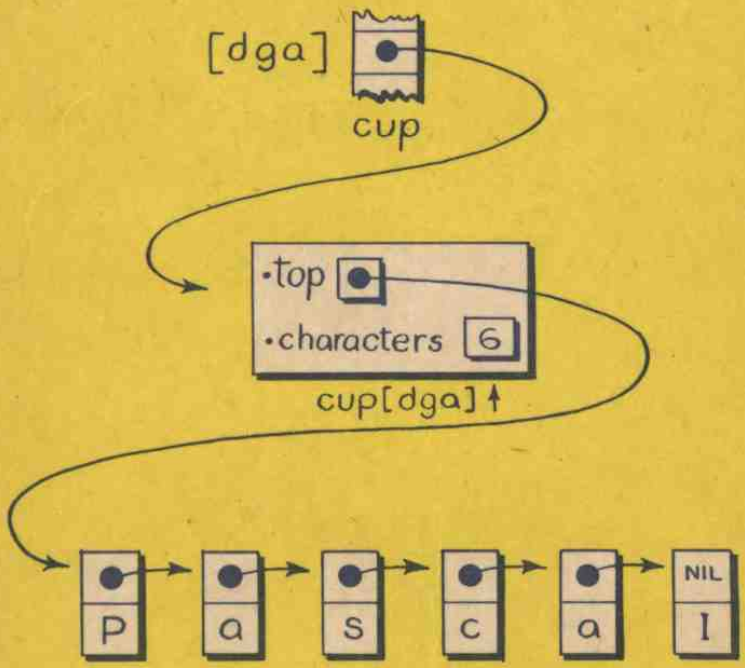
Дональд Алкок

Язык Паскаль в иллюстрациях

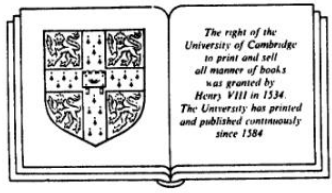


Издательство «Мир»

Язык Паскаль в иллюстрациях

ILLUSTRATING PASCAL

DONALD ALCOCK



CAMBRIDGE UNIVERSITY PRESS

CAMBRIDGE
NEW YORK NEW ROCHELLE
MELBOURNE SYDNEY

Дональд Алкок

Язык Паскаль в иллюстрациях

Перевод с английского
А.Ю. Медникова
под редакцией
А.Б. Ходулёва

Москва «Мир»
1991

ББК 32.973
А 50
УДК 681.3

Алкок Д.

А 50 Язык Паскаль в иллюстрациях: Пер. с англ. – М.: Мир, 1991. – 192 с., ил.

ISBN 5-03-001292-3

Книга английского специалиста, в которой в наглядной и оригинальной форме представлен стандарт языка Паскаль, имеющего реализации практически на всех современных ЭВМ. Изложение рассчитано на изучение языка.

Для программистов разной квалификации.

А $\frac{1702070000-104}{041(01)-91}$ 139-90

ББК 32.973

Редакция литературы по математическим наукам

Учебное издание

Доналд Алкок

ЯЗЫК ПАСКАЛЬ В ИЛЛЮСТРАЦИЯХ

Заведующий редакцией академик В. И. Арнольд. Зам. зав. редакцией А. С. Попов.
Ст. научный редактор М. В. Хатунцева. Художник Ю. Урманчиев.
Художник-график Ю. Н. Соустин. Художественный редактор В. И. Шаповалов.
Корректор Т. М. Подгорная.

Оригинал-макет подготовлен на персональном компьютере и отпечатан на лазерном принтере издательства "Мир".

Подписано к печати 19.02.91. Формат 70×100 1/16. Бумага офсетная.
Гарнитура литературная. Печать офсетная. Объем 6.00 бум. л.
Усл. печ. л. 15,6. Уч.-изд. л. 16,65. Усл. кр.-отт. 31,57.
Изд.№ 1/6498. Тираж 50 000 экз. Зак. 458. Цена 4р.60к.

Издательство «Мир» В/О «Совэкспорткнига» Госкомитета СССР по печати.
129820, Москва, 1-й Рижский пер., 2.

Можайский полиграфкомбинат В/О «Совэкспорткнига» Госкомитета СССР по печати.
143200, Можайск, ул. Мира, 93.

ISBN 5-03-001292-3 (русск.)
ISBN 0-521-33695-3 (англ.)

© Cambridge University Press 1987.
© перевод на русский язык, Медников А. Ю., 1991

ПРЕДИСЛОВИЕ РЕДАКТОРА ПЕРЕВОДА

Открыв эту книгу, не сразу понимаешь, что держишь в руках учебник по языку программирования Паскаль. На одной странице – какие-то падающие пирамидки из кубиков, на другой – переплетение стрелочек, на третьей – вроде бы программа, но по ней почему-то ползает множество жучков. Все это – выразительные средства, делающие чтение книги не только полезным, но и по возможности простым и интересным занятием. Упомянутые жучки, в частности, указывают ошибочные места в программе. Этот символ происходит из омонимии английского слова *bug*, означающего одновременно и жука и ошибку в программе.

Еще один немаловажный момент – расположение материала по страницам. Автор позаботился о том, чтобы каждый разворот представлял собой некоторый сравнительно обособленный фрагмент изложения. Читателю не придется бесконечно перелистывать страницы, сравнивая, скажем, начало коротенькой программы с ее продолжением на обороте. Благодаря широкому использованию наглядных образов и тщательно продуманной форме изложения автору удалось весьма компактно изложить стандартный Паскаль в полном объеме.

Необычный стиль книги создал существенные трудности при ее переводе. Первая проблема – переводить ли имена в программах. Большинство компиляторов с Паскаля не позволяют использовать в именах русские буквы, поэтому было принято решение сохранить в программах английские имена, с тем чтобы читатель мог взять любую программу из книги и выполнить ее на своем компьютере. В необходимых случаях в программы добавлены комментарии на русском языке. Вместе с тем, практически все компиляторы позволяют использовать русские буквы (если, конечно, они есть на клавиатуре компьютера) в *текстах*, предназначенных для печати или обработки. Соответственно, такие тексты, как правило, переводились, так что программы общаются с пользователем на русском языке.

Чтобы сохранить компоновку материала, перевод выполнялся «страница в страницу». Каждый разворот содержит ровно тот же материал, что и английский оригинал. Ради сохранения целостности разворотов примечания редактора перевода пришлось поместить *в конце каждой главы*. Они касаются в основном отличий версии Турбо Паскаль, наиболее распространенной на персональных компьютерах в СССР, от версии Acornsoft ISO Паскаль, на которую ориентировался автор.

Типографский набор не подходит для издания перевода по тем же причинам, что и для оригинала. Не обладая, однако, терпением, необходимым для подготовки манускрипта, мы остановили выбор на компьютерных издательских системах, от которых отказался автор (см. его предисловие). При подготовке на персональном компьютере оригинал-макета этой книги использовались компьютерные шрифты, разработанные в Институте прикладной математики им. М.В. Келдыша АН СССР.

Книгу ввиду ее краткости и «занимательности» можно рекомендовать начинающим программистам – тем, кто уже имел дело с компьютером, но еще не поднаторел в изучении многотомных руководств по программному обеспечению, а также и тем, кто только вступает в мир компьютеров. При изучении книги желательно использовать компьютер, оснащенный каким-либо (годится любой) компилятором с Паскаля, с тем чтобы выполнять упражнения не на бумаге, а по-настоящему.

Язык Паскаль – простой и стройный – весьма удачен как средство обучения программированию; именно для этой цели он и предназначался его создателем – Н. Виртом. (Впоследствии язык Паскаль нашел широкое применение и для создания серьезных производственных программ.) Следует, однако, иметь в виду, что автор почти не касается общих вопросов программирования, сосредоточиваясь на изложении собственно языка. Для углубленного изучения программирования следует обратиться к дополнительной литературе.

А. Б. Ходулёв

СОДЕРЖАНИЕ

Предисловие редактора перевода 5

Предисловие 8

1. Принципы 11
- Что такое программа 12
- Первое знакомство с Паскалем 14
- Ввод программы 15
- Компиляция 16
- Шаги выполнения 17
- Упражнения 18

2. Основные элементы 19
- Пунктуация 20
- Переменные 22
- Константы 22
- Стандартные типы 23
- Выражения 24
- Заем (пример) 25
- Условия 26
- Поля 26
- Геометрические фигуры (пример) 27
- Циклы 28
- Былая слава (пример) 29
- Синус (пример) 29
- Упражнения 30

3. Синтаксис 31
- Стиль написания 32
- Обозначения 33
- Элементы 34
- Компоненты 35
- Синтаксис выражения 36
- Синтаксис оператора 37
- Синтаксис программы 38
- Синтаксис типа 39
- Примечания редактора 40

4. Арифметика 41
- Операции 42
- Размер и точность 44
- Компараторы 45
- Арифметические функции 46
- Тригонометрические функции 47
- Функции преобразования 48
- Логические функции 49
- Функции над дискретными типами 50
- Примечания редактора 52

5. Управление 53
- Блок-схемы 54
- Оператор IF-THEN-ELSE 56
- Цикл FOR 57
- Цикл REPEAT 58
- Цикл WHILE 58
- Фильтр (пример) 59
- Оператор CASE 60
- Автоматный распознаватель (пример) 61
- Упражнения 62
- Примечания редактора 62

6. Функции и процедуры 63
- Определение функции 64
- Примеры функций 66
- Рекурсия 67
- Процедуры 68
- Случайные числа 70
- Снова заем (пример) 72
- Имена функций как параметры 73
- Ссылки вперед 74
- Локальные переменные 75
- Побочные эффекты 76
- Правила видимости 77
- Упражнения 78
- Примечания редактора 78

7. Типы и множества 79
- Стандартные типы 80
- Определение типов 81
- Перечисляемые типы 82
- Интервальные типы 83
- Тип множеств и переменные типа множество 84
- Конструкторы множеств и операции над множествами 85
- Фильтр2 (пример) 86
- МУ-У-У (пример) 87
- Упражнения 88
- Примечания редактора 88

8. Массивы и строки 89
- Знакомство с массивами 90
- Синтаксис объявления массивов 91
- Площадь многоугольника (пример) 92
- Провода (пример) 93
- Сортировка методом пузырька (пример) 94

БЫСТРАЯ СОРТИРОВКА (ПРИМЕР) 96
УПАКОВКА 98
ЗНАКОМСТВО СО СТРОКАМИ 99
ФОКУС (ПРИМЕР) 100
СИСТЕМЫ СЧИСЛЕНИЯ (ПРИМЕР) 102
УМНОЖЕНИЕ МАТРИЦ (ПРИМЕР) 105
НАСТРАИВАЕМЫЕ ПАРАМЕТРЫ-МАССИВЫ 106
УПРАЖНЕНИЯ 108
ПРИМЕЧАНИЯ РЕДАКТОРА 108

9. ЗАПИСИ 109
ЗНАКОМСТВО С ЗАПИСЯМИ 110
СИНТАКСИС ЗАПИСЕЙ 111
ПЕРСОНАЛЬНЫЕ ЗАПИСИ (ПРИМЕР) 112
ОПЕРАТОР WITH 116
ЧТО ТАКОЕ ВАРИАНТЫ 118
УПРАЖНЕНИЯ 120
ПРИМЕЧАНИЯ РЕДАКТОРА 120

10. ФАЙЛЫ 121
ЧТО ТАКОЕ ФАЙЛЫ 122
ОТКРЫТИЕ ФАЙЛОВ 124
ТЕКСТОВЫЕ ФАЙЛЫ 125
ПРОЦЕДУРЫ *WRITE* И *WRITELN* ДЛЯ ТЕКСТОВЫХ ФАЙЛОВ 126
ПРОЦЕДУРА *PAGE* ДЛЯ ТЕКСТОВЫХ ФАЙЛОВ 126
ПРОЦЕДУРЫ *READ* И *READLN* ДЛЯ ТЕКСТОВЫХ ФАЙЛОВ 127
БЕЗОПАСНОЕ ЧТЕНИЕ 128
ПРОЦЕДУРА *GRAB* ДЛЯ БЕЗОПАСНОГО ЧТЕНИЯ 130
ДВОИЧНЫЕ ФАЙЛЫ, ПРОЦЕДУРЫ *PUT* И *GET* 134
СЖАТИЕ (ПРИМЕР) 136
СВОЙСТВА ФАЙЛОВ, СВОДКА 137
УПРАЖНЕНИЯ 138
ПРИМЕЧАНИЯ РЕДАКТОРА 138

11. ИНТЕРАКТИВНЫЙ ВВОД 139
ДИАЛОГ 140
ПРОБЛЕМА ЗАГЛЯДЫВАНИЯ ВПЕРЕД 141
ПРОБЛЕМА БУФЕРА 142
ПРОБЛЕМА КОНЦА ФАЙЛА (*EOF*) 143
ПРИМЕЧАНИЯ РЕДАКТОРА 144

12. ДИНАМИЧЕСКАЯ ПАМЯТЬ 145
ДИНАМИЧЕСКАЯ ПАМЯТЬ 146
ПРОЦЕДУРЫ *NEW* И *DISPOSE* 148
СТЕКИ И ОЧЕРЕДИ 150
ОБРАТНАЯ ПОЛЬСКАЯ НОТАЦИЯ 152
RAJPCKAYA (ПРИМЕР) 154
ПРОСТЫЕ ЦЕПИ 155
КРАТЧАЙШИЙ ПУТЬ (ПРИМЕР) 156
КОЛЬЦА 160
АСТРА (ПРИМЕР) 162
ДВОИЧНЫЕ ДЕРЕВЬЯ 164
ОБЕЗЬЯНЬЯ СОРТИРОВКА (ПРИМЕР) 166
УПРАЖНЕНИЯ 168
ПРИМЕЧАНИЯ РЕДАКТОРА 168

13. ДИНАМИЧЕСКИЕ СТРОКИ 169
ПРОГРАММЫ ОБРАБОТКИ СТРОК 170
READSTRING 172
WRITESTRING 172
MIDDLE 173
CONCAT 174
COMPARE 175
INSTR 176
PEEK 176
POKE 177
ОБРАТНЫЙ СЛЕНГ (ПРИМЕР) 178
ТЕХНИКА ХЕШИРОВАНИЯ (ПРИМЕР) 180
HASHER (ПРИМЕР) 182

ЛИТЕРАТУРА 184

КРАТКАЯ СПРАВКА 185
СТАНДАРТНЫЕ ПРОЦЕДУРЫ 185
СТАНДАРТНЫЕ ФУНКЦИИ 186
СИНТАКСИС 187

УКАЗАТЕЛЬ 190

ПРЕДИСЛОВИЕ

Язык программирования Паскаль был разработан профессором Цюрихского Федерального технологического института (ETH) Никлаусом Виртом. Предварительное описание появилось в 1968 году. С тех пор Паскаль становился все более и более популярен, причем не только как язык для обучения принципам программирования, но и как средство создания достаточно сложного программного обеспечения.

В этой книге дано полное описание версии языка, определенной стандартом BS6192: *Спецификация языка программирования Паскаль.* Этот стандарт разрабатывался так, чтобы обеспечить совместимость со стандартом ISO 7185 Международного Института Стандартов. Чтобы не уходить от действительности, я выполнил программы, приведенные в этой книге, в трех системах:

- ISO Паскаль фирмы *Acornsoft*
- Pro Паскаль фирмы *Prospero*
- Турбо Паскаль фирмы *Borland International*

Мой стиль изложения – картинки. Гораздо больше можно сказать рисунком:



нежели сотней слов о хеш-адресах, указателях, записях и связанных списках. Но я думал также и о выборе правильных слов, имея в виду простоту и краткость. Материал скомпонован так, чтобы каждый разворот из двух страниц содержал законченный фрагмент. В результате вам не нужно переворачивать страницы, встретив, например, ссылку из текста на диаграммы. При такой компоновке, а диаграммы здесь являются столь же важным элементом, как и сам текст, слова необходимо размещать строго на своем месте. Это одна из причин использования рукописного текста – в таких условиях просто легче пользоваться пером, нежели типографским набором. (Современный автор, использующий текстовые процессоры и автоматизированный набор, – все это заметно упрощает процесс подготовки и компоновки текстового материала, – поддается соблазну и думает: «Как бы мне изложить эту идею без рисунка?», тогда как на самом деле ему следовало бы задаться вопросом: «Как мне заменить все эти скучные слова одним рисунком?».)

Содержание книги излагается в стиле руководства по языку программирования. В главе 1 приводится пример для начинающих – демонстрируется сама идея хранимой программы. В гл. 2 дается краткий обзор элементов программирования (переменные, стандартные типы, выражения, условия и циклы). У тех, кто писал программы на других языках, все это не должно вызвать затруднений. Материала этих двух глав вполне достаточно для того, чтобы все остальные возможности Паскаля излагать при помощи законченных программ.

Глава 3 – короткая, но – важная. В ней вводится система обозначений, которая используется на протяжении всей книги для описания синтаксиса операторов и элементов Паскаля. Эта нотация является смесью формы Бэкуса–Наура и диаграмм, имеющих вид железнодорожных путей. Я полагаю, что такая структура хорошо воспринимается зрительно, не теряя в строгости.

Начиная с гл. 4, каждая возможность Паскаля вводится в контексте работающей программы. Более длинные программы служат не только для демонстрации средств Паскаля, но также и для иллюстрации фундаментальных понятий программирования, к которым, например, относятся быстрая сортировка, рекурсия, кольца, двоичные деревья и хеширование.

Дольше всего мне пришлось ломать голову над проблемой интерактивного ввода. Паскаль был разработан во времена перфокарт и магнитных лент. Логика операторов WRITE и READ не позволяла программам запрашивать ввод данных с клавиатуры. Теперь такая возможность организации диалога доступна – читатели этой книги, вероятно, будут выполнять примеры из книги в интерактивном режиме. К сожалению, в различных версиях языка Паскаль эта проблема решена по-разному. Поэтому в моих примерах используется наипростейшая организация ввода и указываются те места, где читателю, работающему с интерактивной системой, следует поставить запросы, облегчающие использование программы. Вопросам, которые могут возникнуть при попытке интерактивного использования Паскаля, я посвятил короткую гл. 11.

Если на первых порах пунктуация программ, написанных на Паскале покажется Вам излишней,

и

вы

обнаружите,

что

Вас

уводит

вправо,

то не отчаивайтесь, скоро вы к этому привыкнете. Когда вы дойдете до *записей*, вновь засияет солнце. Добравшись до *указателей* (и научившись собирать *цепочки, стеки, кольца и деревья*), Вы почувствуете свою преданность Паскалю. Лекарства от любви к Паскалю еще не найдено.

Доналд Алкок

Ноябрь 1986

Благодарности

Эта книга должна была писаться вместе с соавторами, сначала с Колин Дэй, затем с Ричардом Кайтом. Однако все попытки к совместной работе разбились о сугубо индивидуальную природу рукописного труда. Так или иначе, настоящая книга, вероятно, удалась. Моя сердечная благодарность им обоим.

Благодарю также Пола Шеринга из компании *Euro Computer Systems Ltd.* за предоставленную мне возможность работать на компьютерах его фирмы и за оказанную помощь при выполнении программ в системах Pro Паскаль и Турбо Паскаль. Первоначально я разрабатывал программы в системе Acorn ISO Паскаль.

В заключение я выражаю благодарность моему старшему сыну Эндрю за разработку программы, которую я использовал при составлении и сортировке предметного указателя к этой книге.

1

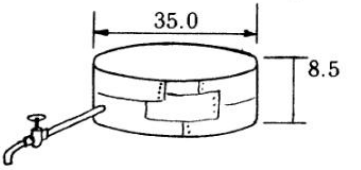
ПРИНЦИПЫ

ЧТО ТАКОЕ ПРОГРАММА
ПЕРВОЕ ЗНАКОМСТВО С ПАСКАЛЕМ
ВВОД ПРОГРАММЫ
КОМПИЛЯЦИЯ
ШАГИ ВЫПОЛНЕНИЯ

ЧТО ТАКОЕ ПРОГРАММА

ЕСЛИ ВЫ ХОТЬ НЕМНОГО ЗНАКОМЫ С ПРОГРАММИРОВАНИЕМ, ПЕРЕВЕРНИТЕ СТРАНИЦУ

Маляру поручили покрасить крышу и стенки этого бака для бензина. Каким образом он сможет определить, сколько ему потребуется банок с краской, если у него нет под рукой компьютера?





Производитель красок утверждает, что емкости одной банки достаточно для покраски поверхности площадью 236.0

Вспомним, что площадь круга вычисляется по формуле πr^2 (где r – радиус круга) или $\pi d^2/4$ (где d – диаметр круга). Вспомним также, что длина окружности равна πd. Таким образом, маляр может вычислить

площадь крыши (top) = 3.14 * 35.0^2 : 4 = 961.63

площадь стенки (wall) = 3.14 * 35.0 * 8.5 = 934.15

Площадь поверхности , которую надо покрасить, равна сумме двух площадей, подсчитаных выше. Разделив величину площади на емкость одной банки, маляр получит необходимое ему количество банок:

банки (pots) = (961.63 + 934.15) : 236 = 8.03

Число с дробной частью называется вещественным (real). Разумеется, нельзя купить часть банки с краской. Поэтому количество банок следует округлить до ближайшего большего целого числа (integer). Для этого *отбросим дробную часть* и прибавим 1

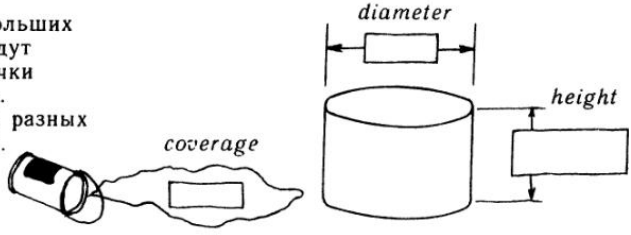
полные банки (fullpots) = 8.03 + 1 = 9



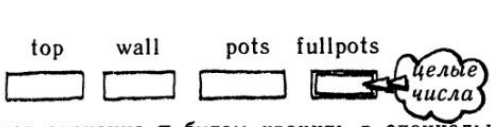
Если при других размерах количество банок получилось бы равным 8.00 вместо 8.03, то в ответе все равно было бы 9. Это не вполне точно, но маляр, вероятно, будет чувствовать себя уверенней в такой ситуации, чем если бы у него было всего 8 банок.

Теперь представим себе, что маляр захотел решить эту задачу в общем виде, т. е. сделать так, чтобы при возникновении подобной задачи еще раз потребовалось бы лишь ввести несколько чисел и «нажать кнопку», чтобы получить ответ.

Заведем несколько небольших коробочек, в которых будут храниться числа. Коробочки будут снабжены именами. *Содержимое* коробочек в разных задачах будет различным.



Надо не забыть про коробочки, где будут храниться промежуточные результаты:





Приближенное значение π будем хранить в специальной коробочке. Это значение одно и то же вне зависимости от размеров бака или емкости банки ~ поэтому коробочка на замке.

Списку инструкций ~ называемому программой ~ можно дать имя. Выглядит он так:

PROGRAM painter (введем (INPUT) данные, выведем (OUTPUT) результат);

константы (CONSTANTS)

запертые коробочки

pi = 3.14 { *можно использовать, но не изменять* };

переменные (VARIABLES)

открытые коробочки

diameter, height, coverage,
top, wall, pots, fullpots;
все коробочки содержат вещественные числа (REAL), *за исключением* fullpots, *предназначенной для целых* (INTEGER).

объявим имена всех используемых коробочек и укажем типы хранящихся чисел

Начало инструкций

- прочесть исходные данные: записать в коробочки diameter, height, coverage соответствующие числа;
- *в коробочку* top *поместить результат умножения содержимого коробочки* pi *на квадрат содержимого коробочки* diameter, *деленный на 4*;
- *в коробочку* wall *поместить результат перемножения содержимого коробочек* pi, diameter, height;
- *в коробочку* pots *поместить сумму чисел из коробочек* top *и* wall, *поделенную на число из коробочки* coverage;
- *в коробочку* fullpots *поместить число с отброшенной дробной частью из коробочки* pots, *сложенное с 1;*
- *Напечатать сообщение маляру* ('ВАМ НЕОБХОДИМО ', *напечатать число из коробочки* fullpots,' БАНОК');

Конец инструкций.

Пусть строка исходных данных состоит из чисел

35.0	8.5	236.0

Если вы поработаете за компьютер и сами проделаете все инструкции программы, то результатом будет записка для маляра

ВАМ НЕОБХОДИМО 9 БАНОК

Другие исходные данные, естественно, приведут к *другому* результату. В этом заключается суть программы ~ программа – это обобщенные вычисления.

ПЕРВОЕ ЗНАКОМСТВО С ПАСКАЛЕМ

ПЕРЕВОД ПРОГРАММЫ, ПРИВЕДЕННОЙ НА ПРЕДЫДУЩЕЙ СТРАНИЦЕ

Инструкции на предыдущей странице слишком многословны, и поэтому их нельзя использовать как программу для компьютера. Однако эти инструкции можно без потери информации переписать на Паскале.

Часто повторяющееся словосочетание *«содержимое коробочки»* можно безболезненно опустить. Например, третья инструкция программы будет теперь выглядеть так

```
☆      в коробочку wall поместить результат вычисления:
       pi умножить на diameter умножить на height;
```

Вместо словосочетания *«в такую-то коробочку поместить результат вычисления:»* будем писать просто название коробочки с символом :=

```
☆      wall :=
```

где := читается как «принимает значение».

Теперь заменим слова *«умножить»*, *«сложить»*, *«вычесть»*, *«поделить»* на знаки *, +, –, / . Третья инструкция станет совсем короткой

```
☆      wall := pi * diameter * height
```

и будет читаться как «wall *принимает значение* pi *умножить на* diameter *умножить на* height».

В Паскале существуют и другие сокращения, речь о которых пойдет позже, как и о некоторых важных правилах пунктуации. Однако уже сейчас мы можем записать нашу программу на Паскале.

```
PROGRAM painter (INPUT, OUTPUT);
  CONST pi = 3.14;                                   (объявления)
  VAR   diameter, height, coverage,
        top, wall, pots: REAL; fullpots: INTEGER;
BEGIN                                                (разделители)
  READ (diameter, height, coverage);                 (инструкции)
  top := pi * SQR(diameter) / 4.0;
  wall := pi * diameter * height;                    SQR( ) и TRUNC( ) – функции из
  pots := (top + wall) / coverage;                   набора функций, заранее пре-
  fullpots := TRUNC(pots) + 1;                       дусмотренного в Паскале
  WRITE ('ВАМ НЕОБХОДИМО', fullpots, ' БАНОК')
END.
```

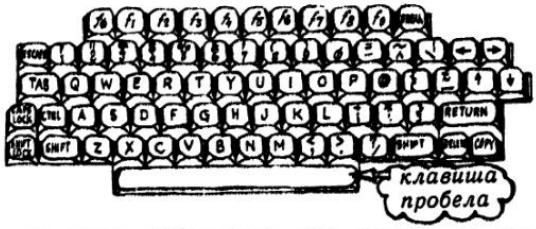
Объявления и инструкции этой программы в точности соответствуют объявлениям и инструкциям программы на обычном языке.

Смысл появления прописных и строчных букв скоро будет разъяснен.

ВВОД ПРОГРАММЫ

ПАСКАЛЬ СТАНДАРТИЗОВАН, КЛАВИАТУРЫ КОМПЬЮТЕРОВ РАЗЛИЧАЮТСЯ

Ниже нарисована клавиатура обычного домашнего компьютера. Клавиатуры других персональных компьютеров и терминалов систем с разделением времени похожи на эту.



В правой части клавиатуры обязательно есть кнопка, на которой написано enter, return, ввод или нарисовано ↵ . Это – кнопка перехода на новую строку. Любая клавиатура имеет клавиши от A до Z, цифры от 0 до 9, точку, запятую, двоеточие и точку с запятой, а также знаки арифметических операций +, –, *, /, которые используются в нашей программе.

Перед вводом программы необходимо загрузить редактор и это делается по-разному в разных системах. Если используется Acornsoft Паскаль на компьютере BBC модели B, то надо набрать EDIT и нажать return. В Pro Паскале используется встроенный редактор аналогичный Word Star, в Турбо Паскале надо нажать кнопку E.

Находясь в редакторе, безбоязненно вводите текст программы: у вас всегда есть возможность стереть или исправить неверно введенный символ. На большинстве компьютеров это делается кнопкой delete или del или backspace. При вводе программы, однако, необходимо внимательно следить за пунктуацией в программе, которая в языке Паскаль довольно неочевидна.

Возможности редакторов могут сильно отличаться. Редактор Турбо Паскаля основан на текстовом процессоре Word Star. Вначале любой редактор немного страшен, но с появлением опыта работы даже самый «страшный» редактор становится не так уж плох. Терпение и настойчивость.

Различием между прописными и строчными буквами можно пренебречь: вводите программу либо с нажатой, либо с отжатой кнопкой Caps Lock. Единственное место в программе, где размер букв может оказаться важен – это строка

```
write ('Вам необходимо',fullpots,' банок')
```

где слова внутри апострофов в точности воспроизводятся при выводе результата.

Вводя программу, имейте в виду, что компьютер *не выполняет* инструкций программы и вовсе может не знать, что вводится программа на Паскале; для компьютера это просто *файл*. Можно вводить даже «Гори, гори, моя звезда» на португальском – компьютер будет невозмутим.

КОМПИЛЯЦИЯ

И ПРОБЛЕМА ВВОДА ДАННЫХ С КЛАВИАТУРЫ

Программу на Паскале в отличие от программы на Бейсике, которая запускается командой RUN, надо предварительно *скомпилировать*. Компиляция означает перевод *исходной* программы с языка Паскаль в *объектную программу* – на язык компьютера. При запуске программы, вычисления производятся по программе в объектном коде, а не по исходной программе.

После компиляции имеются две версии программы: одна на Паскале, другая на языке компьютера (или близком к нему). Если посмотреть на объектную программу, то на экране будет просто тарабарщина.

Паскаль выполняется быстрее Бейсика, потому что объектная программа на языке близком к языку компьютера (или непосредственно в командах компьютера) выполняется весьма эффективно, в то время как инструкции программы на Бейсике интерпретируются в исходном виде. Платой за выигрыш в скорости выполнения скомпилированной программы служат неизбежные затраты времени на компиляцию и связанные с этим неудобства. Правда, в большинстве систем предусмотрена возможность сохранения объектных программ, а значит и повторного их выполнения без рекомпиляции. На следующей странице показан случай, когда копия скомпилированной программы сохраняется на диске.

Этапы от ввода программы до ее выполнения изображены напротив. Слева написаны *команды*, которые надо набрать на клавиатуре, чтобы выполнить очередной шаг. Эти команды зависят от системы, с которой вы работаете: выписанные команды вымышлены, однако они довольно типичны. MYSOURCE и MYOBJECT – имена исходной и объектной программ, придуманные программистом.

Последний шаг – выполнение программы. Здесь предполагается ввод данных (input) с клавиатуры и вывод результатов (output) на экран. Это довольно распространенная схема ввода-вывода данных, стандартная в Паскале, но, разумеется, не единственная. Язык был разработан еще тогда, когда файлы хранились на магнитной ленте, ввод осуществлялся с перфокарт, а вывод – на печатающее устройство. Современные компиляторы позволяют выводить сообщения Паскаля на экран и вводить данные с клавиатуры. Если вы располагаете подобным компилятором, у вас не будет особых затруднений при выполнении примеров из этой книги. Однако, если ваша программа задает вопросы после получения ответов, загляните в гл. 11, где проливается свет на возможные трудности и пути их преодоления. Вероятно, ваш компьютер не работает в интерактивном режиме ~ в этом случае вы все же сможете выполнить примеры, но исходные данные придется хранить на диске, а не вводить с клавиатуры. Такая схема *пакетной обработки* показана на рисунке.

GO – ВЫПОЛНЕНИЕ



ШАГИ ВЫПОЛНЕНИЯ

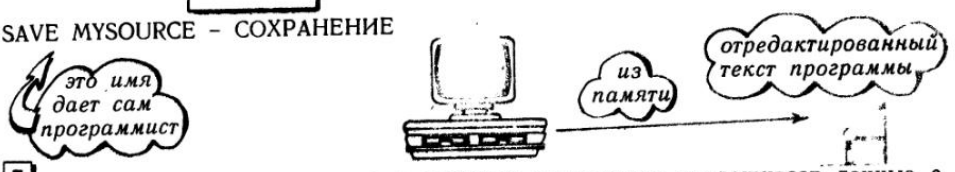
КОМАНДЫ НЕ СТАНДАРТИЗОВАНЫ

Команды операционной системы на разных компьютерах могут отличаться, однако процесс, который изображен ниже, весьма типичен:

EDIT – РЕДАКТИРОВАНИЕ

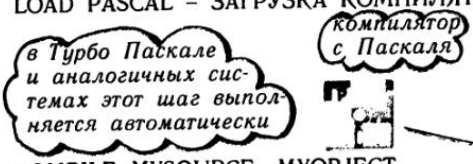


SAVE MYSOURCE – СОХРАНЕНИЕ

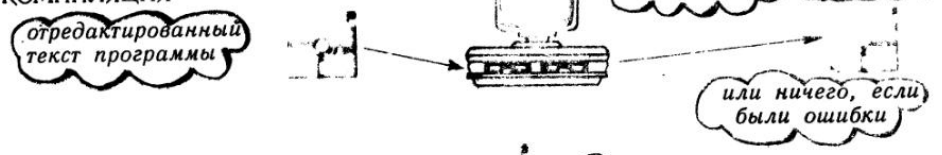


Предполагается, что программа в процессе выполнения запрашивает данные с клавиатуры. Если же исходные данные будут вводится из файла на диске, то их необходимо ввести, отредактировать и записать на диск так же, как и саму программу.

LOAD PASCAL – ЗАГРУЗКА КОМПИЛЯТОРА



COMPILE MYSOURCE, MYOBJECT – КОМПИЛЯЦИЯ



LOAD MYOBJECT – ЗАГРУЗКА



GO – ВЫПОЛНЕНИЕ



УПРАЖНЕНИЯ

1. Выполните на компьютере программу о покраске. Это упражнение заставит вас воспользоваться редактором и компилятором. Знакомство с неизвестной системой всегда непростое дело: возможно это упражнение – самое трудное во всей книге.

2

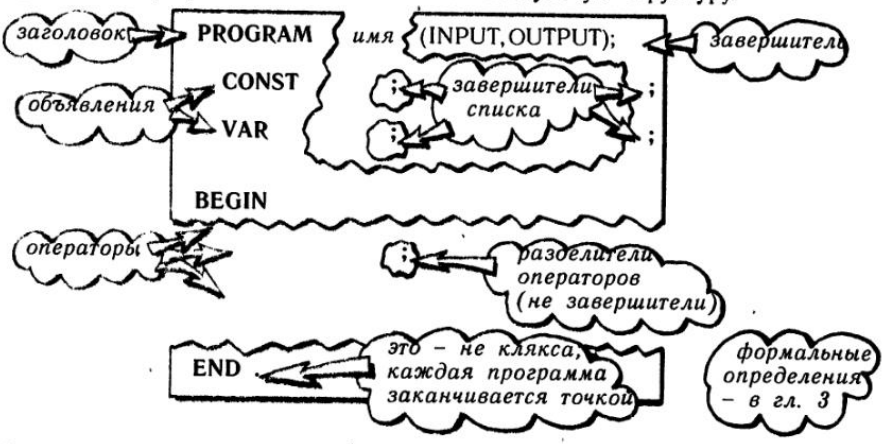
ОСНОВНЫЕ ЭЛЕМЕНТЫ

ПУНКТУАЦИЯ
ПЕРЕМЕННЫЕ
КОНСТАНТЫ
СТАНДАРТНЫЕ ТИПЫ
ВЫРАЖЕНИЯ
ЗАЕМ (ПРИМЕР)
УСЛОВИЯ
ПОЛЯ
ГЕОМЕТРИЧЕСКИЕ ФИГУРЫ (ПРИМЕР)
ЦИКЛЫ
БЫЛАЯ СЛАВА (ПРИМЕР)
СИНУС (ПРИМЕР)

ПУНКТУАЦИЯ

СИНТАКСИС ОПРЕДЕЛЕН В СЛЕДУЮЩЕЙ ГЛАВЕ, СЕЙЧАС – ОБЩИЕ ПОЛОЖЕНИЯ

Типичная программа на Паскале имеет следующую структуру:

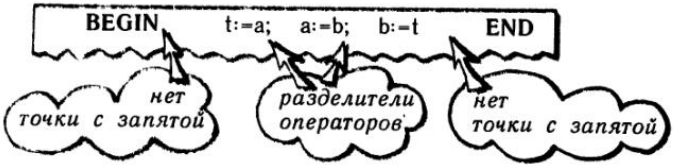


Заголовок *завершается* точкой с запятой.

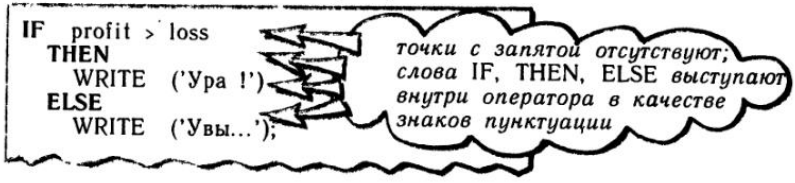
В любом объявлении каждый список *завершается* точкой с запятой.

Операторы *отделены* один от другого точкой с запятой.

Слова BEGIN и END не являются операторами – они служат знаками пунктуации. Слово BEGIN выступает в качестве левой, а END – правой скобки. Так как они сами – знаки пунктуации, то точка с запятой после BEGIN и перед END не обязательна. В программах на Паскале слова BEGIN и END используются преимущественно для образования *составных* операторов. Составной оператор может быть использован в любом месте, где мог бы быть использован простой оператор. Пример составного оператора:



Слова в других операторах также действуют как знаки пунктуации. Ни одно из этих слов еще не встречалось, но вот пример:



Операторы разделены знаками пунктуации, поэтому расположение программы на странице с точки зрения компилятора значения не имеет. Вполне достаточно придерживаться двух правил:

- Не писать слова вместе:

```
PROGRAMpainter(INPUT,OUTPUT);
    CONSTpi=3.14;
```

- Не разрывать слово пробелами или переходом на новую строку:

```
PRO GRAM  painter  ( INPUT, OUT
    PUT);  CONST p i = 3.1 4;
```

(пробелы, которые *не* помечены жучками, разрешены)

В остальном компилятору все равно, как будет расположена программа, однако, это совсем не безразлично для программиста. Польза *отступов*, проясняющих структуру программы, видна из вступительного примера. В этой книге не предлагается никаких специфических правил отступов; все принципы излагаются на примерах. Если же примеры из этой книги выполняются в системе, где есть *автоматическое* форматирование отступов, то внешне программы, вероятно, будут отличаться от моих. Взгляды на выбор отступов весьма различны, но все согласны в одном – отступы должны делать структуру программы максимально наглядной. (Загляните вперед, на с. 27, и вы увидите там программу с широким использованием отступов.)

Слова PROGRAM, CONST, VAR, BEGIN, END (и еще три десятка, знакомство с ними – впереди) называются *зарезервированными словами*. Зарезервированные слова нельзя расширять:

```
CONSTANT        VARI
```

и сокращать:

```
PROG painting ( INPUT, OUTPUT );
```

Использовать можно либо прописные, либо строчные буквы, либо те и другие вместе. В этой книге используются и те и другие. О причинах этого речь пойдет чуть позже.

```
PROGRAM PAINTER(INPUT,OUTPUT);   program painter(input,output);
Program PAINTER(INput,OUTput);
```

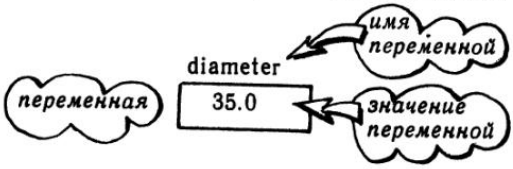
Однако в *строках* разница между прописными и строчными буквами существует:

```
WRITE(' ВАМ НЕОБХОДИМО',fullpots,' БАНОК')  ⇒  ВАМ НЕОБХОДИМО 9 БАНОК
WRITE(' Вам необходимо',fullpots,' Банок')  ⇒  Вам необходимо 9 Банок
```

Имена задавайте такой длины, какая вам больше нравится. При этом только убедитесь что первые восемь символов во всех именах различаются. Так, например, некоторые компиляторы воспримут имена NUMBEROFMEN и NUMBEROFWOMEN как одинаковые.

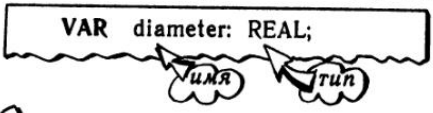
ПЕРЕМЕННЫЕ

НА ПРИМЕРЕ ПРОСТОЙ ПЕРЕМЕННОЙ ИЛЛЮСТРИРУЕТСЯ ОБЩЕЕ ПОНЯТИЕ ПЕРЕМЕННЫХ

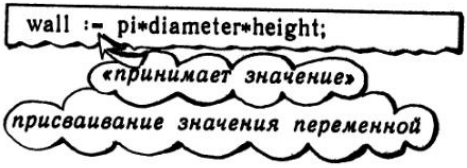


Незапертые маленькие коробочки из вступительного примера называются *переменными*. Простая переменная – это некоторая коробочка, имеющая *имя* и *значение*.

В компьютере переменная создается в результате объявления ее в разделе **VAR**. Объявление одновременно специфицирует имя переменной и тип ее значения. Типы переменных обсуждаются напротив.



Символ, состоящий из двоеточия и знака равенства ~ произносится как «принимает значение» ~ и указывает на то, что значение (обычно результат вычисления выражения) должно быть помещено в коробочку.



Во вступительном примере значения переменных оставались постоянными; переменным были присвоены определенные числа, которые не менялись. Однако в этой программе можно обойтись меньшим числом переменных. В предлагаемой версии используется несколько присваиваний переменной x:

```
PROGRAM painting ( INPUT,  OUTPUT );

  CONST  pi = 3.14;

  VAR diameter, height, coverage, x : REAL;
      fullpots : INTEGER;

BEGIN
  READ(diameter, height, coverage);
  x := pi*SQR(diameter)/4.0;
  x := x+pi*diameter*height;
  x := x / coverage;
  fullpots := TRUNC(x) + 1;
  WRITE('Вам необходимо',fullpots,' банок')
END.
```

x: 961.63 → 1895.78 → 8.03

каждое новое присваивание стирает то, что было до этого

КОНСТАНТЫ

ПОНЯТИЕ ИЛЛЮСТРИРУЕТСЯ НА ПРИМЕРЕ ПРОСТОЙ ИМЕНОВАННОЙ КОНСТАНТЫ



Запертая маленькая коробочка из вводного примера называется *константой*. Константы создаются в результате объявления их в разделе **CONST**. *Тип* константы задается автоматически, исходя из *формы* ее значения. Например, pi – константа типа REAL. Это определяется десятичной точкой в числе 3.14.

```
CONST  pi = 3.14;
```

обратите внимание, здесь стоит знак равенства, а не :=

СТАНДАРТНЫЕ ТИПЫ

INTEGER, REAL, CHAR, BOOLEAN

Целые (INTEGER) числа включают положительные, отрицательные и нуль:

- все *константы* типа INTEGER должны быть объявлены так

```
CONST dozen=12, decr= -1;
```
(*нет десятичной точки*)

- все *переменные* типа INTEGER должны быть объявлены так

```
VAR i, j, k : INTEGER;
```

- *выражение*, присваиваемое целой переменной, должно принимать целое значение; поэтому деление вида i/j использовать нельзя, см. ниже.

```
i:= dozen*decr+6;
j:= TRUNC(3.14)+7;
k:= i/j
```
(*нет десятичной точки*; *результат типа REAL*)

Вещественные числа (тип REAL) – это числа с дробной частью. Вещественное число может быть отрицательным, нулем или положительным:

- все *константы* типа REAL должны быть объявлены так

```
CONST pi=3.1415926; couple=2.0;
```
(*десятичная точка – важна*)

- все *переменные* типа REAL должны быть объявлены так

```
VAR x, y, z : REAL;
```

- *выражение*, присваиваемое вещественной переменной, может принимать целое или вещественное значение; перед присваиванием целые значения автоматически преобразуются в вещественные. (В выражении допускается одновременное использование целых и вещественных членов, последствия этого описаны на следующей странице.)

```
x:= pi/180.0;
y:= i/j;
z:= i+2
```
(*или 180*; *веществ. результат*; *автом. преобраз. из int. в real*)

Литеры (тип CHAR) – это буквы, цифры и символы; тип CHAR подразумевает одиночную литеру.

- все *константы* типа CHAR при объявлении надо заключать в апострофы

```
CONST p='A'; q='*'; r='6';
```

- все *переменные* типа CHAR должны быть объявлены так

```
VAR a,b,c : CHAR;
```

- литеры можно сравнивать, результат будет логического типа. Литеры сравниваются на основе их *порядковых* номеров: 'A'<'B', 'B'<'C' и т.д., '0'<'1', '1'<'2' и т.д.

```
IF (p>a) AND (c>'X') THEN
```

Логические значения (тип BOOLEAN) – это *false* (ложь) и *true* (истина). (В Паскале *false* «меньше чем» *true*.)

- логические *константы* уже предусмотрены и нет нужды в их объявлении программистом



- логические переменные должны быть объявлены так

```
VAR ok, alive: BOOLEAN;
```

- Логическое *выражение* должно приводиться к значению *true* или *false*

```
IF a=b THEN ok:= TRUE;
IF alive AND ok THEN WRITE('Great')
```

ВЫРАЖЕНИЯ

СТАРШИНСТВО.
СКОБКИ МЕНЯЮТ ПОРЯДОК ДЕЙСТВИЙ.
ТИПЫ: INTEGER, REAL, BOOLEAN

Использование арифметических выражений в операторах присваивания проиллюстрировано во вводном примере. Вот два из них:

```
pots := (top + wall)/ coverage;
fullpots := TRUNC(pots) + 1 ;
```

присваивание вещественного

присваивание целого

Скобки обеспечивают необходимый порядок вычислений. Если бы в первом примере скобки были опущены:

```
pots := top + wall / coverage;
```

то сначала было бы выполнено деление, приоритет которого выше. Приоритет в арифметических выражениях:

выше	*	/
ниже	+	–

** и /* *имеют равный приоритет*

+ и – *имеют равный приоритет*

Во втором из приведенных примеров производится присваивание значения целой переменной. Функция TRUNC( ) дает целый результат, а число 1 записано без десятичной точки; таким образом, оба слагаемых в сумме дают целое значение. Вообще, когда все члены выражения – целые, само выражение принимает целое значение.

У сформулированного выше правила существует важное исключение: деление (с использованием знака «/») всегда дает вещественный результат:

```
6.5 / 2 ⟶  3.25   (вещественное)
6   / 2 ⟶  3.0    (вещественное)
```

Деление нацело ~ нахождение частного и остатка ~ может быть выполнено при помощи операций **DIV** и **MOD**, речь о которых пойдет позже.

Выражение может включать в себя и целые и вещественные члены. Наличие хотя бы одного вещественного члена или знака «/» приводит к тому, что значение результата будет вещественным. Функции TRUNC( ) и ROUND( ) могут быть использованы для преобразования вещественного числа в целое.

Функция SQR( ) возводит значение аргумента (записанного внутри скобок) в квадрат. В Паскале нет оператора возведения в произвольную степень (подобного ↑ в Бейсике). Возведение в степень здесь осуществляется с использованием логарифмов – это показано напротив. Нематематики должны принять на веру, что вместо A↑X в Бейсике, на Паскале можно написать EXP(LN(A)*X).

Кому-то это может показаться странным, но знаки >, <= и т.д. тоже могут быть использованы как операции. Например, 1 > 2 имеет значение false, а 1 + 2 = 3 – true. Выражения, содержащие подобные операции, принимают логические значения и называются логическими выражениями или еще *условиями*. В состав логических выражений могут входить логические операции NOT (*не*), AND (*и*), OR (*или*), а также члены типа CHAR:

```
ok := (1=2) OR (ch >= 'A');
IF ok THEN
```

ЗАЕМ

ПРИМЕР ДЛЯ ИЛЛЮСТРАЦИИ АРИФМЕТИЧЕСКИХ ВЫРАЖЕНИЙ, ВОЗВЕДЕНИЯ В СТЕПЕНЬ, ПЕЧАТИ ВЕЩЕСТВЕННОГО РЕЗУЛЬТАТА

Месячная выплата m по займу в s фунтов на n лет под процент p вычисляется по формуле:

$$m = \frac{sr(1 + r)^n}{12((1+r)^n - 1)}$$

где $r = p \div 100$

Здесь представлена программа вычисления m по известным значениям s, p и n.

```
PROGRAM loans( INPUT, OUTPUT );
  VAR
     m, s, p, n, r, a : REAL ;
BEGIN
  READ(s, p, n);
  r := p/100;
  a := EXP( LN(1 + r)* n);
  m := (s*r*a)/(12*(a-1));
  m := TRUNC(100*m + 0.5)/100;
  WRITELN; WRITELN;
  WRITE(' Взято £', s:4:2);
  WRITE(' под',p:5:2,'%');
  WRITE(' на',n:5:2,'лет');
  WRITELN;
  WRITE(' Месячная выплата £',m:5:2);
  WRITELN;
  WRITE(' Общая прибыль равна £',m*n*12-s:5:2);
  WRITELN
END.
```

клавиатура *экран*

Если ваш Паскаль – интерактивный, то вставьте сюда подходящую подсказку для пользователя

$a = (1+r)^n$ *через логарифм*

округление до пенни

каждый WRITELN начинает новую строку

:5:2 означает поле из 5 позиций с 2 позициями под дробную часть

1 . 6 3
1 2 3 4 5

s — *сумма займа*
p — *процент*
n — *срок выплаты (лет)*
r — $p \div 100$
a — $(1+r)^n$
m — *выплата*

Экран после выполнения программы будет выглядеть так:

```
99.99  14.5  10

Взято £99.99 под 14.5% на 10 лет
Месячная выплата £ 1.63
Общая прибыль равна £95.61
```

исходные данные

результат

Если ваша версия Паскаля поддерживает интерактивную работу, то, чтобы на экране появился запрос необходимых данных, перед оператором READ вставьте еще оператор WRITE.

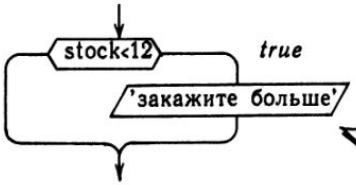
Приведенная выше программа вовсе не проверяет исходные данные. Если пользователь программы введет неверное число (например, букву o вместо числа 0), то программа не сработает. Большинство программ из этой книги столь же уязвимы в этом отношении. Причина такой уязвимости в том, что тщательная проверка данных заметно удлинит программы ~ цель которых – компактная иллюстрация различных аспектов программирования. Читателю оставляется в качестве упражнения сделать программы дружественными и устойчивыми к ошибкам.

УСЛОВИЯ

РЕЗУЛЬТАТ ЛОГИЧЕСКОГО ВЫРАЖЕНИЯ ПОЗВОЛЯЕТ ВЫБРАТЬ ХОД ВЫЧИСЛЕНИЙ

В зависимости от результата программа может выполнять различные действия.
Вот элементарный пример:

```
IF stock<12 THEN WRITELN('Закажите больше');
```



Логическое выражение stock < 12 может принять значение *true* или *false*. Если stock < 12 принимает значение *false*, то оператор WRITELN не выполняется: управление будет просто передано следующему оператору.

Количество операторов, входящих в оператор **IF** и выполняемых в зависимости от значения логического выражения, не ограничено.

```
READ(key);
IF  key = 'y'
  THEN
    BEGIN
  { эти операторы выполняются, если ответом является Y RETURN }
    END
  ELSE
    BEGIN
  { эти операторы выполняются, если ответ отличен от Y RETURN }
    END;
  { эти операторы выполняются после, независимо от ответа }
```

Условия, проиллюстрированные здесь, есть просто сравнение двух величин. Допускаются и более сложные условия, включающие логические операции AND (и), OR (или), NOT (не). Например,

```
(initial>='E') AND (initial<'L')
```

где *initial* – переменная типа CHAR, в которой записана начальная буква фамилии. Результат *true* будет означать, что в телефонной книге фамилия находится между E и K.

```
keen := (x = y) OR (z >= 3);
IF NOT keen THEN
```

Значение логического выражения можно присвоить логической переменной и лишь затем проанализировать.

ПОЛЯ

ПОЛЯ ДЛЯ ПЕЧАТИ ЧИСЕЛ, СТРОК И ДЕСЯТИЧНЫХ ДРОБЕЙ

Ширина поля и количество дробных десятичных разрядов указываются после двоеточия – это показано ниже:

```
i := 123;        r:= 123.456;
WRITELN( i:8);
WRITELN( -r:8:2);
WRITELN('String':8)
```

ширина поля

число позиций под дробную часть (только для вещественных)

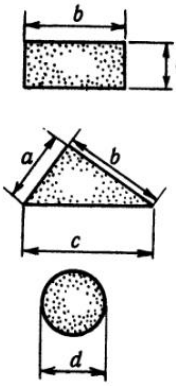


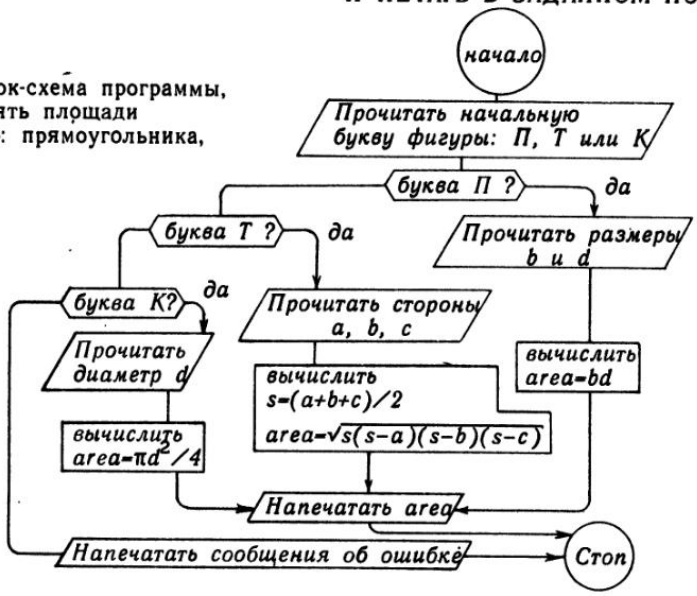
Задавая ширину поля *выражением*, можно рисовать кривые, но об этом чуть позже.

