Р. ХЕРШЕЛЬ

TURBO PASCAL 4.0/5.0

Р. Хершель

TURBO PASCAL 4.0 / 5.0

Издание МП «МИК»

1991 г.

ББК 22.18
Х39
УДК 519.682

Хершель Рудольф

Х39 Турбо Паскаль/2-е изд., перераб., — Вологда: МП «МИК», 1991. — 342 с. при участии МП ТПО «Квадрат», г. Москва

Х $\frac{2404010000 - 1}{030015-91}$ Без объявл. ББК 22.18

ТУРБО ПАСКАЛЬ 4.0/5.0
Доктор Рудольф Хершель,
профессор Высшего технического училища, г. Улм

TURBO Paskal 4.0/5.0
Prof/ Dr. Rudolf Herschel

Настоящая монография, включая содержащиеся в ней рисунки, защищена в соответствии с авторским правом. Любое использование ее кроме случаев, предусмотренных авторским правом, без согласования с издательством недопустимо и преследуется по закону. Это касается также копирования, перевода, микрофильмирования, записи и обработки содержимого монографии с помощью электронных систем.

ISBN 5—7382—0001—2

© Издательство «Р.Олденбург», Мюнхен — Вена, 1989
© R. Oldenbourg Verlag Muenchen Wien, 1991

Предисловие редакторов перевода

Вниманию читателей предлагается книга, посвященная современному языку программирования высокого уровня Турбо Паскаль, получившему широчайшее распространение для персональных компьютеров благодаря своему программному окружению, обеспечивающему пользователю максимальный комфорт и высокую скорость работы с языком.

Книга весьма полно описывает язык и методологию его применения. Она ориентирована на пользователя, владеющего базовыми знаниями о работе на IBM - совместимом компьютере.

Большое количество примеров, практически по всем возможностям Турбо Паскаля, позволяет и неподготовленному пользователю включиться в изучение языка и подняться до уровня практического программирования на Турбо Паскале.

С помощью данной книги читатель может изучить всю систему Турбо Паскаля в целом и отдельные особенности ее составных частей.

Книга хорошо структурирована. При описании частей системы Турбо Паскаля автор использует стандартизированную форму записи, короткую по форме и емкую по содержанию. Это позволяет использовать данную книгу в качестве справочника при практической работе с Турбо Паскалем.

Детально рассматривается версия Турбо Паскаля 4.0, но если необходимо, то приводятся отличия для версии 5.0, что обеспечивает возможность использования книги для работы с обеими версиями Турбо Паскаля.

В качестве дополнения к книге имеется дискета в формате MS DOS с исходными текстами программ примеров, рассмотренных в книге. Это дает читателю возможность в минимальные сроки и с минимальными трудозатратами, освоить их и использовать как подпрограммы в своей практической работе.

Книга издана в различных странах мира (Германия, Италия, Франция и др.). Перевод книги выполнялся с целью наиболее точно отразить мысли автора. Из этих соображений все идентификаторы в исходных текстах программ примеров сохранены без изменений.

Книга может быть рекомендована для изучения и использования в работе самому широкому кругу читателей в качестве учебника и справочника по языку программирования Турбо Паскаль 4.0/5.0

С.В. Сорокин, А.В. Тупицын

1. Введение

Турбо Паскаль появился на рынке программных продуктов в 1984 году и, несомненно, совершил революцию в программировании. До того времени предпочтение отдавалось Бейсику - простому, дешевому и легко усвояемому. Паскаль же был аппаратно зависимым, дорогим и сложным в обращении. С появлением Турбо Паскаля положение в корне изменилось. Замечателен в Турбо Паскале не язык, который является лишь языком Паскаля, а программное окружение, в котором теперь можно использовать Паскаль и которое обеспечивает пользователю комфорт и высокую скорость работы. С успехом Турбо Паскаля на рынке программных продуктов связано также развитие многочисленных пакетов, облегчающих применение Паскаля для самых разных целей.

При составлении настоящего руководства автор исходил из того, что читатель имеет известный опыт работы с компьютером. Он должен знать операционную систему своего компьютера и уметь писать программы на каком-нибудь языке программирования. Во всяком случае настоящее руководство не является учебником по программированию, а позволяет лишь получить общее представление о Турбо Паскале, свойствах этого языка программирования и о работе с ним. Естественно, дается большое количество примеров, при выборе которых автор руководствовался стремлением продемонстрировать возможности Турбо Паскаля.

При изучении нового языка программирования речь, в первую очередь, идет о том, чтобы изучить свойства языка, а в случае Турбо Паскаля требуется изучить всю систему Турбо Паскаля в целом. Но на практике оказалось, что читатель предпочитает пользоваться руководством в качестве справочника. При написании данной книги автор постарался учесть такое стремление, а потому описывал Турбо Паскаль в некоторой единой форме записи, сравнительно короткой, но наглядной и абсолютно понятной. Однако, лучше всего использовать настоящую книгу вместе с каким-либо курсом по Паскалю для школы или высших учебных заведений.

Концепция Паскаля была разработана Н.Виртом примерно в 1970 году и Паскаль быстро получил широкое распространение благодаря легкости его изучения, наглядности составленных на нем текстов программ. Поскольку Паскаль послужил основой для разработки других языков программирования, таких как Ада и Модула-2, и поскольку такие языки как Си и ФОРТРАН77 содержат аналогичные Паскалю элементы языка, знание Паскаля является солидной базой для изучения других языков программирования.

В середине 70-х годов была сделана попытка разработать международный стандарт на Паскаль. В результате в 1982 году появился стандарт ИСО 7185 (Международной организации по стандартизации). Здесь следует различать этап 0 и заключительный

этап 1. В Германии стандарт был опубликован в 1983 году в качестве стандарта ДИН 66256 на немецком языке [6]. Описание стандартного языка Паскаль на уровне учебного пособия появилось в 1987 году [5]. Основные различия между Турбо Паскалем и стандартным Паскалем указываются в тексте.

Расскажем коротко о создании Турбо Паскаля. После просуществовавшей сравнительно недолго и не получившей широкого распространения версии 1 в середине 1984 года появляется версия 2, распространение которой пошло стремительными темпами. К осени 1985 года появляется версия 3.0, отличающаяся от версии 2 следующими особенностями: компилятор и редактор стали работать существенно быстрее, появилась возможность передачи параметров в программу с помощью команды RUN, стал возможным вызов MS-DOS из программы, стала более удобной работа с файлами [4].

С начала 1988 года начинает распространяться версия 4.0. Здесь Турбо Паскаль представлен в совершенно новой форме. Не только становится еще более быстрым компилятор, но и появляется совершенно новое программное окружение. Существенно то, что компилятор стал встроенным, так что появилась возможность разбиения программы на части (так называемые модули), компилируемые по отдельности. Поскольку каждый такой модуль имеет собственный сегмент кодов, программы могут занимать и более 64 Кбайт. Осенью 1988 года появилась версия 5.0 с еще более развитым программным окружением. Здесь заслуживает внимания прежде всего встроенный отладчик. Модули могут теперь описываться как оверлейные, причем программа также может занимать более 64 Кбайт.

В то время как версии 1, 2 и 3 Паскаля последовательно сменяли друг друга, версии 4 и 5 предлагаются одновременно с версией 3. Это имеет определенный смысл, так как версия 3 дешевле, требует меньше памяти и проще в эксплуатации. Начинающим на долгое время хватит тех возможностей, которые обеспечиваются версией 3. И лишь при наличии достаточного опыта работы с Турбо Паскалем и при реализации задач, требующих особо сложных программных решений, стоит обратиться к версиям 4 и 5. Различия между версиями 3.0 и 4.0 Турбо Паскаля значительны, в то время как версии 4.0 и 5.0 чрезвычайны близки. В 1989 году появилась версия 5.5, позволяющая перейти к объектноориентированному программированию.

В настоящем руководстве описывается версия 4.0. Если это необходимо, в конце главы дается примечание применительно к версии 5.0, так что руководство пригодно при работе с обеими версиями.

Турбо Паскаль может использоваться в большинстве существующих для персональных компьютеров операционных систем: хотя

подавляющее большинство читателей, безусловно, использует IBM-совместимые компьютеры с операционными системами PC-DOS/MS-DOS, этот вариант в тексте каждый раз оговаривается.

К настоящему руководству имеется дискета с программами в формате MS-DOS. Из соображений стоимости эта дискета прилагается не к каждой книге, поскольку, вероятно, она нужна не каждому читателю. На дискете записаны примеры, приведенные в этой книге.

2. Принципы Паскаля и его программное окружение

Прежде чем в главе 4 будут подробно описываться особенности Паскаля, поясним на одном примере, каковы принципы построения этого языка и в каком программном окружении может использоваться Турбо Паскаль.

Пример 2.1

```
(* Это комментарий*)
program willkommen(input,output);              (* 1  *)
uses crt ;                                     (* 4  *)
(**************Соглашения**************)       (* 2  *)
type wort = string[20]                         (* 2a *)
var name : wort;                               (* 2b *)

procedure eingabe(var name : wort);            (* 2c *)
begin
  clrscr;  (*погасить экран*)                  (* 2d *)
  write('Ваше имя, пожалуйста:');
  readln(name);
end;

procedure starline(n:integer);                 (* 2c *)
var i:integer;
begin
   for i := 1 to n do write('*');
   writeln;
end;

procedure zaehle(n:integer);                   (* 2c *)
var i:integer;
begin
   for i := 1 to n-1 do begin
                         write(i:3);
                         delay(500);
                        end;
   writeln; writeln('Здесь было что-то еще...');
   delay(2000);                                (* 2d *)
   writeln; writeln(n:3*n);
end;
```

```
procedure begruessung(name : wort);            (* 2c *)
var c:char; n : integer;
begin
  clrscr;(*Погасить экран*)                     (* 2d *)
  starline(50);
  lowvideo; writeln('Приветствую Вас');        (* 2d *)
  highvideo; writeln(name:20);
  starline(50);
  writeln('Могу я Вам еще что-то перечислить?');
  writeln(' J/N');
  read(c);
  case c of
  'j', 'J' : begin writeln('Пожалуйста, что дальше?');
                   readln(n);
                   zaehle(n);
                   delay(1000);
                   writeln('Увидимся позже');
                   writeln('Пока!');
             end;
  else        writeln('Только не это!');
             writeln('Пока!':40);
             end (* case *);
end;  (*Приветствие*)

(*************Операторы***************)        (* 3 *)
begin
  eingabe(name);                                (* 3a *)
  begruessung(name);                            (* 3b *)
end.
```

Написанная на Паскале программе состоит из трех частей: заголовок программы (*1*), описательная часть (соглашения) (*2*) и собственно программные операторы (исполняемая часть) (*3*).

Заголовок отмечает начало программы, присваивает ей некоторое имя (даже если такое имя бессмысленно) и описывает средства, через которые можно взаимодействовать с этой программой. В приведенном выше примере программе присвоено имя willkommen, а взаимодействовать с программой можно через input (это клавиатура) для ввода информации и через output (это дисплей) для вывода информации.

Важнейший принцип Паскаля: все используемые в программе имена должны быть описаны до их употребления. Отсюда строгое деление на описательную часть (*2*) и исполняемую часть (*3*). В исполняемой части, содержащей собственно операторы, должны

использоваться лишь те имена, которые описаны в части соглашений:

- процедуры eingabe ("ввод"), begruessung ("приветствие"), zaehle ("числа") и starline ("строка звездочек"),
- переменная name ("имя").

В описательной части (*2*) заданы:

- тип wort ("слово") (*2a*)
- наименование переменной name (*2b*)
- имена процедур eingabe, begruessung, zaehle и starline (*2c*), причем begruessung использует описанные ранее процедуры starline и zaehle.

В качестве имен должны использоваться имена, позволяющие точно определить, какие имена введены пользователем (к ним относятся все перечисленные выше имена), а какие имена стандартные (к ним относятся наименования типов переменных integer, char или имена процедур clrscr, read, write, lowvideo, highvideo, delay), чтобы пользователю не пришлось заниматься их описанием.

Часть программы, содержащая исполняемые операторы (*3*), позволяет с помощью операторов описать, какие операции над описанными объектами должна выполнить программа. В нашем примере такие операции выполняются с помощью процедур (*3a*) eingabe и begruessung.

Для написания программ программное окружение Турбо Паскаля содержит удобный редактор. Написанный на языке Паскаль текст называется исходным текстом или исходной программой. Исходный текст должен иметь атрибут .PAS. Компилятор должен сгенерировать из исходной программы объектные коды. Эта операция выполняется в два этапа. Сначала компилятор генерирует неполные, непригодные для исполнения объектные коды, содержащие ссылки на объекты, неописанные в исходном тексте; это могут быть, например, стандартные наименования типов переменных (integer, char) и процедур (clrscr, read, write, delay и пр.). Они помещаются в модулях библиотеки Турбо Паскаля TURBO.TPL, а компоновщик должен дополнить сгенерированные компилятором объектные коды кодами из библиотеки TURBO.TPL.

В принципе компилятор должен работать с полным исходным текстом. Но для больших программ возникает необходимость компиляции отдельных частей программы и передачи таких скомпилированных частей компоновщику. В принципе это возможно. Такие части программы называются модулями, а сгенерированные компилятором их объектные коды имеют атрибут .TPU (Turbo Pascal Unit). Библиотека TPL есть не что иное как некоторое собрание модулей TPU. Если в программе используются модули TPU, на это после заголовка программы с помощью uses делается ссылка. В нашем примере это сделано в строке (*4*). В модуле CRT.TPU определяются использованные в (*2d*) процедуры

clrscr = clear screen очистить экран
delay(2000) = задержка на 2000 мс
lowvideo = половинная яркость
highvideo = увеличенная яркость
и потому в строке (*4*) стоит uses crt;.

Если Вы уже работали с версией 3, то должны были убедиться в том, что не все написанные для версии 3 программы работают под версией 4. Если Вы столкнетесь с такими трудностями, воспользуйтесь оператором uses turbo3; в этом модуле описаны некоторые обычные для версии 3 имена.

Кроме того, существуют опции компилятора и директивы компилятора. И опции и директивы оказывают определенное воздействие на работу компилятора. Опции могут задаваться с помощью описанного в следующей главе основного меню, директивы помещаются в исходный текст и имеют форму специальных комментариев. Информацию о них Вы найдете в приложении D.

3. Программное окружение Турбо Паскаля

Эта глава должна дать читателю общее представление о программном окружении Паскаля, чтобы он сразу же получил возможность приступить к работе с Паскалем. Паскаль позволяет использовать две версии: с интегрированным, выполненным в технике меню программным окружением (TURBO.EXE) и в виде интерпретатора командных строк (TPC.EXE). В этой главе речь пойдет лишь о TURBO.EXE. Версия TPC рассматривается в главе 20.

3.1 Инсталляция

Приступая к работе с Паскалем нужно прежде всего создать программное окружение, в котором должен работать Турбо Паскаль. При этом различают режим работы с двумя накопителями на гибких магнитных дисках и режим работы с винчестером (жестким диском). Можно работать и с одним лишь дисководом, но всерьез этим никто заниматься не будет.

Работа с двумя накопителями на гибких дисках

Первая дискета (дисковод А) содержит такие файлы:
MS-DOS (т.е. отформатированную согласно FORMAT A:/S, примерно 45 Кбайт)
TURBO.EXE (интегрированное программное окружение примерно 115 Кбайт)
TURBO.TPL (стандартные библиотеки примерно 37 Кбайт)
TURBO.HLP (пояснения-"подсказки" примерно 125 Кбайт)

Все эти файлы занимают в совокупности около 300 Кбайт. Итак, остается еще место для нескольких вспомогательных программ. Тогда вторая дискета (дисковод B) могла бы содержать написанные на Паскале программы. Файл подсказок TURBO.HLP занимает так много места лишь в том случае, если пояснения даются на немецком языке. Исходная английская версия занимает лишь 85 Кбайт!

Работа с винчестером

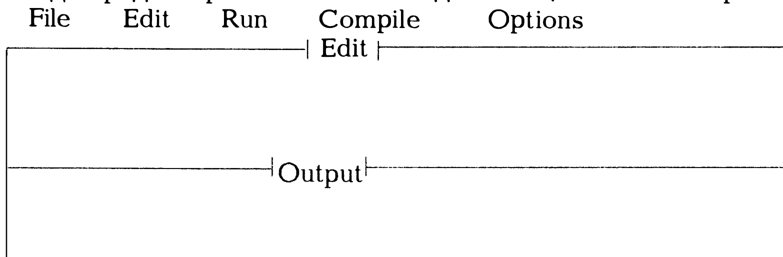
Создайте в корневом каталоге каталог TP или TP4 (или назовите этот каталог иначе, если Вам того хочется). Скопируйте в него три дискеты с версией 4.0 Турбо Паскаля.

Кроме того, существует установочная программа TINST.EXE, в которой действуют специальные соглашения относительно каталогов, команд редактора, режимов работы с экраном и цветовой гаммы экрана. Поскольку Турбо Паскаль устанавливается на ПЭВМ достаточно легко, а программа TINST.EXE по ходу своей работы выдает все необходимые пояснения, не будем на этом останавливаться подробно.

3.2. Интегрированная среда TURBO.EXE

После запуска по команде TURBO, Турбо Паскаль выдает на экран основное меню (рис. 3.1).

В верхней половине экрана можно генерировать и обрабатывать тексты, т.е. написанные на Паскале программы. В нижнюю часть экрана выводятся сообщения компилятора, компоновщика или программы. Нажав клавишу F5, получите возможность использовать для редактирования или вывода сообщений весь экран.



F1 Help F2 Save F3 Load F5 Zoom F6 Edit F9 Make F10 Main Menu

Рис. 3.1. Основное меню Турбо Паскаля версии 4.0

В строке заголовка перечисляются основные функции системы. Выбрать нужную функцию можно либо с помощью курсора (в этом случае актуальная функция высвечивается в инверсном представлении) либо путем ввода начальной буквы наименования функции. Почти для каждой функции при ее выборе на экране появляется окно со вспомогательным меню, предоставляющим пользователю возможность сделать следующий выбор (а может быть, и вызвать на экран новое окно). В любом случае пользователь может пользоваться клавишей курсора или выбирать нужную функции путем ввода начальной буквы ее имени.

Для того чтобы быстро перемещаться по окнам с соблюдением их иерархии, не сталкиваясь с необходимостью каждый раз закрывать все появившиеся на экране окна, существуют так называемые функциональные клавиши (hot keys - "горячие" клавиши), с помощью которых можно легко перейти в нужный режим, не заботясь при этом о том, в каком режиме Вы находитесь в данный момент. Прежде чем приступить к описанию предлагаемых в основном меню возможностей, перечислим функциональные клавиши и их назначение (рис. 3.2).

Функциональные клавиши	Назначение клавиши
F1	Вызов подсказки
F2	Сохранение рабочего файла редактора
F3	Загрузка файла
F5	Увеличение/уменьшение поля, отведенного для редактирования/вывода
F6	Переключение с одного окна редактирования/вывода на другое
F9	Создание EXE-файла (исходная программа в виде нескольких файлов)
F10	Вызов основного меню
Alt-F1	Удаление с экрана текста вызванной последней подсказки
Alt-F3	Выбор имен только что обработанных исходных файлов и загрузка одного из них в редактор
Alt-F5	Переключение с окна редактирования на окно вывода или наоборот
Alt-F9	Компиляция файла в редакторе
Alt-F10	Вывод на экран номера версии
Alt-C	Вызов меню компилятора
Alt-E	Вызов редактора
Alt-F	Вызов меню файлов
Alt-O	Вызов меню опций

Alt-R	Запуск программы (возможно, вместо компиляции и компоновки)
Alt-X	Выход из Турбо Паскаля (возврат в MS-DOS)
Ctrl-F6	Следующее окно

Рис. 3.2. Функциональные клавиши, используемые в версии 4.0

Функциональная клавиша F1 работает в зависимости от ситуации, т.е. в зависимости от того, в каком из сгенерированных в иерархической последовательности окон Вы находитесь. При нажатии клавиши F1 выдается соответствующая этому окну подсказка. Выход из режима подсказки путем нажатия клавиши Escape. Такой механизм поддержки пользователя настолько удобен, что в дальнейшем изложении мы ограничимся лишь самым общим описанием. Если, например, с помощью клавиши F1 вызван режим подсказки, повторное нажатие F1 приведет к запросу "подсказки о режиме подсказки", в результате чего пользователь получит о соответствующем программном окружении информацию, более подробную, чем та, которую можно дать в рамках настоящей монографии. При этом иерархия окон будет отмечаться косой чертой /, как и в случае указания пути доступа к файлу.

Функции основного меню и важнейшие подфункции и вспомогательные меню:

File	Загружает и сохраняет файлы, работает с каталогами, вызывает MS-DOS и завершает работу с Турбо Паскалем
File/Load	Загружает в редактор файл с дискеты
File/Pick	Выводит на экран имена восьми обработанных последними файлов и обеспечивает возможность загрузить их в редактор
File/New	Создает новый файл NONAME.PAS
File/Save	Записывает файл из редактора на дискету под тем же именем (как и F2)
File/Write to	Записывает файл из редактора в файл на дискете, запрашивая при этом имя файла
File/Directory	Распечатывает список файлов текущего каталога
File/Chang Dir	Позволяет перейти в новый каталог
File/OS shell	Позволяет вызвать MS-DOS, не выходя из Турбо Паскаля
File/Quit	Выход из Турбо Паскаля
Edit	Переход в редактор. Содержимое рабочего файла редактора выдается в окно редактирования, где его можно обрабатывать. Редактор описан в главе 24. Информация о работе

	в режиме редактирования выдается при нажатии клавиши F1
Run	Находящаяся в редакторе программа, если это не было сделано ранее, компилируется, компонуется и запускается
Compile	Компилирует программу
Compile/Compile	Компилирует находящийся в редакторе файл и при наличии ошибок возвращается в редактор
Compile/Make	Компилирует и компонует разделенную на несколько файлов программу (см. гл. 19.5) (Действует аналогично F9)
Compile/Build	Компилирует и компонует все задействованные в некотором проекте файлы (см. гл.19.5).
Compile/Destination	Дает возможность выбора записи результата компиляции в оперативную память или на дискету
Compile/Find error	Выдает место обнаружения ошибки при прогоне программы в формате seg:ofs (сегмент: смещение). При возврате в редактор это место в исходном тексте соответствующим образом выделяется (см. гл. 23)
Compile/Primary file	Осуществляет выдачу файла .PAS, к которому происходит обращение при формировании и компоновке. Если нет специальной оговорки, файл помещается в редактор
Compile/Get info	Выводит информацию (через исходный текст .PAS и объектные коды .EXE (соответственно .TPU)
Options	Это важнейшее меню, позволяющее установить параметры программной среды
Options/Compiler	Вызывает новое меню, позволяющее управлять компиляцией. Здесь можно выбрать многие подфункции, такие как контроль по прерываниям, проверка состояния стека, контроль за процессами ввода/вывода, параметрами и пр.(см. также описание директив компилятора в приложении D)
Options/Environment и Options/Directories	описывают программное окружение Турбо Паскаля, т.е. указывает, где компилятор и компоновщик должны брать необходимые файлы, следует ли перед запуском сохранить содержимое рабочего файла редактора и генерировать ли при сохранении резервную копию
Options/Parameters	Позволяет задавать параметры, используемые при запуске программы
Options/Save Options	Все варианты компилятора и компоновщика, как и данные о каталогах в Options/ Enviroment помещаются в файл конфигурации

	TURBO.TP, который используется затем при вызове TURBO.EXE
Options/Load Options	Загружает сгенерированный ранее с помощью Options/Save Options файл конфигурации

В заключение заметим, что с помощью системы подсказок (F1) выдается информация не только о программном окружении Турбо Паскаля, но и о самом Паскале. Находясь в режиме редактирования, можно с помощью клавиш Ctrl-F1 вызвать на экран информацию о стандартных процедурах и функциях Паскаля и модулях библиотеки, а при работе с версией 5.0 и о служебных словах, представленных на рис. 4.1. Причем, если курсор устанавливается на имя соответствующей процедуры или функции, выдаются пояснения, касающиеся только этой процедуры или функции. Для версии 5.0 то же самое справедливо и для стандартных типов данных. Вообще говоря, клавиши Ctrl-F1 при изучении возможностей Турбо Паскаля являются превосходным вспомогательным средством.

Заключение

В очень многих случаях можно обходиться следующими несколькими приемами.

1. Запуск с помощью C> TURBO.

2. Загрузка исходной программы в редактор с помощью функциональной клавиши F3 (аналогичного результата можно достигнуть с помощью строки

C>TURBO путь_доступа_имя_исходной_программы).

3. Использование под исходный текст экрана осуществляется по нажатию клавиши F5.

4. Генерация и изменение исходной программы с помощью редактора.

5. Компиляция, компоновка и запуск находящейся в редакторе программы путем нажатия клавиш Alt-R .

6. Возврат в редактор с помощью клавиш Alt-E с последующей обработкой исходного текста. (Вновь клавиши Alt-R и т.д.).

7. Сохранение исходной программы с помощью клавиши F2.

8. Выход из Турбо Паскаля путем нажатия клавиш Alt-X.

Примечания относительно версии 5.0

Версия 5.0 имеет встроенный отладчик, а потому основное меню содержит два дополнительных меню (рис. 3.3), описанных вместе с отладчиком в разделе 23.2. Но из-за отладчика изменилось и функция RUN основного меню. Теперь команда Run открывает новое дополнительное меню, а в версии 4.0 Run была командой запуска программы. Теперь же программа запускается по команде

Run/Run. Появились и новые функциональные клавиши (см. рис. 3.4), кроме представленных на рис. 3.2.

После запуска Турбо Паскаля версии 5.0 на экране появится изображенное на рис. 3.3 меню. Экран делится на два окна для редактирования и контроля (Watch). Естественно, имеется и окно вывода как для версии 4.0 (рис. 3.1). Нужно лишь его вызвать.

Если в изображенном на рис. 3.3 окне редактирования записать программу, выводящую, например, три числа, и запустить ее с помощью клавиш Ctrl-F9, то результата на экране мы не получим, поскольку эти три числа запишутся в окно вывода. Нажав клавиши Alt-F5 (рис. 3.2), увидим на экране результат работы программы. Если кому-то это покажется слишком сложным, можно и при использовании версии 5.0 иметь на экране изображенное на рис. 3.1 меню.

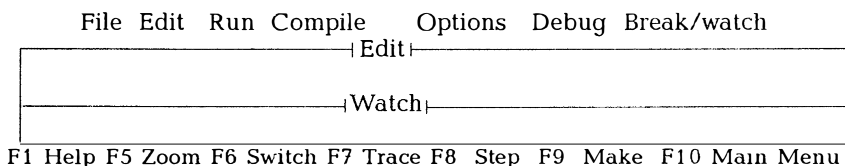


Рис. 3.3. Основное меню Турбо Паскаля версии 5.0

После запуска версии 5.0 с помощью клавиши F6 вызывается окно контроля, которое затем нажатием клавиш Alt-F6 заменяется окном вывода. Теперь экран, как и для версии 4.0 делится на окно для редактирования и окно вывода. Если теперь запустить программу, записанную в окне редактирования, нажав сочетание клавиш Ctrl-F9, результат работы программы будет сразу же выводиться в окно вывода.

F4	Выполнение программы до места, помеченного курсором (и запуск отладчика)
F7	Выполнение одного шага программы
F8	То же, что и F7, но без пошагового выполнения вызванной функции
Shift F10	Вывод на экран номера версии (для 4.0 Alt-F10)
Ctrl-F2	Возврат в отладчик
Ctrl-F3	Отладчик выводит на экран содержимое стека вызванной функции
Ctrl-F4	Отладчик вычисляет значение выражения
Ctrl-F8	Установить/исключить точку прерывания для отладчика
Ctrl-F9	Компиляция и запуск программы (для версии 4.0 клавиши Alt-R)
Alt-F6	Смена текущего окна (окна редактирования на окно вывода или окно контроля)

Alt-B	Вызов меню функции Break/Watch
Alt-D	Вызов меню функции Debug
Alt-R	Вызов меню функции Run

Рис.3.4. Дополнительные функциональные клавиши для версии 5.0 Турбо Паскаля

Такое разбиение экрана на окна может оказаться для начинающих сложным и потребовать определенного времени для привыкания. Повторим еще раз важнейшие моменты работы в многооконном режиме.

Версия 4.0:

Имеются лишь два окна: окно редактирования и окно вывода (см. F5 и F6). Если записанная в окне редактирования программа компилируется и запускается с помощью клавиш Alt-R, экран переключается на окно вывода, в котором появляются результаты работы программы. После завершения работы программы на экране появится сообщение "Press any key to return to turbo pascal" ("Нажмите любую клавишу для возврата в Турбо Паскаль"). После нажатия какой-либо клавиши экран вновь будет переключен на окно редактирования.

Версия 5.0:

Имеются три окна: окно редактирования, контроля и вывода. После запуска программы в окне редактирования с помощью клавиш Ctrl-F9, экран переключается на окно вывода, где появляются выводимые программой результаты. После завершения работы программы экран вновь переключается на окно редактирования. Выведенные программой данные будут теперь не видны. Нажав клавишу F6 (переключение на окно контроля) или клавиши Alt-F6 (переключение на окно вывода) можно еще раз просмотреть полученные в ходе работы программы результаты. Тот, кому такой способ покажется затруднительным, может просто завершить программу строкой

```
readln;
```

Тогда программа остановится в конце своей работы и будет ожидать нажатия любой клавиши. Вплоть до нажатия клавиши окно вывода будет оставаться на экране.

4. Структура программ на Паскале

В главе 2 были описаны основные особенности языка Паскаль, а в главе 3 - программное окружение Турбо Паскаля. Теперь рассмотрим эти вопросы подробнее. Приведем еще один пример.

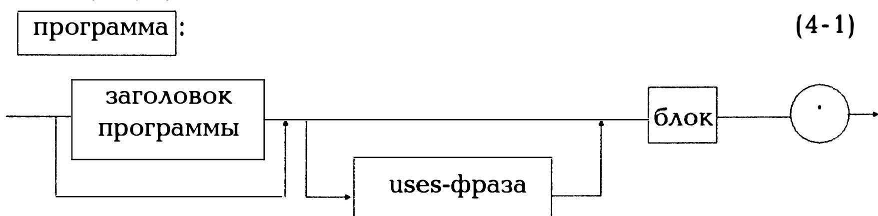
Пример 4.1

```
program lieblingszahl_erraten (input, output);
uses crt;
(*Эта программа должна делать следующее:
  - дружески поприветствовать Вас,
  - предложить Вам угадать ее любимое число,
  - сообщить результат угадывания такого числа.*)

(************************Описания**********************)
const lieblingszahl = 367;
var zahl, anzahl : integer;

procedure begruessung;
begin
    clrscr;
    writeln('Привет партнеру!');
    writeln('Я рад, что Вы состязаетесь');
    writeln('со мной!');
    writeln;
    writeln('Отгадайте мое любимое число!');
    writeln('Для справки: оно меньше тысячи');
end;

procedure beurteilung(a:integer);
begin
    case a of
      1..9  : writeln('Это случайность');
      10    : writeln('Чудесно!');
      11,12 : writeln('Очень хорошо!');
      13,14 : writeln('Посредственно');
      else writeln('Неудачно') end;
end;

begin   (************Исполняемая часть************)
   begruessung;
   anzahl := 0;
   repeat
      readln(zahl);
      if zahl < lieblingszahl
         then writeln('Больше');
      if zahl > lieblingszahl
         then writeln('Меньше');
      anzahl := anzahl +1;
   until zahl = lieblingszahl;
   beurteilung(anzahl);
   writeln('Пока!');
end.
```

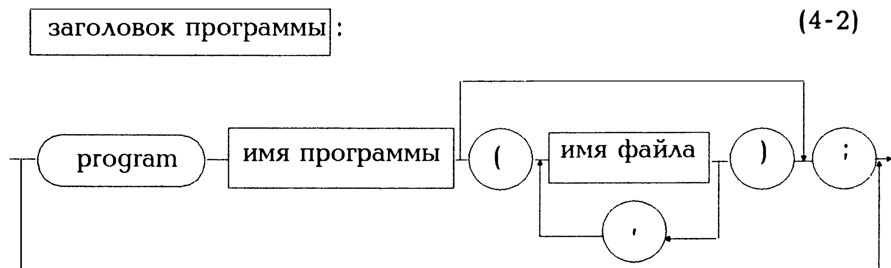
Если хотят описать некий язык программирования, хорошо бы различать синтаксис и семантику. Синтаксис языка - это грамматика, то есть правила, следуя которым можно сформировать корректный для данного языка текст. Синтаксис Паскаля устанавливает, как можно писать программы на Паскале. Синтаксически правильная программа компилируется без ошибок. Семантикой является содержание синтаксически корректной формулировки. Описание синтаксиса сравнительно просто. Для этого используются так называемые BNF (Backes-Naur-Form - форма Бэкуса-Наура, текстуальное описание) и синтаксические диаграммы (в виде графического описания).В данной книге используются синтаксические диаграммы, как наиболее наглядные. Тем, кто плохо понимает, что такое синтаксическая диаграмма, рекомендуем обратиться к приложению А, в котором они приведены. Ниже в тексте синтаксические диаграммы используются для пояснения определенных положений. Не всегда целесообразно приводить сразу же самую общую форму. Итак, общее описание дано в приложении А.

Семантику вряд ли можно описать так же точно и кратко. Для описания смысла формулировок Паскаля воспользуемся немецким разговорным языком.

Структуру программы можно представить следующей схемой:



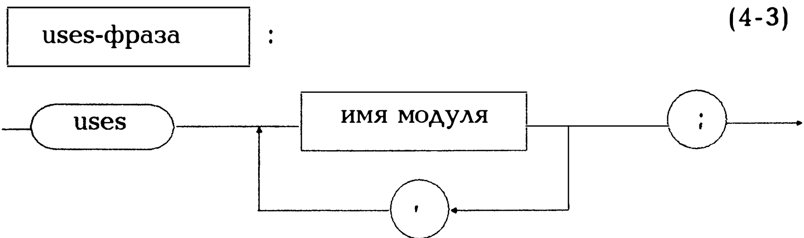
Согласно (4.1) заголовок программы является лишь "декорацией". Мы будем приводить заголовок всегда, поскольку он позволяет ясно увидеть начало программы и использование его предписано стандартом Паскаля. Вот описание заголовка программы:



Итак, пример 4.1 можно было начать тремя различными способами:

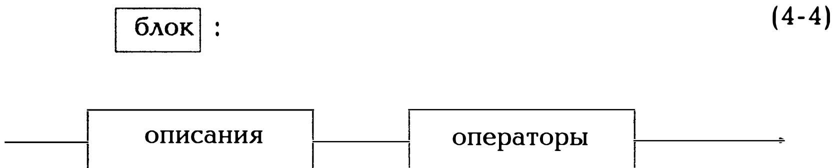
без заголовка,
program lieblingszahl_erraten;
program lieblingszahl_erraten (input, output);

В этой книге всюду будет использоваться форма "program имя_программы;". Имя программы для выполняемых программой операций никакого значения не имеет. Нельзя лишь использовать его внутри программы для каких-то других целей. Возможно приведенные затем имена файлов описывают программное окружение, с которым взаимодействует программа. За заголовком программы могут следовать так называемые uses-фразы, имеющие следующую форму:



Такая фраза указывает на используемые в программе, но описанные в другом месте объекты. Это связано с важнейшим понятием модуля, рассмотренным в главе 19. В примере 4.1 uses crt; говорит о том, что может использоваться процедура clrscr (= clear screen = очистить экран), описанная в модуле crt.

Согласно схеме (4-1) после заголовка программы и uses идет собственно программа в виде некоторого блока:



Структура блока отвечает принципу Паскаля "Прежде чем использовать величины, все их нужно описать". Под нейтральным названием "величина" имеется в виду следующее:

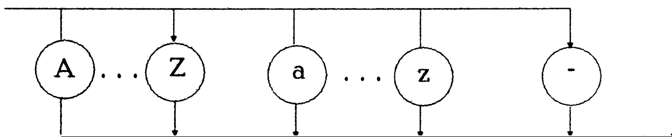
(4-5)

В приведенной в примере 4.1 программе использованы описания:

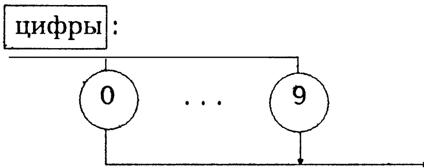
- меток	(таковых нет),
- констант	(lieblingszahl),
- типов данных	(стандартное наименование типа integer),
- переменных	(zahl, anzahl),
- процедур	(begruessung, beurteilung, стандартные процедуры writeln, readln, clrscr),
- функций	(нет).

Этим шести объектам даются имена, а потому следует сказать, как эти имена образуются.

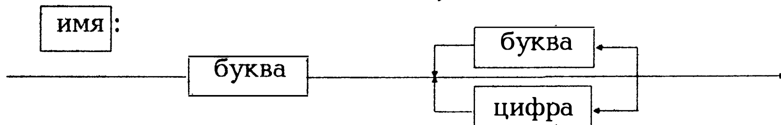
буквы : (4-6)



(4-7)



(4-8)



Имя или идентификатор является центральным понятием языка программирования. С его помощью можно именовать, идентифицировать объект языка. В нашем примере имеют имя:

константа lieblingszahl, переменная anzahl, процедура beurteilung и (поскольку символ подчеркивания также является буквой для языка программирования) программа lieblingszahl_erraten ("отгадайте_любимое число").

Итак, имя начинается с буквы, далее могут следовать буквы или цифры. Причем большие и маленькие буквы не различаются: ZAHL,Zahl и zahl обозначают один и тот же объект. Напротив, NEW-ULM и 12a недопустимо использовать в качестве имени, поскольку здесь либо присутствует отличный от буквы и цифры символ, либо имя начинается не с буквы. Обычно "умлаут" и "эсцет" в идентификаторах не употребляются (поэтому в нашем примере слово "приветствие" пишется begruessung!). Длина имени ограничена максимальной длиной строки, т.е. 127 символами, причем значащими из них являются только первые 63 символа.

Согласно схеме (4-5) порядок следования описаний произвольный. Это существенное отличие от стандартного Паскаля, где после меток, констант, типа данных, переменных должны идти в произвольной последовательности процедуры и функции. Но и в Турбо Паскале действует определенное правило:

Имя можно использовать лишь после того, как оно описано в тексте программы. Исключение встречается лишь в случае рекурсии.

В нашей книге мы будем придерживаться правил стандартного Паскаля и отклоняться от них лишь тогда, когда возможности Турбо Паскаля обеспечивают значительное преимущество перед стандартным Паскалем.

Существует целый ряд служебных (зарезервированных) слов, которые нельзя использовать в качестве имен(рис.4.1).

absolute	end	inline	procedure	type
and	external	interface	program	unit
array	file	interrupt	record	until
begin	for	label	repeat	uses
case	forward	mod	set	var
const	function	nil	shl	while
div	goto	not	shr	with
do	if	of	string	xor
downto	implementation	or	then	
else	in	packed	to	

Рис. 4.1. Служебные слова Турбо Паскаля 5.0

Естественно, эти слова также можно писать и большими буквами, значение их от этого не изменится.

Наряду с этим есть ряд стандартных имен, имеющих некоторое фиксированное, раз и навсегда определенное значение, таких как, например, integer и read. Подчеркнем принципиальные отличия служебных слов от стандартных имен. Каждое неверное использование служебного слова приводит к ошибке при компиляции. Каждое новое определение стандартного имени просто отменяет его первоначальное значение. Так например, если по недосмотру сократить Donnerstag ("четверг") до do, такое сокращение будет воспринято при компиляции как оператор do, что приведет к ошибке. Если некоторой переменной присвоить имя random (var random:real), то просто нельзя будет больше использовать стандартную функцию random, т.е. генератор случайных чисел.

Программу на Паскале можно писать в произвольном формате, лишь бы ее было удобно читать. Следует лишь помнить, что элементы языка (имена, константы, служебные слова) нельзя писать слитно, они должны отделяться один от другого разделителями (пробелом, переводом строки или комментарием).

Строка программы может иметь максимальную длину 127 символов. Если строка будет длиннее, ее часть, выходящая за пределы 127 символов, будет компилятором игнорироваться. В редакторе Турбо для строки отводится максимум 127 символов. Если же текст программы подготовлен с помощью другого редактора, при считывании его с помощью Турбо Паскаля автоматически осуществляется перевод строки и выдается соответствующее сообщение.

Комментарии заключаются в фигурные скобки { } или обрамляются круглыми скобками со звездочками (* *):

{Это комментарий,}   (*и это тоже.*)

При вложении должна использоваться другая пара ограничителей:

{Это (*совершенно специфический*) комментарий.}

Практическое использование такой конструкции целесообразно тогда, когда всюду по тексту встречаются (* *). Тогда с помощью фигурных скобок можно при компиляции просто пропустить кусок исходного текста.

Комментарии можно помещать всюду, где есть разделитель. Комментарий не должен начинаться с символа $, так как конструкция {$...} и (*$...*) относится к директивам компилятора и при компиляции приводит к вполне определенным результатам (см. приложение D).

5. Простые стандартные типы данных

Тип данных задает область значений, к которой может принадлежать принимаемое переменной данного типа значение. Вначале покажем, как следует записывать константы этого типа. Каждый тип данных имеет собственное наименование. Обращение к известным элементарным типам данных выполняется стандартным образом и пользователю нет необходимости описывать их специально. Если же пользователь захочет ввести какой-то свой тип данных, он должен указать это в той части программы, где описываются типы данных. Как это делается, поясняется в главе 12. Пока же мы будем использовать стандартно предлагаемые пользователю простейшие типы данных.

5.1. Целые числа

Целые числа представлены в ПЭВМ с помощью двоичных чисел. Двоичное число, состоящее из n бит, можно интерпретировать двумя способами:

n бит без знака числа: $0...2^n-1$

n бит со знаком числа: $-2^{n-1}...2^{n-1}-1$

На рис.5.1 представлено несколько типов целых чисел.

Наименование типа	Формат	Область значений
byte	8 бит без знака	0..255
word	16 бит без знака	0..65535
shortint	8 бит со знаком	-128..127
integer	16 бит со знаком	-32768..32767
longint	32 бита со знаком	-2147483648.. 2147483647

Рис. 5.1. Формат целых чисел

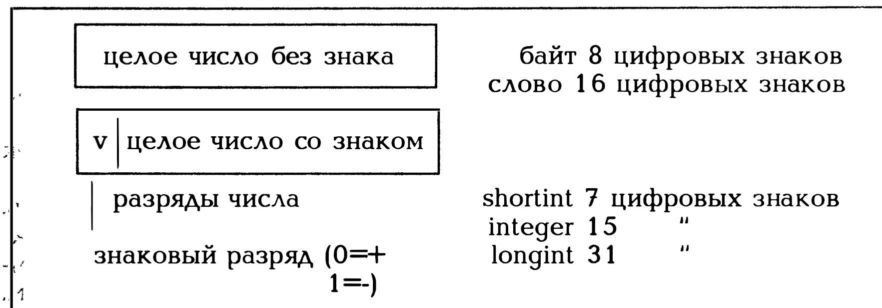


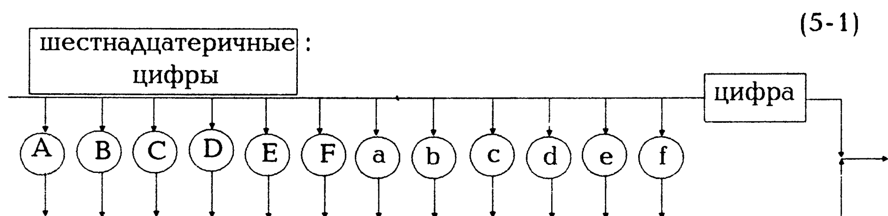
Рис. 5.2. Внутреннее представление целых чисел

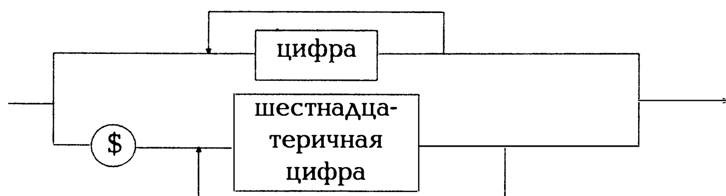
Покажем читателю, как представлены в ПЭВМ несколько значений типа shortint. В представлении отрицательной величины присутствует знаковый разряд 1, а значащие разряды записываются в дополнительном двоичном коде:

```
01101101 = +109
10010011 =  -109
11111111 =  -1
11111110 =  -2
10000000 =  -128 (наименьшее отрицательное число)
01111111 = +127 (наибольшее положительное число)
```

Как отдельные двоичные разряды целого числа обрабатываются программой, можно увидеть в примере 9.4.

Константы такого типа могут быть десятичными и шестнадцатеричными.





Константы в десятичной форме записи могут иметь знак:

234   -4567   +1988

Вот примеры констант такого типа в шестнадцатеричной форме записи:

$0012 $2AB5

Константы в шестнадцатеричной форме записи знакового разряда не имеют.

Если с помощью read(x) ввести некоторую константу, которая в двоичном представлении будет меньше $FFFFFFFF, для x : byte и x : shortint в качестве значение будут браться последние (справа) 8 бит, для x : word и x : integer - последние 16 бит, а для x : longint - все 32 бита. Если введенная постоянная больше указанного двоичного числа, это приведет к ошибке.

Чтобы не вдаваться в детали, для всех указанных выше пяти типов целых чисел в настоящей книге будет использоваться собирательное название "целое" (integer).

5.2. Вещественные числа

Дробные и все вещественные числа представляются внутри ЭВМ в полулогарифмической форме, состоящей из мантиссы и экспоненты. Мантисса содержит значащие цифровые разряды, а экспонента задает числовой диапазон. Существуют следующие типы вещественных чисел (E обозначает 10 в степени):

Тип данных	Десятичные разряды	Диапазон значений	Кол-во битов
single	7-8	1.5E-45 ... 3.4E38	32
real	11-12	2.9E-39 ... 1.7E38	48
double	15-16	5.0E-324 ... 1.7E308	64
extended	19-20	1.9E-4951 ... 1.1E4932	80
comp	19-20	-9.2E18 ... 9.2E18	64

Рис. 5.3. Форматы вещественных чисел

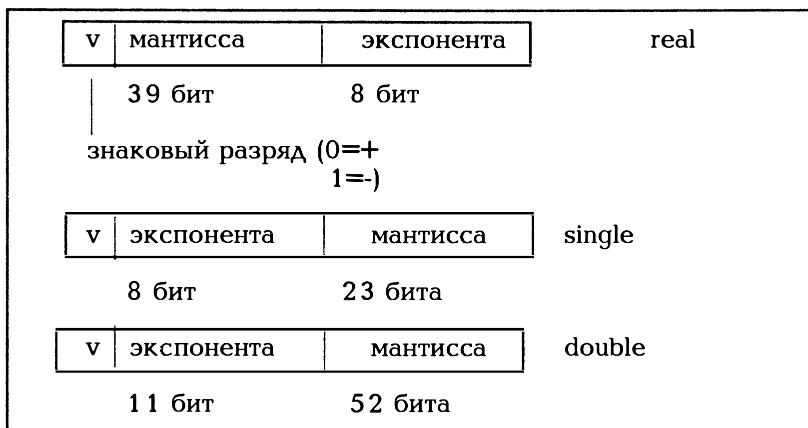


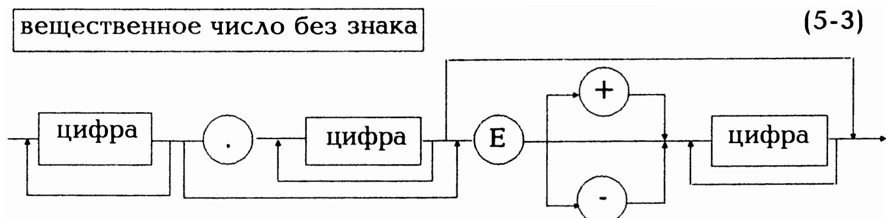
Рис.5.4. Внутреннее представление вещественных чисел

Как из мантиссы (m) и экспоненты (e) получается число z, рассмотрим лишь на одном случае real:

if $0 < e <= 255$ then $z := (-1)^e * 2^{(e-129)} * (1.m)$
if $e = 0$ then $z = 0$

В примере 15.5 показано, как программа может обрабатывать отдельные биты вещественного числа.

К сожалению, приходится реализовывать столь удивительное многообразие форматов дробных чисел. Все типы данных кроме real требуют использования сопроцессора 8087. Тип данных comp занимается совершенно особое место, поскольку значения таких величин могут быть лишь целочисленными. Он является в известной степени расширением типа longint до 19 разрядов. Постоянные такого типа могут быть записаны в следующей форме:



Итак, для вещественного числа без знака имеется три формы записи, например:

23.5678 0.5E-12 34E+8

Пример 5.1:

Для использования любого типа представления вещественного числа, кроме типа real, необходима директива

(*$N+*) (*см. приложение D)

```
program reals;
var x,y:double;
begin
  x := 2.345;
  y := 10*x;
  writeln(x:10:4, y:10:4);
end.
```

Для того, чтобы не было путаницы, форму внутреннего представления числа определяет не его значение, а его тип. Согласно описанию

var i :integer;x :real

выполняется присвоение

i := 12; (*записать 12 во внутреннем формате согласно схеме, преставленной на рис. 5.2*)

x := 12 (*записать 12 во внутреннем формате по схеме 5.4*)

Примечания относительно версии 5.0

Версия 5.0 различает оба состояния (*$N+*) (все 5 типов вещественных величин) и (*$N-*) (только тип real). Но сейчас имеется эмулятор, который формирует команды сопроцессора 8087 с по-

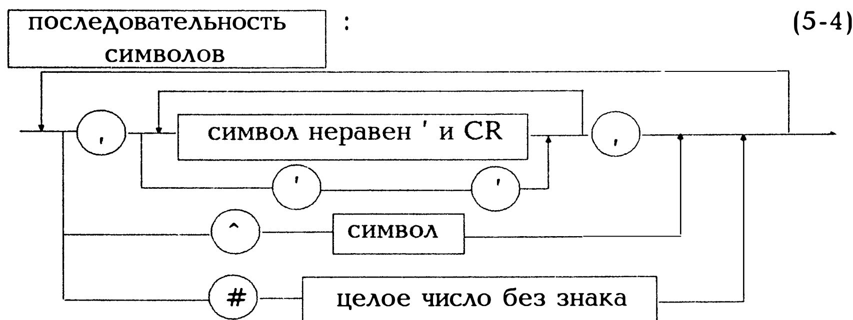
мощью лишь 8086. Соответственно, существует директива компилятора (*$E+*), подключающая эмулятор 8087, и директива (*$E-*), когда подключаются лишь небольшие стандартные подпрограммы для управления 8087. Чтобы не было путаницы, используют и директиву $E и директиву $N. Итак, пара (*$N+*)(*$E+*) обеспечивает использование всех 5 типов вещественных переменных, не требуя при этом наличия сопроцессора, а пара директив (*$N+*)(*$S-*) требует наличия сопроцессора 8087. Согласно (*$S-*) состояние $E значения не имеет, используется лишь тип real. Итак, директива $N устанавливает, какие типы данных имеются, а директива $E определяет, как реализуются операции.

5.3. Символы

Тип данных char обозначает множество символов кода ASCII (см. приложение С). Один символ требует для своего внутреннего представления 8 бит = 1 байту. Символы пронумерованы от 0 до 127 (для PC-DOS от 0 до 255). Константы такого типа обрамляются апострофами:

'a' 'B' '+'

Последовательность символов (строка) относится собственно к структурированным данным, а потому рассматривается лишь в главе 14.3. Здесь мы лишь приведем примеры констант такого типа:



Итак, строка является просто заключенной в апострофы последовательностью символов:

'Привет, друзья!' 'Это строка.'

Если в строке символов встречается апостроф ' , его следует "удвоить":

'Тогда это было ''.'

В приложении С приводится множество символов ASCII. Первые 32 символа являются управляющими. Для большинства из них

не существует соответствующей клавиши. Для того, чтобы включить в последовательность символов такие символы ASCII, введено понятие управляющего символа, для которого предусмотрено два способа записи:

предшествует некоторому целому числу, лежащему в диапазоне 0..255, которое задает номер управляющего символа ASCII,

^ предшествует управляющему символу.

Итак, имеем

#27 или #$1B или ^[для Escape ,
#7 " #$7 " ^G для звукового сигнала.

При следовании один за другим управляющие символы пишутся без разделителей:

#13#10 или ^M^K для последовательности "возврат каретки", "перевод строки".

Согласно приведенному выше описанию (5-4), смешанная последовательность символов может выглядеть следующим образом:

'Привет, пожалуйста, введите '^G^G^G', чтобы проснуться!'

Для типа данных char существуют следующие стандартные функции:

Вызов:	ord(x)	Значение функции: integer
Параметр:	x.char	
Действие:	Значением функции является номер ASCII-символа x	
Вызов:	chr(x)	Значение функции: char
Параметр:	x:integer	
Действие:	Значением функции является ASCII-символ с номером x	
Вызов:	upcase(x)	Значение функции: char
Параметр.	x:char	
Действие:	Значением функции является большая буква x, если она существует (в противном случае x остается неизменным)	

Рис.5.5. Стандартные функции для типа данных char

Пример 5.2:

ord('a') = 97, chr(63) = '?'
ord(chr(97)) = 97, chr(ord('a')) = 'a'

Приведенная ниже маленькая программа считывает номера и выдает соответствующие символы кода ASCII, являясь одновременно примером законченной программы.

```
program ASCII_Zeichen;
var i:integer;
begin
   writeln('введите номер символа: (конец i>127)');
   readln(i);
   while i < 128 do
      begin
         writeln('N',i:4,'   Символ:',chr(i):3);
         readln(i);
      end;
end.
```

5.4. Boolean

Тип данных boolean характеризуется двумя значениями вероятности true и false. При внутреннем их представлении они занимают 8 бит. В смысле перечисляемого типа (см. раздел 13.1) false<true.

В главах 12-18 будет описано, как можно описать собственный тип данных, не отказываясь от использования стандартных типов данных. Все перечисленные выше типы данных относятся к простейшим типам. Стандартные типы данных integer, byte, char и boolean (но не real) относятся к тому же к порядковым типам. Итак, если в дальнейшем говорится, что в определенном месте может встретиться порядковый тип, это означает любой тип данных за исключением real, причем real употребляется здесь как собирательное понятие для пяти изображенных на рис. 5.3 типов данных.

В заключение упомянем о функции sizeof, с помощью которой определяется число байтов, занимаемое переменной определенного типа (рис.5.6).

Вызов:	sizeof(x)	Значение функции: word
Параметр:	x является наименованием типа или именем переменной	
Действие:	Значением функции является число байтов, занимаемых в памяти x	

Рис. 5.6. Функция sizeof

Приведенная ниже небольшая программа может использоваться для того, чтобы выдать потребность в памяти для данных перечисленных выше стандартных типов.

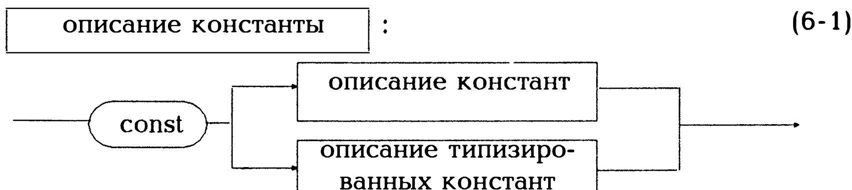
Пример 5.3:
Объем памяти, занимаемой данными стандартного типа:

```
program speicherplatz;

begin
 writeln('byte      :  ',sizeof(byte),' Byte');
 writeln('word      :  ',sizeof(word),' Byte');
 writeln('shortint :  ',sizeof(shortint),' Byte');
 writeln('integer  :  ',sizeof(integer),' Byte');
 writeln('longint  :  ',sizeof(longint),' Byte');
 writeln('single   :  ',sizeof(single),' Byte');
 writeln('real     :  ',sizeof(real),' Byte');
 writeln('double   :  ',sizeof(double),' Byte');
 writeln('extended :  ',sizeof(extended),' Byte');
 writeln('comp     :  ',sizeof(comp),' Byte');
 writeln('char     :  ',sizeof(char),' Byte');
 writeln('boolean  :  ',sizeof(boolean),' Byte');
end.
```

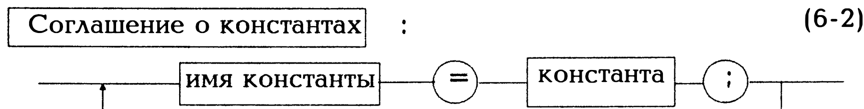
Функцию sizeof можно использовать и для того, чтобы определить потребность в памяти для структурированных типов данных, как это описано в главе 14.

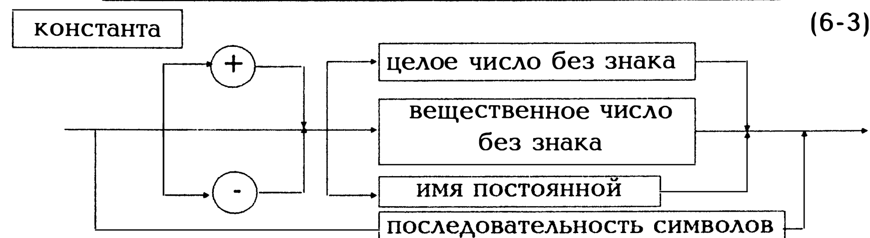
6. Соглашения о постоянных

Постоянные описываются согласно схеме (4-5) в той части программы, которая специфицирует данные. В Турбо Паскале различаются два вида констант:



Указанные выше возможности описания констант относятся к константам в строгом смысле этого слова:





Вот несколько примеров:

```
const  oben    = 324;
       unten   = -oben;
       parole  = 'абракадабра';
       escape  = #27;
```

Имена oben, unten, parole и escape здесь заменяют указанные выше значения. oben здесь не что иное как 324, и это значение не может в дальнейшем изменяться. Учтите, что согласно схеме (6-3) значение константы должно задаваться непосредственно. В крайнем случае в качестве значения можно использовать имя другой константы со знаком:

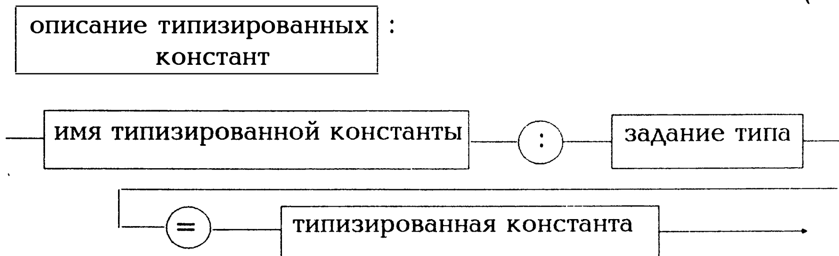
```
unten = -oben
```

Можно не вычислять значение, например:

```
const max = 500;
      min = 100;
      mitte = (max + min) div 2;
```

После mitte компилятор выдаст сообщение "constant expected". Это только задание компилятору перевести исходный текст в объектные коды, а не сразу же произвести вычисления. Но в отличие от стандартного Паскаля существуют и так называемые типизированные константы. В синтаксической диаграмме (6-1) это нижняя ветвь "описания типизированных констант". В описании типизированной константы присутствуют описание типа и одно из допустимых значений для этого типа типизированных констант.

(6-4)



Типизированные константы являются, собственно говоря, переменными, которым в той части программы, где описываются константы, присваивается некоторое начальное значение. Они не только могут описываться как переменные, их можно использовать в следующей программе в качестве переменных, т.е. им можно присвоить новое значение или использовать их в качестве параметра в процедурах или функциях.

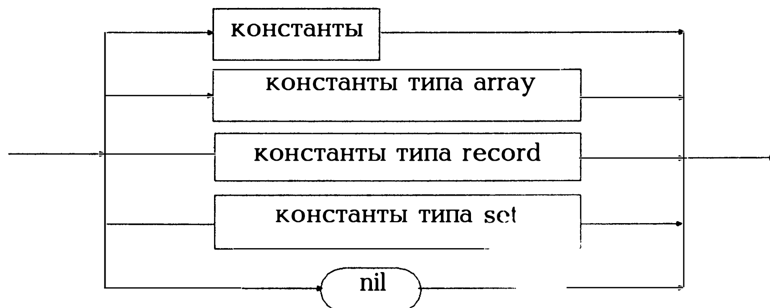
Вот примеры таких типизированных констант:

```
const zinsen : real = 7.5;
      anzahl : integer = 589;
      ein    : char = ^Q;
```

Итак, имена zinsen, anzahl и ein обозначают переменные заданного типа и начальные значения.

Вообще говоря, существуют следующие типизированные константы:

типизированные константы : (6-5)



Итак, для типизированных констант можно еще использовать структурированные типы данных: "массив", "запись", "множество", и тип данных "указатель". Они описаны в главах 14 (массив), 15 (запись), 16 (множество) и 18 (указатель).

Примечания относительно версии 5.0

В версии 5.0 компилятор может определять константы через выражения, такие как

```
const max = 500;
      min = 100;
      mitte = (max + min) div 2;
```

Выражения описываются в главе 9, а здесь мы лишь подчеркнем, что выражения, через которые определяются константы, не являются, вообще говоря, выражениями в смысле главы 9. Например, не все функции могут вызываться. Так описание

```
const max = 500;
      min = 100;
      mitte = sqrt(max * min);
      (* sqrt = корень квадратный)
```

приведет к выдаче сообщения об ошибке "Cannot evaluate this expression" ("Невозможно вычислить данное выражение") при компиляции, поскольку синтаксическая диаграмма (6-3) не допускает расширения за счет ветви с выражением. Нельзя ожидать от ком-

пилятора производительности, которую можно достичь лишь с помощью исполняемого программного модуля. Так, скажем, компилятору неизвестна функция sqrt (квадратный корень), поскольку она будет известна программе только после компоновки. В выражениях для определения констант можно использовать следующие функции:

abs	chr	hi	length	lo	odd	ord
pred	prt	sizeof	succ	swap	trunc	

Сразу же договоримся, что всюду, где в дальнейшем будут встречаться константы, и в случае версии 5.0 будут использоваться выражения с учетом указанных ограничений. Точнее говоря, это все функции модуля SYSTEM.TPU (так называемой библиотеки run time - библиотеки исполняющей системы), к которым нельзя обратиться с помощью uses.

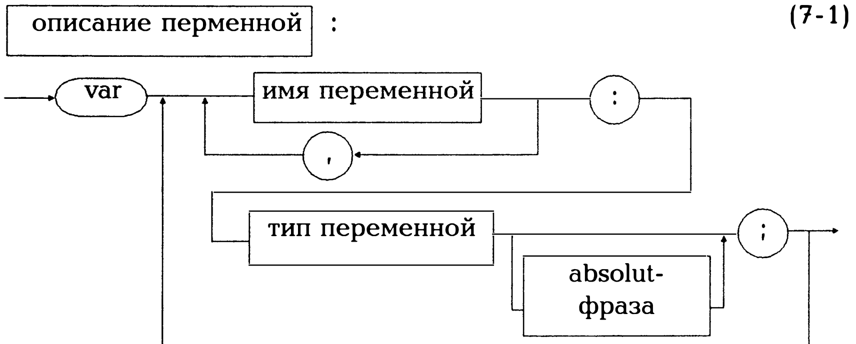
7. Соглашения о переменных

Понятие переменной является центральным понятием любого языка программирования. Для того, чтобы полностью описать переменную, необходимо указать четыре характеристики:

Имя	alpha
Тип переменной	real
Значение	12.34
Адрес	$1000:$2A34

Рис. 7.1. Составляющие описания переменной

Описание переменной имеет следующую форму:



Итак, в описании первые два параметра можно задать, например, таким образом:

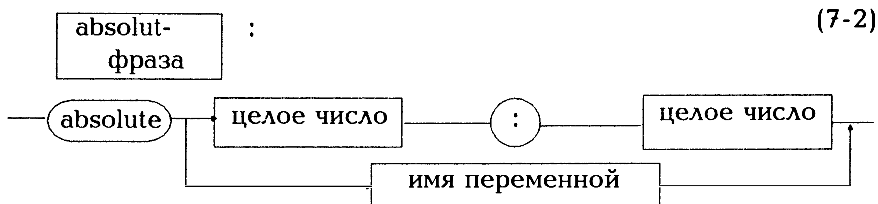
```
var maximum,minimum  :integer;
    alpha            :real;
    fertig           :boolean;
    buchstabe,zeichen:char;
```

Имя, называемое иначе идентификатором, служит для того, чтобы в программе можно было обратиться к этому объекту (то есть его идентифицировать).

Задание типа определяет область значений и вид внутреннего представления значения переменной. В приведенном выше примере значения maximum и minimum должны, как целые числа, занимать в памяти по 16 бит, переменная alpha - 32 бита в формате вещественных чисел (см. гл. 5). Здесь тип задается стандартным именем типа данных. Другие возможности задания типа переменной описываются в главах 12-18.

Значение переменной присваивается затем в программе путем считывания соответствующей константы (гл.8.), либо с помощью оператора присваивания (раздел 10.2). Следует отметить, что при запуске написанной на Паскале программы, в отличие от других языков программирования, переменные инициализируются, т.е. получают автоматически некоторое начальное значение. В Турбо Паскале этого можно достичь с помощью типизированных констант (гл.6). Следует непременно придерживаться такого принципа: применять переменную лишь тогда, когда ей присвоено некоторое определенное значение.

Адрес является адресом в памяти, начиная с которого значение переменной записывается в память. Он задается в типичной для MS-DOS форме "сегмент:смещение" в шестнадцатеричном представлении. Если объект имеет длину более 1 байта, это адрес первого байта. При компиляции имени (alpha) ставится в соответствие некоторый адрес ($1000:$2A34), так что имя становится, вообще говоря, ссылкой на собственное значение. С помощью absolut-фразы можно, как это показано на рис.7-1, отвести описанной переменной соответствующее место в памяти.



При этом адрес записывается в обычной для MS-DOS форме: адрес_сегмента:адрес_смещения (см. раздел 18.1), например:

```
var i:integer absolute $0020:$00A4
```

Естественно, перед absolute можно привести лишь одну переменную, поскольку можно указать лишь один адрес. Адреса долж-

ны лежать в диапазоне $0000...$FFFF (десятичные значения адреса 0...65535). Не имеет смысла задавать адреса. Встречающиеся в программе глобальные переменные расположены в сегменте данных. Согласно рис.18.3 функция dseg позволяет получить базовый адрес этого сегмента, который может использоваться в качестве первой константы после оператора absolute:

```
var a :array[1..8] of integer absolute dseg:$00FF;
```

Согласно (7-2) в качестве задания адреса можно воспользоваться именем некоторой другой переменной, через которую можно получить адрес:

```
var wort :string[40];
    laenge :byte absolute wort;
```

Переменные laenge и wort начинаются с одной и той же ячейки памяти, что имеет смысл, когда хотят получить, как это описано, скажем, в разделе 14.4, текущую длину строки wort в wort[0].

На рис.7-1 показано то же самое, но для простой переменной, имеющей одно значение. Начиная с раздела 14.1 будут использоваться так называемые структурированные переменные, состоящие из нескольких значений. В главе 18 мы увидим, что такие переменные полезны в тех случаях, когда переменная имеет тип "адрес некоторой другой переменной" и значение является лишь адресом.

Весьма важно различать представленные на рис.7-1 составные части переменной.

8. Элементарный ввод и вывод

Каждая программа взаимодействует со своим окружением через операторы чтения/записи (рис.8.1). Для каждого процесса чтения и записи следует указать, откуда и куда осуществляется чтение или запись и как вообще это должно выполняться. Из-за множества существующих в каждом языке программирования возможностей это довольно сложное дело.

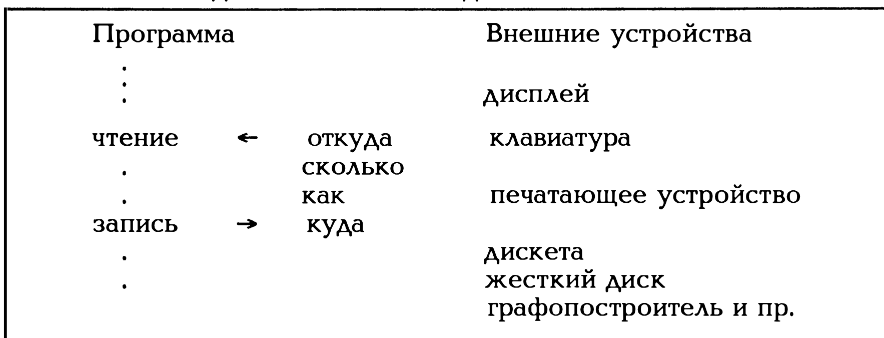


Рис. 8.1. Связь между программой и внешними устройствами

В Паскале связь программы с внешними устройствами осуществляется через имена файлов. Для нетерпеливых читателей сразу же поясним, что в наиболее простом случае с клавиатуры выполняется ввод данных, а на экран дисплея - вывод. Для ввода с клавиатуры существует имя файла input, а для вывода output. Согласно правилам Паскаля (которые необязательно соблюдаются в Турбо Паскале) в заголовке программы (4-2) распечатывается список имен файлов, с которыми взаимодействует программа. В простейшем случае это выглядит так:

```
program program_name(input,output);
```

Различное использование операторов ввода и вывода, в частности, при чтении с дискеты и при записи на дискету, а также при выводе на печатающее устройство, описывается в главе 17.

8.1. Ввод

Для ввода с клавиатуры (input) существует процедура read, представленная на рис.8.2.

Вызов: read(v1,v2,...,vn) процедура	
Параметры: var v1,v2,...,vn:integer или real или char	
	или string
Действие:	Если vi имеет тип - integer или real, считывается одно число соответствующего формата и значение его присваивается переменной vi.Знаки пробела или перевода строки перед числом игнорируются. - char,считывается один символ и присваивается переменной vi. - string, при длине n строковой переменной vi считывается максимум n символов.
Вызов:	readln(v1,v2,...,vn) процедура
Параметры:	как для read
Действие:	как для read с последующим переходом на начало новой строки.

