае, когда при выполнении программы возникнет исключительная ситуация, как, например, конец файла или деление на нуль
PROCEDURE	Задаёт начало ПЛ/1-программы или подпрограммы
STOP	Завершает выполнение программы и возвращает управление системе

SIMPLE: PROCEDURE OPTIONS(MAIN);

DCL A (5) FLOAT, (H,I,J) FIXED BIN (15,0),

B (5,4) FIXED DEC (2,0);

DCL 1 PERSON,

2 NAME CHAR (25),

2 GROSS-PAY FIXED DEC (7,2);

**/*ЭТА ПРОГРАММА ЛИШЬ ИЛЛЮСТРИРУЕТ ПРИМЕНЕНИЕ
НЕКОТОРЫХ ОСНОВНЫХ ОПЕРАТОРОВ ПЛ/1*/**

```

DO I = 1 TO 5;
A(I) = 6 - I;
END;
H = 1;
LOOP: IF H <= 5
THEN DO;
    B(H,1) = 1;
    B(H,2) = 5 + 3 * (H - A(H));
    H = H + 1;
    GO TO LOOP;
END;
B(*,3) = A; /* ЗАПОЛНЕНИЕ 3-ГО СТОЛБЦА В */
B(*,4) = B(*,1) + B(*,2) + B(*,3); /* СУММА СТРОК В */
NAME = 'ALLEN B. TUCKER';
GROSS-PAY = 25400;
STOP;
END SIMPLE;

```

Эта программа имеет небольшое практическое значение, однако в ней присутствуют все типы операторов, которые рассматриваются в этом разделе.

ПЛ/1-программа начинается с оператора, имеющего форму
имя: **PROCEDURE OPTIONS(MAIN);**

и заканчивается оператором, имеющим форму

END имя;

Слова "**OPTIONS(MAIN)**" указывают, что данная программа не является подпрограммой и поэтому выполняется в первую очередь. В написанных выше операторах *имя* идентифицирует программу и должно быть одним и тем же в обоих случаях. В нашем примере именем программы является **SIMPLE**. Именем может быть любая последовательность букв (A—Z), цифр (0—9) или знаков @, #, \$ или -, начинающаяся с буквы или одного из знаков @, # или \$.

Во всех операторах, включая эти, соседние ключевые слова и имена должны быть отделены друг от друга по крайней мере одним пробелом (—). Например, в первом операторе между словами **PROCEDURE** и **OPTIONS** должен быть пробел. Однако в тех местах, где имеются ограничители, такие, как двоеточие (:), точка с запятой (;), скобки, знак плюс (+) и т. д., пробелы не требуются.

В седьмой и восьмой строках программы находится комментарий. Комментарием может быть любая последовательность знаков, которая начинается символами /* и заканчивается сим-

волами */. Комментарии могут появляться в любом месте программы. Они не влекут за собой никаких действий и являются только лишь программной документацией.

Оператор GO TO

Операторы в ПЛ/1-программе обычно выполняются в том порядке, в котором они записаны, если не встретится оператор, который может изменить этот порядок. Одним из таких операторов является оператор **GO TO**, который имеет следующую общую форму:

GO TO метка;

Здесь *метка* — это метка некоторого оператора, расположенного между первым оператором (**PROCEDURE**) программы и ее оператором **END** (включая сам оператор **END**). *Целевым* для оператора **GO TO** может быть любой оператор из приведенного в табл. 17.1 списка, кроме операторов **PROCEDURE**, **DECLARE**, **FORMAT** и **ON**. Существуют также и некоторые другие ограничения, которые будут рассмотрены позже.

Оператор помечается располагаемыми перед ним меткой и двоеточием, как показано ниже:

метка: оператор

Здесь *меткой* может быть любая последовательность знаков, первым в которой является алфавитный (A — Z, \$, @, #) знак, а остальные — алфавитные, цифровые (0—9) или знак разбивки (_). Оператор может быть помечен указанным способом произвольное число раз. (По мнению автора, это средство не дает большой пользы.) Например, в приведенной выше программе содержатся оператор с меткой **LOOP** и оператор **GO TO**, который осуществляет возврат управления на оператор с меткой **LOOP**.

Оператор STOP

Выполнение оператора **STOP** просто приводит к завершению работы программы. Если выполнение программы завершается нормально, т. е. при достижении последнего оператора (оператора **END**), то оператор **STOP** может включаться, а может и не включаться в программу. В противном случае он должен быть помещен в том месте программы, где ее выполнение должно завершиться. Оператор **STOP** всегда имеет следующую форму:

STOP;

Выражения

В этом разделе нам осталось рассмотреть операторы присваивания, **DO** и **IF**. Для этого необходимо ввести очень важное понятие языка ПЛ/1 — выражение.

В ПЛ/1 выражение является основным средством записи выполняемых вычислений. Вычисления могут выполняться над числами или строками. Результатом вычислений может быть как одно-единственное значение, так и несколько значений (при обработке массивов и структур). Хотя это не совсем строго соответствует соглашениям, принятым в языке ПЛ/1, с методической целью разобьем все выражения на следующие классы:

- Элементарные выражения: арифметические выражения, строковые выражения, логические выражения
- Выражения над массивами
- Выражения над структурами

Элементарные выражения обладают общим свойством, которое заключается в том, что результатом их вычисления является одно-единственное значение. Для арифметического выражения результатом всегда является числовое значение, для строкового выражения — символьная или битовая строка, для логического выражения — однобитовая строка. Значения '1'В и '0'В интерпретируются как *истина* и *ложь* соответственно.

Арифметическое выражение — это либо один-единственный терм, либо несколько термов, разделенных арифметическими операторами и, возможно, скобками (как показано ниже). Термом может быть числовая константа, имя числовой переменной, обращение к числовой функции или элемент числового массива, как показано в следующих примерах:

10 1 GROSS-PAY SQRT(X) A(N)

Каждый из этих термов определяет одно-единственное значение. Первый из них является константой, следующие два — именами переменных, четвертый — обращением к функции и последний — элементом массива.

Ниже приводятся арифметические операторы и указывается их смысл.

Оператор	Смысл
+	сложение
-	вычитание
*	умножение
/	деление
**	возведение в степень

Каждый из этих операторов выполняется с двумя операндами, один из которых располагается в арифметическом выражении слева от знака операции, а другой — справа. Например,

$$N + 2$$

означает сложение текущего значения переменной **N** с константой 2.

Когда требуется выполнить две или более операций, они выполняются слева направо. Например, для того чтобы прибавить значение **I** к сумме **N + 2**, нужно записать следующее выражение (опуская занумерованные стрелки):

$$\begin{array}{c} N + 2 + I \\ \uparrow \quad \uparrow \\ \textcircled{1} \quad \textcircled{2} \end{array}$$

Как показано занумерованными стрелками, вначале выполняется сложение **N + 2**, а затем к полученной сумме прибавляется значение **I**.

Последовательность из двух или более операций обычно выполняется слева направо, однако здесь имеются два исключения. Во-первых, существует иерархия операций. Вначале выполняются все операции возведения в степень (******) (справа налево), затем — все операции умножения (*****) и деления (**/**) (слева направо) и наконец — все операции сложения (**+**) и вычитания (**-**) (слева направо). Следовательно, выражение

$$\begin{array}{c} N + 2 * I \\ \uparrow \quad \uparrow \\ \textcircled{2} \quad \textcircled{1} \end{array}$$

является суммой **N** и **2*I**, а не произведением **N + 2** и **I**.

Во-вторых, всякий раз, когда требуется, программист может изменить этот порядок, указывая в скобках те операции, которые должны быть выполнены в первую очередь. Например, произведение величин **N + 2** и **I** записывается следующим образом:

$$\begin{array}{c} (N + 2) * I \\ \uparrow \quad \uparrow \\ \textcircled{1} \quad \textcircled{2} \end{array}$$

Как показано занумерованными стрелками, вначале здесь выполняется сложение.

Строковое выражение — это либо один-единственный терм, либо несколько термов, разделенных знаком операции над строками **||**.

Отдельным термом в строковом выражении может быть строковая константа, имя строковой переменной, элемент строкового массива или обращение к строковой функции, как это показано ниже:

'ABABAB__' NAME S(I) SUBSTR(NAME, 1, 5)

Знак **||** означает сцепление двух строк, в результате которого образуется новая строка. Например, если требуется объединить строки **'ABA'** и **'BAB__'** в одну строку **'ABABAB__'**, следует записать

'ABA' || 'BAB__'

Хотя операция сцепления является единственной операцией над строками в языке ПЛ/1, он предоставляет большие возможности для обработки строк благодаря встроенным строковым функциям.

Логическим выражением может быть однобитовая строковая константа (**'0'B** или **'1'B**), однобитовая строковая переменная, обращение к функции, значениями которой являются однобитовые строковые константы, отношение между двумя арифметическими или двумя строковыми выражениями, а также последовательность таких логических выражений, соединенных логическими операторами **&** или **|**, причем перед каждым выражением может стоять логический оператор **¬**. Отношение образуется из двух арифметических или двух строковых выражений, разделенных одним из следующих операторов отношения: **<**, **<=**, **=**, **¬=**, **>=**, **>**.

В табл. 17.2 приводятся примеры различных видов логических выражений. В этих примерах предполагается, что **P** — это переменная, описанная как **BIT(1)**, **H** — как **FIXED BIN**, **GROSS_PAY** — как **FIXED DEC (7,2)**, **NUMERIC** — функция, возвращающая значение **'0'B** или **'1'B**, а **S** и **NAME** — строковые переменные.

Операторы отношения, **<**, **<=**, **=**, **¬=**, **>=** и **>**, а также логические операторы **&** и **|** являются бинарными. Они используются для представления условий, перечисленных в табл. 17.3. Выражение "А операция В" истинно или ложно в зависимости от того, выполнено или нет соответствующее условие. Наконец, логический оператор **¬** является унарным. Выражение **¬А** представляет условие, значение которого есть истина, когда значением логического выражения **А** является ложь. В противном случае значением **¬А** является ложь.

Все операторы отношения имеют одинаковый приоритет, который выше приоритетов логических операторов и ниже приоритетов арифметических операторов. Кроме того, приоритет оператора **&** выше приоритета оператора **|**, а приоритет оператора

Таблица 17.2

Вид логического выражения	Пример	Смысл
Однобитовая константа	'0'B	"Ложь"
Однобитовая строковая переменная	P	"Истина", если значением является '1'B, и "ложь", если '0'B
Обращение к функции	NUMERIC(S)	"Истина", если возвращаемое значение есть '1'B, и "ложь", если '0'B
Отношение между двумя арифметическими выражениями	$H <= 5$	"Истина", если значение H меньше или равно 5, и "ложь" в противном случае
Несколько таких выражений, соединенных логическими операторами	$H <= 5 NAME = 'ALLEN'$ $0 < H \& H < 5$ $\neg NUMERIC(S)$	"Истина", если значение H меньше или равно 5 или если значением переменной NAME является 'ALLEN'; "ложь" в противном случае "Истина", если 0 меньше значения H и значение H меньше 5; "ложь" в противном случае "Истина", если дополнением возвращаемого значения является '1'B, и "ложь", если '0'B

Таблица 17.3

Оператор	Условие, представляемое выражением "А оператор В"
<	А меньше В
<=	А меньше или равно В
=	А равно В
!=	А не равно В
>=	А больше или равно В
>	А больше В
	Либо А, либо В (либо оба) истинно
&	И А и В истинны

\neg выше приоритетов операторов & и |. Для того чтобы изменить порядок выполнения операторов, определяемый их приоритетами, в логических выражениях, так же как и в арифметических, используются скобки. Таким образом, выражение

$$0 < H \& H < 5$$

интерпретируется так, как это было сделано выше. С другой стороны, при вычислении выражения

$$P \mid Q \& (H \leq 5 \mid NAME = 'ALLEN')$$

$\uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow$
 ⑤ ④ ① ③ ②

операторы выполняются в порядке, указанном занумерованными стрелками, так как в результате использования скобок приоритет оператора $\&$ стал ниже приоритета второго оператора \mid .

В применении к арифметическим выражениям операторы $<$, \leq , $=$, \neq , $>$ и $>=$ отражают обычные отношения, существующие между числами. Например, условие

$$H \leq 5$$

устанавливает, справедливо ли, что число H меньше или равно числу 5. Для любого из следующих значений H ответом является "истина" ('1'В):

$$5 \quad 4.9999 \quad 0 \quad -3 \quad -100$$

В применении к строковым выражения отношения отражают так называемую *лексикографическую упорядоченность*, или неявное отношение порядка, которое существует между знаками из набора символов. Это отношение может зависеть от реализации. Однако всегда справедливо следующее полезное правило:

$$_ < \{\text{все специальные знаки}\} < A < B < \dots < Z < 0 < 1 < \dots < 9$$

Под *специальными знаками* подразумеваются те, которые не являются ни буквами, ни цифрами, например "+", ";", и т. п.

Если S и T — односимвольные строки, то S меньше T в том и только в том случае, когда S предшествует T в лексикографическом порядке. Например, ' A ' $<$ ' B ' есть истина, ' $_$ ' $<$ ' A ' есть истина и т. д. Если S и T имеют одну и ту же длину, превосходящую 1, то S меньше T в том случае, когда выполнено одно из следующих условий:

1. Крайний слева знак в S меньше крайнего слева знака в T .
 2. Первые k знаков в S и T совпадают ($1 \leq k < \text{длины } S \text{ или } T$), но $(k+1)$ -й знак в S меньше $(k+1)$ -го знака в T .
- Это означает например, что ' $ALLEN$ ' $<$ ' $BROWN$ ' есть истина, так как ' A ' $<$ ' B '. Это также означает, что ' $ALLAN$ ' $<$ ' $ALLEN$ ' есть истина, так как (для $k=3$) первые три знака в этих двух строках совпадают, а четвертый знак (A) в строке ' $ALLAN$ ' меньше четвертого знака (E) в строке ' $ALLEN$ '.

Наконец, если строки **S** и **T** имеют различные длины, то более короткая из них дополняется справа необходимым числом пробелов (**_**), с тем чтобы выравнивать длины строк, а затем осуществляется описанная выше проверка. Так, '**ALL**' < '**ALLEN**' есть истина, поскольку '**ALL__**' < '**ALLEN**' есть истина.

Отношение между двумя битовыми строками определяется аналогично с учетом того, что '**0'B** < '**1'B**' есть истина. Если две сравниваемые битовые строки имеют *различные* длины, перед проверкой описанных выше условий (1) и (2) более короткая из них дополняется справа нулями (а не пробелами).

Символьные строки **S** и **T** *равны* (т. е. **S** = **T** есть истина), если выполнено одно из следующих условий:

1. **S** и **T** имеют равные длины и состоят из одних и тех же знаков, расположенных в одном и том же порядке (т. е. они идентичны).

2. **S** и **T** имеют различные длины, но если более короткую из них дополнить справа достаточным для выравнивания длин числом пробелов, строки станут идентичными.

Таким образом, '**ABC**' равно только '**ABC**', '**ABC__**', '**ABC____**' и т. д. Однако '**ABC**' не равно '**__ABC**' или '**BAC**'.

Равенство между двумя битовыми строками определяется аналогично, за исключением того, что более короткая из них дополняется справа нулями (а не пробелами).

Другие отношения (**<=**, **¬=**, **>=** и **>**) определяются для строк на основе отношений **<** и **=** согласно следующей таблице.

Отношение	Смысл выражения "А отношение В"
<=	Либо A < B , либо A = B есть истина
¬=	A = B не есть истина
>=	Либо B = A , либо B < A есть истина
>	B < A есть истина

Этим завершается рассмотрение элементарных выражений.

Помимо элементарных величин выражения в ПЛ/1 могут содержать в качестве операндов массивы или структуры. Выражение, содержащее имена массивов, называется выражением над массивами, и результатом его вычисления является не одно значение, а массив. Выражение, содержащее имена структур, называется выражением над структурами, и результатом его вычисления является структура. Мы не будем рассматривать здесь выражения над структурами.

Пусть **A** и **B** — два массива размером 4×3 , элементы которых имеют атрибуты **REAL FIXED DEC (5,0)**:

$$\mathbf{A} = \begin{bmatrix} 5 & 2 & 0 \\ -1 & 0 & 0 \\ 3 & 7 & 0 \\ 9 & 4 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 1 & 2 & 4 \\ 3 & 2 & 3 \\ 2 & 1 & 1 \\ 0 & 5 & 3 \end{bmatrix}$$

Выражение **3*A** определяет массив размером 4×3 , образованный умножением каждого элемента массива **A** на 3:

$$3*\mathbf{A} = \begin{bmatrix} 15 & 6 & 0 \\ -3 & 0 & 0 \\ 9 & 21 & 0 \\ 27 & 12 & 0 \end{bmatrix}$$

Выражение **A + B** определяет массив размером 4×3 , образованный сложением соответствующих элементов массивов **A** и **B**:

$$\mathbf{A} + \mathbf{B} = \begin{bmatrix} 6 & 4 & 4 \\ 2 & 2 & 3 \\ 5 & 8 & 1 \\ 9 & 9 & 3 \end{bmatrix}$$

Аналогично выражение **A op B**, где *op* — один из операторов —, *, / или **, определяет массив размером 4×3 , образованный в результате выполнения операций вычитания, умножения, деления или возведения в степень над соответствующими элементами массивов **A** и **B**.

Хотя между массивом в ПЛ/1 и алгебраическим понятием матрицы существует естественная аналогия, программист с математическим уклоном должен обратить особое внимание на то, что при рассмотрении умножения этой аналогии нет; умножение массивов, т. е. **A*B**, не соответствует произведению матриц. Последнее не может быть получено одним оператором ПЛ/1.

Существует несколько общих ограничений, о которых необходимо помнить при записи выражений над массивами. Во-первых, число измерений и число элементов в каждом измерении должны быть одинаковыми для всех массивов, к которым обращаются в одном выражении. Например, с массивом размером 4×3 можно складывать только массивы размером 4×3 . Во-вторых, в арифметических выражениях могут использоваться сечения массивов. Например, если требуется вычислить одномерный массив из трех элементов, образованный умножением соответствующих элементов третьей строки массива **A** и второй

строки массива **B** (массивы заданы выше), то следует записать выражение

$$A(3, *) * B(2, *)$$

При этом будет получен массив [9 14 0].

Атрибуты результата вычисления выражения

Результатом вычисления элементарного выражения всегда является одно число, символьная строка или битовая строка. Хотя все операнды арифметического выражения являются числами, обычно не все они имеют одни и те же атрибуты. Например, операнды даже такого простого выражения, как

$$1 + 1$$

по всей вероятности, будут иметь различные атрибуты; для **1** — **REAL FIXED (15,0)**, а для константы **1** — **REAL FIXED DEC (1,0)**.

Тип, формат и основание результата выполнения арифметических операций определяются в соответствии со следующей таблицей (для случая, когда атрибуты одного операнда не полностью совпадают с атрибутами другого).

Атрибуты одного операнда	Атрибуты другого операнда	Атрибуты результата
REAL	COMPLEX	COMPLEX
FIXED	FLOAT	FLOAT
DECIMAL	BINARY	BINARY

Если два операнда согласуются по всем атрибутам, результат имеет их общие атрибуты. Точность результата зависит как от типа арифметической операции, так и от реализации. Правила определения точности являются настолько громоздкими, что любое достаточно подробное для понимания изложение их заняло бы здесь слишком много места. Отметим только, что точность, определяемая для результата выполнения операции, имеющей форму

$$A \text{ on } B$$

где *on* — знак $+$, $-$, $*$, $/$ или $**$, в достаточной степени соответствует тому, чтобы записать наибольшее (наименьшее) значение, которое может быть получено при выполнении этой операции. Например, результат выполнения операции $1 + 1$ имеет

атрибуты **FIXED BIN (15,0)**, если **I** определено как **FIXED BIN (15,0)**.

Точность результата арифметической операции ограничена также некоторыми максимальными значениями, которые зависят от реализации. Для ПЛ/1(F) эти значения перечислены в следующей таблице.

Основание и формат результата	Максимальная точность результата
FIXED DECIMAL	(15,q), где $0 \leq q \leq 15$
FIXED BINARY	(31,q), где $0 \leq q \leq 31$
FLOAT DECIMAL	(16)
FLOAT BINARY	(53)

Когда результатом выполнения арифметической операции над числами с атрибутами **FIXED (DEC или BIN)** является столь большое значение, что его крайняя левая значащая цифра переходит допустимую границу точности, возникает так называемое состояние **FIXEDOVERFLOW** (переполнение при фиксированном формате). Например, при выполнении следующего сложения над числами с атрибутами **FIXED DEC**

$$9999999999999999 + 1$$

результат будет содержать 16 значащих цифр. Возникновение указанного состояния приводит к прерыванию выполнения программы. Вопрос о том, возобновлять ли выполнение программы и, если да, то с какого места, решается программистом.

Оператор присваивания

Основным средством вычисления и присвоения нового значения переменной, массиву или структуре является оператор присваивания. Он имеет следующую форму:

переменная = выражение;

Здесь *переменная* — это элементарная переменная, массив или структура, а *выражение* — элементарное выражение, выражение над массивами или выражение над структурами. Ниже будет показано, что не все варианты операторов, удовлетворяющие этому определению, являются допустимыми.

При выполнении оператора присваивания вначале вычисляется выражение, записанное в правой части. Затем вычисленный результат принимается в качестве нового значения (*при-*

сваивается) переменной. Предположим, например, что к моменту выполнения оператора

$$I = I + 1;$$

переменная **I** с атрибутами **FIXED BIN (15,0)** имеет значение, равное 43. Тогда вначале вычисляется значение выражения $I + 1$, которое равно 44, а затем оно становится новым значением переменной **I**.

Переменная, расположенная слева от знака присваивания "=", может быть также элементом массива. Предположим, например, что **KA** — одномерный массив 10 величин с атрибутами **FIXED BIN (15,0)**. Тогда в результате выполнения оператора $KA(3) = I + 1$ третьему элементу массива **KA** будет присвоено значение $I + 1$.

Слева от знака присваивания "=" может быть указано более одной переменной. В этом случае оператор присваивания называется **групповым**. Он удобен для присвоения одного и того же начального значения нескольким переменным. Например, для того чтобы переменным **I**, **J** и **K** присвоить нулевое начальное значение, можно записать

$$I, J, K = 0;$$

Такая запись является более удобной, чем следующая эквивалентная ей последовательность операторов:

$$I = 0; J = 0; K = 0;$$

Если атрибуты переменной, расположенной слева от знака присваивания "=", не совпадают с атрибутами значения выражения, стоящего справа от него, то выполняется преобразование. При преобразовании значение с одним набором атрибутов заменяется эквивалентным или почти эквивалентным значением с другим набором атрибутов. Например, значение 15 имеет атрибуты **REAL FIXED DEC (2,0)**. При преобразовании его в значение с атрибутами **REAL FIXED BIN (15,0)** будет получено

000000000001111B

Для того чтобы полностью разобраться в вопросе о преобразовании, требуется немало усилий. Основная причина этого заключается, по-видимому, в том, что правила преобразования зависят от реализации. Тем не менее программист должен знать, что, если атрибуты переменной, стоящей в левой части оператора присваивания, не согласуются с атрибутами присваиваемого ей значения, при каждом выполнении такого оператора осуществляется преобразование. Преобразование имеет место не только при выполнении операторов присваивания, но и при вы-

полнении так называемых операторов потокоориентированного ввода.

В большинстве реализаций ПЛ/1 все “разумные” преобразования четко определены. Даже преобразование символьной строки, содержащей некоторое допустимое число, например ‘1.23’, в ее эквивалентное арифметическое значение выполняется в большинстве реализаций. Однако некоторые преобразования невозможны. Например, если бы нас попросили преобразовать символьную строку ‘ALLEN’ в эквивалентное арифметическое значение, то это вызвало бы серьезные трудности. В ситуациях, аналогичных данной, когда преобразование практически невозможно, возникает так называемое состояние **CONVERSION** (преобразование). Это исключительное состояние, которое вызывает прерывание выполнения программы. Вопрос о том, возобновлять ли выполнение программы и если да, то с какого места, решается программистом.

Взаимные преобразования чисел с фиксированной (**FIXED**) и плавающей (**FLOAT**) точкой и десятичных (**DECIMAL**) и двоичных (**BINARY**) чисел являются довольно простыми; они осуществляются обычным арифметическим способом. Однако преобразование точности арифметического значения не всегда является подходящим. В частности, при преобразовании числа с фиксированной точкой, имеющего одну точность, в число с другой точностью происходит либо добавление ведущих или конечных нулей к исходному числу, либо отбрасывание ведущих или конечных цифр. В следующей таблице показано, что происходит, когда исходное значение преобразуется в значение с заданными атрибутами.

Исходное значение	Заданные атрибуты	Результат преобразования
3.19	FIXED DEC (2,1)	3.1
3.19	FIXED DEC (2,2)	.19
3.19	FIXED DEC (4,3)	3.190
3.19	FIXED DEC (4,2)	03.19

Из таблицы видно, что в результате преобразования могут отбрасываться как конечные, так и ведущие значащие цифры. Отбрасывание конечных цифр приводит к потере точности, но при этом не возникает состояния ошибки. Однако при отбрасывании ведущих значащих цифр, как 3 во втором примере таблицы, возникает состояние **SIZE** (размер), которое прерывает выполнение программы. Вопрос о том, возобновлять ли выпол-

нение программы и если да, то с какого места, решается программистом.

Преобразование другого вида, так называемое преобразование длины, выполняется в тех случаях, когда символично-строчной (**CHAR**) (или битово-строчной (**BIT**)) переменной фиксированной длины в качестве значения присваивается символьная (или битовая) строка, длина которой отлична от длины переменной. Предположим, например, что **S** и **T** — строковые переменные, описанные следующим образом:

DCL S CHAR (5), T CHAR (1);

Если длина переменной больше длины присваиваемого ей значения, последнее дополняется справа необходимым для выравнивания длин числом пробелов. Например, в результате выполнения оператора

S = 'ALL';

переменной **S** присваивается значение **'ALL__'**. Если длина переменной меньше длины присваиваемого ей значения, последнее усекается справа соответствующим образом. Например, в результате выполнения оператора

T = 'ALL';

переменной **T** присваивается значение **'A'**.

Такое преобразование часто очень полезно для программиста. Например, если требуется заполнить пробелами длинную, скажем 132-символьную, строку, отведенную под переменную **Y**, программист может просто записать

Y = '_';

Однако при присваивании строковым переменным переменной длины значений их длины, так же как и значения, меняются. Например, если **S** имеет атрибуты **CHAR (5) VARYING**, то в результате выполнения оператора **S = 'ALL'** переменной **S** присваивается значение **'ALL'**, а ее длина становится равной 3.

До сих пор мы рассматривали лишь операцию присваивания элементарных значений элементарным переменным. Такое использование оператора присваивания действительно является наиболее общим. Однако иногда требуется присвоить новые значения всем элементам массива или структуры. Для этих целей можно воспользоваться операторами присваивания для массивов или структур.

Одна из часто возникающих ситуаций связана с присвоением начальных значений. Если всем элементам массива **A** требуется

присвоить некоторое элементарное значение, например 0, можно просто записать

$$A = 0;$$

Отметим, что имя массива **A** пишется без индексов. Аналогично некоторое значение можно присвоить всем элементам сечения массива, указав это сечение в левой части оператора присваивания. Наконец, аналогичным образом можно присвоить одно и то же значение всем элементам структуры, указывая слева ее имя.

Другая ситуация, которая часто возникает, связана с присвоением массива значений, полученного в результате вычисления выражения над массивами, всему массиву или сечению. Предположим, например, что **B**, **C** и **D** — массивы, описанные следующим образом:

$$DCL (B(5,3), C(3), D(5,3)) FIXED BIN (15,0);$$

В этом случае мы могли бы элементам массива **B** присвоить значения соответствующих элементов массива **D**, просто записав

$$B = D;$$

С другой стороны, мы могли бы присвоить элементам второй строки **B** значения соответствующих элементов массива, представляющего сумму **C** и удвоенной четвертой строки **D**, следующим образом:

$$B(2,*) = C + 2 * D(4, *);$$

При присваивании массиву значений, полученных в результате вычисления выражения над массивами, всегда следует придерживаться следующего безусловного правила. Массив, которому присваиваются новые значения, и массив, являющийся результатом вычисления выражения над массивами, должны иметь одинаковую размерность и одинаковое число элементов в каждом измерении.

Оператор IF

В тех случаях, когда оператор (или последовательность операторов) **S** должен быть выполнен только при выполнении некоторого условия, представленного логическим выражением (т. е. значение **e** должно быть равным '1'В), используется оператор **IF**. Он имеет одну из следующих двух форм:

1. IF **e** THEN **S**

2. IF **e** THEN **S₁** ELSE **S₂**

Здесь **e** — логическое выражение, а **S**, **S₁** и **S₂** — любой оператор, группа **DO** или блок **BEGIN**. Группы **DO** и блоки **BEGIN** (они будут рассмотрены позже) являются чрезвычайно полезными средствами для написания структурированных программ. Отметим также, что в конце оператора **IF** точка с запятой явно не указывается. Однако каждый из операторов **S**, **S₁** и **S₂** заканчивается точкой с запятой, так что конец оператора **IF** всегда определен однозначно.

Оператор **IF** в первой форме выполняется в два этапа. Вначале вычисляется выражение **e**; в результате вырабатывается значение **'1'B** (истина) или **'0'B** (ложь). Если результат — истина, выполняется оператор **S**. В противном случае **S** обходится.

Оператор **IF** во второй форме также выполняется в два этапа. Вначале вычисляется выражение **e** и вновь результатом является истина или ложь. Затем один из операторов **S₁** или **S₂** выполняется, а другой обходится в зависимости от того, какой результат был получен на предыдущем шаге — истина или ложь.

Рассмотрим несколько примеров с использованием оператора **IF**. Предположим, что нам требуется вычислить значение **FED_TAX** (государственный налог), равное 22 процентам от величины **GROSS_PAY** (основной оклад) при условии, что последняя меньше 18 000 долл. Предполагается, что **GROSS_PAY** и **FED_TAX** — либо символично-цифровые переменные, описанные как **PICTURE '(5)9V99'**, либо переменные с атрибутами **FIXED DEC (7,2)**. Тогда мы могли бы записать следующий оператор **IF** (в первой форме):

```
IF GROSS_PAY < 18000
  THEN FED_TAX = .22 * GROSS_PAY;
```

Здесь оператор присваивания используется в качестве **S**, а выражением **e** является **GROSS_PAY < 18 000**.

Предположим далее, что если величина **GROSS_PAY** не меньше 18 000, значение **FED_TAX** должно быть вычислено равным 25 процентам от **GROSS_PAY**. Тогда мы могли бы записать следующий оператор **IF** (во второй форме):

```
IF GROSS_PAY < 18000
  THEN FED_TAX = .22 * GROSS_PAY;
  ELSE FED_TAX = .25 * GROSS_PAY;
```

Теперь значение **FED_TAX** вычисляется одним или другим способом в зависимости от величины **GROSS_PAY**. В предыдущем примере значение **FED_TAX** вычислялось только в том случае, когда величина **GROSS_PAY** была меньше 18 000 долл.

Группа **DO** имеет несколько форм, бóльшая часть которых будет рассмотрена нами позже. Здесь мы остановимся на рассмотрении простейшей ее формы, поскольку она естественным образом связана с оператором **IF**. Группа **DO** в своей простейшей форме состоит из последовательности выполняемых операторов, которая начинается словом **DO** и заканчивается словом **END**. Пример программы, приведенный выше, содержит этот вид группы **DO**, которая переписана ниже и обведена.

LOOP: IF H <= 5

THEN DO; B(H,1) = 1; B(H,2) = 5 + 3 * (H - A(H)); H = H + 1; GO TO LOOP; END;
--

Как видно, эта группа **DO** находится внутри оператора **IF** и поэтому выполняется только в том случае, когда выполнено условие $H \leq 5$. В противном случае она полностью обходится.

Рассмотрим более близкий к практике пример: нахождение вещественных корней x квадратного уравнения

$$ax^2 + bx + c = 0 \quad (a \neq 0),$$

где a , b и c — заданные вещественные числа.

Сначала можно определить число корней и их значения, вычисляя дискриминант d по формуле

$$d = b^2 - 4ac.$$

Если $d < 0$, то вещественных корней не существует. Если $d = 0$, существует один вещественный корень x_1 , вычисляемый по формуле

$$x_1 = -b/(2a).$$

Если $d > 0$, существуют два вещественных корня, x_1 и x_2 , вычисляемые по формулам

$$x_1 = (-b + \sqrt{d})/(2a),$$

$$x_2 = (-b - \sqrt{d})/(2a).$$

Фрагмент программы, вычисляющей число корней (**NROOTS**) и их значения (**X1** и **X2**) по заданным коэффициентам (A , B и C), выглядит следующим образом:

```

D = B ** 2 - 4 * A * C;
IF D < 0 THEN NROOTS = 0;
ELSE IF D = 0
    THEN DO;
        NROOTS = 1;
        X1 = - B / (2 * A);
        END;
    ELSE DO;
        NROOTS = 2;
        X1 = (- B + D **.5) / (2 * A);
        X2 = (- B - D **.5) / (2 * A);
        END;

```

Как видно, за словом **ELSE** в первом операторе **IF** следует другой оператор **IF** (во второй форме), в каждой части которого содержится отдельная группа **DO**. Как видно, такое структурирование позволяет составлять программу просто и ясно.

Оператор **DO** и циклы

При программировании часто имеют дело с соответствующим образом определенными управляемыми (в противоположность бесконечным) циклами. В ПЛ/1 имеется несколько форм оператора **DO**, который позволяет писать такие циклы.

Управляемый цикл — это последовательность операторов, выполнение которых повторяется до тех пор, пока не будет выполнено определенное условие. Многие такие циклы являются *управляемыми с помощью счетчика*. В них переменной, используемой в качестве счетчика, присваивается начальное значение, которое затем увеличивается и проверяется при каждом выполнении последовательности операторов. Когда значение этой переменной выходит за указанную границу, выполнение цикла завершается. Процесс выполнения двух различных форм такого цикла описан на рис. 17.6. Здесь i — переменная, управляющая циклом, а m_1 , m_2 и m_3 — выражения, представляющие соответственно начальное значение, конечное значение и шаг изменения управляющей переменной. Пример блок-схемы рис. 17.6, б содержится в приведенной ниже программе. Здесь N — управляющая переменная, ее начальное значение равно $m_1 = 1$, конечное значение $m_2 = 5$ и шаг $m_3 = 1$. Блок-схема рис. 17.6, б может быть записана в виде *цикла DO* следующим образом:

```
DO i = m1 TO m2 BY m3;
```

Последовательность операторов

```
END;
```

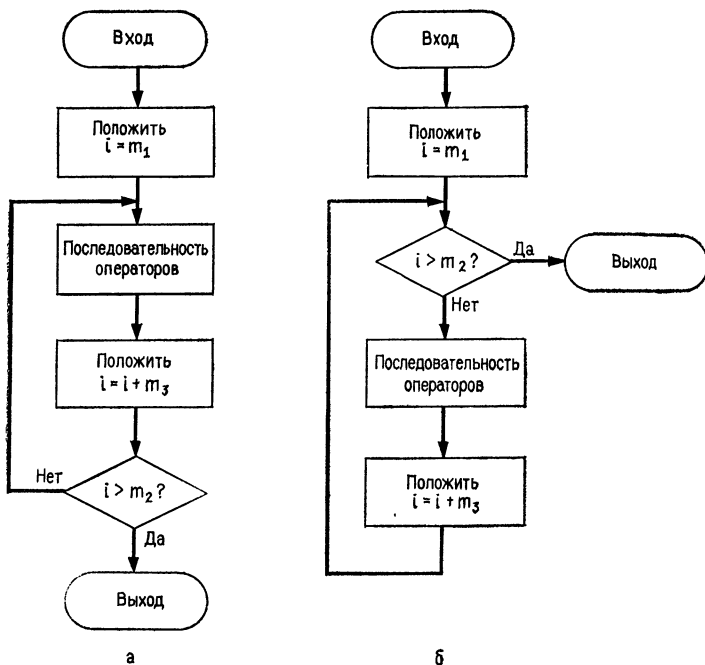


Рис. 17.6.

Цикл, содержащийся в приведенной выше программе, может быть переписан следующим образом:

```

LOOP: DO N = 1 TO 5 BY 1;
      B(N,1) = 1;
      B(N,2) = 5 + 3 * (N - A(N));
END;

```

Как видно, один этот оператор **DO** включает в себя инициализацию, проверку и приращение, выполняемые в цикле, описанном на блок-схеме рис. 17.6, б.

Следует отметить, что блок-схемы *а* и *б* рис. 17.6 не эквивалентны, поскольку первая не может быть использована в тех случаях, когда последовательность операторов не должна выполняться. Поэтому при написании цикла **DO** в ПЛ/1-программах этот случай должен учитываться.

Существует много форм оператора **DO**, которые позволяют легко описывать различные циклы, часто используемые при программировании. Мы рассмотрим следующие формы, которые, по нашему мнению, являются наиболее полезными:

1. **DO** $i = m_1$ **TO** m_2 **BY** m_3 ;
2. **DO** $i = m_1$ **TO** m_2 ;
3. **DO** $i = m_1$ **BY** m_3 ;
4. **DO** $i = v_1, v_2, \dots, v_n$;
5. **DO WHILE**(e);
6. **DO** $i = m_1$ [**TO** m_2] [**BY** m_3] **WHILE**(e);

Здесь i — переменная, $m_1, m_2, m_3, v_1, v_2, \dots, v_n$ — арифметические выражения и e — логическое выражение. Каждая из этих шести форм оператора **DO** используется для управления выполнением повторяющейся последовательности операторов. После последнего оператора пишется оператор **END** независимо от того, какая форма оператора **DO** используется. Таким образом, цикл **DO** в ПЛ/1 имеет следующую форму:

оператор **DO**

Последовательность операторов

оператор **END**

В оставшейся части этого раздела мы остановимся на вопросе о назначении каждой из шести форм и покажем, в каких случаях каждая из них используется.

Форму 1 мы уже в основном рассмотрели. Мы не упомянули о том, что значение переменной, управляющей циклом, можно не только увеличивать, но и уменьшать, если задать отрицательное значение для m_3 . Например, рассмотренный выше цикл **DO** можно переписать следующим образом:

```

LOOP: DO H = 5 TO 1 BY -1;
      B(H,1) = 1;
      B(H,2) = 5 + 3 * (H - A(H));
END;
```

Если m_3 отрицательно, то вместо указанной на блок-схеме рис. 17.6,б проверки, которая предшествует последовательности операторов, следует выполнять проверку, указанную на рис. 17.7.

Форма 2 является сокращенным вариантом записи формы 1 для случая, когда величина шага m_3 равна 1. Например, следующие операторы **DO** эквивалентны:

```

DO H = 1 TO 5 BY 1;
DO H = 1 TO 5;
```

Форма 3 используется в тех случаях, когда выход из цикла не может быть достаточно просто описан ни с помощью указания

верхней границы, ни условием **WHILE**. Ее смысл такой же, как и у формы 1, за исключением того, что не выполняется проверка, обеспечивающая выход из цикла. Программист должен позаботиться о том, чтобы таким циклом управляли некоторые другие средства, которые позволили бы в конце концов выйти из цикла. Наиболее часто данная форма цикла используется, вероятно, для управления последовательным считыванием записей из файла. В этом случае выход из цикла осуществляется при достижении конца файла, а переменная, управляющая циклом, играет роль счетчика записей.

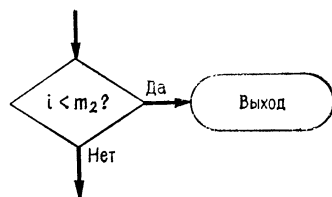


Рис. 17.7.

Форма 4 используется в тех случаях, когда последовательность значений, принимаемых переменной, управляющей циклом, не может быть описана заданием шага ее возрастания или убывания. Предположим, например, что требуется выполнить последовательность операторов по одному разу для каждого из следующих значений **Н**: 1, 3 и 4. В этом случае можно воспользоваться формой 4, записав следующий оператор:

DO Н = 1, 3, 4;

Последовательность операторов

END;

Такая запись эквивалентна той, которая дана на рис. 17.8, где “Последовательность операторов” повторяется трижды.

Н = 1;

Последовательность операторов

Н = 3;

Последовательность операторов

Н = 4;

Последовательность операторов

Рис. 17.8.

Форма 5 используется в тех случаях, когда выход из цикла осуществляется только при невыполнении определенного условия, которое задается с помощью выражения **е**. Это условие проверяется перед каждым выполнением последовательности операторов, поэтому не исключена возможность того, что эти операторы вообще не будут выполняться. Эта форма часто используется в численном анализе, где последовательность приближенных решений вычисляется до тех пор, пока не будет выполнено заданное условие сходимости. Предположим, например, что требуется вычислить приближенное значение квадратного корня \sqrt{A} методом Ньютона. В этом случае следующее приближение **У** получается из предыду-

щего приближения X по формуле

$$Y = \frac{1}{2}(X + A/X)$$

и это вычисление повторяется до тех пор, пока абсолютное значение разности между двумя последовательными приближениями не будет достаточно малым, скажем, меньше 0,0001. Для организации этих вычислений может быть использован следующий цикл:

```
DO WHILE(ABS(Y - X) >= 0.0001);
  X = Y;
  Y = 0.5 * (X + A/X);
END;
```

Форма 6 используется в тех случаях, когда цикл должен завершить работу либо при достижении верхней границы, либо при выполнении условия выхода, независимо от того, какое из этих условий будет выполнено первым. Семантика этой формы определена на рис. 17.9 (для случая $m_2 > 0$). Из рисунка видно, что выполнению последовательности операторов предшествуют обе проверки.

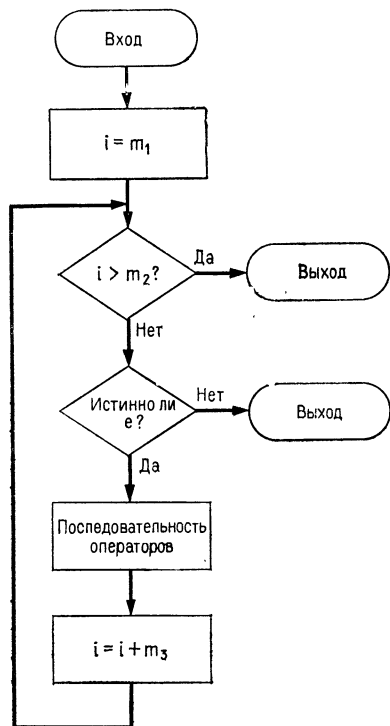


Рис. 17.9.

Эта *смешанная* форма оператора цикла может быть использована, например, в тех случаях, когда требуется предотвратить заикливание программы. В предыдущем примере предполагалось, что условие

$$ABS(Y - X) >= 0.0001$$

в конце концов не будет выполняться. Однако в некоторых случаях это не так. Поэтому программист должен обеспечить выход из цикла тем или иным способом, например задавая верхнюю границу числа его выполнений. В этом случае записанный выше оператор **DO** надо заменить на следующий:

```
DO I = 1 TO 50 WHILE(ABS(Y -
  - X) >= 0.0001)
```

Теперь, записав этот цикл, можно проверить, выполнение какого из условий привело к выходу из цикла.

Другим примером использования формы 6 является последовательный поиск в таблице. Предположим например, что **NAMES** — массив, состоящий из 50 15-символьных строк, а **NAME** — простая 15-символьная переменная. В следующем фрагменте программы проверяется, содержится ли значение переменной **NAME** в массиве **NAMES**.

```
DO I = 1 TO 50 WHILE (NAMES (I)  $\neq$  NAME);  
END;  
IF I > 50 THEN GO TO NOT-FOUND;  
  
FOUND: _____  
_____  
_____  
  
NOT-FOUND: _____
```

Этот пример показывает также, что значение переменной, управляющей циклом, в момент выхода из цикла, которым она управляет, является четко определенным и иногда может быть использовано.

Существует несколько разумных ограничений в использовании циклов **DO**, о которых следует упомянуть. Во-первых, извне цикла **DO** нельзя передавать управление ни на какой оператор, находящийся внутри его; исключение составляет передача управления самому оператору **DO** (что вызывает инициацию цикла). Во-вторых, передача управления из цикла любому оператору, расположенному внутри или вне его, конечно, допустима. Следует, однако, отметить, что передача управления из цикла оператору **END** вызывает возобновление этого цикла, а передача управления оператору **DO** вызывает повторную инициацию цикла. В-третьих, цикл **DO** может быть полностью вложенным в другой цикл **DO** при условии, что соблюдены правила передачи управления. В-четвертых, значение переменной, управляющей циклом, не должно явно изменяться внутри цикла. Однако внутри цикла можно обращаться к значению этой переменной, и эта возможность обычно используется программами.

17.4. СОГЛАШЕНИЯ О ВВОДЕ-ВЫВОДЕ

ПЛ/1 предоставляет разнообразные средства для чтения и записи данных. Существуют два класса операций ввода-вывода: потокоориентированный ввод-вывод и записеориентированный ввод-вывод. Необходимо иметь основные понятия о каж-

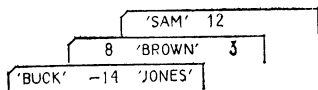


Рис. 17.10.

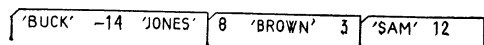


Рис. 17.11.

дом из этих классов, поскольку оба они играют существенную роль при решении широкого круга задач.

Когда данные в программе считываются или записываются с использованием потокоориентированной операции ввода-вывода, они рассматриваются как непрерывная последовательность, или поток, отдельных значений. Действительно, отдельные записи (на перфокартах) воспринимаются так, как будто они соединены друг с другом концами и образуют одну очень длинную карту, размещающую все значения данных в файл. Рассмотрим, например, колоду с входными данными, изображенную на рис. 17.10. При чтении с помощью потокоориентированных операций ввода-вывода эти данные поступают в программу слева направо, как будто перфокарты соединены так, как показано на рис. 17.11. При потокоориентированном вводе-выводе именно отдельное значение данных, а не отдельная запись, является основной единицей ввода (или вывода).

Потокоориентированные операции ввода всегда осуществляются с помощью оператора **GET**, а потокоориентированные операции вывода — с помощью оператора **PUT**. При выполнении оператора **GET** с текущей позиции потока данных считывается указанное число отдельных значений данных. При выполнении оператора **PUT** в текущую позицию потока данных размещается указанное число значений данных. После выполнения каждого из этих операторов текущая позиция потока данных продвигается вперед. Это можно представить себе, мысленно помещая указатель в начальный момент рядом с первым значением, как показано на рис. 17.12. Предположим, что при выполнении первого оператора **GET** в качестве входных данных считываются два значения: **'BUCK'** и **—14**. Теперь указатель передвигается за эти два значения и его *текущая позиция* становится такой, как показано на рис. 17.13. При следующем выполнении оператора **GET** данные будут считываться, начиная со значения **'JONES'**.

Наконец, потокоориентированный ввод-вывод всегда выполняется последовательно. Значения данных считываются из фай-

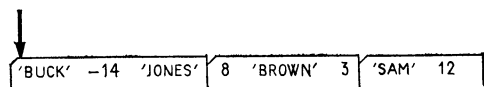


Рис. 17.12.

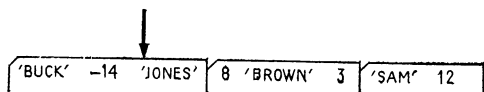


Рис. 17.13.

ла в том порядке, в котором они появляются; значения данных записываются в файл в порядке выполнения операторов **PUT**. Файл нельзя “переместить назад”, так что значение не может быть считано заново. Файл нельзя также “переместить вперед”, минуя некоторые данные, так что значения не могут быть пропущены.

Когда данные считываются или записываются с использованием записеориентированных операций ввода-вывода, файл рассматривается как набор записей. При выполнении отдельной операции ввода или вывода осуществляется считывание или запись всей записи. Записеориентированный ввод-вывод обычно выполняется последовательно. Записеориентированный ввод-вывод позволяет также считывать (писать) записи произвольно при условии, что рассматриваемый файл размещается на устройстве ввода-вывода с прямым доступом. Эта возможность часто используется при решении многих задач. В этом разделе будут рассмотрены вопросы обработки прямых файлов.

Записеориентированные операции ввода всегда осуществляются с помощью оператора **READ**, а записеориентированные операции вывода — с помощью операторов **WRITE** или **REWRITE**. При выполнении оператора **READ** (для файла, который обрабатывается последовательно) осуществляется считывание следующей записи. Аналогично при выполнении оператора **WRITE** в файл пишется следующая запись.

Независимо от того какой ввод-вывод используется, потокоориентированный или записеориентированный, математическое и физическое понятия файла остаются неизменными. Файлом называется любой набор данных, размещенный на носителе, с которого эти данные могут быть считаны вычислительной машиной (такими носителями являются, например, колода перфокарт, магнитная лента, диск, бумага для печати). **Входным файлом** называется файл, используемый для поставки данных программе. **Выходным файлом** называется файл, используемый для хранения результатов, полученных при выполнении программы.

Обновляемым файлом называется файл, используемый как для поставки результатов, так и для хранения результатов.

Потокоориентированный ввод-вывод

При потокоориентированном вводе-выводе данные в файле могут представляться либо как последовательность значений данных, разделенных пробелами, либо как последовательность записей одного и того же формата (рис. 17.14). В первом слу-

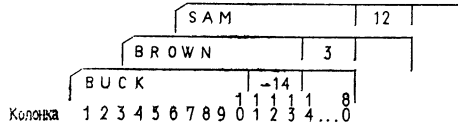


Рис. 17.14.

чае мы имеем дело с так называемым **вводом-выводом, управляемым списком**, а во втором — **вводом-выводом, управляемым редактированием**.

Ввод, **управляемый списком**, осуществляется с помощью оператора **GET**, записанного в одной из следующих форм:

1. **GET LIST** (список-переменных);
2. **GET FILE** (имя-файла) **LIST** (список-переменных);

Здесь *список-переменных* — список имен переменных, разделенных запятыми; эти переменные примут значения считываемых данных. Предположим, например, что в программе описаны следующие переменные:

```
DCL NAME CHAR (10),
    SCORE FIXED DEC (3,0);
```

Тогда в результате выполнения оператора

```
GET LIST (NAME, SCORE);
```

с данными, указанными на рис. 17.14, переменным **NAME** и **SCORE** присвоятся значения **'BUCK_____'** и **—14** соответственно.

Порядок, в котором переменные перечислены в операторе **GET**, определяет порядок, в котором присваиваются входные значения. Отметим также, что символьные строки заключаются в кавычки и это требование относится к любой строке, считываемой с помощью операции ввода, управляемого списком. Наконец, отметим, что при вводе, управляемом списком, отдельные значения данных должны быть отделены друг от друга по крайней мере одним пробелом.

Отличие между формами 1 и 2 оператора ввода, управляемого списком, заключается в том, что при использовании формы 1 предполагается, что входные данные будут поступать из *стандартного входного файла*, в то время как форма 2 позволяет считывать данные из любого файла, обозначенного как *имя-файла*.

Каждый файл, используемый в ПЛ/1-программе для ввода или вывода, имеет уникальное имя. Именем файла является последовательность знаков, начинающаяся с алфавитного знака (A — Z, @, #, \$) и содержащая только алфавитные, цифровые (0—9) знаки и (или) знак разбивки. Два файла вследствие их частого использования резервируются специально; это — **стандартный входной файл** и **стандартный выходной файл**. Первый — это просто стандартное системное устройство ввода, обычно устройство считывания с перфокарт. Второй — это стандартное системное устройство вывода, обычно печатающее устройство. Имена этих файлов зависят от реализации; для компилятора ПЛ/1(F) IBM они называются **SYSIN** и **SYSPRINT** соответственно.

Важно отметить, если используется форма 1 оператора **GET LIST**, система предполагает, что программисту нужно, чтобы данные считывались из стандартного входного файла (**SYSIN**). Если требуется любой другой файл, для его указания следует использовать форму 2.

Аналогично, **вывод, управляемый списком**, осуществляется с помощью оператора **PUT**, записанного в одной из следующих форм:

1. **PUT LIST** (список-выражений);
2. **PUT SKIP LIST** (список-выражений);
3. **PUT FILE** (имя-файла) **LIST** (список-выражений);

Здесь *список-выражений* — это несколько выражений, отделенных друг от друга запятыми. В тех случаях, когда требуется стандартный выходной файл, могут быть использованы формы 1 или 2; в остальных случаях может быть использована форма 3 для того, чтобы указать другое *имя-файла*. При выполнении оператора **PUT** значения указанных выражений передаются в выходной файл в том порядке, в котором эти выражения перечислены.

В качестве примера вновь рассмотрим описанные выше переменные **NAME** и **SCORE**, имеющие значения '**BUCK_____**' и **—14** соответственно. Предположим, что эти значения требуется напечатать. В этом случае можно воспользоваться следующим оператором:

PUT LIST (NAME, SCORE);

В результате будут напечатаны значения **BUCK_____** и **—14**.

	Позиция					
Строка	1	21	41	61	81	101
1						
2						
3						
4						
5						

Рис. 17.15.

	Позиция					
Строка	1	21	41	61	81	101
1						
2						
3						
4						
5						

Рис. 17.16.

	Позиция					
Строка	1	21	41	61	81	101
1						
2						
3						
4						
5						
6						

Рис. 17.17.

Их расположение на бумаге зависит как от текущей позиции выходного файла, так и от определяемого системой табличного формата для этого файла.

При печати, осуществляемой с помощью оператора вывода, управляемого списком, каждая строка в странице разбивается на отрезки с фиксированным, одинаковым числом горизонтальных позиций. В этой главе будем предполагать, что для 120-знаковой строки граничными позициями являются 1, 21, 41, 61, 81 и 101. Читателю предлагается узнать соответствующие позиции для используемой им реализации.

Предположим, что к моменту выполнения записанного выше оператора **PUT** текущая позиция файла печати была такой, как

показано на рис. 17.15. Тогда в результате выполнения этого оператора значения переменных **NAME** и **SCORE** будут помещены в позиции, указанные на рис. 17.16. Отметим, что текущая позиция также изменится, т. е. к моменту выполнения следующего оператора **PUT** она будет другой. Отметим также, что символьные строки печатаются без кавычек, в которые они заключены.

При выполнении формы 2 оператора **PUT** печать осуществляется от начала следующей строки. Например, с помощью оператора "**PUT SKIP LIST (NAME,SCORE);**" результат будет напечатан так, как показано на рис. 17.17. Форма 3 оператора **PUT** используется в тех случаях, когда выходные данные предназначены для файла, отличного от стандартного выходного. В этом случае файл может не печататься, а отдельные значения данных могут быть отделены друг от друга не несколькими пробелами, расположенными до следующей позиции табуляции, а одним пробелом. Позиции табуляции определяются только для выходных файлов печати.

Ввод, управляемый редактированием, осуществляется с помощью оператора **GET**, записанного в одной из следующих форм:

1. **GET EDIT** (список-переменных) (список-форматов);
2. **GET FILE** (имя-файла) **EDIT** (список-переменных) (список-форматов);

Выбор между формами 1 и 2 вновь зависит от того, должны ли входные данные считываться из стандартного входного файла или нет. *Список-переменных* здесь используется в тех же целях, что и в операторе **GET**, управляемом списком. *Список-форматов* описывает расположение и форму отдельных значений данных на носителе (например, на перфокартах), на котором они находятся, и состоит из последовательности элементов формата, отделенных друг от друга запятыми. Элемент формата выполняет одну из следующих функций:

1. Управление продвижением вперед указателя, определяющего место, начиная с которого должно быть считано следующее значение данных. Этот тип элементов формата называется **управляющим элементом формата**.

2. Описание физической формы представления входных данных на носителе (например, на перфокартах). Этот тип элементов формата называется **элементом формата данных**.

В табл. 17.4 описываются некоторые наиболее часто используемые управляющие элементы формата и элементы формата данных.

В качестве примера рассмотрим вначале следующий оператор ввода, управляемого редактированием, используемый для считывания первых двух значений из форматизованных данных.

Таблица 17.4

Управляющий элемент формата	Смысл (при вводе)
<u>COLUMN</u> (n)	Продвижение вперед к колонке (позиции) n в текущей записи (например, перфокарте), выполняемое до чтения следующего значения. Здесь n — любое целочисленное выражение. Если колонка n в текущей записи уже была пройдена, то продвижение вперед к колонке n следующей записи
X(n)	Продвижение вперед на n позиций от текущей позиции во входном потоке; n — любое целочисленное выражение, значения которого больше 0
SKIP(n)	Продвижение к началу n-й записи относительно текущей записи во входном потоке; n — любое целочисленное выражение, значения которого больше 0
Элемент формата данных	Смысл (при вводе)
F(w,d)	Арифметическое значение данных с атрибутами FIXED DECIMAL, хранимое в следующих w позициях во входном потоке. Ширина поля w, а также d могут быть любыми целочисленными выражениями; d — число цифр, начиная с крайней правой позиции в поле, которые должны восприниматься как дробная часть значения. Если значение в поле уже имеет десятичную точку, то принимается во внимание ее расположение, а d значения не имеет (см. приводимые ниже примеры). Если d = 0, вместо F(w,0) можно записать F(w) (т.е. входные значения целые)
E(w,d)	Арифметическое значение данных с атрибутами FLOAT DECIMAL, хранимое в следующих w позициях во входном потоке; d — число цифр в мантиссе, расположенных после десятичной точки. Если десятичная точка явно указана в числе, d не имеет значения. (См. приводимые ниже примеры.)
A(w)	Символьно-строчное значение данных, хранимое в следующих w позициях (без открывающей и закрывающей кавычек) во входном потоке; w — любое выражение, принимающее целые положительные значения
B(w)	Битово-строчное значение данных, хранимое в следующих w позициях (без открывающей и закрывающей кавычек и символа "B") во входном потоке; w — любое выражение, принимающее целые положительные значения

изображенных на рис. 17.14, в переменные **NAME** и **SCORE**:
GET EDIT (NAME, SCORE) (COL(1), A(10), F(3,0));

Во-первых, перечисленным переменным элементы формата данных сопоставляются в том порядке (слева направо), в котором они указаны в списке форматов (управляющие элементы формата — в данном случае **COL(1)** — в этом процессе сопоставления игнорируются). Каждой переменной сопоставляется элемент формата, используемый для управления вводом данного, которое станет ее новым значением. Во-вторых, при выполнении оператора отдельные элементы формата обрабатываются слева направо (в порядке их появления).