ГЕОМЕТРИЧЕСКИЕ ФИГУРЫ

ИСПОЛЬЗОВАНИЕ УСЛОВИЙ И ПЕЧАТЬ В ЗАДАННОМ ПОЛЕ

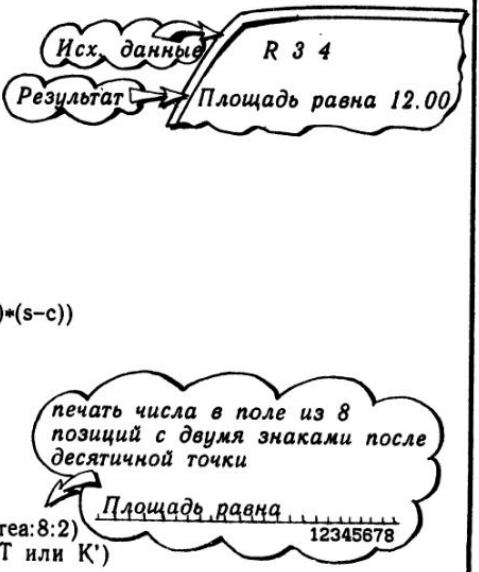
Здесь приведена блок-схема программы, позволяющей вычислять площади геометрических фигур: прямоугольника, треугольника, круга.





Вот программа, которая соответствует этой блок-схеме:

```
PROGRAM  shapes(INPUT,OUTPUT);
  CONST
    pi = 3.1415926;
  VAR
    letter: CHAR;   s,area,a,b,c,d: REAL;  ok: BOOLEAN;
BEGIN
  ok:= TRUE;
  READ(letter);
  IF (letter='П') OR (letter='п')
    THEN
      BEGIN
        READ(b,d);
        area:= b*d
      END
    ELSE IF (letter='Т') OR (letter='т')
           THEN
             BEGIN
               READ(a,b,c);
               s:= 0.5*(a+b+c);
               area:= SQRT(s*(s-a)*(s-b)*(s-c))
             END
    ELSE IF (letter='К') OR (letter='к')
           THEN
             BEGIN
               READ(d);
               area:= pi*SQR(d)/4
             END
    ELSE ok:= FALSE;
  IF ok THEN WRITE('Площадь равна',area:8:2)
        ELSE WRITE('Должно быть П, Т или К')
END.
```



ЦИКЛЫ

ФУНДАМЕНТАЛЬНОЕ ПОНЯТИЕ В ЛЮБОМ ЯЗЫКЕ ПРОГРАММИРОВАНИЯ

Можно заставить программу выполнять некоторую последовательность инструкций несколько раз подряд:

```
PROGRAM  xmas( OUTPUT );
  VAR    humbug: INTEGER;
BEGIN
  FOR humbug := 1 TO 3 DO
    WRITELN(' Мы желаем Вам веселого Рождества');
  WRITELN('И счастливого Нового года');
END.
```

Этот оператор выполняется, когда humbug *равно 1, когда* humbug *равно 2 и когда* humbug *равно 3*

Этот оператор выполняется один раз, когда цикл завершен

Чаще употребляется вот такая организация цикла:

```
PROGRAM  tables( INPUT, OUTPUT);
  VAR  valu, product, multiplier : INTEGER;
BEGIN
 READ(valu);
 FOR multiplier:=1 TO 10 DO
  BEGIN
    product:= multiplier*valu;
    WRITELN(muliplier:2,'*',valu:2,'=',product:4)
  END
END.
```

Замечание: WRITELN('Что-нибудь') *эквивалентно* WRITE('Что-нибудь'); WRITELN

BEGIN *и* END *– это «скобки» в которые заключен составной оператор, следующий за* DO

Если результат этих простых программ сразу не очевиден для вас, то перед дальнейшим чтением выполните их на компьютере. Циклы – основа программирования.

Цикл FOR называется «конечным», потому что число повторений определено до начала цикла. Цикл REPEAT таковым не является. Часть между внешними BEGIN и END в последней из приведенных программ можно заменить следующим фрагментом:

```
READ(valu);
multiplier:=1;
REPEAT
  product:=multiplier*valu;
  WRITELN(muliplier:2,'*',valu:2,'=',product:4);
  multiplier:=multiplier+1;
UNTIL multiplier > 10
```

приращение в цикле

Примеры уместного использования цикла REPEAT *встретятся позже*

Циклы FOR и REPEAT выполняются хотя бы один раз (если не произойдет что-нибудь столь серьезное, что они вообще не выполнятся). Однако существует цикл, в котором проверка на выполнение осуществляется в начале цикла и, если эта проверка не проходит, то цикл пропускается:

```
READ(valu);
multiplier:=1;
WHILE multiplier <= 10 DO
 BEGIN
    product:=multiplier*valu;
    WRITELN(muliplier:2,'*',valu:2,'=',product:4)
    multiplier:=multiplier+1;
 END
```

пропускается, когда multiplier>10

Примеры уместного использования цикла WHILE *даны позже*

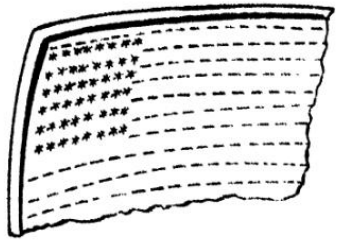
БЫЛАЯ СЛАВА

ПРОГРАММА ДЛЯ ИЛЛЮСТРАЦИИ ЦИКЛОВ НА ПРИМЕРЕ ЗВЕЗДНОПОЛОСАТОГО ФЛАГА (1912г.)

В 1912 году Американский флаг «Былая слава» имел 48 звезд (по одной на союзный штат) и 13 полос (по одной на колонию). Нижеследующая программа печатает грубое приближние «Былой славы» 1912 г. Сегодня больше штатов, а, следовательно, и звезд больше.

```
PROGRAM  glory( OUTPUT);
  VAR  row,col : INTEGER;
BEGIN
  FOR col:=1 TO 19 DO WRITE('__');
  WRITELN;
  FOR row:=1 TO 13 DO
    BEGIN
      FOR col:=1 TO 19 DO
        IF (col<9) OR (row<7)
          THEN
            WRITE('* ')
          ELSE
            WRITE('__');
      WRITELN;
    END
END.
```

подчеркивание



СИНУС

ПРОГРАММА ПЕЧАТИ СИНУСОИДАЛЬНОЙ КРИВОЙ, ИСПОЛЬЗУЮЩАЯ ЦИКЛ И ПЕРЕМЕННУЮ ШИРИНУ ПОЛЯ

Предлагаемая программа печатает график функции sin(x). График размещается на экране таким образом, чтобы колебания происходили относительно середины экрана. Вся хитрость такой печати – использование для задания ширины поля *выражения*. Ширина поля изменяется от строки к строке; в каждом строке звездочка прижимается к правому краю поля.

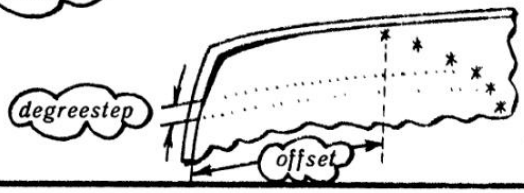
```
PROGRAM sinuous(OUTPUT);
  CONST
    offset=20; scale=18; degreestep=8;
  VAR
    i: INTEGER;   k: REAL;
BEGIN
  k:= degreestep * 3.1415926 / 180;
  FOR i:=0 TO MAXINT DO
    WRITELN('*':ROUND(offset+scale*SIN(k*i)))
END.
```

эти три значения рассчитаны на ТВ монитор; подберите их так, чтобы они подходили к вашему оборудованию

радианы из градусов

0,1,2,... градусы

ширина поля – выражение



УПРАЖНЕНИЯ

1. Выполните программу о займах *loans*. Поэкспериментируйте с различными исходными данными. Если вы введете нулевое значение процента, то программа не сработает. Включите в программу проверку такой возможности и обеспечьте для этого случая печать правильных результатов. Если ваш Паскаль допускает интерактивный ввод, предусмотрите в программе сообщения пользователю о необходимых исходных данных.

2. Выполните программу о геометрических фигурах *shapes*. Усовершенствуйте программу – сделайте после печати результата возврат к решению новой задачи. Пусть программа воспринимает букву Z как признак остановки (т.е. будут распознаваться буквы П, Т, К и Z).

3. Выполните программу построения графика *sinuous*. Постройте график затухающих колебаний, задав вместо y=sin x функцию y=sin x / exp x .

3

СИНТАКСИС

СТИЛЬ НАПИСАНИЯ
ОБОЗНАЧЕНИЯ
ЭЛЕМЕНТЫ
КОМПОНЕНТЫ
СИНТАКСИС ВЫРАЖЕНИЯ
СИНТАКСИС ОПЕРАТОРА
СИНТАКСИС ПРОГРАММЫ
СИНТАКСИС ТИПА

СТИЛЬ НАПИСАНИЯ

ЗАРЕЗЕРВИРОВАННЫЕ СЛОВА,
ПРЕДОПРЕДЕЛЕННЫЕ ИМЕНА и
имена, которые
придумывает программист

Обратите внимание на разницу в стилях написания самой первой программы; здесь она приводится еще раз:

```
PROGRAM painting(INPUT,OUTPUT);
  CONST pi = 3.14;
  VAR   diameter,height,coverage,top,
        wall,pots :  REAL;
        fullpots  :  INTEGER;
BEGIN
  READ( diameter, height, coverage);
  top:= pi * SQR( diameter)/4.0;
  wall:= pi * diameter * height;
  pots:= ( top + wall )/coverage;
  fullpots:= TRUNC( pots ) + 1;
  WRITE('Вам необходимо',fullpots,' банок')
END.
```

Компьютер, работая с программой на Паскале, не различает прописные и строчные буквы. Исключением здесь являются буквы внутри апострофов. Таким образом, программа может быть целиком набрана прописными буквами:

```
PROGRAM PAINTING(INPUT,OUTPUT);
  CONST PI = 3.14;
```

или целиком строчными:

```
var     diameter,height,coverage,top,
        wall,pots :  real;
```

или прописными и строчными буквами вместе:

```
FullPots:= TRUNC(Pots) + 1;
```

Разница существенна в промежутке между апострофами:

```
Write('Вам необходимо',FullPots,' банок')
```

разница несущественна | разница существенна | разница несущественна | разница существенна

Тем не менее во вводном примере ~ как и во всей книге в дальнейшем ~ используются три стиля написания с тем, чтобы выделить три типа слов в Паскале:

- **PROGRAM, CONST, VAR, BEGIN,...** – *зарезервированные слова,* которые ведут себя подобно знакам пунктуации. Каждое такое слово в Паскале имеет свое определенное значение

- INPUT, REAL, READ, TRUNC,... – *предопределенные имена;* эти имена указывают на возможности, предоставляемые Паскалем для объявления файлов (INPUT), типов (REAL) или вызова полезных функций (WRITE( ), TRUNC( )). Однако программист вправе игнорировать подобные возможности и использовать эти имена для других целей

- painting, pi, diameter, height,... – имена, которые составляет программист с целью идентификации переменных, констант, процедур и прочих объектов, которые еще появятся.

Программу проще понять, если смысл имени или слова ясен из его написания.

ОБОЗНАЧЕНИЯ

ДЛЯ ОПИСАНИЯ ФОРМ ЗАПИСИ ОБЪЯВЛЕНИЙ И ОПЕРАТОРОВ ПАСКАЛЯ

Форму написания и пунктуацию Паскаля помогают определить схематические обозначения. Обозначения, которые описаны ниже, – смесь двух общепринятых систем: диаграмм, похожих на железнодорожные пути, которые используются в нескольких книгах по Паскалю, и формы Бэкуса–Наура (БНФ), которая используется в определении Паскаля в стандарте ISO. Железнодорожные диаграммы зрительно могут показаться сложными при разборе сколько-нибудь нетривиальной конструкции; БНФ хороша для формального определения, но не столь удобна для быстрых справок или общей оценки синтаксической структуры. Представленная ниже система обозначений предназначена для быстрых справок и общих оценок практически без потери строгости.

курсив — Курсивные буквы используются в названиях определяемых объектов: *цифра, оператор, выражение* и так далее

::= — как и в БНФ означает «определено как...»

БУКВЫ
& + (* /
012 и т.д. — Эти литеры означают сами себя; в определениях они понимаются как есть. Буквы по желанию можно заменять на строчные: A на a, B на b, C на c и т.д.



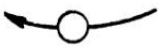
Вертикальные скобки, содержащие несколько строк, позволяют выбирать только одну строку



Эта стрелка говорит, что объект(ы), над которым она нарисована, не обязателен (может быть пропущен)



Эта стрелка разрешает возврат ~ таким образом, можно выбрать еще один объект из вертикальных скобок или тот же самый



Круг (или колбаска) содержит разделитель, который необходимо использовать при возврате к другому объекту. Отсутствие кружка означает отсутствие разделителя

имя
пер
конст
файла
типа
фун
проц — Подпись у *имени* означает, что именуется этим именем; будь то *переменная, константа, файл, тип, функция* или *процедура*. (Этот механизм уже из области семантики, а не синтаксиса.)

▶ — Этот символ располагается перед примером вместо слова «например»

Некоторые слова, используемые в нижеследующих определениях, отличаются от тех, что используются в стандартных работах по Паскалю. В частности, я использую слово *имя* вместо *идентификатор, терм* вместо *множитель*, и мне не нужно слово, обозначающее *слагаемое*. Я использую слово *компаратор* вместо *операции сравнения*.

ЭЛЕМЕНТЫ

СИНТАКСИСА ПАСКАЛЯ: буква, цифра, символ, пробел, операция, компаратор

буква ::= A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

цифра ::= 0 1 2 3 4 5 6 7 8 9

символ ::= + – * / = < > [ ] . , : ; ↑ ( )

строчные буквы эквивалентны прописным. Например, div ≡ DIV. *Исключением являются цепочки литер.*

Обратите внимание, что апостроф и фигурные скобки {'} отсутствуют в определении *символа*. Они явно фигурируют в диаграммах

В *цепочках* или *комментариях* могут также использоваться и другие имеющиеся на клавиатуре символы, такие как £, ! или $.

Добавьте сюда строчные и русские буквы, если их допускает компилятор.

пробел ::= *клавиша пробела, нажатая один раз*

Пробелы важны в цепочках и комментариях. Переход на новую строку в цепочках и комментариях не допустим. [1] *)

операция ::= * / DIV MOD AND — *операции, старшие по приоритету*

+ – OR — *операции, младшие по приоритету*

компаратор ::= < <= = > >= <> IN

приоритет ниже, чем у любой из операций

В выражении 3+4*5 умножение (*) выполняется перед сложением (+), потому что приоритет умножения выше. Чтобы изменить порядок операций, используйте скобки, например, (3+4)*5.

Выражение 5–3=2 *истинно*, потому что оно рассматривается как (5–3)=2, а не как 5–(3=2). Другими словами, компаратор имеет более низкий приоритет.

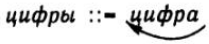
*) См. примечания редактора перевода в конце главы – *Прим. ред.*

КОМПОНЕНТЫ

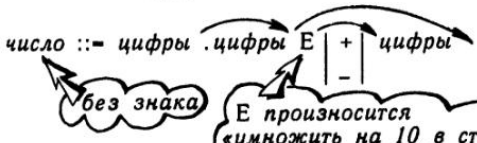
СИНТАКСИСА ПАСКАЛЯ: имя, цифры, число, константа, переменная, цепочка, комментарий



▶ X
▶ H2SO4 ▶ h2so4



▶ 6 ▶ 0123444

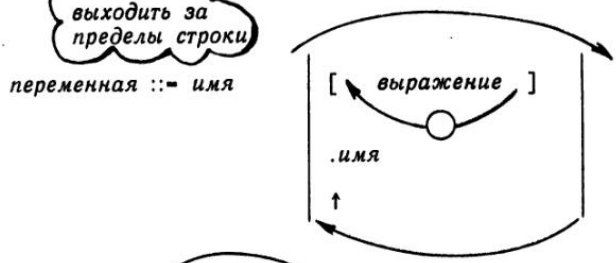


▶ 66
▶ 66.2
▶ 662E-01



▶ -55.4e-03 *т.е. -0.0554*
▶ k
▶ -k
▶ 'Me'

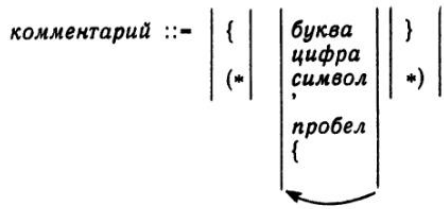
допускается CONST k=2; m=-k;



▶ k
▶ array[6,2*k]
▶ person.age
▶ ptr↑
▶ array[6][2*k]



▶ 'Вам необходимо'
▶ '£'



▶ {Комментарий программиста}
▶ (* Это тоже комментарий *)

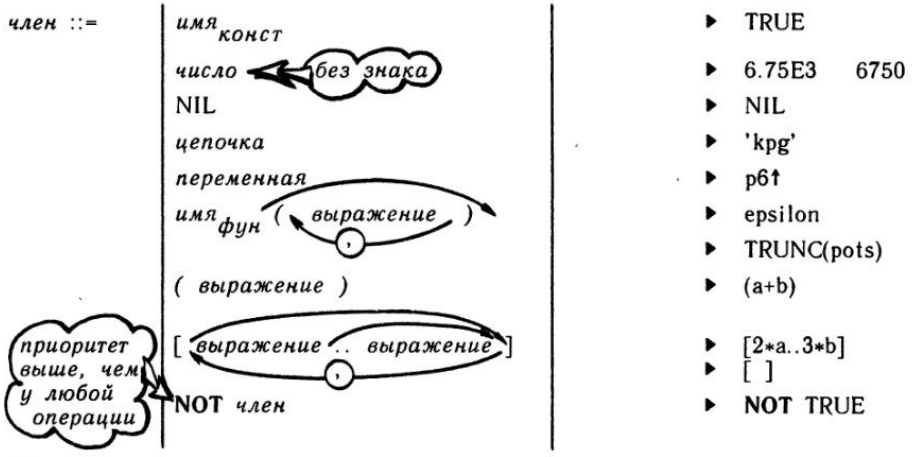
Комментарий воспринимается как один пробел и может быть вставлен всюду, где допустим пробел.

СИНТАКСИС ВЫРАЖЕНИЯ *И ЛОГИЧЕСКОГО ВЫРАЖЕНИЯ, НАЗЫВАЕМОГО ТАКЖЕ УСЛОВИЕМ*

«Элементы» и «компоненты» синтаксиса Паскаля теперь могут быть объединены в определении *выражения*. Во вводном примере фигурирует несколько выражений, следующие два из которых типичны:

```
(top + wall)/coverage;   pi * SQR(diameter)/4.0;
```

Выражение содержит один или более *членов*. Члены связаны между собой скобками и операциями. Членом может быть имя переменной (например, top), или ссылка на функцию (например, TRUNC(pots)) или что-нибудь еще из перечисленного ниже.



Определив *член*, можно определить *выражения* как набор членов, объединенных операциями и компараторами:



▸ (pi * SQR(diameter)/4) + (pi*diameter*height)

обе операции высокого приоритета — низкий приоритет — в этом примере благодаря приоритету скобки необязательны

Выражение, содержащее один или более компараторов или единственный логический член, называется *логическим выражением* или *условием*.

▸ -3 > 1 ← ложно ▸ TRUE ← истинно

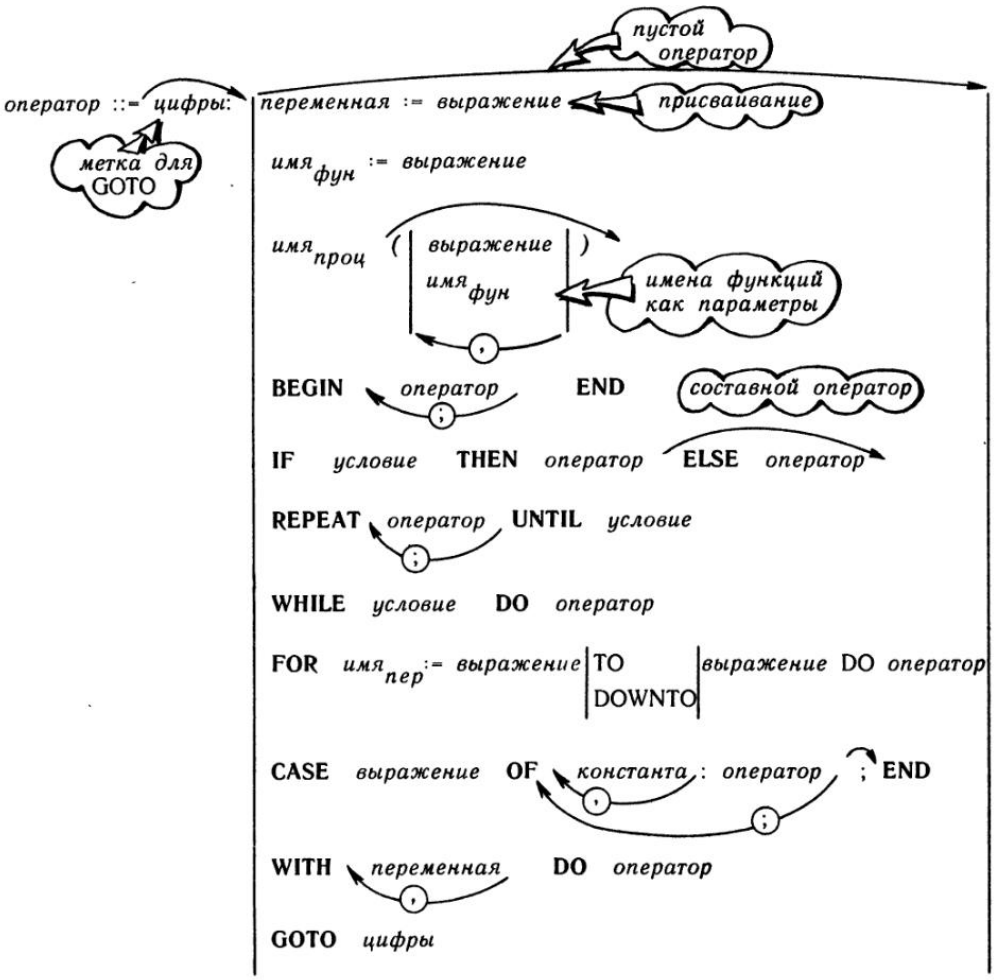
В операторах WRITE и WRITELN может также использоваться нестандартная форма выражения:

нестандартное ::= *выражение* : *выражение* : *выражение*

▸ WRITE(x:8:2) ▸ WRITELN('*':ROUND(offset+scale*SIN(k*i)))

СИНТАКСИС ОПЕРАТОРА *НЕКОТОРЫЕ ФОРМЫ ЕЩЕ НЕ БЫЛИ ПРЕДСТАВЛЕНЫ*

Ниже приводится определение *оператора*. Некоторые из *операторов* упоминаются здесь впервые.



- 100: area:= p*SQR(diameter)/4 ← *оператор присваивания ~ с меткой*
- **BEGIN** temp:=a; a:=b; b:=temp **END** ← *составной оператор*
- **IF** a>b **THEN** **BEGIN** temp:=a; a:=b; b:=temp **END** ← *оператор IF*
- **BEGIN** ; t:=a; ; a:=b; b:=t; **END**

пустые операторы

СИНТАКСИС ПРОГРАММЫ

ЧИТАЯ ПЕРВЫЙ РАЗ, ЭТИ ДВЕ СТРАНИЦЫ МОЖНО ПРОПУСТИТЬ

Вот определение *программы* по принципу сверху вниз:

программа ::= **PROGRAM** *имя* (*имя*$_{файла}$, ...) ; *блок* .

заметьте .

Здесь:

блок ::= **LABEL** *цифры* , ... ;

{ *продолжается на следующей строке* }

CONST *имя* = *константа* ; ... ;

{ *продолжается* }

TYPE *имя* = *тип* ; ... ;

{ *продолжается* }

VAR *имя* , ... : *тип* ; ... ;

{ *продолжается* }

FUNCTION *имя* *параметры* : *имя*$_{типа}$

PROCEDURE *имя* *параметры*

; *блок* ;

{ *продолжается* }

BEGIN *оператор* ; ... **END**

Где:

параметры ::= (**VAR** *имя* , ... : *имя*$_{типа}$

FUNCTION *имя* *параметры* : *имя*$_{типа}$

PROCEDURE *имя* *параметры*

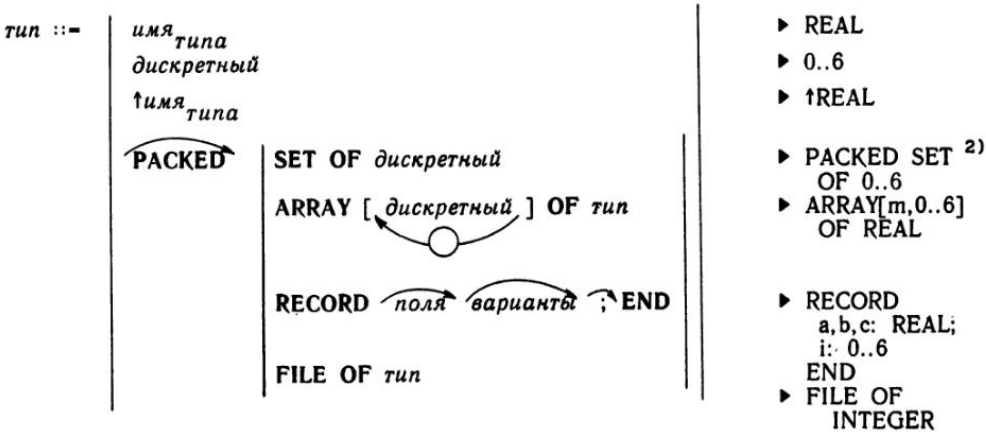
; ...)

функции и процедуры как параметры функций и процедур

СИНТАКСИС ТИПА

ДЛЯ ПОЛНОТЫ ОПРЕДЕЛЕНИЯ ПРОГРАММЫ ПО ПРИНЦИПУ СВЕРХУ ВНИЗ

Вот определение *типа* по принципу сверху вниз:



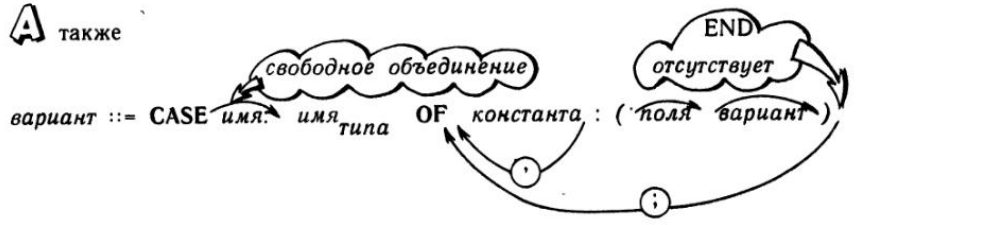
Где



И



А также



На этом завершается определение синтаксиса стандартного Паскаля (ISO).

ПРИМЕЧАНИЯ РЕДАКТОРА

[1] (с. 34) Во всех известных мне версиях Паскаля допускается переход на новую строку внутри комментариев.

[2] (с. 39) В конструкциях **SET OF** *дискретный* и **ARRAY** [*дискретный*] **OF** *тип* в качестве *дискретный* нельзя использовать тип INTEGER.

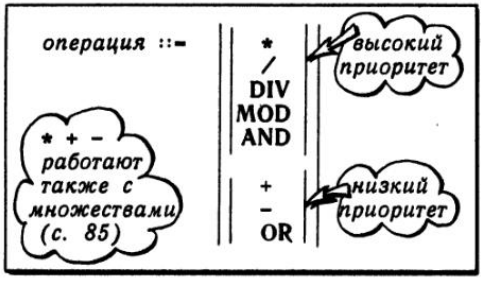
4

АРИФМЕТИКА

ОПЕРАЦИИ
РАЗМЕР И ТОЧНОСТЬ
КОМПАРАТОРЫ
АРИФМЕТИЧЕСКИЕ ФУНКЦИИ
ТРИГОНОМЕТРИЧЕСКИЕ ФУНКЦИИ
ФУНКЦИИ ПРЕОБРАЗОВАНИЯ
ЛОГИЧЕСКИЕ ФУНКЦИИ
ФУНКЦИИ НАД ДИСКРЕТНЫМИ ТИПАМИ

ОПЕРАЦИИ

* / DIV MOD
\+ - AND OR



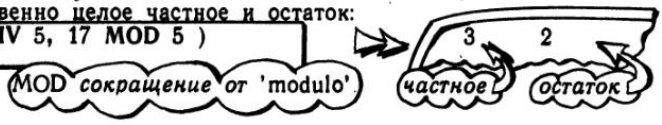
Для удобства, синтаксис *операций* здесь приведен еще раз.

Использование представленных операций поясняется на данном развороте. Если использование скобок не кажется очевидным, то лучше вернуться к с. 36 – там изложен синтаксис *выражений*.

При отсутствии скобок выражения вычисляются слева направо, сначала выполняются операции с высоким приоритетом, затем – с низким приоритетом. Для указания любого нужного порядка вычислений можно ставить скобки. Например, члены a*b/c и (a*b)/c будут вычисляться одинаково, а в члене a*(b/c) скобки влекут изменение порядка вычислений.

Операции **DIV** и **MOD** предназначены для деления нацело; они позволяют получить соответственно целое частное и остаток:

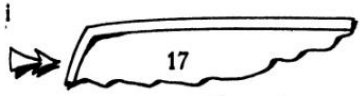
```
WRITELN( 17 DIV 5, 17 MOD 5 )
```



Для положительных значений i и j выполняются следующие соотношения:

(i **DIV** j) * j + (i **MOD** j) = i

```
WRITELN( (17 DIV 5)*5 + (17 MOD 5))
```



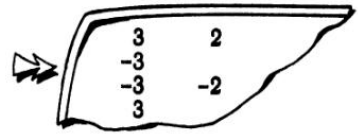
При неположительных значениях ситуация несколько сложнее. Второй операнд у операции **MOD**, например, не может быть отрицательным: [1]

```
WRITELN( 17 MOD -5 )
```



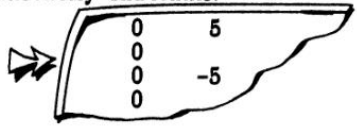
Допустимые сочетания приведены ниже: [2]

```
WRITELN( 17 DIV 5, 17 MOD 5 );
WRITELN( 17 DIV (-5));
WRITELN( -17 DIV 5, -17 MOD 5 );
WRITELN( -17 DIV (-5))
```



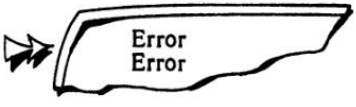
Первый операнд может быть меньше по абсолютному значению:

```
WRITELN( 5 DIV 17, 5 MOD 17 );
WRITELN( 5 DIV (-17));
WRITELN( -5 DIV 17, -5 MOD 17 );
WRITELN( -5 DIV (-17))
```



Если же делитель равен нулю или один из операндов – не целое, то появляется сообщение об ошибке:

```
WRITELN( 17 DIV 0 );
WRITELN( 17.0 MOD 5 )
```



Операции + и - могут использоваться как 'унарные' операции (проще говоря, как знаки) перед целыми или вещественными выражениями:

```
WRITELN( -2, +2*3 );
WRITELN( -2.0:4:1 ).
```

⇒

```
-2     6
-2.0
```

Результат операций *, + и - будет целый, если *оба* операнда - целые:

```
WRITELN( 2*3, 2+3, 2-3 )
```

⇒

```
6  5  -1
```

и вещественный, если хотя бы один или оба операнда - вещественные:

```
WRITELN( 2.0*3:4:1 );
WRITELN( 2+3.0:4:1 )
```

⇒

```
 6.0
 5.0
```

Операция / дает вещественный результат ~ даже если оба операнда являются целыми:

```
WRITELN(   6/2:4:1 );
WRITELN( 6.0/2:4:1 )
```

⇒

```
 3.0
 3.0
```

Делитель не может быть равен нулю:

```
WRITELN( 6/0:4:1 )
```

⇒

```
Error
```

Операции **AND** и **OR**, применяемые к логическим операндам, дают логический результат. Если операнды не принадлежат логическому типу, то появляется сообщение об ошибке:

```
WRITELN( 1 OR 2 );
WRITELN( 'A' AND 'B' )
```

с типом CHAR *допустимо использование только компараторов, например,* 'A'<'B'

⇒

```
Error
Error
```

Нижеследующая *таблица истиности* определяет результат применения операций **AND** и **OR** к логическим операндам:

AND	второй операнд *true*	второй операнд *false*
первый операнд *true*	✓	X
первый операнд *false*	X	X

OR	второй операнд *true*	второй операнд *false*
первый операнд *true*	✓	✓
первый операнд *false*	✓	X

Далее приведены некоторые примеры логических выражений. Обратите внимание на то, как оператор WRITELN печатает логические результаты словами. Эти слова могут быть записаны прописными или строчными буквами в зависимости от конкретной системы:

```
WRITELN( TRUE  AND TRUE , TRUE  AND FALSE );
WRITELN( FALSE AND TRUE , FALSE AND FALSE );
WRITELN( TRUE  OR TRUE , TRUE  OR FALSE );
WRITELN( FALSE OR TRUE , FALSE OR FALSE );
WRITELN((((1=2) OR (1+2=3) OR (1>2)) AND (2+3=5))
```

⇒

```
true   false
false  false
true   true
true   false
true
```

РАЗМЕР и ТОЧНОСТЬ *ЦЕЛЫХ И ВЕЩЕСТВЕННЫХ*

Целое может быть положительным, нулем или отрицательным. Константа с именем MAXINT хранит копию наибольшего целого, которое может передаваться или храниться.



Значение 32767 – типично для систем, где целые хранятся как 16-разрядные слова. В случае 32-разрядного слова максимальное целое обычно равно 2147483647.

Если в программе результат промежуточных вычислений целого выражения превышает величину MAXINT, то появляется сообщение об ошибке. Иногда этого можно избежать, добавив скобки, например, заменив i*j **DIV** k на i*(j **DIV** k).[3)]

Хотя диапазон допустимых целых простирается от –MAXINT до MAXINT, вы, возможно, обнаружите, что и значение «–(MAXINT+1)» не приводит к ошибке. Это объясняется тем, что при традиционной кодировке целых в слове из n битов можно записать числа в диапазоне от -2^{n-1} до $2^{n-1}-1$ (несимметричном относительно нуля).

Вещественное число может быть отрицательным, нулем или положительным. Его наибольшее абсолютное значение – 10^{38}; точность, обычно, – 6 или 7 десятичных значащих цифр. В таких системах наибольшее положительное или отрицательное число будет около

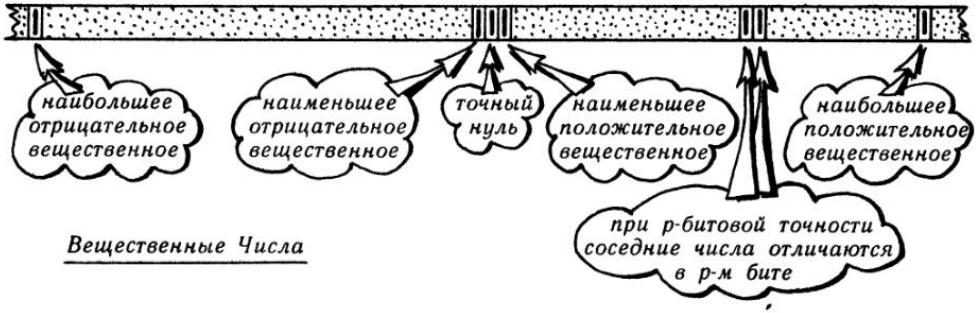
± 100 000 000 000 000 000 000 000 000 000 000 000

Наименьшее положительное или отрицательное число будет около

± 0.000 000 000 000 000 000 000 000 000 000 000 01

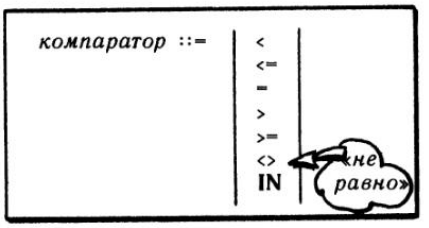
Число 1 000 000 (в случае вышеуказанной точности) еще будет отличаться от числа 1 000 001, но не от 1 000 000.1 .

Числа хранятся в виде двоичных цифр (битов), а вовсе не в десятичном виде; в этом – неизбежная неопределенность двух предыдущих абзацев. Ниже диапазоны вещественных чисел представлен наглядно:



КОМПАРАТОРЫ

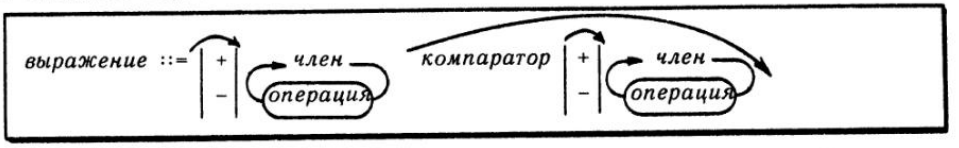
ИЛИ «ОПЕРАЦИИ СРАВНЕНИЯ». ЛОГИЧЕСКИЙ РЕЗУЛЬТАТ ИЗ СОВМЕСТИМЫХ ОПЕРАНДОВ



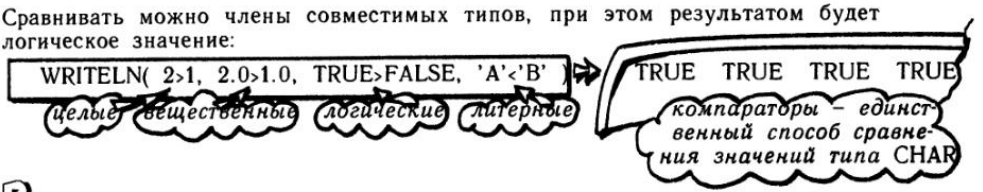
Здесь воспроизведен для удобства синтаксис *компаратора*. Смысл символов отвечает их написанию: например, >= означает «Больше или равно».

Приоритет любого *компаратора* ниже чем приоритет любой *операции*.

Чтобы подчеркнуть разницу между *операцией* и *компаратором*, приведем еще раз и синтаксис *выражения*:



Сравнивать можно члены совместимых типов, при этом результатом будет логическое значение:



Вещественные и целые члены с одинаковыми значениями взаимозаменяемы:



Синтаксическая диаграмма *выражения* допускает только один компаратор. Однако, *выражение в скобках* – это *член*. Таким образом, включая еще компаратор, можно строить более сложные выражения:



Хотя тип множество вводится в гл. 7, ниже для полноты изложения, представлены операции над множествами. «Друзья» и «знакомые» – это имена множеств, «приятель» – имя одного элемента множества.

=	друзья = знакомые	*истинно*, если все друзья – знакомые, и все знакомые одновременно друзья (множества совпадают)
◇	друзья <> знакомые	*истинно*, если среди друзей не все знакомые или среди знакомых не все друзья (различные множества)
<=	друзья <= знакомые	*истинно*, если все друзья – знакомые
>=	друзья >= знакомые	*истинно*, если все знакомые – друзья
IN	приятель **IN** друзья	*истинно*, если приятель – друг
	NOT(приятель **IN** друзья)	*истинно*, если приятель не является другом

АРИФМЕТИЧЕСКИЕ ФУНКЦИИ

В функциях всегда использовались *аргументы*:

```
x := ABS( y )
```

функция — аргумент — *ПАРАМЕТР!*

В Паскале же предпочтительнее термин *параметр*. Параметры, которые используются ниже, – *фактические* параметры. Позднее мы определим *формальные* параметры.

Следующие две функции можно применять к целым параметрам, и в этом случае они возвращают целый результат. Этим функциям можно также передавать вещественный параметр, получая вещественный результат.

ABS(*выражение*) — *абсолютное (т.е. положительное) значение параметра*

```
WRITELN(ABS(-2),ABS(0),ABS(2));
WRITELN(ABS(-2.0):4:1,ABS(0.0):4:1,ABS(2.0):4:1)
```

```
 2   0   2
2.0 0.0 2.0
```

SQR(*выражение*) — *квадрат параметра*

```
WRITELN(SQR(-2),SQR(0),SQR(2));
WRITELN(SQR(-2.0):4:1,SQR(0.0):4:1,SQR(2.0):4:1)
```

```
 4   0   4
4.0 0.0 4.0
```

Остальные арифметические функции воспринимают целый или вещественный параметр; результат в любом случае будет вещественным:

SQRT(*выражение*) — *квадратный корень*

```
WRITELN(SQRT(16):4:1,SQRT(0.64):4:1,SQRT(0):4:1);
WRITELN(SQRT(-16))
```

```
4.0 0.8 0.0
Error
```

LN(*выражение*) — *натуральный логарифм*

```
WRITELN(LN(1):4:1,LN(2.718282):4:1,LN(7.5):4:1);
WRITELN(LN(0),        LN(-1)       )
```

```
0.0 1.0 2.0
Error
```

EXP(*выражение*) — *экспонента*

```
WRITELN(EXP(0):4:1,EXP(1):8:5,EXP(2.014903):4:1);
WRITELN(EXP(-1):7:4,EXP(LN(100):6:1)
```

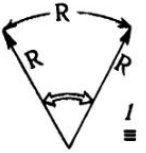
```
1.0 2.71828 7.5
0.3579 100.0
```

$e^{-1} = \frac{1}{e}$

$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$

ТРИГОНОМЕТРИЧЕСКИЕ ФУНКЦИИ

Ниже определяются тригонометрические функции. Аргумент любой из них может быть целым или вещественным; результат в любом случае – вещественный.

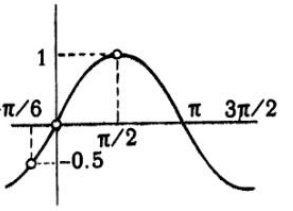


1 радиан ≡ (180/π)

SIN(*выражение*) *синус угла, измеренного в радианах*

$\sin\alpha = \frac{p}{h}$





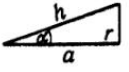

```
CONST    PI=3.1415926;
WRITELN( SIN(-PI/6):4:1);
WRITELN( SIN(0):4:1);
WRITELN( SIN(PI/2):4:1)
```

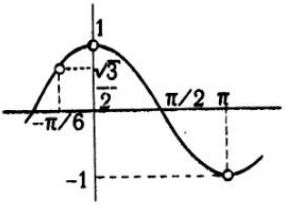
⇒

```
-0.5
0.0
1.0
```

COS(*выражение*) *косинус угла, измеренного в радианах*

$\cos\alpha = \frac{a}{h}$





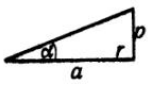

```
CONST    PI=3.1415926;
WRITELN( COS(-PI/6):4:1);
WRITELN( COS(0):4:1);
WRITELN( COS(PI):4:1)
```

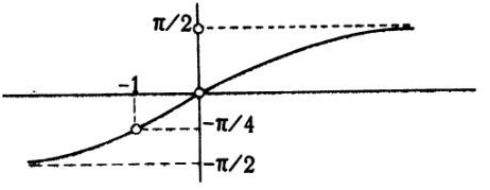
⇒

```
0.86603   √3/2
1.0
-1.0
```

ARCTAN(*выражение*) *АРКТАНГЕНС « угол в радианах, тангенс которого равен ...»*

arctan(p/a) = α *радиан*





можно считать бесконечностью

```
WRITELN( ARCTAN(1E35):8:5);
WRITELN( ARCTAN(0):4:1);
WRITELN( ARCTAN(-1):8:5)
```

⇒

```
1.57080   около π/2
0.0
-0.78540   -π/4
```

ФУНКЦИИ ПРЕОБРАЗОВАНИЯ *ИЗ ВЕЩЕСТВЕННОГО В ЦЕЛЫЙ*

Когда целое значение присваивается вещественной переменной, оно автоматически преобразуется в вещественный тип и никакие функции для этого не требуются. Такое преобразование типов называется *неявным*.

```
VAR  x,y : REAL; i,j : INTEGER;
x:= 2*3+4;   { преобразование из 10 в 10.0 }
y:= 3;       { преобразуется в 3.0 }
```

Обратного неявного преобразования нет: будет ошибкой пытаться присваивать переменной целого типа вещественный результат.

```
i:= 2.0*3+4;
j:=9/3
```

Перед присваиванием целой переменной вещественного значения это значение следует преобразовать к целому типу отбрасыванием дробной части или округлением. Для этих целей служат функции TRUNC( ) и ROUND( ) соответственно.

вещественное

TRUNC(*выражение*) *преобразует вещественное в целый тип, отбрасывая дробную часть*

```
WRITELN(TRUNC(3.1), TRUNC(3.8));
WRITELN(TRUNC(-3.1), TRUNC(-3.8));
WRITELN(TRUNC(3.0), TRUNC(-3.0))
```

```
 3   3
-3  -3
 3  -3
```

вещественное

ROUND(*выражение*) *преобразует вещественное в целый тип, округляя до ближайшего целого*

```
WRITELN(ROUND(3.1), ROUND(3.8));
WRITELN(ROUND(-3.1), ROUND(-3.8));
WRITELN(ROUND(3.0), ROUND(-3.0))
WRITELN(ROUND(3.5), ROUND(-3.5));
```

```
 3   4
-3  -4
 3  -3
 4  -4
```

Здесь возможны недоразумения. Пусть вещественная переменная x имеет значение 3.499999. Если это значение напечатать с использованием оператора WRITE(x:8:5), то получится 3.50000, в то время как WRITE(ROUND(x)) даст 3, а не 4. Это затруднение можно обойти при помощи небольшой поправки, например WRITE(ROUND(x+0.000001)) (в предположении, что значение переменной x заведомо положительное).

Применять функции TRUNC( ) и ROUND( ) к параметрам целого типа нельзя:

```
WRITELN( TRUNC(3));
WRITELN( ROUND(3))
```

```
Error
Error
```

ЛОГИЧЕСКИЕ ФУНКЦИИ

ВОЗВРАЩАЮТ ЗНАЧЕНИЕ TRUE или FALSE

Функция ODD( ) используется для проверки четности или нечетности результата целого выражения.

ODD(*выражение*) — *целое*

возвращает TRUE, если параметр – нечетный ~ в противном случае возвращает FALSE

```
WRITELN(ODD(3), ODD(2),ODD(0));
WRITELN(ODD(-3), ODD(-2));
WRITELN(ODD(3.0))
```

четное; *должно быть целым*

```
TRUE   FALSE   FALSE
TRUE   FALSE
Error
```

Следующие функции служат для определения конца строки или конца файла соответственно. Функция EOLN используется только с *текстовыми* файлами, которые организованы как строки символов. Файлы описываются в гл. 10, однако, приведенной ниже информации вполне достаточно, чтобы воспользоваться функцией EOLN. Функцию EOF не следует использовать при вводе данных с клавиатуры; этот вопрос обсуждается в гл. 11.

EOLN(*имя$_{файла}$*)

возвращает TRUE, если была прочитана последняя литера текущей строки

EOLN — *означает* EOLN(INPUT)

```
WHILE NOT EOLN DO
  BEGIN
    READ(i);
    WRITELN(i:3)
  END
```

целый тип

```
WHILE NOT EOLN DO
  BEGIN
    READ(a);
    WRITE(a:5:1)
  END
```

вещественный тип

Ввод 11 12 13 / 11 12 13 — RETURN

пробелы пропускаются

Ввод 11 12 13 / 11.0 12.0 13.0 — RETURN

EOF(*имя$_{файла}$*)

возвращает TRUE, если была прочитана последняя литера файла (попытка дальнейшего стения ведет к ошибке)

```
WHILE NOT EOF(f) DO
  BEGIN
    WHILE NOT EOLN(f) DO
      BEGIN
        READ(ch);
        WRITE(ch)
      END;
    WRITELN
  END
```

вводимый файл f

пробелы имеют значение при чтении типа CHAR

пробелы имеют значение при чтении типа CHAR

вводимый файл g

```
WHILE NOT EOF(g) DO
  BEGIN
    READ(ch); WRITE(ch)
  END
```

Признак конца строки читается как пробел

Признак конца строки читается как пробел

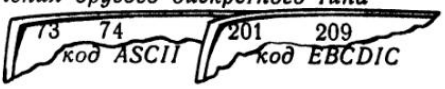
ФУНКЦИИ НАД ДИСКРЕТНЫМИ ТИПАМИ

ПОЗИЦИЯ В ПОРЯДКЕ ВОЗРАСТАНИЯ

Буквы от 'A' до 'Z' следуют в *возрастающем порядке*, иными словами, каждая буква имеет *порядковое значение*, соответствующее ее месту в алфавите. Это порядковое значение может быть получено посредством функции ORD( ):

ORD(*выражение*) *возвращает порядковый номер литеры ~ или значения другого дискретного типа*

```
WRITELN( ORD('I'), ORD('J'))
```



Порядковый номер литеры зависит от компьютера; для персональных и домашних компьютеров общепринятым является код ASCII. Но, независимо от используемого кода, порядковые значения букв следуют по возрастанию:

ORD('A') < ORD('B') < ORD('C')... < ORD('Z')

хотя ORD('Z')-ORD('A') и не обязательно равно 25. Не все компьютеры работают со строчными буквами, но если они их «понимают», то

ORD('a') < ORD('b') < ORD('c')... < ORD('z')

Определенной связи между прописными и соответствующими строчными буквами нет, но можно без опасений полагаться[4] на то, что ORD('a')-ORD('A') имеет то же значение, что и ORD('z')-ORD('Z').

Независимо от используемого кода, порядковые значения *цифр* также расположены по возрастанию:

ORD('0') < ORD('1') < ORD('2')... < ORD('9')

и, более того, порядковые значения соседних цифр отличаются на 1; так, ORD('9')-ORD('0')= 9. Отсюда следует, что численное значение цифры d (типа CHAR) может быть получено как

value:= ORD(d) - ORD('0')

(Пожалуй, *нет* такой версии Паскаля, где бы ORD('0') возвращала нуль.)

Паскаль поддерживает типы CHAR, INTEGER и т.п. В дополнение к ним программист вправе определить и другие типы путем перечисления последовательности констант:

```
TYPE
days = (mon,tue,wed,thu,fri,sat,sun);
```

тип, заданный перечислением, рассматривается на с. 82

Константы типа, заданного перечислением, имеют порядковые значения, отсчитываемые от нуля. Например, ORD(mon) возвращает 0, ORD(sun) возвращает 6; mon<sun.

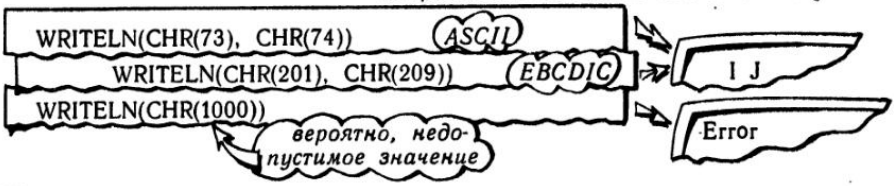
Тип BOOLEAN – перечисляемый тип, который автоматически задается как

```
TYPE
 BOOLEAN = (FALSE,TRUE);
```

следовательно, ORD(FALSE) дает 0, ORD(TRUE) дает 1; FALSE < TRUE.

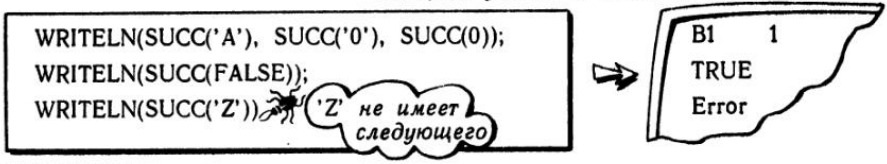
Обратной для ORD( ) является функция CHR( ):

CHR(*выражение*) *возвращает литеру, порядковое значение которой задается параметром ~ неправильное значение влечет ошибку*

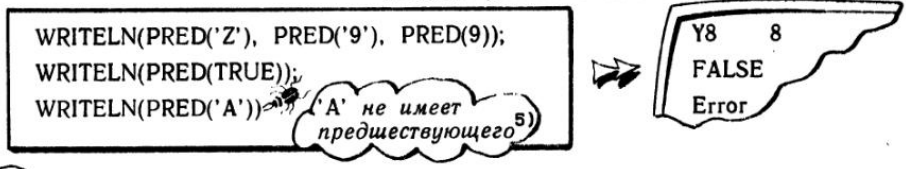


Порядковые значения ~ даже если они существуют ~ редко бывают нужны сами по себе. Часто достаточно знать следующий или предыдущий элемент в установленном порядке. Для этой цели служат функции SUCC( ) и PRED( ):

SUCC(*выражение*) *возвращает элемент, следующий за тем, который указан в качестве параметра*



PRED(*выражение*) *возвращает элемент, предшествующий тому, который указан в качестве параметра*



Эти две функции можно использовать для определения следующих и предшествующих элементов для типа, заданного перечислением. Возьмем тип *days*, определенный напротив:

PRED(sun) *возвращает* sat, SUCC(mon) *возвращает* tue

Однако было бы неверно писать WRITELN(PRED(sun)), поскольку элементы перечисляемого типа нельзя читать или печатать ~ что, конечно, снижает выгоду от использования таких типов. Наилучшее приближение к WRITELN(PRED(sun)) – это оператор WRITELN(ORD(PRED(sun))), печатающий число 5 (порядковое значение элемента sat).

