РОСЖЕЛДОР
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Ростовский государственный университет путей сообщения»
(ФГБОУ ВО РГУПС)

О.Г. Ведерникова, Е.В. Голубенко

Прикладное программирование

Учебное пособие

Ростов-на-Дону
2017

УДК 681.3(07) + 06

Ведерникова, О.Г.

Прикладное программирование: [Электронный ресурс] учебное пособие. / О.Г. Ведерникова, Е.В. Голубенко ; Рост. гос. ун-т путей сообщения. – Ростов н/Д, 2017. – 80 с. : ил., табл., прил. – Библиогр.: 8 назв.

Рассмотрены базовые приемы программирования, начиная с простейших расчетов. Содержится лекционный материал, а также большое количество исходных текстов программ-примеров.

Учебное пособие предназначено для студентов направления подготовки 23.03.03 «Эксплуатация транспортно-технологических машин и комплексов». Одобрено к изданию кафедрой «Вычислительная техника и автоматизированные системы управления».

Содержание

Введение

Язык программирования Паскаль (Pascal) в настоящее время следует рассматривать как учебное средство, позволяющее, при своем простом синтаксисе, сосредоточиться на алгоритмической стороне программирования, не вдаваясь в детали разработки сложных пользовательских интерфейсов и структур данных.

Рекомендуемая среда для работы с примерами – Turbo Pascal 7.1 разработана компанией Borland International в 1983–97 гг. Скачать компактный дистрибутив Turbo Pascal 7.1 можно, например, по ссылке http://tpdn.ru/files/turbo-pascal-download/ . Эта оболочка языка Паскаль создавалась для операционной системы (ОС) MS-DOS, но в современных ОС семейства Windows программа, написанная на языке Паскаль, также будет работать, только без удобных интерфейсных возможностей Windows.

Данное пособие предназначено для изучения базовых приемов программирования. Оно последовательно знакомит читателя с азами программистского искусства, начиная с простейших расчетов.

1 АЛГОРИТМ И ПРОГРАММА

Написанию программы всегда предшествует разработка некоторого плана (*алгоритма*) решения задачи. Кратко опишем основные понятия теоретической информатики, связанные с алгоритмами.

Алгоритм – это однозначно определенная последовательность действий, записанная на понятном исполнителю *алгоритмическом языке* и определяющая процесс перехода от исходных данных к результату.

В этом определении уже указаны основные *свойства алгоритма*. Во-первых, алгоритм состоит из конечного набора инструкций или *шагов*, во-вторых, каждый шаг трактуется исполнителем единственным образом, что позволяет гарантированно получить решение для некоторого набора входных данных, в-третьих, алгоритм всегда сводится к некоторому преобразованию исходных данных в результат или результаты. В этом смысле формулы для решения квадратного уравнения или даже четко составленную инструкцию по варке кофе можно считать алгоритмами, выполнимыми исполнителем-человеком. Для машины, разумеется, требуется более четкая формализация задачи, чем для человека.

Перечислим основные *свойства* алгоритма:

– дискретность – алгоритм состоит из отдельных инструкций (шагов);

– однозначность – каждый шаг понимается исполнителем единственным образом;

– массовость – алгоритм работает при меняющихся в некоторых пределах входных данных;

– результативность – за конечное число шагов достигается некоторый результат.

1.1 Формы записи алгоритма

Принято выделять две основные формы записи алгоритма.

Графическая форма записи (*блок-схема*) характерна тем, что отдельные шаги алгоритма изображаются геометрическими фигурами, а последовательность выполнения шагов – связями между этими фигурами. На рис. 1.1 указаны основные элементы блок-схем.

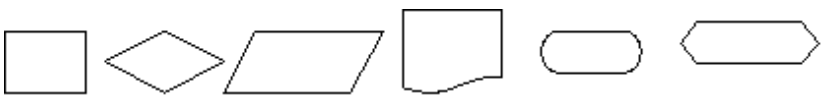


Рис. 1.1. Основные элементы блок-схем

Указанные на рис. 1.1 геометрические фигуры интерпретируются так:

Прямоугольник – любая последовательность действий; внутри прямоугольника записывают формулы или словесное описание выполняемых действий.

Ромб – блок проверки условия; так как любое условие может быть только истинно или ложно, у блока один вход и два выхода, соответствующие действиям, выполняемым в случаях, когда условие истинно и когда оно ложно. Выходы подписывают символами «да» и «нет».

Параллелограмм – блок ввода исходных данных. Внутри фигуры обычно пишут, какие именно данные должны быть введены.

Лист с разрывом – блок вывода данных. Внутри блока указывают, какие данные или сообщения программа выводит для представления пользователю.

Закругленные прямоугольники – необязательные блоки начала и конца программы, внутри блоков обычно указывают ключевые слова «нач» и «кон» соответственно.

Последняя фигура служит для изображения циклов, как правило, у нее два входа (первый и повторный вход в цикл) и один выход, соответствующий завершению циклического процесса.

На рис. 1.2 приведен пример блок-схемы, иллюстрирующей известный процесс решения квадратного уравнения.

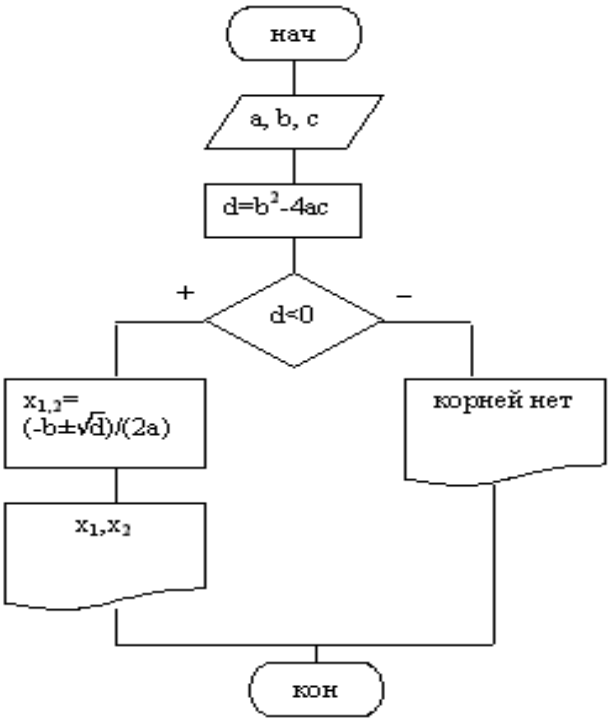


Рис. 1.2. Блок-схема решения квадратного уравнения

1.2 Этапы разработки программы

Разработка любой программы, от несложной учебной задачи до профессионального приложения, может быть разбита на ряд этапов. Кратко опишем и охарактеризуем их.

1 **Определение входных и выходных данных.**

На первом этапе определяются входные и выходные данные программы, способ ее взаимодействия (*интерфейса*) с пользователем, язык и среда программирования.

2 **Разработка алгоритма.**

На этом шаге производится определение последовательности действий, ведущих к решению задачи, и запись их в одной из форм записи алгоритма.

3 **Кодирование** (программирование).

Третий этап – это перевод алгоритма на язык программирования и создание *исходного текста программы* в одной из систем программирования. Программа на любом языке состоит из *операторов* – так называются отдельные действия, разрешенные в языке. Число операторов в любом языке ограничено, и правила их написания жестко заданы.

4 **Компиляция и отладка.**

Исходный текст на языке Паскаль не будет непосредственно исполняться компьютером, для работы программы ее требуется *откомпилировать*, т.е. перевести в машинный код. Эту работу выполняет специальная программа-*компилятор* или *оболочка* языка. Оболочка Паскаля – Turbo Pascal 7.1. В результате преобразования компилятором исходного текста программы в машинный код получается *исполняемый файл* с расширением *exe*, который можно запустить (*выполнить*) в той ОС, для которой разработан компилятор.

Итак, *компиляция* – это процесс преобразования программы в машинный код. Программа, которую удалось откомпилировать, не обязательно работает правильно. Она может содержать ошибки, для выявления которых предназначен этап *отладки* – поиска ошибок в программе. Как правило, компиляция и отладка выполняются программистом в тесной взаимосвязи.

Возможны программные ошибки трех видов:

– *синтаксические* (ошибки в правилах языка);

– *алгоритмические* (ошибки в логике программы);

– *ошибки времени исполнения*, возникающие в процессе работы запущенной программы.

Компилятор способен найти только синтаксические ошибки, для выявления же алгоритмических ошибок служит этап *тестирования* программы. Ошибки времени исполнения возникают как результат некорректных действий пользователя, недопустимых операций над данными (например, попытки из-

влечь квадратный корень из отрицательного числа, поделить на ноль) или ошибок программного и аппаратного обеспечения ЭВМ.

5 **Тестирование**.

Тестированием называют проверку правильности работы программы на наборах «пробных» (*тестовых*) данных с заранее известным результатом.

6 **Документирование и поддержка.**

Этот этап включает в себя создание справочной системы и документации к программе, возможно, расширение ее функциональности, выпуск новых версий, исправление ошибок, которые практически неизбежны в любой сложной программной системе.

2 ДАННЫЕ В ЯЗЫКЕ ПАСКАЛЬ

Любая программа выполняет над исходными данными некоторые расчеты. При этом, как и переменные или константы в математике, отдельные элементы данных обозначаются даваемыми программистом именами (*идентификаторами*). Любые идентификаторы в языке Паскаль строятся по следующим правилам:

– имена могут включать в себя латинские буквы, цифры и знак подчеркивания (для простоты опустим некоторые другие символы, разрешенные в именах);

– имя состоит из одного слова; если требуется пробел в имени, он заменяется подчеркиванием: так, `My_1` будет правильным идентификатором, а `My 1` – нет;

– имя всегда начинается с буквы, возможен объект с именем `A1`, но не `1A`; прописные и строчные буквы в именах не различаются Паскалем: `x1` и `X1` – это одна и та же величина;

– имена не могут совпадать с зарезервированными в языке *служебными словами*, обозначающими разрешенные в языке операции над данными: например, нельзя назвать `Begin` или `BEGIN` ни одну величину в программе, так как `begin` – зарезервированное служебное слово, а прописные и строчные буквы в служебных словах также не различаются. Ознакомиться с большинством служебных слов мы сможем в процессе изучения языка.

2.1 Константы

Константой называют величину, значение которой не меняется в процессе выполнения программы.

Числовые константы служат для записи чисел. Различают следующие их виды:

Целочисленные (целые) константы – записываются со знаком + или −, или без знака, по обычным арифметическим правилам:

```
-10      +5       5
```

Вещественные числа могут записываться в одной из двух форм:

обычная запись: `2.5` `-3.14` `2.` Обратите внимание, что целая часть отделяется от дробной символом *точки*;

экспоненциальная («научная») форма. В этой записи вещественное число представляется в виде $\mathbf{m*10^p}$, где **m** – *мантисса* или основание числа, $0.1 \le |\mathbf{m}| \le 1$, **p** – порядок числа, заданный целочисленной константой. Действительно, любое вещественное число можно представить в экспоненциальной форме:

```
-153.5       -0.1535*10^3
99.005       0.99005*10^2
```

Во всех IBM-совместимых компьютерах вещественные числа хранятся как совокупность мантиссы и порядка, что позволяет упростить операции над ними, используя специальную арифметику, отдельно обрабатывающую мантиссу и порядок. Для программной записи числа в экспоненциальной форме в качестве обозначения «умножить на 10 в степени» используется символ **E** или **e** (латинские):

-153.5 -0.1535*10^3 -0.1535E3 или -1.535E02
99.005 0.99005*10^2 0.99005E+2 или 9.9005e+01

Без принятия специальных мер программа на языке Паскаль будет выводить на экран и принтер вещественные числа именно в такой форме. Кроме того, такая форма удобна для записи очень маленьких и очень больших чисел:

10^{30} 1e30
-10^{20} -1E20
10^{-30} 1E-30

Поскольку размер памяти, отводимой под мантиссу и порядок, ограничен, то *вещественные числа всегда представляются в памяти компьютера с некоторой погрешностью*. Например, простейшая вещественная дробь 2/3 дает в десятичном представлении 0,666666... и, независимо от размера памяти, выделяемой для хранения числа, невозможно хранить все его знаки в дробной части. Одной из типичных проблем программирования является учет возможных погрешностей при работе с вещественными числами.

Кроме числовых констант существуют и другие их виды:

логические константы служат для проверки истинности или ложности некоторых условий в программе и могут принимать только одно из двух значений: служебное слово true обозначает истину, а false – ложь;

символьные константы могут принимать значение любого печатаемого символа и записываются как символ, заключенный в *апострофы* ('одинарные кавычки'):

```
'Y'     'я'     ' '
```

Строковые константы – это любые последовательности символов, заключенных в апострофы. Как правило, строковые константы служат для записи приглашений к вводу данных, выдаваемых программой, вывода диагностических сообщений и т.п.:

```
'Введите значение X:'
'Ответ='
```

Именованные константы перечисляются в разделе описаний программы оператором следующего вида:

```
const Имя1=Значение1;
 Имя2=Значение2;
 ...
 ИмяN=ЗначениеN;
```

Ключевое слово const показывает начало раздела описаний именованных констант. Ясно, что зачастую удобнее обращаться к константе по имени,

чем каждый раз переписывать ее числовое или строковое значение. Приведем пример раздела:

```
const e=2.7182818285;
  lang='Turbo Pascal 7.1';
```

Здесь описана числовая константа `e` со значением основания натурального логарифма и строковая константа с именем `lang`, содержащая строку `'Turbo Pascal 7.1'`.

Каждое даваемое программистом имя должно быть *уникальным* в пределах одной программы. Если мы включим этот раздел в свою программу, мы уже не сможем создать в ней других объектов с именами `e` и `lang`.

2.2 Переменные и типы переменных

Переменными называют величины, значения которых *могут изменяться* в процессе выполнения программы. Каждая переменная задается своим уникальным именем, имена могут включать в себя латинские буквы, цифры и знак подчеркивания. Максимально возможная длина имени зависит от реализации языка Паскаль, теоретически можно давать переменным имена вплоть до 63 символов длиной, что едва ли актуально, обычно имена не длиннее 5–10 символов.

Поскольку любые данные в памяти компьютера хранятся в числовой форме и двоичной системе счисления, кроме имени переменной обязательно следует присвоить и *тип*, определяющий *диапазон значений*, принимаемых переменной, и *способ ее обработки* машиной. В любом языке программирования, в том числе и в языке Паскаль, существует стандартный набор типов, к которым может быть отнесена та или иная совокупность ячеек памяти. В табл. 2.1 представлены не все возможные, а лишь основные типы данных языка Паскаль.

Таблица 2.1

Основные типы данных языка Паскаль

Ключевое слово	Название и описание типа	Объем памяти, байт	Диапазон возможных значений
boolean	Логический: хранит одну логическую переменную	1	true и false
char	Символьный: хранит код одного символа из набора ASCII-кодов	1	От 0 до 255 включительно ($2^8 = 256$)
integer	Целочисленный	2	±215

Окончание табл. 2.1

Ключевое	Название и описание типа	Объем па-	Диапазон воз-

слово		мяти, байт	можных значений
word	Целочисленный без знака	2	±216 – диапазон вдвое больше, так как 16-й бит не занят под знак числа
longint	Длинное целое: для представления больших целочисленных значений	4	±231
real	Вещественное число с точностью представления до 11–12 знака в дробной части	6	~ $2.9*10^{-39}$–$1.7*10^{38}$
double	Вещественное число с точностью представления до 15–16 знака в дробной части	8	~ $5*10^{-324}$–$1.7*10^{308}$
string	Последовательность символов типа char длиной от 1 до 255	2–256 (данные строки + 1 байт для хранения ее длины)	Любые строки текста, состоящие из печатаемых символов

Теоретически для записи переменной типа `boolean` было бы достаточно 1 бита, но минимальная адресуемая единица памяти – 1 байт. В этой таблице уточните, как именно объем памяти в байтах, выделяемой под переменную, влияет на диапазон представляемых ей значений.