Рис. 8.2. Процедура read

В отличие от других процедур read может иметь переменное число параметров, а readln может не иметь их вовсе. Так для

```
var i,j:integer;
    x,y:real;
```

можно записать

```
read(i,x)    или   read(i); read(x);
readln(i,x)  или   read(i); read(x); readln;
```

Тогда с клавиатуры следует ввести одно за другим значения i (например, 123) и x (например, 34.56). Причем числа 123 и 34.56 должны быть разделены по крайней мере одним пробелом. Конец ввода обозначается нажатием клавиши <RETURN>. Введенное число должно завершаться одним пробелом. Для наглядности покажем, как после числа ввести символы:

```
var a,b:char; i:integer;

read(i,a,b)     Ввод: 123 y
                i  <-  123 (завершить пробелом)
                a <-  пробел
                b <-   y
```

Если хотят подчеркнуть, что чтение осуществляется с помощью input, можно использовать input в качестве первого параметра оператора read, т.е. записать

```
read(input,i,a,b)
```

При вводе часто хотелось бы указывать, что ввод завершен (а не договариваться о добавлении некоторого специального символа в конце вводимой строки). Для этого имеется стандартная функция eof (= end of file = конец файла), представленная на рис. 8.3.

Вызов:	eof	Значение функции: boolean
Параметры:	нет (возможно input)	
Действие:	Значение функции всегда false, а при вводе 26 символов в коде ASCII (или при вводе ^Z = Ctrl-Z) true.	

Рис. 8.3. Функция eof

Вот типичная конструкция, используемая для считывания данных:

```
while not eof do begin read()  (*что-то считывается*)
                          (*что-то с этим делается*)
                 end;
```

Причем и здесь можно писать eof(input). Достаточно распространенный на практике случай демонстрирует пример 8.2. Для того, чтобы определить конец строки, используется функция eoln (= end of line).

Вызов:	eoln	Значение функции: boolean
Параметры:	нет (возможно input)	
Действие:	Значение функции true, если достигнут конец строки, и false в противном случае.	

Рис. 8.4. Функция eoln

Вот типичная конструкция, используемая для считывания данных:

```
while not eoln do begin read()  (*что-то считывается*)
                          (*что-то с этим делается*)
               end;        (*достигнут конец строки*)
```

Можно представить себе input как некий буфер, в который записываются введенные с клавиатуры строки. Тогда при вызове read данные берутся из этого буфера ввода. Чтобы определить, остались ли еще в input несчитанные символы, используется функция keypressed.

Вызов: keypressed	Значение функции: boolean	crt
Параметры: нет		
Действие: Значение функции true, если в буфере input не осталось несчитанных символов, и false в противном случае.		

Рис.8.5. Функция keypressed

Вот типичная для использования keypressed конструкция:

```
repeat
  (*что-либо делается*)
until keypressed;
```

Может оказаться желательным, чтобы введенные символы не отображались на экране дисплея. Тогда следует воспользоваться функцией readkey, которая считает ровно один символ из input. Введенный символ на экране не высвечивается.

Вызов: readkey	Значение функции: char	crt
Параметры: нет		
Действие: Считывается один символ из буфера input		

Рис. 8.6. Функция readkey

8.3. Вывод

Вывод на экран осуществляется с помощью операторов write или writeln.

Вызов:	write (p1,p2,...,pn) процедура
Параметры:	p1,p2,...,pn:integer или real или boolean или char или string
Действие:	Выдается на экран значение pi в стандартной форме
Вызов:	writeln(p1,p2,...,pn) процедура
Параметры:	как для write
Действие:	Как для write. В заключение выполняется переход на начало новой строки (writeln = write line)

Рис. 8.7. Процедуры write и writeln

При использовании приведенных выше форм записи процедур значения параметров write выводятся на экран в стандартном формате. Для Турбо Паскаля этот формат таков:

integer	столько разрядов, сколько требуется для записи числа
real	x.xxxxxxxxxxE+-xx (всего 18 разрядов) (x>=0) -x.xxxxxxxxxxE+-xx (всего 18 разрядов) (x<0)
boolean	false или true
char	или
string	символ или символы

Формат, отличный от стандартного, можно выбрать следующим образом:

pi:d	d - выражение типа integer, задающее ширину поля данных, в которое должно быть записано значение pi (i=1,...,n) с выравниванием по правому краю.
pi:d:s	pi (i=1,...,n) имеет тип real. d используется также, как это было только что описано выше. s - выражение типа integer, задающее число знаков после десятичной точки (но тогда без экспоненты!).

Если задаваемая ширина поля данных d будет выбрана слишком маленькой, d расширяется до нужного числа позиций.

Итак,для

```
var i,j:integer;
    x,y:real;
    a,b:char;
```

равносильно

```
write(i,x,a)    и   write(i); write(x); write(a);
writeln(i,x,a)  и   write(i); write(x); write(a); writeln;
```

В отличие от read, где параметрами должны быть имена переменных, параметрами write, являются выражения, а значит запись

```
write(2*i,3+5/(x+2),chr(123));
```

верна. Здесь вычисляется и выдается на экран значение выражения (см. главу 9).

Для начала рекомендуем читателю написать небольшую программу, в которой будут считываться и сразу же протоколироваться несколько значений различного типа. Следующая программа должна лишь подтолкнуть Вас к написанию собственных программ. Попробуйте модифицировать ее!

Пример 8.1:

Опробуйте следующую программу и измените ее так, чтобы понять, как действуют процедуры read и write.

```
program read_write_versuch;
var i,j: integer; x,y: real; a,b: char;
begin
  writeln('1 целое и 1 вещественное значение');
  read(i,x); writeln(i:4,x:10:4);
  writeln('1 целое и 1 вещественное значение');
  readln(i,x); writeln(i,x);
  writeln('1 символ и 1 целое число');
  readln(b,j);
  writeln(j:5,b:3);
  writeln('1 целое и 2 символа');
  readln(i,a,b);
  writeln(i:4, a:3, b:3);
end.
```

Следующий пример показывает, как можно считать и просуммировать некоторое неопределенное количество чисел:

Пример 8.2:

Вводится и суммируется произвольное количество чисел. Ввод завершается нажатием клавиш ^Z=Ctrl-Z, если переменная

checkeof из модуля crt установлена на true (стандартная установка false!).

```
program read_beliebig_viele_Zahlen;  {считать произвольное
                                               кол-во чисел}
uses crt;
var summe, zahl:integer;
begin
  checkeof :=true;
  summe :=0;
  writeln('Задайте несколько  целых чисел.  Конец ввода по
        ^Z = eof.');
  while not eof do
    (*Читать до тех пор,  пока  eof  не  примет  значения
     true.*)
    begin
      readln(zahl);
      (*Протокол считанных чисел*)
      writeln(zahl:20);
      summe := summe + zahl
   end;
  writeln('summe:', summe:6);
end.
```

Распечатка этой программы показывает, что ввод ^Z завершает цикл чтения данных.

Пример 8.2 имеет тот недостаток, что при неверном условии в readln(zahl) происходит прерывание, а значение полученной ранее суммы теряется. Это более чем неприятно. Хотелось бы также, чтобы ввод можно было проигнорировать и повторить заново. Для этого существует стандартная функция ioresult (рис.8.8) и директивы (*$I-*) и (*$I+*)(см. приложение D). По умолчанию устанавливается (*$I+*),т.е. установлена директива компилятору "I/O checking On" ("Проверка ввода/вывода вкл.").

Вызов:	ioresult	Значение функции: word
Параметры:	нет	
Действие:	Значением функции является состояние операции ввода/вывода.Если проверка ввода/вывода отключается, т.е. если имеем (*$I-*), следующие операции ввода/вывода игнорируются, если была вызвана ioresult. Если операция ввода/вывода прошла корректно, значение функции 0.	

Рис. 8.8. Функция ioresult

Если бы хотелось избежать прерывания при ошибке ввода, нужно действовать так, как показано в следующем примере.

Пример 8.3:

```
program ioresult_Demonstration;
uses crt;
var summe, zahl:integer;
begin
  checkeof :=true;
  summe :=0;
  writeln('Задайте несколько  целых чисел.  Конец ввода по
        ^Z = eof');
  while not eof do
     begin
        (*$I-*) readln(zahl); (*$I+*)
        if ioresult = 0 then summe := summe + zahl
                        else writeln('Нет числа');
        writeln(zahl:20);
    end;
  writeln('summe:', summe:6);
end.
```

При протоколировании с помощью writeln(zahl:20) увидим, что при ошибке ввода значение ioresult не равно нулю, а zahl принимает значение 0.

Следует еще пояснить, что собственно считывается при нажатии клавиши <RETURN>. Рассмотрим следующий пример.

Пример 8.4:

```
program read_return;
uses crt;
var c : char;
begin
   checkeof := true;
   writeln('Задайте  несколько  чисел  и нажмите несколько
           раз клавишу <RETURN>');
   writeln('Конец ввода нажатием клавиш ^Z = eof');
   while not eof do
      begin
        read(c);
        writeln(c:4, ord(c):4);
        (* Выдаются считанные числа и их номера в табли-
           це кодов ASCII*)
      end;
end.
```

Как оказалось, нажатие клавиши <RETURN> означает одновременный ввод ASCII-кодов символов возврата каретки (13) и перевода строки (10) (см. приложение С).

Пусть нам хотелось бы не просто выводить на экран числа и символы, но и как-то "оформить" такой вывод. Для этих целей служат представленные на рис.8.9 процедуры. Причем при описании в каждой первой строке стоит crt. Это означает, что с помощью некоторого условия uses должны быть сделана ссылка на модуль crt (см. также раздел 19.3).

Вызов:	clrscr	процедура	crt
Параметры:	нет		
Действие:	Гашение экрана (clear screen)		
Вызов:	gotoxy(i,k)	процедура	crt
Параметры:	i,k:integer;		
Действие:	Курсор перемещается в i-тую позицию k-той строки экрана. Левый верхний угол (1,1).		
Вызов:	lowvideo	процедура	crt
Параметры:	нет		
Действие:	Устанавливает для экрана атрибут "половинная яркость".		
Вызов:	normvideo	процедура	crt
Параметры:	нет		
Действие:	Устанавливает для экрана атрибут "обычная яркость".		
Вызов:	highvideo	процедура	crt
Параметры:	нет		
Действие:	Следующие символы высвечивать с большей яркостью.		
Вызов:	textbackground(farbe)	процедура	crt
Параметры:	farbe:byte;		
Действие:	Устанавливает цвет фона. Атрибутом farbe может быть одна из следующих констант: black (черный) = 0 blue (голубой) = 1 green (зеленый) = 2 cyan (васильковый) = 3 red (красный) = 4 magenta (фиолетовый) = 5 brown (коричневый) = 6 lightgray (светло-серый) = 7		
Вызов:	textcolor(farbe)	процедура	crt
Параметры:	farbe:byte;		
Действие:	Устанавливает текущий цвет. Атрибут farbe		

	может принимать одно из значений 0...7 из textbackground и кроме того darkGray (темно-серый) = 8 lightBlue (светло-голубой)= 9 lightGreen(салатовый) = 10 lightRed (алый) = 11 lightMagenta(сиреневый) = 13 yellow (желтый) = 14 white (белый) = 15
Вызов:	delline процедура crt
Параметры:	нет
Действие:	Стирает строку, в которой находится курсор и смещает все строки за ним на одну строку вверх.
Вызов:	insline процедура crt
Параметры:	нет
Действие:	Добавляет в позицию, на которой находится курсор, пустую строку и смещает все строки за ним на одну строку вниз.

Рис. 8.9. Несколько процедур для оформления выводимых данных

Относительно цветов, используемых в качестве атрибутов в процедурах textcolor и textbackground, следует отметить, что задавать их можно по выбору номером или наименованием цвета, то есть, строки textcolor(yellow) и textcolor(14) равнозначны. Как задать другие атрибуты, такие как мерцание, подчеркивание и пр., показано в главе 21.

Пример 8.5:
Пример цветового оформления текста.

```
program text_farben;
uses crt;
var i, k : integer;
begin
  for i := 0 to 7 do
  begin
     clrscr;
     textbackground(i);
     delay(2000);
     for k := 0 to 15 do
     begin
       textcolor(k);
```

```
      gotoxy(20,k+1);
      write('Привет, друг!');
      delay(800);
    end;
  end;
end.
```

Пример 8.6:

Демонстрация low, norm, highvideo.

```
program low_high_video;
uses crt;
begin
  clrscr;
  normvideo; write('Привет,');
  lowvideo; write('друг!');
  writeln;
  highvideo; write('Привет,');
  lowvideo; write('друг!');
  writeln;
  normvideo; write('Привет,');
  lowvideo; write('друг!');
  writeln;
  (*normvideo никакого действия не оказывает: обычный
   атрибут текста в позиции курсора при запуске*)
end.
```

Пример 8.7:

Далее приводится еще один пример, показывающий, как можно оформить выводимый текст при использовании процедуры write(i:d) с помощью d = ширине поля данных. Следующая программа записывает значение i в позицию i.

```
program nanu;
uses crt;
var i:word;
begin
  clrscr;
  for i :=1 to 10 writeln(i:i);
  (*В позиции i выводится значение i*)
end.
```

Вывод будет выглядеть таким образом:

```
-
 2
  3
   4
    5
     6
      7
       8
        9
         10
```

write или write(output,...) используют для вывода экран. Очень просто вывести данные и на печатающее устройство. Для него нужно определить логическое имя файла lst (=listing), соответствующее физическому устройству LPT1 (принтер) (см. раздел 19.3). Тогда достаточно лишь написать writeln(lst) или writeln(lst,...). В следующем примере персональная ЭВМ используется в качестве пишущей машинки. Копии введенных без ошибок с клавиатуры строк и символов выдаются на печатающее устройство.

Пример 8.8:

```
program schreibmaschine;  {пишущая машинка}
uses crt,printer;
var c:char;
begin
  checkeof :=true;
  while not eof do
  begin
     (*Считать и записать 1 строку.*)
     while not eoln do
       begin
           read(input,c); write(lst,c);
           (*Считать с клавиатуры,
             вывести на принтер.*)
       end;
     readln(input);
     (*Конец строки для input достигнут.*)
     writeln(lst);
     (*Перевод строки на принтере*)
  end;
end.
```

9. Операции и выражения

Выражения состоят из операндов, знаков операции и круглых скобок. Смысл выражения в том, чтобы пассивные составляющие (операнды), такие как константы, переменные и значения функций,

связать через активные составляющие (+,-,*,/ и пр.) и получить некоторое новое значение. Итак, выражение заменяет временно некоторое значение. Строго говоря, при использовании выражения

a op b

нужно было сказать, какой тип имеют операнды a и b операции op. Выражение не просто имеет некоторое значение, но и обладает совершенно определенным типом. Для большинства операндов нужно еще сказать, в какой последовательности должны использоваться операнды op1 и op2 в выражении

a op1 b op2 c

которое можно интерпретировать как

(a op1 b) op2 c или op1 (b op2 c) ?

В математике существует большое количество операций над вещественными и комплексными числами, множествами, векторами, матрицами, логическими выражениями и т.п. Поскольку для большинства знаков таких операций символов ASCII не существует, в языках программирования и, в частности, в Паскале, вынуждены один и тот же знак операции, скажем + или -, использовать довольно широко, употребляя его в разных значениях, или использовать английское написание операции, например, not, and, or. Результат выполнения операции a + b зависит от типов a и b!

Настоящая книга построена таким образом, чтобы описать в следующих главах, как можно образовать арифметические (операндами являются числа типа integer и real) или логические выражения (операнды имеют тип boolean). В других главах рассматриваются другие типы данных (строка, массив, множество, запись, указатель); там же указывается, какие операции и выражения существуют для них.

Для того, чтобы описать последовательность, в которой должны стоять операнды в выражениях и в которой должны расставляться скобки, целесообразно упорядочить операции по уровням. Правила приоритетности операций в выражениях представленны на рис. 9.1.

1. Операции уровня i выполняются до выполнения операции уровня j>i (чем ниже уровень, тем теснее связь)
2. Операции одного уровня выполняются поочередно слева направо.
3. Операции, заключенные в круглые скобки, выполняются раньше операций, записанных за скобками.

Рис. 9.1. Выполнение операций в выражениях

Эти правила действуют для всех типов выражений. Поскольку синтаксические диаграммы учитывают эти уровни приоритета и

распространяются на все операции, они довольно сложны. Синтаксическим диаграммам посвящено приложение А.

9.1. Арифметические операции и выражения

Арифметические выражения имеют тип real или integer, причем мы всегда под real будем понимать также single, double, extended и comp, а под integer byte, word, shortint и longint.

Особое положение с одноместными операциями + и -. Для +a знак "плюс" можно опустить, а при отрицательном значении a запись -a обязательна.

Уровень	Операция	Тип операнда	Тип результата	Значение
1	*	real	real	Умножение
	*	integer	integer	"
	*	integer, real	real	"
	/	integer	real	Деление
	/	real	real	"
	/	integer, real	real	"
	div	real	integer	Деление без остатка
	mod	integer	integer	Остаток от деления
2	+	integer	integer	Сложение
	+	real	real	"
	+	real,integer	real	"
	-	integer	integer	Вычитание
	-	real	real	"
	-	real,integer	real	"

Рис. 9.2. Арифметические операции

Следует отметить, что некоторые знаки операции могут употребляться и в другом значении, а именно:

+-* для операндов типа "множество" (см. гл. 16)

Элементарные операции + -*/ употребляются как и обычно в математике. Поскольку + - относятся к (низшему) уровню 2, а */ к уровню 1, действуют обычные правила скобок:

```
12*5 + 2   →   62
12*(5+2)   →   84
```

Следует особо отметить, что при делении / результат всегда имеет тип real:

10/3 → 3.333333...

Естественно, попытка деления на нуль приведет к ошибке.

Если результат деления целых чисел нужно получить в форме целой части частного и остатка, используют операции mod и div.

Как показывает пример 9.1, существует также некоторое значение i mod k при k<0, приводящее в стандартном Паскале к ошибке. В Турбо Паскале для отрицательного n

i mod n = i - (i div n) * n

Пример 9.1

Настоящий пример демонстрирует действие операций div и mod.

```
program mod_und_div;
var i,n : integer;
begin
   write('i = '); readln(i);
   write('n = '); readln(n);
   writeln(i:3,' div',n:3,' = ', i div n : 3);
   writeln(i:3,' mod',n:3,' = ', i mod n : 3);
end.
```

Несколько результатов:

```
 13 div  4 =  3          13 mod  4 =  1
-13 div  4 = -3         -13 mod  4 = -1
 13 div -4 = -3          13 mod -4 =  1
-13 div -4 =  3         -13 mod -4 = -1
```

Важный вопрос, относительно того, что будет, если в некотором выражении используются операнды различного типа, рассматривается в разделе 9.5.

Для ясности запишем еще несколько обычных математических выражений на Паскале.

$$\frac{\frac{x}{y}}{z}$$ → x/y/z (порядок следования слева направо!

$$\frac{x}{\frac{y}{z}}$$ → x(y/z)

$$\frac{x}{yz} \rightarrow \texttt{x/(y*z)}$$

$$\frac{\frac{a}{x}+\frac{2b}{5y}}{1+\frac{4z}{-2}} \rightarrow \texttt{(a/x +2*b/(5*y))/(1 +4*z/(-2))}$$

a(b+c) → a*(b+c) (* не забывайте *!)

Пробел до и после знака операции чисто декоративен: a + b обозначает то же самое, что и a+b. Если при расстановке скобок у Вас возникнут сомнения, вспомните простое правило: "Лишние скобки не помешают!".

9.2 Битовые операции

Предположим, нам необходимо обрабатывать отдельные биты некоторой константы. Для такой цели используют представленные на рис. 9.3 битовые операции.

Уровень	Операции	Тип операнда	Тип результата	Значение
0	not	integer	integer	Побитное отрицание
1	and shl shr	integer integer integer	integer integer integer	Побитное И Сдвиг влево Сдвиг вправо
2	or xor	integer integer	integer integer	Побитное ИЛИ Побитное исключающее ИЛИ

Рис. 9.3. Битовые операции

Операции сдвига shl и shr требуют некоторого пояснения:

i shl n Значение i сдвигается влево на n разрядов,
i shr n Значение i сдвигается вправо на n разрядов.

При этом вышедшие за пределы поля влево (для shl) или вправо (для shr) n разрядов теряются, а с другой стороны добавляются n нулей. При ассемблировании обычно различают арифметический и логический сдвиг. При арифметическом сдвиге знаковый разряд сохраняется без изменения, при логическом - нет. В примере 9.2

показано, как операции shl и shr осуществляют логический сдвиг. Отрицательное значение n не ведет к сообщению об ошибке. Но при сдвиге берется интерпретация без учета знака (word), так что всегда получается 0.

Пример 9.2:
демонстрирует действие операций сдвига.

```
program shift_operatoren_shl_und_shr;
var i,n : integer;
begin
   write('i ='); readln(i);
   (* Чтобы сгенерировать для i нужное множество битов,
      представим, что могут вводиться шестнадцатеричные
      числа*)
   write('n ='); readln(n);
   writeln(i:3,' shl',n:3,' = ', i shl n :3);
   writeln(i:3,' shr',n:3,' = ', i shr n :3);
end.
```

Здесь получится несколько результатов:

```
 12 shl 2 = 48          12 shr 2 = 3
-12 shl 2 = 48         -12 shr 2 = 16381
 128 shl 1 = 256        128 shr 1 = 64
-1 shl 1 = -2          -1 shr 1 = 32767
```

Одноместная операция not формирует побитовое отрицание значения типа integer. Двуместные операции and, or и xor побитно связывают два целых значения. В примере 9.3 наиболее целесообразно вводить значения в шестнадцатеричном виде ($FFFF вместо -1). Вывод результатов также должен осуществляться в шестнадцатеричном виде или побитно. Это делается в примерах 15.5 (шестнадцатеричное представление) и 9.4 (побитно).

Пример 9.3:
демонстрирует битовые операции not, and, or, xor.

```
program bit_operatoren_not_and_or_xor;
var i,n : integer;
begin
     (* Чтобы сгенерировать нужное множество битов, пред-
ставим,
      что могут вводиться шестнадцатеричные числа*)
   write('i ='); readln(i);
   write('n ='); readln(n);
```

```
    writeln('not',i:3,' =',not i :8);
    writeln(i:3,' and',n:3,' = ', i and n :3);
    writeln(i:3,' or',n:3,' = ', i or n :3);
    writeln(i:3,' xor',n:3,' = ', i xor n :3);
  end.
```

Несколько результатов:

```
-256 and 255 = 0     15 and 1 = 1     -1 and 128 = 128
-256 or  255 = -1    15 or  1 = 15    -1 or  128 = -1
-256 xor 255 = -1    15 xor 1 = 14    -1 xor 128 = -129
```

Чаще всего сталкиваются с необходимостью установить часть битов из некоторого множестваа в 0 или 1 или выделить часть битов.

С помощью операции or можно установить в 1 определенную часть битов:

```
x               0101010101010101
maske           0000111111110000=$0FF0
x or maske      0101111111110101
```

С помощью and и not можно установить часть битов в 0:

```
x               0101010101010101
maske           0000111111110000=$0FF0
x and not maske 0101000000000101
```

С помощью and можно выделить часть битов:

```
x               0101010101010101
maske           0000111111110000=$0FF0
x and maske     0000010101010000
```

С помощью xor можно выделить часть битов и образовать дополнение:

```
x               0101010101010101
maske           0000111111110000=$0FF0
x xor maske     0000101010100000
```

В приведенных выше случаях маска задается сразу в шестнадцатеричном виде (то есть, в приведенных примерах maske=$0FF0). Положим, мы хотели бы формировать в программе маску в зависимости от переменных. Если l номер левого бита маски, а r номер правого бита маски (биты нумеруются справа налево, начиная с нуля)

```
    |       |
0000111111110000

 l=11     r=4
```

то это достигается следующим образом:

```
l := 11;      (*или путем считывания*)
r := 4;       (*или путем считывания*)
maske := not(not 0 shl l-r+1) shl r;
```

С помощью not генерируется 16 единиц, с помощью shl l-r+1 выполняется сдвиг влево на ширину маски, так что справа добавляются l-r+1 нулей. Это множество битов инвертируется с помощью not, так чтобы маска оказалась выровненной по правому краю. В заключение маска с помощью shr r сдвигается на нужную позицию.

Пример 9.4:

В следующей программе с помощью операций and и shl побитно выдается внутреннее представление некоторого целого числа. Для понимания программы нужно лишь знать, что такое двоичное число и как записывается отрицательное число (1 для знака числа минус и двоичное дополнение).

```
program integer_bitweise_zeigen;
var i,counter:integer;
begin
  write('номер : ');
  readln(i);
  for counter:=15 downto 0 do
     if (i and (1 shl counter) <>0)
        then write('l')
        else write('0');
     (* Итак, в i одна 1 сдвигается слева направо
        и по одной связывается с соответствующим
        значением разряда i.*)
  writeln;
end.
```

Важный вопрос относительно того, что получится, если в одном выражении будут встречаться операнды разных типов, рассматривается в разделе 9.5.

9.3. Логические операции

Логические выражения имеют значение типа boolean, т.е. true или false. Переменные и функции типа boolean могут связываться через следующие операции:

Уровень	Операция	Тип операнда	Тип результата	Значение
0	not	boolean	boolean	Отрицание
1	and	boolean	boolean	Конъюнкция И
2	or xor	boolean boolean	boolean boolean	Дизъюнкция ИЛИ Исключающее ИЛИ
3	= <> < <= > >=	простой тип простой тип простой тип простой тип простой тип простой тип	boolean boolean boolean boolean boolean boolean	равно неравно меньше меньше или равно больше больше или равно

Рис. 9.4. Логические операции

Одноместная операция not возвращает значение истинности . Если переменная a:boolean имеет значение true, not a имеет значение false. Другие операции описываются таблицей истинности, представленной на рис. 9.5. a and b тогда и только тогда имеет значение true, когда истинны (true) оба операнда. a or b истинно (true), если a или b или они оба истинны. a xor b истинно, если истинно a или b, но не оба операнда (исключающее ИЛИ).

a	b	a and b	a or b	a xor b
false	false	false	false	false
false	true	false	true	true
true	false	false	true	true
true	true	true	true	false

Рис. 9.5. Таблица истинности логических операций and, or, xor

Согласно правилам приоритета, not выполняется перед выполнением and, a and перед or, т.е.

a and b or c	Вначале a and b, затем or c,
a and (b or c)	Вначале b or c, затем and a.

Очевидно, сравнение является выражением типа boolean: оно либо верно, либо нет. Согласно i : integer
i < 10 true или false

Согласно рис. 9.4 простые типы могут сравниваться между собой. К простым типам относятся стандартные типы данных integer, real, char и boolean, а также описанные в гл. 13 перечисляемый и ограниченный типы. Можно ли и если да, то как, применять операции сравнения для типов структурированных данных, будет говориться позже.

Поскольку сравнение имеет уровень 3, следует внимательно расставлять скобки. В приведенных ниже примерах скобки необходимы.

```
(i < 10) and (k<=0)
(i = 2)  or  (k >20)
(0 < x)  and (x <10)          соответственно 0<x < 10
('a' <= c)  and  (c<= 'z')     соответственно, c является
                               маленькой буквой
```

Логические выражения, используемые в операторах if и при организации циклов, играют в следующих главах большую роль. Там могут встретиться такие формулировки:

```
if (i<10) and (k<=0) then....
while (i = 2) or (k > 20) do ....
```

При оценке логических выражений следует различать так называемые короткие и длинные (или комплексные) оценки. Если выражение имеет форму a or b, то при оценке слева направо при a = true значение всего выражения true, независимо от того, каково значение b. Соответственно, для a and b при a = false значение выражения false, независимо от того, каково значение b. Под короткой оценкой следует понимать ситуацию, когда оценка прекращается, если установлено значение всего выражения. При комплексной оценке оцениваются все компоненты выражения, даже если значение выражение уже, собственно говоря, установлено. По директиве (*$B-*) строится короткая, а по директиве (*$B+*) комплексная оценка. По умолчанию выполняется (*$B-*) (см. приложение D). Итак, читатель уже может различать, что означает запись

```
(*$B-*) if (n <> 0) and (z/n < 20) then ....
```

а что запись

```
(*$B+*) if (n <> 0) and (z/n < 20) then ....
```

При (*$B-*) и n = 0 логическое выражение имеет значение false и z/n < 20 уже не учитывается. При (*$B+*) и n = 0 оценивается

также выполнение условия z/n < 20, чем провоцируется ошибка деления на нуль.

Пример 9.5:

Нужно расчитать координаты x,y точек на плоскости (x,y). При этом следует определить, сколько из расчитанных точек попадает в прямоугольник, ограниченный xmin,ymin и xmax,ymax.

```
program rechteck;      { программа прямоугольник}
uses crt;
var x,y,xmin,xmax,ymin,ymax:real;
    links,rechts,oben,unten:boolean;
    anzahl :integer;
begin
  checkeof :=true;
  xmin := -5;
  xmax := 20;
  ymin := 3.5;
  ymax := 12.8;
  anzahl := 0;
while not eof do
  begin
    write('x,y?');readln(x,y);
    if (x > xmin) and (x < xmax) and (y > ymin) and (y  <
ymax)
          then anzahl := anzahl +1;
  end;
  writeln('В прямоугольнике лежат',anzahl,'точек');
end.
```

Напомним, что в разделе 8.1 уже было рассмотрено несколько функций, имеющий значение типа boolean: eoln, eof и keypressed.

9.4. Стандартные математические функции

В выражениях может присутствовать вызов функций. Если f(x) некоторая функция, значение которой может иметь, например, тип integer, то

a + 2*f(23)

арифметическое выражение. Функция передает свое значение в выражение. В Паскале имеется ряд стандартных математических функций, представленных в приведенной ниже таблице. Здесь же указано, какой тип должен иметь аргумент функции и каков тип значения функции.

Вызов:	abs(x)	Значение функции: тип x
Параметры:	x:integer или real;	
Действие:	Абсолютное значение x, т.е. x	
Вызов:	exp (x)	Значение функции: real
Параметры:	x:real;	
Действие:	Значением функции является e в степени x,т.е. e^x	
Вызов:	cos(x)	Значение функции: real
Параметры:	x:real;	
Действие:	Значение функции - косинус x. x должен задаваться в радианах	
Вызов:	sin(x)	Значение функции: real
Параметры:	x:real;	
Действие:	Значение функции - синус x. x должен задаваться в радианах	
Вызов:	arctan(x)	Значение функции: real
Параметры:	x:real;	
Действие:	Значение функции - арктангенс x, т.е. главное его значение в области $-\pi/2...+\pi/2$	
Вызов:	ln(x)	Значение функции: real
Параметры:	x:real;	
Действие:	Значением функции является натуральный логарифм x. (x>0)	
Вызов:	sqr(x)	Значение функции: тип x
Параметры:	x:real;	
Действие:	Значением функции является квадрат x	
Вызов:	sqrt(x)	Значение функции: real
Параметры:	x:real;	
Действие:	Значением функции является квадратный корень из x (x=0)	
Вызов:	random(x)	Значение функции: real или word
Параметры:	x:word; (может и отсутствовать)	
Действие:	Если x отсутствует, значением функции является случайное число типа real из диапазона 0<=...<1. Если задается параметр x, имеющий тип word, значением функции будет случайное число типа word из диапазона 0<=....<x.	
Вызов:	Pi	Значение функции: real
Параметры:	нет	
Действие:	Значением функции является значение π. Точность зависит от используемого процессора	

Вызов:	odd(x)	Значение функции: boolean
Параметры:	x:longint;	
Действие:	Значение функции true, если x нечетен; и false в противном случае	
Вызов:	inc(x,n)	Значение функции: как x
Параметры:	x:перечисляемый тип; n:integer;	
Действие:	Значением функции является значение x, увеличенное на n. Если n отсутствует, x увеличивается на единицу (x := x+1, образование приращения)	
Вызов:	dec(x,n)	Значение функции: как x
Параметры:	x:перечисляемый тип; n:integer;	
Действие:	Значением функции является значение x, уменьшенное на n. Если n отсутствует, x уменьшается на 1 (x :=x-1, образование отрицательного приращения)	

Рис. 9.6. Математические функции

Ниже приводится еще несколько стандартных функций, которые могут использоваться прежде всего для преобразования типов данных.

Вызов:	int(x)	Значение функции: real
Параметры:	x:real;	
Действие:	Значением функции является целая часть x	
Вызов:	frac(x)	Значение функции: real
Параметры:	x:real;	
Действие:	Значением функции является дробная часть x, т.е. frac(x)=x - int(x).	
Вызов:	trunc(x)	Значение функции: longint
Параметры:	x:real;	
Действие:	Значением функции является целая часть x в формате longint. Значение x должно лежать в области значений longint	
Вызов:	round(x)	Значение функции: longint
Параметры:	x:real;	
Действие:	Значением функции является значение x, округленное до следующего целого числа, точнее round(x)=trunc(x+0.5) для x>0 round(x)=trunc(x-0.5) для x<0 Значение x должно лежать в области значений longint	

Вызов:	hi(x)	Значение функции: byte
Параметры:	x:integer или word;	
Действие:	Значением функции является старший байт x.	
Вызов:	lo(x)	Значение функции: byte
Параметры:	x:integer или word;	
Действие:	Значением функции является младший байт x.	
Вызов:	swap(x)	Значение функции: как параметр
Параметры:	x:integer или word;	
Действие:	Значением функции является x, оба байта которого поменены местами.	

Рис. 9.7. Функции преобразования типа

9.5. Приведение типов

Паскаль является довольно строгим языком в том, что касается совместимости типов данных. В частности, при присвоении значений (раздел 10.2) и в параметрах процедур и функций (гл. 11) требуется, чтобы значение выражения всегда имело некий совершенно определенный тип. Тип выражения автоматически получается из типов операндов, причем действует следующее правило:

Если в некотором выражении встречаются операнды различных типов, тип результата совпадает с самым сложным типом операндов.

Для двуместных операций

a op b

имеет место "выравнивание" типов, причем "простейший" тип преобразуется в тип более сложного из типов двух операндов. Итак, согласно

```
var a:shortint; b:integer; c:longint; x:real;
```

получим

a + b	Тип: integer
a + b + c	Тип: longint
a + b + c + x	Тип: real

Здесь следует быть осторожным. Ведь не просто все операнды приводятся к более сложному типу, а затем выполняется операция; приведение типа осуществляется лишь "по мере необходимости", т.е. при выполнении операции. Согласно рис.9.1 операции одного уровня выполняются слева направо. Это приводит к следующему результату:

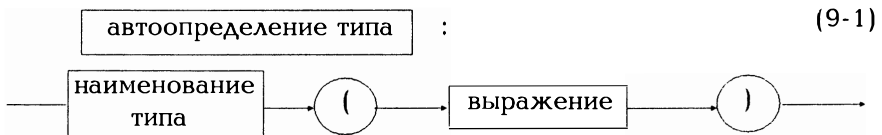
```
var b:integer;
b:=1000;
writeln(50*b:5);          Результат:-15536
```

Произведение 50*b выходит за пределы диапазона целых чисел integer (+32767). Вышедшие за пределы 16 битов разряды теряются! Если же мы захотим перейти в типу вещественных чисел real, добавив коэффициент 1.0, получим

```
writeln (50*b*1.0:8:2);   Результат: -15536.00
writeln (1.0*50*b:8:2);   Результат:  50000.00
```

В силу правила выполнения операций слева направо в первом случае вначале (для integer) неверно формируется произведение 50*b, а затем это неверное произведение приводится к вещественному типу real. Во втором случае образуется произведение 1.0*50, имеющее тип real, а затем оно умножается на преобразованное в вещественный тип значение b! Итак, очень важно знать, в какой последовательности обрабатывается выражение.

Тип выражения зависит от типов операндов. Но может оказаться, что нужно получить тип результата, независящий от типов операндов. В этом случае используется так называемое автоопределение типов. Эта операция типична для языка Си и не предусмотрена в стандартном Паскале.



Наименование типа результата и значение выражения должны оба иметь простой тип или тип указателя. Простой тип подробно описывается в главе 13, тип "указатель" - в главе 18. Здесь отметим лишь, что к простому типу относятся пять типов целых чисел, boolean и char, и не относится тип real. Итак, в приведенном выше примере можно было бы записать:

```
writeln (longint(50*b):5);   Результат: -15536.00
writeln (longint(50)*b:5);   Результат:  50000.00
```

В первом случае вновь будет получен неверный результат умножения 50*b, а затем будет использован тип longint. Во втором случае к типу longint правильно будет приведен лишь сомножитель 50, а затем будет получен верный результат 50000.

Автоопределение типов позволяет сделать так, чтобы значение занимало меньше места в памяти (например, в результате преобразования integer в shortint) или чтобы значение занимало больше ме-

ста (например, путем преобразования integer в longint). Знак числа при таком расширении, естественно, сохраняется:

```
writeln(word(2*b):5);          Результат: 2000
writeln(byte(2*b):5);          Результат:  208
writeln(shortint(-2*b):5);     Результат:   48
```

Приведение типа integer к типу real сложностей не вызывает, поскольку не влечет за собой потери информации. Вообще говоря, всюду, где используется тип real, можно использовать и значение integer. При преобразовании вещественного значения в целое следует указать, с какой точностью нужно получить результат. Здесь можно использовать перечисленные на рис. 9.7 функции.

В математике имеется много пар противоположных по своему действию операций, таких как *,/,+,- и т.п., для которых справедливы следующие равенства:

x/a*a = a
+b+a-a = b

В силу того, что в ЭВМ числа представлены в двоичном виде, совершенно не очевидно, что приведенные выше тривиальные равенства будут выполняться и внутри ПЭВМ. Приведем два небольших примера.

Пример 9.6:

Возьмем первый пример x/a*a=x и рассмотрим следующую программу:

```
program nanu;
var i,j,anzahl:integer;
    x:real;
begin
    anzahl :=0;
    for i :=2000 to 3000 do
        begin
          x :=i/1000;
          x :=x*1000;
          if i=x then begin anzahl :=anzahl+1; end;
        end;
    writeln('равно:',anzahl :10);
    writeln('неравно:',1001-anzahl :10);
end.
```

Здесь требуется дать несколько пояснений. Чисто математически следовало бы ожидать результата: "равно:1001, неравно:0". К сожалению, это не так. На персональном компьютере с процессором 8088 автор получил "равно:9, неравно:992"! Итак, почти всегда x/a*a <>x. Результат станет понятен, если рассмотреть двоичное представление чисел:

Возьмем для цикла значение i=2100. Тогда деление i/1000 дает вещественное число 2.1. Если перевести его в двоичную систему, получим:

2.1[десятичное]=10.001100110011...[двоичное]

2.1 бесконечная периодическая дробь
с повторяющейся группой 0011!!!!

В примере 15.5 наглядно представлены биты вещественного числа. Согласно рис. 5.4, мантисса 1000110011 0011... обрывается после 40 разрядов. Умножение на 1000, необходимое в силу представления 2**10, приводит к тому, что мантисса сдвигается влево на 10 разрядов, причем утраченные справа повторяющиеся группы 0011 не компенсируются, в результате чего приходим к неравенству:

2100/1000*100021000

Если может выдаваться меньшее значение i, для которого
i/1000*1000=i
увидим, что точный результат получим в том случае, двоичное представление которого имеет мантиссу с конечным числом разрядов, например, i=2125, 2250, 2375. Возьмем для наглядности первое значение

2.125[десятичное]=10.001[двоичное]
\- - - -
2. 0.125=1/8=0.001 Итак, мантисса =10001

Если используется сопроцессор 8087, можно добиться нужного результата с помощью директивы (*$N+*). Теперь арифметические действия над вещественными числами выполняются не программным путем, а реализуются на аппаратном уровне. Результат впечатляет: в приведенном выше примере получим: "равно:992", "неравно:9".

Пример 9.7:

Следующий пример показывает, что не всегда b+a-a должно равняться b:

```
program test;
uses crt;
var b :integer;
    a,x : real;
begin
    writeln ('b:');
    readln(b);
    writeln ('a');
    readln(a);


    x :=b + a -a;
    clrscr;
    writeln('x:',x);
    writeln('b:',b:10);
end.
```

Возьмем в качестве примера b=1 и a=1e+15. Перед выполнением операции сложения должны быть выравнены экспоненты обоих слагаемых:

0.000000000000001e+15+1e+15

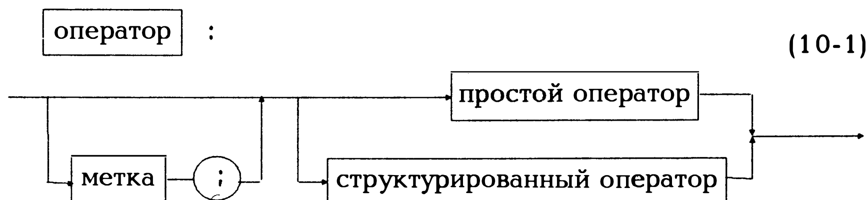
или

1e+0 + 1000000000000000e+0

В обоих случаях используется 15-разрядная мантисса. Но согласно рис. 5.3 вещественное число имеет мантиссу лишь из 11-12 десятичных разрядов. Сумма a+b (при b : integer, a : real) может быть образована, вообще говоря, лишь тогда, когда a и b различаются меньше, чем на 12 десятичных порядков.

10. Операторы

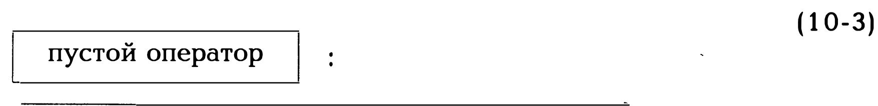
В исполняемой части программы помещаются операторы, которые должны быть выполнены с определенными в описательной части данными.



Обычно операторы выполняются в той последовательности, в которой они встречаются в тексте программы. Эту последовательность можно изменить, воспользовавшись оператором перехода, причем переход осуществляется к оператору, снабженному соответствующей меткой. Метка ставится перед оператором и отделяется от него двоеточием. Операторы перехода рассматриваются подробно в разделе 10.5.

Операторы подразделяются на простые и структурированные. Простой оператор не может составляться из других операторов.

Оператор присваивания описывается в разделе 10.2, оператор процедуры - в разделе 11.1, оператор перехода - в разделе 10.5. Пустым оператором является



то есть такой оператор не выполняет никаких операций и ничего не изменяет в данных и в программе. Он ставится по соображений синтаксиса там, где синтаксис требует наличия некоторого оператора, а никакой оператор там стоять не должен.

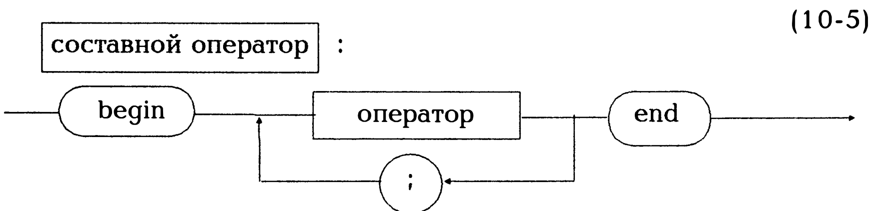
Встроенный оператор позволяет добавлять объектные коды непосредственно в написанный на Паскале текст. Как это делается в MS-DOS, описано в разделе 2.1.

Структурированные операторы составляются из других операторов. Различают следующие структурированные операторы:

Оператор with имеет значение лишь в сочетании с типом данных record и рассматривается в главе 11. Другие операторы описываются ниже.

10.1 Составной оператор

Составной оператор состоит из ряда операторов, выполняемых в той последовательности, в которой они встречаются в тексте программы.



Составной оператор обрамляется командами begin и end. Операторы разделяются точкой с запятой ; . Исполняемая часть программы (4.4) является составным оператором такого рода.

исполняемая часть : (10-6)

→ составной оператор →

Составной оператор служит в первую очередь для того, чтобы несколько операторов синтаксически объединить в один. Это часто требуется там, где нужно выполнить несколько операторов, когда допустим лишь один. Понятие составного оператора позволяет с помощью команд begin и end объединить несколько операторов и рассматривать их с точки зрения синтаксиса как один оператор.

При исполнении операторов точка с запятой служит разделителем для двух операторов. Собственно говоря, точкой с запятой перед заключительным end можно пренебречь. Но поскольку существует пустой оператор, обе записи

```
begin                      begin
  read(i);                   read(i);
  write(i)                   write(i);
end;                       end;
```

верны. При общепринятом стиле написания программ между write(i); и end ставится пустой оператор.

10.2. Оператор присваивания

Оператор присваивания является важнейшим оператором любого языка программирования. С его помощью мы можем присвоить переменной значение выражения. Выражение оценивается, т.е. определяется его значение, и это значение присваивается переменной. В результате прежнее значение переменной перезаписывается, а потому утрачивается.

Это определение будет расширено в разделе 11.2 при рассмотрении функций.

оператор присваивания : (10-7)

→ переменная → (:=) → выражение →

Сформулируем его более аккуратно. Каково различие между формулировками «одного и того же типа» и «одинакового типа», читатель сможет увидеть, обратившись к формулировке стандартного Паскаля «Переменная v и выражение e должны иметь идентичный тип». Как и всегда, будь это тот же самый, одинаковый или идентичный тип, следует сказать, что под этим понимается. О ти-

пах и наименованиях типов будет говориться лишь в главе 12. Там же будет пояснено, когда два типа являются идентичными.

При использовании оператора присваивания v := e следует непременно учитывать, что переменная v и выражение e должны иметь одинаковый тип. Имеется лишь одно исключение из этого правила: переменная может иметь тип real, а выражение значение типа integer.

Для простых типов (см. гл. 12) это утверждение несколько ограничено.

Рис. 10.1. Оператор присваивания

Пример 10.1:

Пусть

```
var i,j:integer; x,y:real; a,b:char; p,q:boolean;
```

Следующие операторы присваивания верны:

```
p := i < 5;
(*Выражение справа имеет тип boolean*)
a := '+';
x := i + j mod 7;
q := odd(j + i div 5);
j := round(x/2);
y := 275;
```

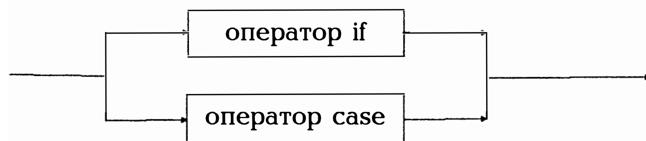
Следующие операторы неверны, поскольку не учитывают типа:

```
i := 3.678;
b := 'hans';
a := +;
```

10.3. Условные операторы

С помощью условных операторов можно выбрать один из нескольких возможных операторов.

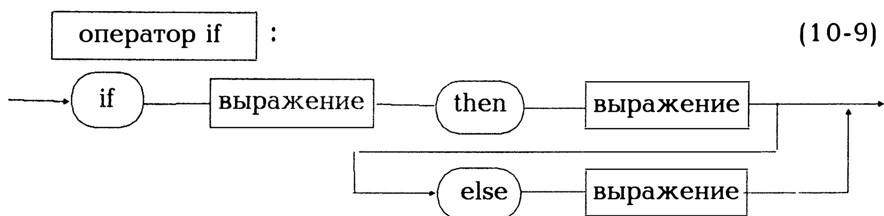
По практическим соображениях существует два типа условных операторов. Оператор if специально предусмотрен для тех случаев,



(10-8)

когда требуется выбрать одну из двух возможностей; оператор case позволяет выбирать из любого числа возможностей. Естественно, можно было бы обойтись и лишь одним из этих операторов.

Выражение после if должно быть логическим выражением (типа boolean). Если такое выражение имеет значение true, оператор,



следующий за then, выполняется; если же выражение имеет значение false, выполняется оператор, следующий за else. Если оператор else и оператор за ним опускаются, в случае false осуществляется переход к следующему оператору.

Если требуется выполнить после then или else несколько операторов, они обрамляются командами begin и end, образуя тем самым составной оператор. Учтите, что перед else никогда не ставится точка с запятой (поскольку это еще не конец оператора if).

Если else отсутствует,а после оператора then вновь стоит оператор if, то мы может столкнуться с многозначностью в выражении

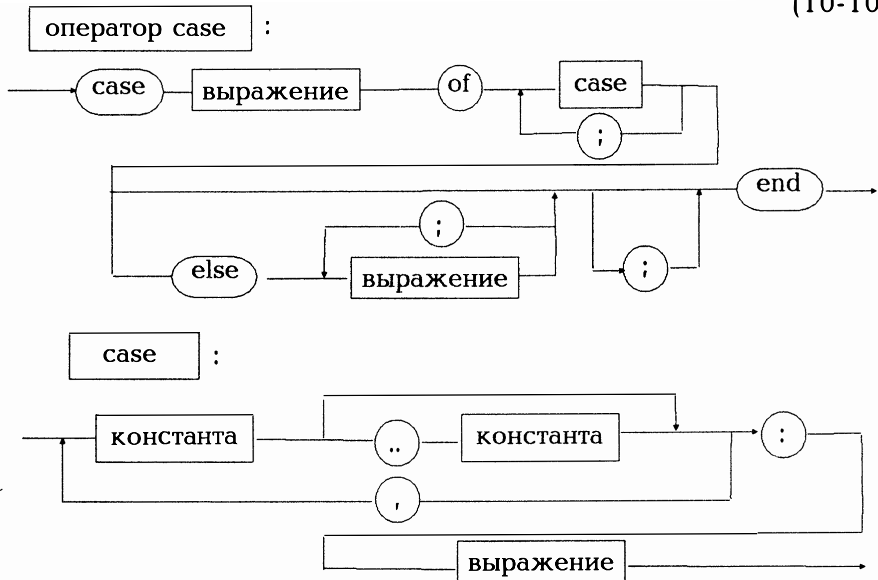
```
if выражение1 then if выражение2 then выражение1
                                    else выражение2
```

поскольку непонятно, относится else к первому или второму if. Существует правило, согласно которому else всегда относится к последнему if, не имеющему начинающегося с else продолжения, т.е. приведенное выше выражение понимается как

```
if выражение1 then begin if выражение2 then выражение1
                                    else выражение2 end
```

После служебного слова case стоит некоторое выражение, после of перечисление операторов, предоставленных для выбора. Эти операторы маркируются значениями, которые может принимать выражение. Значение выражения, называемое также селектором, позволяет выбрать то выражение, которое помечено данным значением. Одно выражение может иметь в качестве меток несколько констант или целый массив (см. раздел 12.2). Значение выражения должно иметь перечисляемый тип. Если записанное после case выражение принимает значение, не встречающееся среди констант, выполняется выражение после else. Значение селектора ограниче-

(10-10)



но диапазоном -32768...32767, а потому не может иметь тип word, longint или string.

Пример 10.2:

Возникшие у читателя вопросы поможет снять приведенный ниже кусок программы:

```
var c: char;
begin
   writeln('Еще раз? Ответьте J/N');
   read(c);
   case c of
     'j','J' :  {Оператор для повторения};
     'n','N' : {Оператор не для повторения};
   else      : {Запрашивается новый ввод}
   end;
end
(*На вопрос можно ответить с использованием
      больших и маленьких букв*)
```

Естественно, оператор if можно записать и в другой форме:

```
var i:integer;
begin
{После этого i принимает некоторое значение}
    if i < 25 then {этот true} else {каждый true}
```

что равносильно

```
case i < 25 of
true : {этот true};
false : {каждый true}   end;
end
```

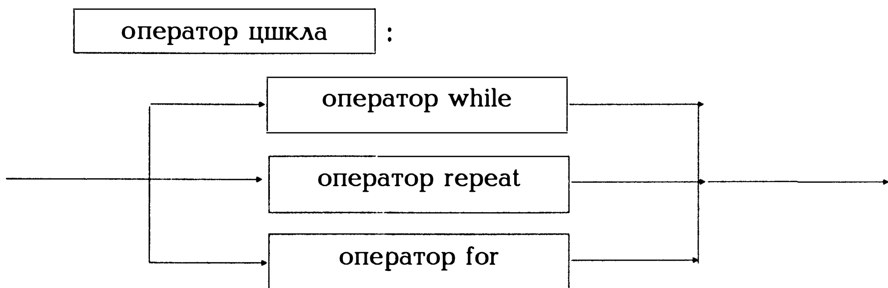
Выражению может соответствовать целое множество значений (см. раздел 12.2):

```
var c:char;
begin
  read(c);
  case c of
    '0'..'9': {c было цифрой};
    'a'..'z': {c было маленькой буквой};
    'A'..'Z': {c было большой буквой};
    else : {c некоторый другой символ} end
end
```

Учтите, что после двоеточия : может стоять лишь одно выражение. Если в этом случае хотят выполнить несколько операций, нужно записать их в виде составного оператора, воспользовавшись командами begin ... end.

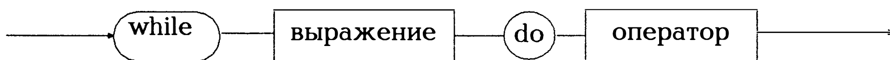
10.4. Операторы цикла

Операторы цикла служат для того, чтобы выполнять операторы повторно. При этом следует различать два вида циклов. Первый из них используют в том случае, когда знают заранее, сколько раз нужно выполнить цикл. В противном случае используют оператор for. Здесь число повторений цикла заранее неизвестно и может меняться по ходу выполнения программы в зависимости от некоторого условия. Потому различают три случая в соответствии с тем, стоит ли это условие в начале, середине или в конце цикла. В Паскале предусмотрен опрос по условию лишь в начале (оператор while) или в конце (оператор repeat) цикла. Если цикл прерывается в середине, то нужно воспользоваться оператором перехода (см. раздел 10.5) или построить цикл по-другому, что всегда возможно. Итак, в Паскале существуют три вида операторов цикла:



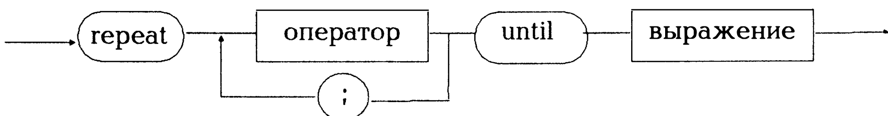
При использовании оператора while условие повторения цикла запрашивается в начале цикла:

(10-11)



Выражение после while должно быть логическим выражением. Оператор после do выполняется, если логическое выражение имеет значение true. Если после do, т.е. в цикле, должны быть выполнены несколько операторов, их следует записать в виде составного оператора, обрамляя командами begin и end. Если логическое выражение в начале имеет значение false, цикл вообще не выполняется.

(10-12)



Выражение после until должно быть логическим выражением. Выражения после repeat выполняются до тех пор, пока логическое выражение после until не примет значение true. Итак, построенный с помощью оператора repeat цикл выполняется по крайней мере один раз.

Обратите внимание на различие: построенный с помощью while цикл выполняется до тех пор, пока выражение имеет значение true (пока выражение не примет значение false), а цикл repeat выполняется тогда, когда выражение имеет значение false (или до тех пор, пока выражение false не примет значения true). Для прерывания цикла требуется, чтобы изменилось значение истиности выражения, стоящего после while и false. Сложности возникают в том случае, когда такое выражение составное. Так

```
repeat ... until (i < 10) or (x = 0);
while not ((i < 10) or (x = 0)) do ....
```
или
```
while (i>=10) and (x<>0) do ....
```

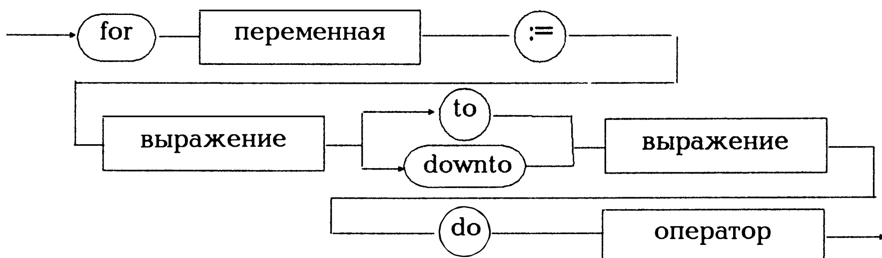
совершенно равнозначны (разве только в образованном с помощью while цикле не будет выполняться первый проход).

Пример 10.3:

Ввести все целые положительные числа в порядке их возрастания. Конец ввода по 0. Вывести все те числа, которых нет среди введенных.

```
program fehlende_zahlen_drucken;
      {распечатать отсутствующие числа}
var ein,aus: integer;
begin
   aus :=1;
   writeln('новое число(конец=0):');
   readln(ein); writeln('считано:',ein:3);
   while ein <> 0 do
     begin
        writeln('отсутствуют:');
        while aus < ein do
           begin writeln(aus);
                 aus :=aus+1;
           end;
        writeln('новое число(конец=0):');
        readln(ein); aus := aus+1;
        writeln('считано:'ein:3);
     end;
end.
```

оператор for : (10-13)



Переменная цикла и оба выражения должны иметь одинаковый тип, который должен быть порядковым (т.е. не быть типом real). Первое выражение является начальным значением, второе - конечным. При выполнении to начальное значение увеличивается каждый раз, пока не достигнет конечного значения (большего или равного начальному значению); при выполнении downto начальное значение каждый раз уменьшается, пока не достигнет конечного значения (меньшего или равного начальному значению). Переменная цикла может принимать значения ограниченного типа (см. гл. 13). Ширина шага в цикле for при прямом счете реализуется с помощью функции succ, при обратном счете - с помощью функции pred. Если переменная цикла имеет тип integer, это означает ширину шага +1 (для to) и -1 (для downto).

Пример 10.4:

```
var i:integer; summe,x:real;
begin
    summe :=0;
    for i := 1 to 10 do
      begin
         read(x);
         summe := summe+x; end;
(*Итак, ширина шага 1, succ(5) = 6 и т.д.*)
      end;
end.

var c:char;
begin
  for c := 'z' downto 'a' do
        writeln(c:2,'nr.', ord(c):4);
        { pred('z') = 'y' и т.д.
         Ширина шага от одного символа ASCII к другому
         в обратном направлении}
end.
```

Пример 10.5:

Следующая программа считывает некоторое число n (0<n<10) и заменяет звездочками * все те числа в диапазоне 0...99, сумма цифр которых равна n, в которых n цифр или которые делятся на n.

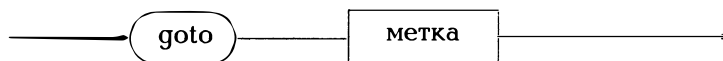
```
program zahlenspiel;
uses crt;
var i,n,zehner,einer,quersumme,teiler:integer;
```

```
        c:char;
    begin
        clrscr;
        writeln('Введите число n из интервала 0<n<10 ');
        read(n);
        clrscr;
        for i := 0 to 99 do
           begin
              zehner := i div 10;
              einer := i mod 10;
              quersumme := zehner + einer;
              teiler := i mod n;
              if i mod 10 = 0 then writeln;
              if (zehner=n) or (einer=n) or
                 (quersumme=n) or (teiler=0)
                 then write('*':4)
                 else write(i:4);
        end;
        readln;
        (* В результате при работе с версией 5.0 после завер-
         шения программы на экране останется окно вывода *)
    end.
```

10.5. Оператор перехода

Исполняемая часть программы является некоторым составным оператором, в котором составляющие его операторы выполняются в той последовательности, в которой они записаны. Если нужно из-

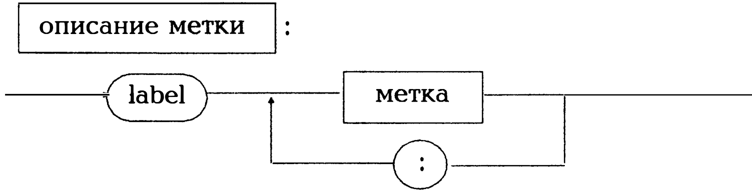
оператор перехода : (10-14)

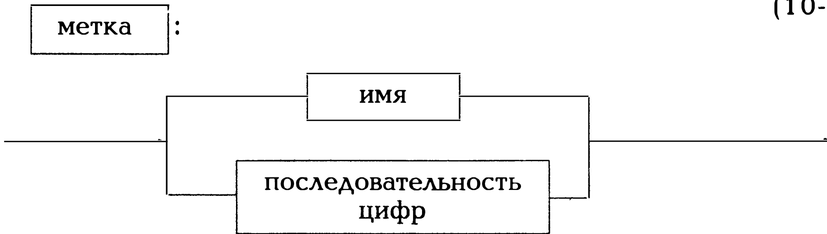


менить такую последовательность, пользуются оператором перехода.

Указанная в операторе перехода метка стоит перед оператором, который нужно выполнить следующим, и отделяется от него двоеточием (см. 10-1).

Метки должны быть описаны. Для этого согласно диаграмме (4-5) в описательной части программы должны присутствовать описания меток:





Здесь «последовательность цифр» означает некоторое число в интервале 0-9999. В отличие от стандартного Паскаля, в Турбо Паскале меткой может быть и имя.

Пример 10.6:

Следующая программа считывает числа до тех пор, пока их сумма не превысит некоторое определенное значение или пока числа не закончатся.

```
program sprung_marken;
    {метки перехода}
uses crt;
label fertig,next;
const maxsumme = 500;
var summe,anz,z : integer;
    c : char;
begin  (******* операторы ********)
        checkeof := true;
        anz := 0;
        summe := 0;
        clrscr;
        writeln('Вводите числа:');
        writeln('Конец по ^Z или при summe > maxsumme');
next:    if eof then goto fertig;
        readln(z);
        summe := summe + z;
        anz := anz + 1;
        if summe > maxsumme then goto fertig;
        goto next;
```

```
fertig: writeln('Сумма:',summe:5);
        writeln('Кол-во чисел: ',anz:3);
        readln;
end.
```

Согласитесь, что все это не так уж сложно. Реализуйте тот же алгоритм, не пользуясь операторами перехода!

В стандартном Паскале в качестве меток могут использоваться только целые числа! Благодаря блочной структуре (раздел 11.1) устанавливается область действия имен. В отличие от стандартного Паскаля относительно области действия меток существует следующее правило:

Метки имеют силу только в том блоке, в котором они описаны, и не действуют в предыдущем и последующем блоках. Итак, в процедуру или функцию нельзя попасть извне, как нельзя выйти за пределы процедуры или функции.

Операторы перехода используются чаще всего тогда, когда процедура или функция должны быть прерваны раньше времени. В Паскале программа или функция завершается лишь тогда, когда выполнены все операторы. Итак, если нужно из определенного места в программе перейти в ее конец или в конец блока:

```
label stop;
var error:boolean;
begin
   ....
   if error then goto stop;
   ....
stop:end.
```

В силу сделанных ранее замечаний относительно области действия меток, таким образом нельзя выйти из процедуры или функции. В этом случае проще воспользоваться стандартными процедурами halt и exit.

Вызов:	halt	Процедура
Параметры:	нет	
Действие:	Программа завершается	
Вызов:	exit	Процедура
Параметры:	нет	
Действие:	Переход в конец того блока, из которого сделан вызов	

При вызове halt, в каком бы месте программы он не стоял, выполнение программы прерывается, вызов exit завершает обработку

текущего блока. Итак, приведенные ниже формулировки равнозначны:

```
label 100;
var error:boolean;              var error:boolean;
begin                           begin
  ....                          ....
  if error then goto 100;         if error then exit;
  ....                          ....
100: end.                         end.
```

Операторы перехода приобрели дурную славу. Надежность программы повышается, если статическая запись текста программы совпадает с динамическим ходом ее обработки. При применении оператора перехода такая взаимосвязь нарушается. Потому относитесь к употреблению операторов перехода столь же бережно, как к ношению белой одежды, а лучше всего избегайте их применения.

11. Процедуры и функции

Процедуры и функции дают возможность снабдить последовательность операторов именем и обращаться затем к ней с помощью этого имени. При вызове можно также передавать различные параметры, меняя их от случая к случаю. Это означает, что программу можно разделить на части, снабдив их именами, что сделает программу более обозримой и надежной.

Стандартные функции и процедуры Паскаля предоставляют пользователю широкие возможности. Функции и процедуры вызываются с помощью их имени и хранятся в библиотеке Паскаля TURBO.TPL в виде отдельных модулей. Как пользоваться модулями библиотеки, описано в главе 19.

Здесь же поясним лишь, как пользователь может описать собственные процедуры и функции. Пользователь может компилировать эти написанные им процедуры и функции в отдельные модули, которые может затем вызывать из других программ. Как это сделать, описано в разделе 19.2.

11.1. Процедуры

Под процедурой понимают некоторое число операторов, снабженных именем и вызываемых с помощью этого имени. Прежде чем говорить о процедурах подробнее, приведем для наглядности один небольшой пример (читателю советуем еще раз обратиться к примерам 2.1 и 4.1).

Пример 11.1:

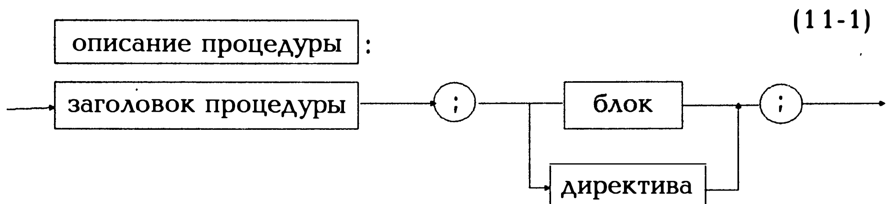
Следующая программа содержит процедуру quersumme, с помощью которой можно определить сумму цифр целого числа.

```
program quersummen_bestimmung;
   {программа определения суммы цифр}
uses crt;
var n : longint;
    quer : word;
(* описание процедуры*)
(*заголовок процедуры*)
procedure quersumme(x:longint;var q:word);
(*описательная часть*)
var h:integer;
begin   (***** Исполняемая часть *)
        (* Тело процедуры quersumme*)
    h := 0;
    while x <> 0 do
       begin h := h + x mod 10; x := x div 10 end;
    if h < 0 then q :=-h else q :=h;
end;  (*сумма цифр*)

begin  (*********Операторы*********)
   checkeof :=true;
   writeln('Введите целое число. Конец eof');
   while not eof do
      begin
         readln(n);
         quersumme(n,quer);
         (*Вызов процедуры*)
         writeln('Сумма цифр',n:5,'равна');
         writeln(quer:3);
      end;
end.
```

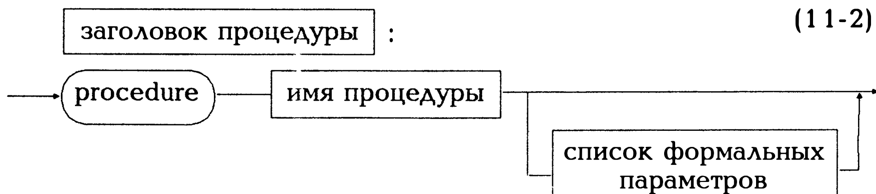
Процедура должна быть описана. Тогда вызов процедуры осуществляется из исполняемой части программы.

Описание процедуры имеет следующий вид:

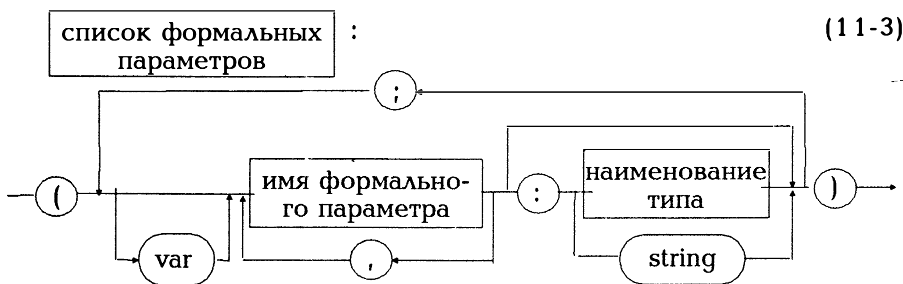
(11-1)



Чаще всего встречается случай, когда тело процедуры оформляется в виде блока. Но можно в этом месте процедуру не записывать. Директива в том месте, где должна стоять процедура, позволит вызвать нужный текст оттуда, где компилятор отыщет запись собственно процедуры. Мы вернемся к этому вопросу в конце данной главы.



За служебным словом procedure следует имя процедуры (в нашем примере это quersumme), а за ним может следовать список формальных параметров (в нашем примере x : longint; var q : word).



Список формальных параметров состоит из наименования параметра и задания соответствующего типа. Параметры одного типа могут перечисляться через запятую, например:

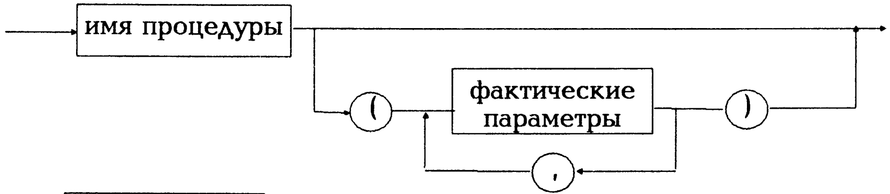
```
procedure nasowas(i,j : integer; var x : real);
```

Формальные параметры заменяются при вызове фактическими параметрами (в нашем примере, n, quer). Формальные параметры резервируют место для фактических параметров. Процедуры вызываются с помощью оператора процедуры.

При вызове фактические параметры должны совпадать с данными в описании формальными параметрами по их числу, типу и последовательности вызова.

Имена формальных параметров выбираются произвольно. Это не должно сбивать читателя с толку. Если в примере 11.1 формальные параметры имеют имена x и q, фактические параметры n и quer, это не означает, что имена должны быть непременно различными. Если в последующих примерах формальные и фактиче-

оператор процедуры : (11-4)



фактические параметры : (11-5)

ские параметры будут иметь одинаковые имена, это также не означает, что они должны быть одинаковыми всегда. Фактический параметр при вызове процедуры должен стоять на том же месте, что и соответствующий формальный параметр (в нашем примере n и quer описываются как переменные).