Функцию SUCC( ) удобно использовать для управления циклом:

```
i:=0;
REPEAT
  i:=SUCC(i);
  операторы
UNTIL i = 10
```

ПРИМЕЧАНИЯ РЕДАКТОРА

1) (с. 42) Выражение 17 **MOD** –5 является ошибочным (в некоторых компиляторах) не потому, что операция **MOD** не допускает отрицательных аргументов, а потому, что синтаксис Паскаля запрещает располагать подряд две операции. В равной мере будет ошибочным выражение 17 * –5. Если же записать выражение так: 17 **MOD** (–5), то сообщения об ошибке не будет и в результате получится число 2. Впрочем, версия Турбо Паскаль допускает и выражение 17 **MOD** –5 , и 17 * –5 .

2) (с. 42) Обратите внимание на тонкий момент: в каком порядке выполняются операции в выражениях типа –17 **DIV** 5 ? Операция «–» формально имеет более низкий приоритет, чем «*», следовательно, выражение –17 **DIV** 5 должно трактоваться, как –(17 **DIV** 5). Именно так работает компилятор на больших ЭВМ серии ЕС. Вместе с тем, в версии Турбо Паскаль приоритет *унарного* минуса (т.е. минуса, перед которым нет операнда) считается выше приоритета любой операции, поэтому то же выражение обрабатывается как (–17) **DIV** 5 . Оба варианта дают одинаковый результат, разница между ними может проявиться в случае переполнения.

3) (с. 44) Замена i * j **DIV** k на i * (j **DIV** k) не только уменьшает риск переполнения, но и изменяет результат выражения, следовательно, такая замена далеко не всегда допустима.

4) (с. 50) К сожалению, русские буквы, даже если они есть на компьютере, вовсе не обязательно расположены по алфавиту. Самая лучшая в этом смысле ситуация – на ПЭВМ ЕС-1840 при работе в операционной системе М86, поставляемой заводом изготовителем. Здесь русские буквы расположены по алфавиту, без промежутков и разница между прописными и соответствующим строчными буквами постоянна. На той же ПЭВМ и других, совместимых с IBM PC, при работе в системе MS DOS обычно используется кодировка (и не одна), в которой русские буквы (кроме ё) расположены по алфавиту, но разность строчных и прописных букв неодинакова для разных букв. На больших же компьютерах серии ЕС порядок русских букв (А, Б, Ц, Д, Е, Ф, Г, Х,...) совсем не алфавитный.

5) (с. 51) В коде ASCII PRED('A'), как и SUCC('Z'), определены:
PRED('A')='@'; SUCC('Z')='['

5

УПРАВЛЕНИЕ

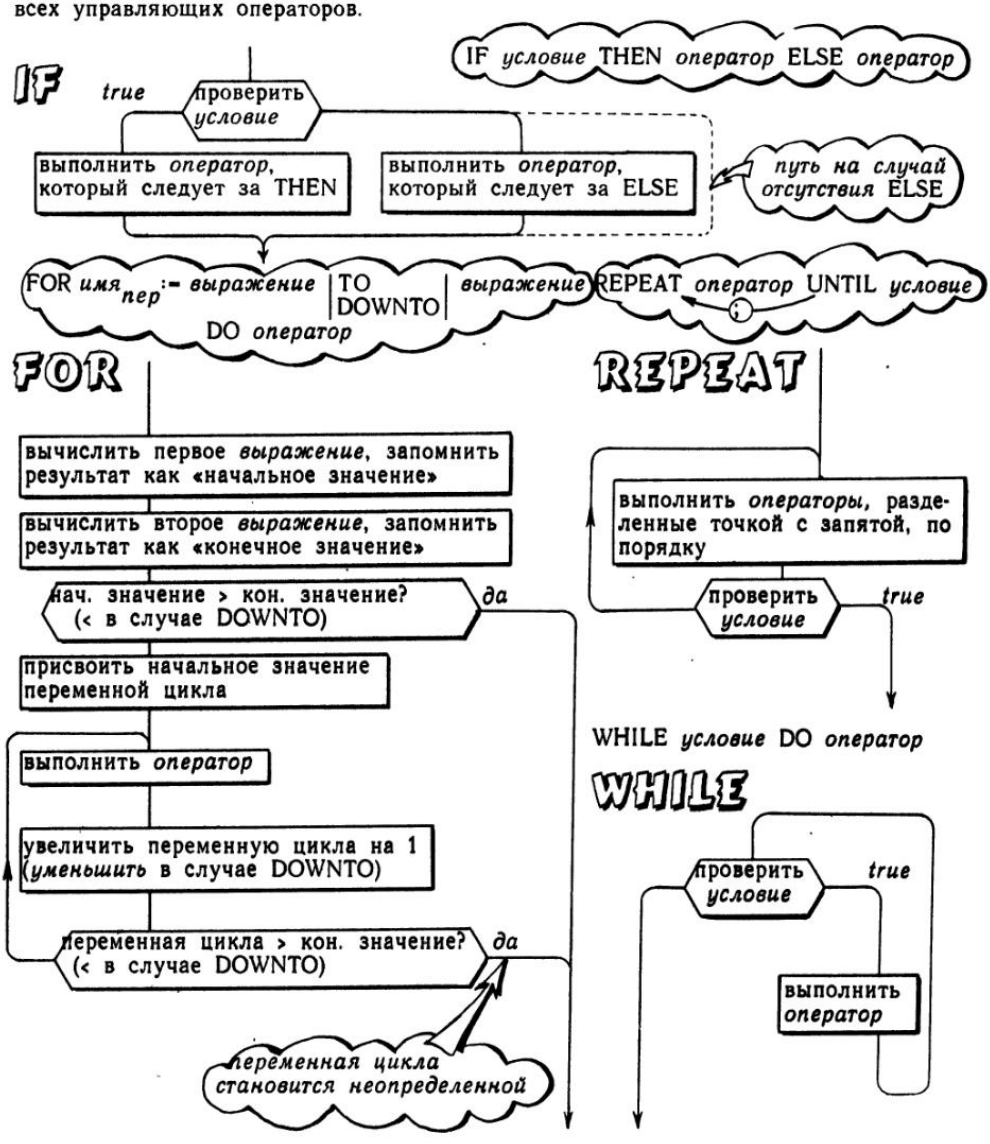
БЛОК-СХЕМЫ
ОПЕРАТОР IF-THEN-ELSE
ЦИКЛ FOR
ЦИКЛ REPEAT
ЦИКЛ WHILE
ФИЛЬТР (ПРИМЕР)
ОПЕРАТОР CASE
АВТОМАТНЫЙ РАСПОЗНАВАТЕЛЬ (ПРИМЕР)

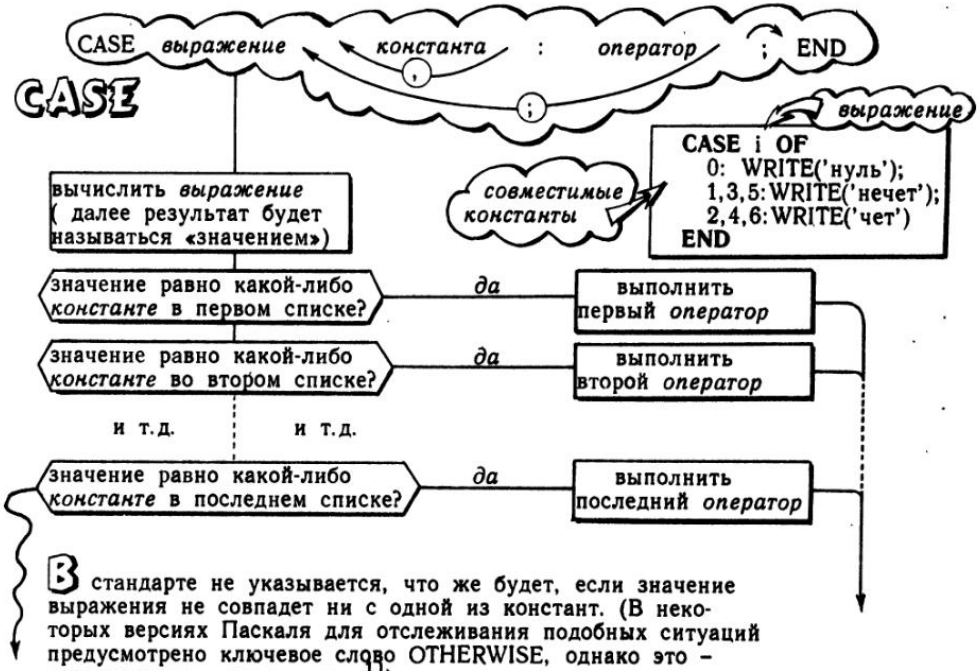
БЛОК-СХЕМЫ

ОПЕРАТОР IF-THEN-ELSE, ЦИКЛ FOR, ЦИКЛ REPEAT, ЦИКЛ WHILE, ОПЕРАТОР CASE, ОПЕРАТОР GOTO

Большинство управляющих операторов уже использовались в примерах из предыдущих глав. В этой главе даются точные определения и обсуждаются особенности. Только эти операторы способны изменить порядок выполнения программы, без них управление передается от оператора к оператору последовательно.

На этом развороте на рисунках в виде блок-схем изображено функционирование всех управляющих операторов.





В стандарте не указывается, что же будет, если значение выражения не совпадет ни с одной из констант. (В некоторых версиях Паскаля для отслеживания подобных ситуаций предусмотрено ключевое слово OTHERWISE, однако это – нестандартная возможность.[1])



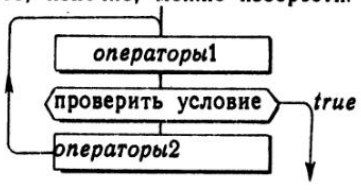
В некоторых версиях Паскаля GOTO и помеченный оператор должны обязательно находится в одном программном блоке.

Оператор GOTO полезен в интерактивных системах для исправления ошибок пользователя ~ тема, обсуждение которой выходит за рамки этой книги.

EXIT?

ПАСКАЛЮ НЕДОСТАЕТ ОПЕРАТОРА EXIT

Стандартным Паскалем не предусмотрен выход из середины цикла*). Хотя какой-то способ, конечно, можно изобрести:





*) не считая GOTO

ОПЕРАТОР IF-THEN-ELSE

Синтаксис оператора:

IF *условие* **THEN** *оператор* **ELSE** *оператор*

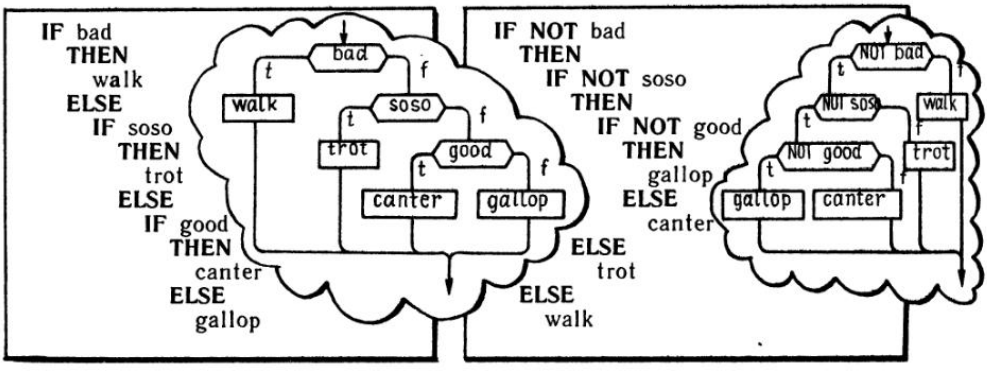
- **IF** profit > loss **THEN** WRITE('Ура!')
- **IF** profit > loss **THEN** WRITE('Ура!') **ELSE** WRITE('Увы...')
- **IF** initial>'k' **AND** initial<'s' **THEN** WRITE('Смотри в каталоге между L и R')

Когда *условие* вычислено и его значение оказалось *true*, выполняется оператор, следующий за **THEN** ~ оператор, следующий за **ELSE**, игнорируется. Если, наоборот, значение условия оказалось *false*, то игнорируется оператор, который следует за **THEN** ~ а выполнен будет оператор, следующий за **ELSE** (если такой имеется).

Если вычисление *условия* не дает ни *true* ни *false*, то выдается сообщение об ошибке.

Оператор, следующий за **THEN** или **ELSE**, может быть составным оператором (т.е. иметь вид **BEGIN**... ...**END**). Нет никаких ограничений на число или сложность операторов, входящих в составной оператор.

Будьте внимательны при использовании вложенных операторов **IF**. Предпочтительнее пользоваться схемой **ELSE-IF**, нежели **THEN-IF**, заставляющей «хранить в уме» соответствующие **ELSE**. Злоупотребление **THEN-IF** обычно заканчивается неприятным нагромождением закрывающих **ELSE**:



Общее правило таково: каждый **ELSE** относится к ближайшему предшествующему **IF**, еще не имеющему парного **ELSE**.

ЦИКЛ FOR

ИСПОЛЬЗУЕТСЯ, ЕСЛИ ВЫ МОЖЕТЕ УКАЗАТЬ В ЗАГОЛОВКЕ ЧИСЛО ПОВТОРЕНИЙ

Синтаксис оператора FOR:

FOR *имя* := *выражение* | TO / DOWNTO | *выражение* **DO** *оператор*

имя переменной цикла

- **FOR** humbug:= 1 **TO** 3 **DO** WRITELN('Мы желаем Вам веселого Рождества'); WRITELN('И счастливого Нового года')
- **FOR** m:= 12 **DOWNTO** 2 **DO** WRITELN(m:3,' человек,'); WRITELN('1 человек и его собака пошли косить луг')

Переменная цикла может быть любого упорядоченного типа (обычно – типа INTEGER и ни в коем случае не REAL). Оба *выражения* должны быть того же типа, что и переменная цикла.

Блок-схему со с. 54 неплохо подкрепить примерами, которые и приведены ниже.

Оба выражения вычисляются перед выполнением операторов цикла; впоследствии они не перевычисляются. Если эти выражения отвечают невозможной последовательности, то цикл вообще не выполняется:

```
FOR i:= 2 TO 1 DO WRITE('Робкий')
```

Ничего не печатается ~ сообщения об ошибке нет

Невозможно выскочить за «финишную черту», которая установлена в самом начале:

```
finish:=3;
FOR i:=3 TO finish DO
  BEGIN
    finish:=finish+1;
    WRITELN(finish)
  END
```

установлено конечное значение 3

цикл работает ровно три раза

4
5
6

Всякое изменение переменной цикла является ошибкой. Среди таких некорректных изменений – присваивание переменной цикла и чтение в нее значений:

```
FOR i:= 1 TO 3 DO
  BEGIN
    i:=i-1;
    READ(i);
    FOR i:=1 TO 3 DO WRITE('Боже мой!')
  END
```

Будет неверно делать какие-либо предположения о значении переменной цикла FOR по выходе из цикла (если только выход не через GOTO):

```
FOR i:= 1 TO 3 DO
  WRITE(i:4);
WRITELN;
WRITE(i:4)
```

может быть что угодно

1 2 3

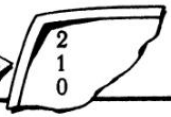
цикл REPEAT



Синтаксис оператора REPEAT:

```
REPEAT  оператор  UNTIL  условие
```

```
▸ n:=3;  REPEAT  n:=PRED(n); WRITELN(n) UNTIL n=0;
```



Из блок-схемы видно, что операторы выполняются по крайней мере один раз. Если при определенных условиях цикл должен быть пропущен, то следует предусмотреть специальные меры ~ скажем, воспользоваться оператором **IF**. В такой ситуации лучше использовать цикл **WHILE**.

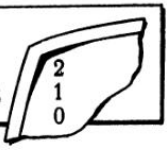
цикл WHILE



Синтаксис оператора WHILE:

```
WHILE  условие  DO  оператор
```

```
▸ n:=3;  WHILE n>0 DO BEGIN n:=PRED(n); WRITELN(n) END;
```



Как следует из блок-схемы, проверка условия производится перед выполнением *оператора*, что позволяет пропустить цикл в случае невыполнения условия (в отличие от цикла REPEAT)

Типичное использование цикла WHILE – при копировании текстовых файлов. Текстовый файл – это файл, организованный в строки и состоящий из некоторых элементов, разделенных пробелами, как описано на с. 125.

```
VAR  ch: CHAR;

WHILE NOT EOF(f) DO
  BEGIN
    WHILE NOT EOLN(f) DO
      BEGIN
        READ(f,ch);
        WRITE(ch)
      END;
    WRITELN
  END;
```







Не используйте EOF при вводе с клавиатуры. Подробнее об этом пойдет речь в гл. 10 и 11.

ФИЛЬТР

НА ПРИМЕРЕ ПРОГРАММЫ ДЛЯ ЧТЕНИЯ ИЗ ТЕКСТА МАЛЕНЬКИХ ЧИСЕЛ ИЛЛЮСТРИРУЮТСЯ ЦИКЛЫ REPEAT *И* WHILE

Чтобы прочитать числа из приведенного ниже файла недостаточно лишь операторов READ – встречающиеся *на пути слова и знаки пунктуации* препятствуют этому. Программа filter предназначена для считывания именно чисел и отсеивания прочих данных.

Здесь изображен файл исходных данных. Его следует вводить не нажимая клавишу RETURN до самой последней точки.

За 6 месяцев, если повезет, у меня будет 350 фунтов +46.47 дохода -8.12 налоги. Этого должно хватить для приобретения домашнего компьютера МК2 с памятью 32К.

Это – выходной файл, который должна создать программа из изображенного выше входного файла.

```
 6.00
350.00
 46.47
 -8.12
  2.00
 32.00
```

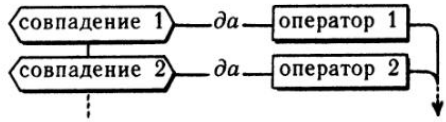
А вот и программа, выполняющая эту работу:

```
PROGRAM filter(INPUT,OUTPUT);
  VAR
    ch,sgn: CHAR;  fraction: INTEGER;  number: REAL;
BEGIN
  ch:=' ';                                  { пробел }
  WHILE NOT EOLN DO
    BEGIN
      number:=0.0;  fraction:=0;
      sgn:=ch;  READ(ch);
      IF (ch>='0') AND (ch<='9')
        THEN
          BEGIN          { если цифра }
            REPEAT
              REPEAT
                number:=10*number+ORD(ch)-ORD('0');
                fraction:=fraction*10;
                IF NOT EOLN THEN READ(ch);
              UNTIL((ch<'0') OR (ch>'9') OR EOLN;
              IF (ch='.') AND NOT EOLN
                THEN
                  BEGIN
                    READ(ch);  fraction:=1;
                  END
            UNTIL((ch<'0') OR (ch>'9') OR EOLN;
            IF fraction>0 THEN number:=number/fraction;
            IF sgn='-' THEN number:=-number;
            WRITELN(number:8:2);
          END      { если цифра }
    END    { WHILE NOT EOLN }
END.
```

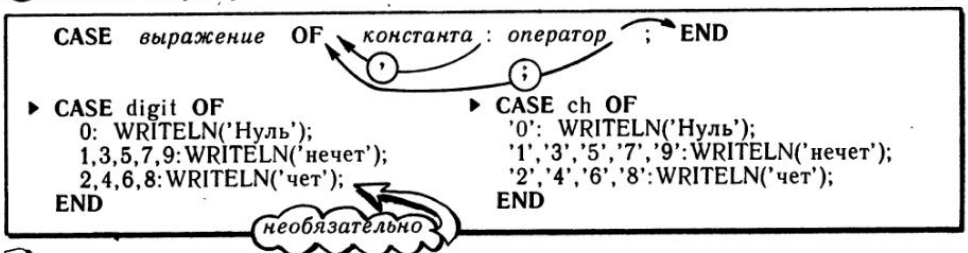
недостаток: если в числе более одной десятичной точки, то учтена будет только последняя; например, 12.3.4 даст 123.4 без сообщения об ошибке

Любой фрагмент программы, имеющий дело с вводом, трудно сделать абсолютно неуязвимым ввиду огромного множества потенциально необходимых проверок. В этом отношении приведенная программа особенно плоха. Лучшая версия этой программы, в которой использованы пока не введенные в рассмотрение возможности Паскаля, приводится на с. 86.

ОПЕРАТОР CASE



Синтаксис оператора CASE:




```
▸ CASE digit OF
    0: WRITELN('Нуль');
    1,3,5,7,9:WRITELN('нечет');
    2,4,6,8:WRITELN('чет');
  END
```

```
▸ CASE ch OF
    '0': WRITELN('Нуль');
    '1','3','5','7','9':WRITELN('нечет');
    '2','4','6','8':WRITELN('чет');
  END
```

Выражение может давать значение любого упорядоченного типа, как правило, – это INTEGER или CHAR и ни в коем случае REAL. *Выражение* и *константы* должны принадлежать одному типу.

Действие оператора CASE определено блок-схемой на с. 55. Как только обнаруживается совпадение, выполняется соответствующий оператор; ни один из других операторов не выполняется. Если совпадений нет вообще, то результат непредсказуем. Поэтому будьте внимательны и постарайтесь предусмотреть все значения, которые может принять выражение (что не всегда просто).

Можно использовать и вложенные операторы CASE, это удобно, например, при реализации *автоматных распознавателей*, которые дают способ наглядной записи алгоритмов распознавания текстов Представленная таблица предназначена для перевода римских чисел, составленных из цифр X, V, I.

состояние \ символ		'X'	'V'	'I'
⇨	1	n:=10; state:=2	n:=5; state:=3	n:=1; state:=6
	2	n:=n+10; state:=2	n:=n+5; state:=3	n:=n+1; state:=6
	3	ok:=FALSE	ok:=FALSE	n:=n+1; state:=4
	4	ok:=FALSE	ok:=FALSE	n:=n+1; state:=5
	5	ok:=FALSE	ok:=FALSE	n:=n+1; state:=7
	6	n:=n+8; state:=7	n:=n+3; state:=7	n:=n+1; state:=5
	7	ok:=FALSE	ok:=FALSE	ok:=FALSE

Для расшифровки XIV начинаем с состояния 1, как указано стрелкой. Первый символ – 'X', поэтому смотрим столбец 'X' и находим n:=10; state:=2. Итак, полагаем n равным 10 и сдвигаем стрелку на вторую строку. Теперь смотрим столбец, определяемый вторым символом, т.е. 'I', и находим n:=n+1; state:=6. Значение n, таким образом, становится 10+1=11. Сдвигаем стрелку к строке 6. Теперь в столбце 'V' находим n:=n+3; state:=7. Значение n становится равным 11+3=14. Сдвигаем стрелку на строку 7 и замечаем, что любая следующая цифра 'X', 'V' или 'I' будет теперь ошибкой (например, XIVX).

Эта таблица позволяет декодировать римскую запись чисел, содержащих любое количество цифр X (в начале) и цифры V, I, записанные по обычным правилам:

I, II, III, IV, V, VI, VII, VIII, IX, X, XI и т.д.

Вместе с тем такое число как IIII, будет воспринято как ошибочное и переменная ok примет значение FALSE. Для работы с цифрами M, D, C и L таблицу можно расширить.

АВТОМАТНЫЙ РАСПОЗНАВАТЕЛЬ

ИЛЛЮСТРИРУЕТ ВЛОЖЕННЫЕ ОПЕРАТОРЫ CASE

```
PROGRAM  roman(INPUT,OUTPUT);

  VAR  n,state: INTEGER; symbol: CHAR; ok: BOOLEAN;

BEGIN   { программа }

   state:=1;  ok:=TRUE;  n:=0;
   WHILE NOT EOLN DO
    BEGIN
      READ(symbol);
      IF ((symbol='X') OR (symbol='V') OR (symbol='I')
        THEN
          CASE state OF
           1: CASE symbol OF
                'X': BEGIN n:=10; state:=2  END;
                'V': BEGIN n:=5;  state:=3  END;
                'I': BEGIN n:=1;  state:=6  END
              END;
           2: CASE symbol OF
                'X': BEGIN n:=10;   state:=2  END;
                'V': BEGIN n:=n+5;  state:=3  END;
                'I': BEGIN n:=n+1;  state:=6  END
              END;
           3: CASE symbol OF
                'X','V': ok:=FALSE;
                'I': BEGIN n:=n+1;  state:=4  END
              END;
           4: CASE symbol OF
                'X','V': ok:=FALSE;
                'I': BEGIN n:=n+1;  state:=5  END
              END;
           5: CASE symbol OF
                'X','V': ok:=FALSE;
                'I': BEGIN n:=n+1;  state:=7  END
              END;
           6: CASE symbol OF
                'X': BEGIN n:=n+8;  state:=7  END;
                'V': BEGIN n:=n+3;  state:=7  END;
                'I': BEGIN n:=n+1;  state:=5  END
              END;
           7: ok:=FALSE;
          END     { CASE state }
        ELSE
          BEGIN
            IF ok
              THEN WRITELN(n:2)
              ELSE WRITELN('PECCAVISTI');
            state:=1;  ok:=TRUE
          END   { ELSE }
    END { WHILE NOT }
END.   { программы }
```

Приятнее писать:
IF symbol **IN** ['X','V','I'],
см. гл. 7

декодированное число – в n

Замечание:
перед нажатием RETURN
следует ввести точку или пробел

```
XIV XIVX XX.
14
PECCAVISTI
20
```
пробел

УПРАЖНЕНИЯ

1. Выполните программу roman. Дополните программу так, чтобы она справлялась с цифрами:

M - 1000, D - 500, C - 100, L - 50 .

Если ваш Паскаль допускает интерактивную работу, то включите в программу сообщения-подсказки для удобства пользователя.

ПРИМЕЧАНИЯ РЕДАКТОРА

[1] (с. 55) В версии Турбо Паскаль в операторе **CASE** можно использовать слово **ELSE**, после которого записывается оператор, исполняемый, если значение *выражения* не совпадает ни с одной из *меток*.

6

ФУНКЦИИ И ПРОЦЕДУРЫ

ОПРЕДЕЛЕНИЕ ФУНКЦИИ
ПРИМЕРЫ ФУНКЦИЙ
РЕКУРСИЯ
ПРОЦЕДУРЫ
СЛУЧАЙНЫЕ ЧИСЛА
СНОВА ЗАЕМ (ПРИМЕР)
ИМЕНА ФУНКЦИЙ КАК ПАРАМЕТРЫ
ССЫЛКИ ВПЕРЕД
ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ
ПОБОЧНЫЕ ЭФФЕКТЫ
ПРАВИЛА ВИДИМОСТИ

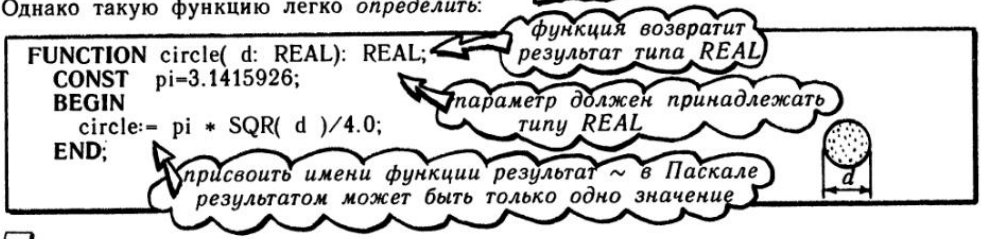
ОПРЕДЕЛЕНИЕ ФУНКЦИИ

ОПРЕДЕЛЯЙТЕ СВОИ СОБСТВЕННЫЕ ФУНКЦИИ

Паскаль сам по себе *не предоставляет* функции, вычисляющей площадь круга по фактическому параметру – диаметру круга.

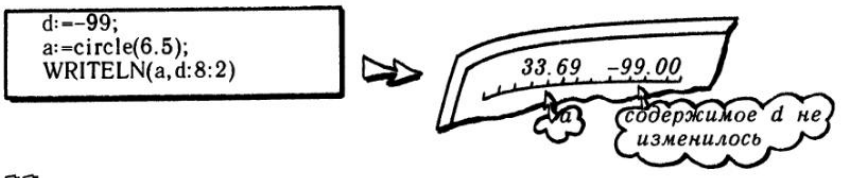


Однако такую функцию легко *определить*:



Теперь функцию circle( ) (или CIRCLE( )) можно использовать в программе точно так же, как ранее использовались функции SQR( ) или TRUNC( ).

В первой строке определения функции параметр *d* как бы говорит «Делай то, что делается со мной, но используй значение, которое будет на моем месте». В примере в начале этой страницы на место *d* ставится число 6.5, которое, таким образом, возводится в квадрат, затем результат умножается на 3.1415926 и делится на 4.0. Параметр *d* – это *формальный параметр*, тогда как число 6.5 – *фактический параметр*. В программе, обращающейся к функции circle( ), можно использовать имя *d* как имя переменной (или любого другого объекта), не опасаясь помех со стороны функции.



Ниже приведен синтаксис определения функции (пока без учета случая, когда параметры сами являются функциями или процедурами):



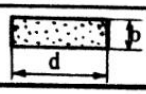
Элемент *блок* имеет структуру программы в программе. Синтаксис *блока* определен на с. 38; эта схема всего лишь иллюстрирует расположение определений функций и процедур в программе.



Определения функций и процедур могут в свою очередь сами содержать вложенные в них определения функций и процедур.

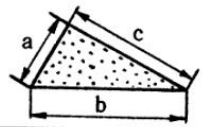
Ниже приведена функция, вычисляющая площадь прямоугольника; ее параметрами являются длины сторон:

```
FUNCTION rectangle( b,d: REAL): REAL;
  BEGIN rectangle := b*d   END;
```



А вот похожая функция для вычисления площади тругольника:

```
FUNCTION triangle( a,b,c: REAL): REAL;
  VAR     x: REAL;
   BEGIN
     x:= (a+b+c)/2;
     triangle := SQRT(x*(x-a)*(x-b)*(x-c))
   END
```



Все эти три функции (circle( ), rectangle( ), triangle( )) могут быть вызваны из следующей программы, которая представляет собой переработку программы со с. 27.

```
PROGRAM shapes2(INPUT, OUTPUT);
  VAR  letter: CHAR; a,x,y: REAL;
{ здесь поместите три функции в любом порядке }
BEGIN
  REPEAT
    READ(letter);
    CASE letter OF
      'в', 'В' : a:=0;

      'п', 'П' :   BEGIN
                     READLN(x,y); a:=rectangle(x,y)
                   END

      'т', 'Т' :   BEGIN
                     READLN(x,y,z); a:=triangle(x,y,z)
                   END

      'к', 'К' :   BEGIN
                     READLN(x); a:=circle(x)
                   END
    END;   { CASE letter }
    WRITELN('Площадь: ',a:8:2)
  UNTIL  (letter = 'q') OR (letter = 'Q')
END.
```

```
П 3.5 2
Площадь:  7.00
К 6.5
Площадь:  33.69
Т 3 4 5
Площадь:  6.00
Все
Площадь:  0.00
```

Обратите внимание на то, что функции вызываются с *фактическими* параметрами *x, y, z*, тогда как *формальные* параметры в их определениях – *a, b, c, d*. Переменная *a* в основной программе никак не связана с формальным параметром *a* в функции *triangle(, ,)*. Точно так же отсутствует связь переменной *x* в основной программе и локальной переменной *x* в функции *triangle(, ,)*. Подробнее об этом – несколько позже.

Определенные здесь функции различаются по числу параметров. В функциях может быть любое фиксированное число параметров; определить функцию с переменным числом параметров (например, как в случае READ(a), READ(a,b), READ(a,b,c)) нельзя – такой возможностью пользуется только сам Паскаль.

В приведенных примерах все типы – REAL, однако, допускается и любое сочетание: например mixfun(a: REAL; b: INTEGER; c: CHAR): BOOLEAN;

ПРИМЕРЫ ФУНКЦИЙ

ДЛЯ ИЛЛЮСТРАЦИИ ОПРЕДЕЛЕНИЙ ФУНКЦИИ

В Паскале нет функции, для вычисления кубического корня. Вот ее определение:

```
FUNCTION cubrt(x: REAL): REAL;
 VAR  old, noo: REAL;
  BEGIN
    IF x=0 THEN cubrt:=0 ELSE
     BEGIN  old:=1;
      REPEAT
        noo:=x/SQR(old);
        old:=(noo+old)/2
       UNTIL ABS(x/(noo*noo*noo)-1)<1E-6 ;
       cubrt:= noo;
  END
END;  { функции }
```

выход, когда $\frac{x}{(noo)^3} \approx 1$

cubrt(-27) возвращает -3
cubrt(0) возвращает 0
cubrt(27) возвращает 3

Программирующие на Бейсике и сожалеющие об отсутствии функции SGN( ), могут определить ее либо «в лоб»:

```
FUNCTION sgn(x: REAL): INTEGER;
  BEGIN
    IF x>0 THEN sgn:=1 ELSE
    IF x<0 THEN sgn:=-1 ELSE sgn:=0
  END;
```

возвращает 1, если x > 0
возвращает -1, если x < 0
возвращает 0, если x = 0

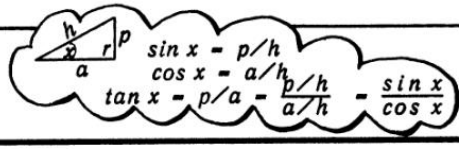
либо хитрее:

```
FUNCTION sgn(x: REAL): INTEGER;
  BEGIN  sgn:=ORD(x>0) - ORD(x<0) END;
```

работает потому, что ORD(TRUE)=1, ORD(FALSE)=0

В Паскале нет функции TAN( ) (тангенс угла, измеренного в радианах). Вот ее определение:

```
FUNCTION tan(x: REAL): REAL;
  BEGIN
    tan:= sin(x)/cos(x)
  END;
```



Далее приведены функции для вычисления арксинуса (угла в радианах, синус которого равен ...) и арккосинуса:

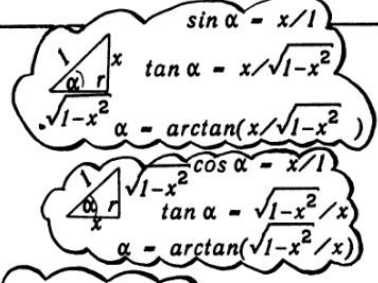
```
FUNCTION arcsin(x: REAL): REAL;
   BEGIN
    IF ABS(x)=1
     THEN arcsin:=x*1.5707963
     ELSE arcsin:=ARCTAN(x/SQRT(1-SQR(x)))
   END;

FUNCTION arccos(x: REAL): REAL;
   BEGIN
    IF x=0
     THEN arccos:=1.5707963
     ELSE arccos:=ARCTAN(SQRT(1-SQR(x))/x) + 3.1415926*ORD(x<0)
   END;
```

±π/2

+π/2

+π, когда x≤0



Функции arcsin( ) и arccos( ) рассматриваются также на с. 78.

РЕКУРСИЯ

ОПРЕДЕЛЕНИЕ РЕКУРСИВНОЙ ФУНКЦИИ ПОМОГАЕТ ПОНЯТЬ ИДЕЮ РЕКУРСИИ

Наибольший общий делитель (НОД) чисел 1470 и 693 – это 21. Другими словами, 21 – наибольшее число, на которое и 1470 и 693 делятся без остатка. Чтобы убедиться в этом, разложим оба числа на простые сомножители:

1470 = 2 × 3 × 5 × 7 × 7

693 = 3 × 3 × 7 × 11

и выделим пары общих сомножителей ~ в данном случае это пары чисел 3 и 7. Наибольший общий делитель – это произведение совпадающих сомножителей; в данном случае это 3*7=21.

Более изящный метод поиска НОД – алгоритм Евклида. Найдем остаток от деления 1470 на 693:

```
1470 MOD 693  =  84
```

Так как этот остаток не равен нулю, повторим то же действие, подставив вместо первого числа второе, а вместо второго – остаток:

```
693 MOD 84  =  21
```

Этот остаток также не нуль, поэтому еще одно деление:

```
84 MOD 21  =  0
```

Теперь остаток – нуль, следовательно, НОД равен 21. Вот и отлично.

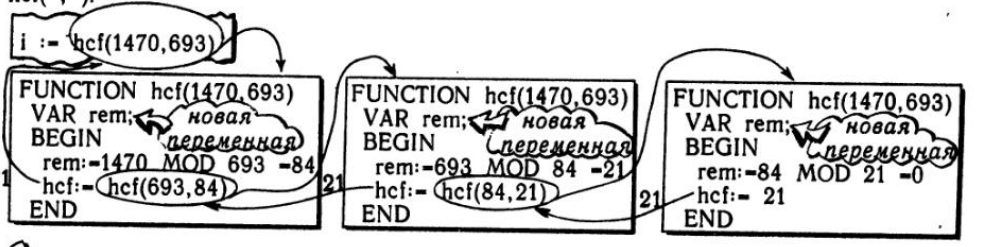
Следующая функция на Паскале использует метод Эвклида:

```
FUNCTION  hcf(n,m: INTEGER): INTEGER;
  VAR rem: INTEGER;
  BEGIN
    rem := n MOD m;
    IF rem=0 THEN hcf:=m ELSE hcf:=hcf(m,rem)
  END;
```

рекурсивное обращение

работает правильно как для n≥m так и для n<m

Сразу ясно, как вычисляется hcf(84,21): остаток rem будет равен нулю и функция возвратит число 21. При обращении hcf(1470,693) rem будет равен 84, и, следовательно, функция вызовет сама себя как hcf(693,84). Теперь уже rem примет значение 21, и функция вызовет себя еще раз в виде hcf(84,21). Таким образом, при каждом обращении Паскаль как бы создает новую копию функции hcf(,):



Способность функции обращаться к себе самой называется *рекурсией*. Подробнее о рекурсии говорится в этой и следующих главах.

ПРОЦЕДУРЫ

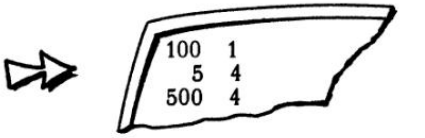
И ОТЛИЧИЕ ПАРАМЕТРОВ-ЗНАЧЕНИЙ ОТ ПАРАМЕТРОВ-ПЕРЕМЕННЫХ

Когда какая-либо часть программы используется более одного раза, то вовсе не обязательно повторять текст; эту часть можно оформить в виде процедуры, дав этой ей имя и вызывая каждый раз, когда необходимо выполнить эту часть программы. Вот простейший пример: процедура печати двух целых чисел в обратном порядке:

```
PROCEDURE reverse(a,b: INTEGER);
 BEGIN
  WRITELN(b:3,a:3)
 END;
```

Из основной программы эта процедура может быть вызвана так:

```
x:=1; y:=100;
reverse(x,y);
reverse(4,5);
reverse(4*x,5*y)
```



Глупо, конечно, использовать такую процедуру, но на этом примере видно, что фактические параметры могут быть константами (4,5), выражениями (4*x,5*y) или именами переменных (x,y). Каждый раз при вызове процедуры reverse(,) вычисляются фактические параметры, и их *значения* подставляются на место формальных параметров *a* и *b*. По этой причине *a* и *b* называются параметрами-значениями.

Предположим, что вместо печати значений в обратном порядке, необходимо поменять местами значения двух переменных целого типа. Приведенная ниже процедура для этого совсем непригодна:

```
PROCEDURE swop(a,b: INTEGER);
  VAR
    tempry: INTEGER;
  BEGIN
    tempry:=a; a:=b; b:=tempry
  END;
```



Предположим, процедура вызывается следующим образом, причем значения переменных x и y равны соответственно 1 и 100:

```
swop(x,y)
```

Далее произойдет вот что: значение 1 будет помещено в *a*, 100 – в *b*; затем значения *a* и *b* поменяются; после чего произойдет возврат в программу. Переменные x и y останутся нетронутыми. Процедура работает только лишь со *значениями* своих параметров; обращения swop(4,5) или swop(4*x,5*y) равным образом не дадут результата.

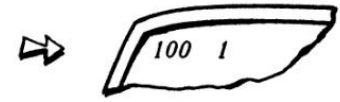
Выход в том, чтобы описать параметры как параметры-переменные. Если перед именем параметра записать слово **VAR**, то процедура получит доступ к переменной в вызывающей программе:

```
PROCEDURE swop( VAR a,b: INTEGER);
  VAR
    tempry: INTEGER;
  BEGIN
    tempry:=a; a:=b; b:=tempry
  END;
```

теперь вы сможете изменить значения переменных, относящихся к вызывающей программе

Теперь, вызывая процедуру, получаем:

```
x:=1;  y:=100;
swop(x,y);
WRITELN(x,y)
```



Попросту говоря, *пишите* **VAR** *перед теми параметрами, значения которых должны быть изменены процедурой.*

Более строго, наличие **VAR** в заголовке процедуры означает прямую связь с вызывающей программой. Оператор a:=b в процедуре означает x:=y в вызывающей программе (речь идет о последнем примере). На жаргоне: параметры-переменные *передаются по адресу* или *по ссылке*, тогда как параметры-значения передаются *по значению* ~ процедура для хранения каждого переданного значения создает локальную переменную.

Вызывать процедуру с параметрами-значениями так, как показано ниже – бессмысленно. Вызов процедуры имеет смысл, только если оба параметра – имена переменных, значения которых следует поменять местами.

```
swop(4,5);
swop(4*x,5*y)
```

Здесь возможно недоразумение: раздел **VAR** в процедуре служит для определения *локальных* для данной процедуры пременных, тогда как слово **VAR** в заголовке процедуры означает ссылку на *нелокальные* переменные:

```
PROCEDURE swop(VAR a,b: INTEGER);
  VAR tempry: INTEGER;
BEGIN
  tempry:=a; a:=b; b:=tempry
END;
```

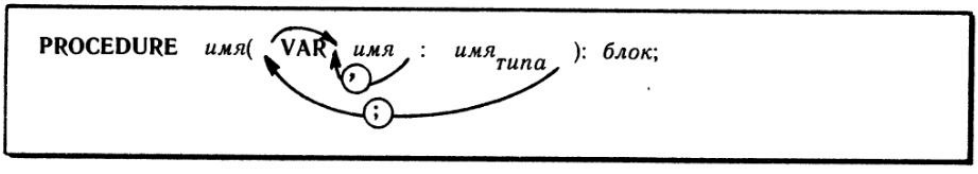
нелокальные

локальная

```
PROCEDURE reverse(a,b: INTEGER);
BEGIN
  WRITELN(b,a)
END;
```

локальные переменные

Далее приведен синтаксис определения процедуры (пока не учитываются параметры, являющиеся в свою очередь именами функций).



Элемент *блок* имеет структуру программы внутри программы. Синтаксис *блока* детально изложен на с. 38

Эта схема просто иллюстрирует расположение определений функций и процедур в программе. Определения функций и процедур могут в свою очередь сами содержать вложенные определения функций и процедур.



СЛУЧАЙНЫЕ ЧИСЛА *ФУНКЦИЯ, ВОЗВРАЩАЮЩАЯ ЗНАЧЕНИЕ И МЕНЯЮЩАЯ ПАРАМЕТР*

Рассмотрим следующую функцию:

```
FUNCTION next(VAR seed: INTEGER): INTEGER;
  CONST multiplier=37; increment=3;  cycle=64;
  BEGIN
    next:=seed;
    seed:=(multiplier*seed+increment) MOD cycle
  END;
```

обратите внимание на **VAR** *в заголовке, необычный прием для функции*

Будучи вызвана с параметром *s*, содержащим число 16

```
s:=16; WRITE(next(s))
```

⇒ 16

эта функция должна, очевидно, возвратить число 16. Вместе с тем, возвращая 16, функция изменяет значение, хранящееся в переменной *seed*, на 19. Если функцию вызвать снова, с полученным значением *s*, то она возвратит число 19 и изменит значение *s* на 2. Продолжая вызывать функцию next( ), получим определенную последовательность целых чисел, начинающуюся с исходного значения *s*:

```
s:=16;
FOR i:=1 TO 64 DO WRITE(next(s):3)
```

```
 16 19  2 13 36 55 54 17 56 27 42 21 12 63 30 25
 32 35 18 29 52  7  6 33  8 43 58 37 28 15 46 41
 48 51 34 45  4 23 22 49 24 59 10 53 44 31 62 57
  0  3 50 61 20 39 38  1 40 11 26  5 60 47 14  9
```

Замечательным свойством этой последовательности является то, что каждое значение от 0 до 63 встречается в ней ровно один раз. Более того, шестьдесят пятый вызов функции next( ) даст число 16 и начнется новый цикл. Другими словами, начиная с любого желаемого целого, функция генерирует фиксированную перестановку чисел от 0 до 63.

Такой прием используется преимущественно для генерации «случайных» чисел (правильнее их называть *псевдослучайными* – чтобы подчеркнуть их предсказуемость). Цикл из 64 чисел, конечно, слишком мал; Грогоно (см. литературу) предлагает константы для генерации перестановки чисел от 0 до 65 535;

```
CONST multiplier=25173; increment=13849;  cycle=65536;
```

Подбор констант с нужными свойствами – нетривиальная задача. Чтобы получить приведенные выше значения 37 и 3 для цикла из 64 чисел мне пришлось изрядно поэкспериментировать с простыми числами.[1]

Приведенная выше функция возвращает значение и *изменяет значение параметра*. Такой способ действий применяется нечасто. В большинстве функций нет необходимости изменять параметры, и, следовательно, нет смысла использовать слово **VAR** в заголовке.

В задачах моделирования, а также в играх часто удобнее использовать случайные дроби в интервале от 0 до 1, нежели случайные целые. Для получения дробей нужно слегка изменить функцию:

```
FUNCTION rnd(VAR seed: INTEGER): REAL;
  CONST multiplier=25173; increment=13849; cycle=65536;
  BEGIN
    rnd:=seed/cycle;
    seed:=(multiplier*seed+increment) MOD cycle
  END;
```

прежде INTEGER

константы Гроголо

имя изменено на rnd( )

добавлено деление

0.0≤rnd<1.0

Эта функция не будет работать, если значение MAXINT меньше, чем $2^{31}-1$. Тем не менее, следующая остроумная модификация программы генерирует циклы из 32 768 дробей даже тогда, когда MAXINT имеет значение всего лишь $2^{15}-1$ (32 767):

```
FUNCTION rnd(VAR seed: INTEGER): REAL;
  VAR a,b,c,d: INTEGER;
  BEGIN
    rnd:=seed/32767;
    a:=seed DIV 256;
    b:=seed MOD 256;
    c:=((b*93) MOD 256) + 13;
    d:=(b*26)+((b*93) DIV 256)+(a*93)+(c DIV 256)+27;
    seed:=((d MOD 128)*256)+(c MOD 256)
  END;
```

0.0≤rnd≤1.0

Следующая программа моделирует процесс бросания пары игральных костей. Ее цель – показать, насколько выгоднее ставить на 7, чем на любую другую сумму очков. (Применив массивы ~ см. гл. 8 ~ можно сделать программу существенно проще.)

```
PROGRAM bones(OUTPUT);
 VAR score,throws,seed,a,b,c,d,e,f,g,h,i,j,k: INTEGER;
  сюда вставьте первую версию rnd( )
BEGIN
  seed:=0; a:=0; b:=0; c:=0; d:=0; e:=0; f:=0;
  g:=0; h:=0; i:=0; j:=0; k:=0;
  FOR throws:=1 TO 3600 DO
    BEGIN    { броски }
      score:=TRUNC(1+6*rnd(seed))+TRUNC(1+6*rnd(seed));
      CASE score OF
        2: a:=a+1;          12: k:=k+1;
        3: b:=b+1;          11: j:=j+1;
        4: c:=c+1;          10: i:=i+1;
        5: d:=d+1;           9: h:=h+1;
        6: e:=e+1;           8: g:=g+1;
        7: f:=f+1;
      END  { CASE }
    END;  {FOR throws }
  WRITELN(2,3,4,5,6,7,8,9,10,11,12);
  WRITELN(a,b,c,d,e,f,g,h, i, j, k)
END.
```

6*rnd(seed) меняется от 0.0 до почти 6.0

2	3	4	5	6	7	8	9	10	11	12
89	194	298	396	523	558	532	418	298	202	92
100	200	300	400	500	600	500	400	300	200	100

сравните с «идеальным» ответом

подберите подходящий формат с учетом устройства вывода; например a:4,b:4,c:4 и т.д.

Результат примерно симметричен относительно 7. Вдохновляет сопоставление результата с «идеальным»; загляните на с. 78 – там предлагается более объемный тест.

СНОВА ЗАЕМ

ПРОГРАММА ИЛЛЮСТРИРУЕТ ОПРЕДЕЛЕНИЕ ПРОЦЕДУРЫ

Программа на с. 25 вычисляет месячную выплату по ссуде *s* выданной на *n* лет под процент *p*. Сложнее однако, ответить на обратный вопрос: под какой процент выдана ссуда величиной *s*, которая гасится месячными выплатами величиной *m* в течение *n* лет.

$$m = \frac{sr(1+r)^n}{12[(1+r)^n - 1]},$$

где $r = p \div 100$

Чтобы решить приведенное выше уравнение относительно *r* можно воспользоваться методом проб и ошибок. Возьмем какое-нибудь *r*, подставим в формулу и вычислим *m1*. Если *m1* совпало с *m*, то выбор *r* был верным. Если *m1* оказалось меньше, значит *r* было выбрано слишком малым, поэтому увеличим *r*, умножив его на *m/m1*, и попробуем еще раз. Если же *m1* больше чем нужно, значит *r* было выбрано слишком большим, поэтому снова умножим его на *m/m1*, на сей раз чтобы уменьшить, и вновь вычислим *m*. Короче говоря, если разница между *m* и *m1* велика, то умножаем *r* на *m/m1* и повторяем вычисления. Рано или поздно значение *r* окажется достаточно близким к точному решению уравнения.

Этот метод хорошо работает лишь до тех пор, пока увеличение искомой величины обусловливает увеличение (или уменьшение) результата. Если же результат колеблется или имеется разрыв, как например, банкротство, то этот метод не годится.

Ниже приведена программа:

```
PROGRAM loanrate(INPUT,OUTPUT);

  VAR
    s,m,m1,r,percent: REAL;
    n: INTEGER;

  PROCEDURE formula(VAR m: REAL; n: INTEGER; s,r: REAL);
    VAR a: REAL;
    BEGIN
      a:=EXP(LN(1+r)*n);
      m:=s*r*a/(12*(a-1))
    END;

BEGIN
  READ(s,m,n);
  r:=0.1;
  REPEAT
    formula(m1,n,s,r);
    r:=r*m/m1
  UNTIL ABS(m/m1-1) < 1E-6;
  percent:=r*100;
  WRITELN('Общая сумма: £',s:4:2);
  WRITELN('Месячная выплата: £',m:4:2);
  WRITELN('Число лет: ',n:4);
  WRITELN;
  WRITELN('Процент: ',percent:4:2,'%')
END.
```

a=(1+r)^n

sr(1+r)^n / 12[(1+r)^n -1]

Если ваш Паскаль интерактивный, вставьте сюда необходимые сообщения

начальный предположительный процент

округлять будет WRITELN

округлено до сотых

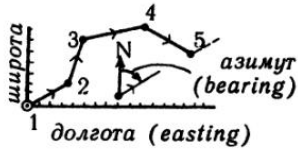
```
99.99 1.63 10
Общая сумма: £99.99
Месячная выплата: £1.63
Число лет: 10

Процент: 14.32%
```

ИМЕНА ФУНКЦИЙ КАК ПАРАМЕТРЫ

ГЛУБОКИЙ ВДОХ...

Здесь приведены операторы программы, вычисляющей приращения географических координат точек на земной поверхности по известным азимуту и расстоянию, пройденному от предыдущей точки (траверсу).




```
BEGIN
  northing:=0; easting:=0;
  WHILE NOT EOF(f)
    BEGIN
      READLN(f,bearing,distance);
      northing:=northing+projection(bearing,distance,cosine);
      easting:=easting+projection(bearing,distance,sine);
      WRITELN(northing:10:2,easting:10:2)
    END  { while }
END.   { program }
```

начало координат в точке 1

входной файл

нуль в точке 1

имя функции

имя функции

не может быть параметром-переменной

Вот определение функции projection(, ,):

```
FUNCTION projection(bng,dist: REAL; FUNCTION ratio(x:REAL):REAL):REAL;
  BEGIN
    projection:=dist*ratio(bng);
  END;
```

определяет третий формальный параметр

А вот как определяются функции, имена которых используются в качестве *фактических* параметров функции projection(, ,):

```
FUNCTION sine(b:REAL):REAL;
  BEGIN  sine:= SIN(3.1415926*b/180) END;
FUNCTION cosine(b:REAL):REAL;
  BEGIN  cosine:= COS(3.1415926*b/180) END;
```

Обратите внимание на определение третьего формального параметра функции projection(, ,):

```
FUNCTION  ratio( x : Real ): REAL
```

фактическим параметром должна быть функция, определенная пользователем

эта функция должна иметь параметр типа REAL

функция должна возвращать результат типа REAL

Единственная роль имени *x* – указать, что здесь должен быть параметр; тем самым синтаксис функции-параметра согласуется с определением функции.

Для полноты картины ниже приведено начало программы:

```
PROGRAM traverse(f,OUTPUT);
  VAR northing,easting: REAL;
```

здесь следует поместить определения функций, затем основную программу

То, что здесь описано, будет работать далеко не во всех Паскаль-системах. Многие компиляторы не позволяют использовать имена функций в качестве параметров, и я не могу утверждать, что осуждаю их за это. Единственная известная мне задача, где параметры-функции действительно нужны – это интегрирование.

... ВЫДОХ!