Переменные описываются в программе оператором следующего вида:

```
var   Список1:Тип1;
      Список2:Тип2;
      ...
      СписокN:ТипN;
```

Здесь *список* – набор имен переменных, разделенных запятыми (или одна переменная), а *тип* – любой из рассмотренных выше типов данных. Например, конструкция

```
var t,r:real;
 i:integer;
```

описывает две вещественные переменные с именами `t` и `r`, а также целочисленную переменную с именем `i`. Ключевое слово `var` можно продублировать, но обычно такой необходимости нет. Сокращение `var` образовано от английского «**variable**» (переменная).

2.3 Символьные переменные

Компьютер способен обрабатывать не только числовые данные. Задачи обработки символьных данных распространены не менее, а, возможно, и более чем чисто арифметические расчеты. Для работы с отдельными символами описываются переменные символьного типа char: Это можно использовать, например, для отслеживания действий пользователя по нажатию клавиш, а также для обработки текста.

Символьные переменные могут принимать значение любого *печатаемого символа* клавиатуры и записываются как символ, заключенный в *апострофы* ('одинарные кавычки'): 'Y' 'я' ' ' , а также *непечатаемые символы,* такие как перевод строки [enter], перевод курсора вверх, вниз, влево, вправо и т.д.

Объявляются *символьные переменные* в операторе описания переменных **var** типом **char.**

```
var <список имен символьных переменных >:char;
```

Например:

```
var symbol,s2, ch :char;
{ ... } symbol:='!'; s2:='3'; ch:='y';
```

Одна переменная типа **char** может хранить *только один* символ. Для хранения набора символов (текста) используется другой тип переменных – тип **string**.

Каждому символу приписывается целое число в диапазоне 0...255. Это число служит кодом внутреннего представления символа (представления символа внутри памяти ПК) и называется **кодом ASCII**. Аббревиатура **ASCII** расшифровывается как *American Standard Code for Information Interchange* – американский стандартный код для обмена информацией.

Таблица 2.2

Кодировка символов в соответствии со стандартом ASCII

Код	Символ	Код	Символ	Код	Символ	Код	Символ
0	NUL	32	BL	64	®	96	'
1	ЗОН	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j

Окончание табл. 2.2

Код	Символ	Код	Символ	Код	Символ	Код	Символ
11	VT	43	+	75	k	107	k
12	FF	44	,	76	L	108	i
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	0
16	DEL	48	0	80	P	112	P
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	V
23	ETB	55	7	87	w	119	w
24	CAN	56	8	88	X	120	X
25	EM	57	9	89	У	121	У
26	SUB	58	:	90	Z	122	z
27	ESC	59	/	91	[	123	{
28	FS	60	<	92	\	124	l
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	—	127	п

Символы с кодами 0...31 относятся к служебным кодам.

Заглавные латинские буквы от ‘A’… ‘Z’ расположены в таблице под номерами от 65 до 90, аналогично строчные латинские буквы от ‘a’…’z’ – под номерами от 97 до 122.

Для того чтобы узнать значение кода **ASCII,** соответствующее данному символу, применяют функцию **ORD**. *Пример*: **`x:=ord('A');`** {**x=65** – код ASCII буквы «A»}

Итак, для работы с кодами символов существуют следующие функции:

1 **ORD(sim`:char):integer`** – позволяет получить значение кода для указанного символа **sim**. Это функция, обратная к следующей функции **CHR.**

Пример:

```
var simbol:char; x,y:integer;
 x:=ord('A'); {x=65 – код ASCII буквы «A»}
simbol:='+';
y:= ord(simbol); {y=75 – код ASCII символа «+»}
```

2 CHR(x: integer):char – позволяет получить символ, соответствующий указанному коду x. Результат будет иметь тип **char**. Это функция, обратная к функции **ORD.**

Пример:

```
var simbol:char;  x:integer;
simbol:=chr(90); { simbol = 'Z' – потому что буква z соответствует ASCII коду 90}
x:=36;
symbol:= chr(x); { simbol = '$' – потому что $ соответствует ASCII коду 36}
```

3 PRED (sim:char) :char – позволяет получить символ, **предыдущий** указанному символу **sim** (предыдущий в таблице кодов).

Пример:

```
var sim, sim2 :char;
sim:= pred('c'); {sim= 'b'предстоит перед 'c'}
sim2:= pred(sim);{sim2='a' предстоит перед b'}
```

4 **SUCC(sim)** – позволяет получить символ, **последующий** указанному символу **sim** (последующий в таблице кодов).

Пример:

```
var  sim, sim2 :char;
sim:= succ('c'); {sim='d' следует за 'c'}
sim2:= succ(sim);{sim2='f' следует за 'd'}
```

Обратите внимание: число 3 и соответствующий ему символ-цифра '3' различны, так как число 3 может храниться только в переменной типа **integer**, а символ '3' может храниться только в переменной типа **char.** *Например:* **symbol:='3';**

Задача (Char). Вычислить сумму порядковых номеров (код ASCII) всех букв, входящих в слово SUM.

1-й способ

```
program example3;
var var simbol1,  simbol2,  simbol3 :char ;
S: integer;
begin
simbol1:='s';{присвоение символьной переменной символа 's'}
simbol2:= 'u';
```

```
simbol3:=’m’;
s:=ord(simbol1) + ord(simbol2)+ ord(simbol3); {вычисление кодов ASCII каждой буквы и сложение их}
writeln(‘s=’, s);
end.
```

2-й способ

```
program example3;
var var;
S: integer;
begin
s:=ord(’s’) + ord(‘u’) + ord(’m’); {вычисление кодов ASCII  каждой буквы и сложение их}
writeln(‘s=’, s);
end.
```

3-й способ

```
program example3;
var var kod1,kod2, kod3 , S: integer;
begin
kod1:= ord(’s’);{ вычисление кода ASCII   буквы’s’ }
kod2:= ord(’u’);{ вычисление кода ASCII   буквы u }  ;
kod3:= ord(’m’ );{ вычисление кода ASCII  буквы  ’m’ }
s:=kod1+  kod2+ kod3;  { сложение кодов}
writeln(‘s=’, s);
end.
```

3 АРИФМЕТИЧЕСКИЕ ВЫРАЖЕНИЯ

Арифметические выражения (АВ) строятся из *операндов*, которыми могут быть константы, переменные и *стандартные функции*. В АВ также входят *арифметические операции* и круглые скобки. В языке Паскаль определены шесть арифметических операций, перечислим их в соответствии с *приоритетом*, т.е. старшинством (табл. 3.1). Операции с одинаковым приоритетом равноправны между собой и выполняются слева направо, как и в математике.

Таблица 3.1

Арифметические операции языка Паскаль

Приоритет	Знак операции	Описание операции
1	*	Умножение
	/	Деление
	div	Деление двух целых значений с отбрасыванием остатка
	mod	Взятие остатка от деления двух целых значений
2	+	Сложение
	-	Вычитание

Операции `div` и `mod` определены только для целочисленных операндов. Приведем пример их использования:

```
var y,c,n:integer;
. . .
y:=2009;
c:=y div 100;
n:=y mod 100;
```

Здесь переменная `c` получит значение `20`, а `n` – значение `9`.

Примеры арифметических выражений мы приведем после рассмотрения стандартных функций языка Паскаль.

Стандартные функции служат для выполнения элементарных математических расчетов, часто требуемых при написании программ. Физически коды стандартных функций хранятся в *стандартной библиотеке* Паскаля – файле с именем **turbo.tpl**. Все функции оформляются одинаково: после имени функции следует ее аргумент, заключенный в круглые скобки. Если аргументов несколько, они разделяются запятыми. Информация об основных стандартных функциях представлена в табл. 3.2.

Таблица 3.2

Стандартные функции языка Паскаль

Математическая запись	Запись на языке Паскаль	Пояснение	Тип аргумента и результата
$\|x\|$	`abs(x)`	Модуль аргумента *x*	Integer (*I*) или Real (*R*)
x^2	`sqr(x)`	Квадрат аргумента *x*	Аргумент – *I* или *R*, результат – *R*
sin *x*	`sin(x)`	Остальные тригонометрические функции выражаются через приведенные три	Аргумент – I или *R*, результат – *R*
cos *x*	`cos(x)`		
arctg *x*	`arctan(x)`		
e^x ln *x*	`exp(x)` `ln(x)`	Экспонента и натуральный логарифм	Аргумент – *I* или *R*, результат – *R*
$\sqrt{x}$	`sqrt(x)`	Квадратный корень от аргумента *x*	Аргумент – *I* или *R*, результат – *R*
π	`pi`	Функция без аргументов, вернет число π	*R*
	`trunc(x)`	Функция отбрасывает дробную часть аргумента, аргумент не округляется	Аргумент *R*, результат *I*
	`frac(x)`	Функция выделяет дробную часть своего аргумента	*R*
	`round(x)`	Округление вещественного числа до ближайшего целого	Аргумент *R*, результат *I*

В табл. 3.2 `x` обозначает любую подходящую по типу переменную, либо результат вычисления выражения соответствующего типа, либо соответствующий по типу результат, вычисленный другой стандартной функцией.

Приведем примеры арифметических выражений.

1 Возвести величину `x` в пятую степень. Выражение может быть записано как `x*x*x*x*x` или `sqr(x)*sqr(x)*x` или `sqr(sqr(x))*x`, последнее показывает, что результаты одних функций могут быть аргументами других, это называют *вложением* функций. Разумеется, тип результата, возвращаемый вложенной функцией, должен быть подходящим для аргумента внешней функции.

2 Возвести величину `a` в произвольную степень `x`. Так как в языке Паскаль нет функции возведения в произвольную степень, воспользуемся формулой $a^x = e^{x*\ln a}$:

```
a:=2.5; x:=0.25;
ax:=exp(x*ln(a));
```

Обратите внимание, что все круглые скобки в выражении должны быть парными. Другой пример применения этого способа:

$\sqrt[3]{x} = x^{1/3}$ = `exp(1/3*ln(x))`.

3 Вычислить $\sin^2 x$. Запись на языке Паскаль: `sqr(sin(x))`. Сравните с выражением $\sin x^2$, которое записывается как `sin(sqr(x))`.

4 Вычислить $k = \mathrm{tg}(t)$. Так как функции тангенса в языке Паскаль нет, распишем тангенс в виде `k:=sin(t)/cos(t);`.

5 При необходимости изменить обычное старшинство операций в записи выражения используют *дополнительные круглые скобки*. Например, правильная запись выражения $y = \frac{a+b}{2}$ выглядит как `y:=(a+b)/2;`. Запись `y:=a+b/2;` неверна, так как это означает $a + \frac{b}{2}$.

6 В записи выражений нельзя пропускать знак *, как часто делается в математике: $b^2 - 4ac$ записывается как `sqr(b)-4*a*c`. Нельзя писать `sin*x` или `sin x`, после имени функции может следовать только ее аргумент в круглых скобках.

Тип выражения определяется старшим из типов входящих в него операндов (т.е. стандартных функций, переменных, констант). Старшинство типов можно определить по табл. 2.1 (в первой строке таблицы находится самый младший тип).

Например, для переменных

```
var i,j:integer; f:real;
```

выражение `i+4*j` имеет целый тип, и его результат можно записать в целую переменную. Выражение `f+i*0.5` дает вещественный результат, который должен быть записан в вещественную переменную.

Операция деления `/` в языке Паскаль всегда дает вещественное число. Для деления целых чисел с целым результатом (остаток отбрасывается) используйте `div`, для взятия остатка от деления двух целых – `mod`.

4 ЛИНЕЙНЫЙ ВЫЧИСЛИТЕЛЬНЫЙ ПРОЦЕСС

Линейный вычислительный процесс (ЛВП) представляет собой набор операторов, выполняемых последовательно, один за другим. Основу программы ЛВП составляют операторы присваивания, ввода и вывода данных.

4.1 Оператор присваивания

Оператор присваивания используется для сохранения результата вычисления арифметического выражения в переменной. Он имеет следующий общий вид:

```
переменная := выражение;
```

Знак := читается как «присвоить».

Оператор присваивания работает следующим образом: сначала вычисляется выражение, стоящее *справа* от знака :=, затем результат записывается в переменную, стоящую *слева* от знака. Например, после выполнения оператора

```
k:=k+2;
```

текущее значение переменной k увеличится на 2.

Тип переменной слева от знака присваивания должен быть *не младше* типа выражения. В частности, это означает, что если выражение дает целое число, результат можно писать и в целую, и в вещественную переменную, если результат вычисления выражения вещественный, писать его в целую переменную нельзя, так как может произойти потеря точности.

Приведем примеры.

1 Записать оператор присваивания, который позволяет вычислить расстояние между двумя точками на плоскости с координатами ($x1$, $y1$) и ($x2$, $y2$).

Оператор будет иметь вид

```
d:=sqrt(sqr(x1-x2)+sqr(y1-y2));
```

2 Записать последовательность операторов присваивания, обеспечивающих обмен значениями переменных x и y в памяти компьютера.

```
c:=x; x:=y; y:=c;
```

Здесь c – дополнительная переменная того же типа, что x и y, через которую осуществляется обмен. Грубой ошибкой было бы, например, попытаться выполнить обмен операторами x:=y; y:=x; – ведь уже после первого из них мы имеем два значения y, а исходное значение x потеряно.

4.2 Оператор ввода

Базовая форма оператора ввода позволяет пользователю ввести с клавиатуры значения одной или нескольких переменных. Оператор ввода с клавиатуры может быть записан в одной из следующих форм:

```
read(список_переменных);
readln(список_переменных);
```

Имена переменных в списке перечисляются через запятую. Здесь и далее список данных, передаваемых любому оператору (а позднее и написанным нами подпрограммам), мы будем называть *параметрами*. Таким образом, параметрами оператора (точнее, *стандартной процедуры*) `read` являются имена переменных, описанных ранее в разделе `var`.

По достижении оператора ввода выполнение программы останавливается и ожидается ввод данных пользователем. Вводимые значения переменных разделяются пробелом или переводом строки (нажатием **Enter**). После ввода значений всех переменных из списка работа программы продолжается со следующего оператора.

Оператор `readln` отличается от `read` только тем, что все переменные должны быть введены в одну строку экрана, клавиша **Enter** нажимается один раз по окончании ввода.

Если пользователь вводит данные недопустимого типа (например, строку текста вместо числа), то выводится системное сообщение об ошибке и работа программы прерывается.

В качестве примера организуем ввод исходных данных для решения квадратного уравнения:

```
var a,b,c:real;
...
read (a,b,c);
```

Для задания значений $a = 1$, $b = 4$, $c = 2.5$ на экране вводится:

```
1_4_2.5¬
```

Здесь и далее «_» означает пробел, а «¬» – нажатие **Enter**. Другой вариант ввода с клавиатуры:

```
1¬
4¬
2.5¬
```

Третий вариант:

```
1¬
4_2.5¬
```

Во всех вариантах пробелов может быть и несколько, лишние будут проигнорированы оператором.