Например, этот оператор буквально говорит следующее: "Перейти к колонке 1 следующей перфокарты, если вы там еще не находитесь. Следующие десять колонок (1—10) содержат символьную строку, которая должна быть записана в **NAME**. Следующие три колонки содержат трехзначное целое число, которое должно быть записано в **SCORE**". Это в точности то, что выполнено в случае, если мы воспользуемся этим оператором **GET**, чтобы прочитать последовательность перфокарт (одна за другой), форматированных так, как показано на рис. 17.14. Особо отметим, что спецификация **COL(1)** необходима. Если ее опустить, при втором выполнении этого оператора **GET** считывание начнется с 14-й позиции первой перфокарты, а не с 1-й позиции второй.

Значение во входном потоке	Элемент формата	Считанное значение
1234	F(4,3)	1.234
—34	F(4,3)	0.034
—34—	F(4,3)	0.034
1.23	F(4,3)	1.230
———	F(4,3)	0.000
1234	F(4)	1234
—34	F(4)	34
—34—	F(4)	34
1.23	F(4)	1.230

При использовании средств ввода, управляемого редактированием, в ряде ситуаций могут возникнуть некоторые вопросы. Мы рассмотрим большую часть таких ситуаций и приведем соответствующие примеры, что позволит лучше понять определения различных элементов формата данных. Рассмотрим вначале некоторые значения с атрибутами **FIXED DECIMAL** и соответствующие элементы формата **F**. При считывании любого значе-

ния, не являющегося допустимой десятичной константой с фиксированной точкой и не состоящего полностью из пробелов, возникает состояние **CONVERSION**. Кроме того, атрибуты переменной, в которой *считанное значение* будет храниться, должны согласовываться с атрибутами самого значения. Например, было бы необычным считывание с помощью элемента формата **F** значения 1.234 в переменную, имеющую не арифметические, а, скажем, строковые атрибуты. Кроме того, если переменная имеет атрибуты **FIXED DECIMAL**, было бы необычным считывание значения, число значащих цифр которого не согласуется с точностью переменной, например точность (3,3) не согласовалась бы со значением 1.234. В этом случае возникнет состояние **SIZE**.

Однако и в тех случаях, когда переменная достаточно согласуется со *считанным значением*, может потребоваться его преобразование, с тем чтобы оно соответствовало атрибутам переменной, указанным в описании. Предположим, например, что значение 1,236 должно храниться в переменной **X**. В следующей таблице показаны различные случаи, в которых имеет место преобразование этого значения.

Атрибуты переменных	Результат преобразования
REAL FIXED DEC (6,3)	001.236
REAL FIXED DEC (6,2)	0001.23
REAL FIXED DEC (5,0)	00001
REAL FLOAT DEC (6)	0.123600E1

То есть значение, присваиваемое переменной, хранится в том виде, который соответствует ее атрибутам. Отметим, что конечные (справа) цифры отбрасываются без округления, и это не приводит к возникновению состояния ошибки.

Рассмотрим несколько примеров, иллюстрирующих элемент формата **E**.

Значение во входном потоке	Элемент формата	Считанное значение
1234_____	E(8,3)	1.234E0
12.34_____	E(8,3)	12.34E0
-1234E1_	E(8,3)	-1.234E1
1234_____	E(8,0)	1234E0
12.34_____	E(8,0)	12.34E0
-1234E1_	E(8,0)	-1234E1

При считывании любого значения, не являющегося допустимой десятичной константой с фиксированной или плавающей точкой, возникает состояние **CONVERSION**. В частности, если используется формат **E**, состояние **CONVERSION** возникнет при считывании значения, полностью состоящего из пробелов. Кроме того, атрибуты переменной, в которой *считанное значение* будет храниться, должны согласовываться с атрибутами самого значения, аналогично тому, как было описано для значений, считываемых с помощью формата **F**.

Наконец, рассмотрим несколько примеров, иллюстрирующих элементы формата **A** и **B**.

Значение во входном потоке	Элемент формата	Считанное значение
TITLE-17	A(8)	'TITLE-17'
TITLE-17	A(7)	'TITLE-1'
-TITLE-17	A(7)	'-TITLE-'
1011	B(4)	'1011'B
1011	B(3)	'101'B

Следует отметить два момента. Во-первых, входные значения не заключаются в кавычки. Во-вторых, значение считывается слева направо и берется столько знаков, сколько указано в элементе формата **A** или **B** с помощью величины *w*. Отметим также, что ведущие пробелы и пробелы, расположенные внутри строки, сохраняются в *считанном значении*.

Переменная, в которой *считанное значение* будет храниться, должна согласовываться с этим значением, т. е. она должна быть символьно-строчной переменной (если используется элемент формата **A**) или битово-строчной (если используется элемент формата **B**). Если длина переменной меньше длины считанного значения, от последнего отбрасывается справа необходимое число знаков. Если длина переменной больше длины считанного значения, то справа к нему добавляется необходимое число пробелов (если переменная символьно-строчная) или нулей (если переменная битово-строчная). Предположим, например, что **S** и **T** описаны следующим образом:

DCL S CHAR(4), T CHAR(10);

Если считанное значение **'TITLE-17'**, оно запишется как **'TITL'** в **S** и как **'TITLE-17__'** в **T**. Наконец, следует отметить, что любое значение во входном потоке, которое считывается с помощью элемента формата **B**, должно целиком состоять из

нулей и единиц. В противном случае возникнет состояние ошибки.

Вывод, управляемый редактированием, осуществляется с помощью оператора **PUT**, записанного в одной из следующих форм:

1. **PUT EDIT** (список-переменных) (список-форматов);

2. **PUT FILE** (имя-файла) **EDIT** (список-переменных) (список-форматов);

Выбор между формами 1 и 2 вновь зависит от того, должны ли выходные данные посылаться в стандартный выходной файл **SYSPRINT** или нет. *Список-переменных* здесь используется в тех же целях, что и в операторе **PUT**, управляемом списком. Назначение списка форматов здесь такое же, как и при вводе, управляемом редактированием: он описывает расположение и форму отдельных значений данных на носителе (например, на бумаге для печати), на который они будут помещаться при выполнении оператора **PUT**.

Различные типы управляющих элементов формата и элементов формата данных, используемых при выводе, аналогичны соответствующим элементам, используемым при вводе, управляемом редактированием. В табл. 17.5 указан смысл этих элементов. Управляющие элементы формата **PAGE** и **LINE** используются только для обработки так называемых **файлов печати**, т. е. файлов, предназначенных для принтера.

Таблица 17.5

Управляющий элемент формата	Смысл (при выводе)
PAGE	Продвижение к первой позиции (клонке) первой строки следующей страницы, выполняемое перед печатью следующего значения
LINE(n)	Продвижение вперед к первой позиции n-й строки текущей страницы, выполняемое перед печатью следующего значения. Если n-я строка ранее уже была достигнута, то продвижение к первой строке следующей страницы ¹⁾
COLUMN(n)	Продвижение вперед к n-й колонке текущей строки, выполняемое перед печатью следующего значения данных. Если n-я колонка уже была пройдена, то будет достигнута n-я колонка следующей строки. ¹⁾
X(n)	Продвижение вперед на n позиций в потоке; границы строки или страницы могут пересекаться. Продвижение вперед к началу n-й строки, следующей за текущей. SKIP(1) может быть сокращенно записано как SKIP

Элемент формата данных	Смысл (при выводе)
F(w,d)	Арифметическое значение данных с атрибутами FIXED DECIMAL будет храниться в следующих w позициях выходного потока. Значение округляется до d цифр справа от десятичной точки. Для целых чисел F(w,0) может быть сокращенно записано как F(w)
E(w,d)	Арифметическое значение данных с атрибутами FLOAT DECIMAL хранится в следующих w позициях выходного потока. Значение округляется до d цифр справа от десятичной точки, как показано ниже: $\underbrace{x.xxx \dots xE \pm uu}_w$
A(w)	Интерпретация этого значения такая же, как и для значений данных с атрибутом FLOAT. Символьная строка без открывающей и закрывающей кавычек записывается в следующие w позиций выходного потока. Значение w может быть опущено; в этом случае для спецификации A оно полагается равным длине записываемой символьной строки
B(w)	Битово-строочное значение данных без открывающей и закрывающей кавычек и символа "B" записывается в следующие w позиций выходного потока. Значение w может быть опущено; в этом случае оно полагается равным длине записываемой битовой строки
P 'спецификация-изображения'	Символьно-цифровое значение данных записывается в выходной поток, как указано спецификацией изображения

¹⁾ Эти особые действия зависят от реализации. Здесь дается описание языка ПЛ/1 (F) IBM.

Существует несколько исключительных ситуаций, которые могут возникнуть при выводе, управляемом редактированием. Мы рассмотрим наиболее общие из этих ситуаций и приведем соответствующие примеры. Сначала используем средства вывода, управляемого редактированием, для печати значений переменных NAME и SCORE, которые описаны и инициализированы следующим образом:

```
DCL NAME CHAR (10) INIT ('BUCK'),
SCORE FIXED DEC (3) INIT (-14);
```

Предположим, что текущей позицией выходного потока данных является первая позиция первой строки страницы. При выполнении оператора

```
PUT EDIT (NAME, SCORE) (A(10), X(10), F(3,0));
```


будут напечатаны значения, изображенные на рис. 17.18. Аналогичный результат будет получен в том случае, если элемент формата A(10) заменить на A (так как длина переменной NAME равна 10), элемент формата F(3,0) заменить на F(3) или элемент формата X(10) заменить на COL(21).

		Позиция														
										111			2222			
		1	2	3	4	5	6	7	8	9	0	1	2	3	...	
Строка																
	1	BUCK														-14
	2															
	3															

Рис. 17.18.

Примеры, приведенные в следующей таблице, иллюстрируют ситуации, которые могут возникнуть при выводе, управляемом редактированием. Внимательное рассмотрение этих особых ситуаций позволит читателю более глубоко их осмыслить и понять.

Хранимое значение	Элемент формата	Значение, выдаваемое на печать
1.236	F(6,3)	—1.236
1.236E2	F(7,3)	123.600
1236	F(8,3)	1236.000
1236	F(6)	—1236
1.236	F(6,2)	—1.24
1.236	F(6,1)	—1.2
1.236	E(12,6)	0.123600E+—1
1.236E2	E(12,3)	—0.124E+—3
'BUCK'	A(6)	BUCK—
'BUCK'	A(3)	BUC
'101'B	B(6)	101—
'101'B	B(2)	10

Попытка вывести любое значение, не соответствующее предусмотренному элементу формата, как, например, 'BUCK' или 1000.5 для элемента формата F(6,3), приведет к возникновению состояния **ERROR**.

Перед тем как завершить наше рассмотрение ввода-вывода, управляемого редактированием, укажем несколько важных обстоятельств. Во-первых, если число элементов формата данных превосходит число значений, которые должны быть считаны (записаны), то лишние элементы формата данных и распо-

женные между ними управляющие элементы формата игнорируются. Например, при выполнении оператора

```
PUT EDIT (NAME, SCORE) (A,F(3), COL(50), A);
```

обведенные элементы в списке форматов игнорируются. Аналогично, если за элементом формата данных, соответствующим последнему значению, которое должно быть напечатано (в данном случае **SCORE**), следует управляющий элемент формата, последний также игнорируется независимо от того, следует за ним элемент формата данных или нет. В предыдущем примере элемент **COL(50)** игнорируется даже в том случае, если опущено **“A”**, непосредственно следующее за ним.

Во-вторых, если число элементов формата данных меньше числа значений, которые должны быть считаны (записаны), то список форматов просматривается заново. Например, действия операторов

```
PUT EDIT (NAME, SCORE, NAME, SCORE) (A, F(3));
```

и

```
PUT EDIT (NAME, SCORE, NAME, SCORE) (A, F(3), A, F(3));
```

эквивалентны.

В-третьих, для повторения некоторой части списка форматов она заключается в скобки и перед ней пишется целочисленное выражение, указывающее число требуемых повторений. Например, действия операторов

```
PUT EDIT (NAME, SCORE, SCORE, NAME) (A, F(3), F(3), A);
```

и

```
PUT EDIT (NAME, SCORE, SCORE, NAME) (A, 2_F(3), A);
```

эквивалентны. Скобки используются в том случае, если повторяемая часть состоит более чем из одного элемента. Отметим, что перед повторяемой частью должен быть пробел ().

Наконец, перед тем как завершить рассмотрение потокоориентированного ввода-вывода, следует сделать одно общее замечание. При потокоориентированном вводе-выводе массив или структура могут быть считаны (записаны) различными способами. Для их рассмотрения опишем массив **B** и структуру **S** следующим образом:

```
DCL B (5,4) FIXED DEC (2,0),
      1 S,
      2 T FIXED DEC (3,2),
      2 U CHAR (5),
      2 V FLOAT DEC (6);
```

Для того чтобы прочитать следующие 20 значений в **B**, а затем следующие три значения в **S**, можно использовать один из следующих операторов, действия которых эквивалентны:

1. GET LIST (B(1,1)B(1,2),B(1,3),B(1,4),B(2,1),...,B(5,4),T,U,V);
2. GET LIST (((B(I,J) DO J=1 TO 4) DO I=1 TO 5),T,U,V);
3. GET LIST ((B(I,*) DO I=1 TO 5),S);
4. GET LIST (B,S);

Во втором примере показано, как можно использовать спецификацию **DO** для сокращения ввода-вывода. В третьем примере показано, как можно использовать сечение массива и (или) имя структуры для дальнейшего сокращения списка, в котором имена всех рассматриваемых переменных указаны явно. Четвертый пример показывает наиболее краткую форму записи и иллюстрирует тот факт, что при отсутствии явных спецификаций **DO** для вводимого или выводимого массива его элементы обрабатываются по строкам. В каждом из этих четырех случаев 23 значения из входного потока будут присвоены элементам массива **B** и структуры **S** одним и тем же способом. Если бы вместо ввода, управляемого списком, использовался ввод, управляемый редактированием, то независимо от способа задания **B** и **S** в списке данных был бы выписан список форматов, управляющий формой и расположением всех 23 считываемых значений.

Записеориентированный ввод-вывод

В этом разделе мы рассмотрим два различных способа доступа к файлам с помощью записеориентированного ввода-вывода: **последовательные** ввод и вывод и **прямые** ввод, вывод и обновление. При последовательном доступе записи в файле обрабатываются в том порядке, в котором они хранятся, по одной, начиная с первой. При прямом доступе записи в файле обрабатываются не в заранее установленном порядке, а так, как указано в программе. Так, при прямом доступе файл может быть считан или записан произвольное число раз за один прогон программы.

Для того чтобы определить атрибуты файла, его нужно описать. Описание файла имеет следующую форму:

$$\text{DCL имя-файла FILE RECORD } \left\{ \begin{array}{l} \text{SEQUENTIAL} \\ \text{DIRECT} \end{array} \right\} \left\{ \begin{array}{l} \text{INPUT} \\ \text{OUTPUT} \\ \text{UPDATE} \end{array} \right\}$$

Здесь *имя-файла* — имя файла, а ключевые слова **FILE** и **RECORD** (в противоположность **STREAM**) указывают, что опре-

делен файл для записеориентированного ввода-вывода в противоположность потокоориентированному. Описатели **SEQUENTIAL** (последовательный) и **DIRECT** (прямой) задают способ доступа к файлу, а описатели **INPUT**, **OUTPUT** и **UPDATE** указывают, что файл предназначается для ввода, вывода и обновления. В последнем случае при одном и том же прогоне программы записи могут как считываться из файла, так и записываться в него.

Для обработки последовательного входного файла записей может быть использован следующий оператор:

READ FILE (имя-файла) INTO (переменная);

Здесь *имя-файла* определяет сам файл, а *переменная* указывает место в памяти, где будет храниться содержимое записи. Предположим, например, что файл **CARDS** и переменная **CARD** описаны следующим образом:

**DCL CARDS FILE RECORD INPUT SEQUENTIAL,
CARD CHAR (80);**

Тогда в результате выполнения оператора

READ FILE (CARDS) INTO (CARD);

следующая запись файла **CARDS** будет считана и записана в переменную **CARD**.

Предполагается, что длина записи (т. е. одной перфокарты) составляет 80 знаков. Длина переменной, в которую считывается входная запись, всегда должна быть равна длине этой записи в файле. Кроме того, эта переменная не обязательно должна быть описана как символьная (с атрибутом **CHARACTER**). Для размещения одной входной записи часто бывает удобно использовать структуру.

Колонка	1-10	11-15	16-30	31-32	33	34-35	36-38	39-47	48-80
	Имя сотрудника			Возраст		Рост		Вес	Номер страхового талона
	Фамилия	Имя	Отчество		Футы	Дюймы			

Рис. 17.19.

Предположим, например, на перфокарте содержится информация о сотруднике, указанная на рис. 17.19. Одну такую за-

пись можно разместить в следующей структуре:

```
DCL 1 PERSON,  
  2 NAME,  
    3 FIRST CHAR (10),  
    3 MIDDLE CHAR (5),  
    3 LAST CHAR (15),  
  2 AGE PIC '99',  
  2 HEIGHT,  
    3 FEET PIC '9',  
    3 INCHES PIC '99',  
  2 WEIGHT PIC '999',  
  2 SS# PIC '(9)9',  
  2 REST CHAR (33);
```

В результате выполнения следующего оператора **READ**
READ FILE (CARDS) INTO (PERSON);

запись считается в эту структуру, и все элементарные поля последней заполняются значениями.

Важно отметить, что при выполнении записеориентированных операций ввода-вывода преобразование данных не осуществляется. Таким образом, программист должен сам позаботиться о том, чтобы элементы данных, содержащихся в записи, соответствовали атрибутам переменных, в которых эти данные будут храниться после выполнения операции ввода-вывода, такой, как, например, **READ**. В реализациях IBM 360/370 внутреннее представление значений с атрибутами **FIXED (FLOAT) DECIMAL (BINARY)** не идентично их внешнему представлению; для получения одного из другого требуется преобразование. Таким образом, записеориентированный ввод-вывод подходит для таких значений только в том случае, если в файл пишется их внутреннее представление. В большинстве случаев это означает, что переменные должны быть описаны как символьные строки (**CHAR**) или символично-цифровые данные (**PICTURE**).

Для обработки последовательного выходного файла записей может быть использован следующий оператор:

WRITE FILE (имя-файла) FROM (переменная);

Здесь *имя-файла* и *переменная* определяют соответственно файл и место памяти, откуда будет выбрана запись. Например, для вывода образов перфокарт выходной файл можно описать следующим образом:

DCL IMAGES FILE RECORD OUTPUT SEQUENTIAL;

Тогда для вывода образа одной перфокарты из переменной **CARD**, описанной выше, может быть использован следующий оператор **WRITE**:

WRITE FILE (IMAGES) FROM (CARD);

При обработке файла записей с прямым доступом для ввода, вывода или обновления каждая запись должна иметь соответствующий ей ключ, который используется для того, чтобы однозначно определить эту запись и таким образом отличить ее от других записей в файле. Предположим например, что к нашему файлу с записями, изображенными на рис. 17.19, должен быть организован прямой доступ и что элемент **SS#** (позиции 39—47) используется в качестве ключа записи.

Для обработки прямого файла записей, предназначенного для ввода, в операторе **READ** необходимо не только определить имя файла и место в памяти, в которое входная запись должна быть считана, но и указать требуемую запись с помощью ее ключа. Таким образом, оператор **READ** имеет следующую форму:

READ FILE (имя-файла) **INTO** (переменная) **KEY** (выражение);

Выражение вычисляется для определения ключа требуемой записи. Например, если из нашего файла (рис. 17.19) требуется считать запись по ключу **SS#** = '025308235', надо записать следующее:

READ FILE (CARDS) INTO (PERSON) KEY ('025308235');

Если в этом файле существует запись с указанным ключом, она будет найдена и помещена в структуру **PERSON**; в противном случае возникнет состояние **KEY**.

При обработке прямого файла записей, предназначенного для вывода, используется оператор **WRITE**. В операторе **WRITE** также должно быть определено значение ключа, которое будет храниться в записи. Таким образом, оператор **WRITE** имеет следующую форму:

WRITE FILE (имя-файла) **FROM** (переменная) **KEYFROM** (выражение);

Значение *выражения* рассматривается как символьная строка и помещается в ту часть записи, где хранится ключ. Например, если требуется добавить к нашему файлу запись, которая содержится в структуре **PERSON**, с ключом '025308235', надо записать следующее:

WRITE FILE (CARDS) FROM (PERSON) KEYFROM ('025308235');

Если в этом файле уже существует запись с таким же ключом, то возникает состояние **KEY**.

При обработке прямого файла записей, предназначенного для обновления, могут выполняться следующие три элементарные операции:

1. Изменение существующей записи в файле.
2. Добавление новой записи к файлу.
3. Удаление существующей записи из файла.

Первая операция выполняется с помощью следующей пары операторов:

READ FILE (имя-файла) **INTO** (переменная) **KEY** (выражение);

REWRITE FILE (имя-файла) **FROM** (переменная) **KEY** (выражение);

Между этими двумя операторами могут находиться другие операторы, которые осуществляют некоторые изменения записи, хранимой в *переменной*. В любом случае запись, которая окончательно переписывается из *переменной*, физически замещает запись, которая была только что считана.

Вторая операция, добавление записи, выполняется с помощью обычного оператора **WRITE** следующим образом:

WRITE FILE (имя-файла) **FROM** (переменная) **KEYFROM** (выражение);

Программа должна быть составлена таким образом, чтобы *выражение*, указывающее ключ новой записи, отличалось от других ключей в файле. В противном случае возникнет состояние **KEY**.

Третья операция, удаление записи, выполняется с помощью следующего оператора:

DELETE FILE (имя-файла) **KEY** (выражение);

В результате выполнения этого оператора из файла удаляется запись, ключ которой совпадает с *выражением*. Если такой записи не существует, возникает состояние **KEY**.

Состояния, возникающие при вводе-выводе

В предыдущих разделах отмечалось, что в определенных случаях при выполнении операций ввода-вывода в ПЛ/1-программе могут возникать особые состояния. В этом разделе будут рассмотрены наиболее важные из них, а именно состояния **ENDFILE**, **ENDPAGE**, **KEY** и **TRANSMIT**.

Состояние **ENDFILE** возникает в тех случаях, когда должна выполняться операция ввода с последовательным доступом

(оператор **READ** или **GET**), но в файле больше не осталось данных для чтения.

Состояние **ENDPAGE** возникает в тех случаях, когда при выполнении операции последовательного потокоориентированного вывода (оператор **PUT**) для файла печати возникает переход за пределы последней строки текущей страницы.

Состояние **KEY** может возникнуть в тех случаях, когда должна выполняться операция ввода или вывода записи с прямым доступом. Различные ситуации, в которых возникает состояние **KEY**, определены в предыдущем разделе.

Состояние **TRANSMIT** может возникнуть при выполнении любой операции ввода или вывода, как записеориентированного, так и потокоориентированного. Оно означает возникновение некорректируемой ошибки передачи данных при попытке чтения (записи) текущей записи из файла (в файл), т. е. запись была неправильно передана с устройства в память (из памяти в устройство).

Если для данного файла возникает какое-либо из указанных выше состояний, результирующее действие зависит от того, содержит ли программа так называемый *ОН-элемент* для этого состояния и этого файла. Если такого *ОН-элемента* в программе нет, при возникновении этих состояний выполняются действия, указанные в табл. 17.6.

Таблица 17.6

Состояние	Действие, выполняемое при отсутствии ОН-элемента
ENDFILE KEY TRANSMIT	Печатается сообщение о возникновении данного состояния, после чего возникает состояние ERROR
ENDPAGE	
	Осуществляется переход на начало следующей страницы и затем возобновляется выполнение оператора PUT с того места, в котором возникло состояние ENDPAGE

Если программист желает, чтобы при возникновении какого-либо из этих состояний выполнялось другое действие, он может воспользоваться так называемым **оператором ОН**, который имеет следующую форму:

ОН состояние (имя-файла) ОН-элемент;

Здесь *состояние* — одно из состояний **ENDFILE**, **ENDPAGE**, **KEY** или **TRANSMIT**, а *имя-файла* определяет конкретный файл, для которого отслеживается возникновение указанного

состояния. Например, если в программе используются два различных последовательных входных файла, то для обработки прерывания по возникновению состояния **ENDFILE** могут быть использованы два различных оператора **ON**. *ON-элемент* определяет действие, которое должно выполняться при возникновении указанного состояния с указанным файлом. *ON-элемент* может быть либо отдельным оператором, либо последовательностью операторов, начинающейся с "**BEGIN;**" и заканчивающейся "**END;**". Такая последовательность называется **блоком BEGIN**.

Предположим например, что при возникновении состояния **ENDPAGE** для файла **SYSPRINT** нужно перейти на начало следующей страницы и до возобновления печати вывести следующий заголовок:

1 позиция ↑ — СПИСОК ВХОДНЫХ ДАННЫХ	91 позиция ↑ — СТРАНИЦА xxx
---	-----------------------------------

Здесь страницы будут нумероваться в правом верхнем углу. Предположим, что программа должна распечатать входную коду перфокарт. Тогда эта программа может выглядеть следующим образом:

```

LISTER: PROC OPTIONS(MAIN):
  DCL CARD CHAR (80), PNO FIXED BIN (31) INIT (0);
  ON ENDFILE (SYSIN) STOP;
  ON ENDPAGE (SYSPRINT) BEGIN;
    PNO = PNO + 1;
    PUT PAGE EDIT ('СПИСОК ВХОДНЫХ ДАННЫХ',
      'СТРАНИЦА', PNO) (A, COL(91), A, F(4));
    PUT SKIP;
  END;
  SIGNAL ENDPAGE (SYSPRINT);
  DO I=1 BY 1;
    GET EDIT (CARD) (A(80));
    PUT SKIP EDIT (CARD) (A(80));
  END;
END LISTER;

```

Каждый раз, когда должен выполняться оператор **PUT**, но его часть **SKIP** вызывает выход за пределы последней строки страницы, возникает состояние **ENDPAGE**. Оператор **ON** определяет предусмотренное на этот случай действие, которое заключается в увеличении номера страницы (**PNO**) и печати заголовка в верхней части следующей страницы.

Если *ON-элемент* не содержит ни оператор **STOP**, ни оператор **GO TO**, который передал бы управление другой части про-

граммы, то непосредственно после выполнения этого ON-элемента осуществляется *нормальный возврат* управления оператору, расположенному около того места программы, где первоначально возникло это состояние. Точное расположение точки нормального возврата зависит от состояния, как это показано в табл. 17.7.

Таблица 17.7

Состояние	Точка нормального возврата
ENDFILE	Должен выполняться оператор, непосредственно следующий за оператором GET или READ, при выполнении которого возникло состояние ENDFILE
ENDPAGE	Выполнение оператора PUT, вызвавшего возникновение состояния ENDPAGE, возобновляется так, как будто этого состояния не возникало
KEY	Должен выполняться оператор, непосредственно следующий за оператором (READ WRITE, REWRITE или DELETE), при выполнении которого возникло состояние KEY. Однако операция ввода-вывода в действительности не будет выполнена
TRANSMIT	Должен выполняться оператор, непосредственно следующий за оператором ввода-вывода, при выполнении которого возникло состояние TRANSMIT. Однако операция ввода-вывода в действительности не будет выполнена

Таким образом, в нашем примере в результате нормального возврата из ON-элемента **ENDPAGE** управление вновь будет передано расположенному внутри цикла **DO** оператору **PUT**, при выполнении которого возникло состояние **ENDPAGE**. Строка, которая должна быть напечатана, будет выведена сразу же после заголовка на следующей странице.

В заключение рассмотрим оператор **SIGNAL**, содержащийся в программе. В результате его выполнения имитируется указанное состояние (в данном случае **ENDPAGE (SYSPRINT)**), даже если оно и не возникало в действительности. Таким образом, этот оператор **SIGNAL** позволяет печатать заголовок сверху первой страницы.

17.5. ПОДПРОГРАММЫ

На практике в большинстве случаев размеры программы бывают весьма большими и становится оправданным ее разбиение на несколько функциональных блоков. Они объединяются с помощью *основной* программы, которая управляет последова-

тельностью выполнения функциональных блоков. Другим преимуществом подпрограмм является то, что они могут использоваться многократно и при этом не требуется каждый раз их переписывать заново.

ПЛ/1 располагает средствами, позволяющими осуществлять такое разбиение программы. Подпрограмма в ПЛ/1 — это процедура, которая может быть вызвана при выполнении основной программы [определенной как **PROCEDURE OPTIONS(MAIN)**] или другой подпрограммы. В ПЛ/1 имеется два вида подпрограмм: **процедуры-функции** и **процедуры**. В каждом из этих случаев существуют два круга проблем, связанных с использованием подпрограмм: написание самой подпрограммы и вызов подпрограммы для выполнения ее функций.

Некоторые функции используются настолько часто, что в языке ПЛ/1 были предусмотрены подпрограммы, вычисляющие их. Они называются **встроенными функциями** и записаны как

Таблица 17.8. Некоторые встроенные арифметические функции в ПЛ/1

Функция	Имя	Аргументы	Результаты
1. Абсолютное значение	ABS	x	Абсолютное значение аргумента x , где x — любое арифметическое выражение
2. Ближайшее большее значение	CEIL	x	Наименьшее целое число i , большее или равное значению x
3. Косинус	COS	x	Косинус числа x , где x — угол, измеренный в радианах
4. Экспонента	EXP	x	Результат возведения числа e в степень x
5. Натуральный логарифм	LOG	x	Логарифм (по основанию e) числа x , где $x > 0$
6. Десятичный логарифм	LOG10	x	Логарифм (по основанию 10) числа x , где $x > 0$
7. Максимальное значение	MAX	x_1, x_2, \dots, x_n	Наибольшее среди n ($n \geq 2$) заданных значений
8. Минимальное значение	MIN	x_1, x_2, \dots, x_n	Наименьшее среди n ($n \geq 2$) заданных значений
9. Остаток (модуль)	MOD	x_1, x_2	Остаток от деления x_1 на x_2
10. Синус	SIN	x	Синус числа x , где x — угол, измеренный в радианах
11. Квадратный корень	SQRT	x	Квадратный корень числа x , где $x \geq 0$

процедуры-функции. Поэтому, когда требуется какая-нибудь из этих функций, например вычисление квадратного корня чйсла, программист должен только вызвать ее. Полный список встроенных функций можно найти в соответствующем справочном руководстве по ПЛ/1. В табл. 17.8, 17.9 и 17.10 описаны те из них, которые являются наиболее полезными при решении широкого круга задач.

Таблица 17.9. Некоторые встроенные функции над массивами в ПЛ/1

Функция	Имя	Аргументы	Результат
1. Верхняя граница	HBOUND	a, i	Наибольшее допустимое значение индекса для i-го измерения массива a
2. Произведение	PROD	a	Произведение всех элементов массива a
3. Сумма	SUM	a	Сумма всех элементов массива a

Таблица 17.10. Некоторые встроенные строковые функции в ПЛ/1

Функция	Имя	Аргументы	Результат
1. Индекс	INDEX	s, t	Если строка t входит в строку s, то результат — целое число i, определяющее позицию самого левого знака, начиная с которого t входит в s. В противном случае результат 0
2. Длина	LENGTH	s	Длина (число знаков) строки s
3. Подстрока	SUBSTR	s, i, l	Подстрока строки s, начинающаяся с i-й позиции и имеющая длину l. Если l опущено, то подстрока включает весь остаток строки s, начиная с i-й позиции
4. Проверка	VERIFY	s, t	Если все знаки строки s входят также в строку t, то результат 0. В противном случае результатом является целое число i > 0, определяющее позицию самого левого знака в s, не входящего в t

Аргументами каждой из данных встроенных арифметических функций (табл. 17.8) могут быть выражения, принимающие числовые, а не строковые значения. Таким образом, различные функции являются общими в том смысле, что их аргументы могут иметь атрибуты **FIXED** или **FLOAT**, **DEC** или **BIN** и т. д. Результат, возвращаемый каждой из этих функций, обычно имеет атрибуты, которые согласуются с атрибутами аргументов и видом функции.

Аналогично встроенные функции над массивами являются общими в том смысле, что аргументом каждой из них может быть массив значений, имеющих числовые атрибуты. Кроме того, одним из аргументов функции **HBOUND** может быть любой массив, включая массив строк. Результат, возвращаемый этой функцией, является целым числом. Результат, возвращаемый каждой из последних двух функций, имеет такие же атрибуты, какие имеет сам массив.

Отметим, что общим свойством всех этих встроенных функций является то, что они возвращают не несколько, а одно-единственное значение. Это требование связано со спецификой их вызова.

Вызов функции должен осуществляться с помощью *обращения* к ней на месте переменной или константы в выражении. Обращение к функции всегда имеет следующую форму:

имя (аргументы)

Здесь *имя* — имя функции, а *аргументы* — список выражений (отделенных друг от друга запятыми), задающих значения аргументов функции. Вызванная таким способом функция возвращает одно-единственное значение (число или строку), которое затем используется в качестве операнда при вычислении выражения, содержащего обращение к этой функции. Предположим, например, что имеются переменные **A**, **B** и **C** с атрибутом **FLOAT**, значения **A** и **B** равны 2 и 3 соответственно и требуется вычислить **C** следующим образом:

$$C = \frac{A + B}{\sqrt{A^2 + B^2}}$$

В этом случае можно использовать встроенную функцию **SQRT** следующим образом:

$$C = (A + B) / \text{SQRT}(A^{**}2 + B^{**}2)$$

↑ ↑
↑ ↑ ↑

① ③
② ④ ③

Как видно, при обращении к функции вычисляется выражение **A**2+B**2**, значение которого (13) передается в качестве ар-

гумента функции **SQRT**. Возвращаемый результат, имеющий атрибут **FLOAT**, является близким приближением квадратного корня из 13, скажем 3.60555. При дальнейшем вычислении выражения это значение становится делителем.

Для того чтобы показать, как могут быть использованы некоторые другие встроенные функции, предположим, что заданы переменные и их значения, изображенные на рис. 17.20.

Переменная	Атрибуты	Значения
X	FLOAT (6)	-3,5
Y	"	+2,9
I	FIXED BIN (31)	17
A	(5) FLOAT (6)	8 3 0 -1 5
B	(5,4) FLOAT (6)	1 -1 2 0 7 2 -1 2 9 2 4 -1 6 7 8 -7 3 1 0 2
S	CHAR (25)	THESE ARE THE TIMES

Рис. 17.20.

В табл. 17.11 приводятся результаты, получаемые при обращении к этим функциям.

Таблица 17.11

Обращение к функции	Получаемый результат
COS(Y)	-0.970958E0 (2,9 рад \approx 166 град)
SUM(A)	15
SUM(A*B(*,3))	"Внутреннее произведение" A и 3-го столбца матрицы B, т.е. $A_1*B_{13} + A_2*B_{23} + A_3*B_{33} + A_4*B_{43} + A_5*B_{53} = 5$
SUBSTR(S, I, 6)	IMES.
SUBSTR(S, I)	IMES.
INDEX(S, 'THE')	2
INDEX(S, 'THE-')	12
INDEX(S, '?')	0
VERIFY(S, '-')	2 (позиция в строке S самого левого отличного от пробела знака)
INDEX(SUBSTR(S, VERIFY(S, '-')), '-')	6 (позиция самого левого пробела в подстроке, которая начинается с 'THESE-ARE...')

Отметим, что последние два примера показывают, как встроенные функции **INDEX** и **VERIFY** могут быть использованы для выделения слова (т. е. последовательности знаков, за которой следует пробел) из текста. В частности, с помощью функции **VERIFY** определяется расположение самого левого знака слова, а с помощью функции **INDEX** — самого правого.

Написание определений функций

Если в программе требуется вычислить функцию, которой нет среди встроенных функций, программист может определить **процедуру-функцию** и затем вызывать ее точно так же, как и встроенную функцию. Поскольку функция, написанная программистом, вызывается так же, как и встроенная функция, результатом ее вычисления всегда должно быть одно-единственное значение. Подпрограммы, в которых в качестве результата вычисляется несколько различных значений, должны быть записаны как процедура. Этот вопрос будет рассмотрен позже.

Для того чтобы определить процедуру-функцию, программист должен (1) указать параметры, которые потребуются для вычисления результата; (2) определить атрибуты этих параметров, а также атрибуты вычисляемого значения функции; (3) описать алгоритм, по которому должны проводиться вычисления; (4) оформить все это в виде процедуры-функции языка ПЛ/1.

Первые три требования должны быть выполнены независимо от используемого языка программирования. Рассмотрим в качестве примера следующую задачу.

Написать процедуру-функцию, вычисляющую факториал любого целого числа n .

Как видно, здесь требуется только один параметр — целое число n . Факториал определяется следующим образом:

$$\text{fact}(n) = \begin{cases} n \cdot (n-1) \cdot \dots \cdot 3 \cdot 2, & \text{если } n > 1, \\ 1, & \text{если } n \leq 1. \end{cases}$$

Таким образом, возвращаемым результатом является целое число, равное либо 1 (если $n \leq 1$), либо произведению первых n положительных целых чисел (если $n > 1$).

Функция в ПЛ/1 всегда начинается с оператора **PROCEDURE**, имеющего следующую форму:

имя-входа: **PROCEDURE** [(список-параметров)] [**RETURNS** (атрибуты)];

Здесь *имя-входа* — имя процедуры, а необязательный список параметров — список параметров, отделенных друг от друга за-

пятой. Атрибуты возвращаемого результата либо указываются в части *атрибуты* необязательной статьи **RETURNS**, либо назначаются по умолчанию, если она опущена. Правила назначения атрибутов возвращаемому значению по умолчанию аналогичны правилам назначения атрибутов по умолчанию обычным переменным.

За оператором процедуры следует тело процедуры, определяющее вычисления, которые будут выполняться при вызове функции. Функция выглядит аналогично обычному сегменту программы, предназначенному для той же цели, за исключением того, что вместо имен переменных, используемых в сегменте программы, пишутся параметры. Тело процедуры всегда заканчивается оператором **END**, имеющим следующую форму:

END имя-входа;

Здесь *имя-входа* — вновь имя функции, которое совпадает с именем, указанным в операторе **PROCEDURE**.

В качестве примера напомним процедуру-функцию **FACT**, вычисляющую факториал целого числа n . Здесь мы определяем один параметр, **N**, указывающий число, факториал которого будет вычисляться.

```
FACT: PROC (N) RETURNS(FIXED BIN);  
      DCL (F, I, N) FIXED BIN;  
      F = 1;  
      DO I = 2 TO N;  
        F = F*I;  
      END;  
      RETURN(F);  
END FACT;
```

Отметим, что вычисления, указанные в теле процедуры, выполняются так же, как в случае, когда **N** — обычная переменная. При вызове функции вместо **N** будет подставлено некоторое значение. Этот процесс будет описан в следующем разделе.

В тех местах, где вычисленный результат следует возвратить вызывающей программе, функция должна содержать оператор **RETURN**, имеющий следующую форму:

RETURN (выражение);

Результат вычисления *выражения* возвращается в вызывающую программу. Поскольку этот результат должен быть единственным значением, то и выражение должно быть таким, которое вырабатывает одно значение, арифметическое или строковое.

Как видно, в нашем примере вызывающей программе возвращается вычисленное значение переменной F. Как F, так и N являются локальными переменными, используемыми в функции для вычисления требуемого результата.

Вызов функций

Функции, которые пишутся программистом, вызываются точно так же, как и встроенные функции, с помощью обращения к функции. Однако функции, определенные программистом, обладают некоторыми дополнительными особенностями, которые рассматриваются в этом разделе.

Предположим, что требуется вычислить биномиальные коэффициенты

$$a_i = \frac{n!}{i!(n-i)!}, \quad i = 0, 1, \dots, n,$$

для хорошо известного полинома

$$(x + y)^n = a_n x^n + a_{n-1} x^{n-1} y + \dots + a_i x^i y^{n-i} + \dots + a_0 y^n$$

Можно написать следующую программу, которая печатает требуемую последовательность коэффициентов A:

```
P: PROC OPTIONS(MAIN);
  DCL (A, I, N) FIXED BIN;
  FACT ENTRY RETURNS (FIXED BIN);
  GET LIST (N);
  DO I=0 TO N;
    A = FACT(N)/(FACT(I)*FACT(N - I));
    PUT SKIP LIST (I, A);
  END;
END P;
```

Обратим внимание на две подчеркнутые строки в этой программе. В первой из них определяются атрибуты результата, возвращаемого функцией **FACT**. Это требуется в тех случаях, когда выполнены следующие условия:

- Определение функции является внешним по отношению к процедуре, вызывающей ее.

Результат, возвращаемый функцией, имеет атрибуты, отличные от тех, которые были бы назначены по умолчанию. Если определение функции является *внутренним* (т. е. не внешним) по отношению к процедуре, которая ее вызывает, то атрибуты возвращаемого результата известны системе заранее. Таким образом, данное здесь описание **ENTRY** в этом случае не потребовалось бы.

Во втором подчеркнутом операторе показаны три различных вызова функции **ФАСТ**. Каждый аргумент, задаваемый в обращении к функции, может быть любым выражением, как, например, " $N - I$ " в третьем вызове. Эти выражения вычисляются до того, как вызывается функция. Обращение к функции всегда должно содержать в точности столько параметров, сколько их указано при определении функции. Между аргументами и параметрами устанавливается взаимно однозначное соответствие слева направо.

При вызове функции выполняются следующие шаги:

1. Каждый параметр в теле процедуры-функции заменяется соответствующим аргументом или его значением.

2. Полученное в результате такой замены тело процедуры выполняется так же, как и в случае, если бы оно было обычной секцией вызывающей программы.

3. Возвращаемым результатом является значение выражения, записанного в находящемся в теле процедуры операторе **RETURN**. Впоследствии этот возвращаемый результат используется при вычислении выражения, в котором содержится обращение к функции.

К этому довольно сжато описанию следует добавить ряд пояснений. Вначале проследим выполнение приведенной выше программы. Предположим, что в результате выполнения оператора **GET** переменной **N** присваивается значение, равное 5. Тогда выходом этой программы будут следующие значения:

0	1
1	5
2	10
3	10
4	5
5	1

Рассмотрим последнее выполнение оператора присваивания (при $I = 5$), содержащего три обращения к нашей функции. При вычислении арифметического выражения первым обращением к функции **ФАСТ** является **ФАСТ(I)**. Поскольку значение аргумента **I** равно 5, выполнение процедуры-функции заключается в вычислении факториала 5, который равен 120. Затем это возвращаемое значение подставляется в исходное арифметическое выражение так, как будто оператор присваивания был записан следующим образом:

$$A = \text{ФАСТ}(N) / (120 * \text{ФАСТ}(N - I));$$

Следующим обращением к функции **ФАСТ** является **ФАСТ(N - I)**. Значение аргумента, равное 0 (текущее значение выражения $N - I$), передается функции, заменяя параметр **N**. Затем возвращаемый результат, 1, подставляется на место

FACT(N — 1) для последующего вычисления выражения. После вычисления произведения 120×1 выполняется последний вызов функции **FACT** (обращение **FACT(N)**), в результате которого возвращается значение 120. Затем вычисляется арифметическое выражение и величина частного, равная 1, присваивается переменной **A**.

Важный момент, который следует помнить при написании и вызове функций, определенных программистом, заключается в том, что вызывающая программа должна удовлетворять следующим двум условиям:

1. Все атрибуты каждого аргумента согласуются с атрибутами соответствующего ему параметра.

2. Если какие-либо атрибуты одного или более аргументов не согласуются с атрибутами соответствующих им параметров, то для указания этого несоответствия в вызывающей программе должно содержаться описание входа. Это позволит выполнить соответствующее преобразование в момент вызова.

Описание входа является немного более общим, чем то, которое подчеркнуто в приведенном выше примере. Оно имеет следующую форму:

DCL имя-входа **ENTRY** [(список-атрибутов)] [**RETURNS** (атрибуты)];

Здесь *список-атрибутов* — список наборов атрибутов; каждый набор отделяется от следующего запятой. Число наборов атрибутов в этом списке совпадает с числом параметров функции. Каждый такой набор должен согласовываться с атрибутами соответствующего ему параметра, указанными в определении функции.

Поскольку в нашей программе в каждом из трех обращений аргументы имеют атрибуты **FIXED BINARY**, то выполняется первое условие. Поэтому описание входа указанного выше вида не требуется. Предположим теперь, что требуется вызвать функцию **FACT** для вычисления факториала некоторого числа, например 5. Тогда для выполнения этого вычисления мы могли бы записать

FACT(5)

Однако аргумент в этом обращении к функции имеет атрибуты **FIXED DECIMAL (1,0)**. Для того чтобы указать, что вследствие такого несоответствия необходимо преобразование, требуется следующее новое описание входа для функции **FACT**:

DCL FACT ENTRY (FIXED BIN) RETURNS(FIXED BIN);

Второй важный момент, который следует иметь в виду при написании и вызове функций, определенных программистом, заключается в том, что преобразование должно быть возможным в случае, когда атрибуты аргумента не согласуются с атрибутами соответствующего параметра. В частности, это означает, что аргумент не может быть массивом или структурой, если соответствующий параметр является элементарной переменной, и наоборот, аргумент должен быть массивом или структурой, если таковым является соответствующий параметр.

Кроме того, важны и некоторые другие моменты, касающиеся соответствия между аргументами и параметрами. На них мы остановимся при рассмотрении примеров определения и вызова процедур. Другие детали читателю станут очевидными, когда он изучит ПЛ/1. Наилучшим подходом к определению и вызову подпрограмм является, по-видимому, консервативный, который заключается в следующем. В тех случаях, когда определяемая функция является внешней по отношению к вызывающей программе, следует давать полное описание входа. Кроме того, необходимо, чтобы аргументы подставлялись в нужном порядке. Наконец, при определении каждой процедуры-функции следует дать точное описание каждого параметра, а также результата.

Определение процедур

Процедура не возвращает значение вызывающей программе, как это делает функция. Результатом вызова процедуры может быть некоторое количество элементарных значений, массивов или структур. Для обозначения результата в определении процедуры используются дополнительные параметры. Обычно эти параметры указываются последними в операторе **PROCEDURE**, с которого начинается определение процедуры.

Помимо указанного выше существует еще одно незначительное различие между определением процедуры и определением функции. В определении процедуры оператор **RETURN** не должен содержать выражения (поскольку возвращаться может не один результат). Вместо этого, те параметры, которые служат для обозначения результата, возвращаемого процедурой, записываются в нужных местах тела процедуры, с тем чтобы во время вызова соответствующим им аргументам могли присваиваться значения.

Проиллюстрируем эти соглашения, переписав функцию **ФАКТ** как процедуру. В этом случае для обозначения вычисленного значения факториала используется дополнительный параметр **F**. Напомним, что в первоначальном определении функции переменная **F** была обычной.

```

FACT: PROC (N, F);
    DCL (N, F, I) FIXED BIN;
    F = 1;
    DO I = 2 TO N;
        F = F*I;
    END;
    RETURN;
END FACT;

```

Сравнивая эту процедуру с первоначальным определением функции, можно заметить, что оператор **RETURN** в последнем случае сокращен. Он указывает то место, в котором управление будет возвращено вызывающей программе. Отметим также, что статья **RETURNS** была опущена, так как для процедур она теряет смысл.

Приведем пример, иллюстрирующий два дополнительных соглашения, касающиеся процедур. А именно, будет дано определение процедуры, которая в качестве результата вычисляет не одно, а несколько значений и обрабатывает массив. Задача формулируется следующим образом:

Написать процедуру, которая для любого одномерного массива значений с атрибутами **FLOAT DEC** вычисляет их среднее, максимальное и минимальное значения.

В тех случаях, когда массив является параметром процедуры (или функции), число его измерений должно быть фиксированным в определении процедуры (функции). Однако не требуется, чтобы число элементов (область изменения индекса) массива в каждом измерении было фиксированным или чтобы оно передавалось в качестве дополнительного параметра.

Для рассматриваемой задачи определим четыре параметра: **A** — массив значений, передаваемый вызывающей программой, **MEAN**, **MAX** и **MIN** — результаты вычисления процедуры. Назовем процедуру **MMM**.

```

MMM: PROC(A, MEAN, MAX, MIN);
    DCL A(*) FLOAT DEC,
        (MEAN, MAX, MIN) FLOAT DEC;
    N = HBOUND(A, 1);
    MEAN, MAX, MIN = A(1); /*ЗАДАНИЕ НАЧАЛЬНЫХ ЗНАЧЕНИЙ*/
    DO I = 2 TO N;
        MEAN = MEAN + A(I);
        IF A(I) > MAX THEN MAX = A(I).
        ELSE IF A(I) < MIN THEN MIN = A(I);
    END;
    MEAN = MEAN/N;
END MMM;

```

Эта подпрограмма является достаточно простой, но в то же время она знакомит читателя с работой с индексами. Она может быть использована для любого одномерного массива чисел независимо от его размера.

Вызов процедур

Процедура может быть вызвана с помощью оператора **CALL**, имеющего следующую форму:

CALL имя-входа [(список-аргументов)];

Здесь *имя-входа* — имя вызываемой процедуры, а необязательная часть (*список-аргументов*) — фактические аргументы, которые должны быть использованы при данном вызове.

В качестве примера перепишем приведенную выше основную программу, вычисляющую биномиальные коэффициенты. Однако вместо функции **FACT**, вычисляющей факториал, теперь будет использоваться процедура **FACT**, вычисляющая факториал.

```
P: PROC OPTIONS(MAIN);
  DCL (A, I, N, F1, F2, F3) FIXED BIN,
      FACT ENTRY (FIXED BIN, FIXED BIN);
  GET LIST (N);
  DO I = 0 TO N;
    CALL FACT (N, F1);
    CALL FACT (I, F2);
    CALL FACT (N - I, F3);
    A = F1/(F2*F3);
    PUT SKIP LIST (I, A);
  END;
END P;
```

Наиболее заметное отличие этой программы от ее первоначальной версии заключается в добавлении трех новых переменных, **F1**, **F2** и **F3**. Им присваиваются результаты трех различных вызовов процедуры **FACT**. Отметим также, что описание **ENTRY** для процедуры **FACT** не содержит статью **RETURNS**, которая имеет смысл только для функций. Наконец, отметим, что в описании **ENTRY** мы определили атрибуты двух параметров. В нашей программе в этом не было необходимости, так как фактические аргументы в трех операторах **CALL** уже имеют эти атрибуты. (Тем не менее это дополнительное описание помогает документировать программу.)

Соответствие между аргументами и параметрами для подпрограмм точно такое же, как и для функций. Здесь мы рас-

смотрим некоторые дополнительные вопросы, касающиеся этого соответствия. Глубокое понимание их позволит легко избегать часто встречающиеся ловушки.

При выполнении процедуры или функции для каждого аргумента, удовлетворяющего одному из следующих условий, будет создан **фиктивный аргумент**.

1. Аргумент является арифметическим выражением, содержащим знаки операций, например "**N — 1**" в третьем операторе **CALL** приведенной выше программы.

2. Аргумент является константой (например, "**7**" в **CALL FACT(7,F1);**)

3. Аргумент является именем переменной, массива или структуры, и его атрибуты не совпадают с атрибутами соответствующего ему параметра (например, "**P**" в **CALL FACT(P,F1);**, где **P** имеет атрибуты **FLOAT DEC**, а не **FIXED BIN**).

4. Аргумент заключен в скобки.

5. Аргумент сам является обращением к функции.

Создание фиктивного аргумента означает, что имя временной ячейки, содержащей текущее значение аргумента (а не само имя аргумента), заменяет соответствующий параметр всюду в теле процедуры или функции и затем выполняется полученная в результате такой замены программа.

Когда аргумент используется только как входной параметр вызываемой программы, создание фиктивного аргумента не нарушает правильного функционирования подпрограммы. Например, не имеет значения, какие атрибуты у переменной **N** в первом операторе **CALL** приведенной выше программы **P — FIXED BIN (31), FIXED BIN (15), FIXED DEC (5), FLOAT DEC (6)** или любой другой набор числовых атрибутов. Если атрибуты переменной **N** и соответствующего ей параметра (**N**), используемого в определении процедуры, не совпадают, то будет создан фиктивный аргумент, который будет содержать значение **N**. Этот фиктивный аргумент затем обрабатывается точно так же, как и переменная **N** в случае, когда она имеет подходящие атрибуты. Таким образом, в любом случае факториал числа **N** будет правильно возвращен **F1**.

Однако, если аргумент используется как выход вызываемой подпрограммы, он не должен быть таким, для которого будет создан фиктивный аргумент. В противном случае получаемый результат станет значением фиктивного аргумента и будет не доступен вызывающей программе. В частности, это означает, что выходной аргумент всегда должен быть простой переменной (или массивом, или структурой), атрибуты которой совпадают с атрибутами соответствующего параметра в описании процедуры. Например, для того чтобы в результате выполнения

первого оператора **CALL** приведенной выше программы значение факториала **N** было присвоено аргументу **F1**, последний должен иметь атрибуты **FIXED BIN**.

Внутренние подпрограммы, внешние переменные и побочные эффекты

До сих пор при рассмотрении подпрограмм в ПЛ/1 мы предполагали, что вызывающая программа и вызываемая подпрограмма являются внешними по отношению друг к другу и что единственный способ обмена данными между ними заключается в использовании параметров и соответствующих аргументов. Однако существуют и другие средства работы с подпрограммами. Определение подпрограммы может целиком находиться внутри подпрограммы, которая ее вызывает. Помимо способа, основанного на использовании параметров, существуют два других способа передачи данных между вызывающей программой и подпрограммой. В этом разделе мы рассмотрим оба этих способа и покажем их преимущества и недостатки.

Рассмотрим вновь функцию **FACT** и программу **P**, которая вызывала ее. Определение функции **FACT** могло бы быть целиком помещено внутри **P**, как показано ниже.

```

P: PROC OPTIONS(MAIN);
  DCL (A, I, N) FIXED BIN;
  FACT: PROC(N) RETURNS (FIXED BIN);
    DCL (F, I, N) FIXED BIN;
    F = 1;
    DO I = 2 TO N;
      F = F*I;
    END;
    RETURN(F);
  END FACT;
  GET LIST (N);
  DO I = 0 TO N;
    A = FACT(N)/(FACT(I)*FACT(N - I));
    PUT SKIP LIST (I, A);
  END;
END P;

```

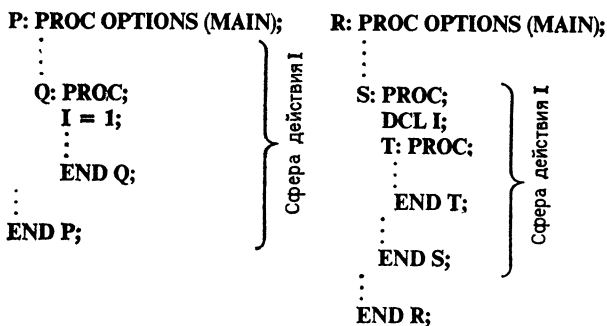
В этом случае необходимость в описании **ENTRY** для функции **FACT** отпадает. В противном случае программа **P** выглядит так же, как и раньше.

Следует также отметить, что здесь **I** — рабочая переменная как в основной программе, так и в подпрограмме **FACT**. Описание **I** в **P** определяет одну переменную, а описание **I** в **FACT** — другую переменную, хотя их имена и совпадают. При

обращении к **I** в основной программе, например в операторе “**DO I = 0 TO N;**”, программа ссылается на переменную **I**, которая была описана в **P**. Однако при обращении к **I** в подпрограмме **FACT**, например в операторе “**DO I = 2 TO N;**”, подпрограмма ссылается на переменную **I**, которая была описана в **FACT**. Таким образом, путаницы здесь не возникает.

Множество операторов, в которых допустимо обращение к переменной, называется **сферой действия** этой переменной. Сфера действия переменной определяется таким образом, что ни к каким двум переменным с одинаковым именем невозможно обращение в одном и том же операторе. Сфера действия переменной определяется в зависимости от того, описана ли эта переменная явно или нет. Если она описывается явно, то сферой действия этой переменной является та процедура (или блок **BEGIN**), в которой эта переменная описана, включая все процедуры, являющиеся внутренними по отношению к этой процедуре и не содержащие описания переменной с тем же именем. В противном случае сфера ее действия включает все процедуры, содержащие ту процедуру, в которой эта переменная используется.

Ниже дается графическое изображение сферы действия переменной для каждого из этих двух случаев. В первом случае переменная **I** не описана, и поэтому сфера ее действия включает обе процедуры **P** и **Q**



Во втором случае предполагается, что переменная **I** описана только в **S**, и поэтому сфера ее действия включает процедуры **S** и **T**, но не **R**. Таким образом, любое обращение к **I** из **R** будет воспринято как ссылка на другую неописанную переменную с тем же именем.

Программист должен хорошо разбираться в вопросах, касающихся сферы действия переменных. Например, если бы мы не ограничили сферу действия переменной **I** и не описали бы ее внутри функции **FACT**, когда последняя была внутренней

в **P**, то при обращении к **I** внутри **FACT** изменилось бы значение той же самой переменной **I**, которая управляла циклом в **P**. Результат был бы ошибочным.

Кроме того, можно умышленно не ограничивать сферу действия переменной, с тем чтобы обращаться к ней как изнутри, так и извне внутренней подпрограммы. Это средство иногда позволяет обходиться без параметров.