Важно различать передачу параметров по значению и по ссылке.

Передача параметров по значению (call by value):

Формальные параметры, вызов которых должен обеспечить передачу значения, присутствуют в списке параметров только в виде имен. В качестве фактических параметров они должны заменяться некоторым выражением того же типа, значение которого передается в начале процедуры формальному параметру. В примере 11.1 передача параметра по значению применяется для x.

Передача параметров по ссылке (call by reference):

Формальным параметрам, вызов которых должен обеспечить передачу параметров по ссылке, в списке параметров предшествует var. В качестве фактических параметров они должны заменяться именем переменной того же типа. В примере 11.1 передача параметров по ссылке используется для параметра q.

Пример 11.2:
Следующий пример демонстрирует принципиальное различие между передачей параметров по значению и передачей параметров по ссылке.

```
program value_reference_parameter;
      {передача параметров по значению и по ссылке}
var i,k:integer;
procedure count(x:integer; var y:integer);
begin
  writeln('В процедуре:');
  x:=x+1; y:=y+1;
  writeln('Передача параметра по значению',x:4);
  writeln('Передача параметров по ссылке',y:4);
end; (*счет*)
begin    (****** Исполняемая часть программы ******)
  i:=5; k:=5;
  writeln('Значение перед выполнением процедуры');
  writeln('Передача параметров по значению:',i:4);
  writeln('Передача параметров по ссылке:',k:4);
  count(i,k);
  writeln('Значение после выполнения процедуры');
  writeln('Передача параметров по значению:',i:4);
  writeln('Передача параметров по ссылке:',k:4);
write('<RET>':40); read (c);
end.
```

Более точно представить результат вызова процедуры можно таким образом:

вызов count(i,k); соответствует

```
begin
  x:=i;
  begin
    x:=x + 1; k:=k + 1;
  end;
end;
```

Фактический параметр i передает свое значение формальному параметру x, а фактический параметр k ставится на место формального параметра y. Процедура увеличивает значение копии i и значение оригинала k. После вызова процедуры i имеет свое прежнее значение 5, k - новое значение 6.

> Внимание!
>
> При передаче по ссылке процедура передает адрес фактического параметра. Изменения соответствующего формального параметра внутри процедуры изменяют и фактический параметр. При передаче по значению процедура передает копию значения параметра. Изменения соответствующего формального параметра изменяют лишь эту копию. Итак, выходные параметры процедуры всегда должны задаваться в виде передаваемых по ссылке параметров.

Локальные и глобальные величины

Телом процедуры является блок, в котором могут описываться имена. Следовательно, возможно вложение блоков, а значит, сразу же возникает вопрос об области действия (или области видимости - scope) имен.

1. Все описанные в некотором блоке имена имеют силу только для этого блока и вложенных в него блоков. Для такого блока имена являются локальными. Локальных имен вне блока не существует (в примере 11.1 используется локальная переменная h).

2. Все используемые в некотором блоке имена, описанные в блоке более высокого уровня иерархии, являются для блока более низкого уровня глобальными.

3. Если в некотором блоке локальная и глобальная величины имеют одно и то же имя, внутри блока с этим именем всегда связывается локальная величина. Значение имеющей то же имя глобальной величины на время обработки блока «замораживается».

Пример 11.3:

Локальные и глобальные переменные с одинаковыми именами (случай 3):

```
program vertauschen;
uses crt;
var hilf, i, k : word;

procedure tausch(var x,y:word);
var hilf : word; (*локальная переменная hilf*)
begin hilf :=x;
      x := y;
      y := hilf;
end; (*замена*)

begin (******* Исполняемая часть программы *******)
  hilf :=56;
  (* Это глобальная переменная hilf*)
  write('Введите два числа:');
  readln(i,k);
  writeln(i:6,k:6);
  tausch(i,k);
  writeln(i:6,k:6);
  writeln('Глобальная переменная hilf',hilf:4);
  (* Вывод 56
    Глобальная переменная hilf после вызова процедуры
    также имеет свое прежнее значение*)
end.
```

В Турбо Паскале согласно диаграмме (11-3) существуют также формальные параметры без идентификатора типа. Назовем такие параметры нетипизированными. Если нетипизированным параметром является параметр,передаваемый по ссылке, то соответствующим фактическим параметром может быть адрес объекта любого типа. Внутри процедуры нетипизированный параметр несовместим с переменными всех других типов без явного приведения типа к типу соответствующей переменной. Это демонстрируется в следующем небольшом примере. Пример показывает также, что благодаря оператору приведения типа отсутствие типа упраздняется : Посредством real(p1) фактический параметр приобретает смысл как адрес вещественной переменной.

Пример 11.4:

Пример неимеющего типа параметра.

```
program typlose_parameter(input,output);
uses crt;
var x : real; i : integer;
procedure test(var p1, p2);
var x : real;
begin
   x := p1; (* Несоответствие типов*)
   writeln('параметр p1:');
   writeln(real(p1):8:3);
   writeln('параметр p2:');
   writeln(integer (p2):9);
end;

begin
   i := 123;
   x := 1.3;
   test(x,i);
end.
```

В библиотеках TURBO.TPL и GRAPH.TPU имеется большое число стандартных процедур (см. раздел 19.3 и главу 22). Важнейшие из этих процедур поясняются в соответствующих разделах. Приведем здесь лишь несколько из них.

Путем чередования вызовов процедур sound/delay()/nosound можно генерировать звуковые сигналы различного тона, получая интересный эффект. Приведенная ниже программа позволяет считывать с клавиатуры символы и выводить через динамик их морзянку.

Вызов:	delay(t)	процедура	crt
Параметры:	t : integer;		
Действие:	Выполнение программы приостанавливается на t мс		
Вызов:	sound(hz)	процедура	crt
Параметры:	hz : word;		
Действие:	Запускается звукогенератор с частотой тона hz Гц		
Вызов:	nosound	процедура	crt
Параметры:	нет		
Действие:	Отключается сгенерированный с помощью sound звуковой сигнал		

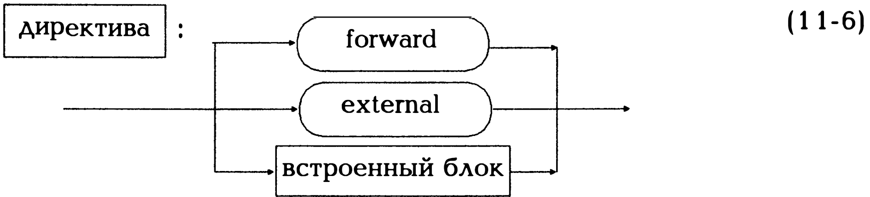
Рис. 11.1. Некоторые общеупотребительные стандартные процедуры

Пример 11.5:

```
program morsen;
{программа "морзянка }
uses crt;
var c:char;

procedure punkt;
{процедура "точка }
begin
  sound(2000);
  delay(50);
  nosound;
  delay(100);
end;

procedure strich;
{процедура "тире }
begin
  sound(2000);
  delay(200);
  nosound;
  delay(100);
end;

procedure pause;
{процедура "пауза }
begin delay(500); end;
```

```
procedure morse(c:char);
begin
    case c of
    'a','A': begin punkt; strich; end;
    'b','B': begin strich; punkt; punkt; punkt; end;
    'c','C': begin strich; punkt; strich; punkt; end;
    'd','D': begin strich; punkt; punkt; end;
    'e','E': begin punkt; end;
    'f','F': begin punkt; punkt; strich; punkt; end;
    'g','G': begin strich; strich; punkt; end;
    'h','H': begin punkt; punkt; punkt; punkt; end;
    'i','I': begin punkt; punkt; end;
    'j','J': begin punkt; strich; strich; strich; end;
    'k','K': begin strich; punkt; strich; end;
    'l','L': begin punkt; strich; punkt; punkt; end;
    'm','M': begin strich; strich; end;
    'n','N': begin strich; punkt; end;
    'o','O': begin strich; strich; strich; end;
    'p','P': begin punkt; strich; strich; punkt; end;
    'q','Q': begin strich; strich; punkt; strich; end;
    'r','R': begin punkt; strich; punkt; end;
    's','S': begin punkt; punkt; punkt; end;
    't','T': begin strich; end;
    'u','U': begin punkt; punkt; punkt; end;
    'v','V': begin punkt; punkt; punkt; strich; end;
    'w','W': begin punkt; strich; strich; end;
    'x','X': begin strich; punkt; punkt; strich; end;
    'y','Y': begin strich; punkt; strich; strich; end;
    'z','Z': begin strich; strich; punkt; punkt; end;
    else pause end;
end;

begin    (******* Операторы *******)
    checkeof :=true;
    writeln('Введите некоторый текст. Конец по ^Z)' );
    while not eof do
      begin
        read(c); morse(c); pause;
      end;
end.
```

За заголовком процедуры может следовать одна из директив:



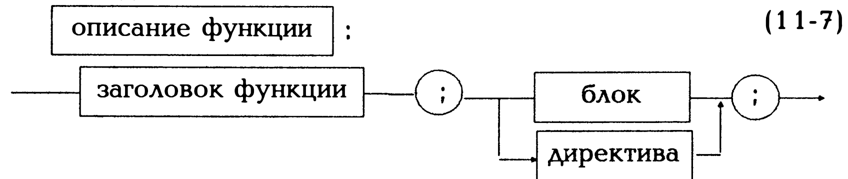
Если в Паскале за заголовком процедуры следует тело процедуры, оно должно оформляться в виде блока. Если тело процедуры записывается где-то в другом месте той же программы, нужно использовать директиву forward (см. пример 11.13). Если процедура существует в виде файла .OBJ, полученного, например, при ассемблировании, обращаться к процедуре следует через директиву external. Соответствующие объектные коды связываются с помощью директивы (*$L имя_файла*) (см. приложение D).

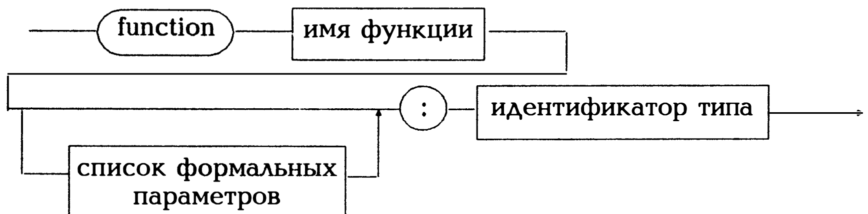
Если процедура не слишком велика, ее объектные коды можно представить в виде встроенного блока (inline-блок) в шестнадцатеричной форме записи. Тогда вызов такой процедуры равносилен вызову макрокоманды. Пример такой процедуры читатель найдет в конце главы 21 (пример 21.8).

Из соображений экономии места приведем лишь несколько примеров. Почти все приведенные ниже примеры содержат процедуры и достаточно наглядно иллюстрируют все изложенное выше.

11.2. Функции

Часто встречается случай, когда процедура генерирует лишь некоторое значение, получаемое при вычислении выражения. В математике такие процедуры называются функциями (входные величины являются аргументами). Описание функции имеет следующий вид:

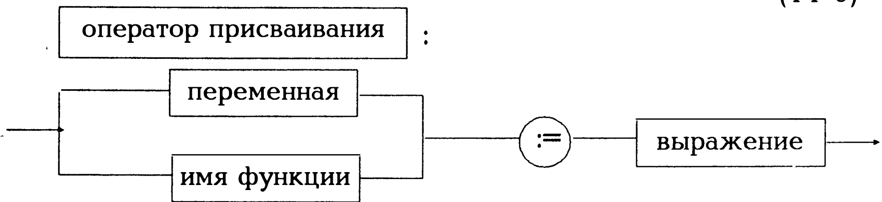




Все, что было сказано относительно процедур, справедливо и для функций со следующими поправками:

1. В теле функции имени функции должно присваиваться одно значение. Поэтому понятие присваивания значения, представленное диаграммой (10-7), расширяется.

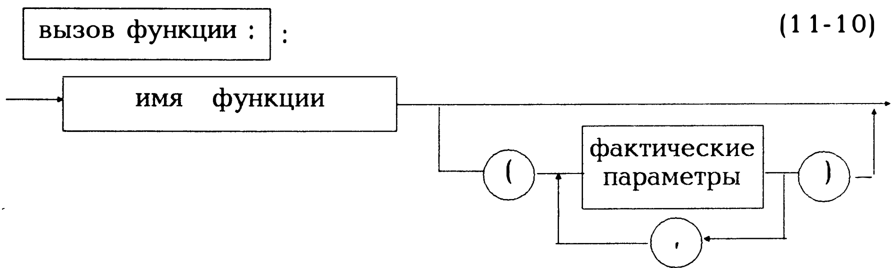
(11-9)



2. Идентификатор типа в конце заголовка функции определяет тип значения функции. Это может быть простой тип (см.гл. 12), строковый (см. раздел 14.3) или тип "указатель (см. гл. 18).

3. Функция вызывается внутри выражения таким образом:

(11-10)



Такого рода была процедура quersumme в примере 11.1. Если мы хотим, например, определить, делится ли на 3 число n, то должны будем написать:

```
quersumme(n,quer);          (*Вызов процедуры*)
if quer mod 3 = 0 then ...
```

Практичнее было бы вызвать функцию quersumme(n) внутри выражения по if:

```
if quersumme(n) mod 3 = 0 then ...
```

Пример 11.6:

В этом примере описанная в примере 11.1 процедура quersumme оформляется в виде функции:

```
program quersummen_bestimmung;
    {Определение суммы цифр}
uses crt;
var n:longint; q:integer;

(*Описание функции*)
function quersumme(x:longint):integer;
(*Заголовок функции*)
(*Описательная часть*)
var h:integer;
begin (*****Исполняемая часть функции quersumme*****)
  h :=0;
  while x <> 0 do
    begin h :=h + x mod 10; x := x div 10 end;
  if h < 0 then quersumme := -h
           else quersumme := h;
end; (* of quersumme *)
begin  (****** Операторы ********)
   checkeof := true;
   writeln('Введите целое число. Конец ввода по eof');
   while not eof do
      begin
         readln(n);
         write('Сумма цифр',n:5,'равна');
         writeln(quersumme(n):3);
         (* Вызов функции в выражении *)
      end;
end.
```

В этом примере даны все важнейшие признаки функции:

1. В теле функции имени функции присваивается некоторое значение:

```
quersumme :=h
```

Сгенерированное для функции значение связывается с именем функции. В описанной в примере 11.1 процедуре то же самое значение передавалось через ссылку q.

2. В процедуре передаваемый по ссылке параметр q имел тип integer. Поскольку теперь значение привязано к имени функции, тип его задается в конце заголовка функции:

```
function quersumme(x:longint):integer;
```

3. Вызов функции осуществляется внутри некоторого выражения:

```
write('Сумма цифр',n:5,'равна');
writeln(quersumme(n):3);
```

Напомним еще раз, что аргументы write являются выражениями.

Пример 11.7:
Требуется определить нули функции f(x) в интервале a.
Задаются некоторая функция f(x), интервал (a,b) и степень точности epsilon.

Найти: все нулевые точки функции f(x) в интервале ($a \le x \le b$) с точностью epsilon.

Способ решения: Интервал разбивается на подинтервалы длиной deltax и сканируется. Если найден подинтервал (x,x+deltax), в котором функция меняет знак, содержащий нулевую точку подинтервал делится пополам до тех пор, пока не будет найдена нулевая точка функции (с учетом заданной точности).

1. Построение алгоритма в псевдокодах

```
BEGIN
   Считать начальные значения;
   установить x в начальное значение;
   WHILE конец не достигнут DO
      BEGIN
        один шаг вправо;
        IF нулевая точка найдена
           THEN распечатать нулевую точку
                один шаг вправо
           ELSE IF найден интервал, содержащий нуль
                THEN определить нулевую точку
      END (* конец поиска *)
END.
```

2. Уточнение

```
Считать начальные значения:
        a,b,deltax,epsilon считать;

Установить начальное значение x:
        x1:=a;y1:=f(x1);x2:=x1+deltax;y2:=f(x2);
        IF abs(y1) < epsilon
           THEN нулевую точку x1 напечатать;
        IF abs(y2) < epsilon
           THEN нулевую точку x2 вывести на печать;
        IF найден интервал, содержащий нуль функции
           THEN  определить нуль;
```

Один шаг вправо:

```
        x1:=x2;y1:=y2;
        x2:=x1+deltax;y2:=f(x2);
```

Содержащий нуль функции интервал найдет:

```
        y1*y2<= 0
```

(*Тем самым учитывается и тот случай, когда нуль лежит на границе интервала*)

Определить нулевую точку:

```
        REPEAT
          mitte:=(x1 + x2)/2;
          if f(mitte)*y1>0
          then y1:=f(mitte); x1:=mitte;
          else y2:=f(mitte); x2:=mitte;
        UNTIL |f(mitte)|< epsilon;
        Нулевую точку вывести на печать;
```

```
(************** Программа *****************)
program nullstellen(input,output);
uses crt;
var a,b,deltax,epsilon,x1,x2,y1,y2:real;
c:char;
function f(x:real):real;
begin
 (* f:=2*x*(1-ln(x)) - cos(4*x) + 3; *)
   f:=(x-1)*(x-2)*(x-3);
   (* Нулевые точки для 1,2,3 *)
end;

procedure Anfangswertelesen(var a,b,deltax,epsilon:
                              real);
{процедура считывания начальных значений}
begin
  clrscr;
   writeln('a?'); readln(a);
   writeln('b?'); readln(b);
   writeln('epsilon?'); readln(epsilon);
   writeln('deltax?'); readln(deltax);
end; (*Начальные значения*)

procedure nullstelledrucken(x:real);
{процедура распечатки начальных значений}
begin
   writeln('Нулевая точка при x=',x:8:5,'     y = ');
   writeln(f(x):7:6);
end;
```

```
procedure Schrittnachrechts(var x1,y1,x2,y2:real);
{процедура сдвига на шаг вправо}
begin
   x1 := x2; y1 :=y2;
   x2 := x1 + deltax; y2 := f(x2);
end;  (*Шаг вправо*)

procedure Nullstellebestimmen(x1,y1,x2,y2:real);
{процедура определения нулевой точки}
var mitte:real;
begin
   repeat
      mitte := (x1 + x2)/2;
     if f(mitte)*y1 > 0
       then begin y1 := f(mitte); x1 := mitte end
       else begin y2 := f(mitte); x2 := mitte end;
   until abs(f(mitte)) < epsilon;
   Nullstelledrucken(mitte);
end;   (*Определить нулевую точку*)

procedure Anfangsetzen(var x1,y1,x2,y2:real);
{процедура установки начальных значений*}
begin
    x1:=a; y1:=f(x1); x2:=x1 + deltax; y2:=f(x2);
    if abs(y1) < epsilon then nullstelledrucken(x1);
    if abs(y2) < epsilon then nullstelledrucken(x2);
    if y1*y2 <= 0
       then Nullstellebestimmen(x1,y1,x2,y2);
end;

begin    (********Исполняемая часть*********)

  Anfangswertelesen(a,b,deltax,epsilon);
  Anfangsetzen(x1,y1,x2,y2);
    begin
      Schrittnachrechts(x1,y1,x2,y2);
      if abs(y2) < epsilon
        then begin
               Nullstelledrucken(x2);
               Schrittnachrechts(x1,y1,x2,y2);
             end
        else
         if y1*y2 <=0
           then Nullstellebestimmen(x1,y1,x2,y2);
    end;     (* Конец поиска *)
  write ('<RET'>); read(c);
end.
```

Поскольку функции ниже будут встречаться довольно часто, для первого знакомства приведенных примеров достаточно.

11.3. Связь процедур и функций

Центральным вопросом является вопрос о том, как программа может взаимодействовать с процедурами и функциями, то есть каким образом осуществляется обмен данными. Как можно узнать процедуру или функцию, какие данные могут существовать в программе и как программа узнает, какой результат сгенерирован процедурой или функцией? Основополагающие понятия "формальных и фактических параметров, передача параметров по значению и по ссылке, локальные и глобальные переменные уже обсуждались в разделе 11.1, но учитывая их исключительную важность, вернемся к ним еще раз применительно к процедурам и функциям. Для иллюстрации воспользуемся следующим небольшим примером.

```
var x,y,m: real;
procedure mittel;
   (* Процедура должна сгенерировать из 2 значений,
      взятых из программы, среднее значение *)
begin
   x := 3.7;        y := 2.0;
   (* процедура mittel должна определять
      среднее m двух значений x и y*)
end.
```

Для того, чтобы вычислить среднее, процедура mittel должна получить три параметра:

```
(*1*)      procedure mittel(a,b,mit:real);
           begin
              mit := (a+b)/2;
           end;
```

Тогда вызов

```
(*2*)      mittel(x,y,m)
```

не дает желаемого результата. m не равно среднему значению 2.85. Причина станет понятной, если точно проанализируете, что происходит при вызове (*2*).

```
begin
    a :=x;
    b :=y;
    mit := (a+b)/2;
end;
```

Оба параметра x,y передают свои значения 3.7 и 2.0 формальным параметрам a,b. Тогда среднее значение (a+b)/2 передается через параметр mit, в то время как это значение должно быть присвоено переменной m. Ситуацию иллюстрирует рис. 11.2. Как процедура может передать среднее значение переменной m?

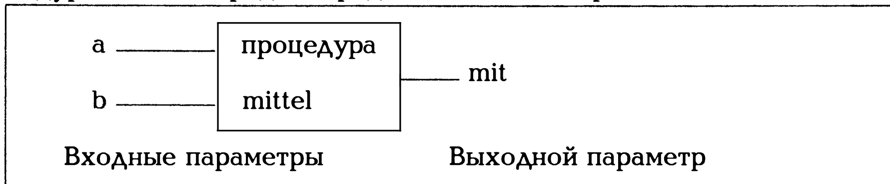


Рис. 11.2. Входные и выходные параметры процедуры mittel

Учитывая сказанное в разделе 11.1, передадим параметр mit по ссылке:

```
procedure mittel (a,b:real; var mit:real);
begin
  mit := (a+b)/2;
end;
```

Теперь вызов

```
mittel (x,y,m)
```

равносилен

```
begin
   a := x;
   b := y;
   m := (a+b)/2;
end;
```

Теперь при передаче параметра по ссылке var m:real формальный параметр mit заменяется адресом, указывающим на m, где и находится среднее значение. m присвоено нужное значение.

Пример 11.8:

```
program mittelwert;
    {программа формирования среднего значения}
var x,y,m:real;
procedure mittel (a,b:real; var mit:real);
begin
   mit := (a+b)/2;
end; (* среднее *)
```

```
begin
    writeln ('x?'); readln(x);
    writeln ('y?'); readln(y);
    mittel (x,y,m);
    writeln('Среднее значение    ',x:8:2,'  и ');
    writeln(y:8:2,'  =  ',m:8:3);
end.
```

Итак, что мы вынесли из всего сказанного? Если программе необходимо сгенерированное процедурой значение, его можно передать только по ссылке.

Наряду со связью через список параметров имеется и другая возможность: процедура использует глобальные для нее переменные, которые известны и доступны и программе и процедуре. Итак, мы получили переданный по ссылке параметр m и присвоили это среднее значение глобальной переменной.

```
program mittelwert;
var x,y,m:real;
procedure mittel (a,b:real);
begin
   m := (a+b)/2; (* m является для процедуры глобальной*)
end;
begin
    readln(x);
    readln(y);
    mittel (x,y);
    writeln('Среднее значение ',x:8:2,'  и   ');
    writeln(y:8:2,'  =  ',m:8:3);
end.
```

Использовать глобальные величины , вообще говоря, не рекомендуется. Процедура должна, в принципе, лишь изменить параметр, т.е. след, который процедура оставляет после себя в программе, должен явно заимствоваться из списка параметров.

Если же процедура, как это имеет место в нашем случае, лишь генерирует некоторое значение, лучше оформить это в виде функции. Тогда связь осуществляется по имени функции.

```
program mittelwert;
var x,y,m:real;
function mittel (a,b:real):real;
begin
   mittel := (a+b)/2;
end;
```

```
begin
  readln(x);
  readln(y);
  m := mittel(x,y);
  writeln('Среднее значение ',x:8:2, '   и    ');
  writeln(y:8:2,'  =   ',m:8:3);
end.
```

Тогда значение функции привязывается к ее имени.

11.4. Рекурсия

Рекурсией называется ситуация, когда процедура или функция сама себя вызывает. Вот типичная конструкция такого рода:

```
procedure proc(i:integer);
begin
  anweisungen1;
  if bedingung then proc(i+1);
  anweisungen2;
end;
```

Вызов proc(1) означает, что proc вызывает себя раз за разом с помощью proc(2), proc(3),.. до тех пор, пока условие bedingung не отменит новый вызов. При каждом вызове выполняестя оператор anweisungen1, после чего порядок выполнения операторов прерывается новым вызовом proc(i+1). Чтобы для каждого вызова был отработан и оператор anweisungen2, все локальные переменные процедуры сохраняются в стеке. Стеком является структура магазинного типа LIFO (Last In First Out), т.е. если, например, при proc(10) условие более не выполняется, anweisungen2 выполняется со значениями, обрабатываемыми в обратном порядке для proc(9),...,proc(1). Локальные параметры помещаются в стек один за другим и выбираются из стека в обратной последовательности (латинское recurrere означает «возвращение назад»).

В Паскале можно пользоваться именами лишь тогда, когда в тексте программы этому предшествует их описание. Рекурсия является единственным исключением из этого правила. Имя proc можно использовать сразу же, не закончив его описания.

Пример 11.9 представляет собой бесконечную рекурсию, с помощью которой можно установить, насколько велик стек. При этом помните, что при использовании директивы (*$S+*) при переполнении стека получим сообщение об ошибке, а при использовании директивы (*$S-*) - нет, а значит, мы скорее всего столкнемся с зависанием системы. Установкой по умолчанию является (*$S+*) (см. Options/Compiler "Stack checking ON"). Программа будет пре-

рвана с выдачей сообщения об ошибке "Error 202: stack overflow error ("Ошибка 202: переполнение стека").

Пример 11.9:

```
program stack_test;
  {программа проверки стека}
procedure proc(i:integer);
begin
   if i mod 1024 = 0
      then writeln(i:6);
   proc(i+1);
end;

begin
   proc(1);
end.
```

Стек связан с другой структурой памяти - с динамической областью. С помощью директивы (*$M*) можно управлять размером стека. К рассмотрению динамической области мы вернемся в разделе 18.2.

Рекурсия не должна восприниматься как некий программистский трюк. Это скорее некий принцип, метод. Если в программе нужно выполнить что-то повторно, можно действовать двумя способами:

- с помощью последовательного присоединения (или итерации в форме цикла);
- с помощью вложения одной операции в другую (а именно, рекурсии).

В следующем примере 11.10 один раз счет от 1 до n ведется с помощью цикла, а второй - с помощью рекурсии. При этом хорошо видно, как заполняется, а затем освобождается стек. В процедуре rekursion операция writeln(i:30) выполняется перед рекурсивным вызовом, после чего writeln(i:3) освобождает стек. Поскольку рекурсия выполняется от n до 1, вывод по команде writeln(i:30) выполняется в обратной последовательности n,n-1,...,1, а вывод по команде writeln(i:3) - в прямой последовательности 1,2,...,n (согласно принципу LIFO - последним пришел, первым обслужен).

Пример 11.10:

Показывает принципиальное различие между итерацией и рекурсией: итерации необходим цикл и локальная переменная k как переменная цикла. Рекурсии ничего этого не требуется!

```
program iterativ_zu_rekursion;
 var n:integer;

 procedure rekursion (i:integer);
 begin
    writeln(i:30);
    if i < 1 then rekursion(i-1);
    writeln(i:3);
 end;  (* Рекурсия *)

 procedure schleife(i:integer);
 var k:integer;
 begin
    k :=1;
    while k <= i do begin
                      write(k:3);
                      k :=k+1;
                    end;
 end; (* Цикл *)

 begin
    write('Введите n:'); readln(n);
    writeln('Пока:');
    schleife(n);
    writeln;
    writeln('Рекурсия:');
    rekursion(n);
 end.
```

Пример 11.11:

Рекурсивная процедура convert переводит десятичное число z в восьмеричную систему путем деления его на 8 и выдачи остатка в обратной последовательности.

```
program dezimal_oktal_konvertierung;
   {преобразование из десятичной системы в восьмеричную}
var z:integer;

procedure convert(z:integer);
begin
   if z > 1 then convert(z div 8);
   (* Это рекурсивный вызов *)
    write(z mod 8:1);
end;
```

```
begin
   writeln('Введите некоторое положительное число:');
   readln(z);
   writeln('Десятичное число:',z:6);
   write('Восьмеричное число:     ');
   convert(z);
end.
```

Один из наиболее ярких примером применения рекурсии дают числа Фибоначчи. Они определяются следующим образом:

```
x[1]=x[2]=1
x[n]=x[n-1]+x[n-2] при n > 2
```

Каждый элемент ряда Фибоначчи является суммой двух предшествующих элементов, т.е.

1 1 2 3 5 8 13 21 34 55 ...

Следующий пример позволяет вычислить n-ный элемент ряда Фибоначчи как итеративно (то есть в цикле, начиная с x[1] до x[n]), так и рекурсивно (n-ный элемент ряда является суммой двух предшествующих элементов). Причем рекурсивная функция вызывает себя дважды.

Пример 11.12:

С использованием рекурсии вычисляются числа Фибоначчи, причем глубина рекурсии индицируется. Перед каждым рекурсивным вызовом выводится ASCII-символ с номером 8 (Backspace), а после вызова вновь стирается. Тем самым можно наблюдать за работой программы, поскольку программа за счет delay(300) приостанавливается на 0.3 с.

```
program fibonacci(input,output);
uses crt;
var n,result:integer;

function fibit(n:integer):integer;
var a,b,c,i:integer;
begin
   a := 1; b := 1;
   if (n=1) or (n=2)
      then fibit :=1
      else begin
            for i := 3 to n do
```

```
            begin c := a+b; a := b; b := c; end;
            fibit :=c;
          end;
end;

function fibrek(n:integer):integer;
begin
  write('-'); delay(300);
  if (n=1) or (n=2)
     then fibrek := 1
     else fibrek := fibrek(n-1)+fibrek(n-2);
  write(chr(8),' ',chr(8)); delay(300);
end;

begin
  clrscr;
  write('n = ');
  readln(n);
  writeln('Итеративно:',fibit(n):5);
  writeln('Рекурсивно:');
  write('                ....!....#....!....#....');
  writeln('!....#....!....#....!....#');
  write  ('Глубина рекурсии:');
  result := fibrek(n);
  writeln;
  write(result);
end.
```

Этот пример демонстрирует прежде всего различия между итерацией и рекурсией. Итерации необходим цикл и вспомогательные величины; итерация сравнительно ненаглядна (см. fibit в приведенном выше примере). Рекурсия обходится без вспомогательных величин и обычно проще для понимания, что демонстрирует следующая запись:

```
if (n=1) or (n=2) then fibrek := 1
                   else fibrek := fibrek(n-1)+fibrek(n-2);
```

Итерация требует меньше места в памяти и машинного времени, чем рекурсия, которой необходимы затраты на управление стеком. Итак, если для некоторой задачи возможны два решения, предпочтение следует отдать итерации. Правда, для многих задач рекурсивная формулировка совершенно прозрачна, в то время как построение итерации оказывается весьма сложным делом. В разделе 19.2 даны примеры этого для двоичных деревьев.

Если процедура или функция вызывает себя сама, это называют прямой рекурсией. Но может встретиться ситуация, когда процедура A вызывает процедуру B, вызывающую C, а процедура C вновь обращается к A. В этом случае мы имеем дело с косвенной рекурсией, что демонстрирует приведенный ниже пример. С таким типом рекурсии мы сталкиваемся там, где использована директива forward.

Пример 11.13:

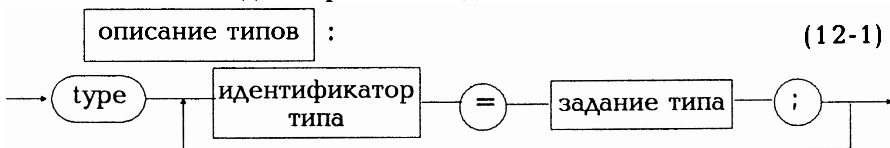
Следующая программа выдает простые числа от 1 до n, для чего используются функции next и prim, которые вызываются перекрестно, то есть рекурсивно. Одновременно это является примером применения директивы forward.

```
program primzahlen_rekursiv_berechnen;
   {программа рекурсивного вычисления простых чисел}
var n,i : integer;
    c : char;

function next(i:integer):integer;forward;
{Это прямая ссылка вперед на функцию next,
 которая будет определена позже}

function prim(j:integer):boolean;
{prim имеет значение true, если j простое число,
 и false в противном случае}
var k:integer;
begin
   k :=2;
   while (k*k <= j) and (j mod k <> 0) do
       k := next(k);
    {k пробегает последовательность простых чисел, начиная
     с 2, вплоть до корня из j, при этом проверяется,
     делится ли j на одно из таких простых чисел. При этом
     используется следующая функция next}
  if j mod k = 0 then prim := false
                 else prim := true;
end  {prim};

function next;
{Параметры уже стоят в ссылке вперед,
 next вычисляет, каково следующее за j простое число}
var l:integer;
begin
   l := i+1;
```

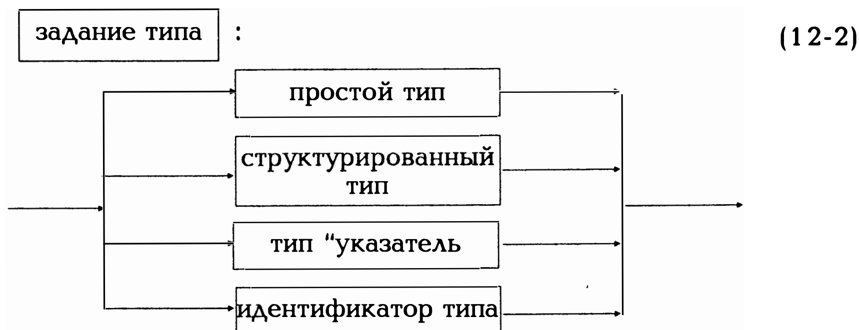
```
      while not(prim(l)) do l := l+1;
      {Итак, next вызывает в свою очередь prim}
      next :=1;
   end  {next};
   begin   (******* Исполняемая часть программы *******)
     write('Введите положительное число n,');
     writeln('Должны быть определены все');
     writeln('предшествующие ему простые числа');
     readln(n);
     for i :=2 to n do
       begin
          if prim(i) then writeln(i:14)
                     else writeln(i:4);
          if i mod 23 = 0 then begin
                      write('<RET>':60);  read(c,c);
                    end;
       end;
   end.
```

12. Типы и идентификаторы типов

В той части программы (4-5), где описываются типы используемых данных, пользователь может описать собственные типы данных. При этом типу данных присваивается имя и устанавливается область значений для переменных данного типа.

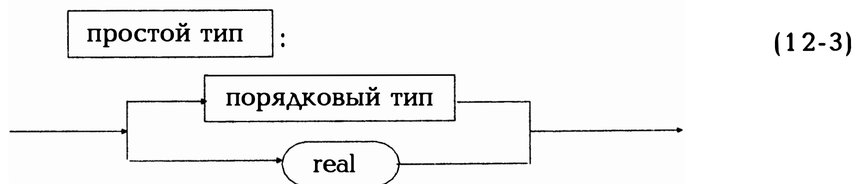


После служебного слова type перечисляются описания типов. Имя переменной и идентификатор типа разделяются знаком равенства =.



Задание типа означает, что теперь можно в разделе переменных при их описании пользоваться вновь введенным идентификатором.

Простые типы описываются в следующей главе 13, структурированные данные - в главе 14, тип "указатель" - в главе 18.



Здесь идентификатор real используется для типов real, extended, single, double и comp. Порядковые типы рассматриваются в главе 13.

Наименования типов данных должны отвечать всем правилам, существующим для любых других имен, а значит, они должны прежде всего отличаться от всех других имен. Для наименований типов справедливы и правила, касающиеся глобальных и локальных величин (см. раздел 11.1). Рассмотрим в связи с этим следующий небольшой пример.

Пример 12.1:

Познакомьтесь с этим небольшим примером, вспомнив действующие для наименований типов правила. В какой строке следует ожидать сообщения компилятора об ошибке?

```
program parametertest;

(*1*) type ganz = integer;
      var  j : integer;
(*2*) procedure probe(a:ganz);
(*3*)     type fest = ganz;
(*4*)          ganz = boolean;
(*5*)     var  vf : fest;
(*6*)          vg : ganz;
               vi : integer;
   begin
(*7*)          vf := a;
(*8*)          vi := a;
(*9*)          vg := a;
   end  (*Конец теста*);

begin
(*10*)   probe(5);
end.
```

Ответ: При компиляции строки (*9*) будет выдано сообщение об ошибке "type mismatch ("несоответствие типов"). В строке (*1*) идентификатор типа ganz вводится в качестве другого имени для integer. Имя действительно для всего блока. Процедура probe в строке (*2*) имеет значение параметра типа ganz, так что обращение в строке (*10*) корректно. В строке (*3*) тип fest вводится как равнозначный с типом integer согласно (*1*). В строке (*4*) мы сталкиваемся с противоречием, поскольку локальное имя ganz является согласно строке (*1*) глобальным. Это означает, что в строке (*4*) имя ganz будет определено заново, то есть начиная со строки (*4*) локальная переменная ganz получает тип boolean, в то время как глобальная переменная ganz из строки (*1*) утрачивает силу для остальной части процедуры probe. Итак, переменная vf из строки (*5*) имеет тип integer, переменная vg из строки (*6*) тип boolean. Поскольку формальный параметр a имеет тип integer, операторы присваивания (*7*) и (*8*) корректны; оператор же (*9*) некорректен, так как здесь переменной типа boolean должно присваиваться значение типа integer.

Это прекрасный пример того, что глобальная переменная действительна в блоке более низкого уровня иерархии до тех пор, пока она не будет переопределена заново.

После этого примера мы может вернуться к вопросу из раздела 10.2 : а что же такое идентичные типы?

Два типа t1 и t2 называются идентичными, если выполнено одно из двух следующих утверждений: t1 и t2 имеют один и тот же тип или t1 описан как эквивалент идентичному t2 типу. Обратимся к примеру:

```
type t1 = integer;
     t2 = t1;
var a:t1; b := t2; (*Типы t1 и t2 идентичны integer*)
                   (*Итак, a :=b; является допустимым
                    оператором присваивания*)

type ganz : integer;
     t1 = ganz;
     t2 = integer;

var a :t1; b:t2 (*t1 и t2 идентичны*)
```

Две переменные a и b также имеют идентичный тип, им при их описании соответствует одно и то же задание типа:

```
var a,b: typangabe;
```

Для оператора присваивания v := e требуется, чтобы выражение e и переменная v имели идентичный тип. Из этого правила есть исключение, когда e имеет тип integer, а v тип real. Для оператора присваивания запрещен, в принципе, тип данных file (см. главу 17).

Примечания относительно версии 5.0

Здесь различаются задание типа и идентификаторы типа. При задании типа тип определяется сразу (возможно через уже существующее наименование типа). В этом смысле все стандартные типы данных, перечисленные в главе 5 (integer, word, longint, real, extended, char, boolean и пр.) относятся к идентификаторам типа. При описании переменной согласно диаграмме (7-1)

```
var variablenname : typangabe;
```

Итак, если записано

```
var x,y : real;
```

мы задаем тип через стандартный идентификатор типа real. Как мы только что видели, после x,y: можно непосредственно определить и некоторый новый тип данных. Учтите, что формальные параметры процедур и функций согласно (11-3) должны специфицироваться идентификатором типа. Для версии 5.0 такой принцип последовательно распространен и на сами процедуры и функции. Так согласно

```
type func = function (x,y:integer):integer;
```

func является идентификатором типа функций двух целочисленных переменных x,y с целочисленными значениями функции. Итак, весьма логично, что процедура или функция может иметь также формальный параметр типа func:

```
procedure nanu(x:real;i:integer: fkt:func; p:boolean);
```

Тогда соответствующим фактическим параметром для fkt будет как раз имя описанной в другом месте программы функции такого типа. Этот случай демонстрируется на следующем примере:

Пример 12.2:

Процедура operation связывает два аргумента x,y. Тип связи (сложение или умножение) реализуется через параметр типа func.

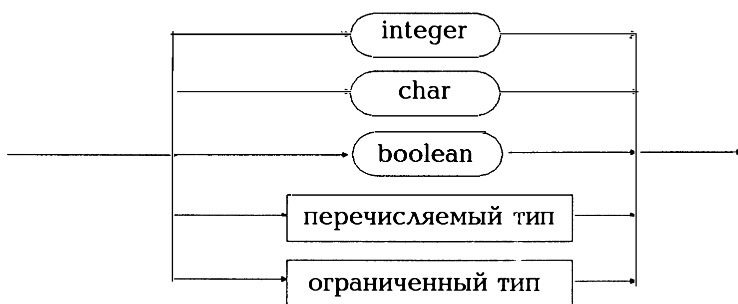
```
program prozedur_parameter;
uses crt;
(*$F+*)  (см. приложение D*)
type func= function(x,y:integer):integer;
var i,k,m:integer;

procedure
operation(x,y:integer; var z:integer; fkt:func);
begin
  z := fkt(x,y);
end;  (*конец операции*)

function add(x,y:integer):integer;
begin
  add := x + y;
end;

function mult(x,y:integer):integer;
begin
  mult := x*y;
end;

begin  (*******Исполняемая часть программы********)
   writeln('i=');    readln(i);
   writeln('k=');   readln(k);
   operation(i,k,m,add);
   writeln (i:4,k:4,M:4);
   operation(i,k,m,mult);
   writeln(i:4, k:4, m:4);
end.
```

Полноты ради скажем, что и в стандартном Паскале процедуры и функции могут являться параметрами. Но там это реализовано менее удобно, чем в Турбо Паскале версии 5.0.

13. Простые типы данных

В (12-3) было введено понятие простых типов данных. Они называются так потому, что "не могут состоять ни из каких других типов . Здесь же было использование понятие порядкового типа. В математике порядковым числом является номер элемента при перечислении. Под порядковым типом понимают тип данных, областью значений которых является упорядоченное счетное множество. Каждому элементу такого множества соответствует некоторое порядковое число, являющееся как раз его номером при перечислении.

порядковый тип : (13-1)



Здесь под integer понимаются также типы byte, word, shortint, longint. Целые числа можно пересчитать. К типу данных char относится множество символов ASCII, тип данных boolean имеет только два значения true и false. Очевидно, данные типа real сюда не относятся. Какое порядковое число может отвечать значению 56.777321? Правда, real является простым, но не порядковым типом (см.12-3).

В силу этого свойства порядковых типов, для них существуют функция следования (successor) и функция предшествования (predecessor); порядковым типом является один из (см. рис. 13.1). Итак:

```
ord('z') = 122        succ('z') = '{'     pred('z') ='y'
ord(true) = 1         ord(false) = 0
ord(0) = 0            succ(45) = 46
```

Понятие порядкового типа уже встречалось ранее. Например, для оператора for

```
for i := anfang to ende do anweisung;
```

i должно иметь порядковый тип. Длина шага для to - succ(i), pred(i) - для downto. Итак, для i:char запись

```
for i := 'a' to 'z' do anweisung
```

также была бы правильной. i пробегает все маленькие буквы. Для оператора case

```
case ausdruck of
```

значение выражения должно иметь порядковый тип. После того, как в главе 5 были рассмотрены первые три порядковых типа

данных, согласно (13-1) осталось лишь рассмотреть перечисляемый и ограниченный типы.

Вызов:	succ(x)
Значение функции:	порядковый тип x
Параметры:	x:порядковый тип
Действие:	последователь x в описании типа (если таковой существует)
Вызов:	pred(x)
Значение функции:	порядковый тип x
Параметры:	x:порядковый тип
Действие:	предшественник x в описании типа (если таковой существует)
Вызов:	ord(x)
Значение функции:	integer
Параметры:	x:порядковый тип
Действие:	номер x в описании типа (начиная с 0)

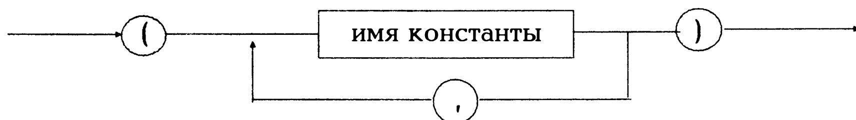
Рис. 13.1. Функции порядкового типа

13.1. Перечисляемый тип

Для перечисляемого типа значения, которые может принимать переменная этого типа, перечисляются по именам.

Значения образуют некоторое упорядоченное множество и являются константами этого типа. Если x является переменной пере-

перечисляемый тип : (13-2)



числяемого типа, для нее, естественно, существуют три стандартные функции, представленные на рис. 13.1.

Для констант перечисляемого типа существуют операторы сравнения. Для оператора for в качестве переменной цикла допускается использование переменной этого типа. Длина шага при прямом счете задается с помощью succ, при обратном счете - с помощью pred.

Помните, что для read и write недопустимо использование переменных и значений перечисляемого типа. Для этого нужно писать собственные процедуры.

Пример 13.1:

Следующие процедуры демонстрируют применение переменных перечисляемого типа.

```
program aufzaehlung;
type color = (rot, gruen, blau, gelb);
var f,g:color;

procedure druckfarbe(x:color);
begin
  case x of rot :write('rot':6);
            (*Обратите внимание на различие между
            константой rot и строкой 'rot' (="красный")*)
            gruen:write('gruen':6);
            blau :write('blau':6);
            gelb :write('gelb':6)      end;
  end;

  begin
    f:=rot;
    g:= succ(f);    {Это зеленый}
    if g < blau then write('меньше')
                else write('не меньше');
                     {Итак, напечатан меньший}
    for g := rot to gelb do druckfarbe(g);
end.
```

Естественно, оператор read(f) недопустим. Нужно было бы для этой цели написать процедуру liesfarbe(var x:color).

Пример 13.2:

Постоянная kinderzahl позволяет расставить детей в ряд. С помощью некоторой "считалочки silbenzahl подсчитываются слоги. Следующая программа должна продемонстрировать процесс счета на экране дисплея. В примерах 14.7 и 16.3 мы вернемся к программе и удалим пересчитанных детей из ряда.

```
program kinder_auszaehlen;
{программа пересчета детей}
uses crt;
(*Имеется kinderzahl детей, которые пересчитываются
  ведущим "считалочки  silbenzahl в соответствии с
  числом слогов*)
const kinderzahl = 8;
type namenstyp =
     (eva,otto,hans,inge,karl,ulla,monika,gabi);
var  kind:namenstyp;
     i,silbenzahl:integer;
procedure anfaenger_lesen(var k:namenstyp);
{процедура считывания начинающего}
var c:char;
begin
```

```
   writeln('Кто должен начинать?');
   writeln('Введите начальную букву:');
   writeln('eva otto hans inge karl ulla');
   writeln(' monika gabi');
   readln(c);
   case c of
     'e' : k := eva;
     'o' : k := otto;
     'h' : k := hans;
     'i' : k := inge;
     'k' : k := karl;
     'u' : k := ulla;
     'm' : k := karl;
     'g' : k := gabi          end;
end;    (* Считать начинающего *)

procedure schreibe(k:namenstyp);
begin
   case k of
     eva :  writeln('eva ');
     otto:  writeln('otto');
     hans:  writeln('hans');
     inge:  writeln('inge');
     karl:  writeln('karl');
     ulla:  writeln('ulla');
     monika :  writeln('monika ');
     gabi:  writeln('gabi')          end;
end; (* Запись *)

procedure anfaenger_zeigen(k:namenstyp);
{Показать начинающего}
begin
   gotoxy(10,ord(k)+1);
   write('<— начинает');  delay(3000);
   gotoxy(10,ord(k)+1);  write('             ');
end;  (* Показать начинающего *)

procedure anzeigen(k:namenstyp);
{ Процедура указания выбывающего из игры }
begin
   gotoxy(10,ord(k)+1);
   write ('<—Выйди вон!');
   delay(1000);
   gotoxy(10,ord(k)+1);  write('             ');
   delay(1000);
   gotoxy(1,ord(k)+1);
   lowvideo; schreibe(k); highvideo; delay(1000);
```

```
end;    (* Указать *)

procedure anfangsstellung;
{Установка в исходное положение}
var k:namenstyp;
begin
   clrscr;
   for k := eva to gabi do schreibe(k);
   gotoxy(50,1);
   write('Число слогов:',silbenzahl:3);
   anfaenger_zeigen(kind);
end;  (* Установка с исходное положение *)

procedure abzaehlen(var k:namenstyp);
{Процедура пересчета}
var i : integer;
begin
   for i := 1 to silbenzahl do
     if k < gabi then k := succ(k) else k := eva;
end;   (* Пересчитать *)

begin    (******* Исполняемая часть *******)
   clrscr;
   writeln('Сколько слогов в считалочке?');
   readln(silbenzahl);
   anfaenger_lesen(kind);
   clrscr;
   anfangsstellung;
   for i := 1 to kinderzahl do
      begin
        abzaehlen(kind);
        anzeigen(kind);
      end;
   gotoxy(1,20);
   end.
```

Стандартные типы данных char и boolean являются перечисляемыми типами. Для char множеством допустимых значений является все множество кодов ASCII, поэтому

```
ord('y') = 121
succ('*') = '+'
pred('V') = 'U'
```

Тип boolean определяется следующим соотношением:

```
type boolean = (false,true);
```

то есть

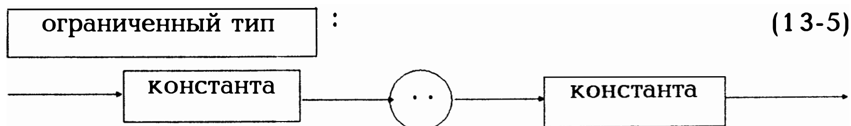
```
false < true
ord(false) = 0
pred(true) = false
```

В соотношении (9-1) была введена операция приведения типов. И выражение и результат должны иметь порядковый тип. В то время как функция ord предполагает присвоение значению перечисляемого типа его номера, наоборот, с помощью операции приведения типов можно получить для некоторого номера соответствующее значение.

Для
```
type farbe = (kreuz,pik,herz,karo)
```
получим
```
ord (karo) = 3      farbe(3) = karo
```
Тогда и
```
boolean(1) = true
```

13.2. Ограниченный тип

Для каждого порядкового типа можно в качестве множества значений определить некоторый интервал:



Первая константа задает наименьшее, а вторая наибольшее значение переменной этого типа. Естественно, вторая константа не может быть меньше первой.

```
type letter     = 'a'..'z';
     lottozahl  = 1..49;
     tag        = (mon,die,mit,don,frei,sam,son)
     werktag    = mon .. frei;
var w:werktag;  lot:lottozahl; let:letter;
```

Тогда переменные w,lot и let могут принимать значения только из указанного интервала. Стандартный тип данных byte является интервалом типа integer:

```
type byte = 0..255;
```

Настоятельно рекомендуем пользоваться ограниченным (интервальным) типом по следующим причинам:

- Программу будет легче читать.
- Если интервал содержит не более 256 элементов, для его хранения нужен лишь 1 байт (таков стандартный тип данных byte=0..255).
- Воспользовавшись директивой компилятору (*$R+*) (см. приложение D) можно при выполнении программы контролировать соблюдение установленных для значений границ.

Относительно директивы $R следует сделать небольшое замечание. Стандартной установкой является $R-, т.е. контроль должен быть включен явно. Причем выход за границы заданного интервала может обнаружиться лишь при компиляции, если попытаться некоторой переменной i=1..10 присвоить значение i :=13; т.е. за пределами данного подмножества. Чаще всего выход значения за пределы заданного интервала можно обнаружить только во время выполнения программы.

Пример 13.3:

С помощью директивы $R+ проверим соблюдение заданной области изменения значений:

```
(*$R+*)
program unterbereichs_grenzen;
var i,k:1..10;
begin
  writeln ('i ');
  readln(i);  (*Сообщение об ошибке, если введенное
                значение лежит вне диапазона 1..10*)
writeln (i:8);
i := 13;  (*Ошибка при компиляции согласно директиве
            $R+*)
i := 4;
k := 10*i;  (*Ошибка при выполнении*)
end.
```

В разделе 9.2 при описании оператора присваивания подчеркивалось, что переменная слева и стоящее справа выражение должны иметь одинаковый тип. Очевидно, что выражение может иметь ограниченный тип; тогда допустима следующая форма записи:

```
var i :='a'..'z'; c:char;
    c :=i;
```

То же самое справедливо и применительно к параметрам. Фактический параметр может передать значения, соответствующие интервалу, определяемому формальным параметром. Для ссылочных параметров должна строго соблюдаться идентичность типов.

Пример 13.4:
Следующая программа может помочь смоделировать игру в лото:

```
(*$r+*)

program zahlenlotto;
{Программа задания чисел для игры в лото}
var i:1..6; lottozahl:1..49;
begin (******** Исполняемая часть программы **********)
  for i := 1 to 6 do
     begin
        lottozahl := random(49) + 1;
        writeln('Nr.',i:2,lottozahl:6);
    end;
end.
```

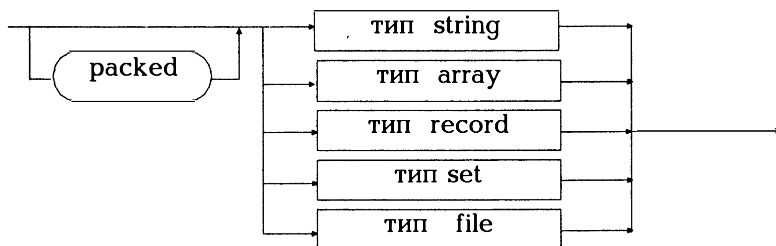
Эта программа имеет тот недостаток, что не все шесть выпавших в лото чисел могут оказаться различными. Этот недостаток легко устранить, воспользовавшись описанным в главе 16 типом данных "множество" (пример 16.4).

14. Тип array

14.1. Типы структурированных данных

Мы начинаем рассмотрение структурированных типов данных с типа данных array. Для начала приведем несколько общих положений об этих типах данных.

Структурированные данные составляются из других типов данных, т.е. структурированные данные имеют в своем составе некоторые компоненты.

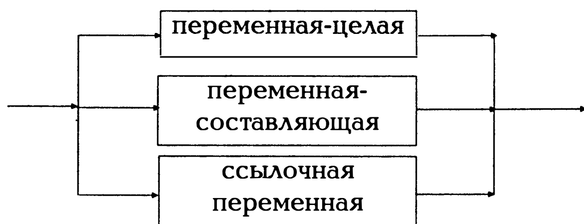


Служебное слово packed должно обеспечить наиболее эффективное размещение в памяти постоянных этого типа. Поскольку в Турбо Паскале это и так имеет место всегда, использование данного служебного слова в Турбо Паскале никакого действия не оказывает и лишь предусматривается на случай использования соответствующей программы в стандартном Паскале без внесения дополнительных изменений.

Типы структурированных данных обсуждаются в следующих главах. Итак, существуют переменные, значения которых могут строиться по очень сложным формулам. При поименовании переменных следует обратить внимание, должно ли значение переменной пониматься как "нечто целое" **или** как некоторая составляющая.

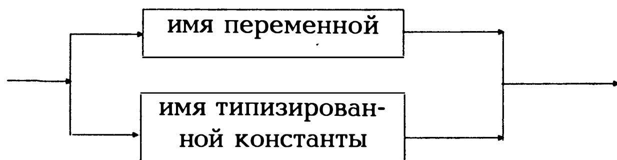
Поэтому для переменных различают:

переменная : (14-2)



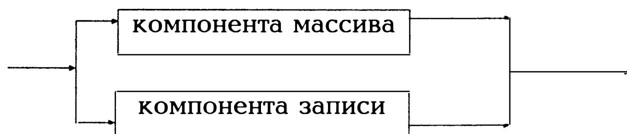
Под переменной-целой понимается некоторая переменная, воспринимаемая как "нечто целое", т.е. как совокупность всех компонент, из которых она состоит. Все описанные в главе 13 простые типы данных являются такими переменными (хотя они и не содержат никаких компонент). Структурированные переменные имеют компоненты, которые сами по себе могут быть переменными. С математической точки зрения, вектор является переменной, но переменной являтеся и каждая из его компонент. Вектор является переменной-целой, его компоненты - переменными-составляющими. Вместе с описанием переменной-целой описываются одновременно и переменные, являющиеся ее компонентами.

переменная-целая : (14-3)



Согласно (7.1) и (6.4) для описания переменных и типизированных констант переменные могут определяться как переменные-целые, а не как переменные-составляющие.

переменная-составляющая : (14-4)

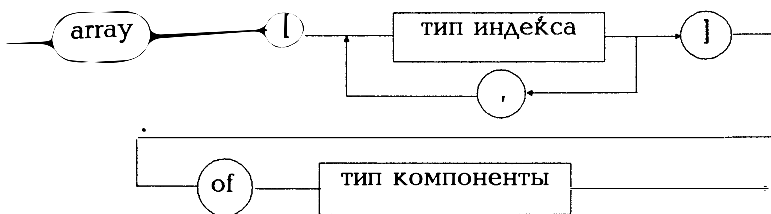


Переменные составляющие используютя с типами данных "массив" (array) и "запись" (record). Последние рассматриваются в разделах 14.2-14.4 и в главе 15. К ссылочные переменные реализуются через тип данных "указатель", рассмотренный в главе 18.

Массив - это совокупность фиксированного числа одинаковых компонент. Каждую компоненту можно снабдить индексом. Для описания массива требуется задать: тип компонент и тип индекса.

Тип данных "массив" описывается следующим образом:

тип array : (14-5)

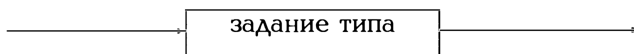


Для типа индекса имеет силу следующая диаграмма:

тип индекса : (14-6)

Итак, тип индекса реализуется упорядоченным множеством значений, устанавливающих количество и порядок следования компонент. При этом следует учитывать, что сегмент данных может иметь длину только 64 Кбайта, т.е. использовать array [integer] нельзя.

тип компоненты : (14-7)



Компоненты могут иметь какой-либо один тип. В следующих главах приведено много примеров этого.

14.2. Одномерные массивы

Согласно диаграмме (14-5) допускается использовать несколько индексов. Однако, рассмотрим вначале одномерные массивы. В математике такие структуры называются векторами. Массив имеет фиксированное количество однородных компонент и описывается числом и типом этих компонент. Значения такого типа хранятся в оперативной памяти. К каждой компоненте можно обращаться непосредственно. Описание типа имеет следующий вид:

```
type наименование_типа = array[индекс] of тип_компонент;
```

Индекс должен иметь порядковый тип и задавать число компонент. Если индекс выражается целыми числами, берется интервал целых чисел. На рис. 7.1 приведено описание простой переменной, на рис. 14.1 - описание элементов массива, состоящего из трех вещественных чисел.

Имя переменной:	x	(Переменная-целая)
Задание типа	array[1..3] of real;	
Значение	3.23 -2.5 12.5	

Элементы массива

Имя переменной	x[1]	x[2]	x[3]	(Переменные-составляющие)
Задание типа	real	real	real	
Значение	3.23	-2.5	12.5	

Рис. 14.1. Составляющие описания массива

С помощью описания

```
    var x : array[1..3] of real;
или
    type feld = array[1..3] of real;
    var x : feld;
```

одновременно с описанием переменной-целой x описываются и переменные-составляющие x[1], x[2], x[3] типа real.

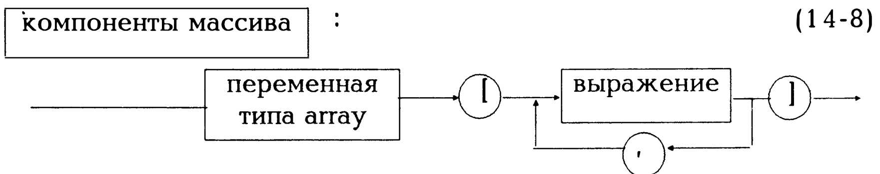
Согласно рис. 5.6 можно для такой переменной x типа feld с помощью sizeof(feld) или sizeof(x) определить потребность в памяти.

Пример:

```
type vektor       = array[1..20] of real;
     zeichenkette = array[0..10] of char;
     feld         = array['a'..'z'] of integer;

type vektor       = array[1..10] of real;
     tag          = (mon,die,mit,don,frei,sam,son);
     arbeitszeit  = array[mo..frei] of real;
     uppercase    = 'A'..'Z';
     letter       = array[uppercase] of integer;
var v:vektor; a:arbeitszeit; l:letter;
```

Переменные v, a и l являются переменными-целыми. Их значения устанавливаются через описание типа: v состоит из 10 вещественных чисел, a - из 5 вещественных чисел, a l из 26 целых чисел.

С помощью приведенного выше описания типа одновременно описываются и компоненты массива.



Итак, за именем массива следует индекс в квадратных скобках; каждый элемент массива имеет тип, указанный в конце описания. Согласно приведенным выше описаниям переменная

```
v[2]    имеет тип real,
a[die] имеет тип real,
l['G'] имеет тип integer.
```

Обратите внимание на синтаксическое понятие "задание типа". Обе формулировки

```
type feld = array['a',,'z'] of boolean;
var f:feld;
```

и

```
var f:array['a'..'z'] of boolean;
```

равносильны, поскольку задание типа согласно (12-2) может восприниматься и как указание некоторого типа и как задание наименования типа. Если наименование типа feld в программе не используется, можно выбрать вторую формулировку. Так как согласно (11-30) формальные параметры процедуры/функции требуют наименования типа (или полного отсутствия задания типа), то здесь рекомендуется первая формулировка.

Пример 14.1 показывает, как использовать массив в качестве параметров и в операторах присваивания. Прежде всего отметим, что все сказанное ранее о передаче параметров по значению и по ссылке остается в силе и для массивов. Первый параметр x процедуры rechts_shift передается по значению. Итак, из фактического массива a формируется копия, которая передается в x. В процедуре эта копия x сдвигается вправо. После вызова процедуры исходный массив a остается без изменения. Второй параметр y процедуры rechts_shift передается по ссылке, следовательно, в y будет передан адрес фактического массива b. В результате вправо сдвигается исходный массив b. Отсюда вывод: обращение по ссылке к массиву выполняется быстрее и требует меньше памяти. Если процедура изменяет формальный массив, изменяется и фактический. Впрочем, в процедуре lies_array параметр x должен передаваться по ссылке (так как в противном случае фактический массив a не получит никаких значений), в то время как параметр процедуры druck_array должен передаваться по значению или по ссылке (выводится лишь одна копия или оригинал).

Пример 14.1:

Массив считывается, сдвигается на одну позицию вправо, после чего выдается результат.

```
program shift (input,output);
const n = 8;
type feld = array[1..n] of integer;
var a,b : feld;
```

```
    i:integer;

  procedure lies_array(var x : feld);
  {процедура считывания массива}
  var i : integer;
  begin
    writeln('Введите',n:3,'чисел:');
    for i := 1 to n do read(x[i]);
  end;

  procedure druck_array(var x : feld);
  {процедура распечатки массива}
  var i : integer;
  begin
    for i := 1 to n do write(x[i]:4);
    writeln;
  end;

  procedure rechts_shift(x:feld; var y:feld);
  {процедура сдвига вправо}
  var i:integer;
  begin
    for i := n downto 2 do x[i] := x[i-1]; x[1] := 0;
    for i := n downto 2 do y[i] := y[i-1]; y[1] := 0;
  end;

  begin      (****** Исполняемая часть ******)
    lies_array(a);
    b := a;
    writeln('Массив перед сдвигом вправо:');
    druck_array(a);
    druck_array(b);
    rechts_shift(a,b);
    writeln('Массив после сдвига вправо:');
    druck_array(a);
    druck_array(b);
  end.
```

Следует сделать небольшое замечание относительно оператора присваивания

```
  b := a;
```

Естественно, для записанной слева переменной и стоящего справа выражения должно иметь место соответствие типов. Но тогда оператор присваивания означает, что b получит копию значения a.

Пример 14.2:

Следует отсортировать максимум 1000 чисел. С помощью таймера можно приостанавливать выполнение в интервале между двумя звуковыми сигналами

```
program sortieren_fuer_testlauf;
uses crt;
const n   = 1000;
type feld = array[1..n] of integer;
var  z:feld;
     anz,i:integer;

procedure druckfeld(f:feld);
var i:integer;
begin
  for i := 1 to 10 do write(f[i]:6); writeln;
  for i := anz-9 to anz do write(f[i]:6); writeln;
end;  (* Поле для печати *)

procedure sortieren(var a:feld; anz:integer);
var i,j,x:integer;
begin (* Сортировка "в лоб", называемая также
       пузырьковой сортировкой *)
  for  i := 2 to anz do
     for j := anz downto i do
          if a[j-1] > a[j] then  (*замена*)
                 begin
                    x:=a[j-1];
                    a[j-1]:=a[j];
                    a[j]:=x;
                 end;
end;  (* сортировка *)

begin    (******* Исполняемая часть *******)
   write('Сколько чисел');
   writeln(' должно быть отсортировано?');
   writeln('Задайте n <= 1000');
   readln(anz);
   for  i := 1 to anz do z[i] := random(10000);
   druckfeld(z);
   (* Неотсортированы первые и последние 10 чисел *)
   writeln('Теперь все в порядке");
   delay(1000);
   write(^g); (* или write(chr(7)), что равносильно*)
```

```
    sortieren(z, anz);
    writeln(^g'fertig');
    druckfeld(z);
    (* Первые и последние 10 чисел отсортированы *)
end.
```

Использованный метод сортировки (последовательное сравнение двух соседних элементов) просто записывается и легко понимается. Но этот способ сортировки неэффективен. Лучшими вариантами являются так называемые динамическая сортировка, быстрая сортировка и другие, описанные в примере 14.6. В главе 21 показано, как вообще можно использовать таймер. Там время между двумя звуковыми сигналами заменяется заданием времени в явном виде.

Пример 14.3:

Следующий пример использует массив array['a'..'z'] of integer, т.е. индексом является строчная буква, а значение элемента массива целое число. Такой массив может использоваться для того, чтобы определить частоту появления букв в тексте. h['r'] = 12 означает, что ранее были считаны 12 букв r. С помощью h['r']:=h['r']+1 или inc(h['r']) число считанных букв r увеличивается на 1.

```
program buchstabenzaehlen;
uses crt;
type haeufigkeit = array['a'..'z'] of unteger;
var h:haeufigkeit;
    c:char;

procedure drucken(h:haeufigkeit);
var c:char;
begin
   for c := 'a' to 'z' do
      begin
         if (ord(c) - 97) mod 8 = 0 then writeln;
         write(c:5,':',h[c]:3);
      end;
end;  (* печатать *)

function ist_grossbuchstabe(c:char):boolean;
begin
   ist_grossbuchstabe := ('A' <= c) and (c <= 'Z');
end;
```

```
function wird_kleinbuchstabe(c:char):char;
begin
  wird_kleinbuchstabe := chr(ord(c) + 32);
end;

function ist_kleinbuchstabe(c:char):boolean;
begin
  ist_kleinbuchstabe := ('a' <= c) and (c <= 'z');
end;

begin  (********* Исполняемая часть *********)
  checkeof :=true;
  (* h установить в нуль: *)
  for c := 'a' to 'z' do h[c] := 0;
  writeln('Введите текст. Конец ввода по ^Z');
  while not eof do

    begin
      read(c);
      if ist_grossbuchstabe(c)
        then c := wird_kleinbuchstabe(c);
      if ist_kleinbuchstabe(c)
        then h[c] := h[c]+1;
    end;
  drucken(h);
  writeln;
end.
```

В разделе 17.2 будет показано, как считать текст из записанного на дискете файла.

Пример 14.4:

При выборах партиями получены голоса избирателей. После этого места в парламенте распределяются согласно методу наибольшего числа д'Хондта.

```
program wahl;
type partei      =(cdu, csu, spd, fdp, gruene);
     stimmen_typ = array[partei] of longint;
     mandate_typ = array[partei] of longint;

var stimmen:stimmen_typ;
    mandate:mandate_typ;
    abgeordnetenzahl:integer;
```

```
procedure drucke_partei(p:partei);
begin
   case p of
     cdu :write('Христианско-демократический союз':8);
     csu :write('Христианско-социальный союз':8);
     spd :write('Социал-демократическая партия':8);
     fdp :write('Свободная демократическая партия':8);
     gruene:write('Партия "зеленых"':8)  end;
end {список партий};

procedure stimmeneinlesen(var st:stimmen_typ);
var p:partei;
begin
  writeln ('Подсчитайте отданные за партии голоса:');
  for p := cdu to gruene do begin
  drucke_partei(p);
  write(':        ');
  readln(st[p]);
                            end;
end;  {подсчет голосов}

procedure mandate_vergeben(st:stimmen_typ;
                           var man:mandate_typ);
type dhondt_zahl_typ = array[partei] of real;
var vergebene_sitze:integer;
    max:real;
    p,maxp:partei;
    dh:dhondt_zahl_typ;

begin
  for p := cdu to gruene do dh[p] := st[p];
  {Число мандатов установить в нуль}
  for p := cdu to gruene do man[p]:=0;
  vergebene_sitze:=0;
  while vergebene_sitze  abgeordnetenzahl do
      begin
           max:=cdu;
           max:=dh[cdu];
           for p := csu to gruene do
             if dh[p] > max then
                 begin
                   max := dh[p];
                   max := p;
                 end;
```

```
            man[maxp]:=man[maxp]+1;
            {Партия maxp получила 1 голос}
            vergebene_sitze:=vergebene_sitze+1;
            dh[maxp]:=st[maxp]/(man[maxp]+1);
        end;  {Распределение мандатов}

procedure ergebnis_drucken(st:stimmen_typ;
                           man:mandate_typ);
var p:partei;
begin
  writeln; write('Партии:');
  for p := cdu to gruene do drucke_partei(p);
  writeln;
  write('Голоса:');
  for p := cdu to gruene do write(st[p]:8);
  writeln;
  write('Мандаты:');
  for p := cdu to gruene do write(man[p]:8);
  writeln;
end;      {Распределение мандатов}

begin       (******* Исполняемая часть ********)
  write('Сколько мест должно быть выдано:');
  readln(abgeordnetenzahl);
  stimmeneinlesen(stimmen);
  mandate_vergeben(stimmen,mandate);
  ergebnis_drucken(stimmen,mandate);
end.
```

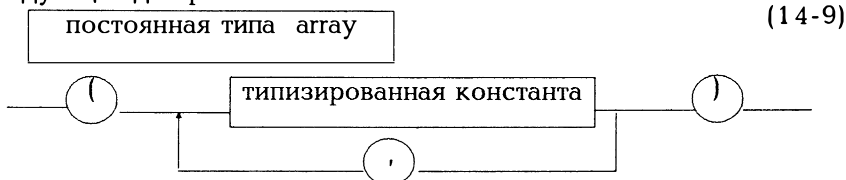
Символьные строки, то есть массивы типа,

```
type wort = array[1..8] of char;
```

рассматриваются в разделе 14.4. При компиляции можно выбрать, нужно ли контролировать во время выполнения программы соблюдение границ заданного интервала или нет. Для этого используется директива компилятору {$R-} и {$R+} (см. приложение D).