Как правило, перед оператором ввода ставится оператор вывода, служащий приглашением к вводу и поясняющий пользователю, что именно следует сделать (см. п. 4.3).

4.3 Оператор вывода

Базовая форма оператора вывода позволяет отобразить на экране значения переменных, АВ или констант, а также строки текста в апострофах. Оператор записывается в одной из следующих форм:

```
write(список);
writeln(список);
```

Элементы списка перечисляются через запятую.

Элементы списка выводятся в пользовательское консольное окно программы. Вещественные значения выводятся в экспоненциальной форме. Строки выводятся «как есть». После вывода работа программы продолжается со следующего оператора.

Оператор `writeln` отличается от `write` тем, что после вывода значения последнего элемента списка выполняется перевод курсора на следующую строку экрана.

Приведем примеры.

1 Нужно дать пользователю возможность ввести с клавиатуры число, затем программа возведет это число в квадрат и выведет результат на экран.

```
var a,a2:integer;
...
writeln ('Введите целое число:');
             {это приглашение к вводу}
read (a);
a2:=sqr(a);
writeln ('Квадрат числа=',a2);
```

Если ввести значение $a = 2$, на экране будет напечатано

```
Квадрат числа=4
|
```

Символ `|` здесь и далее обозначает курсор. Видно, что оператор `writeln` перевел курсор на следующую строку.

После вывода результата выполнение программы продолжится, а если оператор `writeln` был в ней последним, то и завершится. Чтобы пользователь успел прочитать результат, следует в конце программы добавить оператор

```
readln;
```

который будет ждать нажатия клавиши **Enter**.

2 Требуется вывести на экран результаты решения квадратного уравнения: значения x1 = 1.5 и x2 = 2.5:

```
write ('x1=',x1,'_x2=',x2);
```

Пробел в строковой константе `'_x2='` нужен, чтобы значение `x1` не слилось со строкой `'x2='`. На экране будет напечатано:

```
x1= 1.5000000000E+00 x2= 2.5000000000E+00|
```

Курсор остался в конце строки, так как использована форма оператора `write`.

Вещественные числа читать в подобной форме неудобно, для их вывода используйте решение из следующего раздела.

4.4 Управление выводом данных

В операторе `write` или `writeln` вещественное значение (а также целое или строковое) зачастую удобнее записывать в виде:

`переменная:ширина:точность`

Здесь *ширина* – целое положительное число, определяющее, сколько экранных позиций отводится для вывода всего числа. Ширина определена для числовых значений любого типа и строк.

Точность – целое положительное число, определяющее, сколько цифр из ширины отводится на вывод дробной части числа. Значение точности определено только для вещественных чисел. Оно не учитывает позицию десятичной точки. Разумные значения точности – от `0` до `ширина-2` включительно. Недопустимые значения ширины и точности не будут учтены при выводе.

В качестве примера выведем на экран значения нескольких переменных:

```
var x1,p:real;
i:integer;
...
x1:=2.5; p:=-3.175; i:=2;
writeln ('x1=',x1:8:2,'_p=',p:9:4);
write ('I=','_':5,i:2);
```

На экране будет напечатано:

```
x1=____2.50_p=__-3.1750
I=_____2
```

4.5 Структура простой программы на языке Паскаль

Программа на языке Паскаль не просто состоит из операторов, порядок следования этих операторов не случаен и образует определенную структуру. Структура простейшей программы описана в табл. 4.1.

Таблица 4.1

Структура простой программы на языке Паскаль

Название раздела	Операторы раздела
Заголовок программы (необязателен)	**program** ИмяПрограммы;
Раздел описаний необязателен, но, как правило, присутствует	**const** список констант; **var** список переменных;
Тело программы обязательно, содержит операторы программы	**begin** операторы; **end.**

Пара операторов `begin` и `end` называется *операторными скобками*, они служат для того, чтобы объединить группу операторов, выполняемых вместе, например, в цикле или по условию. Ключевые слова `begin` и `end` следует рассматривать как единый оператор, поэтому после `begin` точка с запятой не ставится, а количество `begin` и `end` в программе всегда одинаково. Таким образом, тело программы заключено в операторные скобки, объединяющие все ее операторы.

Только последний оператор программы завершается точкой: `end.` , все остальные – символом `;`.

Если в программе нет констант, в ней будет отсутствовать раздел `const`, если нет и переменных – раздел `var`.

При написании текста программы следует соблюдать несложные правила, облегчающие его последующие чтение и модификацию:

- На каждой строке обычно пишется один оператор (это облегчает и отладку программы).
- Операторы одного уровня вложенности пишутся с одинаковым отступом слева; например, хорошим тоном считается после начала каждого блока (`begin`) отступать в следующей строке на символ или несколько символов вправо, а закрывать блок так, чтобы соответствующий `end;` находился под своим `begin`.

Приведем пример неправильного и правильного структурирования:

```
program p1; var
a,b,c:real;  begin
writeln ('Введите значения A и B:');read(a,b);
```

```
c:=a+b; writeln ('A+B=',c); c:=a-b;
writeln ('A-B=',c); end.
```

Текст этой программы структурирован явно неудачно, гораздо лучше он воспринимается так:

```
program p1;
var a,b,c:real;
begin
  writeln ('Введите значения A и B:');
  read (a,b);
  c:=a+b;
  writeln ('A+B=',c);
  c:=a-b;
  writeln ('A-B=',c);
end.
```

Основные действия программы *комментируются*. Комментарием в языке Паскаль считается любой текст, ограниченный фигурными скобками `{ ... }` или символами `(* ... *)`. Количество комментариев в программе никак не влияет на объем генерируемого машинного кода, они призваны, прежде всего, облегчить последующее чтение и модификацию исходного текста программы.

В качестве примера приведем законченную программу на языке Паскаль, вычисляющую вещественные корни произвольного квадратного уравнения:

```
program Equation;
var a,b,c,d,x1,x2:real; begin
 writeln ('Введите коэффициенты a,b,c:');
 read (a,b,c);
 d:=sqr(b)-4*a*c;
 x1:=(-b+sqrt(d))/(2*a);
 x2:=(-b-sqrt(d))/(2*a);
 writeln ('Корни уравнения');
 writeln (x1:10:2,x2:10:2);
 readln;
end.
```

В разделе описаний программы всем переменным, требуемым для решения задачи, присвоен тип `real`, и этот выбор вполне очевиден: коэффициенты `a`, `b` и `c` не обязательно целые значения. После вычисления дискриминанта и корней `x1`, `x2` (условие `d≥0` мы пока не проверяем), на экране печатается информационное сообщение «Корни уравнения», а затем с новой строки выводятся значения `x1` и `x2` с соблюдением указанных ширины и точности вывода. Наконец, оператор `readln;` в конце программы позволяет ей дождаться, пока пользователь не нажмет клавишу **Enter**. Если бы ввод данных был организован в виде `a:=1; b:=2; c:=0;` , это уменьшило бы до нуля полезность программы.

5 РАЗВЕТВЛЯЮЩИЙСЯ ВЫЧИСЛИТЕЛЬНЫЙ ПРОЦЕСС И УСЛОВНЫЙ ОПЕРАТОР

В первой части данного учебно-методического пособия были рассмотрены линейные вычислительные процессы, в которых порядок следования операторов последователен и неизменен. Однако этим программам недостает гибкости и умения принимать решения. Ведь уже несложный алгоритм решения квадратного уравнения предусматривает два варианта расчета, реальные же алгоритмы могут выдавать результаты, зависящие от десятков и сотен условий.

Разветвляющийся вычислительный процесс (РВП) реализуется по одному из нескольких направлений вычисления (*ветвей* алгоритма). Выбор одной из ветвей зависит от истинности или ложности некоторого условия (*логического выражения*), включенного в состав *условного оператора*. Программа должна учитывать *все* возможные ветви вычислений. При запуске программы, в зависимости от данных, выполняется *только одна* из возможных ветвей.

Логические выражения

Логические выражения строятся из арифметических выражений, операций отношения, логических операций и круглых скобок.

Результатом вычисления логических выражений (ЛВ) является одно из двух значений: `true` или `false`.

Операции отношения

Операции отношения (*сравнения*) имеют следующий общий вид:

```
AB1 OO AB2
```

где АВ – арифметические выражения, ОО – один из следующих знаков операции отношения:

```
<    <=   >    >=   =    <>
```

Последний знак обозначает отношение «не равно». Обратите также внимание на запись отношений «меньше или равно», «больше или равно».

В любое логическое выражение должна входить хотя бы одна операция отношения.

Приведем примеры ЛВ, включающих одну ОО:

`d<0` – выбор ветви вычислений зависит от значения `d`;

`sqr(x)+sqr(y)<=sqr(r)` – результат будет равен `true` для точек с координатами (*x*, *y*), лежащих внутри круга радиусом *R* с центром в начале координат;

`cos(x)>1` – результат этого ЛВ всегда равен `false`.

К вещественным значениям в общем случае *неприменима* операция = («равно») из-за неточного представления этих значений в памяти компьютера. Поэтому для вещественных переменных отношение вида `a=b` часто заменяется на `abs(a-b)<eps`, где `eps` – малая величина, определяющая допустимую погрешность.

Логические операции

Логические операции применимы только в логических выражениях и служат для составления сложных условий, требующих более одной операции отношения. В языке Паскаль определены логические операции, описанные в табл. 5.1.

Таблица 5.1

Логические операции языка Паскаль

Математическая запись	Запись на языке Паскаль	Название
¬	`not`	Отрицание
∧	`and`	Операция «И» (логическое умножение) – пересечение множеств
∨	`or`	Операция «ИЛИ» (логическое сложение) – объединение множеств
⊻	`xor`	Операция «исключающее ИЛИ»

Операция NOT применима к одному логическому выражению (является *унарной*). Ее результат равен `true`, если выражение ложно, и наоборот.

Например, выражение `NOT (sin(x)>1)` всегда даст значение `true`.

Операция `AND` связывает не менее двух логических выражения (является *бинарной*). Ее результат равен `true`, если все выражения истинны, или `false`, если хотя бы одно из выражений ложно.

В качестве примера распишем выражение $x \in [a,b]$. Так как операции принадлежности в языке Паскаль нет, используем операцию `AND` и операции отношения: `(x>=a) and (x<=b)`.

Математическое выражение *a, b, c* > 0 (одновременно) будет иметь вид `(a>0) and (b>0) and (c>0)`.

Операция `OR` также связывает не менее двух логических выражений. Ее результат равен `true`, если хотя бы одно выражение истинно, и `false`, если все выражения ложны.

Распишем выражение $x \notin [a,b]$. На языке Паскаль оно будет иметь вид `(x<a) or (x>b)`. Другой способ связан с применением операции NOT: `not ((x>=a) and (x<=b))`.

Условие «хотя бы одно из значений *a, b, c* положительно» может быть записано в виде `(a>0) or (b>0) or (c>0)`.

Условие «только одно из значений *a, b, c* положительно» потребует объединения возможностей операций AND и OR:

```
(a>0) and (b<=0) and (c<=0) or
```

```
(a<=0) and (b>0) and (c<=0) or
(a<=0) and (b<=0) and (c>0).
```

Операция XOR в отличие от OR возвращает значение «ложь» (false) и в том случае, когда все связанные с ней логические выражения истинны. Чтобы лучше уяснить это отличие, составим так называемую *таблицу истинности* двух логических операций (табл. 1.2). Для краткости значение false обозначим нулем, а true – единицей. Для двух логических аргументов возможны всего четыре комбинации значений 0 и 1.

Таблица 1.2

Таблица истинности операций OR, XOR и AND

Аргумент A	Аргумент B	A OR B	A XOR B	a AND b
0	0	0	0	0
0	1	1	1	0
1	0	1	1	0
1	1	1	0	1

В качестве примера использования операции XOR запишем условие «только одно из значений *a, b* положительно»:

```
(a>0) xor (b>0).
```

Приоритет логических операций следующий: самая старшая операция – not, затем and, следующие по приоритету – or и xor (равноправны между собой), самый низкий приоритет имеют операции отношения. Последнее служит причиной того, что в составных условиях отдельные отношения необходимо заключать в круглые скобки, как и сделано во всех примерах раздела.

5.1 Виды условных операторов

Короткий условный оператор

Это первый вид условного оператора, позволяющий программе выполнить или пропустить некоторый блок вычислений. Общий вид короткого условного оператора следующий:

```
if <логическое_выражение> then <оператор1>;
```

Здесь и далее треугольные скобки < > будут означать, что это выражение нужно будет заменить на соответствующее выражение согласно синтаксису языка Паскаль.

Сначала вычисляется логическое выражение. Если оно имеет значение true, то выполняется оператор1, иначе оператор1 игнорируется. Блок-схема соответствующего вычислительного процесса представлена на рис. 1.1.

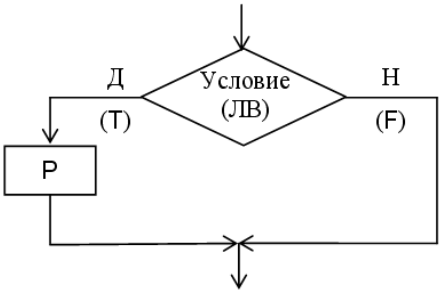


Рис. 5.1. Блок-схема короткого условного оператора

Если по условию требуется выполнить несколько операторов, их необходимо заключить в операторные скобки `begin...end;`, образуя единый *составной оператор*:

```
if d>0 then
begin
    x1:=(-b+sqrt(d))/(2*a);
    x2:=(-b-sqrt(d))/(2*a);
    writeln (x1:8:3,x2:8:3);
end;
```

Здесь по условию `d>0` выполняется три оператора, первые два из которых вычисляют корни `x1` и `x2` квадратного уравнения, а последний выводит на экран найденные значения корней.

Следующий пример иллюстрирует поиск значения $y = \max(a, b, c)$. Поскольку стандартной функции для нахождения максимума в языке Паскаль нет, применим два коротких условных оператора:

```
y:=a;
if b>y then y:=b;
if c>y then y:=c;
```

Вообще, для условной обработки N значений требуется N-1 короткий условный оператор.

Полный условный оператор

Эта форма условного оператора позволяет запрограммировать две ветви вычислений. Общий вид полного условного оператора следующий:

```
if <логическое_выражение> then <оператор1> else <оператор2>;
```

Оператор работает следующим образом: если логическое выражение имеет значение `true`, то выполняется `оператор1`, иначе выполняется `оператор2`. Всегда выполняется только один из двух операторов. Перед ключевым словом `else` («иначе») точка с запятой *не ставится*, так как `if-then-else` – единый оператор.

Вычислим значение $m = \min(x, y)$ с помощью полного условного оператора:

```
if x<y then m:=x else m:=y;
```

В следующем примере выполним обработку двух переменных: если значения *a* и *b* одного знака, найти их произведение, иначе заменить нулями.

```
if a*b>0 then c:=a*b
  else begin
       a:=0;
       b:=0;
       end;
```

Из примера видно, что к ветви алгоритма после ключевого слова `else`, состоящей более чем из одного оператора, также применяются операторные скобки.