Наконец, аналогичного эффекта можно достичь, если две (под)программы являются внешними по отношению друг к другу, как показано ниже:

```
U: PROC OPTIONS(MAIN);
  .
  .
  .
  END U;
V: PROC;
  .
  .
  .
  END V;
```

В этом случае, если в **U** и **V** потребуется обратиться к общей переменной, скажем **I**, то эту переменную в каждой процедуре можно описать как **EXTERNAL** (внешняя) следующим образом:

```
U: PROC OPTIONS(MAIN);
  DCL I EXTERNAL;
  .
  .
  .
  END U;
V: PROC;
  DCL I EXTERNAL;
  .
  .
  .
  END V;
```

Теперь существует одна переменная **I**, к которой можно обращаться как из **U**, так и из **V**. Если такая переменная используется в двух или более внешних по отношению друг к другу процедурах, то все ее описания должны быть идентичными. Таким образом, при обмене данными между вызывающей и вызываемой программами вместо параметров и аргументов с таким же успехом могут использоваться переменные типа **EXTERNAL**.

Последний вопрос, который мы рассмотрим в связи с подпрограммами, касается так называемых побочных эффектов. Побочный эффект — это изменение подпрограммой значения переменной, которая входит в число аргументов, указанных при вызове и не является локальной в этой подпрограмме. На практике в большинстве случаев побочные эффекты являются не-

желательными и ненужными. Например, если из определения функции **ФАКТ**, вложенной в программу **P**, исключить описание локальной переменной **I**, то это приведет к нежелательному побочному эффекту. Некоторые побочные эффекты являются неизбежными или создаются преднамеренно. В этом случае программист должен предупредить об их существовании, включив в определение подпрограммы соответствующую документацию.

Рекурсия

В некоторых задачах требуются функции, которые определяются рекурсивно. Это означает, что функция использует себя в своем определении. Примеры рекурсивных функций можно найти в элементарном анализе, теории чисел и приложениях теории формальных языков. ПЛ//1 позволяет описывать процедуры, которые вызывают себя; тем самым можно непосредственно запрограммировать рекурсивно определенные функции. Такие процедуры называются **рекурсивными**. В качестве примера вновь рассмотрим функцию, вычисляющую факториал, который может быть определен рекурсивно следующим образом:

$$\text{факториал } (n) = \begin{cases} 1, & \text{если } n < 2, \\ n \times \text{факториал } (n - 1), & n \geq 2. \end{cases}$$

Средства ПЛ/1 позволяют непосредственно запрограммировать вычисление этой функции в виде следующей рекурсивной процедуры:

```

FACT: PROC(N) RETURNS(FIXED BIN);
      DCL N FIXED BIN;
      IF N < 2 THEN RETURN(1);
      ELSE RETURN(N*FACT(N - 1));
END FACT;

```

В языке ПЛ/1 IBM дополнительно требуется, чтобы рекурсивная процедура была явно определена с помощью ключевого слова **RECURSIVE**, которое записывается в операторе **PROCEDURE** перед словом **RETURNS**. При использовании некоторых других компиляторов программист сам решает, включать это ключевое слово или нет.

Читателю следует обратить внимание на то, что каждый раз, когда вычисляется выражение **N*FACT(N - 1)**, функция **ФАКТ** вызывает себя. Это приводит к следующей последовательности вызовов при вычислении **ФАКТ(4)**:

Вызов	Возвращаемый результат
1	4*ФАКТ(3)
2	3*ФАКТ(2)
3	2*ФАКТ(1)
4	1

Результат четвертого вызова (1) используется при третьем вызове, результат которого (2*1) используется при втором, и т. д.

При практическом использовании рекурсивных процедур необходимо понимать процесс их выполнения. В большинстве современных ЭВМ рекурсия не является эффективным средством и более предпочтительным обычно бывает использование итераций (циклы).

17.6. ЗАКОНЧЕННЫЕ ПРОГРАММЫ

При подготовке ПЛ/1-программы к выполнению программист должен учитывать существующие при этом требования, которые зависят от реализации. Здесь будут рассмотрены общие требования. Читателю предлагается также познакомиться со специальными требованиями, предъявляемыми к используемой им системе.

На перфокартах ПЛ/1-программа может располагаться достаточно произвольно. В системах IBM обычно требуется, чтобы программа не занимала колонки 1 и 73—80 на каждой карте. Вне этих колонок операторы и их метки могут располагаться, начиная с любых позиций. Оператор можно начать на одной строке и продолжить на другой, не указывая этого явно никаким дополнительным символом. Аналогично два или более операторов могут быть записаны на одной перфокарте. Во всех случаях ограничителем для операторов является точка с запятой (;), а не граница карты.

Учитывая эти синтаксические требования, программист должен так расположить операторы программы, чтобы была понятна логика ее работы. Обычно под этим понимается по меньшей мере следующее:

1. Метки операторов должны начинаться с позиции, расположенной левее начальной позиции для операторов.
2. Операторы должны начинаться с общей для них позиции.
3. Группы операторов, имеющие различные логические уровни, должны быть смещены по отношению друг другу (например, группа **DO** или блок **BEGIN** смещаются относительно расположенного рядом с ними текста).

В некоторых реализациях требуется, чтобы в случаях, когда ПЛ/1-программа состоит из одной основной программы, за которой следуют одна или более внешних подпрограмм, последние отделялись друг от друга и от основной программы дополнительной картой. В ПЛ/1(F) IBM эта разделительная карта имеет форму, изображенную на рис. 17.21. Более подробно об этом читатель может узнать из справочного руководства по ПЛ/1.

Колонка 1

```
* PROCESS;
```

Рис 17.21.

Внутри оператора в любое место можно вставлять произвольное число пробелов, однако они не должны располагаться

внутри имени переменной, метки оператора, имени процедуры и ключевого слова ПЛ/1. Например, слово **DECLARE** внутри себя не должно содержать пробелов.

С другой стороны, любые два соседних ключевых слова ПЛ/1 или два имени должны разделяться по меньшей мере одним пробелом. Например, следующие предложения не эквивалентны:

```
DCL X FLOAT    DEC;
DCL X FLOATDEC;
DCL XFLOAT DEC;
DCLX FLOAT DEC;
```

Второе и четвертое являются неверно записанными операторами, первое описывает переменную **X**, а третье — переменную **XFLOAT**.

17.7. ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ

Было бы невозможно рассмотреть все дополнительные возможности языка ПЛ/1. В этом разделе мы остановимся на следующих двух вопросах:

1. Состояния, прерывания и ON-элементы.
2. Классы памяти и автоматическое распределение памяти. Здесь будут рассмотрены только основные стороны этих вопросов. Более подробные сведения о них читатель может получить из справочного руководства по ПЛ/1.

Состояния, прерывания и ON-элементы

В предыдущих разделах мы упоминали о так называемых **состояниях**, которые возникают при выполнении программы в тех или иных исключительных ситуациях. Помимо уже рассмотренных нами, в языке ПЛ/1 имеется несколько других

состояний. Мы не будем рассматривать их всех, а опишем лишь наиболее важные состояния, которые приводятся в табл. 17.12.

Способы использования этих состояний будут рассмотрены ниже. Сейчас важно понять, что происходит в тех случаях, когда при выполнении программы возникает одно из этих состояний.

При выполнении программы каждое состояние может быть либо *включенным*, либо *выключенным*. При возникновении одного из указанных в табл. 17.12 состояний выполнение программы прерывается только в том случае, если это состояние является *включенным* в момент своего возникновения. Если состояние *выключено*, выполнение программы продолжается так, как будто это состояние не возникало.

Некоторые из этих состояний постоянно включены для системы в течение всего времени выполнения программы. Такими являются состояния **ENDFILE**, **ENDPAGE**, **KEY** и **TRANSMIT**. Некоторые другие состояния первоначально включены для системы, но могут быть выключены с помощью явного указания внутри программы. Такими являются состояния **CONVERSION**, **FIXEDOVERFLOW**, **OVERFLOW**, **UNDERFLOW** и **ZERODIVIDE**. Остальные из указанных состояний первоначально выключены для системы, но могут быть включены с помощью явного указания внутри программы.

Включить на время выполнения программы одно или более выключенных состояний можно с помощью **префикса состояний**, который предшествует программе и имеет следующую форму:

(список-состояний):

Здесь *список-состояний* — список имен этих состояний, отделенных друг от друга запятыми. Например, если требуется включить состояния **SIZE** и **SUBSCRIPTRANGE** на время выполнения программы **P**, то можно записать следующее:

```
(SIZE, SUBRG):  
P: PROC OPTIONS(MAIN);  
.  
.  
END P;
```

Аналогично выключить на время выполнения программы одно или более включенных состояний можно с помощью префикса состояний, в котором каждому выключаемому состоянию слева приписывается **NO**. Например, если в приведенной выше программе требуется также выключить состояние **CONVER-**

Таблица 17.12

Состояние	Смысл
<u>CONVERSION</u>	Символьно-строчное значение, как, например, 'ABC', не может быть преобразовано в эквивалентное арифметическое значение
<u>FIXEDOVERFLOW</u>	Результат арифметической операции, имеющей атрибут FIXED, превышает допустимый максимум по количеству цифр для данной реализации
<u>OVERFLOW</u>	Значение с атрибутом FLOAT превышает допустимый максимум для данной реализации
SIZE	Теряется значащая цифра наивысшего порядка при присвоении значения переменной с атрибутом FIXED
<u>UNDERFLOW</u>	Значение с атрибутом FLOAT меньше допустимого минимума для данной реализации
<u>ZERODIVIDE</u>	При выполнении операции деления (/) делитель равен 0
ENDFILE	Сделана попытка считать запись из последовательного входного файла, когда последняя запись была уже считана
ENDPAGE	Сделана попытка выполнить оператор PUT для файла печати, когда страница заполнена полностью. Число строк зависит от реализации, но обычно равно 60
KEY	(1) Сделана попытка считать запись из прямого файла, но в файле не существует записи с указанным ключом; (2) сделана попытка писать запись в прямой файл, но в файле уже существует запись с аналогичным ключом
TRANSMIT	При передаче записи на внешнее устройство или с него возникла ошибка ввода-вывода, имеющая физический характер
<u>SUBSCRIPTRANGE</u>	Значение индекса вышло за пределы области, определенной для данного измерения рассматриваемого массива
<u>STRINGRANGE</u>	При обращении к функции SUBSTR подстрока определена таким образом, что она частично или полностью выходит за пределы рассматриваемой строки
CHECK	Одной из переменных в так называемом <i>контрольном списке</i> было присвоено новое значение или был выполнен один из операторов, метки которых указаны в контрольном списке

SION, то префикс состояний надо записать следующим образом:
(SIZE, SUBRG, NOCONV):

Предположим теперь, что включенное состояние действительно возникает и выполнение программы прерывается. То, что происходит дальше, зависит от того, предусмотрено ли в программе ее собственное действие на случай возникновения этого состояния или будет выполняться *стандартное действие системы* для данного состояния. Свой выбор программист указывает, помещая или соответственно не помещая в программу оператор **ON** для этого состояния.

Предположим, например, что в программе ни для одного состояния не указан оператор **ON** и что все состояния включены. В табл. 17.13 описано стандартное действие системы на случай возникновения каждого из этих состояний.

Таблица 17.13

Состояние	Стандартное действие системы
<u>CONVERSION</u> <u>FIXEDOVERFLOW</u> <u>OVERFLOW</u> <u>SIZE</u> <u>UNDERFLOW</u> <u>ZERODIVIDE</u> <u>ENDFILE</u> <u>KEY</u> <u>TRANSMIT</u> <u>SUBSCRIPTRANGE</u> <u>ENDPAGE</u>	Печатается сообщение об ошибке и выполнение программы завершается
<u>STRINGRANGE</u> <u>CHECK</u>	Файл печати продвигается на начало следующей страницы; затем выполнение оператора вывода продолжается так, как будто состояние <u>ENDPAGE</u> не возникало Выполнение продолжается с некоторой строкой, отличной от неверно определенной подстроки Это специальное состояние, используемое для трассировки программы: оно будет рассмотрено отдельно

Читатель уже знаком с тем, как операторы **ON** используются для управления выполнением программы при возникновении состояния ввода-вывода. Оператор **ON** имеет следующую форму:

ON состояние **ON**-элемент

Здесь *состояние* — любое из перечисленных выше состояний, а *ON-элемент* — либо отдельный непомеченный оператор, либо непомеченный блок **BEGIN**.

Блок **BEGIN**, так же как и группа **DO**, используется для того чтобы разграничить последовательность операторов. Он имеет следующую форму:

BEGIN;

Последовательность операторов

END;

Если блок **BEGIN** используется в качестве ON-элемента, он может содержать любые операторы, кроме оператора **RETURN**.

В качестве примера использования оператора **ON** рассмотрим следующую программу:

```
P: PROC ORTIONS(MAIN);
  DCL NZ FIXED BIN INIT (0);
  ON ENDFILE (SYSIN) BEGIN;
    PUT SKIP LIST (NZ, 'ДЕЛЕНИЙ НА НУЛЬ.');
```

STOP;

END;

ON ZDIV NZ = NZ + 1;

```
  LOOP: DO I = 1 BY 1;
    GET LIST (A, B);
    C = A/B;
    END:
  END P;
```

При выполнении этой программы считываются пары чисел, **A** и **B**, первое делится на второе и определяется число делений на нуль.

Сначала выполняется оператор с меткой **LOOP**. Этот цикл выполняется нормально до тех пор, пока не возникнет одно из двух состояний: **ENDFILE (SYSIN)** или **ZDIV**. Состояние **ZDIV** может возникнуть только при выполнении оператора "**C=A/B;**". Когда возникнет это состояние, выполнение программы прерывается и выполняется оператор **ON** для **ZDIV**. При этом увеличивается значение счетчика **NZ**.

Если в результате выполнения оператора **ON** не осуществляется передача управления программе (с помощью оператора **GO TO** внутри ON-элемента) и не происходит завершение ее выполнения (с помощью оператора **STOP** внутри ON-элемента), происходит так называемый **нормальный возврат** в программу. Место в программе, куда в этом случае передается

управление, зависит от того, какое состояние возникло. В табл. 17.14 указана точка нормального возврата для каждого рассмотренного здесь состояния.

Таблица 17.14

Состояние	Нормальный возврат из ON-элемента
<u>CONVERSION</u>	Делается повторная попытка преобразовать строку, которая вызвала возникновение этого состояния, в том операторе, в котором оно возникло
<u>FIXEDOVERFLOW</u>	Возврат управления в точку, непосредственно следующую за местом в операторе, где возникло это состояние. Численный результат выполнения операции, вызвавшей возникновение состояния, не определен
<u>OVERFLOW</u>	
<u>SIZE</u>	
<u>UNDERFLOW</u>	
<u>ZERODIVIDE</u>	Возврат управления оператору, непосредственно следующему за оператором, который вызвал возникновение этого состояния
<u>ENDFILE</u>	
<u>KEY</u>	
<u>TRANSMIT</u>	
<u>ENDPAGE</u>	Возврат управления в точку в операторе PUT, в которой возникло это состояние
<u>SUBSCRIPTRANGE</u>	Возврат управления в точку, непосредственно следующую за местом в операторе, где возникло это состояние
<u>STRINGRANGE</u>	
<u>CHECK</u>	

Вернемся к рассмотрению нашей программы. Из табл. 17.14 видно, что из оператора **ON** управление будет передано оператору **"C = A/B"**, а переменной **C** будет присвоено неопределенное значение **A/0**. Таким образом, выполнение цикла продолжается, и при этом каждый раз, когда возникает состояние **ZDIV**, выполняется оператор **"NZ = NZ + 1;"**. При возникновении состояния **ENDFILE (SYSIN)** управление передается соответствующему оператору **ON**, в результате чего печатается значение счетчика **NZ** и завершается выполнение программы. Нормальный возврат в данном случае не осуществляется, так как этому препятствует оператор **STOP**.

В заключение настоящего раздела рассмотрим несколько практических примеров, в которых используются эти средства прерывания-захватывания.

Напомним, что состояние **CONVERSION** возникает в тех случаях, когда символьная строка (например, **'ABC'**) не может быть корректно преобразована в число. Таким образом, оно может возникнуть либо при выполнении оператора присваивания, либо при выполнении оператора потокоориентированного

ввода (**GET**). При возникновении состояния **CONVERSION** программист (с помощью оператора **ON**) может проверить или просто отобразить исходную строку, т. е. строку, в результате обработки которой это состояние возникло. Эта строка хранится в специальной системной переменной **ONSOURCE**. При обращении к **ONSOURCE** внутри **ON**-элемента **CONVERSION** в качестве результата будет получена исходная строка.

Состояния арифметического типа (**FIXEDOVERFLOW**, **OVERFLOW**, **SIZE**, **UNDERFLOW** и **ZERODIVIDE**), как правило, должны быть включенными при решении любой задачи. К сожалению, в языке ПЛ/1(F) IBM они не включены в начале работы программы, и поэтому программист должен указать его явно в префиксе состояний.

Состояния **SUBSCRIPTRANGE** и **STRINGRANGE** используются главным образом для сохранения целостности программы в процессе ее отладки. Однако, после того как программа полностью отлажена и подготовлена к использованию, необходимость оставлять эти состояния включенными обычно отпадает. Во-первых, на данном этапе в программе должны быть предусмотрены средства, позволяющие предотвратить возникновение этих состояний. Во-вторых, включение этих состояний может существенно снизить эффективность программы (в плане скорости ее выполнения и необходимого объема памяти), содержащей большое число обращений к массивам или символьным строкам.

Наконец, вернемся к рассмотрению состояния **CHECK**. Обработка прерываний для этого состояния является основным средством ПЛ/1 для трассировки программы и проверки промежуточных результатов. Хотя состояние **CHECK** используется при отладке, оно тем не менее является существенной частью языка ПЛ/1.

Для того чтобы при каждом присваивании нового значения некоторой переменной, скажем **I** (в результате выполнения оператора присваивания или оператора ввода), печатать это значение, к программе может быть добавлен префикс **CHECK** следующим образом:

(CHECK(I)):

Если программисту нужно, чтобы при каждом выполнении некоторого оператора печаталась его метка, скажем **L**, он должен добавить к программе префикс **CHECK** следующим образом:

(CHECK(L)):

Пример программы и ее выход, данные на рис. 17.22, показывают, как может быть использовано состояние **CHECK**. Напомним, что по определению оператора **DO** переменной **I** будет

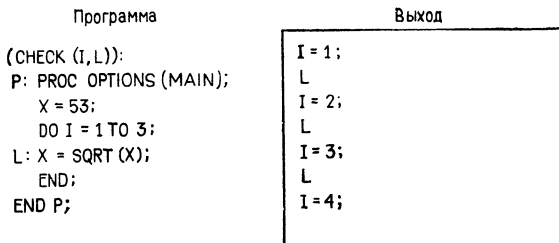


Рис. 17.22.

четыре раза присваиваться значение, хотя оператор с меткой **L** будет выполняться три раза.

Использование состояния **CHECK**, так же как и состояний **STRINGRANGE** и **SUBSCRIPTRANGE**, приводит к снижению скорости выполнения программы и увеличению объема требуемой памяти. Поэтому его следует использовать только при отладке программы.

Классы памяти и автоматическое распределение памяти

В некоторых случаях до начала работы программы нельзя предсказать, какой объем памяти потребуется для некоторых данных. Обычно такая ситуация возникает в системном программировании и ряде других областей, где широко используются средства **динамического распределения памяти**.

Каждая переменная, включая массивы и структуры, в ПЛ/1-программе имеет атрибут **класс памяти**, который определяет, когда, как и кем отводится память для этой переменной. Класс памяти для переменной должен быть одним из тех, которые перечислены в табл. 17.15.

Если класс памяти для переменной явно не описан в программе, то память назначается автоматической (**AUTOMATIC**) по умолчанию.

Использование класса статической (**STATIC**) памяти является наиболее эффективным в плане экономии времени, поскольку распределение памяти и присвоение начальных значений осуществляется при загрузке программы для ее выполнения. Обычно его используют при работе с массивами, представляющими таблицы констант. Например,

```
DCL MONTNS(12) CHAR(3) STATIC
  INIT ('JAN', 'FEB', 'MAR', ..., 'DEC');
```

Однако при использовании класса статической памяти программист должен помнить, что присвоение начальных значений осуществляется только один раз в начале выполнения программы.

Таблица 17.15

Класс памяти	Смысл
STATIC (статическая)	Память для переменной отводится один раз, и (если требуется) при загрузке всей программы для выполнения система присваивает этой переменной начальное значение
AUTOMATIC (автоматическая)	Отведение памяти для переменной и (если требуется) переприсвоение ей начального значения осуществляется системой каждый раз, когда начинает выполняться подпрограмма или блок BEGIN, где описана эта переменная
CONTROLLED (управляемая)	Память для переменной отводится программой в результате выполнения оператора ALLOCATE. До этого момента к переменной нельзя обращаться, поскольку ее еще не существует в памяти. Однако одновременно могут существовать несколько копий переменной. В этом случае можно обращаться только к самой новой копии
BASED (базированная)	Как и в предыдущем случае, память для переменной отводится программой. Однако в отличие от управляемого (CONTROLLED) размещения для базированной (BASED) переменной могут существовать несколько копий, к каждой из которых можно обращаться при выполнении программы

Поскольку автоматическая память (**AUTOMATIC**) используется наиболее часто, она назначается по умолчанию. Для переменной с автоматическим размещением система отводит память только после начала фактического выполнения подпрограммы или блока **BEGIN**, в которых эта переменная описана. Эта особенность имеет важное практическое применение в тех случаях, когда до начала выполнения программы нельзя определить размер **N** массива **A**. Рассмотрим следующий пример:

P: PROC OPTIONS(MAIN);

```

. . .
GET LIST (N);
  BEGIN;
    DCL A(N) FLOAT;
    . . .
  END;
. . .
END P;
```

Здесь описание переменной **A** мы локализовали таким образом, что сфера ее действия включает только те операторы, которые расположены внутри блока **BEGIN**. При каждом входе в этот

блок переменная **N** будет иметь целое значение, которое определит размер массива **A**. При выходе из блока **BEGIN** память, отведенная под переменную **A**, будет освобождена, и значение **A**, таким образом, вновь станет не определенным. (Эта особенность переменных с автоматическим размещением проявляется в отношении блоков **BEGIN**, а также подпрограмм, несмотря на то что вход в подпрограмму осуществляется только с помощью оператора **CALL** или обращения к функции. Эта особенность не проявляется в отношении группы **DO**, так как оператор **DO** не разграничивает сферу действия описанной переменной.)

При использовании класса управляемой (**CONTROLLED**) памяти ответственность за отведение памяти для переменных и ее освобождение ложится на программиста. Память отводится с помощью оператора **ALLOCATE**, а освобождается с помощью оператора **FREE**. Эти операторы имеют следующие основные формы:

ALLOCATE список-переменных;

FREE список-переменных;

Здесь *список-переменных* — это список из одного или более имен переменных (включая массивы и структуры), каждое из которых имеет атрибут **CONTROLLED**. Имена отделяются друг от друга запятыми.

К переменной с управляемым размещением, скажем **X**, нельзя обращаться до того момента, пока не будет выполнен оператор "**ALLOCATE X**:". Однако одновременно может существовать несколько размещений **X**, поскольку в результате каждого выполнения оператора "**ALLOCATE X**:" определяется новое размещение. При обращении к **X**, например в операторе "**Y = X + 1**;", используется самое последнее размещение **X**. Предпоследнее размещение **X** становится доступным только после выполнения оператора "**FREE X**;", который освобождает память, отведенную под самое последнее размещение **X**. Таким образом, в результате многократного отведения памяти под одну и ту же переменную образуется стек, в качестве вершины которого определяется самое последнее размещение переменной. Операторы **ALLOCATE** и **FREE** используются для выполнения элементарных операций над стеками — заталкивания и выталкивания. Поскольку за пределами области системного программирования эти понятия используются редко, мы не будем больше останавливаться на их рассмотрении. Читателю, которого интересует этот вопрос, рекомендуется обратиться к справочному руководству по ПЛ/1.

Отметим, однако, что, если переменной с управляемым размещением является массив, размеры которого заранее не из-

вестны, память для него может быть явно отведена в программе следующим образом. Во-первых, неизвестный размер массива обозначается в его описании звездочкой. Например, в описании

DCL A(*) FLOAT CONTROLLED;

указано, что **A** является одномерным массивом с неизвестным числом значений типа **FLOAT**, для которых память будет отводиться и освобождаться программой. В программе должны быть предусмотрены эти действия. Например, при выполнении следующей программы будет получен такой же результат, как и при выполнении предыдущей программы, однако вместо блока **BEGIN** здесь используется массив **A** с управляемым размещением.

```
P: PROC ORTIONS(MAIN);
  DCL A(*) CONTROLLED FLOAT;
  . . .
  GET LIST (N);
  ALLOCATE A(N);
  . . .
  FREE A;
  . . .
END P;
```

Отметим, что число элементов (**N**) в **A** определяется в операторе **ALLOCATE**, но не в операторе **FREE**.

Класс базированной (**BASED**) памяти аналогичен классу управляемой памяти, однако ко всем существующим копиям (а не только к самой последней) базированной переменной можно обращаться в любое время при выполнении программы. Наше рассмотрение базированных переменных будет носить поверхностный характер, так как они применяются главным образом в системном программировании в тех областях, где широко используются такие понятия, как стеки, очереди и другие связанные списки.

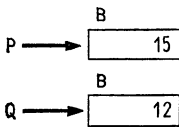


Рис. 17.23.

Обращение к базированной переменной осуществляется с помощью так называемой переменной типа указатель (**POINTER**). Этот тип переменных отличается от тех типов, который мы до сих пор рассматривали. Единственное назначение переменной типа указатель заключается в том, чтобы указывать базированную переменную, а не в том, чтобы содержать само значение. Это показано на рис. 17.23. Если требуется обратиться к той копии переменной **B**, значение которой равно 15, нужно записать следующую составную ссылку к **B**:

P —> B

К другой копии можно обратиться следующим образом:
 $Q \rightarrow B$.

Хотя в программе может быть использовано любое число переменных, одна из них должна быть определена как **собственный указатель** базированной переменной. Это делается с помощью следующего описания:

DCL переменная **BASED** (переменная-типа-указатель);

Например, для того чтобы описать, что **B** — базированная переменная, а **P** — ее собственный указатель, можно записать

DCL B FIXED BASED (P);

Для отведения памяти базированной переменной используется одна из следующих форм оператора **ALLOCATE**:

1. **ALLOCATE** переменная;

2. **ALLOCATE** переменная **SET** (указатель);

В первом случае создается новое размещение *переменной*, а в качестве ее указателя выбирается собственный указатель этой переменной. Форма 2 эквивалентна форме 1, за исключением того, что в форме 2 указатель задается явно. Таким образом, в результате выполнения одного из следующих операторов:

ALLOCATE B;

ALLOCATE B SET (P);

переменной **B** будет отведена память, а в качестве указателя для этого размещения будет взята переменная **P** (рис. 17.24). В результате выполнения операторов

ALLOCATE B SET (Q);

$P \rightarrow Q = 15;$

$Q \rightarrow B = 12;$

(1) будет создано другое размещение для **B** с указателем **Q**;
 (2) этим двум размещениям переменной **B** будут присвоены значения 15 и 12 соответственно (см. рис. 17.23). Как здесь показано, конкретное размещение рассматриваемой базированной переменной может быть явно

указано с помощью составной ссылки, как, например, $P \rightarrow B$ или $Q \rightarrow B$. В тех

случаях, когда требуемым размещением базированной переменной является то, которое соответствует ее собственному указателю, квалификатор ($P \rightarrow$) может быть опущен. Таким образом, второй из приведенных выше операторов может быть переписан в следующей эквивалентной форме: **B = 15**.

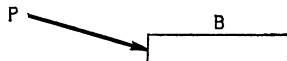


Рис. 17.24.

Переменную типа указатель можно описать, объявляя, что она является собственным указателем базированной переменной, или явно, с помощью атрибута **POINTER**. Например, указатель **Q**, использованный выше, должен быть описан следующим образом:

DCL Q PTR;

Переменной типа указатель может быть присвоено значение с помощью оператора **ALLOCATE** или оператора присваивания. Например, оператор

Q = P;

присваивает указателю **Q** значение **P**, в результате чего **Q** становится указателем того же самого размещения базированной переменной, с которым в текущий момент связан указатель **P** (рис. 17.25). Существует специальное значение **NULL**, смысл

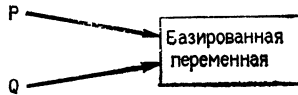


Рис. 17.25.

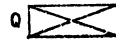


Рис. 17.26.

которого заключается в том, что если его присвоить переменной типа указатель, то последняя не будет указывать ни на какой адрес. Это показано на рис. 17.26 для указателя **Q**. Значение **NULL** может быть присвоено явно, как показано ниже:

Q = NULL;

Для того чтобы показать простейшие случаи использования базированных переменных и переменных типа указатель при обработке списков, рассмотрим задачу создания списка имен, считываемых с входной колоды перфокарт (рис. 17.27). Предположим, что программисту не известно число имен, но в любом случае требуется создать внутренне связанный список, форма которого показана на рис. 17.28 (для данных, приведенных выше).

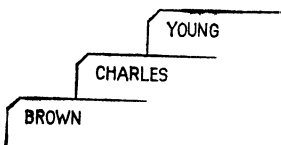


Рис. 17.27.

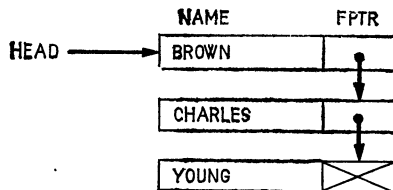


Рис. 17.28.

Каждое размещение базированной переменной состоит из двух частей: имени (**NAME**) и прямого указателя (**FPTR**) следующего размещения. Конец списка задается пустым (**NULL**) прямым указателем, а начало, или заголовок, списка — указателем **HEAD**. Этот пример показывает наиболее общую структуру списка, так называемого **односвязанного списка**.

Таким образом, базированная переменная логически описывается как структура. Приведем программу, создающую этот список, а затем покажем ее работу на примере указанных данных.

```

LISTER: PROC OPTIONS(MAIN);
      DCL 1 NODE BASED(P);
          2 NAME CHAR (15);
          2 FPTR POINTER;
      (Q, HEAD) PTR;
      ON ENDFILE (SYSIN) BEGIN;
          FREE NODE;
          IF Q = NULL THEN Q → FPTR = NULL;
          ELSE HEAD = NULL;

          STOP;
          END;
      ALLOCATE NODE SET (P);
      HEAD = P; Q = NULL;
      DO I = 1 BY 1;
          GET EDIT (P → NAME) (COL(I), A(15));
          Q = P;
          ALLOCATE NODE SET (P);
          Q → FPTR = P;

      END;
END LISTER;

```

Первым выполняемым оператором является первый из операторов **ALLOCATE**, который создает копию базированной переменной **NODE**, как показано на рис. 17.29. Отметим, что ни **NAME**, ни **FPTR** пока не имеют значений. При выполнении следующих двух операторов в указатель **HEAD** засылается значение, соответствующее первому узлу в списке, а во вспомогательный указатель **Q** — значение **NULL**. После первого выполнения цикла **DO** формируется список, структура которого

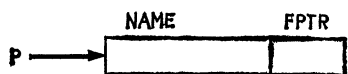


Рис. 17.29.

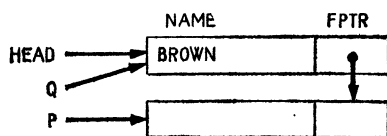


Рис. 17.30.

показана на рис. 17.30 (для данных из рассматриваемого примера). А именно, считывается первое имя, устанавливается значение **Q** и затем размещается новый узел (**NODE**), который связывается с указателем **P**. Наконец, прямой указатель (**FPTR**) предыдущего узла устанавливается таким образом, чтобы указывать этот новый узел, соединяя тем самым два узла вместе.

Этот процесс продолжается до тех пор, пока не будут считаны все перфокарты. Читатель должен был заметить, что пе-

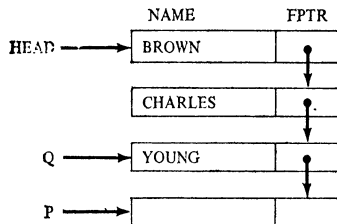


Рис. 17.31.

ременная **Q** всегда указывает предпоследнее размещение переменной **NODE**, а **P** указывает следующий узел, с тем чтобы разместить в нем имя в случае, если не все имена еще считаны. При возникновении состояния **ENDFILE** для этих данных ситуация будет такой, как показано на рис. 17.31. Таким образом, при выполнении **ON**-элемента освобождается самое последнее размещение переменной **NODE** и в *прямой* указатель **FPTR**, соответствующий узлу (**NODE**), связанному с **Q**, засылается значение **NULL**, указывающее конец списка. Отметим, что оператор **IF** в **ON**-элементе предназначен на тот случай, когда не будет считываться ни одной перфокарты. В этом случае в указатель **HEAD** засылается значение **NULL**, указывающее, что список является пустым.

18

ДОКУМЕНТАЦИЯ ПО АППАРАТНЫМ СРЕДСТВАМ И ПРОГРАММНОМУ ОБЕСПЕЧЕНИЮ

Д. Эрикссон

18.1. ВВЕДЕНИЕ

Под термином **документация** будем понимать отпечатанный материал, содержащий описание аппаратных средств и программного обеспечения ЭВМ. В последние годы вопросу составления документации уделяется большое внимание, поскольку вычислительная техника стала широко использоваться людьми, не являющимися специалистами в области обработки данных и ЭВМ. Часто документация является последней частью создаваемой вычислительной системы, первой частью, с которой знакомится пользователь, и частью, которая наиболее широко обсуждается.

Существуют две основные причины, по которым документация так легко поддается критике:

1. Качество документации определяется качеством информации, которой располагает автор документации.

2. Узкая специализация авторов является нежелательным обстоятельством, поскольку документация должна отражать широкий круг вопросов, связанных с различными компонентами вычислительной системы.

Следовательно, для каждого человека, связанного с литературой по ЭВМ (будь то пользователь или автор), важно уметь точно определить характерные черты хорошей документации.

18.2. ТРЕБОВАНИЯ К ХОРОШЕЙ ДОКУМЕНТАЦИИ

Основное требование к любой документации заключается в том, чтобы она была полной и точной. Большинство претензий пользователей к документации возникает из-за неточности информации. И это понятно. Тот, кто потратил несколько сотен долларов за пакет программного обеспечения или несколько тысяч долларов за вычислительную систему, ожидает получить точную и исчерпывающую информацию по вопросам использования программного и аппаратного обеспечения.

“Точность” означает нечто большее, чем подробное описание работы программ и функционирования аппаратных средств. Помимо этого, документация должна содержать точное описание предметной области, в которой могут использоваться данные программные и аппаратные средства. Например, пакет программ для решения деловых задач должен быть правильным с точки зрения стандартной деловой практики и нормальных канцелярских процедур.

Другим общим недостатком документации является то, что часто она не содержит исчерпывающей информации (хотя о некоторых программных и аппаратных средствах, по-видимому, невозможно дать полную информацию). Если документация содержит достаточно информации, но написана плохо, то пользователю приходится самому доискиваться до необходимых для себя сведений. Такую документацию также нельзя назвать хорошей, однако она предпочтительнее той, в которой необходимая информация вообще отсутствует. Если документация не содержит достаточного количества информации, пользователь имеет полное право (в действительности он почти обязан) подать жалобу на ответственного за сбыт программного обеспечения для аппаратных средств.

18.3. ТИПЫ ДОКУМЕНТАЦИИ

Существуют следующие основные типы документации по ЭВМ:

• *Системные руководства* • *Руководства по применению* • *Руководства для обучения* • *Справочные руководства*.

В системном руководстве описывается функция некоторой части аппаратных средств или работа набора программ. В руководстве по применению указывается, как конкретные устройства или программы могут быть использованы для решения определенных задач, получения некоторых результатов или выполнения требуемых функций. В руководстве для обучения содержатся сведения для пользователя о работе и использовании программных и аппаратных средств, а в справочном руководстве информация о программном обеспечении и аппаратных средствах собрана таким образом, чтобы к ней можно было легко и быстро обращаться. Иногда встречаются документации смешанного типа. Например, руководство для обучения может содержать отдельный справочный отдел.

Перед тем как оценивать качество документации, следует определить ее назначение. Хорошее руководство для обучения не всегда является хорошим справочным руководством, так же как хорошее справочное руководство не является хорошим руководством для обучения.

Таким образом, некоторые претензии к документам возникают вследствие того, что пользователи ожидают получить из руководства информацию, которую в нем и не предполагалось давать. По самому своему характеру руководство для обучения в отличие от справочного руководства редко предоставляет возможность просто и быстро находить нужную информацию. С другой стороны, справочное руководство не так полезно для пользователя при первом знакомстве с новым материалом, как руководство для обучения.

Из сказанного выше следует, что при подготовке материала автор документации должен знать, для какого круга пользователей она предназначена и каково ее назначение.

18.4. СПРАВОЧНАЯ ДОКУМЕНТАЦИЯ

В большинстве случаев предполагается, что пользователь справочной документации уже знаком с рассматриваемыми в ней вопросами. Точно так же, как для пользования словарем надо знать английский язык, для пользования справочным руководством по Коболу необходимо иметь некоторые понятия о Коболе.

Один из наиболее важных вопросов, который следует рассмотреть при написании документации, состоит в выборе способа компоновки материала. Например, предположим, что требуется написать справочное руководство по использованию различных операторов и команд языка высокого уровня. Как скомпоновать этот материал? Один способ заключается в том, чтобы расположить операторы и команды в алфавитном порядке; другой — сгруппировать их по типам, посвящая отдельные главы операторам ввода и вывода, управляющим операторам и операторам передачи, функциям и т. д.

Если автору документации не известен уровень подготовки пользователя, то лучше всего располагать материал в алфавитном порядке. При высоком уровне подготовки пользователя можно использовать более сложные способы изложения материала. Важно, чтобы пользователи могли легко и быстро получить необходимую информацию.

Следует отметить, что во многих случаях одна информация требуется чаще, чем другая. Часто используемая информация должна быть помещена в отдельном разделе. Другой способ выделения такой информации заключается в использовании жирного шрифта или курсива.

Необходимо стремиться к тому, чтобы каждая статья в справочной документации была независимой от других статей. Статьи, в которых есть ссылки на предыдущие статьи, часто огорчают пользователей и приводят к путанице.

18.5. ДОКУМЕНТАЦИЯ ДЛЯ ОБУЧЕНИЯ

Этот тип документаций предназначен для того, чтобы научить пользователя программировать и работать на вычислительной системе. При написании документации для обучения основным предположением является то, что рассматриваемые в ней вопросы являются новыми для читателя и он желает изучить их.

В документации для обучения материал должен быть изложен в том порядке, в котором он будет изучаться пользователем. Обучение программированию на языке Бейсик надо начать с рассмотрения оператора CLS. Это следует не из того, что он расположен в начале списка операторов Бейсика, упорядоченных по алфавиту, а из того, что он является одним из первых операторов, используемых при программировании на Бейсике.

Хорошее руководство для обучения должно содержать много практических примеров. Эти примеры должны нести содержательную информацию, представляющую интерес для пользователя. Многие руководства по микроЭВМ составлены таким образом, что их изучение требует взаимодействия между пользователем и микроЭВМ; пользователь вводит данные в микроЭВМ и становится активным участником процесса обучения.

Общим недостатком многих руководств для обучения является то, что их авторы переоценивают знания пользователя. При написании документации их авторы используют устоявшиеся понятия; при этом легко забыть, что пользователь может не знать многих основных понятий и терминов. Полезно также весь материал в руководстве для обучения разбивать на небольшие части.

Было бы ошибочно полагать, что составлять руководство для обучения легче, чем справочное руководство. Руководство для обучения должно содержать по каждому вопросу примерно столько же информации, сколько ее содержится в справочном руководстве.

18.6. СОСТАВЛЕНИЕ ДОКУМЕНТАЦИИ

Документация не может быть написана одним автором. Она должна быть результатом совместного труда автора, специалистов по сбыту, а также программистов или инженеров.

Специалисты по сбыту смогут дать информацию о покупателях продукции и об особенностях торговли. Если продукцией является пакет программного обеспечения стоимостью 500 долл., предназначенный для научных работников, то и документация должна быть написана на соответствующем уровне. Докумен-

тация на пакет программ для телеигры стоимостью 15 долл. должна быть достаточно простой, с тем чтобы ее смогли понять и использовать дети. Однако во многих случаях покупателями продукции являются специалисты по сбыту, и документация пишется специально для них.

Программисты или инженеры предоставляют информацию технического характера, касающуюся пакета программного обеспечения или элемента аппаратной части. Программисты и инженеры имеют почти такое же отношение к специалистам по сбыту, какое имеет автор документации. Они должны предоставить окончательную продукцию, которая удовлетворяет требованиям, предъявляемым специалистами по сбыту, и передавать нужную информацию автору документации.

В результате документация обычно пишется параллельно с разработкой программного обеспечения или аппаратных средств. При этом необходимо планирование, иначе компания не сможет продать готовый пакет программного обеспечения или аппаратные средства из-за незавершенной документации.

Это может потребовать от авторов документации и пользователей достаточного запаса знаний. Часто у автора документации нет возможности до опубликования документации сверить ее с последней версией аппаратных или программных средств. В этом случае почти неизбежны некоторые незначительные (а возможно, серьезные) ошибки в документации. Кроме того, может быть нарушена согласованность действий между автором документации, специалистами по сбыту и техническими работниками.

Эффективный способ проверки пригодности документации для предполагаемых пользователей состоит в том, что группе типичных пользователей предлагается на практике использовать аппаратные средства или программное обеспечение с помощью этой документации. По крайней мере перед тем как автор документации будет писать ее последнюю версию, она должна быть рассмотрена специалистами по сбыту, а также инженерами или программистами.

Если документация удовлетворяет рассмотренным требованиям, ее качество является вполне удовлетворительным. В то время как в некоторых крупных издательствах на издание книги объемом в 200 страниц может потребоваться несколько лет, руководство такого же объема может быть написано за несколько недель.

19

БАЗЫ ДАННЫХ И ОРГАНИЗАЦИЯ ФАЙЛОВЫХ СИСТЕМ

Д. Уидерхолд

19.1. ВВЕДЕНИЕ

Когда о базе данных говорят не формально, имеют ввиду набор взаимосвязанных данных, аппаратные средства ЭВМ, которые используются для хранения данных, и программы, используемые для манипулирования этими данными.

Взаимосвязанными мы называем такие данные, которые представляют сведения об определенном предприятии, например компании, университете, правительственном учреждении. Данные могут быть связаны также в силу того, что они относятся к определенному кругу проблем, например касающихся болезни, которой интересуются сотрудники нескольких больниц. Данные должны быть организованы таким образом, чтобы их можно было обрабатывать для получения информации.

Организация данных в базе данных должна правильно передавать их основное смысловое значение, или семантику, и позволять эффективно обращаться к ним. В обычной программе структура данных организована таким образом, чтобы обеспечить удобный доступ к ним из данной программы. База данных содержит данные, которые используются множеством разнообразных программ. Следовательно, при определении структуры базы данных нельзя ограничиваться критериями, используемыми при программировании конкретных функций.

19.2. ФАЙЛЫ

База данных является набором связанных данных. Хранение данных в базе данных осуществляется с помощью одного или более файлов. Файл определяется как набор однотипных записей, хранимый на внешних запоминающих устройствах. Типичными внешними запоминающими устройствами являются

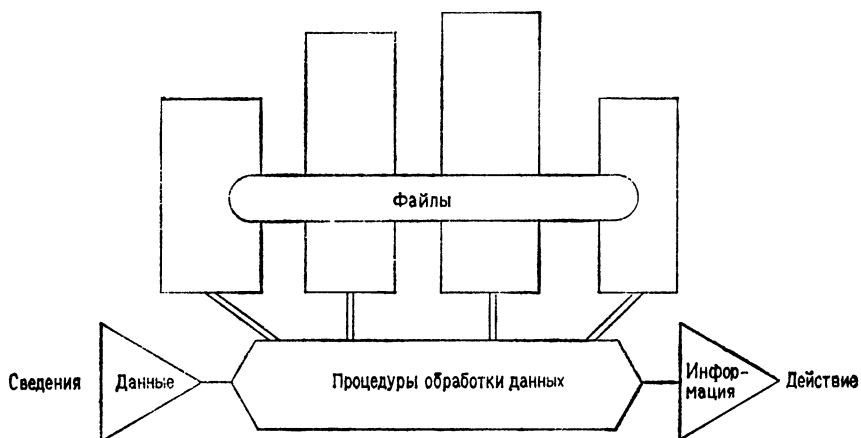


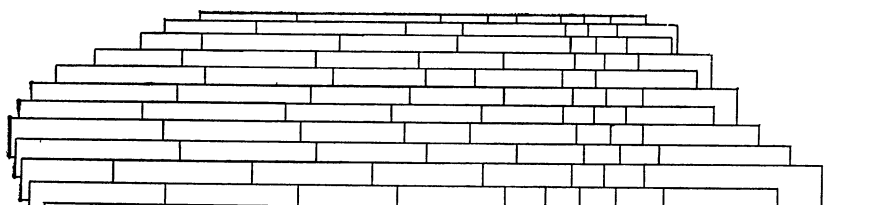
Рис. 19.1. База данных.

дисководы с магнитными дисками. **Запись** мы определим здесь как набор связанных полей, содержащих элементы данных. **Элемент данных** обычно представляет собой значение, которое является частью описания объекта или события. С помощью вычислительных процессов можно манипулировать этими значениями.

Размер

Для того чтобы использование предлагаемых в этой главе методов было оправданным, база данных должна быть достаточно большой. Мы будем рассматривать только те процессы, которые применимы к большим внешним файлам. Методы, применимые к набору данных, который может быть целиком обработан в памяти с прямой адресацией, или оперативной памяти ЭВМ, здесь рассматриваться не будут. При использовании термина **база данных** предполагается также, что с ней связано некоторое число людей. Помимо того что запись данных может осуществляться людьми, не связанными с пользователями, данные могут содержать информацию, которая применима для различных целей. Количество обрабатываемых данных может изменяться от больших до очень больших значений и зависит от используемых аппаратных и программных средств.

Большим называется такое значение, которое превосходит количество данных, обрабатываемых одним человеком, даже если он имеет доступ к вычислительной системе. Фактическое количество будет изменяться в зависимости от сложности дан-



Джон Джонс	513-72-3411	14 окт 41	Н-Й, Н-Й	3	...	11	1980	14 Флора, Напа
Имя	Номер страхового полиса	Дата рождения	Место рождения	Налоги	Категория зарплаты	Год назначения зарплаты	Адрес	

Рис. 19.2. Файл сведений о служащих.

ных и решаемых задач. Примером большой базы данных является система, содержащая сведения о 6000 служащих промышленной компании и производимой ею продукции. Эта база данных может, например, содержать 300 000 записей 21 типа.

Очень большая база данных является важной частью предприятия и находится в постоянном пользовании многих людей. В то же время для нее требуется большое число запоминающих устройств. Примером очень большой базы данных может служить база данных телефонной компании, обслуживающей 5 миллионов абонентов. Значительно большие базы данных используются ведомством по социальному страхованию и другими государственными учреждениями.

Во многих случаях, как, например, при периодически проводимом анализе данных, повторениях с целью повышения надежности, а также при инспекциях данных, важно иметь копию содержимого файла на определенный момент времени. Чтобы избежать дополнительных трудностей, лучше всего запретить использовать и модифицировать файл во время его копирования. Это полезное требование накладывает определенные ограничения на работу систем, которые будут нами рассмотрены. В очень больших базах данных это требование является источником противоречий.

Организация файлов

Файлы характеризуются не только размерами, но и организацией. От организации файлов зависит время выполнения операций записи и выборки.

Позже мы подробно рассмотрим шесть основных типов организации файлов. Для базы данных часто требуется более чем один тип файлов.

Ввод-вывод

При считывании или записи файлов данные передаются с одного запоминающего устройства вычислительной системы на другое. При считывании входных данных или записи выходных данные соответственно поступают в вычислительную систему или покидают ее. База данных связана с теми данными, которые остаются внутри системы. Данные, которые записываются на ленту, хранятся на ней, а затем снова считываются, могут быть частью базы данных. Данные, которые выбираются, затем модифицируются и снова вводятся, должны рассматриваться как новые данные.

Для ввода и вывода данных используются, например, перфокарты, печатаемые отчеты, ленты, переносимые на другие вычислительные системы, и микрофильмы, генерируемые ЭВМ.

Примерами устройств, используемых для хранения файлов, являются фиксированные диски и барабаны, основные ленты или диски, хранимые рядом с ЭВМ, архивные ленты, хранимые в отдаленном хранилище в целях их защиты, и колоды перфокарт, содержащие списки клиентов или служащих.

Во многих вычислительных системах нет четкого разделения между файлами, используемыми для ввода, и файлами, используемыми для вывода. В тех случаях, когда речь будет идти о системах баз данных, мы будем предполагать, что имеются все необходимые средства ввода и вывода, включая, где это необходимо, терминалы, работающие в оперативном режиме.

Мы не будем касаться таких организаций файлов, которые связаны с особенностями организации ввода и вывода. С учетом их данные рассматриваются как непрерывный поток литер. Как следует из описания языка ПЛ/1, в основе определения потокоориентированных файлов и их эквивалентов в других системах лежат операции чтения и записи непрерывных строк текста. Непрерывные потоки текста важны для связи, но не подходят для манипулирования данными. Термин **файл** не будет также использоваться для ссылки на аппаратные средства, применяемые для хранения данных, образующих файлы.

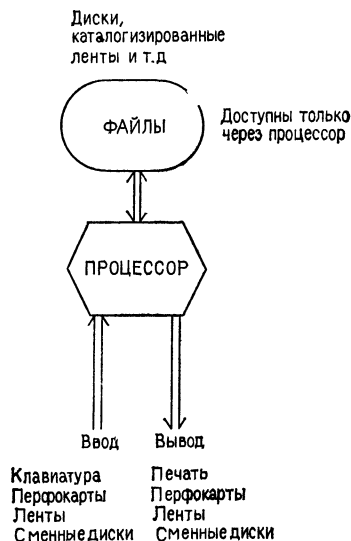


Рис. 19.3. Файлы в связи с вводом-выводом.

19.3. ВЫЧИСЛЕНИЯ, ВЫПОЛНЯЕМЫЕ ПРИ РАБОТЕ С БАЗОЙ ДАННЫХ

В первом разделе мы описали базу данных в статическом состоянии, а именно рассматривался вопрос о хранении данных в виде файлов, записей и полей. Сейчас мы рассмотрим базу данных в динамике. Изучение динамики структур программирования является неотъемлемой частью анализа работающих систем. Термин **вычисление** определяет ту группу процедур, которая используется для манипулирования данными в базе данных. Большая часть вычислений, используемых для манипулирования наборами данных, может быть легко описана. Существуют четыре типа вычислений, связанные с базами данных:

1. Построение набора данных.
2. Обновление элементов данных в наборе данных.
3. Выборка данных из набора данных.
4. Сокращение большого числа данных с целью получить удобную для пользования форму.

Схематически они изображены на рис. 19.4. Очевидно, что алгебраические действия являются только частью вычислений, которые могут выполняться. При работе с базами данных используются все четыре типа вычислений, однако в некоторых случаях какой-то один тип может использоваться чаще других.

Построение базы данных включает в себя сбор данных, организацию набора данных и его хранение. Эта часть работы с базой данных часто является самой дорогой.

Обновление базы данных включает в себя добавление новых данных, изменение значений данных, когда это необходимо, и удаление недействительных или устаревших элементов. Частота обновления зависит от решаемой задачи. Статическая база данных может быть использована при ретроспективном анализе, когда все данные собираются до проведения эксперимента. Динамическая, или непостоянная, база данных используется например в системах резервирования.

Выборка данных может заключаться либо в нахождении определенного элемента с целью получить хранящееся значение или сведение, либо в сборе ряда взаимосвязанных элементов с целью получить данные, касающиеся некоторой зависимости, которая проявляется только в том случае, когда данные собраны вместе. Нужная запись в базе данных находится с помощью аргумента поиска, который сравнивается с ключом в записях базы данных. К сожалению, значение слова **ключ**, употребляемого при вычислениях, не соответствует значению этого слова в обычном житейском смысле. Человек при входе в дом использует **ключ** для того, чтобы открыть замок двери. В нашем случае человек использует **аргумент** для сравнения с **ключом** нуж-

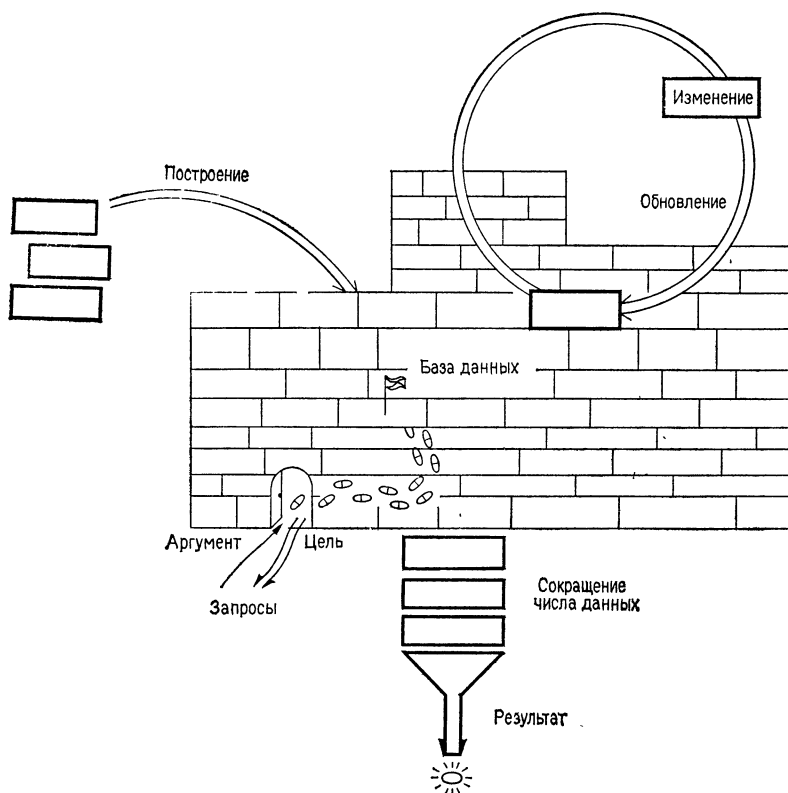


Рис. 19.4. Вычисления, выполняемые в системе баз данных.

ной записи. Аргумент поиска часто тоже называют ключом поиска. Различие обычно определяется по контексту.

В тех случаях, когда требуемые данные рассредоточены по всей базе данных, может возникнуть необходимость в хранении большей их части в более сжатой форме, которая упростила бы доступ к этим данным и их обработку. Примерами часто используемых методов сокращения числа данных являются методы статистической обработки, методы составления годовых отчетов и графические методы представления данных.

Концептуальное описание вычислений для частной задачи часто может занимать несколько строк, а соответствующие процедуры могут быть изображены в виде блок-схемы, занимающей не более одной страницы, и быстро запрограммированы. Однако приведение системы баз данных в рабочее состояние часто требует больших временных и финансовых затрат.

Процессы

В то время как для пользователя запрос на обновление или выборку данных является основной единицей вычислений, операционная система, управляющая ходом вычислений, может представлять их как совокупность нескольких различных **процессов**. Процессы являются основными единицами вычислений, управляемыми операционной системой. Процессы одного вычисления могут требовать различные аппаратные средства ЭВМ и выполняться параллельно. Различные типы операционных систем предъявляют различные требования к свойствам процессов, которыми они управляют. Порядок выполнения процессов (а следовательно, и вычислений) и их производительность определяют производительность системы баз данных.

Секции процесса

Сам процесс может состоять из нескольких секций. **Секция** — это последовательность шагов программы, при выполнении которых операционная система не должна затрагивать процесс. Если процесс остановлен, запрашивает данные из файла, требует выделения дополнительной области памяти и т. д., то управление должно быть передано операционной системе. Например, если запрашивается запись из файла, то операционная система должна проверить, что эта запись в текущий момент не модифицируется другим процессом. Секция процесса, с помощью которой осуществляется такое взаимодействие, называется **критической секцией**. На рис. 19.5 изображен процесс и его секции.

19.4. ИЕРАРХИЧЕСКОЕ ПРЕДСТАВЛЕНИЕ ДАННЫХ

В вопросе, касающемся среды, в которой функционирует база данных, существуют два важных аспекта. С одной стороны, база данных является компонентом общей системы, включающей людей, организации и средства сообщения. С другой стороны, элементами, составляющими базу данных, являются биты и байты в аппаратных средствах. В тех случаях, когда содержание базы данных рассматривается как структурированное представление информации о некотором реальном объекте, нас интересует значение информации, представленной данными, и правильность результатов, которые могут быть получены. Когда база данных рассматривается с точки зрения особенностей ее функционирования, мы обращаем внимание на структуру электронных устройств, с помощью которых возможна автоматизация процесса обработки данных. На рис. 19.6 изо-

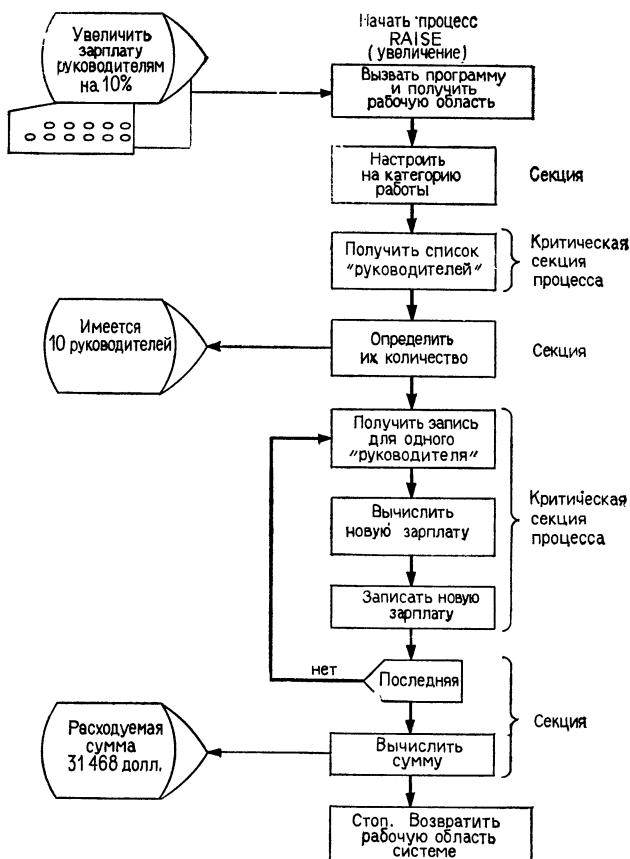


Рис. 19.5 Секции процесса.