Согласно изложенному в главе 6 в части описания констант для массива можно задать начальные значения его компонент. В (6-5) введено понятие постоянной типа "массив". Для нее справедлива следующая диаграмма:

(14-9)



Такое описание говорит лишь о том, что значения отдельных компонент должны считываться из массива. Поскольку компоненты могут быть какого-то типа, они являются, вообще говоря, типизированными константами.

Константы должны иметь значения, соответствующие типу компонент.

```
type vektor = array[1..5] of real;
const a:vektor = (1.0,1.0,1.0,2.5,2.5);

type farbe = (gelb,gold,schwarz,rot,blau,weise);
     fahne = array[1..3] of farbe;
const brd:fahne = (schwarz,rot,gold);
```

Следует сказать еще об одной особенности. Поскольку массив имеет некоторое фиксированное число элементов, корректно следующее описание:

```
const min  = 5; max = 12;
type feld = array[min..max] of boolean;
```

Поскольку типизированные константы являются, в принципе, переменными, описание

```
const min:integer=5;
      max:integer=12;
type feld = array[min..max] of boolean
```

некорректно, так как min и max могут принять в ходе выполнения программы другие значения.

Две важнейшие операции над массивами - сортировка и поиск. При поиске по некоторому массиву следует, очевидно, различать, отсортированы или нет элементы массива. Если они не отсортированы, то при поиске нужно просто просматривать все элементы массива подряд, пока нужный элемент не будет найден. Если массив отсортирован, процесс поиска можно существенно ускорить. В этом случае просматриваемый интервал делится пополам и определяется, в какой части - левой или правой - лежит искомое значение. Если искомое значение находится в левой половине интервала, она вновь делится пополам и все повторяется заново. Такой поиск называется методом половинного деления. В приведенном ниже примере берется массив случайных чисел; массив

сортируется, после чего с применением обоих указанных выше методов поиска осуществляется поиск элемента x.

Пример 14.5:

```
program suchen;
{программа поиска}
uses crt;
const max = 100;
type feld = array[1..max] of integer;
var f:feld; x:integer;

procedure binaersuchen(x:integer; f:feld):
(* Поиск методом половинного деления.
   Массив должен быть отсортирован.*)
var i,rechts,links,mitte:integer;
    found:boolean;
begin
   links:=1; rechts:=max;
   repeat mitte := (links+rechts) div 2;
          if x < f[mitte] then rechts:=mitte-1
                          else links:=mitte+1;
          found:=x=f[mitte];
   until found or (links>rechts);
   if found
      then
        begin
          write(x:3,'при i= ',mitte:3);
          writeln('  успешный поиск');
        end
      else writeln(x:3,'  безрезультатный поиск')
end;  (* по методу половинного деления*)

procedure feld_anlegen(var f:feld);
var i:integer;
begin
  for i :=1 to max do f[i] := random(99)+1;
end;  {заполнение массива}

procedure druckfeld(f:feld);
var i:integer;
begin
   for i := 1 to max do write(f[i]:4);
   writeln;
end;   { поле печати}

procedure suchen(x:integer; f:feld);
(* Простой перебор сверху вниз *)
```

```
var i:integer; found:boolean;
begin
  i := 1; found := false;
  while (i <= max) and not found do
    begin
      found := x = f[i];
      inc(i);
    end;
  if found
    then writeln(x:3,' при i = ',i-1:3,' элемент найден')
    else writeln(x:3,' элемент не найден')
end;  (* поиск *)

procedure sortiere(var f:feld);
var i,j,hilf:integer;
begin
  (* Здесь просто сравниваются два соседних элемента
    и соответствующим образом меняются местами (см.
    следующий пример); это простейший, но к сожалению,
    малоэффективный метод сортировки *)
  for i := 1 to max-1 do
     for j :=i+1 to max do
        if f[j] < f[i] then {f[i] и f[j] меняются
                                местами}
           begin
              hilf := f[i];
              f[i]:=f[j];
              f[j]:=hilf
           end;
end;    {сортировка}

begin       (********** Исполняемая часть **********)
   feld_anlegen(f);
   druckfeld(f);
   sortiere(f);
   druckfeld(f);
   writeln('Какое число искать? (Конец = 0)');
   readln(x);
   while x <> 0 do begin
                     suchen(x,f);
                     binaersuchen(x,f);
                     write('Какое число');
                     writeln('искать?');
                     readln(x);
                  end;
end.
```

Центральной проблемой обработки данных является сортировка множества данных. Существует большое число методов сортировки. Следующий пример демонстрирует три элементарных способа поиска, добавления и замены элементов массива с некоторыми их модификациями (heapsort, shellsort, quicksort). Эти методы можно охарактеризовать лишь очень кратко. Интересующийся читатель найдет более подробные описания в книге Вирта [7].

Пример 14.6:

```
(* Здесь приведено три наиболее распространенных
   алгоритма, применяемых при сортировке массива:
   перебор, добавление, замена. Реализуется основной
   принцип сортировки. Каждый способ  допускает
   модификацию. Они известны
     для: перебора          как heapsort
          добавления            shellsort
          замены               quicksort.
  Частично отсортированный массив состоит из:
     выходной последовательности =
       уже отсортированный кусок в начале массива;
     исходной последовательности =
       неотсортированный кусок в конце массива.   *)

program sortieren;
{программа сортировки}
uses crt;
const n = 1000;
type feld = array[-9..n] of integer;
            (* -9 по методу shellsort *)
var a:feld;
    anz, nr:integer;

procedure eingabe(var f:feld);
var i:integer;
begin
   for i := 1 to anz do f[i] := random(99)+1;
end;  (*Ввод*)

procedure druckfeld(f:feld);
(* Выдаются первые 20 элементов *)
var i:integer;
begin
   for i := 1 to 20 do write(f[i]:4); writeln;
end;    (* Поле печати *)
```

```
procedure austausch(var a:feld);
(* Последовательно сравниваются между собой два
   соседних элемента и соответствующим образом
   меняются местами. Это самый примитивный способ
   сортировки (называемый также пузырьковым методом
   или методом британского музея). *)

var i,j,x:integer;
begin (* Прямая замена *)
   for i := 2 to anz do
      for j := anz downto i do
          if a[j-1] > a[j] then (* поменять местами *)
                   begin
                     x :=a[j-1];
                     a[j-1]:=a[j];
                     a[j]:=x;
                   end;
end;     (* Замена *)

procedure quicksort(var a:feld);
(* Число операций перемены местоположения элементов
  внутри массива значительно сократится, если менять
  местами далеко отстоящие друг от друга элементы. Для это-
го выбирается для сравнения один элемент x (наиболее
  целесообразно выбрать элемент из середины массива),
  отыскивается слева первый элемент, который не меньше x,
  а справа первый элемент, который не больше x. Найденные
  элементы меняются местами. После первого же прохода
  все элементы, которые меньше x, будут стоять слева от x,
  а все элементы, которые больше x, - справа от x. С двумя
  половинами массива поступают точно так же. Из-за такой
  рекурсии метод оформляется как процедура. *)

procedure quicks(links,rechts:integer);
var i,j,x,w:integer;
begin
  i:=links; j:=rechts;
  x:=a[(links+rechts) div 2];
  repeat
     while a[i] < x do i:=i+1;
     while x < a[j] do j:=j-1;
     if i <= j then begin
     w:=a[i]; a[i]:=a[j]; a[j]:=w;
     i:=i+1; j:=j-1;
                end;
  until i > j;
```

```
  if links < j  then quick(links,j);
  if i < rechts then quick(i,rechts);
end;   (* quicks *)

(* Работа с алгоритмом заключается тогда
   в серии отдельных обращений *)
begin
  quicks(1,anz);
end;  (* quicksort *)

procedure einfuegen(var a:feld);
(* Из исходной последовательности берется следующий
   элемент и добавляется в выходной массив, причем
   для него с шагом 1 ищется соответствующее место,
   начиная с конца массива. *)
var i,j,x:integer;
begin  (* Непосредственное добавление *)
   for i := 2 to anz do
      begin
         x:=a[i]; a[0]:=x; j:=i-1;
         while x < a[j] do
               begin a[j+1]:=a[j]; j:=j-1 end;
         a[j+1]:=x;
      end;
end;  (* Добавление *)

procedure shellsort(var a:feld);
(* Алгоритм добавления выполняется t раз с уменьшающим
   ся  каждый раз шагом s[1], s[2],...,s[t] для "следующего
   x". Пусть s[1]=anf, а s[t]=1. Для того, чтобы устано-
   вить конечную метку для добавления, нужно массив а
   сначала продлить на начальную длину шага anf. Итак,
   нужно задать type feld = array[-anf..n] of integer;
   Для выбора шага рекомендуются, например, 40, 13, 4, 1
   или 31, 15, 7, 3, 1 или 9, 5, 4, 1 *)
var s:array[1..4] of integer;
    marke, m, t, i, j, k, x:integer;
begin (* shellsort *)
  t := 4; s[4] := 1; s[3] := 3; s[2] := 5;
          s[1] := 9;
  for m :=1 to t do
   begin
     k:=s[m]; marke:=-k;
     for i := k+1 to anz do
       begin
         x:=a[i];
```

```
        j:=i-k;
        if marke = 0 then marke:=-k;
        marke:=marke+1;
        a[marke]:=x;
        while x < a[j] do
           begin a[j+k]:=a[j]; j:=j-k end;
        a[j+k]:=x;
     end;
  end;
end; (* shellsort *)

procedure auswahl(var a:feld);
(* Из исходной последовательности выбираются
   те элементы, которые следует добавить в
   конец выходной последовательности *)
var i,j,k,x:integer;
begin  (* Прямой выбор *)
  for i := 1 to anz-1 do
     begin
       k:=i; x:=a[i];
       for j:= i+1 to anz do
       {В оставшейся части ищется наименьший элемент}
       if a[j] < x then
                begin
                   k:=j;
                   x:=a[j];
                end;
       a[k]:=a[i]; a[i]:=x;
     end;
end;  (* Перебор *)

procedure heapsort(var a:feld);
(* При выборе сохраняется появляющаяся по пути
   информация о соотношениях между элементами
   (теряющаяся при прямом переборе), так что
   следующий шаг выбора значительно сокращается.
   Согласно предварительному условию о том, что
   место в памяти должно использоваться лишь для
   хранения a, весь массив a предварительно
   упорядочивается таким образом, чтобы всеми
   элементами выполнялись следующие соотношения:
      a[i] >= a[2i] для всех i=1,...,n/2
      a[i] >= a[2i+1]
   Упорядоченный таким образом массив называется
   "кучей" (heap - динамическая область). Вначале
   в состояние "кучи" приводится левая половина
```

```
   массива a. Затем берется первый элемент справа
   (поскольку он имеет наибольшее значение), правая
   граница сдвигается влево на 1 и остальной массив
   вновь отфильтровывается, чтобы опять получить "кучу
   Затем повторяется тот же процесс. Итак, в отличие
   от простого перебора выходная последовательность
   формируется справа. *)

var rechts, links, x:integer;
procedure sieb;
(* Массив a, как и переменные links, rechts
   является глобальным *)
var i,j:integer;
begin
   i:=links; j:=2*i; x:=a[i];
   while j <= rechts do
     begin if j < rechts then
              if a[j] < a[j+1] then j:=j+1;
                              if x < a[j]
                                 then
                                   begin
                                      a[i]:=a[j];
                                      i:=j;
                                      j:=2*i
                                   end
                                 else j:=rechts+1;
     end;
 a[i]:=x;
end  (* Фильтрация *);

begin   (* heapsort *)
   rechts:=anz;
   for links := (anz div 2) downto 1 do sieb;
   (* В результате получим массив в форме "кучи" *)
   (* Теперь начнем сортировать *)
   while rechts > 1 do
     begin links:=1;
           x:=a[links];
           a[links]:=a[rechts];
           a[rechts]:=x;
           rechts:=rechts-1;
           sieb;
     end
end;   (* heapsort *)
```

```
begin  (******* Исполняемая часть ********)
  write('Сколько элементов ( <=1000):');
  readln(anz);
  eingabe(a);
  clrscr;
  druckfeld(a);
  writeln(anz:50);
  writeln('Какой метод:');
  writeln('1 = einfuegen');
  writeln('2 = shellsort');
  writeln('3 = auswaehlen');
  writeln('4 = heapsort');
  writeln('5 = austauschen');
  writeln('6 = quicksort');
  readln(nr);
  writeln('Внимание:'); delay(500); write(^g);

  case nr of
    1: einfuegen(a);
    2: shellsort(a);
    3: auswahl(a);
    4: heapsort(a);
    5: austausch(a);
    6: quicksort(a);
    else writeln('Ничего не нужно'); end;
  write(^g);
  writeln('Выполнить с сортировкой:');
  druckfeld(a);
end.
```

Пример 14.7:

Здесь, как и в примере 13.2 речь пойдет о пересчете детей. Но теперь ряд детей будет массивом. Следующие ребенок определяется не с помощью считалочки, а с помощью индекса и выходит из игры. Образовавшееся "окно" закрывают следующие дети.

```
program kinder_auszaehlen;
{Программа пересчета детей}
uses crt;
(* Имеется некоторое число kinderzahl детей, которых
  ведущий пересчитывает с помощью детской считалочки,
  построенной на подсчете слогов. *)
const kinderzahl = 8;
type namenstyp = (eva,otto,hans,inge,karl,ulla,
                 monika,gabi);
```

```
     kreistyp = array[1..kinderzahl] of namenstyp;
var kind:namenstyp;
    kreis:kreistyp;
    kind_index,rest,silbenzahl:integer;

procedure anfaenger_lesen(var k:namenstyp);
var c:char;
begin
  writeln('Кто будет ведущим?');
  writeln('Введите начальную букву имени:');
  write ('Eva Otto Hans Inge Karl Ulla');
  writeln('       Monika Gabi');
  readln(c);
  case c of
    'e' : k := eva;
    'o' : k := otto;
    'h' : k := hans;
    'i' : k := inge;
    'k' : k := karl;
    'u' : k := ulla;
    'm' : k := monika;
    'g' : k := gabi     end;
end;   (* Указать ведущего *)

procedure schreibe(k:namenstyp);
begin
    case k of
      eva:    writeln('Eva');
      otto:   writeln('Otto');
      hans:   writeln('Hans');
      inge:   writeln('Inge');
      karl:   writeln('Karl');
      ulla:   writeln('Ulla');
      otto:   writeln('Otto');
      monika: writeln('Monika');
      gabi:   writeln('Gabi')      end;
end;  (* Запись *)

procedure anfaenger_zeigen(k:namenstyp);
begin
   gotoxy(10,ord(k)+1); write('<— начинает');
   delay(3000);
   gotoxy(10,ord(k)+1); write('              ');
end;    (* Указать ведущего *)
```

```
procedure anzeigen(k:namenstyp);
{ Процедура указания выбывающего из игры }
begin
   gotoxy(10,ord(k)+1);
   write ('<—Выйди вон!');
   delay(1000);
   gotoxy(10,ord(k)+1);  write('             ');
   delay(1000);
   gotoxy(1,ord(k)+1);
   lowvideo; schreibe(k); highvideo; delay(1000);
end;    (* Указать *)

procedure anfangsstellung;
{Установка в исходное положение}
var k:namenstyp;
begin
   clrscr;
   for k := eva to gabi do schreibe(k);
   gotoxy(50,1); write('Число слогов:',silbenzahl:3);
   anfaenger_zeigen(kind);
end;  (* Установка с исходное положение *)

porcedure aufstellen;
var i:integer;
begin
    kreis[1] := eva;
    for i := 2 to kinderzahl do
        kreis[i] := succ(kreis[i-1]);
    rest := kinderzahl;
end;   (* Установить *)

procedure abzaehlen(var k:integer);
{Процедура пересчета}
begin
   k:= (k+silbenzahl-2) mod rest + 1;
end;   (* Пересчитать *)

procedure herausnehmen(k:integer);
var i:integer;
begin
    for i := k+1 to rest do kreis[i-1]:=kreis[i];
    rest := rest-1;
end;        (* Исключить из игры *)

begin    (******* Исполняемая часть *******)
   clrscr;
```

```
    writeln('Сколько слогов в считалочке?');
    readln(silbenzahl);
    aufstellen;
    anfaenger_lesen(kind);
    anfangsstellung;
    kind_index := ord(kind)+1;
    while rest > 1 do
      begin
        abzaehlen(kind_index);
        anzeigen(kreis[kind_index]);
        herausnehmen(kind_index);
      end;
  gotoxy(1,20);
  end.
```

В заключение скажем несколько слов об очень больших массивах, таких как например

```
var a:array[1..20000] of real;
```

Каждое вещественное число занимает в памяти 6 байт, следовательно, для записи вектора a требуется 120000 байт. Но сегмент данных может иметь длину лишь 64 Кбайта. Попытка компиляции программы, содержащей описание такого большого массива, приведет к выдаче сообщения

Structure too large
(Структура имеет слишком большой размер)

Если непременно нужно работать с таким количеством вещественных чисел, имеется возможность воспользоваться так называемыми динамическими переменными, хранящимися в динамической области. Об этом будет говориться в главе 18.

14.3. Многомерные массивы

Согласно определению (14-5), массив может иметь несколько индексов. Массивы с двумя индексами называются в математике матрицами. Рассматривать матрицы можно с двух точек зрения. Первая из них опирается на представление, что матрица является прямоугольным упорядочением чисел. Согласно

```
type matrix = array[1..6,1..8] of integer;
var  a:matrix;
```

a является матрицей размерности 6x8 с 6 строками и 8 столбцами. a[2,3] - целое число, стоящее в третьем столбце второй строки матрицы. С другой точки зрения, a есть массив из 6 строк по 8 элементов в каждой строке, а a[2][3] - третий элемент второй строки. В принципе, оба представления равнозначны, и в Паскале можно использовать обе формы записи a[2,3] или a[2][3].

Пример 14.8:

Матрица считывается и проверяется на симметричность относительно главной диагонали.

```
program matrix_test;
uses crt;
const n = 4;
type matrix = array[1..n,1..n] of integer;

(* Эта запись идентична
      zeile = array[1..n] of integer;
      matrix = array[1..n] of zeile;    *)

var a:matrix;

procedure liesmatrix(var a:matrix);
var i,k:integer;
begin
   writeln('Введите',n*n:6,'   чисел');
   for i := 1 to n do
     for k := 1 to n do read(a[i,k]);
end;

function symmetrisch(a:matrix):boolean;
var i,k:integer;
begin
    symmetrisch :=true;
    for i := 2 to n do
        for k := 1 to i-1 do
           if a[i,k] <> a[k,i]
           then begin
                  symmetrisch := false;
                  exit;
                end;
end;
```

```
begin        (******* Исполняемая часть *******)
   liesmatrix(a);
   if symmetrisch(a) then writeln('Матрица симметрична')
                     else writeln('Матрица несимметрична')
end.
```

Естественно, индекс может иметь лишь порядковый тип.

Пример 14.9:

Вновь считывается некий текст и определяется, как часто встречается в нем пара букв. Для этого формируется матрица размерности 26*26 с большими буквами в качестве индексов: h['E','I'] - это число, показывающее, как часто в тексте встречается пара EI.

```
program buchstaben_paerchen;
uses crt;
type letter = 'A'..'Z';
     haeufigkeit = array[letter,letter] of integer;

var h : haeufigkeit;
    c, c1, c2 : char;

function ist_buchstabe(var c:char):boolean;
begin
   c := upcase(c);
   if ('A' <= c) and (c <= 'Z')
      then ist_buchstabe := true
      else ist_buchstabe := false;
end;

begin      (******* Исполняемая часть *******)
   checkeof := true;
   for c1 := 'A' to 'Z' do
      for c2 := 'A' to 'Z' do h[c1,c2] :=0;
   writeln('Введите текст. Конец ввода по ^Z');
   read(c1);
   while not eof do
     begin
       read(c2);
       if ist_buchstabe(c1) and ist_buchstabe(c2)
          then h[c1,c2] := h[c1,c2] + 1;
       c1 := c2;
     end;
   writeln ('     ');
   write('       ');
```

```
        for c1 := 'A' to 'Z' do write(c1:2);
        writeln;
        for c1 := 'A' to 'Z' do
          begin
            write(c1,' ');
            if c1 = 'S' then c := readkey;
            for c2 := 'A' to 'Z' do write(h[c1,c2]:2);
            writeln;
          end;
      end.
```

В диаграмме (14-9) определены константы типа "массив". В соответствии с этим для нескольких индексов справедливы следующие записи:

```
type punkt = array[1..2] of integer;
     dreieck = array[1..3] of punkt;
const d:dreieck = ((3,12),(2,6),(5,-2));

type triangle = array[1..3,1..2] of integer;
const t:triangle = ((3,12),(2,6),(5,-2));
```

Для многомерных массивов константы для записанных справа индексов заключаются во внутренние скобки.

Естественно, можно использовать и больше двух индексов, но при этом нужно помнить, что число элементов будет быстро возрастать.

```
type raum = array[1..10,1..500,1..20] of integer;
var a : raum;
```

a является трехмерной матрицей с 10*500*20=100000 элементами, что равносильно 20 матрицам размерности 10*500. В несколько утрированном виде это можно было бы записать таким образом:

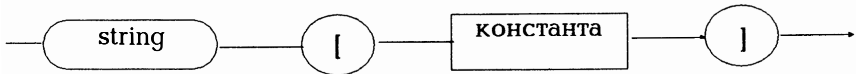
```
type zeile = array[1..64] of char;
     seite = array[1..50] of zeile;
     buch  = array[1..250] of seite;
     bibliothek = array[1..1000] of buch;
     { строка = массив от 1 до 64 символов;
       страница = массив от 1 до 50 строк;
       книга = массив от 1 до 250 страниц;
       библиотека = массив от 1 до 1000 книг}

var b:bibliothek;
```

Тогда элемент b[367, 56, 17, 10] был бы десятым символом в семнадцатой строке пятьдесят шестой страницы 367-й библиотечной книги.

14.4. String

Строкой называется последовательность символов определенной длины. Тип данных string определяется следующим образом:

тип string : (14-10)



Константа должна лежать в пределах 1...255 и задавать максимальную длину строки. Длина переменной такого типа может динамически изменяться между 1 и значением константы. Символы в строке следует воспринимать как пронумерованные интервале от 1 до значения константы. Если n - текущая длина строки, то в компоненте 0 помещается номер n последнего ASCII-символа относительно начала строки.

Пример 14.10:

Следующая программа демонстрирует считывание строки.

```
program string_test;
var s:string[10];
begin
    writeln('Задайте несколько символов:');
    readln(s);
    writeln(s,'        длина:',ord(s[0]):5);
end.
```

Этот очень простой пример показывает, что в одной строке можно записать сколько угодно символов (максимум 127), но с помощью read(s) можно считать максимум 10 символов. Прочие символы игнорируются. Но считываемых символов может быть и меньше 10. Мы вернемся к этому еще раз в конце раздела.

Согласно описанию

```
var a:string[5];
    b:string[10];
```

можно присвоить значение

```
b := a;
```

хотя типы переменных различны. При присвоении

```
a :=b;
```

в а запишутся только первые 5 символов строки b.

Строка может стоять с любой стороны оператора присваивания, имеющего 3 уровня (см. рис. 9.4). Результат будет иметь тип boolean.

```
'balkon' > 'balken'   → true
'Pascal' = 'pascal'   → false
'+' < '-'             → true
```

Обе строки сравниваются посимвольно слева направо. При обнаружении первого несовпадающего символа принимается решение о "больше" или "меньше" в соответствии с таблицей кодов ASCII. Так символ "+" стоит в этой таблице перед символом "-", "Р" перед "р", а "о" после "е". Если две сравниваемые строки имеют различную длину, но совпадают вплоть до последнего символа более короткой строки, последняя строка считается меньшей. Две строки считаются совпадающими только тогда, когда они имеют одинаковую длину и все их символы совпадают.

В Турбо Паскале имеется большое число процедур и функций для обработки строк. Прежде всего следует упомянуть о тех, которые воспринимают строку как параметр. Заголовок процедуры вида

```
procedure das(x:string[10]);
```

недопустим, поскольку string[10] не является наименованием типа, как того требует задание формального параметра. Итак, следовало бы записать:

```
type      wort = string[10];
procedure das(x:wort);
```

Но string как наименование типа данных можно использовать:

```
procedure das(x:string);
```

Тогда формальный параметр воспринимается как string[255]. Значит, имеется существенное различие, передается ли параметр по значению или по ссылке. Поскольку Паскаль чрезвычайно тре-

бователен при проверке типов, здесь имеется широкое поле деятельности для применения директивы $V (см. приложение D).

Пример 14.11:

Строка в качестве параметра.

```
(*$V-*)
(* Иначе выдается ошибка несовпадения типов для
   процедур  dies и jenes *)
program string_als_parameter;
{программа использования строки в качестве параметра}
type wort = string[10];
var  a:string[20];

procedure das(x:wort);
begin
   writeln(x,'          Длина x:',ord(x[0]):5);
end;

procedure dies(var x:wort);
begin
   writeln(x,'          Длина x:',ord(x[0]):5);
end;

procedure jenes(x:string);
begin
   writeln(x,'          Длина x:',ord(x[0]):5);
end;

begin  (********* Исполняемая часть *********)
   writeln('Введите пару символов:');
   readln(a);
   writeln(a,'          Длина a:',ord(a[0]):5);
   das(a);          (* первые 10 символов из а *)
   dies(a);            (* все символы из а *)
   jenes(a);           (* все символы из а *)
end.
```

Пример показывает, как использовать строку в качестве параметра:

- При передаче параметра по значению производится довольно поверхностная проверка на соответствие типов. Компилятор безропотно берет исходную программу, хотя для das формальный параметр x и фактический параметр a различаются по типу. Правда, при передаче параметров "лишние" символы отсекаются, что в свете сказанного выше о присвоении значений не так уж здорово. Если a состоит более, чем из 10 символов, x передаются только

первые 10 символов. Директива компилятору $V в этом случае никакой роли не играет.

- При передаче параметра по ссылке проверка типов требует полного их соответствия. Компилятор выдает сообщение об ошибке "type mismatch" (несовпадение типов). Если кому-то это покажется неудобным, он может отключить на время проверку, воспользовавшись директивой $V-. Стандартной установкой является $V+, т.е. контроль за совпадением типов включен (см. приложение D). Но согласно директиве $V- обработке ведется не так, как это имело место при передаче параметров по значению. Поскольку теперь в процедуру dies передается начальный адрес a, процедура выдает все символы из a, в том числе те, которые выходят за пределы 10 указанных в описании символов! Согласно этому замечанию, в следующих процедурах и функциях при использовании строковых параметров можно опускать их длину. Итак, помните о передаче параметров по значению и по ссылке.

Вызов: delete(s,p,n) Процедура
Параметры: var s:string;p,n:integer;
Действие: В строке s, начиная с позиции p, стираются следующие n байтов. Если p+n > максимальной длины s , стираются только эти байты в строке. Если p> максимальной длины s, никакие символы не стираются.

Вызов: insert(q,z,p) Процедура
Параметры: q:string;var z:string; p:integer;
Действие: В выходную строку z, начиная с позиции p, добавляется исходная строка q. Новая строка z обрывается при длине z.

Вызов: str(v,s) Процедура
Параметры: v:integer или v:real; var s:string;
Действие: Заданное в форме v:d (v:integer) или v:d:s (v:real) числовое значение преобразуется в строку и запоминается в s (см. описание оператора write в разделе 8.2).

Вызов: val(s,v,c) Процедура
Параметры: s:string; var v:integer; или var v:real; var c: integer;
Действие: Если строка s стоит из цифр, они преобразуются в некоторое числовое значение и передаются переменной v (как и в операторе read, описанном в главе 8). Если нет ошибок, c=0; в противном случае c получает значение разряда первого содержащего ошибку символа из s, a v становится неопределенной. С помощью директивы $R можно установить для v выход за пределы заданной области.

Вызов: concat(s1,s2,...,sn) Значение функции:string
Параметры: s1,s2,...,sn:string

Действие:	Строки s1,s2,...,sn записываются одна за другой. Если результат превысит 255 символов, строка обрывается.
Вызов:	copy(s,p,n) Значение функции:string
Параметры:	s:string; p,n:integer;
Действие:	Из строки s, начиная с позиции p, берутся n символов. Если p > length(s), берется 0. Если p+n > length(s), из строки s берутся лишь имеющиеся в строке символы, начиная с позиции p.
Вызов:	length(s) Значение функции:integer
Параметры:	s:string
Действие:	Определяется длина s, т.е. число символов, из которых состоит строка s.
Вызов:	pos(q,z) Значение функции:integer
Параметры:	q,z:string;
Действие:	Просматривается выходная строка z, чтобы отыскать, где впервые встречается исходная строка q. Значением функции является номер такой позиции. Если q в строке z не обнаружена, значение функции равно 0.

Рис. 14.2. Некоторые процедуры/функции для обработки строк

Процедуры str и val делают то же самое, что и операторы read и write с числами. Вот несколько примеров их использования.

```
var i,c:integer; x:real;
    s:string[10];

begin
  i:=324; x:=23.567;
  str(i:6,s);       {соответствует: s:='   324'}
  str(x:6:2,s);     {соответствует: s:' 23.57'}
  s:='5867';
  val(s,i,c);       {соответствует: i:=5867; c:=0}
  s:='3.567';
  val(s,x,c);       {соответствует: x:=3.567; c:=0}
  s:='123  ';
  val(s,i,c);       {соответствует: i:=?????; c:=4}
end.
```

Существуют выражения, имеющие своим значением строку. Строковые выражения могут состоять из строковых констант, строковых переменных, строковых функций и знаков операций над строками. Для строки существует лишь одна операция + (уровень 2 на рис. 9.2). Использование + позволяет разместить строки одну за другой:

```
'NEU' + '-' + 'ULM' = 'NEU-ULM'
```

Итак, операция + дает тот же результат, что и функция concat:

```
concat('NEU','-','ULM') = 'NEU-ULM'
```

что равносильно

```
var titel:string[10];
titel := 'Dr.' + 'Ing.';
```

Функция concat присутствует в Турбо Паскале для обеспечения его совместимости с другими версиями Паскаля.

Пример 14.12:

Предлагается игра в угадывание слов. Для того, чтобы можно было замаскировать предлагаемое к отгадыванию слово, то есть осуществлять ввод с клавиатуры без его отображения на экране, простоты ради воспользуемся предусмотренной в версии 3.0 возможностью read(kbd,...). При использовании этого оператора вводимая с клавиатура информация на экране не повторяется. Но поскольку в версии 4.0 имя файла kbd не предусмотрено, с помощью uses turbo3 вызовем версию 3.0.

```
program wort_raten;
{программа угадывания слов}
uses crt,turbo3;
var gedacht, geraten:string[20];
    anzahl:integer;
(* gedacht, geraten, Anzahl для используемых
   в программе процедур являются глобальными *)

procedure eingabe;
var sl,i:integer;
begin
   clrscr;
   write('Введите слово, которое  нужно отгадать:');
   read(kbd,gedacht);
   sl := length(gedacht);
   for i := 1 to sl do
       gedacht[i] := upcase(gedacht[i]);
   geraten := '————————————————';
   delete(geraten,sl+1,20-sl);
   anzahl := 0;
```

```
end;     (* Ввод *)

procedure zeigen;
begin
   clrscr;
   writeln('угадали:',geraten);
   writeln; writeln('попытайтесь',11-anzahl:3,'еще раз');
end;  (* Вывод на экран *)

procedure raten;
var c:char; i:integer;
begin
   writeln('Знаете слово? j/n:'); readln(c);
   case c of
   'j','J': begin writeln('Какое?');
                readln(geraten);
                for i := 1 to length(geraten) do
                   geraten[i] := upcase(geraten[i]);
                if gedacht <> geraten
                   then writeln('неверно')
                   else writeln('браво');
                halt;
            end;
   'n','N': begin writeln('Какая буква?');
                readln(c);
                for i := 1 to 20 do
                if gedacht[i] = upcase(c)
                   then geraten[i] := upcase(c);
                anzahl := anzahl+1;
            end; end; (* case *)
end;        (* Отгадывание *)


begin   (******** Исполняемая часть *********)
    eingabe;
    repeat
      zeigen;
      raten;
    until (gedacht = geraten) or (anzahl = 10);
end.
```

Пример 14.13:

Здесь будут считаны n чисел и отсортированы в соответствии с таблицей ASCII.

```
program string_sortieren;
{программа сортировки строк}
uses crt;
const n = 10;
type feld = array[1..n] of string[10];
var z:feld;

procedure lies(var a:feld);
var i:integer;
begin
    clrscr;
    for i := 1 to n do
      begin
         write('слово: ',i:2,'    '); readln(a[i]);
      end;
end;     (*  Считывание *)

procedure drucklinks(f:feld);
(* Выдается массив слов, выравненных по
   левому краю *)
var i:integer;
begin
   for i := 1 to n do writeln(f[i]);
end;     (*  Печать *)

procedure druckrechts(f:feld);
(* Выдается массив слов, выравненных по
   правому краю *)
var i:integer;
begin
   for i := 1 to n do writeln(f[i]:10);
end;     (*  Печать *)

procedure sortieren(var a:feld);
var i,j:integer; x:srting[10];
begin (* Прямой перебор, как и в примере 14.2 *)
   for i := 2 to n do
      for j := n downto i do
         if a[j-1] > a[j] then (* Перемена местами *)
                begin
                   x := a[j-1];
                   a[j-1]:=a[j];
                   a[j]:=x
                end;
end;  (* Сортировка *)
```

```
begin  (******** Исполняемая часть ********)
   lies(z);
   writeln('до сортировки:');
   drucklinks(z);
   sortieren(z);
   writeln(^j'после сортировки:');
   druckrechts(z);
end.
```

Хотя данные типа string относятся к структурированным данным, такие константы уже встречались среди констант в (5-4). Итак, можно просто записать

```
const city : string[10] = 'Мюнхен';
```

в результате чего city получит начальное значение 'Мюнхен'.

Поскольку в стандартном Паскале символьные последовательности используются совершенно иначе и работать там с ними весьма затруднительно, сделаем еще несколько замечаний. В стандартном Паскале

```
     const n = 6;
type zeichen = array[1..n] of char;
     wort    = packed array[1..n] of char;
var z:zeichen; w:wort;
    s:string[n];         (* в Турбо Паскале *)
```

Если переменная z состоит из n отдельных символов (различающихся индексами), w является последовательностью из n символов. Поскольку служебное слово packed в Турбо Паскале никакого действия не оказывает, в нем типы данных zeichen и wort равнозначны. Итак, чтобы написанная на стандартном Паскале программа оставалась работоспособной и в Турбо Паскале, выполняются следующие правила:

> Типы данных array[] of char и packed array[] of char стандартного Паскаля могут использоваться и в Турбо Паскале. В принципе, они обрабатываются так же, как и тип данных string.

Тогда с учетом приведенных выше соглашений в Турбо Паскале получим:

```
w:='pascal';                          Точно 6 символов!
s:=w;
writeln(s,w,length(s),length(w));     Корректно
```

s:='ulm';
w:=s; — Ошибка несовпадения типов: w должна иметь 6 знаков

delete(w,2,2) — "String variable exected" (Должна быть строковая переменная)

Отсюда ясно, что w в стандартном Паскале всегда должна иметь длину n (w[0] недопустимо), а s в Турбо Паскале имеет текущую длину s[0]. Тем самым все приведенные выше строковые процедуры, использующие s[0], принимаются, а packed array[1..n] of char в качестве параметра отклоняется.

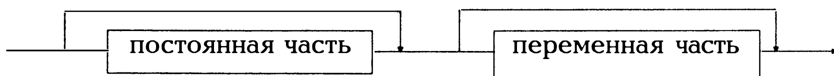
15. Тип record

Под данными типа record понимают тип данных, состоящих из фиксированного числа компонент (возможно различного типа). Такие данные называются также составными. Обычно файлы разбиваются на отдельные совокупности данных, называемые записями, которые в свою очередь разбиваются на поля. Поэтому компоненты записи называются кратко полями.

тип record : (15-1)

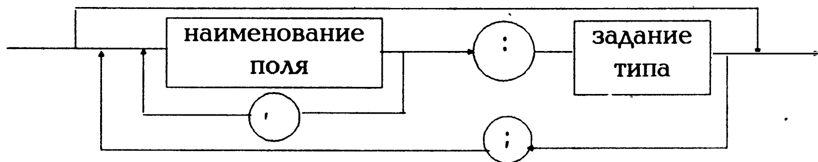
Итак, тип данных record характеризуется тем, что после служебного слова record перечисляются компоненты записи, называемые также полями. Описание типа заканчивается оператором end.

список полей : (15-2)



Переменную часть списка полей, которая значительно усложняет использование записи, мы пока опустим и ограничимся вначале записями с одной лишь постоянной частью.

постоянная часть : (15-3)



Итак, описание данных типа record требует задания наименования типа, указания имен компонент (полей) и их типов:

```
type наименование_типа = record
          имя_компоненты_1:тип_компоненты_1;
          имя_компоненты_2:тип_компоненты_2;
             ...       end;
```

На следующем рисунке показана конкретная запись и ее компоненты.

Переменная-целая

Имя переменной	ding			
Задание типа	record nr:integer; name:string[6]; qualitaet:(gut,mittel,schlecht); preis:real; end			
Значение	1245	'roller'	mittel	76.34

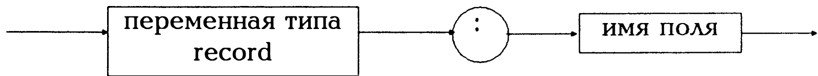
Компоненты записи (поля)

Переменная-составляющая

Имя переменной	ding.nr	ding.name	ding.qualitaet	ding.preis
Задание типа	integer	srting[6]	(gut,mittel, schlecht)	real
Значение	1245	'roller'	mittel	76.34

Рис. 15.1. Составляющая описания записи

компоненты записи : (15-4)



При описании переменной ding одновременно описаны и переменные ding.nr, ding.name, ding.qualitaet и ding.preis, являющиеся компонентами ding.

Итак, информация о персонале может быть записана следующим образом:

```
type datum  = record tag  :1..31;
            monat:1..12;
            jahr :0..99 end;
     person = record
            nummer    :integer;
            name      :string[20];
            adresse   :record
                      plz     :0..9999;
                      strasse :string[20];
                      stadt   :string[16];
                      end;
            stand     :(ld,vh,vw,gesch,getr);
            geburt    :datum;
            geschlecht:(m,w)      end;

var p    :person;
   leute:array[1..100] of person;
```

Переменная p выступает в качестве значения некоторой записи из шести компонент с именами nummer,name,adresse,stand,geburt и geschlecht, типы которых задаются. Если p является переменной-целой, шесть полей записи p будут согласно (14-4) переменными-составляющими. Функция sizeof (рис.5.6) даст следующий результат:

```
sizeof(person) :   60 байт
sizeof(leute)  : 6000 байт
```

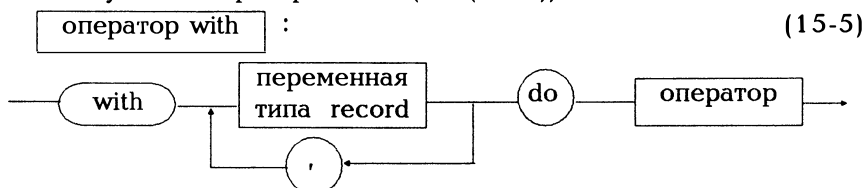
Итак, согласно приведенному выше описанию, в исполняемой части программы может стоять:

```
p.nummer := 2387;
p.name   := 'Нейман,Хайнц';
p.stand  := vw;
p.geburt.monat := 5;
```

```
if (p.geschlecht = m) and (p.geburt.jahr > 50)
        then ...
      { Это все люди, родившиеся после 1950 года}
leute[12].adresse.stadt:='Мюнхен';
{Это город, в котором родился 12-й человек из списка}
```

Приведенная выше формулировка затруднительна, поскольку к отдельным полям нужно обращаться через имя переменной типа "запись". Для того, чтобы иметь более короткую форму записи, воспользуемся оператором with (см. (10-4)).

оператор with : (15-5)



Смысл этого оператора заключается в том, чтобы задать имя переменной типа record только один раз с помощью with, после чего в стоящем после do операторе останется лишь указать имена полей:

```
                          with p,geburt do
                          begin
p.nummer:= 2198;             nummer := 2198;
p.name:='Майер';             name := 'Майер';
p.geburt.tag:=12;            tag := 12;
p.geburt.monat:=5;           monat := 5;
p.geburt.jahr :=25;          jahr := 25;
                          end;
```

Пример 15.1:

Задается и выводится массивов артикулов товаров, хранящихся на складе. Обратите особое внимание на то, как используются компоненты записи artikel и массив lager.

```
program artikel;
uses crt;
type artikeltyp = record
            nummer      :integer;    {номер}
            name        :string[20]; {наименование}
            stueckzahl :integer;     {кол-во штук}
            preis       :real;       {цена}
            lieferdatum:record tag:1..31;
            {дата поставки:день,месяц,год}
                              monat:1..12;
```

```
                              jahr:0..99
                        end;
              end;
     lagertyp   = array[1..100] of artikeltyp;
var lager :lagertyp;
    anzahl:integer;

procedure liesartikel (var a:artikeltyp);
begin
   with a,lieferdatum do
   begin
    write('номер:'); readln(nummer);
    write('наименование:'); readln(name);
    write('штук:'); readln(stueckzahl);
    write('цена:'); readln(preis);
    writeln('дата поставки:');
    write('день:'); readln(tag);
    write('месяц:'); readln(monat);
    write('год:'); readln(jahr);
   end;
end;    (* Считать данные по товарам *)

procedure lieslager(var l:lagertyp; anz:integer);
var i:integer;
begin
   for i := 1 to anz do liesartikel(l[i]);
   l[anz+1].nummer := -1;
   (* Тем самым можно узнать последнее
      поступление на склад *)
end;   (*  Состояние склада *)

procedure druckartikel(a:artikeltyp);
begin
    with a,lieferdatum do begin
        write(nummer:5,name:24,stueckzahl:10);
        write(preis:10:2,tag:12,'.',monat:2);
        writeln('.',jahr:2);  end;
end;   (* Распечатка данных о товаре *)

procedure drucklager(l:lagertyp);
var i:integer;
begin
   i := 1;
   while   l[i].nummer > 1 do
      begin
        druckartikel(l[i]);
```

```
      i := i + 1;
    end;

  end;  (* Распечатка данных по складу *)

  begin  (******* Исполняемая часть *******)
    write('Сколько товаров: n  <100   ');
    readln(anzahl);
    lieslager(lager,anzahl);
    drucklager(lager);
  end.
```

Дополните программу процедурой типа

```
minimum(l:lager;min:integer),
```

которая будет выдавать данные о товарах, количество которых на складе меньше установленного минимума min.

Приведенная выше программа хотя и решает задачу ввода и хранения информации о товарах, она вряд ли отвечает даже самым простым требованиям. Как уже говорилось, записями являются множества данных из файлов и ввод их требует слишком частого обращения к файлу. Такой ввод записи должен быть чрезвычайно надежным, простым и комфортабельным. Приведенная выше процедура liesartikel вряд ли удовлетворяет этим требованиям. К тому же нет проверки по длине цепочки символов и контроля за вводом числовых данных. Неверный ввод числа может даже привести к прерыванию! Пригодный для использования на практике ввод записей приводит нас к проблеме масок и требует при программировании больших затрат. Это демонстрирует следующий пример.

Пример 15.2:

В приведенной в примере 15.1 программе нужно модифицировать процедуру leisartikel таким образом, чтобы считывание записи было удобным.

```
program artikel;
uses crt, turbo3;
(*$V-*)
type artikeltyp = record nummer      :integer;
                         name        :string[20];
                         stueckzahl :integer;
                         preis       :real;
```

```
                    lieferdatum:record
                              tag:1..31;
                              monat:1..12;
                              jahr:0..99;
                         end;
               end;
     lagertyp   = array[1..100] of artikeltyp;
var lager :lagertyp;
    anzahl:integer;

procedure liesname (zeile:integer; var n:string;
                    laenge:word);
begin
  repeat
     gotoxy(15,zeile);
     readln(n);
  until length(n) <= laenge;
end;   (* Считать наименование *)

procedure druckartikel(a:artikeltyp);
begin
    with a,lieferdatum do begin
        write(nummer:5,name:24,stueckzahl:10);
        write(preis:10:2,tag:12,'.',monat:2,'.');
        writeln(jahr:2);  end;
end;   (* Распечатка данных о товаре *)

procedure drucklager(l:lagertyp);
var i:integer;
begin
   writeln;
   i := 1;
   while  l[i].nummer > 1 do
      begin
        druckartikel(l[i]);
        inc(i);
      end;
end;  (* Распечатка данных по складу *)

procedure liesinteger(var z:integer;
                      unten,oben,zeile:integer);
var ok:boolean;
begin
  gotoxy(15,zeile);
  repeat (*$i-*) readln(z); (*$i+*)
         ok :=ioresult = 0;
```

```
         if not ok then
                  begin write(^g);
                       gotoxy(15,zeile);
                       write('                 ');
                       gotoxy(15,zeile);
                  end;
         if (z < unten) or (z > oben)
                  then
                    begin
                       write(^g);
                       gotoxy(15,zeile);
                       write('                 ');
                       gotoxy(15,zeile);
                       ok := false;
                    end;
  until ok;
end;

procedure liesreal(var z:real; zeile:integer);
var ok:boolean;
begin
   gotoxy(15,zeile);
   repeat (*$i-*) read(z); (*$i+*)
          ok := ioresult = 0;
          if not ok
              then begin
                     write(^g);
                     gotoxy(15,zeile);
                     write('                   ');
                     gotoxy(15,zeile);
                  end;

   until ok;
end;

procedure liesartikel (var a:artikeltyp);
var x:integer;
begin
   clrscr;
   lowvideo;
   writeln('номер          :');
   writeln('наименование  :');
   writeln('штук           :');
   writeln('цена           :');
   writeln('дата поставки :');
   writeln('    день       :');
```

```
    writeln('    месяц     :');
    writeln('     год      :');
    highvideo;
  with a,lieferdatum do
  begin
    liesinteger(nummer,0,9999,1);
    gotoxy(15,2); liesname(2,name,20);
    liesinteger(stueckzahl,0,32767,3);
    liesreal(preis,4);
    liesinteger(x,1,31,6); tag := x;
    liesinteger(x,1,12,7); monat := x;
    liesinteger(x,0,99,8); jahr := x;
  end;
end;

procedure lieslager(var l:lagertyp; anz:integer);
var i:integer;
begin
   for i := 1 to anz do liesartikel(l[i]);
   l[anz+1].nummer := -1;
   (* Тем самым можно узнать последнее
      поступление на склад *)
end;   (*  Состояние склада *)

begin  (******* Исполняемая часть *******)
   write('Сколько товаров: n      ');
   readln(anzahl);
   lieslager(lager,anzahl);
   drucklager(lager);
end.
```

Для того, чтобы обойти контроль на совпадение типов для данных ограниченного типа, границы интервала передаются в процедуру liesinteger. Кроме того, процедуры liesinteger, liesreal и liesname получают номер строки, которая только что считывалась. Относительно ioresult см. рис. 8.8.

Пример 15.3:

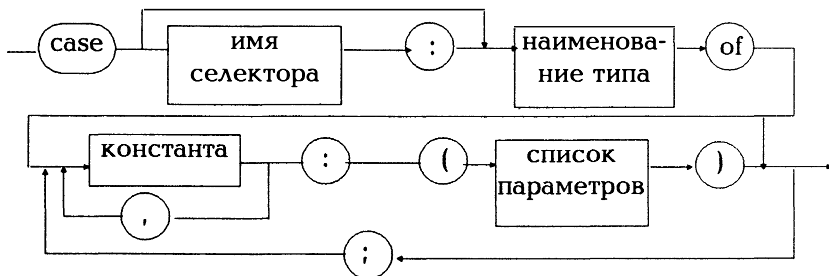
```
(*$R+*)
program zeitunterschied;
{программа определения разницы во времени}
type zeit = record stunde : 0..23;
                   minute, sekunde : 0..59 end;

var jetzt, vorher, unterschied : zeit;
```

```
procedure zeitdifferenz(alt, neu : zeit;
                        var diff : zeit);
var sek, sekalt, sekneu : longint;
begin
   sekalt := longint(3600)*alt.stunde +
             60*alt.minute + alt.sekunde;
   sekneu := lingint(3600)*neu.stunde +
             60*neu.minute + neu.sekunde;
   sek := sekalt - sekneu;
   diff.stunde := sek div 3600;
   sek := sek - 3600*diff.stunde;
   diff.minute := sek div 60;
   diff.sekunde := sek - diff.minute*60;
end;

procedure lies_zeit(var z : zeit);
begin
   writeln;
   write('Часы:       '); readln(z.stunde);
   write('Минуты:     '); readln(z.minute);
   write('Секунды:    '); readln(z.sekunde);
end;

begin
   lies_zeit(jetzt);
   lies_zeit(vorher);
   zeitdifferenz(vorher, jetzt, unterschied);
   with unterschied do begin
      write(stunde:3,'  ч',minute:3,'  мин');
      writeln(sekunde:3,'  с'); end;
end.
```

Иногда возникает ситуация, когда записи различаются только одним элементом. Может, например, оказаться, что все люди, информация о которых записана в виде записей, имеют одинаковые фамилии, адреса и пр. данные, в то время как в зависимости от поколения такой семьи требуются различные компоненты. В примере 15.1 могло оказаться, что по номеру артикула различается, изготовлен ли данные товар на месте или получен от дочерней или другой фирмы. Чтобы не создавать для таких товаров три различных записи, можно воспользоваться переменной (или как ее еще называют, вариантной) частью записи.

Переменная часть аналогична оператору выбора. С помощью селектора выбираются элементы, запрошенные в специальной части. Отдельные элементы помечаются как константы, которые се-

переменная часть : (15-6)



лектор может брать в качестве значений. Каждая переменная вновь состоит из списка полей, так что согласно (15-2) каждая переменная может состоять из одной постоянной и одной переменной части.

Пример 15.4:

В программе используется запись с одной лишь переменной частью. Программа вычисляет площади различных геометрических фигур с применением одной единственной функции.

```
program flaechenberechnung;
{программа вычисления площадей геометрических фигур}
type bild = (rechteck,dreieck,kreis);
     figur = record
               case art : bild of
                 rechteck:  (laenge,breite:real);
                 kreis :    (radius:real);
                 dreieck:   (seite1,seite2,winkel:real);
              end;
var x:figur;

function flaeche(f:figur):real;
   begin
      with f of
      case art of
       rechteck: flaeche := laenge*breite;
       kreis:    flaeche := pi*sqr(radius);
       dreieck:  flaeche := seite1*seite2*
                              sin(winkel)/2;
      end;
end  { Площадь }

begin
   writeln('Задайте длину и ширину прямоугольника:');
```

```
    with x do

        begin
            art := rechteck;
            readln(laenge,breite);
            write('ширина:',breite:8:2,'длина:');
            writeln(laenge:8:2);
            writeln('площадь:',flaeche(x):8:2);
        end;
end.
```

Чтобы в записи с переменной частью не столкнуться по ошибке с неверной компонентой, рекомендуется для надежности использовать, как и в примере 15.4., запись с соответствующим оператором case.

Переменная часть записи использует одно и то же место в памяти. Компилятор для переменной такого типа отводит столько места, сколько занимает наибольший из переменных элементов.

Пример 15.5:

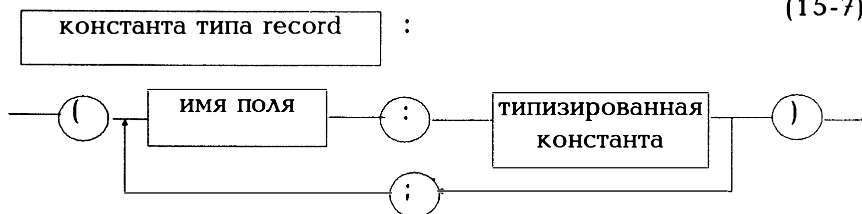
Для демонстрации внутреннего представления вещественных чисел используется переменная запись. Одна варианта имеет тип real, другая является массивом из шести байт. В вещественную переменную запись считывается число, а затем то же самое место в памяти интерпретируется как последовательность шести байтов, содержимое которых выдается побитно в шестнадцатеричном представлении.

```
program realbyte;
uses crt;
type fall = 1..2;
    rec = record
            case f:fall of
            1: (x:real);
            2: (s:array [1..6] of byte); end;
            (* Вещественное число x и массив s
               занимают одно и то же место в памяти *)
var i:integer;
    r:rec;
    f:fall;

procedure bitwrite(b:byte);
(* Содержимое байта b выдается побитно *)
var counter:integer;
begin
    write('  ');
    for counter:=7 downto 0 do
```

```
        if (b and (1 shl counter) <> 0)
            then write('1')
            else write('0');
end;
procedure hexwrite(b:byte);
(* Содержимое байта b выдается в шестнадцатеричном
   виде*)
var nr,links,rechts:integer;
begin
   write('       $');
   links:=b div 16;
   case links of
      0..9:   write(chr(links+48));
      10..15: write(chr(links+55)); end;
    rechts:= b mod 16;
    case rechts of
      0..9:   write(chr(rechts+48));
      10..15: write(chr(rechts+55)); end;
end;

begin   (********* Исполняемая часть *********)
  writeln('Введите вещественное число:');
  r.f := 1; readln(r.x);
  r.f := 2; for i := 1 to 6 do hexwrite(r.s[i]);
  writeln;
  for i := 1 to 6 do bitwrite(ord(r.s[i]));
  writeln;
  writeln;
end.
```

Приведенный пример позволяет понять, каково внутреннее представление вещественных чисел в ЭВМ: байт 1 = экспонента в 0=$80. Байты 2-6 = нормализованная мантисса. Байт 6 содержит старшие разряды, байт 2 - младшие. Знак числа записывается в левый бит байта 6. При этом следует помнить, что байты выводятся в порядке возрастания их адресов.

(15-7)



Согласно диаграмме (6-5), переменная типа record может содержать в описании константы задание начального значения.

Приведем несколько примеров:

```
type datum = record tag:1..31;
                    monat:1..12;
                    jahr:0..99;
             end;
const date : datum = (tag:2; monat:7; jahr:84);

type stadt = record plz:0..9999;
                    name:string[20];
                    einwohner:longint;
                    vorwahl:string[8]    end;
const bonn : stadt = (plz:5300; name:'Бонн';
                   einwohner:288148; vorwahl:'0228');
```

При описании констант элементам записи должны присваиваться значения в той последовательности, в которой они стоят в описании типа. Если элемент имеет тип "файл" или "указатель", значения ему можно не присваивать. Если запись содержит переменную часть, значения присваиваются только значащим элементам.

Вообще говоря, тип данных "запись" более всего подходит для того, что сделать программу, работающую с данными сложной структуры, "читабельной". Так например, для шахматной задачи можно было бы использовать следующие типы данных:

```
type spalte = 1..8;
     zeile  = 'A'..'H';
     figurtyp = (bauer, laeufer, turm, dame, koenig,
                springer, keine);
     {тип_фигур: пешка, слон, ладья, ферзь, король,
                конь, нет_фигуры}
     farbentyp = (schwarz, weiss, nichts);
     {Цвет фигур: черный, белый, никакой}
     feldtyp   = record s: spalte;
                     z: zeile;
                     figur: figurtyp;
                     farbe: farbentyp;
                     end;
var brett : array[zeile, spalte] of feldtyp;
```

Тогда пустое поле можно было бы описать следующим образом:

```
brett['E' ,5].figur := keine;
brett['E' ,5].farbe := nichts;
```

Итак, элемент farbe поля должен задавать цвет стоящей там фигуры (или не задавать никакого цвета, если на этом поле нет фигуры). Попробуйте воспользоваться функцией

```
function turm(f:feldtyp) : boolean;
```

Значением этой функции должно быть true, если стоящая на поле f ладья (цвета f.farbe) угрожает фигуре противника, и false в противном случае.

Каждый мог бы написать программу карточной игры скат и вывести соответствующие типы данных:

```
type werttyp  = ( sieben, acht, neun, zehn, bube,
                  dame, koenig, as);
     {семерка, восьмерка, девятка, десятка, валет,
                  дама, король, туз}
     farbentyp = ( karo, herz, pik, kreuz);
     {Масть: бубны, черви, пики, крести}
     kartentyp = record wert:werttyp;
                        farbe:farbentyp;
                 end;
       blattyp = array[1..10] of kartentyp;
var vorhand, mittelhand, hinterhand : blattyp;
```

Попытайтесь использовать функцию

```
function reizen(b:blattyp):integer;
```

Значением функции должна быть "игра цвета", если во взятке b присутствуют по крайней мере пять карт одной масти и двое бубей. Обратиться к функции можно следующим образом: reizen(mittelhand).

Чтобы избежать ошибок при использовании многих названий полей, рекомендуется все идентификаторы типа оканчивать буквами ...typ. Как показывают последние два примера, решение задачи в значительной степени зависит от того, насколько умело введены подлежащие обработке типы данных.

В одной записи могут быть (но не обязательно) элементы различных типов. Так можно в качестве записи ввести комплексные числа:

```
type komplex = record realteil, imaginaerteil:real
               end;
```

6*

```
var z:komplex;
z.realteil - действительная часть z;
z.imaginaerteil - мнимая часть z.
```

Поскольку обе компоненты записи komplex имеют одинаковый тип, их можно описать также в качестве массива:

```
type komplex = array[1..2] of real;
var z:komplex;
z имеет действительную часть z[1] и мнимую часть z[2].
```

Формулировка в качестве записи лучше. Здесь не нужно ничего запоминать. Процедура перемножения двух комплексных чисел для двух описаний типов будет выглядеть следующим образом:

```
procedure mult(x,y:komplex, var z:komplex);
begin
   z.realteil :=x.realteil*y.realteil
               - y.imaginaerteil*x.imaginaerteil;
   z.imaginaerteil :=x.imaginaerteil*y.realteil
               + y.imaginaerteil*x.realteil;
end;
```

или в качестве массива

```
procedure mult(x,y:komplex, var z:komplex);
begin
  z[1] :=x[1] * y[1] - x[2] * y[2];
  z[2] :=x[1] * y[2] + x[2] * y[1];
end;
```

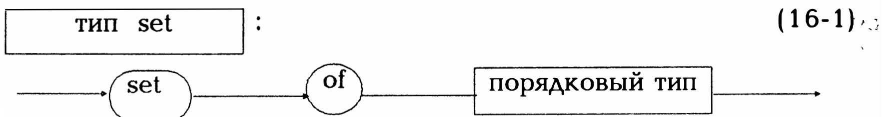
Вообще говоря, приведенная выше формулировка в виде функции

```
function mult(x,y:komplex):komplex;
```

некорректна, поскольку значение функции не является структурированным типом данных, а значит, в нашей ситуации могут быть использованы либо запись, либо массив.

16. Тип set

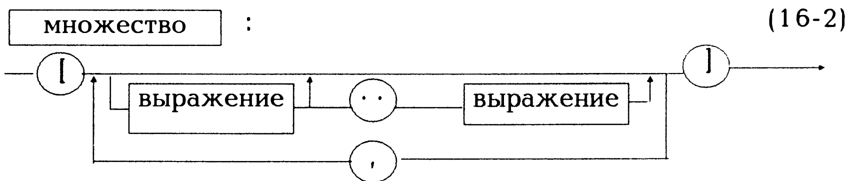
С помощью типа данных множества вводятся множества и операции над множествами.



Указанный порядковый тип, называемый также базисным множеством, описывает элементы, из которых могут быть образованы множества. В Турбо Паскале число элементов ограничено 256. Вот примеры такого типа данных:

```
type ziffer  = set of 0..9;
     farbe   = (rot,gelb,blau,lila,rosa,gruen);
     bunt    = set of farbe;
     letter  = set of 'a'..'z';
var z:ziffer; b:bunt; l:letter;
```

Если мы вводим новый тип данных, нужно еще указать, как должны описываться константы такого типа и какие операции позволяют образовывать выражения. Переменные z, b и l могут восприниматься как значения множеств из заданных базисных множеств. С математической точки зрения, переменные типа "множество" имеют в качестве значений элементы множества всех подножеств базисного множества. Для констант этого типа имеет силу следующая диаграмма:



Выражения должны иметь значения из заданного базисного множества порядкового типа. Итак, можно воспользоваться следующими операторами присваивания:

```
z := [3,5,7..9];
b := [rot,blau,gelb];
l := ['c','k'..'s','y'];
```

Константы этого типа содержат в квадратных скобках [] перечисление элементов. Обратите внимание на то, что элементы множества не упорядочены. Так [3,1,7], [7,3,1] и [3,7,1,3,7] являются одним и тем же множеством. [] пустое множество. Элементы множества могут задаваться и в виде выражений. Множество [2+3,2*4] означает множество [5,8].

В свою очередь, из множеств можно образовывать выражения, состоящие из констант, переменных и знаков операций над типом set.

Все описанные в главе правила исчисления выражений переносятся без каких-либо изменений на множества. Существуют приведенные ниже операции, для которых также имеют силу перечисленные на рис. 8.1 правила.

Уровень	Операция	Тип операндов	Результат действия оператора	Значение
1	*	Множество	Множество	Пересечение множеств
2	+ -	Множество Множество	Множество Множество	Объединение множеств Разность множеств
3	= <> >= <= in	Множество Множество Множество Множество слева:выражение справа: множество	boolean boolean boolean boolean	Проверка на равенство Проверка на неравенство Проверка на подмножество Проверка на подмножество Проверка на принадлежность

Рис. 16.1 . Операции над множествами

Пример 16.1:

В следующем примере демонстрируются операции над множествами:

```
program operationen_mit_mengen;
{программа выполнения операций над множествами}
type menge = set of 1..10;
var a,b,c:menge; i,anz:integer;
procedure druckmenge(x:menge);
var i:1..10;
begin
   for i := 1 to 10 do if i in x then write(i:3);
   writeln;
end;  (* Выводимое на печать множество *)
procedure liesmenge(var x:menge);
var k:integer;
begin
    x :=[];
    repeat
      read(k);
      if k in [1..10] then x := x + [k];
    until k = 0;
end;  (* Считываемое множество *)

begin    (********** Исполняемая часть **********)
   writeln ('Введите элементы множества.
```

```
            (Конец ввода по 0):');
    liesmenge(a);
    writeln ('Введите элементы множества.
            (Конец ввода по 0):');
    liesmenge(a);
      c := a*b;
      writeln('Пересечение множеств :');
      druckmenge(c);
      c := a + b;
      writeln('Объединение множеств :');
      druckmenge(c);
      c := a - b;
      writeln('Разность множеств:');
      druckmenge(c);
      if a <= b then writeln('a содержится в b');
              else writeln('a не содержится в b');
end.
```

При проверке на подмножество выполняется тест на "меньше или равно", а не только проверка на собственное подмножество, т.е. без "равно". Можно заменить такую проверку на

```
(x <= y) and (x <> y)
```

Тип данных "множество" особенно удобен, поскольку предоставляет возможность с помощью символа сравнения in проверять принадлежность некоторому множеству. Это особенно удобно для опросов, которые часто вырождаются в сложные операторы if.

Пример 16.2:

После считывания символа с нужно установить, является ли этот символ гласной.

```
if c in ['a','o','e','i','u'] then writeln ('да')
                                else writeln ('нет');
```

Можно записать это иначе:

```
  if (c='a') or (c='o') or (c='i') or (c='u')
     or (c='e')
  then writeln('да') else writeln('нет');
program vokale;
uses crt;
(* Считывается текст и гласные заменяются
   звездочками *)
```

```
  var c:char;
begin
   checkeof := true;
   while not eof do
     begin
       read(c);
       if upcase(c) in ['A','I','E','U','O']
          then write('*')
          else write(c);
     end;

end.
```

Пример 16.3:

При пересчете детей (см. примеры 13.2 и 14.7) выстроенные в ряд дети идентифицируются как множество. При отображении на экране исключенные из ряда дети помечаются окном половинной яркости.

```
program kinder;
{Программа  дети}
uses crt;
const kinderzahl = 8;
type namenstyp = (eva,otto,hans,inge,karl,ulla,
                  monika,gabi);
     kreistyp = set of namenstyp;
var kind       :namenstyp;
    silbenzahl:integer;
    kreis      :kreistyp;

procedure anfaenger_lesen(var k:namenstyp);
var c:char;
begin
  writeln('Кто будет ведущим?');
  writeln('Введите начальную букву имени:');
  write ('Eva Otto Hans Inge Karl Ulla');
  writeln('      Monika Gabi');
  readln(c);
  case c of
    'e' : k := eva;
    'o' : k := otto;
```

```
    'h' : k := hans;
    'i' : k := inge;
    'k' : k := karl;
    'u' : k := ulla;
    'm' : k := monika;
    'g' : k := gabi    end;
end;   (* Указать ведущего *)

procedure schreibe(k:namenstyp);
begin
    case k of
      eva:    writeln('Eva');
      otto:   writeln('Otto');
      hans:   writeln('Hans');
      inge:   writeln('Inge');
      karl:   writeln('Karl');
      ulla:   writeln('Ulla');
      otto:   writeln('Otto');
      monika: writeln('Monika');
      gabi:   writeln('Gabi')      end;
end;  (* Запись *)

procedure anfaenger_zeigen(k:namenstyp);
begin
   gotoxy(10,ord(k)+1); write('<— начинает');
   delay(3000);
   gotoxy(10,ord(k)+1); write('             ');
end;    (* Указать ведущего *)

procedure anzeigen(k:namenstyp);
{ Процедура указания выбывающего из игры }
begin
   gotoxy(10,ord(k)+1);
   write ('<— Выйди вон!');
   delay(1000);
   gotoxy(10,ord(k)+1);  write('             ');
   delay(1000);
   gotoxy(1,ord(k)+1);
   lowvideo; schreibe(k); highvideo; delay(1000);
end;    (* Указать *)

procedure anfangsstellung;
{Установка в исходное положение}
var k:namenstyp;
begin
   clrscr;
```

```
   highvideo;
   for k := eva to gabi do schreibe(k);
   gotoxy(50,1); write('Число слогов:',silbenzahl:3);
   anfaenger_zeigen(kind);
end;  (* Установка с исходное положение *)

procedure aufstellen;
begin
    kreis := [eva..gabi];
end;

procedure abzaehlen(var k:namenstyp);
{Процедура пересчета}
var i:integer;
procedure next(var n:namenstyp);
begin
   repeat
      if n < gabi then n:=succ(n) else n:=eva;
   until n in kreis;
end;   (* Следующий *)

begin   (**** Счет ****)
   for i := 1 to silbenzahl do
     begin next(k);
           gotoxy(10,ord(k)+1); write('<—(',i,')');
           delay(500);
           gotoxy(10,ord(k)+1); write('        ');
     end;
end;  (* Рассчитаться *)

procedure herausnehmen(k:namenstyp);
begin
    kreis := kreis - [k];
end;        (* Исключить из игры *)

begin    (******* Исполняемая часть *******)
   clrscr;
   writeln('Сколько слогов в считалочке?');
   readln(silbenzahl);
   aufstellen;
   anfaenger_lesen(kind);
   anfangsstellung;
   while kreis <> [] do
      begin
        abzaehlen(kind);
        anzeigen(kind);
```

```
        herausnehmen(kind);
      end;
   gotoxy(1,ord(kind)+1);
   highvideo;
   schreibe(kind);
end.
```

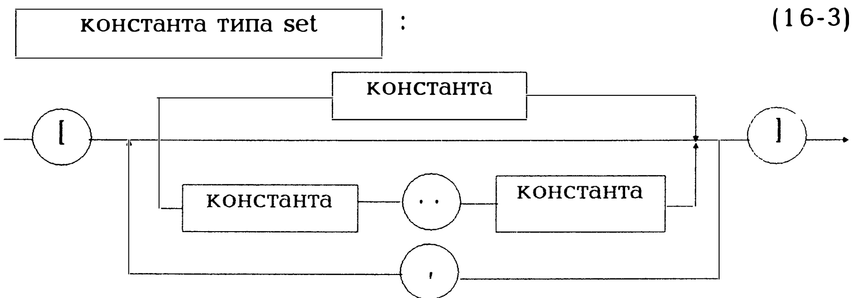
Пример 16.4:

В примере 13.4 генерируются шесть чисел, выпадающих при игре в лото, причем не гарантируется, что все шесть случайных чисел различны. Следующая программа позволяет сгенерировать шесть различных чисел.

```
program lottospiel;
{программа игры в лото}
uses crt;
(* Карточка лото заполняется с помощью random(49)+1 *)
const n = 6;
type lottozettel = set of 1..49;
var lotto : lottozettel;
    k,i : 1..n; zahl : 1..49; ok : boolean;

procedure ausgabe(l:lottozettel);
var i:1..49; k:1..6;
begin
   clrscr;
   lowvideo;
   for i := 1 to 49 do
     begin
        if i in l then begin
                        highvideo;
                        write(i:4);
                        lowvideo;
                        end;
                  else write(i:4);
         if i mod 7 = 0 then writeln;
     end;
end;

begin      (******* Исполняемая часть *******)
    lotto :=[];
    for i := 1 to n do
    begin
      repeat
         zahl := random(49)+1;
```

```
            if zahl in lotto
            then ok := false
            else begin lotto :=lotto + [zahl];
                       ok :=true;
                 end;
         until ok;
    end;
       ausgabe(lotto);
       gotoxy(1,23);
    end.
```

Согласно диаграмме (6-5), переменные типа "множество" при описании констант задаются начальными значениями.