ССЫЛКИ ВПЕРЕД

В ПРОЦЕССЕ КОМПИЛЯЦИИ

Объявления констант и переменных в любом *блоке* располагаются перед скобками BEGIN и END, которые заключают в себе собственно операторы. Вследствие этого компилятору никогда не приходится иметь дело с оператором, содержащим константы и переменные, о которых он (компилятор) еще не знает. Появление необъявленной константы или переменной вызовет во время компиляции сообщение об ошибке.

То же рассуждения справедивы и в отношении *подпрограмм* (т.е. функций и процедур). Если компилятор встречает вызов подпрограммы, о которой он не знает, то появляется сообщение об ошибке. Следить за правильным порядком следования определений – обязанность программиста.

```
PROGRAM demo(INPUT,OUTPUT);
  VAR a,b,c: REAL;
  PROCEDURE ring(VAR area,circumf: REAL; diam: REAL);
    BEGIN
      circumf:=3.14*diam;
      area:=circle(diam)
    END

FUNCTION circle(d:REAL):REAL;
   BEGIN  circle:=3.14*SQR(d)/4  END;
```

в этом месте компилятор еще не знает о функции circle( )

Очевидный выход – поменять порядок строк так, чтобы функция circle( ) была определена перед процедурой ring(, ,). Однако можно обойтись и меньшей кровью (перестановка строк в реальных программах, значительно более длинных, чем наши простенькие примеры, может быть весьма серьезной операцией):

- оставьте злополучную подпрограмму на своем месте, лишь упростите ее заголовок, вычеркнув все параметры
- вставьте полный заголовок там, где ему надлежит быть, ~ т.е. *перед* подпрограммой, которая его вызывает.
- после полного заголовка добавьте предопределенное слово FORWARD[2]:

```
PROGRAM demo(INPUT,OUTPUT);
  VAR a,b,c: REAL;

FUNCTION circle(d:REAL):REAL;
FORWARD;

  PROCEEDURE ring(VAR area,circumf: REAL; diam: REAL);
    BEGIN
      circumf:=3.1415926*diam;
      area:=circle(diam)
    END;

FUNCTION circle;
BEGIN  circle:=3.1415926*SQR(d)/4.0  END;
```

вставьте полный заголовок перед всеми подпрограммами, вызывающими эту функцию

предупредите компилятор, что определение следует

оставьте от заголовка только имя

саму подпрограмму не трогайте

Помимо определения подпрограмм ссылки вперед используются в Паскале только для указателей в списках. Это описано в гл. 12.

ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ

СОЗДАЮТСЯ ЗАНОВО ПРИ КАЖДОМ ОБРАЩЕНИИ, ИСЧЕЗАЮТ ПРИ ВОЗВРАТЕ

Приведенные ниже примеры использовались на с. 69, чтобы показать различия между *локальными* и *нелокальными* переменными в процедуре:

```
PROCEDURE swop(VAR a,b: INTEGER);
  VAR tempry:INTEGER;
  BEGIN
    tempry:=a; a:=b; b:=tempry
  END;
```

нелокальные

локальная

```
PROCEDURE reverse(a,b: INTEGER);
  BEGIN
    WRITELN(b,a)
  END;
```

локальные переменные

Локальные переменные создаются при вызове процедуры. Затем текущие значения параметров-*значений* копируются в соответствующие локальные переменные. Например, вызов:

```
reverse(4,5);
```

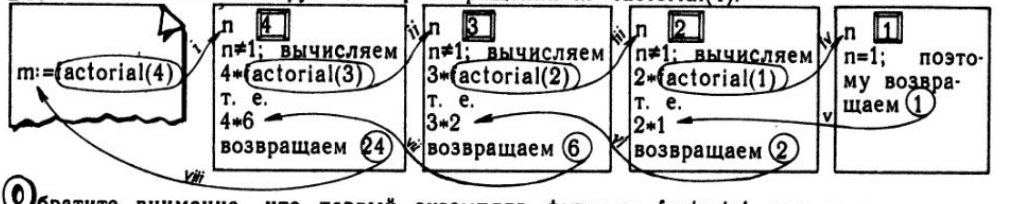
повлечет копирование числа 4 в локальную переменную с именем *a* и числа 5 в локальную переменную с именем *b*.

После этого начинает выполняться процедура. При завершении, когда управление возвращается в вызвавшую программу, локальные переменные забываются, их содержимое теряется навсегда. *Но пока управление не возвращено вызывающей программе, локальные переменные сохраняются.* Это очень важно в случае рекурсивных подпрограмм, что видно уже из банального примера о «факториале»:

```
FUNCTION factorial(number: integer): INTEGER;
  VAR n: INTEGER;
  BEGIN
    n:=number;
    IF n=1 THEN factorial:=1
           ELSE factorial:=n*factorial(n-1)
  END;
```

локальная переменная n

Проследите поведение функции при обращении m:=factorial(4):



Обратите внимание, что первый экземпляр функции factorial хранит в локальной переменной *n* значение 4 вплоть до того момента, когда в *m* будет возвращено число 24. Аналогично, вторая копия хранит 3, пока 6 не возвратится в первую копию и так далее. Локальная переменная является локальной для данного экземпляра функции. В некоторый момент выполнения изображенной выше программы будет одновременно существовать *четыре* различных копии переменной *n*.

Объявлять переменную *n*, как это было сделано выше (VAR n) – вовсе не обязательно. Параметры-значения автоматически объявляются локальными переменными:

```
FUNCTION factorial(n: integer): INTEGER;
  BEGIN
    IF n=1 THEN facto
           ELSE
```

параметры-значения – уже локальные переменные

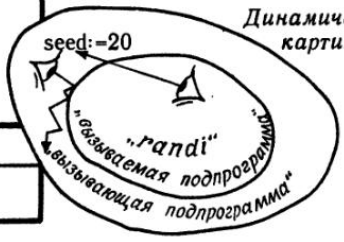
ПОБОЧНЫЕ ЭФФЕКТЫ

ИХ ЛУЧШЕ ИЗБЕГАТЬ, НО ИНОГДА ОНИ МОГУТ БЫТЬ ПОЛЕЗНЫМИ

Следующий генератор случайных чисел – альтернатива программе со с. 71:

```
FUNCTION randi;
  BEGIN
    randi:=seed/(65536-1);
    seed:=(25173*seed + 13849) MOD 65536
  END;
```

Динамическая картинка



Функцию можно вызвать, скажем, так:

```
seed:=20;
throw:=(1+5*randi)+(1+5*randi)
```

Этот фрагмент программы выполняется как задумано, потому что компьютер, работая внутри функции randi, способен видеть переменную с именем seed. Более того, функция randi может изменить значение, хранящееся в переменной seed. Вызываемая подпрограмма способна видеть свою вызывающую программу, но не наоборот.



Динамическая картинка

Когда подпрограмма обращается к переменной с именем *a*, она подразумевает *локальную* переменную *a*. Если локальной переменной с именем *a* нет, то взгляд обращается в вызвавшую подпрограмму (возможно, в свою же рекурсивную копию) в поисках *локальной* переменной с именем *a* в этой подпрограмме. Если и там нет локальной переменной *a*, то смотрим еще дальше наружу...

Тот же принцип применяется ко всем именованным объектам: переменным, константам, функциям, процедурам, файлам и типам.

Когда подпрограмма *изменяет* значение переменной, объявленной вне этой подпрограммы, говорят, что подпрограмма имеет *побочный эффект*. Функция randi имеет побочный эффект; она изменяет значение переменной seed, которая объявлена вне функции randi.

Побочные эффекты часто появляются случайно. Многократно используя переменные с короткими именами вроде a,b,c и забывая объявить их локально, программист имеет потенциальный источник неприятностей. Некоторые книги по Паскалю, с целью предотвратить такую опасность, ратуют за использование длинных имен переменных.

В маленьких программах, вероятно, проще всего делать все переменные глобальными. При использовании *множеств* (они описываются в следующей главе), возможно, единственный разумный путь – сделать глобальными все переменные типа множество. В длинных программах, возможно, имеет смысл определить несколько глобальных переменных, чтобы обращаться к ним из процедур. Однако, если побочные эффекты используются бесконтрольно или небрежно, то это уже – плохой стиль.

На противоположной странице приведена структура типичной программы. Чтобы подчеркнуть вложенность подпрограмм, они заключены в рамки. Надписи поясняют, какие переменные доступны на данном уровне вложения. Выделены и те переменные, которые могут вызвать побочные эффекты. Обратите внимание на то, что сама программа выступает в качестве подпрограммы (отличающейся нестандартным заголовком, определяющим файлы INPUT и OUTPUT, и нестандартной концовкой), вложенной в «Паскаль-оболочку».

ПРАВИЛА ВИДИМОСТИ

«СТАТИЧЕСКИЙ» РИСУНОК ВЛОЖЕННОЙ ПРОГРАММЫ



ПАСКАЛЬ-ОБОЛОЧКА: стандартные файлы (INPUT, OUTPUT), типы (REAL и т.д.) функции (SQR() и т.д.), процедуры (WRITE() и т.д.), константы (TRUE, FALSE)

```
PROGRAM twigs(INPUT,OUTPUT);
  VAR   a,b,c: REAL;

  PROCEDURE lining(p,q: REAL; VAR x,y: REAL);
    VAR   a,b: REAL;

    PROCEDURE chick(p: REAL; VAR x: REAL);
      VAR   a,d: REAL;
      BEGIN                    в этих операторах вы можете:
☆ использовать a,d,p, относящиеся к chick
☆ использовать b, относящееся к lining, c, относящееся к twigs
☆ использовать x для возврата результата при выходе из chick
☆ использовать y  "  "  "  "  "  "  "  "  "  "  lining[3]
☆ вызвать chick или lining рекурсивно
☆ использовать все файлы, типы, функции, процедуры, константы Паскаля
      END;

    PROCEDURE egg(p: REAL; VAR x: REAL);
      VAR   a,e: REAL;
      BEGIN                    в этих операторах вы можете:
☆ использовать a,e,p, относящиеся к egg
☆ использовать b, относящееся к lining, c, относящееся к twigs
☆ использовать x для возврата результата при выходе из egg
☆ использовать y  "  "  "  "  "  "  "  "  "  "  lining
☆ вызвать chick (для вызова chick из egg надо использовать FORWARD)
☆ вызвать egg или lining рекурсивно
☆ использовать все файлы, типы, функции, процедуры, константы Паскаля
      END;

    BEGIN { lining }    в этих операторах вы можете:
☆ использовать a,b,p,q, относящиеся к lining
☆ использовать c, относящееся к twigs
☆ использовать x,y для возврата результата при выходе из lining
☆ вызывать chick,egg
☆ вызывать lining рекурсивно
☆ использовать все файлы, типы, функции, процедуры, константы Паскаля
    END;

  BEGIN { twigs }    в главной программе вы можете:
☆ использовать a,b,c, относящиеся к twigs
☆ вызвать lining
☆ использовать все файлы, типы, функции, процедуры, константы Паскаля
  END.
```

возможны побочные эффекты

возможны побочные эффекты

возможны побочные эффекты

УПРАЖНЕНИЯ

1. Для оценки диапазона результатов, выдаваемых функциями arcsin( ) и arccos( ), которые определены на с. 66, напишите программу построения таблицы результатов при значениях параметров в интервале от -1 до 1 с шагом 0.1. Основой программы может быть, к примеру, такой оператор:

```
FOR n:=-10 TO 10 DO
  WRITELN(n/10:6:2,arcsin(n/10):6:2,arccos(n/10):6:2)
```

2. Выполните на компьютере программу *bones*, приведенную на с. 71. Если вы располагаете свободным компьютерным временем, увеличьте число бросаний игральной кости с 3600 до 32768. Посмотрите, станет ли результат ближе к «идеальному».

3. Выполните на компьютере программу *loanrate*, (с. 72). Как и предыдущая программа *loan*, эта программа не работает, в случае нулевого процента. Исправьте этот дефект. Если ваша Паскаль-система допускает интерактивный ввод, предусмотрите в программе сообщения для ее пользователя обо всех требуемых данных.

ПРИМЕЧАНИЯ РЕДАКТОРА

[1] (с. 70) Теоретический анализ этого и других методов получения случайных чисел можно найти в книге: Д. Кнут. Искусство программирования для ЭВМ. т.2 Получисленные алгоритмы, М.: «Мир», 1977.

[2] (с. 74) Слово **FORWARD** является не предопределенным, а зарезервированным словом в языке Паскаль. Это означает, что его нельзя переопределить в программе, в отличие, скажем, от предопределенного слова REAL.

[3] (с. 77) В примере программы *twigs* переменная *y* не является локальной по отношению к внутренним процедурам, поэтому использование ее для возврата результата - тоже побочный эффект.

7

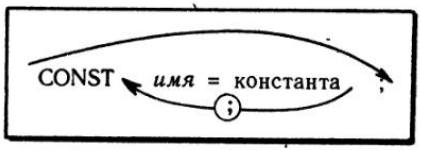
ТИПЫ И МНОЖЕСТВА

СТАНДАРТНЫЕ ТИПЫ
ОПРЕДЕЛЕНИЕ ТИПОВ
ПЕРЕЧИСЛЯЕМЫЕ ТИПЫ
ИНТЕРВАЛЬНЫЕ ТИПЫ
ТИП МНОЖЕСТВ И ПЕРЕМЕННЫЕ ТИПА МНОЖЕСТВО
КОНСТРУКТОРЫ МНОЖЕСТВ И ОПЕРАЦИИ НАД МНОЖЕСТВАМИ
ФИЛЬТР2 (ПРИМЕР)
МУ-У-У (ПРИМЕР)

СТАНДАРТНЫЕ ТИПЫ

REAL, INTEGER, CHAR, BOOLEAN ~ СВОДКА ~

Константы стандартных типов могут быть определены в разделе CONST любого блока. Объявлять тип каждой константы не надо: он узнается по форме написания:




```
CONST  pi=3.14;   increment=1;   star='*';
```

десятичная точка, следовательно, pi – типа REAL

нет десятичной точки, значит тип INTEGER

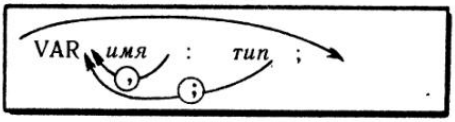
*апострофы, следовательно, * – типа CHAR*

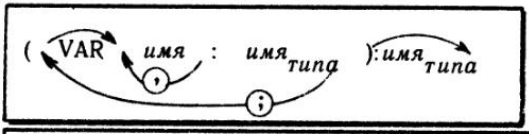
или из сопоставления с некоторой ранее определенной константой:

```
p=pi;    stella=star;    verily=TRUE;    decrement=-increment;
```

выражения недопустимы; предел сложности x=-y

Тип каждой *переменной* должен быть объявлен в разделе VAR того блока, в котором эта переменная будет использована:





Тип каждого *параметра* должен быть объявлен в заголовке процедуры или функции.

```
FUNCTION  mix(r:REAL; i:INTEGER; c:CHAR):BOOLEAN;
  VAR  s:REAL; j:INTEGER; letter:CHAR; ok:BOOLEAN
```

Арифметические действия над стандартными типами рассмотрены в гл. 4; в частности, описано смешение типов REAL и INTEGER в *выражении* и преобразование результата из одного типа в другой.

Выражения, использующие тип CHAR или операции **NOT**, **AND**, **OR** приводят к результату типа BOOLEAN.

Приведенные ниже правила касаются *порядковых значений*:

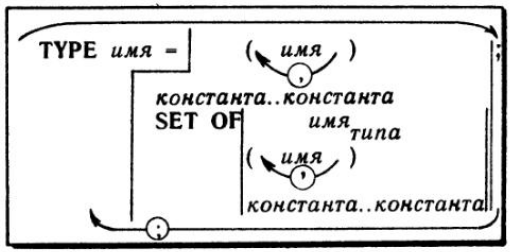
- Целое имеет порядковое значение, равное себе (ORD(6) – это 6), а, следовательно, имеет предшествующее и последующее (PRRED(6) – это 5, SUCC(6) – это 7).
- Величина типа REAL не имеет порядкового значения.
- Порядковые значения величин типа CHAR таковы, что ORD('A')<ORD('B') и т.д.; ORD('1')–ORD('0') равно 1, ORD('2')–ORD('1')=2 и т.д.
- При написании *условия* неявно используется функция ORD( ); таким образом, ORD('I')<ORD('J') может быть упрощено до 'I'<'J'. Однако напомню еще раз, что SUCC('I') – это не обязательно 'J' и не обязательно ORD('J')–ORD('I') равно 1.

ОПРЕДЕЛЕНИЕ ТИПОВ

ПЕРЕЧИСЛЯЕМОГО, ИНТЕРВАЛЬНОГО И ТИПА МНОЖЕСТВО

У программиста есть возможность определить свои собственные простые типы, отличные от четырех стандартных типов. Для этого служит раздел определения типов соответствующего блока. Раздел определения типов (TYPE) располагается между разделами CONST и VAR, что иллюстрируется ниже на этой странице.

Здесь дан синтаксис раздела TYPE (опущены структурные типы, которые рассматриваются, начиная со следующей главы).



Три типа имеют соответственно названия: *перечисляемые типы, интервальные типы* и *типы – множества.*

Вот пример перечисляемого и двух интервальных типов:

```
PROGRAM dodo(INPUT,OUTPUT);
  CONST pi=3.14;
  TYPE  daytype= (mon,tue,wed,thu,fri,sat,sun);
        weekdaytype= mon..fri;
        dicetype= 2..12
```

перечисляемый тип

интервальные типы

Впоследствии имена daytype, weekdaytype, dicetype могут быть использованы для определения переменных наравне с именами REAL, INTEGER, CHAR и BOOLEN.

```
VAR x: REAL;

    today: daytype;
    throw,score: dicetype;
PROCEDURE egg(VAR d:daytype);
```

Можно поступить иначе, убрав определение типа из раздела TYPE и включив его в раздел VAR:

```
PROGRAM dodo(INPUT,OUTPUT);
  CONST pi=3.14;
  TYPE  daytype= (mon,tue,wed,thu,fri,sat,sun);
  VAR x: REAL;
      today: mon..fri;
      throw,score: 2..12
```

определения типов, перенесены в раздел VAR

но в заголовках функций и процедур такая свобода не допустима:

```
PROCEDURE egg(VAR d:daytype);
```

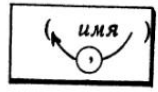
тип параметра должен быть определен в разделе TYPE

Перечисления и интервалы полезны при проверке правильности программ – благодаря им обеспечивается автоматическая проверка диапазона изменения переменных:

```
WHILE throw >= score DO simulate(throw,score);
CASE today OF
  mon,tue,wed,thu,fri: WRITE('Работай');
  sat,sun: WRITE('Играй')
END { CASE }
```

выдается сообщение об ошибке, если хоть одна переменная выйдет за границы

ПЕРЕЧИСЛЯЕМЫЕ ТИПЫ



Ниже даны определения двух перечисляемых типов и соответствующих переменных:

```
TYPE  days= (mon, tue, wed, thu, fri, sat, sun);
      status=(wedded, unwed);
VAR   today, tomorrow: days
```

здесь использовано слово wedded, а не wed, поскольку в перечислении все имена должны быть различными

Вы не можете считывать или печатать значения перечисляемого типа:

```
READ(today, tomorrow);
WRITE(fri, today)
```

Вы *можете* присваивать значения переменным перечисляемого типа:

```
today:=mon;
tomorrow:=today
```

но только в том случае, когда переменная и значение относятся к разным перечислениям:

```
today:=unwed;
```

К перечисляемому типу *неприменимы* арифметические действия:

```
today:=sat + sun
```

Константы перечисляемого типа имеют *порядковые значения*, начинающиеся с нуля:

```
WRITELN(ORD(mon), ORD(tue), ORD(sun));
```

```
0 1 6
```

Следовательно, можно вычислять предшествующие и последующие элементы:

```
today:=PRED(sun);
tomorrow:=SUCC(today);
WRITELN(ORD(today), ORD(tomorrow));
```

```
5 6
```

Однако первая константа не имеет предшествующего элемента, а последняя – последующего:

```
today:=PRED(mon);
tomorrow:=SUCC(sun);
```

В логических выражениях для всех элементов Паскаля, имеющих порядковые значения, можно не писать ORD( ):

```
IF ORD(today)>ORD(mon) THEN sayso;
IF today>mon THEN sayso
```

эти два оператора работают одинаково

Тип BOOLEAN – перечисляемый тип, определение которого автоматически присутствует в Паскаль-программе:

```
TYPE
  BOOLEAN = (FALSE, TRUE)
```

откуда заключаем, что ORD(FALSE) есть нуль, ORD(TRUE) – 1 и FALSE < TRUE.

ИНТЕРВАЛЬНЫЕ ТИПЫ

интервалы из ПЕРЕЧИСЛЕНИЙ, ЦЕЛЫХ и ЛИТЕР

константа .. константа
(нижняя граница) (верхняя граница)

Вот определения переменных нескольких интервальных типов:

```
TYPE  daytype= (mon,tue,wed,thu,fri,sat,sun);
VAR   weekday: mon..fri;        <- интервал типа daytype
      throw,score: 2..12        <- интервал типа INTEGER
      musketeer: 1..3;
      grade: 'A'..'D'           <- интервал типа CHAR
```

Интервальный тип может быть определен на основе любого типа, имеющего *порядковое значение*. Тем самым исключаются интервалы типа REAL:

```
VAR  price = (1.99..5.99)   <- Real
```

Интервалы перечислений подчинены тем же самым ограничениям, что и сам перечисляемый тип. Так, элементы типа mon..fri не могут читаться или печататься, к ним нельзя применять арифметические действия, их нельзя присваивать переменным никаких других типов, кроме mon..fri и daytype (где daytype – *базовый тип*, по отношению к которому mon..fri – интервал и они, следовательно, совместимы).

Константы *любого* интервального типа имеют порядковые значения, совпадающие с их порядковыми значениями в базовом типе. Так, в интервале sat..sun, базовый тип которого – daytype, значения ORD(sat) и ORD(sun) будут, соответственно, 5 и 6, но не 0 и 1.

Если базовый тип интервала – INTEGER, то значения такого интервального типа могут обрабатываться как целые числа. Такая обработка включает чтение, печать и целую арифметику:

```
READ(throw);
score:=SQR(throw);
Write(score)
```

Более того, можно смешивать значения из различных интервалов:

```
musketeer:= score + 2   <- -MAXINT..MAXINT
            1..3          2..12
```

Вместе с тем всякий раз перед изменением значения переменной (посредством присваивания, чтения и т.п.) проверяются ее границы. В этой автоматической проверке объявленных границ и есть смысл интервалов. В результате программист может не отвлекаться на частые проверки вида **IF** (score>12) **OR** (score<2) **THEN** WRITE('score лежит вне границ'). В хорошо написанных программах вы скорее увидите **VAR** score: 2..12 нежели **VAR** score: INTEGER.

Если базовый тип интервала – тип CHAR, то значения из интервала могут обрабатываться как литеры, в том числе их можно читать, печатать и использовать в логических выражениях:

```
READ(grade);
IF grade <= 'B'
 THEN WRITE('Хорошо сделано!')
 ELSE WRITE('Надо поработать')
```

ТИП МНОЖЕСТВ *и* ПЕРЕМЕННЫЕ ТИПА МНОЖЕСТВО

В общих словах, *множество* – это набор элементов одинакового типа. В Паскале вы можете создавать и именовать множества для отслеживания элементов любого *упорядоченного* типа (исключая тип REAL и структурные типы). «Отслеживание» здесь означает сохранение информации о наличии или отсутствии каждого элемента.



В следующем примере сначала определяется перечисляемый тип, а затем – *тип множества*. Базовым типом является тип daytype:

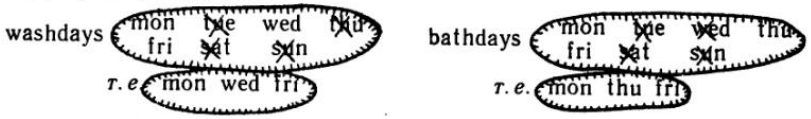
```
TYPE
  daytype= (mon,tue,wed,thu,fri,sat,sun);
  dayset = SET OF daytype
```

перечислены все дни недели

а здесь определены две переменные, хранящие множества дней по описанной ниже схеме:

```
VAR
  washdays, bathdays: dayset;
```

В какой-либо момент выполнения программы эти две переменные могут выглядеть, например, так:



Видно, что информация, содержащаяся в переменной типа *множество*, включает по одному логическому значению (присутствует или нет) для каждого возможного элемента множества.

Как при определении перечислений или интервалов, здесь также допускаются сокращения за счет перенесения определения типа в раздел VAR:

```
TYPE
  daytype= (mon,tue,wed,thu,fri,sat,sun);
  washdays,bathdays: SET OF daytype
```

можно и вовсе опустить раздел TYPE:

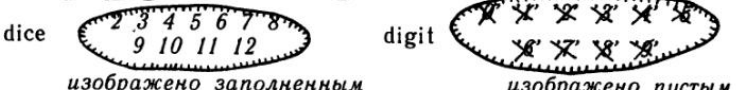
```
  washdays,bathdays: SET OF (mon,tue,wed,thu,fri,sat,sun);
```

В следующем разделе VAR определены несколько переменных типа *множество*:

```
VAR
  washdays,bathdays: SET OF (mon,tue,wed,thu,fri,sat,sun);
  teaset: SET OF CHAR;
  letters: SET OF 'A'..'Z';
  digits: SET OF '0'..'9';
  dice: SET OF 2..12
```

полное множество литер зависит от системы

для некоторых систем **SET OF** *INTEGER слишком велико*



КОНСТРУКТОРЫ МНОЖЕСТВ и ОПЕРАЦИИ НАД МНОЖЕСТВАМИ

Конструктор множества определяет *множество* ~ которое затем можно присвоить переменной, или как-либо обработать, или и то и другое вместе. *Конструктор множества* можно рассматривать как *константу типа множество.*



- [2*3..3*3,5+6,5]
- [5,6,7,8,9,11]

одинаковые множество

Множество можно сделать пустым:

```
dice:=[];
```

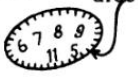




Или присвоить такое значение:

все эти элементы законны в интервале 2..12

```
dice:=[2*3..3*3,5+6,5];
```



Объединение двух множеств обозначается знаком плюс:

```
dice:= [2..5] + [4..6];
```





Пересечение обозначается звездочкой:

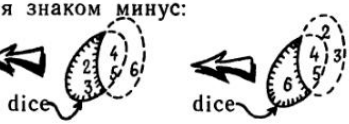
```
dice:= [2..5] * [4..6];
```





Разность двух множеств обозначается знаком минус:

```
dice:= [2..5] - [4..6];
dice:= [4..6] - [2..5];
```



Множества можно сравнивать. Использование компараторов **IN**, >=, <=, =, <> применительно к переменным типа *множество* или конструкторам позволяет строить логические выражения.

Принадлежность отдельного элемента множеству может быть выявлена при помощи операции **IN**:

```
WRITELN(6 IN [4..6],6 IN [2..5]);
```

TRUE FALSE

Для выяснения, *содержит* ли одно множество другое, используем >=

```
WRITELN([2..12] >= [3..5]);
WRITELN([3..5] >= [2..12]);
```

TRUE
FALSE

Для выяснения, *содержится* ли одно множество в другом, используем <=

```
WRITELN([2..12] <= [3..5]);
WRITELN([3..5] <= [2..12]);
```

FALSE
TRUE

Для проверки *совпадения* двух множеств используем = или <>

```
WRITELN([3..5] = [5,4,3]);
WRITELN([3..5] <> [4,5,3]);
```

TRUE
FALSE

ФИЛЬТР2

ИЛЛЮСТРИРУЕТ ПРОЦЕДУРЫ, КОТОРЫЕ ВЫЗЫВАЮТ ДРУГ ДРУГА, ПЕРЕЧИСЛЯЕМЫЕ ТИПЫ И КОНСТРУКТОРЫ МНОЖЕСТВ

Программа filter (на с. 59) считывает входной файл, и печатает в выходной файл все распознанные числа. Представленная ниже программа имеет то же назначение. Она несколько длиннее предыдущей версии, хотя, возможно, понятнее, поскольку она более прямолинейна. Процедуры используются наипростейшим образом; они работают только с глобальными переменными:

```
PROGRAM filter2(INPUT,OUTPUT);
  VAR state: (ignoring,pending,reading);      { перечисляемый тип }
      fraction: 0..MAXINT;                    { интервал }
      ch: CHAR; positive: BOOLEAN; number: REAL;

  PROCEDURE initialize;          { начальная установка }
    BEGIN
      state:=ignoring; positive:= TRUE;
      number:=0; fraction:=0
    END;

  PROCEDURE display;             { затем начальная установка }
    BEGIN
      IF fraction>0 THEN  number:= number/fraction;
      IF NOT positive THEN  number:= -number;
      WRITELN(number:10:2);
      initialize
    END;

  PROCEDURE accumulate;          { накапливать число и установить чтение }
    BEGIN
      number:=10*number + ORD(ch)-ORD('0');
      fraction:=10*fraction;
      state:=reading
    END;

  PROCEDURE negate;              { и установить ожидание }
    BEGIN
      positive:=FALSE;    state:=pending
    END;

BEGIN   { программа }
  initialize;
  WHILE NOT EOLN DO
    BEGIN  { WHILE }
      READ(ch);
      CASE state OF
{ игнор. } ignoring: IF ch='-'
                     THEN negate
                     ELSE IF ch IN ['0'..'9']
                          THEN accumulate;
{ ждать }  pending: IF ch IN ['0'..'9']
                    THEN accumulate;
                    ELSE initialize;
{ читать } reading:  IF ch='.'
                     THEN fraction:=1
                     ELSE IF ch IN ['0'..'9']
                          THEN accumulate;
                          ELSE display
      END  { CASE }
    END;   { WHILE }
  IF state=reading THEN display
END.
```

state \ symbol	IN [0..9]	'.'	'-'	прочие
игнорировать:	accumulate и перейти к чтению	игнор.	negate и ожид.	игнор.
ждать:		initialize		
читать:		frac.:=1	display и initialize	

попробуйте эту программу с данными, приведенными на с. 59

недостаток: любая литера работает как признак конца числа, например, $-8-9 даст: -8.00 9.00*

чтобы не упустить число в самом конце файла

МУ–У–У

ИГРА ДЛЯ ИЛЛЮСТРАЦИИ ИНТЕРВАЛЬНОГО ТИПА И МАНИПУЛЯЦИЙ НАД МНОЖЕСТВАМИ

Компьютер задумывает четырехзначное число, не содержащее двух одинаковых цифр. Вы набираете свое число и компьютер сообщает количество быков (точно угаданных цифр) и количество коров (цифр, которые есть в задуманном числе, но на другом месте). Например, пусть задуманное число 5734, а вы набрали 0755. Результат будет 1 бык и 2 коровы. Игра продолжается до тех пор, пока вы не получите четыре быка.

```
PROGRAM mooo(INPUT,OUTPUT);
  TYPE playtype = '0'..'9';
       seedtype = 0..65535;
       scoretype = 0..4;
  VAR  pool,target: SET OF playtype;
       a,b,c,d: playtype;
       seed: seedtype;
       bulls,cows: scoretype;

FUNCTION random: REAL;
  BEGIN
    random:=seed/65536;
    seed:=(25173*seed + 13849) MOD 65536;
  END; { random }

FUNCTION unique: playtype;
  VAR ch: CHAR;
  BEGIN
    REPEAT
      ch:=CHR(TRUNC(10*random)+ORD('0'));
    UNTIL ch IN pool;
    unique:=ch;
    pool:=pool-[ch];
    target:=target+[ch]
  END; { unique }

PROCEDURE try(thisone: CHAR);
  VAR ch: CHAR;
  BEGIN
    READ(ch);
    IF ch IN target
     THEN IF ch=thisone
            THEN bulls:=SUCC(bulls)
            ELSE cows:=SUCC(cows)
END;  { try }

BEGIN  { программа }
  WRITE('Задайте случайное число,');
  WRITELN(' затем отгадывайте');
  READLN(seed);
    pool:=['0'..'9']; target:=[];
    a:=unique; b:=unique; c:=unique; d:=unique;
    REPEAT
      bulls:=0; cows:=0;
      try(a); try(b); try(c); try(d);
      WRITELN('Быков: ',bulls:1,'; коров: ',cows:1);
      READLN
    UNTIL bulls=4;
END. { программа }
```

pool — например — *target*

6, а не 5, поэтому результат всегда меньше 1.0

возвращает случайное число в интервале 0.0 ≤ random < 1.0

переместить случайную цифру из 'pool' в 'target'

разность множеств

объединение множеств

прочитать следующую цифру и, если необходимо, увеличить число быков или коров

все цифры

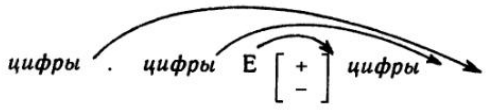
пустое множество

компьютер задумывает число abcd

```
Задайте случайное
35109
1234
Быков: 0; коров: 1
5678
Быков: 1; коров: 0
5990
Быков: 2; коров: 0
...
Быков: 4; коров: 0
```

УПРАЖНЕНИЯ

1. Дополните программу filter2 на с. 86 так, чтобы она могла обрабатывать числа, записаные в «научном» формате, где «E» означает «умножить на 10 в степени»:



Для этого потребуется расширить таблицу переходов.

2. Попробуйте игру МУ-У-У со с. 87. Усовершенствуйте игру. Для этого модифицируйте программу так, чтобы она:

- предлагала сыграть еще раз, после того как игра закончилась
- останавливала игру и считала ее выигранной компьютером, если задуманное число не было правильно угадано после десяти попыток
- отдельно подсчитывала число выигрышей игрока и компьютера и выводила счет на экран.

ПРИМЕЧАНИЯ РЕДАКТОРА

[1] (с. 87) Пример работы программы *тооо* в правом нижнем углу страницы вымышленный. Как можно установить ~ даже без помощи компьютера ~ *не существует* числа из четырех различных цифр, для которого программа *тооо* даст ответы, показанные в примере.

Действительно, из первых двух ответов видно, что среди цифр 1, 2, 3, 4, 5, 6, 7, 8 есть всего две цифры задуманного числа; но всего в числе – 4 различные цифры, значит, обе оставшиеся цифры – 0 и 9 – обязательно входят в задуманное число. Но тогда, рассматривая последние три цифры 9, 9, 0 вопроса 5990 программа *тооо каждую* из них посчитает коровой или быком (при наличии в вопросе совпадающих цифр программа *тооо* действует не вполне логично), следовательно, сумма быков и коров в третьем ответе должна быть не меньше трех.(Фактически, при указанном начальном значении случайного числа, программа задумывает число 5920.)

8

МАССИВЫ и СТРОКИ

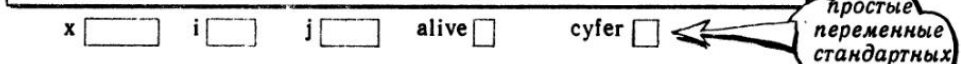
ЗНАКОМСТВО С МАССИВАМИ
СИНТАКСИС ОБЪЯВЛЕНИЯ МАССИВОВ
ПЛОЩАДЬ МНОГОУГОЛЬНИКА (ПРИМЕР)
ПРОВОДА (ПРИМЕР)
СОРТИРОВКА МЕТОДОМ ПУЗЫРЬКА (ПРИМЕР)
БЫСТРАЯ СОРТИРОВКА (ПРИМЕР)
УПАКОВКА
ЗНАКОМСТВО СО СТРОКАМИ
ФОКУС (ПРИМЕР)
СИСТЕМЫ СЧИСЛЕНИЯ (ПРИМЕР)
УМНОЖЕНИЕ МАТРИЦ (ПРИМЕР)
НАСТРАИВАЕМЫЕ ПАРАМЕТРЫ-МАССИВЫ

ЗНАКОМСТВО С МАССИВАМИ

ПРЯМОУГОЛЬНЫЙ МАССИВ КОРОБОЧЕК КАКОГО-ЛИБО ОДНОГО ТИПА

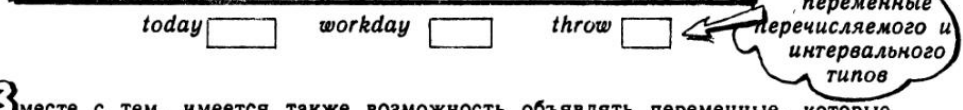
До сих пор переменные стандартного типа изображались отдельными маленькими коробочками:

```
VAR x: REAL;   i,j: INTEGER;   alive: BOOLEAN;   cyfer: CHAR;
```



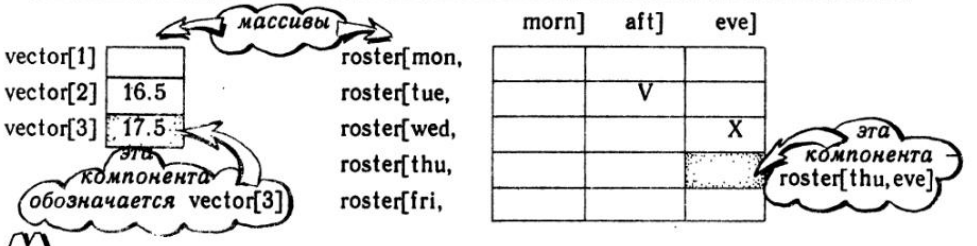
это относится и к переменным *перечисляемого* и *интервального* типов:

```
TYPE  daytype = (mon,tue,wed,thu,fri,sat,sun);
VAR   today: daytype; workday: mon..fri; throw: 2..12;
```



Вместе с тем, имеется также возможность объявлять переменные, которые являются *массивами* таких маленьких коробочек:

```
TYPE  daytype = (mon,tue,wed,thu,fri,sat,sun);
      session = (morn,aft,eve);
VAR   vector: ARRAY[1..3] OF REAL;
      roster: ARRAY[mon..fri,session] OF BOOLEAN;
```



Маленькие коробочки массива называются *компонентами*; в квадратных скобках стоят *индексы*. *Базовый тип* массива – это тип маленьких коробочек, из которых составлен массив (в каждом массиве все компоненты одного типа).

Компоненты можно обрабатывать так же, как переменные базового типа:

```
vector[2]:=16.5;  READ(vector[3]);
roster[tue,aft]:=TRUE;  WRITE(roster[tue,aft]);
roster[wed,eve]:= NOT roster[tue,aft];
```

Однако использование компонент массива в качестве обычных переменных не дает никакой выгоды. Массивы ценны тем, что индексы *могут быть переменными или выражениями, обеспечивая доступ к последовательным компонентам.* Взгляните на следующий фрагмент:

```
FOR day:= mon TO tue DO
  FOR time:= morn TO eve DO
    roster[day,time]:=FALSE;
FOR i:= 1 TO 3 DO vector[i]:=0;
```

записываем FALSE *во все компоненты массива* roster *и 0 во все компоненты массива* vector

Предполагается, что предварительно в разделе **VAR** объявлено i: 1..3, day: mon..fri и time: morn..eve .

СИНТАКСИС ОБЪЯВЛЕНИЯ МАССИВОВ

Массивы, изображенные напротив, могли быть объявлены после предварительного определения соответствующих типов:



Синтаксис *типа массива*:



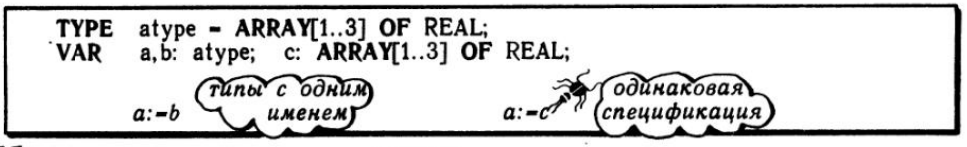
Синтаксис обращения к компоненте массива:



Обработка массивов производится путем изменения индексов компонент, как показано напротив. Есть, однако, одно важное исключение: копия *всего* содержимого массива может быть присвоена другому массиву *того же типа* одной операцией:



Здесь слова «того же типа» означают тип с *таким же именем*. Тип, имеющий такую же спецификацию, но другое имя, не подходит:



Исключением тут является тип PACKED ARRAY[ ] OF CHAR, для которого в некоторых версиях Паскаля эквивалентность не обязательна. Измените в последних примерах REAL на CHAR, и запись a:=c станет корректной.

Двумерный массив, такой как массив roster, на самом деле является *массивом массивов*. Нижеследующий синтаксис вполне правомерен, но слишком неуклюж:

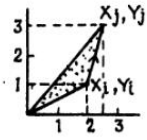
```
TYPE rostertype = ARRAY[mon..fri] OF ARRAY [session] OF BOOLEAN;
     roster[day][time]:=FALSE
```

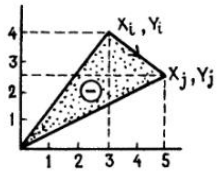
ПЛОЩАДЬ МНОГОУГОЛЬНИКА

ПРИМЕР, ИЛЛЮСТРИРУЮЩИЙ ИСПОЛЬЗОВАНИЕ ОДНОМЕРНЫХ МАССИВОВ

Посмотрите на схему справа: Заштрихованная площадь A_{ij} дается выражением:

$$A_{ij} = \frac{1}{2}(x_i y_j - x_j y_i) = \frac{1}{2}(2*3 - 2.5*1) = 1.75$$

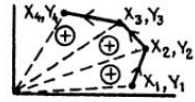


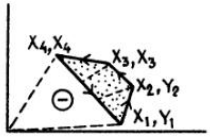


Та же формула может быть использована для вычисления площади на рисунке слева. Но здесь площадь оказывается *отрицательной*:

$$A_{ij} = \frac{1}{2}(x_i y_j - x_j y_i) = \frac{1}{2}(3*2.5 - 5*4) = -6.25$$

Формулу можно применять к последовательным сторонам многоугольника. Сумма площадей треугольников даст изображенную здесь площадь





Если многоугольник *замкнут*, как показано слева, то сумма площадей даст площадь, заключенную внутри него.

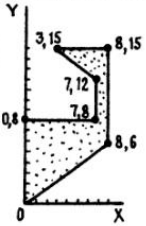
Ограниченная контуром область должна лежать *слева* от каждой стрелки; стороны фигуры не должны пересекаться, как, например, у восьмерки.

Следующая программа вводит координаты граничных точек и вычисляет площадь многоугольника:

```
PROGRAM polygon(INPUT,OUTPUT);
  TYPE
    spantype = 1..30;          { поставьте сюда требуемый наибольший размер задачи }
  VAR
    i,j,n: spantype; area: REAL;
    x,y: ARRAY[spantype] OF REAL;

BEGIN
  { если ваш Паскаль интерактивный, вставьте: WRITELN('Число вершин?'); }
  READLN(n);
  FOR i:=1 TO n DO             { инициализация }
    READLN(x[i],y[i]);
  area:=0;
  FOR i:=1 TO n DO             { если i=1, то j=2;
    BEGIN                        если i=2, то j=3;
      j:=(i MOD n) + 1;          и т.д.,
      area:=area+0.5*(x[i]*y[j]-x[j]*y[i])   но если i=n, то j=1.
    END;
  WRITELN('Площадь равна ',area:8:2)

END.
```

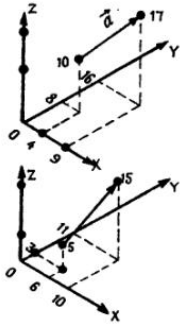



```
Число вершин?
7
0 0
8 6
8 1
3 15
7 12
7 8
0 8
Площадь равна 53.00
```

ПРОВОДА

РАБОТА С МАССИВАМИ КАК С ВЕКТОРАМИ ~ И СОВСЕМ НЕМНОГО МАТЕМАТИКИ

Два силовых кабеля $\vec{a}$ и $\vec{b}$, если наложить эти схемы, кажутся до страшного близки друг к другу. Каково же наименьшее расстояние между $\vec{a}$ и $\vec{b}$?

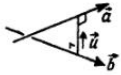


Тригонометрическое решение будет очень громоздким; здесь нас выручит векторная алгебра. Представим $\vec{a}$ и $\vec{b}$ как векторы:

$$\vec{a} = (9-4)\vec{i} + (16-8)\vec{j} + (17-0)\vec{k}$$
$$\vec{b} = (10-6)\vec{i} + (11-3)\vec{j} + (15-5)\vec{k}$$

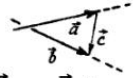
Их векторное произведение $\vec{a} \times \vec{b}$ – это вектор, перпендикулярный и $\vec{a}$ и $\vec{b}$. Промасштабируйте его, поделив на его же длину $|\vec{a} \times \vec{b}|$ и вы получите *единичный* вектор, параллельный вектору $\vec{a} \times \vec{b}$:

$$\vec{u} = \vec{a} \times \vec{b} \div |\vec{a} \times \vec{b}|$$



Возьмите вектор $\vec{c}$, соединяющий любую точку вектора $\vec{a}$ с любой точкой вектора $\vec{b}$. Здесь изображен один из них; он соединяет конец $\vec{a}$ с концом $\vec{b}$:

$$\vec{c} = (10-9)\vec{i} + (11-16)\vec{j} + (15-17)\vec{k}$$



Кратчайшее расстояние d между $\vec{a}$ и $\vec{b}$ дается проекцией $\vec{c}$ на $\vec{u}$ (т.е. скалярным произведением $\vec{c}$ и $\vec{u}$), которая равна:

$$d = \vec{c} \cdot \vec{u}$$

в этом примере получится $d = 3 \cdot 52$

```
PROGRAM cables(INPUT,OUTPUT);
  TYPE vector = ARRAY[1..3] OF REAL;
  VAR  a,b,c,u: vector; d,length: REAL;
       coord: ARRAY[1..12] OF REAL;
       i: 1..12;
BEGIN
  FOR i:=1 TO 12 DO READ(coord[i]);
  FOR i:=1 TO 3 DO
    BEGIN
      a[i]:=coord[3+i] - coord[i];
      b[i]:=coord[9+i] - coord[6+i];
      c[i]:=coord[9+i] - coord[3+i]
    END;
  u[1]:= a[2]*b[3] - b[2]*a[3];
  u[2]:= a[3]*b[1] - a[1]*b[3];
  u[3]:= a[1]*b[2] - b[1]*a[2];
  length:=SQRT(SQR(u[1])+SQR(u[2])+ SQR(u[3]));
  FOR i:=1 TO 3 DO u[i]:=u[i]/length;
  d:=c[1]*u[1]+c[2]*u[2]+ c[3]*u[3];
  WRITELN('Кратчайшее расстояние',d:6:2)
END.
```

чтение данных

формирование $\vec{a}, \vec{b}, \vec{c}$

$$\vec{u} = \vec{a} \times \vec{b} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}$$

$d = \vec{c} \cdot \vec{u}$

	coord	
[1]	4	начало $\vec{a}$
[2]	8	
[3]	10	
[4]	9	конец $\vec{a}$
[5]	16	
[6]	17	
[7]	6	начало $\vec{b}$
[8]	3	
[9]	5	
[10]	10	конец $\vec{b}$
[11]	11	
[12]	15	

	a			b
[1]	5	×	[1]	4
[2]	8		[2]	8
[3]	7		[3]	10

	c			u
[1]	1	·	[1]	.716
[2]	-5		[2]	-.656
[3]	-2		[3]	.239

СОРТИРОВКА МЕТОДОМ ПУЗЫРЬКА

ЗАКЛЕЙМЕННАЯ ОДНИМ ИЗ МОИХ РЕЦЕНЗЕНТОВ КАК «НЕПРЕВЗОЙДЕННАЯ В СВОЕЙ НЕЭФФЕКТИВНОСТИ»

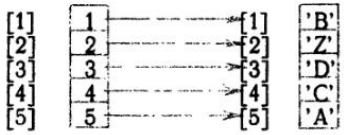
Существует много методов сортировки (упорядочения) содержимого массива. Ниже описан несложный *метод пузырька*.

Чтобы разобраться в нем, возьмем список из букв. «Пометим» первую и следующую за ней буквы. Если порядок букв правильный, то так их и оставим и передвинем метку вниз на одну строчку. Если порядок букв неверный, то поменяем их местами и также сдвинем метку вниз. Закончим процесс, не доходя одной строчки до конца списка, с тем чтобы предотвратить сравнение с элементом за пределами списка. Вот картинка, поясняющая метод:



«Утопив» самую тяжелую букву, осталось отсортировать список букв, лежащих сверху. С этим списком мы поступим так же, как и с полным списком. Другими словами, обратимся к уже известной процедуре рекурсивно.

Более аккуратный подход к сортировке – поставить в соответствие сортируемому массиву элементов массив *указателей* (pointers):



Теперь можно переставлять указатели, а не сами элементы. Когда сортировка закончится, массивы будут выглядеть так:



Если цель – отсортировать всего несколько букв, то этот подход не дает особых выгод. Но в реальных условиях с каждым сортируемым элементом может быть связано значительное количество информации, и гораздо проще двигать указатели нежели всю информацию, на которую они указывают.

Вот полиая программа сортировки букв:

```
PROGRAM bubbles(INPUT,OUTPUT);

  TYPE sizetype = 0..30;

  VAR pointers: ARRAY [sizetype] OF sizetype;
      letters: ARRAY [sizetype] OF CHAR;
      key: CHAR; n,i: sizetype;

  PROCEDURE swop(VAR p,q: sizetype);
    VAR  tempry: sizetype;
    BEGIN
     tempry:=p; p:=q; q:=tempry
    END;

  PROCEDURE sort(first,last: sizetype);
    VAR  i: sizetype; sorted: BOOLEAN;
    BEGIN  { sort }
     IF first < last THEN
      BEGIN
        sorted:= TRUE;
        FOR i:=1 TO last-1 DO
         BEGIN
           IF letters[pointers[i]]>letters[pointers[i+1]]
             THEN
               BEGIN
                swop(pointers[i],pointers[i+1]);
                sorted:=FALSE;
               END { if letters }
         END; { for i }
         IF NOT sorted THEN sort(first,last-1)
      END { if first < last }
    END; { sort }

BEGIN  { bubbles }
  READLN(n);
  FOR i:=1 TO n DO
   BEGIN
    READLN(letters[i]);
    pointers[i]:=i
   END;
  sort(1,n);
  WRITELN;
  FOR i:=1 TO n DO
    WRITE(letters[pointers[i]])
END. { bubbles }
```

ПРОЦЕДУРА СОРТИРОВКИ

рекурсивный вызов

если ваш Паскаль интерактивный, вставьте: WRITELN('Число букв?');

прочитать последовательно все буквы и установить указатели

сортировать буквы

вывести их по порядку

```
Число букв?
5
B
Z
D
C
A

ABCDZ
```

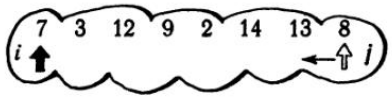
Хотя метод пузырька неэффективен при сортировке запутанного списка, список, в котором всего одно или два значения стоят не на месте, сортируется очень быстро.

БЫСТРАЯ СОРТИРОВКА

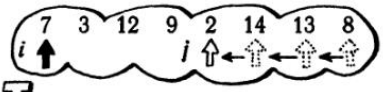
ПРИМЕР ДЛЯ ИЛЛЮСТРАЦИИ РЕКУРСИИ И ЕЩЕ ОДНОГО МЕТОДА СОРТИРОВКИ

Метод, называемый быстрой сортировкой, был придуман профессором Ч. Хоаром. Ниже приведена интерпретация этого метода, которая скорее иллюстрирует принципы, нежели прдставляет собой рабочий продукт.

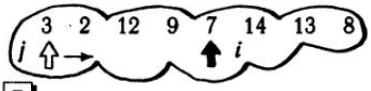
Возьмем для сортировки несколько чисел:



Установим указатели *i* и *j*, как это показано, на концы списка. Будем двигать *j* по направлению к *i*. Если число, на которое указывает *j*, *больше*, чем то, на которое указывает *i*, то передвигаем *j* на один шаг.



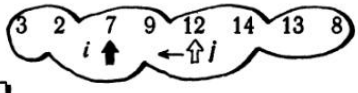
Теперь *j* указывает на *меньшее* число, чем *i*. Поэтому меняем местами эти два числа и сами указатели:



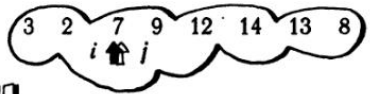
Продолжаем двигать *j* по направлению к *i* (теперь это будет движение направо, а не налево). Если *j* указывает на число *меньшее*, чем то, что относится к *i*, то передвигаем *j* на шаг вправо. (Обратите внимание, что *условие* движения *j* изменилось на обратное.)



Теперь *j* указывает на число, *большее*, чем то, что относится к *i*. Меняем числа, указатели, направление и условие, как уже делали раньше:

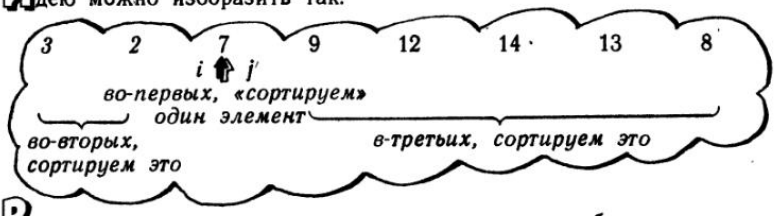


Продолжаем в том же духе, когда нужно переключаясь, пока *j* не встретится с *i*:



На этом этапе можно утверждать, что все числа слева от числа с указателем *i* не превосходят, а все числа справа – не меньше этого числа. Другими словами, отмеченное число находится на своем законном месте. Тем не менее, числа слева от *i* не отсортированы; не отсортированы и числа правее *i*. Однако уже описана процедура размещения одного элемента, которая разделяет группу на две; теперь остается только отсортировать группы справа и слева от *i*, начав сортировку по схеме, которая подробно изложена выше.

Идею можно изобразить так:



Рекурсия применима тогда, когда задача может быть сведена к такой же задаче или таким же задачам меньшего размера. Когда размер задачи станет достаточно малым, в рекурсивной процедуре должно быть, разумеется, предусмотрено завершение. В случае сортировки это должно произойти тогда, когда процедура вызывается для сортировки одного элемента.