бражена структура в форме пирамиды, показывающая широту базы, требуемой для получения информации.

Поскольку тема этой главы связана с хранением данных и манипулированием ими, то следует остановиться на вопросе об информационном содержании данных. Техническая сторона проектирования базы данных может быть рассмотрена независимо от информационного значения хранимых данных. Тот факт, что в этой главе мы не регулярно обращаемся к вопросу о значении данных, не означает, что проектировщики систем обработки информации должны опускать рассмотрение этого вопроса в своей практической работе. Для специалиста в области ЭВМ было бы ошибочно не анализировать разветвления рассматриваемой задачи и не исследовать пути их применения



Рис. 19.6. Уровни базы данных.

в различных областях. Вопросы использования информации в коммерции, политике и области принятия решений рассматриваются в ряде учебников.

В этой главе не рассматривается также и другой вопрос, касающийся баз данных, а именно проектирование и конструирование устройств хранения данных. Хотя вопрос о характеристиках аппаратных средств ЭВМ является центральным при оценке производительности систем баз данных, ему уделено немного внимания. Очевидно, что хорошее знание аппаратных средств, из которых строятся блоки вычислительных систем, необходимо только разработчикам систем. Для инженеров в области ЭВМ также имеются учебники различных уровней сложности. В этой главе содержится достаточно большое количество материала, соответствующего средней части рис. 19.6.

В табл. 19.1 представлены на обсуждение различные вопросы и показана взаимосвязь между ними. В этой таблице перечислены также некоторые области исследования, в которых может быть получен основной или дополнительный материал для каждого из указанных уровней. Не предполагается, что читатель хорошо разбирается в этих областях; они перечисляются просто как вспомогательные сведения.

Термин **именование** указывает, что для обращения к данным создается символическое имя, в минимальной степени связанное с местоположением и формой данных. Термин **адресация** указывает, что определяется точное расположение и форма представления данных. Из табл. 19.1 видно, что при переходе на более низкие уровни термин **именование** все чаще заменяется термином **адресация**.

Таблица 19.1. Уровни структуры базы данных ¹⁾

Уровень 1.	На концептуальном уровне мы имеем дело с информацией, которая несет в себе достаточно смыслового значения для того, чтобы можно было предпринять какие-либо действия. Информация характеризуется новизной и отсутствием избыточности, так что ее получение обогащает наши знания
А.	Некоторые процессы, используемые для получения информации: Перекрестное табулирование Сообщение об особых случаях Математический и статистический анализ Линейное и динамическое программирование Анализ решений Дедуктивное заключение
Б.	Интеллектуальные средства обработки информации: Интуитивные и формальные модели рассматриваемого объекта Теория информации Эвристические процедуры Формализованные концептуальные соотношения
В.	Необходимое для получения информации: Именованные и описанные данные Средства отбора и сбора данных в соответствии с описанием и содержанием Средства интеллектуального или автоматического анализа таких данных
Уровень 2.	На описательном уровне мы имеем дело с данными, текстом или числами, описывающими элементарные данные или события
А.	Процессы, используемые для получения данных и манипулирования ими: Сбор данных Отбор данных Организация данных Выборка данных Сокращение числа данных
Б.	Интеллектуальные средства манипулирования данными: Логика Понятия, связанные с ЭВМ Теория языков и метаязыки Исчисление высказываний
В.	Необходимое для манипулирования большим числом данных: Именованные файлы Именованные или допускающие адресацию записи Элементы данных, допускающие адресацию Каталоги и описания имеющихся средств Средства получения ресурсов системы Библиотеки процедур

¹⁾ Автор предлагает модель разбиения на уровни, не соответствующую общепринятой. (См Дж Мартин. Организация баз данных в вычислительных системах. М., Мир, 1980.) — *Прим. ред.*

Таблица 19.1 (продолжение)

Уровень 3.	На организационном уровне мы имеем дело с образом данных, т.е. данными, оторванными от их содержания, но именнованными и допускающими обработку
А.	Процессы, используемые для представления данных и манипулирования результатами вычислений: Кодирование и декодирование Определение памяти Управление файлами Управление размещением и составление каталога Перемещение и сравнение полей, содержащих закодированные данные Проверка ошибок
Б.	Интеллектуальные средства, используемые для представления данных и манипулирования ими: Теория информации Арифметические процессы Языки программирования Сведения об архитектуре ЭВМ и работе ее компонент
В.	Необходимое для представления данных: Устройства хранения закодированных данных Аппаратные средства для записи полей, имеющих адреса, в память Элементарные средства для сравнения кодов
Уровень 4.	Наконец, на материальном уровне появляются аппаратные средства, оборудование, используемое для обеспечения в значительной степени неформатизованной адресуемой памятью
А.	Процессы, используемые для получения аппаратных средств: Инженерное проектирование Производство
Б.	Инженеру, проектирующему аппаратные средства, необходимо: Обладать знаниями в области электроники, физики и логики Иметь практический опыт
В.	На что в свою очередь должен полагаться инженер: Компоненты, устройства и измерительные приборы Производственное оборудование

Сравнение. Можно провести сравнение между уровнями, описанными в табл. 19.1, и этапами проектирования железной дороги.

1. *Концептуальный уровень* связывается с идеей создания транспортной сети для перевозки товаров и определением ее экономического значения.

2. *Описательный уровень* связывается с планированием, определением размеров поездов и оборудования, необходимых для перевозки товаров за приемлемую стоимость и обеспечения достаточной гибкости.

3. *Организационный уровень* связывается со строительством рельсовых путей, работой сортировочных станций и определением производительности комплекса.

4. *Материальный уровень* связывается с проектированием локомотивов, грузовых вагонов, сигнальных механизмов, рельсов и шпал и несметного числа различного оборудования, необходимого для работы железной дороги.

Описание и организация. В этом разделе рассматриваются два центральных уровня, показанных в таблице: описательный и организационный.

На **описательном уровне** мы представляем процессы, функции которых определяются в соответствии со значением данных или полезностью последних для некоторых задач. Данные должны быть описаны на языке, способствующем их отбору с помощью таких процессов. Весьма важным является содержание элементов данных. Системы баз данных представляют средства, позволяющие связать данные в соответствии с их значением.

На **организационном уровне** мы представляем процессы, которые локализуют и перемещают данные в соответствии с требованиями эффективности. На этом уровне наряду с манипулированием самими данными возможно манипулирование описаниями данных. Здесь не учитывается содержание элемента данных, а играет роль его размер и расположение. Файловые системы обеспечивают возможность выборки записей, содержащих данные, в соответствии с относительным расположением или именами записей.

Эти два уровня находятся в зависимости друг от друга. На описательном уровне можно легко задать требования, которые на организационном уровне не могут быть в достаточной мере удовлетворены с помощью имеющихся аппаратных средств. С другой стороны, организация файлов, которая лишь немного расширяет возможности основных аппаратных средств, может наложить ограничения, снижающие эффективность процессов описательного уровня.

Эти два уровня нужны для того, чтобы отделить системы поддержки баз данных и прикладные системы. На уровне файлов структуры проектируются с использованием общих требований поддерживаемости в качестве основы. Эти требования могут быть удовлетворены с помощью программ, которые в различных приложениях остаются схожими.

Характер применения базы данных зависит от того, как лицо, принимающее решение, представляет задачи обработки информации, и все время изменяется в зависимости от потребностей пользователей базы данных, используемых ими методов анализа и способностей пользователей.

Для того чтобы гарантировать разделение уровней, в системах, в которых поддерживается концепция различных уровней, можно скрывать структурные детали низких уровней от пользователей высоких уровней. Эта идея **засекречивания инфор-**

мации может быть реализована только в тех случаях, когда функции низких уровней являются надежными, полными и достаточно эффективными.

19.5. СОВРЕМЕННАЯ ПРАКТИКА

В предыдущем разделе был введен ряд понятий на абстрактном уровне. Эту теоретическую базу следует дополнить несколькими практическими примерами, которые рассматриваются ниже.

Программа обработки файлов

Для того чтобы показать возможности использования файлов, приведем пример программы, выполняющей действия, указанные на рис. 19.5. При этом ставится цель привести пример не хорошей или плохой, а типичной программы.

Пример 19.1.

```
/* Программа обработки транзакций „raise“ (увеличение) */
raise: PROCEDURE(message);
    DECLARE message CHAR(200);
s1: DECLARE emp_list(5000) INITIAL((5000)0);
s2: DECLARE workspace(20), title CHAR(8), pct CHAR(2);
    title = SUBSTR(message, 6, 8); pct = SUBSTR(message, 17, 2);
s3: OPEN FILE(job_classification) DIRECT;
s4: READ FILE(job_classification) KEY(title) INTO(emp_list);
    CLOSE FILE(job_classification);
    DO i = 1 TO 5000 WHILE emp_list(i) ≠ 0; END;
    no_employees = i - 1;
    PUT SKIP EDIT (no_employees, title) ('Имеется', I5, ' ', A8);
    OPEN FILE(payroll) SEQUENTIAL UPDATE;
    total_increase = 0;
    DO j = 1 TO no_employees;
s5: READ FILE(payroll) KEY(emp_list(j)) INTO(workspace);
    /* Увеличение значения в поле зарплаты */
    increase = workspace(18) * pct / 100;
    total_increase = total_increase + increase;
    workspace(18) = workspace(18) + increase;
    REWRITE FILE(payroll) FROM(workspace); END;
CLOSE FILE(payroll);
PUT SKIP EDIT(total_increase) ('Расходуемая сумма', F(8, 2));
RETURN;
END raise;
```

Операторы **s1** и **s2 (DECLARE)** определяют рабочие области для размещения копии записи, используемой в программе. Все элементы одной записи связаны благодаря тому, что они хранятся в соседних ячейках файла и в описанных массивах. Операторы **OPEN** идентифицируют файлы, определяют характер их использования и подготавливают файлы к обработке. Имя запрашиваемого файла ищется в справочнике имен файлов. Автор программы полагает, что с этого момента файл находится под монопольным управлением программы, которое завершится в тот момент, когда файл будет закрыт с помощью оператора **CLOSE**. При выполнении оператора **OPEN** может потребоваться выполнение нескольких модулей. Поэтому время выполнения оператора **OPEN** может быть сравнительно большим.

Оператор **s4(READ)** выбирает особую запись, названную как **'Managers'**, с тем чтобы получить список номеров служащих. Оператор **s5(READ)** идентифицирует запись файла, соответствующую указанному служащему (**emp_list(j)**), и передает копию этой записи в область **workspace**. Система должна найти указанную запись в файле и вычислить ее адрес¹⁾. Для области **workspace** система использует оперативную память. Адрес для **workspace** определяется при компиляции и загрузке программы.

С помощью приведенной программы, используя поля адресации²⁾ в копии записи, можно извлечь необходимую информацию, например размер зарплаты. При этом все манипуляции производятся с копией записи. Комментарий помогает понять содержание полей адресации. Оператор **REWRITE** завершает обновление записи в файле. В своей работе он использует адрес записи, генерируемый файловой системой при выполнении оператора **READ**. Файловая система должна хранить этот адрес в течение всего времени выполнения оператора **READ**, чтобы затем использовать его при выполнении оператора **REWRITE**.

Использование имен или адресации? В этом примере записи в файле специфицируются с помощью имен, а именно заглавия (**title**) или номера служащего (**emp_list(j)**). Файловая система выполняет некоторую вычислительную процедуру с целью определить действительный адрес соответствующей записи и затем выбирает запись. Система запоминает этот адрес, для того чтобы изменять файл, т. е. чтобы надлежащим образом выполнить оператор **REWRITE**. Внутри выбираемой записи мы разместили

¹⁾ Предполагается использование операции «поиск по ключу», с помощью которой отыскивается запись. Адрес записи становится известным, когда она найдена. — *Прим. ред.*

²⁾ Имеется ввиду, очевидно, относительная адресация, подразумевающаяся форматом записи. — *Прим. ред.*

данные, указывая их адреса. Так, поле зарплаты было определено как поле 18. При использовании адреса программист указывает расположение и размер элемента данных.

Управление с помощью операционной системы

Для обработки больших наборов данных, файлов, используются системные средства. Это обеспечивает независимость описания задачи от спецификации используемых аппаратных средств. Автор программы обработки файлов предполагает, что где-то существует эксперт, который управляет (непосредственно или с помощью программ) фактическим размещением записей. Этот эксперт управляет работой файловой системы, которая, как правило, является частью **операционной системы ЭВМ**.

Для того чтобы операционной системе передать информацию о программе, используются операторы, не содержащиеся в языке программирования. Эти операторы образуют **язык управления**, предназначенный для работы с операционной системой. Отсутствие связи между языком программирования и языком управления часто является препятствием на пути эффективного использования вычислительных систем.

Оператор языка управления для приведенного выше примера может выглядеть так, как показано в примере 19.2.

Пример 19.2.

```
FILE(data): (NAME("payroll"), DEVICE(diskdrive),  
            LOCATION(disk5), ORGANIZATION(fixed records  
            (80 bytes), indexed, sequential, buffered),  
            SIZE (60 tracks), и т. д.)
```

Для лучшего понимания этот оператор записан более подробно, чем допускается в большинстве языков программирования. При выполнении этого оператора файл **payroll** назначается программе на время ее работы.

Однако ответ на вопрос о том, принадлежит ли файл **payroll** программе в течение всего времени ее выполнения или в интервале между моментами выполнения операторов **OPEN** и **CLOSE**, или же только в интервале между моментами выполнения операторов **READ** и **REWRITE**, не является однозначным. Он зависит от используемой операционной системы. Для того чтобы не допустить ошибки, другая транзакция, вычисление суммарной зарплаты для составления бюджета, должна быть отложена до полного завершения процедуры **raise**.

Язык программирования можно изучать отдельно от операторов языка управления, однако для разработки эффективных

систем баз данных необходимо знать оба этих языка. Однако для того чтобы проектировать прикладные системы, использующие базы данных, требуются более обширные знания. Анализ полученных данных может потребовать опыта, отличного от того, который требуется для проектирования, организации или сбора и изменения данных. Для того чтобы управлять базой данных как целым, может потребоваться специалист — администратор баз данных.

Модульность

Поскольку разработка прикладных систем с использованием баз данных становится все более сложной, возможно дальнейшее разделение функций. Программы, которые работают с базой данных, называются **модулями**. Они используют другие программные модули и сами используются модулями более высокого уровня. Программа **raise** реализует один тип операций в системе обработки транзакций и вызывается в тех случаях, когда во входном сообщении **message** указано, что требуется повысить зарплату. Операторы **OPEN**, **READ**, **REWRITE** и т. д. являются вызовами модулей файловой системы. Высокая степень модульности необходима по следующим причинам:

1. Масштабы разрабатываемых проектов требуют совместной работы многих людей, а каждый человек в отдельности выполняет специальную функцию.

2. Одну и ту же базу данных могут использовать многие проекты, поскольку спецификация базы данных становится суммой многих требований.

3. Информация, содержащаяся в базе данных имеет определенную ценность, поэтому неразумно предоставлять неограниченный доступ к ней каждому, кто связан с базой данных. Для обеспечения защиты личных данных, доступ к ним контролирует операционная система.

В описанном выше примере все строки программы составлялись человеком, знающим содержание задачи (специалистом). Оператор языка управления мог быть написан помощником специалиста. Процессы, выполняемые системой (вызываемые операторами **OPEN**, **READ** и **REWRITE**), вероятно, были написаны заранее и в другом месте лицом, не знающим содержания задачи.

Разделение функций и средств при разработке системы диктуется главным образом соображениями удобства и зависит от объема вычислений и знаний отдельных программистов. Разработка более формальных критериев модульности осуществляется в настоящее время и становится важной стороной управления вычислениями.

Приводя этот пример, мы не преследовали цель критиковать ПЛ/1 как язык программирования. Не существует машинного языка, который вынуждал бы пользователя создавать программу с блочной структурой в соответствии с уровнями, которые мы описали. Однако имеется ряд языков, сильно затрудняющих структурирование.

19.6. ОПИСАНИЯ

Участие многих людей в разработке и использовании системы, становится возможным только при наличии документации, в которой были бы отражены вопросы, касающиеся как статики системы, г. е. **структуры файлов**, так и ее динамики, т. е. **вычислений**, с помощью которых осуществляются преобразования данных.

К вопросам статики относится описание данных с точки зрения их разбиения на файлы, записи, поля и указание взаимосвязи между отдельными данными, содержащимися в этих элементах. Такое описание структуры данных может иметь форму документа, содержащего указания для людей, составляющих программы обработки файлов с использованием этой базы данных. Альтернативой этому является преобразование описания структуры в набор кодов, читаемых машиной, или в схемы, которые обеспечивают автоматическую обработку файлов. Большая часть этой главы связана с проектированием и использованием таких схем.

Описание вычислений может быть дано в виде формул, описания некоторого числа секций, которые должны быть выполнены, или блок-схемы. Во многих коммерческих программистских группах много усилий направлено на системный анализ, в результате которого до мельчайших подробностей составляется описание процесса, используемое в дальнейшем кодировщиками. Однако при обмене информацией между специалистами по анализу систем и кодировщиками легко могут быть допущены ошибки. Специалист по анализу может предполагать, что кодировщики знакомы с условиями, которых на самом деле они не знают. Кодировщики могут составить программы для ситуаций, которые никогда не возникают. Часто затруднения вызывает определение критических секций. Кроме того, не существует формальных методов проверки полноты описаний для интерактивных процессов. Иногда для постериорной проверки проекта или для наглядности документации автоматически составляются блок-схемы написанных программ.

Сопоставление описаний статики и динамики системы. При описании файлов вопросы, касающиеся статики и динамики, не

всегда различимы. При решении простых задач обработки данных описание процесса может быть полностью исключено. Вычисления описываются косвенно заданием двух копий файла, определяющих форму и содержание до и после вычислений. Этот метод может быть также применен при передаче данных между двумя отдельными файлами. Полное исключение процесса кодирования является задачей **генераторов отчетов**, на вход которых поступают описания исходных файлов и формы выдаваемого отчета. Все шаги обработки подразумеваются. Та же цель преследуется при использовании справочной системы высокого уровня, где информационно-поисковый запрос формулируется в виде формулы, а получение ответа на него является задачей информационно-поисковой системы.

Иногда организация данных однозначно определяется указанием шагов процесса, осуществляющего размещение данных в базу данных. Таким способом часто описываются сложные взаимосвязанные структуры, поскольку статическое изображение может быть неясным. Указанным двум вариантам соответствуют два термина для описания структуры хранения: **организация файлов и методы доступа**.

Если файл обрабатывается несколькими процессами, то существенными могут быть статическое описание вместе с указанием условий, при выполнении которых это описание верно. До того как эти процессы начнут соответствующим образом обрабатывать файл, указанные условия должны быть проверены либо самим процессом, либо вызванной критической секцией.

Одним из таких условий может быть, например, полное обновление файла. Более детальные условия (информация о состоянии) могут указывать, что определенные записи еще не были обновлены, например, из-за отсутствия информации, такой, как количество часов, отработанных некоторыми служащими. Некоторые вычисления могут быть запрограммированы с целью принять решение о завершении обработки; другие процессы продолжают выполняться, если для них не требуются еще необновленные записи.

Как видно, существует взаимосвязь между качеством, или полнотой, описания данных и нашей способностью усовершенствовать и классифицировать секции, объединенные в программу обработки данных.

Блок-схемы. Для описания сложных процессов и потоков данных иногда в этой главе мы будем пользоваться графическим изображением динамики алгоритма, или блок-схемой (рис. 19.7). В блок-схемах различают три вида элементов:

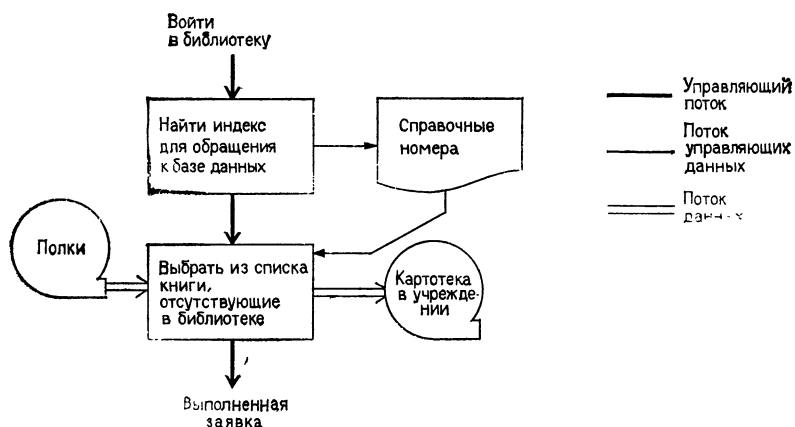


Рис. 19.7. Блок-схема.

Управляющий поток — последовательность команд, выполняемых устройством обработки при различных условиях. Эта последовательность описывается в тех случаях, когда блок-схемы используются для документирования алгоритма работы программы.

Поток управляющих данных — поток элементов данных, воздействующий на управляющий поток обработки. Типичными примерами являются счетчики, переключатели управления, которые указывают, разрешается ли выполнение операции, заполнена или пуста область и т. д. Они могут быть установлены в одной программе, а использоваться в другой.

Поток данных — поток большого числа закодированных данных, используемый с целью передачи информации для пользователей процессом, а не для управления самим процессом. Этот поток обычно находится между файлами и устройствами ввода или вывода.

Последние два вида элементов иногда трудно различимы, поскольку некоторые управляющие данные часто извлекаются при обработке данных. Однако во многих случаях знание этих различий очень полезно для понимания функций процессов. На рис. 19.7 показаны три типа потоков в процессе.

С использованием современных средств программирования, включая языки высокого уровня, четко определенные модули, хорошо структурированные процессы и документированные стандартные алгоритмы, необходимость в описании динамики программы уменьшается. Если взаимодействие процессов не является чрезмерно сложным, то более предпочтительным представляется описание состояния файла и режима работы программы.

19.7. ПРИВЯЗКА

До сих пор многие из тех, кто занимается обработкой данных, полагают, что содержание и структура базы данных должны быть полностью определены до написания процедур, использующих базу данных. Такое допущение лишает нас основной выгоды, которая может быть получена при передаче наших процессов ЭВМ. При проектировании и реализации системы баз данных желательно использовать такие методы, которые позволяют постоянно усовершенствовать и изменять ее, в результате чего отпадает необходимость предусматривать все детали заранее. Тот факт, что обработка данных и выполнение процедур могут осуществляться с помощью ЭВМ, делает возможным добавление данных и связей в базу данных в более позднее время. Операции, необходимые для осуществления таких изменений, могут быть нетривиальными, однако система, в которой предусмотрены возможности расширения, может длительное время не устаревать.

Каждый раз, когда при разработке базы данных реализуется решение, определяющее фиксированную взаимосвязь между элементами данных, число возможных вариантов проекта базы данных уменьшается. Процесс упорядочения данных, или установление связей, называется **привязкой**. На последней стадии обработки данных должны быть установлены все связи, необходимые для получения нужной информации.

Время привязки. Понятие привязки обычно используется для того, чтобы различать процессы компиляции (**ранняя привязка**) и процессы интерпретации (**поздняя привязка**). Диапазон вариантов привязки при проектировании баз данных значительно шире, чем в программировании. Для реализации привязки в новой системе баз данных часто требуются годы.

Варианты привязки покажем с помощью расширения примера 19.1. Полагая, что запись 'managers' (руководители) не найдена в файле 'job-classification', но должна быть получена посредством сканирования файла 'department' с записями, описывающими отделы, можно записать

```
s3: OPEN FILE(department) SEQUENTIAL; no_employees = 0;
    ON ENDFILE(department) GO TO tell;
read_next: READ FILE(department) INTO(workspace);
s6: no_employees = no_employees + 1;
    emp_list(no_employees) = workspace(8);/*номер руководителя*/
    GO TO read_next;
tell: PUT SKIP EDIT(no_employees, ...
```

Здесь привязка осуществляется в предположении, что существует взаимно однозначное соответствие между отделами и

руководителями. Если один человек руководит двумя отделами, то с помощью процедуры **raise** ему будет дана 21%-ная надбавка вместо 10%-ной. Если между **read_next** и **s6** включить оператор

```
DO i = 1 TO no_employess/*поиск руководителей двух отделов*/
  IF emp_list(i) = workspace (8) THEN GO TO read_next;
END;
```

то привязка будет отложена до времени выполнения процедуры **raise**. Если привязка откладывается до того момента, когда будет осуществлен доступ к каждому отдельному элементу, то большой объем вычислений выполняется при фактической обработке. Поздняя привязка может быть особенно дорогой в результате того, что, во-первых, она выполняется в критическое время, когда требуется получить ответ, и во-вторых, она должна выполняться поэлементно, а не один раз для всех элементов, когда определяется структура.

Ранняя привязка может быть выбрана по той причине, что она может обеспечить высокую исходную эффективность. В базе данных большое число связей возникает сразу после того, как проектировщик указывает позиции элементов данных в файлах. Последующие модификации связанных структур будут чрезвычайно затруднены и часто являются причиной неэффективной обработки. В тех случаях, когда необходимо добавить новые типы данных или связей, требуется полная реорганизация структуры файла и модификация программ, в которых они используются.

Чтобы избежать затрат, возникающих в результате проводимых изменений, дополнительные элементы данных можно помещать в позиции, первоначально отведенные для других целей. Такая ситуация аналогична той, когда изменения на уровне машинного языка вносятся в объектный текст программы, создаваемой компилятором. Подобная ситуация теперь не встречается в программировании, поскольку затраты на повторную компиляцию невелики. В области управления файлами использование вставок в структуры данных все еще является распространенным средством. Ниже приводится пример такой вставки.

Пример 19.3. Записи в файле сведений о служащих имеют следующие поля: **имя, адрес, пол, зарплата, служба в армии, дата принятия на работу.**

Требуется добавить поле **декретный отпуск**. Записи используются многими программами, хотя только некоторые из них обращаются к полю **служба в армии**. Решение может заключаться в том, чтобы использовать это поле в качестве поля **декретный отпуск** в каждом случае, когда указывается женский пол.

В этом примере решения, принятые во время ранней привязки, ограничивают реализацию средств более поздней системы.

Отсутствие подходящей методологии программирования и стремление использовать вставки вместо соответствующих изменений явились причиной неудач многих программистов. Результатом часто является низкая производительность специалистов. Значительная часть процесса проектирования, предлагаемого в этой главе, может быть сведена к принятию решений относительно выбора наилучшего способа привязки. Решение проектировщика относительно времени привязки заключается в выборе оптимального компромисса между эффективностью и гибкостью.

19.8. КЛАССИФИКАЦИЯ ОПЕРАЦИОННЫХ СИСТЕМ

При проектировании баз данных могут быть использованы любые из многочисленных операционных систем, существующих в настоящее время. Операционные системы выполняют разнообразные функции, основными из которых являются следующие:

- Управление процессами
- Поддержка файловой системы
- Поддержка ввода и вывода
- Учет расхода ресурсов ЭВМ

В этой главе будут рассмотрены файловые системы. Управление процессами осуществляется системным процессом. Он выделяет имеющиеся ресурсы процессам пользователей или заданиям, содержащим последовательность процессов, подлежащих выполнению на ЭВМ. Ресурсы ЭВМ включают в себя центральный процессор, оперативную память, содержащую процессы, память для хранения файлов и устройства ввода и вывода, включая терминалы пользователей. Планирующая система при необходимости также использует ресурсы ЭВМ, причем чем сложнее эта система, тем больше ресурсов для нее требуется.

Пакетная обработка

При последовательном выполнении вычислений, или пакетной обработке, все ресурсы ЭВМ или большая их часть находится в распоряжении одного процесса в течение требуемого промежутка времени. На рис. 19.8 изображена диаграмма, иллюстрирующая этот метод, а также методы, которые будут рассмотрены в оставшейся части этого раздела. Очевидно, что при использовании метода однопоточковой пакетной обработки небольшим числом заданий все имеющиеся ресурсы будут использоваться эффективно.

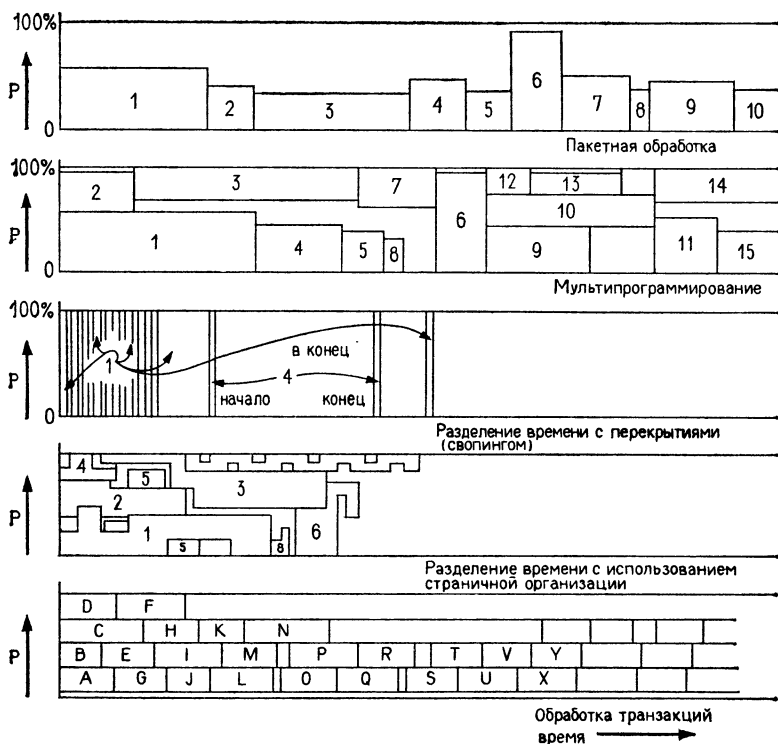


Рис. 19.8. Распределение ресурсов для пяти типов операционных систем.

На рис. 19.8 показана схема распределения одного ресурса. На самом деле при выполнении вычислений используются все упомянутые выше ресурсы. Как правило, один из ресурсов обладает ограниченными возможностями, в результате чего другие ресурсы не могут быть использованы в полной мере.

Типичным является пример, когда данные считываются и затем обрабатываются. Считывание может выполняться с максимальной скоростью устройством ввода, однако при этом может случиться так, что используется лишь несколько процентов производительности центрального процессора, выполняющего требуемые вычисления со считанными данными. Впоследствии может выполняться большой объем математических вычислений и считывающее устройство должно будет находиться в состоянии ожидания команд, дающих необходимую информацию о следующей операции считывания.

Мультипрограммирование

При мультипрограммировании предпринимается попытка более эффективно использовать ресурсы, разделяя их между несколькими процессами в соответствии с текущими запросами последних. Если некоторому процессу требуется ресурс, который не может быть выделен немедленно, то этот процесс откладывается на более позднее время, так что операционная система может начать выполнение процесса, ожидающего освобождение ресурса. Каждый отдельный процесс вычислений выполняется дольше, однако, если удастся достаточно хорошо распределить ресурсы между процессами, то общая производительность системы возрастет. Заметим, что переключения между процессами требуют некоторых затрат процессорного времени.

Разделение времени

Разделение времени — это режим работы ЭВМ, при котором процессы делятся на кванты, а распределяются только дефицитные ресурсы между активными процессами в соответствии с требованиями баланса производительностей системы и пользователей. Пользователь за терминалом теперь также может рассматриваться как дефицитный ресурс. К сожалению, производительность пользователей мала, и в тоже время они очень нетерпеливы, когда им нужно инициировать вычисления. Быстрый ответ таким пользователям обеспечивается тем, что с помощью прерываний по таймеру время выполнения конкурирующих квантов вычислений ограничивается долей секунды. Когда пользователь сообщает, что он готов использовать вычислительную машину, его процесс помещается в очередь и выполняется после очень небольшой задержки. В большинстве систем с разделением времени наиболее дефицитными аппаратными ресурсами являются центральный процессор и оперативная память. В периоды ожидания процессы пользователей могут “выгружаться” из основных ресурсов. Неактивные процессы занимают область на диске или барабане, организация которой может не удовлетворять требованиям, предъявляемым к файлам данных.

Режим разделения времени позволяет эффективно использовать центральный процессор и оперативную память в тех случаях, когда тип вычислений такой, что для их выполнения ресурсы требуются на небольшие периоды времени. Возникающие паузы часто связаны с ожиданием ответов пользователей, работающих за терминалами. Требования, предъявляемые ко многим операциям базы данных, определяются тем, что с ней будут связаны пользователи, работающие за терминалами. По-

этому взаимосвязь систем баз данных и методов разделения ресурсов становится очень важной. Этот режим выполнения операций обеспечивает эффективную работу пользователя за счет значительной потери в производительности ЭВМ, возникающей из-за частых переключений между квантами процессов.

Страничная организация памяти

Один квант процесса, выполняемого в режиме разделения времени, редко требует большое число ресурсов, поэтому разбиение такого ресурса, как область памяти, используемая пользователями, на страницы способствует более эффективному распределению этого ресурса между процессами и сокращению объема области перекачки. Число страниц, действительно необходимых для выполнения требуемого объема вычислений, будет изменяться от кванта к кванту. Страницы выделяются по требованию или с помощью прогноза, основанного на анализе опыта предыдущей работы системы. Страницы, которые использовались недавно, по возможности сохраняются в оперативной памяти. Страничная организация может также использоваться для распределения памяти при мультипрограммировании.

Обработка транзакций

В тех случаях, когда объемы вычислений невелики, может быть выбран тип управления, называемый обработкой транзакций. Например, небольшим может быть объем вычислений, требуемых для обработки запроса на поиск одного элемента хранимых данных. В этом случае процессы стартуют сразу после поступления запроса на выполнение вычисления и могут занимать центральный процессор до тех пор, пока это возможно. В тот момент, когда процесс переходит в состояние ожидания ответа от терминала или прихода данных из файла, управление передается системе и может быть начата или продолжена обработка другой транзакции. Если процесс находится в критической секции, то он может даже не передавать управление системе, с тем чтобы предотвратить освобождение файла, состояние которого не определено четко. Когда обработка транзакции завершается, система оповещается об этом и освобождает все ресурсы, отведенные для данной обработки. В примере 19.1 показана типичная программа обработки транзакций.

Объем работ по контролю вычислений, которые будут выполняться, увеличивается, чтобы предупредить монополизацию всей системы одним процессом. Этот тип контроля используется в тех случаях, когда вычисления точно определены или ограничены. Поскольку затраты на контроль процесса выполнения

транзакции невелики, то операционная система может быть значительно менее сложной. Процесс, для которого расход ресурсов значительно превышает допустимые величины, может быть принудительно завершен. В режиме обработки транзакций могут эффективно выполняться многие действия с базой данных.

Затраты, связанные с разделением ресурсов

Любая форма разделения ресурсов требует определенных затрат на создание соответствующих операционных средств и средств управления. Если режим разделения ресурсов не дает существенного увеличения эффективности системы, предпочтительнее является специализированная система подходящих размеров. Конечно, с помощью такой системы нельзя будет успешно обрабатывать неоднородный высокоинтенсивный поток требований. Наибольший интерес для проектировщиков систем представляют смешанные системы, которые являются специализированными и в то же время имеют доступ к основным разделяемым ресурсам. Разработка таких систем находится в начальной стадии, поэтому в этой главе мы не будем останавливаться на их рассмотрении.

19.9. ПРИМЕНЕНИЕ БАЗ ДАННЫХ

В настоящее время системы баз данных используются в следующих областях:

- Обработка промышленности (управление запасами, обработка списков материалов и производственное календарное планирование)
- Отрасли инфраструктуры (обработка списков услуг и составление схем размещения)
- Экономика (обработка данных о производстве и потреблении с целью планирования и распределения)
- Управление финансами (составление отчетов, подведение баланса, анализ конвертируемости фондов)
- Научные исследования (обработка ранее собранных данных с целью определить направления будущих исследований)
- Медицинское обслуживание (обработка записей о пациентах, изучение историй болезней, классификация задач, поиск эффективных методов лечения)
- Обработка описательных данных (о людях или объектах)
- Обслуживание библиотек (составление каталогов, поиск с помощью индексов)

Управление запасами в обрабатывающей промышленности, особенно на предприятиях, состоящих из многих частей и подразделений, — это одна из областей, в которых системы баз данных используются наиболее длительное время и наиболее продуктивно. Основная цель этих систем обработки списков материалов заключается в анализе производства и планировании подачи деталей, являющихся компонентами различных готовых изделий и, следовательно, наиболее дефицитными. Мы приведем немало примеров файлов сведений о служащих, так как содержание и функции таких файлов часто являются очевидными сами по себе. Более интересные и более важные приложения связаны с планированием производства пищевых продуктов и энергетических ресурсов.

Влияние области применения базы данных

При решении любой задачи с помощью ЭВМ необходимо подробно описать поставленную цель. Только после этого можно определять и оценивать релевантность содержания элементов данных, процессов, требуемых для манипулирования данными, и структур файлов, необходимых для эффективного выполнения процессов. Необходимо хорошо представлять себе область приложения базы данных, чтобы была ясна семантика получаемой информации. Необходимо также знать требуемое время отклика, с тем чтобы выходные данные поступали из системы своевременно. Все эти сведения являются исходными данными для анализа системы и последующего выбора аппаратных средств. Информация о динамике среды области приложения будет полезна при определении степени формализации организации данных и выборе соответствующих времен привязки.

Формулирование целей и ограничений, описывающих задачу и среду области приложения, является важной частью любого проекта, осуществляемого на практике. Оно должно также присутствовать при описании любой решаемой в качестве упражнения задачи по теме этой главы. Частой ошибкой, встречающейся при изложении целей, является то, что методы и технология, которые предполагается использовать, описываются как часть цели. Описание методов и технологии следует помещать в раздел, который содержит результаты анализа системы.

Формулировка цели, в которой указано, что причиной создания базы данных явилось желание использовать ЭВМ, должна вызвать содрогание у любого специалиста в области вычислительной техники. Если выбор методов или технологии был преднамеренно ограничен, то это обстоятельство должно быть пояснено в отдельном разделе формулировки цели.

19.10. ОРГАНИЗАЦИЯ ФАЙЛОВЫХ СИСТЕМ

Основными характеристиками, которыми должны обладать системы, предназначенные для хранения большого количества данных, являются быстрота доступа к данным с целью их выборки, удобство средств и методов обновления данных и эффективное использование памяти. Другими важными характеристиками являются наличие возможности представлять реально существующие структуры, надежность, защита секретных данных и целостность. Для обеспечения возможности проведения анализа характеристик система должна проектироваться с использованием четко выделяемых абстрактных структур. Приведенные требования к системе противоречат друг другу. Выбор способа организации файла определяет эффективность функционирования системы относительно указанных критериев. Вначале мы дадим оценку файлам в соответствии с основными характеристиками. Это важно сделать, так как хорошее соответствие возможностей файловой системы и требований к ней со стороны системы управления базой данных определяет успех совокупной системы.

Опишем здесь шесть основных способов организации файлов и оценим эффективность их использования. Большая часть типов структур, используемых на практике, либо совпадает с одним из тех, которые будут рассмотрены, либо является их комбинацией.

Шесть типов файлов, которые будут рассмотрены в настоящей главе, следующие: файл-множество, последовательный файл, индексно-последовательный файл, индексированный файл, прямой файл и многокопцевой файл. Выбор в качестве основных моделей этих шести был продиктован тем обстоятельством, что все они тесно связаны с системами, которые используются на практике. Иными словами, при выборе этих шести моделей не преследовалась цель представить минимальный набор независимых способов организации файлов.

Предыдущее определение файла теперь можно расширить, указав, что файл не только состоит из однотипных записей, но также имеет определенную организацию.

Справочники файлов

Файл может иметь заголовки, или справочную запись. Эта запись содержит информацию, описывающую расположение и формат записей файла. Для различных типов организации файлов существуют различные требования, накладываемые на содержание справочника файла. Значительная часть информации, хранящейся в справочнике, связана с распределением памяти

и понятиями, касающимися базы данных. Поэтому при последующем анализе файлов справочные записи отдельно рассматриваться не будут. Типичными элементами данных, хранимых в справочнике, являются имя, владелец, начальная точка, конечная точка, допустимый объем памяти и фактически используемый объем памяти. Набор справочных записей для некоторого числа файлов в свою очередь также может образовывать файл. Владельцем этого справочного файла обычно бывает операционная система.

Реально действия, связанные со справочными записями — это чтение записей, осуществляемое в момент начала вычислений, использующих соответствующий файл, и хранение информации для последующих ссылок. Этот процесс называется **открыть файл**. В тех случаях, когда в файле были проведены такие изменения, которые должны быть отражены в справочнике, последний будет обновлен с помощью соответствующего процесса, выполняемого при завершении использования файла. Это называется **закрыть файл**.

Описания файлов

В каждом описании организации файлов будут встречаться определения, относящиеся также и к способам организации, излагаемым ниже. Это означает, что изучать материал этой главы лучше всего в той последовательности, в которой он изложен.

При описании и анализе каждого способа организации файлов рассматривается одна определенная организация записи. Хотя рассматриваемая комбинация является типичной в практике файловых систем, следует понимать, что допустимы и другие комбинации. Для каждой такой комбинации имеются соответствующие формулы, оценивающие эффективность использования системы. Эти формулы выводятся достаточно просто, так что другие варианты читатель может исследовать самостоятельно.

Оценки эффективности

Для определения эффективности использования файловой системы необходимы количественные оценки. Для каждого из шести способов организации файлов существуют семь следующих видов оценок:

- Память, требуемая для одной записи
- Время, требуемое для выборки произвольной записи из файла
- Время, требуемое для получения следующей записи в файле

- Время, требуемое для обновления путем включения записи в файл
- Время, требуемое для обновления путем изменения записи в файле
- Время, требуемое для считывания всего файла
- Время, требуемое для реорганизации файла

Как видно, существует шесть операций над файлами, которые в качестве компонентов включают установку механизма чтения-записи, а также считывание и запись блоков. Таким образом, вывод искомых формул будет основываться на оценках параметров аппаратуры. Использование обобщенных параметров обеспечивает некоторую независимость от физических характеристик аппаратных средств, так что анализ способов организации файлов можно продолжить без рассмотрения деталей, касающихся возможных реализаций аппаратных средств. При оценке эффективности использования файловой системы в каждом конкретном случае необходимо ответить на следующие вопросы:

- Требуется ли установка механизма чтения-записи или он находится в нужном положении, т. е. должна ли использоваться величина s ¹⁾?
- Осуществляется ли непосредственный переход к следующей записи или нет, т. е. какой из величин, 0 , r ²⁾ или $2r$, равно время ожидания?
- Осуществляется ли считывание только значащих данных или данных вместе с промежутками между ними, т. е. какая из величин, t или t' , используется в качестве скорости обмена данными³⁾?
- Определяется ли оценка для чистого количества данных или рассматривается требуемое пространство, т. е. какая из величин, R или $(R + W)$, используется в качестве меры?

В этой главе мы определим затраты на выполнение указанных операций с точки зрения требуемого времени.

Размер записи. В целях экономии памяти следует стремиться хранить данные с минимальной **избыточностью**. Наличие избыточности при хранении данных требует также больших усилий для их изменения. Избыточность возникает в тех случаях, когда поля данных дублируются (например, в записях 1 и 8

¹⁾ s — время подвода головок к нужному цилиндру. — *Прим. ред.*

²⁾ r — время полуоборота ЗУПД. — *Прим. ред.*

³⁾ В оригинале мысль выражена нечетко; очевидно, имеется в виду разница между временем считывания записи, расположенной полностью в пределах одного физического блока и так называемой расширенной (как в ОС ЕС), располагающейся в нескольких физических блоках. — *Прим. ред.*

табл. 19.2в) или когда описание содержания полей данных повторяется для каждого элемента. Для сокращения последней формы избыточности можно заранее фиксировать структуру записи, с тем чтобы позиция элемента данных служила в качестве компонента описания. Однако если данные очень неоднородные, то такая табличная организация в действительности может оказаться расточительной, поскольку необходимо будет содержать много незаполненных позиций. В табл. 19.2 показаны плотный, разреженный и избыточный файлы. Каждый из файлов содержит информацию в форме, удобной для определенной цели.

Выборка записи. Для того чтобы иметь возможность использовать данные, содержащиеся в файле, запись, объединяющая данные, должна быть считана в процессор. Выборка записи состоит из двух шагов: локализация записи и фактическое счи-

Таблица 19.2. Плотный, разреженный и избыточный файлы в базе данных, содержащей сведения о посещении лекций студентами

(а) Плотный файл

	Номер студента	Группа	Суммы	Незавершенные курсы	Текущая работа	Возраст	Специальность
1	721	Т	43	5	12	20	БВ
2	843	Т	51	0	15	21	Per
3	1019	Ф	25	2	12	19	Per
4	1021	Ф	26	0	12	19	Per
5	1027	Ф	28	0	13	18	Per
6	1028	Ф	24	3	12	19	БВ
7	1029	Ф	25	0	15	19	Per
8	1031	Ф	15	8	12	20	БУ
9	1033	В	23	0	14	19	БВ
0	1034	Ф	20	3	10	19	Per

(б) Разреженный файл

	Номер студента	Курсы лекций					Практ. работа
		ГС101	ГС102	Биз3	Р5	ИО103	
1	721	Пя72	Пя73		Ср73		
2	843	Пя72	Ср73				
3	1019		Су72	Су73			1
4	1021		Су72			Пя73	
5	1027	Пя73		Су73			
6	1028				Ср73		1
7	1029	Пя73	Ср73				
8	1031	Пя73					
9	1033						3
10	1034					Пя73	

(в) Избыточный файл

	Обязательный курс	Номер студента	Когда	Практ. работа	Накопленные суммы	...	Специальность
1	ГС102	721	Пя73		43	...	БВ
2	ГС102	843	Ср73		51	...	Пер
3	ГС102	1019	Су72		25	...	Пер
4	ГС102	1021	Су72		26	...	Пер
5	ГС102	1029	Ср73		25	...	Пер
6	Биз3	1019	Су73		25	...	Пер
7	Биз3	1027	Су73		28	...	Пер
8	Р5	721	Ср73		43	...	БВ
9	Р5	1027	Ср73		28	...	Пер
10	ИО103	1021	Пя72		26	...	Пер
11	ИО103	1034	Пя73		20	...	Пер
12	Практ.	1019		3	25	...	Пер
13	Практ.	1028		1	24	...	БВ
14	Практ.	1033		1	23	...	БВ
15	Никакой	1031			20	...	БУ

тывание. Термин **выборка** используется в тех случаях, когда поиск записи осуществляется без подготовки, т. е. выборке не предшествуют никакие операции, выполняемые для упрощения локализации и чтения записи. Для эффективного считывания данных необходимо быстро определять местоположение элемента, который должен быть считан. Наиболее предпочтительным кажется простой способ вычисления адреса, аналогичный тому, который используется при определении местоположения элемента массива в оперативной памяти. Однако такой способ приводит к потере гибкости в плане хранения данных всякий раз, когда данные по своей природе не могут быть записаны в виде таблицы или когда не все элементы таблицы заполнены, так что местоположение элемента не может быть определено непосредственно по значению индекса. Справочные таблицы или индексы помогают при поиске данных в тех случаях, когда местоположение не может быть определено непосредственно. Однако использование справочных таблиц и индексов увеличивает избыточность. Справочная таблица, которая может быть использована для доступа к другим файлам, показана в табл. 19.3.

Получение следующей записи. Единичные данные редко могут служить исчерпывающей информацией. Обычно полную картину создают данные, связанные друг с другом. При этом возникает необходимость получать последовательность записей в соответствии с некоторым критерием. И если выборку можно определить как ассоциативный поиск элемента данных, основанный на использовании ключа, то операцию получения сле-

Таблица 19.3. Справочная таблица

Имя	Номер студента	Обязательные элементы		
Башмен Уилд	1028	13		
Джейсон Пит	1027	7	9	
Джи Отто	1021	4	10	
Конт Мэри	843	2		
Мейкл Верна	1019	3	6	12
Олмейер Джон	1031	15		
Рьюноут Ремингтон	1033	14		
Хестон Чарльз	721	1	8	
Хоттен Донна	1029	5		
Эриксон Сильвия	1034	11		

дующей записи можно охарактеризовать, основываясь на структурной зависимости. Наиболее эффективно следующая запись может быть получена в тех случаях, когда связанные данные хранятся вместе, т. е. когда данные располагаются в строгой последовательности (одно за другим). Поскольку отношения между данными могут быть многомерными, а последовательностей эффективного доступа к физическим устройствам, на которых находятся данные, может быть лишь несколько, то может оказаться полезным использование памяти с высокой избыточностью или применение большого числа указателей, обеспечивающих связь со следующими записями. Чтение (или запись) записей в определенном порядке называется упорядоченным (serial) чтением (упорядоченной записью). Если, кроме того, записи упорядочены физически, то файл, содержащий их, может быть считан последовательно (sequentially). В табл. 19.2(в) показан пример, когда избыточность позволяет упростить группирование записей, указывающих обязательные курсы.

Включение записи. Во многих случаях для того, чтобы данные в файлах не устаревали, необходимо постоянно осуществлять включение (или добавление) новых записей. Операция записи в файл часто бывает намного более сложной, чем чтение файла. Добавление записей легче всего осуществляется в том случае, когда их можно поместить в конец файла, удлиняя его. Для того чтобы конечную точку файла можно было легко локализовать, ее адрес часто хранят в справочнике. В тех случаях, когда для упрощения упорядоченного доступа запись нужно расположить в определенном месте, включение записи потребует сдвига или модификации других записей. Хранение избыточных записей влечет за собой при обновлении файла большое число дополнительных операций записи.

Если в файлах, изображенных в табл. 19.2, требуется изменить элемент, соответствующий, например, графе “незавершенный курс”, то достаточно обновить только одну запись. Однако если же необходимо ввести в файл информацию о новом обязательном курсе и указать его завершение, то потребуется выполнить большое число операций.

При выполнении каждой операции записи в заблокированный файл необходимо прочитать блок, с тем чтобы иметь возможность до перезаписи всего блока объединить данные из соседних записей. В том случае, если мы получаем информацию в процессе вычислений, а не выбираем ее из архива, частота выполнения операции чтения будет значительно превышать частоту операции обновления файлов и записей. Поэтому часто стремятся использовать такую организацию данных, которая обеспечивает их быстрый поиск, даже если это приводит к усложнению операций обновления.

Обновление записи. Если данные, расположенные внутри существующей записи, должны быть заменены на новые, то обновленная запись формируется с помощью информации, получаемой из копии этой записи. Обновление файла осуществляется путем обновления записи. Если при этом размер записи увеличивается, для нее не может быть использовано старое место. В этом случае старую запись нужно сделать недействительной. Подобная задача возникает, например, в том случае, когда в справочную таблицу, изображенную в табл. 19.3, требуется добавить обязательный элемент.

Полное считывание файла. При решении некоторых задач требуется считывать весь файл. В этом случае для того, чтобы уменьшить время обработки и избежать ошибок¹⁾, которые могут возникнуть в результате многократного появления одних и тех же данных, вновь предпочтительнее использовать плотный файл, не содержащий избыточных данных. Если этого достичь нельзя, то в процессе полного считывания необходимо использовать дополнительную информацию, содержащуюся либо внутри этого файла, либо в отдельных таблицах. Например, не существует простого пути использования файла, изображенного в табл. 19.2 (в), для подсчета числа студентов или вычисления средних сумм, накопленных ими.

Реорганизация. Наконец, может возникнуть необходимость в периодической чистке файлов, особенно тех, которые подвер-

¹⁾ Ошибки могут возникнуть, если не учитывать ассоциативных связей между элементами данных (полями) в записях и между записями (например, табл. 19.2 (в)). — *Прим. ред.*

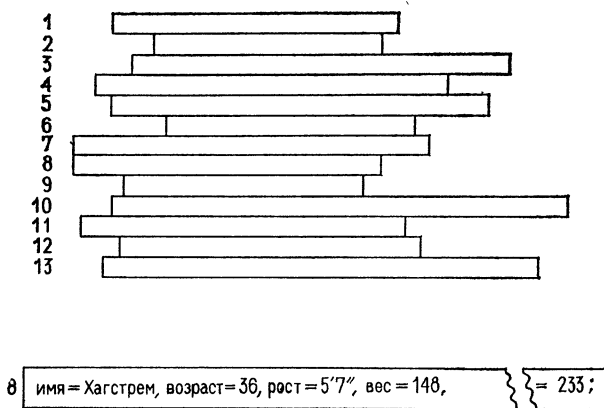


Рис. 19.9. Файл-множество с типичной записью.

гаются постоянным динамическим изменениям. Частота выполнения этой операции в большей степени зависит от используемого типа организации файла, чем от решаемой задачи. Эта операция, называемая реорганизацией файла, очень похожа на процесс “сбора мусора”, выполняемый в некоторых вычислительных системах с динамическим распределением памяти.

В дальнейшем под словом **запрос** мы будем понимать как выборку, так и получение следующей записи, а под словом **доступ** — любое обращение к файлу. Слово **“поиск”** иногда используется для описания некоторой работы с файлом, предшествующей чтению или записи блока.

19.11. ФАЙЛ-МНОЖЕСТВО

Первый из предлагаемых здесь способов организации данных является простейшим способом. Данные в файле-множестве (рис. 19.9) собираются в том порядке, в котором они поступают. Данные не анализируются, не распределяются по категориям и не нормализуются. В лучшем случае порядок может быть хронологическим. Записи могут быть переменной длины и могут содержать разнотипные наборы элементов данных.

Структура файла-множества и работа с ним

При обработке данных, проводимой с целью получения некоторой информации, следует учитывать определенные ограничения. Запись должна состоять из связанных элементов данных, значение каждого из которых идентифицируется. Эта идентификация может заключаться в указании имени, кода или

позиции, которые определяют атрибут элемента (например, его высоту). Если в записи существует несколько идентичных доменов, то атрибут определяет также взаимосвязь с объектом или событием, описываемым записью. Например: высота дверного проема, высота башни. При рассмотрении файлов-множеств мы будем описывать атрибут посредством явного указания имени, поскольку такой способ описания больше всего подходит для файлов с неструктурированной формой.

Пример 19.4. Элемент данных

высота = 95

В этом примере значение элемента данных равно 95, а описанием является высота.

Пара, указанная в примере 19.4, называется парой *имя атрибута — значение*.

Запись, которая содержит только одну такую пару, не является значимой. Для того чтобы правильно описать объект, необходимо несколько таких пар. В этом случае, если требуется объединить фактические данные об объекте, можно определить дополнительные пары атрибутов, которые содержат добавочные данные. Для различных вариантов поиска могут использоваться различные наборы атрибутов, так что нет необходимости заранее назначать атрибуты ключа или целевой функции данных.

Пример 19.5. Запись данных

имя = Хувер, тип = Башня, улица = Серра-Роуд, высота = 95;

Ключевые атрибуты записи должны сопоставляться с аргументом поиска, указанным при запросе на выборку. Термины **ключ** и **цель** определяют две части записи. Ключ идентифицирует требуемую запись в файле, а цель определяется как оставшаяся часть записи. Число атрибутов, необходимых для поиска, является функцией от **эффективности разбиения** для атрибутов ключа. Разбиение имеет место в тех случаях, когда мы проверяем набор данных по указанному атрибуту и затем разделяем этот набор на подмножество, содержащее только те записи, которые по-прежнему представляют для нас интерес, и подмножество данных, которые с этого момента можно игнорировать. Эффективность разбиения может быть задана как абсолютное число или коэффициент. Можно представить себе, что после первого разбиения образуется файл с меньшим числом записей. Важно выделить это подмножество записей таким образом, чтобы для сопоставления со вторым указанным атрибутом не нужно было исследовать все исходные данные. Второй поиск

следует оценивать, исходя из коэффициента разбиения, соответствующего записям, отобранным при предыдущем поиске в качестве интересующих нас записей.

Оценка получается достаточно просто, если первая спецификация поиска не зависит от второй. Последовательное применение всех других спецификаций поиска сокращает число возможных вариантов до тех пор, пока не будет получена требуемая запись или набор записей.

В предыдущем примере применение спецификации Башня к файлу, содержащему описания всех объектов в мире (или по крайней мере всех башен), ограничит наш поиск 10^7 башен [предполагается, что на каждые 300 человек существует одна башня, а всего в мире $3(10^9)$ человек]. Имя Хувер может иметь коэффициент разбиения $4(10^{-6})$ (доля всех башен с именем Хувер в мире была получена на основе экстраполяции данных, взятых из телефонного справочника Сан-Франциско, а также в предположении, что половина всех башен в мире называются именами людей), так что с помощью второго атрибута поиска будет получено 40 возможных записей. (Предполагается, что имя Хувер дается башням так же часто, как и другие имена людей.) Указание третьей спецификации (название улицы) должно быть достаточным для того, чтобы определить требуемую башню или установить, что башни, указанной в данном запросе, не существует.