Составной условный оператор

Эта форма условного оператора применяется, когда есть более двух вариантов расчета. Общий вид составного оператора может включать в себя произвольное число условий и ветвей расчета:

```
if <логическое_выражение1> then <оператор1>
else if <логическое_выражение2> then <оператор2>
...
else if <логическое_выражениеN> then <операторN>
else <оператор0>;
```

При использовании оператора последовательно проверяются логические выражения `1`, `2`, ... ,`N`. Если некоторое выражение истинно, то выполняется соответствующий оператор и управление передается на оператор, следующий за условным. Если все условия ложны, выполняется `оператор0` (если он задан). Число ветвей *N* неограниченно, ветви `else <оператор0>;` может и не быть.

Существенно то, что если выполняется более одного условия из *N*, обработано все равно будет только первое истинное условие. В показанной ниже программе составной условный оператор неверен, если ее разработчик хотел отдельно учесть значения `x`, меньшие единицы по модулю:

```
var x:real;
begin
 write ('Введите x:'); readln (x);
 if x<0 then writeln ('Отрицательный')
 else if x=0 then writeln ('Ноль')
 else if abs(x)<1 then
  writeln ('По модулю меньше 1')
 else writeln ('Больше 1');
end.
```

Условие `x<0` сработает, например, для значения `x=-0.5`, что не позволит программе проверить условие `abs(x)<1`.

Еще одну распространенную ошибку работы с составным условным оператором показывает произведенный ниже расчет знака *n* переменной *a*:

```
if a<0 then n:=-1;
if a=0 then n:=0
else n:=1;
```

Применение одного короткого и одного полного условных операторов является здесь грубой ошибкой, ведь после завершения короткого условного оператора для всех ненулевых значений *a* будет выполнено присваивание `n:=1`. Правильных вариантов этого расчета по меньше мере два.

1 вариант – с помощью трех коротких условных операторов.

```
if a<0 then n:=-1;
if a=0 then n:=0;
if a>0 then n:=1;
```

Вариант не очень хорош тем, что проверяет лишние условия даже тогда, когда знак уже найден.

2 вариант – с помощью составного условного оператора.

```
if a<0 then n:=-1;
else if a<0 then n:=1;
else n:=0;
```

Этот вариант лучше.

Можно сделать вывод, что при программировании многовариантных расчетов следует соблюдать осторожность, чтобы не допустить «потерянных» или неверно срабатывающих ветвей алгоритма.

В качестве еще одного примера рассчитаем значение функции, заданной графически (рис. 1.2).

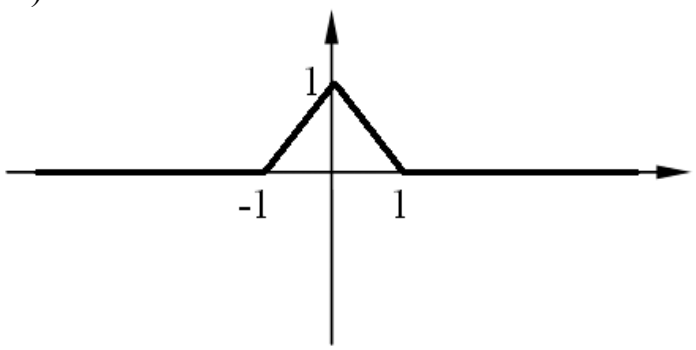


Рис. 5.2. Функция, заданная графически

Перепишем функцию в аналитическом виде:

$$y(x)=\begin{cases} x+1, & \text{если } x\in[-1,0[, \\ 1-x, & \text{если } \quad x\in[0,1], \\ 0, & \text{если } \quad |x|>1. \end{cases}$$

Для программирования вычисления `y(x)` можно использовать следующий вариант:

```
if abs(x)>1 then y:=0
else if x<0 then y:=x+1
else y:=1-x;
```

Возможна и последовательная проверка условий слева направо, которая породит немного избыточный, но легче воспринимающийся код:

```
if x<-1 then y:=0
else if x<0 then y:=x+1
else if x<1 then y:=1-x
else y:=0;
```

Вложенные условные операторы

Когда после ключевых слов `then` или `else` вновь используются условные операторы, они называются *вложенными*. Число вложений может быть произвольно, при этом действует правило: `else` всегда относится к ближайшему `then`. Часто вложением условных операторов можно заменить использование составного.

В качестве примера рассмотрим программу для определения номера координатной четверти `p`, в которой находится точка с координатами `(x,y)`. Для простоты примем, что точка не лежит на осях координат. Без использования вложений основная часть программы может иметь следующий вид:

```
if (x>0) and (y>0) then p:=1
else if (x<0) and (y>0) then p:=2
else if (x<0) and (y<0) then p:=3
else p:=4;
```

Однако использование такого количества условий представляется явно избыточным. Перепишем программу, используя тот факт, что каждое из условий `x>0`, `x<0` оставляет в качестве значения `p` только по две возможные четверти из четырех:

```
if x>0 then begin
 if y>0 then p:=1
 else p:=4;
end
else begin
 if y>0 then p:=2
 else p:=3;
end;
```

В первом фрагменте программы проверяются от двух до шести условий, во втором – всегда только два условия. Здесь использование вложений позволило существенно повысить производительность.

Рассмотренный выше пример с определением знака `n` числа `a` может быть переписан и с использованием вложения:

```
if a>0 then n:=1
else begin
 if a<0 then n:=-1
 else n:=0;
end;
```

5.2 Оператор выбора

Для случаев, когда требуется выбор одного значения из конечного набора вариантов, оператор `if` удобнее заменять оператором выбора (*переключателем*) `case`:

```
case <выражение> of
 <список1>: <оператор1>;
 <список2>: <оператор2>;
 . . .
 <списокN>: <операторN>;
 else <оператор0>;
end;
```

Оператор выполняется так же, как составной условный оператор.

Выражение должно иметь порядковый тип (целый или символьный). Элементы списка перечисляются через запятую, ими могут быть константы и *диапазоны* значений того же типа, что тип выражения. Диапазоны указываются в виде:

```
<Мин.значение> .. <Макс.значение>
```

Оператор диапазона записывается как *два* рядом стоящих символа точки. В диапазон входят все значения от минимального до максимального включительно.

В качестве примера по номеру месяца `m` определим число дней `d` в нем:

```
case m of
 1,3,5,7..8,10,12: d:=31;
 2: d:=28;
 4,6,9,11: d:=30;
end;
```

Следующий оператор по заданному символу `c` определяет, к какой группе символов он относится:

```
case c of
 'A'..'Z','a'..'z':    writeln ('Латинская буква');
 'А'..'Я','а'..'п','р'..'я':writeln ('Русская буква');
 '0'..'9':    writeln ('Цифра');
 else writeln ('Другой символ');
end;
```

Здесь отдельные диапазоны для русских букв от «а» до «п» и от «р» до «я» связаны с тем, что между «п» и «р» в кодовой таблице DOS находится ряд небуквенных символов.

Если по ветви оператора `case` нужно выполнить несколько операторов, действует то же правило, что и для оператора `if`, т.е. ветвь алгоритма заключается в операторные скобки `begin ... end;`.

5.3 Примеры программ с условным оператором

Приведем несколько примеров законченных программ, использующих РВП.

1 Проверить, может ли быть построен прямоугольный треугольник по длинам сторон `a`, `b`, `c`.

Проблема с решением этой задачи – не в проверке условия теоремы Пифагора, а в том, что в условии не сказано, какая из сторон может быть гипотенузой. Подходов возможно несколько: запрашивать у пользователя ввод данных по возрастанию длины сторон, проверять все три возможных условия теоремы Пифагора и т.п. Используем наиболее естественное решение: перед проверкой условия теоремы Пифагора упорядочим величины `a`, `b`, `c` так, чтобы выполнялись соотношения $a \le b \le c$. Для этого применим прием с обменом значений переменных.

```
var a,b,c, {Длины сторон}
 s:real;{Буферная переменная для обмена}
begin
{ Секция ввода данных }
 writeln;
 write ('Введите длину 1 стороны:');
 readln (a);
 write ('Введите длину 2 стороны:');
 readln (b);
 write ('Введите длину 3 стороны:');
 readln (c);
{ Сортируем стороны по неубыванию }
 if (a>b) then
     begin
      s:=a; a:=b; b:=s;
     end;
 if (a>c) then
     begin
      s:=a; a:=c; c:=s;
     end;
 if (b>c) then
     begin
      s:=b; b:=c; c:=s;
```

```
      end;
{ Проверка и вывод }
 if abs(a*a+b*b-c*c)<1e-8 then writeln
  ('Прямоугольный треугольник ',
    'может быть построен!')
 else writeln('Прямоугольный треугольник ',
    'не может быть построен!')
end.
```

2 Определить, попадает ли точка плоскости, заданная координатами (a, b), в прямоугольник, заданный координатами двух углов (x_1, y_1) и (x_2, y_2).

Как и в предыдущей задаче, было бы не совсем корректно требовать от пользователя вводить данные в определенном порядке, гораздо лучше при необходимости поменять *x* и *y* координаты прямоугольника так, чтобы пара переменных (x_1, y_1) содержала координаты левого нижнего угла прямоугольника, а (x_2, y_2) – правого верхнего.

```
var x1,y1,x2,y2,a,b:real;
begin
 writeln ('Введите координаты 1 угла:');
 read (x1,y1);
 writeln ('Введите координаты 2 угла:');
 read (x2,y2);
 if x1>x2 then begin
  a:=x1; x1:=x2; x2:=a;
 end;
 if y1>y2 then begin
  a:=y1; y1:=y2; y2:=a;
 end;
 writeln ('Введите координаты точки:');
 read (a,b);
 if (x1<=a) and (a<=x2)
    and (y1<=b) and (b<=y2) then writeln
   ('Точка попадает в прямоугольник')
 else writeln
   ('Точка не попадает в прямоугольник');
end.
```

3 Вводится денежная сумма в рублях и копейках. Программа печатает введенную сумму с правильной формой слов «рубли» и «копейки», например, «123 рубля 15 копеек».

Окончание, используемое для слов «рубли» и «копейки», зависит от последней цифры суммы, которую можно получить, взяв остаток от деления на 10 (1058 *рублей*, 38 *рублей* и т.д.). Исключения – суммы с последними двумя цифрами от 11 до 19 включительно, которые всегда произносятся «рублей» и «ко-

пеек» (511 *рублей*, но 51 *рубль*). Используя эту информацию, составим программу.

```
var r,k,o10,o100:integer;
begin
 writeln;
 write ('Введите количество рублей, ',
     'затем пробел и количество копеек:');
 read (r,k); writeln;
 o10:=r mod 10;   {Взяли последнюю цифру}
 o100:=r mod 100; {...и 2 последних цифры}
 write ('Правильно сказать: ',r,' ');
 {Печатаем число рублей, затем пробел}
 if (o100>10) and (o100<20)
     or (o10>4) or (o10=0) then
  write ('рублей')
 else if (o10>1) and (o10<5) then
  write ('рубля')
 else
  write ('рубль');
 {аналогично для копеек:}
 o10:=k mod 10;
 o100:=k mod 100;
 write (' ',k,' ');
 {печатаем число копеек с пробелами}
 if (o100>10) and (o100<20) or
     (o10>4) or (o10=0) then
  write ('копеек')
 else if (o10>1) and (o10<5) then
  write ('копейки')
 else write ('копейка');
end.
```

6 ОПЕРАТОРЫ ЦИКЛА

6.1 Понятие цикла

Большинство практических задач сводится к многократному выполнению однотипных операций. Для их реализации используется циклический вычислительный процесс (ЦВП), или цикл. Термин ***цикл*** происходит от латинского слова *circle* – круг, т.е. это многократное повторение одних и тех же вычислений над некоторым набором данных.

Повторяемый блок вычислений называют ***телом цикла***.

Количество повторений в цикле отслеживается с помощью ***счетчика цикла*** – специально выделенной переменной.

На счетчик накладывается ***условие выхода*** из цикла, определяющее, до каких пор или сколько раз следует выполнять тело цикла. Для того чтобы не было **зацикливания**, условие выхода из цикла должно быть **выполнимо** и обеспечивало некоторое изменение входящих в него переменных

В теле цикла должен быть оператор, изменяющий значение счетчика ***на шаг*** или на какое-либо другое так, чтобы условие выхода стало в какой-то момент выполнимо и цикл мог завершиться, а *не зациклиться*.

Однократное выполнение тела цикла, или один виток цикла, называют ***итерацией***.

Таким образом, для организации цикла достаточно определить условие, управляющее числом его повторений и описать операторы, образующие тело цикла.

Пример 1. Составить словесный алгоритм для вычисления и вывода на экран элементов последовательности: 1^2, 3^2, 5^2, 7^2…11^2.

Решение

1. Начало
2. $y = 1$
3. Вывести y^2 – *тело цикла*
4. $y = y + 2$
5. Если $y \leq 11$, то переход к п. 3 иначе к п. 6 – *условие выхода из цикла*
6. Конец

Пункты 3 и 4 – это тело цикла, пункт 5 – условие выхода из цикла. Переменная y – счетчик цикла, который организует переход к следующей итерации цикла.

Блок-схема, соответствующая этому алгоритму, представлена на рис. 1.

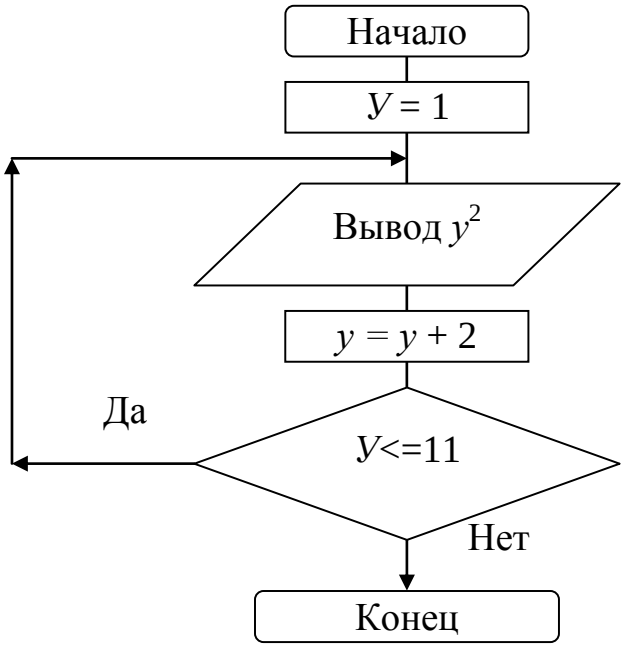


Рис. 6.1. Блок-схема алгоритма

Пример 2. Составить словесный алгоритм для вычисления и вывода на экран элементов знакопеременной последовательности: -7, 10, -13, 16,…, -25.

Решение

Для организации этого цикла введем три переменные:
znak = -1, 1, …. -1 отвечает за знак элемента последовательности
y = 7, 10, 13, 16,… 25 отвечает за модуль элемента
a = -7, 10, -13, 16…,-25 – элемент заданной последовательности
a = znak **y*

1. Начало
2. znak = -1
3. y:=7;
4. a:=znak*y;
5. Вывести a;
6. znak=-znak;
7. y:=y + 3;
8. Если y ≤ 25, то переход к п. 4, иначе к п. 9
9. конец

Пункты 4–7 – это тело цикла, пункт 8 – условие выхода из цикла. Переменная *y* – счетчик цикла, который организует переход к следующей итерации цикла.

Итак, для создания цикла необходимо задать счетчику *начальное значение,* описать операторы, образующие тело цикла, оператор, изменяющий значение *счетчика на шаг*, и определить условие выхода из цикла.