Пример:
const vokal : set of 'a'..'z' = ['a','e','i','o','u'];

17. Файлы

Каждая программа общается со своим окружением через операторы read и write, будь это внешнее устройство, множество данных на дискете или файл на диске. Итак, с помощью read и write можно указать, откуда считываются данные и куда данные должны записываться. Для этого вводится понятие файла (File).

Файл является некоторой структурой данных с однородными компонентами, число которых при описании файла жестко не фиксируется. Файл может оставаться открытым и число его компонент может соответствующим образом изменяться. Компоненты файла называются также записями (record).

Файл целесообразнее всего просматривать сверху вниз. Имеется так называемый указатель записи в файле, который указывает на обрабатываемую в настоящий момент запись. При каждом обращении к файлу (для чтения или записи) указатель записи автоматически перемещается. Это называется также последовательным доступом (а указатель записи называется иногда индексом файла). Конечно, указатель можно установить на любую запись и начать последовательную обработку с нее. Но поскольку на позиционирование требуется довольно много времени, нужно обращаться к этой операции лишь при крайней необходимости.

Существуют три типа файлов:

- Типизированные файлы, записи которых имеют жесткую структуру, чаще всего типа record. Такие файлы рассматриваются в разделе 17.1.

- Текстовые файлы, состоящие из символов и строковых структур. Эти файлы рассматриваются в разделе 17.2.

- Нетипизированные файлы, при работе с которыми исходят лишь из того ограничения, что в них могут передаваться блоки по 128 байтов каждый. Такие файлы рассматриваются в разделе 17.3.

Для всех трех типов файлов существуют общие правила, более подробно описанные в следующих разделах:

1. В Паскале файл является некоторой переменной, как и любая другая переменная, а значит, ему можно присвоить имя, например, filename. С другой стороны, существует операционная система, которая в свою очередь использует имена файлов, например, A:MM.DAT или B:TEST.PAS или LST: для печатающего устройства. Прежде всего нужно установить связь между именами файлов в Паскале и именами файлов, присваиваемых операционной системой. Для этого имеется стандартная процедура assign, которая с помощью директивы

```
assign(filename,dos_name)
```

устанавливает соответствие между файловой переменной filename и системным файлом dos_name. Такое соответствие означает, что все операции, выполняемые в программе над переменной filename, будут выполняться над файлом dos_name. dos_name является некоторой строкой и может иметь путь доступа.

2. Файл filename следует позиционировать в начало, причем следует различать, существовал ли файл ранее или нет:

reset(filename) для уже существующего файла;
rewrite(filename) для вновь создаваемого файла.

3. Затем можно с помощью стандартной процедуры

read(filename,...); считать данные из filename или
write(filename,...); записать данные в filename.

4. С помощью стандартной функции

eof(filename);

типа boolean можно установить при чтении, достигнут ли конец файла filename(eof=конец файла) или нет.

5. С помощью стандартной процедуры

close(filename);

запись в файл filename завершается, файл после добавления метки конца файла eof закрывается и может быть в обычном порядке включен в каталог дискеты.

6. С помощью стандартной функции

ioresult

типа integer можно запросить, нормально ли прошло обращение к файлу (см. рис. 8.8).

7. Для стирания файла используется

erase(filename);

для переименования

rename(filename, dos_name);

8. Файлы в параметрах процедур могут передаваться только по ссылке (var filename). Переменные типа file не могут входить в состав выражения (а также сравниваться с помощью = или <>) и не могут стоять в левой части оператора присваивания.

Вызов: assign(f,dos_name) Процедура
Параметры: var f:filename; dos_name:string;
Действие: Ставит в соответствие файловую переменную f и системный файл dos_name, т.е. все операции, выполняемые над f, будут выполняться над dos_name. Имя dos_name может включать в себя и указание пути доступа к файлу.

Вызов: reset(f) Процедура
Параметры: var f:file_typ;
Действие: Открывает уже существующий файл f и позиционирует указатель записи на начало файла. Если файл f не существует, выдается сообщение об ошибке.

Вызов: rewrite(f) Процедура
Параметры: var f:file_typ;
Действие: Создается новый пустой файл f и указатель записи устанавливается на начало файла. Если файл f уже существует, он уничтожается.

Вызов: close(f) Процедура
Параметры: var f:file_typ;
Действие: Закрывает открытый с помощью reset, rewrite или append файл.

Вызов: eof(f) Значение функции:boolean
Параметры: var f:file_typ;
Действие: Значение функции true, если индекс файла f указывает позицию после последней компоненты f, в противном случае false.

Вызов: erase(f) Процедура
Параметры: var f:file_typ;
Действие: Стирает файл f. Перед этим файл нужно закрыть с помощью процедуры close.

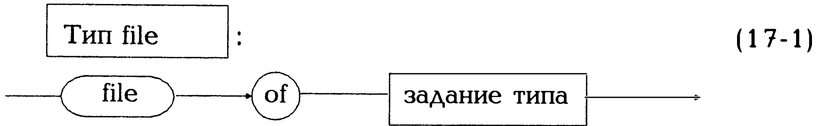
Вызов: rename(f,s) Процедура
Параметры: var f:file_typ; s:string;
Действие: Файл f переименовывается в s.В каталог вносится соответствующее изменение. Перед этой процедурой файл f должен быть закрыт с помощью close.

Примечание: файл filename может быть типизированным (17-1), текстовым (text, см. раздел 17.2) или не имеющим типа файлом (раздел 17.3). Все обращения могут проверяться с помощью ioresult.

Рис.17.1. Стандартные процедуры для файлов всех типов

17.1. Типизированные файлы

Общее описание данных типа file задается следующей диаграммой:



С помощью задания типа описывается тип множества данных, который не может быть в свою очередь типом file или типом структурированных данных, если хотя бы одна компонента множества данных имеет файловый тип. Практически, описание могло бы выглядеть следующим образом:

```
type artikel_typ = record key        :integer;
                        (* Номер артикула *)
                        name      :string[16];
                        stueck   :integer;
                        preis     :real;
                        lieferant:string[16];
                        datum     :record
                                   tag  :1..31;
                                   monat:1..12;
                                   jahr :0..99;
                                 end;
                   end;
     file_typ = file of artikel_typ;
var  f:file_typ; a:artikel_typ;
```

Каждое множество данных файла f состоит из записи a типа artikel_typ. Компоненты записи являюся полями данных. Для того, чтобы однозначно идентифицировать множество данных, записи в файле должны различаться по какому-то признаку. Назовем такое поле данных ключевым (или ключом). В приведенном выше примере ключевым является номер артикула.

Теперь займемся созданием файла, изменением отдельных записей в файле, удалением записей в файле и добавлением записей в файл. Чтобы процедуры для таких операций над файлами не зависели от вида записи, воспользуемся satz_typ:

```
type satz_type = (* Какой-либо тип записи данных,
                    например, artikel_typ *)
```

```
    file_typ = file of satz_typ;
var f:file_typ; s:satz_typ;
```

Для того, чтобы продемонстрировать основной механизм чтения и записи для файлов, в следующем примере создадим файл с информацией о городах. При этом наряду с представленными на рис. 17.1 процедурами будем использовать процедуры чтения/записи.

Вызов:	read(f,s1,...,sn)	Процедура
Параметры:	var f:file_typ; var s1,...,sn:satz_typ	
Действие:	Считывает записи s1,...,sn из файла f. Указатель устанавливается каждый раз на следующую запись.	
Вызов:	write(f,s1,...,sn)	Процедура
Параметры:	var f:file_typ; var s1,...,sn:satz_typ	
Действие:	В файл f записываются записи s1,...,sn. Указатель устанавливается каждый раз на следующую запись.	

Рис. 17.2. Операции чтения и записи для типизированных файлов

Пример 17.1:

Следующая программа позволяет записывать в файл данные о городах и выводить их.

```
program datei_anlegen_und_ausgeben;
{Программа создания файла и вывода из него данных}
uses crt;

type stadt = record key        : 0..9999;
                   (* Почтовый индекс *)
                   name      : string[20];
                   einwohnwer: longint;
                   kfz       : string[3];
             end;
    satz_typ = stadt;
    file_typ = file of satz_typ;

var f : file_typ; s : satz_typ;
    name :string[30];
```

```
procedure satz_lesen(var s : satz_typ);
begin
   clrscr;
   with s do
   begin
      writeln('Конец ввода plz = 0');
      write ('Почтовый индекс:      '); readln(key);
      if key <> 0 then begin
      write('Название:         '); readln(name);
      write('Жителей:          '); readln(einwohner);
      write('Транспортных средств:       '); readln(kfz);
                     end;
   end;
end;

procedure satz_ausgabe(s:satz_typ);
begin
   with s do
   writenl(key:4, name:22, einwohnwer:9, kfz:5);
end;

procedure datei_anlegen(var f : file_typ);
{процедура создания файла}
var s : satz_typ;
begin
    rewrite(f);
    satz_lesen(s);
    while s.key <> 0 do
       begin
          write(f,s);
          satz_lesen(s);
       end;
    close(f);
end;

procedure datei_ausgeben(var f:file_typ);
var s:satz_typ;
begin
   reset(f);
   while not eof(f) do begin
                      read(f,s);
                      satz_ausgaben(s);
                    end;
end;
```

```
begin      (******** Исполняемая часть ********)
   write('Как назвать файл: ');
   readln(name);
   assign(f,name);
   datei_anlegen(f);
   datei_ausgeben(f);
end.
```

В последнем примере обратите внимание на две важные операции создания файла и вывода из него данных.

Создадим файл f:

```
rewrite(f);
(*  Позиционировать на начало файла, записать *)
while существуют еще записи do
   begin
      write(f,s);
      (* Записать в f множество данных s *)
      satz_lesen(s);
      (* Считать следующую запись *)
   end;
close(f);
```

Выведем данные из файла f:

```
reset(f);
(* Позиционировать на начало файла, считать *)
while not eof(f) do
    begin
       read(f,s);
       (* Считать следующую запись из f *)
       satz_ausgeben(s);
       (* Вывести запись на экран *)
    end;
```

Обратите также внимание на то, что процедуры datei_anlegen и datei_ausgeben не зависят от того, что при обработке отдельной записи речь идет, собственно говоря, о типе stadt. Для другого файла пришлось бы лишь заново написать процедуры satz_einlesen и satz_ausgeben.

Файл f является типизированным файлом и, конечно, не может просматриваться с помощью команды type операционной системы или редактора, поскольку числовые значения записываются в файле в двоичном виде.

Рассмотрим теперь для типизированного файла три важные операции:

- добавление записи,
- изменение записи,
- удаление записи.

При этом целесообразно обратиться к представленным на рис. 17.3 процедурам.

Вызов:	seek(f,snr)	Процедура
Параметры:	var f:file_typ; snr:longint;	
Действие:	Указатель записи позиционируется на запись с номером snr. Первая запись имеет номер 0.	
Вызов:	filepos(f)	Значение функции: longint
Параметры:	var f:file_typ;	
Действие:	Выдается номер записи, на которую указывает указатель записи. Первая запись имеет номер 0.	
Вызов:	filesize(f)	Значение функции: longint
Параметры:	var f:file_typ;	
Действие:	Число записей в файле f. Пустой файл имеет 0 записей. Это всегда имеет место после процедуры rewrite(f).	
Вызов:	truncate(f)	Процедура
Параметры:	var f:file_typ;	
Действие:	Файл f усекается по текущему положению указателя. В этой позиции записывается метка конца файла eof. (Для всех типов файлов).	

Рис. 17.3. Процедуры для типизированных файлов

Добавить запись s в файл f можно таким образом:

```
reset(f);
{Добавлять записи можно только в конец файла.
 Позиционирование в конец файла:}
  seek(f,filesize(f));
  satz_lesen(s);
  write(f,s);
  close(f);
```

Изменить запись s с ключом k в файле f:

```
reset(f);
while not eof(f) and (s.key <> k) do read(f,s);
if s.key <> k
```

```
        then writeln('Запись не найдена')
        else begin
              (* Изменить запись s с ключом k *)
              {Указатель записи уже установлен на следующую
                запись, то есть нужно вернуться на одну
                запись назад;}
              seek(f,filepos(f)-1);
            end;
        close(f);
```

Стереть в файле f запись с ключом k=key:

```
   (* Файл f без записи с ключом key=k переносится во вре-
      менный файл TEMP.COP.  Затем f стирается, а файл
      TEMP.COP переименовывается в f. *)
      found := false;
      assign(kopie,'TEMP.COP'); rewrite(kopie);
      reset(f);
      while not eof(f) do begin
         read(f,s);
         if s.key <> k then write(kopie,s)
                       else begin
                              found := true;
                              (* s можно высветить на экране
                             и стереть после подтверждения *)
                            end;
                         end;
     close(f); close(kopie);
     if not found
        then begin
              writeln('Запись не существует');
              erase(kopie);
            end
        else begin
              erase(f);
              rename(kopie,name);     (* name - имя
                                   файла f в  DOS *)
            end;
```

Следующий пример показывает действие названных процедур на конкретном случае файла с данными о городах.

Пример 17.2:

```
program datei_anlegen_und_ausgeben;
{Программа создания файла и вывода из него данных}
uses crt;
```

```
type stadt = record key        : 0..9999;
                  (* Почтовый индекс *)
                  name      : string[20];
                  einwohner : longint;
                  kfz       : string[3];
             end;
     satz_typ = stadt;
     file_typ = file of satz_typ;

var kopie, f : file_typ;
   name : string[20];
   quit : boolean;

procedure satz_lesen(var s : satz_typ);
begin
   clrscr;
   writeln('Конец ввода: почтовый индекс = 0');
   write ('Почтовый индекс:     '); readln(s.key);
   if s.key  <> 0 then begin
   write('Название:        '); readln(s.name);
   write('Жителей:         '); readln(s.einwohner);
   write('Транспортных средств:       '); readln(s.kfz);
                  end;
end;

procedure satz_ausgeben(s:satz_typ);
begin
   writenl(s.key:4, s.name:22, s.einwohner:9, s.kfz:5);
end;

procedure datei_anlegen(var f : file_typ);
{процедура создания файла}
var s : satz_typ;
begin
    rewrite(f);
    satz_lesen(s);
    while s.key  <> 0 do
       begin
          write(f,s);
          satz_lesen(s);
       end;
    close(f);
end;
```

```
procedure datei_ausgeben(var f:file_typ);
var s:satz_typ;
begin
   reset(f);
   while not eof(f) do begin
                       read(f,s);
                       satz_ausgeben(s);
                     end;
end;       (* Вывести файл *)

procedure vorbereiten;
var c:char;
begin
    assign(kopie, 'temp.cop');
    write('Имя файла ? ');
    readln(name);
    assign(f,name);
    writeln('Файл    ',name,'   уже существует? Д/Н');
    readln(c);
    case c of
    'Н','н': begin rewrite(f); close(f); end;
    else end;
end;      (* Подготовительные операции *)

procedure delete(var f:file_typ);
var s:satz_typ; k:integer; c:char; found:boolean;
begin
   writeln('Какую запись стереть?');
writeln('Задайте ключевое поле:  ');
readln(k);
reset(f); rewrite(kopi);
found :=false;
while not eof(f) do
  begin
     read(f,s);
     if s.key <> k then write(kopie,s)
                 else
                 begin
                    found :=true;
                    sazt_ausgaben(s);
                    write('Именно эту запись');
                    writeln('уничтожить? Д/Н');
                    readln(c);
                    case c of
                      'Д','д': ;
                      else write(kopie,s) end;
                    end;
```

```
  end;
close(f); close(kopie);
if not found
   then begin
          write('Запись с ключом  ',k:6);
          write('   не найдена.');
          erase(kopie);
          writeln('<ret>'); read(c);
        end
   else begin
          erase(f);
          rename(kopie,name);
        end;
end;  (* Удаление записей *)

procedure append(var f:file_typ);
(* Для текстового файла существует стандартная
   процедура append *)
var s:satz_typ; c:char;
begin
   reset(f);
   seek(f,filesize(f));
   sazt_lesen(s);
   writeln; satz_ausgaben(s);
   writeln('Все правильно? Д/Н'); readln(c);
   case c of
     'Д','д':   write(f,s);
   end;
   close(f);
end;   (* Добавить *)

procedure change(var f:file_typ);
var s:satz_typ; k:integer; c:char;
begin
   writeln('Какую запись нужно изменить?');
   writeln('Задайте ключевое поле:  ');
   readln(k);
   reset(f);
   while not eof(f) and (s.key <> k) do read(f,s);
   if s.key = k then begin
                       satz_ausgeben(s);
                       write('Новое число жителей:  ');
                       readln(s.einwohner);
                       writeln('Все правильно? Д/Н');
                       satz_ausgeben(s);
                       readln(c);
```

```
                        case c of
                          'Д','д': begin
                                      seek(f,filepos(f)-1);
                                      write(f,s);
                                   end;
                        end;
                     end
                else begin
                        write('Запись с ключом',k:6);
                        writeln(' не найдена');
                     end;
                close(f);
              end;  (* Изменение *)

proceudre menue;
var c:char;
begin
   clrscr;
   quit := false;
   writeln('Что будете делать?');
   highvideo; write('d'); lowvideo; writeln('elete');
                                    ("удалять")
   highvideo; write('a'); lowvideo; writeln('ppend');
                                    ("добавлять")
   highvideo; write('c'); lowvideo; writeln('hange');
                                    ("изменять")
   highvideo; write('p'); lowvideo; writeln('rint');
                                    ("печатать")
   highvideo; write('q'); lowvideo; writeln('uit');
                                    ("выход")
   highvideo;
   read(c);
   clrscr;
   case c of
      'a', 'a': append(f);
      'd', 'd': delete(f);
      'c', 'c': change(f);
      'p', 'p': begin datei_ausgeben(f);
                   writeln('<ret>'); readln(c);
                end;
      'q', 'q': quit := true;
   end;  (* case *)
end;
```

```
begin  (******** Исполняемая часть *********)
   vorbereiten;
   repeat
      menue;
   until quit;
end.
```

Приведенный выше пример демонстрирует основной принцип работы с типизированными данными. Запись данных отыскивается последовательно, прямым перебором. Естественно, на практике с большими массивами данных так работать нельзя. С помощью типа данных "указатель" (глава 18) можно организовать чрезвычайно элегантный доступ к записям файла, когда соответствие "ключевое поле - номер записи" отображается в неком дереве индексов и свободные записи данных приводятся к некоторому связному списку. Для проблемно-ориентированной обработки файлов следует использовать соответствующее готовое программное обеспечение (например, Toolbox Database фирмы Borland).

Приведем еще один пример записи в файл после операции reset, поскольку в стандартном Паскале после reset можно выполнять лишь операцию считывания, а после rewrite - только операцию записи.

Пример 17.3. Запись после reset

```
program reset_rewrite_test;
uses crt;
var f : file of integer;
    i : integer;

begin
   clrscr;
   assign(f,'test.t');
   rewrite(f);
   for i := 1 to 10 do write(f,i);
   close(f);

   reset(f); writeln('1 1 1 1 1 1');
   while not eof(f) do
   begin read(f,i); write(i:4); end; writeln;

   reset(f);
   for i := 1 to 3 do read(f,i);
   i := 77;
   write(f,i);
   (*  close(f);  никакого значения не имеет *)
   (* truncate(f); Конец файла без close после записи 77.*)
```

```
    reset(f);   writeln('22222');
    while not eof(f) do
    begin read(f,i); write(i:4); end; writeln;
end.
```

17.2. Текстовые файлы

Наряду с описанными в разделе 17.1 типизированными файлами, существует также понятие текстового файла. Такой файл не задается как file of ..., для него существует стандартное наименование типа text, например:

```
var f:text;
```

Обратите внимание, что описанный как file of char файл является типизированным файлом, но не файлом типа text. Тип text имеют оба стандартных файла input и output, через которые обычно осуществляется ввод и вывод.

Текстовые файлы отличаются от типизированных двумя признаками:

- Их элементами являются символы и они имеют строковую структуру.
- При чтении и записи числа преобразуются автоматически.

В главе 8 были описаны элементарные операции ввода и вывода. Рассмотрим теперь эти операции более подробно и с привязкой к изложенному ранее. Наряду с общими процедурами и функциями, представленными на рис. 17.1, для текстовых файлов существуют и специальные процедуры и функции (см. рис. 17.4).

Вызов:	readln(f)	Процедура
Параметры:	var f:text	
Действие:	Указатель записи файла f устанавливается на начало новой строки, то есть пропускаются все символы, вплоть до следующих знаков CR/LF.	
Вызов:	writeln(f)	Процедура
Параметры:	var f:text	
Действие:	В файл f в качестве метки конца строки записывается последовательность символов CR/LF.	
Вызов:	eoln(f)	Значение функции: boolean
Параметры:	var f:text	
Действие:	Функция типа boolean принимает значение true, если указатель записи стоит в конце строки (точнее, указывает на CR), и false в противном случае. Если eof(f) имеет значение true, значение true имеет и eoln(f).	

Вызов:	seekeoln(f)	Значение функции: boolean
Параметры:	var f:text	
Действие:	То же, что и eoln. Но пробелы и табуляция обходятся.	
Вызов:	seekeof(f)	Значение функции: boolean
Параметры:	var f:text	
Действие:	То же, что и eof, но знаки пробела, табуляции и перевода строки игнорируются.	
Вызов:	flush(f)	Процедура
Параметры:	var f:text	
Действие:	В f записывается содержимое буфера.	
Вызов:	settextbuf(f,buf,size)	Процедура
Параметры:	var f:text; var buf:какой-либо тип; size:word;	
Действие:	Для файла f создается буфер в buf. Параметр size может отсутствовать; в этом случае он берется как sizeof(buf).	
Вызов:	append(f)	Процедура
Параметры:	var f:text	
Действие:	Открывает файл f и устанавливает указатель записи на eof. Затем можно добавлять символы.	

Рис. 17.4. Процедуры и функции для текстовых файлов

Относительно операций чтения и записи для текстовых файлов действуют следующие общие правила:

```
read(f,v1,...,vn);          = read(f,v1);...; read(f,vn)
readln(f,v1,...,vn);        = read(f,v1,...,vn); readln(f)
write(f,e1,...,en);         = write(f,e1);...; write(f,en)
writeln(f,e1,...,en);       = write(f,e1,...,en);
                              writeln(f);
```

Если в качестве f используется стандартный текстовый файл input, его имя можно опускать:

```
read(input,v1,...,vn);      = read(v1,...,vn);
readln(input);              = readln;
eof(input);                 = eof;
eoln(input);                = eoln;
```

Если с качестве f используется стандартный текстовый файл output, его имя можно опускать:

```
write(output,e1,...,en);    = write(e1,...,en);
writeln(output);            = writeln;
```

Вызов: read(f,v); Процедура

Рис. 17.5. read/write для текстовых файлов

Текстовый файл делится на строки с помощью меток конца строки (end of line mark), генерируемых при нажатии клавиши <RETURN> и представленных внутри файла символами ASCII-кода carriage return/line feed (CR/LF). С помощью команды readln(f) метка строки считывается, с помощью writeln(f) записывается в файл f. В следующем примере один текстовый файл копируется в другой и показывается, что происходит при чтении "метки конца строки" с помощью read(f,c) (var c:char).

Пример 17.4:

Текстовый файл quelle копируется в текстовый файл ziel построчно и посимвольно. При этом считанный из quelle символ протоколируется путем выдачи его номера по таблице кодов ASCII.

```
program text_file_copieren;
var quelle,ziel:text;
    qname,zname:string[20];

procedure copy(var q,z:text);
var c:char;
begin
   while not eof(q) do
    begin
       read(q,c);
       writeln(ord(c):4);
       (* Номер считанного символа в таблице ASCII
          → экран *)
       write(z,c);
    end;
   close(q); close(z);
end;

begin
  write('Исходный файл:      '); readln(qname);
  write('Выходной файл:      '); readln(zname);
  assign(quelle,qname); assign(ziel,zname);
  reset(quelle); rewrite(ziel);
  copy(quelle,ziel);
end.
```

При выполнении этой программы оказалось, что при чтении "метки конца строки" выдаются символы 13 (CR) и 10 (LF) кодов ASCII. Тогда при чтении с помощью read(q,c) считываются именно эти символы.

Если нужно распознать конец строки, следует воспользоваться функцией eoln. В следующем примере текстовый файл quelle копируется в файл ziel таким образом, что содержимое quelle сдвигается на n символов вправо.

Пример 17.5:

Текстовый файл quelle копируется построчно и посимвольно в файл ziel. При этом текст в копии ziel сдвигается на n символов вправо.

```
program text_file_copieren;
var quelle,ziel:text;
    qname,zname:string[20];
    n:integer;

procedure copyn(var q,z:text; n:integer);
var c:char;
begin
  while not eof(q) do
  begin      (* Передать 1 строку *)
     write(z,' ':n);
     while not eoln(q) do
       begin read(q,c); write(z,c); end;
       readln(q);  (* Конец строки считать из q *)
       writeln(z); (* Конец строки записать в q *)
  end;

 close(q); close(z);
end;

begin
   write('Исходный файл:   '); readln(qname);
   write('Выходной файл:   '); readln(zname);
   write('На сколько символов ');
   write('вправо сдвинуть?');
   readln(n);
   assign(quelle,qname); assign(ziel,zname);
   reset(quelle); rewrite(ziel);
   copyn(quelle,ziel,n);
end.
```

Вторая особенность текстовых файлов заключается в том, что при выполнении операций чтения или записи числа преобразуются автоматически и при этом учитывается задание формата.

Вызов:	read(f,v) Процедура
Параметры:	var f:text; var v:char или integer или real или string
Действие:	
v:char	Считывается только один символ и присваивается переменной v. Если считываемый файл находится на дискете, значение функции eoln(f) будет true, если следующий символ cr или Ctrl-Z, а функция eof(f) будет иметь значение true, если далее следует Ctrl-z или физический конец файла.
v:string[n]	Считывается максимум n символов и записывается в v, если ранее функции eoln(f) или eof(f) не имели значения true. eoln(f) имеет значение true при считывании cr или если значение true имела функция eof(f). Функция eof(f) имеет значение true, если считаны символы Ctrl-Z или достигнут физический конец файла.
v:integer или v:real	Ожидается последовательность символов, образующих численную константу (см. гл. 5). Знаки пробела, табуляции, перевода строки cr или возврата каретки lf перед значащими символами игнорируются. В последовательности не может быть более 30 символов, и численная константа не может завершаться пробелом, знаком табуляции, cr или Ctrl-Z. Численная константа преобразуется и записывается в v. Если цепочка символов не отвечает ожидаемому формату, выдается сообщение об ошибке.

Рис. 17.6. Процедура read для текстовых файлов

Вызов:	write(f,e); Процедура
Параметры:	var f:text; е является выражением с указанием формата (см. ниже).
Действие:	m и n обозначают выражения типа integer.
char	e:n (е имеет тип char). После n-1 пробела записывается символ е. Если n отсутствует, берется n=1.
string	e:n (е имеет тип string[k] или array[1..k] of char) Строка е выравнивается по правому краю и записывается в поле из n символов после n-length(e) пробелов. Если n отсутствует, берется n = length(e).
boolean	e:n (е имеет тип boolean). Если имеет место true или false, в поле записывается n символов, выравненных по правому краю. Если n отсутствует, записывается true или false.

integer	e:n (e имеет тип integer) Значение e преобразуется из десятичной системы и записывается в поле из n символов с выравниванием по правому краю. Если n отсутствует, записывается значение e.
real	e:n:m (e имеет тип real) Значение e преобразуется из десятичной системы и записывается в поле из n символов с m разрядами после десятичной точки. m должно принадлежать интервалу [0,24]. Если m отсутствует, значение e после выравнивания по правому краю записывается в поле из n символов в формате x.xxxxxxxxxxE+-dd (e>=0) -x.xxxxxxxxxxE+-dd (e<0) Если m и n отсутствуют, значение e, выравненное по правому краю, записывается с 18 знаками в указанном выше формате.

Рис. 17.7. Процедура write для текстовых файлов

Внимание!	Если m или n выбраны меньшими, чем это требуется для представления e, записывается соответствующее число символов, большее, чем n или m.

Пример 17.6:

Следующая программа записывает числа 1..n как в типизированный, так и в текстовый файл.

```
program filetyp;
var i,n:integer;
    zahl_text:text;
    zahl_typ : file of integer;
begin
   writeln('n?'); readln(n);
   assign(zahl_text,'ZAHL.TXT');
   assign(zahl_typ,'ZAHL.TYP');
   rewrite(zahl_text);
   rewrite(zahl_typ);
   for i := 1 to n do begin
                      writeln(i:i);
                      (* Вывод на экран *)
                      writeln(zahl_text,i:i);
                      writeln(zahl_typ,i);
                    end;
   close(zahl_text);
   close(zahl_typ);
end.
```

Файл ZAHL.TXT можно просмотреть с помощью редактора. Его содержимое в точности совпадает с выводимыми на экран данными. В файле ZAHL.TYP числа записываются в двоичном виде, а потому прочитать его содержимое с помощью редактора затруднительно.

Печатающее устройство воспринимается как текстовый файл. В модуле принтера имеется описание

```
var lst : text
```

которое связывает имя lst с каналом устройства LPT1. Итак, в примере 17.6 нужно добавить лишь строки

```
uses printer;
writeln(lst,i:i);
```

чтобы выдать на печатающее устройство n чисел.

Пример 17.7:

Небольшую программу для вывода текстового файла на печать и управления принтером мог бы написать каждый. При этом можно было бы задать шрифт, левую границу, а также число символов в строке. Печатающим устройством можно управлять с помощью последовательности переключения кодов. Предположим, мы имеем графический принтер фирмы IBM (или совместимый с ним). Он позволяет использовать в качестве управляющих символов даже не существующие в таблице кодов ASCII символы. Например:

Желаемый результат: включить курсив
Управляющие символы в тексте: номер 133 в таблице ASCII
Управляющие символы принтера: <ESC>%G

Другие соответствия демонстрирует следующая программа:

```
program druck_program;
uses printer;
const esc = #27;
var quelle:text;
    rand:integer;
    c:char;
    filename:string[18];
begin
    writeln('Введите имя исходного файла для печати:');
    readln(filename);
    assign(quelle,filename);
```

```
    reset(quelle);
  writeln('Введите ширину левого полz');
   readln(rand);
    while not eof(quelle) do begin
       write not eof(quelle) do begin
          write(lst,' ':rand);
          while not eoln(quelle) do begin
                   read(quelle,c);
                   case ord(c) of
                   133: write(lst,esc,'%g');
                        (* Включить курсив *)
                   134: write(lst,esc,'%h');
                        (* Отключить курсив *)
                   137: write(lst,esc,'e');
                        (* Включить жирный шрифт *)
                   138: write(lst,esc,'f');
                        (* Отключить жирный шрифт *)
                   139: write(lst,esc,'w1');
                        (* Включить разрядку *)
                   140: write(lst,esc,'w0');
                        (* Отключить разрядку *)
                   145: write(lst,chr(18));
                        (* Установить 10 точек/дюйм *)
                   146: write(lst,esc,':');
                        (* Установить 12 точек/дюйм *)
                   147: write(lst,chr(15));
                        (* Установить 17.1 точек/дюйм *)
                   else write(lst,c) end;
                                   end;  (* eoln *)
                   readln(quelle); writeln(lst);
                        end;  (* eof *)
end.
```

В программу легко добавить строки для задания числа строк на странице распечатки, разбивки на страницы, добавления колонтитула и пр.

Пример 17.8:

Выше была приведена функция seekeoln. Поясним ее использование на нескольких примерах. Если в файле f:text в некоторой строке стоит

Это одна строка

то строки (c:char)

```
while not eoln(f) do
    begin read(f,c); write(c) end;
```

обеспечивают вывод на экран строки

Это одна строка

в то время как включение в программу строк

```
while not seekeoln(f) do
    begin read(f,c); write(c) end;
```

приведет к выводу на экран такой строки:

Это одна строка

Чтение и запись для текстовых файлов осуществляются через буфер. Обычно буфер имеет размер 128 байтов. При интенсивных операциях чтения/записи может оказаться желательным иметь буфер большего размера. Поэтому предусмотрен выбор размера буфера с помощью процедуры settextbuf, представленной на рис. 17.4. Обращение

```
settextbuf(f,buf);
```

означает, что в качестве буфера используется не внутренний буфер длиной 128 байт, а переменная buf. Если, например,

```
buf:array[1..10240] of char;
```

то буфер имеет длину 10 Кбайт. Если третий параметр в settextbuf отсутствует, под buf используется весь объем памяти. В приведенном ниже примере для копирования используется буфер, размер которого превышает стандартный. Тогда процедура settextbuf должна задаваться, естественно, с одним и тем же буфером. Если же процедуру settextbuf вызвать тогда, когда операции ввода/вывода уже выполнены со стандартным буфером, это приведет к потере информации.

Пример 17.9:

Скопировать текстовый файл из примера 17.4, используя буфер длиной 1 Кбайт.

```
program text_file_copieren;
var quelle,ziel:text;
```

```
    qname,zname:string[20];
    puffer: array[1..1024] of char;
procedure copy(var q,z:text);
var c:char;
begin
   while not eof(q) do
    begin
      read(q,c);
      write(z,c);
    end;
   close(q); close(z);
end;

begin
  write('Исходный файл:     '); readln(qname);
  write('Выходной файл:     '); readln(zname);
  assign(quelle,qname);
  assign(ziel,zname);
  settextbuf(quelle,puffer);
  settextbuf(ziel,puffer);
  reset(quelle);
  rewrite(ziel);
  copy(quelle,ziel);
end.
```

17.3. Нетипизированные файлы

Рассматриваемый нетипизированный файл является просто последовательностью байтов без какой-либо структуры. Тогда можно на примитивном уровне очень быстро и эффективно реализовать доступ к "записям данных" или, что лучше, к блокам, содержащим требуемое число байтов. Поскольку нетипизированные файлы совместимы с другими типами файлов, их предпочтительно использовать тогда, когда переменная типа file требуется лишь для операций erase, rename или других операций ввода/вывода.

Нетипизированный файл описывается с помощью служебного слова file без указания типа записи, то есть достаточно задать

```
var f:file;
```

Для нетипизированных файлов следует иметь в виде некоторые особенности применения стандартных процедур и функций:

rewrite(f) и reset(f) расширяются благодаря параметру blockgroesse типа word:

```
rewrite(f,blockgroesse); reset(f,blockgroesse);
```

blockgroesse задаает размер передаваемого блока. Если параметр blockgroesse отсутствует, размер блока принимается равным 128 байт. Процедуры и функции filesize, filepos и seek используют компоненты по blockgroesse байт каждая.

Процедуры read и write заменяются более быстрыми процедурами передачи данных blockread и blockwrite (рис.17.8).

Вызов:	blockread(f,puffer,anzahl,erg);	Процедура
Параметры:	var f:file; puffer какая-либо переменная;	
Действие:	Из файла f в буфер puffer считывается максимум anzahl блоков. Передача начинается с первого байта переменной puffer. Действительно считанное число блоков стоит в переменной erg. Буфер puffer должен иметь размер по крайней мере anzahl*blockgroesse. blockgroesse задается в reset или устанавливается по умолчанию равным 128. Указатель записи перемещается на erg блоков.	
Примечание:	С помощью blockread можно считать самое большее 64 Кбайта.	
Вызов:	blockwrite(f,puffer,anzahl,erg);	Процедура
Параметры:	var f:file; puffer какая-либо переменная; anzahl:word; var erg:word;	
Действие:	Из буфера puffer в файл f записывается максимум anzahl блоков. Передача начинается с первого байта переменной puffer. Действительное число записанных блоков фиксируется в переменной erg. Буфер puffer должен иметь размер по крайней мере anzahl*blockgroesse. blockgroesse задается в rewrite или устанавливается по умолчанию равным 128. Указатель записи перемещается на erg блоков.	
Примечание:	С помощью blockwrite можно записать самое большее 64 Кбайта.	

Рис. 17.8. Процедуры blockread и blockwrite

Пример 17.4 генерировал посимвольную копию файла, записанного на дискете. Преимущество такой структуры программы состоит в том, что символы считываются и записываются отдельно, то есть при копировании можно выполнять какие-то операции над данными. Если же требуется лишь чистое копирование, прибегают к нетипизированным файлам, задавая размер блока в reset или rewrite равным 1, а затем передавая столько записей, сколько вмещает буфер.

Пример 17.10:

Нужно скопировать в файл ziel файл quelle.

```
program datei_kopieren;
const bytezahl = 10000;
var quelle,ziel:file;
    quellname,zielname:string[30];
    puffer:array[1..bytezahl] of char;
    gelesen, geschrieben : word;
begin
      writeln('Задайте имя исходного файла:');
      readln(quellname);
      assign(quelle,quellname);
      reset(quelle,1);
      writeln('Задайте имя выходного файла:');
      readln(zielname);
      assign(ziel,zielname);
      rewrite(ziel,1);
      repeat
         blockread(quelle, puffer,
                   sizeof(puffer), gelesen);
         blockwrite(ziel, puffer,
                    gelesen, geschrieben);
      until (gelesen = 0) or (geschrieben <> gelesen);
      close(quelle); close(ziel);
end.
```

18. Тип данных "указатель"

В главе 7 были описаны переменные. Согласно рис. 7.1 переменная задается тремя параметрами: именем, типом, значением и адресом. Благодаря типу данных "указатель" появляется возможность использовать в качестве значения переменной адрес другой переменной, так чтобы такая переменная как бы "указывала" на другую (см. рис. 18.1).

Имя	z
Задание типа	"указывает на переменную типа real"
Значение	$24A2:$1000
Адрес	$1000:$2A34

Рис. 18.1. Переменная типа "указатель"

Значением переменной z с адресом $1000:$2A34 является адрес переменной типа real (значение которой записывается начиная с адреса $23A2:$1000). Прежде чем показать, как описывается переменная такого типа и какие значения она может принимать, напомним, как можно в Турбо Паскале работать с адресами.

18.1. Адреса и операции над адресами

Вначале кратко поясним, как в процессоре семейства Intel 8086 (а также 8088, 80286, 80386) образуется адрес. Наименьшей адресуемой единицей памяти является байт, состоящий из 8 битов. Два байта образуют слово, четыре байта - двойное слово, 16 байт называются разделом. Напомним, что 2**10 байт = 1 килобайт (Кбайт), 2**20 байт = 1 мегабайт (Мбайт), 2**30 байт = 1 гигабайт (Гбайт). Адресное пространство состоит из адресов длиной в 16 или 32 бит:

16 бит: 2**16 = (2**6)*(2**10) = 64 Кбайта = 65536 адресов
32 бита:2**32 = (2**2)*(2**30) = 4 Гбайта адресов

Поскольку для одного адреса 16 бит слишком мало, а 32 бита слишком много, в процессоре 8086 нашли "золотую середину" и адрес здесь образуется из 20 битов, так что таким образом можно представить 2**20 = 1 Мбайт адресов. В частности, это делается следующим образом:

Память разбивается на сегменты по 64 Кбайта. Начальный адрес сегмента всегда кратен 16, следовательно, два сегмента отстоят один от другого по крайней мере на 16 байт (= 1 параграфу). Адрес сегмента состоит из 16 бит, так что может быть самое большее 65536 сегментов.

Линейная адресация в сегменте называется смещением и также имеет 16-разрядные адреса. Сегмент задается своим базовым адресом, так что один адрес всегда состоит из пары сегмент:смещение, длиной по 16 бит каждый (а значит, каждый имеет тип word). Тогда абсолютный адрес образуется следующим образом:

сегмент*16 + смещение

то есть адрес сегмента сдвигается влево на четыре разряда и складывается со смещением:

сегмент:	$0060	(16 бит)	$0060	+
смещение:	$01A0	(16 бит)	$01A0	+
	Абсолютный адрес (20 бит)		$007A0	

Отсюда видно, что сегменты могут перекрываться, т.е. один абсолютный адрес может образовываться несколькими способами:

сегмент:	\$0020	-1	—	сегмент: \$001F
смещение:	\$0010	+16	—	смещение: \$0020
абс.адрес:	\$00210			абс.адрес:\$00210

Такая многозначность является следствием того, что из 16+16 бит сегмента и смещения для образования адреса используются только 20 бит. Сегменты всегда имеют размер 64 Кбайта, но минимальное расстояние между сегментами 16 байт!

В Турбо Паскале сегмент и смещение адреса представлены типом word, весь адрес имеет длину 32 бита, причем левые 16 бит являются адресом сегмента, а правые 16 бит - адресом смещения.

Тип данных "указатель" вводится следующим образом:

Тип "указатель" : (18-1)

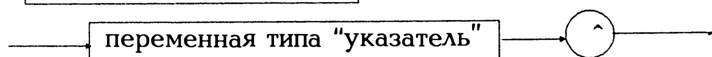
Вот пример описания переменной такого типа:

```
type real_zeiger = ^real;
var dies, das  : real_zeiger;
```
или
```
var dies, das : ^real;
```

даже если наименование типа real_zeiger в программе не используется.

Переменная типа "указатель" указывает на переменную типа, заданного в описании типа. Переменная связывается в этим типом. В качестве значения такая переменная имеет адрес той переменной, на которую она указывает.

Переменной, на которую указывает указатель, не обязательно присваивать какое-либо имя. К ней можно обращаться через имя указателя, а потому она называется ссылочной переменной (см.14-2).



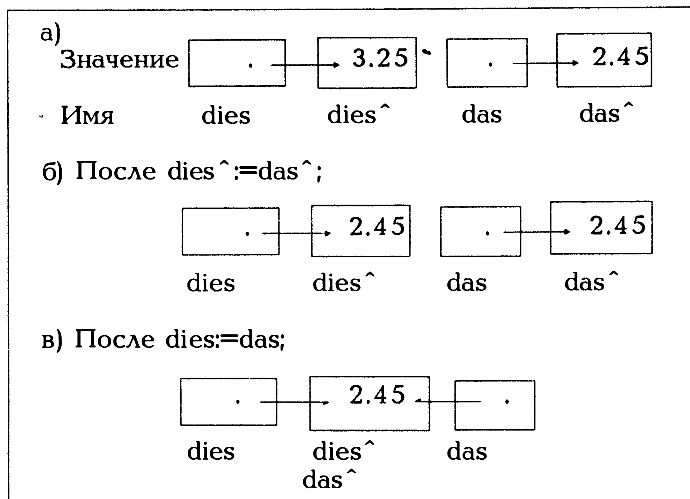


Рис. 18.2. Элементарные операции присваивания для переменных типа "указатель

Переменная dies типа "указатель" имела в качестве значения адрес вещественного числа 3.25, и имя dies^, под которым можно было обратиться к этому числу. Точно так же das указывала на число 2.45. Соответственно нужно различать два типа операций присваивания значения (см. рис. 18.2):

```
dies^ := das^;
```

Ссылочная переменная dies^ получает значение das^, т.е. dies и das указывают теперь на одинаковые объекты 2.45 (рис.18.2б). Согласно

```
dies := das;
```

переменные-указатели имеют одно то же значение, то есть указывают на один и тот же объект 2.45 (обратите внимание на различие формулировок "одинаковые объекты" и "один и тот же объект", рис. 18.2в).

В последней операции присваивания следует иметь в виду, что dies и das должны указывать на объект одного и то же типа. Оператор присваивания

```
var rptr : ^real;  wptr : ^word;
rptr := wprt;
```

приводит при компиляции к ошибке, поскольку rptr и wprt - объекты различного типа. Между тем в качестве значения интерес представляет только один адрес, а потому существует стандартный тип pointer, совместимый со всеми типами указателей.

Прежде чем приступить к практическому использованию переменных типа "указатель , ответим на следующие три вопроса:

- Как присваивается значение переменной типа "указатель ?
- Какие операторы и константы существуют для переменной типа "указатель ?
- Как можно выдать значение переменной типа "указатель ?

Начнем с первого вопроса: Как присвоить переменной p типа "указатель некоторое значение? Для этого существует две принципиально различных возможности:

- с помощью стандартной процедуры new(p) переменная p получает адрес динамической памяти и там резервируется место для переменной p^. Этот случай настолько важен, что ему посвящен целый раздел 18.2;
- через оператор присваивания p:=допустимый адрес.

Рассмотрим вначале второй случай. Поскольку прежде всего существует оператор @ для задания адреса, применимый ко всем переменным:

```
var i : integer; r : real;
    zi: ^integer; zr : ^real;

i := 135; zi := @i;  (* Итак, zi^ = 135 *)
r := 1.3;  zr := @r;  (* Итак, zr^ = 1.3 *)
```

Кроме того, для работы с адресами существуют представленные на рис. 18.3 функции.

Вызов:	addr(x)	Значение функции: pointer
Параметры:	x какая-либо переменная, процедура или функция	
Действие:	Значением функции является указатель на x, совместимый со всеми типами указателей. Оператор адресации @ имеет то же самое действие.	
Вызов:	ptr(srg,ofs)	Значение функции: pointer
Параметры:	seg,ofs : word;	
Действие:	Значением функции является адрес, образованный с помощью seg или ofs в качестве сегмента. Этот указатель совместим со всеми типами указателей. На значение функции можно сослаться с помощью ^	

Вызов:	seg(x)	Значение функции: word
Параметры:	x какая-либо переменная, процедура или функция	
Действие:	Значением функции является адрес сегмента объекта x.	
Вызов:	ofs(x)	Значение функции: word
Параметры:	x какая-либо переменная, процедура или функция	
Действие:	Значением функции является смещение объекта x.	
Вызов:	dseg	Значение функции: word
Параметры:	нет	
Действие:	Значением функции является адрес сегмента данных, то есть содержимое регистра ds (см. рис. 21.1).	
Вызов:	cseg	Значение функции: word
Параметры:	нет	
Действие:	Значением функции является адрес кодового сегмента, то есть содержимое регистра cs (см. рис. 21.1).	
Вызов:	sseg	Значение функции: word
Параметры:	нет	
Действие:	Значением функции является адрес сегмента стека, то есть содержимое регистра ss (см. рис. 21.1).	

Рис. 18.3. Стандартные функции для работы с адресами

Перейдем ко второму вопросу: Какие константы и операторы существуют для переменных типа "указатель"? Для такого типа переменных имеется единственная константа nil, т.е. после

```
var p : pointer;
    p := nil;
```

переменная p не указывает ни на что (внутреннее представление $0000:$0000). На вопрос об операторах также ответить очень просто, поскольку их просто нет, следовательно, ничего подобного

```
p + 2
p - sizeof(real)
```

не существует.

Здесь необходимо сделать одно критическое замечание. В стандартном Паскале оператора @ не существует, а создатель Паскаля Н.Вирт предполагал генерировать указатель только с помощью new(p), так чтобы он всегда указывал на динамическую область. Указатель никогда не может указывать на статическую переменную программы. Этот принцип сохранился и в версии 3.0 Турбо Паскаля. В версии 4.0 имеется оператор @, введенный прежде все-

го для того, чтобы обеспечить совместимость Турбо Паскаля с соответствующими операциями над адресами, применяемыми в языке Си. К сожалению, этого не получилось. Следовательно,

```
var a : array[1..10] of real;
    rz: ^real;

rz := @a;            (* следовательно, rz^ = a[1] *)
rz := rz +1           (* но не rz^ = a[2] *)
```

неприменимо (что для языка Си является обычным). Переменные типа "указатель" могут проверяться на равенство с помощью операторов сравнения = и <> :

```
var p, p1, p2 : pointer;
if p1 = p2 then .........
while p <> nil do........
```

При сравнении адресов следует быть осторожными:

Сравниваются части, относящиеся к сегменту и смещению.

Указатели логически могут оказаться не равными, хотя занимают одну и ту же ячейку памяти. Итак, адреса $0020:$00A1 и $0000:$02A1 будут признаны различными!

Генерируемые с помощью new и getmem указатели нормализуются, т.е. смещение всегда лежит в диапазоне $0000...$000F, следовательно, приведенные выше адреса были бы нормализованы к виду $002A:$0001.

Сгенерированные с помощью ptr адреса не нормализуются, так что при сравнении адресов будьте осторожны!

В заключение остановимся на третьей вопросе: Как можно выдать значение переменной типа "указатель"? Внутреннее представление указателя - это 32-разрядное число, у которого левые 16 бит являются адресом сегмента, а правые 16 бит смещением. Несмотря на это такое число не принадлежит к типу longint. Запись

```
var p : pointer;
p := какой-либо адрес;
write(p);
```

приведет к сообщению об ошибке "Cannot write variable of this type" ("Нельзя записать переменную такого типа"). Итак, для того, чтобы вывести значение p в нужной Вам форме в виде абсолютного адреса (отдельно сегмент:смещение) в десятичном или шестнадцатеричном представлении, нужно написать собственную процедуру. Такая процедура приведена в примере 18.1.

После всего сказанного может сложиться представление, что после

```
var p:^integer;
```

переменная p^ является совершенно обычной целой переменной. Это так лишь тогда, когда p имеет некоторое определенное значение, а именно адрес p^. Наиболее распространенная ошибка, когда программу начинают строкой

```
p^ := 17;
```

не присвоив p значения, например, с помощью p:=@variable или new(p). Тогда неясно, где же находится константа 17.

Пример 18.1:

Выдать адрес в форме сегмент:смещение, соответственно, в десятичном и шестнадцатеричном виде.

```
program zeiger_test;
uses crt;
var a,b : integer;
    za : ^integer;
    w : word;

procedure hexwrite_byte(b:byte);
(* Вывести байт b в шестнадцатеричном представлении *)
var nr,links,rechts:integer;
begin
(*   write('        $'); *)
   links:= b div 16;
   case links  of
      0..9:   write(chr(links+48));
      10..15: write(chr(links+55));   end;
   rechts:= b mod 16;
   case rechts of
      0..9:   write(chr(rechts+48));
      10..15  write(chr(rechts+55));   end;
end;

procedure hexwrite_word(w:word);
(* w выдать в шестнадцатеричном виде *)
begin
   write('$');
   hexwrite_byte(hi(w));
```

```
   hexwrite_byte(lo(w));
end;

begin
   clrscr;
   a := 17;
   za := @a;
   b := za^;
   writeln(a:6, b:6);
   writeln(longint(@a));
   (* 32 бита: левые 16 бит - сегмент,
     правые 16 бит - смещение *)
   write(longint(@a) div 65536:6);
   write(longint(@a) mod 65536:5);
   writeln('seg:ofs  ',seg(a):5,':',ofs(a):5);
   writeln('  ':9); hexwrite_word(seg(a)); write(':');
   hexwrite_word(ofs(a)); writeln;
   writeln('seg:ofs  ',seg(b):5,':',ofs(b):5);
   writeln(dseg);
   write('  ':9); hexwrite_word(seg(b)); write(':');
   hexwrite_word(ofs(b)); writeln;
end.
```

Пример 18.2:

Пример указателя массива и указателя компоненты.

```
program zeiger_test;
uses crt;
type feld = array[1..10] of integer;
var a,b,i:integer;
    za : ^integer;
    w : word;
    f : feld;
    zf : ^feld;

begin
   clrscr;
   for i := 1 to 10 do f[i] := 10*i;
   zf := @f;
   writeln(longint(@f));
   writeln(zf^[1]);
   zf := @f[2];
   writeln(zf^[1]);   (* —> 20 *);
   zf := ptr(seg(f[3]),ofs(f[3])+sizeof(integer));
   writeln(zf^[1]);   (* —> 40 *);
end.
```

18.2. Динамические переменные и структуры связанных данных

С переменной типа "указатель" связано понятие динамической переменной. Все переменные Паскаля статичны в том смысле, что они действуют во всем том блоке, в котором они описаны, что касается и переменных типа "указатель". В то же время ссылочные переменные, на которые они указывают, могут быть динамическими. Они вызываются стандартной процедурой new и действуют до их отмены с помощью dispose.

Вызов:	new(p)	Процедура
Параметры:	var p:pointer;	
Действие:	Резервирует в динамической области место для ссылочной переменной p^ и присваивает p в качестве значения начальный адрес p^. Если места в памяти недостаточно, выдается сообщение об ошибке	
Вызов:	dispose(p)	Процедура
Параметры:	var p:pointer;	
Действие:	Занятое ссылочной переменной p^ место в памяти вновь объявляется доступным (и может вновь использоваться при следующем вызове new(p)). После этого значение p не определено.	

Рис. 18.4. Процедуры new и dispose

Управление памятью, занятой ссылочной переменной p^, должно осуществляться new(p) и dispose(p) динамически. Назовем управляемую таким образом область памяти динамической памятью или "кучей". В начале программы следует установить указатель динамической области на ее начало, которое при каждом new(p) сдвигается соответственно потребности в памяти для p^. Для dispose(p) соответствующее место в памяти объявляется свободным, так что динамическая область может иметь "пропуски", которые заполняются при следующих вызовах процедуры new(p).

new и dispose являются предусмотренными стандартным Паскалем процедурами для работы с переменными типа "указатель" и динамической областью. Мы пока остановимся на этом, а в конце главы рассмотрим другие имеющиеся в Турбо Паскале возможности.

Пример 18.3:

С помощью следующей программы можно проверить, сколько памяти можно отвести в Вашем компьютере под динамическую память.

```
program heaptest;
type feld = array[1..1024] of byte;
     zeiger =  ^feld;
var  z:zeiger; i:integer;

begin
   i := 1;
   repeat
      new(z);
      if i mod 10 = 0
           then writeln(i:6,'   Кбайт занято');
         i:=i+1;
   until false;
end.
```

Программа прерывается, если динамическая область израсходована, следовательно, когда new(p) нельзя больше выполнить. Позже мы еще раз вернемся к этому в примере 18.9.

Сложности могут возникнуть в том случае, когда два указателя p и q указывают на одну и ту же ячейку динамической памяти. Тогда встает вопрос, есть ли после dispose(p) доступ к значению q^.

Пример 18.4:

```
program test;
var p,q:^integer;
begin
(*1*)    new(p);
(*2*)    p^ := 17;
        writeln(p^:10);
(*3*)    q := p;
        writeln(q^:10);
(*4*)    dispose(p);
(*5*)    p^ := 23;
(*6*)    writeln(q^:20);
end.
```

Согласно строке (*1*) p получает в качестве значения адрес динамической области. В ту же ячейку в строке (*2*) записывается значение 17. Согласно строке (*3*) q получит тот же адрес, что и p, а следовательно, q^=p^=17. В строке (*4*) память, на которую

указывают p и q, освобождается. До сих пор было неясным, будет ли при этом утрачено значение p или лишь соответствующая ячейка памяти будет освобождена для других применений. Из приведенного выше примера выяснилось, что p сохраняет свое значение и после выполнения процедуры dispose(p), а 17 изменяется на 23.

Приведенный пример служит прежде всего предостережением на тот случай, когда на одну и ту же ячейку указывает несколько указателей,а затем dispose() применяется к одному или нескольким из этих указателей.

Переменные типа "указатель служат прежде всего для того, чтобы создавать структуры связанных данных (связанные списки и деревья), где элемент вызывается с помощью указателя на предшественник или последователь такого элемента. При использовании таких структур можно создавать весьма элегантные и эффективные программы. Примеры тому приведены в [7]. Здесь же дадим лишь один пример простого связанного списка (рис. 18.5).

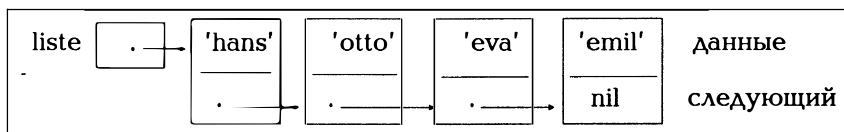


Рис. 18.5. Пример связанного списка

Каждый элемент связанного списка является записью, состоящей из двух частей: данные и указатель на следующий элемент списка. Конец списка помечается указателем nil. Начало списка формирует переменная типа "указатель (liste), содержащая первый элемент списка. С точки зрения техники программирования, список характеризуется только переменной liste, заголовком списка, являющейся указателем на первый элемент списка.

Пример 18.5:

Следующая программа демонстрирует создание и печать одного такого списка.

```
program gekettete_liste;
uses crt;
type datentyp = string[4];
     zeiger = ^elementtyp;
     elementtyp = record daten:datentyp;
                         next:zeiger   end;
var liste:zeiger; d:datentyp;
```

```
procedure anfuegen(var l:zeiger; d:datentyp);
{В список, начинающийся с l, добавляется элемент
 содержащий в качестве части данных d.
 Процедура написана в виде итерации, т.е. указатель
 p просматривает список сверху вниз, пока не
 найдет nil.}
var p,q:zeiger;
begin
    new(q); q^.next:=nil; q^.daten:=d;
    {Это добавляемый элемент. Теперь отыскивается
     конец списка}
    if l = nil then l := q {Список пуст}

              else begin
                     p := l;
                     while p^.next <> nil
                       do p := p^.next;
                     {Добавить элемент:} p^.next :=q;
                      end;
end   {Добавление};

procedure liste_drucken(l:zeiger);
{Распечатать начинающийся с l список.}
{Процедура написана в виде рекурсии}
begin
   if l = nil then writeln('      Конец списка')
             else begin writeln(l^.daten);
                     liste_drucken(l^.next);
                  end;
end  {Распечатка списка};

begin      (******** Исполняемая часть *********)
   clrscr;
   liste := nil;
   writeln('Введите часть списка, содержащую данные
          (Конец = ****)');
    readln(d);
    while d <> '****' do begin anfuegen(liste,d);
                             readln(d);
                         end;
    liste_drucken(liste);
end.
```

Пример 18.6:

Теперь продемонстрируем еще две важнейшие операции над списками, а именно

- стирание, то есть удаление некоторого элемента списка;

- добавление элемента в список после определенного элемента. Воспользуемся для этого списком из примера 18.5.

```
procedure loeschen(var l:zeiger; d:datentyp);
{В списке, начинающемся с l, стереть элемент,
содержащий в качестве данных d}
var q:zeiger;
   {Указывается на удаляемый элемент}
begin
  if l = nil
      then writeln('Элемент, содержащий ',d,' не найден')
      else
  if l^.daten <> d
      then loeschen(l^.next,d)
      else begin q := l;
              l := l^.next;  {Удалить элемент}
              dispose(q);  {Ячейка памяти свободна}
                    end;
end   {Удаление};

procedure einfuegen_nach(var l:zeiger; x,d:datentyp);
{В список, начинающийся с l, после элемента, содержащего
 в качестве данных x, добавляется элемент, содержащий d}
var q:zeiger;
begin
   if l = nil
     then writeln('Элемент, содержащий x, не найден')
     else
   if l^.daten <> x then einfuegen_nach(l^.next,x,d)
                  else begin new(q);
                          q^.daten := d;
                          q^.next := l^.next;
                          l^.next := q;
                       end;
end   {Добавить после...};
```

Особенно интересными структурами связанных данных являются упорядоченные двоичные деревья. Дерево состоит из узлов и ветвей, за которыми вновь следуют узлы. Узлы, в которые не входит никаких ветвей, называются корневыми. Узлы, из которых не выходит ветвей, называют листьями. Тем самым можно строить иерархические отношения. Каждый узел вновь состоит из части, содержащей данные (скажем, некий элемент key, позволяющий однозначно идентифицировать узел), и указателя, показывающего на следующий узел (то есть ветви).

Если каждый узел имеет самое большее по две ветви, говорят о бинарном или двоичном дереве. Тогда каждый узел имеет одного

левого и одного правого последователя. Подобное дерево можно было бы описать, например, таким образом:

```
type zeiger   = ^knoten;
     datentyp = (* какой-то тип данных *);
     knoten   = record daten : datentyp;
                       links, rechts : zeiger end;
var baum : zeiger;
```

Если кроме того для каждого узла выполняется правило, что все левые примыкающие к этому узлу узлы меньше (то есть имеют меньший ключ), а все правые узлы больше, то такое бинарное дерево называют упорядоченным. На рис. 18.6 показано такое дерево. В силу своего свойства упорядоченности применение подобных деревьев для поиска исключительно удобно. Назовем их также поисковыми деревьями или деревьями поиска.

В следующем примере строится и выводится на печать упорядоченное бинарное дерево.Функция liesknoten() отводит место в памяти для нового узла и считывает данные для него. Значением функции является указатель на такой новый узел hilf. Функция einfuegen() добавляет такой новый узел в дерево. wurzel указывает на корневой узел (на начало nil). Добавление управляется через ключи: если hilf.key меньше значения ключа рассматриваемого в настоящий момент узла дерева, новый узел holf относится к левому поддереву, в противном случае - к правому. В конце выполнения процедуры einfuegen() новый узел hilf добавляется в дерево в качестве листа.

Существует три способа просмотра всех узлов двоичного дерева. Следует начинать от корня рекурсивно для каждого узла :
preorder: узел, левая ветвь, правая ветвь (прямой просмотр);
inorder: левая ветвь, узел, правая ветвь (обратный просмотр);
postorder: левая ветвь, правая ветвь, узел (концевой просмотр).

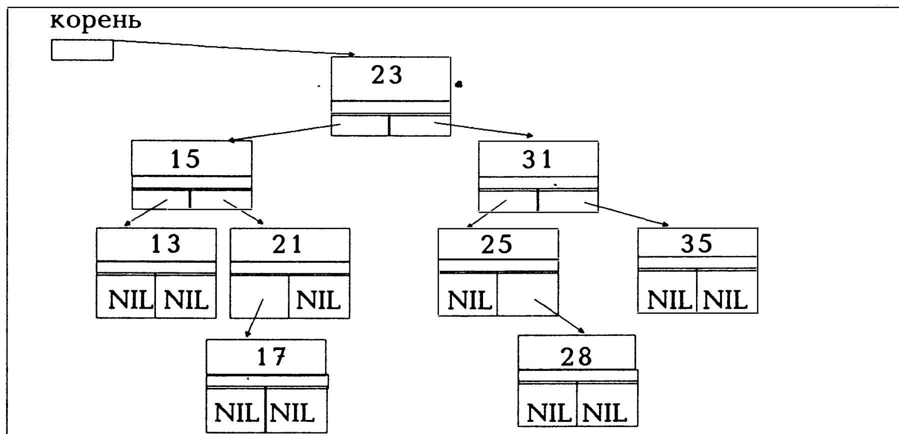


Рис. 18.6. Упорядоченное двоичное дерево

При этом можно, конечно, поменять местами правую и левую части. При просмотре упорядоченного двоичного дерева согласно inorder получаются узлы в отсортированной последовательности (узлы слева направо увеличиваются, а справа налево уменьшаются). Функция baum_drucken() использует inorder, так что узлы выдаются в возрастающей последовательности.

Пример 18.7:

```
program binaerer_baum;
uses crt;
type zeiger = ^knoten;
     knoten = record key : integer;
                     links,rechts:zeiger     end;
var baum : integer;
    z    : integer;

proceudre einfuegen(var b:zeiger; k:integer);
begin
   if b = nil
     then begin new(b);
              with b^ do
              begin
                links:=nil;
                rechts:=nil;
                key:=k;
              end;
            end
     else with b^ do
          begin
            if key < k then einfuegen(links,k);
            if key > k then einfuegen(rechrs,k);
          end;
end   (* Добавить *);

procedure druckbaum(b:zeiger; tiefe:integer);
begin
  if b = nil
     then writeln('  Дерево пусто   ')
     else with b^ do
            begin
              tiefe:=tiefe+1;
              if links<>nil
                 then druckbaum(links,tiefe);
                 write(' ':4*tiefe); writeln(key:4);
                 if rechts<>nil
```

```
                then druckbaum(rechts,tiefe);
             tiefe:=tiefe-1;
          end;
end   (*   Дерево распечатки *);

procedure baum_loeschen(var b:zeiger);
(* Занятое в памяти деревом b место
   вновь освобождается *)
begin
   if b <> nil then
      begin
         if b^.rechts <> nil
            then baum_loeschen(b^.rechts);
         if b^.links <> nil
            then baum_loeschen(b^.links);
         dispose(b);
       end;
end;  (* Освободить *)

begin    (******** Исполняемая часть ********)
   baum := nil;
   checkeof := true;
   writeln('Введите числа, завершение по ^z');
   while not eof do
      begin
        readln(z);
       einfuegen(baum,z);
      end;
   druckbaum(baum,0);
   baum_loeschen(baum);
end.
```

Двоичные деревья являются замечательными структурами для получения упорядоченных множеств данных, поскольку позволяют хранить их отсортированными. Форма дерева в примере 18.7 зависит, естественно, от порядка следования входящих в него узлов. Дерево может быть "несбалансированным" (если ключи отсортированы в порядке возрастания или убывания), при благоприятных обстоятельствах оно может быть полностью "сбалансированным". От этой сбалансированности зависит качество поиска.

Настоящая книга является введением в Турбо Паскаль, а не руководством программиста. Поэтому из соображений экономии места ряд интересных операций (удаление, поиск, изменение и пр.) над списками и деревьями далее обсуждаться не будет. Всю эту информацию можно найти в книге [7]. Удобство работы с адреса-

ми, предоставляемое Паскалем, делает его особенно интересным в случае использования структур связанных данных. В качестве примера смоделируем ниже стек с помощью простого связанного списка.

Пример 18.8:

С помощью операций popup и pushdown формируется простой связанный список. Конец списка является основанием стека, первый элемент списка является текущей вершиной стека. Тогда операция pushdown означает добавление некоторого нового элемента в список сверху. Операция popup удаляет первый элемент списка. Итак, заголовок списка может выполнять роль указателя стека.

```
program stack_simulation;
uses crt;
const stackmax = 10;
type stackprt     = ^stackelement;
     datentyp     =  integer;
     stackelement = record daten : datentyp;
                           next  : stackprt end;
var st            : stackprt;
    fertig        : boolean;
    stacksize, d  : integer;
(*  stacksize является глобальной переменной для всех
    процедур *)

procedure printstack(var st:stackprt);
begin
   if st = nil then write('    Конец стека ')
            else begin
                    writeln('|',st^.daten:4,' | ');
                    writeln('└──┘ ');
                    printstack(st^.next)
                 end;
end;  (* Распечатка содержимого стека *)

procedure push(var st:stackprt; d:datentyp);
var hilf:stackprt; c:char;
begin
   new(hilf); hilf^.daten := d;
   hilf^.next := st;
   st := hilf;
   stacksize := stacksize + 1;
end;   (* Поместить в стек *)
```

```
procedure pop(var st:stackprt; var d:datentyp);
var hilf:stackprt;
begin
   hilf := st; st := st^.next;
   d := hilf^.daten;
   dispose(hilf);
   stacksize := stacksize - 1;
end; (* Выборка из стека с удалением выбранного элемента*)

procedure create(var st:stackprt);
begin
   st := nil;
   stacksize := 0;
end;  (* Формирование *)

function isfull(st:stackprt):boolean;
begin
    isfull := stacksize >= stackmax;
end;

function isempty(st:stackprt):boolean;
begin
   isempty := st = nil;
end;

procedure menue;
var c:char;
begin
   clrscr;
   fertig := false;
   printstack(st);
   gotoxy(40,5);
   writeln('Что Вы хотите:');
   gotoxy(40,7);
   writeln('+ = push down');
   gotoxy(40,8);
   writeln('- = pop up');
   gotoxy(40,9);
   writeln('e = конец');
   gotoxy(40,10);
   readln(c);
   case c of
     '+': if isfull(st)
                then
                begin
                   gotoxy(40,20);
```

```
                write('К сожалению,');
                write('стек полон');
                readln(c);
             end
             else begin
                   gotoxy(40,12);
                   writeln('Введите данные:');
                   gotoxy(40,13);
                   readln(d);
                   push(st,d);
                end;
 '-': if isempty(st)
         then
         begin
            gotoxy(40,20);
            writeln('К сожалению, стек пуст');
            readln(c);
         end
         else
         begin
            pop(st,d);
            gotoxy(10,1);
            writeln('Извлечь из стека: ',d:4);
            readln(c);
         end;

 'e','E':fertig := true;   end;  (* Выбор *)
end;   (* Меню *)

begin (******** Исполняемая часть *********)
   create(st);
   repeat
      menue;
   until fertig;
end.
```

Мы хотели бы теперь рассмотреть динамическую область и управление ею несколько подробнее (рис. 18.7). В системных модулях имеются следующие стандартные переменные:

```
var   heaporg : pointer;
      heapptr : pointer;
      freeptr : pointer;
      freemin : word;
```

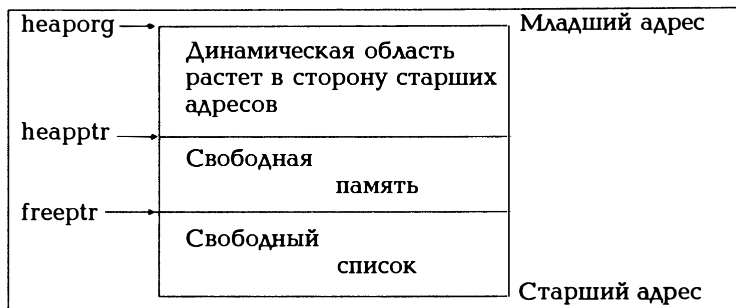


Рис. 18.7. Организация динамической области

Для управления динамической областью существуют следующие стандартные процедуры, функции и переменные:

Вызов:	mark(p)	Процедура
Параметры:	var p:pointer;	
Действие:	В переменную p записывается текущее значение heapptr.	
Вызов:	release(p)	Процедура
Параметры:	var p:pointer;	
Действие:	Динамическая область освобождается, начиная со значения p, причем прежде переменная p должна быть помечена с помощью mark(p).	
Вызов:	getmem(p,i)	Процедура
Параметры:	var p:pointer; i:word;	
Действие:	В динамической области под переменную p^ отводится ровно i байтов памяти.	
Вызов:	freemem(p,i)	Процедура
Параметры:	var p:pointer; i:word;	
Действие:	В динамической области памяти освобождается один блок длиной i байт. Переменная i должна точно соответствовать своему значению в предыдущем вызове процедуры getmem.	
Вызов:	maxavail	Значение функции:longint
Параметры:	нет	
Действие:	Значением функции является размер наибольшего непрерывного свободного блока динамической области памяти.	
Вызов:	memavail	Значение функции:longint
Параметры:	нет	
Действие:	Значением функции является объем доступной памяти в байтах.	