Если в нашей записи не остается ни одной пары атрибутов помимо тех, которые требуются для поиска, то единственной информацией, которая может быть получена, является тот факт, что данный объект существует в нашем файле. Может существовать много типов атрибутов, которые нельзя использовать для разбиения, однако башня может быть использована как элемент целевых данных.

Пример 19.6. Составной атрибут

место = (улица = Серра-Роуд, город = Станфорд, округ = Санта-Клара,
штат = Калифорния);

Для того чтобы упростить схему организации записи, допускается разбиение самого значения атрибута на несколько пар имя атрибута — значение, как показано в примере 19.6.

В оставшейся части этой главы символом a мы будем обозначать общее число типов атрибутов в рассматриваемом файле, а символом a' — среднее число атрибутов, встречающихся в записи. Если запись, приведенная в примере 19.5, является типичной для данного файла, то величина a' для этого файла равна 4. В примере 19.6 величина a' должна быть увеличена на 4. Для файла с организацией файла-множества нет необходимости знать общее число a атрибутов.

Использование файлов-множеств

Файлы-множества используются в тех случаях, когда сбор данных предшествует их обработке или когда данные трудно организовать, а также при исследовании структуры файлов. В данной главе мы будем использовать их в качестве базиса при сравнении эффективностей использования файлов.

Форму файлов-множеств иногда имеют банки данных, предназначенные для военной разведки. Это связано с тем, что дальнейшее использование записи трудно предугадать. Большое количество атрибутов, входящих в файл-множество, не подлежат предварительному разбиению. Многие справочные наборы данных, как, например, медицинские записи, также имеют форму файла-множества. Анализ данных в файлах-множествах может быть очень дорогим, поскольку поиск достаточного для статистической обработки числа записей требует значительных временных затрат.

Эффективность использования файлов-множеств

Организация файлов-множеств характеризуется следующими параметрами эффективности.

Размер записи. При определении плотности данных в файле-множестве следует учитывать два обстоятельства. Во-первых, необходимость хранить имена атрибутов вместе с данными. Это уменьшает плотность. Во-вторых, несуществующие данные вообще не нужно указывать¹⁾. Это увеличивает плотность. В результате в тех случаях, когда собираемые данные разнородны или разрежены (много несуществующих данных), плотность относительно высокая, а когда данные плотные (несуществующих данных мало) и последовательные записи содержат имена одних и тех же атрибутов, плотность относительно низкая. Имена и данные в файле-множестве имеют переменные длины, поэтому для того, чтобы отметить конец одного данного и начало другого, используются некоторые разделительные знаки (в приведенных выше примерах такими знаками были =, и ;). Через A и V будем обозначать среднюю длину (в байтах) тех частей пары имя атрибута — значение, которые указывают имя и значение соответственно. В примере 19.5 $A = 4,25$, $V = 5,5$.

Из этих определений следует, что ожидаемая средняя длина записи равна

$$R = a'(A + V + 2).$$

¹⁾ При других организациях файлов избыточность может возникать из-за требования соблюдать ограничения, накладываемые структурой записи. — *Прим. ред.*

Величины a' , A и V должны быть получены на основе анализа достаточно большого числа записей.

Выборка записей. Для локализации записи в файле-множестве требуется большое количество времени, поскольку возможно, что при поиске элемента данных, который представлен в файле в единственном экземпляре, нужно будет исследовать все записи. Будем предполагать, что вероятности появления элемента в любом месте файла равны и что для его поиска необходимо считать по меньшей мере один блок, а в худшем случае — все блоки (число которых равно b). Тогда ожидаемая средняя величина времени поиска равна сумме всех времен, необходимых для поиска и считывания каждого блока, деленной на число возможных вариантов, т. е.

$$\begin{aligned}\text{Среднее число считываемых блоков} &= \sum_{i=1}^b \frac{i}{b} = \frac{1}{2} (1 + b) \approx \\ &\approx \frac{1}{2} b, \text{ если } b \gg 1.\end{aligned}$$

Время, необходимое для последовательного считывания этого числа блоков, равно ¹⁾

$$T_F = \frac{1}{2} b \frac{B}{t'}.$$

Этот процесс показан на рис. 19.10. Блоки считываются в один буфер в то время, как обрабатывается содержимое предыдущего буфера. С учетом соотношения между размерами блоков и записей время, необходимое для последовательной выборки, можно выразить в более удобной форме:

$$T_F = \frac{1}{2} n \frac{R}{t'}.$$

Здесь уместно использование скорости t' последовательной передачи данных по той причине, что мы считаем файл от начала, в том числе и все межблочные промежутки, перемещаемые через границы цилиндров, до тех пор пока не будет найден блок, содержащий требуемую запись.

Пакетная обработка запросов. Проблема, связанная с уменьшением затрат на поиск, частично может быть решена путем сбора соответствующих запросов в пакет, поскольку пакет из многих запросов может быть обработан за один проход по все-

¹⁾ Автор, очевидно, предполагает, что B — длина части дорожки, на которой умещается один блок (ниже t' определяется как скорость последовательной передачи). — *Прим. ред.*

²⁾ С учетом примечания 1), n — число записей в файле, R — длина части дорожки, на которой умещается одна запись. — *Прим. ред.*

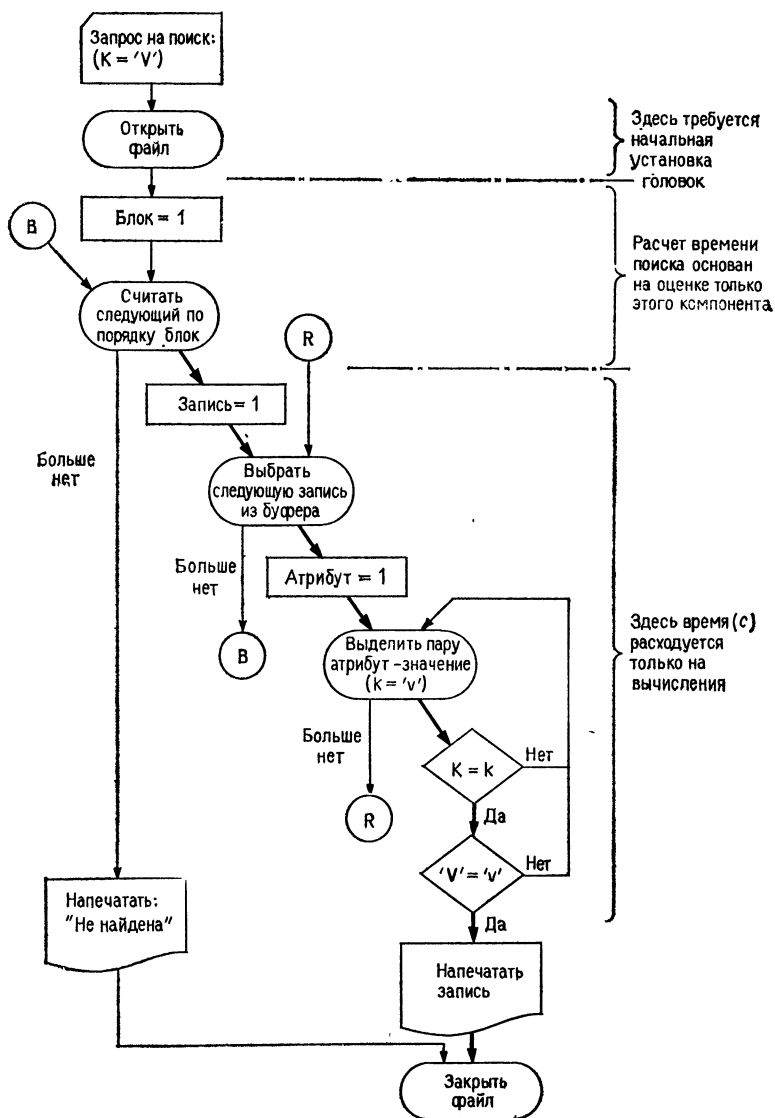


Рис. 19.10. Поиск в файле-множестве.

му файлу. Коэффициент, фигурирующий в выражении для среднего времени поиска, возрастет при этом с $\frac{1}{2}$ до $\frac{3}{4}$ или $\frac{7}{8}$ для двух или трех запросов соответственно. При увеличении числа запросов на выборку этот коэффициент будет стремиться к 1,

так что ожидаемые временные затраты на обработку пакета из L запросов составят

$$T_F(L) = 2T_F, \text{ если } L \gg 1.$$

Хотя затраты на поиск одного элемента сокращаются в $2/L$ раз, время ответа на каждый отдельный запрос по сравнению с первоначальной величиной T_F удваивается. Кроме того, возникает задержка, связанная со сбором в пакет достаточного числа (L) заявок. Обработка таких пакетов часто проводится раз в сутки, с тем, чтобы сделать ее оправданной. Например, время поиска в файле-множестве данных медицинских исследований при пакетировании запросов в действительности превысило 24 часа. В случае когда запросы поступают очень часто, используется одна-единственная программа, способная одновременно обрабатывать несколько запросов. Она периодически просматривает файл. Такая программа выбирает запросы по мере их поступления и сразу же обрабатывает каждый запрос, просматривая h записей файла. В этом случае, как и в предыдущем, затраты на поиск уменьшаются по сравнению со случаем обработки каждой отдельной транзакции, а ожидаемая задержка для отдельного запроса равна T_F . При обработке большого числа запросов, содержащихся в одном пакете, необходимо иметь эффективный алгоритм обработки записей, с тем чтобы выполнялось предположение о том, что $c < R/t$ ¹⁾.

Получение следующей записи из файла-множества. Записи в файле-множестве не упорядочены, поэтому следующая запись может располагаться в любом месте файла. Поскольку место нахождения записи заранее не известно, время, необходимое для поиска произвольной следующей записи, так же как в случае, описанном выше, равно

$$T_N = T_F.$$

Здесь предполагается, что для поиска следующей записи требуется информация, содержащаяся в предыдущей записи. Если требуемые атрибуты поиска для следующей записи были известны в начальный момент, то анализ этой записи проводится в ходе одного комбинированного поиска аналогично тому, как это делается при описанной выше пакетной обработке запросов.

Включение записи в файл-множество. Отсутствие структуры у файла-множества позволяет осуществлять быстрое включение в него новой записи. Адрес конца файла известен и данные

¹⁾ c — время вычислений, расходуемое на операции в процессоре при обработке записи. — *Прим. ред.*

просто добавляются к концу, а указатель конца корректируется. Для того чтобы получить плотноупакованные записи, последний блок считывается в оперативную память. Там к нему присоединяется новая запись и затем блок вновь переписывается в файл. Время операции включения записи в файл составляет ¹⁾

$$T_I = s + r + btt + T_{RW}$$

или

$$T_I = s + 3r + btt.$$

Новая запись не всегда включается в последний блок файла. При простой, без перекрытий, организации файла с записями переменной длины, если значение указателя конца файла, хранящееся в справочнике, говорит о том, что блок уже заполнен и поместить новую запись в него нельзя — он в оперативную память не считывается. В этом случае в файле следует отвести место для нового блока и запись поместить в него. При организации файла с перекрытиями, содержащего записи переменной длины, последний блок считывается в оперативную память, в него частично помещается запись и блок переписывается в файл. Затем в файле отводится место для нового блока и в него помещается оставшаяся часть записи.

Относительная частота появления границы блока, в результате чего потребуется выполнение обеих указанных процедур, равна R/B . С помощью этого коэффициента можно скорректировать приведенную выше формулу для времени включения записи.

В дальнейшем мы не будем принимать во внимание эту поправку и будем предполагать, что чтение и перезапись для расширения файла осуществляются за два обращения. При обработке пакета расширений файла время включения в расчете на одну запись можно уменьшить, если последний блок не переписывать в файл, а оставить в оперативной памяти.

Обновление записи в файле-множестве. Для обновления записи необходимо определить местоположение старой записи, сделать ее недействительной и записать новую, возможно больших размеров, запись в конец файла, так что

$$T_U = T_F + T_{RW} + T_I.$$

Если требуется только стереть запись, то величину T_I следует опустить. На самом деле стирание осуществляется путем замены старой записи нуль-символами ²⁾ или сообщением "Deleted xxx...x".

¹⁾ btt — очевидно, время считывания (записи) блока. — *Прим. ред.*

²⁾ Нуль-символ — признак отсутствия информации. — *Прим. ред.*

Полное считывание файла-множества. Для организации файла-множества затраты на полное считывание только вдвое превосходят затраты на поиск определенной записи:

$$T_X = 2T_F.$$

Однако если требуется просчитать записи файла упорядоченно по некоторому атрибуту, то затраты на повторное выполнение n отдельных выборок составят

$$T_X (\text{упорядоченное}) = nT_F = \frac{1}{2} n^2 \frac{R}{I'}.$$

Эти затраты могут быть уменьшены путем предварительного упорядочения записей в соответствии со значениями данного атрибута. Хорошо известно, что для выполнения процедуры сортировки требуется порядка $n \log_2 n$ шагов. Типичный шаг процедуры сортировки может включать в себя последовательное считывание и последовательную запись одной записи. Поэтому в случае использования процедуры сортировки файла получаем следующую оценку:

$$T_X (\text{упорядоченное}) = T_X + T_{\text{sort}} + T_X = (2n + 2n \log_2 n) \frac{R}{I'}.$$

Для любого нетривиального файла эта величина значительно меньше, чем nT_F .

Реорганизация файла-множества. Если файл-множество обновляется так, как было описано выше, то нужно периодически исключать неиспользуемые области. Это осуществляется путем копирования файла, удаления записей, помеченных как недействительные, и переблокирования оставшихся записей. Если число записей, добавленных в течение некоторого периода, равно o , а число записей, помеченных как удаляемые, равно d , то размер файла возрастет с n до $n + o$, так что время копирования файла составит

$$T_Y = (n + o) \frac{R}{I'} + (n + o - d) \frac{R}{I'}.$$

Число o записей переполнения для различных способов организации файла оценивается не одинаково и зависит от используемого алгоритма. Здесь

$$o = \# (\text{новые записи}) + \# (\text{обновленные записи}),$$

а число записей, которые должны быть удалены при реорганизации, равно

$$d = \# (\text{удаленные записи}) + \# (\text{обновленные записи}).$$

Здесь и в других местах, где оценивается величина T_Y , предполагается также, что процессы чтения и записи при реорганизации не мешают друг другу. Это означает, в частности, что если используются диски с перемещаемыми головками, то ста-

рый и новый файлы нужно хранить на разных устройствах. Данное условие требует наличия достаточного количества буферов, с тем чтобы диски использовались полностью. Поскольку процедуру реорганизации, как правило, планируют проводить во время невысокой загруженности ЭВМ, последнее требование обычно выполняется.

19.12. ПОСЛЕДОВАТЕЛЬНЫЙ ФАЙЛ

Последовательная организация файлов имеет две особенности, отличающие ее от организации файла-множества. Первое усовершенствование организации заключается в том, что записи данных упорядочиваются в определенной последовательности; второе — в том, что атрибуты данных распределены по категориям, так что отдельные записи содержат значения всех атрибутов данных в одном и том же порядке и, возможно, в одной и той же позиции. В этом случае имена атрибутов данных в описании файла должны встречаться только один раз. Вместо того чтобы хранить пары имя атрибута — значение, с каждым именем связывается полный набор (столбец) значений. Такая организация похожа на хорошо известную табличную организацию, которая обычно используется при выводе данных. Как показано на рис. 19.11, записи здесь имеют фиксированную длину.

	Имя	Возраст	Рост	Коэфф. умств. разв.
1	Антверп	55	5' 8"	95
2	Беррингер	39	5' 6"	75
3	Бигли	36	5' 7"	70
4	Бреслоу	25	5' 6"	49
5	Гарсон	61	5' 6"	169
6	Калхоун	27	5' 11"	80
7	Кронер	59	5' 5"	145
8	Майасма	27	5' 2"	75
9	Мак-Клоуд	26	5' 8"	47
10	Мирро	38	5' 8"	52
11	Москвич	23	5' 7"	50
12	Парди	37	5' 9"	48
13	Поп	38	5' 3"	53
14	Протьюз	41	5' 8"	152
15	Розберри	38	5' 7"	70
16	Финнери	42	5' 9"	178
17	Хагстрем	36	5' 7"	83
18	Халгард	31	5' 6"	95
19	Уилер	23	5' 8"	67
20	Янг	18	5' 8"	89

Рис. 19.11. Последовательный файл.

Структура последовательных файлов и работа с ними

Одна или более пар атрибут — значение используется в качестве ключевых атрибутов для записей файла. Часто возникает необходимость в однозначном определении записей по их

ключам. В этом случае записи в файле упорядочиваются в соответствии с ключевыми атрибутами. Один ключевой атрибут выполняет роль первичного ключа сортировки, или ключа высшего порядка, а если этот ключ не однозначно определяет порядок, то до тех пор, пока порядок не будет полностью определен, можно задавать вторичный и дополнительные ключевые атрибуты. Теперь поочередное считывание записей файла в этом порядке может быть выполнено последовательно. В табл. 19.2(а, б) таким атрибутом является номер студента. В табл. 19.2(в) не существует поля, которое могло бы использоваться в качестве ключа, однако ключом здесь может быть пара: обязательный курс и номер студента.

Иногда, для того чтобы получить единственный ключевой атрибут, добавляются искусственные поля, содержащие номера следования, или номера идентификации. В этом случае разбиение файла, описанное при рассмотрении файла-множества, выполняется явно: для каждой записи выбирается уникальный номер идентификации и, следовательно, файл разбивается на n отдельных записей. К сожалению, для определения номера идентификации, соответствующего требуемой записи данных, могут потребоваться дополнительные вычисления.

Наряду с усовершенствованием структуры файла и повышением эффективности таблично ориентированной обработки происходит существенная потеря гибкости, поскольку затрудняется выполнение операций обновления и расширения последовательного файла. Вследствие того что только ключевой атрибут определяет последовательность записей, возникает асимметрия, из-за которой в последовательных файлах затруднен общий поиск информации. При выполнении общей процедуры включения записей в последовательный файл записи собираются в файл-множество, или **файл транзакций**, до тех пор пока последний не станет достаточно большим, а затем осуществляется **пакетное обновление**. Эта процедура выполняется с помощью реорганизации файла. Файл транзакций сортируется в соответствии с теми же ключами, которые используются для основного файла, а затем создается новая копия последовательного файла.

Последовательный файл характеризуется ограниченным и заранее определенным набором атрибутов. Все записи определяются одним-единственным описанием и имеют идентичную структуру. Если к некоторой записи требуется добавить новый атрибут, то необходимо реорганизовать весь файл. При этом каждая запись файла переписывается, с тем чтобы выделить место для нового элемента данных. Поэтому для последовательных файлов иногда с самого начала отводят резервную область (несколько столбцов оставляют пустыми).

При составлении программ обработки удобно использовать

записи фиксированного формата, поскольку программы легче писать в тех случаях, когда однотипная информация находится в одних и тех же полях последовательных записей. Часто записанная запись является простой копией информации, содержащейся в памяти процессора. Даже в тех случаях, когда с помощью языков обработки осуществляется преобразование данных, большая поддержка таким записеориентированным данным обеспечивается спецификациями PICTURE в Коболе и операторами FORMAT в Фортране и ПЛ/1.

Пример 19.7. Описание записи

```
DECLARE 1 payroll_record,  
        2 name,  
        3 initials CHAR(2),  
        3 last_name CHAR(28),  
        2 da'e_born CHAR(6),  
        2 date_hired CHAR(6),  
        2 salary FIXED BINARY,  
        2 exemptions FIXED BINARY,  
        2 sex CHAR(1),  
        2 maternity_leave FIXED BINARY;  
и т. д.  
...  
...  
WRITE (payroll_file) FROM (payroll_record);
```

Запись в файле из примера 19.7 будет иметь формат, в точности соответствующий оператору **DECLARE**, имеющемуся в программе.

Использование последовательных файлов

Последовательные файлы представляют наиболее часто используемый при решении коммерческих задач в пакетном режиме тип файлов. Для того чтобы объединить данные из нескольких последовательных файлов и логически упорядочить их записи, выполняются операции сортировки (рис. 19.12). Тогда все требуемые данные могут быть найдены путем последовательного просмотра заданных файлов. Неудобство заключается в том, что файлы могут быть упорядочены только в соответствии с одним первичным ключом. Поэтому часто возникает необходимость в повторной сортировке такого файла по другому ключу, с тем чтобы объединить другие наборы файлов. В тех случаях, когда данные обрабатываются только периодически, как, например, при ежемесячном составлении накладных, последовательные файлы предпочтительнее других с точки зрения затрат.

ФАЙЛ ДОПОЛНЕНИЙ Студенты, посещавшие лекции ГС		ОСНОВНОЙ ФАЙЛ Другие студенты	
721	...	721	...
843		1019	
1019		1021	
1021		1027	
1029	...	1028	
...		1031	...
		1033	
		1034	...
		...	

Рис. 19.12. Логически последовательные файлы.

Эффективность использования последовательных файлов

В зависимости от выполняемой операции эффективность использования последовательных файлов может быть как очень высокой, так и недопустимо низкой.

Размер записи в последовательном файле. Объем памяти, требуемый для хранения файла с записями фиксированного формата, зависит от всех a возможных атрибутов. (Памятью, необходимой для хранения имен атрибутов, можно пренебречь, поскольку описание атрибутов, которое часто располагается за пределами самого файла, встречается только один раз.) Однако и в тех случаях, когда атрибуты имеют неопределенное значение или являются незначимыми в комбинации с другими атрибутами, будет использоваться память. В примере 19.7 между последними двумя элементами существует зависимость, так как для класса **sex** = 'Male' (пол = 'мужской') элемент **maternity_leave** (декретный отпуск) становится незначимым (NULL). Фиксированный размер записи равен произведению числа полей и их среднего размера, т. е.

$$R = aV.$$

Плотность файла будет низкой в том случае, когда не определено большое число значений атрибутов. Если же значение a' близко к значению a , то плотность файла будет высокой. Если ожидается добавление записей, то для o новых записей длины R должна быть выделена память в соседней области.

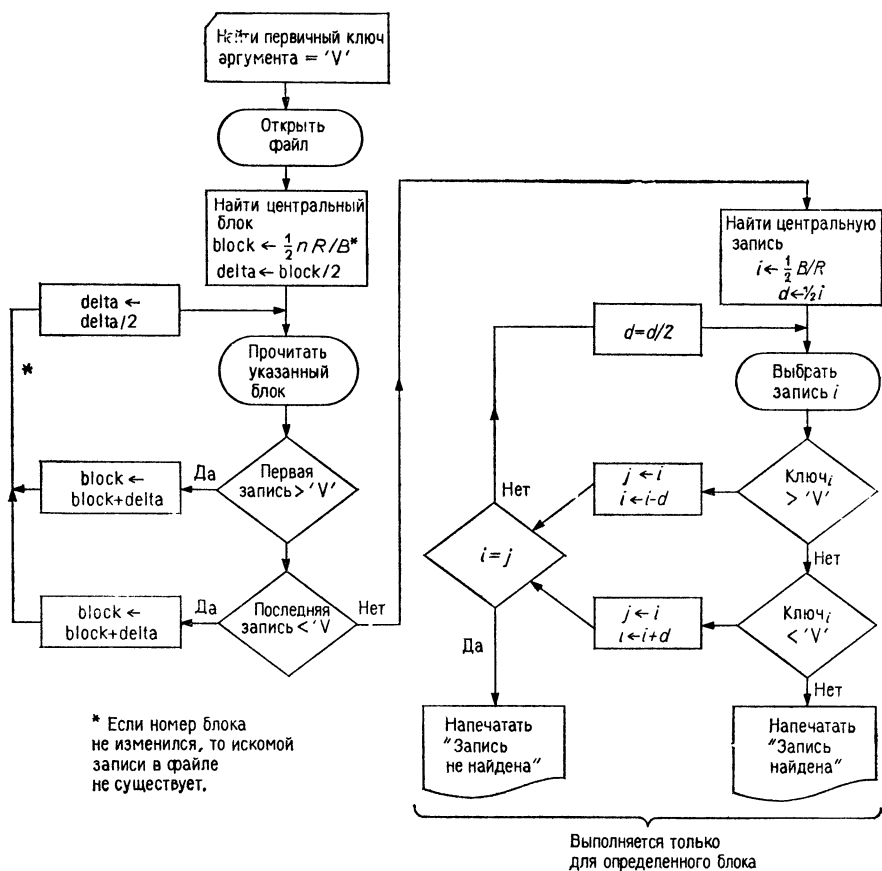


Рис. 19.13. Вложенный двоичный поиск в заблокированном файле.

Выборка записи из последовательного файла. Общий подход к выборке записи из последовательного файла заключается в последовательном переборе. Если файл хранится на устройстве прямого доступа, то с помощью метода двоичного поиска можно значительно сократить время, необходимое для выборки произвольной записи. Такой поиск возможен только для того типа атрибута, по которому файл был упорядочен.

Двоичный поиск (рис. 19.13). Всякий раз, когда осуществляется выборка блока, анализируется его первая и последняя записи, с тем чтобы определить, находится ли в этом блоке иско-

мая запись¹⁾. Поэтому число выборок зависит не от числа записей n , а от числа блоков nR/B . Используя выражение для числа ожидаемых выборок блоков при двоичном поиске, получаем

$$T_F = \log_2 \left(n \frac{R}{B} \right) (s + r + btt + c).$$

Величина c , равная времени обработки, присутствует здесь по той причине, что до тех пор, пока не выполнена проверка записи, не известно, какой блок должен быть считан следующим. Получение оценок эффективности для последовательного считывания файла с использованием дополнительных буферов в этом случае мы опускаем. По сравнению с другими рассматриваемыми временами значение c мало, и поэтому им вполне можно было бы пренебречь, но к скорости t' передачи массивов это замечание неприменимо.

В тех случаях, когда аргумент поиска не является ключевым атрибутом, используемым для упорядочения файла, поиск становится таким же, как и в файле-множестве. Поскольку из-за различий в организации записей общий размер последовательного файла отличается от размера файла-множества, то относительная эффективность использования зависит от плотности атрибутов a'/a , а также от относительной длины дескрипторов A и значений данных V . Для выборки одной записи в среднем потребуется провести поиск по половине файла, так что

$$T_F = \frac{1}{2} n \frac{R}{t'}.$$

Для небольших файлов данная величина сравнима с временем, затрачиваемым на двоичный поиск.

Если при добавлении к файлу o' новых записей они были включены в файл транзакций (или файл переполнений), то в нем также необходимо провести поиск. Записи в файле транзакций будут располагаться в порядке их поступления, так что дополнительное слагаемое, соответствующее времени последовательного поиска, всегда равно

$$T_{FO} = \frac{1}{2} o' \frac{R}{t'}.$$

Если в обеих частях файла проводится последовательный поиск, то общее время выборки составляет

$$T_F = \frac{1}{2} (n + o') \frac{R}{t'}.$$

¹⁾ Подробное описание метода двоичного поиска можно найти, например, в книге: Кнут Д. Искусство программирования для ЭВМ, т. 1, — М.: Мир, 1972. — Прим. ред.

или, предполагая, что файл транзакций в среднем заполнен наполовину,

$$T_F = \frac{1}{2} \left(n + \frac{1}{2} o \right) \frac{R}{t'},$$

где o — вместимость файла транзакций.

Если выборка отдельных записей производится часто, то последовательный поиск в файле транзакций может стать чрезмерно дорогим. В этом случае записи в файле транзакций следует хранить в такой последовательности, которая позволяет осуществлять быстрый поиск, а добавление записей нужно осуществлять с использованием таких методов, как сцепление и прямой доступ, которые будут рассмотрены в последующих разделах.

Зондирование. Третий способ поиска, зондирование, является более сложным в плане получения количественных оценок. Он применяется только в тех случаях, когда аргумент поиска является ключом упорядочения для файла. Этот способ заключается в следующем. Вначале выполняется операция прямой выборки (зондирование), исходя из некоторой оценки позиции записи в файле, а затем — последовательный поиск. Если эффективно может выполняться только последовательный поиск в прямом направлении, то начальное зондирование осуществляется, исходя из оценки нижней границы значения ключа записей в блоке. Выбор наиболее вероятных начальных значений для зондирования может быть основан, например, на анализе ведущих цифр номера страхового полиса, если он используется в качестве ключа, или на оценке вероятности появления ведущих знаков в именах, если последние используются в качестве ключей. Например, имена, начинающиеся с букв "Sc", могут быть найдены после просмотра 0,79*n* записей файла, упорядоченного по именам. Более совершенные методы доступа к записям, методы прямого доступа, применяются для описываемых ниже организаций файлов.

Как видно, в рассматриваемом случае существуют три метода доступа, применимых к одной организации файлов. Таким образом, методы программирования не однозначно определяются организацией данных.

Получение следующей записи из последовательного файла. В последовательном файле следующей записью является та, которая достигается непосредственно из предыдущей, и вполне возможно, что она располагается в том же самом блоке, в котором находится предыдущая запись. Если необходимость в получении следующей записи возникает часто, то файловая система должна обеспечить сохранение буфера с содержимым те-

кущего блока, с тем чтобы не уничтожались оставшиеся записи блока. Вероятность того, что следующая запись располагается в одном блоке с предыдущей, равна $(B/R - 1)/(B/R)$, так что ожидаемое время получения следующей записи составляет

$$T_N = \left(1 - \frac{B/R - 1}{B/R}\right) \frac{B}{t'} = \frac{R}{t'}.$$

Если в файловой системе не предусмотрено выполнение операции получения следующей записи и содержимое текущего блока не сохраняется, то

$$T_N = r + btt,$$

так как в этом случае буфер необходимо заполнять заново.

Включение записи в последовательный файл. Добавление записи к основному файлу обычно осуществить невозможно, поскольку при включении новых записей в конец файла нарушается упорядоченность. Для небольших наборов данных можно сдвинуть записи, расположенные за точкой включения, с тем чтобы освободить пространство для новой записи. Для этого потребуется прочитать и переписать в среднем половину всех блоков файла, так что

$$T_I = \frac{1}{2} n \frac{R}{B} \left(\frac{B}{t'} + T_{RW} \right).$$

На практике это редко осуществимо. Обычно новые записи собирают в файл транзакций и затем, спустя некоторое время, выполняя пакетное обновление. Символом o мы будем обозначать число записей, собранных для отсроченного включения.

Следовательно, фактические затраты на расширение файла являются функцией от времени, необходимого для того, чтобы записать записи в файл транзакций, и затрат на реорганизацию. Затраты T_Y на реорганизацию распределяются по o записям, которые были собраны в файл транзакций между периодами реорганизации. Для простого несблокированного файла транзакций время обновления и реорганизации в расчете на одну запись составляет

$$T_I = s + r + \frac{R}{t} + \frac{T_Y}{o}.$$

Время реакции системы при вводе записи для обновления, включает в себя только три первых слагаемых¹⁾. Время реорганизации T_Y и число o записей в файле транзакций определяются ниже. Для заблокированного файла транзакций следует

¹⁾ Реорганизация, как было отмечено выше, производится не во время работы пользователя; затраты времени на подготовку файла транзакций не включаются во время ответа. — *Прим. ред.*

учесть время считывания и перезаписи последнего блока, так что

$$T_I = s + r + btt + T_{RW} + \frac{T_Y}{o}.$$

Если записи заблокированы и обновления производятся часто, то последний блок файла транзакций может храниться в буфере обновления, так что необходимость поиска и чтения блока возникает реже. В этом случае

$$T_I = \frac{R}{B} (s + r + btt) + \frac{T_Y}{o}.$$

Если поиск должен выполняться по таким необработанным файлам¹⁾, то необходимо просматривать как последовательный файл, так и файл транзакций.

Обновление записи в последовательном файле. В силу определения последовательного файла размер включаемой записи не должен превосходить размер исходной хранимой записи. Если значение ключа включаемой записи то же, что и обновляемой, то она может быть записана в основной файл вместо старой записи. В противном случае процесс обновления аналогичен процессу добавления записи к файлу, но одновременно включает в себя удаление записи из последовательного файла. Вместо фактического изменения записи в основном файле в файл транзакций может быть добавлена специальная запись, указывающая необходимость этого действия. Эта транзакция будет использоваться для выборки модифицируемой записи, а также в процессе реорганизации. Такому типу действий соответствуют две записи в файле транзакций. Если d записей удаляется, а v записей обновляется, то к величине o , размеру файла транзакций, должна быть добавлена величина $d + 2v$. Тогда $T_U \approx T_I$. Мы не будем останавливаться на получении других оценок эффективности выполнения операции обновления, поскольку при этой организации файла сложное обновление выполняется редко.

Полное считывание последовательного файла. Полное считывание файла заключается в последовательном считывании основного файла и файла транзакций. Это означает, что данные будут считываться упорядоченно в соответствии с ключом, используемым для задания физического упорядочения записей файла. Для создания этого порядка вначале следует рассорти-

¹⁾ Еще не завершена обработка файла транзакций на включение, а уже производится поиск записей. — *Прим. ред.*

ровать файл транзакций. Тогда

$$T_x = \text{sort}(o) + n \frac{R}{t} + o \left(s + r + \frac{R}{t} \right),$$

если записи переполнений не заблокированы, или

$$T_x = \text{sort}(o) + (n + o) \frac{R}{t},$$

если файл транзакций заблокированный и используются два буфера, позволяющих обрабатывать оба файла как массивы.

Если очередность при обработке записей не важна, то файл транзакций можно не сортировать. Очевидно, что время, необходимое на последовательное считывание записей, значительно меньше времени, необходимого на считывание записей в любом другом порядке, а именно

$$T_x(\text{посл.}) \approx 2T_s,$$

в то время как

$$T_x(\text{не посл.}) = nT_s.$$

Если файл транзакций относительно большой ($o \gg n$), то лучше всего, по-видимому, вначале реорганизовать этот файл. Записи можно анализировать во время реорганизации, так что

$$T_x = T_y.$$

Из замечаний, сделанных в предыдущих параграфах, следует, что

$$o = (n_{\text{нов}} - n_{\text{стар}}) + 2v + d,$$

или, другими словами, величина o , используемая для подсчета транзакций, здесь равна сумме числа новых записей, удвоенного числа изменяемых записей и числа удаляемых записей.

Реорганизация последовательного файла. Реорганизация последовательного файла заключается в создании нового файла путем объединения старого файла и файла транзакций. Для эффективного выполнения операции объединения следует вначале рассортировать файл транзакций в соответствии с ключом, который используется в старом файле. В процессе объединения рассортированные данные из файла транзакций и записи из старого последовательного файла копируются в новый файл; при этом исключаются записи из файла транзакций, которые помечены как удаляемые. Время прогона реорганизации складывается из времени сортировки файла транзакций и времени объединения. Таким образом, время, необходимое для реорганизации файла, равно сумме времени считывания обоих фай-

лов, времени записи нового файла и времени сортировки файла транзакций:

$$T_Y = n_{\text{стар}} \left(\frac{R}{I'} \right) + \text{sort}(o) + o \left(\frac{R}{I'} \right) + n_{\text{нов}} \left(\frac{R}{I'} \right),$$

или, если числом удаляемых записей $d + v$ можно пренебречь,

$$T_Y = 2(n + o) \frac{R}{I'} + \text{sort}(o).$$

Оценка времени сортировки файла была дана при рассмотрении полного считывания файла-множества.

19.13. ИНДЕКСНО-ПОСЛЕДОВАТЕЛЬНЫЙ ФАЙЛ

Появление индексно-последовательных файлов объясняется стремлением преодолеть недостатки последовательной организации файла, проявляющиеся при доступе к данным, не теряя при этом всех преимуществ этой организации. Одна отличительная особенность индексно-последовательных файлов заключается в наличии индекса, позволяющего осуществлять менее упорядоченный доступ к записям; другая особенность заключается в наличии средств обработки дополнений к файлу. На рис. 19.14 дан пример индексно-последовательного файла. Здесь показаны три важных компонента: последовательный файл, индекс и область переполнения. Показаны также и другие детали, которые будут рассмотрены ниже.

Структура индексно-последовательных файлов и работа с ними

При упорядоченном считывании данных индексно-последовательная организация позволяет осуществлять доступ к записям файла в последовательности их размещения. Добавленные записи находятся в отдельном файле, аналогичном файлу транзакций для последовательных файлов. Однако их местоположение определяется с помощью указателя из предшествующей им записи, находящейся в последовательном файле. Упорядоченное считывание записей комбинированного файла выполняется последовательно до тех пор, пока не встретится указатель на файл переполнения; затем считывается файл переполнения до тех пор, пока не встретится пустой (**NULL**) указатель; после этого возобновляется считывание последовательного файла. Индекс позволяет обращаться к записям в произвольном порядке.

Индекс. Индекс состоит из набора элементов каждый из которых содержит значение ключевого атрибута, соответствующее некоторой записи данных, и указатель, позволяющий осуществ-

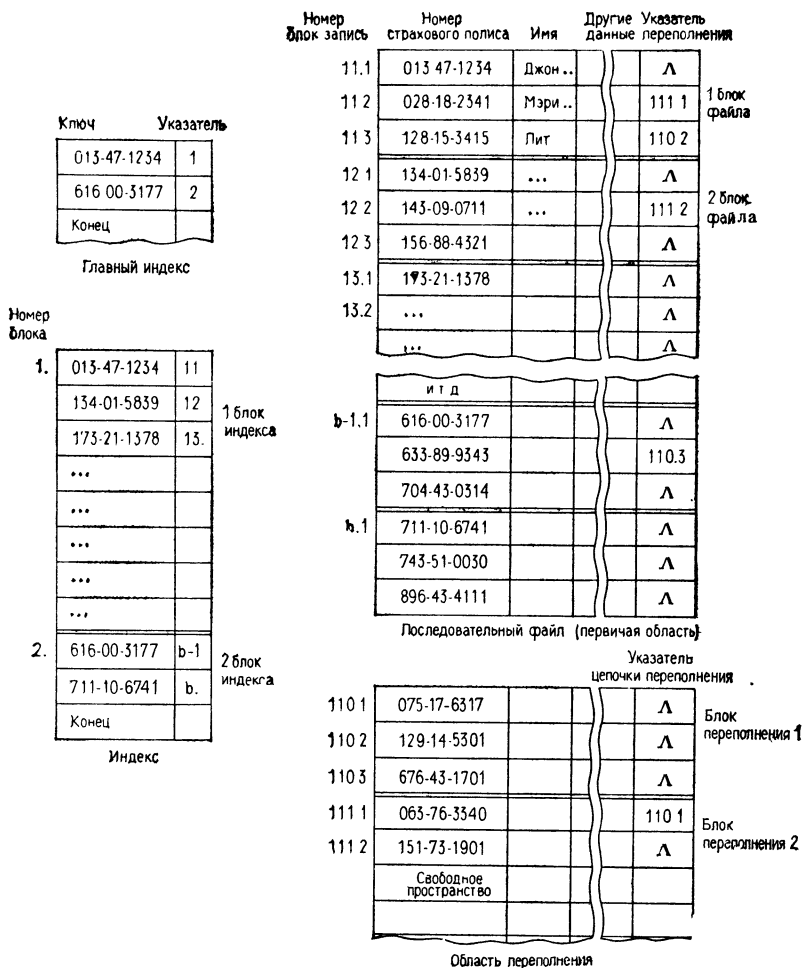


Рис. 19.14. Индексно-последовательный файл.

лять непосредственный доступ к этой записи. Для записей больших размеров элемент индекса занимает значительно меньше места, чем сама запись данных. Следовательно, весь индекс будет меньше, чем сам файл, так что поиск нужно будет проводить по меньшей области. Для того чтобы осуществлять быстрый поиск в самом индексе, его элементы упорядочивают в соответствии с заданным атрибутом, даже если файл упорядочен другим образом.

Пример 19.8. Индекс. Имеется файл сведений о служащих, упорядоченный по номеру страхового полиса.

Запись	Номер страхового полиса	Имя	Дата	Пол	Профессия
1	013—47—1234	Джон	1/1/43	Муж.	Сварщик
2	028—18—2341	Пит	11/5/45	Муж.	Рыболов
3	128—15—3412	Мэри	6/31/39	Жен.	Инженер

Для того чтобы находить служащих по именам, расположенным в алфавитном порядке, индексный файл нужно организовать следующим образом:

Джон	1
Мэри	3
Пит	2

Поиск в большом индексе упрощается путем индексирования подмножеств индекса. Например, такими подмножествами могут быть 26 групп служащих¹⁾, в каждой из которых имена начинаются с одной и той же буквы. Часто размер подмножества определяется размером буферов, блоков и дорожек, имеющих в файловой системе. С ростом уровня индекса размер индекса уменьшается. Индекс высшего уровня имеет небольшие размеры и может храниться в оперативной памяти. Позже будет показано, что необходимость иметь много уровней индексации возникает редко. Альтернативой многоуровневому индексированию является использование метода двоичного поиска по индексному файлу. Двоичный поиск был описан в предыдущем разделе при получении оценок для последовательных файлов. Для файла, изображенного на рис. 19.14, разбиение индекса на подмножества проводилось по числу (8) элементов индекса в блоке.

Первичный индекс. Индекс для индексно-последовательного файла обычно организуется с помощью того же самого ключевого атрибута, который использовался для упорядочения записей самого файла. Для такого первичного индекса можно ввести ряд усовершенствований.

Анкерные точки блоков. Отдельные записи в блоке могут быть найдены с помощью последовательного поиска в этом блоке. Поэтому нет необходимости для каждой записи хранить

¹⁾ Число групп соответствует числу букв в английском алфавите. — *Прим. ред.*

соответствующий ей элемент индекса, а достаточно лишь указать одну запись в каждом блоке. Эта справочная запись называется **анкерной точкой**, и в индексе хранится только значение ключа анкерной точки и указатель блока. Обычно анкерные точки выбираются для блоков, дорожек или цилиндров. На рис. 19.14 в качестве анкерной точки выбирается первая запись блока. Иногда более предпочтительными могут оказаться другие варианты. Затраты на поиск записи в блоке незначительны, поскольку всякий раз, когда требуется, в память считывается весь блок, который может храниться в буфере. Число записей в блоке равно B/R . Следовательно, число элементов в индексе равно $n(R/B)$. Размер элемента индекса равен $V + P$.

Если в индексе хранятся только анкерные точки блоков, то одной лишь проверкой индекса нельзя определить, существует ли запись, соответствующая заданному аргументу. Необходимо также прочитать соответствующий блок данных. Для того чтобы определить, выходит ли значение аргумента поиска за значение последнего элемента файла, необходимо выбрать последний блок данных, так как анкерная точка указывает на первую запись в этом блоке. Если необходимость включать записи в конец файла возникает часто, удобнее хранить в индексе значение ключа последней (а не первой) записи каждого блока. Тогда для заданного аргумента соответствующий блок выбирается путем поиска минимального элемента индекса, не меньше, чем значение аргумента.

Индексы цилиндров. Значительная часть накладных расходов на выборку блока приходится на достижение нужного цилиндра. Однако затраты на установку можно сократить, располагая подмножества индекса в соответствии с границами, определяемыми аппаратными средствами. В этом случае появится главный индекс, который содержит только значения ключевого атрибута и адреса анкерных точек цилиндров. На начальной дорожке каждого цилиндра будет находиться индекс цилиндра, указывающий дорожки, блоки или записи в качестве анкерных точек для данного цилиндра. Тогда задержки на установку между индексом цилиндра и записями данных не возникает.

Коэффициент расширения индекса. Для оценки требуемого числа уровней индексации рассмотрим пример сравнительно большого файла (один миллион записей) и индекса скрепленного с блоками данных. Для того чтобы оценить время доступа при выборке записи, следует определить требуемое число уровней индекса для файла такого размера.

Пример 19.9. Организация индекса. Заданы размер блока $B = 2000$ байт, размер значения $V = 14$ байт, размер указателя $P = 6$ байт и длина записей данных $R = 200$ байт. С данным коэффициентом блокирования ($B/R = 10$) для 10^6 записей потребуется 10^5 блоков и, следовательно, столько же указателей индекса. Отношение размера блока B к размеру элемента индекса определяет информационную вместимость одного блока индексной памяти, или коэффициент расширения y индекса каждого уровня. Каждый блок индекса содержит

$$y = \frac{B}{V + P}$$

элементов индекса. Размер элемента индекса в нашем примере равен $(14 + 6) = 20$ байт, а размер блока B по-прежнему равен 2000. Следовательно, $y = 100$, так что 10^5 элементов индекса нижнего уровня занимают 10^3 блоков, для указания которых требуется 10^3 элементов индекса второго уровня. Индекс второго уровня будет занимать память объемом $20 \cdot (1000) = 20\,000$ байт, который является слишком большим для оперативной памяти, так что необходимо определить третий уровень индекса. На высшем уровне потребуется всего $20\,000/2000 = 10$ элементов, которые будут занимать 200 байт. Термин **главный индекс** используется для указания самого верхнего уровня. Уровни индекса нумеруются от 1 для ближайшего к данным уровня до x (здесь 3) для главного уровня.

Можно заметить, что блоки индексов содержат большое число элементов, т. е. имеют большой коэффициент расширения, так что число уровней невелико. В рассмотренном выше примере в ходе обработки два уровня индекса будут считываться с диска, а главный индекс остается доступным в оперативной памяти. Поэтому для выборки записи данных потребуется прочитать три блока.

Записеориентированный индекс для того же файла был бы в $B/R = 10$ раз больше, однако число уровней в нем осталось бы прежним (это видно из рассмотрения примера 19.9 для 10^6 элементов индекса). В следующем примере рассмотрен случай, когда индекс, скрепленный с блоками данных, для того же файла хранится на том же цилиндре, где располагаются данные.

Пример 19.10. Организация индекса с учетом особенностей аппаратных средств. При использовании диска, емкость цилиндров которого составляет 266 000 байт, в каждом цилиндре будут располагаться $266\,000/2000 = 133$ блока. Элементы индекса, которые соответствуют последовательным аппаратным секциям, могут не содержать поля указателя, так как элемент 1 просто соответствует блоку 1 и т. д. Для индекса потребуется

$133(14) = 1862$ байт, или один блок на каждый цилиндр, а 132 блока остаются для данных. Помимо индекса для данных используется также индекс более высокого уровня. Файл занимает $10^6(200)/(132(2000)) = 758$ цилиндров. Поскольку индекс цилиндров содержит по одному элементу на каждый из последовательно расположенных цилиндров, то для него также не требуется поля указателя. Для индекса цилиндров требуется $758(14) = 10\,612$ байт, или 6 блоков, каждый из которых содержит по 142 элемента. На практике при использовании индексов первого уровня, размещаемых в цилиндре, индекс цилиндров также располагается в пакете дисков. Если пакеты дисков содержат 200 цилиндров, то для каждого индекса цилиндров потребуется 2800 байт (два блока) и главный индекс будет содержать по одному элементу на каждый пакет дисков, или четыре элемента для данного файла. Если для индекса размеры блоков увеличить нельзя, то для доступа к индексу цилиндров здесь потребуется считывание двух блоков. Имея два буфера, можно уменьшить затраты на величину, равную времени одного оборота диска.

Из этого примера видно, что структура индекса, ориентированного на аппаратные средства, имеет большие коэффициенты расширения (132, 142) и, следовательно, небольшой главный индекс. Фактические их значения будут зависеть от размера блока и размера файла. Можно заметить, что деревья, описывающие такие организации индекса, направлены больше вширь, чем ввысь. Структура индекса, ориентированного на средства программирования, является более гибкой с точки зрения приспособления ее к определенным требованиям, предъявляемым к файлам.

Ширина дерева определяет коэффициент расширения. На рис. 19.15 даны символические примеры, иллюстрирующие низкий и высокий коэффициенты расширения. Деревья изображаются не в традиционном виде (перевернутыми вверх корнем), поэтому процесс выборки листа начинается с нижней части, или корня, дерева. Вопрос, связанный с оценкой коэффициента расширения, является очень важным для анализа индексированных файлов, и мы будем часто к нему возвращаться.

Переполнение. Для того чтобы осуществлять включение записей в файл, необходимо зарезервировать некоторую свободную область. Добавляемые записи можно размещать в отдельном файле или зарезервировать для них область в каждом блоке или в каждом цилиндре. Если для добавления записей используется отдельный файл, то при обращении к каждой включенной записи необходим отдельный доступ, что приводит к дополнительным расходам, связанным с установкой головок

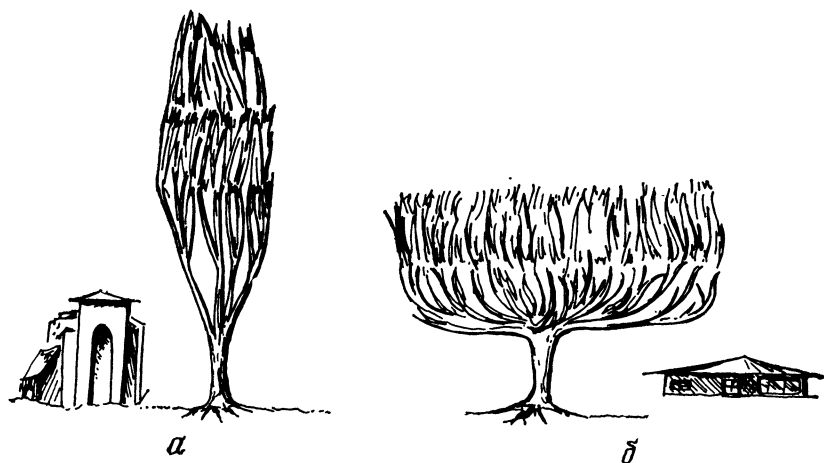


Рис. 19.15. Итальянский и монтерейский кипарисы: а — небольшое расширение; б — большое расширение.

и ожиданием нужного блока записей. Выделение области в каждом блоке возможно только в тех случаях, когда блоки имеют большие размеры и включения достаточно равномерно распределены; в противном случае эта область может быстро заполниться.

Разумного компромисса можно достичь, если резервную область оставлять в каждом цилиндре. В этом случае при локализации записи переполнения возникает задержка на время оборота диска, а установки головок не требуется. Адрес цилиндра определяется с помощью значения ключа добавляемой записи. Для этого ищется элемент индекса, соответствующий ближайшему предшественнику добавляемой записи. Новая запись помещается в следующую по порядку свободную позицию в области переполнения цилиндра.

Размеры областей переполнения цилиндров необходимо выбирать с особой тщательностью. Если в определенные области записи включаются чаще, чем в другие, то для соответствующих цилиндров потребуются большие области переполнения. Если система распределяет память таким образом, что все области переполнения цилиндров имеют равные размеры (как это чаще всего делается), то в цилиндрах, в которые включается мало записей, может остаться много незанятого пространства. В качестве дополнительной можно зарезервировать вторичную область переполнения, которая используется в тех случаях, когда область переполнения какого-либо цилиндра сама переполняется. В этом случае при обработке записей, расположенных

во вторичной области переполнения, не удастся избежать затрат, связанных с установкой головок, в частности упорядоченный доступ осуществляется очень медленно. Существует два способа организации доступа к первичной области переполнения.

Косвенный доступ к области переполнения с помощью последовательного файла. Указатели на включенные записи обычно размещаются с предшествующими записями в первичных блоках данных (см. рис. 19.14). Ключ включенной записи здесь не хранится; в последовательный файл помещается только указатель, так что поиск любой промежуточной записи переносится на область переполнения. При включении записей с помощью данной процедуры не требуется модифицировать индекс, однако для каждой выборки включенной записи дополнительно считывается один блок. Запрос на несуществующую запись потребует перехода к файлу переполнения, если аргумент поиска следует за ключом первичной записи с указателем на область переполнения.

Непосредственный доступ к области переполнения с помощью индекса. Для того чтобы показать существование включенной записи, используется указатель. Если каждой записи файла соответствует элемент индекса, то в индекс может быть помещен также указатель переполнения. При обработке последовательной части файла эти указатели в элементах индекса используются для доступа к включенным записям. В этом случае при поиске определенной записи решение о переходе к области переполнения может быть принято на основе информации, содержащейся в индексе, так что выборки первичного блока не требуется. Соответствующий индекс показан на рис. 19.16; первичный файл не содержит полей, указывающих на область переполнения. В тех случаях, когда файл обновляется, необходимо ведение расширяющегося индекса. Для предложенной выше схемы ведение упрощается путем помещения указателя переполнения с каждым элементом записи, т. е. путем удвоения размера индекса. Однако с ростом размера индекса возрастает также время его обработки. Для индексов, присоединенных к записям данных, увеличение размеров будет очень значительным. Этот метод представляет интерес главным образом в тех случаях, когда область переполнения и соответствующие первичные записи расположены на различных цилиндрах так что экономия времени на выборку становится значительной.

Сцепление записей переполнения. Для того чтобы определить местоположение нескольких записей переполнения из одной первичной записи, указатели помещаются также в записи,

		Первичные элементы		Элементы переполнения		
Главный индекс	013-47-1234	1	013-04-1234	11	075-17-6317	110.3
	307-10-4837	2	134-01-5839	12	156-88-4321	111.2
	616-00-3177	3	173-21-1378	13	—	Λ
	Конец	

Индекс записей данных	616-00-3177	b-1	704-43-0314		111.1	
	711-10-6714	b	—		Λ	
	Конец					

Рис. 19.16. Индекс для непосредственного доступа к области переполнения.

находящиеся в областях переполнения. Все такие записи связываются в **цепочку**, возможно, при помощи нескольких блоков области переполнения. К этой цепочке может быть присоединена новая запись, так что последовательный порядок сохраняется.

В тех случаях, когда выборка осуществляется среди большого числа включенных записей, в большом числе блоков, следование по цепочке к определенной записи в действительности может оказаться менее эффективным, чем обычный исчерпывающий поиск по всей области переполнения. С другой стороны, возможность просмотра цепочки существенно упрощает упорядоченную обработку. Для того чтобы при обработке данных не потерять данные из буфера последовательного файла, необходимо иметь отдельный буфер переполнения. Для того чтобы избежать этих затрат, в методе IBM ISAM используются блоки, размер которых определяется с учетом размера записей в области переполнения.

Проталкивание. Другой метод размещения записей переполнения основан на сохранении последовательности ключей в блоках первичного файла. Новые записи включаются после своих непосредственных предшественников; последующие записи проталкиваются по направлению к концу блока. Записи из конца первичного блока проталкиваются в область переполнения. На рис. 19.17 описан процесс **проталкивания**, использующий та-

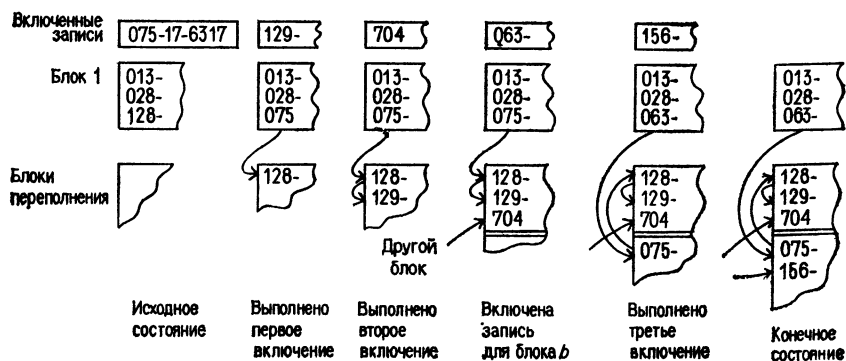


Рис. 19.17. Последовательное включение записей в блок индексно-последовательного файла с помощью проталкивания.

кую же последовательность включения записей, какая приведена на рис. 19.14. Конечное состояние файла показано на рис. 19.18. Индекс такой же, как и прежде, и изображён символически. Теперь в первичном блоке необходим только один указатель на область переполнения, и из каждого блока осуществляется только один переход к файлу переполнения. Тем не менее файл переполнения обрабатывается так же, как было показано выше. Однако цепочки будут длиннее. Выборка записи, помещенной в область переполнения, в среднем будет осуществляться дольше. Если время доступа из области последовательного файла в область файла переполнения больше, чем время перемещения между блоками переполнения (т. е. указанные две области расположены на различных цилиндрах), то упорядоченная обработка записей будет осуществляться быстрее с использованием метода проталкивания.

Реорганизация. Необходимость в реорганизации файла возникает в тот момент (или до того момента), когда переполняются сами области переполнения. Реорганизация может потребоваться и в тех случаях, когда из-за образования длинных цепочек время, необходимое на выборку или упорядоченную обработку записей, становится недопустимо большим. В процессе реорганизации файл считывается так же, как при выполнении упорядоченной обработки, и записывается заново. При этом исключаются все записи, которые помечены как удаляемые, а все новые и оставшиеся старые записи последовательно записываются в основную область нового файла. В процессе этой обработки программами реорганизации организуются новые индексы, в которых используются новые анкерные точки.