Ниже приведена процедура быстрой сортировки, которую можно использовать вместо процедуры сортировки методом пузырька на предшествующем развороте:

```
PROCEDURE sort(first,last: sizetype);
  VAR
    i,j: sizetype; jstep: -1..1; condition: BOOLEAN;
  BEGIN
    IF first<last
      THEN
        BEGIN
          i:= first; j:= last;
          jstep:= -1;
          condition:= TRUE;
          REPEAT
            IF condition=(letters[pointers[i]]>letters[pointers[j]])
              THEN
                BEGIN
                  swop(pointers[i],pointers[j]);
                  swop(i,j);
                  jstep:= -jstep;
                  condition:= NOT condition;
                END;
                j:= j+jstp
          UNTIL j=i;
          sort(first,i-1);
          sort(i+1,last)
        END   { if first<last }
  END;  { sort }
```

если first≥last, то сортировать нечего

ПОДСТАВЬТЕ ЭТУ ПРОЦЕДУРУ В ПРОГРАММУ НА ПРЕДЫДУЩЕМ РАЗВОРОТЕ

по-существу, перестановка элементов

перестановка указателей

изменение направления

изменение условия

рекурсивные вызовы

выход, если сортировать нечего

Обратите внимание, как переключается условие между <= и >. Логическое выражение letters[pointers[i]]>letters[pointers[j]] принимает значение *true* или *false*. Это значение сравнивается с тем значением, которое хранится в переменной condition. Это значение, посредством **NOT**, переключается между *true* и *false*.

Каждый раз, когда процедура вызывает себя, компьютер должен запомнить значения параметров и локальных переменных для возможного использования их при возврате. Это было проиллюстрировано на простом примере рекурсии на с. 75. В примере выше, переменные *jstep* и *condition* можно было бы сделать глобальными и, таким образом, сэкономить память. Но поступать так в столь маленьких задачах, как приведенные в этой книжке, было бы глупо.

УПАКОВКА

КОМПРОМИСС МЕЖДУ БЫСТРОДЕЙСТВИЕМ И ОБЪЕМОМ ПАМЯТИ (НЕКОТОРЫЕ КОМПИЛЯТОРЫ УПАКОВЫВАЮТ АВТОМАТИЧЕСКИ)

Для представления *логического значения* требуется один бит ▯ (т.е. двоичная цифра); *литера* обычно требует четыре бита ▭; *целое* – 16 или 32 бита ▭▭▭▭. Однако, в компьютере единица хранения – – это *слово*. Размер этого слова зависит от марки и модели компьютера, обычно это – 32 бита. Отсюда следует, что хранение логических значений и литер (возможно, даже целых) по одному в слове – расточительство памяти.

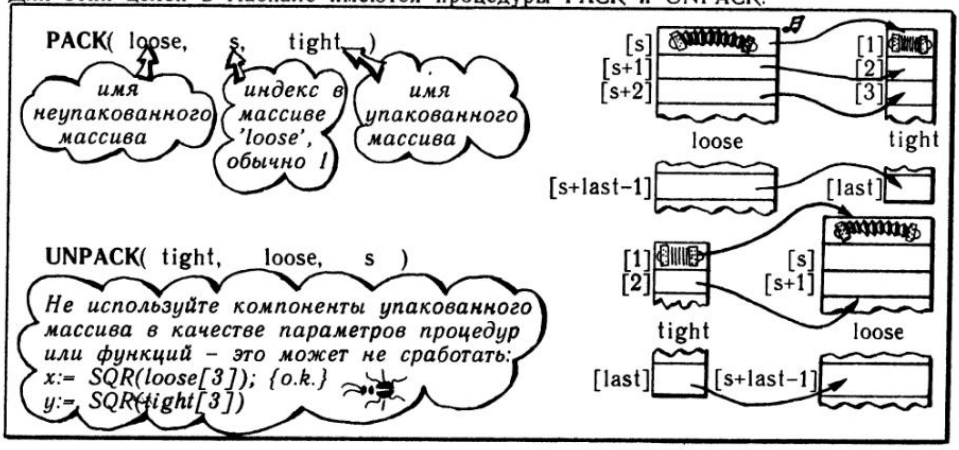
В Паскале слово PACKED в определении массива (или записи) дает компилятору право упаковать информацию более плотно, чем один элемент на слово. Например, строчка:

PACKED ARRAY [1..32768] **OF** BOOLEAN

если, конечно, компилятор упакует компоненты этого массива до тридцати двух в слове, позволит, быть может, достьчь чего то, что в другом случае было бы невозможно. (Некоторые современные компиляторы выполняют упаковку автоматически.)

Цена за сэкономленную память – пониженная скорость обращения к памяти во время выполнения.

Баланс между быстродействием и памятью в некоторых системах может достигаться выборочной упаковкой: скажем, обрабатывается неупакованный массив, затем для хранения его содержимое копируется в упакованный массив. Для этих целей в Паскале имеются процедуры PACK и UNPACK.



Ниже приведены два типичных обращения к этим стандартным процедурам:

```
VAR prolix: ARRAY[1..1000] OF CHAR;
    pith: PACKED ARRAY[1..1000] OF CHAR;

PACK(prolix,1,pith);        UNPACK(pith,prolix,1);
```

ЗНАКОМСТВО СО СТРОКАМИ

В НЕКОТОРЫХ ВЕРСИЯХ П ИМЕЕТСЯ СПЕЦИАЛЬНЫЙ СТРОКОВЫЙ ТИП

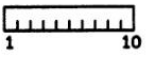
Строковая константа состоит из букв, заключенных в апострофы. Если в строку нужно включить апостроф, то он должен быть записан в виде пары апострофов:

```
WRITELN('Ooh! ', 'It''s cold!')
```

⇒ Ooh! It's cold!

Под *строковую переменную* в стандартном Паскале выделяется PACKED ARRAY[] OF CHAR:

```
VAR shiver: PACKED ARRAY[1..10] OF CHAR
```

shiver 

Строковые константы можно присваивать строковым переменным:

```
shiver := 'It''s cold!';
WRITELN('Ooh! ',shiver)
```

⇒ Ooh! It's cold!

Но в стандартном Паскале присваивание допустимо, только если количество литер константы совпадает с размером упакованного массива:

```
shiver := 'Ooh!';
shiver := 'Ooh!!!!!'; { o.k. }
```

ВО МНОГИХ СОВРЕМЕННЫХ КОМПИЛЯТОРАХ НЕ ТРЕБУЕТСЯ СОВПАДЕНИЯ ДЛИН

Строки можно сравнивать, для этого надо, чтобы число литер в них совпадало. Можно использовать любой компаратор (=, <>, >= и т.д.):

```
WRITELN(shiver = 'Ooh!!!!!');
WRITELN(shiver > 'Ooh!');
```

⇒ TRUE
Error

Сравнение выполняется на основе *порядковых значений*. Литеры двух строк сопоставляются начиная с левого края, пока не встретится отличие. Строка, в которой первая несовпадающая литера имеет большее порядковое значение, считается большей. Отсутствие неравных литер означает равенство строк:

'a' < 'b' → *true, ORD('a')<ORD('b')*

'abcz' < 'abda' → *true, ORD('c')<ORD('d')*

'abcdef'='abcdef' → *true, нет отличий*

Порядок литер в интервалах от '0' до '9', от 'A' до 'Z' и от 'a' до 'z' определяется Паскалем; в остальном порядковые значения литер зависят от символьного кода конкретной системы, это, как правило, – код ASCII.

Возможна обработка отдельных литер строки:

```
FOR i:=1 TO 5 DO
  shiver[i+5]:=shiver[i];
WRITELN(shiver)
```

⇒ OooohOoooh

Однако не все версии Паскаля допускают использование компонент *упакованного массива* в качестве параметров процедур (см. напротив). Например, WRITE(shiver[i]), возможно, придется заменить на ch:=shiver[i]; WRITE(ch).

Этих довольно ограниченных средств все же достаточно для создания набора мощных процедур строковой обработки, что демонстрируется в гл. 13.

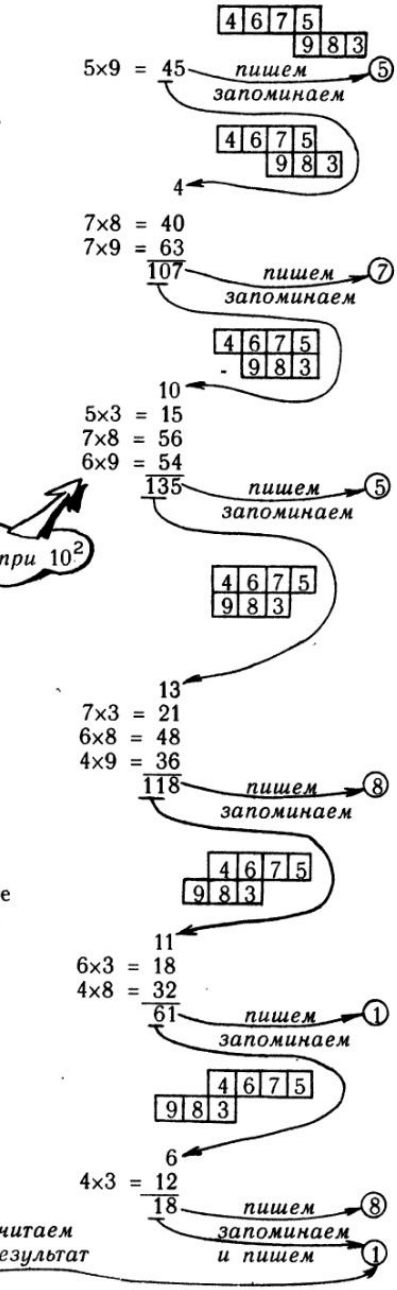
ФОКУС

ИЛЛЮСТРАЦИЯ ОБРАБОТКИ СТРОК КАК МАССИВОВ

Удивите своих друзей. Запишите пример на умножение длинных чисел, вроде этого, а затем начинайте записывать ответ, цифру за цифрой справа налево выполняя все действия в уме.

```
  4 6 7 5
x   3 8 9
---------
```

Фокус состоит в том, чтобы мысленно перевернуть нижнее число и постепенно сдвигать его влево под верхним числом. На каждом шаге перемножьте цифры, лежащие друг под другом и сложите результаты. Запишите последнюю цифру, а остальное запомните для последующего шага. Справа изображен весь процесс целиком.



Чтобы понять, как это все работает, рассмотрим каждое число как полином по степеням 10. На каждом шаге результаты перемножения лежащих одна над другой цифр относятся к одной степени 10. Более того, эти числа – *все* с данной степенью 10 (и еще, конечно, перенос от предыдущего шага).

$$4 \times 10 + 6 \times 10^2 + 7 \times 10^1 + 5 \times 10^0$$

$$9 \times 10^0 + 8 \times 10^1 + 3 \times 10^2$$

$$54 \times 10^2 + 56 \times 10^2 + 15 \times 10^2$$

это, например, – все коэффициенты при 10^2

Программа, изображенная напротив, автоматизирует описанный выше метод умножения. Эта программа может справится с любой разумной длиной сомножителей. Надо только подобрать константы termlimit и prodlimit. В приведенном варианте программа может перемножать числа длиной до 20 цифр. Длина результата при этом – до 40 цифр.

Чтобы воспользоваться программой, наберите два числа, разделив их звездочкой, а в конце поставьте знак равенства. Затем нажмите клавишу RETURN.

```
4675*389=
1818575

11111111111111111111*20000000000000000000=
22222222222222222220000000000000000000000
```

```
PROGRAM  parlour(INPUT,OUTPUT);

  CONST
    termlimit=20;  prodlimit=40;

  TYPE
    termspan  = 0..termlimit;
    prodspan  = 0..prodlimit;
    termtype  = PACKED ARRAY [termspan] OF CHAR;
    prodtype  = PACKED ARRAY [prodspan] OF CHAR;

  VAR
    a,b: termtype;  c: prodtype;  sum,offset: INTEGER;
    na,nb: termspan;  i,k: prodspan;

  PROCEDURE backhand(VAR x: termtype; VAR count: termspan);
    VAR
      i: INTEGER;  buffer: termtype;
    BEGIN
      i:=0;
      REPEAT
        READ(buffer[i]);
        i:=SUCC(i)
      UNTIL  (buffer[i-1]='*')
          OR  (buffer[i-1]='-');
      count:=i-2;
      FOR i:=0 TO count DO
        x[i]:=buffer[count-i];
    END;  { backhand }

  BEGIN { parlour }

    backhand(a,na);
    backhand(b,nb);
    sum:=0;
    offset:=ORD('0');

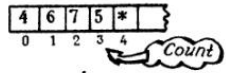
    FOR k:=0 TO na+nb DO
      BEGIN
        FOR i:=0 TO k DO
          IF (i<=na) AND ((k-i)<=nb)
            THEN
              sum:=sum+(ORD(a[i])-offset)*(ORD(b[k-i])-offset);
        c[k]:=CHR(sum MOD 10 + offset);
        sum:=sum DIV 10
      END;

    c[na+nb+1]:=CHR(sum + offset);
    IF sum=0 THEN i:=na+nb ELSE i:=na+nb+1;
    FOR k:=1 DOWNTO 0 DO
      WRITE(c[k]);
    WRITELN

END.  { parlour }
```

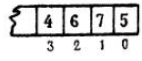
процедура backhand делает три действия:

(i) считывает число в буфер:

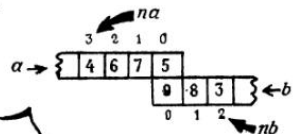


(ii) считает цифры начиная с 0

(iii) переворачивает цифры в x[ ]



сдвиги, как описано рядом



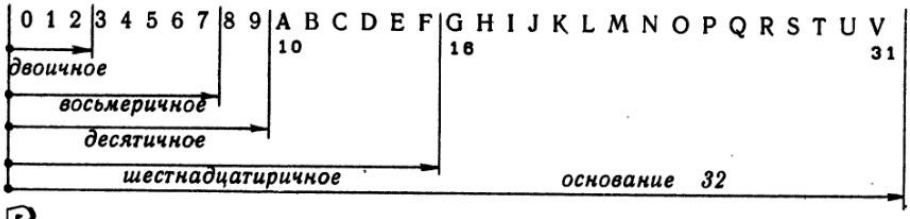
последний перенос

отбросим нули спереди

СИСТЕМЫ СЧИСЛЕНИЯ

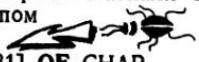
ЕЩЕ ОБ ОБРАБОТКЕ СТРОК КАК МАССИВОВ ЛИТЕР

Как было отмечено на предыдущей странице, десятичное число (основание 10) – полином по степеням десятки. Аналогично, шестнадцатеричное число (основание 16) – полином по степеням шестнадцати, восьмеричное число (основание 8) – полином по степеням восьми и т.д. Вообще, число в системе счисления с основанием b – это полином по степеням b и для представления таких чисел необходимо b цифр. Для цифр больше 9 используются прописные буквы; букв от A до V хватит для работы с основаниями вплоть до 32.



В следующей программе литеры от '0' до 'V' хранятся в строковой константе refconst, которая присваивается упакованному массиву литер с именем refstring. Этот массив используется двояко. По заданной литере, представляющей цифру (скажем, шестнадцатеричную), методом последовательного поиска может быть найдено соответствующее ей числовое значение – при совпадении индекс массива указывает на порядковое значение. Наоборот, используя порядковое значение как индекс массива, можно извлечь соответствующую литеру без всякого поиска.

На этих идеях основаны процедуры *find* и *outdigit* соответственно. К сожалению, некоторые версии Паскаля запрещают присваивать строковые константы массиву с типом



```
PACKED ARRAY [0..31] OF CHAR
```

требуя, чтобы нижняя граница была всегда единицей, например [1..32]. Сам индекс массива, таким образом, не может считаться порядковым значением и должен быть уменьшен на 1. Не слишком изящно.

Программа предназначена для того, чтобы считывать число, записанное в одной системе счисления, и печатать это число в записи по другому основанию. Например, если программе задать



то она преобразует 112D из *шестнадцатиричной* формы в *восьмеричную* и выдаст число 10455.

Сначала программа отыскивает порядковые значения цифр D,2,1,1 и вычисляет полином по степеням 16:

$$13\times16^0 + 2\times16^1 + 1\times16^2 + 1\times16^3 = 4397$$





Поиск осуществляется процедурой *find*, а полином вычисляется в процедуре *decimal*. Заметьте, что если в диапазоне данного основания найти соответствие не удается, то *find* возвращает –1. Если же процедура *decimal* получает –1 от *find*, то в основную программу процедура *decimal* возвращает нуль.

```
PROGRAM bases(INPUT,OUTPUT);

CONST
  stringlength=32;
  refconst='0123456789ABCDEFGHIJKLMNOPQRSTUV';

TYPE
  stringrange = 1..stringlength;
  stringtype = PACKED ARRAY [stringrange] OF CHAR;
  basetype = 2..32;
  number = 0..MAXINT;

VAR
  instring,outstring,refstring: stringtype;
  inlength,outlength: stringrange;
  ch: CHAR; dec,i: number;
  basenow,baserequired: basetype;

FUNCTION find(ch: CHAR; base: basetype): INTEGER;
  VAR
    found: BOOLEAN; i: number;
  BEGIN
   i:=1;
   REPEAT
    found:= (ch = refstring[i]);
    IF NOT found THEN i:= SUCC(i)
   UNTIL found OR (i>base);
   IF found
    THEN find:=i-1
    ELSE find:=-1
  END;

FUNCTION decimal(string: stringtype; length: stringrange;
                 base: basetype: INTEGER;
  VAR
   digit,power: INTEGER; n: number;
   i: stringrange; silly: BOOLEAN;
  BEGIN
   i:=0; silly:= FALSE; power: 1;
   FOR i:=length DOWNTO 1 DO
    BEGIN
     digit:=find(string[i],base);
     IF digit<0
      THEN
        silly:=TRUE
      ELSE
        BEGIN
          n:=n+digit*power;
          power:= power*base
        END
    END;
   IF silly
    THEN decimal:=0;
    ELSE decimal:=n;
  END;
```

в основной программе refconst присваивается массиву refstring

за границей данного основания поиск не ведется

например, если основание = 16, то «степень» (power) принимает значения 1, 16, 16^2, 16^3

продолжение на обороте

СИСТЕМЫ СЧИСЛЕНИЯ *(ПРОДОЛЖЕНИЕ)*

Для преобразования промежуточного десятичного результата к числу, выраженному в системе с новым основанием, программа использует деление на новое основание и запоминает остатки. Остатки – это порядковые значения цифр результата, записанные в обратной последовательности.

```
4397 : 8 ост. 5
 549 : 8 ост. 5
  68 : 8 ост. 4
   8 : 8 ост. 0
   1 : 8 ост. 1
   0
```

Цифры результата извлекаются из массива литер по порядковым значениям. Результат 10455, очевидно, совсем не требует обращения к массиву. Но при переводе к основанию, скажем, 32, порядковые значения были бы 4, 9, 13. Соответствующий поиск в массиве даст число 49D.

```
4397 : 32 ост. 13
 137 : 32 ост.  9
   4 : 32 ост.  4
   0
```

Преобразование к требуемому основанию выполняется процедурой *outdigit*. Чтобы справиться с обратным порядком вычисления цифр, используется рекурсия.

```
PROCEDURE outdigit(n: number; base: basetype);
VAR
    m: number; c: CHAR;
BEGIN
  m:= n DIV base;
  c:= refstring[1+(n MOD base)];
  IF m<>0
   THEN
     outdigit(m,base);
  WRITE(c)
END;
```

«поиск» цифры, например, 8 дает 8; 13 дает D

рекурсия для вывода цифр в обратном порядке

```
BEGIN  { программа }
  refstring:= refconst;
  i:=1;
  REPEAT
    READ(ch);
    instring[i]:=ch;
    i:=SUCC(i)
  UNTIL ch=' ';
  inlength:=i-2;
  READLN(basenow,baserequired);
  dec:=decimal(instring,inlength,basenow);
  outdigit(dec,baserequired);
  WRITELN

END.
```

ввод 112D 16 8

вывод 10455

10455 8 16

112D

УМНОЖЕНИЕ МАТРИЦ

ИЛЛЮСТРИРУЕТ ДВУМЕРНЫЕ МАССИВЫ В КАЧЕСТВЕ МАТРИЦ

Три продавца продают четыре вида товаров. Количество продаваемого сведено в таблицу A.

A

продавец	товар 1]	2]	3]	4]
[1,	5	2	0	10
[2,	3	5	2	5
[3,	20	0	0	0

B

товар	1] цена	2] комиссионные
[1,	1.20	0.50
[2,	2.80	0.40
[3,	5.00	1.00
[4,	2.00	1.50

В таблице B представлены цена каждого товара и коммиссионные, получаемые от продажи.

Вырученные от продажи деньги подсчитываются так:

```
продавец [1    5*1.50 +  2*2.80 +  0*5.00  +10*2.00 =  33.10
         [2    3*1.50 +  5*2.80 +  2*5.00 +  5*2.00 =  38.50
         [3   20*1.50 +  0*2.80 +  0*5.00 +  0*2.00 =  30.00
```

A полученные комиссионные так:

```
продавец [1    5*0.20 +  2*0.40 +  0*1.00  +10*0.50 =  6.80
         [2    3*0.20 +  5*0.40 +  2*1.00 +  5*0.50 =  7.10
         [3   20*0.20 +  0*0.40 +  0*1.00 +  0*0.50 =  4.00
```

Эти вычисления называются *умножением матриц* и лучше смотрятся в такой вот записи:

$$\begin{array}{c} \\ A[1, \\ A[2, \\ A[3, \end{array}\begin{array}{c} \begin{array}{cccc} 1] & 2] & 3] & 4] \end{array} \\ \begin{bmatrix} 5 & 2 & 0 & 10 \\ 3 & 5 & 2 & 5 \\ 20 & 0 & 0 & 0 \end{bmatrix} \end{array} * \begin{array}{c} \\ B[1, \\ B[2, \\ B[3, \\ B[4, \end{array}\begin{array}{c} \begin{array}{cc} 1] & 2] \end{array} \\ \begin{bmatrix} 1.50 & 0.20 \\ 2.80 & 0.40 \\ 5.00 & 1.00 \\ 2.00 & 0.50 \end{bmatrix} \end{array} = \begin{array}{c} \\ C[1, \\ C[2, \\ C[3, \end{array}\begin{array}{c} \begin{array}{cc} 1] & 2] \end{array} \\ \begin{bmatrix} 33.10 & 6.80 \\ 38.50 & 7.10 \\ 30.00 & 4.00 \end{bmatrix} \end{array}$$

число столбцов A должно быть таким же как

число строк B

а результат имеет столько строк, сколько у A, и столько столбцов, сколько у B

Следующая программа вводит матрицы A и B, перемножает эти матрицы и затем печатает их произведение – матрицу C:

```
PROGRAM sales(INPUT,OUTPUT);
  TYPE
    atype = ARRAY[1..3,1..4] OF INTEGER;
    btype = ARRAY[1..4,1..2] OF REAL;
    ctype = ARRAY[1..3,1..2] OF REAL;
  VAR
    a: atype; b: btype; c: ctype; n,i,j,k: INTEGER;
BEGIN
  FOR n:=1 TO 3 DO
   READLN(a[n,1],a[n,2],a[n,3],a[n,4]);
  FOR n:=1 TO 4 DO
   READLN(b[n,1],b[n,2]);
  FOR i:=1 TO 2 DO
    FOR j:=1 TO 3 DO
      BEGIN
        c[i,j]:=0;
        FOR k:=1 TO 4 DO
          c[j,i]:=c[j,i] + a[j,k]*b[k,i];
      END;
      FOR n:=1 TO 3 DO WRITELN(c[n,1]:8:2,c[n,2]:8:2)
END.
```

попробуйте эту программу со значениями A и B, что даны сверху

НАСТРАИВАЕМЫЕ ПАРАМЕТРЫ-МАССИВЫ

ГЛУБОКИЙ ВДОХ ...

Если умножение матриц выделить в процедуру, то программа с предыдущей страницы могла бы быть записана несколько иначе:

```
PROCEDURE matmul(VAR p: atype; VAR q: btype; VAR r: ctype);
  VAR
    i,j,k: INTEGER;
BEGIN
  FOR i:=1 TO 2 DO
    FOR j:=1 TO 3 DO
      BEGIN
        r[j,i]:=0;
        FOR k:=1 TO 4 DO
          r[j,i]:=r[j,i] + p[j,k]*q[k,i];
      END
END;
```

массивы p и q в этой процедуре не изменяются, тем не менее параметры, являющиеся массивами, всегда лучше делать параметрами-переменными. В противном случае при каждом обращении к подпрограмме массивы будут копироваться!

Главная программа тогда упростится:

```
BEGIN { программа }
  FOR n:=1 TO 3 DO
    READLN(a[n,1],a[n,2],a[n,3],a[n,4]);
  FOR n:=1 TO 4 DO
    READLN(b[n,1],b[n,2]);

  matmul(a, b, c);

      FOR n:=1 TO 3 DO WRITELN(c[n,1]:8:2,c[n,2]:8:2)
END.
```

вызов процедуры для массивов a,b,c

В такой программе необходимо, чтобы пределы изменения *i*, *j* и *k* в циклах FOR процедуры *matmul* соответствовали размерам массивов типов *atype*, *btype* и *ctype*, которые объявлены в разделе TYPE основной программы:

```
TYPE
  atype = ARRAY[1..3,1..4] OF INTEGER;
  btype = ARRAY[1..4,1..2] OF REAL;
  ctype = ARRAY[1..3,1..2] OF REAL
```

Но что делать, если обстоятельства вынуждают программиста изменить размерность этих массивов? Придется изменить и диапазон изменения *i*, *j* и *k* в циклах FOR процедуры *matmul*, чтобы установить соответствие новым размерностям, а это – источник возможных неприятностей.

Частично эта проблема решена в Паскале BS6192. Суть идеи: используемые в качестве параметров массивы (такие как *p*, *q* и *r* в процедуре *matmul*) объявлять как *настраиваемые массивы*. Настраиваемый массив способен *совмещаться* по размеру и типу компонент с массивом, объявленном во внешнем блоке ~ обычно в разделе TYPE основной программы. О том, что массив – *настраиваемый*, программист сообщает компилятору с Паскаля, специфицируя *настраиваемые параметры-массивы*. В начале следующей страницы процедура *matmul* переписана с использованием настраиваемых параметров-массивов.

не запятая, а точка с запятой

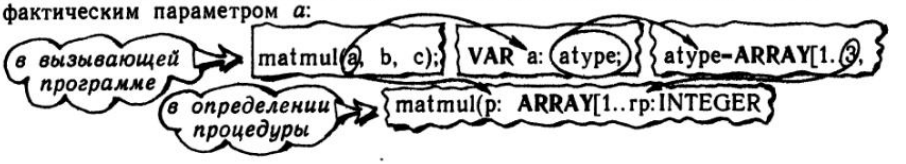
```
PROCEDURE matmul(VAR p: ARRAY[1..rp:INTEGER;1..cp:INTEGER]OF INTEGER;
                 VAR q: ARRAY[1..cp:INTEGER;1..cq:INTEGER]OF REAL;
                 VAR r: ARRAY[1..rp:INTEGER;1..cq:INTEGER]OF REAL);
  VAR
    i,j,k: INTEGER;
BEGIN
  FOR i:=1 TO cq DO
    FOR j:=1 TO rp DO
      BEGIN
        r[j,i]:=0;
        FOR k:=1 TO cp DO
          r[j,i]:=r[j,i] + p[j,k]*q[k,i];
      END
END;
```

это настраиваемый параметр-массив; массив r должен совмещаться по размерности, типу компонент и упаковке с любым массивом, передаваемым в качестве фактического параметра

Вызов процедуры *matmul* остается неизменным:

```
matmul( a, b, c);
```

Так как же *matmul* узнает значения *cq*, *rp* и *cp*? В этом и есть изюминка. Настраиваемые параметры-массивы передают в процедуру *matmul* достаточно информации, чтобы *matmul* могла подглядеть в вызывающей программе объявления этих массивов. Вот картинка для массива *p*, когда программа вызывает *matmul* с фактическим параметром *a*:



В случае совместимости массивов *p* и *a* (оба – двумерные, у обоих компоненты типа INTEGER, оба – неупакованные) каждое имя, такое как *rp*, сопоставляется со значением размерности, например 3.

Настраиваемее параметры-массивы *не* обеспечивают динамических границ массивов, они позволяют только автоматически устанавливать соответсвие с фиксированной размерностью из исходного объявления типа. Сложная возможность для достижения малого [4]. Лишь немногие версии Паскаля допускают настраиваемые параметры-массивы.

Динамические границы массива в ограниченных пределах можно промоделировать объявлением массивов большего размера и заданием параметров текущей размерности. Идея поясняется следующим фрагментом программы:

```
TYPE    atype = ARRAY[1..20,1..20];
PROCEDURE matmul(p: atype; q: btype; r: ctype; i,j,k: INTEGER);
  matmul( a, b, c, 2, 3, 4);
```

увеличенная размерность

вызов

размеры как параметры

Настраиваемые параметры-массивы позволили бы сообщить процедуре *matmul* лишь то, что максимально возможные размеры равны 20.

ВЫДОХ...!

УПРАЖНЕНИЯ

1. Введите программу *bubbles* с более существенным, нежели 30, размером константы *sizetype* ~ скажем 100 или 150. Затем измерьте время сортировки:

- когда входная последовательность задана случайными нажатиями клавиш без какой-либо системы
- когда входная последовательность в основном отсортирована:

  AAAABBCCCCCCDDE...

  но какая-либо буква, выпадает из последовательности:

  ...EZFFFGGGG...

2. Повторите предыдущее упражнение, используя вместо сортировки методом пузырька быструю сортировку (с. 97). Какие выводы можно сделать из результатов?

3. Взяв за основу программу *bases*, разработайте специальные процедуры для:

- преобразования чисел из *шестнадцатеричной* формы записи в *десятичную*
- преобразования чисел из *десятичной* формы записи в *шестнадцатеричную*

Избавившись от чрезмерной общности в программе *bases*, вы в конечном итоге получите две коротких, элегантных и полезных процедуры.

4. Если вы знакомы с матричной алгеброй, разработайте набор процедур, подобных процедуре *matmul*, для сложения, транспонирования и (крепкий орешек) обращения матриц. Используйте параметры для передачи текущей размерности как предлагается в конце предыдущей страницы.

ПРИМЕЧАНИЯ РЕДАКТОРА

[1] (с. 98) Для хранения литеры в подавляющем большинстве компьютеров отводится 8 бит; изредка используется 6-битовый код (БЭСМ-6, код TEXT); но никогда – 4 бит.

[2] (с. 98) В версии Турбо Паскаль нет никаких ограничений на использование компонент упакованных структур в качестве параметров процедур и функций ~ по очень простой причине: этот компилятор игнорирует слово **PACKED**.

[3] (с. 99) В версии Турбо Паскаль допускается сравнение упакованных массивов литер разной длины и, кроме того, имеется специальный строковый тип, позволяющий присваивать одной переменной последовательности литер разной длины.

[4] (с. 107) То принципиально новое, что обеспечивают настраиваемые массивы-параметры – это возможность с помощью *одной* подпрограммы обрабатывать массивы *разных* размеров.

9

ЗАПИСИ

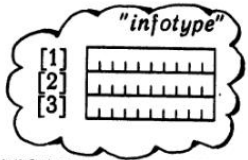
ЗНАКОМСТВО С ЗАПИСЯМИ
СИНТАКСИС ЗАПИСЕЙ
ПЕРСОНАЛЬНЫЕ ЗАПИСИ (ПРИМЕР)
ОПЕРАТОР WITH
ЧТО ТАКОЕ ВАРИАНТЫ

ЗНАКОМСТВО С ЗАПИСЯМИ

ЛИШЬ РАЗ УВИДЕВ ПРЕЛЕСТЬ ЗАПИСИ ...

В то время как *массив* – объединение компонент одинакового типа, *запись* – объединение компонент, вообще говоря, *различного* типа. Сравните следующий тип массива:

```
TYPE
  nametype = PACKED ARRAY[1..10] OF CHAR;
  infotype = ARRAY[1..3] OF nametype;
```



массив упакованных массивов

с этим вот типом записи:

```
TYPE
  nametype = PACKED ARRAY[1..10] OF CHAR;
  detailtype =
    RECORD
      surname,forename: nametype;
      age: 18..65;
      grade: (jr,sr,exec)
    END
```



запись с полями разного типа

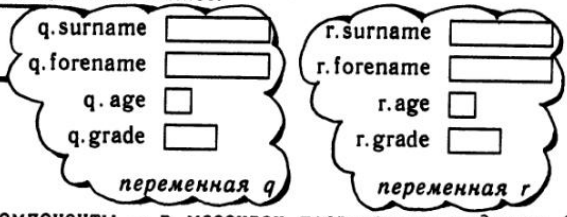
Переменная может содержать массив целиком:

```
VAR
  a,b: infotype;
```



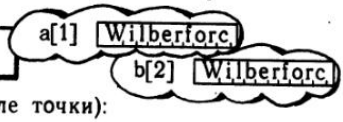
точно так же она может содержать целиком *запись*:

```
VAR
  q,r: detailtype;
```



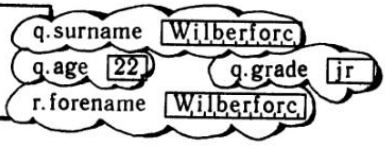
Похожим образом адресуются компоненты – в массивах посредством *индексов* (в квадратных скобках):

```
a[i]:= 'Wilberforc';   b[2]:= a[1];
```



а в записях – посредством *имени поля* (после точки):

```
q.surname:= 'Wilberforc';
q.age:= 22;  q.grade:= jr;
r.forename:= q.surname;
```



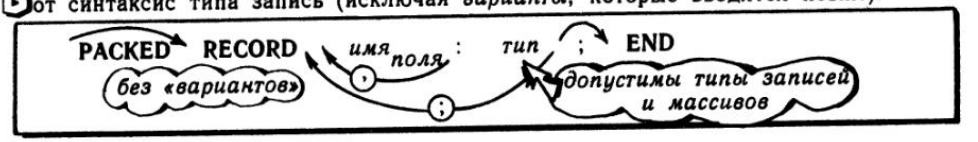
На рисунках показаны записи, в состав которых входят компоненты различных типов, в том числе упакованные массивы. С другой стороны, *записи* сами могут являться компонентами массива. Единственное ограничение на смешение типов в массивах и записях касается *массивов*: в каждом конкретном массиве *все* компоненты должны быть одного типа. Пример массива записей:

```
VAR people: ARRAY[1..100] OF detailtype
```

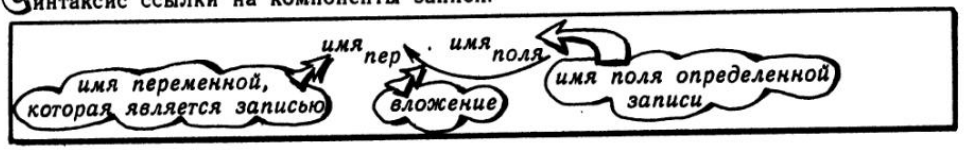
people теперь – массив из 100 записей

СИНТАКСИС ЗАПИСЕЙ

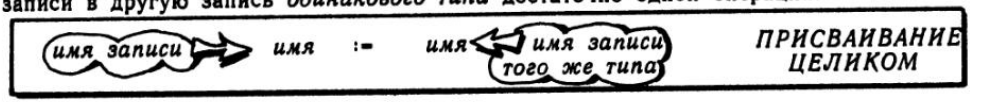
Вот синтаксис типа запись (исключая *варианты*, которые вводятся позже):



Синтаксис ссылки на компоненты записи:



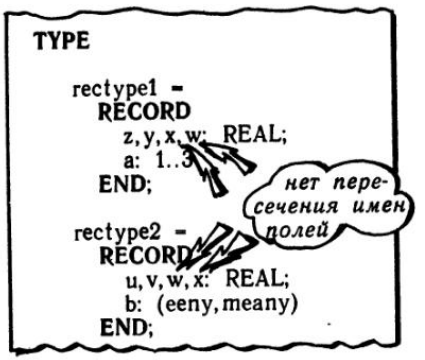
Для работы с массивами используются *индексы*; аналогом индексов в случае записей являются *имена полей*. Пододно массивам, правило покомпонентной обработки имеет важное исключение: для копирования *всего* содержимого одной записи в другую запись *одинакового типа* достаточно одной операции:

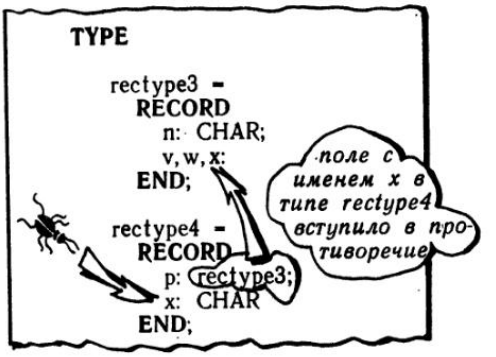


«Тот же» тип здесь означает тип с *совпадающим именем*; только совпадения спецификации недостаточно. Похожее требование относительно присваивания массивов целиком встречалось на с. 91.

Слово PACKED перед RECORD имеет тот же смысл, что и перед ARRAY. Упакованная запись занимает меньше места, нежели соответствующая неупакованная. Расплатой за это служит замедление обращения к компонентам во время выполнения программы. Процедуры PACK и UNPACK (с. 98) применимы только к массивам, но не к записям.

В каждом конкретном типе записи ~ включая все вложенные записи ~ имена полей должны различаться[1]. Однако в различных типах записей допускаются совпадающие имена полей:





ПЕРСОНАЛЬНЫЕ ЗАПИСИ

ПРИМЕР ИЛЛЮСТРИРУЕТ ИСПОЛЬЗОВАНИЕ ЗАПИСЕЙ

Эта программа запрашивает фамилию, имя, возраст и звание сотрудника. Каждый ответ заканчивайте нажатием на клавишу RETURN. Когда записей больше не останется, программа отсортирует все имеющиеся записи по каждому из четырех признаков:

- фамилия (алфавитный порядок)
- имя (алфавитный порядок)
- возраст (в порядке возрастания)
- звание (в порядке возрастания порядкового значения: JR,SR,EXEC)

```
Еще? (Д/Н): Д
Фамилия? (<= 10 букв): HAIG
Имя? (<= 10 букв): JOHN
Возраст? (от 18 до 65): 40
Звание? (JR,SR,EXEC): EXEC
Еще? (Д/Н): Д
Фамилия? (<= 10 букв): DAVIS
Имя? (<= 10 букв): SAMUEL
Возраст? (от 18 до 65): 64
Звание? (JR,SR,EXEC): JR
Еще? (Д/Н): Н
```

ВВОД

РЕЗУЛЬТАТ

по фамилии

```
DAVIS  SAMUEL  64  Junior
HAIG   JOHN    40  Executive
****
```

по имени

```
HAIG   JOHN    40  Executive
DAVIS  SAMUEL  64  Junior
****
```

по возрасту

```
HAIG   JOHN    40  Executive
DAVIS  SAMUEL  64  Junior
****
```

по званию

```
DAVIS  SAMUEL  64  Junior
HAIG   JOHN    40  Executive
```

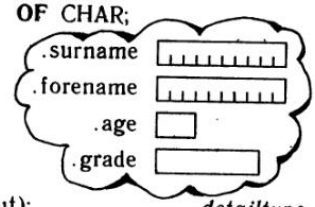
Проверки данных сведены к минимуму. Звание, отличное от JR, SR или EXEC, воспринимается как JR; ответ на «Еще?», отличный от Д, воспринимается как Н; прочие ошибки (например, имя длиннее 10 букв) обнаруживаются Паскаль-процессором.

Пример, который здесь приведен, подразумевает использование интерактивного Паскаль-процессора. Ряд возможных затруднений, обусловленных интерактивным вводом, обсуждается в гл. 11 .

Допустимая длина имени и допустимое число записей, из соображений удобства модификации, задаются константами. Запись о сотруднике (персональная запись) уже обсуждалась; ее тип воспроизведен в программе.С полями этой записи (*surname, forename, age* и *grade*) ассоциированы элементы перечисляемого ключевого типа (*lastname, firstname, decrepitude* и *clout*), который используется в процедуре сортировки для выделения соответствующего признака сортировки. Персональные записи хранятся в массиве *a*, соответствующие указатели – в массиве *p*. Указатели используются для сортировки, принцип которой описан на с. 94.

Вот объявления:

```
PROGRAM personel(INPUT,OUTPUT);
  CONST namelength = 10; listlength = 30; space = ' ';
  TYPE nametype = PACKED ARRAY[1..namelength] OF CHAR;
       detailtype =
        RECORD
          surname,forename: nametype;
          age: 18..65;
          grade: (jr,sr,exec)
        END
       indextype = 0..listlength;
       keytype = (lastname,firstname,decrepitude,clout);
       ordertype = (gt,eq);
```



```
VAR a: ARRAY[indextype] OF detailtype;   { массив персональных записей }
    p: ARRAY[indextype] OF indextype;    { указатели для сортировки }
    key: keytype;                        { признаки сортировки }
    count: indextype;                    { счетчик записей }
```

Главная программа приведена на следующем развороте. Главная программа (A) вызывает процедуру ввода (B), затем вызывает процедуру сортировки (C) и процедуру вывода (D) ~ каждую четыре раза. Для ввода строки литер процедура ввода (B) вызывает специальную процедуру (E); она также вызывает функцию (F) для проверки двух строк на равенство.



Процедура сортировки (C) также использует функцию (F), чтобы выяснить, какая из двух строк «больше». Во избежание использования директивы FORWARD эти подпрограммы должны располагаться так, чтобы (E) и (F) предшествовали (B), а (F) предшествовала (C). Основная программа (A) должна располагаться последней.

Далее приведены все эти подпрограммы. Процедура (E) принимает данные с клавиатуры:

```
PROCEDURE accept(VAR linebuf: nametype); { Прием }

  VAR  i: 0..namelength; ch: CHAR;

BEGIN
  FOR i:=1 TO namelength DO linebuf[i]:= space;   { заполнение буфера строки пробелами }
  REPEAT READ(linebuf[1]) UNTIL linebuf[1]<>space;   { пробелы в начале игнорируются }
  i:=1;                                          { первая литера в linebuf[1] }
  WHILE NOT EOLN DO
    BEGIN
      i:=i+1;
      READ(linebuf[i]);                          { продолжать с linebuf[2] }
    END;
  READLN
END;
```

Функция (F) сравнивает строки на равенство или на больше-меньше:

```
FUNCTION order(c: ordertype; a,b: nametype): BOOLEAN; { Сравнение }

  VAR  i: 0..namelength; c1,c2,null: CHAR;

BEGIN
  i:=0; null:= CHR(0);   { невидимая литера с меньшим порядковым значением, чем у любой буквы или цифры }
  REPEAT                 { глобальная константа }
    i:=i+1;
    IF a[i]=space THEN c1:= null ELSE c1:=a[i];
    IF b[i]=space THEN c2:= null ELSE c2:=b[i];
  UNTIL ((i=namelength) OR (c1<>c2)) OR ((c1=null) AND (c2=null));
  CASE c OF
    gt:   order:= (c1>c2);
    eq:   order:= (c1=c2)
  END { CASE }
END;
```

ПЕРСОНАЛЬНЫЕ ЗАПИСИ *(ПРОДОЛЖЕНИЕ)*

Процедура сортировки (C) использует описанный ранее метод пузырька, который теперь приспособлен для работы с различными ключами сортировки. Каждый ключ подразумевает свой критерий сортировки. Разветвление по ключу осуществляется оператором CASE, структура которого напоминает структуру персональной записи.

```
PROCEDURE sort(n: indextype; k: keytype); { Сортировка }
  VAR s,sorted: BOOLEAN; i,tempry: indextype;
  BEGIN
    IF n>1 THEN
      BEGIN
        sorted:= TRUE;
        FOR i:=1 TO n-1 DO
          BEGIN
            CASE k OF                          { выбирает, по какому из четырех ключей сортировать }
              lastname:
              s:=order(gt,a[p[i]].surname,a[p[i+1]].surname);   { алфавитные ключи }
              firstname:
              s:=order(gt,a[p[i]].forename,a[p[i+1]].forename);
              decrepitude:
              s:=a[p[i]].age>a[p[i+1]].age;                     { числовой ключ }
              clout:
              s:=ORD(a[p[i]].grade)>ORD(a[p[i+1]].grade)        { порядковый ключ }
            END; { CASE }
            IF s THEN                          { s равно TRUE или FALSE }
              BEGIN
                sorted:= FALSE;                { если TRUE, то меняем указатели }
                tempry:= p[i];
                p[i]:=p[i+1];
                p[i+1]:= tempry
              END
          END; { FOR i }
          IF NOT sorted THEN sort(n-1,k)       { рекурсивное обращение }
      END  { IF n>1 }
END;
```

Дальше следует бесхитростный текст процедуры (D):

```
PROCEDURE list(n: indextype); { Печать }
  VAR i: indextype;
  BEGIN
    FOR i:=1 TO n DO
      BEGIN  { FOR i }
        WRITE(a[p[i]].surname,space);
        WRITE(a[p[i]].forename,space);
        WRITE(a[p[i]].age:3,space);
        CASE a[p[i]].grade OF
          jr: WRITELN('Junior');
          sr: WRITELN('Senior');
          exec: WRITELN('Executive');
        END  { CASE }
      END  { FOR i}
END;
```

].surname CANDLEWICK
].forename JOSIAH
].age 19
].grade jr
ПРИМЕР

не забывайте, что значения перечисляемого типа нельзя выводить на печать поэтому здесь используется оператор CASE

Даже несмотря на отсутствие проверок, написание процедуры (B) является наиболее утомительным. Таковы процедуры ввода в любом языке.

Если ваша программа икает, запрашивая данные, которые уже были введены (см. гл. 11), то можно убрать все запросы и создать файл с исходными данными. Посмотрите в вашем руководстве, как следует создавать, редактировать и сохранять файл данных для последующего его чтения в программе на Паскале.

```
PROCEDURE inputter(VAR n: indextype); { Ввод }

  VAR  indicator: CHAR;
       string: nametype;
       buffer: detailtype;

  BEGIN
     n:=0;
     REPEAT
       WRITE('Еще? (Д/Н): ');
       READLN(indicator);
       IF indicator = 'Д'
          THEN
            BEGIN
               n:=n+1;
               p[n]:=n;
               WRITE('Фамилия? (<=10 букв): ');
               accept(string); buffer.surname:=string;
               WRITE('Имя? (<=10 букв): ');
               accept(string); buffer.forename:=string;
               WRITE('Возраст? (от 18 до 65): ');
               READLN(buffer.age);
               WRITE('Звание? (JR,SR,EXEC): ');
               buffer.grade:=jr;
               accept(string);
               IF order(eq,string,'EXEC      ')THEN buffer.grade:=exec;
               IF order(eq,string,'SR        ')THEN buffer.grade:=sr;
               a[n]:=buffer
            END
          ELSE
            IF indicator='Н'     THEN WRITELN('Нормальное завершение')
                                 ELSE WRITELN('Ненормальное завершение');
     UNTIL indicator<>'Д';
  END; { inputter }
```

indicator = 'Д', если Да

string

buffer.surname
buffer.forename
buffer.age 18..65
buffer.grade (JR,SR,EXEC)

p[1] 1
p[2] 2
p[3] 3
p[4] 4

присваивание компоненте a[n] целиком записи «buffer»

Главная программа (A) – простая:

```
BEGIN

  inputter(count);

  FOR key:= lastname TO clout DO
    BEGIN
      sort(count,key);
      list(count);
      WRITELN('****');
    END
END.  { PROGRAM }
```

ГЛАВНАЯ ПРОГРАММА

ключ сортировки пробегает в цикле все четыре поля записи

ОПЕРАТОР WITH

ЭКОНОМИТ ВРЕМЯ И ЧЕРНИЛА

Обратите внимание на повторяющийся элемент a[p[i]]. (*точка*) в тексте процедуры на с. 114. Строчки отличаются, главным образом, именем поля, которое следует за точкой.

```
WRITE(a[p[i]].surname;
WRITE(a[p[i]].forename
WRITE(a[p[i]].age
CASE a[p[i]].grade
```

имена полей

Оператор WITH приписывает единожды указанное имя записи (до точки) к последующим именам полей с тем, чтобы операторы, как, например изображенные сверху, можно было сократить до возможно меньших размеров. Вот снова эти же операторы, но уже в законченном виде и с использованием WITH.

```
WITH a[p[i]] DO
  BEGIN
     WRITE(surname,space);
     WRITE(forename,space);
     WRITE(age,space);
     CASE (grade) OF
  END { WITH }
```

точки нет

слово WITH относится ко всем именам полей внутри составного оператора, следующего за WITH...DO

Синтаксис оператора WITH:

WITH *переменная* **DO** *оператор*

заметьте, что «DO» здесь не означает повторения, как в циклах FOR и WHILE

где

переменная := *имя* [*выражение*] *.имя*

пример: a[p[i]]
пример: nest.field1.field2
еще допускается ↑, *см. гл. 12*

Ниже приведен раздел объявлений, который потребуется в программе напротив. Эта программа демонстрирует использование оператора WITH применительно к вложенным записям:

```
TYPE
  nesttype = RECORD
               field1 = RECORD
                          field2 = RECORD
                                     field3 = BOOLEAN
                                   END
                        END
             END;
VAR
  nest: nesttype;
```

NEST **field1** field2 field3 TRUE

Первый из приведенных ниже примеров показывает, что оператор WITH может достигать любого уровня вложения. (Имеет ли вложение «уровни»? Быть может, лучше было бы говорить о «слоях», но «уровни» – общепринятый термин.)

```
PROGRAM nesting(OUTPUT);
    здесь поместите объявления TYPE и VAR
BEGIN
      nest.field1.field2.field3:= TRUE;
    WITH nest.field1.field2 DO WRITELN(field3);
    WITH nest.field1 DO WRITELN(field2.field3);
    WITH nest DO WRITELN(field1.field2.field3)
END.
```

```
TRUE
TRUE
TRUE
```

Следующий пример иллюстрирует вложенные операторы WITH, отражающие структуру вложенных записей:

```
PROGRAM nesting2(OUTPUT);
    здесь поместите объявления TYPE и VAR
BEGIN
      nest.field1.field2.field3:= TRUE;
    WITH nest DO
     WITH field1 DO
      WITH field2 DO
        WRITELN(field3)
END.
```

```
TRUE
```

Третий пример призван проиллюстрировать использование запятых вместо точек. Такие обозначения позволяют указывать более одного типа записи. Этому, однако, может помешать то, что компилятор не всегда будет знать, к какой записи относится каждое поле (вспомните, что в разных записях возможны одноименные поля). Запятые – не более чем альтернатива точкам. Сравните следующую программу с программой вверху страницы.

```
WITH  переменная  DO
        ,
```

```
PROGRAM nesting3(OUTPUT);
    здесь поместите объявления TYPE и VAR
BEGIN
      nest.field1.field2.field3:= TRUE;
    WITH nest,field1,field2 DO WRITELN(field3);
    WITH nest,field1 DO WRITELN(field2.field3);
    WITH nest DO WRITELN(field1.field2.field3)
END.
```

```
TRUE
TRUE
TRUE
```

Запятая имеет смысл только после WITH; не пытайтесь писать:

```
WITH nest DO WRITELN(field1,field2,field3);
```

и не переставляйте поля после WITH:



```
WITH field2,nest,field1 DO WRITELN(field3);
```



ЧТО ТАКОЕ ВАРИАНТЫ

Рассмотрим программу для управления транспортными ресурсами, рассчитанную на случай забастовки железнодорожников и водителей автобусов. Наличие транспортных средств можно описать такой записью:

```
PROGRAM carshare(INPUT,OUTPUT);
  TYPE
    modetype = (foot, pushbike, motorbike, car);
    gotype = RECORD
               surname: PACKED ARRAY[1..10] OF CHAR;
               initial: CHAR;
               mode: modetype;
               year: 1900..1990;
               sidecar: BOOLEAN;
               mpg: REAL;
               seats: 1..6;
             END;
  VAR
    person: gotype; people ARRAY[1..100] OF gotype;
    i: 1..100;
```

пешеход *велосипед* *мотоцикл* *автомобиль*

фамилия .surname
инициал .initial
вид .mode
год .year
коляска .sidecar
расх. гор. .mpg
места .seats

Эту запись надо заполнять внимательно, потому что некоторые поля не всегда имеют смысл; например, пешеход не расходует горючее и у него нет мест для пассажиров. Значение поля *mode* в каждом случае определяет набор существенных полей. Таким образом, для заполнения или печати записей естественно использовать оператор CASE. Например:

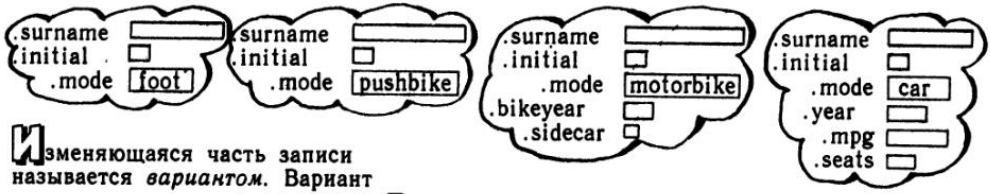
```
WITH people[i] DO
  BEGIN
    WRITELN(initial,surname:11);
    CASE mode OF
      foot, pushbike: ;
      motorbike: BEGIN WRITE('Мотоцикл сделан в',year);
                   IF sidecar THEN WRITELN('1 место в коляске')
                              ELSE WRITELN('сиденье сзади')
                 END;  { motorbike }
      car: WRITELN(year:4,mpg:4:1,' салои на',seats-1:2)
    END  { CASE mode }
  END; { WITH }
```

этот оператор WRITE – общий для всех видов транспорта

пустой оператор

обычный оператор CASE

Такое решение, однако, не свободно от проблем: размер каждой записи должен быть достаточным для хранения всех вообще возможных полей. Память расходуется попусту. В реальных программах потери могут быть очень велики. Поэтому Паскаль позволяет варьировать структуры от записи к записи так, как на этом рисунке:



Изменяющаяся часть записи называется *вариантом*. Вариант всегда располагается *в конце*. Поле (в данном случае *.mode*), позволяющее различать варианты, называется *полем признака*.