В языке Pascal существуют три типа цикла:

1 Оператор цикла с предусловием (**while**).

2 Оператор цикла с постусловием (**repeat**).

3 Оператор цикла со счетчиком (**for**).

6.2 Оператор цикла с предусловием while

Название этого цикла обусловлено тем, что проверка условия выхода в нем предшествует выполнению тела цикла. Изобразим этот цикл в виде блок-схемы (рис. 2).

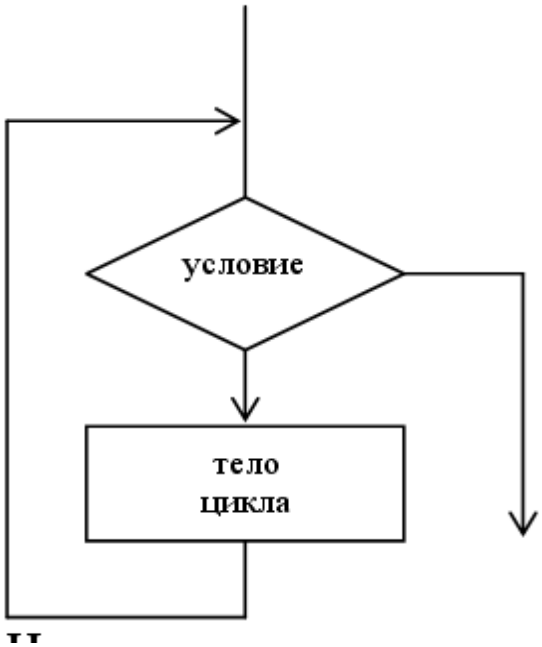


Рис. 6.2. Блок-схема цикла с предусловием

Цикл с предусловием имеет следующий синтаксис:

```
while <логическое_выражение> do
 begin
    <операторы тела цикла>
 end;
```

Опишем порядок работы этого оператора. Сначала проверяется условие стоящее после слова ***while*** **(**«пока»**)**, если оно ложно, то тело цикла не выполняется ни разу и происходит выход из цикла, если оно истинно, то выполняется тело цикла, стоящее после ***do*** («делать», «выполнять»). Далее, после завершения тела цикла управление вновь передается на проверку условия и т.д. Таким образом, цикл выполняется до тех пор, пока условие будет истинно. Естественно, предполагается, что в теле цикла было обеспечено некоторое изменение входящих в условие переменных, в противном случае произойдет *зацикливание.*

Если тело цикла состоит более чем из одного оператора, оно заключается в операторные скобки `begin ... end;`.

Пример. Составить программу с оператором **`while`** для вычисления и вывода на экран элементов последовательности: 1^2, 3^2, 5^2, 7^2…11^2.

```
program r1;
var y:integer;
   begin
    y:=1;
     while y<=11 do
       begin
         writeln('a=',y*y);
         y:=y+2;
       end;
end.
```

6.3 Оператор с постусловием repeat

Название данного цикла обусловлено тем, что проверка условия выхода последует после выполнения тела цикла. Изобразим этот цикл в виде блок-схемы (рис. 6.3).

Цикл с постусловием имеет следующий синтаксис:

```
repeat
   <операторы тела цикла>
until <логическое_выражение>;
```

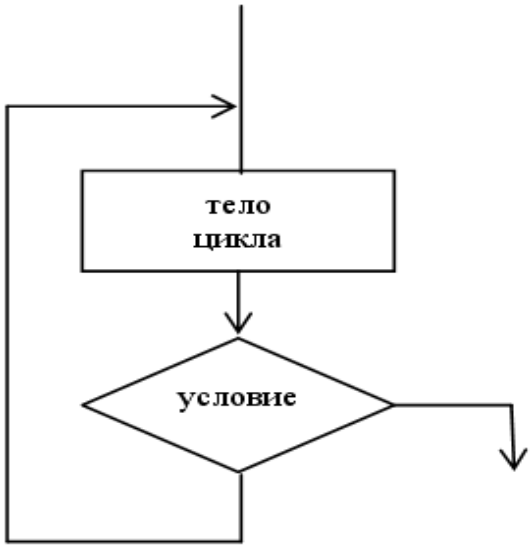


Рис. 6.3. Блок-схема цикла с постусловием

Опишем порядок работы оператора **repeat** («*повторять*»). Сначала выполняется тело цикла в любом случае, далее проверяется логическое выражение, стоящее после слова ***until*** («*до тех пор, пока не*»), если выражение ложно, то тело цикла выполняется еще раз и так до тех пор, пока логическое выражение не станет истинным. Выход из цикла и переход к следующему оператору за циклом происходит, когда условие станет истинным.

Обратите внимание, что в отличие от `while` цикл `repeat` работает, пока условие *ложно*. Это отличие подчеркивается использованием ключевого слова `until` («до тех пор, пока не») вместо `while` («до тех пор, пока»).

С учетом приведенных описаний очевидно основное различие двух циклов: цикл с постусловием гарантированно *выполняется хотя бы раз*, а цикл с предусловием может не выполняться ни разу, если условие сразу же окажется ложным.

Тело цикла `repeat`, даже если оно состоит из нескольких операторов, можно не заключать в операторные скобки **`begin ... end;`**

Для наглядности приведем пример применения оператора **`repeat`** для той же задачи, что и в предыдущем примере.

Пример. Составить программу с оператором **`repeat`** для вычисления и вывода на экран элементов последовательности: 1^2, 3^2, 5^2, 7^2…11^2.

```
program r2;
var y:integer;
   begin
    y:=1;
     repeat
       writeln('a=',y*y);
       y:=y+2;
   until y>11;
end.
```

6.4 Оператор цикла со счетчиком for

Для обработки заранее известного количества повторений с шагом счетчика, равным единице, удобнее использовать *цикл со счетчиком* (цикл *for*). Он имеет следующий синтаксис:

```
for <счетчик> := <НЗ> to <КЗ> do
begin
 <операторы тела цикла>
end;
```

Здесь <**счетчик**> – это переменная, обязательно типа **integer** (или **char**, словом, любого перечислимого типа), **НЗ** (начальное) и **КЗ** (конечное) значения счетчика – целочисленные выражения или константы.

Работает цикл *for* следующим образом. Сначала присваивается начальное значение счетчику, если оно меньше конечного значения, то тело цикла выполняется первый раз, далее счетчик увеличивается на 1, если счетчик по-прежнему меньше конечного, то тело цикла выполняется второй раз и т.д. до тех пор, пока значение счетчика не станет больше конечного.

Если требуется, чтобы значение счетчика уменьшалось, а не увеличивалось, вместо ключевого слова **to** используется **downto**.

Если в теле цикла операторов больше одного, то необходимо заключить их в операторные скобки **begin ... end;**

Пример. Составить программу с оператором **for** для вычисления и вывода на экран элементов последовательности: 1^2, 3^2, 5^2, 7^2…11^2.

```
program r3;
var y,i:integer;
   begin
    y:=1;
     for i:=1 to 6 do
       begin
         writeln('a=',y*y);
         y:=y+2;
       end;
end.
```

6.5 Алгоритмы накопления суммы и произведения

Данные алгоритмы применяются, когда требуется сложить или перемножить выбранные данные. Эти простейшие задачи относятся к итерационным процессам, так как последующее значение суммы или произведения вычисляется через предыдущие значения.

В общем виде алгоритм накопления суммы можно описать так:

1 Для подсчета суммы описать переменную того же типа, что суммируемые данные.

2 До начала цикла переменная-сумма очищается нулем, так как прибавление к сумме 0 не меняет ее значения.

3 В теле цикла вычисляется очередное слагаемое.

4 В теле цикла переменная-сумма увеличивается на очередное слагаемое. Происходит накопление суммы, т.е. к старому значению переменной-суммы прибавляется очередное слагаемое и результат присваивается в ту же переменную оператором вида `s:=s+a;`

5 Как и в любом цикле, увеличение счетчика на шаг и проверка условия выхода из цикла.

Вывод результата суммирования происходит после цикла.

Очевидно, что начальное значение произведения равно `1`, а не `0`. После оператора `p:=0;` оператор `p:=p*t;`, расположенный в теле цикла, будет возвращать только нули.

Пример. Составить программу для вычисления суммы элементов последовательности: sum = $1^2 + 3^2 + 5^2 + 7^2 + \ldots + 11^2$. Использовать оператор цикла `while`.

```
program r1;
var y, a,sum:integer;
   begin
    y:=1; sum:=0;
     while y<=11 do
       begin
         a:=y*y;
         sum:=sum+a;
         y:=y+2;
       end;
writeln('sum=',sum);
end.
```

Пример. Составить программу для вычисления суммы элементов последовательности sum = $1^2 + 3^2 + 5^2 + 7^2 + \ldots + 11^2$. Использовать оператор цикла `for`.

```
program r3;
var y,I,a,sum:integer;
   begin
    y:=1;sum:=0;
     for i:=1 to 6 do
       begin
         a:=y*y;
```

```
          sum:=sum+a;
          y:=y+2;
        end;
 writeln('sum=',sum);
end.
```

Пример. Вычислить сумму $y = \sum_{n=1}^{15} \frac{1}{n}$.

```
program r5;
var a,s:real;
    n:integer;
 begin
S:=0;
for n:=1 to 15 do
 begin
  a:=1/n;
  S:=S+a;
 end;
writeln('S=',S);
end.
```

Пример. Вычислить сумму

$$S = \frac{1}{\sqrt[3]{1}} + \frac{1}{\sqrt[3]{2}} + \frac{1}{\sqrt[3]{4}} + \frac{1}{\sqrt[3]{8}} + \frac{1}{\sqrt[3]{16}} + \ldots + \frac{1}{\sqrt[3]{256}}.$$

```
program r5;
var k,a,s:real;
 begin

S:=0; k:=1;
repeat
   a:=1/exp(1/3*ln(k));
   S:=S+a;
   k:=k*2;
until k>256;
writeln('S=',S);
end.
```

7 СОСТАВНЫЕ ТИПЫ ДАННЫХ

7.1 Одномерные массивы и их описание

Массив – упорядоченный набор однотипных переменных, называемых *элементами* массива. Каждый элемент имеет целочисленный порядковый номер, называемый *индексом*. Число элементов в массиве называют его *размерностью*.

Массивы используются там, где нужно обработать сразу несколько переменных одного типа, например, оценки всех 20 студентов группы или координаты 10 точек на плоскости. Элементы массива занимают определенный объем оперативной памяти, и однажды полученные остаются доступными в течение всего сеанса работы программы и не требуют повторного вычисления или чтения из файла. Это порождает круг приложений, связанных с типовой обработкой наборов данных, таких как вычисление математических и статистических характеристик векторов, поиск нужных значений в массивах и т.д.

Массив описывается в разделе `var` оператором следующего вида:

```
var <ИмяМассива>: array [<НЗ> .. <КЗ>] of <Тип>;
```

где `НЗ` – начальное значение диапазона изменения индекса,

`КЗ` – конечное значение диапазона изменения индекса,

`Тип` – любой из известных типов данных Паскаля. Каждый элемент массива будет рассматриваться как переменная соответствующего типа.

Опишем несколько массивов разного назначения.

```
var a: array [1..20] of integer;
```

Здесь мы описали массив с именем `A`, состоящий из 20 целочисленных элементов;

```
var x,y : array [1..10] of real;
```

Описаны 2 массива с именами `x` и `y`, содержащие по 10 вещественных элементов;

```
var t : array [0..9] of string;
```

Массив `t` состоит из 10 строк, которые занумерованы с нуля.

Легко увидеть, что *размерность* (число элементов) массива вычисляется как `ВИ – НИ + 1`.

Замечание. Диапазон изменения индекса не должен быть описан через переменные, т.е. можно указывать только константы или константные выражения.

Например:

```
const n=10;
var mass : array [1 . . n] of real;
b : array [-5 . . 5] of integer;
a : array [0 . . 2*n] of integer;
```

Для обращения к отдельному элементу массива используют оператор вида `ИмяМассива [Индекс]`.

Здесь Индекс – целочисленный номер элемента. Индекс не должен быть меньше значения нижнего или больше верхнего индекса массива, иначе возникнет ошибка «Constant out of range». Отдельный элемент массива можно использовать так же, как переменную соответствующего типа, например:

```
A[1]:=1;
x[1]:=1.5; y[1]:=x[1]+1;
t[0]:='Hello';
```

Каждый элемент массива имеет один номер (индекс), характеризующий его положение в массиве. В математике понятию одномерного массива из n элементов соответствует понятие *вектора* из n компонент: $A = \{A_i\}$, $i = 1, 2, ..., n$.

Ввод, вывод и обработка массивов

Как правило, ввод, обработка и вывод массива осуществляются поэлементно, с использованием цикла for.

Простейший способ ввода – ввод массива с клавиатуры:

```
const n = 10;
var a: array [1..n] of real;
 i:integer;
begin
writeln ('Введите элементы массива');
for i:=1 to n do read (A[i]);
```

Размерность массива определена константой n, элементы вводятся по одному в цикле for, при запуске этой программы пользователю придется ввести 10 числовых значений. При решении учебных задач вводить массивы «вручную», особенно если их размерность велика, не всегда удобно. Существует альтернативное решение.

Формирование массива из случайных значений уместно, если при решении задачи массив служит лишь для иллюстрации того или иного алгоритма, а конкретные значения элементов несущественны. Для того чтобы получить очередное случайное значение, используется стандартная функция random(N), где параметром N передается значение порядкового типа. Она вернет случайное число того же типа, что и тип аргумента, лежащее в диапазоне от 0 до N–1 включительно. Например, оператор вида a[1]:=random(100); запишет в a[1] случайное число из диапазона [0,99].

Для того чтобы при каждом запуске программы цепочка случайных чисел была новой, перед первым вызовом random следует вызвать стандартную процедуру randomize;, запускающую генератор случайных чисел. Приведем пример заполнения массива из 20 элементов случайными числами, лежащими в диапазоне от –10 до 10:

```
var a:array [1..20] of integer;
 i:integer;
begin
 randomize;
```

```
 for i:=1 to 20 do
  begin
  a[i]:=random(21)-10;
  write (a[i]:4);
 end;
end.
```

К массивам применимы все типовые алгоритмы, изученные в теме «Циклы». Приведем один пример, в котором вычисляется сумма s положительных элементов массива:

```
var b:array [1..5] of real;
s:real; i:integer;
begin
writeln ('Введите 5 элементов массива');
for i:=1 to 5 do read (b[i]);
s:=0;
for i:=1 to 5 do if b[i]>0 then s:=s+b[i];
```

Вывод массива на экран также делается с помощью цикла for:

```
for i:=1 to 5 do write (b[i]:6:2);
```

Здесь 5 элементов массива b напечатаны в одну строку. Для вывода одного элемента на одной строке можно было бы использовать оператор writeln вместо write.

Существенно то, что, если обработка массива осуществляется последовательно, по одному элементу, циклы ввода и обработки зачастую можно объединить, как в следующем примере.

Найти арифметическое среднее элементов вещественного массива t размерностью 6 и значение его минимального элемента.

```
var b:array [1..6] of real;
 s, min:real;
 i:integer;
begin
s:=0; min:=1e30;
writeln ('Ввод B[6]');
for i:=1 to 6 do
 begin
 read (b[i]);
 s:=s+b[i];
 if b[i]<min then min := b[i];
 end;
writeln ('min=',min,' s=', s/6);
end.
```

Теоретически в этой программе можно было бы обойтись и без массива, ведь элементы b[i] используются только для накопления суммы и поиска максимума, так что описание массива вполне можно было заменить описанием

вещественной переменной b. Однако в реальных задачах данные, как правило, обрабатываются неоднократно и без массивов обойтись трудно. Приведем пример учебной задачи, где использование массива дает выигрыш за счет уменьшения объема вычислений, выполняемых программой.