Частота такой реорганизации зависит от интенсивности включения записей в файл. На практике реорганизация прово-

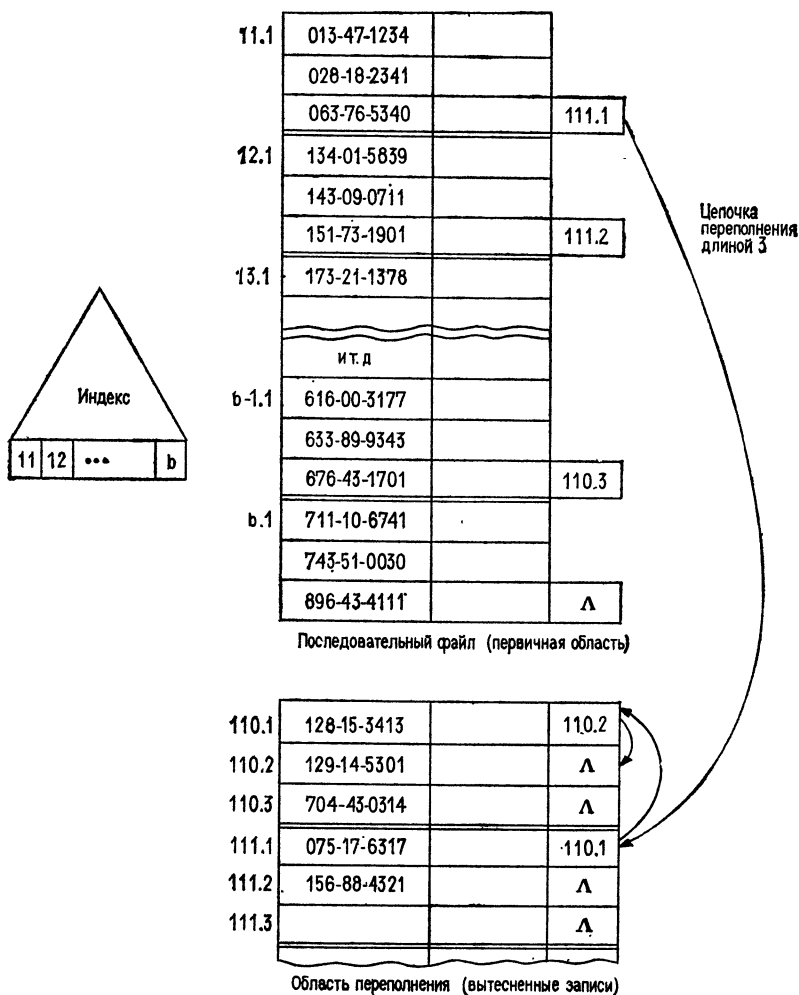


Рис. 19.18. Область переполнения индексно-последовательного файла, заполненная путем проталкивания записей.

дится от одного раза в день до одного раза в год. Поскольку для реорганизации может потребоваться большое количество времени, она обычно выполняется до того, как файл полностью заполняется, с тем чтобы избежать неприятных неожиданностей во время работы с файлом. Реорганизация может выполняться либо периодически с заранее вычисленным периодом, либо в удобный момент после того, как число элементов в области переполнения превысит определенный предел.

Использование индексно-последовательных файлов

Индексно-последовательные файлы определенного типа, рассмотренного выше, широко используются при обработке экономической информации. Особенно часто они используются в тех случаях, когда временные интервалы, в течение которых необходимо сохранить самую новую копию файла, меньше, чем интервалы обработки, допустимые при периодической реорганизации последовательных файлов¹⁾. Например, для составления списка товаров индексно-последовательный файл может использоваться ежедневно, а его реорганизация (совместно с процессом, формирующим заказы на товары, запасы которых не достаточны) проводится еженедельно.

Индексно-последовательные файлы также широко используются для обработки информационных запросов, но при этом требуется, чтобы в запросе был определен ключевой атрибут. Это требование подчас приводит к тому, что в отдельных индексно-последовательных файлах хранятся копии одних и тех же данных, но упорядоченных по различным ключам. В этом случае возрастают затраты на обновление и увеличивается необходимый объем памяти.

Пользователи часто не понимают назначения того или иного варианта организации данных. Вследствие этого использование, например, индексно-последовательных файлов в некоторых системах ведет к неоправданному увеличению времени обработки данных. Случаи, когда новые данные поступают в файл группами, тоже могут привести к большим затратам времени при их просмотре, поскольку получающиеся цепочки могут оказаться очень длинными. В действительности группы новых данных часто присоединяются к концу файла. Если их обрабатывать отдельно или если заранее, до реорганизации индексно-последовательного файла, выделять в нем дополнительную область и резервировать значения индекса, то можно избежать ряда проблем, возникающих при включении записей. В пределах конкретного метода обработки индексно-последовательного файла возможности для модификации режимов невелики, однако авторы программного обеспечения и изготовители ЭВМ все же предоставляют ряд дополнительных вариантов обработки. Но ограничение, согласно которому только один ключевой атрибут определяет основную последовательность записей файла, является общим для всех последовательных файлов.

¹⁾ Интервал обработки определяет фактическое «время жизни» конкретной структуры некоторого файла, после этого проводится реорганизация, и файл приобретает другую структуру. — *Прим. ред.*

Эффективность использования индексно-последовательных файлов

Оценки эффективности использования индексно-последовательных файлов получить труднее, чем оценки для двух предыдущих способов организации файлов, поскольку при реализации этого метода могут учитываться различные виды возможностей (опций). Мы получим такой файл, который используется в большинстве экономических систем широкого профиля.

С точки зрения порядка следования элементов индекс аналогичен самому файлу данных. Индекс первого уровня присоединяется к блокам данных, а индекс второго уровня имеет по одному элементу на блок индекса первого уровня. Индекс первого уровня, области данных и область переполнения хранятся на одном и том же цилиндре. Весь файл занимает несколько цилиндров.

После реорганизации область данных и область индекса заполнены полностью, а область переполнения пустая. Когда к файлу добавляется запись, она помещается в область переполнения и связывается с записями данных, с тем чтобы сохранить функциональную последовательность. Для этого в каждой записи отводится место для указателя. Записи, которые должны быть удалены, на самом деле не удаляются, а только помечаются как недействительные. Поэтому в каждой записи должно быть дополнительное поле, или нуль-индикатор, для флажка удаления. Для ссылок на область переполнения должен использоваться указатель. В периодах между реорганизациями не делается никаких изменений индекса, упрощающих процедуру включения записей. Все области состоят из блоков размера V .

Размер записи в индексно-последовательном файле. В последовательной части файла для каждой записи требуется пространство, необходимое для расположения a значений атрибутов данных и указателя на возможную область переполнения, т. е.

$$R = aV + P.$$

Дополнительное пространство отводится для o записей переполнения. Каждая из этих записей также имеет указатель, используемый для сцепления. Размер области, требуемый для каждой записи переполнения, равен

$$R_o = aV + P.$$

Кроме того, необходимо учесть индекс со значениями ключа и указателями. Размер элемента индекса равен $V + P$. Индекс первого уровня содержит по одному элементу на блок данных,

так что требуется

$$i_1 = \left\lceil n / \left\lfloor \frac{B}{R} \right\rfloor \right\rceil$$

его элементов. Индекс второго уровня содержит по одному элементу на каждый блок индекса первого уровня. Коэффициент расширения для самого индекса равен

$$y = \left\lfloor \frac{B}{V + P} \right\rfloor,$$

так что число элементов для второго и более высоких уровней индекса равно

$$i_{\text{уровень}} = \left\lceil \frac{i_{\text{уровень}-1}}{y} \right\rceil.$$

Число блоков, необходимых для индекса первого уровня, равно i_2 , а число блоков на каждом из последующих уровней равно числу элементов на соседнем уровне, расположенном выше. На последнем уровне один блок содержит весь корень индексного дерева. Объем области, необходимый для трехуровневого индекса, равен

$$SI = (i_2 + i_3 + i_4) B = (i_2 + i_3 + 1) B.$$

В тех случаях, когда блоки индекса первого уровня расположены на том же цилиндре, на котором находятся индексированные данные, необходимо зарезервировать область, достаточную для хранения одного элемента индекса на каждый блок данных.

В примере 19.11 вычисляется суммарный объем памяти (в расчете на один цилиндр), необходимый для хранения файла, изображенного на рис. 19.19, с использованием индексов, присоединенных к блокам данных.

Пример 19.11. Определение вместимости цилиндра для индексно-последовательного файла.

Параметры:

$R = 200$ (включая связующий указатель длиной 6 байт);

$B = 2000$;

$V = 14, P = 6$;

Число блоков на дорожке = 7, число поверхностей в цилиндре = 19, распределение под область переполнения = 20 процентов

Распределение памяти:

Размер первичного индекса в расчете на одну запись = $(14 + 6) / \lfloor 2000/200 \rfloor = 2$ байт

Объем области переполнения в расчете на одну запись = $0,20(2000) = 40$ байт

Максимальное число первичных записей данных в цилиндре = $\lfloor (19(7)(2000)) / (200 + 2 + 40) \rfloor = 1099$

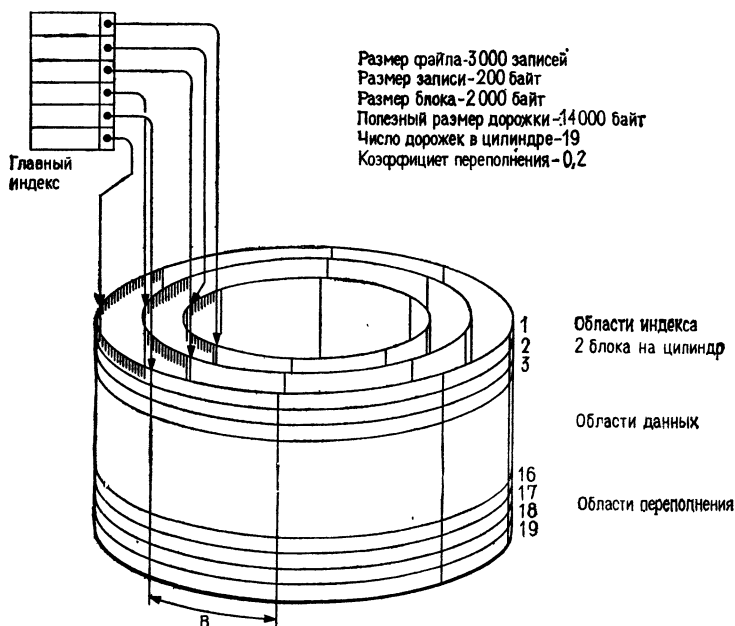


Рис. 19.19. Формат индексно-последовательного файла.

Полезное число записей данных в цилиндре, усеченное по кратному $B/R = 10, = 1090$

Область для записей переполнения во всех блоках/цилиндр = $1090(0,20) = 220$

Число элементов первичного индекса/цилиндр = $= 1090(200/2000) = 109$

Число блоков первичного индекса/цилиндр = $= \lceil (109(20))/2000 \rceil = 2$

Проверка: число используемых блоков в сравнении с числом имеющихся блоков $= (1090 + 220) / \lceil (2000/200) \rceil + 2$ в сравнении с 19(7) или $133 \leq 133$, что в точности соответствует требуемому соотношению.

Замечание: если при проверке обнаруживается, что потребовалось большее число блоков вследствие округления, то число блоков данных следует уменьшить. Общий необходимый объем памяти в расчете на одну запись (исключая относительно небольшие индексы более высоких уровней) теперь составляет $133(2000)/1090 = 245$ байт.

Для файла, изображенного на рис. 19.19, теперь требуется $\lceil 3000/1090 \rceil = 3$ цилиндра, а главный индекс состоит из 6 эле-

ментов. Общий требуемый объем памяти в расчете на одну запись после реорганизации составляет

$$R_{\text{общий}} = R + \frac{o}{n} R_0 + \frac{SI}{n}.$$

Обычно размеры записей R и R_0 равны.

Выборка записи из индексно-последовательного файла. Для локализации указанной записи используется индекс. Если файл имеет размеры, подобные тем, которые были заданы в примере 19.11, то при обращении к главному индексу доступа к диску не потребуется, поскольку при открытии файла главный индекс помещается в таблицу, расположенную в оперативной памяти. Процесс выборки состоит из просмотра главной таблицы, установки механизма чтения-записи, считывания индекса и считывания блока данных. Время на выборку составляет

$$T_{F0} = c + s + r + btl + r + btl, \text{ если } o = 0.$$

Однако если производилось включение записей, то данной процедуры будет не достаточно для того, чтобы найти включенную запись, и поиск необходимо будет продолжить по области переполнения. Вероятность Pov того, что требуемая запись находится в области переполнения, зависит от числа o' записей, которые включены в файл. Если выбор любой записи равновероятен, то

$$Pov = \frac{o'}{n + o'}.$$

Число rop записей в последовательном файле, которые будут содержать указатели переполнения, не превосходит o' , так как несколько записей переполнения могут быть сцеплены из одной первичной записи. Длина этой цепочки определяет затраты на доступ к записям переполнения. Для доступа к каждой записи цепочки, вообще говоря, потребуется считывание блока, так как записи в области переполнения не упорядочены.

Длина цепочки переполнения. Если в первичной области находится rop записей с указателями на область переполнения и $rop \leq o'$, то вероятность того, что первичная запись имеет указатель на область переполнения, равна

$$Plf = \frac{rop}{n}.$$

Распределение по цепочкам переполнения зависит от значений ключей поступающих записей. Рассмотрим здесь простейший случай, когда распределение равномерное. Тогда вероятность

того, что будет найдена вторая запись, сцепленная с данной записью переполнения, также равна rop/n . Поэтому полная вероятность существования второй записи равна

$$P2f = \frac{rop}{n} P1f = P1f^2 \text{ или в общем случае } Pif = P1f^i.$$

Количество записей в цепочке ограничено числом o' записей переполнения. Следовательно, ожидаемая сумма Ptf для записей, сцепленных с первичной записью, равна

$$Ptf = P1f + P2f + \dots + P{o'}f$$

и эта величина рассматривается для всех записей переполнения, так что ¹⁾

$$Pov = Ptf.$$

Выражение для Ptf , записанное в виде суммы величин Pif , является геометрической прогрессией (без начального члена 1, соответствующего $i = 0$), так что

$$Pov = Ptf = \sum_{i=1}^{o'} P1f^i = \frac{1 - P1f^{o'+1}}{1 - P1f} - 1.$$

Если значение o' достаточно большое (> 20), а $P1f \ll 1$ ($< 0,5$), то

$$Pov = Ptf = \frac{P1f}{1 - P1f}, \text{ так что } P1f = \frac{Pov}{1 + Pov}.$$

Тогда ожидаемая длина Lc цепочки равна

$$Lc = \frac{Pov}{P1f} = 1 + Pov.$$

При включении записи необходимо просмотреть всю цепочку, с тем чтобы связать запись с концом цепочки. Поэтому

$$Lc_l = Lc = 1 + Pov.$$

¹⁾ При $rop = o'$ данная формула не верна, так как $Pov = \frac{o'}{n + o'} < \frac{o'}{n} = \frac{rop}{n} \leq Ptf$. Формулу для длины цепочки переполнения следует выводить следующим образом. Вероятность того, что с заданной первичной записью связана некоторая цепочка переполнения, составляет $p = 1 - \left(1 - \frac{1}{n}\right)^{o'}$. Поэтому ожидаемое число цепочек переполнения составляет np , а их средняя длина равна $o'/np \approx 1 + o'/(2n)$. Тогда $Lc_l = Lc = 1 + o'/(2n)$, $Lc_r = (1 + Lc)/2 = 1 + o'/(4n)$. Соответствующим образом следует изменить формулы, содержащие эти величины. — *Прим. перев.*

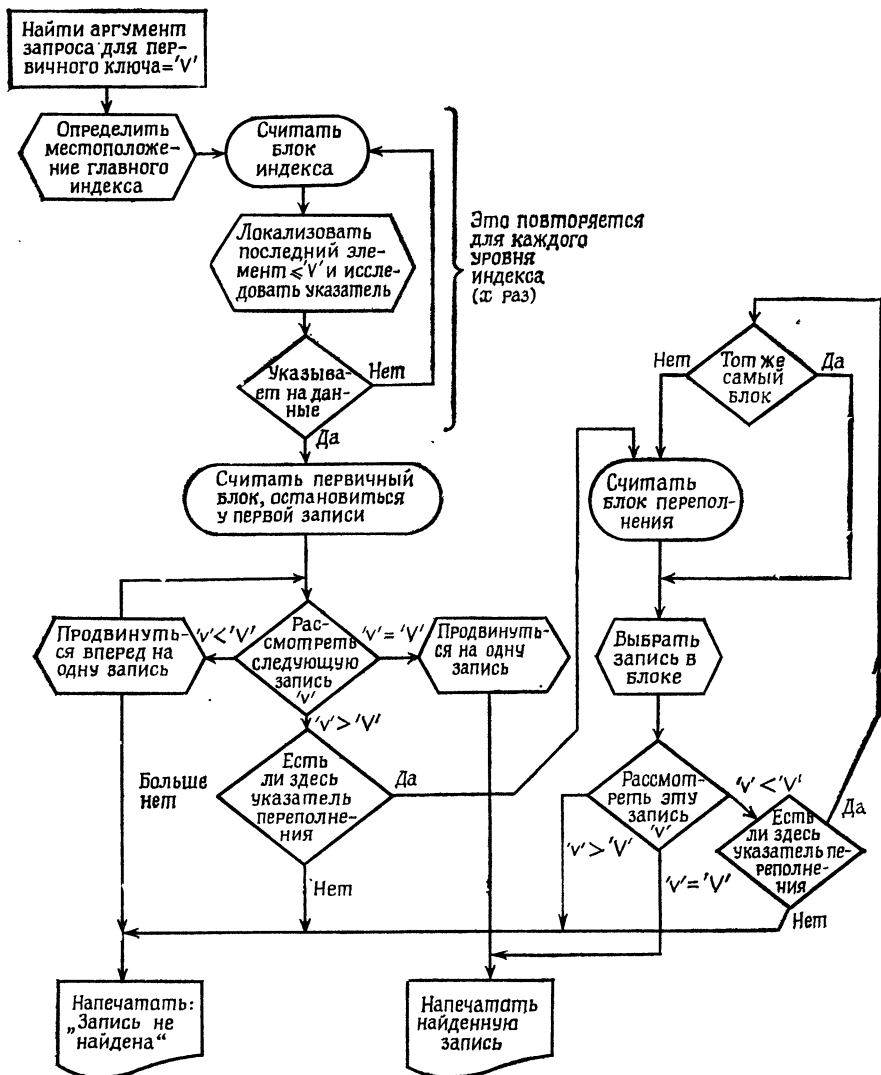


Рис. 19.20. Выборка записи из индексно-последовательного файла.

При выборке учитывается средняя длина:

$$Lc_F = \frac{1 + Lc}{2} = 1 + \frac{1}{2} Pov.$$

Если s_{ov} — время установки головок к области переполнения, то

$$T_F = T_{F0} + Pov(s_{ov} + Lc_F(r + btt)).$$

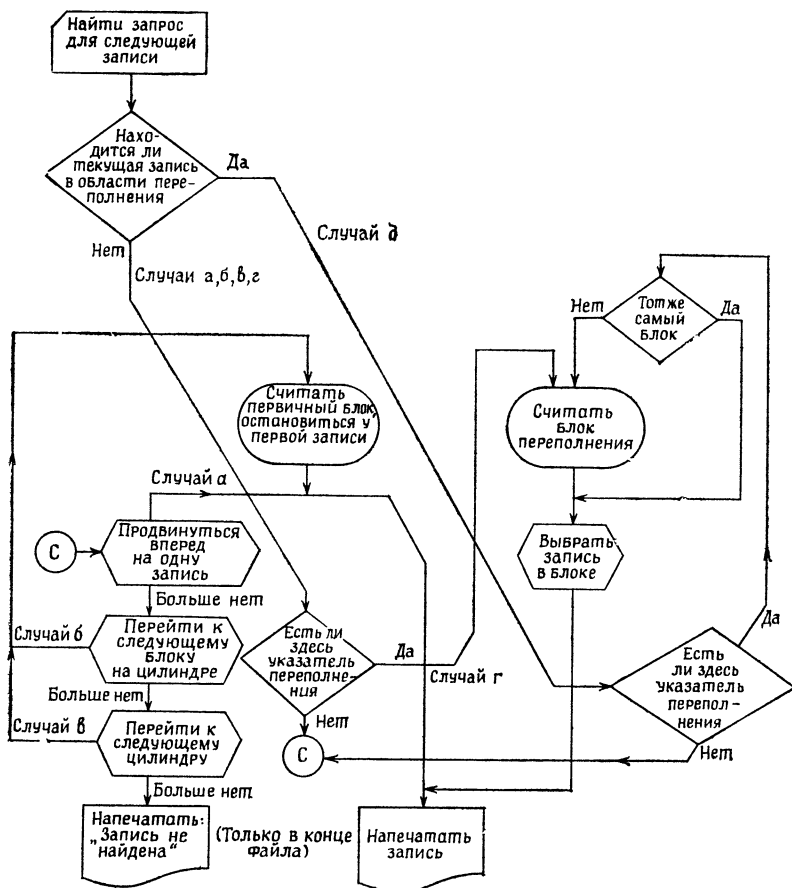


Рис. 19.21. Получение следующей записи из индексно-последовательного файла. Отметим, что большая часть действий, указанных на этой схеме, совпадает с теми, которые изображены на рис. 19.20.

Эта процедура схематически изображена на рис. 19.20. Если область переполнения и первичная область расположены на одном и том же цилиндре, то $s_{ov} = 0$ и ¹⁾

$$T_F = T_{F0} + PovLc_F(r + btt) = \\ = c + s + \left(2 + Pov \left(1 + \frac{1}{2} Pov\right)\right)(r + btt).$$

Оценка времени выборки. Если реорганизация выполняется в тех случаях, когда область переполнения заполнена на

¹⁾ См. примечание на с. 369.— *Прим. перев.*

80 процентов, то среднее значение o' равно 0,4 о. Если в этом случае объем области переполнения составляет 20 процентов объема области, отведенной для первичного файла, то

$$o' = (0,20)(0,4)n = 0,08n.$$

Из полученных выше соотношений следует, что ¹⁾

$$Pov = \frac{0,08}{1,08} = 0,0741 \text{ и } Lc_F = 1 + \frac{0,0741}{2} = 1,037.$$

Отсюда следует, что среднее время выборки равно

$$T_F = c + s + 2,077(r + btt).$$

Если записи добавляются не равномерно, то величина T_F может значительно возрасти.

Получение следующей записи из индексно-последовательного файла. Для локализации следующей записи поиск следует начать от текущей записи данных, не принимая во внимание индекс. Необходимо определить, может ли упорядоченное считывание записей выполняться последовательно или следует перейти к другой области. В зависимости от расположения ²⁾ предшествующей и последующей записей возможны различные варианты поиска. Используя числовые данные рис. 19.14, проиллюстрируем пять различных случаев. Данная процедура в виде блок-схемы изображена на рис. 19.21.

Пути доступа к следующей записи

а) После последней записи включения не производилось, и следующая запись расположена в том же самом первичном блоке данных, который уже находится в оперативной памяти (последней является запись 12.1).

б) Промежуточного включения не производилось, но запись находится в следующем блоке на том же цилиндре (последней является запись 12.3).

в) Включения не производилось, но запись находится в новом блоке на другом цилиндре [если в одном цилиндре находится β блоков и файл начинается с нового цилиндра, то это произошло бы между записями $\beta.3$ и $(\beta + 1).1$, $2\beta.3$ и $(2\beta + 1).1$ и т. д.].

г) Включение производилось, и следующая запись находится в блоке переполнения (последней является запись 12.2).

¹⁾ См. примечание на с. 369. — *Прим. перев.*

²⁾ Имеется в виду физическое местоположение записи: в одном блоке, в разных блоках и т. п. — *Прим. ред.*

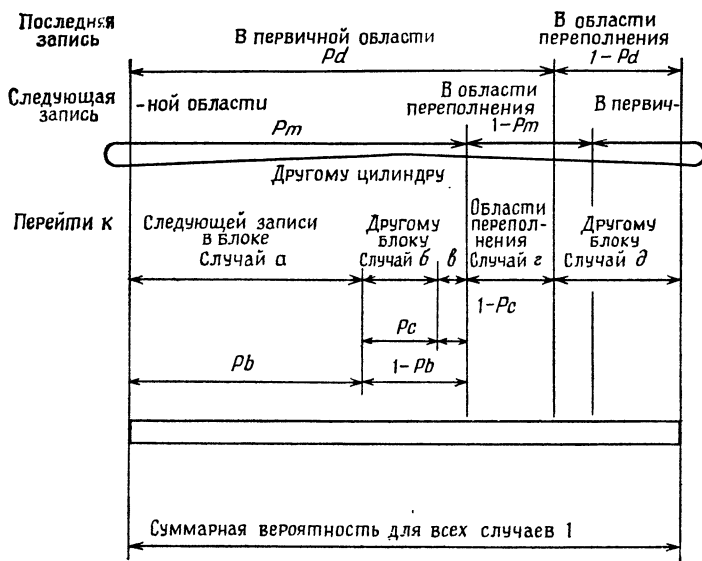


Рис. 19.22. Условия, возникающие при поиске следующей записи.

д) Последней является включенная запись, так что для поиска следующей записи необходимо считать новый блок (данных или записей переполнения) на том же цилиндре (последней является запись 110.1 или 111.1).

Для определения вероятностей возникновения указанных событий введем следующие обозначения:

P_d Текущая запись находится в первичном блоке данных.

$$P_d = 1 - P_{ov}.$$

P_m Промежуточного включения не производилось. $P_m = 1 - P_{1f}$ (см. данное выше определение).

P_b Запись находится в том же самом блоке. $P_b = 1 - R/B$.

P_c Блок находится на том же самом цилиндре. $P_c = 1 - 1/\beta$.

На рис. 19.22 схематически описаны условия, введенные на рис. 19.21. Суммируя возможные затраты, умноженные на соответствующие вероятности, получаем среднее время поиска следующей записи, равное

$$\begin{aligned}
 T_N = & (P_d)(P_m)(P_b)(c) + & (* \text{ случай } a *) \\
 & (P_d)(P_m)(1 - P_b)(P_c)(r + btt) + & (* \text{ случай } б *) \\
 & (P_d)(P_m)(1 - P_b)(1 - P_c)(s + r + btt) + & (* \text{ случай } в *) \\
 & (P_d)(1 - P_m)(r + btt) + & (* \text{ случай } г *) \\
 & (1 - P_d)(r + btt) & (* \text{ случай } д *)
 \end{aligned}$$

Как Pd , так и Pm могут быть выражены через Pov с помощью полученных выше соотношений для длины цепочки¹⁾.

Если можно пренебречь временем установки механизма чтения-записи (т. е. значение β достаточно большое, так что $Pc \approx 1$) и временем c поиска записей в блоке, расположенном в оперативной памяти, а также если предположить, что величина Pov достаточно мала (так что слагаемые, содержащие Pov^2 , могут быть опущены), то полученное выражение можно упростить следующим образом:

$$T_N = \left(\frac{R}{B} + 2Pov \left(1 - \frac{R}{B} \right) \right) (r + btt).$$

Включение записи в индексно-последовательный файл. При добавлении записей к файлу необходимо считать и переписать предшествующую запись для того, чтобы включить или изменить указатель. Кроме того, необходимо считать и переписать текущий блок переполнения. Время выборки предшествующей записи равно T_F , блок переполнения расположен на том же самом цилиндре, и время, необходимое для его считывания, равно $r + btt$, а для каждой переписи записей требуется один полный оборот диска, т. е. $T_{RW} = 2r$. Следовательно,

$$T_I = T_F + 2r + r + btt + 2r = T_F + 5r + btt.$$

Обновление записи в индексно-последовательном файле. Обновленная запись, имеющая прежний размер и не измененный ключ, может быть помещена в первоначальном месте, так что эта процедура аналогична выборке, но содержит дополнительную операцию записи, время выполнения которой T_{RW} . Поэтому $T_{Ud} = T_F + 2r$ для изменений неключевых полей.

В общем случае предыдущая копия записи удаляется и соответствующим образом включается новая запись. При удалении старая запись переписывается с нуль-индикатором, однако поля ключа и указателя остаются неизменными, так что структура файла сохраняется прежней. Поэтому

$$T_U = T_{Ud} + T_I = 2T_F + 7r + btt.$$

Первая процедура (когда время обновления составляет T_{Ud}) во многих системах не реализована (за исключением явных удалений), так что время обновления записей для этих систем всегда составляет T_U .

Полное считывание индексно-последовательного файла. Исчерпывающий поиск в файле необходим в тех случаях, когда аргумент поиска не является индексированным атрибутом. Здесь возможны два варианта. В первом осуществляется упо-

¹⁾ См. примечание на с. 369. — *Прим. перев.*

рядоченное считывание записей файла. Во втором последовательно считываются все записи области данных на цилиндре, после чего последовательно считываются все записи области переполнения. В большинстве систем возможно только упорядоченное считывание, так что

$$T_X = T_F + (n + o' - 1)T_N \approx (n + o')T_N.$$

Если первичный блок данных может быть сохранен, то считывание из первичной области можно выполнять со скоростью обмена больших массивов, в то время как записи переполнения по-прежнему будут считываться упорядоченно. Тогда для упорядоченного считывания с буферизацией.

$$T_X = n \frac{R}{t'} + o' (r + btt).$$

Во время считывания блока переполнения первичной блок может храниться в оперативной памяти и обрабатываться в дальнейшем.

При оценке времени последовательного считывания учитывают скорость эффективного обмена, пренебрегая задержкой, возникающей при переходе от блоков данных к блокам переполнения, поскольку она возникает не более одного раза в расчете на цилиндр. Для последовательного считывания

$$T_X = (n + o') \frac{R}{t'}.$$

Реорганизация индексно-последовательного файла. Для реорганизации старого файла все записи считываются упорядоченно и переписываются без использования областей переполнения. Одновременно организуется новый индекс. Старый индекс не используется, поскольку записи считываются упорядоченно. Предположим, что в областях переполнения находится o' новых записей. Предположим также, что программа реорганизации может одновременно обрабатывать несколько блоков, так же как в рассмотренном выше случае упорядоченного считывания с буферизацией. Для сбора новых данных и информации для индекса требуются два дополнительных буфера в оперативной памяти. При их заполнении запись содержащихся в них данных можно выполнить последовательно. Тогда

$$T_Y = n \frac{R}{t'} + o' (r + btt) + (n + o' - d) \left(\frac{R}{t'} + \frac{SI}{t'} \right).$$

Значение o' меньше o , числа записей, для которых была отведена область переполнения. Ранее мы предполагали, что $o' = 0,8o$. Такое предположение было бы оправданным в том случае, если стратегия реорганизации была следующей: реорганизация файла должна выполняться в первую же ночь после того,

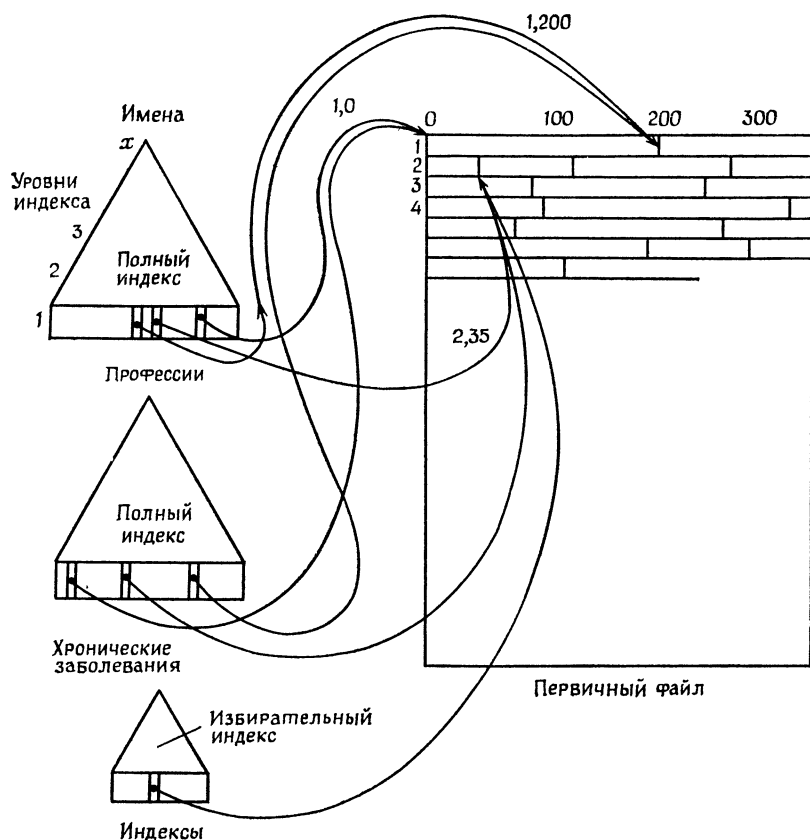


Рис. 19.23. Связь записей в индексированном файле.

как область переполнения заполняется на 75 процентов, и среднее ежедневное увеличение количества данных в области переполнения составляет 10 процентов.

Если предположить, что $o' = 0$, то для числа записей переполнения, которые должны быть обработаны, будет получена завышенная оценка.

19.14. ИНДЕКСИРОВАННЫЙ ФАЙЛ

При отсутствии требования последовательной упорядоченности записей, благодаря которой может быть обеспечен эффективный последовательный доступ, возможно использование индексированных файлов. Доступ к записям индексированного

фрагмент текста

значение ключевого атрибута	указатель
, quant vit pasmer Rollant, / dunc out tel	doel unkes mais n'out si grant. / Tendit sa mai 2223
la sele en-remeint guaste. / Mult ad grant	doel Carlemagnes li reis, / quant Naimun veit 3451
c. / Co dist li reis: "Seignurs, vengez voz	doels, / si esclargiez voz talenz e voz coers, 3627
chevalier." / Respont li quens: "Deus le me	doinst venger!" / Sun cheval brochet des esperu 1548
ad mort France ad mis en exill. / Si grant	dol ai que ne voldreie vivre, / de ma maisnee, 2936
d sanc. / Franceis murrunt, Casles en ert	dolent. / Tere Majur vos metrum en present. 951
ent, / e cil d' Espaigne s'en cleiment tuit	dolent. / Dient Franceis: "Ben fiert nostre gu 1651
alchet ireement, / e li Franceis curucus e	dolent; / n'i ad celui n'i plurt e se dement, 1835
ma gent." / E cil respunt "Tant sy jo plus	dolert. / Ne pois a vos tenir lung parlement: 2835
sal duluset: / jamais en tere n'orrez plus	dolent hume! / Or veit Rollant gue mort est su 2023
devers les porz d' Espaigne: / veoir poez,	dolente est la reregarde: / ki ceste fait, jan 1104
e vient curant cuntre lui; / si li ad dit:	"Dolente, si mare fui! / A itel hunte, sire, mon 1813
pereres cevalchet par irur / e li Franceis	dolenz e curucus; / n'i ad celui ki durement ne 2695
alenur, / plurent e crient, demeneint grant	dolor, / pleignent lur deus, Tervagan e Mahum 2946
perere, co dist Gefrei d' Anjou, / "ceste	dolor ne demenez tant fort! / Par tut le camp f 489
ance ad en baillie, / que me remembre de la	dolur e l'ire, / co est de Basan e de sun frer 2577
amumede, / pluret e criet, mult forment se	doluset; / ensembl'od li plus de xx. mil humes, 2521
out mais en avant. / Par tuz les prez or se	dorment li Franc. / N'i ad cheval ki puisse e 2525
ad apris ki bien conuist ahan. / Karles se	dort cum hume travaillet. / Seint Gabriel li a 2494
poent plus faire. / Ki mult est las, il se	dort cuntre tere. / Icele nuit n'unt unkes esca 718
it le jur, la nuit est aserie. / Carles se	dort, li empereres riches. / Sunjat qu'il eret 736
ent liuels d'els la veintrat. / Carles se	dort, mie ne s'esveillat. AOT. / Tresvait la no 724
le cel en volent les escicles. / Carles se	dort, qu'il ne s'esveillat mie. / Apres iceste, 2569
s Deu co ad mustret al barun. / Carles se	dort tresqu'al demain, al cler jur. / Li reis 1201
et les os, / tute l'eschine li desevert del	dos, / od sun espiet l'anme li getet fors, 1588
gemmet ad or, / e al cheval parfundement el	dos; / ambure ocit, ki quel blasme ne quil lot, 1945
eruns a or, / fiert Oliver derere en mi le	dos. / Le blanc osberc li ad descust el cors, 3222
ros; / sur les eschines qu'il unt en mi les	dos / cil sunt seiect ensement cume porc. AOT. 3922
re joe en ad tute sanglante; / l'osberc del	dos jusque par sum le ventre. / Deus le guarit 1649
ele les dous alves d'argent / e al cheval le	dos parfundement: / ambure ocist seinz nul reco 2445
t li ber. / De cels d' Espaigne unt lur les	dos turnez, / tenent l'enchalz, tuit en sunt cu 1440
a fuis: / de cent millers n'en poent guarir	dous. / Rollant dist: "Nostre jume sunt mult p 1648
s e l'osberc jazerenc, / de l'oree sele les	dous alves d'argent / e al cheval le dos parfund 2874
tet en ad, ne poet muer n'en plurt. / Desuz	dous arbres parvenuz est . . . li reis. / Les c 3874
Dedesuz Ais est la pree mult large: / des	dous baruns justee est la bataille. / Cil sunt 539
agne, ki est canuz e vielz! / Men esgientre	dous cenz anz ad e mielz. / Par tantes teres ad 524
t vielz, si ad sun tens usef: / men escient	dous cenz anz ad passet. / Par tantes teres at

Рис. 19.24. Пример алфавитного указателя для *Chanson de Roland*. (Взято из Joseph J. Duggan, «A Concordance of the Chanson de Roland», Ohio State University Press, Columbus, 1969.)

файла (рис. 19.23) осуществляется только с помощью одного или более индексов. Ограничение на порядок взаимного расположения записей данных отсутствует, поскольку в некотором индексе имеется указатель, который позволяет осуществить выборку нужной записи. Одному ключевому атрибуту соответствует только один индекс (совокупность записей индекса). Индексы могут быть организованы для всех атрибутов, для которых возможно задание аргумента поиска. Несмотря на отсутствие физической упорядоченности записей относительно первичного ключевого атрибута, большая гибкость может сделать эту организацию файлов более предпочтительной по сравнению с индексно-последовательной.

Инвертированные файлы. Файл, для которого были организованы индексы, иногда называют **инвертированным файлом**. Происхождение этого термина связано с процессом индексирования библиотечной картотеки. Элементом полного индекса файла, содержащего английский текст, могут быть все различные слова в файле. Такой индекс по существу является словарем с указателями на все места в тексте, где встречаются эти слова. Если вместе с указателями даются фрагменты текста, то такой словарь принимает форму **алфавитного указателя**. Конечно, текст может быть написан не только на английском языке. Часть алфавитного указателя показана на рис. 19.24. В текстовом файле имеется только один домен, **слова**, а индекс содержит по несколько элементов на запись, одну для каждого слова. При частичной инверсии могут быть исключены часто повторяющиеся слова или слова, не представляющие интереса, как, например, инициалы авторов. Термины: инвертированный индекс, инвертированный список, инвертированный файл и частично инвертированный файл — в литературе используются очень несогласованно. Часто их связывают с определенным способом индексирования. Индексирование с точки зрения создания картотеки — это отбор и организация существенных значений атрибутов для последующего поиска. Инвертированным файлом иногда называют копию последовательного файла, рассортированного в соответствии с другим ключевым атрибутом,

Структура индексированных файлов и работа с ними

Число индексов в индексированном файле может доходить до числа типов атрибутов. Индекс, соответствующий некоторому атрибуту, содержит по одному элементу для каждой записи. Эти элементы упорядочены в соответствии со значениями данного атрибута. Каждый элемент состоит из значения атрибута и указателя на запись. В индексированных файлах выборка следующей записи осуществляется не с использованием физической упорядоченности записей или указателя из предшествующей записи, а с помощью следующего элемента индекса. Каждый индекс может содержать несколько уровней, так же как индекс индексно-последовательного файла.

Формат записи данных может быть таким же, как и в любой из рассмотренных ранее организаций. Использование записей, содержащих пары имя атрибута — значение (как в файле-множестве), предпочтительнее в тех случаях, когда плотность атрибутов a'/a низкая; в остальных случаях могут быть использованы структурированные записи. Поскольку указатели в индексе определяют для каждой записи адрес блока и расположение записи, то ограничений на размеры записей или их распо-

ложение внутри отдельного блока по существу не накладывается. Записи могут включаться всюду, где имеется достаточно свободного пространства.

Полный и избирательный индексы. Индексы могут быть **полными**, т. е. содержащими указатели на все записи, и **избирательными**, т. е. содержащими указатели только на те записи, в которых значения атрибута являются значащими. Значащими иногда называют все существующие (не являющиеся неопределенными или нулевыми) значения атрибута, а в некоторых случаях ограничиваются определенной областью значений, которые больше всего подходят для использования в качестве ключей поиска.

Избирательный индекс может быть использован, например, при обработке файла сведений о состоянии здоровья служащих, когда организованы только индексы временных и хронических заболеваний, хотя для ведения статистики или в других целях в самом файле данных хранится более полная запись. Соответствующий пример (индекс “хронических заболеваний”) символически изображен на рис. 19.23. Другой избирательный индекс для файла сведений о состоянии здоровья служащих мог бы использоваться для указания всех лиц, уровень холестерина в крови которых превышает 250. Необходимость в таком индексе возникает в тех случаях, когда на эти данные поступают частые запросы, например для правильного приготовления пищи.

Если не организован ни один полный индекс, то для правильного ведения файла необходимо иметь таблицу распределения памяти. Такая таблица содержит информацию о всем пространстве, отведенном для файла. При упорядоченном считывании записей файла в соответствии с этой таблицей проявляется такой же порядок их расположения, как в файле-множестве.

Ведение индексов. Основная проблема, возникающая при использовании индексированных файлов, связана, во-первых, с необходимостью корректировать все индексы при каждом добавлении, удалении или перемещении записи и, во-вторых, с необходимостью изменять отдельные индексы при изменении значения поля. Необходимость корректировать индексы для индексно-последовательных файлов не возникала благодаря использованию цепочек указателей на включенные записи. Использование цепочек переполнения не соответствовало бы назначению индексированной организации, поэтому необходимо ведение индекса. Но при этом немедленное изменение всех индексов не всегда целесообразно. Например, в некоторых случаях обработку высокоприоритетных запросов на выборку дан-

ных из файла желательно проводить до обновления всех индексов.

Непрерывное ведение индексов может привести к большим временным затратам. Иногда индексы организуются для специального анализа, но после изменения файла не обновляются. В некоторых случаях индексы корректируются только периодически, в результате чего последние данные до проведения корректировки недоступны.

Корректировка индексов может выполняться в то время, когда ЭВМ относительно свободна. Если такую корректировку можно выполнять поочередно для небольших участков, то отпадает необходимость в периодическом проведении полной реорганизации, так что файл остается доступным. Такая корректировка индекса выполняется процессом, который активен с низким приоритетом до тех пор, пока она не завершится. Старую запись необходимо сохранять на прежнем месте до тех пор, пока не будет гарантии, что скорректированы все справочные индексы. При проведении корректировки (пока файл находится в неактивном состоянии) необходимо тщательно упорядочить корректирующие операции и сохранять информацию о состоянии незавершенных операций. При решении некоторых задач старые данные не используются. Однако, если это так, старые записи лучше всего пометать как недействительные: **удаленные** или **перемещенные**. В последнем случае новое расположение записи можно указывать с помощью метки. Если изменяются значения атрибутов, то поиск новой записи по соответствующему ключу не будет возможен до тех пор, пока не будет скорректирован соответствующий индекс. Процессу, корректирующему индексы важных атрибутов, может быть назначен более высокий приоритет, чем другим процессам корректировки индексов.

Анкерные точки. Метод блокировки и назначения анкерных точек, который используется для индексно-последовательной организации файлов и позволяет уменьшать размер индекса, в данном случае не применим, поскольку способы упорядочения индекса и данных различны. Поэтому необходимо, чтобы индекс самого низкого уровня содержал элемент для каждой значащей записи.

Использование индексированных файлов

Индексированные файлы используются главным образом в тех областях, где крайне необходимо своевременное получение информации. Например, они находят применение в системах резервирования авиабилетов, рабочих банках, военных инфор-

мационных системах и других областях, связанных с управлением запасами. Здесь редко встречается упорядоченная обработка данных, если не считать проводящийся время от времени (возможно, раз в год) переучет.

При получении некоторой информации, например сведений о наличии свободного места на определенный рейс, данные должны отражать положение дел на текущий момент, а если информация обновляется, т. е. билет на данный рейс продается, то этот факт должен быть немедленно передан системе.

С помощью мультииндексирования информации можно получить номер рейса по имени пассажира, по записи производимой при пересадке, и т. д., не проводя реорганизации файла и не допуская избыточности данных. Конечно, в отношениях между индексом и данными существует избыточность.

Индексированные файлы широко применяются также в тех случаях, когда изменение данных происходит часто. Гибкость, существующая в индексированном файле при выборе формата и порядка взаимного расположения записей, отсутствует в других файловых системах. Однако в случае, когда используется только один ключевой атрибут, подобную гибкость обеспечивают древовидные файлы. В тех случаях, когда данные полностью индексированы, вся информация содержится в индексе и сам файл данных может быть исключен. Получаемый при этом файл называется фиктивным. В тех случаях, когда важен упорядоченный доступ, альтернативой индексированному файлу может быть транспонированный файл.

Эффективность использования индексированных файлов

Оценки эффективности использования индексированных файлов получаются проще, чем соответствующие оценки для рассмотренных ранее индексно-последовательных файлов. Заметим, что следует критически подходить к выбору числа атрибутов, подлежащих индексированию. В противном случае полные индексы для всех атрибутов будут просто превосходить по размеру исходный файл. На практике всегда существует несколько атрибутов, для которых индексирование не оправдано. Это обстоятельство позволяет уменьшить размер индекса, но не влияет на другие факторы эффективности, поскольку эти индексы никогда не используются. Атрибуты, которые имеют низкую эффективность разбиения, плохо подходят для индексирования.

При получении оценок рассматриваются полностью индексированные файлы. Предполагается, что для каждого атрибута имеется только один уровень индекса и что значения атрибутов данных являются разреженными, так что используются записи данных переменной длины и избирательные индексы.

Размер записи в индексированном файле. Размер области, необходимой для хранения данных, содержащихся в индексированном файле, такой же, как и в файле-множестве. Кроме того, для того чтобы организовать индекс для каждого атрибута, потребуется a индексов. Поскольку атрибуты данных разреженные, на каждую запись приходится только a' атрибутов; в каждом индексе в среднем содержится по na'/a элементов, указывающих на записи данных, а размер каждого элемента равен $V' + P$. Требуемый объем области в расчете на одну запись складывается из объема области для индекса и объема области для данных. Поскольку значения всех атрибутов, хранящиеся в заданной записи, индексированы, то размер области, отводимой для записи (включая индекс и данные), равен

$$R_{\text{общий}} = a'(V' + P) + a'(A + V + 2).$$

Для осуществления быстрого поиска в индексе может потребоваться, чтобы поле значения в индексе имело фиксированную длину, так что V' может быть больше, чем средний размер поля значения в записи данных. С другой стороны, если часто встречаются группы записей, содержащих одни и те же значения атрибута (например, если атрибутом является “профессия”, то в индексе существует много элементов, соответствующих такому значению этого атрибута, как “сварщик”), то один элемент значения может использоваться для многих указателей. Кроме того, могут использоваться другие приемы, позволяющие уменьшить размеры значений ключей в индексах, так что $V' < V$. Если предположить, что $V' = V$, то из предыдущего соотношения следует, что

$$R_{\text{общий}} = a'(A + 2V + P + 2).$$

В тех случаях, когда требуется многоуровневое индексирование, используются дополнительные блоки индекса. В большинстве проектов объем памяти, необходимый для их хранения, составляет лишь несколько процентов от общего требуемого объема памяти (см. пример 19.12).

Степень заполнения и рост индекса. В тех случаях, когда включение записей осуществляется часто, желательно зарезервировать дополнительное пространство в блоках индекса. Если индекс первоначально заполнен на 80 процентов, то можно включить еще 20 процентов дополнительных записей, т. е. $o = n/4$. Иными словами, начальная плотность заполнения равна 0,8. Возможное расширение индекса обратно пропорционально начальной плотности заполнения, так что

$$R_{\text{общий}} = a' \frac{n+o}{n} (V' + P) + a'(A + V + 2),$$

однако область данных будет расширяться только пропорционально фактическому числу записей. Алгоритм, управляющий ростом блоков индекса (образующих **В-дерево**), при заполнении одного блока индекса строит два наполовину заполненных блока, заменяющих первоначальный. При каждом расщеплении в следующий расположенный выше уровень помещается дополнительный элемент. При расщеплении блока самого высокого уровня создается новый главный уровень, исходными в котором являются только что полученные два элемента. Если не считать начальное заполнение, то число элементов в блоке индекса колеблется от $\frac{1}{2}(y + 1)$ (сразу после расщепления) до y (непосредственно до расщепления), так что средняя плотность заполнения составляет

$$\text{Плотность} = \frac{1}{2} \frac{\frac{1}{2}(y + 1) + y}{y} = 75 \text{ процентов.}$$

Эта величина определяет требование к объему памяти для индекса: по мере того как индекс будет расти, потребуются новые блоки, поскольку в среднем этот коэффициент заполнения будет постоянным. Распределение элементов по смежным блокам до расщепления некоторого блока предполагается таким, что плотность **В-дерева** увеличивается.

Для определения местоположения индекса, соответствующего имени атрибута, необходим также справочник атрибутов.

Выборка записи из индексированного файла. Оценка ожидаемого времени выборки записи из индексированного файла получается аналогично соответствующей оценке для индексно-последовательного файла. Однако областей переполнения в индексированном файле нет, и поэтому слагаемое, соответствующее времени поиска среди записей переполнения, исключается. Поскольку, вообще говоря, существует несколько подфайлов индекса, то нельзя предполагать, что эти индексы и данные будут находиться на одном и том же цилиндре. Элементы индексов соответствуют записям данных. Кроме того, индексы содержат элементы, соответствующие добавленным записям, так что значение n здесь включает в себя число o' добавленных записей (которое было определено при рассмотрении индексно-последовательных файлов). Складывая время доступа к индексу и время доступа к данным, получаем, что для файлов с одноуровневым индексом

$$T_F = s + r + btt + s + r + btt,$$

или

$$T_F = 2(s + r + btt).$$

Если предположение о том, что размеры индекса невелики, не верно, то возникают дополнительные задержки. Рассмотрим вначале один-единственный индекс. Существуют два метода доступа. Во-первых, можно использовать метод двоичного поиска. Во-вторых, могут быть добавлены дополнительные уровни индекса.

Многоуровневый индекс. Число уровней индексации является функцией числа элементов индекса, которые могут быть размещены в одном блоке индекса. Для оценки времени доступа будем предполагать, что все индексы содержат равное число элементов. Тогда одному индексу соответствует $n(a'/a)$ записей, где a' — как и раньше, ожидаемое число атрибутов в записи. Число записей данных, достижимых с помощью одного блока индекса, равно коэффициенту расширения $B/(V+P)$, так что число блоков индекса первого уровня составляет

$$N1IB = n \frac{a'/a}{B/(V+P)}.$$

На каждый блок индекса более низкого уровня требуется один элемент индекса следующего более высокого уровня, так что число блоков индекса второго уровня равно

$$N2IB = \frac{N1IB}{B/(V+P)}.$$

Аналогичные рассуждения можно провести для уровней 3, 4, ..., x до тех пор, пока значение $NxIB$ не станет меньше или равным 1. Поэтому необходимое число уровней равно

$$x = \left\lceil \log_y \left(n \frac{a'}{a} \right) \right\rceil,$$

$$\text{где } y = \left\lfloor \frac{B}{V+P} \right\rfloor.$$

Процесс поиска в виде блок-схемы изображен на рис. 19.25.

В предыдущем разделе величина x была оценена итеративно. В случае индексно-последовательных файлов элементы первого уровня индекса соответствуют блокам записей. Поэтому если полученную здесь формулу использовать для индексно-последовательных файлов, то n следует заменить на nR/B .

Хранение главных индексов в оперативной памяти. Если все или используемые в текущий момент блоки индекса уровня x могут храниться в оперативной памяти, то фактическое значение x может быть уменьшено на 1. Для хранения всех главных индексов в оперативной памяти может потребоваться область размера до aB знаков. Блоки индекса главного уровня обычно заполняются не полностью, и на практике потребуется около

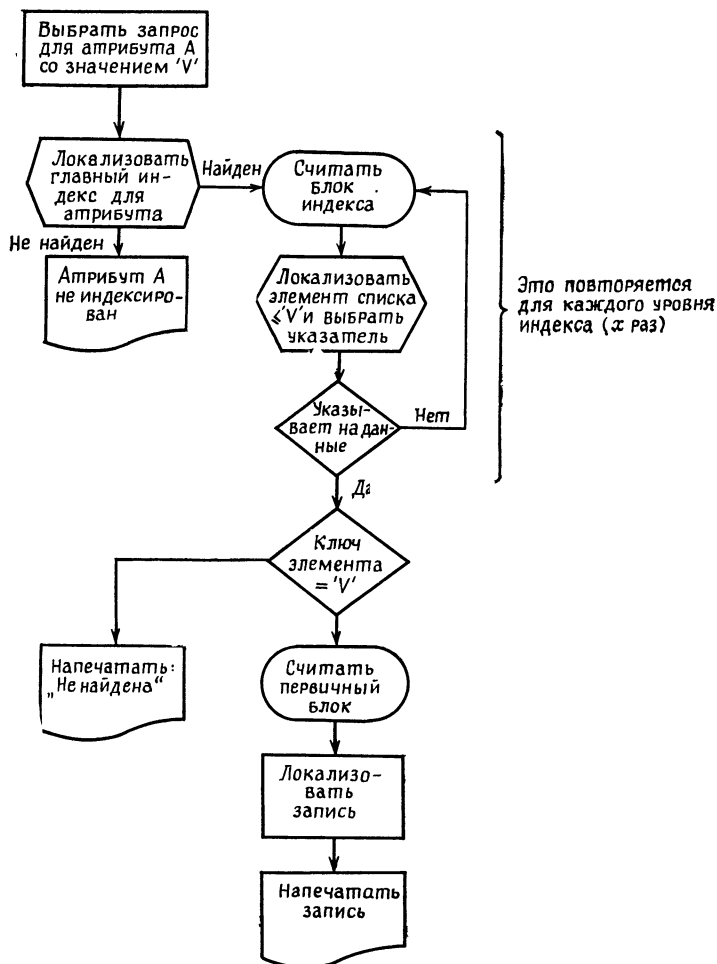


Рис. 19.25. Выборка с использованием одного индекса мультииндексированного файла.

половины указанной области. Однако и этого, как правило, хватает с избытком. В дальнейшем эта возможность не будет нами рассматриваться, поскольку для ее реализации требуются важные дефицитные ресурсы и сложное управление оперативной памятью.

Пример 19.12. Рассмотрим файл, содержащий сведения о специальностях $n = 20\,000$ служащих. Каждый служащий

имеет в среднем 2,5 специальности, так что в действительности здесь a'/a больше 1.

Каждая специальность задается кодом, состоящим из 6 знаков. Для нахождения записи используется указатель, состоящий из 8 цифр и занимающий 4 знаковые позиции. Поэтому элемент индекса занимает 10 байт. Длина блока в файловой системе составляет 1000 знаков, так что в блоке может содержаться 100 элементов индекса.

$$N1IB = 20\,000(2,5)/100 \rightarrow 500$$

$$N2IB = 500/100 \rightarrow 5$$

$N3IB = 5/100 \rightarrow < 1$ блок (т. е. в оперативной памяти содержится таблица из пяти элементов)

Вычисляя x непосредственно, получаем

$$x = \log_{100}(20\,000(2,5)) = 2,69897 < 3.$$

Для многоуровневого индексированного файла время выборки равно

$$T_F = (1 + x)(s + r + btt).$$

Если весь индекс можно хранить в одном цилиндре, то

$$T_F = 2s + (1 + x)(r + btt).$$

Получение следующей записи из индексированного файла. При поиске следующей записи предполагается, что последний блок индекса находится в оперативной памяти, так что необходимо выбрать только новую запись. Таким образом,

$$T_N = s + r + btt.$$

Это справедливо в тех случаях, когда коэффициент расширения $y \gg 1$, так как вероятность того, что текущий блок индекса содержит указатель на следующую запись, равна $(y - 1)/y$.

Включение записи в индексированный файл. При добавлении записи к индексированному файлу она помещается в любую свободную область, после чего корректируются все a' индексов, соответствующих атрибутам этой записи. Если в блоках индекса имеется достаточно места (плотность заполнения индекса < 1), то для включения записи необходимо только найти нужные блоки индекса, прочитать их, а затем скорректировать и записать на прежнее место.

Суммируя времена, требуемые для того, чтобы выбрать и перезаписать блок данных, просмотреть a' индексов по x уровням и перезаписать индексы первого уровня, получаем

$$T'_I = s + r + btt + 2r + xa'(s + r + btt) + a'2r,$$

или

$$T'_I = (1 + xa')s + (3 + xa' + 2a')r + (1 + xa')btt.$$

Эту величину также можно уменьшить, если все индексы хранить на одном цилиндре. Если размеры всех индексов небольшие (так что $x = 1$), то

$$T'_I = (1 + a')(s + 3r + btt).$$

Переполнение индексов. В худшем случае включение записи приводит к переполнению всех блоков индексов, в результате чего они должны быть расширены, или расщеплены. Тогда дополнительно необходимо записать $a'x$ блоков индексов.

Для организации одного блока индекса необходимо переслать данные из блока, который переполнен, в новый буфер, переписать старый блок, записать новый блок и скорректировать блок индекса следующего более высокого уровня. В записанном выше выражении для T'_I учитывалось время, необходимое для перезаписи старого блока низшего уровня и считывания блока более высокого уровня. Если блок более высокого уровня по-прежнему находится в буфере, то время выполнения дополнительной работы для одного переполнения равно сумме времен, необходимых для записи нового блока и перезаписи блока второго уровня, т. е.

$$T_{Iov} = c + s + r + btt + r + btt.$$

Поскольку перезапись блока второго уровня здесь была прервана операцией записи блока первого уровня, то может возникнуть необходимость в дополнительной установке головок, в результате чего для индексов, которые не располагаются полностью на одном цилиндре, величину $r + btt$ нужно будет заменить на $s + r + btt$.

Во всех a индексах, вместе взятых, используется na' элементов индекса. Если применяется алгоритм В-дерева¹⁾, то блоки индексов будут заполнены на 50—100 процентов своего объема. Число блоков индекса первого уровня в одном индексе равно $N1IB$. В некоторый момент, когда все блоки индексов заполнились и расщепились, имеется na' пустых элементов. Для каждого включения записи требуется a' новых элементов индексов. Если предположить, что распределение корректировок индексов оптимальное и равномерное, то после n включений будет $a N1IB$ расщеплений индексов, так что

$$Pov = \frac{a}{n} N1IB = \frac{a'}{B/(V + P)}.$$

¹⁾ Алгоритм В-дерева — разновидность метода двоичного поиска (дихотомии). — Прим. ред.