Рис. 18.8. Процедуры и функции для работы с динамической областью

Напомним еще раз: место в динамической области можно отводить с помощью процедур new или getmem. Освободить блок памяти можно с помощью dispose (при этом в динамической области могут образоваться "пропуски) или с помощью freemem (при этом имеющиеся "пропуски закрываются за счет сдвига содержимого памяти или с помощью mark/release (при этом освобождается вся динамическая область, лежащая выше указанного адреса). При освобождении памяти нужно соблюдать осторожность, то есть пользоваться либо new/dispose либо new/mark/release либо getmem/freemem, но ни в коем случае не перепутать в сочетании этих процедур.

Пример 18.9:

Вернемся еще раз к описанному в примере 18.3 грубому способу определению размера динамической области и несколько уточним его.

```
program heaptest;
uses crt;
type feld = array[0..1023] of byte;
     zeiger = ^feld;
var i:integer;
    y,z:zeiger;

procedure write_address(z:pointer);
begin
   writeln('сегмент:смещение',seg(z^):4,':',ofs(z^):4);
end;

begin
   clrscr;
    write('heaporg     '); write_address(heaporg);
    write('heapptr     '); write_address(heapptr);
    write('freeptr     '); write_address(freeptr);
    write('Свободно    '); writeln(memavail);
for i := 1 to 3 do
  begin
    new(z); writeln;
    write_address(z);
    write('heaporg    '); write_address(heaporg);
    write('heapptr    '); write_address(heapptr);
    write('freeptr    '); write_address(freeptr);
    write('Свободно   '); writeln(memavail);
    new(y);
    dispose(z);
  end;
end.
```

В заключение продемонстрируем возможности указателя еще на одном примере. Компьютерная игра ADVENTURE состоит в том, что игрок может перемещаться в некотором здании в двух или трехмерном пространстве и в отдельных помещениях переживать всевозможные приключения при встрече с призраками и грабителями.

Путешественник должен пройти в некоторое помещение двумерного здания, соединенное дверьми с другим помещением (тогда путешественник может выйти на свободу). Помещения здания могут соединяться между собой, например так, как показано на рис. 18.9.

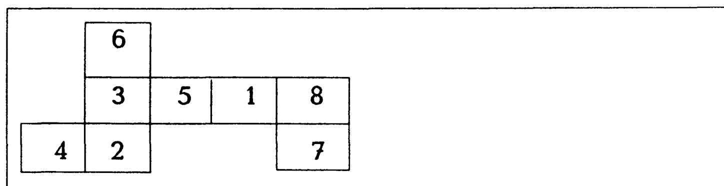


Рис. 18.9. План помещений для примера 18.10

Тогда, как показано на рис. 18.10, информация о плане здания должна храниться в файле.

	Дверь в комнату				
Номер комнаты	Норд	Ост	Зюйд	Вест	(0 = нет двери)
7	8	0	0	0	
1	0	8	0	5	
8	0	0	7	1	
5	0	1	0	3	
6	0	0	3	0	
3	6	5	2	0	
2	3	0	0	4	
4	0	2	0	0	

Рис. 18.10. Информация о плане здания, представленном на рис. 18.9

Очевидно, что порядок следования помещений в этом списке может быть любым, как и последовательность номеров помещений на плане. В следующем примере функция einlesen() обеспечивает размещение плана здания в памяти в виде структуры связанных данных. Файл с данными считывается и помещается в список, исходя из номера помещения (рис. 18.11а). Затем файл считывается еще раз и с учетом наличия дверей генерируется связь помещений (рис.18.11б).

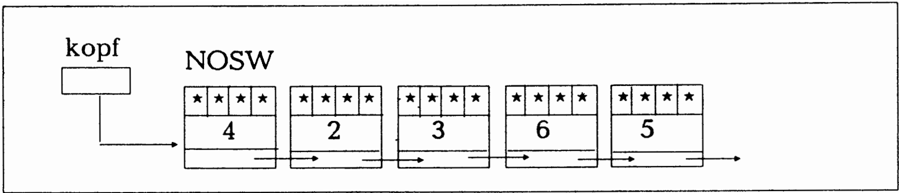


Рис.18.11а. Начало списка помещений после первого прочтения представленного на рис. 18.10 файла и расшифровки номеров помещений

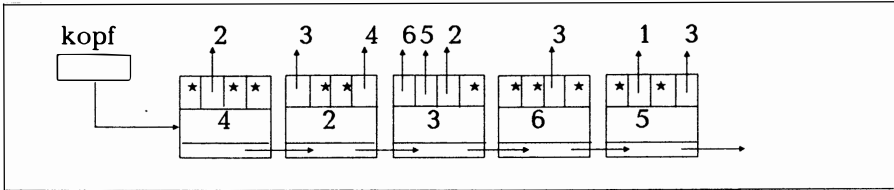


Рис. 18.11б. Начало списка помещений после второго прочтения представленного на рис. 18.10 файла и расшифровки дверей (*=nil)

Пример 18.10:

```
program raum_wandern;
{программа путешествия по лабиринту}
uses crt;
type richtung = (nord, ost, sued, west);
     raumzeiger = ^raum;
     raum = record nummer : integer;
                   tuer   : array[richtung] of
                            raumzeiger;
                   next   : raumzeiger;
                   end;
var kopf : raumzeiger;

function raum_suchen(kopf:raumzeiger; n:integer):
         raumzeiger;
  (***************** Лабиринт *************************)
  (* В начинающемся в kopf списке помещений          *)
  (* по номеру помещения n отыскивается нужное *)
  (* Значением функции указывает на это помеще- *)
  (*ние (рис.18.11а).                                  *)
  (*************************************************)
var p : raumzeiger;
begin
   if n=0 then p :=nil;
```

```
  p := kopf;
  while (p <>nil) and (p^.nummen <>n)
      do p := p^.next;
  raum_suchen := p;
end;

procedure einlesen (var kopf:raumzeiger);
  (***************** Считать ************************)
  (* Файл с планом помещений считывается       *)
  (* дважды. Вначале первый столбец использу-  *)
  (* ется под номера помещений, чтобы сформи- *)
  (* ровать связанный список (см. рис. 18*11a)   *)
  (* Затем используются данные о наличии две-  *)
  (* рей, чтобы сгенерировать путь между по-    *)
  (* мещениями (рис. 18.11 б).                   *)
  (************************************************)

var filename:string[20];
   r:richtung;
   f:text;
   p:raumzeiger; i,nr:integer;
begin
  write('Как называется файл с планом помещений?');
  readln(filename);
  assign(f,filename);
  reset(f);
  (* Вначале считываются только номера помещений,
     а записывается связанный список помещений,
     представленный на рис. 18.11a. *)

  kopf := nil;
  while not eof(f) do
    begin
      new(p);
      readln(f,i);
      p^.nummer := i;
      for r :=nord to west do p^.tuer[r]:=nil;
      p^.next := kopf;
      kopf := p
    end;
   (* Файл filename считывается еще раз: соответственно
      учитываются номера помещений и наличие дверей в
      помещении (список представлен на рис. 18.10).  *)
    reset(f);
    while not eof(f) do
      begin
```

```
            read(f,i);
            p := raum_suchen(kopf,i);
            for r := nord to west do
               begin read(f,nr);
                     if nr = 0
                        then p^.tuer[r] := nil
                        else
                          p^.tuer[r] :=
                          raum_suchen(kopf,nr);
               end;
          end;
end;  (* Считывание *)

procedure wandern;
var platz:raumzeiger;
    nr :integer;
    c: char;
begin
    writeln('С какого помещения Вы хотите начать?');
    readln(nr);
    platz := raum_suchen(kopf,nr);
    if platz = nil
       then begin
            writeln('Такого помещения нет');
                     exit;
            end;
       writeln('Вы в помещение No',platz^.nummer:3);
       repeat
          write('В каком направлении ');
          writeln('Вы хотите идти?');
          writeln('Введите n, o, s, w или e для выхода');
          readln(c);
          case c of
          'n','N':begin
                   platz :=platz^.tuer[nord];
                   if platz = nil
                      then writeln('Это невозможно')
                      else begin
                      write('Вы в помещение No');
                      writeln(platz^.nummer:3);
                           end;

                  end;
          'o','O':begin
                   platz :=platz^.tuer[ost];
                   if platz = nil
```

```
                then writeln('Это невозможно')
                else begin
                write('Вы в помещение No');
                writeln(platz^.nummer:3);
                     end;
             end;
      's','S':begin
               platz :=platz^.tuer[sued];
               if platz = nil
               then writeln('Это невозможно')
               else begin
               write('Вы в помещение No');
               writeln(platz^.nummer:3);
                    end;
             end;
      'w','W':begin
               platz :=platz^.tuer[west];
               if platz = nil
                  then writeln('Это невозможно')
                  else begin
                  write('Вы в помещение No');
                  writeln(platz^.nummer:3);
                      end;
             end;  end;  (*case*)
    until (c ='e') or (c = 'E');
end;  (* Путешествие *)

begin     (******** Исполняемая часть ********)
   einlesen(kopf);
   wandern;
end.
```

19. Построение программ по модульному принципу и модули

19.1. Построение программ по модульному принципу

Когда программа превышает определенный объем или когда одновременно несколько человек хотят работать над программой, можно разбить программу на части. Такое разбиение программы называется построением ее по модульному принципу. При этом можно выделить три ступени:

- разбиение на модули исходной программы;
- разбиение на модули программы в объектных кодах;
- раздельная трансляция модулей исходной программы.

Разбиение исходной программы на модули

Это самый простой этап построения программы по модульному принципу. Исходный текст хранится в нескольких файлах. Это лишь упрощает работу с редактором текстов программ. В основной программе, являющейся как бы "каркасом программы, с помощью директив компилятору "вставить ("Include" или "Insert) обращаются к подпрограммам. Такая директива в Турбо Паскале имеет вид

(*$Iимя_файла*)

В соответствующем месте основной программы добавляется текст из указанного файла. Компилятор всегда работает с полным исходным текстом. Это было обычным для версии 3.0. Программные средства версии 3.0 включают в себя соответствующие специальные программы в виде исходного текста на Паскале, которые добавляются в написанный программистом текст по директиве Include. В результате короткая графическая программа (с графическими стандартными средствами) или система управления файлами (со средствами управления базами данных) оказывается длиной в несколько тысяч строк. И тогда даже очень быстрый компилятор Турбо Паскаля оказывается не столь уж быстрым.

Разбиение на модули объектной программы (оверлей)

Обычно при запуске программы в оперативной памяти находятся все объектные коды. Здесь в случае, когда программа большая, а память ограничена, могут возникнуть сложности. Под оверлейным модулем понимают часть программы в объектных кодах, которая загружается в оперативную память лишь на время ее обработки. Путем разбиения объектной программы на оверлейные модули можно добиться использования одной и той же области памяти несколькими модулями поочередно. В результате оказывается работоспособной программа, не помещающаяся целиком в оперативной памяти.

В Турбо Паскале версии 3.0 процедуры и функции можно снабжать заголовком overlay. Такой заголовок означает, что компилятор помещает объектные коды не в тот файл, в котором записана основная программа, а в особый, так называемый оверлейный файл, из которого во время выполнения программы вызываются отдельные модули. Поскольку в версии 4.0 существует возможность раздельной трансляции модулей, формирование оверлейных структур там не предусмотрено. В версии 5.0 оверлейная структура в форме модулей вновь существует.

При модульном построении программы были разбиты на части исходный текст для редактирования и объектные коды для выполнения программы. Но подлинное разбиение на модули означает, что исходный текст программы можно компилировать по частям. Тогда после компиляции исходной программы, содержащей вызовы модулей, компилятор включит их в сгенерированный объектный код. Для этого должны выполняться две предпосылки:

- Компилятор должен быть в состоянии оттранслировать исходный текст, не являющийся полной программой. Такие части программы называются модулями или units.

- Должен иметься компоновщик, который после компиляции программы, содержащей ссылки на оттранслированные ранее модули и не являющейся полной программой, добавит эти модули в сгенерированный объектный код.

Поскольку компоновщик добавляет объектный код, вовсе не обязательно, чтобы тот был результатом трансляции написанного на Паскале модуля. Следовательно, такие модули можно писать и на других языках.

Версия 3.0 не располагает компоновщиком такого рода. Есть библиотека исполняющей системы (Run-Time-Library), содержащая стандартные программы и включающая сгенерированные компилятором объектные коды. Если самую короткую программу на Паскале

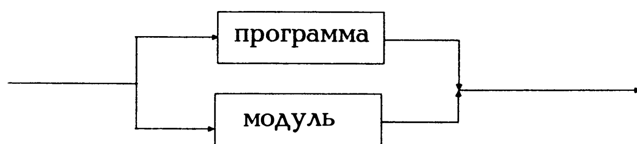
```
program nix; begin end.
```

скомпилировать с помощью Com, объем NIX.COM составит около 11 Кбайт!

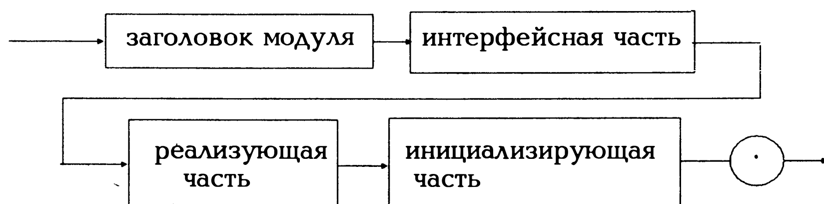
Существенным преимуществом версии 4.0 является отсутствие этого недостатка. Если приведенную выше программу скомпилировать в версии 4.0 с помощью Compile/Destination Disk, объектный код займет лишь 1360 байт!

19.2. Модули

Модули позволяют реально построить программу по модульному принципу, когда части программы можно компилировать по отдельности. Соответственно, имеются два вида компилируемых исходных текстов на Паскале.



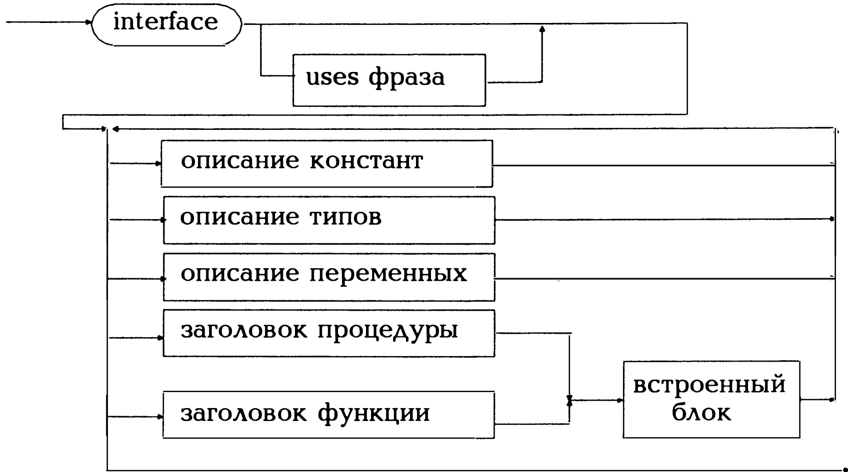
Синтаксическое понятие программы подробно пояснялось на диаграмме (4-1) и все рассмотренные до сих пор программы были построены по такому принципу. Теперь же возникают модули следующей структуры:



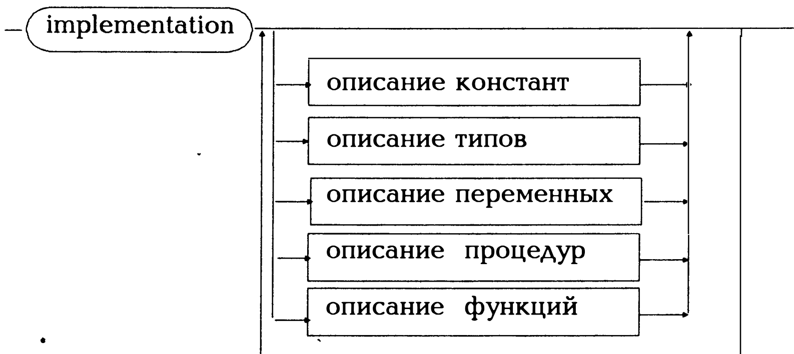
Модули служат для определения констант, типов данных, переменных, процедур и функций, которые могут использоваться затем другими программами. Заголовок модуля имеет такой вид:

Имя модуля используется, если хотят сослаться на него в предложении uses (4-3).

Интерфейсная часть описывает константы, типы, переменные, процедуры и функции "общего пользования , к которым с помощью uses имя_модуля могут обращаться другие программы или модули.



Вызывающая программа может использовать эти модули, обратившись к ним с помощью uses имя_модуля, также, как если бы они были описаны в самой программе. Все вызванные величины являются для вызывающей программы глобальными. В интерфейсной части стоят лишь заголовки процедур и функций. Тело процедуры или функции записывается в реализующую часть, имеющую следующий вид:

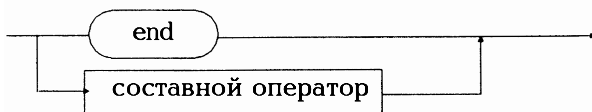


Наряду с этим здесь могут описываться константы, типы, переменные, процедуры и функции, которые не общедоступны, а используются модулем "приватно . Функции и процедуры не

нуждаются более в списке параметров. Параметры стоят уже в заголовках интерфейсной части, которые имеют то же действие, что и ссылка forward. Но в интерфейсной части заголовки могут повторяться, так что корректны обе приведенные ниже формулировки:

```
unit dies;
interface
procedure p(a,b:real);
implementation
procedure p;
(* Список параметров уже записан в интерфейсной части *)
begin
.
.
.
end;  (* of p *)
end.

unit das;
interface
procedure p(a,b:real);
implementation
procedure p(a,b:real);
(* Список параметров может повторяться.
   Но он должен совпадать с указанным в интерфейсной
   части. *)
begin
.
.
.
end; (* of p *)
end.
```

В инициализирующей части



(19-6)

можно выполнять операторы, введенные через uses имя_модуля. Эта часть состоит по крайней мере из завершающего модуль end.

Из программы компилятор генерирует согласно директиве Compile/memory disk работоспособную объектную программу .EXE или .COM. Из некоторого модуля с помощью Compile/memory disk получают объектный модуль .TPU, который добавляется компоновщиком в другие программы или модули с помощью соответствующего предложения uses.

Вернемся к рассматриваемому в примере 18.8 моделированию стека. Простоты ради повторим еще раз:

Пример 19.1:

Здесь лишь еще раз рассматривается пример 18.8. С помощью операций popup и pushdown формируется простой связанный список. Поскольку стек работает по принципу LIFO (последним пришел первым обслужен), помещение в стек (pushdown) и выборка из стека (popup) выполняется всегда с начала списка.

```
program stack_simulation;
uses crt;
const stackmax = 10;
type stackprt     = ^stackelement;
     datentyp     =  integer;
     stackelement = record daten : datentyp;
                           next  : stackprt end;
var st             : stackprt;
    fertig         : boolean;
    stacksize, d   : integer;
(*  stacksize является глобальной переменной для всех
    процедур *)

procedure printstack(var st:stackprt);
begin
   if st = nil then write('    Конец стека ')
             else begin
                    writeln('| ',st^.daten:4,'|');
                    writeln('└────┘ ');
                    printstack(st^.next)
                  end;
end;  (* Распечатка содержимого стека *)

procedure push(var st:stackprt; d:datentyp);
var hilf:stackprt; c:char;
begin
   new(hilf); hilf^.daten := d;
   hilf^.next := st;
   st := hilf;
```

```
   stacksize := stacksize + 1;
end;   (* Поместить в стек *)

procedure pop(var st:stackprt; var d:datentyp);
var hilf:stackprt;
begin
   hilf := st; st := st^.next;
   d := hilf^.daten;
   dispose(hilf);
   stacksize := stacksize - 1;
end; (* Выборка из стека с удалением выбранного элемента*)

procedure create(var st:stackprt);
begin
   st := nil;
   stacksize := 0;
end;  (* Формирование *)

function isfull(st:stackprt):boolean;
begin
    isfull := stacksize >= stackmax;
end;

function isempty(st:stackprt):boolean;
begin
   isempty := st = nil;
end;

procedure menue;
var c:char;
begin
   clrscr;
   fertig := false;
   printstack(st);
   gotoxy(40,5);
   writeln('Что Вы хотите:');
   gotoxy(40,7);
   writeln('+ = push down');
   gotoxy(40,8);
   writeln('- = pop up');
   gotoxy(40,9);
   writeln('e = конец');
   gotoxy(40,10);
   readln(c);

   case c of
```

```
        '+': if isfull(st)
                 then
                 begin
                    gotoxy(40,20);
                    write('К сожалению,');
                    write('стек полон');
                    readln(c);
                 end
                 else begin
                        gotoxy(40,12);
                        writeln('Введите данные:');
                        gotoxy(40,13);
                        readln(d);
                        push(st,d);
                      end;
      '-': if isempty(st)
               then
               begin
                  gotoxy(40,20);
                  writeln('К сожалению, стек пуст');
                  readln(c);
               end
               else
               begin
                  pop(st,d);
                  gotoxy(10,1);
                  writeln('Извлечь из стека: ',d:4);
                  readln(c);
               end;

      'e','E':fertig := true;   end;  (* Выбор *)
   end;   (* Меню *)

   begin (******** Исполняемая часть *********)
      create(st);
      repeat
         menue;
      until fertig;
   end.
```

Разобьем программу на модули:

TYPEN.PAS будет содержать все типы данных, константы и переменные:

```
   unit typen;
   interface
```

```
const stackmax = 10;
type stackprt     = ^stackelement;
     datentyp     = integer;
     stackelement = record daten : datentyp;
                           next  : stackprt end;
var st            : stackprt;
    fertig        : boolean;
    stacksize, d  : integer;

implementation
begin end.
```

Модуль EA.PAS содержит меню операций ввода/вывода, считывания чисел, вывода содержимого стека. Модуль EA использует типизованные переменные и следующий модуль STACKOP.

```
unit ea;
interface
uses typen, stackop,crt;
procedure printstack(var st:stackprt);
procedure create(var st:stackprt);
procedure menue;
var c:char;
implementation
procedure printstack(var st:stackprt);
begin
  if st = nil then write('    Конец стека')
              else begin
                     writeln('│',st^.daten:4,'T');
                     writeln('└────┘');
                     printstack(st^.next)
                   end;
end;  (* Распечатка содержимого стека *)

procedure create(var st:stackprt);
begin
  st := nil;
  stacksize := 0;
end;  (* Формирование *)

procedure menue;
var c:char;
begin
   clrscr;
   fertig := false;
   printstack(st);
```

```
  gotoxy(40,5);
  writeln('Что Вы хотите:');
  gotoxy(40,7);
  writeln('+ = push down');
  gotoxy(40,8);
  writeln('- = pop up');
  gotoxy(40,9);
  writeln('e = конец');
  gotoxy(40,10);
  readln(c);
  case c of
    '+': if isfull(st)
                then
                begin
                   gotoxy(40,20);
                   write('К сожалению,');
                   write('стек полон');
                   readln(c);
                end

                else begin
                       gotoxy(40,12);
                       writeln('Введите данные:');
                       gotoxy(40,13);
                       readln(d);
                       push(st,d);
                     end;
 '-': if isempty(st)
            then
            begin
               gotoxy(40,20);
               writeln('К сожалению, стек пуст');
               readln(c);
            end
            else
            begin
               pop(st,d);
               gotoxy(10,1);
               writeln('Извлечь из стека: ',d:4);
               readln(c);
            end;

 'e','E':fertig := true;   end;  (* Выбор *)
end;   (* Меню *)

begin end.
```

Модуль STACKOP.PAS содержит операции со стеком pushdown и popup. Здесь используется модуль typen.

```
unit stackop;
interface
uses typen;
procedure push(var st:stackprt; d:datentyp);

procedure pop(var st:stackprt; var d:datentyp);

function isfull(st:stackprt):boolean;

function isempty(st:stackprt):boolean;

implementation
procedure push(var st:stackprt; d:datentyp);
var hilf:stackprt; c:char;
begin
   new(hilf); hilf^.daten := d;
   hilf^.next := st;
   st := hilf;
   stacksize := stacksize + 1;
end;   (* Поместить в стек *)

procedure pop(var st:stackprt; var d:datentyp);
var hilf:stackprt;
begin
   hilf := st; st := st^.next;
   d := hilf^.daten;
   dispose(hilf);
   stacksize := stacksize - 1;
end; (* Выборка из стека с удалением выбранного элемента*)

function isfull(st:stackprt):boolean;
begin
    isfull := stacksize >= stackmax;
end;

function isempty(st:stackprt):boolean;
begin
   isempty := st = nil;
end;

begin end.
```

При компиляции из приведенных выше текстов на Паскале должны быть сгененированы три модуля TYPEN.TPU, STACKOP.TPU и EA.TPU (именно в такой последовательности!).

Тогда вся программа сократится до

```
program stack (input,output);
uses crt, typen, ea, stackop;

begin (*********** Исполняемая часть ************)
   create(st);
   repeat
     menue;
   until fertig;
end.
```

Если Вы написали важный для Вас модуль, может оказаться полезным включить его в стандартную библиотеку TURBO.TPL, что сразу же избавит Вас от необходимости заботиться о размещении этого модуля в нужном каталоге. Для этого существует программа TPUMOVER, которая может помещать модули .TPU в библиотеку TURBO.TPL и удалять их из библиотеки, когда она станет слишком большой.

В разделе 11.1 говорилось об использовании допустимых имен. Сюда же относятся понятия локальных и глобальных имен. Все сказанное можно перенести и на модули. Поясним это на следующем небольшом примере:

```
unit a;
interface
const kennung = 'unit a';
implementation
begin
end.

program demo;
uses a;
const kennung = 'demo';
begin
   writeln(kennung);          Выводится: demo
end.
```

Модуль a определяет имя kennung, которое заново определяется в программе. Тогда в исполняемой части программы kennung выступает в качестве локального имени. Имя kennung из модуля a является для программы глобальным и заменяется тем же самым локальным именем. Итак, если в некоторой программе случайно

будет использовано имя из стандартного модуля, это имя в стандартном модуле потеряет для программы силу (будет определено заново). В отличие от вложенных блоков для процедур, глобальные величины из модуля доступны программе. Имя можно задать также следующим образом:

квалификатор.имя

А квалификатором может быть имя модуля. Итак, в приведенном выше примере можно было бы обратиться к определенному в а имени kennung:

```
program demo;
uses a;
const kennung = 'demo';
begin
  writeln(kennung);        Выводится: demo
  writeln(a.kennung);      Выводится: unit a
end.
```

Остается вопрос, играет ли роль последовательность перечисления модулей в предложении uses, то есть равнозначны ли

```
program demo;
uses a, b, c, d;
```

и

```
program demo;
uses a, c, d, b;
```

Пример 19.2:

Продемонстрируем вложение модулей:

```
unit a;
interface
const kennung = 'unit a';
implementation
begin
end.

unit b;
interface
const kennung = 'unit b';
implementation
```

```
begin
end.

unit c;
interface
const kennung = 'unit c';
implementation
begin
end.

program demo;
uses a, b, c;  (* См. рис. 19.1 *)
const kennung = 'demo';
begin
  writeln (kennung);      (*—->demo *)
  writeln (a.kennung);    (*—->unit a *)
  writeln (b.kennung);    (*—->unit b *)
  writeln (c.kennung);    (*—->unit c *)
end.
```

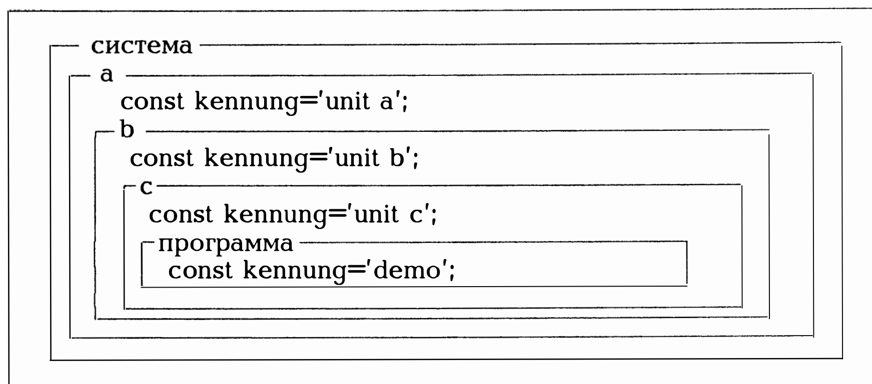


Рис. 19.1. Вложение модулей согласно примеру 19.2

Модули будут компоноваться в той последовательности, в которой они перечисляются в uses. Тогда они окажутся вложенными так, как это показано на рис. 19.1. Каждый внутренний модуль отменяет объект с тем же именем. При следующей конструкции

```
unit a;
interface
const kennung = 'unit a',
implementation
begin
end.
```

```
unit b;
interface
uses a;
const kennung = 'unit b';
implementation
begin
end.

program demo;
uses b;
begin
  writeln (kennung);      (*—->unit b *)
  writeln (a.kennung);    (*—->unit a *)
end.
```

модуль a вложен в модуль b, а тот в свою очередь - в программу demo.

19.3. Стандартные модули

Что такое модуль и как написать собственный модуль, описано в предыдущем разделе 19.2. Здесь же рассмотрим имеющиеся стандартные модули. Вообще говоря, существует семь стандартных модулей, а именно: SYSTEM, CRT, GRAPH, DOS, PRINTER, GRAPH3, TURBO3. В них определены сотни имен констант, типов, переменных, процедур и функций. На установочной дискете Турбо Паскаля находится несколько файлов с атрибутами .DOC, например, CRT.DOC. Затем следует интерфейсная часть модуля crt, следовательно, общедоступная часть модуля. Крайне желательно, чтобы такие файлы можно было выводить один раз. Проще всего получить информацию через стандартные модули, для чего существует вызываемый при нажатии клавиш Ctrl-F1 редактор и меню, откуда с его помощью можно выбрать соответствующий пункт.

В файле TURBO.TPL (TPL = Turbo Pascal-Library) содержится семь стандартных модулей.

Модуль SYSTEM.TPU

Это важнейший модуль. Он содержит основные стандартные процедуры и функции Турбо Паскаля. На такой модуль не требуется ссылки через uses system (да она и невозможна). Модуль автоматически связывается с любой программой.

Модуль CRT.TPU

содержит необходимые для эффективного ввода и вывода процедуры, обеспечивающие прямой контроль через экран, клавиатуру и динамик.

assigncrt	ставит в соответствие экрану текстовый файл
clrscr	стирает содержимое экрана или экранного окна и устанавливает курсор в левый верхний угол экрана
clreol	стирает все символы, начиная с позиции, в которой находится курсор, вплоть до правой границы окна
delay(n)	на n миллисекунд задерживает выполнение программы
delline	стирает строку, в которой находится курсор, и перемещает курсор на одну строку вверх
insline	в позицию, в которой находится курсор, добавляет пустую строку и перемещает курсор на одну строку вниз
gotoxy(s,z)	устанавливает курсор в позицию s строки z. Левым верхним углом окна будет (1,1)
readkey	считывается один символ из input
keypressed	генерирует значение true, если клавиша нажата, и false в противном случае
highvideo	устанавливает цвет символа на "подчеркивание"
lowvideo	устанавливает цвет символа на половинную яркость
normvideo	устанавливает цвет символа на нормальную яркость
textbackground	устанавливает цвет фона для последующего вывода текста
textcolor	устанавливает цвет символа для последующего вывода текста
textmode	устанавливает режим отображения для текста
window	определяет текстовое окно на экране
wherex	возвращает позицию (столбец), в которой находится курсор
wherey	возвращает позицию (строку), в которой находится курсор, относительно верхней кромки окна
sound(n)	генерирует звуковой сигнал частотой n Гц
nosound	отключает звуковой сигнал

Рис. 19.2. Процедуры и функции, содержащиеся в модуле CRT.TPU

Процедура assigncrt при инициализации crt вызывается следующим образом:

```
assigncrt(input); reset(input);
assigncrt(output); rewrite(output);
```

в результате чего стандартные текстовые файлы input и output назначаются не стандартному устройству CON, а специальному

драйверу в CRT. В версии 3.0 Турбо Паскаля многие такие процедуры были также стандартными.

Приведенных на рис.19.2 кратких сведений, пожалуй, достаточно, так как при работе с редактором можно с помощью клавиш Ctrl-F1 получить более подробную информацию обо всех стандартных процедурах и функциях. Впрочем, следует привыкнуть включать uses crt; во все программы. С помощью процедур sound, delay и nosound, как показано в примере 11.5, можно генерировать специальные звуковые сигналы.

Из практических соображений приведем еще несколько стандартных переменных из CRT.TPU:

```
var checkbreak,checkeof:boolean;
```

checkbreak обычно устанавливается на true, то есть при нажатии клавиш Ctrl-Break программа при следующем вводе или выводе текста прерывается. Если задать явно

```
checkbreak := false;
```

контроль по прерываниям не производится.

checkeof установлена по умолчанию на false, то есть ввод Ctrl-Z не воспринимается как конец файла. Лишь после

```
checkeof := true;
```

Ctrl-Z будет вновь восприниматься как конец файла (см. пример 11.5).

Модуль DOS.TPU

является интерфейсом с операционной системой MS-DOS. Он содержит процедуры обработки прерываний (intr) и вызовы функций (msdos), процедуры для работы с датой и временем суток (getdate, gettime, setdate, settime), функции опроса состояния накопителей на гибких дисках и жестких дисков (disksize,diskfree), процедуры обработки записей файла (getfattr, findfirst, findnext), а также процедуры и функции для процессов (dosexitcode, execute, keep). Многие из перечисленных процедур и функций более точно описываются в главе 21.

Модуль PRINTER.TPU

описывает текстовый файл lst и связывает его с устройством LPT1, то есть печатающим устройством. Используя названия lst в операторах write и writeln, можно выводить результаты работы программы на печать более простым способом, что демонстрирует приведенный ниже небольшой пример.

```
prigram drucker;
uses printer;
var x : integer;
begin
  readln(x);
  writeln(lst,'Значение x:',x:6);
end.
```

Модуль TURBO3.TPU

служит прежде всего для обеспечения совместимости с версией 3.0 Турбо Паскаля. В нем описываются типовые для версии 3.0 имена переменных и процедур. Следует подчеркнуть, что предполагается использование модуля CRT. Итак, нужно писать

```
uses crt, turbo3;
```

Модуль GRAPH3.TPU

В версии 3.0 стандартные графические программы описывались в файле GRAPH.P, который добавлялся в программу с помощью директивы (*$I GRAPH.P*). Теперь имеется модуль GRAPH3.TPU, который содержит и графику версии 3.0, использующую только относительные команды. Этот модуль предполагает подключение CRT.TPU:

```
uses crt,graph3;
```

Модуль GRAPH.TPU

содержит пакет более чем 50 стандартных графических программ. Сюда же относятся графические драйверы в файлах .BGI (BGI = Borland Graphics Interfaces - графический интерфейс фирмы Borland) для наиболее распространенных видеоадаптеров. Более подробное описание Вы найдете в главе 22.

В заключение еще раз сошлемся на программу TPUMOVER.EXE, с помощью которой можно работать с библиотекой TURBO.TPL, добавляя или удаляя из нее модули.

Примечания относительно версии 5.0

Модуль OVERLAY.TPU

В версии 5.0 на уровне модуля существует понятие оверлея, который будет рассматриваться в разделе 19.4.

19.4. Оверлейные блоки (версия 5.0)

Может случиться так, что большая программа не помещается в оперативной памяти. Классическим выходом из этой ситуации является разбиение объектных кодов на части и размещение их в оверлейных блоках, загружаемых в память лишь по мере необходимости. Итак, оверлейные блоки используют часть оперативной памяти совместно. Для версии 3.0 Турбо Паскаля процедуры и функции могли считаться оверлейными блоками. Для версии 4.0 оверлейные блоки не предусмотрены. В версии 5.0 оверлейные блоки появились вновь. Причем обрабатываются они на уровне модулей (units), т.е. наименьшей оверлейной структурой является модуль. Опишем вначале, как из модуля получается оверлейный блок. Рассмотрим пример:

```
unit das;
(*O+*)  (*F+*)
interface
procedure das_proc;
implementation
procedure das_proc;
begin
writeln('Здесь das_proc из модуля das');
end;
end.
```

При компиляции обе директивы $F и $O должны стоять на +. (*$F+*) означает, что при вызове стандартной программы и выходе из нее этот модуль может выходить за границы сегмента (far). Директива (*$O*), то есть Options/Compile Overlays allowed ON, позволяет компилятору выполнить известные дополнительные проверки. Говоря яснее, оттранслированный с использованием директивы (*$O+*) модуль может позднее использоваться как оверлейный блок, но может быть также использоваться как обычный модуль с помощью uses ... . Точно также можно написать

```
unit dies;
(*O+*)  (*F+*)
interface
procedure dies_proc;
implementation
procedure dies_proc;
begin
writeln('Здесь dies_proc из модуля dies');
end;
end.
```

Используем оба эти модуля в программе OVER.PAS:

```
program over;
(*F+*)  (*O+*)
uses overlay, crt, dies, das;
(*O dies*)
(*O das*)
begin
ovrinit('over.ovr');
if ovrresult <> ovrok then
    begin
      writeln('Ошибка в оверлейной структуре',ovrresult:4);
      halt
    end;
dies_proc;
das_proc;
end.
```

После uses первым следует назвать стандартный модуль OVERLAY, затем в любой последовательности могут следовать другие используемые модули. Используемые в качестве оверлейных блоков модули следует включить в директиву компилятору (*$O имя_модуля*). Тогда компилятор сгенерирует два файла OVER.EXE и OVER.OVR, в которые включит все объявленные оверлейными модули (в нашем случае это dies.TPU и das.TPU). Содержащиеся в модуле OVERLAY стандартные программы показаны на рис. 19.3. Первым должен вызываться overinit, инициализирующий управление оверлеями. В оверлейных блоках имеется переменная

```
var ovrresult:integer;
```

которая после вызова из overlay (аналогичной ioresult) стандартной программы получает некоторое значение (0 = ovrok при безошибочном выполнении или значение < 0 при возникновении ошибки).

ovrinit создает некий оверлейный буфер, который имеет размер не меньший размера наибольшего модуля из .OVR. Буфер размещается непосредственно за динамической областью.

При обращении к стандартной программе из оверлейного блока проверяется, есть ли такая программа в буфере. Если нет, она загружается туда (а если места для размещения стандартной программы недостаточно, одна из находящихся в буфере программ выгружается из памяти). С помощью oversetbuf можно увеличить буфер, чтобы одновременно размещать в буфере как можно больше оверлейных блоков. Если Ваш компьютер оснащен платой рас-

ширения памяти EMS (EMS = Expanded Memory Specification), можно разместить оверлейный буфер и там (с помощью ovrinitems). Тогда экономится время на обращении к дискете.

Вызов:	overinit(имя_файла)	Процедура	overlay
Параметры:	имя_файла:string;		
Действие:	Инициализирует управление оверлейными блоками и открывает файл имя_файла, содержащий объявленные оверлейными блоками модули. overinit ищет указанный файл в текущем каталоге. Создает оверлейный буфер, размер которого соответствует размеру наибольшего оверлейного модуля (а динамическая область сдвигается соответственно вверх). overinit следует вызывать до вызова других стандартных программ.		
Вызов:	oversetbuf(size)	Процедура	overlay
Параметры:	size:longint;		
Действие:	Оверлейный буфер имеет длину в size байтов. Начало динамической области соответствующим образом сдвигается!		
Вызов:	ovrgetbuf	Процедура	overlay
Параметры:	нет		
Действие:	Значением функции является текущий размер оверлейного буфера в байтах.		
Вызов:	ovrclearbuf	Процедура	overlay
Параметры:	нет		
Действие:	Из буфера удаляются все загруженные в оверлейный буфер модули.		
Вызов:	ovrinitems	Процедура	overlay
Параметры:	нет		
Действие:	Проверяется, имеет ли система плату расширения памяти EMS, которой достаточно для размещения оверлейного файла .OVR. Соответственно, указанный файл загружается туда. Тогда нет необходимости обращаться к дискетам.		

Рис. 19.3. Стандартные программы из модуля OVERLAY

Следует рассмотреть случай, когда программа использует динамические переменные (см. раздел 18.2) в динамической области. Прежде чем их разместить, нужно окончательно установить размер оверлейного буфера. В заключение отметим, что стандартные модули CRT, GRAPH, TURBO3 и GRAPH3 нельзя сделать оверлейными. Не могут быть оверлейными и самостоятельно написанные

модули, использующие стандартные программы обработки прерываний.

19.5. Компоновка

Для больших программ целесообразно действовать по старой римской пословице "Divide et impera" (Разделяй и властвуй). Задача раскладывается на подзадачи, которые оформляются в виде модулей. В результате вся программа становится не только более надежной и обозримой, но и появляется возможность одновременной работы над программой нескольких человек. Пример 18.8 был построен по этому принципу.

В задаче "моделирование стека" все константы, типы данных и глобальные переменные были объединены в модуле TYPEN.TPU, все операции ввода и вывода в модуле EA.TPU, а все операции над смоделированным стеком в модуле STACKOP.TPU. Тогда сама программа STACK.PAS стала более короткой и легко читаемой. При этом были установлены следующие зависимости:

```
STACK.PAS          uses crt, typen, ea, stackop;
STACKOP.TPU        uses typen;
EA.TPU             uses crt,typen,stackop;
TYPEN.TPU          uses нет
```

Эти зависимости можно представить в виде дерева (рис.19.4). Причем стандартный модуль crt пропущен.

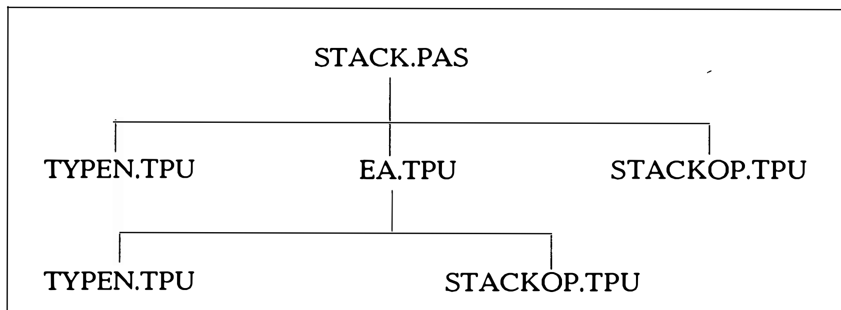


Рис. 19.4. Связь между модулями из примера 18.8

При таком подходе возникает одна довольно серьезная трудность: Как обеспечить, чтобы узлы такого дерева всегда имели актуальное состояние? Если кто-то изменит что-либо в модуле STACKOP.PAS, нужно будет заново скомпилировать STACKOP.PAS в STACKOP.TPU, и точно также EA.TPU должен

быть заново сформирован путем компиляции EA.PAS. Наконец, придется заново скомпилировать и STACK.PAS.

Итак, если сразу несколько человек работают с модулями, являющимися узлами представленного выше дерева, нужно будет вести учет времени формирования каждого узла (то есть модуля). Ни один узел не может иметь более позднюю дату возникновения, чем лежащие выше узлы. В Турбо Паскале пункт меню Compile/Make предоставляет механизм, который позволяет автоматически отслеживать эти временные связи и заново компилировать соответствующие модули. Функция Compile/Make (для версии 5.0) имеет следующий результат:

- Если с помощью Compile/Primary File создан некий "головной файлэ, он используется для компоновки, в противном случае берется записанный в редакторе исходный текст. Если он содержит предложение uses xxx, проверяется, существует ли XXX.TPU с более старой датой, чем XXX.PAS. Если это так, из XXX.PAS заново компилируется XXX.TPU.
- Если интерфейсная часть модуля изменена, заново компилируются все модули, которые ее используют.
- Если модуль содержит в себе директиву включения (*I имя_файла*), а файл с указанным именем имеет более позднюю дату создания, чем сам модуль, он также компилируется заново.
- Если в некоторый модуль с помощью директивы (*L*) включается некоторый объектный файл .OBJ, который создан позже модуля, модуль компилируется заново.

Естественным условием применения функции Compile/Make является наличие исходного текста модуля. Для стандартных модулей в TPL сравнение даты и времени создания файлов не предусмотрено.

Тот, кто хочет быть абсолютно уверен в правильности результата, может воспользоваться функцией Compile/build. При этом даты и время создания связанных между собой модулей не сравниваются, а просто заново компилируются все используемые программой модули.

Обычно используется программа MAKE.EXE, которая осуществляет такое управление разработкой программных средств, она описана в руководстве пользователя [1].

20. Командные строки TPC

В версиях 4.0 и 5.0 Турбо Паскаля наряду с интегрированным развитым программным окружением имеется и некая версия интерпретатора командных строк TPC, с помощью которого можно вызывать компилятор на уровне MS-DOS. Интерпретатор вызывается следующим образом:

C>TPC опции имена_файлов опции

Разнообразные опции позволяют выбирать интегрированное программное окружение через Options/Compiler или Compile. Использование TPC.EXE аналогично возможностям использования Паскаля под UNIX. Для UNIX командная строка записывается в виде:

$ команда опции имена_файлов

Знак доллара является для UNIX идентификатором командной строки. Команда описывает, что нужно выполнить, имена файлов указывают, над какими файлами должны выполняться эти операции, а необязательные параметры, начинающиеся с дефиса -, - как эти операции должны выполняться. Команда "Вызов компилятора Паскаляэ записывается как pc, а типичная строка обращения к компилятору может иметь, например, такой вид:

$ pc -f -o prog prog.p

то есть вызывается компилятор Паскаля, исходный текст находится в файле prog.p, генерируемый объектный файл (-o) должен иметь имя prog, -f должен эмулировать выполнение арифметических операций с плавающей запятой.

Обычно (из-за встроенного редактора и имеющегося в версии 5.0 отладчика) программу на Паскале пишут с использованием программного окружения TURBO. Тогда готовая программа на Паскале может обрабатываться аналогично тому, как это делается в UNIX, с помощью командной строки TPC:

C>TPC опции имена_файлов опции

В UNIX опции могут начинаться с - или /. При использовании дефиса опции параметры должны отделяться друг от друга пробелом, при использовании косой черты опции могут разделяться пробелами. Причем параметры могут стоять справа и слева от имен файлов по выбору пользователя.

Если исходная программа записана в файл PROG.PAS, проще всего вызвать ее строкой

C:TPC PROG

Если имя файла не имеет атрибутов, по умолчанию принимается расширение .PAS. Приведенная выше строка позволяет скомпилировать исходный текст и скомпоновать работоспособную программу, помещаемую в PROG.EXE.

Список возможных опции выводится, если запустить TPC без указания параметров и имен файлов:

C>TPC

Параметр	Режим в основном меню
/$B+	Options/Compiler/Boolean evaluation Полная (+) или краткая (-) оценка логического выражения.
/$D+	Options/Compiler/Debug information Ввести или вывести информацию для отладчика.
/$F+	Options/Compiler/Force far calls Указатель FAR установить (+) или отменить (-).
/$I+	Options/Compiler I/Ochecking Проверку ввода/вывода включить (+) или отключить (-).
/$L+	Options/Compiler/Link buffer.. Память (+), диск (-).
/$Mxxx	Options/Compiler/Memory sizes
/$N+	Options/Compiler/Numeric processing С 8087 (аппаратные средства (+), программное обеспечение (-)).
/$R+	Options/Compiler/Range checking... On(+),Off(-).
/$S+	Options/Compiler/Stack checking... On(+),Off(-).
/$T+	Options/Compiler/Turbo Pascal map file generation...On(+)Off(-)
/$V-	Options/Compiler/Var_string checking... Включить (+) или отключить (-) проверку на длину строк
/B	Compile/Build
/Dxxx	Options/Compiler/Conditional defines
/Exxx	Options/Directories/Executable directory
/Fxxx	Compile/Find error
/M	Compile/Make
/Oxxx	Options/Directories/Object directories
/Q	Не в основном меню (только компиляция без выдачи сообщения о номере строки).
/Rxxx	Compile/Destination...memory (Только для версии 4.0). Options/Parameters Run
/Txxx	Options/Directories/Turbo directories
/Uxxx	Options/Directories/Unit directories
/Xxxx	Compile/Destination...Disk (Только для версии 4.0) Options/Parameters Run

Рис. 20.1. Опции командной строки

Опции влияют на ход компиляции и компоновки. Они выбираются в интегрированном программном окружении через режимы основного меню. Опции являются директивами (см. приложение D), выбираемыми через Options/Compiler или помещаемыми в исходном тексте программы.

При отсутствии опции по умолчанию берутся значения, установленные в основном меню Options/Compiler. Например, возьмем

```
program menu;             Имя исходного файла NN.PAS
var k,i : 1..10;
begin
  i := 2;
  k := 10*i;
  writeln(i:8,k:8);
end.
```

Запустим ее с помощью

C:TPC -X NN —> 2 20 , поскольку по умолчанию берется $R-

Запустим с помощью командной строки

C:TPC -X -$R+ NN —> Ошибка по времени исполнения при k:=10*i

Если вызов TPC содержит несколько опций, можно перечислить их через косую черту /:

C:TPC /$R-/$I-/$V-/$F+ имя_файла

или просто через запятую:

C:TPC/$R-,I-,V-,F+ имя_файла

Если опции вводятся с дефисом -, они всегда должны разделяться пробелом:

C:TPC -$R- -$I- -$V- $F+ имя_файла

Если программа записана не в том же каталоге, что и TPC, нужно, естественно, задать полное имя пути доступа:

C:TPC -X A:\KAT\PROG.PAS

Опции, указанных на рис. 20.1 внизу, являются параметрами применительно к Compile (/B/Q/F/M) или к каталогам (/E/I/O/T/U) или касаются выполнения программы (/R/X).

TPC содержит собственный компилятор и компоновщик. Но ему необходима библиотека TURBO.TPL (а может быть, и TPC.CFG). Если TPC запущена, в текущем каталоге ведется поиск этих двух файлов. Если TURBO.TPL (и TPL.CFG) находится в каталоге C:\TURBO4, можно указать на это с помощью параметра /T:

C:TPC -X -TC:\TURBO4 PROG

Может возникнуть ситуация, когда TPC всегда должна запускаться с одними и теми же параметрами. Тогда можно избежать задания их каждый раз. Параметры можно поместить в файл TPC.CFG и ссылаться на них с помощью /T. При использовании /R и /X можно задавать так называемые параметры вызова: /R"par1 par2 par3" или /X"par1 par2 par3".

Для того, чтобы указанные в вызове параметры включить в программу, имеются стандартные функции paramcount и paramstr, представленные на рис. 20.2.

Вызов:	paramcount	Значение функции:word
Параметры:	нет	
Действие:	Значением функции является число указанных в строке вызова параметров.	
Вызов:	paramstr(i)	Значение функции:string
Параметры:	i:word;	
Действие:		

Рис. 20.2. Функции для параметров вызова

Пример 20.1:

Протокол параметров вызова:

```
program aufrufparameter_protokollieren;
uses crt;
var i : integer;
begin
  writeln('Число параметров вызова: ',paramcount);
  for i := 1 to paramcount do
    writeln(paramstr(i));
end.
```

Если программа из примера 20.1 имеет имя PROT.PAS, вызов

```
C>TPC -x"dies und das"PROT     (только для версии 4.0)
```

приведет к выводу

```
Число параметров вызова:3
dies
und
das
```

Обычно параметры вызова можно задавать и для интегрированного программного окружения. Для этого используется предоставляемое функцией Options/Parameters окно, в которое можно ввести параметры вызова, доступ к которым обеспечивается функциями paramcount и paramstr.

В примере 17.4 текстовый файл quelle копировался в выходной файл zeil, причем из программы считывались имена входного и выходного файлов. В приведенном ниже примере имена обоих файлов задаются в вызове.

Пример 20.2:

```
program text_file_copieren;
var quelle,ziel:text;
    qname,zname:string[20];

procedure copy(var q,z:text);
var c:char;
begin
  while not eof(q) do
   begin
     read(q,c);
     (* writeln(ord(c):4);
        Номер считанного символа в таблице кодов ASCII ->
        Bildschirm *)
     write(z,c);
   end;
 close(q); close(z);
end;

begin
  if paramcount <> 2
     then writeln('Вызываются: prog_name quelle ziel')
     else begin
           assign(quelle,paramstr(1));
           assign(ziel,paramstr(2));
           reset(quelle); rewrite(ziel);
```

```
                copy(quelle,ziel);
              end;
      end.
```

Если эта программа имеет имя KOPIE.PAS, нужно было бы осуществлять вызов с помощью

```
C>TPC -x"quelle.t ziel.t"KOPIE
```

Примечания относительно версии 5.0

Развитие в этой версии интегрированных программных средств по сравнению с версией 4.0 соответствует расширению возможностей задания опций в командной строке TPC. В дополнение к уже приведенным на рис. 20.1 опциям могут использоваться и опции, указанные на рис. 20.3.

Параметр	Режимы в основном меню
/$E+	Options/Compiler/Emulation..On(=)Off(-) (см. раздел 5.2).
/$O+	Options/Compiler/Overlays allowed..On(+)Off(-)
/Exxx	Options/Directories/EXE и каталог TPU
/GD	Options/Linker/Map file/Detailed Генерирует файл MAP с именами и адресами всех имен программы (при условии /$D+ или /$L+).
/GP	Options/Linker/Map file/Publics Генерирует файл MAP с именами и адресами всех глобальных имен программы (при условии /$D+ или /$L+).
/GP	Options/Linker Map file/Segments Генерирует файл MAP с теми же именами, что и .EXE. Файл содержит имена, величины и адреса сегментов программы.

Рис. 20.3. Дополнительные опции версии 5.0

21. Турбо Паскаль и MS-DOS

Если Турбо Паскаль работает в среде MS-DOS, возникает вопрос, какие функции MS-DOS можно вызывать из написанной на Паскале программы. Для этого в библиотеке Паскаля TURBO.TPL, в частности, в DOS.TPU, имеется множество констант, типов данных, процедур и функций. В этой главе дается лишь краткий обзор важнейших аспектов предоставляемых Турбо Паскалем возможностей. От читателя здесь требуется достаточное знание MS-DOS.

Операционная система MS-DOS предоставляет многочисленные служебные функции, которые можно вызвать с помощью команд. Было бы логичным, чтобы команды Турбо Паскаля оформлялись в виде стандартных процедур. В ряде случаев это действительно так. На рис. 21.1 приведено несколько таких команд. Другие будут описаны позже.

Вызов:	mkdir(s)	Процедура
Параметры:	s:string	
Действие:	Создает подкаталог с именем s. s может быть наименованием дисковода или путем доступа к файлу. Подкаталог s может эти данные и не содержать.	
Вызов:	rmdir(s)	Процедура
Параметры:	s:string	
Действие:	Стирает подкаталог с именем s.s может быть наименованием дисковода или путем доступа к файлу. Если подкаталог s не пуст или если он является текущим каталогом, возникает ошибка исполнения.	
Вызов:	chdir(s)	Процедура
Параметры:	s:string	
Действие:	Создает заново существующий актуальный каталог. s может содержать наименование дисковода. Изменение осуществляется относительно текущего каталога, причем тогда, когда s начинается с \.	
Вызов:	getdir(lw,s)	Процедура
Параметры:	lw:byte; var s:string	
Действие:	Определяет текущий каталог дисковода lw и размещает его в строку s. Для lw 0 = текущий каталог 1 = дисковод A 2 = дисковод B и т.д.	

Рис. 21.1. Некоторые команды DOS в качестве процедур

Пример 21.1:

```
program directories;
var d : byte;
    name,dir : string[64];

begin
    getdir(0,name);
    writeln(name);
    write('Новый каталог: ');
```

```
    readln(dir);
    (*$I-*) mkdir(dir); (*$I+*)
    if ioresult <> 0
       then begin
              write('Этот каталог ', dir);
              writeln( ' уже существует.')
            end
       else begin
              chdir(dir);
              getdir(0,name);
              writeln('Новый каталог: ',name);
              chdir('..');
            end;
    rmdir(dir);
    getdir(0,name);
    writeln('Новый каталог: ',name);
end.
```

Прежде чем приступить к подробному описанию функций MS-DOS перечислим регистры и их назначение (см. рис. 21.2). Все регистры имеют длину 16 бит. Регистры *x подразделяются на два байта: *h - левый байт (старший), *l - правый байт (младший).

Наименование	Содержимое
ax (ah,al)	Общего назначения
bx (bh,bl)	"
cx (ch,cl)	"
dx (dh,dl)	"
	Регистры сегментов:
ds	сегмент данных
es	дополнительный сегмент
cs	кодовый сегмент
ss	сегмент стека
	Специальные регистры:
bp	указатель базы (для индекса стека)
di	индекс приемника (назначения)
si	индекс источника
sp	указатель стека

Рис. 21.2. Регистры процессора 8086

Для использования этих регистров в стандартном модуле DOS.TPU имеется тип данных registers (рис. 21.3). Здесь речь идет о вариантной записи, чтобы иметь возможность по выбору опрашивать 16-битовые регистры ax, bx, cx, dx, а также отдельные байты ah, al и пр. Поскольку в типе данных registers регистры ss и sp не

содержатся, с помощью приведенных ниже процедур intr и msdos нельзя обрабатывать прерывания или вызывать функции DOS, использующие эти регистры.

```
type registers =  record
                  case integer of
                  0: (ax, bx, cx, dx, bp, si, di, es,
                     flags : word);
                  1: (al, ah, bl, bh, cl, ch, dl,
                     dh : byte);
                  end;
```

Рис. 21.3. Тип данных registers в модуле DOS

Функции MS-DOS вызываются через прерывания (так называемые стандартные программы BIOS). Важнейшие из них приведены на рис. 21.4. При вызове номер прерывания должен стоять в регистре ah. Большинство из этих прерываний имеют несколько вариантов, в соответствии с тем, какие параметры стоят в регистрах al, bx, cx и dx. Эти прерывания можно использовать через процедуру intr (см. рис. 21.5).

Прерывание	Результат
$5	Распечатка содержимого экрана
$10	Ввод/вывод визуальной информации Управление экраном для текстов и графики
$11	Проверка внешних устройств (подключения устройств)
$12	Размер памяти
$13	Ввод/вывод на дискету для прямого чтения/записи на дискету
$16	Ввод с клавиатуры
$17	Печатающее устройство
$19	Горячий запуск
$21	Вызов функции
$1A	Дата и время по часам

Рис. 21.4. Прерывания BIOS

Вызов:	intr(intnr,regs)	Процедура dos
Параметры:	intnr : byte; var regs : registers;	
Действие:	Выполняется прерывание intnr. Переменная regs должна содержать запрошенное прерыванием intnr значение. После вызова сгенерированный прерыванием результат помещается в regs.	

Рис. 21.5. Процедура intr

Пример 21.2:

```
program interrupt_beispiel;
uses dos;
var size : integer;

procedure memory_size(var size:integer);
var    result:registers;
begin
  (* Прерывание $12 проверяет оперативную память
     и помещает размер памяти в регистр  ax *)
     intr($12,result);
     size:=result.ax;
end;

begin
   memory_size(size);
   write('Ваш компьютер имеет',size:4);
   writeln(' Кбайт оперативной памяти');
end.
```

Прерывание $21, через которое можно вызвать многие функции DOS (так называемые функции высокого уровня), имеет особое значение (см. рис. 21.6). Более подробную информацию об этом прерывании читатель может почерпнуть из Технического руководства.

Номер	Результат
$5	Вывод на печатающее устройство
$B	Ввод с клавиатуры
$19	Задание текущего дисковода
$23	Размер файла
$2A	Выбор даты
$2C	Выбор времени по часам
$30	Выбор версии DOS
$33	Проверка на <Ctrl>Break
$39	Создание подкаталога (MKDIR)
$3A	Удаление подкаталога (RKDIR)
$3D	Открытие файла
$3E	Закрытие файла
$41	Удаление файла
$47	Выбор текущего подкаталога
$4B	Загрузка и выполнение программы
$57	Выбор даты и времени создания файла/установка даты и времени создания файла

Рис. 21.6. Вызов функций с помощью прерывания DOS $21

Для вызова указанных функций через прерывание $21 существует стандартная процедура msdos, представленная на рис. 21.7.

Вызов:	msdos(regs)	Процедура	DOS
Параметры:	var regs:registers;		
Действие:	Вызывается функция DOS, номер которой записан в регистре ax (точнее в ah). В переменной regs должны храниться параметры вызова. После вызова функции результат вновь записывается в regs.		
Примечание:	Тип данных registers описан в модуле DOS (см. рис. 21.3). Вызов msdos(regs) идентичен вызову intr($21,regs).		

Рис. 21.7. Процедура msdos

Пример 21.3:

```
program freier_speicherplatz;
uses dos;
var    lw:byte;

function speicher(lw:byte):longint;
var regs:registers;
begin
 with regs do
 begin
   ax := $3600;
   (*Вызов функции для определения свободного места
     в памяти*)
   dx := lw;     (* dl - номер дисковода *)
   msdos(regs);
   (* Теперь в ax записано число сторон дискеты
      или $FFFF, если lw задает недействительный дисковод,
                  bx = число свободных секторов
                  cx = число байтов в секторе *)
   if ax = $FFFF
      then speicher := 0
      else speicher :=longint(ax)*bx*cx;
      (* Коэффициент longint(ax) позволяет выполнить умно-
         жение в формате longint;
         см. раздел 9.5 *)
 end;
end; (* память *)

begin (********Исполняемая часть*********)
 writeln('Дисковод (0=текущий, 1=A, 2=B, и т.д.):');
 readln(lw);
```

```
 write('Свободной памяти на дисководе ',lw);
 writeln(speicher(lw):8);
end.
```

Часто возникает ситуация, когда нужно обратиться к установленным в компьютере дате и времени. Сделать это можно лишь с помощью MS-DOS через функции $2A и $2C. Но и для этого есть стандартные процедуры getdate и gettime (см. рис. 21.8).

Вызов:	getdate(j,m,t,wt)	Процедура	DOS
Параметры:	var j,m,t,wt:word;		
Действие:	Процедура позволяет получить используемые MS-DOS дату в виде: год j(1980..2099), месяц m (1..12), число t (1..31) и день недели wt (0..6, 0 = воскресенье)		
Вызов:	gettime(h,m,s,s1 00)	Процедура	DOS
Параметры:	var h,m,s,s100:word;		
Действие:	Процедура возвращает текущее время суток в формате: час h(0..23), минуты m(0..59), секунды s (0..59) и сотые доли секунды s100 (0..99).		

Рис. 21.8. Выбор даты и времени по часам

Пример 21.4:

```
program datum_holen;
uses dos;
type datum_typ = record tag, monat, jahr,
                        wochentag : word
                 end;
var  datum : datum_typ;

begin
  with datum do
  getdate(jahr, monat, tag, wochentag);
  with datum do
  begin
    write('Дата: ');
    case wochentag of
      0: write('воскресенье  ');
      1: write('понедельник  ');
      2: write('вторник      ');
      3: write('среда        ');
      4: write('четверг      ');
      5: write('пятница      ');
      6: write('суббота      ');  end;
```

```
      writeln(tag:2,'.', monat:3,'.', jahr:5);
   end;
end.
```

Пример 21.5:

```
program zeit_holen;
uses dos, crt;
type zeit_typ = record stunde, minute, sekunde,
                       sek100 : word
                end;
var  zeit, zeit1, zeit2 : zeit_typ;
     i, anzahl : longint;

procedure uhrzeit(var z : zeit_typ);
begin
   with z do
   gettime(stunde, minute, sekunde, sek100);
end;

procedure uhrzeit_ausgeben(z : zeit_typ);
begin
   with z do
   begin
      write('Uhrzeit: ');
      write(stunde:2,'h :', minute:2,'m :');
      writeln( sekunde:2,'.',sek100,'s');
   end;
end;

function zeitdifferenz(var z2, z1 : zeit_typ)
         : real;
var sek2, sek1, msek2, msek1 : real;

begin
   sek2 := z2.stunde*3600.0 + 60*z2.minute +
           z2.sekunde + z2.sek100/100;
   sek1 := z1.stunde*3600.0 + 60*z1.minute +
           z1.sekunde + z1.sek100/100;
   zeitdifferenz := sek2 - sek1;
end;

begin
   clrscr;
   writeln('Определение времени счета для цикла');
   write('Время цикла: '); readln(anzahl);
```

```
  uhrzeit(zeit1);
  for i := 1 to anzahl do;
  uhrzeit(zeit2);
  uhrzeit_ausgeben(zeit1);
  uhrzeit_ausgeben(zeit2);
  write('Время счета: ');
  writeln(zeitdifferenz(zeit2, zeit1):6:3,' sec');
end.
```

Выбирать режим экрана можно двумя способами. Во-первых, с помощью управляющей последовательности, во-вторых, путем прямой записи в память дисплея. С помощью символа номер 27 по таблице кодов ASCII (escape) можно управлять внешними устройствами. Для печатающего устройства найдем управляющие последовательности в руководстве по принтеру (см. пример 17.7). Для экрана существуют так называемые последовательности стандарта ANSI, которые описаны в руководстве по MS-DOS. Они имеют вид

```
esc[параметр
```

и используются в определенном в файле ANSI.SYS драйвере, то есть в файле CONFIG.SYS должна иметься запись

```
DEVICE=ANSI.SYS
```

Этот драйвер используется в приведенном ниже примере.

Пример 21.6:

Управление экраном с помощью драйвера ANSI.

```
program ansi_treiber;
const blinken_ein = ^['[5m';
      clrscr      = ^['[2J';
      invers_ein  = ^['[7m';
      unter_ein   = ^['[4m';
      attrib_aus  = ^['[0m';
      esc         = #27;

procedure gotoxy(spalte, zeile:integer);
begin
  write(chr(27),'[',zeile,';',spalte,'H');
end;

begin
  write(clrscr);
```

```
  gotoxy(5,10);
  writeln(blinken_ein,'Привет,',attrib_aus);
  gotoxy(10, 12);
  writeln(invers_ein,'дорогие',attrib_aus);
  gotoxy(15,14);
  writeln(unter_ein, 'друзья!',attrib_aus);
end.
```

Память дисплея располагается, начиная с адреса $B000:000. Если в одном байте записан некий символ, в следующем байте хранится так называемый атрибут (задающий представление символа, например, в инверсном отображении, с мерцанием и т.д.). Для прямой записи в определенное место памяти существует стандартная переменная mem типа array of byte, то есть с помощью строки

```
mem[$0050:$0065] :=17;
```

можно записать число 17 по указанному после mem адресу.

Пример 21.7:

С помощью этой программы можно записать цепочку символов в нужное место экрана, введя некий атрибут. Номера атрибутов приводятся ниже.

Атрибуты экрана:

N бита	7	6	5	4	3	2	1	0	Значение
		0	0	0		1	1	1	Обычный (белый на черном)
		0	0	0		0	0	0	Невидимый (черный на черном)
		1	1	1		0	0	0	Инверсный (черный на белом)
		0	0	0		0	0	1	С подчеркиванием

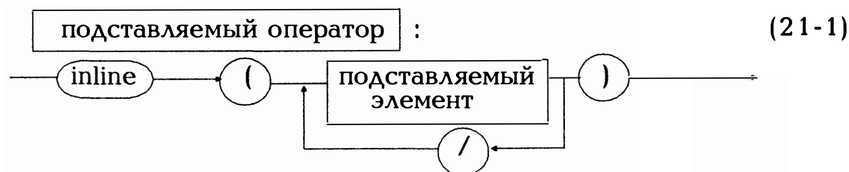
Бит 7: 1 = мерцание включено 0 = мерцание выключено
Бит 3: 1 = яркость норм. 0 = половинная яркость

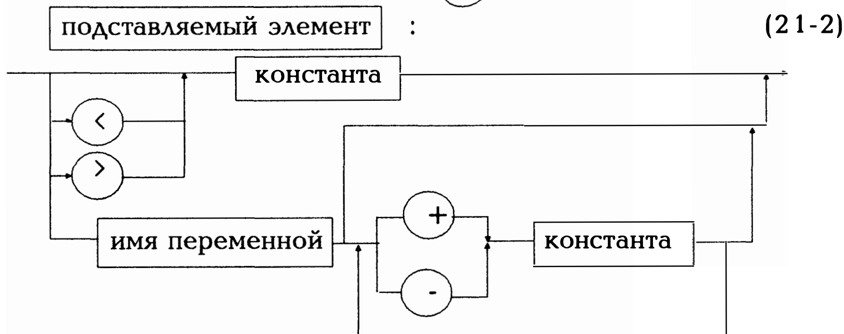
```
program bildspeicher;
uses Crt;
type wort = string[80];
var st : wort;
    a,sp,z,i:integer;

procedure writescreen(z,sp,attr:integer;st:wort);
const line = 160;
begin
   clrscr;
   for i := 1 to length(st) do
      begin
```

```
      mem[$B000:2*i+z*line+sp] := ord(st[i]);
      mem[$B000:2*i+1+z*line+sp] := attr;
    end;
  gotoxy(1,24);
end;

begin    (*********Исполняемая часть**********)
  repeat
    writeln('слово: конец = stop');
    readln(st);
    if st <> 'stop'then
    begin
    writeln('строка:'); readln(z);
    writeln('столбец:'); readln(sp);
    writeln('атрибут:'); readln(a);
    (* Атрибут целесообразно задавать
      в шестнадцатеричном представлении *)
    writescreen(z,sp,a,st);
    end;
  until st = 'stop';
end.
```

Для записи в память дисплея можно было использовать также прерывание $10 (см.рис.21.4).