Для описания вариантов применяется специальный оператор. Его имя CASE, хотя он и отличается от *управляющего* оператора с тем же именем. Однако вполне очевидно и сходство между двумя операторами. Приведем новое определение типа gotype:

```
TYPE
   modetype =
   gotype = RECORD
               surname: PACKED ARRAY[1..10] OF CHAR;
               initial: CHAR;
               CASE mode: modetype OF
                  foot, pushbike: ();
                  motorbike: (bikeyear:1900..1990; sidecar: BOOLEAN);
                  car: (year:1900..1990; mpg:REAL; seats:1..6)
               END { RECORD }
```

у CASE нет закрывающего END

пустое определение

заметьте, что вместо «year» использовано имя «bikeyear», чтобы сделать все имена полей в записи различными

Выше определен тип записи, который нарисован во всех возможных видах в конце страницы напротив.

Обращение к новой записи не упростилось; оно даже стало сложнее, потому что теперь в ней используются *различные* компоненты для хранения года изготовления (при модификации программы напротив измените WRITE('Мотоцикл сделан в', year) на WRITE('Мотоцикл сделан в',bikeyear)). Оператор CASE по-прежнему необходим, чтобы пешеходам и велосипедистам не пришлось перевозить пассажиров.

Синтаксис *варианта* рекурсивно определяется следующим образом:



Обратите внимание, у CASE нет закрывающего END. Так как вариант должен следовать в конце, считается, что вариант и вся запись закрываются одним общим словом END.

Заметьте, что конструкция (поля вариант) допускает отсутствие обоих элементов. Такая пустая пара скобок означает пустое определение полей (что и используется в примере выше). С другой стороны, присутствие *варианта* открывает возможность появления еще CASE, позволяя описывать вложенные варианты. А так как *поля* в любом варианте могут опускаться, то отсюда вытекает, что правило о следовании варианта в конце не накладывает никаких ограничений на сложность.

Пропуск *имени:* подразумевает отсутствие поля признака, служащего для распознавания вариантов. Такая запись называется *свободным объединением* (в отличие от *размеченного объединения* при наличии поля признака). Свободное объединение допускает хранение элемента, например, под видом литеры с последующим извлечением его как целого числа ~ аналогично и для других случаев «эквивалентности» типов. Свободное объединение, придуманное для чтения по указателям (смешно) предложено Грогоно одновременно с соответствующими предостережениями против такой практики. Смотри библиографию.

УПРАЖНЕНИЯ

1. Выполните программу personel. Модернизируйте программу, определив более реалистичные записи.

2. Напишите процедуру быстрой сортировки, чтобы заменить ею процедуру сортировки методом пузырька на с. 114. Сортирует ли она записи сколько-нибудь быстрее? (Объем данных в этом упражнении так невелик, что ни одна процедура сортировки не имеет преимуществ над другими. Лучшей поэтому будет наипростейшая процедура.)

ПРИМЕЧАНИЯ РЕДАКТОРА

[1] (с. 111) Стандарт Паскаля допускает такое пересечение имен, как в примере справа внизу. Действительно, оно не приводит ни к каким неоднозначностям: если переменная *r* имеет тип *rectype4*, то поле *x* на ее верхнем уровне обозначается как r.x, тогда как поле с тем же именем во вложенной записи будет обозначено r.p.x .

[2] (с. 117) В перечислении записей между WITH и DO запятая совсем не эквивалентна точке. Запятая служит сокращением для вложенных операторов WITH. Оператор **WITH** a,b **DO**... эквивалентен **WITH** a **DO WITH** b **DO** ... ; таким образом, во внутреннем операторе можно использовать компоненты обеих записей. При этом запись *b* вовсе не обязательно должна быть полем в записи *a*. Оператор **WITH** a.b **DO** «присоединяет» к внутреннему оператору только одну запись – *a.b* .

10

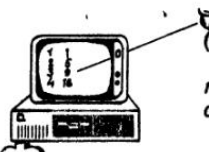
ФАЙЛЫ

ЧТО ТАКОЕ ФАЙЛЫ
ОТКРЫТИЕ ФАЙЛОВ
ТЕКСТОВЫЕ ФАЙЛЫ
ПРОЦЕДУРЫ *WRITE* И *WRITELN* ДЛЯ ТЕКСТОВЫХ ФАЙЛОВ
ПРОЦЕДУРА *PAGE* ДЛЯ ТЕКСТОВЫХ ФАЙЛОВ
ПРОЦЕДУРЫ *READ* И *READLN* ДЛЯ ТЕКСТОВЫХ ФАЙЛОВ
БЕЗОПАСНОЕ ЧТЕНИЕ
ПРОЦЕДУРА *GRAB* ДЛЯ БЕЗОПАСНОГО ЧТЕНИЯ
ДВОИЧНЫЕ ФАЙЛЫ, ПРОЦЕДУРЫ *PUT* И *GET*
СЖАТИЕ (ПРИМЕР)
СВОЙСТВА ФАЙЛОВ, СВОДКА

ЧТО ТАКОЕ ФАЙЛЫ

СРЕДСТВО СВЯЗИ МЕЖДУ ПРОГРАММАМИ

Файл с именем OUTPUT уже встречался. Имя OUTPUT, будучи опущено в операторе WRITE (или WRITELN), *подразумевается*, но при желании его можно явно указать




```
PROGRAM squares(OUTPUT);
 VAR i: INTEGER;
BEGIN
 FOR i:=1 TO 4 DO
   WRITELN(OUTPUT,i,SQR(i))
END.
```

имя файла существенно

имя файла необязательно

Файл с именем INPUT также уже встречался. Аналогично имя INPUT *подразумевается* в процедурах READ, READLN и функциях EOF, EOLN, но может быть и явно указано:

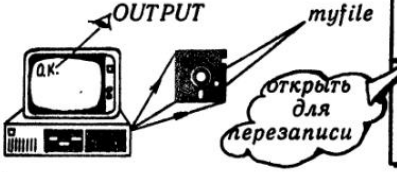



```
PROGRAM anysquares(INPUT,OUTPUT);
 VAR i,j,k: INTEGER;
BEGIN
 WRITELN(OUTPUT,'диапазон:');
 READLN(INPUT,i,k);
 FOR i:=j TO k DO
   WRITELN(OUTPUT,i,SQR(i))
END.
```

имена файлов существенны

имя файла необязательно

Результаты работы можно отправить и в другие файлы, не только в файл с именем OUTPUT. Каждый такой файл должен быть указан в операторе PROGRAM, а его *тип* объявлен в разделе VAR. Однако файл OUTPUT должен быть указан всегда, хотя бы в качестве канала для сообщений ~ в частности, сообщений Паскаль-процессора об ошибках:



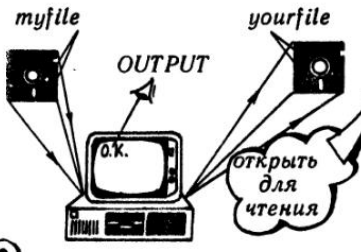

```
PROGRAM filesquares(OUTPUT,myfile);
 VAR i: INTEGER; myfile: TEXT;
BEGIN
 REWRITE(myfile);
 FOR i:=1 TO 4 DO
   WRITELN(myfile,i,SQR(i));
 WRITELN(OUTPUT,'O.K.')
END.
```

тип файла – текстовый

имя файла

имя файла существенно

В качестве источника данных могут использоваться файлы, отличные от файла с именем INPUT. Каждый такой файл должен быть указан в операторе PROGRAM, а его *тип* объявлен в разделе VAR:



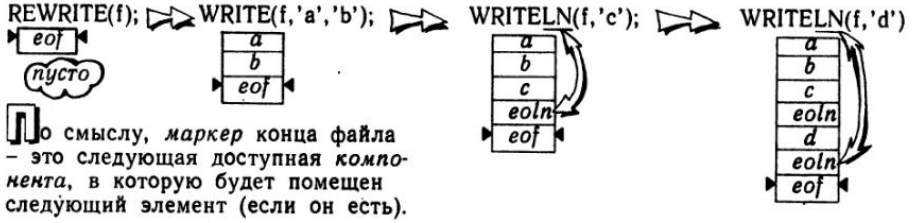

```
PROGRAM filecubes(OUTPUT,myfile,yourfile);
 VAR i,j: INTEGER; myfile,yourfile: TEXT;
BEGIN
 RESET(myfile); REWRITE(yourfile);
 WHILE NOT EOF(myfile) DO
  BEGIN
    READLN(myfile,i,j);
    WRITELN(yourfile,i,i*j)
  END;
  WRITELN(OUTPUT,'o.k.)
END.
```

тип файла

Одновременно могут быть открыты несколько файлов; с другой стороны, в ходе выполнения одной программы один и тот же файл может быть открыт для записи и впоследствии установлен на чтение.

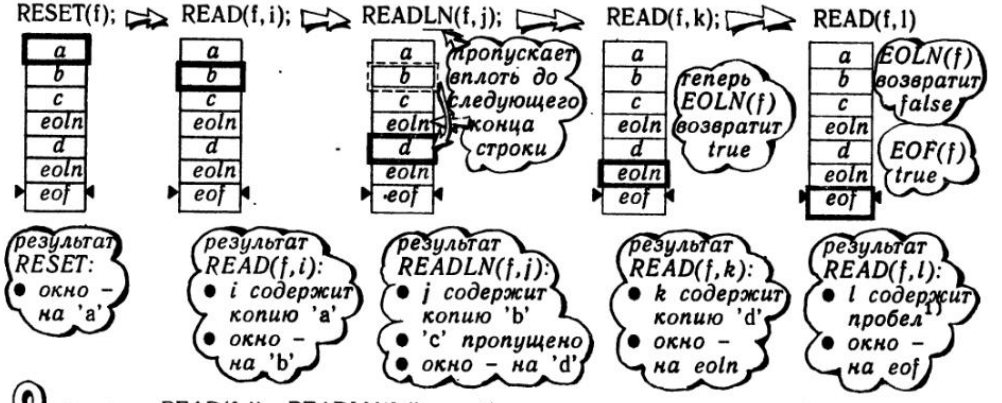
Обратите внимание, что файлы myfile и yourfile перед записью должны быть открыты процедурой REWRITE, а перед чтением – процедурой RESET. Однако для открытия специальных файлов с именами OUTPUT и INPUT использовать REWRITE и RESET *нельзя* (эти файлы открываются автоматически). Ошибкой также будет попытка открыть уже открытый файл.

В ISO Паскале все файлы – *последовательные*. Открытый на запись файл изначально является пустым, он содержит лишь маркер конца файла. Каждый оператор WRITE или WRITELN осуществляет добавление новой информации, после чего маркер сдвигается к новому концу файла. Оператор WRITELN (в отличие от WRITE), перед тем как вернуть управление, добавляет в файл еще и литеру конца строки.



По смыслу, *маркер* конца файла – это следующая доступная *компонента*, в которую будет помещен следующий элемент (если он есть).

Открытый для чтения файл имеет «окно», расположенное над первой компонентой. Выполнение первого оператора READ приводит к считыванию элемента в окне, после чего окно сдвигается к следующей компоненте и так далее. Оператор READLN (в отличие от READ), перед тем как вернуть управление, сдвигает окно *за ближайшую литеру конца строки*. Если такой литеры нет, то окно устанавливается на маркер конца файла.



Операторы READ(f,i), READLN(f,j), изображенные выше, можно объединить в один оператор – READLN(f,i,j). Вообще:

READLN(f,p,q,r,...) ≡ READ(f,p);READ(f,q);READ(f,r);...READLN(f)
WRITELN(f,p,q,r,...) ≡ WRITE(f,p);WRITE(f,q);WRITE(f,r);...WRITELN(f)

Литера конца строки имеет рассмотренный выше специальный смысл только в текстовых файлах наподобие тех, что изображены напротив. Текстовый файл в соответствие с названием состоит из слов и чисел, которые разделены пробелами и объединены в строки. Нже рассматриваются файлы другого вида – *двоичные*.

В следующей главе речь пойдет об изменениях логики работы с файлами в случае *интерактивного* ввода.

ОТКРЫТИЕ ФАЙЛОВ

Каждый файл, используемый в программе на Паскале для чтения или записи, должен быть указан в операторе PROGRAM:

PROGRAM *имя*$_{прог}$ (*имя*$_{файла}$) ;

▸ **PROGRAM** myprog(OUTPUT, mydata, mydump);

указывается всегда

Тип каждого файла должен быть объявлен в разделе VAR главной программы. Синтаксис объявления приведен ниже. Определение FILE OF REAL упреждает события, это – *двоичный файл;* речь о них – впереди.

VAR *имя*$_{файла}$: *тип* ;

не включайте сюда файлы с именами INPUT или OUTPUT, которые по умолчанию определены как файлы типа TEXT

▸ **VAR** mydata: TEXT; mydump: **FILE OF** REAL

Запись в файл, если это не файл OUTPUT, может вестись только после того, как этот файл открыт процедурой REWRITE:

REWRITE(*имя*$_{файла}$)

в некоторых системах возможны расширения

▸ REWRITE(mydump)

не исполняйте REWRITE с файлом OUTPUT

Чтение при помощи операторов READ или READLN из файла, если это не файл INPUT, может производиться только после того, как этот файл открыт процедурой RESET:

RESET(*имя*$_{файла}$)

в некоторых системах возможны расширения

▸ RESET(mydata)

не исполняйте RESET с файлом INPUT

Процедуры WRITE, WRITELN, READ и READLN подробно обсуждаются на следующем развороте.

Приведенные выше определения применимы как к текстовым, так и к двоичным файлам. Двоичные файлы будут рассматриваться несколько позже.

ТЕКСТОВЫЕ ФАЙЛЫ

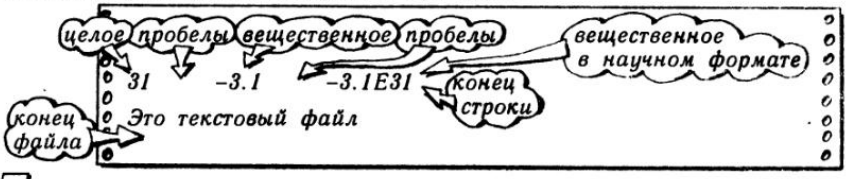
ПОСМОТРИТЕ В СВОЕМ РУКОВОДСТВЕ, КАК ВВОДИТЬ, РЕДАКТИРОВАТЬ И СОХРАНЯТЬ ФАЙЛЫ

Файлы с именами INPUT и OUTPUT – текстовые файлы (файлы типа TEXT). Файлы, указаные программистом, также могут быть объявлены как текстовые.

```
PROGRAM(INPUT,OUTPUT,hisfile,herfile);
  VAR hisfile,herfile: TEXT;
```

текстовые файлы, объявленные программистом

Текстовый файл состоит из литер в коде ASCII ~ или из литер того кода, который используется на вашем компьютере ~ таким образом, напечатанный текстовый файл можно воспринять зрительно:



Текстовый файл организован в виде строчек элементов, элементы разделены пробелами. Программа на Паскале, предназначенная для чтения такого файла, может считывать файл по одной литере используя только оператор READ(f,ch) (*ch* – типа CHAR). С другой стороны, в операторе READ можно использовать несколько параметров; каждый параметр того же типа, как предполагаемый очередной элемент файла. Например, оператор READLN(file,i,x,y) правильно прочитает первую строку текстового файла, который изображен выше. Здесь *i* типа INTEGER, *x,y* – типа REAL.

Отличительным свойством текстовых файлов является автоматическое преобразование элементов из внутренней формы представления в символьную в процедуре WRITE и из символьной во внутреннюю в процедуре READ. Это преобразование выполняется в соответствии с *типом* используемых параметров. Если соответствия типов нет, то чтение прерывается, поэтому безопаснее отказаться от автоматического преобразования и считывать данные по одной литере. Соответствующая процедура ввода приведена на с. 128–133.

Текстовые файлы могут создаваться процедурой WRITE, это было показано на с. 122. Текстовые файлы можно также вводить с клавиатуры и сохранять на диске. Как это делается, зависит от вашей системы; посмотрите свое руководство. Обычно файл вводится под управлением «строчного редактора» или «экранного редактора». Редактор позволяет вводить, исправлять, вставлять и удалять текст. Когда файл введен и отредактирован, он может быть сохранен на диске для последующего использования в качестве файла INPUT в программе на Паскале. Для этого используется команда наподобие:

```
SAVE 'INPUT'
```

возможно, без кавычек

Для многих компиляторов недостаточно указать файлы в операторе PROGRAM; необходимо еще связать эти имена с именами файлов, которые понимает операционная система. Для этих целей в Pro Паскале и Turbo Паскале используется процедура ASSIGN; в Acornsoft ISO Паскале – расширенные процедуры RESET и REWRITE:

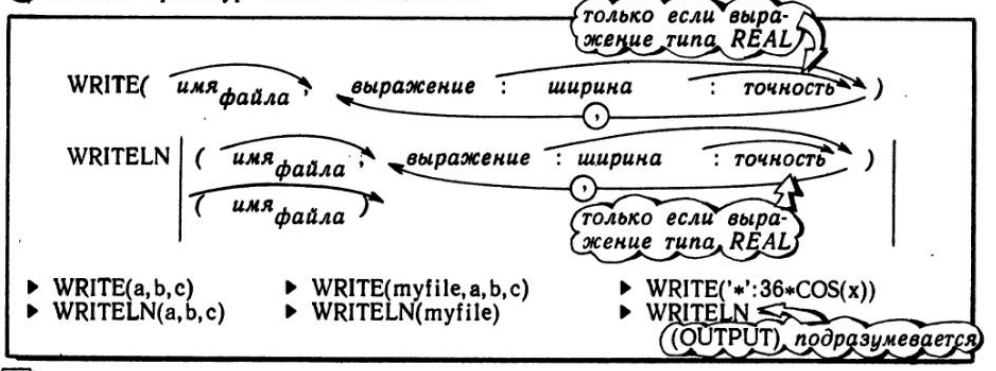
имя файла для Паскаля

```
ASSIGN(mydata2,'MYDAT2.TX')
RESET(mydata2,'MYDAT2.TX')
```

имя файла для операционной системы

ПРОЦЕДУРЫ WRITE и WRITELN ДЛЯ ТЕКСТОВЫХ ФАЙЛОВ

Синтаксис процедур WRITE и WRITELN:



Первый выполняемый оператор WRITE или WRITELN располагает первое поле вывода в начале выходного файла. (*Поле* – последовательность смежных литерных позиций, в которые записывается выводимый элемент ~ сдвинутый к правому краю). Следующие поля, порождаемые той же или следующими процедурами WRITE и WRITELN, добавляются в выходной файл последовательно и вплотную друг к другу.

Если *ширина* поля вещественного или целого типов не указана, то используется некоторое стандартное значение, зависящее от системы (обычно это 14 позиций). Не указанная точность (число *знаков* после десятичной точки) для поля типа REAL подразумевает вывод в «научном» формате; например, значение -0.000123456 представляется в виде -1.23456E-04. Число значащих цифр, которые печатаются перед E, зависит от системы (обычно 6 или 9). Для строк не заданная *ширина* подразумевает ширину, равную числу литер в строке, не считая открывающего и закрывающего апострофов (3 для 'abc'). Для элемента типа PACKED ARRAY[1..n] OF CHAR при отсутствии *ширины* используется *n*. Для логического элемента подразумеваемая *ширина*, зависит от системы (обычно 4 для *true* и 5 для *false*). Если заданное значение *ширины* слишком мало, чтобы вместить соответствующий элемент, то поле растягивается вправо.

Процедура WRITELN (в отличие от WRITE) после записи последнего параметра автоматически добавляет литеру конца строки. Процедура WRITELN без параметров также добавляет в файл литеру конца строки.

ПРОЦЕДУРА PAGE ДЛЯ ТЕКСТОВЫХ ФАЙЛОВ И ТОЛЬКО ДЛЯ НИХ

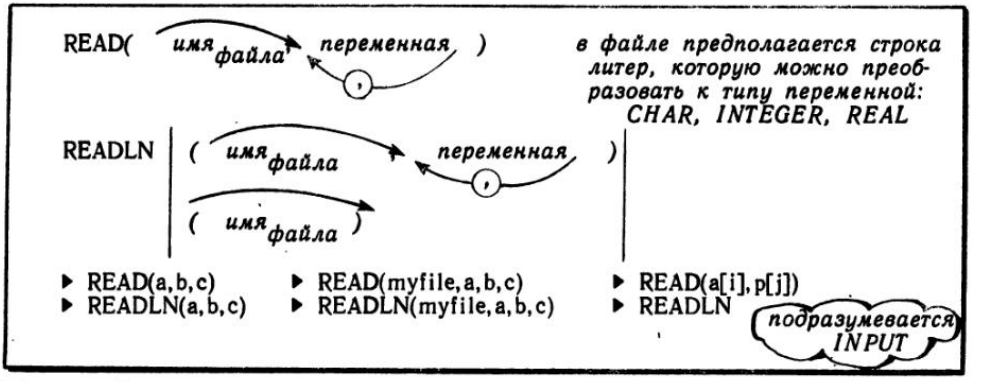
Синтаксис процедуры PAGE:



При вызове этой стандартной процедуры в указанный или подразумеваемый выходной файл посылается признак конца страницы (можно использовать, только если система распознает этот признак).

ПРОЦЕДУРЫ READ и READLN для ТЕКСТОВЫХ ФАЙЛОВ

Синтаксис процедур READ и READLN:



Если параметр имеет тип CHAR, то считывается одна литера в окне. Если это – литера конца строки, то она читается так, как если бы это был *пробел*. Тем не менее ее все же можно отличить от *пробела*, потому что когда в окне – литера конца строки ~ и ни в каком другом случае ~ функция EOLN для этого файла возвратит значение *true*. После успешного считывания литеры в окне, окно сдвигается к ближайшей следующей литере. Если эта литера оказалась маркером конца файла, то функция EOF (от *End Of File* – конец файла), будучи вызвана для этого файла, возвратит значение *true*. Функция EOF возвращает значение *true*, если только в окне – маркер конца файла. Попытка *прочитать* маркер конца файла ведет к ошибке.

Если параметр процедуры READ или READLN имеет тип INTEGER или REAL, то окно сдвигается вперед, пропуская пробелы и литеры конца строки, до тех пор, пока не встретится первый значащий символ новой строки (или же поиск не закончится аварийно на маркере конца файла). Строка (если она правильно записана) преобразуется в элемент стандартного типа, согласующегося с соответствующим параметром. (Инструкция READ(x), например, сработает неверно для строки 1.5, если переменная *x* – типа INTEGER.) После успешного прочтения строки окно устанавливается над следующей за этой строкой литерой. Эта следующая литера может быть пробелом или литерой конца строки; в последнем случае функция EOLN, будучи вызвана, возвратит значение *true* (а EOF – *false*).

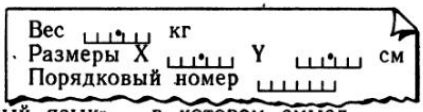
После успешного прочтения последнего параметра процедурой READLN (в отличие от READ) окно пропускает все литеры, оставшиеся в данной строке файла, и останавливается, оказавшись над *первой* литерой *следующей* строки. Эта первая литера может оказаться маркером конца файла. В этом случае функция EOF возвратит значение true. То же относится к использованию процедуры READLN без параметров.

Концептуально, окно для текстовых файлов имеет гибкие рамки. Большую часть времени оно просматривает одну литеру. Однако, когда встречается строка литер, определяющая *число*, окно «растягивается», с тем, чтобы охватить все литеры в этой строке. В этом – отличие от окон *двоичных файлов*; окна для них могут иметь сложную структуру, но не являются гибкими. Двоичные файлы описываются ниже.

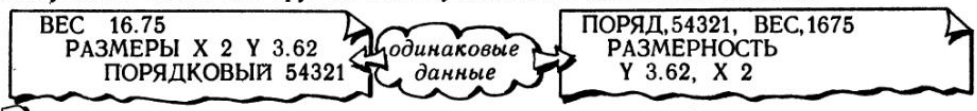
БЕЗОПАСНОЕ ЧТЕНИЕ

ЭТО – ПРОБЛЕМА В БОЛЬШИНСТВЕ ПОПУЛЯРНЫХ ЯЗЫКОВ

Всем нам время от времени приходится заполнять «форматные» бланки, преимущество которых сказывается разве что в некотором облегчении жизни программиста:



Однако, процедуру ввода сложных наборов данных весьма желательно сделать более гибкой. Можно, например, придумать некоторый «проблемно-ориентированный язык», в котором смысл следующего числа или группы чисел указывается ключевыми словами:



Вполне естественно, что в программе, разработанной для чтения форматных данных программист оставляет проверку данных Паскалю, например используя для чтения первого элемента приведенного выше бланка оператор READ(INPUT,weight). Однако, если пользователь такой программы по ошибке введет, скажем, 16.7S вместо числа 16.75, то Паскаль выдаст сообщение о неверном числе ~ и программа остановится. Для программы, считывающей более чем пару чисел, такой подход неприемлем.

Единственный способ не утратить контроль над программой – это читать данные по одной литере и конструировать числа или ключевые слова и находить ошибки пользователя в самой программе. В Паскале есть лишь одна безопасная процедура чтения – READ(file,ch) с предварительной проверкой конца файла.

Если эти рассуждения представили вам Паскаль в дурном свете, то не сомневайтесь – ряд других известных языков в смысле работы с вводом ничуть не лучше. Язык Фортран предлагает ряд привлекательных описателей ввода (см. мою книгу *Illustrating Fortran, C.U.P., 1982, гл. 10*), и тем не менее единственный практичный из них – тот, что считывает одну литеру. Несколько лучше обстоят дела в тех версиях Бейсика, где есть оператор «ON ERROR...», потому что этот оператор в случае ошибки при чтении позволяет вернуть управление – но такой подход явно неуклюж.

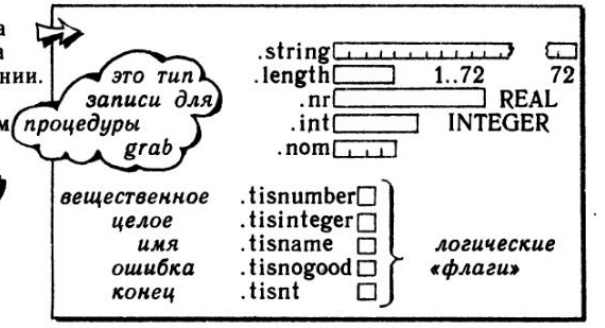
Описанная ниже процедура сконструирована так, чтобы продолжать работу, какая бы глупость не встретилась во входном файле. Называется процедура *grab*.

Для использования процедуры *grab* просто вызывайте ее, когда потребуется следующий элемент. Проверки конца файла перед вызовом делать не надо. Считается, что каждый элемент заканчивается *пробелом, признаком новой строки* или *концом файла*. Процедура возвращает запись, описывающую все аспекты прочитанного элемента. Процедура *grab* различает четыре типа элементов:

- *имя*; имя начинается с буквы и содержит только буквы и цифры. Значащими являются только четыре первые литеры (РАЗМЕРНОСТЬ=РАЗМЕРЫ).
- *число*; число может быть записано с десятичной точкой или без нее. Процедура отличает эти две формы записи числа.
- *nogood*; строка литер, которая не является ни именем, ни числом (например, +R6).
- *tisn't*; пустой элемент, означающий конец файла; (любое последующее обращение к *grab* даст тот же результат).

Здесь изображен формат записи, которую возвращает процедура. Эта запись выглядит громоздкой, но на самом деле очень проста в обращении. Пусть, к примеру, программист ожидает, что следующим элементом входного файла будет число. Обращение может выглядеть так:

```
grab(it);
IF   it.tisnumber
  THEN remember:=it.nr
  ELSE complain(it);
```



Здесь предполагается, что *complain* – процедура диагностики. Таким образом, если элемент окажется не числом, а чем-то другим, то диагностическая процедура сможет точно указать, где ошибка (IF it.tisnt THEN ... IF it.tisnogood THEN ...) и, обратившись к компоненте it.string, сможет точно определить, что ввел пользователь.

Возможно, программист задействует операторы WITH it DO ... и, таким образом, упростит ссылки на запись: IF tisnumber THEN ... IF tisnogood THEN ... .

Ввод числа вроде 12345 повлечет за собой установку в положение *true* обоих флагов – флага *tisnumber* и флага *tisinteger*. Затем в поле *.nr* будет помещено значение 12345.0, а в поле *.int* – значение 12345 . Однако, за вводом 12345000000 последует установка в *true* только *tisnumber*, потому что (в стандартном компиляторе) это число превосходит MAXINT.

Ниже логика процедуры grab представлена таблицей состояний. Использование таких таблиц описано на с. 60.

литера / состояние	1]	2]	'0'..'9' 3]	'A'..'Z' 'a'..'z 4]	5]	другие 6]	, пробел, нов. строка 7]
[1,	[2,	sign:=-1 [2,	nr:=digit(ch) [3,	nom[i]:=ch [6, 10	[7,	[7,	[1,
[2,	[7,	[7,	действие 2	[7,	[7,	[7,	tisnumber:=TRUE; nr:=sign*nr; 3
[3,	[7,	[7,	nr:=10*nr+digit(ch) [3, 3	[7,	frac:=1 [4, 4	[7,	if nr<MAXINT then tisinteger:=TRUE и int:=sign*TRUNC(nr)
[4,	[7,	[7,	frac:=10*frac; nr:=nr+digit(ch)/frac; [5, 6	[7,	[7,	[7,	tisnogood:=TRUE 7
[5,	[7,	[7,	[7,	[7,	[7,	[7,	tisnumber:=TRUE; nr:=sign*nr 8
[6,	[7,	[7,	i:=i+1; nom[i]:=ch; [6, 11	[7,	[7,	[7,	tisname:=TRUE 9
[7,	[7,	[7,	[7,	[7,	[7,	[7,	tisnogood:=TRUE 7

Различные *действия* в этой таблице представлены номерами в маленьких облачках, например 7 . Изменение *состояния* отмечается широкой стрелкой, например [7,. Сама таблица хранится как массив table[1..7,1..7] (см. следующую страницу), а значение каждой компоненты представляет собой код вида

$$100 * \textit{действие} + \textit{состояние}$$

Эта таблица формируется в компьютере при помощи файла.

ПРОЦЕДУРА GRAB (ЗАПИСЬ) ОБЕСПЕЧИВАЕТ БЕЗОПАСНОЕ ЧТЕНИЕ

Основная программа начинается с задания констант. Значение *Stringlength* должно быть равным максимально возможной длине вводимых строк (в случае, если пользователь вообще забудет ввести пробелы или запятые). В константе *Namelength* надо задать число значащих символов в имени – обычно это четыре литеры. Константы *Minord* и *Maxord* – порядковые значения первой и последней литеры в имеющемся наборе литер. Для кода ASCII – это 32 и 127. В случае работы с кодом EBCDIC или каким-либо другим кодом измените эти значения.

```
PROGRAM saferead(INPUT,OUTPUT,f);                          (* файл *)
  CONST
    stringlength=72; namelength=4; minord=32; maxord=127;
  TYPE
    stringtype = PACKED ARRAY[1..stringlength] OF CHAR;
    nametype  = PACKED ARRAY[1..namelength] OF CHAR;
    lookuptype = ARRAY[minord..maxord] OF 1..7;
    tabletype = ARRAY[1..7,1..7] OF 1..1200;

    intype = RECORD
               string: stringtype;
               length: 0..stringlength;
               nr: REAL;
               int: INTEGER;
               nom: nametype;
               tisnumber,tisinteger,tisname,
               tisnogood,tisnt: BOOLEAN;
             END;

  VAR
    it: intype; lookup: lookuptype; table: tabletype;
    i: INTEGER; f: TEXT;                                   (* текстовый файл *)
```

```
lookup[32]  7  пробел
lookup[33]  6
lookup[34]  6
lookup[46]  5  '.'
lookup[57]  3  '9'
lookup[66]  4  'B'
        примеры
```

Массивы *lookup* и *table* необходимо проинициализировать. Массив *lookup* предназначен для определения номера столбца в таблице *table*, который соответствует прочитанной литере. Например, если в *ch* находится литера '9', то *lookup[ORD(ch)]* сразу даст 3. Аналогично, если *ch* содержит '.', то *lookup[ORD(ch)]* будет 5. Инициализация производится специальной процедурой, которая должна быть вызвана только один раз: перед первым обращением к *grab*. Вот эта процедура:[2)]

```
PROCEDURE initialization(VAR l: lookuptype; VAR t: tabletype);
  VAR
    c: CHAR;   i,j: 1..7;   k: minord..maxord;
  BEGIN
    FOR k:= minord TO maxord DO l[k]:=6;
    l[ORD('+')]:=1;   l[ORD('-')]:=2;
    FOR c:= '0' TO '9' DO l[ORD('c')]:=3;
    FOR c:= 'A' TO 'Z' DO l[ORD('c')]:=4;
    FOR c:= 'a' TO 'z' DO l[ORD('c')]:=4;
    l[ORD('.')]:=5;
    l[ORD(',')]:=7;
    l[ORD(' ')]:=7;
```

заполняем шестерками, потом некоторые компоненты переустановим

READ(ch) читает признак конца строки как пробел

Объявление VAR в основной программе содержит объявление файла типа TEXT: «f: TEXT». Запись и последующее чтение этого файла избавляет от необходимости записывать сорок девять отдельных присваиваний:

```
t[1,1]:=002;   t[1,2]:=102;   t[1,3]:=203;  и т.д.
```

(Если ваш компилятор Паскаля допускает «врéменные» файлы, то можно перенести все ссылки на *f* из основной программы в раздел VAR процедуры инициализации ~ единственное место, где используется *f*.)

```
REWRITE(f);   { открытие файла f на запись }
WRITE(f,  002,  102,  203,  1006,  007,  007,  001);
WRITE(f,  007,  007,  203,  007,   007,  007,  300);
WRITE(f,  007,  007,  403,  007,   504,  007,  300);
WRITE(f,  007,  007,  605,  007,   007,  007,  700);
WRITE(f,  007,  007,  605,  007,   007,  007,  800);
WRITE(f,  007,  007,  1106, 1106,  007,  007,  900);
WRITE(f,  007,  007,  007,  007,   007,  007,  700);
  RESET(f);   { установка на чтение }
  FOR i:=1 TO 7 DO
    FOR j:=1 TO 7 DO
      READ(f,t[i,j]);
END;  { initialization }
```

сравните эту таблицу с таблицей на с. 129

[0, подразумевает выход

ключ: 504

действие 5

новое состояние [4,

Ниже показано начало процедуры *grab*. Сюда включено определение локальной функции для нахождения целого значения литеры. Например, *digit('6')* возвращает 6.

```
PROCEDURE grab( VAR rec: intype);
  VAR
   i: 1..stringlength; sign: -1..1;   state: 0..7;
   ch: CHAR;   action: 0..11;   frac: INTEGER;

  FUNCTION digit(c: CHAR): INTEGER;
    BEGIN
      digit:= ORD(ch) - ORD('0');
    END;

  BEGIN   { grab }
    WITH rec DO
      BEGIN  { WITH rec }
        tisnumber:= FALSE; tisinteger:= FALSE;
        tisname:= FALSE; tisnogood:= FALSE;   tisnt:= FALSE;
        length:= 0; stat:=1; sign:=1;
      FOR i:=1 TO stringlength DO string[i]:=' ';
      FOR i:=1 TO namelength DO nom[i]:=' ';
        i:=1;
```

установить во все флаги false

заполнение приемной строки пробелами

снова i - в начало

продолжение на следующей странице

Далее приведена логика содержательной части процедуры *grab*:

```
      REPEAT {для каждой цифры}
        IF EOF(INPUT)
          THEN
            BEGIN
              action:=table[state,7] DIV 100;
              tisnt:= (state = 1)
            END
          ELSE
            BEGIN
              READ(INPUT,ch);
              length:=length+1; string[length]:=ch;
              action:=table[state,lookup[ORD(ch)] DIV 100;
              state:=table[state,lookup[ORD(ch)] MOD 100;
            END;  { END IF }
        CASE action OF
         0:          ;
         1: sign:= -1;
         2: nr:= digit(ch);
         3: BEGIN
              IF nr <= MAXINT
                THEN
                  BEGIN
                    tisinteger:=TRUE;
                    int:= sign*TRUNC(nr)
                  END;
              tisnumber:= TRUE;
              nr:= sign*nr
            END;
         4: nr:= 10*nr + digit(ch);
         5: frac:= 1;
         6: BEGIN
              frac:= 10*frac;
              nr:= nr + digit(ch)/frac;
            END;
         7: tisnogood:= TRUE;
         8: BEGIN
              tisnumber:= TRUE;
              nr:= sign*nr
            END;
         9: tisname:= TRUE;
        10: nom[1]:= ch;
        11: BEGIN
              i:=i+1;
              IF i<= namelength THEN nom[i]:= ch
            END;
      END { CASE }
    UNTIL (state = 0) OR tisnt
  END
END;  { procedure grab }
```

проверка конца файла

столбец 7 – для разделителей

конец файла в состоянии (state) 1 означает, что grab ничего не извлекает ... пустой элемент

в процедуре grab – это единственный оператор READ

заполнение строки

действие и новое состояние

ничего не делать

начальное значение было +1

первая цифра

nr может быть преобразовано в целое функцией TRUNC( ) только если оно не превосходит MAXINT

построение целого в nr

после десятичной точки делим очередную цифру на 10, 100, 1000 и т.д.

первая литера в nom

построение nom длины namelength

нормальное завершение

встретился конец файла и никакого элемента перед ним

Приведенная ниже главная программа служит лишь для демонстрации процедуры *grab*:

```
BEGIN  { главная программа }
  initialization(lookup,table);
  REPEAT
    grab(it);
    WITH it DO
      BEGIN

        IF tisnumber  THEN WRITELN(nr);
        IF tisinteger THEN WRITELN(int);
        IF tisname    THEN WRITELN(nom);

        IF tisnogood THEN
          FOR i:=1 TO length DO
            WRITE(string[i]);
        WRITELN
      END { WITH it }
  UNTIL it.tisnt

END. { program }
```

Поэкспериментируйте с программой, например, так:



О двоичных файлах рассказывается на следующей странице. Файл с именем *f* из предыдущего примера было бы лучше сделать двоичным файлом. Для соответствующих изменений программы поставьте вместо объявления f: TEXT в разделе VAR главной программы объявление f: FILE OF INTEGER.

Проверка конца файла (EOF) в начале предыдущей страницы рассчитана на неинтерактивную работу, однако, эта проверка не помешает и при интерактивной работе. Функция EOF будет возвращать значение *false*, пока с клавиатуры не послан специальный сигнал (в Турбо Паскале это CTRL Z). Если у Вас возникнут неполадки при работе процедуры *grab*, загляните для воодушевления в гл. 11.

ЧТО ТАКОЕ ДВОИЧНЫЕ ФАЙЛЫ

Элемент *текстового* файла преобразуется из цепочки литер во внутреннюю форму при выполнении процедуры READ; из внутренней формы в цепочку литер – при выполнении процедуры WRITE. В *двоичном файле* в отличие от текстового информация хранится во внутренней (двоичной) форме. Двоичные файлы имеют ряд преимуществ в сравнении с текстовыми. Отсутствие необходимости в преобразовании позволяет их считывать и записывать с большей скоростью. Они также компактнее текстовых файлов и лишены ошибок округления, которые возникают при преобразовании во внутреннюю форму и обратно. Недостаток двоичных файлов в том, что даже если напечатать двоичный файл, то нельзя будет понять, что в нем содержится (исключением является FILE OF CHAR).

Двоичные файлы полезны для временного хранения информации в процессе выполнения. Обычно в конце работы программы, отслужив свое, такие файлы могут быть уничтожены. Однако иногда необходимо сохранять огромные объемы промежуточных данных, полученных в одной программе, для последующего их использования в другой программе. Двоичные файлы – компактные и точные – идеальны для этих целей.

Файл в Паскале – это *переменная*. Посмотрите на последнюю строчку раздела VAR на с. 130, она воспроизведена ниже:

```
i: INTEGER;  f: TEXT;
```

переменная типа TEXT

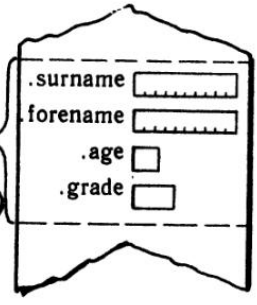
Видно, что файл *f* объявлен как переменная типа TEXT точно таким же образом, как объявлена переменная *i* типа INTEGER. Вообще файлы могут быть любого типа; все файлы, *за исключением* файлов типа TEXT – *двоичные*.

В следующем примере каждая компонента файла *binfile* является записью того же типа, что использовались в программе о персональных записях на с. 112.

```
TYPE
  nametype = PACKED ARRAY[1..10] OF CHAR;
  detailtype =
    RECORD
      surname,forename: nametype;
      age 18..65;
      grade: (jr,sr,exec)
  END;
VAR
  binfile:  FILE OF detailtype;
```

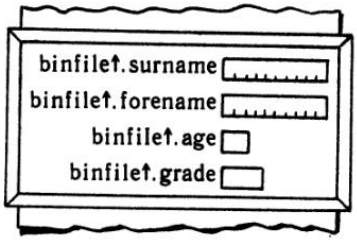
Одна компонента

обратите внимание на «FILE OF»

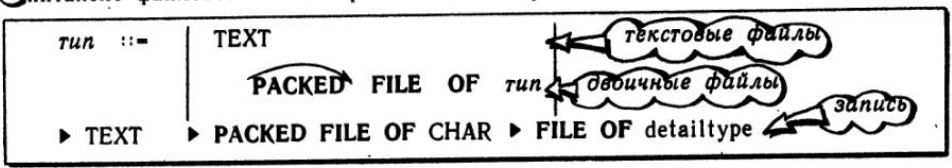


Сверху схематически нарисована одна незаполненная компонента файла *binfile*. Файл может содержать любое количество таких записей, какое необходимо для работы программы.

Объявление любого файла сопровождается неявным объявлением еще одной переменной – *переменной-окна*, относящейся к этому файлу. Имя окна такое же, как и имя файла; только, как показано, добавляется ↑: Вся связь с файлом *binfile* осуществляется через окно с именем *binfile*↑, которое иногда также называется *буфером* или *буферной переменной* файла.



Синтаксис файлового типа определяется следующим образом:



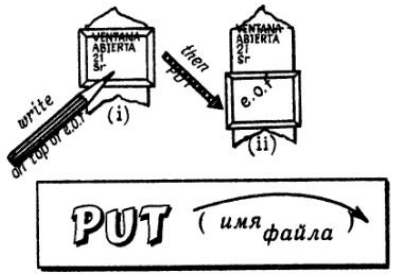
Не путайте FILE OF CHAR с текстовым файлом. Автоматическое прямое и обратное преобразование цепочек литер ~ а также обнаружение конца строки ~ привилегия исключительно текстовых файлов. Только к текстовым файлам можно применять процедуры WRITELN и READLN.

Запись в двоичный файл состоит, вообще говоря, из двух этапов: (1) то, что должно быть записано, следует присвоить переменной-окну; (2) для сдвига вперед рамки окна и записи нового конца файла вызывается процедура PUT:

```
REWRITE(binfile);
WITH binfile↑ DO
 BEGIN
   surname:= 'VENTANA'
   forename:='ABIERTA'
   age:= 21;
   grade:= sr
 END;
 PUT(binfile);
```

пустой файл, в окне виден конец файла

перемещает окно на новый конец файла

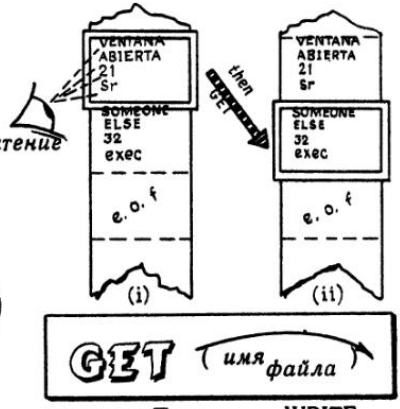


После проверки конца файла, чтение из файла также состоит из двух этапов: (1) прочесть содержимое окна; (2) используя процедуру GET, сдвинуть рамку окна вперед к следующей компоненте (или к концу файла, если следующей компоненты нет):

```
RESET(binfile);
IF NOT EOF(binfile)
 THEN
   BEGIN
   WITH binfile DO
     BEGIN
       s:= surname;
       f:= forename
       a:= age;
       g:= grade;
     END;
   GET(binfile)
END;
```

окно устанавливается на первую компоненту

установить окно на следующую компоненту (или конец файла)



Процедуры PUT и GET – это процедуры «нижнего уровня». Процедуры WRITE и READ могут быть описаны в терминах PUT и GET ~ и переменных-окон ~ следующим образом:

```
WRITE(filename,item) ≡ filename↑:= item; PUT(filename);
READ  (filename,item) ≡ item:= filename↑; GET(filename);
```

В Турбо Паскале процедуры PUT и GET не определены; вместо этого расширены процедуры WRITE и READ.

СЖАТИЕ

ПРИМЕР ИЛЛЮСТРИРУЕТ ОДНОВРЕМЕННУЮ РАБОТУ С ТЕКСТОВЫМИ И ДВОИЧНЫМИ ФАЙЛАМИ

Эта программа разработана для чтения текстового файла и записи соответствующего двоичного файла. Таким способом достигается сжатие хранимой информации.

Допустим, что текстовый файл был верифицирован другой программой, так что при вводе уже не будет надобности делать проверку формы и законченности. Пусть известно, что форма файла в точности такая, как здесь нарисовано:

температура в СОЛНЕЧНОЙ КОМНАТЕ

День	Месяц	темп. в полдень	отметки
2	ФЕВ	2.5	ХОЛОДНО
4	ФЕВ	-10	МОРОЗ
17	ФЕВ	-15.5	БРРР!!!

```
PROGRAM compressor(textin,binaryout,OUTPUT);
 CONST
   monthchars = 3;
   remchars = 8;

 TYPE
   monthtype = PACKED ARRAY[1..monthchars] OF CHAR;
   remtype = PACKED ARRAY[1..remchars] OF CHAR;
   grouptype = RECORD
                 day: 1..31;
                 month: monthtype;
                 temp: REL;
                 remarc: remtype
               END;

 VAR
   textin: TEXT;
   binaryout: FILE OF grouptype;
   count: INTEGER;   i: 1..monthchars; j:= 1..remchars;

BEGIN
  count:=0;
  RESET(textin);
  REWRITE(binaryout);
  WITH binaryout↑ DO
    WHILE NOT EOF(textin) DO
     BEGIN
       READ(textin,day);
       FOR i:=1 TO monthchars DO
         READ(textin,month[i]);
       READ(textin,temp);
       FOR j:=1 TO remchars DO
         READ(textin,remark[j]);
       count:= count + 1;
       READLN(textin);
       PUT(binaryout);
     END  { WHILE }
  { end WITH }
  WRITELN(OUTPUT,'Обработано ', count,' строк данных')
END.
```

СВОЙСТВА ФАЙЛОВ: СВОДКА

<table>
<tr><th>ТИП ФАЙЛА / СВОЙСТВО</th><th colspan="2">ТЕКСТОВЫЕ ФАЙЛЫ</th><th>PACKED FILE OF CHAR (не текст. файлы)</th><th>Прочие типы (примеры: файлы массивов, записей смешаного типа)</th></tr>
<tr><td></td><td>стандартные текст. файлы: INPUT, OUTPUT</td><td>Файлы, определенные программистом</td><td colspan="2"></td></tr>
<tr><td>Включение имен файлов в оператор PROGRAM</td><td>INPUT – необязателен, OUTPUT надо включать хотя бы для сообщений об ошибках</td><td colspan="3">В общем случае файл должен быть упомянут в операторе PROGRAM.

(Некоторые компиляторы допускают «временные» файлы, которые не указываются в операторе PROGRAM)</td></tr>
<tr><td>Определение файла как переменной в разделе VAR</td><td>По умолчанию, типа TEXT</td><td colspan="3">В общем случае переменную-файл следует объявить в разделе VAR главной программы (или, если компилятор допускает временные файлы, то в локальном разделе VAR)</td></tr>
<tr><td>RESET и REWRITE</td><td>Не следует явно употреблять RESET и REWRITE</td><td colspan="3">Для чтения файл должен быть открыт процедурой RESET(имя файла). Для записи файл прежде должен быть открыт процедурой REWRITE(имя файла)</td></tr>
<tr><td>Операторы ввода</td><td>READ, READLN и GET: если 1-й параметр опущен – используется INPUT</td><td>READ, READLN и GET: подразумеваемых парам. нет</td><td colspan="2">READ и GET, но не READLN</td></tr>
<tr><td>Преобразования при вводе</td><td colspan="2">Каждая цепочка литер автоматически, в соответствии с типом параметра, преобразуется к типу CHAR, REAL или INTEGER</td><td>Двоичный код файла преобразуется только в элементы типа CHAR</td><td>Двоичный код файла преобразуется в REAL, INTEGER, CHAR в зависимости от типа файловой переменной</td></tr>
<tr><td>Операторы, используемые для вывода</td><td colspan="2">WRITE, WRITELN, PUT и PAGE (некоторые компиляторы с Паскаля не допускают PUT)</td><td colspan="2">WRITE и PUT, но не WRITELN</td></tr>
<tr><td>Преобразования при выводе</td><td colspan="2">Элементы типа CHAR, INTEGER, BOOLEAN, а также PACKED ARRAY OF CHAR преобразуются в печатаемые цепочки литер</td><td>Элементы типа CHAR преобразуются в двоичный код выходного файла</td><td>Элементы всех типов преобразуются в двоичный код выходного файла</td></tr>
<tr><td rowspan="2">EOLN</td><td colspan="2">Литера конца строки прочитывается как пробел. EOLN( ), когда в окне литера конца строки, возвращает true</td><td colspan="2" rowspan="2">Конец строки не обнаруживается; функция EOLN( ) используется только применительно к текстовым файлам</td></tr>
<tr><td>EOLN означает – EOLN(INPUT)</td><td>Нет подразум. параметров</td></tr>
<tr><td>EOF</td><td colspan="4">Если в окне файла – маркер конца файла, то функция EOF возвращает true; во всех остальных случаях – false</td></tr>
<tr><td rowspan="2"></td><td>EOF означает – EOF(INPUT)</td><td colspan="3">У EOF( ) подразумеваемых параметров нет</td></tr>
<tr><td colspan="2">При интерактивном вводе сигнал EOF зависит от системы</td><td colspan="2">Интерактивный ввод невозможен</td></tr>
</table>

УПРАЖНЕНИЯ

1. Выполните программу *saferead*. Попробуйте с клавиатуры «вывести из строя» эту программу. Это не должно получиться. При каждой попытке ошибочная строка будет отображаться на экране для проверки.

2. Измените программу *personel* со с. 112 так, чтобы она записывала массив персональных записей в форме двоичного файла. Файл следует записать после прочтения всех данных, но до их сортировки. Кроме того, вставьте в программу чтение этого файла перед вводом каждого нового пакета записей. Обладая такими возможностями, программа будет выглядеть как простенькая управляющая система.

3. Возьмите любую программу из предыдущих глав (например, программу *loanrate* со с. 72) и замените ее примитивные операторы ввода обращением к процедуре *grab*. Если ваш Паскаль допускает интерактивные программы, добавьте соответствующие запросы и диагностику ошибок с тем, чтобы придать программе определенную дружественность по отношению к ее потенциальному пользователю.

ПРИМЕЧАНИЯ РЕДАКТОРА

[1] (с. 123) В версии Турбо Паскаль чтение из текстового файла в состоянии конца строки возвращает не пробел, а специальную литеру – признак конца строки.

[2] (с. 130) Процедура инициализации опирается на то, что коды всех строчных букв, равно как и коды всех прописных букв, располагаются подряд. Если это свойство не выполнено, то процедуру придется усложнить.

11

ИНТЕРАКТИВНЫЙ ВВОД

ДИАЛОГ
ПРОБЛЕМА ЗАГЛЯДЫВАНИЯ ВПЕРЕД
ПРОБЛЕМА БУФЕРА
ПРОБЛЕМА КОНЦА ФАЙЛА (*EOF*)

ДИАЛОГ

ПАСКАЛЬ ПРОЕКТИРОВАЛСЯ БЕЗ РАСЧЕТА НА ДИАЛОГ

Пользователи многих современных программ работают в режиме диалога. Программа выдает на экран вопросы и подсказки, пользователь отвечает, набирая ответы на клавиатуре. Ответы зависят от уже полученных результатов. Если бы от пользователя требовалось сообщить всю информацию заранее, то результат, возможно, был бы иным. Другими словами, пользователь и программа приближаются к результату рука об руку, взаимодействуя друг с другом. Идея диалога сейчас является общепринятой, хотя в истории вычислительной техники она сформировалась относительно недавно.

Язык Паскаль был спроектирован до того, как диалог стал общепринятым. Язык разрабатывался в те времена, когда программисты набивали программы на перфокартах и оставляли колоду карт оператору компьютера, который загружал ее в устройство чтения перфокарт. Данные также набивались на перфокартах и вручались оператору. Обе колоды впоследствии возвращались программисту завернутыми в «нотную бумагу» печатающего устройства с результатами (или же печальной диагностикой). Операторы обычно ожидали, пока для загрузки не накопится несколько таких программ. Поэтому такой режим обработки был назван «пакетным режимом».