Задана последовательность T_i = max {sin i, cos i}, i= -5, -4, ..., 5. Найти элемент последовательности, имеющий минимальное отклонение от арифметического среднего положительных элементов.

Здесь в первом цикле можно сформировать массив по заданному правилу и найти арифметическое среднее положительных элементов. Во втором цикле, когда среднее известно, можно искать отклонение. Без использования массива нам пришлось бы считать элементы последовательности дважды.

```
var t : array [-5..5] of real;
 i,k:integer;
 s, ot : real;
begin
 s:=0; k:=0;
 for i:=-5 to 5 do
 begin
  t[i]:=sin(i);
  if t[i]<cos(i) then t[i]:=cos(i);
  if t[i]>0 then
   begin
   k:=k+1; s:=s+t[i];
   end;
 end;
 s:=s/k;
 ot:=1e30;
 for i:=-5 to 5 do
 begin
  if abs(t[i]-s)<ot then ot:= abs(t[i]-s);
 end;
 writeln ('Ot=', ot:8:2);
end.
```

Распространена обработка в одной задаче сразу нескольких массивов. Приведем пример.

Координаты 10 точек на плоскости заданы массивами x={x_i}, y={y_i}, i=1, 2, ..., 10. Найти длину ломаной, проходящей через точки (x_1, y_1), (x_2, y_2), ..., (x_{10}, y_{10}), а также номер точки, лежащей дальше всего от начала координат.

При решении задачи используем формулу для нахождения расстояния между двумя точками на плоскости, заданными координатами (x_1, y_1) и (x_2, y_2):

$$r = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Обозначим через r расстояние между текущей точкой и следующей, Len – искомую длину ломаной, Dist – расстояние от текущей точки до начала координат, max – максимальное из этих расстояний, Num – искомый номер точки.

```
var x,y : array [1..10] of real;
I, num:integer;
r, Len, Dist, max : real;
begin
 {найдем max при вводе данных}
 max:=0; {т.к. расстояние не м.б. <0 }
 num:=1; {на случай, если все точки (0,0)}
 writeln ('Введите координаты 10 точек');
 for i:=1 to 10 do
  begin
  read (x[i], y[i]);
  Dist:=sqrt (sqr(x[i]) + sqr (y[i]));
  if dist > max then
   begin
    max:=dist; {запомнили новое расстояние}
    Num:=i;    {и номер точки}
   end;
 end;
 writeln ('Номер точки=',num,
  ' расстояние=',dist:8:2);
 Len:=0;{длина ломаной - сумма длин сторон}
 for i:=1 to 9 do
  begin   {у 10-й точки нет следующей!}
   r:= sqrt(sqr(x[i]-x[i+1])+sqr(y[i]-y[i+1]));
   Len:=Len+r;
  end;
 writeln ('Длина ломаной=',len:8:2);
end.
```

7.2 Сортировка массивов

Сортировка – распределение элементов множества по группам в соответствии с определенными правилами.

Например, сортировка «по невозрастанию» – это сортировка элементов массива, в результате которой получается массив, каждый элемент которого, начиная со второго, не больше стоящего от него слева. Существует достаточно большое число методов сортировки. Приведем лишь простейшие из них.

Линейная сортировка (отбором). Пусть необходимо упорядочить массив по возрастанию (убыванию) элементов.

А л г о р и т м:

1. Просматриваем элементы, начиная с первого. Ищем минимальный (максимальный). Меняем его местами с первым элементом.
2. Просматриваем элементы, начиная со второго. Ищем минимальный (максимальный). Меняем его местами со вторым элементом.
3. И т. д. до предпоследнего элементата.

П р и м е р. Отсортировать массив символов по алфавиту.

```
Program Linsort;
  Const Maxkol=26;
  Var
    C:Array[1..Maxkol] Of Char;
    K,I,J:Integer;
    Vr:Char;
  Begin
    Write('введите количество символов <= 26');
    Readln(K);
    Writeln('введите ',K,' символов ');
    For I:=1 To K
     Do Readln(C[I]);
    For I:=1 To K-1   {измен. размер неотсортирован.части
     Do For J:=I+1 To K                          массва}
         Do If C[J]<C[I]       {сравнивать по очереди I-й
             Then Begin         элемент неотсортированной
                    Vr:=C[I];   части массива со всеми от
                    C[I]:=C[J];   I+1-го до конца, если в
                    C[J]:=Vr;     неотсортированной части
                  End;     массива нашли элемент, больший
   For I:=1 To K             I-го, то обменять их местами}
      Do Write(C[I]);
   Writeln
End.
```

Сортировка методом «пузырька». Данный метод получил такое название по аналогии с пузырьками воздуха в стакане воды. Более «легкие» (макси-

мальные или минимальные) элементы постепенно «всплывают». В отличие от линейной сортировки, сравниваются только пары соседних элементов, а не каждый элемент со всеми (поэтому такая сортировка выполняется за меньшее число шагов, а следовательно, быстрее).

П р и м е р. Отсортировать по убыванию массив методом «пузырька».

А л г о р и т м:

1. Последовательно просматриваем пары соседних элементов массива (m).
2. Если для соседних элементов выполняется условие m[i-1]<m[i], то значения меняются местами.

```
Program Sortpuz;
  Const Maxkol=26;
  Var
     C:Array[1..Maxkol] Of Char;
     K,I,J:Integer;
     Vr:Char;
  Begin
     Write('введите количество символов '); Readln(K);
     Writeln('введите ',K,' символов ');
     For I:=1 To K Do Readln(C[I]);
     For I:=2 To K Do
     For J:=K Downto I
      Do If C[J-1]<C[J]
          Then Begin           {вытеснить элемент справа
                  Vr:=C[J-1];  влево -пузырек "всплывет"}
                  C[J-1]:=C[J];
                  C[J]:=Vr
               End;
    For I:=1 To K
     Do Write(C[I])
    Writeln
End.
```

7.3 Двумерные массивы (матрицы)

Определение типа, значениями которого являются матрицы, выполняется следующим образом:

```
Type <имя типа>=Array[<диапазон индекса строки>,
          <диапазон индекса столбца>] Of <тип компонент>;
```

Количество индексов ***2*** определяет размерность массива, а сами индексы разделяются запятыми и заключаются в квадратные скобки.

П р и м е р

```
Type
   Matr=Array[1..2,1..12] Of Real;
Var
```

A,B,C:Matr;

Массив можно описать в разделе ***Var*** следующим образом:

```
<идентификатор>:array[<диапазон первого индекса>,...,
        <диапазон n-го индекса>] of <тип компонент>;
```

П р и м е р:

```
Var
   A,B,C:Array[1..10] Of Integer;
```

Для обращения к элементам массива используются конкретные значения индексов. Индекс представляет собой выражение любого простого (скалярного) типа (кроме real). К примеру, оператор **`B[3]:=10;`** присваивает третьему элементу одномерного массива с именем ***B*** значение 10.

П р и м е р.

Пусть двумерный массив описан следующим образом:
`Var A:Array[1..2,1..4] Of Integer`; а в памяти эвм записана таблица чисел, представляющая этот массив:

17	11	4	5
22	8	16	12

Все элементы в таблице имеют тип ***Integer***. При обращении к элементам матрицы ***A*** первый индекс указывает номер строки таблицы (изменяется в данном случае от 1 до 2), второй – номер столбца (в нашем примере изменяется от 1 до 4). Если задать оператор присваивания в виде `X:=A[2,3];` то после его выполнения значение некоторой переменной ***X*** будет равно 16.

Ввод и вывод значений элементов массива производится поэлементно.

Пример.

```
{программа нахождения произведения двух матриц A размером
M*N и B размером N*K. Элементы результирующей матрицы C
```

размером M*K рассчитываются по формуле $C_{ij} = \sum_{k=1}^{N} a_{ik} \cdot b_{kj}$. Значения m, n, k не превышают 10}

```
Program Msg3;
   Const Em=10;
   Type
      Matr=Array[1..Em, 1...Em] Of Real;
   Var
      A,B,C:Matr;
      M,N,K,I,J,T:Integer;
```

```
   Begin
      Write('введите значения M, N, K ');
      Readln(M,N,K);
      For I:=1 To M
        Do For J:=1 To N
            Do Readln(A[I,J]);{ввод элементов матрицы A}
      For I:=1 To N
        Do For J:=1 To K
             Do Readln(B[I,J]);{ввод элементов матрицы B}
For I:=1 To N
        Do For J:= To K
             Do Begin
                   C[I,J]:=0;
                   For T:=1 To N       {умножение матриц}
                    Do C[I, J]:=C[I,J]+A[I,T]* B[T,J];

                End
End.
```

В Паскале разрешается присваивать значения одной переменной массива другой (если элементы массива имеют один тип и одинаковую размерность). К примеру, если массивы ***A*** и ***B*** имеют одинаковую размерность и тип элементов real, то допустимо присваивание: `A := B.`

7.4 Записи

Запись – это структура данных, состоящая из фиксированного числа компонентов, называемых полями записи. В отличие от массива, поля записи могут быть различного типа. Чтобы можно было ссылаться на тот или иной компонент записи, поля именуются.

Объявление типа записи выглядит следующим образом:

```
Type
   <имя типа>=Record
                  <список полей>
               End;
```

Здесь <имя типа> – правильный идентификатор; <список полей> – список полей; представляет собой последовательность разделов записи, разделяемых точкой с запятой. Каждый раздел записи представляет собой один или несколько идентификаторов полей, отделяемых друг от друга запятыми. За идентификаторами ставится двоеточие и описание типа поля.

Как было указано в п. 3.1., запись может быть элементом массива.

П р и м е р

```
Type
```

```
Rec=Record
          A:Integer;
          B:Real;
          C:String[10]
       End;
Var
   Elem:Rec;
   Tabl:Array[1..100] Of Rec;
```

Обращение к элементам записи осуществляется с помощью составных имен. Составное имя начинается с имени записи и содержит список имен полей (разделенных точками), входящих в цепочку, ведущую к требуемому элементу.

Например, к полям вышеприведенной записи можно обратиться по именам:

Elem.A, Elem.B, Elem.C, Elem.C[5],

А к полям ***i***-го компонента массива записей – по именам:

Tabl[I].A, Tabl[I].B, Tabl[I].C, Tabl[I].C[1].

Для сокращения обозначения полей записи (когда ведется работа с несколькими полями одной и той же записи) используется оператор присоединения:

```
With <переменная запись>
 Do <оператор>;
```

Внутри оператора, входящего в оператор присоединения, компоненты записи обозначаются с помощью только имен полей (имя переменной-записи перед ними не указывается). Заголовок операторов может содержать список переменных-записей, разделенных запятыми:

```
With <переменная-запись_1>, …<переменная-запись_N>
 Do <оператор>;
```

П р и м е р ы

1.
```
With Elem
    Do Begin
         A:=1998;
         B:=37.5,
         C:='И.И. Петров'
      End;
```

2.
```
For I:=1 To 10 Do
   With Tabl[I]
    Do Begin
         Read(A,B);
         Readln(C)
      End;
```

П р и м е р. Пусть запись содержит сведения о студенте и состоит из полей: личный номер студента, фамилия и инициалы, номер курса, номер группы, средние оценки за каждый год учебы. Необходимо ввести исходные данные.

{ввод элементов записи}

```
ConstNmax=30;
Type
   Zap=Record
          Nom:Byte;
           Fio:String[20];
         Kurs, Group : Byte;
         Ocen : Array[1..5] of Real;
         End;
     Arr_zap = Array[1..Nmax] Of Zap;
Var
   List:Arr_Zap;
   N:1..Nmax;
Begin
   Write('задайте количество студентов в списке');
   Readln(N);
  For I:=1 To N
   Do With List[I]
        Do Begin
              Write('задайте личный номер студента');
              Readln(Nom);
              Write('задайте фамилию ');
              Readln(Fio);
              Write('курс и группа ');
              Readln(Kurs, Gruppa);
              Write('средние оценки за каждый год ');
              For J:=1 To 5
               Do     Readln(Ocen[I]);
              Readln
           End
 End.
```

7.5 Строки

Для обработки текстов в Паскале используется тип ***String*** (строка). Строка трактуется как цепочка символов. К любому символу можно обратиться так же как к элементу одномерного массива ***Array[0..N] Of Char***. Количество символов в строке может меняться от 0 до N, где N – максимальное количество символов в строке. Значение n объявляется определением типа ***String[N]*** и может быть любой константой порядкового типа, но не больше 255. Самый первый байт в строке имеет индекс 0 и содержит текущую длину строки.

Действия над строками реализуются с помощью стандартных процедур и функций.

Length(St) – функция типа ***Integer***, возвращает длину строки ***St***.

Concat(S1…Sn) – функция типа ***String***, возвращает строку, представляющую собой сцепление строк ***S1…Sn.***

Copy(St, Index, Count) – функция типа ***String***, копирует из строки st count символов, начиная с символа с номером index.

Delete(St, Index, Count) – процедура, удаляющая ***Count*** символов из строки ***St***, начиная с символа с номером ***Index***.

Insert (St, Index, Count) – процедура, вставляющая count символов в строку ***St***, начиная с символа с номером ***Index***.

Pos(Subst, St) – функция типа ***Integer***, отыскивает в строке ***St*** первое вхождение подстроки subst и возвращает номер позиции, с которой она начинается. Если подстрока не найдена, возвращается ноль.

Str (X, St) – процедура, преобразует число ***X*** любого вещественного или целого типа в строку символов ***St*** так, как это делает процедура ***Writeln*** перед выводом. После ***X*** можно задать формат преобразования (как в процедуре вывода).

Val(St, X, Code) – процедура, преобразует строку ***St*** во внутреннее представление целой или вещественной переменной ***X***, которое определяется типом этой переменной. Параметр code содержит ноль, если преобразование прошло успешно и порядковый номер первого ошибочного символа в строке ***St*** в противном случае.

П р и м е р ы использования приведенных процедур и функций

```
Var
     X:Real;
     Y:Integer;
     St,St1:String;
Begin
  St:=Concat('12'; '345');     {строка St содержит 12345}
  St1:=Copy(St,3,Length(St)-2);       {St1 содержит 345}
  Insert('-', St1,2);          {Строка St1 содержит 3-45}
  Delete(St,Pos('2',St),3);       {Строка St содержит 15}
  Str(Pi:6:2,St);                   {Строка St содержит
3.14}
  Val('3,1415',X,Y);          {Y Содержит 2, X остался Без
End.                                              изменений}
```

П р и м е р ы программ с использованием строкового типа

1.
```
{Подсчитать количество цифр в произвольной строке}
Program St_1;
   Var S:String;
       I,Nd:Byte;
```

```
   Begin
      Write(‘ введите произвольную строку ');
      Readln(S);
      Nd:=0;
      For I := 1 To Length(S)
       Do If S[I] In [‘0’..’9’]
           Then Inc(Nd);
      Writeln(Nd);
      Readln
   End.
```

2. {Подсчитать сумму цифр в целом положительном числе}

```
Program St_2;
   Var
      X:Word;
      I,d,Sum:Byte;
      Code:Integer;
   Begin
      Write(‘ введите целое положительное число');
      Readln(X);
      Str(X, S);
      For I := 1 To Length(S)
       Do Begin
             Val(S[I], D, Code);
             Sum := Sum + D;
             If S[I] In [‘0’..’9’]
              Then Inc(Nd);
          End;
      Writeln(Nd);
      Readln
   End.
```

7.6 Множества

Наряду с числом множество является фундаментальным математическим понятием. К операциям и отношениям со множествами сводится большинство математических моделей. Паскаль – один из немногих языков, который имеет встроенные средства для работы со множествами. Давайте посмотрим, как «переводятся» на паскаль некоторые понятия теории множеств.