Добавляя полученные выше слагаемые для корректировки индекса низшего уровня, получаем

$$T_I = T'_I + \text{Pov}T_{Iov} = \\ = (1 + a')(s + 3r + btt) + a' \frac{V+P}{B} (c + s + 2r + 2btt).$$

Поскольку величина $y = B/(V + P)$ обычно является достаточно большой, то слагаемыми, содержащими $1/y^2$ и т. д., можно пренебречь. Поэтому расщепление на втором и более высоких уровнях можно не принимать в расчет. Главные индексы обычно бывают одинаково слабо заполненными, а их размеры могут быть выбраны с помощью полученного выше соотношения и с учетом максимального размера файла.

Обновление записи в индексированном файле. Процедура обновления записи в индексированном файле состоит из поиска, после которого проводится изменение индексов, соответствующих измененным значениям полей записи. При минимальном обновлении (только одного поля) изменяется только одна запись данных и только один индекс. В тех случаях, когда новое значение поля настолько отличается от старого, что старый и новый элементы индекса находятся в различных блоках, для обновления необходимо считать и переписать два блока индекса (старый и новый), т. е.

$$T_U = T_F + 2r + 2(s + 3r + btt).$$

Если как старое, так и новое значения находятся в одном и том же блоке индекса, то отпадает необходимость в выполнении одной выборки и одной перезаписи, так что

$$T_U = T_F + s + 5r + btt.$$

Трудно предсказать, какой именно из этих двух случаев возникнет; это зависит от характера изменения атрибута. Например, если типом атрибута является вес человека, то можно ожидать, что преобладающим будет более простой случай. Кроме того, если в результате обновления неопределенное значение становится определенным, то необходимо только создать новый элемент индекса. Будем предполагать, что изменения индекса произвольно распределены по его блокам. Если средняя плотность заполнения составляет 75 процентов, то число блоков первичного индекса для одного атрибута равно

$$b_i = \frac{n}{y} \frac{a'}{a} \frac{1,0}{0,75}.$$

Тогда если изменения данных произвольны, то вероятность того, что потребуется другой блок индекса, равна $P_i = (b_i -$

— $1)/b_i$, так что

$$T_U = T_F + s + 5r + btt + P_i(s + 3r + btt).$$

Если размеры файлов больше, а атрибуты часто повторяются, то значение P_i близко к 1, так что

$$T_U = T_F + 2s + 8r + 2btt.$$

При поиске в индексе вновь можно избежать одной установки головок.

Если изменяется несколько (a_U) значений атрибутов, то необходимо найти и изменить столько же блоков индексов. В этом случае

$$T_U = T_F + 2r + 2a_U(s + 3r + btt).$$

Блоки индексов здесь также могут переполняться. В этом случае справедливы те же соображения, которые приводились при рассмотрении процедуры включения записи.

Случай равномерного распределения, предполагаемый в моделях включения и обновления записей, является наиболее благоприятным, если период времени достаточно длительный. Однако обычно дело обстоит таким образом, что сначала чаще других обновляются одни атрибуты, затем другие и т. д., причем такая смена происходит периодически. Такая неравномерность может повлиять на время ответа системы. Для оценки дополнительного слагаемого, соответствующего случаю неравномерного распределения, необходимы статистические данные о характере обновления. Такими данными мы здесь не располагаем и поэтому не будем проводить соответствующих оценок. Если известны частоты изменения атрибутов и заданы области изменения значений, то с помощью описанных выше приемов можно получить более точную оценку затрат на обновление.

Полное считывание индексированного файла. Полностью индексированная организация осложняет полный поиск. При необходимости такой поиск осуществляется с использованием информации о распределении памяти или путем упорядоченного считывания записей файла с помощью некоторого полного индекса. Полный индекс организуется в тех случаях, когда требуется, чтобы в каждой записи существовал идентифицирующий ее элемент данных (атрибут записи). При непосредственном использовании такого индекса затраты на поиск составят

$$T_X = nT_F.$$

Если просматривать указатели распределения памяти, то на каждый блок потребуется только одна установка головок, так что в этом случае время считывания для последовательно раз-

мешенного файла было бы равно времени, необходимому для считывания последовательного файла. Если блоки распределяются произвольно или находятся из распределения памяти случайным образом, то

$$T_x = n \frac{R}{B} \left(s + r + \frac{B}{l'} \right).$$

Порядок появления записей в этом случае непредсказуем. Из-за обработки пустых областей, образованных в результате удаления записей, возможно снижение фактической эффективности поиска.

Реорганизация индексированного файла. Для индексированных файлов необходимость в проведении периодической реорганизации не столь велика, как для ранее рассмотренных организаций файлов. Для некоторых типов индексированных файлов реорганизацию вообще можно не проводить. Реорганизация затрагивает не базу данных, а лишь определенный индекс. Поэтому можно выполнять поочередную реорганизацию каждого индекса. Индекс в среднем состоит из

$$b_i = (n + o') \frac{a'}{a} \frac{V + P}{B} + bhi$$

блоков, где bhi — число блоков, принадлежащих индексам более высокого уровня. Тогда время, необходимое для реорганизации всех a индексов, равно

$$T_Y = 2ab_i(s + r + bitt).$$

Новые блоки индексов, вообще говоря, помещаются в новые области на диске, так что соотношения, содержащие T_{RW} , не справедливы. Если для старого и нового индексов использовать отдельные устройства, то среднее время установки можно уменьшить.

19.15. ПРЯМОЙ ФАЙЛ

Прямой файл позволяет наиболее тесно связать аргумент поиска, используемый для выборки записи, с физическими возможностями устройств прямого доступа. Смысл прямого доступа к файлу заключается в том, что заранее известно, где находятся данные на устройстве (рис. 19.26). Впервые файлы с прямым доступом нашли применение в электромеханических счетных машинах, в которых некоторое отперфорированное на карте число использовалось для того, чтобы определить, где хранится остаток информации, содержащейся на карте. Метод прямого доступа обеспечивает относительно быстрый поиск записи, поскольку он позволяет избежать выполнения промежу-

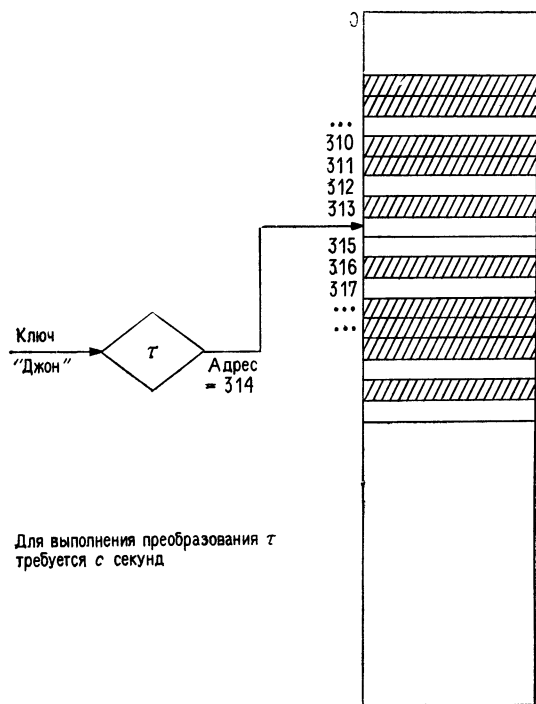


Рис. 19.26. Доступ к записи в прямом файле.

точных операций над файлом. Но так же, как при использовании последовательных и индексно-последовательных файлов, поиск ведется только по одному ключевому атрибуту. При этом не требуется, чтобы данные были связаны с какими-либо предшествующими записями. Вместо поиска в индексных таблицах, с помощью которых определяется местоположение записи в индексированном файле, проводится вычисление τ , целью которого является определение адреса записи.

Структура прямых файлов и работа с ними

В простейшей реализации метода прямого доступа каждой записи данных приписывается идентификационный номер, который определяет адреса диска, дорожки и записи файла. Так, если служащему Джо приписан номер 25-4-7, то это говорит нам о том, что соответствующую запись в платежной ведомости следует искать на диске с номером 25, дорожке 4 как запись 7. Мы перечислим ряд проблем, связанных с использо-

ванием адресов файла для идентификации записей данных, и, минуя рассмотрение промежуточных решений, остановимся на наиболее распространенной в настоящее время методологии.

Адреса файла. На устройстве прямого доступа адреса обычно не являются смежными, т. е. они могут изменяться от 0 до 200, а затем от 1000 до 1200 и т. д. Это вызывает большие трудности у работников учреждений, плохо разбирающихся в ЭВМ, поскольку от них требуется знание аппаратных средств.

Если устройство устаревает и заменяется на новое, то всем данным присваиваются новые идентификационные номера.

Идентификационные номера могут понадобиться более чем в одном файле, в результате чего пользователю придется иметь дело с множествами различных чисел.

Для того чтобы после удаления записи заново использовать занимаемую ею область, необходимо присвоить новый идентификационный номер, что является причиной ошибок в тех случаях, когда старые и новые данные обрабатываются вместе.

Для идентификации записей нельзя ограничиваться использованием плотных множеств чисел. В тех случаях, когда смысловое значение имеют группы последовательных цифр, естественными ключами являются имена, страховые или инвентарные номера. Вообще говоря, число людей или предметов, на которое можно ссылаться с помощью этих ключей, намного больше, чем число записей, которые должны храниться в файле. Другими словами, область ключа намного больше, чем адресное пространство файла.

Преобразование ключа в адрес. Для решения проблем, связанных с применением адресов файла, используется вычислительная процедура, преобразующая существующие данные идентификации, т. е. значения ключевых атрибутов, в адресное пространство на диске. Эта процедура может быть написана таким образом, что будут решены проблемы, связанные с изменениями данных и многократным использованием ключа. Используемые при этом методы называются **преобразованиями ключа в адрес**. Мы рассмотрим два таких метода: **детерминированные методы**, позволяющие преобразовывать поля идентификации в уникальные адреса, и **вероятностные методы**, с помощью которых ключи преобразуются в адреса, являющиеся уникальными в такой степени, в какой это возможно. На рис. 19.26 схематически изображена процедура доступа к прямому файлу, выполняющая включение новой записи с ключом "Джон". С помощью алгоритма τ преобразования ключа в адрес, примененного к строке "Джон", относительный адрес записи был вычислен равным "314". Аналогичный алгоритм преоб-

разования ключа в адрес применяется к аргументу поиска в тех случаях, когда необходимо осуществить выбор записи.

При **детерминированном методе** множество всех значений ключа исследуется с целью найти алгоритм преобразования. А затем, следуя этому алгоритму, вычисляется уникальный адрес на диске. Если число элементов файла превосходит несколько десятков, то разработка алгоритмов преобразования ключа в уникальный адрес представляет некоторую трудность. Добавление новых элементов к файлу также вызывает затруднения, поскольку алгоритм преобразования зависит от распределения исходных ключей. Из сказанного выше следует, что детерминированный метод удобен только при обработке статических файлов. Заметим, что если вместо вычислительного алгоритма использовать таблицу, то задача преобразования упрощается; мы вновь получаем индексированный файл! В дальнейшем мы не будем рассматривать детерминированный прямой доступ.

Вероятностные методы позволяют преобразовывать значения идентификационных номеров в числовые адреса, расположенные в адресном пространстве файла. Для этого используется специальная процедура. Желательно, чтобы распределение адресов было равномерным, поскольку в этом случае каждому участку будет соответствовать равная доля ключей.

При получении адреса из ключа может случиться так, что разные ключи преобразуются в один и тот же адрес. Это приведет к коллизиям. Позже в этом разделе мы покажем, как поступать в данном случае. Можно попытаться определить вероятностное преобразование таким образом, чтобы сохранить порядок расположения записей. Однако в большинстве случаев при определении преобразования преследуется единственная цель: максимизировать степень уникальности результирующего адреса. Первый из указанных классов преобразований называется **сохраняющим последовательность**, остальные алгоритмы называются **случайными преобразованиями ключа в адрес**, или **методами перемешивания (хеширования)**. Родословное дерево методов преобразования ключа в адрес изображено на рис. 19.27. Один простой метод описан в примере 19.13.

Пример 19.13. Преобразование ключа в адрес. В преобразовании рандомизации, выполняемом над файлом сведений о служащих, в качестве ключа используется номер страхового полиса. Предполагается, что значения цифр низкого порядка в этих номерах распределены равномерно. Следовательно, вероятность того, что из этих цифр будет образован уникальный номер для каждого служащего, достаточно большая. Если требуется вести учет 500 служащих, то значение ключа можно раз-

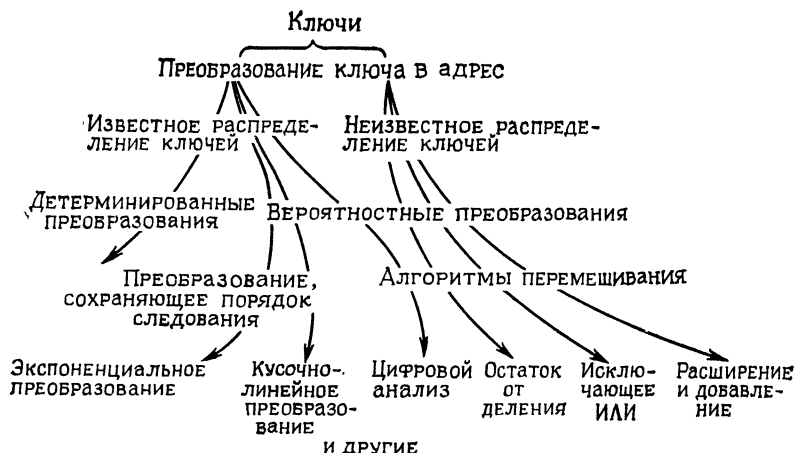


Рис. 19.27. Типы преобразований ключа в адрес.

делить на 500, а полученный остаток будет лежать в диапазоне от 0 до 499. Очевидно, что в этом случае могут быть получены одинаковые адреса. Например,

Ал	322-45-6678	адрес	178
Джо	123-45-6392	адрес	392
Мэри	036-23-0373	адрес	373
Пит	901-23-4892	адрес	392

Здесь записям “Джо” и “Пит” будет назначен один и тот же номер, 392, поэтому если эти записи непосредственно поместить в файл, то возникнет коллизия. Вопросы, касающиеся коллизий, рассматриваются ниже.

Область ключей и адресное пространство. Значения ключа могут изменяться в широком диапазоне, ограниченном только максимальным размером V поля ключа. Для числового ключа количество допустимых значений составляет 10^V , для простого алфавитного ключа — 26^V ¹⁾. Для номера страхового полиса размер области ключей составляет 999 999 999. Соответствующая область в файле будет значительно меньше, поскольку большая часть пользователей не зарегистрирована в Управлении по социальному страхованию. Размер адресного пространства (объем) может быть выражен числом записей m . Фактическое число записей, помещенных в файл, не может превышать этот объем. Следовательно,

$$\text{основание}^V \gg m \geq n.$$

¹⁾ Если используются только буквы латинского алфавита (их 26). — *Прим. ред.*

Поскольку во многих алгоритмах преобразования ключа в адрес используются числовые значения ключей, то может возникнуть необходимость представить алфавитные значения ключей в числовой форме. Если 26 используемым буквам приписать значения (**lettervalue**) от 0 до 25, то с помощью следующего полиномиального преобразования для заданного слова может быть получено компактное представление без потери информации ¹⁾:

```
/*Преобразование букв в целые числа*/  
numeric_value = 0;  
DO i = 1 TO number_of_letters;  
    numeric_value = numeric_value*26 + lettervalue(letter(i));  
END;
```

Для обработки чисел и пробелов эту программу необходимо соответствующим образом модифицировать. Кроме того, следует учесть случай переполнения машинного слова.

Для упрощения этого процесса, предположим, что можно использовать относительную адресацию. Такая возможность обеспечивается алгоритмом, преобразующим последовательные номера записей, лежащие в диапазоне от 0 до $m-1$, в пространство физических адресов на диске или на любом другом устройстве, используемом для хранения данных. В этом случае нужно только из исходного ключа сгенерировать целое число, лежащее в диапазоне от 0 до $m-1$. Часто требуется вместо адресов определенных областей записей вычислять адреса блоков. (В блок можно поместить несколько записей.) Вопросы, связанные с использованием бакетов, будут рассматриваться позже.

Методы, учитывающие статистическое распределение значений ключей, основаны на использовании по крайней мере приближенных сведений об ожидаемых ключах. Преимущества методов, зависящих от распределения, проявляются при использовании **открытой адресации**. Их эффективность зависит от размеров бакетов, плотности файлов и преобразования. Если бакеты небольшие и выбран хороший алгоритм, учитывающий закон распределения, то по сравнению с рандомизацией может быть получено значительное улучшение. С другой стороны, на разработчика методов преобразования, учитывающих закон распределения, ложится большая ответственность, так как при их использовании изменение распределения ключей может при-

¹⁾ Здесь **number — of — letters** — число букв в слове, **letter(i)** — *i*-я по порядку буква в слове, а **numeric — value** — его числовое представление.—
Прим. перев.

вести к значительно большему числу коллизий, чем при использовании метода рандомизации. Преимущество некоторых методов преобразования ключа в адрес, учитывающих закон распределения, заключается в том, что они позволяют сохранить порядок следования записей. Это достигается путем увеличения адресов в соответствии с увеличением ключей. Такое преобразование ключа в адрес называется **преобразованием, сохраняющим порядок следования**. Оно позволяет осуществлять упорядоченный доступ. В остальных случаях при использовании основных прямых файлов возможность упорядоченного доступа отсутствует.

Обзор методов. Мы рассмотрим только два метода, зависящие от распределения (цифровой анализ и преобразование, сохраняющее порядок следования). Предлагаемые методы перемешивания (остаток от деления, исключающее ИЛИ, а также расширение и добавление) основаны на использовании случайных свойств цифр ключа. Использование таких операций, как арифметическое умножение и сложение, при выполнении которых обычно образуются нормально распределенные случайные значения, нежелательно при перемешивании.

Метод цифрового анализа основан на использовании распределений цифр, составляющих ключи. С помощью выборочной совокупности записей, которые должны храниться в файле, для каждой из последовательных цифровых позиций в ключах выводится оценка или составляется таблица. В примере 19.13 предполагалось, что были исследованы три цифровые позиции младших разрядов в номере страхового полиса и что соответствующие значения оказались равномерно распределенными.

При составлении таблиц определяются частоты появления нулей, единиц, двоек и т. д. Цифровые позиции, в которых распределения оказались в достаточной степени равномерными, являются кандидатами для использования их в адресе файла. Для того чтобы составить полный адрес, необходимо найти достаточное число таких цифровых позиций. Можно также исследовать комбинации других цифровых позиций (взяв соответствующие значения по модулю 10 или как-нибудь иначе).

Аналогичные проверки могут быть выполнены для алфавитных ключей. В этом случае набор из 26 возможных букв может быть разбит на 10 групп, с тем чтобы получить цифровые значения, или на группы различных размеров, с тем чтобы получить множители, отличные от 10, которые могут быть использованы для определения номера записи.

Функция преобразования, сохраняющая порядок следования, может быть получена путем простой инверсии найденного рас-

пределения ключей. Адреса генерируются таким образом, чтобы сохранить порядок следования по отношению к исходному ключу. При использовании **кусочно-линейного преобразования** наблюдаемое преобразование аппроксимируется (автоматизированно или вручную) отрезками прямой. Затем полученное приближение используется, чтобы распределить адреса в виде их дополнений.

Для получения требуемого адреса можно также использовать **остаток от деления** ключа на делитель, равный числу m отведенных областей записей. Операция деления в каком-то смысле аналогична использованию цифр младших разрядов. Однако в тех случаях, когда делитель не кратен основанию (10 в примере 19.13) системы счисления, в которой записывается ключ (или системы счисления, используемой в ЭВМ), дополнительно используется информация, содержащаяся в цифрах высокого порядка. Эта дополнительная информация увеличивает число вариантов и, следовательно, сгенерированные адреса располагаются более равномерно. В качестве делителей обычно используются достаточно большие простые числа, так как они позволяют получить хорошее распределение значений частных даже в тех случаях, когда для частей ключа этого достичь не удастся. Вообще говоря, подходящими являются делители, которые не содержат маленьких простых чисел (≤ 19). Исследования показали, что метод деления лучше остальных позволяет сохранять исходно существующую равномерность распределений, особенно равномерность, обусловленную последовательностями цифр низкого порядка в присвоенных идентификационных номерах. Этот метод не позволяет сохранить порядок следования записей. При использовании метода деления следует учесть ограниченные возможности самой операции деления. Часто поле ключа, которое должно быть преобразовано, превосходит наибольший допустимый размер делимого. В некоторых ЭВМ отсутствуют команды деления с остатком. В этом случае остаток может быть вычислен с помощью выражения

$$\text{адрес} = \text{ключ} - \text{FLOOR}(\text{ключ}/m) * m.$$

Оператор **FLOOR** используется для того, чтобы “сообразительный” оптимизатор не сгенерировал для каждого ключа адрес $= 0$, что привело бы к максимальному числу коллизий ($n - 1$). Необходимость в выполнении этих операций отпадает, если деление заменить умножением на число, обратное m . Это означает, что в операции используются двоичные дроби и результат должен быть получен с помощью сдвига. В тех случаях, когда длины ключей чрезмерно большие, этому преобразованию может предшествовать выполнение операции исключающего ИЛИ.

Операция исключающего ИЛИ имеется в большинстве ЭВМ с двоичной арифметикой или может быть выполнена с помощью следующих операторов:

```

DECLARE (address, keypart1, keypart2) BIT(19);
. . .
x_or: PROCEDURE(arg1, arg2); DECLARE(arg1, arg2) BIT(*);
      RETURN((arg1, arg2) ( $\neg$ (arg1, arg2)));
      END x_or;
. . .
address = x_or(keypart1, keypart2);

```

Как видно из этого примера, ключ делится на сегменты, которые соответствуют требуемому размеру адреса. В результате выполнения этой операции образуются случайные числа для случайных входных данных в двоичной форме. Эти сегменты могут содержать алфавитные знаки. Например, **x_or** ('MA', 'RA') равно **x_or** ('RA', 'MA'), так что "MARA" будет конфликтовать с "RAMA". Следует проявлять внимание в тех случаях, когда двоичное представление десятичных цифр или символьных знаков является таковым, что в определенных позициях всегда будут находиться нули или единицы. Для решения этих проблем можно выбрать размеры сегментов такими, чтобы они не имели общего делителя по отношению к размерам символов или слов. Эта операция обычно является одной из самых быстрых аппаратно реализованных операций.

Метод расширения и добавления ключевой цифровой строки позволяет использовать для вычисления адреса более короткую строку. Значения соответствующих битов в дополнительных сегментах противоположны друг другу. Эта операция аппаратно реализована в некоторых больших современных ЭВМ HONEYWELL.

Известны также и другие методы, описание которых можно найти в литературе.

Резюме. Как видно, существует большое число методов преобразования ключа в адрес. Необходимо помнить, что доступ к произвольному прямому файлу может быть осуществлен только с помощью одного конкретного значения атрибута, а не путем упорядоченного считывания или указания диапазона значений. Упорядоченный доступ может быть осуществлен только путем перебора всех возможных значений ключа. Если множество реальных значений ключей не очень плотное, то результатом подавляющего большинства выборок будет возникновение ситуации "запись-не-найдена". На практике такой способ не применяется, Другое ограничение в вычислении адресов запи-

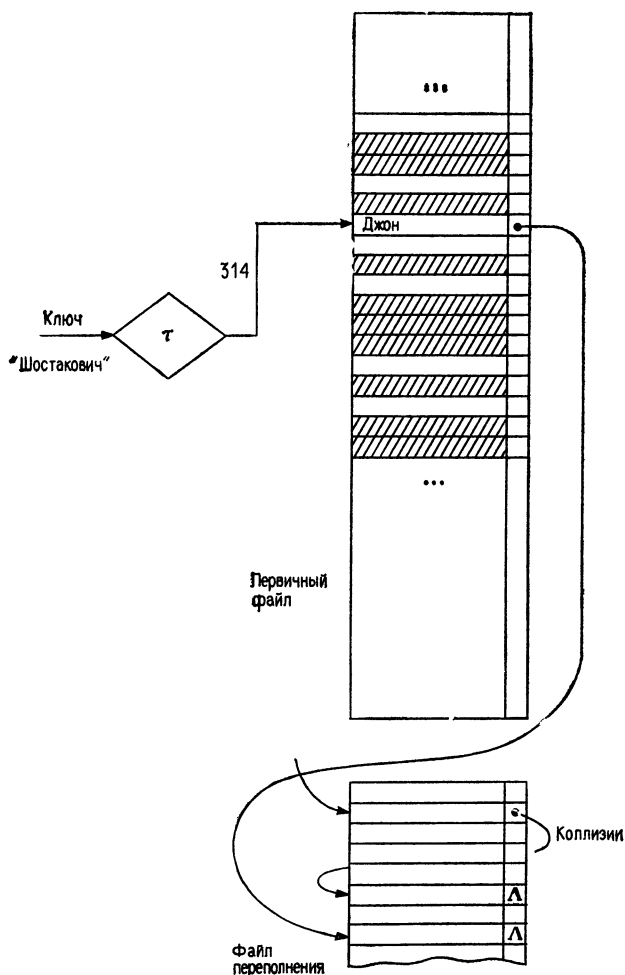


Рис. 19.28. Прямой файл с коллизией.

сей заключается в том, что записи данных должны иметь фиксированную длину.

Коллизии (рис. 19.28). При использовании любой вероятностной процедуры возможны случаи, когда нескольким различным ключам ставится в соответствие один и тот же адрес. В этом разделе будут рассмотрены пути разрешения коллизий.

Размер большей из указанных на рис. 19.28 областей адресов записей соответствует числу записей в файле. Размер нижней области определяется с учетом вероятности коллизии. Ниже

будет дана оценка для этой вероятности. Сейчас отметим, что при использовании области, размер которой на 50 процентов превосходит минимально допустимый размер ($m/n = 1,5$), затраты на разрешение коллизий становятся относительно небольшими: ожидаемое число коллизий составляет менее чем 34 процента от числа доступов к файлу.

Для того чтобы установить возникновение коллизии, следует сравнить значение ключа, найденное по вычисленному адресу, с заданным значением ключа. Предположим, что считается файл, построенный на основе примера 19.13. Для записи “Пит” с номером страхового полиса 901-23-4892 с помощью алгоритма был вычислен номер 392. Однако это не означает, что запись “Пит” обязательно будет помещена в область, соответствующую номеру 392. Необходимо проверить содержимое самой области записи. Эта область может быть пустой, может содержать более старую запись “Пит” или содержать конфликтующую с ней запись, например “Джо”. Для того чтобы определить, какой случай имеет место, следует сопоставить ключевые поля. Если ключевое поле в файле не является пустым (**NULL**) и новое и старое ключевые поля не совпадают, то имеет место коллизия. Аналогичные действия выполняются с аргументом поиска во время выборки записи. Возможны следующие случаи: запись не найдена, запись найдена, продолжить поиск, исследуя результаты коллизий. Мы рассмотрим методы разрешения коллизий с точки зрения добавления новой записи к прямому файлу. Существуют три метода разрешения коллизии, которые иногда применяются в комбинации друг с другом. В двух из них (линейный поиск и повторная рандомизация) для области переполнения используется основной файл. Эти два метода называются методами **открытой адресации**. В третьем методе используется отдельный файл переполнения.

Линейный поиск. Поиск неиспользованной области для новой записи может проводиться путем последовательного просмотра всех позиций, отведенных для записей.

Поиск в бакете. Один из важных подходов к разрешению коллизий заключается в группировании всех записей, претендующих в файле на одну позицию, в один блок, называемый бакетом (рис. 19.29). При доступе к записям, расположенным внутри бакета время расходуется только на вычисления. Время доступа к диску следует учитывать только в тех случаях, когда заполнен сам бакет. Если размер бакета совпадает с размером физического блока, то допустимое число *ovrb* коллизий в бакете равно

$$ovrb = \left(1 - \frac{n}{m}\right) \left\lfloor \frac{B}{R} \right\rfloor$$

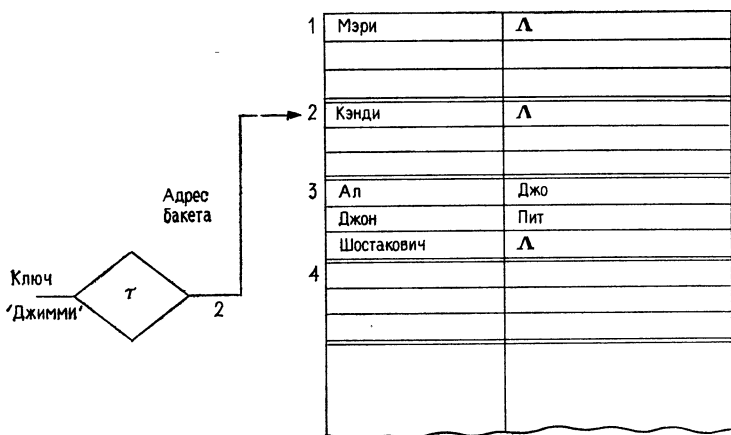


Рис. 19.29. Бакеты.

(т. е. доля свободного пространства умножается на число записей в физическом блоке).

Поиск в файле. В случае переполнения поиск может быть продолжен в следующем по порядку блоке. При этом затрат на установку головок для доступа к новым областям не возникает, однако записи переполнения группируются в одном месте. На практике этот метод часто используется как для файлов, так и для оперативной памяти. Поскольку число элементов меньше числа доступных областей записей ($n < m$), свободное пространство для записи в конце концов будет найдено. Это можно определить заранее, и поэтому дополнительного условия завершения процедуры здесь не требуется. Проблема, связанная с кластеризацией, заключается в том, что в тех случаях, когда область памяти заполнена плотно, для выборок необходимо выполнять большое число шагов поиска. Включение дополнительных записей приводит к увеличению размера плотной области. Соответствующий пример показан на рис. 19.34. Здесь показано, что кластеризация приводит к большим затратам даже в тех случаях, когда память умеренно загружена.

Повторная рандомизация. Для того чтобы не допустить образования кластеров или рассредоточить их, используются методы, отличные от метода поиска свободного пространства в следующем бакете. Новый адрес записи в том же пространстве может быть определен путем повторенного вычисления с другими данными идентификации или с помощью другого алгоритма. Применение этого метода к таким устройствам, как диски,

приводит к большим накладным расходам, поскольку при этом обычно возникает необходимость в дополнительной установке головок. Кроме того, могут возникнуть повторные коллизии и возможно, что эту процедуру придется выполнить несколько раз. В действительности нельзя гарантировать, что этот процесс всегда позволит найти свободное пространство. Обычно подобные методы используются для оперативной памяти, хотя в системах со страничной организацией памяти их, вероятно, следует применять с особой осторожностью. Такой подход находит применение в тех случаях, когда ассоциативная память реализуется с помощью программного обеспечения. В этом методе следует предусмотреть дополнительную возможность завершения.

Вероятность выбора удачного метода рандомизации. Для того чтобы лучше понять смысл различных методов получения случайных распределений, можно воспользоваться графическим описанием соответствующих процессов. Число возможных методов преобразования n ключей в m адресов огромно. Действительно, существует m^n возможных функций. Однако только $m!/(m-n)!$ из них позволяют избежать коллизий. Очевидно, что лучше всего было бы найти одну из них. С другой стороны, возможны m вариантов полных коллизий (т. е. все записи помещаются в один и тот же участок), и в этих случаях ожидаемое число доступов для выборки с помощью цепочки равно $(n+1)/2$. Поскольку метод рандомизации выбирается без каких-либо предварительных знаний о ключах, то вероятность того, что при использовании метода, выбранного из числа возможных, не возникнет коллизий, составляет всего лишь $m!/(m-n)!/m^n$. С другой стороны, вероятность того, что будет выбран худший метод и возникнет необходимость в выполнении $1/2n$ дополнительных доступов, составляет только m/m^n . Если сам метод преобразования ключа в адрес выбирать случайным образом, то, как видно из рис. 19.30, для $m=5$, $n=4$ ожидаемое число дополнительных доступов составляет $p=0,30$. Для других значений m и n кривая выглядит аналогично. Можно получить также оценки для числа коллизий в общем случае. Однако цель этого подраздела заключается в том, чтобы показать, что выбор метода рандомизации, если он заведомо не принадлежит числу худших, не играет существенной роли. Следует отметить, что при замене ключей, т. е. при выборе нового набора данных, которые должны быть рандомизированы, взаимное расположение m^n методов на рисунке полностью изменится, однако форма распределения останется прежней.

Область переполнения. По аналогии с построением цепочек переполнения, используемых в индексно-последовательных фай-

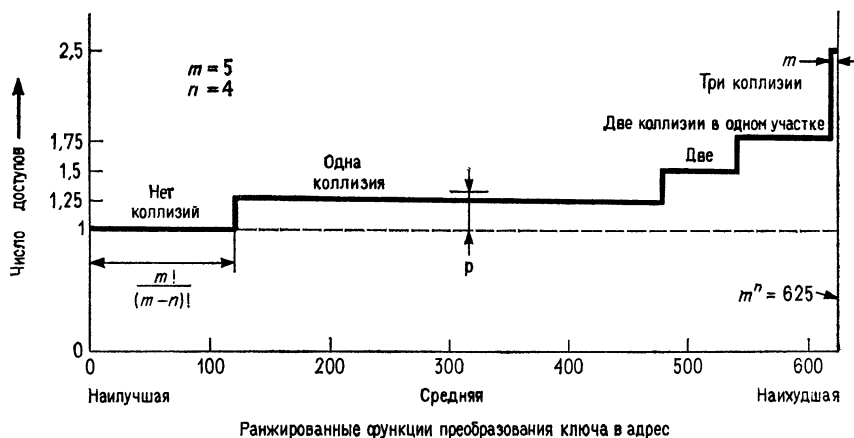


Рис. 19.30. Распределение длин выборок.

лах, все записи, вызывающие коллизии или переполнения бакетов, также можно помещать в отдельный файл со ссылкой из первичной записи. Создание такой отдельной области переполнения позволит избежать кластеризацию. При переполнении появляется необходимость в новой установке головок, однако проблемы рекурсивного поиска, как в первом методе, не возникает. Трудность заключается в том, что необходимо выделить достаточный объем памяти для области переполнения, хотя число коллизий заранее предсказать нельзя. Степень переполнения может быть существенно уменьшена с помощью начального линейного поиска в бакете.

Как и прежде, затраты, связанные с переполнением, могут быть снижены, если на каждом цилиндре выделять свою область переполнения. Для этого необходимо модифицировать преобразование ключа в адрес таким образом, чтобы использовать пропуски в первичном пространстве адресации. Если бакеты уже используются, то с помощью этого пространства можно уменьшить плотность их заполнения.

Адреса бакетов. Выше рассматривался вопрос об использовании бакетов с целью сокращения затрат на разрешение коллизий. В тех случаях, когда в одном физическом блоке хранится несколько записей и когда для получения одной записи выбирается весь физический блок, может быть использован более короткий адрес. Если в физическом блоке B/R записей, то размер адресного пространства сокращается с m до mR/B .

Использование прямых файлов

Прямые файлы часто используются для составления справочников, таблиц стоимостей, расписаний, списка имен и т. д. Прямая организация файлов особенно удобна в тех случаях, когда записи имеют небольшие размеры и фиксированную длину, когда необходим быстрый доступ и когда доступ к данным всегда осуществляется просто. Кроме того, прямые файлы часто используются в качестве вспомогательных элементов в более сложных организациях файлов.

Эффективность использования прямых файлов

С точки зрения эффективности использования прямые файлы исследованы глубже, чем другие рассмотренные выше файлы. Исходным параметром в этих оценках является число m областей записей или бакетов, имеющих для хранения n записей. Вообще говоря, n меньше, чем m . Число записей, вызывающих коллизии, будем обозначать через o . Будет исследована простая структура прямого файла с бакетами, содержащими одну-единственную запись фиксированной длины, и отдельной областью для хранения o записей, вызвавших коллизии. При необходимости даются комментарии, относящиеся к использованию прямых файлов с открытой адресацией и бакетов с несколькими записями, поскольку соответствующие методы находят широкое применение.

Размер записи в прямом файле. Как следует из предыдущих разделов, объем SF области для хранения файла равен

$$SF = m(aV + P) + o(aV + P),$$

или, в расчете на одну запись,

$$R_{\text{эффект}} = \frac{m+o}{n} (aV + P) = \frac{m+o}{n} R.$$

Выборка записи из прямого файла. Для вычисления среднего времени локализации записи в прямом файле необходимо найти число p дополнительных обращений при коллизии, поскольку ожидаемое время выборки определяется как сумма времени, требуемого для рандомизации, времени одного прямого доступа и среднего времени, расходуемого на разрешение коллизии, т. е.

$$T_F = c + s + r + btl + p(s + r + btl).$$

Для оценки величины p в рассматриваемом случае был проведен специальный анализ. Результаты анализа показали, что для прямых файлов с бакетами размера R и областями пере-

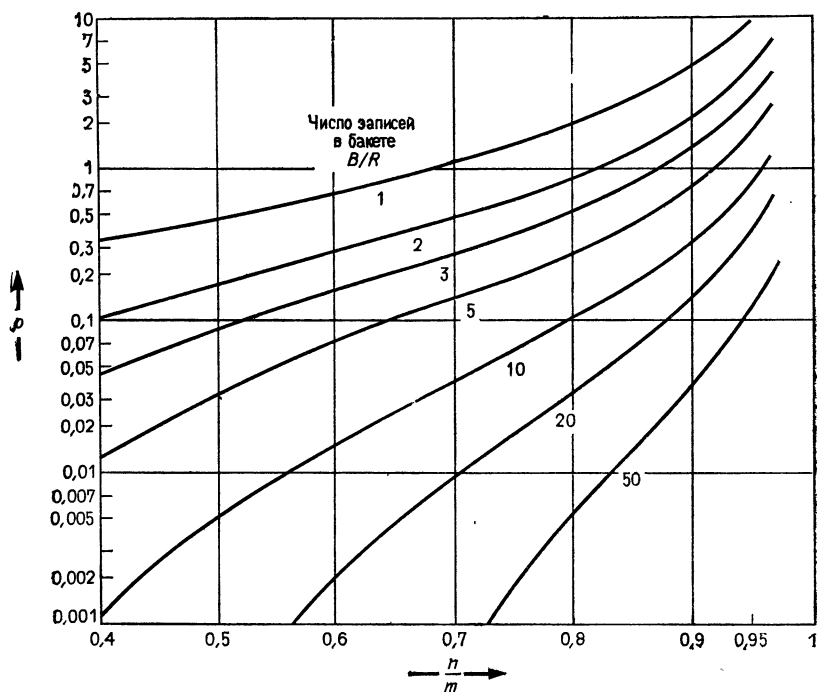


Рис. 19.31. Число дополнительных обращений для случая открытой адресации и линейного поиска.

полнения, используемыми при коллизиях, среднее значение p равно

$$p = \frac{1}{2} \frac{n}{m}.$$

Результаты численного эксперимента хорошо согласуются с этой формулой.

Влияние открытой адресации. Исследования показали также отрицательное влияние кластеризации, имеющей место при открытой адресации с линейным поиском. Затраты, возникающие из-за образования кластеров, быстро возрастают с увеличением плотности заполнения n/m (рис. 19.31). С помощью аргументов рассмотренной выше рандомизации со случайной выборкой Кнут установил, что в этом случае доля числа дополнительных доступов становится равной

$$p = \frac{1}{2} \frac{n}{m - n}.$$

Ввиду сложности случая, когда размеры бакетов (в записях) $B/R > 1$, соответствующие результаты представляются здесь в графической форме. Указанные значения вполне соответствуют результатам численного эксперимента, основанного на использовании метода остатка от деления. Для плохо распределенных ключей эти результаты подтверждаются в тех случаях, когда размеры бакетов $B/R \geq 5$. Следует отметить, что если используется открытая адресация, то области переполнения не требуется, так что эквивалентное значение m может быть соответствующим образом увеличено. Однако это не возместит потерь, вызванных увеличением числа коллизий.

Пример 19.14. Согласно сформулированному выше предположению в бакете содержится только одна область записи и, кроме того, существует отдельная область переполнения. Если размер первичной области на 50 процентов больше, чем требуется для самого файла, то число дополнительных обращений равно $p = 0,5n/(1,5n) = 0,3333$, т. е. при полном заполнении файла ожидается 33 процента дополнительных доступов из-за коллизий.

Если ту же первичную область размера m и бакет размера R использовать в случае открытой адресации, то значение p становится равным

$$p = \frac{1}{2} \frac{1}{1,5 - 1} = 1,0,$$

указывая на 100 процентов дополнительных доступов. Если размер записи $R = 200$, а размер блока $B = 2000$, то размер бакета (в записях) может быть сделан равным 10; при этом возрастут только накладные расходы на вычисления. Значение p , получаемое из рис. 19.31, теперь равно

$$p = 0,03,$$

так что доступ к другому блоку потребуется только для 3 процентов выборок.

Рассмотрим случай линейного поиска. Если решение о считывании следующего блока может быть принято немедленно, то задержки, равной времени оборота диска, не возникает. Однако в общем случае, когда решение о считывании принимается в зависимости от содержания предыдущего блока, для считывания следующего блока требуется один полный оборот диска, так что время доступа к записи переполнения равно $2r + btt$. С помощью дополнительного блока (рис. 19.32) время доступа может быть сокращено до $2btt$; при этом для анализа записей, расположенных внутри бакета, отводится время $c < btt$. При использовании этого метода не требуется какой-либо перенуме-

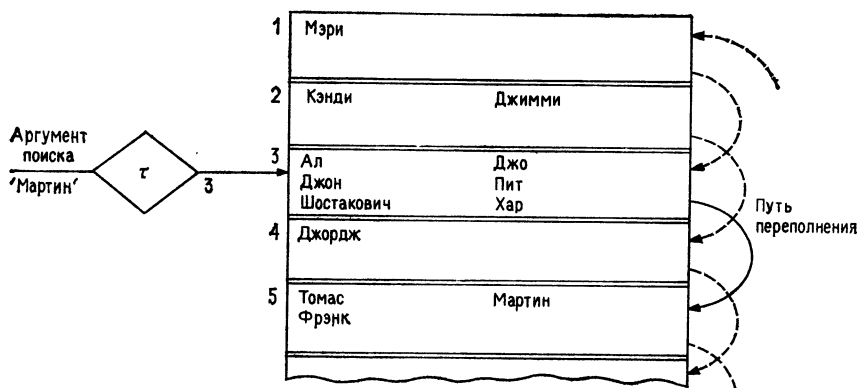


Рис. 19.32. Доступ с дополнительным бакетом переполнения.

рации блоков, поскольку место для записей переполнения выбирается произвольным образом.

Таким образом,

$$T_F = c + s + r + btt + pt \text{ переполнение.}$$

Значения для p были даны выше для случаев, когда используются отдельные области переполнения, а также когда применяется линейный поиск (для различных размеров бакетов). Значения для $t_{\text{переполнение}}$ следующие:

$s + r + btt$	для отдельной области переполнения
$r + btt$	для отдельных областей переполнения на одном и том же цилиндре
$2r + btt$	для линейного поиска, последовательных блоков
$2btt$	для линейного поиска, дополнительных блоков

Время c зависит от сложности алгоритма перемешивания и размера бакета.

Если характер использования файла постоянный и известен заранее, то полезным может оказаться такое упорядочение записей, что те из них, которые требуются чаще, располагаются перед цепочками, образованными с помощью адресов для конфликтующих записей, а записи, обращение к которым требуется реже, помещаются в конец этих цепочек.

Выборка записи, не существующей в файле. При попытке осуществить выборку несуществующей записи поиск проводится до тех пор, пока не будет найден пустой участок. Соответствующее время вычисляется при получении оценки для времени включения записи (T_{lov}), так как для локализации пустого участка необходимо проверить такое же число участков. Время

выборки, выполняемой с помощью просмотра цепочки, определяется длиной этой цепочки. Одним из путей уменьшения затрат на выборку несуществующих записей при открытой адресации является использование цепочек указателей. Другой путь заключается в хранении записей цепочек переполнения в упорядоченной последовательности, как показано на рис. 19.18. Помимо того что осуществляется доступ к каждой записи цепочки и проверяется, совпадает ли ее ключ с аргументом выборки, ключ хранимой записи преобразуется также в ее адрес, который сравнивается с исходным адресом выборки, с тем чтобы определить, является ли эта запись элементом данной цепочки. Поиск несуществующей записи может быть завершён в том случае, когда значение ключа записи, являющейся элементом цепочки, больше, чем значение аргумента выборки. Кроме того, необходимо проверять, содержит ли рассматриваемая запись цепочки метку конца цепочки. Эта метка приписывается последней записи цепочки и корректируется при включении новых записей в файл. Для каждого из указанных методов величина p (вероятность того, что записи в файле не существует) в полученной выше оценке для T_F заменится на $2p$ или меньшее значение.

Получение следующей записи из прямого файла. В тех случаях, когда преобразование ключа в адрес осуществляется с помощью рандомизации, понятия упорядоченного доступа для прямого файла не существует. Если ключ следующей записи известен, то она может быть найдена с помощью той же процедуры, которая использовалась для нахождения произвольной записи. В этом случае

$$T_N = T_F.$$

Если ключ следующей записи не известен, то для ее выборки не существует никакого практического метода. Для случая линейных преобразований ключа в адрес, которые сохраняют порядок следования записей, следующая запись может быть найдена путем последовательного чтения с пропуском каждого неиспользованного пространства. Время получения следующей записи может быть оценено с помощью процедуры, используемой для индексно-последовательного файла, если имеется отдельная область переполнения, или с учетом величины p (вероятности того, что записи в файле не существует), если используется линейный поиск.

Включение записи в прямой файл. Процесс включения записи в файл был рассмотрен выше, при описании структуры прямого файла. Необходимо осуществить выборку участка, адрес которого был вычислен, и определить, содержит ли эта область

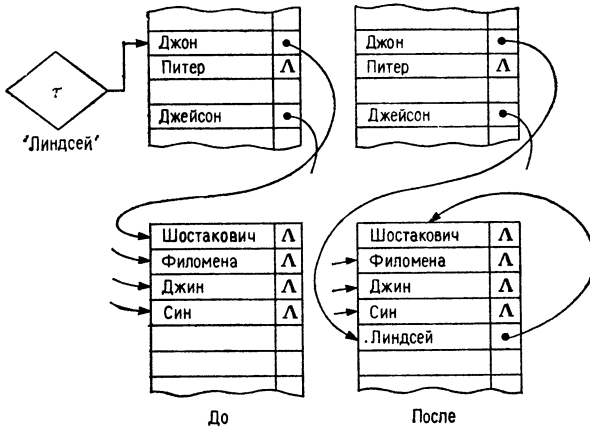


Рис. 19.33. Запись переполнения, связанная как второй элемент цепочки.

записи другую запись с другим ключом. Если бакет состоит из одной записи и при переполнении используется сцепление, то

$$Plu = 1 - e^{-n/m}.$$

Если используется открытая адресация, то

$$Plu = \frac{n}{m}.$$

Затраты на включение записи можно представить в виде суммы ожидаемых затрат для случая, когда первичный участок является пустым, и затрат для случая, когда первичный участок заполнен и вызывается процедура обработки переполнения, т. е.

$$T_l = T_{lpr} + T_{lov}.$$

Если участок пустой, то запись переписывается в него, так что в этом случае

$$T_{lpr} = (1 - Plu)(s + r + btt + 2r).$$

Если участок заполнен, то следует воспользоваться одной из рассмотренных выше процедур обработки переполнения.

В тех случаях, когда используется **отдельная область** переполнения, с помощью простейшей процедуры новую запись следует добавить к цепочке в качестве ее второго элемента (см. рис. 19.33). Здесь новая запись помещается в область переполнения со значением указателя, полученным из первичной записи, а первичная запись переписывается с адресом последней записи переполнения, т. е.

$$T_{lov} = Plu(s + r + btt + 2r).$$

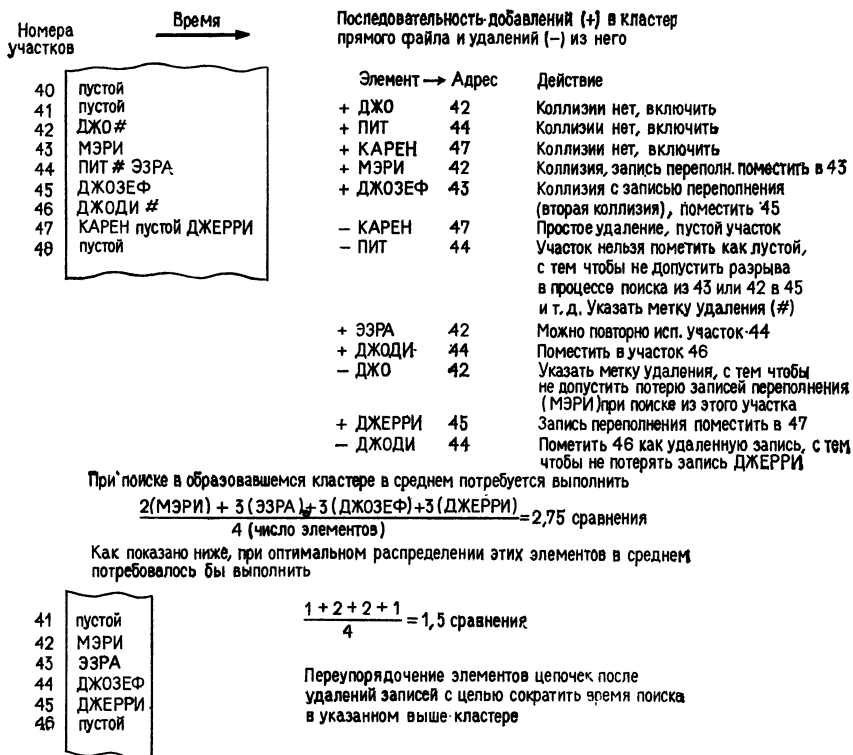


Рис. 19.34. Операции над файлом с открытым рехешированием.

Если требуется включить запись в цепочку, сохраняя возрастающую последовательность ключей, то цепочку следует просматривать до тех пор, пока не встретится запись с большим ключом. (Необходимость в выполнении этой процедуры возникает также в тех случаях, когда для того, чтобы не допустить дублирования записей с одним и тем же ключом, цепочка просматривается до включения записи.) Соответствующее время равно времени выборки. Затем переписывается предшествующая запись (из неизвестной позиции) и включается новая запись, так что

$$T_{lov} = Plu(T_F + s + r + btt + s + r + btt).$$

В тех случаях, когда используется открытая адресация, для нахождения свободного пространства необходимо просмотреть всю последовательность и соответствующий кластер (рис. 19.34). Автору не известно ни одной удовлетворительной формулировки

этой проблемы для линейного поиска. Начальной оценкой является удвоенное время выборки, так что в случае коллизии, суммируя T_{ipr} и T_{lov} и подставляя в полученное выражение вероятность коллизии для открытой адресации, получаем

$$T_I = \frac{2n}{m} T_F + s + 3r + btt.$$

Кнудом было получено, что при использовании метода повторной рандомизации $T_I \approx (m/(m-n))(s+r+btt)$.

Обновление записи в прямом файле. Процесс обновления записи состоит из поиска записи и переписи ее на первоначальный участок, так что

$$T_U = T_F + 2r.$$

В тех случаях, когда изменяется ключ, запись следует удалить и записать заново.

Необходимо сделать ряд замечаний, касающихся удаления записи. Записи переполнения располагаются в цепочке. Необходимо либо удалить запись из цепочки переполнения, с тем чтобы заполнить этот участок, либо указать, что, несмотря на то что эта запись удалена, она все еще содержит соответствующий указатель на записи переполнения. При использовании открытой адресации аналогичные замечания справедливы не только для цепочки, из которой удаляется запись, но и для любой другой цепочки, которая пересекает этот участок записи.

На рис. 19.34 показано, какие изменения могут происходить в кластерах в тех случаях, когда используется открытая адресация с последовательным поиском участков для переполнений.

Как показано выше, некоторые области удаленных записей могут быть использованы повторно. Однако при определении длины пути выборки следует учитывать все области, которые не помечены явно как пустые. Поэтому величина n , используемая для вычисления значений p и P_{lu} , может превосходить текущее число записей.

Полное считывание прямого файла. Полный поиск в файле с использованием преобразования рандомизации ключа в адрес может быть выполнен только путем поиска по всему пространству, отведенному для файла, поскольку какие-либо преобразования значений ключей не позволяют выполнять упорядоченное считывание. Ввиду того что рабочая область имеет большой размер, такой поиск требует значительных затрат. Необходимо проверить также каждую область, используемую записями переполнения. Область переполнения является плотной (если не учитывать удалений). Таким образом,

$$T_X = (m + o) \frac{R + W}{t'}.$$

Реорганизация прямого файла. Реорганизация не является неотъемлемой частью обработки прямых файлов. Однако в тех случаях, когда используется открытая адресация и удаляется большое число записей, реорганизация приносит большую пользу. Если необходима высокая степень готовности системы, то реорганизации может подлежать один-единственный кластер (пространство между двумя пустыми участками). Возможно, что для переупорядочения сцепленных записей с учетом частоты доступа к ним потребуется прогон реорганизации, время выполнения которого составляет

$$T_Y = T_X + T_{Ld},$$

где T_{Ld} — время перезагрузки, оценка которого дана ниже.

Перезагрузка прямого файла. Наиболее очевидный путь перезагрузки прямого файла — переписать по очереди все записи в новую область. Тогда

$$T_{Ld} = nT_F,$$

где значение T_F возрастает с ростом величины n/m . Для приближенной оценки можно использовать величину $1/2(n/m)$. В этом случае затраты на перезагрузку будут большими, и если используется открытая адресация, то в новом файле уже будут образованы кластеры.

Для того чтобы уменьшить влияние кластеров, может быть использована двухпроходная процедура. При первом проходе записи помещаются только в первые участки; при втором проходе оставшиеся записи помещаются в незаполненные участки файла.

Сокращение затрат на загрузку может быть достигнуто путем сортировки загружаемого файла. Для этого к ключам записей, которые должны быть загружены в файл, применяется преобразование ключа в адрес и полученные адреса приписываются соответствующим записям в качестве ключей сортировки. Затем файл загрузки сортируется и записывается в область прямого файла. При этом участки, для которых нет записей с соответствующими адресами, пропускаются, а конфликтующие записи помещаются в последующие участки. Время записи здесь равно времени последовательного полного считывания, так что

$$T_{Ld} = c + \text{sort}(n) + T_X,$$

а время, необходимое для реорганизации, равно

$$T_Y = c + 2T_X + \text{sort}(n).$$

Благодаря значительному сокращению времени загрузки реорганизация с успехом может использоваться вместо более сложных методов, применяемых для решения проблем, связанных с кластеризацией, и управления процессом удаления записей в динамической среде. Этот метод также может быть использован для включения пакета записей в файл. Включаемый пакет должен быть достаточно большим. Использование указанного метода включения оправдано только в том случае, когда $n_l > (T_x/T_l)$, где n_l — размер пакета.

Реорганизация файла проводится также в тех случаях, когда размер файла стал настолько большим, что отношение n/m превысило допустимое значение. В этом случае для первичной области файла следует выделить больше пространства, а процедуру рандомизации необходимо переписать или видоизменить. Все данные следует скопировать путем полного считывания файла и переписать в новые участки. При оценке слагаемых T_x и T_{Ld} следует использовать старое значение m для T_x и новое значение m для T_{Ld} .

19.16. МНОГОКОЛЬЦЕВОЙ ФАЙЛ

Три предыдущих способа организации файлов используются в тех случаях, когда требуется быстрый поиск отдельных записей. Последний из шести основных способов позволяет эффективно обрабатывать подмножества записей, содержащих некоторое общее значение атрибута. **Многокольцевой** подход используется во многих системах баз данных; в этом разделе будет рассмотрена только файловая структура, используемая в подобных системах. Подмножество образуется с помощью указателей, которые определяют некоторый порядок для его членов. Одна запись может быть членом многих таких подмножеств. Цепочки могут быть также построены вне заголовков таких подмножеств. Заголовок содержит информацию, относящуюся ко всем подчиненным ему записям.

Специальным типом цепочки, который будет описан в этом разделе, является **кольцо**, т. е. цепочка, в которой поле указателя последнего члена используется для указания ее начала. Аналогичные файловые структуры, называемые в литературе **связными списками** или **мультисписками**, могут быть организованы с помощью колец или простых цепочек. Кольца, рассматриваемые здесь, могут быть вложенными и иметь несколько уровней глубины. Для связных списков или мультисписков это не всегда так. Примером запроса, обработку которого удобно выполнять с помощью такой организации, является следующий:

“Перечислить всех служащих в Туле”.

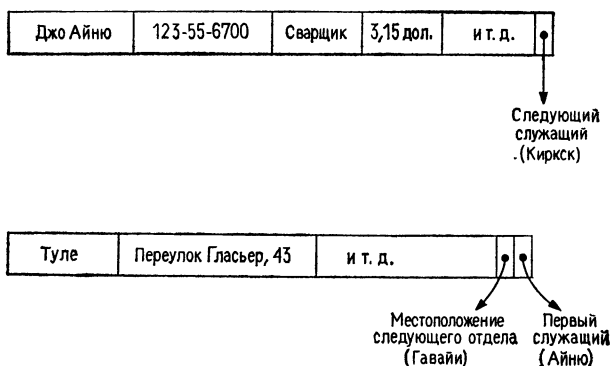


Рис. 19.36. Записи в кольцевой структуре.