Процедура ввода READ в Паскале была спроектирована в расчете на пользователя, работающего в пакетном режиме. Логика работы процедуры READ с перфокарточным файлом следующая: (1) считать с текущей перфокарты специфицированный элемент или элементы, затем (2) заглянуть вперед, чтобы увидеть, есть ли еще литера на текущей перфокарте; если нет, то сделать результатом функции EOLN – *true*. Такая логика дает программисту возможность перед каждым вызовом процедуры READ писать:

```
IF NOT EOLN THEN ...
```

Логика чтения целой строки (READLN) – такая же: (1) считать специфицированный элемент или элементы с текущей перфокарты, игнорируя при этом все оставшиеся позиции; затем (2) заглянуть вперед, чтобы увидеть, есть ли еще перфокарта; если нет, то сделать результатом функции EOF значение *true*. В результате программист имеет возможность перед каждым вызовом процедуры READLN писать:

```
IF NOT EOF THEN ...
```

Однако, заглядывать вперед при интерактивном вводе – это абсурд; программа не может знать, что еще предполагает ввести ее пользователь. Таким образом, в силу того что ввод осуществляется человеком, отвечающим на запросы, логику процедур READ и READLN необходимо модифицировать.

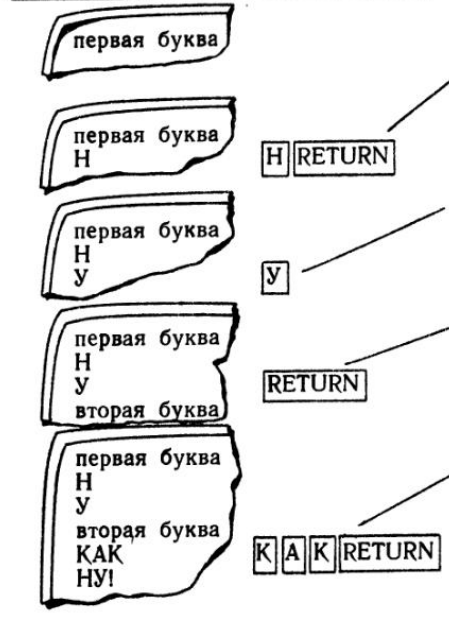
Распространенным подходом к решению этой проблемы (Acornsoft: ISO Паскаль, Prospero: Pro Паскаль) является «отложенный ввод-вывод», который означает, что подглядывание вперед откладывается до тех пор, пока программа не сделает следующего запроса с клавиатуры ~ например, посредством READ или EOF. Иная техника (Borland: Турбо Паскаль) – воспринимать в качестве результата заглядывания вперед *текущую* литеру с клавиатуры[1]. И тот и другой подходы решают проблему заглядывания вперед, которая обсуждается напротив. Об остальных проблемах речь пойдет после.

ПРОБЛЕМА ЗАГЛЯДЫВАНИЯ ВПЕРЕД

ДЕЛАЕТ НЕВОЗМОЖНОЙ ИНТЕРАКТИВНУЮ РАБОТУ

Принцип заглядывания вперед, обсуждавшийся в общих чертах на предыдущей странице, и обуславливает заминки при интерактивной работе. Процедура RESET (при работе с файлом INPUT – неявная) помещает окно на первый элемент файла; при последующем чтении процедурой READ или READLN содержимое окна копируется и *затем окно передвигается к следующей литере или за ближайший конец строки* соответственно. Это фундаментальный принцип Паскаля, поэтому стоит исследовать, что же произойдет, если попытаться выполнить эту маленькую программу ~ скомпилировав ее традиционным компилятором с Паскаля ~ в интерактивном режиме.

```
PROGRAM hiccups(INPUT,OUTPUT);
  VAR a,b: CHAR;
BEGIN
  WRITELN('первая буква');
  READLN(a);
  WRITELN('вторая буква');
  READLN(b);
  WRITELN(a,b,'!');
END.
```



Выполнение начинается с оператора WRITELN('первая буква'); затем выполняется оператор READLN(a). Здесь программа ждет, пока будет что-либо набрано и нажата клавиша RETURN.

Наберем Н и нажмем клавишу RETURN.

Оператор READLN(a) считывает Н, однако, не удовлетворяется, пока не увидит первую литеру следующей строки. Таким образом, мы «зависли». Очевидный выход – ввести первую литеру следующей строки.

Все еще висим! В большинстве систем программа не получает данных, пока не нажата клавиша RETURN. Нажмем ее.

Теперь оператор READLN(a) завершается и управление передается оператору WRITELN('вторая буква') и далее на READLN(b). Оператор READLN(b) считывает У, однако ждет первой литеры следующей строки. Опять висим!

Следующей строки нет. Тем не менее надо ввести что-нибудь. Что угодно!

Наконец, завершается оператор READLN(b), так что управление переходит на оператор WRITELN(a,b,'!') и затем на конец программы. Не блестяще!



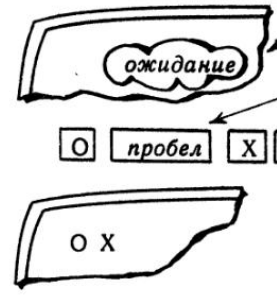
Компиляторы с Паскаля вроде упомянутых на предыдущей странице, не вызывают заминок. Результат будет точно таким, как можно ожидать из текста программы. Иначе говоря, таким, как здесь нарисовано[2]).

ПРОБЛЕМА БУФЕРА

ЕСЛИ ВАШ ПАСКАЛЬ РАБОТАЕТ ТАК ЖЕ, НЕ ПИШИТЕ ИНТЕРАКТИВНЫХ ПРОГРАММ

В те времена, когда под словом «файл» подразумевался «файл на магнитной ленте», обычным делом было использование Паскаль-процессором *буферов* ввода-вывода. Буфер – это область в памяти. Литеры, посланные в выходной файл, сперва обязательно попадали в буфер. И только когда буфер заполнялся, его содержимое копировалось на магнитную ленту. Этот же подход использовался и при вводе. Такая буферизация просто необходима при работе с файлами на магнитной ленте, она полезна при работе с дисками, однако если «файлом» является человек, вводящий данные с клавиатуры, то буферизация – это катастрофа. Ниже, на примере простой программы, иллюстрируются происходящие события.

```
PROGRAM flush(INPUT,OUTPUT);
 VAR a,b: CHAR;
BEGIN
 WRITELN('первая буква');
 READ(a);
 WRITELN('вторая буква');
 READ(b);
 WRITELN(a,b,'!');
END.
```



Выполнение начинается с оператора WRITELN('первая буква'). Слова 'первая буква', несомненно, печатаются, однако печатаются в выходной буфер ~ имеющий достаточный объем, чтобы его содержимое еще не копировалось на экран. На экране ничего нет, а программа ждет.

Наберем строку данных и нажмем клавишу RETURN. Данные появятся на экране, но это еще не означает, что программа получила их.

Если больше ничего не происходит, то это значит, что данные попали во входной буфер и, пока буфер не заполнится или пока с клавиатуры не будет послан сигнал конца файла, данные из буфера не будут выбраны. (Как послать сигнал конца файла – зависит от системы.)

Предположим, что при вводе используется всего лишь *буфер строки*, который активизируется клавишей RETURN. Это означает, что оператор READ(a) выполнится; оператор WRITELN('вторая буква') пошлет слова 'вторая буква' в буфер вывода; оператор READ(b) выполнится; оператор WRITELN(a,b,'!') пошлет слово 'OX!' в буфер вывода.

В конце концов управление достигает завершающего END. и выходной буфер копируется на экран.

Если ваши программы ведут себя подобным образом, то это означает, что Паскаль-компилятор не рассчитан на работу с интерактивными программами. Программы для такого компилятора должны ориентироваться на ввод данных из дискового файла.

Компиляторам Pro Паскаль, Турбо Паскаль и Acornsoft Паскаль не присущи трудности, речь о которых шла выше; с их помощью можно компилировать интерактивные программы.

ПРОБЛЕМА EOF

НЕ ИСПОЛЬЗУЙТЕ ОПЕРАТОР WHILE NOT EOF В ИНТЕРАКТИВНЫХ ПРОГРАММАХ

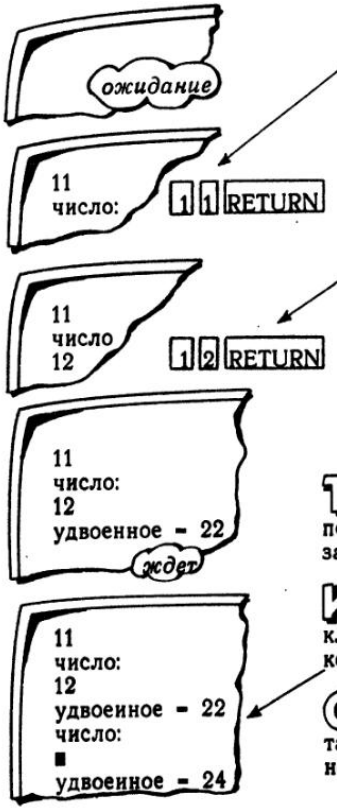
Справа – стандартная в Паскале схема для неинтерактивного ввода. Однако, что *означает* EOF в интерактивной программе? Обычно сигнал конца файла посылается с клавиатуры в виде специальной литеры, которая зависит от настройки системы. Пример сигнала конца файла – одновременное нажатие клавиш CTRL и Z.

```
WHILE NOT EOF(f) THEN
  BEGIN
    • прочитать и обработать информацию в окне, затем
    • сдвинуть окно к следующему элементу
  END
```

```
PROGRAM pardon(INPUT,OUTPUT);
 VAR i: INTEGER
BEGIN
 WHILE NOT EOF(INPUT) DO
   BEGIN
     WRITELN('число:');
     READLN(INPUT, i);
     WRITELN('удвоенное =',2*i)
   END
END.
```

Здесь показано, что же будет, если вы используете этот прием в интерактивной программе.

Работа начинается с проверки WHILE NOT EOF(INPUT), где программа ждет. Еще ничего не было набрано, поэтому функции EOF нечего проверять. (Оператор READLN тут ни при чем, потому что управление до него еще не дошло.) Поможем программе, введя первое число.



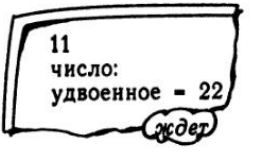
Числа 11 достаточно для проверки конца файла. Функция EOF(INPUT) возвращает *false* и, таким образом, управление передается на оператор WRITELN('число:') и далее на READLN(INPUT,i). Оператор READLN(INPUT,i) считывает '11', но не завершается, пока не заглянет вперед (об этом см. также в конце страницы).

Теперь оператор READLN(INPUT,i) возвращает первое число 11, управление переходит на WRITELN('удвоенное =',2*i); этот оператор печатает 22. Затем цикл начнется сначала с WRITELN('число:') и затем READLN(INPUT,i). Оператор READLN(INPUT,i) считывает число 12 и ждет следующей возможности заглянуть вперед.

Таким образом, программа будет продолжать печатать решение предыдущей задачи, затем запрашивать число, которое ей уже было дано.

И так до тех пор, пока не будет нажата комбинация клавиш, посылающая сигнал конца файла в вашей конкретной системе (здесь она обозначена как ■).

С «отложенным вводом» результаты выглядят чуть разумнее, но все же остаются «не в фазе».



Не используйте WHILE NOT EOF в интерактивных программах.

ПРИМЕЧАНИЯ РЕДАКТОРА

[1] (с. 140) Способ действий в версии 5 системы Турбо Паскаль скорее отвечает «отложенному вводу-выводу», чем использованию текущей литеры. Версия 3 в этом смысле несколько отличается от версии 5.

[2] (с. 141) Проблему заглядывания вперед вполне можно решить программным путем. Основная идея – не использовать процедуру READLN для чтения данных, а вызывать ее непосредственно *перед* чтением данных из новой строки. Например, программу *hiccups* можно модифицировать следующим образом:

```
PROGRAM nohiccups(INPUT, OUTPUT);
  VAR a,b: CHAR;
BEGIN
  WRITELN('первая буква');
  READ(a);
  WRITELN('вторая буква');
  READLN; READ(b);
  WRITELN(a,b,'!');
END.
```

Эта программа будет работать правильно (как показано внизу с. 141) при использовании компилятора с заглядыванием вперед (но *неправильно* в случае современного компилятора типа Турбо Паскаль).

12

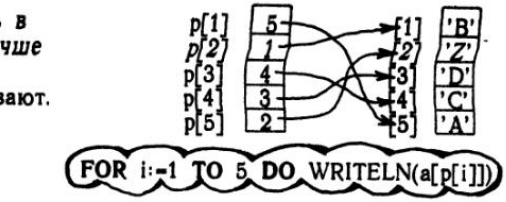
ДИНАМИЧЕСКАЯ ПАМЯТЬ

ДИНАМИЧЕСКАЯ ПАМЯТЬ
ПРОЦЕДУРЫ *NEW* И *DISPOSE*
СТЕКИ И ОЧЕРЕДИ
ОБРАТНАЯ ПОЛЬСКАЯ НОТАЦИЯ
RAJSPCKAYA (ПРИМЕР)
ПРОСТЫЕ ЦЕПИ
КРАТЧАЙШИЙ ПУТЬ (ПРИМЕР)
КОЛЬЦА
АСТРА (ПРИМЕР)
ДВОИЧНЫЕ ДЕРЕВЬЯ
ОБЕЗЬЯНЬЯ СОРТИРОВКА (ПРИМЕР)

ДИНАМИЧЕСКАЯ ПАМЯТЬ

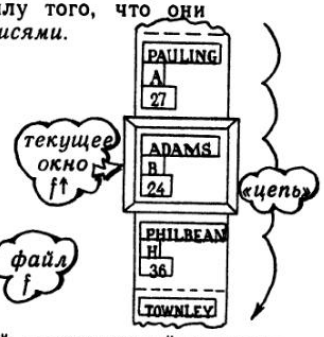
ВВОДЯТСЯ УКАЗАТЕЛИ И ДИНАМИЧЕСКИЕ ЗАПИСИ

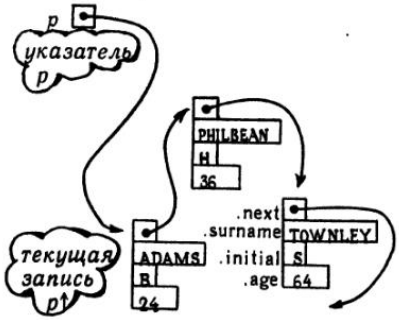
Идея *указателя* уже встречалась в контексте *сортировки массива*. Лучше переставлять указатели, нежели компоненты, на которые те указывают. Указателями были *целые числа*, лежащие в интервале изменения индексов массива.



Если есть возможность разместить сортируемые элементы в обычном массиве, то указателями на эти элементы, как показано выше, могут быть целые числа. Однако ввиду негибкости массивов использование этой структуры не всегда удобно. В ателье проката, скажем, никогда не хранится по одному свадебному платью или выходному костюму для *каждого* из зарегистрированных клиентов, поскольку невероятно, чтобы все они в один день венчались или созывали гостей. По тем же причинам непрактично объявлять каждую переменную-массив массивом с наибольшими допустимыми размерами. Руководствуясь принципом «больше овец – меньше козлов» Паскаль запасает «кучу» коробочек для хранения данных. По мере поступления данных коробочки достаются из кучи и собираются в записи. Если, к примеру, первым элементом считываемых данных является отсчет температуры, то для хранения значения изготовляется контейнер типа REAL. Если следующий элемент содержит сложную персональную запись, то взятые из кучи коробочки собираются в контейнер соответствующего типа. Если запись более не требуется, то от ее контейнера можно избавиться, вернув составляющие его коробочки обратно в кучу. Такие записи в силу того, что они появляются и исчезают, называются *динамическими записями*.

Теперь посмотрим на динамическую запись по аналогии с *файлом*. Вспомните, каждый файл связан с *файловой переменной* в виде *окна*. Если файл называется *f*, то к окну можно обратиться, используя *f*↑. Другими словами, окно не имеет своего имени: ссылка на него производится через имя файла, который представляет собой цепь таких окон





Точно также, с каждой динамической записью связан *указатель*. Если указатель называется *p*, то ссылка на динамическую запись делается посредством *p*↑. Другими словами, динамическая запись не имеет своего имени: на нее можно сослаться по имени любого указателя, который указывает непосредственно на эту запись. Создав в каждой записи компоненту, содержащую указатель на другую запись, можно построить «цепь».

Ссылка на элементы в текущей *записи* делается тем же способом, что и ссылка на элементы в текущем *окне*:

```
WRITELN(f↑.initial);   ≡   WRITELN(p↑.initial);   ⇒   B
```

Приведенные здесь указатели – *не* целого типа. Они имеют специальный тип – *указательный* (напечатать указатель, чтобы посмотреть на что он похож, нельзя). Теперь определим синтаксис объявления указательного типа:



Сравните приведенный выше синтаксис *указательного* типа с синтаксисом *файлового* типа:



Заметим, что слова FILE OF (ФАЙЛ ИЗ) соответствуют *не* словам УКАЗАТЕЛЬ НА (как можно было бы предположить), а всего лишь стрелке вверх. В этом смысле стрелка вверх должна произноситься как «указатель на» и пониматься как сокращение этого словосочетания.

Сравнивая синтаксис, заметим, что за словами FILE OF может следовать имя *типа* или *полное определение* типа. В этом примере можно убрать имя persontype, написав сразу после FILE OF определение записи. Напротив, в случае указателей такое сокращение *недопустимо*: элемент после стрелки вверх должен быть *именем* уже определенного типа.

```
TYPE
  persontype=
    RECORD
      surname: PACKED ARRAY[1..10]OF CHAR;
      initial:  CHAR;
      age:  18..65
    END;
  filetype = FILE OF persontype;

  pointertype = ↑ persontype;
```

обязательно имя

Указатель на запись полезнее всего использовать, если данная запись сама содержит указатель на другую запись. Простейшая структура данных, связанная такими указателями – это изображенная напротив «цепь». Здесь необходимы два объявления:

```
persontype = RECORD
  next: pointertype;
  surname: PACKED ARRAY[1..10] OF CHAR;
  initial: CHAR;
  age: 18..65
END;
```

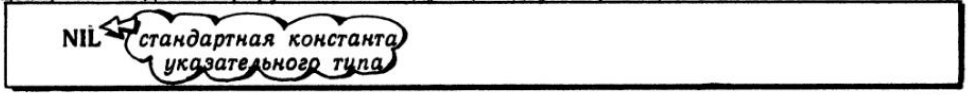
указатель на

```
pointertype = ↑ persontype;
```

Какое объявление поместить первым? Если сначала объявить *persontype*, то будет ссылка вперед на *pointertype*. С другой стороны, объявленный сперва тип *pointertype* будет ссылаться вперед на тип *persontype*. Не гоняйтесь за двумя зайцами, сперва объявляйте то, что содержит стрелку вверх. Ссылки вперед из указателей разрешены; это вынужденное исключение из правила, запрещающего ссылаться на элементы, определение которых появится позже.

Дав имена одному или нескольким указательным *типам*, можно объявить *переменную*-указатель. Это делается обычным способом в разделе VAR. В примере на следующем развороте демонстрируется объявление переменных-указателей с именами *top* и *p*; обе эти переменные принадлежат типу *pointertype*.

Существует одна стандартная *константа* указательного типа, которая не нуждается в объявлении (способов объявления указателей-констант не существует). Стандартный указатель-константа называется NIL. При обработке указателей она аналогична нулю; ее можно использовать для пометки конца цепи, что и демонстрируется на следующих двух страницах.



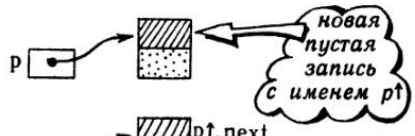
Чтобы понять, что же делает программа, приведенная на следующей странице, проще всего начать объяснение с середины. Пользователь вводит 'A', затем 'B' и программа создает такую цепь:



Теперь пользователь собирается ввести 'C' и присоединить эту букву к изображенному выше списку. Это присоединение происходит в четыре этапа (которые уже выполнены для букв A и B):

(i) создается новая запись, на которую указывает *p*. Это достигается вызовом стандартной процедуры NEW:

```
NEW(p);
```



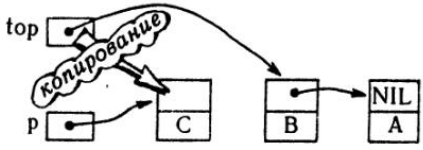
(ii) в запись помещаются данные, например

```
READLN(p↑.data);
```



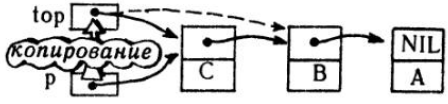
(iii) указатель из переменной *top* копируется в новую запись. Теперь новая запись возглавляет *старую* цепь (наравне с *top*)

```
p↑.next:= top;
```

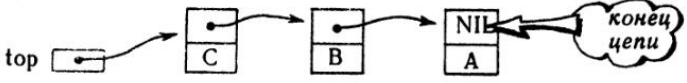


(iv) указатель из переменной *p* копируется в *top*. Теперь *top* (наравне с *p*) возглавляет *удлиненную* цепь:

```
top:= p;
```



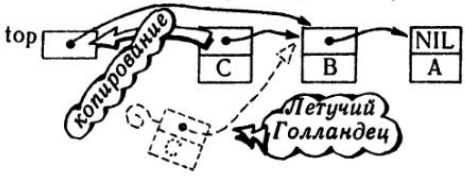
В результате получится:



Чтобы *отсоединить* от цепи первую запись, требуется всего один шаг, если только можно пожертвовать маленьким кусочком памяти (как правило, можно):

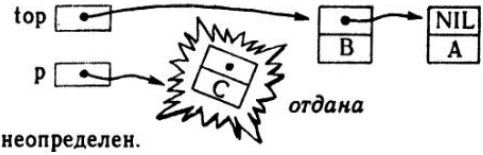
(i) указатель обреченной записи копируется в *top*. Теперь *top* указывает на следующую запись:

```
top:= top↑.next;
```



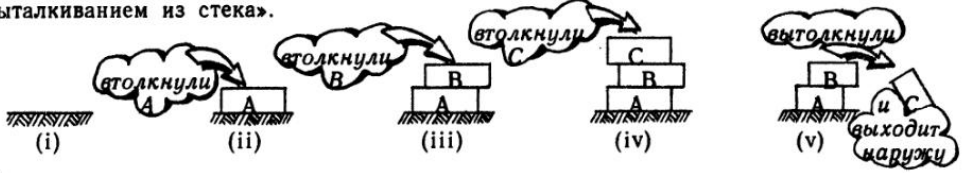
Однако если мы не можем позволить себе Летучих Голландцев, то их остовы можно вернуть в кучу для повторного использования. Для этого следует: (i) указать обреченную запись; (ii) исключить ее, как показано выше; (iii) вызвать стандартную процедуру DISPOSE. Три шага вместо одного:

```
p:= top;
top:= top↑.next;
DISPOSE(p)
```



Теперь буква 'C' исчезла; указатель *p* – неопределен.

Из предыдущего объяснения ясно видно, что запись, которая подключается последней – исключается первой. Следовательно, образ присоединения звена к цепи и отсоединения от нее можно заменить на другой – добавление в верх стопки и снятие с нее. Программистски это называют «вталкиванием в стек» и «выталкиванием из стека».



Для работы с программой, которая написана ниже, необходимо: чтобы втолкнуть букву в стек – ввести +L (или плюс любую букву); чтобы вытолкнуть из стека – ввести один знак минус (в начале строки); чтобы остановить программу – ввести звездочку (в начале строки).

```
PROGRAM  stack(INPUT,OUTPUT);
  TYPE
   pointertype = ↑ recordtype;
   recordtype = RECORD
                  next: pointertype;
                  letter: CHAR
                END;
  VAR
   top,p: pointertype;   ch: CHAR;
BEGIN
   top:= NIL;                    { инициализация пустого стека }
   REPEAT
    READ(ch);
    IF ch IN ['+','-']
     THEN
      CASE ch OF

       '+':  BEGIN  { Вталкивание }
              NEW(p);
              READLN(p↑.letter);
              p↑.next:= top;
              top:= p
             END

       '-':  BEGIN  { Выталкивание }
              IF top <> NIL          { проверка: не пуст ли стек? }
                THEN
                  BEGIN
                   WRITELN(top↑.letter,' вытолкнута');
                   p:= top;                  { возврат }
                   top:= top↑.next;
                   DISPOSE(p)
                  END
                ELSE
                  WRITELN('выталкивать нечего')
             END
      END   { CASE }
   UNTIL ch = '*'
END.
```

```
+L
+T
-
T вытолкнута
-
L вытолкнута
-
выталкивать нечего
+Q
и т.д.
```

стандартная процедура
NEW(*имя указателя*)

стандартная процедура
DISPOSE(*имя указателя*)

ИСПОЛЬЗОВАНИЕ РЕКУРСИИ ПРИ ОБХОДЕ СВЯЗАННОГО СПИСКА

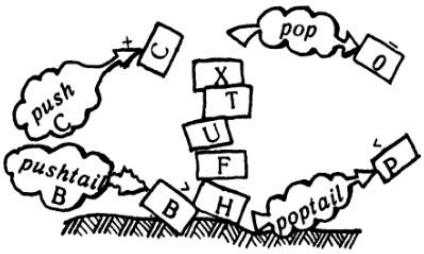
Программа на предыдущей странице была по возможности упрощена с тем, чтобы показать без лишнего тумана механизм исключения записей из головы цепи и подключения к ней. В программе, приведенной ниже, используется та же техника, лишь выделены отдельные процедуры и функции, вызываемые следующим образом:

push(ptr,ch) и ch:= pop(ptr)

Кроме того, добавлены еще две вспомогательные подпрограммы (утилиты), которые, не изменяя простой структуры стека, вталкивают элемент в стек *снизу* и выталкивают элемент из стека *снизу* соответственно:

pushtail(ptr,ch) и ch:= poptail(ptr).

Если пользоваться только подпрограммами *push* и *poptail*, то цепь будет работать как *очередь*. Элементы вталкиваются с одной стороны, стоят в очереди и выталкиваются для обслуживания с другой стороны. Использование только подпрограмм *pushtail* и *pop* означает, что такая же очередь выстраивается в обратном направлении.



Чтобы достичь низа стека, используется рекурсия. Когда вызывается процедура *pushtail*, текущее звено цепи оказывается в одном из двух состояний:

ptr NIL *или* ptr → ptr↑.next

Если ptr=NIL, то мы находимся в конце цепи, и поэтому надо заменить NIL указателем на новую запись. Если ptr<>NIL, то мы – *не* в конце цепи, и поэтому вызываем процедуру *pushtail(ptr↑.next,ch)* для вталкивания *ch*.

Рекурсия используется также и в функции *poptail*, но здесь возможны уже *три* состояния:

ptr NIL *или* ptr → NIL ptr↑.next *или* ptr → →

Если ptr=NIL, то очередь пуста. Если ptr↑.next=NIL, то в очереди – единственный элемент, который можно вытолкнуть, как если бы очередь была стеком. Если ptr↑.next<>NIL, то мы вызываем функцию *poptail(ptr↑.next)*, которая сделает то, что надо.

```
PROGRAM staque(INPUT,OUTPUT);
 TYPE
  pointertype = ↑ recordtype;
  recordtype = RECORD
                 next: pointertype;
                 data: CHAR
               END;
 VAR
  top: pointertype;
  ch:  CHAR;
```



Здесь приведены четыре утилиты; они весьма эффективно используются в программе на с. 154.

```
PROCEDURE push(VAR ptr: pointertype; c: CHAR);
  VAR
    p: pointertype;
  BEGIN
    NEW(p);
    p↑.data:= c;
    p↑.next:= ptr;
    ptr:= p
  END;
```

```
FUNCTION pop(VAR ptr: pointertype): CHAR;
  BEGIN
    pop:= CHAR(0);
    IF ptr<>NIL
      THEN
        BEGIN
          pop:= ptr↑.data;
          ptr:= ptr↑.next
        END
  END;
```

если стек пуст, то pop возвращает этот символ-невидимку

если хотите, то можете избавиться от этого Летучего Голландца

```
PROCEDURE pushtail(VAR ptr: pointertype; c: CHAR);
  BEGIN
    IF ptr=NIL
      THEN
        BEGIN
          NEW(ptr);
          p↑.data:= c;
          p↑.next:= NIL;
        END
      ELSE
        pushtail(ptr↑.next,c)
  END;
```

рекурсия

```
FUNCTION poptail (VAR ptr:pointertype):CHAR;
  BEGIN
    poptail:= CHR(0);
      IF ptr<>NIL
        THEN
          IF ptr↑.next = NIL
            THEN poptail:=pop(ptr)
            ELSE poptail:=poptail(ptr↑.next)
  END;
```

пусто; см. выше

рекурсия

```
BEGIN  { staque }
  top:= NIL;
  REPEAT
    READ(ch);
    IF ch IN ['+', '-', '>', '<']
      THEN
        CASE ch OF
          '+': BEGIN
                 READLN(ch);
                 push(top,ch)
               END;
          '-': WRITELN(pop(top));
          '>': BEGIN
                 READLN(ch);
                 pushtail(top,ch)
               END;
          '<': WRITELN(poptail(top))
        END
  UNTIL ch='*'
END.
```

push pop pushtail poptail

используйте эту программу как программу со с. 149 — но попробуйте две новые возможности:

+L	*втолкнуть 'L' (или любую букву в стек)*
>L	*втолкнуть букву в стек снизу*
-	*вытолкнуть из стека*
<	*вытолкнуть из стека снизу*
***	*остановка*

ОБРАТНАЯ ПОЛЬСКАЯ НОТАЦИЯ

ИЛЛЮСТРАЦИЯ ИСПОЛЬЗОВАНИЯ СТЕКОВ

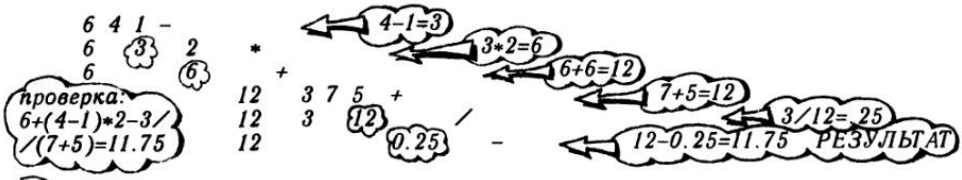
Обычные алгебраические выражения можно записывать также в обратной польской нотации – записи без скобок. Нотация «польская», потому что ее предложил польский математик Jan Łukasiewicz; произнести это правильно могут только поляки, по-русски говорят Ян Лукасевич или Лукашевич. Нотация «обратная», потому что в сравнении с исходным вариантом порядок операндов и операций был обращен. Пример обратной польской нотации:

A + (B – C) * D – F/(G + H) преобразуется в ABC–D*+FGH+/–

Вычислить обратное польское выражение проще, чем это может показаться. Пусть, например, A=6, B=4, C=1, D=2, F=3, G=7, H=5. С этими значениями *вычисляемое выражение запишется так:*

6 4 1 – 2 * + 3 7 5 + / –

Просматриваем все элементы слева направо. Как только встречается операция, *выполняем ее по отношению к двум предыдущим элементам*, заменяя два *элемента* одним:

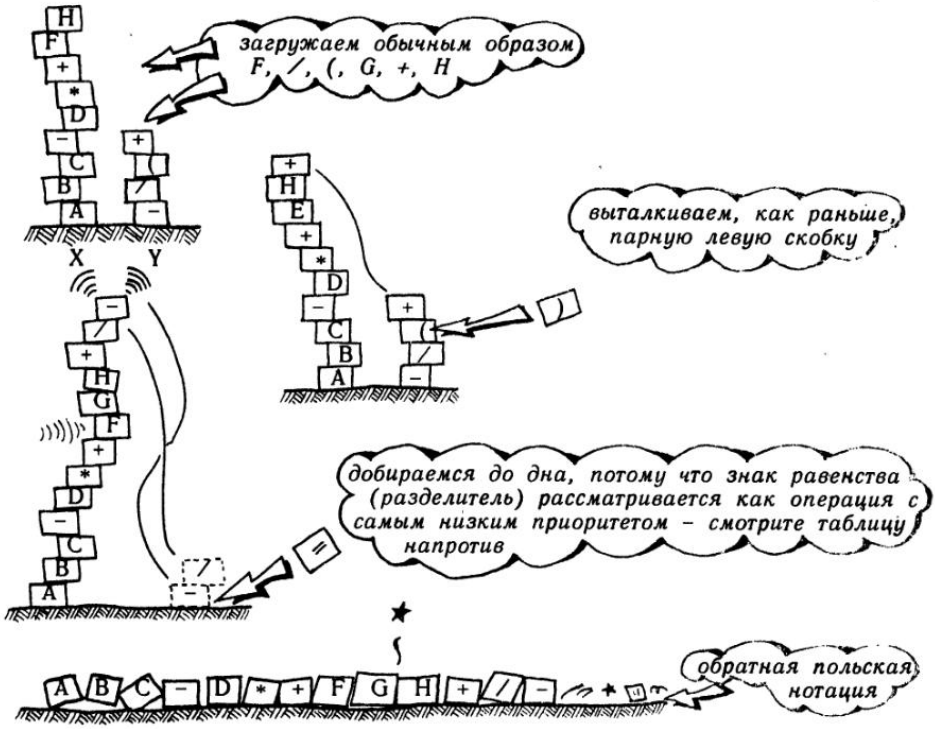


Этот пример призван продемонстрировать, что обратная польская запись может быть весьма полезной при вычислении выражений на компьютере. Итак, с чего начать преобразование выражения вида A+(B–C)*D–F/(G+H)? Здесь используются два стека; последовательность действий поясняется ниже.



оператор	*приоритет*
*	3 (выс-
/	3 ший)
+	2
–	2
(	1
=	0

Заметьте, что левая скобка включена в таблицу приоритетов с низким приоритетом. Этот трюк позволяет избежать проверки дополнительного условия «*~ или не оказывается левой скобкой*». Ловко придумано.



Наряду с процедурой *push(stack,ch)* и функциями *pop(stack)* и *poptail(stack)* необходима функция, которая бы возвращала приоритет операции. Приведенная ниже функция получает в качестве параметра литеру и возвращает целое в соответствии с таблицей приоритетов:

```
FUNCTION prec(c: CHAR): INTEGER;
  BEGIN
   CASE c OF
   '*', '/' : prec:= 3;
   '+', '-' : prec:= 2;
   '(' : prec:= 1;
   '=' : prec:= 0
   END
  END;
```

см. маленькую табличку напротив

На следующей странице приведена программа, которая преобразует выражение в традиционной записи в обратную польскую нотацию. Для использования программы наберите выражение, закончив его знаком равенства:

```
A+(B-C)*D-F/(G+H)=
ABC-D*+FGH+/-
```

введите это

получите в результате

ПРИМЕР ДЛЯ ИЛЛЮСТРАЦИИ ИСПОЛЬЗОВАНИЯ СТЕКОВ (инструкции к использованию – в конце предыдущей страницы)

```
PROGRAM hsilop(INPUT,OUTPUT);  { яаксьлоп }
  TYPE
    pointertype = ↑ recordtype;
    recordtype = RECORD
                   next: pointertype;
                   data: CHAR
                 END;
  VAR
    x,y: pointertype;                    { стеки x и y }
    ch: CHAR;      i: 0..40;   exit: BOOLEAN;

    { здесь вставьте процедуры и функции с
      предыдущих страниц: push, pop, poptail, prec }

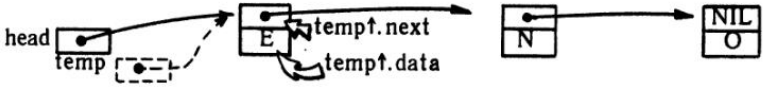
  BEGIN   { hsilop }
    x:= NIL;   y:= NIL;                  { инициализация стеков }
    REPEAT
      READ(ch);
      IF ch IN ['A'..'Z'] THEN push(x,ch);
      IF ch = '(' THEN push(y,ch);
      IF ch = ')'
        THEN
          BEGIN                          { вытолкнуть до соответствующей
            WHILE y↑.data<>'(' DO          левой скобки }
              push(x,pop(y));
            ch:= pop(y)                  { затем выкинуть ее вон }
          END;
      IF ch IN ['+', '-', '*', '/', '=']
        THEN
          BEGIN
            REPEAT
              exit:= TRUE;
              IF y<>NIL                  { если приоритет операции,
                THEN                       что загружается сверху ...
                  IF prec(ch)< prec(y↑.data)   { ... <= приоритета
                    THEN                         операции или левой
                      BEGIN                      скобки снизу
                        push(x,pop(y));
                        exit:= FALSE
                      END;
              UNTIL exit;                { только теперь будет
              push(y,ch)                   правильным втолкнуть в стек
                                           новую операцию }
          END
    UNTIL ch = '=';
    WHILE x<>NIL DO WRITE(poptail(x));   { печать стека снизу вверх }
    WRITELN;
  END.
```

ПРОСТЫЕ ЦЕПИ

МОДЕЛИ ДЛЯ «ОБХОДА» И «ВСТАВКИ ПОСЛЕ»

Отличительной чертой *стека* или *очереди* является то, что вызов записи означает ее *удаление*. (В предыдущем примере программа жульничает, глядя на запись перед тем, как вытолкнуть ее из верхушки стека.) Однако, существует много приложений, в которых последовательные записи извлекаются но не исключаются из цепи. Такая последовательная обработка называется *обходом*.

Ниже показана обычная цепь и фрагмент программы для ее обхода. В этом примере «обработка записи» есть не более чем печать одной из ее компонент, однако в общем случае это может быть более сложная процедура.



МОДЕЛЬ ДЛЯ ОБХОДА (НЕ РЕКУРСИВНАЯ)

```
{ traversal }
temp:= head;
WHILE temp<>NIL DO
  BEGIN
    WRITE(temp↑.data);
    temp:= temp↑.next
  END
```

обработка данных

переход к следующей записи

Чтобы вставить элемент после некоторого:

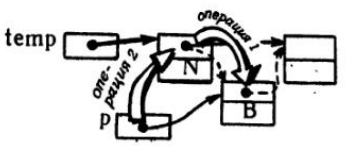
```
{ вставка 'B' после 'N' }
temp:= head;
WHILE temp↑.data<>'N' DO
  temp:= temp↑.next;
NEW(p);
p↑.data:= 'B';
p↑.next:= temp↑.next;
temp↑.next:=p
```

обход для нахождения N

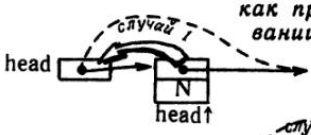
создание 'B'

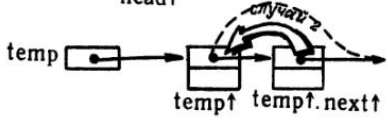
операция 1 затем операция 2

МОДЕЛЬ ДЛЯ «ВСТАВКИ ПОСЛЕ»



как при выталкивании из стека





Чтобы удалить элемент:

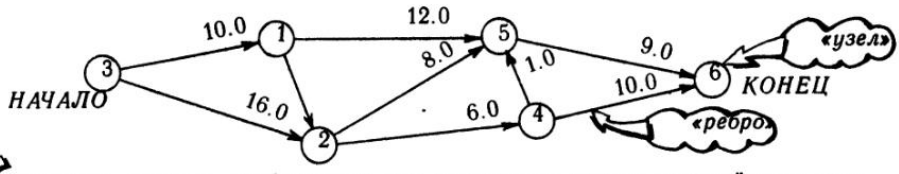
```
{ удаление 'N' }
IF head↑.data = 'N'
  THEN                    { случай 1 }
    head:= head↑.data
  ELSE                    { случай 2 }
    BEGIN
      temp:= head
      WHILE temp↑.next↑.data<>'N' DO
        temp:= temp↑.next;
      temp↑.next:= temp↑.next↑.next
    END
```

Неуклюже! Если необходимы выборочное уничтожение или «вставка перед», лучше использовать *дважды связанные кольца* (см. позже), нежели простые цепочки.

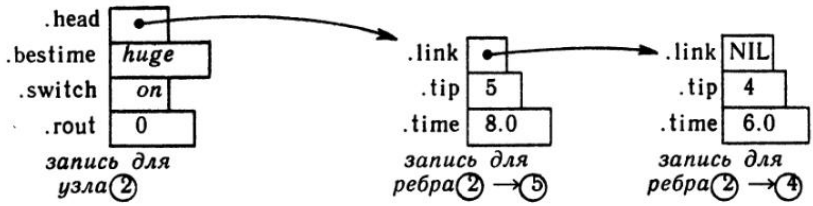
КРАТЧАЙШИЙ ПУТЬ

ПРИМЕР ДЛЯ ИЛЛЮСТРАЦИИ ИСПОЛЬЗОВАНИЯ ЦЕПЕЙ

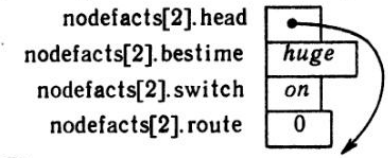
Задача поиска кратчайшего (или самого длинного) пути через сеть возникает во многих приложениях ~ примером может служить определение *критического пути графика работ* в техническом планировании. Пусть имеется сеть наподобие изображенной ниже. Задача заключается в поиске кратчайшего пути из узла с пометкой НАЧАЛО к узлу с пометкой КОНЕЦ. Двигаться можно только в направлении стрелок. Число возле каждой стрелки указывает время в пути.



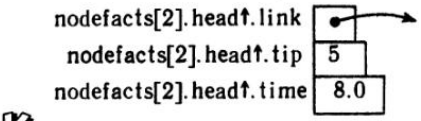
Структура данных, необходимая для программы поиска кратчайшего пути, нарисована ниже. Каждому *узлу* отвечает своя запись и цепь, начинающаяся от этой записи. Каждая такая цепь включает записи с информацией обо всех *ребрах, выходящих из* этого узла.



Записи для всех узлов объединены в массив *nodefacts*. Ниже более подробно рассматривается запись для узла 2. В компоненте *bestime* находится значение *huge* (константа, равная 10^{20}). В компоненте *switch* (переключатель) – логическое значение; первоначально это – *on* (вкл). Использование этих элементов поясняется ниже.



Записи для ребер, выходящих из узла, создаются динамически. В каждой записи есть компонента для хранения связи, компонента для хранения номера узла в конце ребра и компонента для хранения времени путешествия по самому ребру. Этот пример – для ребра (2) → (5).



Кратчайший путь ищется в ходе итеративного процесса. Перед началом процесса должны быть сформированы все цепи и во все компоненты должны быть помещены начальные значения; в дальнейшем они, возможно изменятся. Компонента *bestime* будет содержать лучшее время достижения данного узла по испробованным к текущему моменту путям. Изначально это время устанавливается столь большим, что первый же возможный путь, каким бы он ни был медленным, это время улучшит. Исключением является начальный узел: в начальном узле, по определению, лучшее время – нулевое.

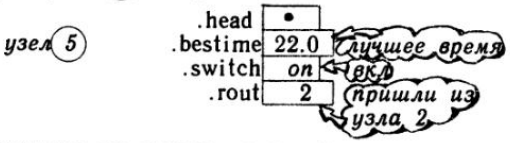
Поначалу все переключатели устанавливаются в положение *on*. Установленный в положение *on* переключатель означает, что ребра, выходящие из этого узла, должны быть еще исследованы (в первый раз или повторно).

Итеративный процесс стартует в начальном узле и вплоть до завершения циклически обрабатывает массив узловых записей. Процесс завершается, когда все переключатели оказываются в положении *off*.

В каждом узле производится обход цепи записей ребер. Для каждого ребра в цепи вычисляется время, необходимое для достижения узла на его конце. Для этого к лучшему на данный момент времени достижения исходного узла прибавляется время путешествия по ребру. Результат сравнивается с лучшим временем, которое хранится в *узловой записи* этого конечного узла. Если новое время лучше, то следует сделать несколько действий, которые нарисованы ниже:



Всякий раз, когда обнаруживается лучший путь к узлу, вместо старого записывается лучшее время и переключатель переводится в положение *on*, как нарисовано выше для узла 5. Для того чтобы впоследствии можно было проследить найденный путь, используется компонента *route*, содержащая номер узла, через который проходит найденный путь. Таким образом, в результате обработки ребра, ведущего из узла (2) в узел (5) получается:



После обхода цепочки ребер, начинающихся во втором узле, переключатель switch узла 2 устанавливается в положение *off*. Тем не менее в результате обработки узла 2 переключатель в узле 5 был установлен в положение *on*, поэтому итерации еще не закончены. Процесс продолжается до тех пор, пока все переключатели не окажутся в положении *off* ~ другими словами, пока в результате цикла по всем узлам не выяснится, что сделать какое-либо улучшение пути уже невозможно.

Узловые записи скомпонованы в массив, а не формируются динамически со связыванием в цепь. Структура массива была выбрана ввиду того, что узловые записи обрабатываются в «случайном» порядке (например, при работе с узлом 2, вам понадобятся также узлы 5 и 4). При использовании массива такие ссылки делаются быстро и просто при помощи варьирования индекса.

Попробуйте выполнить программу для сети напротив. Данные и результаты (предполагается интерактивная работа) должны быть такими, как показано здесь[2)].

Число узлов	Число ребер	Начальный узел	Конечный узел
6	9	3	6
3 1 10.0			
3 2 16.0			
1 2 5.0			
1 5 12.0			
2 4 6.0			
2 5 8.0			
5 6 9.0			
4 6 10.0			
4 5 1.0			
Путь из 6 в 3			
6...4...2...1...3			
Требуемое время 31.0			

КРАТЧАЙШИЙ ПУТЬ

(ПОЛНАЯ ПРОГРАММА)

```
PROGRAM network(INPUT,OUTPUT);
 CONST
   on = TRUE;     off = FALSE;
   huge = 1E20;       nothing =0.0;
   maxnodes = 30;       maxedges = 50;

 TYPE
   nodetype = 0..maxnodes; edgetype = 0..maxedges;
   pointertype = ↑chaintype;

   chaintype = RECORD
                 link: pointertype;
                 tip: nodetype;
                 time: REAL
               END;

   rectype = RECORD
               head: pointertype;
               bestime: REAL;
               switch: BOOLEAN;
               route: nodetype
             END;

   arraytype = ARRAY[nodetype] OF rectype;

 VAR
  nodes, startnode, endnode, i, n, tail: nodetype;
  edges, j: edgetype;
  edge, p: pointertype;
  nodefacts: arraytype;
  cycles: 0..2;   try: REAL;

BEGIN
 WRITELN('Число     Число    Начальный   Конечный');
 WRITELN('узлов     ребер      узел        узел');
 READLN(nodes,edges,startnode,endnode);

 FOR i:=1 TO nodes DO
   WITH nodefacts[i] DO
     BEGIN
       head:= NIL;
       bestime:= huge;
       switch:= on;
       route:= 0
     END;          { WITH }
 nodefacts[startnode].bestime:= nothing;

 FOR j:=1 TO edges DO
   BEGIN
     NEW(p);
     READLN(tail,p↑.tip,p↑.time);
     p↑.link:= nodefacts[tail].head;
     nodefacts[tail].head:= p
   END;
```

массив узловых записей

инициализация

замена времени в начальном узле

формирование всех цепей

чтение данных

подключение к цепи новой записи

```
cycles:= 0;
n:= startnode-1;
WHILE cycles < 2 DO
  BEGIN
    cycles:= SUCC(cycles);
    n:= n MOD nodes + 1;
    IF nodefacts[n].switch = on
      THEN
        BEGIN { IF switch }
          cycles:= 0;
          edge:= nodefacts[n].head;

          WHILE edge<>NIL DO
            BEGIN  { WHILE edge }
              try:= nodefacts[n].bestime + edget.time;
              IF try < nodefacts[edget.tip].bestime
                THEN
                  WITH nodefacts[edget.tip] DO
                    BEGIN
                      bestime:= try;
                      route:= n;
                      switch:= on
                    END;
              edge:= edget.link
            END;    { WHILE edge }
          nodefacts[n].switch:= off
        END  { IF switch }
END; { WHILE cycles }

WITH nodefacts[endnode] DO
  IF (bestime <> huge) AND (bestime <> nothing)
    THEN
      BEGIN
        WRITELN('Путь из',endnode:3,' в',startnode:3);
        n:= endnode;

        WHILE n <> 0 DO
          BEGIN
            WRITE(n:1);
            n:=nodefacts[n].route;
            IF n<>0 THEN WRITE('...')
          END;
        WRITELN;
        WRITELN('Требуемое время ',bestime:6:2)
      END
    ELSE
      WRITELN('Пути нет - или идти некуда')
END.
```

перед использованием n увеличивается на 1, поэтому -1 заранее

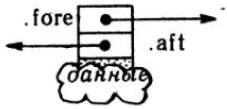
опустите «=on», если так будет яснее

ширина поля автоматически увеличивается до 2, если номер узла состоит из двух цифр

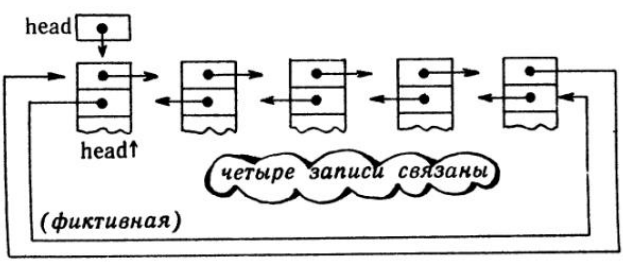
смотрим назад на предыдущий узел

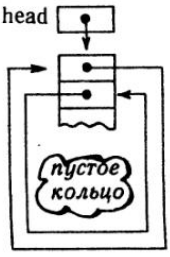
например, 6...4...2...1...3

Каждая запись дважды связанного кольца содержит указатели вперед и назад:



Доступ к записям в кольце упрощается, если добавить к ним еще одну фиктивную головную запись, с которой начинается кольцо, как это показано ниже. Такой подход избавляет от необходимости специально проверять, находится ли уничтожаемая запись в начале или попадет ли туда добавляемая запись.





Выше изображено кольцо с четырьмя записями и пустое кольцо.

Справа – определение записи, подходящей для коиструирования кольца. Для простоты в этой записи хранится всего одна литера.

```
TYPE
  pointertype = ↑ recordtype;

  recordtype =  RECORD
                  fore,aft: pointertype;
                  data: CHAR
                END;
VAR
  head,temp: pointertype;
```

В главной программе пустое кольцо можно создать, например, следующим образом.

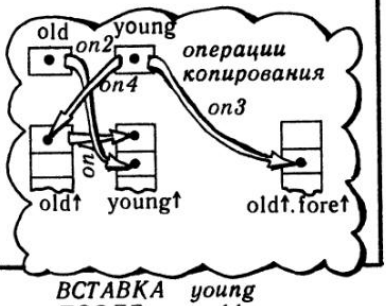
```
NEW(head);
head↑.fore:= head;
head↑.aft:= head;
```

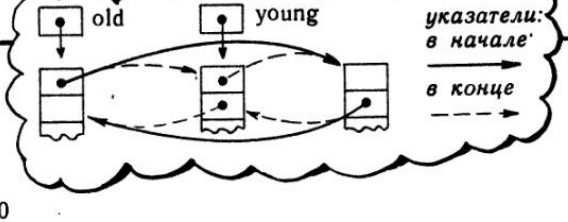
Новую запись можно вставить до *или* после указанной записи. Ниже представлены процедуры, реализующие эти две операции:

```
PROCEDURE  inafter(old,young: pointertype);
  BEGIN

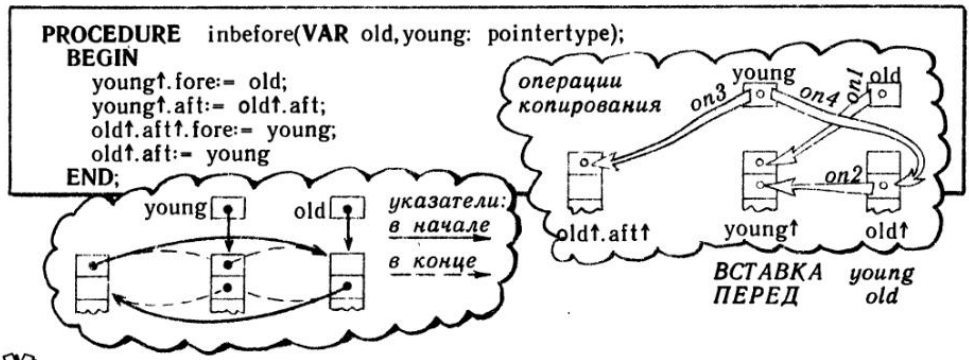
    young↑.fore:= old↑.fore;
    young↑.aft:= old;
    old↑.fore↑.aft:= young;
    old↑.fore:= young

  END;
```



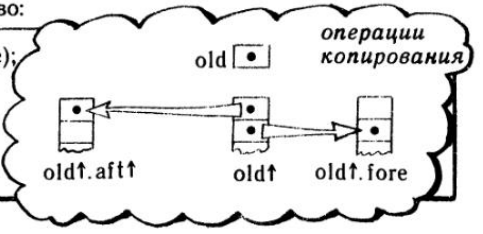


```
PROCEDURE inbefore(VAR old,young: pointertype);
  BEGIN
    young↑.fore:= old;
    young↑.aft:= old↑.aft;
    old↑.aft↑.fore:= young;
    old↑.aft:= young
  END;
```

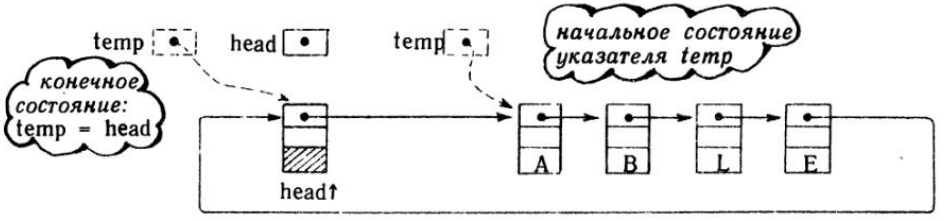


Удаление записи делается просто и красиво:

```
PROCEDURE delete(VAR old: pointertype);
  BEGIN
    old↑.fore↑.aft:= old↑.aft;
    old↑.aft↑.fore:= old↑.fore
  END;
```



Обход в любом направлении прост. Единственная трудность – вовремя остановиться. Если цель – обойти кольцо в точности один раз, то начинайте с указателя на первую запись и предусмотрите остановку, как только указатель укажет на фиктивную головную запись (перед выборкой данных из нее).




```
temp:= head↑.fore;
WHILE temp<>head DO
  BEGIN
    WRITE(temp↑.data);
    temp:= temp↑.fore
  END;
WRITELN
```



Если заменить «fore» на «aft» в обоих местах, то результатом этого кусочка программы было бы не ABLE, а ELBA.