В заключение этой главы кратко расскажем о подставляемых операторах, относящихся согласно (10-2) к простым операторам. Такие операторы имеют следующий вид:





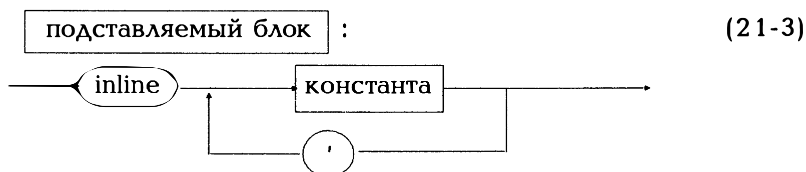
Итак, можно добавлять отдельные битовые последовательности, то есть объектные коды, непосредственно в программу на Паскале. Следовательно,

```
inline($CD/5);       { INT 5 }
```

означает прерывание $5 (см. рис. 21.4), то есть сочетание Shift-PrintScreen, с выводом содержимого экрана на печатающее устройство. Это можно сформулировать и как процедуру:

```
procedure shift_print_screen_proc;
{ Функция Shift-Print-Screen }
begin
   inline($CD/5);   { INT 5 }
end;  { процедура  shift print screen }
```

Согласно (11-6) процедура может формулироваться и как подставляемый блок:



Приведенная выше процедура shift_print_screen_proc может быть записана и в таком виде:

```
procedure shift_print_screen;
inline($CD/5);              { INT 5 }
```

Различие между "обычной" процедурой shift_print_screen_proc и подставляемой процедурой shift_print_screen состоит в том, что подставляемая директива каждый раз встраивается непосредственно в программу, а не вызывается в виде подпрограммы, как это имеет место для "обычной" процедуры.

Пример 21.8:

Из программы можно выбрать форму курсора. Для этого используется прерывание $10 (см. рис. 21.4).

```
program cursor;
const l_cursor = $0B0C;
     b_cursor = $000E;
var c:char;
```

```
procedure setup_cursor(cursor_form:char);
{ Устанавливается форма курсора }
begin
  case cursor_form of
    'B':
inline($B4/$01/
  {        MOV    AH,1                   }
  $8B/$0E/>b_cursor/
  {        MOV    CX,[b_cursor] ; курсор в виде блока}
  $CD/$10);
  {        INT    10H                    }
   'L':

inline($B4/$01/
  {        MOV    AH,1                   }
  $8B/$0E/>l_cursor/
  {        MOV    CX,[l_cursor] ; курсор в виде линии}
  $CD/$10);
  {        INT    10H                    }

    ' ':
inline($B4/1/$B9/13/14/$CD/$10);
     { Ключ курсора включен }
  end;{case}
end; { of setup_cursor }
begin (*******Исполняемая часть*******)
  writeln('Как должен выглядеть курсор?');
  writeln('B = блок, L = линия');
  readln(c);
  setup_cursor(c);
end.
```

22. Графика

В стандартном модуле Турбо Паскаля GRAPH.TPU содержится целый набор весьма эффективных процедур, типов данных и констант для работы с графическими изображениями. Поскольку графические возможности версий 4.0 и 5.0 по сравнению с версиями 3.0 изменились значительно, в библиотеке TURBO.TPL имеется модуль GRAPH3, позволяющий без каких бы то ни было изменений выполнять графические программы, подготовленные в версии 3.0.

Для того, чтобы воспользоваться предлагаемыми модулем GRAPH.TPU возможностями, естественно, необходимо иметь компьютер, оснащенный видеоадаптером. В состав модуля GRAPH входит ряд программ драйверов для наиболее распространенных видеоадаптеров:

CGA
MCGA
EGA
VGA
Hercules
AT&T
3270 PC

Эти драйверы хранятся с файлах с атрибутами .BGI (= Borland Graphic Interface). Для множества символов имеются файлы описаний .CHR. В модуль GRAPH включена дюжина стандартных графических программ. Для экономии места приведем лишь несколько примеров для демонстрации принципов работы с графикой. На поставляемой вместе с Турбо Паскалем демонстрационной дискете есть программа GRDEMO.PAS, которая весьма полезна при изучении графических возможностей Турбо Паскаля.

Рассмотрим "скелет" графической программы:

Пример 22.1:

Скелет графической программы. Используемые графические операции поясняются на рис. 22.1..

```
program graphtest;
uses graph;
var graphdriver :integer;
    graphmode   :integer;
    errorcode   :integer;
begin
(* 1 *)  graphdriver := detect;
(* 2 *)  initgraph(graphdriver, graphmode,'');
   (* Переход в графических режим.
      Теперь можно вызывать графические процедуры *)
(* 3 *)  errorcode :=graphresult;
   closegraph;
   (* Выход из графического режима *)
write ('Графический драйвер:');
writeln(graphdriver:5,graphmode:5);
writeln(errorcode);
   if errorcode <>grok then

      begin
        write ('Графическая ошибка');
        writeln(grapherrormsg(errorcode));
        writeln('программа прервана .....');
```

```
      halt;
    end;
end.
```

В строке (*1*) переменной graphdriver присваивается значение постоянной detect (которое равно 0). В строке (*2*) вызывается процедура initgraph, с которой должна начинаться любая графическая программа. initgraph имеет два передаваемых по ссылке параметра graphdriver и graphmode. Процедура проверяет конфигурацию технических средств на наличие видеоадаптера, загружает соответствующий драйвер, выбирает нужный графический режим и переключается на него. В строке (*3*) по initgraph фиксируется состояние ошибки или по переменной errorcode выполненная последней графическая операция. closegraph завершает графический режим и возвращает в режим работы с текстом. После вывода результата процедурой initgraph в случае ошибки программа прерывается с выдачей соответствующего сообщения.

Вызов:	initgraph(gt, gm, tpf) Процедура graph
Параметры:	var gt, gm:integer;tpf:string;
Действие:	Если передаваемая по ссылке переменная gt имеет значение 0, с помощью некоторой процедуры detectgraph выбирается один из включенных в конфигурация графических драйверов gt и определяется подходящий графический режим gm. Загружается графический драйвер gt и осуществляется переход в графический режим gm. Если gt имеет значение <> 0, берется соответствующий драйвер, а gm должно иметь соответствующее значение (тогда detectgraph не вызывается). В строковой переменной tpf должен задаваться путь доступа к графическому драйверу, т.е. файлу BGI. Если здесь стоит пустая строка '', initgraph осуществляет поиск в текущем каталоге Для графического драйвера в GRAPH.TPU определяются: const detect = 0; CGA = 1; MCGA = 2; EGA = 3; EGA64 = 4; EGAMono = 5; RESERVED= 6; HercMono= 7; ATT400 = 8; VGA = 9; PC3270 =10; Возможные графические режимы gm различных

драйверов описывают разрешение (например, 640x400 точек растра, а также набор цветов).

Вызов:	closegraph Процедура graph
Параметры:	нет
Действие:	Графический драйвер удаляется из памяти и осуществляется переход в тот текстовый режим, который был установлен перед вызовом initgraph.

Вызов:	graphresult; Значение функции:integer graph
Параметры:	нет
Действие:	Значением функции является состояние ошибки последней графической операции. В качестве возможных значений в GRAPH.TPU определены следующие константы:

```
const
grok            = 0;  {без ошибки}
grnoinitgraph   =-1;
{Графический драйвер .BGI не установлен,
 initgraph не вызван.}
grnodetected    =-2;
{Нетграфического адаптера, режим не установлен.}
grfilenotfound  =-3;
{Файл драйвер не найден.}
grinvaliddraver =-4;
{Дефект в программе драйвера.}
grnoloadmem     =-5;
{Для драйвера не хватает места в памяти}
grnoscanmem     =-6;
{Не хватает места в памяти (scanfill).}
grnofloodmem    =-7;
{Не хватает места в памяти (floodfill).}
grfontnotfound  =-8;
{Не найден файл с набором символов .CHR.}
grnofontmem     =-9;
{Для множества символов не хватает места в памяти}
grinvalidmod    =-10;
{Загруженный драйвер не поддерживает графический режим.}
```

Для каждого такого значения с помощью функции grapherrormsg генерируется соответствующая текстовая строка.

Вызов:	grapherrormsg(ec) Значение функции:string graph
Параметры:	ec:integer;
Действие:	Генерируется сообщение об ошибке, соответствующее коду ошибки ec (см. выше).

Рис. 22.1. Графические процедуры из примера 22.1

Для того, чтобы работать с графическими изображениями, нужна система координат. В зависимости от выбранного графического режима используется разрешение в 320x200 или 640x200 точек. В первом случае имеет место такая ситуация:

(0,0) Левый верхний угол экрана (319,0) Правый верхний угол

(159,99) середина

(0,199) Левый нижний угол (319,199) Правый нижний угол

Итак, начало координат (0,0) всегда находится в левом верхнем углу экрана, ось x идет слева направо (столбцы), ось y сверху вниз (строки). Для того, чтобы можно было из программы установить графический режим и соответствующее разрешение независимо от выбранных initgraph графического режима и разрешения, существуют функции getmaxx и getmaxy. Итак, согласно

```
xm := getmaxx div 2;
ym := getmaxy div 2;
```

(xm,ym) всегда центр экрана. Простейшей геометрической фигурой является отрезок прямой между двумя заданными точками. Если в дальнейшем говорится о курсоре, имеется в виду курсор в графическом режиме, который в отличие от текстового режима невидимо мерцает в этой точке. Тем не менее помнить о нем нужно. При выполнении графических операций всегда указывается, что делает курсор. Прямую можно сгенерировать двумя способами: во-первых, с помощью line, когда задаются начальная и конечная точки:

```
line (10, 20, 100, 100);
```

В результате получим отрезок прямой между точками (10,20) и (100,100). Во-вторых, курсор с помощью moveto можно переместить в начальную точку, а затем с помощью lineto нарисовать прямую до конечной точки:

```
moveto(10,20);
lineto(100,100);
```

В результате получим ту же самую прямую. Тип линии можно выбрать с помощью setlinestyle, например:

```
setlinestyle(dashedln,0,thickwidth);
(*Утолщенная пунктирная линия*)
line (10,20,100,100);
```

Если setlinestyle не используется, вычерчивается сплошная, обычной толщины линия.

Пример 22.2:

Вычерчиваются оси координат и семейство из десяти концентрических окружностей.

```
program kreise;
uses crt,graph;
var
  gr_treiber, gr_modus, errorcode:integer;
  xm, ym, i:integer;

begin
  gr_treiber := detect;
  initgraph(gr_treiber, gr_modus,'');
  errorcode :=graphresult;
  if errorcode <> grok then
     begin
        writeln('Ошибка при работе в графике',
                 grapherrormsg(errorcode));
        closegraph;
        halt;
     end;
     xm := getmaxx div 2;
     ym := getmaxy div 2;
     line(xm,0,xm,getmaxy);
    line (0,ym,getmaxx,ym)
     for i := 1 to 10 do circle(xm,ym,i*20);
     moveto(0,0);
     outtext('<RETURN>');
     readln;
     closegraph;
end.
```

Вызов:	cleardevice	Процедура graph
Параметры:	нет	
Действие:	Экран гасится. Все параметры графического драйвера устанавливаются на стандартное значение.	
Вызов:	setviewport(x1,y1,x2,y2,clip)	Процедура graph
Параметры:	x1,y1,x2,y2:word;clip:boolean;	
Действие:	Создается символьное окно с левым верхним углом (x1,y1) и правым нижним углом (x2,y2). Курсор устанавливается в верхний левый угол окна (относительные координаты (0,0). Параметр clip устанавливает, пересекают ли символы границы окна	

	(true - да, false - нет). Имеются константы:
	const
	  clipon = true; {пересекают}
	  clipoff = false; {не пересекают}
	initgraph устанавливает в качестве окна весь экран.

Вызов:	getviewsettings(viewport) Процедура graph
Параметры:	var viewport:viewpotrtype;

Действие:	Определяется информация о текущем окне. Для этого в модуле graph имеется тип:
	type
	  viewporttype=record
	        x1,y1,x2,y2:word;
	        clip:boolean;
	        end;

Вызов:	clearviewport Процедура graph
Параметры:	нет
Действие:	Содержимое текущего окна гасится и окрашивается в заданный palette(0) цвет.
Вызов:	setvisualpage(p) Процедура graph
Параметры:	p:word;
Действие:	Некие графические адаптеры (например, EGA, VGA, Hercules) поддерживают несколько страниц (см. setactvepage). setvisualpage устанавливает, какая страница видна.
Вызов:	setactivepage(p) Процедура graph
Параметры:	p:word;
Действие:	Некоторые графические адаптеры (например, EGA, VGA,Hercules) поддерживают несколько страниц. setactivepage устанавливает, на какой странице строится изображение. Такой страницей может быть и фон, то есть изображение будет невидимым.
Вызов:	getmaxx Значение функции:word graph
Параметры:	нет
Действие:	Значением функции является максимально возможная для данного драйвера и режима координата x (например, 319,639).
Вызов:	getmaxy Значение функции:word graph
Параметры:	нет
Действие:	Значением функции является максимально возможная для установленного драйвера и режима координата y (например, 199,349).

Рис. 22.2. Стандартные программы для формирования экрана, окна, страницы

Вызов:	putpixel(x,y,f)	Процедура	graph
Параметры:	x,y:integer,f:word;		
Действие:	В точке x,y рисуется точка растра цвета f.		
Вызов:	getpixel(x,y)	Значение функции:word	graph
Параметры:	x,y:integer;		
Действие:	Значением функции является номер цвета элемента графического изображения в точке x,y.		
Вызов:	getx	Значение функции:integer	graph
Параметры:	нет		
Действие:	Значением функции является координата x текущей позиции курсора.		
Вызов:	gety	Значение функции:integer	graph
Параметры:	нет		
Действие:	Значением функции является координата y текущей позиции курсора.		

Рис. 22.3. Стандартные программы для работы с точечными изображениями

Вызов:	line(x1,y1,x2,y2)	Процедура	graph
Параметры:	x1,y1,x2,y2:integer;		
Действие:	Вычерчивается отрезок прямой от точки (x1,y1) до точки (x2,y2). При этом используется установленный при последнем вызове setlinestyle тип линии (см. там) и установленный с помощью setcolor цвет. Курсор не меняется.		
Вызов:	lineto(x,y)	Процедура	graph
Параметры:	x,y:integer;		
Действие:	Вычерчивается прямая из текущей позиции курсора до точки (x,y). При этом используется установленный с помощью setlinestyle тип линии и определенный с помощью setcolor цвет. После вызова процедуры курсор устанавливается в позицию (x,y).		
Вызов:	linerel(dx,dy)	Процедура	graph
Параметры:	dx,dy:integer;		
Действие:	Курсор находится в позиции x,y, начиная с которой вычерчивается прямая до точки (x+dx,y+dy).		
Вызов:	moveto(x,y)	Процедура	graph
Параметры:	x,y:integer;		
Действие:	Устанавливает курсор в точку (x,y). x и y относятся к текущему окну.		
Вызов:	moverel(dx,dy)	Процедура	graph
Параметры:	dx,dy:integer;		
Действие:	Если курсор находится в точке (x,y), он переместится в точку (x+dx,y+dy).		

Вызов:	setlinestyle(stil,muster,dicke)	Процедура
Параметры:	stil,muster,dicke:word;	
Действие:	Устанавливает тип линии для следующей операции над символами (line,lineto arc,circle,drawpoly, rectangle). Для stil в модуле graph имеются следующие константы: const solidln = 0; {сплошная линия} dotteln = 1; {пунктирная линия} centerln = 2; {штрихпунктирная линия (точка-тире-точка)} dashedln = 3; {штриховая линия} userbitln = 4; {определенная пользователем линия} Параметр muster имеет значение только при stil=4. Для параметра dicke имеются следующие константы: const normwidth = 1; (обычная толщина) thickwidth = 3; (жирная линия)	
Вызов:	getlinestyle(li)	Процедура graph
Параметры:	var li:linesettingstyp;	
Действие:	Определяется установленный на данный момент тип линии. Для этого в модуле graph имеется: type linesettingstype=record linestyle:word; pattern :word; thickness:word; end;	

Рис. 22.4. Стандартные программы для работы с графическими примитивами типа "линия"

Вызов:	arc(x,y,w1,w2,r)	Процедура graph
Параметры:	x,y:integer;w1,w2,r:word;	
Действие:	Из центра (x,y) вычерчивается дуга радиуса r от угла w1 до угла w2. Угол следует задавать в градусах (0=правая горизонталь, 90=верхняя вертикаль и т.д.)	
Вызов:	getarccoords(ac)	Процедура graph
Параметры:	var ac:arccoordstype;	
Действие:	Процедура возвращает данные о последнем вызове arc. Для этого в модуле graph имеется тип данных: type arccoordstype=record x,y:integer (*центр*) xs,ys (*Точка запуска*)	

```
xend,yend:word (*Конечная точка*)
end;
```

Вызов: circle(x,y,r) Процедура graph
Параметры: x,y:integer;r:word;
Действие: Вычерчивается окружность радиуса r с центром в точке (x,y) (цвет линии текущий).

Вызов: getaspectratio(x,y) Процедура graph
Параметры: var x,y:word;
Действие: Определяется физическое отношение высоты к стороне экрана, которое ставится в соответствие каждому драйверу и режиму. Используется для arc,circle и pieslice, чтобы можно было представить дуги окружности в виде круга.

Вызов: rectangle(x1,y1,x2,y2) Процедура graph
Параметры: x1,y1,x2,y2:integer;
Действие: Вычерчивается прямоугольник с координатами левого верхнего угла (x1,y1) и координатами правого нижнего угла (x2,y2). Причем должны выполняться неравенства:

0 <= x1 < x2 <= getmaxx
0 <= y1 < y2 <= getmaxy

Для символов используется процедура setlinestyle.

Вызов: bar(x1,y1,x2,y2) Процедура graph
Параметры: x1,y1,x2,y2:integer;
Действие: Вычерчивается прямоугольник с координатами левого верхнего угла (x1,y1) и координатами правого нижнего угла (x2,y2). Прямоугольник закрашивается в соответствии с образцом, заданным процедурами setfillstyle или setfillpattern.

Вызов: bar3d(x1,y1,x2,y2,tiefe,top) Процедура graph
Параметры: x1,y1,x2,y2:integer;tiefe:word;
top:boolean;
Действие: Рисуется трехмерный брусок с прямоугольным основанием с координатами (x1,y1) и (x2,y2). tiefe задает пространственную глубину в элементах изображения, например, 25% ширины. Прямоугольник заполняется в соответствии с образцом, заданным процедурами setfillstyle или setfillpattern. Контур выполняется в соответствии с setlinestyle. top задает, должен ли брусок иметь верхнее замыкание или нет, для чего в модуле graph имеются две константы:

```
const
  topon  = true;
  topoff = false;   {Нет крышки}
```

Вызов:	drawpoly(anz,punkte) Процедура graph
Параметры:	anz:word;var punkte;
Действие:	Вычерчивается многоугольник с anz вершинами. punkte является нетипизированным параметром, содержащим координаты вершин. Для пары координат одной точки задается тип: type pointtype=record x,y:word; end; Итак, punkte имеет тип array[1..n] of pointtype. drawpoly использует текущий цвет и заданный процедурой setlinestyle тип линии.
Вызов:	fillpoly(anz,punkte) Процедура graph
Параметры:	anz:word; var punkte;
Действие:	Вычерчивается многоугольник, имеющий anz вершин. Координаты одной точки могут задается через определенный в модуле graph тип: type pointtype=record x,y:word; end; например, как array of pointtype. После того, как многоугольник построен (последняя точка должна совпадать с первой, чтобы ломаная замкнулась), он заполняется в соответствии с определенным процедурами setfillstyle или setfillpattern образцом.
Вызов:	setfillstyle(muster,f) Процедура graph
Параметры:	muster:word;f:word;

Действие·	Устанавливает образец для заполнения площадки. Для muster существуют следующие константы:
	const
	emptyfill = 0;
	{Заполнение цветом фона}
	solidfill = 1; {сплошное заполнение}
	linefill = 2; {—-}
	ltslashfill= 3; {\\\}
	slashfill =4; {\\\, линии утолщенные}
	bkslashfill=5; {\\\" " }
	ltbkslashfill=6; {\\\}
	hatchfill = 7; {легкая штриховка}
	xhatchfill = 8;
	{частая штриховка, пересекающаяся}
	interleavefill=9; {чередующиеся линии}
	widedotfill =10;
	{далеко отстоящие одна от другой точки}
	closedotfill =11; {жирные точки}

	userfill =12; {определяется пользователем, резервная} При заполнении используется цвет f.
Вызов:	setfillpattern(muster,f) Процедура graph
Параметры:	muster:fillpatterntype;f:word;
Действие:	Устанавливает образец заполнения и цвет для одной из вызываемых процедур fillpoly, floodfill, bar, bar3d, pieslice. В модуле graph имеется тип type fillpatterntype=array[1..8] of byte; При этом каждому биту этого массива соответствует один элемент изображения (пиксель). Каждый байт определяет восемь расположенных рядом точек. Восемь байтов устанавливаются один за другим. Образец может определяться либо через setfillstyle, либо через setfillpattern (но не обеими процедурами сразу).
Вызов:	floodfill(x,y,rand) Процедура graph
Параметры:	x,y,rand:word;
Действие:	Если точка (x,y) находится внутри ограниченной некоторыми линиями цвета rand поверхности, она закрашивается (заполняется). Для этого используется установленный с помощью setfilppatern или setfillstyle образец.
Вызов:	pieslice(x,y,w1,w2,r) Процедура graph
Параметры:	x,y:integer; w1,w2,r:word;
Действие:	Вычерчивается дуга радиуса r с центром в точке (x,y) от угла w1 до угла w2. Углы должны задаваться в градусах и отсчитываться как и для arc против часовой стрелки. Затем такая "вырезка заполняется согласно установленному с помощью setfillstyle или setfillpattern образцу.

Рис. 22.5. Стандартные программы для работы с дугами, фигурами и штриховкой

В графическом режиме можно создавать так называемые палитры. Палитра имеет номер и состоит из серии красок. Первой краской является цвет фона, например:

Цветовая палитра для цветной графики:

№ палитры.	Краски 0	1	2	3
0	Фон	зеленый	красный	коричневый
1	Фон	бирюзовый	фиолетовый	светлосерый
2	Фон	салатовый	розовый	желтый
3	Фон	св.бирюзовый	сиреневый	белый

В качестве фона используются цвета N 0-15, представленные на рис. 8.9.

Вызов:	setbkcolor(f)	Процедура	graph
Параметры:	f:word;		
Действие:	f устанавливает, какой цвет должен браться в качестве фона.		
Вызов:	setcolor(f)	Процедура	graph
Параметры:	f:word;		
Действие:	Устанавливается цвет N f текущей палитры. setcolor(0) является цветом фона, setcolor(1) - первая запись в палитре.		
Вызов:	getbcolor	Значение функции:word	graph
Параметры:	нет		
Действие:	Значением функции является номер записи в палитре, используемой в качестве фона.		
Вызов:	getcolor	Значение функции:word	graph
Параметры:	нет		
Действие:	Значением функции является номер той краски в палитре, которая используется в данный момент.		
Вызов:	setallpalette(p)	Процедура	graph
Параметры:	var p:нетипизированная со структурой record anzahl:byte; farben:array[1..max]of shortint; end;		
Действие:	p.anzahl является числом записей в палитре. Отдельные цвета задаются p.farben[i]. Значение -1 означает "без изменения".		
Вызов:	setpalette(fnr,f)	Процедура	graph
Параметры:	fnr:word; f:byte;		
Действие:	Запись fnr текущей палитры устанавливает цвет f. setpalette(1,green) означает, что все помеченные 1 объекты окрашиваются в зеленый цвет.		
Вызов:	getpalette(p)	Процедура	graph
Параметры:	var p:palettetype;		
Действие:	Определяются отдельные записи текущей палитры. В модуле graph для этого имеются: const maxcolors=15 type palettetype=record size:byte; colors:array[1..maxcolors]of shortint end;		

	p.size содержит число использованных записей. p.colors[i] является цветом i-той записи. (См. также setallpalette)

Рис. 22.6. Стандартные программы для работы с цветом и палитрой

Вызов:	outtext(str)	Процедура	graph
Параметры:	str:string;		
Действие:	В позиции курсора выводится строка string. Автоматической переверстки строки не производится Шрифт устанавливается с помощью процедур settextjustify, settextstyle, textwidth и textheight.		
Вызов:	outtextxy(x,y,str)	Процедура	graph
Параметры:	x,y:integer; str:string;		
Действие:	Строка str выдается, начиная с позиции x,y. Прочее как для outtext.		
Вызов:	settextstyle(font,dir,gr)	Процедура	graph
Параметры:	font,dir,gr:word;		

Действие:	Для следующего вывода текста задаются множество символов, наклон и размер. Для этого в модуле graph имеются следующие константы:
	const
	defaultfont = 0;
	{8*8 бит определены для элемента изображения
	triplexfont = 1; {векторное множество символов
	smallfont = 2;
	sansseriffont = 3;
	gothicfont = 4;
	hopizdir = 0; {слева направо}
	vertdir = 1; {снизу вверх}
	normsize = 1; {обычные размеры текста}		

Вызов:	settextjustify(hor,vert)	Процедура	graph
Параметры:	hor,vert:word;		

Действие:	Устанавливаются параметры для выдачи текста. Для этого в модуле graph имеются следующие константы:
	const
	{Для горизонтального расположения}
	lefttext = 0; {текст, начиная с позиции курсора}
	centertext = 1;
	{текст центрирован относительно позиции курсора}
	righttext = 2;
	{Текст завершается в позиции курсора}
	
	{Для вертикали}
	bottomtext = 0; {Текст над позицией курсора}		

```
                cenrettext = 1; {Текст центрирован относи-
                                 тельно позиции курсора}
                toptext    = 2; {Текст под позицией курсора}
```

Вызов:	textheigth(str)	Значение функции:word
Параметры:	srt:string;	
Действие:	Значением функции является высота строки str в элементе изображения.	

Вызов:	textwidth(str)	Значение функции:word
Параметры:	str:string;	
Действие:	Значением функции является ширина строки str.	

Вызов:	gettextsettings(info)	Процедура graph
Параметры:	var info:textsettingstype;	
Действие:	Через переменную info передаются установленные в процедурах settextstyle и settextjustify параметры. textsettingstype определяется в модуле graph как	

```
type
  textsettingstype = record
          font:word;
          direction:word;
          {1..10}
          horiz    :word;
          vert     :word;
        end;
```

Рис. 22.7. Стандартные программы работы с текстом

Если мы хотим начертить кривую y=f(x) в интервале a<x<=b, этот интервал будет проходиться с шагом deltax, а две соседние точки соединяться прямой. Итак, типичная программа имела бы следующий вид:

```
x1: = a;
y1: = f(x1);
while x1  <= b do
 begin
   x2:=x1+deltax;
   y2:=f(x2);
   line(round(x1),round(y1),round(x2); round(y2));
   x1 :=x2; y1 :=y2;
 end;
```

При этом следует иметь в виду следующие две вещи: если x и y вещественные числа, они должны обрабатываться как аргументы line типа integer, например, с помощью round(). Выбранный с помощью initgraph графический режим устанавливает шкалу для осей

x и y. Тогда необходимо выбрать соответствующее деление шкалы, чтобы получилась осмысленная картинка. В следующем примере рисуется функция sin(x). Поскольку |sin(x)|<=1, она должна соответствовать на рисунке диапазону -100..+100. Итак, 100*sin(x) вычерчена. Для того, чтобы получить кривую в диапазоне 0 < x < 2*pi, она нормируется по x = 100 соответственно pi. В качестве длины шага deltax берется pi/100.

Пример 22.3:

```
program sinus;
uses crt,graph;
var
  gr_treiber, gr_modus, errorcode:integer;
  xm, ym, i:integer; pihu, x1, y1, x2, y2:real;
begin
  gr_treiber := detect;
  initgraph(gr_treiber, gr_modus,'');
  errorcode :=graphresult;
  if errorcode <> grok then
     begin
       write('Ошибка в графике');
       writeln(grapherrormsg(errorcode));
       closegraph;
       halt;
     end;
     xm := getmaxx div 2;
     ym := getmaxy div 2;
     line(0,ym-100,0,ym+100);
     line(0,ym, getmaxx, ym);
     line(100,ym-100,100,ym+100);
     outtextxy(100,ym+100,'pi');
     moveto(0,ym);
     pihu :=pi/100;
     x1 := 0;
     while x1 <= 200 do begin
           y1 := 100*sin(x1*pihu);
           x2 := x1+pihu;
           y2 := 100*sin(x2*pihu);
           line(round(x1),ym+round(y1),round(x2),
                ym+round(y2));
           x1 := x2;
           y1 := y2;      end;

     outtextxy(xm+200,ym+90,'<RETURN>');
     readln;
```

```
  closegraph;
end.
```

Пример 22.4:

Демонстрация drawpoly и fillpoly

```
program poly;
uses crt,graph;
const muster:fillpatterntype = ($FF, $00, $FF, $00,
                                $FF, $00, $FF, $00);
    fuenfeck:array [1..6] of pointtype =
                              ( (x:50;y:100),
                                (x:80;y:110),
                                (x:100;y:130),
                                (x:150;y:90),
                                (x:170;y:50),
                                (x:50;y:100));
var
 viewport :viewporttype;
  gr_treiber, gr_modus, errorcode:integer;
  xm, ym, i:integer; x1,y1,x2,y2:real;
  color :word; s:string;
begin
 gr_treiber := detect;
 initgraph(gr_treiber, gr_modus, '');
 errorcode :=graphresult;
 if errorcode <> grok then
   begin
     write('Графическая ошибка');
     writeln(grapherrormsg(errorcode));
     closegraph;
     halt;
   end;
   xm := getmaxx div 2;
   ym := getmaxy div 2;
   drawpoly(6,fuenfeck);
   for i := emptyfill to closedotfill do begin
   setfillstyle(i,i);
   fillpoly(6,fuenfeck);
   outtextxy(300,170,'Return');
   readln;        end;
   closegraph;
end.
```

Пример 22.5:

Пример для bar и bar3d

```
program balken;
uses graph;
var gr_treiber, gr_modus, errorcode:integer;
var xm,ym, i:integer;

begin
  gr_treiber := detect;
  initgraph(gr_treiber, gr_modus,'');
  errorcode :=graphresult;
  if errorcode <> grok then
     begin
       write('Графическая ошибка');
       writeln(grapherrormsg(errorcode));
       closegraph;
       halt;
     end;
     xm := getmaxx div 2;
     ym := getmaxy div 2;
    line(0,ym,getmaxx,ym);
    setfillstyle(linefill, red);
    for i := 1 to 5 do
    begin
      bar(15*i,ym-15*i,15*(i+1),ym);
    end;
    outtextxy(xm,ym+50,'Return');
    readln;
    cleardevice;
    line(0,ym,getmaxx,ym);
    setfillstyle(linefill, red);
    for i := 1 to 5 do
    begin
       bar3d(15*i,ym-15*i,15*(i+1),ym,10,topon);
    end;
    outtextxy(xm,ym+50,'Return');
    readln;
    closegraph;
end.
```

Пример 22.6:

Демонстрация aspect_ratio. Строится окружность радиуса r с центром в точке (xm,ym) согласно формуле

```
x = xm + r*cos(t)
y = ym + r*sin(t)       (0 <= t <= 2*pi)
```

При этом возникает явно выраженное искажение по оси y. Затем та же самая окружность строится с помощью circle(xm,ym,r).

```
program aspect_ratio;
uses graph;
var gr_treiber, gr_modus, errorcode:integer;
    xm,ym,i:integer;
    xasp,yasp:word;

procedure kreis(xm,ym,r:integer);
var t, zweipi, pihu:real;
x,y,xx,yy:integer;
begin
zweipi := 2*pi;
pihu   := pi/100;
x :=xm+r;
y := ym;
t := 0;

while t <=zweipi do
  begin
    xx := round(xm+r*cos(t));
    yy := round(ym+r*sin(t));
    line(x,y,xx,yy);
    t := t+ pihu;
    x := xx;
    y := yy;
  end;
end; (*окружность*)
begin
  gr_treiber := detect;
  initgraph(gr_treiber, gr_modus,'');
  errorcode := graphresult;
  if errorcode <> grok then
     begin
       write('Графическая ошибка');
       writeln(grapherrormsg(errorcode));
       closegraph;
       halt;
     end;
     xm := getmaxx div 2;
     ym := getmaxy div 2;
     line(xm,0,xm,getmaxy);
```

```
    line(0,ym,getmaxx,ym);
    kreis(xm,ym,80);
    outtextxy(xm+80,ym+10,'Построение окружности  с
            помощью процедуры<Return>');
    readln;
    circle(xm,ym,80);
    getaspectratio(xasp,yasp);
    rectangle(xm-80,ym-round((xasp/yasp)*80),
            xm+80,ym+round((xasp/yasp)*80));
    outtextxy(xm,ym+40,'Построение окружности  с
            помощью circle<Return>');
    readln;
    closegraph;
    writeln('aspectx:y= ',xasp:8,':',yasp:8);
end.
```

Пример 22.7:

Создается и обрамляется окно.

```
program fenster_rahmen;
uses graph;
var gr_treiber, gr_modus, errorcode:integer;
   xm, ym:integer;
   vpt:viewporttype;

begin
  gr_treiber := detect;
  initgraph(gr_treiber, gr_modus, '');
  errorcode :=graphresult;
  if errorcode <> grok then
     begin
      write('Графическая ошибка');
      writeln(grapherrormsg(errorcode));
      closegraph;
      halt;
     end;

     xm := getmaxx div 2;
     ym := getmaxy div 2;

     setviewport(xm-80, ym-80, xm+50, ym+50, clipon);
     line(0,0,150,150);
     getviewsettings(vpt);
     with vpt do rectangle(0, 0, x2-x1, y2-y1);
```

```
    outtextxy(20,20,'Return');
    readln;
    closegraph;
end.
```

Пример 22.8:

Демонстрируются различные типы шрифтов.

```
program schrift;
uses graph;
var gr_treiber, gr_modus, errorcode:integer;
   xm, ym, i, x, y:integer;
begin
  gr_treiber := detect;
  initgraph(gr_treiber, gr_modus, '');
  errorcode :=graphresult;
  if errorcode <> grok then
     begin
       write('Графическая ошибка');
       writeln(grapherrormsg(errorcode));
       closegraph;
       halt;
     end;
     xm := getmaxx div 2;
     ym := getmaxy div 2;
     line (xm, 0, xm, getmaxy);
     moveto(xm,20);
     settextjustify(centertext,centertext);
     outtext('Привет, друзья!');
     moveto(xm,30);
     settextstyle(smallfont,horizdir,5);
     outtext('Привет, друзья!');
     moveto(xm,50);
     settextstyle(gothicfont,horizdir,2);
     outtext('Привет, друзья!');
     moveto(30,getmaxy-120);
     settextstyle(gothicfont,vertdir,2);
     settextjustify(centertext,bottomtext);
     outtext('Привет, друзья!');
     settextjustify(centertext,toptext);
     moveto(30,getmaxy-115);
     settextstyle(gothicfont,vertdir,2);
     outtext('Привет...');
     y := 60;
     for x :=1 to 10 do
```

```
    begin
      settextstyle(triplexfont,horizdir,x);
      settestjustify(centertext,centertext);
      inc(y,textheight('m')+5);
      outtextxy(xm,y,'Привет, друзья!');
    end;
  settextstyle(defaultfont,horizdir,1);
  outtextxy(30,20,'Return');
  readln;
  closegraph;
end.
```

Примечания относительно версии 5.0

Модуль graph содержит и несколько других стандартных программ. Вот наиболее интересные из них:

Вызов:	getdefaultpalette(p)	Процедура graph
Параметры:	var p:palettetyp;	
Действие:	Определяется стандартная палитра, выбранная графическим драйвером согласно процедуре initgraph (как и после вызова процедур setpalette или setallpalette (рис.22.6), где также задается тип palettetyp).	
Вызов:	getdrivername	Значение функции: string graph
Параметры:	нет	
Действие:	Значением функции является выбранный с помощью initgraph драйвер.	
Вызов:	getmaxmode	Значение функции: word graph
Параметры:	нет	
Действие:	Определяется максимально возможное значение graphmode (обычно максимальное разрешение).	
Вызов:	getpalettesize	Значение функции: word graph
Параметры:	нет	
Действие:	Значением функции является число записей в палитре текущего графического режима.	
Вызов:	setusercharsize(mx,dx,my,dy)	Процедура graph
Параметры:	mx, dx, my, dy : word;	
Действие:	В отличие от settextstyle могут задаваться коэффициенты увеличения по оси x и y. Ширина одного символа умножается на mx, а затем делится на dx. То же самое относится и к высоте символа my и dy.	

Рис. 22.8. Стандартные графические программы модуля GRAPH версии 5.0

23. Поиск ошибок

Один шутник заявил однажды: "Программирование - хорошая школа жизни. Все, что делается, оказывается вначале ошибкой." Такая утрированная формулировка, к сожалению, довольно близка к действительности. Поиск и устранение ошибок отнимают значительную часть времени, затраченного на создание программного продукта. Поэтому вначале следует получить представление о многочисленных возможностях поиска ошибок.

23.1. Классификация ошибок

Если Вы хотите пройти такую "школу жизни", можно описать эту работу таким образом, используя смесь псевдокодов и Паскаля:

```
repeat
  Quelleprogram erstellen oder aendern;
  {Составить или изменить исходную программу}
    if (compileiren ok) then binden
    {если компиляция прошла нормально, компоновать}
       else fertig := false;
    if (binden ok) then starten
    {если компоновка прошла нормально, запустить}
       else fertig := false;
    if (run-time-abort)
    {если прерывание при исполнении программы}
       then fertig := false;
    if (Ergebnisse korrekt) then fertig := true
    {если результат корректен, то fertig := true }
       else fertig := false;
until fertig;
```

Четыре оператора if в этом описании соответствуют четырем возможным типам ошибок, которые могут встретиться в программе:

- ошибка при компиляции,
- ошибка при компоновке,
- ошибка при исполнении, которая ведет к прерыванию,
- выполненная программа генерирует неверный результат.

Компилятор распознает синтаксические ошибки, то есть отступления от правил грамматики Паскаля. Такие ошибки сравнительно безобидны, поскольку причина ошибки точно описывается, а курсор указывает то место в исходном тексте программы, где и обнаружена ошибка. Компиляция прерывается на первой же ошибке, что на первый взгляд кажется недостатком. Но если неописанная переменная X встречается в разных местах программы 20 раз, то

будет обнаружено соответственно 20 ошибок. Компилятор же при обнаружении первой ошибки выдаст сообщение об ошибке с кодом 3: "unknow identifier". Остальные 19 случаев появления X в программе выдачей сообщения об ошибке не сопровождаются. В результате кажущийся недостаток обеспечивает очень высокую скорость компиляции.

Учтите, что курсор устанавливается не на то место, где находится ошибка, а сразу же за ней. Если в конце оператора отсутствует точка с запятой, это может быть помечено лишь в начале нового оператора. В операторах присваивания корректность может быть проверена лишь после того, как будет зарегистрирован тип выражения справа.

Наконец, следует сослаться на функцию Options/Compiler в основном меню, по которой можно выбрать параметры компилятора.

Наиболее часто при компоновке встречается ошибка, когда компоновщик не может найти заданный предложением uses модуль. В этом случае выдается сообщение об ошибке с кодом 15 "file not found (usesname.TPU)" ("Файл не найден (имя файла)"). Затем используемые ненайденным модулем параметры рассматриваются в программе как неописанные переменные.

Ошибки выполнения особенно обременительны, поскольку обычно сгенерированные до прерывания программы результаты еще находятся в оперативной памяти и при прерывании теряются. Наиболее частая причина прерывания при исполнении программы - неправильное обращение к процедурам ввода/вывода read/write, assign, reset, erase, rename и пр., когда требуемая операция не может быть выполнена, или использование недопустимых операций типа деления на нуль или обращение к функции типа квадратного корня, когда значение подкоренного выражения отрицательно. Ошибка выполнения может привести иногда к фатальным результатам, поскольку программа в этом случае не может быть завершена обычным путем, а значит, можно столкнуться с потерей данных, если открытые к этому моменту файлы не закрылись. Если исходная программа записана лишь в редакторе, а приходится вынужденно выйти из Турбо Паскаля или даже перезапустить компьютер, исходная программа будет потеряна. Чтобы обезопасить себя от таких случаев, перед запуском программы с помощью Alt-R нужно лишь сохранить ее, воспользовавшись клавишей F2. Здесь программиста может утешить лишь то, что место ошибки в программе помечается курсором, а в строке заголовка окна редактирования указывается причина ошибки.

Естественно, ошибка указывается лишь тогда, когда это доступно редактору. Если программа скомпилирована с помощью Compile/Destination ... disk и запущена по команде MS-DOS, в случае ошибки выдается сообщение

runtime error nr.at xxxx:yyyy

Адрес ошибки следует записать. Если затем исходный текст загружается в редактор и выполняется функция Compile/find error, требуется указать шестнадцатеричный адрес; если указать записанный ранее адрес xxxx:yyyy, можно тем самым указать место ошибки в тексте программы.

Самое трудное, когда удалось скомпилировать и скомпоновать программу без ошибок, без каких бы то ни было явных сбоев, но тем не менее программа выдает неверный результат. Можно лишь порадоваться, если вообще удалось заметить, что результат некорректен. Если причину ошибки при просмотре программы обнаружить не удалось, часто оказывается полезным добавление операторов write в критических точках программы, чтобы можно было проконтролировать ход выполнения программы. Чаще всего критическими точками программы являются циклы, точки ветвления, передача параметров через аргументы функции или генерация каких-то значений. При проверке корректности операторов ввода/вывода не забывайте о функции ioresult и директиве (*$I*).

Если включена функция Option/Compiler/Debug information или если в исходный текст включена директива (*$D*), генерируется .EXE-файл с информацией, позволяющей локализовать ошибки исполнения. Для того, чтобы получить эти данные в читабельной форме, нужно установить в ON и Option/Compile/Turbo map file generation. Тогда при каждой компиляции генерируется файл .TPM (Turbo Pascal Map-file). Для того, чтобы сделать его содержимое читабельным, существует программа TPMAP.EXE, которая из файла .TPM генерирует файл .MAP. Такой файл содержит подробную информацию о символических адресах, сегментах и пр. Поможет ли эта информация в поиске ошибок, зависит от искусства того, кто ищет ошибку.

Для больших программ все это весьма утомительно. Тогда нужно обратиться к отладчику. Отладчик - это программа поиска ошибок, работающая следующим образом: после вызова отладчика программа компилируется, связывается и выполняется под его контролем. В исходном тексте можно указать точки останова, в которых выполнение программы должно быть приостановлено для выдачи определенного протокола со значениями переменных. Программа может выполняться в пошаговом режиме, что позволяет отследить результат действия отдельных операторов. Подобные отладчики предлагаются на рынке программных продуктов.

23.2. Турбо-отладчик (примечания относительно версии 5.0)

Пользователь версии 5.0 располагает "встроенным" отладчиком, который мы и попытаемся сейчас описать. Этот Турбо-отлад-

чик является так называемым отладчиком исходного текста, то есть в процессе отладки он работает с исходным текстом. Нет необходимости специально заботиться о запуске отладчика. В основном меню стандартно устанавливается

Option/Compiler/Debug information ON
Option/Compiler/Local symbols ON
Debug/Integrated Debugging ON

то есть отладчик всегда присутствует в фоновом режиме и может быть активизирован простым нажатием клавиш (см. рис. 3.4). С помощью отладчика при выполнении программы можно перейти на уровень исходного текста и проверить или изменить содержимое переменных.

Разъясним вначале несколько понятий. Точкой останова (breakpoint) называет точка программы, в который выполнение программы прерывается, после чего из нее может быть вызван отладчик. С помощью некоторого контрольного выражения (watch expression) можно прерывать программу, если значение данного выражения изменилось.

Отладчик управляется функциональными клавишами. Важнейшими из них являются:

F1 "Подсказка"

F7 = Run/Trace into

Выполняется следующий оператор в строке, после чего осуществляется останов. При этом вызванные функции также выполняются по шагам (trace into). Если это не удается (если Вы находитесь в начале программы), активизируется отладчик. Первая строка программы помечается как рабочая (то есть следующая подлежащая выполнению команда) прямоугольником.

F8 = Run/Step over

То же, что и клавиша F7, но без пошагового выполнения функций (Step over)

F4 = Run/Go to cursor

Программа выполняется вплоть до позиции курсора. Если это невозможно (курсор стоит в начале программы) активизируется отладчик.

Ctrl-F4 = Debug/Evaluate

В одно из окон следующей формы

```
┌──── Evaluate ────┐
└──────────────────┘

┌──── Result ──────┐
└──────────────────┘

┌──── New Value ───┐
└──────────────────┘
```

можно ввести выражение, значение которого индицируется. Если в качестве выражения вводится переменная, можно изменить ее значение, воспользовавшись третьим окном.

Ctrl-F7 = Break/Add watch

В окно формы

```
┌──── Add watch ─────┐
│                    │
└────────────────────┘
```

можно ввести выражение, значение которого должно контролироваться. Оно будет постоянно высвечиваться в нижней части экрана, если там имеется окно для выдачи служебных сообщений.

Break/Delete watch

стирает текущее (то есть выделенное ранее) контроль ное выражение.

Break/Remove all watches

стирает все выделенные ранее контрольные выражение.

Alt-F5

меняет экран, то есть переключает с окна редактирования на окно DOS и обратно.

Ctrl-F3 = Debug/Call stack

Высвечивает в окне "стек функций", то есть последовательность вызовов функций вплоть до текущего момента, с указанием их имени и значений переданных параметров.

Ctrl-Break

В текущей программе точка прерывания помечается в качестве рабочей строки. Команды отладчику можно задавать только в этом состоянии.

Ctrl-F8 = Break/Toggle Breakpoint

Установить в месте нахождения курсора точку останова и пометить это место в программе прямоугольником или удалить существовавшую там точку останова.

Break/Clear all breakpoints

стирает все установленные за время сеанса точки останова.

Break/View next breakpoint

показывает следующую точку останова.

Для знакомства с работой отладчика воспользуемся программой из примера 23.1, скомпилированной и скомпонованной без ошибок, но содержащей логическую ошибку. Программа выполняется без выдачи каких бы то ни было сообщений об ошибках, но полученный результат некорректен. Программа подсчитывает строки, слова и символы введенного текста.

Пример 23.1:

Содержащая ошибку программа для подсчета количества строк, слов и символов в тексте.

```
program text;
(* Программа должна подсчитывать строки, слова и символы
   в заданном тексте*)
uses crt;
var c:char; zeile, zeichen, wort:integer;

function istkleinbuchstabe(c:char):boolean;
begin
   istkleinbuchstabe := (c >= 'a') and (c <= 'z');
end;

function istgrossbuchstabe(c:char):boolean;
begin
   istgrossbuchstabe := (c >= 'A') and (c <= 'Z');
end;

function istbuchstabe(c:char):boolean;
begin
   istbuchstabe := istkleinbuchstabe(c) or
                   istgrossbuchstabe(c);
end;

procedure wortverarbeiten;
begin
    repeat
       read(c);
    until not istbuchstabe(c) or eoln;
end;

procedure ausgabe;
begin
   writeln;
   writeln('Строк:',zeile:6);
   writeln('Слов:',wort:6);
   writeln('Символов:',zeichen:6);
end;

begin   (*********Исполняемая часть*********)
   checkeof := true;
   writeln('Введите текст. Конец ввода по ^Z');
   zeile := 0; zeichen := 0; wort := 0;
```

```
    while not eof do
      begin
        while not eoln do
          begin
            read(c);
            inc(zeichen);
      (* H1 *) if istbuchstabe(c)
                then begin
                       wortverarbeiten;
                       inc(wort);
                     end;
      (* H2 *)
          end;
        inc(zeile);
        readln;
      end;
      ausgabe;
  end.
```

Вначале попробуем обычный процесс тестирования программы без использования отладчика. После запуска Турбо Паскаля исходный текст с помощью клавиши F3 загружается в окно редактирования. Поскольку мы решили вначале не пользоваться отладчиком, заменим окно служебных сообщений окном вывода: клавиша F6 активизирует окно служебных сообщений, а Alt-F6 заменяет его на окно вывода. Такая замена дает то преимущество, что в ходе тестирования программы на экране одновременно будет виден исходный текст и результат работы программы. По ходу тестирования обнаружим, что программа компилируется и компонуется без выдачи сообщений об ошибке. Однако, результат, к сожалению, оказывается неверным. Ввод текста

```
Das ist eine Zeile
und noch eine
^Z
```

приводит к выдаче следующего результата:

```
Строк:    2
Слов:     8
Символов: 8
```

Очевидно, число строк и слов неверно. К сожалению, и число символов подсчитано программой неправильно. Теперь попытаемся найти ошибку с помощью отладчика.

Вначале можно попробовать выполнить программу по шагам, то есть построчно, воспользовавшись функциональными клавишами F7 (Run/Trace into) или F8 (Run/Step over). При нажатии клавиши F7 под строкой begin в исполняемой части программы появляется прямоугольничек, помечающий точку старта. Отсюда начинаются операторы программы. Различие в F7 и F8 заключается в том, что при нажатии клавиши F7 (Trace into subroutines) вызванные процедуры и функции выполняются по шагам, а при нажатии F8 (step over subroutines) вызванные процедуры обходятся. Если последовательно нажимать клавишу F7, прямоугольник курсора перескакивает со строки на строку, показывая, какой оператор будет выполняться следующим. Правда, в нашем случае операторов не так уж много.

Весьма интересно, каково значение переменных в определенной точке. В нашем случае это относится прежде всего к считанным символам с и переменной zeichen.

Для вывода значений таких переменных служит окно служебных сообщений. Вначале нужно указать в этом окне переменные с и zeichen. Далее можно пойти двумя путями. Во-первых, воспользоваться Break/Add watch, в результате чего на экране появится разделенное на три части окно следующего вида:

```
┌──────── Evaluate ──────────┐
│                            │
└────────────────────────────┘
┌──────── Result ────────────┐
│                            │
└────────────────────────────┘
┌──────── New Value ─────────┐
│                            │
└────────────────────────────┘
```

В поле Evaluate записываются переменные, значения которых хотелось бы отследить в процессе отладки.

Но можно пойти более коротким путем. Активизируем с помощью F6 окно служебных сообщений. При этом в нижней строке будет стоять Ins-Add, что показывает, что с помощью клавиши Ins можно добавлять новые записи в окно служебных сообщений. Воспользуемся указанной клавишей дважды для ввода переменных с и zeichen. При этом мы увидим, что окно служебных сообщений расширилось на одну строку. Поскольку пошаговое выполнение программы с помощью F7/F8 может оказаться скучным (но при этом в окне служебных сообщений появятся соответствующие значения с и zeichen), включим в программу точки останова или точки прерывания, в которых отладчик должен будет остановить программу. Точки останова задаются с помощью клавиш Ctrl-F8 в том месте, в котором находится курсор. В нашей программе есть два места, которые представляют интерес для подсчета считанных символов: в строке (* H1 *) символ считывается и регистрируется. В

строке (* H2 *) считывается слово и мы хотим знать, какое значение будет иметь переменная zeichen после этого. Итак, с помощью клавиш Ctrl-F8 установим в этих двух местах точки останова. Функциональные клавиши Ctrl-F8 означают запуск и выполнение или продолжение выполнения программы после ее прерывания вплоть до следующей точки останова.

После того, как мы установили точки останова H1 и H2 (и записали в окно служебных сообщений c и zeichen), запустим программу с помощью Ctrl-F9 и введем следующую строку:

```
<пробел><пробел>abc.^Z
```

Программа отработает до точки H1 и в окне служебных сообщений появятся корректные значения c='', zeichen=1. Поскольку считанный символ не является буквой, процедура wortverarbeiten не вызывается. С помощью Ctrl-F9 продолжим выполнение программы до следующей точки останова, то есть опять до точки H1, поскольку точка H2 была обойдена. Считывается второй пробел и выдаются корректные значения c='' и zeichen=2. После нажатия клавиш Ctrl-F9 программа вновь остановится в точке H1, при этом c='a', zeichen=3. Теперь c является буквой и вызывается процедура wortverarbeiten. Программа останавливается в точке H2 со значениями c='.', но zeichen=3, то есть слово abc считано корректно, но символы при этом не подсчитаны! Итак, ошибка найдена. Процедура wortverarbeiten должна быть переписана правильно:

```
procedure wortverarbeiten;
begin
   repeat
     read(c);
     inc(zeichen);
   until not istbuchstabe(c) or eoln;
end;   (* Обработка слова *)
```

Попробуем ввести

```
<пробел><пробел>abc.^Z
```

и получим следующий результат:

```
Строк:       1
Слов:       1
Символов:    6
```

Обычно в окне служебных сообщений записывают и выражения, значения которых затем выводятся. В нашем случае вместо c

можно было бы записать ord(c). Тогда получим номер считанного символа в таблице кодов ASCII, то есть 32 для двух ведущих пробелов, 46 для последней точки. Для используемых таким образом выражений сохраняются те же правила, которые были указаны в главе для выражений, составленных из констант.

Итак, работа с отладчиком ведется таким образом: Исходный текст помещается в окно редактирования, после чего он начинается проверяться. Экран обычно разбивается на окно редактирования и окно служебных сообщений. Затем отыскиваются критические точки программы, в которых хотелось бы проверить определенную информацию в ходе выполнения программы. Именно с помощью клавиш Ctrl-F8 здесь устанавливаются точки останова. Затем заполняется окно служебных сообщений, в котором будут выдаваться интересующие Вас значения. Это можно сделать либо с помощью клавиш Ctrl-F7, либо активизировав окно служебных сообщений нажатием клавиши F6. Но тогда изменится последняя строка в списке меню, где вновь появится Ins=Add и Del=Delete, то есть нажатием клавиши Insert мы вызываем появление того же самого окна, что и при использовании Ctrl-F7. Теперь можно задать, например, имя переменной, за изменением значения которой в ходе выполнения программы мы хотим проследить. При этом при каждом нажатии клавиши Insert окно служебных сообщений расширяется на одну строку. Это следует иметь в виду в дальнейшем. Если после нажатия клавиши Insert (или Ctrl-F7) ввести имя переменной, скажем, k, в окне служебных сообщений после k появится либо значение k, либо, если k пока не присвоено значения, сообщение k:Unknown identifier. Позже это сообщение будет заменено соответствующим значением k.

После такой подготовки (установка точек останова и подготовка окна служебных сообщений) можно запустить программу с помощью F7 (Trace into) или F8 (Step over) или же запустить ее обычным путем с помощью Ctrl-F9 (прогон до первой точки останова). Затем программа выполняется по шагам с помощью клавиш F7 или F8.

Если Вы проделаете такую процедуру пару раз, Вы быстро привыкните отлаживать новую программу только с помощью отладчика. Это первоклассный инструмент, позволяющий экономить время на поиск ошибок. Автор по своему опыту может сказать, что встроенный отладчик мало помогает только при отладке графических программ, поскольку при работе в графическом режиме окно служебных сообщений на экран не выводится.

Вообще говоря, поиск ошибок, даже если это делается с помощью отладчика, довольно утомительное занятие и в том случае, когда Вы располагаете развитым программным окружением. Если компиляция, компоновка и выполнение программы прошли без выдачи сообщений об ошибках, все же остается определенная доля

скептицизма при оценке полученных результатов. Всегда помните о том, что отладка показывает только наличие ошибок, но не их отсутствие. Программа всегда делает только то, что в нее заложено. И здесь возникает вопрос, то ли Вы заложили в программу, что Вам требовалось. Но это уже относится к сложной проблеме верификации программ, о которой здесь не хотелось бы распространяться.

24. Редактор Турбо

Турбо Паскаль содержит сравнительно комфортабельный редактор, с помощью которого можно записывать текст или манипулировать текстовой информацией. Прежде чем приступить к описанию редактора Турбо, необходимо сделать несколько замечаний по клавиатуре. Когда работают с редактором, нужно различать, должен ли соответствующий нажатой клавише символ принадлежать к тексту или же с его помощью должно осуществляться управление функциями редактора. Для управления редактором используется клавиша <CTRL>, которая должна нажиматься вместе с другими клавишами. В приложении C приведена таблица ASCII-символов, из которой видно, с помощью каких клавиш получаются первые 32 символа. Для краткости будем писать

^K, что равносильно <CTRL>-K,
^KD,что равносильно <CTRL>-K <CTRL>-D.

При вводе второго символа клавиша <CTRL> может быть нажата или нет.

Редактор вызывается с помощью
- функции Edit основного меню или
- функциональных клавиш Alt-E.

Выйти из редактора можно
- по какой-нибудь функциональной клавише (например, нажав F10 для активизации основного меню) или
- <CTRL>-KD.

Подлежащий обработке текст задается с помощью
- функции New в меню файлов (автоматически задается имяNONAME.PAS);
- функциональной клавиши F3 (= Load в меню файлов);
- задания имени при вызове TURBO.EXE.

Обработанный текст сохраняется
- при нажатии клавиши F2 (= Save в меню файлов) или
- путем записи в каталог файлов под некоторым новым именем.

Функциональная клавиша F1 обеспечивает помощью при работе в любых режимах и обеспечивает получение пояснений по всем

функциям редактора, если Вы работаете в режиме редактора. Команда редактора ^F1 обеспечивает особый комфорт. Если подвести курсор к имени стандартной процедуры, функции, модуля или типа, применение команды ^F1 позволит Вам получить все необходимые разъяснения по поводу такой процедуры, функции и пр. Такая возможность не может не порадовать. Но в то же время такой сервис является причиной, почему файл TURBO.HLP занимает 150 Кбайт и непременно включается в тот же каталог или на ту же системную дискету, что и TURBO.EXE (см. раздел 3.1).

При работе с редактором доступны оба состояния "вставить" (insert) и "переписать" (overwrite). Вставка означает, что символы вставляются в том месте, где находится курсор. Перезапись означает, что символ, на который указывает курсор, заменяется. Текущее состояние указывается в строке заголовка. Переход из одного состояния в другое осуществляется нажатием клавиш ^V.

Строка заголовка (называемая также строкой состояния) в редакторе имеет следующий вид:

Line Col. Insert Indent Tab X:имя_файла.тип

где

- Line - строка, в которой находится курсор
- Col. - столбец, в котором находится курсор
- Insert - редактор находится в состоянии "вставить". При вводе находящийся справа от курсора текст сдвигается вправо. С помощью клавиш Ins или ^V можно перейти в состояние Overwrite
- Indent - активизируется функция табуляции Indent, то есть при нажатии клавиши RETURN курсор перемещается в позицию, следующую за самым левым символом предыдущей строки. При нажатии клавиш ^OI функция отключается.
- Tab - включить табуляцию, то есть при нажатии клавиши Tab курсор перейдет в следующую точку установки табуляции. Функция включается и отключается с помощью ^OT.
- X:имя_файла.тип задает дисковод и имя файла, в котором записан загруженный в редактор текст. Если текст не имеет имени, высвечивается стандартное имя NONAME.PAS.

Перечислим вначале комбинации клавиш с клавишей <CTRL>, выполняющие чисто управляющие функции, при этом будем руководствоваться назначением клавиш, принятым в PC-DOS.

Команды управления курсором

На символ влево	^S	Левый край строки	^QS
На символ вправо	^D	Правый край строки	^QD
На строку вверх	^E	Верхняя часть экрана	^QE
На строку вниз	^X	Нижняя часть экран	^QX
На слово влево	^A	Начало текста	^QR
На слово вправо	^F	Конец текста	^QC
Сдвиг строк вверх	^Z	Начало блока	^QB
Сдвиг строк вниз	^W	Конец блока	^QK
На страницу вперед	^C	Последняя позиция	
На страницу назад	^R	курсора	^QP

Команды добавления и стирания

Добавление вкл./выкл.	^V
Стереть символы слева от курсора	^H
Стереть символы после курсора	^G
Стереть слово справа от курсора	^T
Добавить строку	^N
Стереть строку	^Y
Удалить все до конца строки	^QY

Если курсор стоит в конце строки, ^T означает удаление следующего знака перевода строки, то есть следующая строка будет добавлена к текущей.

Команды обработки блоков

Пометить начало блока	^KB	Переместить блок	^KV
Пометить конец блока	^KK	Удалить блок	^KY
Пометить слово	^KT	Записать блок на диск	^KW
Блок видимый/невидимый	^KH	Считать блок с диска	^KR
Скопировать блок	^KC	Выдать блок на печать	^KP

Различные команды

Поиск	^QF	Автоматическую табу-	
Поиск с заменой	^QA	ляцию вкл./выкл.	^QI
Повторить последний		Табулятор	^I
поиск	^L	Добавить символ	
Прерывать команду	^U	<CTRL>	^P
Конец работы с редак-		Конец работы	
тором с сохранением	^KS	с редактором	^KD

Хотя почти все приведенные выше команды совершенно понятны, сделаем все же несколько дополнительных замечаний.

Команда: ^QF Поиск
Параметры: Ведется поиск по тексту, начиная с позиции курсора.
b Поиск ведется начиная с позиции курсора в обратном направлении (к началу текста)
n Поиск n-ного появления искомого фрагмента, начиная с позиции курсора
u Игнорировать различия между большими и маленькими буквами
w Поиск по целому слову, а не по частям слов.

Команда: ^QA Поиск с заменой
Параметры: Те же, что и для ^QF, и дополнительно:
g Поиск по всему тексту
n Замена без запроса подтверждения
Параметры могут комбинироваться, например, b3w.

Команда: ^P
Параметры: Позволяет вставлять в текст и символы <CTRL>, которые сами являются обычно командой.
Пример: После ^P команда ^L означает вставку символа 12 по таблице кодов ASCII (form feed) в текст.

Для тех, кто привык работать с редактором WORDSTAR, приятно будет видеть, что здесь присутствуют те же команды. Редактор Турбо является, по сути, мини-WORDSTAR. Но все же управлять редактором с помощью клавиши <CTRL> утомительно. Большинство клавиатур имеют функциональные клавиши, так что целесообразно назначить этим клавишам наиболее употребительные команды. Для этого следует запустить программу TINST и выбрать команды редактора. Команды редактора появятся на экране в виде трех столбцов: слева будет стоять имя команды, в центре под подзаголовком Primary описанное выше распределение клавиш, под подзаголовком Secondary (быть может уже существующее) второе назначение клавиши. Теперь можно изменить назначение клавиш по своему усмотрению. Сделанное в столбце Secondary назначение имеет более высокий приоритет по сравнению со столбцом Primary.

Можно использовать и функциональные клавиши, например, F10, которые сами являются командами для системы Турбо. Тогда F10 будет иметь вновь присвоенное значение только внутри редактора и не будет командной клавишей (но при выходе из режима

редактора F10 вновь приобретет свое значение "Возврат в основное меню").

Для работающего под PC-DOS Турбо Паскаля уже сделаны следующие назначения:

Действие	Клавиша PC	Команда
На символ влево	←	^S
На символ вправо	→	^D
На слово влево	^ ←	^A
На слово вправо	^ →	^F
На строку назад	↑	^E
На строку вперед	↓	^X
На страницу назад	PgUp	^R
На страницу вперед	PgDn	^C
Левый край строки	Home	^QS
Правый край строки	End	^QD
Верхняя граница экрана	^Home	^QE
Нижняя граница экрана	^End	^QX
Начало текста	^PgUp	^QR
Конец текста	^PgDn	^QC
Добавление вкл./выкл.	Ins	^V
Стереть символы слева от курсора	Del	
Пометить начало блока	F7	^KB
Пометить конец блока	F8	^KK
Табулятор	TAB	^I

Примечания относительно версии 5.0

Строка состояния расширяется (добавляются две функции):

Line Col. Insert Indent Tab Fill Unindent *C:NONAME.PAS

Fill - функция, автоматически превращающая следующие друг за другом пробелы в оптимальную последовательность знаков табуляции и пробелов. В результате может появиться возможность весьма значительно сжать текст. Эта функция включается и отключается с помощью ^QF.

Unindent - если слева от курсора находятся только пробелы, при нажатии клавиши пробела курсор перемещается в позицию, следующую за самым левым символом предыдущей строки (итак, эта функция является

противоположностью функции Indent). Функция включается и отключается с помощью ^QU.

*C:NONAME.PAS - имя находящегося в редакторе файла обозначается символом *, если текст с момента последнего изменения более не сохранялся (клавиша F2).

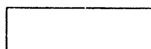
Кроме того, в версии 5.0 существуют две дополнительные команды для работы с блоками:

^KI Помеченный блок смещается на одну позицию вправо.
^KU Помеченный блок смещается на одну позицию влево.

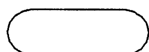
Приложение А

Синтаксис Турбо Паскаля

Синтаксис описывает грамматику языка. Синтаксис - это правила, согласно которым можно формулировать текст на этом языке. Для описания таких правил здесь используется форма синтаксических диаграмм. Синтаксическая диаграмма является направленным графом, при прохождении которого автоматически строится синтаксически правильная программа. При этом следует различать два типа символов:



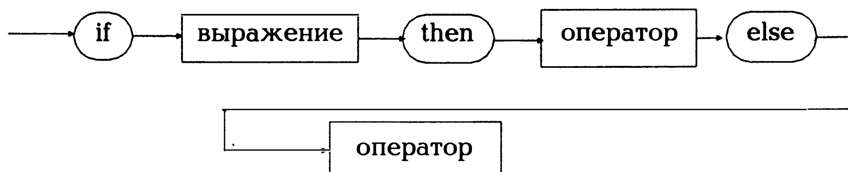
содержит некоторое синтаксическое понятие,



содержит символы, входящие в исходный текст.

Синтаксические диаграммы отражают, обычно, лишь небольшую часть синтаксиса. Каждое понятие, заключенное в прямоугольник, требует, в свою очередь, некоторого определения. Например:

оператор if:

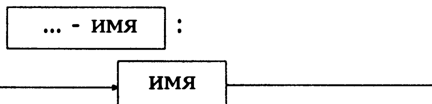


Использованные для описания оператора if понятия "выражение" и "оператор" должны, естественно, определяться в свою очередь другой синтаксической диаграммой, в то время как if, then и else непосредственно включаются в текст программы.

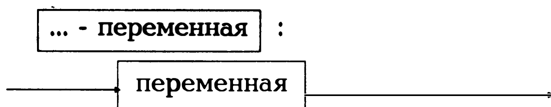
В настоящей книге правила синтаксиса в такой форме приведены в соответствующих местах. При этом должны были использоваться понятия, описание которых следовало бы искать в другом месте. Тогда, конечно, ссылка на (9-7) означает, что имеется в виду 7-е определение из главы 9. В настоящем приложении разбросанные по всей книге диаграммы собраны вместе и упорядочены по алфавиту в соответствии с синтаксическими понятиями, которые они описывают.