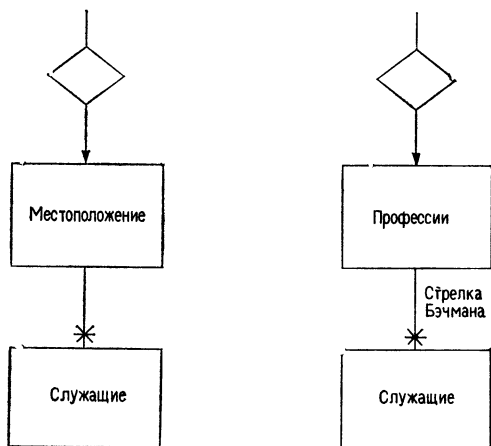


Рис. 19.37. Два файла сведений о служащих.

Описание многокольцевых файлов. Для того чтобы упростить изображение кольцевой структуры, будут использоваться прямоугольники, соответствующие кольцам, и специальные стрелки, указывающие взаимосвязь между ними. На рис. 19.37 показаны кольцевая структура, изображенная на рис. 19.35, и кольцевая структура, позволяющая эффективно обрабатывать второй из указанных запросов. Стрелка данной формы указывает, что в конечной ее точке будет несколько колец, однако на рисунке указано только одно подмножество. Обычная стрелка используется для указания **точки входа**, т. е. кольца, заголовков которого доступен непосредственно. Такие кольца каталогизируются, и с их помощью определяется начальная точка для обработки запроса.

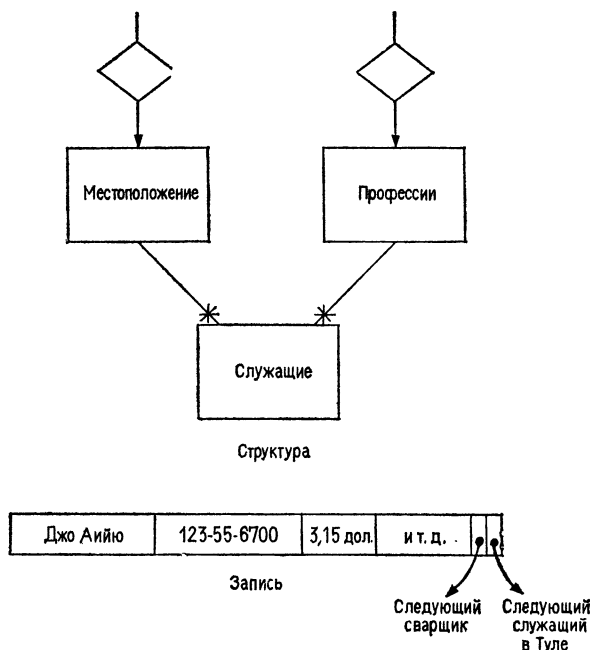


Рис. 19.38. Сцепленные кольца.

Очевидно, что файлы сведений о служащих, изображенные на рис. 19.37, можно объединить, заменяя поле данных для профессии другим полем связи, как показано на рис. 19.38. Кольцевая структура с этими двумя связями уже стала весьма сложной, однако на практике структуры могут содержать значительно большее число колец.

Если расширить приведенный выше пример, распределяя служащих из различных городов по определенным отделам, организовать средства доступа в порядке, определяемом трудовым стажем, добавить в каждом городе товарный склад и хранить информацию об ассортименте товаров, то диаграмма структуры будет такой, как показано на рис. 19.39. Если изобразить фактические указатели связи, то диаграмма станет похожа на чашу со спагетти.

Связи между кольцами не обязательно должны быть иерархическими. Концептуально они могут быть организованы таким образом, чтобы связать члены одного и того же кольца (рис. 19.40), организовать множественные пути между записями или связать нижние кольца с кольцами более высокого уровня.

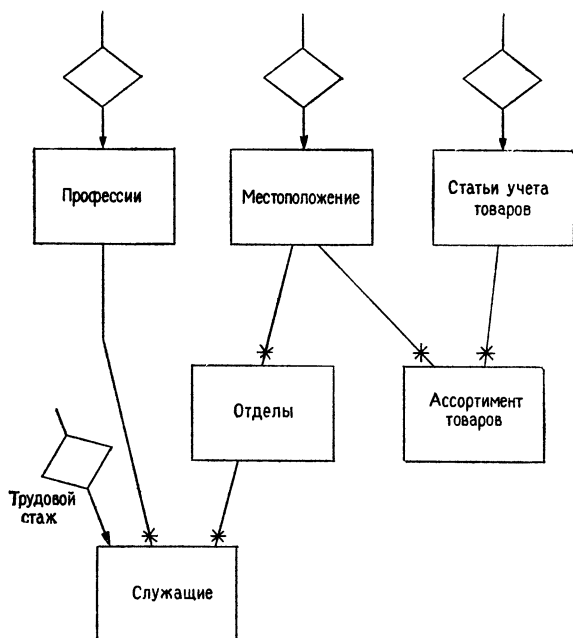


Рис. 19.39. Сложная иерархическая структура.

Для любителей футбола на рис. 19.41 даны два примера множественных путей.

На практике не все конструкции допустимы или реализуемы. За легкостью, с которой можно изображать стрелки связи, скрывается сложность основной структуры. Циклы и другие неиерархические связи между записями нежелательны, поскольку для их организации может потребоваться переменное число полей указателей в записях. Например, для реализации отношения “супружеская пара” (рис. 19.40), можно ограничиться одним входом. Отношение “дети — клиники” лучше всего может быть реализовано, используя поисковый аргумент элемента “дети”, связанный с индексом “клиники”.

При поиске оптимального метода получения фактографических данных (фактов) или их подмножеств в сложных структурах может возникнуть необходимость в выборе одного среди нескольких возможных путей доступа. Такой запрос, как

“Найти сварщика в Туле”,

может быть обработан, начиная с кольца местоположения или с кольца профессий. Требуемые записи расположены на пере-

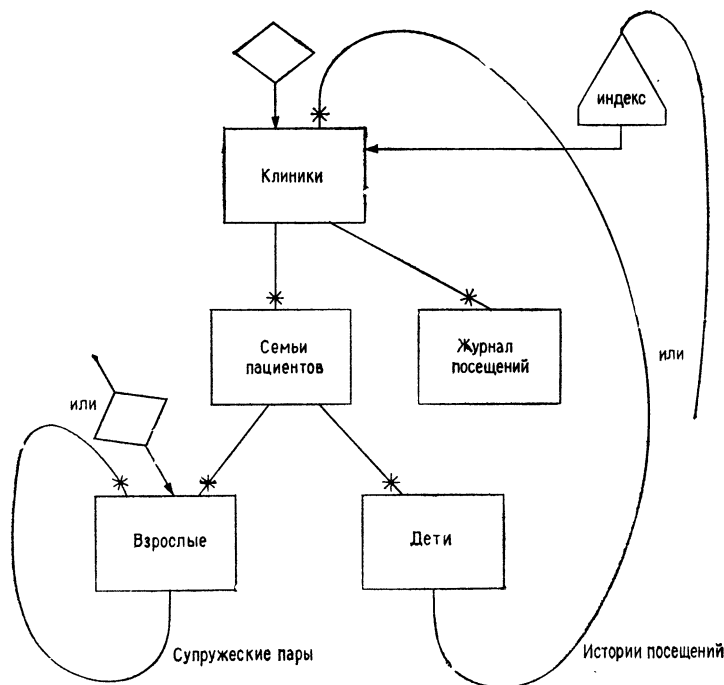


Рис. 19.40. Циклы в кольцевых структурах.

сечении кольца, соответствующего любому отделу в Туле, и кольца, соответствующего сварщикам.

Эффективность процесса локализации записи находится в прямой зависимости от того, насколько пары атрибутов, образующие аргумент запроса, соответствуют структуре факта или подмножества, для которой возможен автоматический выбор среди нескольких путей доступа.

Если отсутствует кольцо профессий, то путь для обработки последнего запроса начинался бы с точки входа в кольцо местоположения. Поскольку кольцо местоположения не содержит точки входа для профессий, необходим исчерпывающий поиск по всем кольцам отделов в Туле. В интерактивном режиме работы система в этот момент может "спросить": "Назовите предполагаемый вами отдел". Процесс нахождения наилучшего пути для обработки запроса в такой базе данных одним из ведущих специалистов, участвовавших в разработке этого проекта файловой системы (Бэчманом), был назван навигацией.

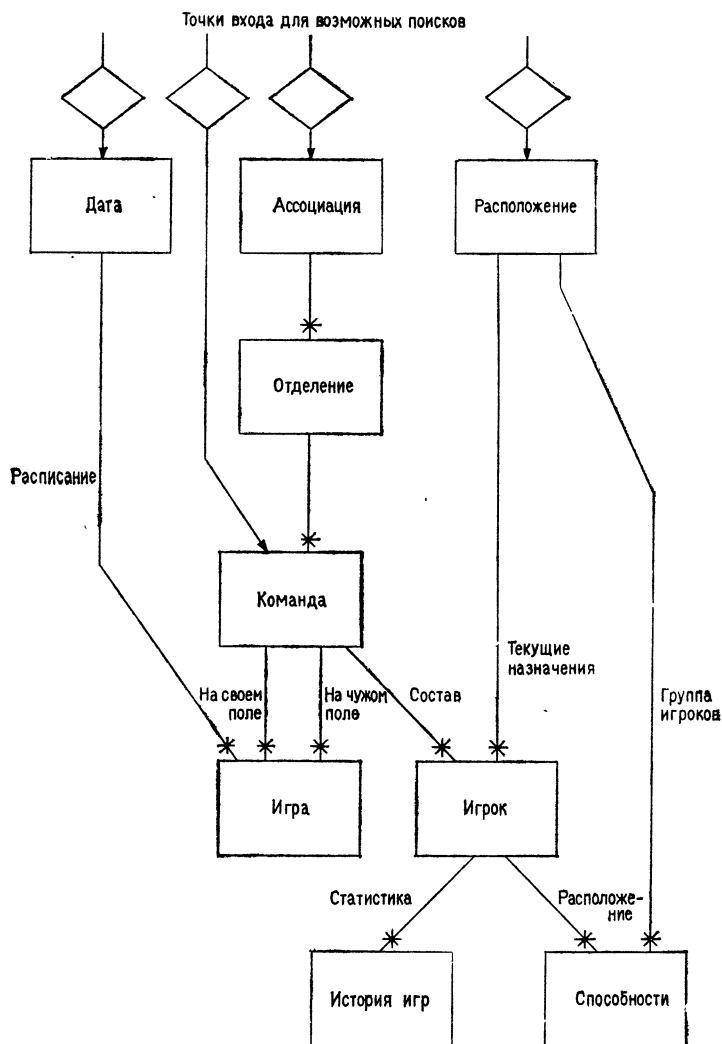


Рис. 19.41. База данных для национальной футбольной лиги.

Структура многокольцевых файлов и манипулирование ими

В многокольцевом файле записи имеют одинаковую структуру, однако их содержание и размер зависят от колец, которым они принадлежат.

Имя		Место жительства и дата рождения				Связи					
243	t	Джо,	201 13-я Улица,	27 сен 1936	317	317	124	231	364	Λ	Λ
244	t	Ф.Мак-Гроу III,	Уединенная Гора	2 июн 1898	001	106	Λ	213	366	110	010

Идентификатор типа записи
 Следующий член отдела
 Следующий служащий с той же профессией
 Следующий служащий в послужном списке, трудовую стажу
 Следующий служащий в медицинской записи
 Начало кольца соответствующего
 семье служащего
 Кольца для спец. профессий

Рис. 19.42. Задание полей записей кольцевого файла.

Форматы записей. Для организации связей между записями используются указатели. Запись может принадлежать стольким кольцам, сколько указателей она содержит. Определенному классу колец обычно назначается заданная позиция для указателя в формате записи. В приведенном выше примере таким классом может быть профессия, а кольцо сварщиков — одно определенное кольцо этого класса. Пример записи сведений о служащем в таком файле дан на рис. 19.42. В тех случаях, когда какая-либо запись не является членом кольца определенного класса, соответствующего типу этой записи, появляется неиспользованное поле указателя, содержащее пустой элемент.

Точный формат записи зависит от комбинации классов колец, членом которых является эта запись. Число различных типов записей, содержащихся в многокольцевом файле, может быть весьма большим. Пары атрибут — значение могут быть самоопределяющимися, как в файле-множестве; в противном случае для каждой записи потребуется идентификатор типа записи. Этот идентификатор позволит обращаться к соответствующему описанию формата записи, хранимому с общим описанием файла.

Отметим, что здесь мы отошли от сформулированного выше определения файла; записи теперь не идентичны по формату, и членство классов колец, а также членство файла должны быть известны до обработки.

Заголовки. Каждое кольцо имеет заголовок. Этот заголовок является либо точкой входа, либо членом другого кольца, либо и тем и другим. Заголовком для кольца служащих отдела является название отдела; заголовком для кольца сварщиков является один из членов кольца профессий; а записи сведений

о служащих являются заголовками для колец, соответствующих их семьям. Когда во время поиска осуществляется вход в кольцо, точка входа помечается, так что при повторном достижении этой точки поиск может быть завершен.

Как видно из приведенных выше примеров, многокольцевая организация файлов позволяет избежать избыточности данных. Для этого данные, общие для всех членов кольца, помещаются в его заголовок. Это означает, что в основном кольцевом проекте при выборке записей по комбинации аргументов поиска нельзя достичь соответствия между аргументами и значениями ключей, если ограничиться использованием информации, содержащейся в записи. Известны следующие два метода поиска.

Параллельный поиск по всем кольцам, указанным в аргументе поиска, завершающийся на записи или записях, расположенных на пересечении этих колец.

Поиск может быть проведен в соответствии с атрибутом, имеющим наибольшую эффективность разбиения. Затем нужные записи ищутся среди отобранных путем локализации заголовков для других типов атрибутов и отбрасывания записей с неподходящими значениями данных. Ниже дается пример использования последнего метода для обработки первого из указанных выше запросов.

Локализуются все записи, соответствующие служащим в Туле. Каждая такая запись исследуется путем просмотра цепочки профессии, с тем чтобы из заголовка определить, какой профессии, бухгалтера, продавца, сварщика и т. д., соответствует эта цепочка.

Важность конечного указателя, который преобразует цепочку в кольцо, очевидна. Однако при получении данных из заголовка могут возникнуть большие затраты. Следовательно, может оказаться полезным хранить в записи избыточные описательные данные или расширить структуру, с тем чтобы упростить доступ к заголовкам.

Членство кольца. Решения о выборе атрибутов, для которых должны быть образованы кольца, принимаются из тех же соображений, на основе которых выбираются атрибуты, подлежащие индексированию. Затраты на поиск в индексированном файле возрастают только логарифмически с ростом размера файла, тогда как затраты на просмотр цепочек возрастают линейно с ростом размеров последних. Размеры отдельных цепочек могут быть уменьшены путем выбора подходящего проекта структуры файла. При возрастании числа уровней (x) в структуре цепочек длины цепочек уменьшаются, так что время поиска сокращается пропорционально корню степени x из числа записей n . Атрибут, который не имеет структур-

ной значимости в файле, как, например, страховой номер, не целесообразно использовать для организации кольца в файле сведений о служащих, так как в этом случае при поиске какого-либо служащего ожидаемое число выборок составит $n/2$. Однако поиск определенного сварщика может быть эффективно осуществлен путем просмотра цепочки профессий, а затем цепочки сварщиков. Проектировщик файлов предполагает, что обе эти цепочки эффективно разобьют область поиска; оптимум достигается в том случае, когда длины обеих цепочек равны \sqrt{n} .

Пример 19.15. Имеется 10 000 служащих с 10 000 страховыми полисами и 50 профессиями (по 200 человек на каждую профессию).

В первом варианте поиска — по номеру страхового полиса — требуется 5000 выборок.

Во втором варианте поиска — по профессии, а затем среди служащих одной профессии — число ожидаемых выборок составляет $25 + 100 = 125$.

Эта величина намного больше, чем ожидаемое число выборок с использованием индексов. Однако выборка следующей записи в этой структуре является более экономичной, чем в тех случаях, когда используются индексированные файлы.

Если требуется получить данные о всех сварщиках, то ожидаемое число доступов составляет $25 + 200 = 225$. Такое число записей, по-видимому, не может быть найдено более эффективно с использованием методов, рассмотренных в предыдущих разделах. Для того чтобы дополнительно сократить время доступа, при разработке проекта файла необходимо также рассмотреть вопрос о расположении колец.

Некоторые атрибуты, как, например, профессия, могут быть разбиты на определенное число дискретных классов (сварщики, бухгалтеры). Такое разделение на классы обеспечит эффективное разбиение записей и будет соответствовать естественной структуре данных. Атрибуты, не представляющие дискретные данные, например вес или высота, не обеспечивают эффективного разбиения, если они не разделены на классы искусственно. Такие искусственные классы в этом случае могут определяться, например, так: меньше, чем пять футов, $5'0''$ — $5'2''$, $5'3''$ — $5'5''$, $5'6''$ — $5'8''$ и т. д. Если непрерывные переменные, подобные этим, используются в качестве вторичных классов, так что число членов в соответствующих им кольцах невелико, то можно ограничиться простым упорядочением членов внутри этих колец. Такое упорядочение часто используется на практике.

При практической реализации кольцевых структур следует учитывать ряд важных дополнительных особенностей.

Использование многокольцевых файлов

Заголовки и описательные записи можно встретить в традиционных процедурах обработки данных. Многокольцевые структуры являются основой для некоторых самых больших из используемых в настоящее время баз данных. С помощью многосвязных списков были реализованы административные информационные системы, большая часть работы которых заключалась в табулировании, суммировании и выдаче сообщений об особых случаях. Примеры таких операций были даны во введении к этой главе.

При решении некоторых задач в географии и архитектуре, связанных с пространственным изображением объектов, также используется многокольцевой подход. Функционирование ряда современных интегрированных многофайловых систем в значительной степени определяется средствами, предоставляемыми кольцевыми структурами. Трудности, возникающие при разработке кольцевой файловой системы, связаны с тем, что до ее реализации необходимо составить точный проект, основанный на заранее полученных сведениях о данных и возможных вариантах ее использования.

Эффективность использования многокольцевых файлов

Эффективность использования многокольцевой системы в значительной степени зависит от того, насколько правильно атрибуты назначены отдельным кольцам. При получении оценок будем предполагать, что файловая структура оптимально соответствует требованиям, определяемым характером ее применения.

Размер записи в многокольцевом файле. Обычно в многокольцевых структурах записи каждого отдельного типа имеют фиксированный формат. Однако, поскольку может существовать большое число типов записей, при правильном проектировании файла не многие поля останутся пустыми. Следовательно, в этом случае можно ожидать большую плотность файла, чем для последовательной или прямой организации. В каждой записи имеется также фиксированное число полей связи, и это число ненамного больше того, которое потребовалось бы для отдельной записи данного типа. Можно предполагать, что суммарное число полей соразмерно со значением параметра a' , который использовался для обозначения фактического числа

атрибутов записи. При детальной разработке проекта необходимо определить размеры записей каждого типа, поскольку в многокольцевых файлах размер записи зависит не только от ее содержания, но в значительной степени и от выбранной структуры файла. Знание размеров записей необходимо для оценки эффективности использования файла. Размер поля данных вновь полагается в среднем равным V , поле связи содержит значения указателей размера P . В записи содержатся $a'_{\text{данные}}$ полей данных и $a'_{\text{связь}}$ полей связи. Для идентификации типа записи используется одно дополнительное поле размера P . Тогда размер записи равен

$$R = a'_{\text{данные}} V + a'_{\text{связь}} P + P.$$

Если при хранении информации отсутствует избыточность, то значение атрибута представляется либо значением данных, либо указателем связи. Если, кроме того, нет пустых полей, то

$$a'_{\text{данные}} + a'_{\text{связь}} = a'.$$

Если запись является членом небольшого числа колец, то различием в размерах значений данных и указателей можно пренебречь, так что

$$R \approx a' V.$$

Если поля атрибутов и поля связей полностью дублируют друг друга, но по-прежнему нет пустых полей, то

$$a'_{\text{данные}} = a'_{\text{связь}} = a'.$$

В этом случае размер записи равен

$$R = a' (V + P) + P.$$

Для оценки среднего размера записи будем предполагать, что в целом для файла число полей атрибутов, отсутствующих благодаря наличию указателей связи, совпадает с числом пустых полей, возникших вследствие фиксированной структуры записи, так что

$$R \approx a' (V + P).$$

Выборка записи из многокольцевого файла. Время выборки записи является функцией числа исследуемых цепочек и их длин. Будем предполагать, что запись содержит достаточное количество данных, так что при поиске ее в соответствии с одной определенной последовательностью доступа она может быть правильно идентифицирована. Следовательно, запись ищется в

иерархии x уровней; этот процесс аналогичен поиску в иерархии индекса. Рассмотрим случай, когда записи одного типа принадлежат только одному кольцу.

Как показано на рис. 19.43, средняя длина y кольца зависит от размера файла и от того, насколько хорошо файл разбит на кольца. В этой простой иерархической кольцевой структуре глубина иерархии определяется путем декомпозиции уровня в кольца размера y , так что

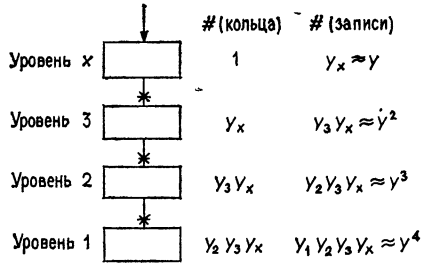


Рис. 19.43. Размер файла при четырех иерархических уровнях.

$$x \approx \log_y n.$$

На нижнем уровне (1) во всех кольцах содержится в общей сложности n записей, не считая другие типы записей на более высоких уровнях и другие иерархии в файле.

При выборке одной-единственной записи число иерархических уровней должно совпадать с числом аргументов поиска a_F в ключе поиска.

Если число аргументов поиска в ключе меньше числа уровней, то результатом поиска будет кольцо, т. е. подмножество записей файла; если число аргументов больше, то существует избыточность и непродуктивные с точки зрения оптимизации стратегии поиска атрибуты могут быть проигнорированы. В противном случае выполняется условие $a_F = x$ и

$$a_F = x = \log_y n \text{ или } y = \exp(\ln(n)/x) = \exp(\ln(n)/a_F) = n^{1/a_F}.$$

При поиске в одном уровне ожидаемое число доступов к записям составляет $(1 + y)/2$. При переходе с одного уровня на другой в одном кольце доступ будет производиться только к $y/2$ новым записям. Для локализации записи на самом низком уровне (а это наиболее вероятная цель поиска) потребуется просмотреть x колец, и если кольца на различных уровнях имеют приблизительно одинаковые размеры, то доступ будет производиться к $1/2xy$ записям. Предполагая, что блоки, содержащие эти записи, размещаются произвольным образом, получаем

$$T_F = \frac{xy}{2} (s + r + bit).$$

Подставляя в полученное выражение ожидаемое число a_F иерархических уровней, используемых в запросе, и значение y ,

вычисленное для оптимальной иерархии, получаем

$$T_F = a_F \left[\frac{1}{2} e^{\ln(n)/a_F} \right] (s + r + btt) = fna (s + r + btt),$$

где fna — функция, зависящая от a_F и n . Как видно, при выводе этого соотношения критическим является допущение об оптимальности структуры данных, поскольку структура файла отражает связи, свойственные данным. На практике в тех случаях, когда время поиска для некоторых ожидаемых комбинаций атрибутов поиска превосходит допустимый предел, используются дополнительные кольца или связи.

Таблица 19.4. Значения коэффициента fna доступа к файлу

a_F	fna		
	$n = 10\ 000$	$n = 100\ 000$	$n = 1\ 000\ 000$
1	5 000	50 000	500 000
2	100	318	1 000
3	33	72	150
4	20	36	72
5	20	25	32
6	18	24	30

fna — коэффициент доступа к файлу,

a_F — число уровней колец в иерархии поиска,

n — число записей на уровне, к которому осуществляется доступ.

В табл. 19.4 перечислены значения fna для некоторых значений n и a_F . Другая возможность сократить время поиска — расположить наиболее важные кольца на одном и том же цилиндре, уменьшая тем самым частоту выполнения операции установки (величина s в приведенном выше равенстве). Реализация больших баз данных, имеющих такую структуру, осуществляется с помощью дисков с фиксированными головками, для которых установка не требуется, так что $s = 0$.

Получение следующей записи из многокольцевого файла. Следующая запись, соответствующая любой из связанных последовательностей, может быть найдена путем следования по нужной цепочке. Поэтому

$$T_N = s + r + btt.$$

Следует отметить, что в рассматриваемой организации файлов информация об очередности следования записей существует для нескольких атрибутов. Единственной другой организацией, которая обеспечивает такую возможность, является мультиин-

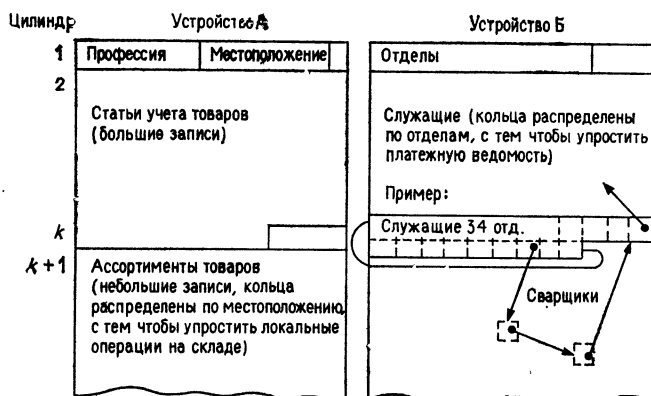


Рис. 19.44. Размещение колец в памяти.

дексированная организация файла. Если упорядоченный доступ часто осуществляется через одно определенное кольцо, то все записи этого кольца можно расположить на одном и том же цилиндре. Поэтому при осуществлении доступа к членам этого кольца $s=0$. Для многокольцевой структуры операция получения следующей записи является наиболее важной операцией обработки данных. Поэтому структура многокольцевого файла выбирается таким образом, чтобы максимально сократить время выполнения этой операции, а не операции выборки.

Размещение колец на дисках, минимизирующее число установок головок. Записи, являющиеся членами только одного кольца, очевидно, лучше всего располагать близко друг от друга. Одно кольцо обычно может быть размещено на одном цилиндре, так что при просмотре этого **первичного** кольца установка не требуется. На более высоких уровнях кольцевой структуры число записей определенного типа может быть небольшим, и эти записи можно хранить на одном или нескольких цилиндрах. Поэтому при просмотре любого кольца либо совсем не требуется установок, либо их требуется минимальное количество. Если файл имеет большие размеры, то для хранения записей определенного типа, расположенных на низких уровнях, требуется большое число цилиндров. Поэтому при просмотре любой последовательности, отличной от первичного кольца, потребуются установки головок.

Возможное размещение данных, изображенных на рис. 19.39, схематически показано на рис. 19.44. Для локализации записи служащего через его отдел потребуется только несколько установок при просмотре кольца отделов и кольца служащих этого

отдела. Если в отделе большое число служащих, то возможно, что это кольцо будет размещаться на нескольких цилиндрах. При поиске служащего по профессии необходимость в установке головок может возникать почти при каждом переходе от одной записи к другой. Ассортимент товаров для заданного местоположения определяется с помощью нескольких установок головок. Соответствующие кольца обычно имеют большую длину, и для их просмотра потребуется несколько установок головок при переходе от одного цилиндра к другому. Далее, можно связать данные об ассортименте товара со статьями учета (в одном направлении). Тогда при переходе от одной записи кольца, указывающего ассортимент, к другой потребуется установка головок. Однако эти кольца могут быть короткими, если статья учета хранится в небольшом числе мест.

При получении оценок эффективности использования кольцевого файла, с одной стороны, делается оптимистическое предположение об оптимальных размерах кольца, а с другой стороны, не учитываются те выгоды, которые могут быть получены в результате оптимального размещения колец. В динамически изменяющейся базе данных трудно постоянно иметь оптимальное размещение, поэтому соответствующие выгоды следует учитывать лишь частично. Для восстановления оптимального размещения может потребоваться реорганизация.

Включение записей в многокольцевой файл. Для добавления записи к многокольцевому файлу определяется подходящая свободная область, локализуются все предшественники новой записи, извлекаются значения соответствующих указателей связи из предшественников и помещаются в новую запись, а в области связи предшественников засылается информация о расположении новой записи.

Следовательно, общие затраты эквивалентны сумме времени, необходимого для $a'_{\text{связь}}$ выборок такой записи, и времени фактической перезаписи. В тех случаях, когда цепочки не упорядочены, затраты на нижних уровнях можно несколько сократить.

Неупорядоченные атрибуты. Если записи упорядочены по значениям атрибута, то в цепочке необходимо найти точку корректного включения. Если цепочка связывает все идентичные значения атрибута, как, например, всех сварщиков, то считыванию, модифицированию и перезаписи подлежит только заголовков. Для коротких колец, в которых порядок не имеет какого-либо значения, новые записи могут быть включены также в начальную позицию. В результате этого записи будут располагаться в обратной хронологической последовательности. Такая последовательность часто используется при поиске, по-

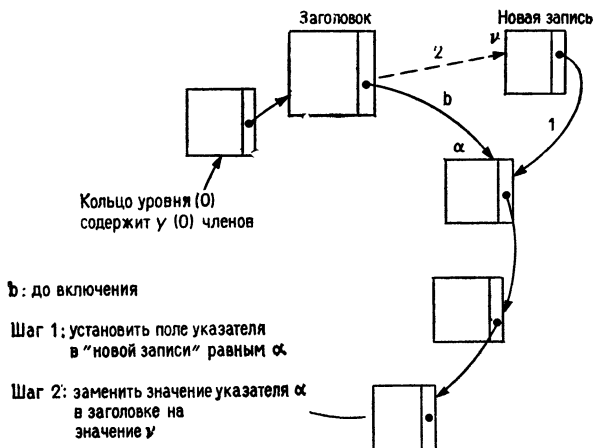


Рис. 19.45. Включение записи в кольцо.

сколько она позволяет легко искать последние поступившие данные. Этот процесс изображен на рис. 19.45. Такое сцепление следует выполнить для всех $a'_{\text{связь}}$ колец, членом которых является новая запись. Количество доступов можно сократить на $a''_{\text{неупорядоченных}} y/2$, где $a''_{\text{неупорядоченных}}$ — число неупорядоченных или идентичных связей атрибутов. Сумма времен, необходимых для выполнения выборок, перезаписей, а также для размещения конечной записи, равна

$$T_I = a'_{\text{связь}} (T_F + 2r) + s + r + btt + 2r - a''_{\text{неупорядоченных}} \frac{y}{2} (s + r + btt).$$

Как видно, затраты на включение новой записи весьма значительны, особенно в тех случаях, когда запись является членом многих колец.

Обновление записи в многокольцевом файле. Если требуется изменить только поля данных, то для обновления необходимо лишь найти запись и переписать ее. Можно предполагать, что при обновлении записей их тип и длина остаются прежними. Тогда

$$T_U = T_F + 2r$$

для случая, когда изменяются только поля данных.

При обновлении записей может также возникнуть необходимость в изменении связей. Поскольку обновленная запись может быть переписана в первоначальную позицию, то коррек-

тировке подлежат только те связи, значения которых изменились. Если измененное значение по-прежнему принадлежит классу атрибута поиска, используемого для локализации записи, то точка включения новой записи находится в том же самом кольце и число ожидаемых выборок составит только $1/2y$. Если, например, записи о товарах определенного типа хранятся в кольце рассортированные по весам, то при изменении одной записи может возникнуть необходимость в изменении расположения другой записи этого кольца. В этом случае, учитывая время на поиск и перезапись предшествующей записи и перепись новой записи, получаем, что

$$T_U = T_F + \frac{1}{2} y (s + r + btt) + 2r + s + r + btt + 2r,$$

если внутри кольца изменяется одно-единственное поле связи.

Если в ходе поиска старой позиции записи делается отметка при прохождении новой позиции, то иногда удастся избежать выборок, требуемых для локализации новой предшествующей записи. В этом случае в качестве ожидаемого времени переписи нельзя использовать величину $2r$, так как поиск новой позиции прерывается.

Если необходимо изменить несколько (a_U) связей или если обновление вызывает изменение местоположения внутри одного и того же кольца, то следует определить местоположение новых точек включения. Это может быть сделано с помощью поиска из вершины по аналогии со случаем включения записи. Альтернатива — использовать указатели местоположения в кольце связи старой записи.

Для поиска заголовков a_U соединений из вершины иерархической структуры, как и прежде, требуется a_U выборок, время выполнения каждой из которых равно T_F . Может оказаться быстрее достичь заголовков с помощью кольца и затем, просматривая их, достичь требуемого места ('Гавайи'). Необходимо также локализовать предшествующую запись ('Калхоун') в старом кольце и переписать ее с указателем на следующую запись ('Моррис') в том же самом кольце. Ожидаемое число доступов вновь равно $y/2$. Этот процесс изображен на рис. 19.46. Если записи в кольце упорядочены, то нахождение места включения ('Кулохело') по-прежнему необходимо. Суммарное число доступов во всем процессе с учетом всех полей равно

$$a_U 4 \frac{y}{2} - a''_{\text{неупорядоченных}} \frac{y}{2},$$

где $a''_{\text{неупорядоченных}}$ — число неупорядоченных связей, включенных в запрос на обновление. При каждом обновлении требуется также переписать новую и старую предшествующие записи.

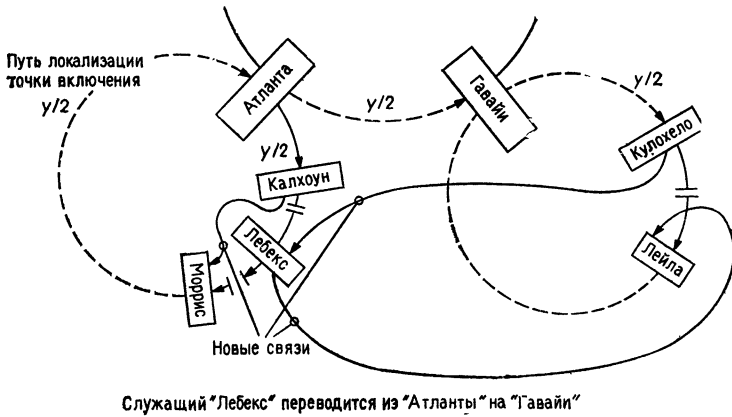


Рис. 19.46. Нахождение новой точки связи.

Поскольку запись, которая должна быть обновлена, в начале обработки локализуется путем выборки или с помощью операции поиска следующей записи, то одна из последовательностей доступов может быть расширением процесса выборки. Эффект от этого, вообще говоря, незначителен и поэтому не будет учитываться при получении оценок. При обновлении записей, расположенных ближе чем на три уровня от вершины (**номер уровня** $> x - 3$), поиск точки включения выгоднее проводить путем выборок, начиная от вершины.

Всякий раз, когда обновляется поле связи, необходимо локализовать и переписать как старую, так и новую предшествующие записи. Тогда, используя описанный выше способ получения оценки, имеем

$$T_U = a_U (2y(s + r + btt) + 2r + 2r) + \\ + s + r + btt + 2r - a''_{\text{неупорядоченных}} \frac{y}{2} (s + r + btt).$$

Некоторое сокращение этого времени возможно в том случае, когда после корректировки обновленная запись остается членом прежнего кольца. В этом случае по-прежнему справедливы многие замечания, касающиеся вопросов хранения данных, а оценки будут зависеть от длин и расположения цепочек и образцов обновления.

Полное считывание многокольцевого файла. Полное считывание производится путем упорядоченного просмотра каждого множества возможных связей. Для того чтобы гарантировать, что ни одна запись не будет считываться более одного раза,

необходимо хорошее знание проекта файла. Поскольку поиск осуществляется с помощью связей, указанных в записях, это приводит к большим затратам. Альтернативный вариант, последовательный поиск по пространству, не является простым, так как записи имеют различные форматы и для понимания смысла считываемых полей необходимо описание типа записи.

Затраты на считывание по цепочкам составляют

$$T_x = n(s + r + btt),$$

однако, помимо этого, более одного раза потребуются заголовки. В некоторых случаях в оперативной памяти можно хранить стек из x заголовков.

Если заголовки не могут быть сохранены в оперативной памяти и должны считываться повторно, то в полученную выше формулу необходимо включить дополнительный множитель, зависящий от средней длины цепочек, так что

$$T_x = n \left(1 + \frac{1}{y} \right) (s + r + btt).$$

Реорганизация многокольцевого файла. Реорганизация не требуется как часть обычных операционных процедур. Например, для одной системы баз данных, основанной на многокольцевых файлах (IDS), которая функционировала приблизительно с 1966 г., не было совместно действующей программы реорганизации до 1975 г. Только в тех случаях, когда потребуется переформатизация типов записей, такие записи необходимо будет переписать. При этом может потребоваться только частичная перезапись файла, затраты на которую такие же, как и затраты на обновление всех записей цепочки, т. е.

$$T_y (\text{част.}) = yT_U.$$

Аналогично затраты на полную реорганизацию составляют

$$T_y = nT_U,$$

если можно избежать повторного считывания заголовков.

Если ожидается частая реорганизация файла, то с помощью соответствующей модификации его основной структуры можно повысить эффективность его использования.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Абсолютный двоичный код 10

Ада 64, 65

Адреса файла 392

Адресное пространство 394

Алгол 63, 65

Анкерные точки 355—356, 380

АПЛ 65

Ассемблер 11—12

— в системе Cyber 170 32—37

— — — IBM 370 24—29

— — — Intel 8080 37—39

— — — PDP-11 30—32

— двухпроходный 12—14

База данных 300—432

Базовый регистр 28—29

Бакет 395, 400, 401, 403

Бейсик 63, 64, 65, 68—90

Библиотека 49—50

Блок 22

Блок-схема 317—318

Блок BEGIN 284, 288—289

Боулз К. 185

Бэкус Дж. 63

Бэчман 418

Ввод-вывод 303

— в Бейсике 86—90

— — Коболе 137—144

— — Паскале 194—195

— — ПЛ/1 239—261

— — — записеориентированный 241, 254—258

— — — потокоориентированный 240, 241, 242—254

— — — управляемый редактированием 245—254

— — — списком 242—245

— — Фортране 180—184

— — потокоориентированный 180—181

— — — форматизованный 181—184

Ведение индекса 379—380

Вектор передач (переходов) 43

Вирт Н. 185

Виртуальная машина 55

Включение записи 332—333

— в индексированный файл 386—388

— — индексно-последовательный файл 374

— — многокольцевой файл 428—429

— — — последовательный файл 350—351

— — — прямой файл 408—411

— — — файл-множество 340—341

Второй проход 13, 22—24

Выборка записи 330—331

— — из индексированного файла 383—384

— — — индексно-последовательного файла 368—371

— — — многокольцевого файла 424—426

— — — последовательного файла 347—349

— — — прямого файла 404—408

— — — файла-множества 338

Выражение в Коболе арифметическое 115—116

— — ПЛ/1 219—227

— — — арифметическое 219—220

— — — логическое 221—224

— — — над массивами 224—226

— — — строковое 220—221

— — — элементарное 219—224

Вычисления с базой данных 304

Группа DO 233

В-Дерево 383

Директива 18

— ассемблеру 20—22

Документация 295—299

— хорошая 295—296

Доступ 334

— в Коболе 142—144

Загрузчик абсолютный 10, 11, 41

— в системе Cyber 170 48—49

— — — IBM 370 47

— — — Intel 8080 49

— — — PDP-11 48

— перемещающий 41

— связующий 46

Запись 59, 301

— в Коболе 139—142

— — Паскале 197

Запрос 334

Зарезервированные слова в Бейсике 90

— — — Коболе 96—98

— — — Паскале 198

— — — Фортране 184

Захватывание 51—54

Защита памяти 52, 53

Зондирование 349

Идентификатор в Коболе 110

— — Паскале 187

Инвариант 51
 Индекс 353—358
 — главный 357, 384
 — избирательный 379
 — многоуровневый 384
 — первичный 355
 — полный 379
 — с учетом аппаратных средств 357—358
 — цилиндров 356
 Интерпретатор 55—56, 62
 — команд 57
 Интерпретация 54—56

Кемени Дж. 63, 68
 Класс памяти 287
 Кластеризация 401
 Ключ записи 304—305, 335
 — — в Коболе 139
 — — — ПЛ/1 257
 Кобол 63, 66, 91—159
 Коллизия 393, 394, 399—400, 402, 404, 410
 Кольцевые уровни 53
 Кольцо 413
 Команды ассемблера 24, 25, 28, 30, 37, 38
 — начальной загрузки 8—11
 — привилегированные 53
 — системные (в Бейсике) 69—72
 Комментарий в Бейсике 72
 — — Коболе 149, 150
 — — Паскале 186
 — — ПЛ/1 217—218
 — — Фортране 161
 Компилятор 62
 Константа в Бейсике 72—75
 — — Коболе 94—96
 — — Паскале 187—188
 — — ПЛ/1 201—204
 — — Фортране 162—163, 177
 — фигуральная 95—96
 Коэффициент расширения индекса 356—357
Куртц Т. 63, 68

Литерал в ассемблере 19, 27—28
 — — Коболе 94—95
 — — — нечисловой 95
 — — — числовой 94

Массив в Бейсике 74
 — — Паскале 189, 195—196
 — — ПЛ/1 209—210, 212—213
 — — Фортране 164—165
 — упакованный 195—196

Метод доступа 317
 — кусочно-линейного преобразования 397
 — линейного поиска 400, 406—407
 — остатка от деления 397
 — открытой адресации 395, 400, 405—406
 — перемешивания (хеширования) 393
 — повторной рандомизации 401—402
 — поиска в бакете 400—401
 — — — файле 401
 — преобразования ключа в адрес 392—399
 — — — — вероятностный 392, 393
 — — — — детерминированный 392, 393
 — — — — зависящий от распределения 395—396
 — — — — сохраняющий последовательность 393, 396—397
 — расширения и добавления 398
 — связи 44—45
 — цифрового анализа 396
 Множество 196
 Модульность 315
 Монитор 51—54
 Мультипрограммирование 322, 323
 Мультисписок 413

Навигация 418
 Начальная загрузка программы 8—11
 Начальное значение переменной 214—215
 Номер уровня 108
 Нормальный возврат 261, 284—285

Область ключей 394
 Область переполнения индексно-последовательного файла 358—362
 — — прямого файла 402—403
 Обновление записи 333
 — — в индексированном файле 388—389
 — — — индексно-последовательном файле 374
 — — — многокольцевом файле 429—431
 — — — последовательном файле 351
 — — — прямом файле 411
 — — — файле-множестве 341
 Обработка транзакций 322, 324—325
 Операторы ввода-вывода (см. ввод-вывод)
 — Бейсика
 — — арифметические 75

- — графические 84—86
- — и программы языка ассемблера 83—84
- — логические 76
- — отношения 75
- — управляющие 77—79
- Кобола
 - — отношения 121
 - — ADD 110, 117
 - — COMPUTE 110, 119
 - — DIVIDE 110, 118—119
 - — GO TO 111, 114
 - — IF 111, 121, 124—125
 - — INSPECT 111, 133—135
 - — MOVE 111, 128—130
 - — MULTIPLY 110, 118—119
 - — PERFORM 111, 125—128
 - — SEARCH 111, 135—136
 - — SET 111, 135—136
 - — STOP 111, 114
 - — STRING 111, 130—133
 - — SUBTRACT 110, 117—119
 - — UNSTRING 111, 130—133
- Паскаля
 - — отношения 190
 - — присваивания 189—190
 - — управляющие 190—193
 - — CASE 192
 - — GO TO 192—193
 - — IF 191—192
 - — REPEAT 191
 - — WHILE 191
- ПЛ/1
 - — арифметические 219—220
 - — логические 221—223
 - — отношения 221—224
 - — присваивания 227—231
 - — сцепления 221
 - — DO 210, 234—239
 - — GO TO 216, 218
 - — IF 216, 231—232
 - — ON 216, 259, 283—284
 - — SIGNAL 261
 - — STOP 216, 218
- Фортрана
 - — арифметические 165—166
 - — логические 169
 - — отношения 166
 - — сцепления 177
 - — управляющие 166—171
 - — эквивалентности 178
 - — DATA 165
 - — DO 170
 - — GO TO 167—168
 - — IF 168—169
 - — PAUSE 170
 - — STOP 161, 170
- Операционная система 321
- Описание данных 316—317
 - — записи 140—141
 - — переменной в Коболе 98—110
 - — — Паскале 187—188
 - — — ПЛ/1 205—209
 - — — Фортране 163—164
 - — файла 328
 - — в Коболе 139—140
 - — — ПЛ/1 254
- Определение данных 17—22
 - — констант в Паскале 187—188
 - — — Фортране 164
 - — типов 189
- Организация файла 317
 - — файловых систем 327—334
- Пакетная обработка 321—322
 - — запросов 338—340
- Память
 - — автоматическая 287, 288—289
 - — базированная 288, 290—294
 - — статическая 287, 288
 - — управляемая 288, 289—290
- Параграф 94, 113
- Паскаль 64, 66, 185—198
- Паскаль Б. 64, 185
- Первый проход 13, 14
- Переменная
 - — алфавитная 100
 - — алфавитно-цифровая 101
 - — редактируемая 101
 - — в Бейсике 72—75
 - — Коболе 96, 100—102
 - — Паскале 187—189
 - — ПЛ/1 204—205
 - — Фортране 162—163
 - — внешняя 277
 - — числовая 101
 - — редактируемая 102
- Перемещаемый адрес 41—42
- Плотность заполнения индекса 382—383
- ПЛ/1 64, 66, 199—294
- Побочный эффект 277—278
- Подпрограмма в Бейсике 79—80
 - — Коболе 145—149
 - — Фортране 171—174
 - — ПЛ/1 261—279
 - — внутренняя 275—277
- Поиск в многокольцевом файле 421
 - — двоичный 347—348
 - — параллельный 421
- Поле 14
 - — переменное 33
- Полное считывание 333

- индексированного файла 389—390
- индексно-последовательного файла 374—375
- многокольцевого файла 431—432
- последовательного файла 351—352
- прямого файла 411
- файла-множества 342
- Получение следующей записи 331—332
 - из индексированного файла 386
 - — — индексно-последовательного файла 372—374
 - — — многокольцевого файла 426—428
 - — — последовательного файла 349—350
 - — — прямого файла 408
 - — — файла-множества 340
- Порядок выполнения операций в Бейсике 76
 - — — Коболе 115—116, 124
 - — — Паскале 190
 - — — ПЛ/1 220—222
 - — — Фортране 166, 169, 178
- Последовательное чтение 332
- Поток данных 318
 - управляющих данных 318
- Предложение 94, 113
- Преобразование данных 228—230, 248
- Прерывание 51—53, 55
- Привязка 319—321
- Программа в Бейсике 72
 - Коболе 93—110, 149—150
 - Паскале 186—187
 - ПЛ/1 201, 217—218, 279—280
 - Фортране 161—162, 177
 - исполнительная 51
 - исходная 62
 - объектная 62
 - трассировки 54
- Проталкивание 361—362
- Процедура в Паскале 193—194
 - ПЛ/1 262, 271—275
 - Фортране 172
 - рекурсивная 278—279
- Процесс 306
- Псевдокоманды 17—22, 34—36
 - загрузки данных 18—20
- Раздел 93—94, 112**
 - данных 94, 113, 137, 139
 - идентификаций 94, 149
 - оборудования 94, 137—139
- процедур 94, 113, 137
- Разделение времени 322—324
- Размер записи 329—330
 - в индексированном файле 382
 - — индексно-последовательном файле 365—366
 - — многокольцевом файле 423—424
 - — — последовательном файле 346
 - — — прямом файле 404
 - — — файле-множестве 338
- Редактор связей 44—47
 - в системе IBM 370 47
- Режим 53
 - пользователя 53
 - супервизора 53
- Реорганизация 333—334
 - индексированного файла 390
 - индексно-последовательного файла 362—363, 375—376
 - многокольцевого файла 42
 - последовательного файла 352—353
 - прямого файла 412—413
 - файла-множества 342—343
- Рост индекса 383
- РПГ 66, 67
- Руководство 296—297
 - для обучения 296—298
 - по применению 296
 - системное 296
 - справочное 296, 297
- Си 65, 66
- Си-Бейсик 66
- Связный список 413
- Секция в Коболе 94, 113
 - процесса 306—307
 - управляющая 22
- Сечение массива 213
- Символ равенства 166
- Слово 94
 - определенное пользователем 96
- Снобол 67
- Справочник файла 327—328
- Состояния в Коболе
 - — — INVALID KEY 143—144
 - — — OVERFLOW 120, 131
 - — — ПЛ/1 280—287
 - — — CHECK 282, 283, 285—287
 - — — CONVERSION 229, 248, 249, 281—283, 285, 286
 - — — ENDFILE 258, 259, 261, 281—283, 285
 - — — ENDPAGE 259, 261, 281—283, 285
 - — — ERROR 252

- — — FIXEDOVERFLOW 227, 281—283, 285, 286
- — — KEY 257—259, 261, 281, 283, 285
- — — OVERFLOW 282, 283, 285, 286
- — — SIZE 229, 248, 282, 283, 285, 286
- — — STRINGRANGE 282, 283, 285, 286
- — — SUBSCRIPTRANGE 282, 283, 285, 286
- — — TRANSMIT 259, 261, 281—283, 285
- — — UNDERFLOW 281—283, 285, 286
- — — ZERODIVIDE 281—283, 285, 286
- Средства объединения 153
 - организации библиотек 152
 - отладки 152
 - сортировки 153—154
 - составления отчетов 155—158
- Статья 99
 - описания записи 107—110
 - — сортировки 153
 - управления файлом 137—138
 - JUSTIFIED 151
 - OCCURS 104—105
 - PICTURE 99—102
 - REDEFINES 151
 - USAGE 99, 102—103
 - VALUE 99, 103, 123, 151
- Страничная организация памяти 322, 324
- Структура 210—214
- Супервизор 51
- Сфера действия переменной 276
- Схема 316
 - загрузить и выполнить 22, 50—51
- Счетчик адресов 14, 36, 37
 - начального адреса 36, 37
- Таблица 103—107
- Типы данных в Бейсике 73
 - — — Коболе 94—96
 - — — Паскале 187
 - — — ПЛ/1 201—204
 - — — Фортране 162—163
- Транслятор 50—51
- Указатель 290, 292
 - собственный 291
- Упорядоченное чтение (запись) 332
- Управляющий поток 318
- Уровни структуры базы данных 309—311
- Условия в Коболе 121—124
- Файл 300—301, 303, 327, 420
 - в Коболе 137
 - — — индексированный 138, 139
 - — — последовательный 137
 - — — с прямым доступом 137, 139
 - — Паскале 196—197
 - — ПЛ/1 241—243
 - — — последовательный 254—257
 - — — с прямым доступом 254, 257
 - — Фортране 179—180
 - инвертированный 378
 - индексированный 376—390
 - индексно-последовательный 353—376
 - многокольцевой 413—432
 - последовательный 343—353
 - прямой 390—413
 - транзакций 344, 350
 - фиктивный 381
- Файл-множество 334—343
- Фиктивный аргумент 274
- Формат входных данных
 - — — фиксированный 14
 - — — переменный 14—15
- Форт 66
- Фортран 63, 66, 160—184
- Функция в Бейсике 80—83
 - — Паскале 193
 - — ПЛ/1 262, 264, 266—271
 - — Фортране 171—172
- Функции внутренние (Фортран) 174—176, 178
 - встроенные в Паскале 194
 - — — ПЛ/1 262—264
- Цепочка 361
- Членство кольца 421—422
- Язык ассемблера и системное программирование 5—61
 - командный 57—61
 - с блочной структурой 64
 - управления заданиями 57
- Языки программирования высокого уровня 62—67
- ON-элемент 259—260, 284

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ РЕДАКТОРА ПЕРЕВОДА	5
Глава 11. ЯЗЫК АССЕМБЛЕРА И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ. <i>С. Гуа</i>	7
11.1. Введение	7
11.2. Незагруженная машина. Начальная загрузка программы	8
11.3. Ассемблер	11
11.4. Перемещающие загрузчики и редакторы связей	39
11.5. Библиотеки	49
11.6. Другие трансляторы	50
11.7. Управление выполнением программ	51
11.8. Команды управления задачами и заданиями	56
Глава 12. ОБЗОР ЯЗЫКОВ ПРОГРАММИРОВАНИЯ ВЫСОКОГО УРОВНЯ <i>Г. Хелмс</i>	62
12.1. Введение	62
12.2. Развитие языков высокого уровня	63
12.3. Описание языков высокого уровня	65
12.4. Резюме	67
Глава 13. БЕЙСИК. <i>Г. Хелмс</i>	68
13.1. Введение	68
13.2. Системные команды	69
13.3. Структура программы	72
13.4. Переменные и константы	72
13.5. Арифметические операторы	75
13.6. Операторы отношения	75
13.7. Логические операторы	76
13.8. Порядок выполнения операций	76
13.9. Управляющие операторы	76
13.10. Подпрограммы	79
13.11. Числовые функции	80
13.12. Строковые функции	81
13.13. Операторы и программы языка ассемблера	83
13.14. Графические операторы	84
13.15. Операторы ввода и вывода	86
13.16. Специальные операторы ввода и вывода	89
13.17. Зарезервированные слова	90

Глава 14. КОБОЛ. А. Таккер-мл.	91
14.1. Введение	91
14.2. Написание Кобол-программ	93
14.3. Операторы Кобола	110
14.4. Соглашения о вводе-выводе	137
14.5. Подпрограммы	145
14.6. Законченные программы	149
14.7. Дополнительные возможности	150
14.8. Сравнение стандартных версий 1968 и 1974 гг.	158
Глава 15. ФОРТРАН. Г. Хелмс	160
15.1. Введение	160
15.2. Формат программы	161
15.3. Ввод с перфокарт и терминала	161
15.4. Константы и переменные	162
15.5. Описание типов	163
15.6. Массивы	164
15.7. Присвоение значений	165
15.8. Арифметические операторы	165
15.9. Операторы отношения	166
15.10. Символ равенства	166
15.11. Управляющие операторы и операторы передачи	166
15.12. Подпрограммы	171
15.13. Внутренние функции	174
15.14. Параметры	177
15.15. Задание имени программы	177
15.16. Манипулирование символьными данными	177
15.17. Операторы эквивалентности	178
15.18. Организация файлов	179
15.19. Ввод-вывод, управляемый списком (потокориентированный)	180
15.20. Форматизованный ввод-вывод	181
15.21. Зарезервированные слова	184
Глава 16. ПАСКАЛЬ. Г. Хелмс	185
16.1. Введение	185
16.2. Структура программы	186
16.3. Идентификаторы	187
16.4. Типы данных	187
16.5. Определения и описания	187
16.6. Массивы	189
16.7. Операторы присваивания	189
16.8. Операторы отношения	190
16.9. Управляющие операторы	190
16.10. Функции и процедуры	193
16.11. Встроенные функции	194
16.12. Ввод и вывод	194
16.13. Упакованные массивы	195
16.14. Множества	196
16.15. Файлы и записи	196
16.16. Зарезервированные слова	198
Глава 17. ПЛ/1. А. Таккер-мл.	199
17.1. Введение	199
17.2. Написание ПЛ/1-программ	201
17.3. Основные операторы	215

17.4.	Соглашения о вводе-выводе	239
17.5.	Подпрограммы	261
17.6.	Законченные программы	279
17.7.	Дополнительные возможности	280
Глава 18.	ДОКУМЕНТАЦИЯ ПО АППАРАТНЫМ СРЕДСТВАМ И ПРОГРАММНОМУ ОБЕСПЕЧЕНИЮ. Д. Эриксон	295
18.1.	Введение	295
18.2.	Требования к хорошей документации	295
18.3.	Типы документаций	296
18.4.	Справочная документация	297
18.5.	Документация для обучения	298
18.6.	Составление документации	298
Глава 19.	БАЗЫ ДАННЫХ И ОРГАНИЗАЦИЯ ФАЙЛОВЫХ СИСТЕМ. Д. Уидерхолд	300
19.1.	Введение	300
19.2.	Файлы	300
19.3.	Вычисления, выполняемые при работе с базой данных	304
19.4.	Иерархическое представление данных	306
19.5.	Современная практика	312
19.6.	Описания	316
19.7.	Привязка	319
19.8.	Классификация операционных систем	321
19.9.	Применение баз данных	325
19.10.	Организация файловых систем	327
19.11.	Файл-множество	334
19.12.	Последовательный файл	343
19.13.	Индексно-последовательный файл	353
19.14.	Индексированный файл	376
19.15.	Прямой файл	390
19.16.	Многокольцевой файл	413
ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ		433

С. Уильям Гна, Аллан Таккер-мл., Джеймс Уидерхолд, Г. Хелмс, Д. Эриксон
КОМПЬЮТЕРЫ, т. 2

Научный редактор Т. Н. Шестакова. Младший научный редактор Л. С. Сысоева.
Художник С. А. Бычков. Художественный редактор Н. М. Иванов. Технический редактор
И. М. Кренделева. Корректор Н. А. Гирия.
ИБ № 5499

Сдано в набор 24.05.85. Подписано к печати 11.02.86. Формат 60×90^{1/16}. Бумага кн.-журн.
Печать высокая. Гарнитура латинская. Объем 13,75 бум. л. Усл. печ. л. 27,50.
Усл. кр.-отт. 27,50. Уч.-изд. л. 26,72. Изд. № 41/4003. Тираж 50 000 экз. Зак. № 640.
Цена 1 р. 70 к.

ИЗДАТЕЛЬСТВО «МИР» 129820, ГСП, Москва, И-110, 1-й Рижский пер., 2.

Ленинградская типография № 2 головное предприятие ордена Трудового Красного Знамени Ленинградского объединения «Техническая книга» им. Евгении Соколовой Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли, 198052, г. Ленинград, Л-52, Измайловский проспект, 29.