В математике рассматривают конечные и бесконечные множества, состоящие из произвольных элементов. В паскале множества всегда конечные, причем состоят не более чем из 256 элементов. Все элементы множества должны быть одного порядкового типа (например, ***Integer, Word, Longint***).

Пусть, например, необходимо задать множества.

Обозначения, принятые:

В математике	В Паскале
{1, 2, 3}	[1, 2, 3]
0	[ ]
{1, 2,…n}	[1…n]

В паскале множество может быть задано выражениями, например **[2+x, 8-3].**

Внутреннее представление множеств. Все значения множества представляются в памяти последовательностями битов одинаковой длины. За каждое значение базового типа «отвечает» 1 бит. Если множество содержит некоторый элемент, в соответствующем бите хранится 1, если не содержит – хранится 0.

Например:

Var X,Y:Set Of 1..10;

Внутреннее представление:

```
X:=[]          0000000000
X:=[2,3,9]     0110000010
Y:=[2..7]      0111111000
```

Операции над множествами сводятся к поразрядным логическим операциям над последовательностями битов. Например, объединение множеств выполняется путем поразрядного логического сложения битов. Объединение множеств $X \cup Y$

```
X: 0110000010
Y: 0111111000
-------------

X ∪ Y: 0111111010
```

Поскольку поразрядные операции входят в набор команд процессора эвм, они выполняются очень быстро.

Множества в Паскале – это наборы однотипных, логически связанных между собой объектов, которые рассматриваются как единое целое. Причем характер связи подразумевается программистом и никак не контролируется паскалем. Например, множество согласованных букв кириллицы; множество простых чисел от 1 до 100.

Каждый объект в множестве называется элементом множества. Все элементы множества должны принадлежать к одному из скалярных типов. Этот тип называется базовым типом. Он задается диапазоном или перечислением. Если множество не имеет элементов, оно называется пустым ([ ]).

Для изображения множеств используют квадратные скобки, в которые заключается перечень элементов.

Например:

```
Type
 Color=(White, Red, Black);
   Number=Set Of 1..31; {явное описание}
Var
   Col:Set Of Color; {неявное описание}
```

```
    N1,N2:Number;
    Etter:Set Of Char; {неявное описание}
```

Попытка присвоить недопустимый символ вызовет прерывание. Контроль диапазонов можно установить опцией компилятора ***{$r+}.***

Операции над множествами. В Паскале разрешены следующие операции над множествами: сравнения ***(=, <>, >=, <=), ∩ (And), ∪ (Or),*** разность множеств (-), включение в множество – ***In***. Рассмотрим их назначение.

"=": два множества равны тогда и только тогда, когда они имеют одинаковые элементы. Порядок следования роли не играет.

1) A:Set Of 1,2,3,4 B:Set Of 1,3,2,4 A=B - True

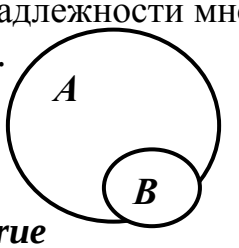
```
2) A:Set Of 'A','B','C'   B: Set Of 'A','B'   A=B - False
```

«< >»: множества не равны, если они отличаются по значению хотя бы одного элемента.

В случае 1) ***A <> B – False;***

В случае 2) ***A <> B – True.***

«>=» используется для определения принадлежности множеств. ***A >= B,*** если все элементы b содержатся в множестве a.



«<=» – аналогично.

A: =1, 2, 3, 4 B: =2, 3, 4 A>=B True B⊂A

B<=A True

«***In***» используется для проверки принадлежности какого-либо значения множеству. Пусть ***X='A', Y='Z', B*** – множество символов ***'A'..'N'.*** Тогда ***X In B – True, Y In B – False***.

Обычно операция ***In*** используется в условном операторе. Например, вместо:

If(A = 1) Or (A = 5) Or (A = 7) Or (A = 11) Or (A = 15) Then…,

Можно записать: ***If A In [1, 5, 7, 11, 15] Then***…,

т. е. «***In***» позволяет более эффективно производить сложные проверки условий. При этом множество не обязательно предварительно описывать в разделе описаний.

Если необходимо проверить, не принадлежит ли ***N*** множеству ***A***, можно записать: ***Not (N In A)*** (неверно: ***N Not In A***).

Объединение множеств ∪: «+». Объединением двух множеств является третье множество, содержащее элементы обоих множеств (выполняется путем поразрядного логического сложения).

Пусть ***A:=[1, 2, 3], B:=[4, 5], C:=A+B,*** тогда ***C=[1,2,3,4,5].***

Пересечение множеств ∩: «*».

Пусть `A:=[1,2,3], B:=[1,4,2,5],` тогда `A*B=[1,2].`

Пусть `A:= ['A'..'Z'], B:= ['B'..'Y'],` тогда

```
A*B=['B'..'Y'],
```

т.е. третье множество, которое содержит элементы, входящие одновременно в оба множества.

Разность множеств «-» (\- в математике) – третье множество, которое содержит элементы первого множества, не входящие во второе:

Пусть `A := [1,2,3,4], B:= [3,4,1],` тогда `A-B=[2].`

Пусть `A:= [X1,X2,X3,X4], B:= [X2,X3],` тогда **`A-B=[X1,X4].`**

Преимущества использования типа ***Set***: значительно упрощаются сложные условия в операторе ***If***, увеличивается степень наглядности, экономятся память, время компиляции и выполнения.

Отрицательные стороны: отсутствуют средства ввода-вывода элементов множеств.

П р и м е р ы

1. В считанном с клавиатуры тексте из строчных латинских букв удалить все гласные буквы, а согласные заменить на прописные. Признак конца ввода – точка. Например: ***Pointer - > PNTR***.

```
Program Sets;
    Type
       Let='A'..'Z';
    Var
       Gl:Set Of Let;
       Tx:String;
       I:Byte;
    Begin
       Readln(Tx);
       Gl:=['A','E','I','O','U','Y'];
       I:=1;
       Repeat
      While Tx[I] In Gl
           Do Delete(Tx,I,1);
          Tx[I]:=Upcase(Tx[I]);
          Inc(I)
       Until Tx[I]='.';
       Writeln(Tx)
    End.
```

2. Заполнить множество ***A*** путем ввода ***N*** значений

```
    Const
       N=20;
    Var
       A:Set Of 1..200;
       J,X:Byte;
    Begin
      A:=[];
       For J := 1 To N
        Do Begin
```

```
            Readln(X);
            A:=A+[X]
          End;
      For X:= 1 To 200
        Do If X In A Then Writeln(X);
   End;
```

3. Вывести в порядке возрастания элементы множества. Пусть ввели последовательность 2, 0, 4, -2, 1000, 7, 9, 100, 100, 40.

```
Const
   N=100;
   D:Set Of Byte=[1]; – типизированная константа
Type
   Num=1..N;
Var
    A,C:Set Of Num;
     B:Set Of 1..10;
       X:Integer;
       I,K:Byte;
Begin
   K:= 10; {заполнить множество A путем ввода K значений}
    A:=[];
    For I := 1 To K
     Do Begin
          Readln(X);
          A := A + [X]
        End;
    For X := 1 To N
     Do If X In A
         Then Write(X);
    Writeln;
End.
```

4. Что будет выведено на экран?

```
A := [1,2,4,6,7];  B := [1,2,4,7];
If A >= B Then Writeln('True');
A:= [1,4,7]; B := [2,4,7,15];
C:= A * B;
A:= [1,2,4,6,7];   B := [1,2,4,7];
C:= A+B;
A:= [1,2,4,6,7]; B := [1,2,4,7,10];
C:=A-B; {Соответствует логическому выражению A And Not B}
C:=B-A; {Соответствует логическому выражению B And Not A}
Readln(x);
If X In [0..3,10..15,20..25]    {проверить принадлежность
```

```
  Then Writeln('True');           одному из интервалов}
```

5. Проверить, является ли введенное с клавиатуры слово правильной записью идентификатора.

```
Type Letter= Set Of Char;
Var
   Ident : Letter;
   Wd : String[10];
 I:Byte;
   Fl:Boolean;
Begin
   Readln(Wd);
   Fl:= False;
   Ident:=['A'..'Z','A'..'Z','_'];
   If Not (Wd[1] In Ident)
    Then Fl := True  {Ошибка}
      Else Begin
              Ident := Ident + ['0'..'9'];
              For I := 2 To Length(Wd)
               Do If Not (Wd[I] In Ident)
                   Then Fl:=True {Ошибка}
          End;
   If Fl
    Then Writeln('не верный идентификатор!');
   Readln
End.
```

8 ПОДПРОГРАММЫ

Подпрограмма – обособленная сформированная в виде отдельной синтаксической конструкции и снабженная именем часть программы.

Использование подпрограмм позволяет, подробно описав в них некоторые операции, в остальной программе указывать только имена подпрограмм, чтобы выполнить эти операции.

Такие вызовы подпрограмм возможны неоднократно из разных участков программы, причем при вызове подпрограмме можно передать некоторую информацию (различную в различных вызовах, чтобы одна и та же подпрограмма могла выполнять решения для разных случаев).

Повышение сложности задач, решаемых с помощью эвм, приводит к увеличению размеров и сложности программ, следовательно, возникают дополнительные трудности при разработке и отладке. Увеличение продолжительности жизненного цикла программ приводит с течением времени к необходимости их модификации (с целью повышения их эффективности и удобства пользования ими). Для разрешения возникших при этом проблем в практике программирования выработан ряд приемов и методов структурного программирования (см. главу 12).

Под структурным программированием понимают такие методы разработки и записи программы, которые ориентированы на максимальные удобства для восприятия и понимания ее человеком.

При прочтении программы в ее фрагментах должна четко прослеживаться логика работы, т. е. не должно быть «скачков».

Структурное программирование – программирование «без ***Goto***», т. е. не используются операторы перехода без необходимости. В связи с этим отдельные фрагменты программы представляют собой некоторые логические (управляющие) структуры, определяюшие порядок выполнения содержащихся в них правил обработки данных. Любая программа получается построенной из стандартных логических структур, число типов которых невелико.

Основные логические структуры: ***следование, ветвление, повторение*** (каждая имеет один вход и один выход).

Простота и надежность программы существенно зависят от того, насколько удобно обрабатывать данные и правила их обработки и как они объединены в логические структуры.

Решение отдельного фрагмента сложной задачи может представлять собой самостоятельный программный блок – подпрограмму.

8.1 Процедуры и функции

Язык Паскаль называется процедурно-ориентированным за наличие подпрограмм как средства структурирования программы. Подпрограммы в Паскале реализованы посредством процедур и функций. Имея один и тот же смысл и

аналогичную структуру, процедуры и функции различаются назначением и способом использования.

Процедура – независимая именованная часть программы, которую можно вызвать по имени для выполнения определенных действий. Структура процедуры повторяет структуру программы. Процедура не может выступать как операнд в выражении. Например, ***Writeln*** – встроенная процедура Паскаля.

Функция – аналогична процедуре, но имеются два отличия:

1) функция передает в точку вызова скалярное значение (возвращает значение);

2) имя функции может входить в выражение как операнд.

Например, ***Arctan(X: Real): Real*** – передает в точку вызова ***Arctg(X).***

Вызов процедуры или функции – указание ее имени в тексте программы, приводящее к ее активизации.

Все подпрограммы паскаля делятся на две группы: встроенные (стандартные) и определенные пользователем.

Все стандартные средства расположены в специализированных библиотечных модулях, основные из которых следующие:

System – содержащиеся в нем подпрограммы обеспечивают работу всех остальных модулей системы. Подключается к программе автоматически, поэтому его имя не указывается в разделе ***Uses*** и любой программе всегда доступны его процедуры и функции;

Crt – средства управления монитором и клавиатурой;

Dos – средства Dos;

Printer – быстрый доступ к печатающему устройству;

Graph – пакет графических средств.

8.2 Процедуры и функции пользователя

Если в программе возникает необходимость частого обращения к некоторой группе операторов (выполняющих действия или вычисляющих значение выражения), то рационально выделить такую группу операторов в самостоятельный блок, к которому можно обращаться, указав его имя. Такие разработанные программистом самостоятельные программные блоки называются ***подпрограммами пользователя.***

Они являются основой ***модульного программирования***. Разбивая задачу на части, формируя логически обособленные модули как процедуры и функции, программист реализует основные принципы системного подхода и методов нисходящего программирования.

При вызове подпрограммы (процедуры или функции), определенной программистом, работа главной программы на некоторое время приостанавливается, и начинает выполняться вызванная подпрограмма. Она обрабатывает данные, переданные ей из главной программы. По завершении подпрограммы функция возвращает главной программе результат. Передача данных из глав-

ной программы в подпрограмму и возврат результата функции осуществляются с помощью параметров.

Параметром называется переменная, которой присваивается некоторое значение. Различают ***формальные параметры*** – определенные в заголовке подпрограммы и ***фактические параметры*** – выражения, задающие конкретные значения при обращении к подпрограмме.

При обращении к подпрограмме ее формальные параметры заменяются фактическими, переданными из главной программы.

Название «формальные» эти параметры получили в связи с тем, что они задают только имена для обозначения исходных данных и результатов работы подпрограммы. При вызове же подпрограммы на их место будут подставлены конкретные значения.

Подпрограмма определяется в разделе описания процедур и функций программы.

Формат описания процедуры:

Procedure <имя процедуры**>[(**список формальных параметров**)];**

Формат описания функции:

Function <имя функции>[(список формальных параметров)]:<тип рез-ата>;

Здесь имя процедуры или функции – идентификатор, список формальных параметров – последовательность имен формальных параметров, их типов и способов подстановки, отделенных друг от друга точкой с запятой.

Описание формальных параметров может отсутствовать.

Блок (тело) подпрограммы имеет ту же структуру, что и блок, являющийся телом программы, т.е. начинается зарезервированным словом ***Begin*** и заканчивается словом ***End.***

При написании обращений к процедуре и к функции необходимо обеспечить соответствие фактических и формальных параметров по количеству, типу и порядку следования. В теле функции должен присутствовать хотя бы один оператор присваивания с именем этой функции в левой части. Функция может возвращать в качестве результата значение скалярного, строкового или ссылочного типа.

П р и м е р. Описать подпрограмму, определяющую возможность построения треугольника по трем сторонам ***A, B*** и ***C*** и вычисляющую его площадь.