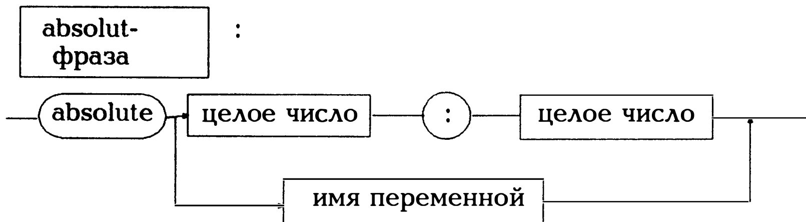
•Следует еще раз отметить, что синтаксические диаграммы передают значение (семантику) текста программы лишь весьма относительно. Для того, чтобы как-то исправить это положение, в

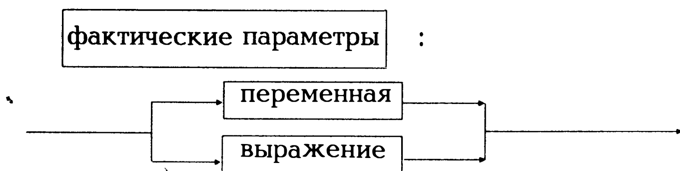
синтаксических диаграммах к невыразительному понятию "имя" добавляются уточнения, такие как "имя функции" и пр. Итак:

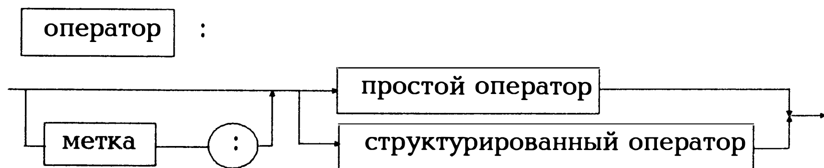


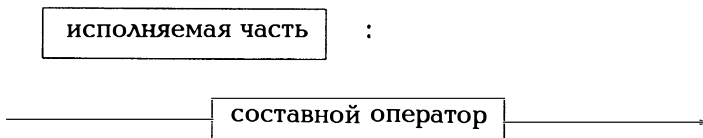
а также

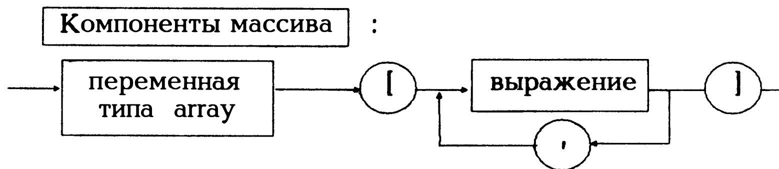


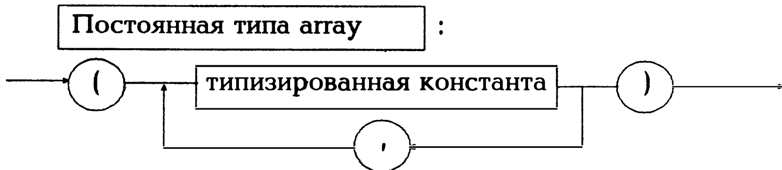


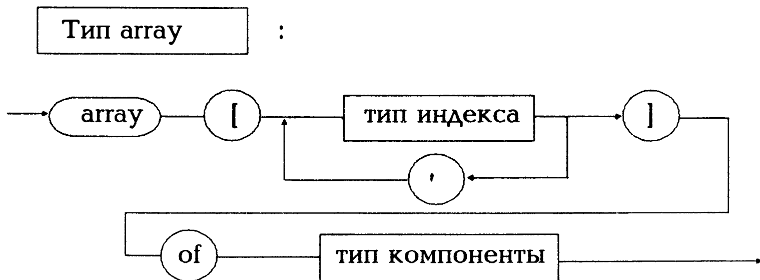


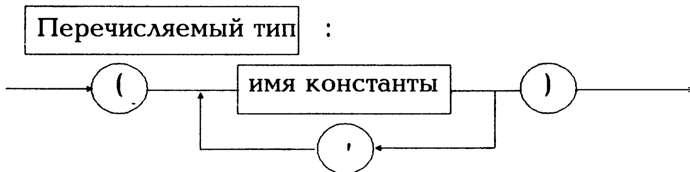


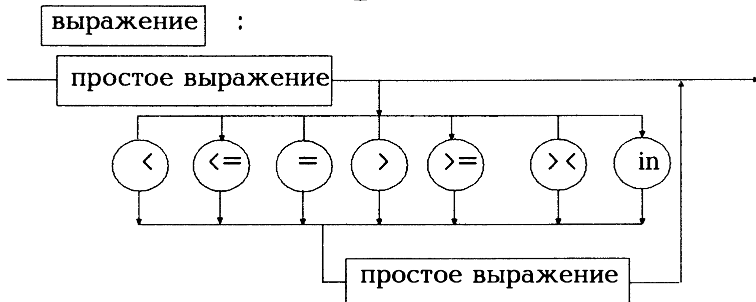



Компоненты массива :
переменная типа array
[
выражение
,
]

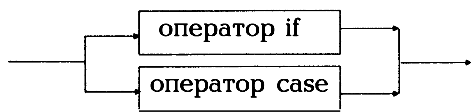

Постоянная типа array :
(
типизированная константа
,
)


Тип array :
array
[
тип индекса
,
]
of
тип компоненты

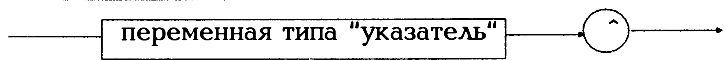

Перечисляемый тип :
(
имя константы
,
)


выражение :
простое выражение
<
<=
=
>
>=
><
in
простое выражение

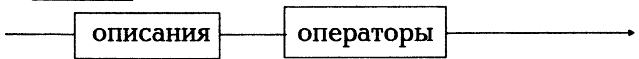
условный оператор :



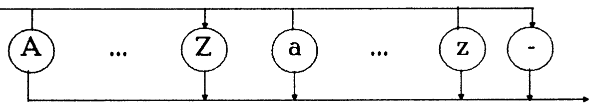
Ссылочная переменная :



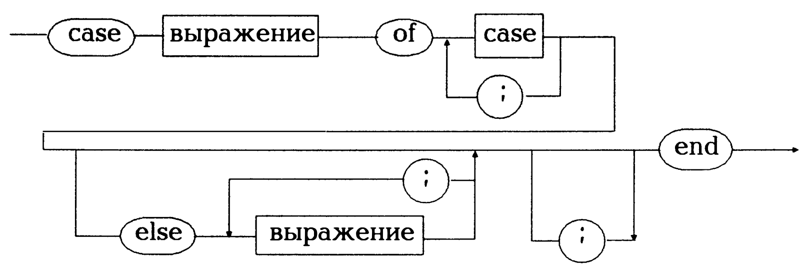
блок :



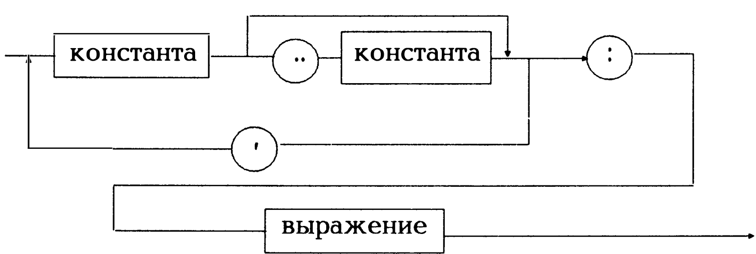
буквы :

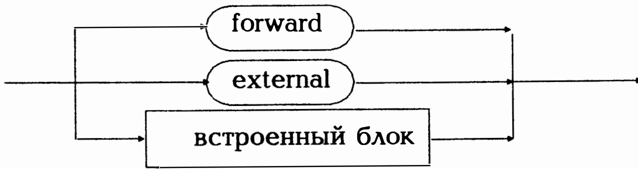


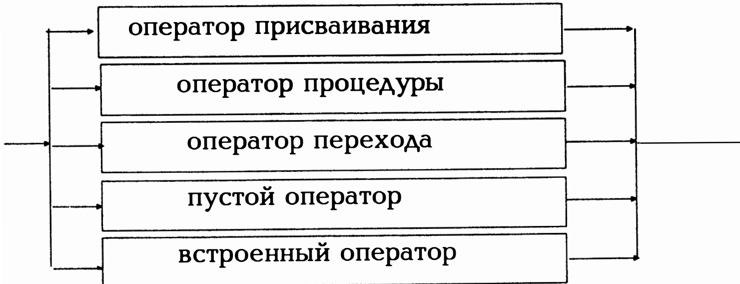
оператор case :

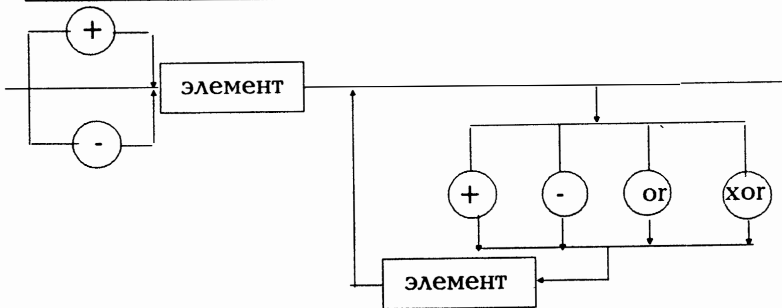


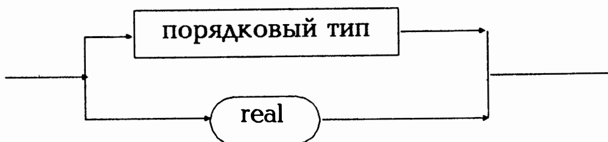
case :




директива :
forward
external
встроенный блок


простой оператор :
оператор присваивания
оператор процедуры
оператор перехода
пустой оператор
встроенный оператор


простое выражение :
+
-
элемент
+
-
or
xor
элемент


простой тип :
порядковый тип
real

коэффициент :

переменная

@

имя процедуры

имя функции

вызов функции

автоопределение типа

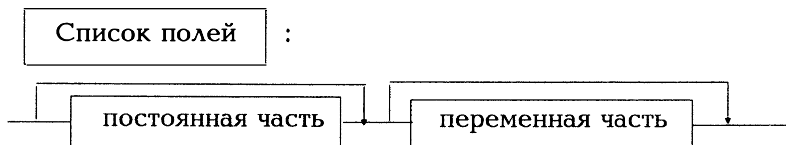
множество

(выражение)

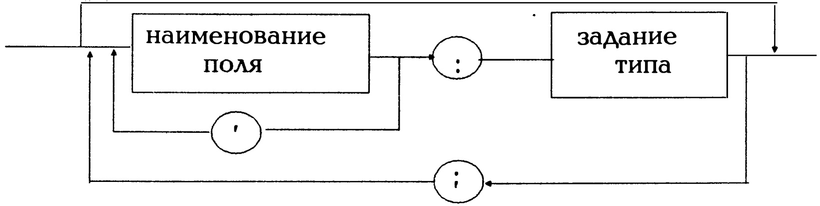
not коэффициент

целое число без знака

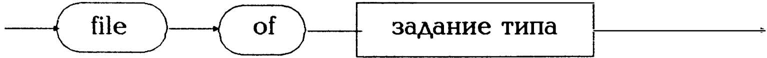
вещественное число без знака



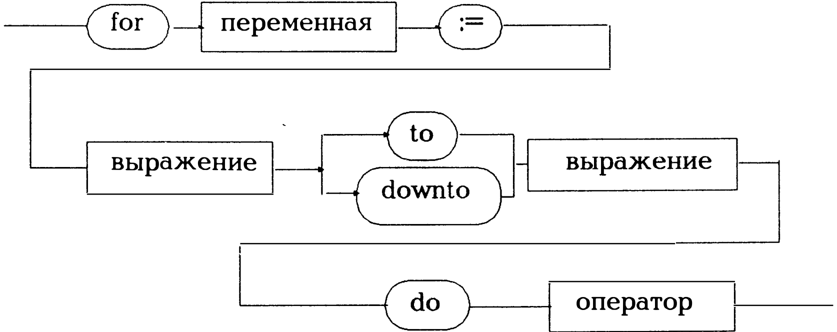
Постоянная часть :



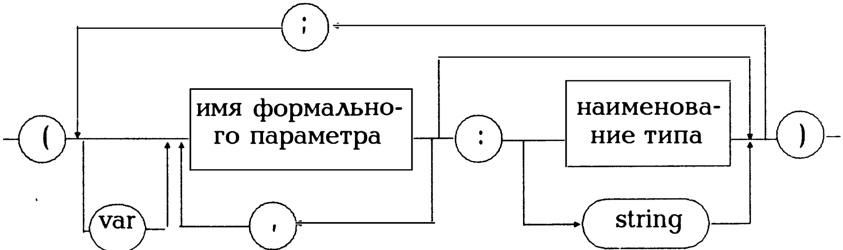
Тип file :

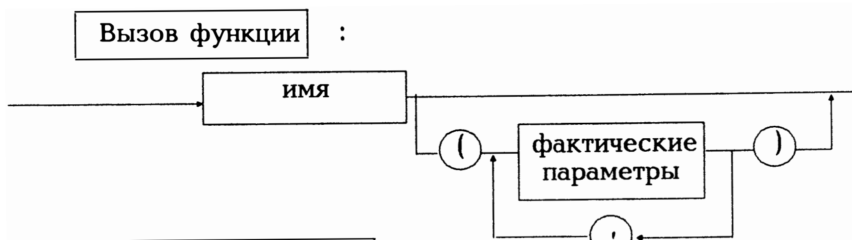


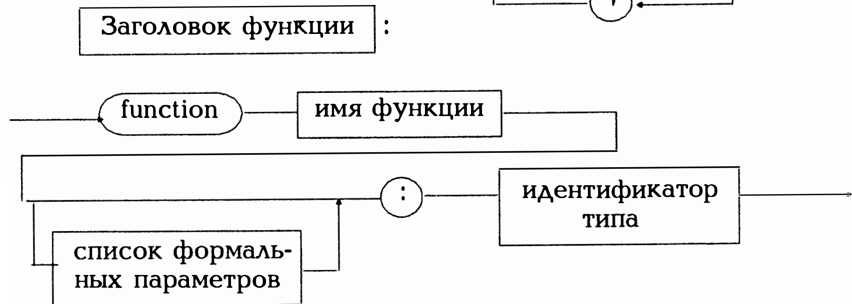
Оператор for :

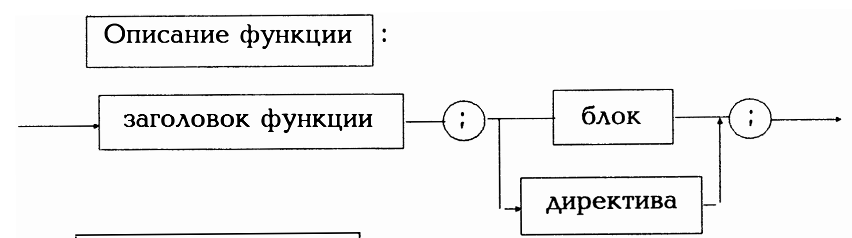


Список формальных параметров :

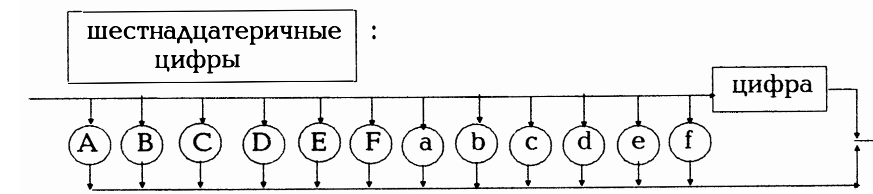



Вызов функции
:
имя
(
фактические
параметры
)
,

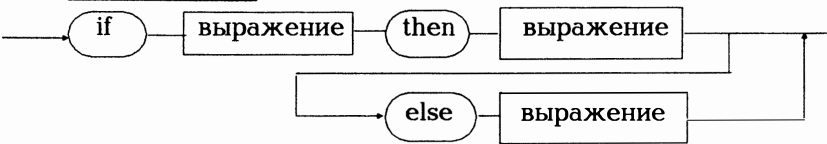

Заголовок функции
:
function
имя функции
:
идентификатор
типа
список формаль-
ных параметров


Описание функции
:
заголовок функции
;
блок
;
директива

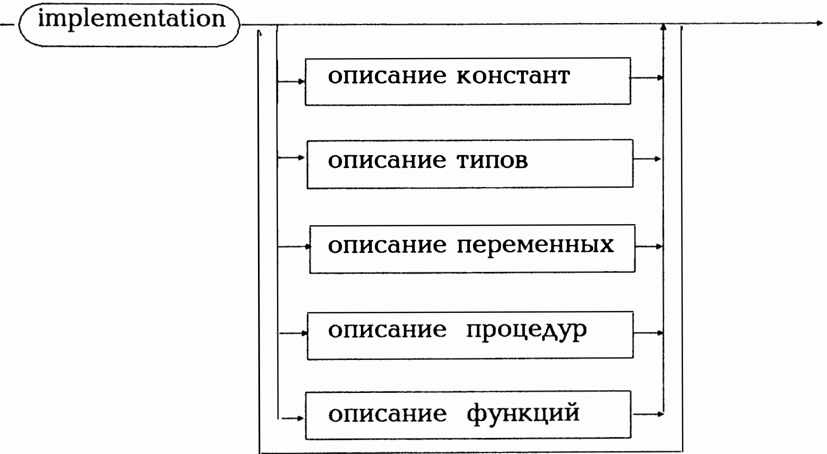
переменная-целая
:
имя переменной
имя типизированной
константы


шестнадцатеричные
цифры
:
цифра
A
B
C
D
E
F
a
b
c
d
e
f

оператор if :



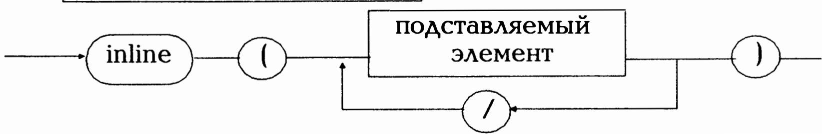
реализующая часть :



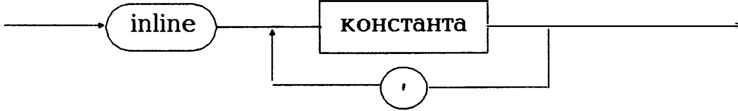
Тип индекса :

Инициализирующая часть :

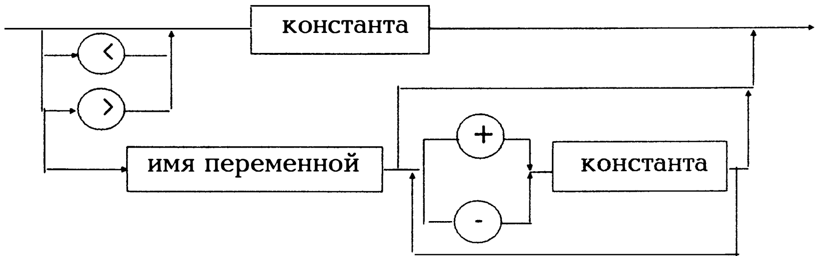
подставляемый оператор :



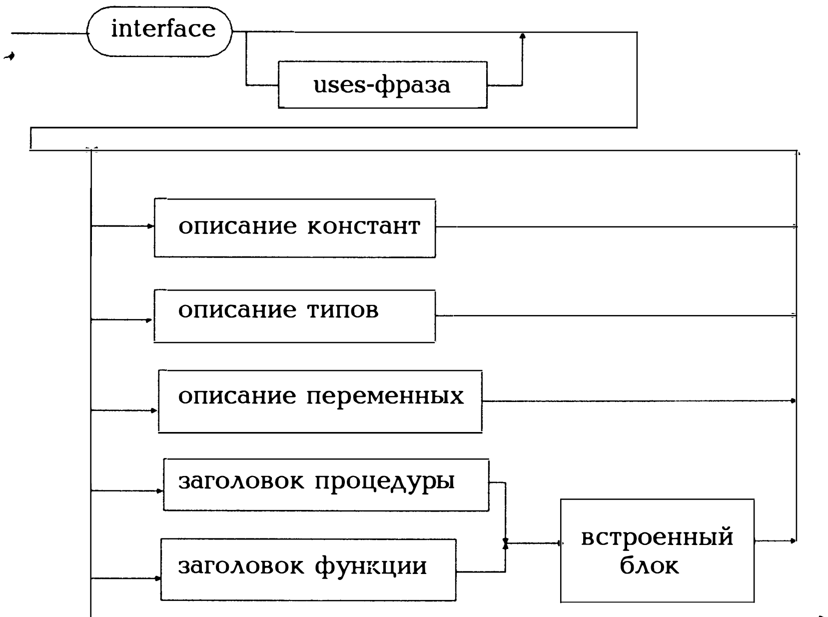
подставляемый блок :



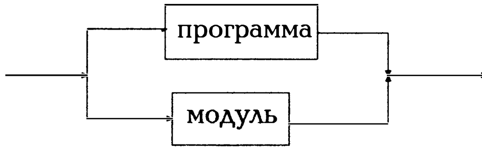
подставляемый элемент :



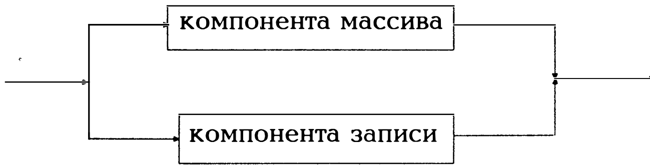
Интерфейсная часть :



Компилируемый блок :

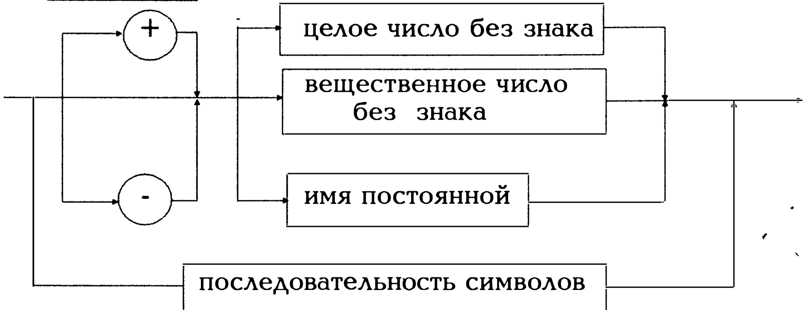


переменные-составляющие :

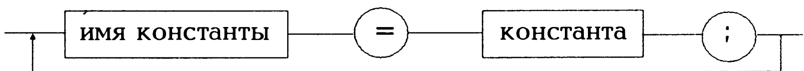


Тип компоненты :

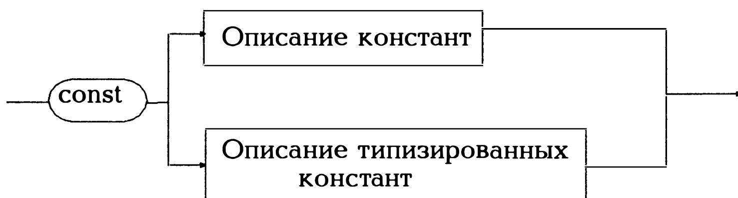
константа :



описание константы :

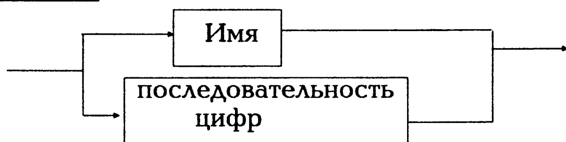


Соглашение о константах :

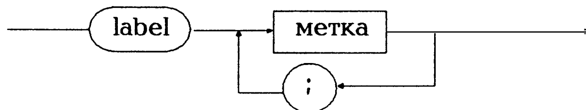


пустой оператор :

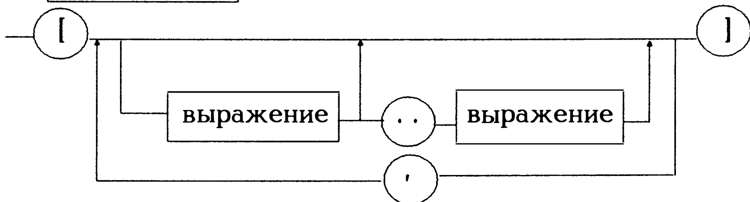
Метка :



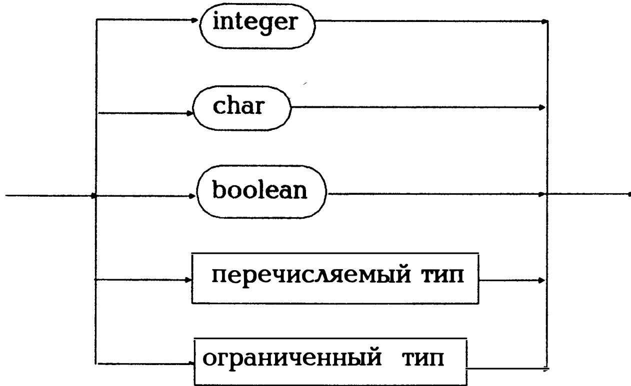
Описание метки :

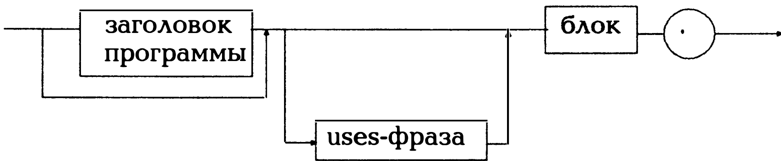


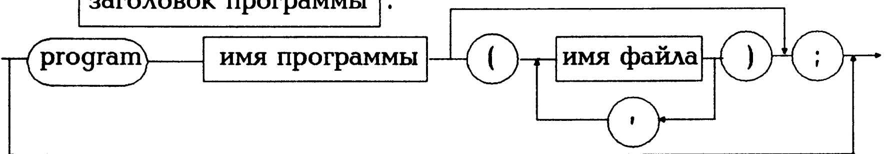
Множество :

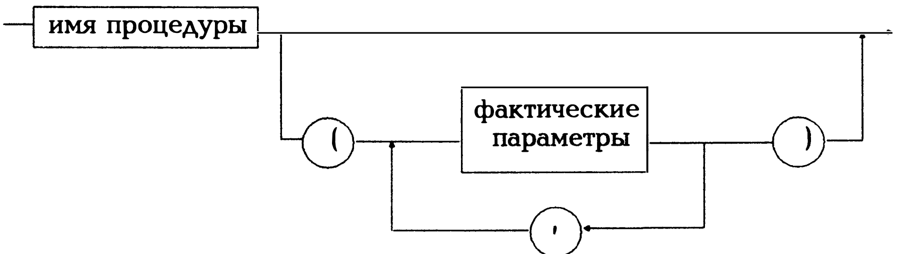


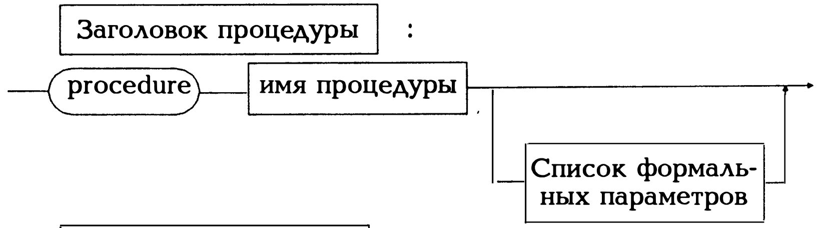
имя :

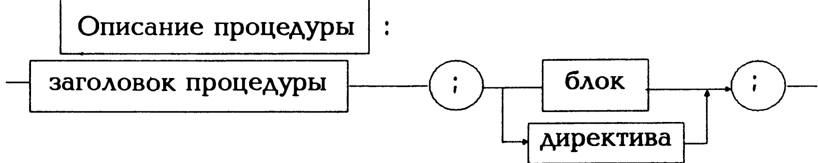

Порядковый тип :
integer
char
boolean
перечисляемый тип
ограниченный тип

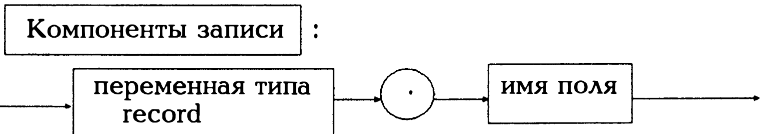

программа :
заголовок программы
блок
.
uses-фраза

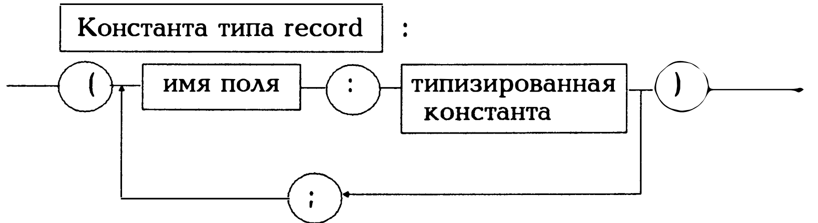

заголовок программы :
program
имя программы
(
имя файла
)
;
,

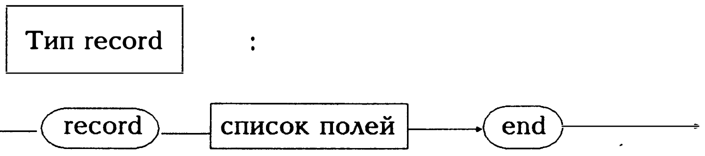

Оператор процедуры :
имя процедуры
(
фактические параметры
)
,

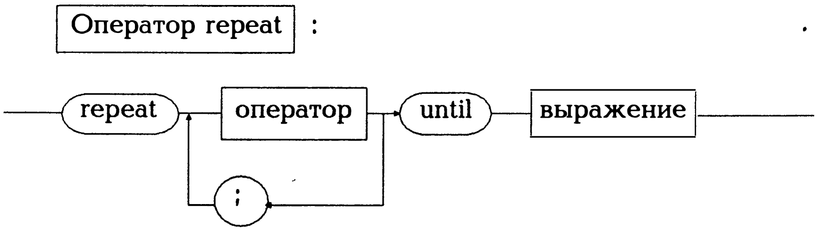

Заголовок процедуры :
procedure
имя процедуры
Список формальных параметров


Описание процедуры :
заголовок процедуры
;
блок
директива
;

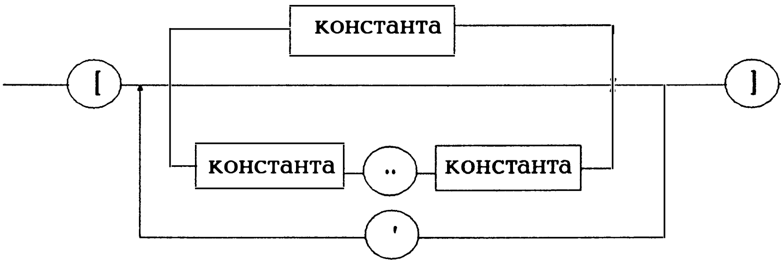

Компоненты записи :
переменная типа record
.
имя поля


Константа типа record :
(
имя поля
:
типизированная константа
)
;

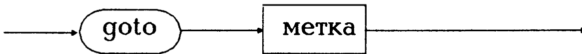

Тип record :
record
список полей
end


Оператор repeat :
repeat
оператор
until
выражение
;

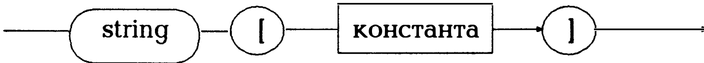
Константа типа set :



Оператор перехода :

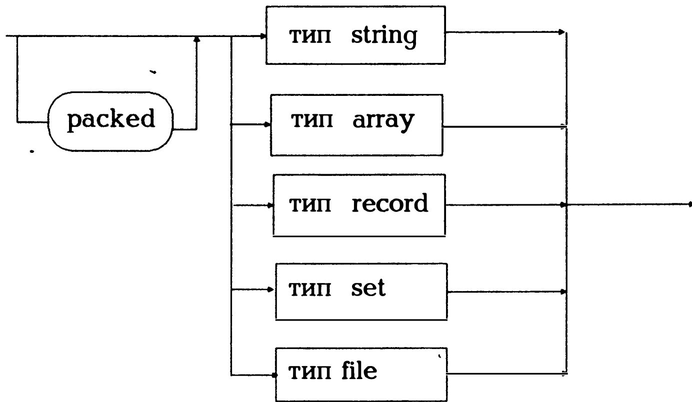


Тип string :



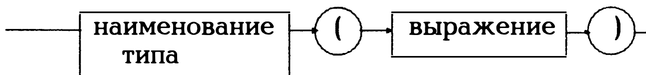
структурированный оператор :

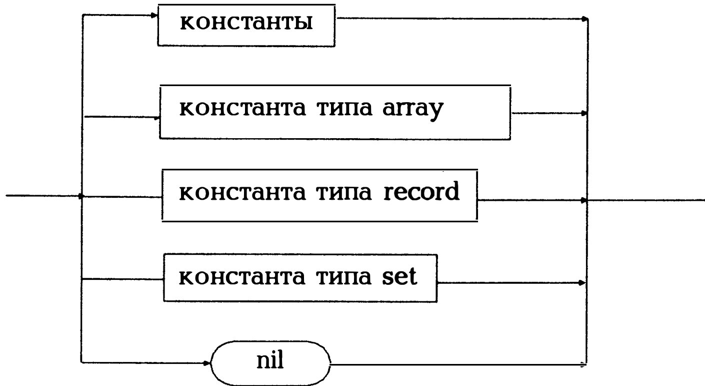
Типы структурированных данных :

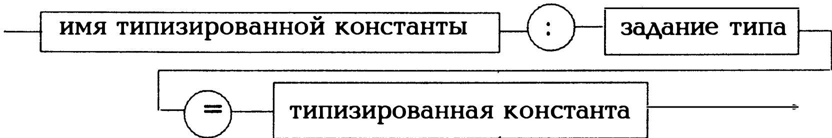


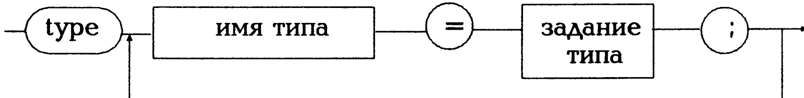
Задание типа :

автоопределение типов :




типизированные константы :
константы
константа типа array
константа типа record
константа типа set
nil

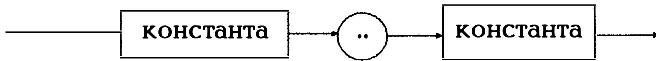

описание типизированных
констант :
имя типизированной константы
:
задание типа
=
типизированная константа


Описание типов :
type
имя типа
=
задание
типа
;

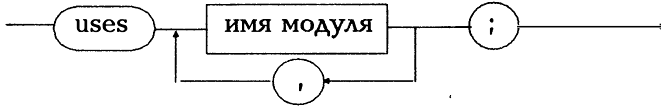
Модуль :
заголовок модуля
интерфейсная часть
реализующая
часть
инициализирующая
часть
.

заголовок модуля :
unit
имя модуля
;

ограниченный тип :

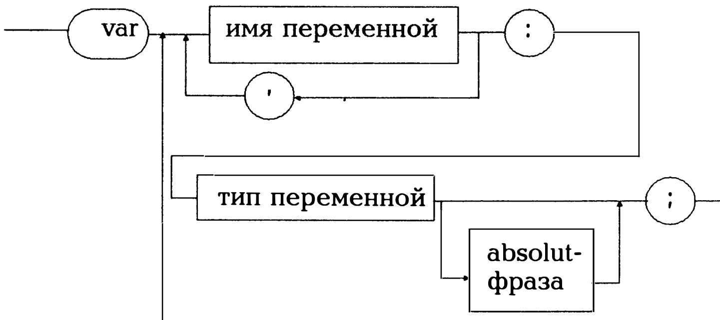


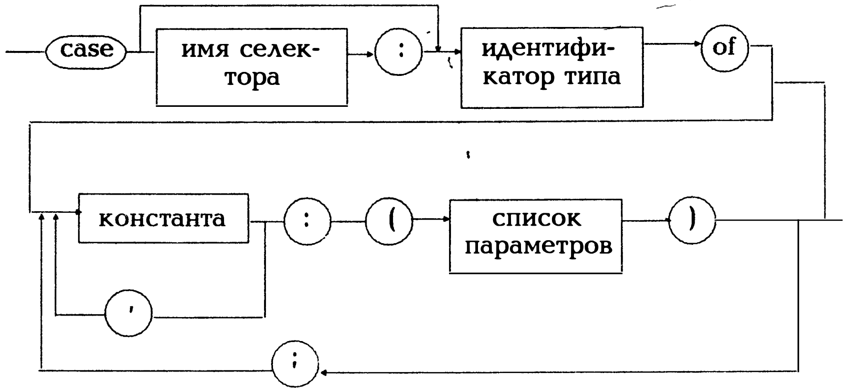
uses-фраза :

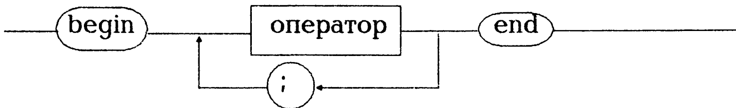


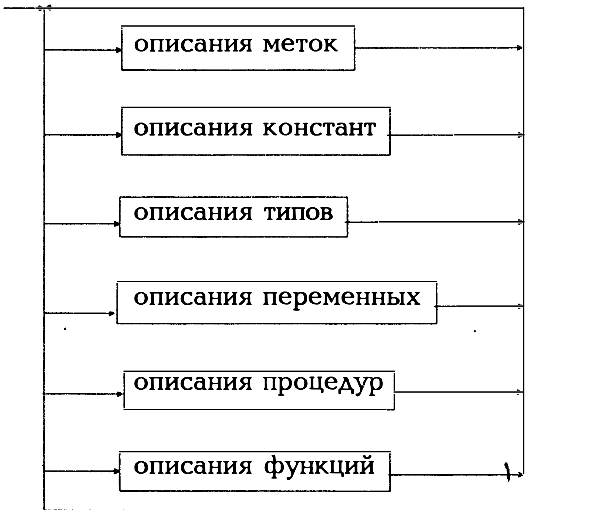
переменная :

описание переменной :

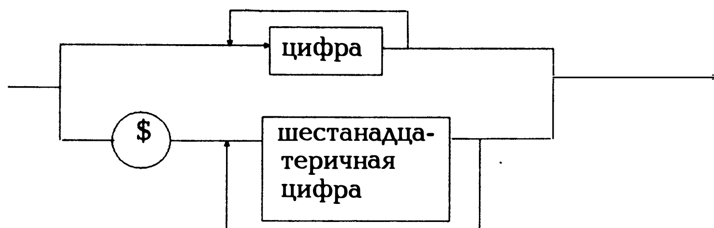



Переменная часть :
case
имя селек-
тора
:
идентифи-
катор типа
of
константа
,
:
(
список
параметров
)
;

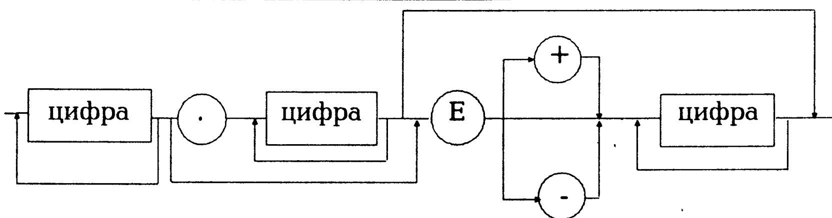

составной оператор :
begin
оператор
end
;


описания :
описания меток
описания констант
описания типов
описания переменных
описания процедур
описания функций

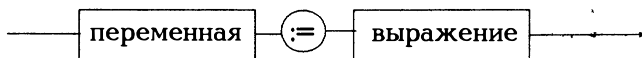
целое число без знака :



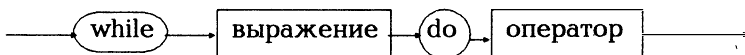
вещественное число без знака



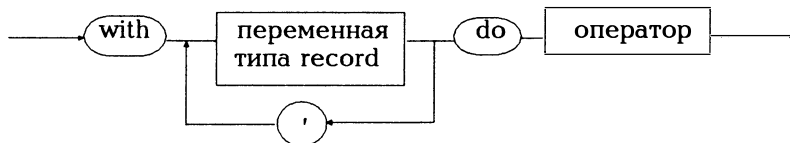
оператор присваивания :

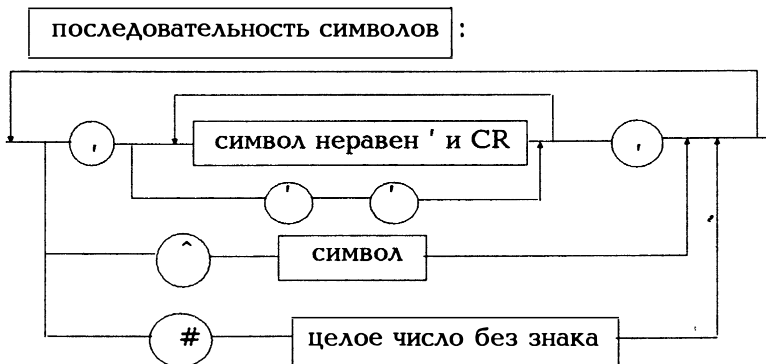


Оператор while :



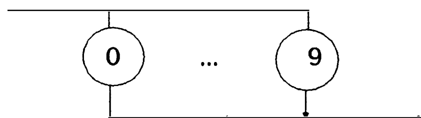
Оператор with :





Тип "указатель" :

цифры :



ПРИЛОЖЕНИЕ В
Операции

Выражения состоят из операндов и операций. Операндами являются константы, переменные и вызовы функций следующих типов:

Исчисление выражений:
1. Операции уровня i выполняются перед выполнением операций уровня j>i (чем ниже уровень, тем теснее связь)
2. Операции одного уровня выполняются последовательно слева направо
3. Операции, заключенные в круглые скобки, выполняются перед операциями, находящимися за скобками

Одноместные (унарные) операции имеют один операнд:

Операция	Тип операнда	Тип результата	Значение
+	integer	integer	Тождество
+	real	real	Тождество
-	integer	integer	Перемена знака
-	real	real	Перемена знака
@	переменная	адрес	адрес ...

Двуместные (бинарные) операции имеют два операнда:

Уровень	Операция	Тип операнда	Тип результата	Значение
0	not	integer	integer	Побитное отрицание
	not	boolean	boolean	Отрицание
1	*	real	real	Умножение
	*	integer	integer	Умножение
	*	integer,real	real	Умножение
	*	множество	множе-ство	Пересечение множеств
	/	integer	real	Деление
	/	real	real	"
	/	integer,real	real	"
	div	integer	integer	Деление без остатка
	mod	integer	integer	Деление по модулю (остаток от деления)
	and	integer	integer	Побитная операция И
	and	boolean	boolean	Конъюнкция, И, ^
	shl	integer	integer	Сдвиг влево
	shr	integer	integer	Сдвиг вправо

2	+	integer	integer	Сложение
	+	real	real	"
	+	real,integer	real	"
	+	string	string	Сцепление
	+	множество	множество	Объединение множеств
	-	integer	integer	Вычитание
	-	real	real	"
	-	real,integer	real	"
	-	множество	множество	Разность множеств
	or	integer	integer	Побитная операция ИЛИ
	or	boolean	boolean	Дизъюнкция, ИЛИ,
	xor	integer	integer	Побитное исключающее ИЛИ
	xor	boolean	boolean	Сложение по модулю, исключающее ИЛИ
3	=	простой тип	boolean	Равно
		string	boolean	
		множество.	boolean	
		указатель	boolean	
	<>	простой тип	boolean	Неравно
		string	boolean	
		множество	boolean	
		указатель	boolean	
	<	простой тип	boolean	Меньше
		string	boolean	
	<=	простой тип	boolean	Меньше или равно
		string	boolean	
		множество	boolean	
	>	простой тип	boolean	Больше
		string	boolean	
	>=	простой тип	boolean	Больше или равно
		string	boolean	
		множество	boolean	
	a in b	a элементов множества b	boolean	Принадлежность

ПРИЛОЖЕНИЕ C

Набор символов ASCII

DEC /	/ HEX	Ctrl-Ch	0 00	Ctrl-Ch	16 10	32 20	48 30	64 4	80 50	96 60	112 70
0	0	^@	NUL	^P	DLE		0	@	P	'	p
1	1	^A	SOH	^Q	DC1	!	1	A	Q	a	q
2	2	^B	STX	^R	DC2	"	2	B	R	b	r
3	3	^C	ETX	^S	DC3	#	3	C	S	c	s
4	4	^D	EOT	^T	DC4	$	4	D	T	d	t
5	5	^E	ENQ	^U	NAK	%	5	E	U	e	u
6	6	^F	ACK	^V	SYN	&	6	F	V	f	v
7	7	^G	BEL	^W	ETB	'	7	G	W	g	w
8	8	^H	BS	^X	CAN	(	8	H	X	h	x
9	9	^I	HT	^Y	EM	)	9	I	Y	i	y
10	A	^J	LF	^Z	SUB	*	:	J	Z	j	z
11	B	^K	VT	^[	ESC	+	;	K	[	k	{
12	C	^L	FF	^\	FS	,	<	L	\	l	|
13	D	^M	CR	^]	GS	-	=	M	]	m	}
14	E	^N	SO	^^	RS	.	>	N	^	n	~
15	F	^O	SI	^_	US	/	?	O	_	o	DEL

Первые 32 символа таблица кодов ASCII являются управляющими символами, используемыми при работе с внешними устройствами и при передаче данных. Важнейшими из таких символов являются:

0	NUL	Нулевой оператор (без операции)
7	BEL	Звуковой сигнал (звонок)
8	BS	Пробел
9	HT	Горизонтальная табуляция
10	LF	Перевод строки
11	VT	Вертикальная табуляция
12	FF	Перевод страницы (подача бланка)
13	CR	Возврат каретки
27	ESC	Переключение (переход)

Для большинства управляющих символов не существует специальной клавиши. Они могут генерироваться с помощью клавиши <CTRL> в сочетании ее с другой клавишей, что в первой позиции столбца Ctrl-Ch. кратко обозначается через ^. Такое сочетание клавиш часто используется в качестве управляющих символов, то есть команд, при работе со служебными программами. В Турбо Паскале они используются как команды редактора (см. раздел 24).

Приведенная выше таблица содержит множество символов кода ASCII в принятой в США форме. В других странах использеют-

ся множества символов с несколько другим их расположением. Для немецкой версии таблицы кодов ASCII

[= Ä \= Ö]= Ü {= ä |= ö }= ü ~= β @ = §

Обычно код ASCII является 7-битным кодом, то есть с его помощью можно представить 128 символов. Восьмой бит может использоваться как бит четности. Но в некоторых персональных ЭВМ используется 8-разрядный код. Тогда имеется 256 символов. Первые 128 символов совпадают с приведенными выше. Символы 128-255 используются для представления специальных символов, таких как $\pm$ $\sum$ Ω .

К этим специальным символам относятся и умлауты. Примером может служить код для IBM-PC. Тогда для строковой переменной допустимы все 256 символов, например, ' 1/2 μm', хотя переменная не может так называться.

Вообще говоря, следует подчеркнуть, что символы 128-255 из набора 8-битовых кодов не должны использоваться, если предполагается передавать этот текст на компьютер с другой кодировкой.

Символ 27 таблицы кодов ASCII (ESCAPE) имеет особое значение. Он используется для управления печатающими устройствами и дисплеями. Если на внешнее устройство поступает символ 27 по таблице кодов ASCII, то он или последующие символы интерпретируются как управляющие (см. пример 17.7).

ПРИЛОЖЕНИЕ D

Директивы компилятора

Было бы целесообразно иметь возможность влиять на ход компиляции, то есть какие-то операции выполнять, а какие-то пропускать. В Турбо Паскале для этого предусмотрены:

- функция меню Options/Compiler;
- /$ в качестве параметра командной строки (см.главу 20);
- директивы компилятору, включаемые в исходный текст.

Такие директивы имеют форму специальных комментариев:

{$директива} или (*$директива*)

Имейте в виду, что перед и после символа $ не должен стоять пробел. Директивы можно подразделить на три типа:

- директивы переключения (+ - включить, - - выключить);
- директивы передачи параметров, содержащие в себе какие-то данные;
- условные директивы, согласно которым в зависимости от некоторого условия куски исходного текста не компилируются.

Директивы переключения

Символ "плюс" (+) означает, что включается (активизируется) соответствующее свойство компилятора; символ "минус" (-) означает, что это свойство отключается. Но впредь нужно иметь в виду, что каждая директива может иметь одно из двух состояний в качестве стандартной установки (так называемое значение по умолчанию, см. Options/Compiler в основном меню). Итак, директива компилятору требуется лишь в том случае, когда нужно изменить стандартную установку. Большинство директив могут в программе многократно включаться и отключаться. Стандартную установку можно вызвать из исходного текста с помощью Ctrl-F7.

Кроме того, следует различать глобальные и локальные директивы. Глобальная директива действует в течение всего процесса компиляции. Глобальные директивы всегда должны стоять в начале программы (после program и unit). Локальные директивы могут помещаться в любом месте исходного текста, где может стоять комментарий. Несколько директив переключения можно заключить в одни скобки комментария в виде списка (например, (*$B+,R-,S-*)). При этом нельзя пользоваться пробелами, то есть нельзя записать (*$B+, R-, S-*).

{$B±} Вычисление логических выражений Локальная
По умолчанию: B-
При B+ логическое выражение оценивается полностью. При B- оценка обрывается, если значение истинности выражения установлено (соответствующий пример приведен в разделе 9.3).

{$D±} Отладочная информация Глобальная
По умолчанию: D+
При D+ компилятор генерирует дополнительную информацию для поиска ошибок, то есть таблицы с номерами строк для каждой стандартной программы, причем показывается связь между адресами отдельных комнад и исходным текстом. При компиляции модулей такая дополнительная информация помещается в соответствующий файл TPU. Если программа компилируется в виде файла EXE на дискете, при использовании директивы *$T+* генерируется файл TPM, из которого с помощью программы TPMAP.EXE можно сгенерировать файл .MAP для символьного отладчика.

{$E±} Эмулятор По умолчанию: E+ Глобальная
Эта директива существует только для версии 5.0. В случае (*$N+*) и (*$E+*) с помощью эмулятора воспроизводятся пять типов real для 8086. Сопроцессор 8087 не требуется. При использовании директив (*$N+*) и (*$E-*) подключаются лишь несколько

стандартных программ для управления 8087, который должен существовать.

{$F±} Использовать дальний вызов (Far Calls) Локальная
По умолчанию: F-
Иногда осуществляются вызовы процедур и функций за пределами кодового сегмента. Здесь речь идет о так называемых far calls. Обычно компилятор разрешает (*F-*) через тип соответствующего вызова. Если заголовок процедуры или функции ограничен с двух сторон (*F+*)...(*F-*), всегда генерируются вызовы far и соответствующий обратный переход.

Примечание относительно версии 5.0.

Процедуры или функции, являющиеся параметрами, должны компилироваться с (*F+*) (см. пример 12.2).

{$I±} Контроль ввода/вывода Локальная
По умолчанию: I+
Директива I устанавливает, должны ли проверяться операции ввода/вывода на наличие ошибок {$I+} или нет {$I-}. Если пользователь выберет I, он может осуществлять обработку ошибок с помощью стандартной функции ioresult (см. пример 8.3).

{$L±} Буфер компоновки Глобальная
По умолчанию: L+
При компоновке нескольких модулей генерируется промежуточная информация, которая при (*L-*) размещается в виде временного файла на дискете. а при (*L+*) - в оперативной памяти.

{$N±} Числовая обработка Глобальная
По умолчанию: N-
В режиме (*N+*) существует пять различных типов real, описанных в разделе 5.2. Тогда для версии 4.0 необходим сопроцессор 80x87. При использовании директивы (*N-*) существует только тип real. См. также директиву *E*±.

{$R±} Проверка на соблюдение диапазона Локальная
По умолчанию: R-
Директива R устанавливает, должны {$R+} или нет {$R-} проверяться во время выполнения соблюдение границ для индексов массивов и дл переменных перечисляемого типа и типа "подмножество" в операторах присваивания. В случае R- программа может выдать непредсказуемый результат. Директиву R+ рекомендуется использовать при написании программы, а директиву R-

при использовании уже отлаженной программы. Понятно, что использование R- значительно экономит машинное время.

{$S±} Проверка на переполнение стека Локальная
По умолчанию: S+
Директива (*$S+*) устанавливает, должен ли проверяться стек перед вызовом каждой процедуры или функции на наличие достаточного места для локальных переменных и возвращаемых данных. При задании директивы (*$S-*) такой контроль не производится, что может привести к зависанию системы.

{$T±} Генерация файла TPM Глобальная
По умолчанию: T-
При задании директивы (*$T-*) на дискете генерируется файл .TPM (Turbo Pascal-MAP-file), содержащий данные об именах и номерах строк. Но при выборе этой директивы нужно задавать и директиву (*D+*). Такой файл .TPM используется для отыскания и индикации ошибок исполнения. С помощью программы TPMAP.EXE можно сгенерировать из файла .TPM файл .MAP, используемый символьным отладчиком. Файл .MAP можно читать непосредственно, поскольку он записан в кодах ASCII.

{$V±} Проверка строки VAR Локальная
По умолчанию: V+
Директива V устанавливает, должна ли проверяться фактическая длина передаваемого по ссылке параметра типа string. V+ означает, что для передаваемых по ссылке параметров типа string длины фактического и формального параметров должны совпадать. При V- эти длины могут быть различными.

Директивы, имеющие параметры, имеют следующий вид

(*$директива параметр*)

(*$I имя_файла*) Добавить файлы (include) Локальная
Директива $I имя_файла позволяет компилятору добавить текст из файла с указанным именем в то место, где стоит данная директива. Тем самым можно разбить исходный текст на несколько файлов. Добавленный с помощью директивы (*$I имя_файла*) текст не может сам содержать в себе директивы включения. При этом поиск указанного файла осуществляется вначале в текущем каталоге, а затем в каталоге, указанном с помощью Option/Directories/Include.

(*$L имя_файла*) Скомпоновать объектные файлы Локальная
Эта директива позволяет компоновщику включить файл с указан-

ным именем в программу. Файл "имя_файла" должен быть объектным (.OBJ).

(*$M размер_стека, минимальный, максимальный размер динамической области*) Распределение памяти Глобальная

Директива устанавливает, сколько места в памяти должно быть отведено для стека и сколько памяти минимально и максимально может быть отведено под динамическую область, например, (*M 32766,0,655360*). Стек может занимать область 1024-65520, динамическая область 0-655360 (640 Кбайт).

(*$U имя_файла*)

Благодаря этой директиве можно использовать и те модули, для которых имя файла и имя модуля различны. Директива помещается после uses перед соответствующим именем модуля. Имя_файла имеет атрибут .TPU.

Директивы по условию (Компиляция по условию)

С помощью этих директив можно исключить из компиляции отдельные куски текста. Существуют две формы директив

(*$IF условие)...(*$ENDIF*)

и

(*$IF условие)...(*ELSE*)...(*ENDIF*)

Действие директив аналогично действию операторов if. Если условие выполнено, в первом случае компилируется стоящий между (*$IF*) и (*$ENDIF*) исходный текст, в противном случае компиляция не выполняется. Во втором случае при выполнении заданного условия компилируется стоящий между (*IF*) и (*ELSE*) исходный текст, если условие не выполняется, компилируется стоящий между (*ELSE*) и (*ENDIF*) текст. Условие можно сформулировать очень просто, определив символические имена (или не определяя их):

(*$DEFINE symbolname*)

Начиная с этого места заданный именем symbolname символ считается определенным.

(*$UNDEF symbolname*)

Начиная с этого места заданный именем symbolname символ считается неопределенным. Если symbolname все же не был определен, директива никакого действия не окажет.

(*$IFDEF symbolname*)
Условие считается выполненным, если symbolname определен, и невыполненным в противном случае.

(*$IFNDEF symbolname*)
Условие считается выполненным, если symbolname неопределен, и невыполненным в противном случае, то есть данная директива является прямой противоположностью директивы $IFDEF.
(*$IFOPT директива_переключения*)
Если достигнута директива переключения, условие считается выполненным (true).

Здесь следует четко сказать, что symbolname не имеет ничего общего с именами других объектов Паскаля, используемыми в программе. Согласно

```
(*$DEFINE dies*)
```

символ dies существует независимо от того, используется ли или нет имя dies в другом месте программы.

```
(*IFDEF dies*)
writeln('Здесь символ dies определен');
(*$ENDIF*)
```

приводит к выполнению оператора writeln. Следующие символы определяются до начала компиляции:

VER40 для версии 4.0;
MSDOS если используется операционная система MS-DOS или PC-DOS;
CPU86 если имеется процессор 80x86;
CPU87 если имеется сопроцессор 80x87 (итак, этот символ определяется не всегда. Можно написать, например, так:
(*IFDEF CPU87*) (*$N+*) (*$ELSE*) (*N-*) (*ENDIF*)

Подчеркнем еще раз: symbolname не является именем переменной, а значит, и не относится к типу boolean. После

```
const das = true;
```

директивы (*$IFDEF das*) не достаточно, если ранее в программе не стояла директива

```
(*$DEFINE das*).
```

СПИСОК ЛИТЕРАТУРЫ

[1] Turbo Pascal 4.0. Band 1: Benutzerhandbuch. Heimsoeth & Borland, Muenchen 1988, 399 S.
Турбо Паскаль, версия 4.0. Том 1: Руководство пользователя.

[2] Turbo Pascal4.0. Band 2: Referenzhandbuch. Heimsoeth & Borland, Muenchen 1988, 392 S.
Турбо Паскаль, версия 4.0. Том 2: Перечень сообщений.

[3] Turbo Pascal 5.0. Update, Heimsoeth & Borland, Muenchen 1988, 207 S.
Турбо Паскаль, версия 5.0. Обновленная редакция.

[4] Herschel R. Standard Pascal, Oldenbourg, 7. Auflage 1990.
Гершель Р. Стандартный Паскаль. 7-е издание.

[5] Herschel R. Turbo Pascal 3.0, Oldenbourg, Muenchen, 7. Auflage 1990.
Гершель Р. Турбо Паскаль, версия 3.0.

[6] Daessler K. und Sommer M. Pascal-Einfuerung in die Sprache, Norm-Entwurf DIN 66256, Erlaeuterungen, Springer, 2. Auflage 1985
Десснер К. и Зоммер М. Введение в язык Паскаль. Стандарт ДИН 66256, пояснения.

[7] Wirth N. Algorithmen und Datenstrukturen, Teubner, 3. Auflage 1983
Вирт Н. Алгоритмы и структуры данных

ПРИЛОЖЕНИЕ F

ГЛОССАРИЙ

Адрес 198

Блок 19

Видеоадаптер 265
Выражение 46

Дерево 211
Динамическая область 217
Директивы компилятора 327
Драйвер ANSI 261

Заголовок программы 7
Задание типа 33
Запись данных 176
Зарезервированное слово 21

Имя 20
  стандартное - 22
Индекс 115
Исполнимая часть 7
Исходный текст 8
Итерация 96

Командные строки TPC 247
Комметнтарий 22
Компоновщик 8
Константа 30
  массив - 124
  множество - 172
  типизированная - 31
  record - 150

Мантисса 25
Матрица 136
Множество 164
  базисное - 165
Модульное
  программирование 226

Оверлей 225
Оператор 64
  перехода - 74
  повторения - 71
  присваивания - 66
  простой - 64
  пустой - 64
  составной - 65
  условный - 67
  цикла - 70
Описательная часть 7
Основное меню 10
Ошибки 287

Палитра 276
Параметр
  по значению - 80
  по ссылке - 80
  фактический - 79
  формальный - 79
Параметры вызова 251
Переменная 33
  глобальная - 35
  локальная - 35
  составляющая - 114
  целая - 114
Подмножество 166
Поле 149
Поле данных 176
Последовательность
  символов 141
Процедура 77
  append 188
  arc 273
  assign 175
  bar 274
  bar3d 274
  blockread 197
  blockwrite 198
  chdir 254
  circle 274
  cleardevice 270

clearviewport 271
close 175
closegraph 268
clrscr 43
delay 84
delete 143
deline 44
dispose 207
drawpoly 275
erase 175
exit 76
fillpoly 275
floodfill 276
flush 188
freemem 218
getarccoords 273
getaspectratio 274
getbkcolor 277
getcolor 277
getdate 259
getdefaultpalette 286
getdir 254
getmem 218
getpalette 277
gettextsettings 279
gettime259
getviewsettings 271
getlinestyle 275
gotoxy 43
halt 76
highvideo 43
initgraph 267
insert 143
insline 44
intr 256
line 272
linerel 272
lineto 272
lowvideo 43
mark 218
mkdir 254
moverel 272
moveto 272
msdos 258
new 207
normvideo 43
nosound 84
outtext 278
outtextxy 278
overinit 245
oversetbuf 245
ovrclearbuf 245
ovrinitems 245
pieslice 276
putpixel 272
read 36, 177
readln 36, 187
rectangle 277
release 218
rename 175
reset 175
rewrite 175
rmdir 254
seek 180
setactivepage 271
setallpalette 277
setbkcolor 277
setcolor 277
setfillpattern 276
setfillstyle 275
setlinestyle 273
setpalette 277
settextbuf 188
settextjistify 278
settextstyle 278
setusercharsize 286
setviewport 270
setvisualpage 271
sound 84
str 143
textbackground 43
texcolor 43
textheight 279
textwidth 279
truncate 180
val 143
write 39, 177
writeln 39, 187
Приведение типов 60

Рекурсия 96

Связанный список 209
Сегмент 198
Семантика 303
Система координат 269
Смещение 199

Тип
  ограниченный - 11
  перечисляемый - 107
Точка останова 291

Указатель 198

Файл
  нетипизированный - 173
  типизированный - 173
  текстовый - 173
Функциональные клавиши 14
Функция
  abs 57
  addr 202
  arctan 57
  chr 28
  concat 144
  copy 144
  cos 57
  cseg 203
  dec 58
  dseg 203
  eof 175
  eoln 187
  exp 57
  filepos 180
  filesize 180
  frac 58
  getdrivername 286
  getmaxmode 286
  getmaxx 271
  getmaxy 271
  getpalettesize 286
  getpaxel 272
  getx 272
  gety 272
  grapherrormsg 268
  graphresult 268
  hi 59
  inc 58
  int 58
  ioresult 41
  keypressed 38
  length 144
  ln 57
  lo 58
  maxavail 218
  memavail 218
  odd 58
  ofs 203
  ord 107
  pi 57
  pos 144
  pred 107
  ptr 202
  ovrgetbuf 245
  paramcount 251
  paramstr 251
  random 57
  readkey 58
  round 58
  seekeoln 188
  seekeof 188
  seg 203
  sin 57
  sizeof 29
  sqr 57
  [illegible] 57
  sseg 203
  succ 107
  swap 59
  trunc 58
  upcase 28

Шестнадцатеричная
  константа 24

A

Array 113

B

BIOS-Routinen 256
boolean 29
byte 23

C

char 27
checkbreak 241
checkeof 41
comp 25

D

Debugger 289
double 25

E

Editor 297
Exponent 25
extended 25

I

input 36
integer 23
Interrupt 254

L

longint 23

M

Make 246

O

Objectcode 8
Ordinaltyp 105
Operand 48
output 36

P

pointer 204

R

real 25
Record 149
varianter 158

S

shortin 23
single 25
Stack 95
String 140
Syntax 18

U

Unit 8

W

word 23

Содержание

Стр.

Предисловие редакторов перевода 3

1. Введение 4

2. Принципы Паскаля и его программное окружение 6

3. Программное окружение Турбо Паскаля 9
- **3.1 Инсталляция** 9
- **3.2 Интегрированная среда TURBO.EXE** 10

4. Структура программ на Паскале 16

5. Простейшие стандартные типы данных 23
- **5.1 Целые числа** 23
- **5.2 Вещественные числа** 25
- **5.3 Символы** 27
- **5.4 Boolean** 29

6. Соглашения о постоянных 30

7. Соглашения о переменных 33

8. Элементарный ввод и вывод 35
- **8.1 Ввод** 36
- **8.2 Вывод** 39

9. Операторы и выражения 46
- **9.1 Арифметические операторы и выражения** 48
- **9.2 Битовые операции** 50
- **9.3 Логические операции** 53
- **9.4 Стандартные математические функции** 56
- **9.5 Приведение типов** 59

10. Операторы 64
- **10.1 Составной оператор** 65
- **10.2 Оператор присваивания** 66
- **10.3 Условные операторы** 67
- **10.4 Операторы цикла** 70
- **10.5 Операторы перехода** 74

11. Процедуры и функции 77
- **11.1 Процедуры** 77
- **11.2 Функции** 86
- **11.3 Связь процедур и функций** 92
- **11.4 Рекурсия** 95

12. Типы и идентификаторы типов 101

13. Простые типы данных....105
13.1 Перечисляемый тип 107
13.2 Ограниченный тип....111

14. Тип array 113
14.1 Типы структурированных данных....113
14.2 Одномерные массивы....116
14.3 Многомерные массивы....136
14.4 String....140

15. Тип record 149

16. Тип set 164

17. Файлы....173
17.1 Типизированные файлы....176
17.2 Текстовые файлы....187
17.3 Нетипизированные файлы 196

18. Тип данных "указатель" 198
18.1 Адреса и операции над адресами 199
18.2 Динамические переменные и структуры связанных данных 207

19. Построение программы по модульному принципу и модули....224
19.1 Построение программы по модульному принципу....224
19.2 Модули....226
19.3 Стандартные модули....239
19.4 Оверлейные блоки (версия 5.0)....243
19.5 Компоновка....246

20. Командные строки TPC....247

21. Турбо Паскаль и MS-DOS....253

22. Графика....265

23. Поиск ошибок....287
23.1 Классификация ошибок 287
23.2 Турбо-отладчик (применительно к версии 5.0) 289

24. Редактор Турбо....297

Приложения

A. Синтаксис Турбо Паскаля 303
B. Операции....324
C. Набор символов ASCII 326
D. Директивы компилятора 327

E. Список литературы 333
F. Глоссарий 334

Перечень примеров

2.1 Вводный пример: приветствие 6
4.1 Угадайте любое число 17
5.1 Тип данных "двойной" с (*$N+*) 26
5.2 Имеющиеся символы кода ASCII 28
5.3 Функция suzeof 29
8.1 Поэкспериментируйте с операторами чтения и записи 40
8.2 Считывание и суммирование чисел (eof и ^Z) 40
8.3 Демонстрация ioresult 42
8.4 Чтение RETURN 42
8.5 Красочное оформление текстов 44
8.6 Демонстрация 45
8.7 Формат числа при записи 45
8.8 Вывод на печатающее устройство (использование ПЭВМ в качестве пишущей машинки 46
9.1 Демонстрация операторов mod и div 49
9.2 Операторы сдвига shl и shr 51
9.3 Битовые операторы not, and, or, xor 51
9.4 Побитовый вывод целого числа 53
9.5 Логические операторы 56
9.6 Вычислительные сложности при использовании выражения x/a*a 61
9.7 Вычислительные сложности при использовании выражения b+a-a 62
10.1 Пример правильного и неправильного применения оператора присвоения 67
10.2 Ответы на запросы 69
10.3 Вывод чисел, не присутствующих среди введенных 72
10.4 Пример оператора for 73
10.5 Числовые игры (заменить известное число *) 73
10.6 Считывание и накопление чисел 75
11.1 Процедура quersumme 78
11.2 Различие между значимыми и ссылочными параметрами . 80
11.3 Локальные и глобальные переменные 82
11.4 Параметры, не имеющие типа 83
11.5 Символы, выводимые в виде знаков Морзе 84
11.6 Функция quersumme 87
11.7 Определение нулевых точек функции f(x) 89
11.8 Значимые и ссылочные параметры 93
11.9 Бесконечная рекурсия 96
11.10 Сравнение итерации и рекурсии 96
11.11 Преобразование десятичных чисел в восьмеричные 97
11.12 Числа Фибоначчи 98
11.13 Простые числа (пример forward) 100
12.1 Наименование типа в качестве глобальной переменной 102
12.2 Функция в качестве параметра 104
13.1 Пример перечислимого типа 107

13.2 Пересчитайте детей!....108
13.3 Контроль за границей подмножества 112
13.4 Числа при игре в лото....113
14.1 Считать и сдвинуть массив 118
14.2 Отсортировать 1000 чисел 120
14.3 Частотность повторения букв в тексте 121
14.4 Распределение мест в парламенте 122
14.5 Поиск по массиву 126
14.6 Сортировка массива 128
14.7 Пересчитайте детей!....133
14.8 Проверка матрицы на симметричность....137
14.9 Частотность парных букв 138
14.10 Чтение строк....140
14.11 Строковые переменные в качестве параметра 142
14.12 Отгадывание слов 145
14.13 Сортировка строк....146
15.1 Массив записей 152
15.2 Чтение защищенных записей 154
15.3 Определение интервала времени 157
15.4 Площадь фигур (переменная запись) 159
15.5 Побитовое представление вещественных чисел (переменная запись) 160
16.1 Операции над множествами....166
16.2 Замена гласных в тексте 167
16.3 Пересчитайте детей!....168
16.4 Определение чисел при игре в лото 171
17.1 Создание и вывод файла определенного типа....177
17.2 Удаление, добавление и изменение записей данных в некотором файле с данными определенного типа 181
17.3 Запись после сброса reset 186
17.4 Копирование текстовых файлов 189
17.5 Копирование текстового файла и сдвиг текста вправо 190
17.6 Сравнение текстового файла и файла с данными заданного типа....192
17.7 Пример программы вывода на печать....193
17.8 Пример seekeoln 194
17.9 Копирование текстового файла с использованием буфера емкостью в 1 Кбайт 195
17.10 Копирование файла данных, не имеющих типа 198
18.1 Вывод десятичного и шестнадцатеричного адресов 205
18.2 Указатель массива и компонентов 206
18.3 Глобальная оценка неупорядоченного массива ("кучи") ...207
18.4 Пример new/dispose 208
18.5 Создание и вывод связанных списков 209
18.6 Операции над связанными списками....210
18.7 Создание и вывод бинарного дерева 213
18.8 Моделирование стека в виде связанного списка 215
18.9 Проверка неупорядоченного массива....219
18.10 Преобразования в двумерном пространстве 221
19.1 Моделирование стека (разложенного на модули) 230
19.2 Вложение модулей 237

20.1 Протокол параметров обращения 251
20.2 Копирование текстового файла (с именем в качестве параметра вызова) 252
21.1 Команды при работе с каталогами 254
21.2 Пример прерывания 257
21.3 Свободное место на дискете 258
21.4 Выбор даты 259
21.5 Выбор времени по часам 260
21.6 Вывод с использованием драйвера ANSI 261
21.7 Запись в память дисплея 262
21.8 Пример использования inline 264
22.1 Скелет графической программы 266
22.2 Представление концентрических окружностей с помощью circle 270
22.3 Синусоиды 280
22.4 Пример drawpoly и fillpoly 281
22.5 Пример использования bar и bar3d 281
22.6 Демонстрация aspect_ratio 282
22.7 Окна с рамкой 284
22.8 Типы шрифтов 285
23.1 Программы с ошибками для отладки 292

Рудольф Хершель

TURBO PASCAL 4.0/5.0

Редакторы: С.В.Сорокин, А.В.Тупицын

Оригинал-макет выполнен на настольной издательской системе «Компьютерная инормационно-издательская служба» 160008, Вологда, ул. Ленина 15 каб. 233

Формат 60x88/16. Усл. печ. л. 21,4.
Тираж 50000. Заказ № 632. Цена договорная.

ПП „Чертановская типография" Москва, Варшавское ш., 129А.

Для заметок

Для заметок

ВНИМАНИЮ ЧИТАТЕЛЕЙ !

К этой книге имеется дискета, содержащая программы в MS-DOS-формате. Дискета содержит все исходные тексты примеров программ этой книги, что позволяет значительно ускорить процесс изучения TURBO PASCAL, а также использовать их в качестве фрагментов в своих собственных разработках. Дискеты высылаются наложенным платежом.

Заявки принимаются по адресу: 160031, г.Вологда, ул.Октябрьская, 19, каб. 2 Телефон для справок: (8-817-22) - 5-09-79