В а р и а н т 1 – процедура:

```
Procedure Triangle(A,B,C:Real; Var S:Real);
  Var
     P:Real;
  Begin
     If (A+B>C)And(A+C>B)And(B+C>A) {Если треугольник
                                      существует}
      Then Begin
             P:=(A+B+C)/2;
             P:=P*(P-A)*(P-B)*(P-C);
```

```
            S:=Sqrt(P);
          End
      Else S:=0
End;
```

Тогда значение площади треугольника может быть использовано в алгоритме, например, следующим образом:

```
{фрагмент программы}
Triangle(2,3,4,Q);
If Q<>0
 Then P:=5.7+2*Q;
```

В а р и а н т 2 – функция.

```
Function Triangle(A,B,C:Real):Real;
   Var
      P:Real;
   Begin
      If (A+B>C)And(A+C>B)And(B+C>A)
        Then Begin
                P:=(A+B+C)/2;
                P:=P*(P-A)*(P-B)*(P-C);
     If P>0
      Then Triangle:=Sqrt(P)

    Else Triangle := 0
   End;
```

Тогда два оператора в разделе действий программы примут вид:

```
Q:= Triangle(2,3,4 );
If Q<>0 Then P:=5.7+2*Q;
```

8.3 Параметры подпрограмм

В заголовке программы необходимо указать способ подстановки фактических параметров. Принято различать два способа подстановки параметров:

– подстановка значения (параметр-значение);

– подстановка переменной (параметр-переменная).

Параметры-значения передаются основной программой в подпрограмму через стек в виде их копий, поэтому фактический параметр подпрограммой измениться не может.

Параметры, которые называют параметрами-переменными, указываются заданием зарезервированного слова ***Var*** перед их идентификаторами в списке формальных параметров. При передаче параметров-переменных в подпрограмму фактически через стек передаются их адреса в порядке, объявленном в заголовке подпрограммы. Следовательно, подпрограмма имеет доступ к этим параметрам и может их изменять.

Входные параметры подпрограммы могут быть как параметрами-значениями, так и параметрами-переменными. Выходные (модифицируемые) – только параметрами-переменными.

Фактическими параметрами, соответствующими параметрам-значениям, могут быть имена переменных, константы, выражения. Фактическими параметрами, соответствующими параметрам-переменным – только имена переменных.

П р и м е р. Отпечатать таблицу значений суммы

$\sum_{i=1}^{m} \frac{1}{i}$ для m = 1, 2, …, 1024.

Опишем процедуру вычисления суммы ***S***.

```
Program Summa;
   Const
      N=1024;
   Var
      X:Real;
      M:Integer;
   Procedure Sum(N: Integer; Var S:Real);
      Var
         I:Integer;
      Begin
         S:=0;
         For I:=1 To N
          Do S:= S+1/I
      End;
   Begin
      For M:=1 To N
       Do Begin
             Sum(M,X);
             Write(M,X)
          End
   End.
```

Здесь в списке параметров процедуры ***Sum N*** – параметр значение, ***S*** – параметр-переменная.

9 РЕКУРСИИ

Рекурсивные алгоритмы и рекурсивные определения

Программист обычно разрабатывает программу, сводя исходную задачу к более простым. Среди этих задач может оказаться и первоначальная, но в упрощенной форме.

Например, вычисление функции ***F(N)*** может потребовать вычисления ***F(N-1)*** и еще каких-то операций. Иными словами, частью алгоритма вычисления функции будет вычисление этой же функции.

Алгоритм, который является своей собственной частью, называется ***рекурсивным***. Часто в основе такого алгоритма лежит рекурсивное определение какого-то понятия.

Например, о факториале числа n можно сказать, что

(определение 1) ***N ! = 1 *2 * … (N - 1) * N*** или 1, если ***N = 0;***

(определение 2) ***N ! =(N - 1) ! * N,*** или 1, если ***N*** = 0.

Второе определение рекурсивное.

Любое рекурсивное определение состоит из 2-х частей. Одна часть определяет понятие через него же, другая часть – через иные понятия.

Записать рекурсивный алгоритм на Паскале можно с помощью рекурсивной процедуры (функции).

Рекурсивные процедуры и функции

Рекурсия – это способ организации вычислительного процесса, при котором процедура или функция в ходе выполнения составляющих ее операторов обращается сама к себе (прямо или косвенно, через другие процедуры).

Рекурсивная процедура осуществляет многократный переход от некоторого текущего уровня организации алгоритма к более низкому последовательно до тех пор, пока не будет получено тривиальное решение поставленной задачи.

Рассмотрим программу с рекурсивным вызовом на примере факториала ***(N! = 1* 2 * 3 * … * N).***

```
Proqram Rec1;
   Var
      F:Longint ;
   Function Fact(F:Inteqer):Longint;  {описание функции,
            F–формальный параметр, значение типа Inteqer,
             результат функции – типа Longint}
      Begin
         If F=0
          Then Fact:=1
           Else Fact:=F*Fact(F-1)
      End;
   Begin
      Write(введите число F>0') ;
      Readln(F);
```

```
    If F>0
     Then Writeln('Для числа ',N,' значение
                        факториала= ',Fact(F))
       Else Writeln(' число неверное !') ;
End.
```

Использование рекурсивной формы организации алгоритма дает более компактный текст программы, но выполняется медленнее и может вызвать переполнение стека. ***Стек*** – специальным образом организованная область памяти, в которой размещаются при каждом входе в подпрограмму ее локальные переменные.

Вызов процедурой самой себя (рекурсивный вызов) ничем не отличается от вызова другой процедуры.

Что происходит, если одна процедура (или программа) вызывает другую? В общих чертах происходит следующее:

1) в памяти размещаются параметры, передаваемые процедуре (но не параметры-переменные);

2) в другом месте памяти сохраняются значения внутренних переменных вызывающей процедуры;

3) запоминается адрес возврата в вызывающую процедуру (или программу);

4) управление передается вызванной процедуре.

Если процедуру (функцию) вызвать повторно из другой процедуры или из нее самой, то будет выполняться тот же алгоритм, но работать он будет с другими значениями параметров и внутренних переменных. Это и дает возможность рекурсии.

П р и м е р. Пусть рекурсивная функция ***Step(A:Real;N:Inteder):Real;*** возводит число ***A*** в степень ***N: (A^N).*** Вычислить ***:*** $Z = X^k + Y^m$.

```
Program Rec_2;
   Var
      A,Y,Z:Real;
      K,M:Integer;
   Function Step(A:Real;N:Integer):Real;
      Begin
         If N=0
          Then Step:=1
           Else Step:=A*Step(A,N-1)
      End;
   Begin
      Readln(X,K);
      Readln(Y,M);
      Z:=Step(X,K)+Step(Y,M);
      Writeln(Z:0:3)
   End.
```

Возникает вопрос: «можно ли использовать рекурсивные процедуры с бесконечным «самовызовом»? нет, так как не существует бесконечной памяти!

П р и м е р. *{бесконечная рекурсия}*

```
Program Smile;
   Procedure Popanddog;
      Beqin
         Writeln('у попа была собака, он ее любил ');
          Writeln(' она съела кусок мяса, он ее убил ');
          Writeln(' похоронил и надпись написал: ');
          Popeanddog;
       End;
   Beqin
       Popanddog
   End.
```

Внесем небольшие изменения в программу.

{правильная рекурсия}

```
Program Smile;
   Procedure  Popanddog(K : Inteqer);
      Beqin
         Writeln(' у попа была собака, он ее любил ');
         Writeln(' она съела кусок мяса, он ее убил ');
         Writeln(' похоронил и надпись написал: ');
        If K>1
         Then Popeanddog(K-1);
      End;
   Beqin
      Popanddog(5)
   End.
```

Следовательно, условие, по которому выполняется вызов процедуры, должно на некотором уровне рекурсии стать ложным.

Пока условие истинно, рекурсивный спуск продолжается. Когда условие становится ложным, спуск заканчивается и начинается поочередный рекурсивный возврат из всех вызванных на данный момент копий рекурсивной процедуры.

Виды рекурсивных процедур

В общем случае рекурсивная процедура recur включает в себя некоторое множество операторов op и один или несколько рекурсивных вызовов recur.

1. Действия выполняются на рекурсивном спуске (до рекурсивного вызова):

```
   Procedure Recur;
       Begin Op;
          If условие
           Then Recur [Else Recur];
```

```
End;
```

2. Действия выполняются на рекурсивном возврате (после рекурсивного вызова):

```
Procedure  Recur;
   Begin
      If условие
       Then Recur [Else Recur ];
      Op;
   End;
```

3. Действия выполняются как на рекурсивном спуске, так и на рекурсивном возврате:

а)
```
Procedure Recur ;
   Begin
      Op1;
      If Условие
       Then Recur;
      Op2;
   End;
```

б)
```
Procedure Recur ;
   Begin
      If Условие
       Then Begin
              Op1;
              Recur;
              Op2
            End;
   End;
```

Все виды практически используются. Причем есть классы задач, при решении которых программисту требуется сознательно управлять ходом рекурсивных процедур и функций.

П р и м е р ы рекурсивных подпрограмм

1. Описать рекурсивную функцию ***Digit*** без параметров, которая подсчитывает количество цифр в тексте, заданном во входном файле (за текстом следует точка).

```
Program Rec_4;
   Var
      K:Integer;
    Function Digits:Integer;
      Var C:Char;
          D:Integer;
      Begin
         Read(C);
         D:=0;
         If C<>'•'
          Then If (C<='9')And(C>='0')
                Then D:=1+Digits
                 Else D:=Digits;
         Digits:=D;
      End;
   Begin
      Write('введите текст, последний символ – точка ');
```

```
    K:=Digits;
    Writeln(K);
End.
```

2. Напечатать в обратном порядке заданный во входном файле текст (за текстом следует точка).

```
Program Rec_5;
    Procedure Perevert;
       Var
          C:Char;
       Begin
          Read(C);
          If C<>'•'
           Then Perevert;
          Write(C);
       End;
    Begin
       Write(' введите текст');
       Perevert;
   End.
```

Библиографический список

1 **Гусева, А.И.** Учимся программировать: Pascal 7.0. Задачи и методы их решения / А.И. Гусева. – М. : Диалог-МИФИ, 2005. – 256 с.

2 **Зеленяк, О.П.** Практикум программирования на Turbo Pascal. Задачи, алгоритмы и решения / О.П. Зеленяк. – СПб. : ДиаСофтЮП : ДМК Пресс, 2007. – 320 с.

3 **Климова, Л.М.** Pascal 7.0. Практическое программирование. Решение типовых задач / Л.М. Климова. – М. : КУДИЦ-образ, 2003. – 528 с.

4 **Культин, Н.Б.** Программирование в Turbo Pascal 7.0 и Delphi / Н.Б. Культин. – СПб. : БХВ-Петербург, 2007. – 400 с.

5 **Марченко, А.И.** Программирование в среде Turbo Pascal 7.0. Базовый курс / А.И. Марченко, Л.А. Марченко. – М. : Век+, 2003. – 464 с.

6 **Немнюгин, С.А.** Изучаем Turbo Pascal / Л.В. Перколаб, С.А. Немнюгин. – СПб. : Питер, 2007. – 320 с.

7 **Фаронов, В.В**. Turbo Pascal 7.0. Практика программирования. Учебное пособие / В.В. Фаронов. – М.: КноРус, 2012. – 414 с.

8 **Тарануха, Н.А.** Обучение программированию: язык Pascal. Учебное пособие / Н.А. Тарануха, Л.С. Гринкруг, А.Д. Бурменский, С.В. Ильина. – М.: Солон-Пресс, 2009. – 384 с.

ПРИЛОЖЕНИЯ

Приложение 1

ОБЩАЯ СТРУКТУРА ПРОГРАММЫ

{-----*ЗАГОЛОВОК ПРОГРАММЫ*------}

PROGRAM *имя программы;*

{-------БЛОК ОПИСАНИЙ ----------}

uses .....
const .....
type .....
var .....
procedure .....
function .....

{------БЛОК ИСПОЛНЯЕМЫХ ОПЕРАТОРОВ ----------}

BEGIN
.....
.....
END.

Приложение 2

ОСНОВНЫЕ ТИПЫ ДАННЫХ

Таблица П2.1

Целые числа

Описатель типа	Длина (байт)	Минимальное число	Максимальное число
Integer	2 (знак)	-32768	+32767
Shortint	1 (знак)	-128	+127
Longint	4 (знак)	-2147483648	+2147483647
Byte	1 (б/зн.)	0	255
Word	2 (б/зн.)	0	65535

Таблица П2.2

Вещественные числа

Описатель типа	Длина (байт)	Число значащих цифр	Директива компилятора
Real	6	11	Не требуется
Single	4	7	{$N+}
Double	8	15	{$N+}
Extended	10	19	{$N+}
Comp	8	19 (целое число,64-bit)	{$N+}

Таблица П2.3

Литеры (символьные величины)

Описатель типа	Длина (байт)	Количество значений	Допустимые значения
Char	1	256	Литера (символ)

Таблица П2.4

Логические (булевские) величины

Описатель типа	Длина (байт)	Количество значений	Допустимые значения
Boolean	1	2	**true, false**

Приложение 3

ВВОД И ВЫВОД ДАННЫХ

Процедура (функция)	Назначение	Пример вызова	Примечания
Read (*список ввода*)	Ввод данных с клавиатуры	read (*a, b, c*);	
Readln (*список ввода*)	Ввод данных, пропуск маркера конца строки	readln (*a, b, c*);	
Write (*список вывода*)	Вывод данных на экран	write (‘*n* = ’,*n*:10); write (‘*n* = ‘,*n:p:q*);	*p, q* – целого типа
Writeln (*список вывода*)	Вывод данных, вывод маркера конца строки	writeln (‘?’:*s*+2); writeln (‘’:spaces);	*s*, spaces – целого типа

Приложение 4

ФУНКЦИИ

Таблица П4.1

Математические функции

Функция	Назначение	Пример вызова	Результат
abs (*число*)	Модуль числа	abs(-3.5)	+3.5
arctan (*тангенс угла*)	Арктангенс числа	arctan(0)	0
cos (*угол*)	Косинус угла (рад)	cos(pi)	-1
exp (*число*)	Экспонента	exp(1)	2.718281828...
frac (*число*)	Дробная часть числа	frac(3.5)	0.5
int (*число*)	Целая часть числа	int(3.5)	3.0
ln (*число*)	Натуральный логарифм	ln(2.718281828)	~1.0
odd (*число*)	Проверка нечетности	odd(3)	True
pi	Число пи	pi	3.141592...
random (*число*)	«Случайное» число	random(10)	Число в [0;10]
sin (*угол*)	Синус угла (рад)	sin(pi)	0
sqr (*число*)	Квадрат числа	sqr(2.0)	4.0
sqrt (*число*)	Квадратный корень	sqrt(25.0)	5.0

Таблица П4.2

Функции преобразования данных

Функция	Назначение	Пример вызова	Результат
round(*число*)	Округлить число	n := round(3.5)	4
trunc(*число*)	Отсечь дробную часть	n := trunc(3.5)	3

Таблица П4.3

Операции над символами

Функция	Значение	Пример вызова	Результат
chr(*номер-символа-n*)	Символ номер *n* (#**n**)	chr(33)	‘!’
ord(*величина*)	Номер величины (код)	ord(‘!’)	33
succ(*величина*)	Следующее значение в последовательности	succ(‘*y*’)	‘*z*’
pred(*величина*)	Предыдущее значение в последовательности	pred(‘*y*’)	‘*x*’

Учебное издание

Ведерникова Ольга Геннадьевна
Голубенко Евгений Владимирович

ПРИКЛАДНОЕ ПРОГРАММИРОВАНИЕ

«Электронный университет» ФГБОУ ВО РГУПС.

Адрес университета:
344038, Ростов н/Д, пл. Ростовского Стрелкового Полка Народного Ополчения, 2.