На следующей странице – демонстрационная программа, разработанная с целью изучения принципов и процедур, приведенных на этом развороте.

ПРОГРАММА-ПРИМЕР ДЛЯ ДЕМОНСТРАЦИИ РАБОТЫ ДВАЖДЫ СВЯЗАННОГО СПИСКА

Следующая программа реализует дважды связанное кольцо, организованное в алфавитном порядке. Чтобы ввести букву, наберите в начале строки +Б (или + любую букву). Чтобы вычеркнуть букву, введите -Б (или - какую-то букву). Для вывода хранящихся букв в алфавитном порядке введите в начале строки литеру >. Для вывода в обратном порядке - введите <. Для остановки введите в начале строки литеру *.

```
PROGRAM aster(INPUT,OUTPUT);

  TYPE

    pointertype = ↑ recordtype;
    recordtype = RECORD
                   fore,aft: pointertype;
                   data: CHAR
                 END;

  VAR
    ch: CHAR;
    head,p,temp: pointertype;
    caps, operators: SET OF CHAR;

  PROCEDURE  inbefore(VAR old,young: pointertype);
    BEGIN
      young↑.fore:= old;
      young↑.aft:= old↑.aft;
      old↑.aft↑.fore:= young;
      old↑.aft:= young
    END;

  PROCEDURE  delete(VAR old: pointertype);
    BEGIN
      old↑.fore↑.aft:= old↑.aft;
      old↑.aft↑.fore:= old↑.fore
    END;

BEGIN

  caps:= ['A'..'Я'];
  operators:= ['+', '-', '>', '<'];

  NEW(head);
  head↑.fore:= head;
  head↑.aft:= head;
  head↑.data:= CHR(0);

  REPEAT

    READ(ch);
    IF ch IN operators
      THEN
```

+A
+C
>
AC
+T
+P
+A
>
AAPCT
-A
<
TCPA
*

процедуры inbefore *и* delete - *как на предыдущей странице*

Создание пустого кольца. Чтобы предотвратить аварийную ситуацию, о которой говорится на следующей странице, поместим в фиктивную головную запись пустую литеру CHR(0)

```
      CASE ch OF

      '+': BEGIN
             READ(ch);
             IF ch IN caps
               THEN
                 BEGIN
                   NEW(p);
                   p↑.data:= ch;
                   temp:= head↑.fore;
                   WHILE (temp<>head) AND (temp↑.data<ch) DO
                     temp:= temp↑.fore;
                   inbefore(temp,p)
                 END
           END;

      '-': BEGIN
             READ(ch);
             IF ch IN caps
               THEN
                 BEGIN
                   temp:= head↑.fore;
                   WHILE (temp<>head) AND (temp↑.data<>ch) DO
                     temp:= temp↑.fore;
                   IF temp<>head
                     THEN
                       delete(temp)
                 END
           END;

      '>': BEGIN
             temp:= head↑.fore;
             WHILE temp<>head DO
               BEGIN
                 WRITE(temp↑.data);
                 temp:= temp↑.fore;
               END;
             WRITELN
           END;

      '<': BEGIN
             temp:= head↑.aft;
             WHILE temp<>head DO
               BEGIN
                 WRITE(temp↑.data);
                 temp:= temp↑.aft;
               END;
             WRITELN
           END
      END  { CASE }

    UNTIL ch = '*'

END.  { aster }
```

Остерегайтесь возможной аварии. Условие temp↑.data<ch *будет проверяться, даже если условие* temp<>head *ложно. Поэтому* temp↑.data *не должно оставаться неопределенным в фиктивной записи. Поэтому там* – CHR(0)

вставка

удаление

вывод в порядке возрастания

вывод в порядке убывания

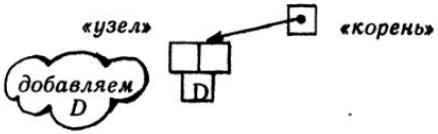
*работа заканчивается на **

ДВОИЧНЫЕ ДЕРЕВЬЯ

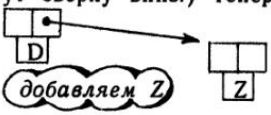
ЕЩЕ ОДНА КРАСИВАЯ СТРУКТУРА

Пусть требуется отсортировать несколько букв:

D, Z, B, E, A, F, C



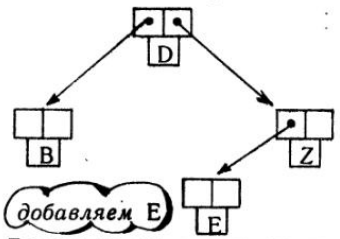
Первую букву (D) запишем в узел, который поместим в корень *дерева*. (В программировании деревья растут сверху вниз.) Теперь возьмем следующую букву – Z,



и подведем ее к корневому узлу. Она «больше», чем D, поэтому идем *вправо* и создаем, как здесь показано, новый узел для хранения Z.

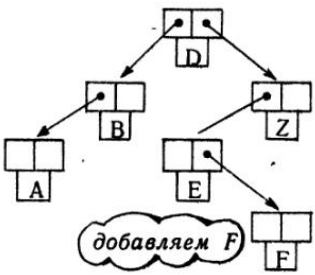
Теперь третья буква, B. Она меньше, чем D, поэтому идем *влево* и создаем новый узел.

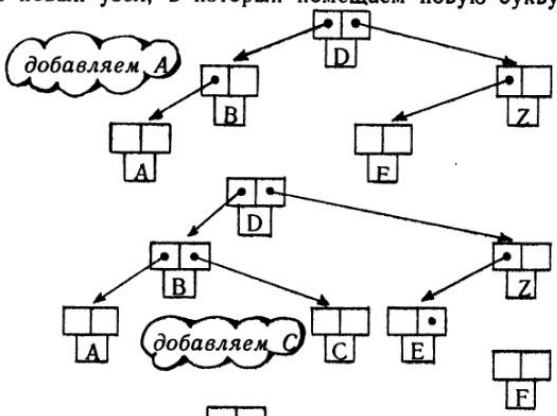




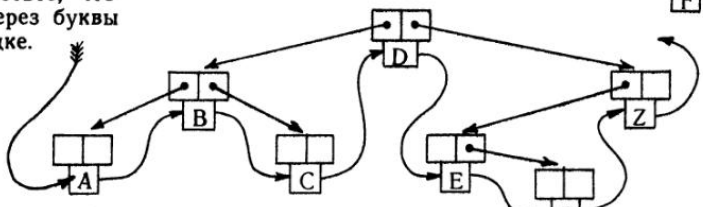
Следующая буква – E. Она больше, чем D, поэтому идем *вправо*. Она меньше, чем Z, поэтому идем *влево*. Затем создаем, как здесь показано, узел для хранения E.

В общем случае: сравниваем очередную букву с буквой в корневом узле. Если новая буква меньше, идем *влево*; если *больше*, идем *вправо*. Делаем то же в каждом из достигнутых узлов, пока узлы с буквами для сравнения не иссякнут. Затем создаем новый узел, в который помещаем новую букву.





В любой момент можно обойти (или *выпрямить*) дерево, как показано ниже. Заметьте, что стрелка проходит через буквы в алфавитном порядке.



Порядок может быть и обратным, если обходить иначе.

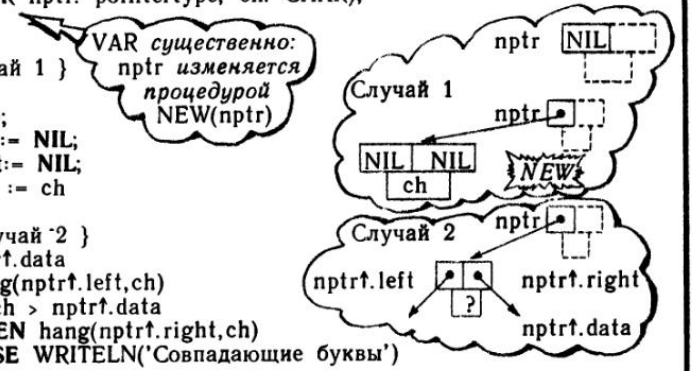
Тип нарисованной напротив узловой записи определяется несложно:

```
TYPE
  pointertype = ↑ nodetype;
  nodetype = RECORD
               left,right: pointertype;
               data: CHAR
             END;
```



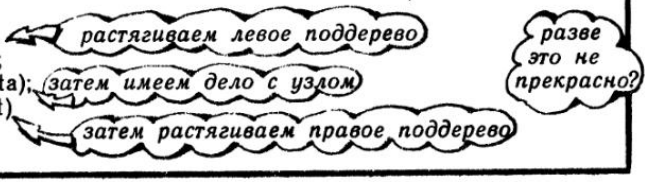
Подвешивать к дереву буквы ~ напротив этот процесс показан по стадиям ~ лучше всего рекурсивно. Если текущий узел – пустой (NIL), то для хранения новой буквы создается новый узел; в противном случае вызываем процедуру добавления *hang* с параметром, специфицирующим правый или левый указатель в зависимости от того, как новая буква соотносится с текущей:

```
PROCEDURE hang(VAR nptr: pointertype; ch: CHAR);
  BEGIN
    IF nptr = NIL
      THEN { Случай 1 }
        BEGIN
          NEW(nptr);
          nptr↑.left := NIL;
          nptr↑.right:= NIL;
          nptr↑.data := ch
        END
      ELSE     { Случай 2 }
        IF ch < nptr↑.data
          THEN hang(nptr↑.left,ch)
          ELSE IF ch > nptr↑.data
                 THEN hang(nptr↑.right,ch)
                 ELSE WRITELN('Совпадающие буквы')
END;
```



Обход дерева можно выполнить рекурсивно:

```
PROCEDURE strip(VAR nptr: pointertype);
  BEGIN
    IF nptr <> NIL
      THEN
        BEGIN
          strip(nptr↑.left);
          WRITE(nptr↑.data);
          strip(nptr↑.right)
        END
END;
```



В обеих приведенных выше процедурах можно использовать «WITH nptr↑ DO», сократив число появлений «nptr↑» ценой лишних строк и меньшей ясности. Слово VAR в процедуре обхода не обязательно с точки зрения логики работы, однако оно предотвращает копирование структуры данных при каждом обращении копии структуры данных. Ох![3]

Перевернув страницу, вы найдете программу, использующую двоичное дерево. Она считывает набранные в любом порядке буквы и выводит их на экран в алфавитном порядке. Добавление возможности вывода букв в обратном порядке оставлено в качестве упражнения.

Двоичные деревья полезны в работе любого сорта, не только в сортировке.

ОБЕЗЬЯНЬЯ СОРТИРОВКА

ИЛИ СОРТИРОВКА С ПОМОЩЬЮ ДВОИЧНОГО ДЕРЕВА

Эта программа использует двоичное дерево во многом аналогично тому, как программа АСТРА использует дважды связанное кольцо. Чтобы подвесить к дереву новую букву, введите +Б (или + любую букву). Для удаления буквы введите -Б (или минус любую букву). Для вывода букв в алфавитном порядке введите в начале строки символ >. Для остановки введите в начале строки символ *.

Добавление к дереву делается просто и изящно, однако удаление узла, который не является «листом» ~ особенно если на дереве возможны одинаковые элементы ~ отнюдь не просто. В этой программе используются счетчики одинаковых элементов. Если элемент удаляется, значение счетчика уменьшается.

```
PROGRAM monkey(INPUT,OUTPUT);

  TYPE
   pointertype = ↑nodetype;

   nodetype = RECORD
                 left,right: pointertype;
                 data: CHAR;
                 count: INTEGER
               END;

  VAR
    root,p: pointertype;
    ch: CHAR;

PROCEDURE hang(VAR nptr: pointertype; ch: CHAR); { добавить букву }
  BEGIN
    IF nptr = NIL
      THEN
        BEGIN
          NEW(nptr);
          nptr↑.left := NIL;
          nptr↑.right:= NIL;
          nptr↑.data := ch;
          nptr↑.count:= 1
        END
      ELSE
        IF ch < nptr↑.data
          THEN hang(nptr↑.left,ch)
          ELSE IF ch > nptr↑.data
                 THEN
                   hang(nptr↑.right,ch)
                 ELSE
                   nptr↑.count:= nptr↑.count + 1
END;
```

```
+Б
+А
+Б
+У
+И
+Н
>
АББИНУ
-Б
>
АБИНУ
*
```

при повторении увеличиваем счетчик

Следующая функция служит для нахождения подлежащей удалению буквы. Написаная рекурсивно, эта функция основывается на тех же принципах, что и процедура *hang*.

```
FUNCTION find(VAR nptr: pointertype; ch: CHAR): pointertype;
  BEGIN                                                    { найти узел }
    IF nptr = NIL
      THEN find:= NIL
      ELSE IF ch < nptr↑.data
             THEN find:= find(nptr↑.left,ch)
             ELSE IF ch > nptr↑.data
                    THEN find:= find(nptr↑.right,ch)
                    ELSE find:= nptr
END;

PROCEDURE strip(VAR nptr: pointertype); { обход дерева }
  VAR
    i: 0..MAXINT;
  BEGIN
    IF nptr <> NIL
      THEN
        BEGIN
          strip(nptr↑.left);
          FOR i:=1 TO nptr↑.count DO
            WRITE(nptr↑.data);
          strip(nptr↑.right)
        END
  END;

BEGIN { monkey }
  root:= NIL;
  REPEAT
    READ(ch);
    IF ch IN ['+','-','>']
      THEN
        CASE ch OF
        '+': BEGIN
               READ(ch);
               IF ch IN ['A'..'Я']
                 THEN
                   hang(root,ch)
             END;

        '-': BEGIN
               READ(ch);
               p:=find(root,ch);
               IF p <> NIL
                 THEN IF p↑.count > 0
                        then p↑.count:= p↑.count - 1
             END;

        '>': BEGIN
               strip(root);
               WRITELN
             END
        END { CASE }
  UNTIL ch = '*'
END.
```

если не нашли, возвращаем NIL

нашли

например, если значение счетчика = 2, печатаем букву дважды; если в счетчике 0, то ничего не печатаем

по существу, удаление одной буквы

УПРАЖНЕНИЯ

1. Напишите программу, которая будет считывать арифметические выражения, наподобие этого:

```
3.5 * (7 + (4 - 6.2) / 32)
```

и печатать результат. Для считывания чисел и операций, входящих в выражение, воспользуйтесь процедурой типа процедуры *grab* (с. 128–133). Используйте логику программы построения обратной польской записи (с. 152–154), внеся в нее важное изменение: непосредственно перед переносом операции из стека Y в стек X, сделайте по-другому:

- вытолкните из стека X два числа
- примените к ним данную операцию
- втолкните результат в стек X

Если действовать по этой схеме, то в конце концов в стеке X останется единственное число – это и будет значение выражения.

2. Напишите приключенческую игру, где игрок исследует волшебный дворец или зловонное подземелье, переходя из одного помещения в другое, поднимая и ставя вещи, одновременно воюя с монстрами. Для написания такой программы потребуются процедуры обработки строк: ведь игроку необходимо будет получать от компьютера вразумительные ответы на свои запросы типа:

```
ВЗЯТЬ ЯД
```

или

```
ИДТИ НА ЗАПАД
```

Нужные процедуры разработаны в следующей главе. Описание простой и вместе с тем законченной приключенческой игры есть в моей книге:

Illustrating Super-BASIC C.U.P. 1985

Там используются кольцевые структуры, чтобы можно было брать предметы в одной комнате и класть их в другой; матрицы состояний – для описания топологии комнат и дверей; таблицы состояний – для кодирования правил игры. Изложенных нами приемов уже вполне достаточно для создания законченной и интересной приключенческой игры.

ПРИМЕЧАНИЯ РЕДАКТОРА

[1] (с. 152) Условие выталкивания текста из стека *Y* в последнем комментарии следует изменить на «пока не встретится левая скобка или дно стека или операция с более низким приоритетом». Программа на с. 154 использует именно это, исправленное условие.

[2] (с. 157) Программа, приведенная на с. 158–159, неправильная. Например, на следующий набор исходных данных (приведены только вводимые строки, разделенные точкой с запятой): 5 7 1 4; 1 2 5.0; 1 3 3.0; 1 5 1.0; 3 2 1.0; 5 3 1.0; 2 4 1.0; 3 4 5.0 программа печатает ответ: Путь из 4 в 1 4...2...3...5...1; Требуемое время 5.0. Здесь путь найден правильно, а время – нет (правильный ответ – 4.0). Для исправления программы, возможно, достаточно изменить условие **WHILE** cycles < 2 (третья строка на с. 159) на **WHILE** cycles < nodes с соответствующим изменением типа переменной cycles.

[3] (с. 165) Слово **VAR** целесообразно убрать из описания параметров процедур обхода; это не приведет к копированию больших структур данных, копироваться будет только один элемент типа *pointertype*, т.е. указатель.

13

ДИНАМИЧЕСКИЕ СТРОКИ

ПРОГРАММЫ ОБРАБОТКИ СТРОК

- READSTRING
- WRITESTRING
- MIDDLE
- CONCAT
- COMPARE
- INSTR
- PEEK
- POKE

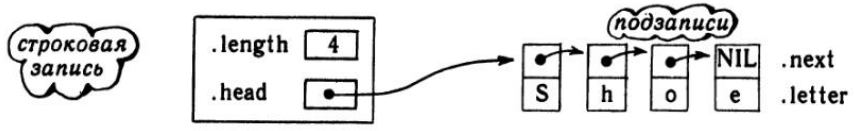
ОБРАТНЫЙ СЛЕНГ (ПРИМЕР)
ТЕХНИКА ХЕШИРОВАНИЯ (ПРИМЕР)
HASHER (ПРИМЕР)

ПРОГРАММЫ ОБРАБОТКИ СТРОК

МОГУТ БЫТЬ ПОЛЕЗНЫ, ДАЖЕ ЕСЛИ В ВАШЕМ ПАСКАЛЕ ЕСТЬ TYPE STRING

В стандартном Паскале предопределено не слишком много процедур по обработке строк, и, как следствие, современные компиляторы предлагают еще и ряд нестандартных процедур. Недостатком использования нестандартных процедур является потеря переносимости: программа, написанная для одного компилятора, не будет работать с другим. Эту проблему можно обойти, определив свой собственный набор программ (утилит), которые строятся только из стандартных частей. Именно такое направление и развивается ниже. Конечной целью является скорее иллюстрация предлагаемой методологии; нежели попытка стандартизации строковых утилит. Читателю наверняка понадобится более мощный и лучшего качества набор процедур, чем предложенный здесь.

Все утилиты используют запись, которая изображена ниже:



Использование динамической памяти снимает ограничения на длину обрабатываемых строк и позволяет строкам иметь различную длину. Ниже представлено определение типов. (Приводится также определение перечисляемого типа, который будет использоваться позже при сравнении строк.)

```
PROGRAM strings(INPUT,OUTPUT);

  TYPE
    stringrange = 0..MAXINT;          { должно быть неотрицательным }
    pointertype = ↑ lettertype;
    lettertype = RECORD
                   next: pointertype;
                   letter: CHAR       { .letter может содержать любую литеру ~ не обязательно букву }
                 END;
    stringtype = RECORD
                   length: stringrange;
                   head: pointertype
                 END;
    relation = (eq, ne, gt, ge, lt, le);   { заменяет отношения (=, <>, >, >=, <, <=) }
```

Первые две процедуры – рекурсивные. Процедура *append* служит для добавления новой литеры в конец строки; процедура *reclaim* предназначена для освобождения подзаписей в том случае, когда в запись помещается новая строка. Эти процедуры «нижнего уровня» используются основными строковыми утилитами. Программисту, который пользуется последними, нет нужды знать о процедурах нижнего уровня.

Во всех процедурах параметры, обозначающие строковые записи, сделаны параметрами-переменными. Смысл этого в том, чтобы процессор не создавал копий строк ~ которые могут быть очень длинными.

```
PROCEDURE append(VAR p: pointertype; c: CHAR);
  BEGIN
    IF p = NIL
      THEN
        BEGIN
          NEW(p);
          p↑.letter:= c;
          p↑.next:= NIL
        END
      ELSE
        append(p↑.next, c)
  END;

PROCEDURE reclaim(VAR p: pointertype);
  BEGIN
    IF p <> NIL
      THEN
        BEGIN
          IF p↑.next<>NIL
            THEN
              reclaim(p↑.next);
          DISPOSE(p);
          p:= NIL
        END
  END;
```

рекурсия

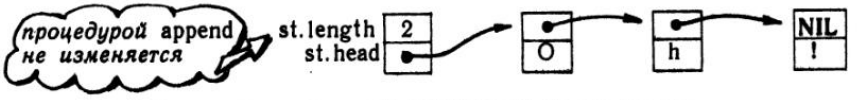
рекурсия

Представим себе строку с именем *st*:

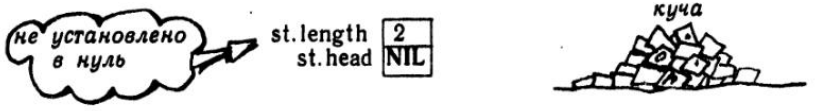
VAR st: stringtype;



В результате работы процедуры *append(st.head,'!')* будем иметь:



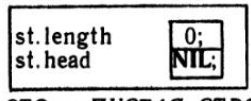
В результате работы процедуры *reclaim(st.head)* будем иметь:



Ниже нарисована *пустая* строка. Любую строку надо обязательно инициализировать, прежде чем ее можно будет использовать в последующих процедурах. Для этого можно написать формальную процедуру, но не стоит тратить усилия – это лишь усложнит дело.

```
st.head := NIL;
st.length := 0;
```

ТАК НАДО ИНИЦИАЛИЗИРОВАТЬ СТРОКУ



ЭТО – ПУСТАЯ СТРОКА

READSTRING (*имя строки*)

ПЕРЕД ВЫЗОВОМ ПРОВЕРЯЙТЕ УСЛОВИЕ EOLN

Эта процедура считывает строку и помещает ее в память под указанным именем. Параметром может быть как пустая, так и непустая строка: предыдущее содержимое строки теряется. Обращение к процедуре с неинициализированной строкой является ошибкой. Предполагается, что строка завершается пробелом или символом EOLN (т.е. нажатием клавиши RETURN). Пробелы перед строкой этой процедурой игнорируются.

```
PROCEDURE readstring( VAR newstring: stringtype);
  CONST
    space = ' ';
  VAR
    ch: CHAR;

  BEGIN
    reclaim(newstring.head);
    newstring.length:= 0;
    REPEAT
      READ(ch);
    UNTIL (ch<>space) OR EOLN;
    IF ch<>space
      THEN
        REPEAT
          append(newstring.head,ch);
          newstring.length:= newstring.length + 1;
          ch:= space;
          IF NOT EOLN THEN READ(ch)
        UNTIL ch=space
  END;
```

если строка newstring *уже пуста, то процедура* reclaim *ничего не делает*

начальные пробелы пропускаются

счетчик числа букв

WRITESTRING (*имя строки*)

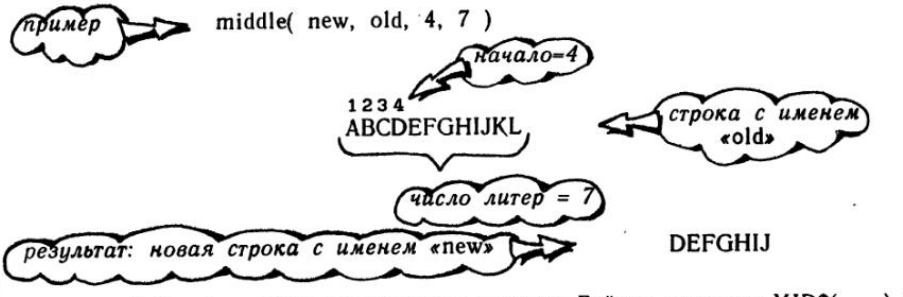
НИЧЕГО НЕ ДЕЛАЕТ С ПУСТОЙ СТРОКОЙ

Приведенная ниже процедура печатает копию указаной строки, не содержащей пробелов в начале и конце строки, а также литер конца строки. Если заданная строка – пустая, то процедура ничего не делает.

```
PROCEDURE writestring(VAR oldstring: stringtype);
  VAR
    p: pointertype;
  BEGIN
    p:= oldstring.head;
    WHILE p<>NIL DO
      BEGIN
        WRITE(p↑.letter);
        p:= p↑.next
      END
  END;
```

MIDDLE (*имя*$_{новойстроки}$, *имя*$_{старойстроки}$, *начало*, *число*$_{литер}$)

Эта процедура создает новую строку, копируя некоторую часть из середины исходной строки. Новая строка делается из «середины» старой строки. Новая строка содержит заданное число литер и начинается с символа, который находится в указаной позиции. Смысл параметров лучше всего объяснить на рисунке:



Процедура *middle* моделирует популярную в языке Бейсик команду MID$(, , ,).

Четвертый параметр может оказаться слишком большим. В этом случае новая строка обрывается на том месте, где кончается старая строка. Процедуру можно использовать для копирования строк целиком. Новая строка может быть записана на место старой.

```
PROCEDURE middle(VAR newstring,oldstring: stringtype;
                             start,span: stringrange);
  VAR
    i: stringrange; p,temp: pointertype;

  BEGIN
    IF (start>0) AND (start<=oldstring.length)
      THEN
        BEGIN
          temp:= NIL;
          p:= oldstring.head;
          i:= 1;
          WHILE i<start DO                  { пробегаем до начала - «start» }
            BEGIN
              p:= p↑.next;
              i:=SUCC(i)
            END;
          i:= 1;                            { отбрасываем хвост, если число литер, «span», слишком велико, }
          WHILE (p<>NIL) AND (i<=span) DO
            BEGIN
              append(temp,p↑.letter);       { формируем результат во временной строке, temp }
              p:= p↑.next;
              i:= i+1
            END;
          newstring.length:= i-1;           { очищаем пространство, затем подставляем указатель на временную строку }
          reclaim(newstring.head);
          newstring.head:= temp;
        END
  END;
```

CONCAT (*имя*$_{новойстроки}$, *имя*$_{левойстроки}$, *имя*$_{правойстроки}$)

Эта процедура создает новую строку, получаемую в результате копирования двух указанных строк одна за другой ~ другими словами, в результате их конкатенации. Указаные для конкатенации левая и правая строки остаются нетронутыми, если только новая строка не будет записана на место одной из них.

```
PROCEDURE concat( VAR newstring,left,right: stringtype);
  VAR
    p,temp: pointertype;
  BEGIN
    temp:= NIL;
    p:= left.head;
    WHILE p<>NIL DO
      BEGIN
        append(temp,p↑.letter);
        p:= p↑.next
      END;

    p:= right.head;
    WHILE p<>NIL DO
      BEGIN
        append(temp,p↑.letter);
        p:= p↑.next
      END;
    newstring.length:= left.length + right.length;
    reclaim(newstring.head);
    newstring.head:= temp;
  END;
```

Следующая функция предназначена для сравнения строк. Критерий сравнения тот же, что используется при составлении алфавитных каталогов. Прописные буквы полагаются «равными» соответствующим строчным буквам. Строки считаются равными, если они одинаковой длины и все литеры в них слева направо попарно равны:

AbCd *считается равной строке* aBCd

Если строки *не равны*, то порядок их следования в каталоге определяется первой с левого конца несовпадающей литерой. Большей считается строка, в которой эта литера имеет большее порядковое значение.

AbCdg *считается больше, чем строка* aBCdefg

первая несовпадающая литера

Если одна строка короче другой, то к более короткой строке мысленно припишите «пустые» (nul) литеры с нулевым порядковым значением. Теперь снова действует правило, приведенное выше:

AbCde *считается больше, чем строка* aBC■

первый несовпадающий символ

воображаемая «пустая» литера

COMPARE (*имя*$_{строки}$, *крит*, *имя*$_{строки}$)

ФУНКЦИЯ ВОЗВРАЩАЕТ ЛОГИЧЕСКОЕ ЗНАЧЕНИЕ: TRUE ИЛИ FALSE

критерий сравнения:
eq, ne, gt, ge, lt, le
=, <>, >, >=, <, <=

примеры: IF compare(responce, eq, affirm) THEN ...
IF compare(left, ge, right) THEN ...

перечислены на с. 170

```
FUNCTION compare(VAR left: stringtype; r: relation;
                 VAR right: stringtype): BOOLEAN;
  VAR
    cp,cq: CHAR;
    same,pmore,qmore: BOOLEAN;
    p,q: pointertype;

  FUNCTION upper( c: CHAR): CHAR;
    BEGIN
      IF c IN ['a'..'z']
        THEN
          upper:= CHR(ORD(c)-ORD('a')+ORD('A'))
        ELSE
          upper:= c
    END;

  BEGIN  { compare }

    p := left.head;      q := right.head;
    pmore := p<>NIL;     qmore := q<>NIL;
    same := TRUE;
    WHILE (pmore AND qmore) AND same DO
      BEGIN
        cp := upper(p↑.letter);
        cq := upper(q↑.letter);
        same := cp=cq;
        p := p↑.next;       pmore := p<>NIL;
        q := q↑.next;       qmore := q<>NIL
      END;

    IF (same AND qmore) AND (NOT pmore)
     THEN cp:= CHR(0);
    IF (same AND pmore) AND (NOT qmore)
     THEN cq:= CHR(0);

    CASE r OF

      eq: compare := cp =  cq;
      ne: compare := cp <> cq;
      gt: compare := cp >  cq;
      ge: compare := cp >= cq;
      lt: compare := cp <  cq;
      le: compare := cp <= cq

    END  { CASE }
  END;  { compare }
```

при сравнении строк любая строчная буква обрабатывается как прописная

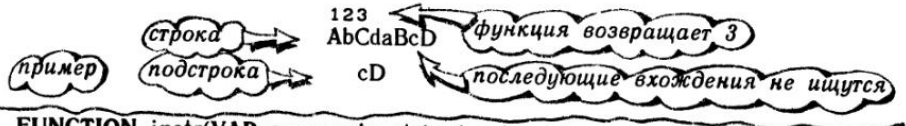
предполагается, что эта разность постоянна:
от $\overset{a}{A}$ *до* $\overset{z}{Z}$

Литера CHR(0) обязана быть меньше, чем любая другая, с которой она сравнивается

INSTR (имя_строки, имя_подстроки)

ФУНКЦИЯ ВОЗВРАЩАЕТ ПОЗИЦИЮ СОВПАДЕНИЯ ~ ИЛИ НУЛЬ, ЕСЛИ СОВПАДЕНИЯ НЕТ

Эта функция моделирует популярную функцию языка Бейсик. Она ищет первое вхождение *подстроки* в *строку* и возвращает номер позиции, в которой начинается подстока. Номера изменяются с 1. Если вхождений нет, то возвращается нуль.




```
FUNCTION instr(VAR super,sub: stringtype): stringrange;
  VAR
    tempstring: stringtype;
    i,j: stringrange;
    match: BOOLEAN;
  BEGIN
    instr := 0;
    tempstring.head := NIL;
    i := 0;
    j := super.length - sub.length + 1;
    IF j >= 1
      THEN
        BEGIN
          REPEAT
            i := SUCC(i);
            middle(tempstring,super,i,sub.length);
            match:= compare(tempstring,eq,sub)
          UNTIL match OR (i=j);
        IF match THEN instr:= i;
        reclaim(tempstring.head);
      END
  END;
```

из строки super, начиная с текущей позиции, извлекаем короткую временную строку

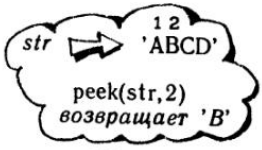
сравниваем временную строку со строкой sub

PEEK (имя_строки, n_позиция)

ФУНКЦИЯ ВОЗВРАЩАЕТ n-Ю ЛИТЕРУ

Эта функция возвращает литеру из *n*-й позиции указаной строки, или CHR(0), если n больше длины строки.

```
FUNCTION peek(VAR old: stringtype; n: stringrange): CHAR;
  VAR
    i: stringrange; p: pointertype;
  BEGIN
    p := old.head;
    i := 1;
    WHILE (i<n) AND (p<>NIL) DO
      BEGIN
        i := SUCC(i);
        p := p↑.next
      END;
    IF p<>NIL
      THEN
        peek := p↑.letter
      ELSE
        peek := CHR(0)
  END;
```



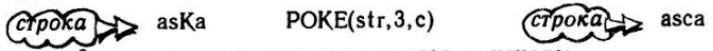
ПРИМЕР

данная позиция больше длины строки

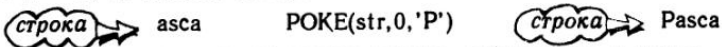
POKE (*имя*$_{строки}$, *n*$_{позиция}$, *c*$_{литера}$) *ЗАМЕНЯЕТ n-Ю ЛИТЕРУ НА c*

Процедура может делать различные действия:

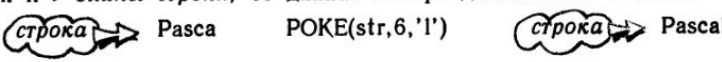
- если 1 ≤ *n* ≤ *длины строки*, то процедура заменяет данной литерой *n*-ю литеру указаной строки:



- если n = 0, то данная литера вставляется в начало:



- если n > *длины строки*, то данная литера добавляется в конец:



Таким способом из пустых строк можно строить строковые «константы». Для длинных строковых констант лучше написать процедуру построения строк из строковых констант Паскаля, присваиваемых упакованным массивам литер.

```
PROCEDURE poke( VAR old: stringtype; n: stringrange; c: CHAR);

  VAR
    p: pointertype;
    i: stringrange;

  BEGIN
    IF n > old.length                          { n > длина; добавляем }
      THEN
        BEGIN
          append(old.head,c);
          old.length:= old.length + 1;
        END
      ELSE IF n = 0                            { n = 0; вставляем в начало }
        THEN
          BEGIN
            NEW(p);
            p↑.next:= old.head;
            p↑.letter:= c;
            old.head:= p;
            old.length:= old.length + 1
          END
        ELSE                                   { 1 ≤ n ≤ длина; заменяем n-ю литеру }
          BEGIN
            p:= old.head;
            i:= 1;
            WHILE (i < n) AND (p <> NIL) DO
              BEGIN
                i:= SUCC(i);
                p:= p↑.next
              END
            IF p <> NIL
              THEN p↑.letter:= c
          END
  END;
```

ОБРАТНЫЙ СЛЕНГ

Isthay isay Ackslangbay!
Ancay ouyay eadray itay?
Erhapspay otnay atay irstfay.

Обратный сленг – это секретный язык, на котором разговаривают в школах. Он совершенно непонятен, когда слышишь его впервые, однако, им легко овладеть, если знаешь грамматические правила. Возможно, существует много диалектов обратного сленга (его еще называют *pig Latin*); этот запомнился еще со школьной скамьи. В каждом английском слове делается циклическая перестановка относительно первой гласной буквы и в конце добавляется *ay* (*tea* → *eatay, tomato* → *omatotay*). Если слово начинается с гласной, то осью вращения становится вторая гласная (*item* → *emitay*). Если второй гласной нет, то ничего не переставляется (*itch* → *itchay*). Две гласные в начале слова воспринимаются как одна гласная (*oil* → *oilay*, а не *iloay; earwig* → *igearway*, а не *arwigeay*).

Прописную букву в начале слова следует преобразовать (*Godfather* → *Odfathergay*, а не *odfatherGay*). Буква *u*, следующая за буквой *q* требует специальной обработки (*Queen* → *Eenquay*, а не *ueenQay*). Знак пунктуации, завершающий слово, так и остается в конце (*Crumbs!* → *Umbscray!* ,а не *Umbs!cray*).

Чтобы все это нормально работало, входной файл нижеследующей программы должен до самого конца вводиться без нажатия клавиши RETURN. Вводите текст строчными буквами, используя, где это нужно, прописные. После каждого знака пунктуации, но не перед ним следует ставить пробел. Кавычки, как двойные так и одинарные, не обрабатываются, и поэтому их следует опустить. Знаки пунктуации внутри слов, такие, как апострофы, трактуются как согласные.

Попробуйте выполнить программу со следующим входным файлом. Программа должна зашифровать текст и получить результат, представленный в самом верху этой страницы:

This is Backslang! Can you read it? Perhaps not at first.

```
PROCEDURE colossus;

  VAR
    puncmark: CHAR;
    recap: BOOLEAN;
    btm: 2..3;
    fold,k,quin: stringrange;
    offset: INTEGER;
    word,fore,aft,qu,ay: stringtype;

  BEGIN
    word.head:= NIL; word.length:= 0;
    fore.head:= NIL; fore.length:= 0;
    aft.head:= NIL;  aft.length:= 0;
    ay.head:= NIL;   ay.length:= 0;
    qu.head:= NIL;   qu.length:= 0;

    offset:= ORD('a') - ORD('A');

    poke(ay,1,'a'); poke(ay,2,'y'); poke(ay,3,' ');
    poke(qu,0,'u'); poke(qu,0,'q');
```

в действительности, цель этого примера – показать использование средств обработки строк, разработанных на предыдущих страницах

инициализация всех строковых переменных

«строковые константы» 'ay' и 'qu'

пробел

```
    WHILE NOT EOLN DO
      BEGIN
        readstring(word);
        recap:=peek(word,1) IN ['A'..'Z'];
        IF recap
          THEN                    { если начальная буква - прописная, делаем ее строчной }
            poke(word,1,CHR(ORD(peek(word,1)) + offset));
        IF NOT (peek(word,word.length) IN ['A'..'Z','a'..'z'])
          THEN
            BEGIN
              puncmark:= (peek(word,word.length);
              IF word.length = 1
                THEN
                  poke(word,0,' ');
              middle(word,word,1,word.length-1)
            END
          ELSE                    { если последняя литера не является буквой, то запоминаем ее как знак пунктуации }
            puncmark:= CHR(0);
        quin:= instr(word,qu);
        IF quin > 0               { если слово содержит 'qu', то заменяем это сочетание на 'q*' }
          THEN
            poke(word,quin+1,'*');
        IF peek(word,1) IN ['A','a','E','e','I','i','O','o','U','u']
          THEN  btm:= 3;          { gear или earwig
          ELSE  btm:= 2;            btm-2    btm-3 }
        fold:= 1;
        FOR k:= word.length DOWNTO btm DO
          IF peek(word,k) IN ['A','a','E','e','I','i','O','o','U','u']
            THEN
              fold:= k;
        IF quin > 0               { восстанавливаем 'u' после 'q' }
          THEN
            poke(word,quin+1,'u');
        middle(fore,word,fold,word.length-fold+1);
        middle(aft,word,1,fold-1);
        concat(word,fore,aft);    { переставляем слово; добавляем 'ay' }
        concat(word,word,ay);
        IF puncmark <> CHR(0)     { если был знак пунктуации, то добавляем его }
          THEN
            BEGIN
              poke(word,word.length,puncmark);
              poke(word,1+word.length,' ')
            END;                  { восстанавливаем, если необходимо, прописную букву }
        IF recap AND (peek(word,1) IN ['a'..'z'])
          THEN
            poke(word,1,CHR(ORD(peek(word,1)) - offset));
        writestring(word)
      END;  { WHILE }
      WRITELN
  END;  { colossus }

BEGIN  { strings }
  colossus                        { главная программа }
END.   { strings }
```

ХЕШИРОВАНИЕ

МЕТОД БЫСТРОГО ПОИСКА

Как вы ищете слово в списке? Простейшее решение – просмотреть список сверху вниз и, найдя совпадение, что-либо предпринять. Здесь приведена несложная программа для поиска буквы 'C' в списке букв. В таком подходе нет ничего плохого, если только список достаточно короткий.

```
FOR i:=1 TO 9 DO
BEGIN
 IF list[i]='C'
  THEN
    WRITELN('C в',i);
 lastposition:= i;
END
```

list[1]	'P'
list[2]	'O'
list[3]	'L'
list[4]	'T'
list[5]	'C'
list[6]	'E'
list[7]	'M'
list[8]	'A'
list[9]	'N'

Трюк в работе с длинными списками состоит в том, чтобы ухитриться сразу попасть в нужное место. В списке латинских букв, имеющем длину 26, техника поиска очевидна. Такой список надо располагать в алфавитном порядке так, чтобы при поиске, например, буквы 'C' надо было бы смотреть элемент list[3]. Для поиска произвольной буквы *x* надо будет проанализировать элемент list[ORD(x) – ORD('A') + 1]. Выражение ORD(x) – ORD('A') + 1 в математической терминологии называется *функцией* от *x*. Эта функция возвращает правильный адрес для произвольной буквы *x*.

Однако в случае списка *слов* обеспечить уникальный адрес для каждого мыслимого слова невозможно. На практике используется следующее решение: устанавливается предельная длина списка и выбирается функция (похожая на ту, что рассматривалась выше), которая дает *вероятный* адрес искомого слова. Такая функция называется *хеш-функцией*.

Хеш-функция напоминает по внешнему виду и поведению функцию, генерирующую случайные числа. Генераторы случайных чисел используют операцию MOD для того, чтобы привести случайное число в определенный диапазон. Подобно этому, в хеш-функции операция MOD используется для того, чтобы адрес находился в пределах длины списка. Приведенная ниже хеш-функция получена из функции, предложенной Керниганом и Плоджером*.

Возьмем слово ANT, которому надо найти место в списке из 17 компонент – с 0 по 16. Хеш-функция использует порядковые значения букв. В примере использован код ASCII, хотя метод работает и с другими кодами.

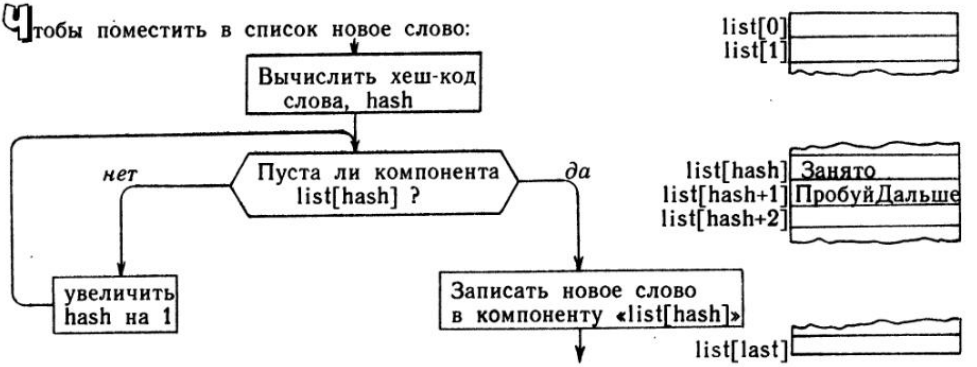


*см. литературу

В соответствии с этим алгоритмом, слово AARDVARK породит код 7 и будет, следовательно, отнесено к элементу list[7]. Из сравнения этих двух адресов ясно, что хеш-коды не располагают слова в алфавитном порядке. Хеширование слова STOAT дает код 2, пересекающийся с кодом слова ANT. Подобные коллизии разрешаются при помощи метода, рассматриваемого на противоположной странице.

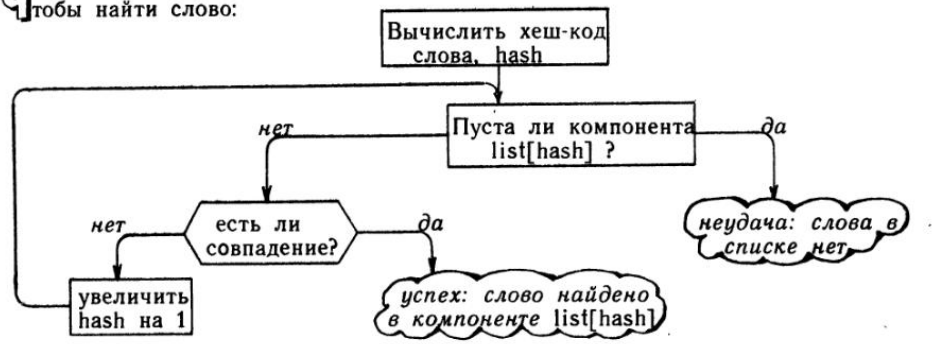
Число 3 – «магическое число»; можно также попробовать 5, 7 или другое небольшое простое число. Длина списка (17 в этом примере) для большей эффективности также должна быть простым числом. Под «большей эффективностью» понимается более равномерное распределение хеш-кодов по списку, с тем чтобы не было скученности. На с. 182 приведена программа, демонстрирующая рассмотренную хеш-функцию. Испытайте ее и посмотрите, не скучиваются ли коды (у меня – не скучивались).

Чтобы поместить в список новое слово:



Список должен быть круговым, с тем чтобы при достижении переменной hash последнего значения увеличение ее на 1 возвращало бы hash в нуль. Вместе с этим должен быть предусмотрен механизм, предотвращающий зацикливание при поиске в случае заполненного списка.

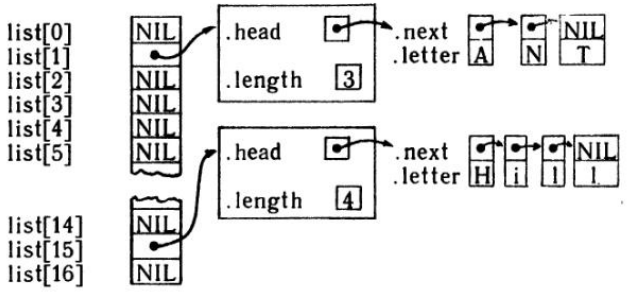
Чтобы найти слово:



Программа на следующей странице разработана для демонстрации метода хеширования. При работе с ней просто вводите слова. Каждое новое слово помещается в список, после чего список полностью выводится; при этом видно, куда помещено слово. Если слово оказывается «старым», то сообщается его местонахождение. Изначально программа предполагает, что данное слово – «старое» и начинает его поиск. Если поиск оказывается безуспешным, то программа запоминает данное слово как «новое».

Программа использует строковые утилиты, которые рассмотрены ранее ~ поэтому прописные буквы рассматриваются как равные соответствующим строчным: ANT ≡ Ant.

Используется структура данных в виде массива указателей, указывающих на записи типа *stringtype*. Массив указателей должен быть объявлен и инициализирован. Остальные данные формируются динамически.



HASHER

ПРИМЕР ПРОГРАММЫ ДЛЯ ДЕМОНСТРАЦИИ ХЕШИРОВАНИЯ

Приведенная здесь программа основана на принципах, разобранных на предыдущем развороте. Для использования программы просто вводите слова и наблюдайте за экраном, чтобы увидеть, куда они записываются. Введите какие-нибудь слова, которые уже были введены, и обратите внимание, что дубли не записываются: вместо этого сообщается их местонахождение.

```
PROGRAM hasher(INPUT,OUTPUT);
```

Здесь вставьте объявления и утилиты обработки строк, которые приведены на с. 170–177 (таким образом, процедура colossus *и основная программа со с. 178–179 не нужны). Процедура хеширования не обращается к процедурам* middle, concat, instr *и* poke, *которые также могут быть опущены.*

```
PROCEDURE hashplay;
  CONST
    size = 17;   siz = 16;
  TYPE
    sizerange = 0..siz;
    nametype = ↑ stringtype;
    arraytype = ARRAY[sizerange] OF nametype;
  VAR
    name: stringtype;
    i,hash,recall: sizerange;
    full,found,ahole: BOOLEAN;
    list: arraytype;
    n: INTEGER;

  PROCEDURE show;
    VAR i: sizerange;
    BEGIN
      FOR i:=0 TO siz DO
        IF list[i]<>NIL
          THEN
            BEGIN
              WRITE(i,'  ');
              writestring(list[i]↑);
              WRITELN
            END
          ELSE
            WRITELN(i,' *')
  END;

BEGIN  { hashplay }

  name.head:= NIL;
  full:= FALSE;
  FOR i:=0 TO siz DO
    list[i]:= NIL;
```

на единицу меньше size

вывод на экран

2 пробела

1 пробел

```
Hill
Записано в 15
 0 *
 1 *
 2  ANT
 3 *
 4 *
 5 *
 6 *
 7 *
 8 *
 9 *
10 *
11 *
12 *
13 *
14 *
15  Hill
16 *
```

```
      REPEAT
        readstring(name);
        hash:=0;
        FOR i:=1 TO name.length DO
          BEGIN
            n:= ORD(peek(name,i));
            IF n IN [ORD('a')..ORD('z')]
              THEN
                n:= n - ORD('a') + ORD('A');
            hash:= (3*hash + n) MOD size;
          END;
        ahole:= list[hash]=NIL;
        IF NOT ahole
          THEN
            BEGIN
              recall:= hash;
              REPEAT
                found:= compare(name,eq,list[hash]↑);
                IF found
                  THEN
                    WRITELN('Найдено в',hash:4);
                  ELSE
                    BEGIN
                      hash:= (1 + hash) MOD size;
                      ahole:= list[hash]=NIL;
                      full:= hash=recall
                    END
              UNTIL (ahole OR found) OR full
            END;

        IF ahole
          THEN
            BEGIN
              NEW(list[hash]);
              list[hash]↑:= name;
              WRITELN('Записано в',hash:3);
              WRITELN;
              show;
              name.head:= NIL
            END
          ELSE IF full
                 THEN
                   BEGIN
                     WRITELN('List full');
                     show;
                   END
      UNTIL full

    END;  { hashplay }

BEGIN  { main program }

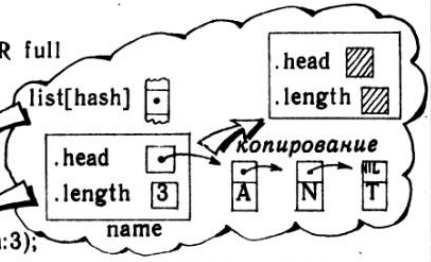
  hashplay;

END.   { main program }
```

с целью сравнения строчные буквы заменяем на прописные

хеш-функция

увеличиваем hash на 1



Важный момент! Если этого не написать, то процедура readstring *сотрет посредством* DISPOSE *слово, на которое указывает* name[1]

---

[1] Оператор name.head:= **NIL** по существу стирает слово, только что записанное в список, и, таким образом, не решает проблему, отмеченную в комментарии к этому оператору. Чтобы обеспечить сохранение слова, следует перед присваиванием **NIL** скопировать хотя бы одно первое звено цепи (так, как это показано на предыдущей картинке, но не так, как записано в программе). – *Прим. ред.*

ЛИТЕРАТУРА

BSI Specification for *Computer programming language Pascal* BS6192: 1982

Британский стандарт, который определяет тот же диалект Паскаля, что представлен в моей книге. Стандарт BS6192 не предназначен для чтения лежа в постели, однако если вам понадобится выяснить точный синтаксис или поведение Паскаль-процессора в каких-либо редких ситуациях, то стандарт BS6192 – как раз то, что нужно. В его предисловии предпринимаются попытки объяснить сложную взаимосвязь между BS6192 и ISO 7185, но мне пока не удалось расшифровать этот текст. По-видимому, предполагалось, что эти стандарты будут определять одно и то же, но фактически это не совсем так.

Jensen, K. & Wirth, N. (1975). *Pascal user manual and report.* (Springer-Verlag). (Русский перевод: К. Йенсен, Н. Вирт. Паскаль. Руководство для пользователя и описание языка. – М.: Финансы и статистика, 1982.).

Это первая книга по Паскалю; один из ее авторов, Никлаус Вирт, является создателем языка. Руководство для пользователя, написанное Кэтлином Йенсеном, являясь примером простоты и точности, вошло в историю.

Grogono, Peter (1980). *Programming in PASCAL* (Addison-Wesley). (Русский перевод: П. Грогоно. Программирование на языке Паскаль. – М.: Мир, 1982.)

Это – классика. Впервые книга была опубликована в 1978 г. путем воспроизведения компьютерной распечатки.Теперь же она и отлично отпечатана. Это до сих пор лучшая из всех книг, содержащих полный курс программирования на Паскале, которую мне доводилось видеть. Книгу отличает ясное изложение и образные примеры. Чтобы взять все из этой книги, вам потребуется много работать и смело погружаться в длинные примеры. Для читателей, которые захотят пойти еще дальше, Грогоно приводит обширную и представительную библиографию.

Brown, P.J. (1982) *Pascal from BASIC* (Addison-Wesley)

Хороший самоучитель, простой для понимания, в котором, увы, не удалось избежать ряда недоразумений. Книгу населяют странные персонажи – проф. Примпл (архаический академик) и Билл Мадд (неопытный, но полный энтузиазма), призванные продемонстрировать различные точки зрения на программирование. Вместе с тем нас учат простить им их предвзятую позицию. Структуры данных и динамическая память рассматриваются кратко. Эта книга должна помочь бывшим приверженцам Бейсика, сохранившим старые привычки, перестроиться на Паскаль.

Kernighan, B.W. & Plauger, P.J. (1981) *Software tools in Pascal* (Addison-Wesley)

Книга полна практичных и проверенных программ на Паскале. Фразу «Картинка стоит тысячи слов» вы найдете под одной из двух картинок во всей книге; остальное – 95 тысяч слов текста. Проза, которую я читал с чувством скуки, наградит вас за упорство обилием и еще раз обилием информации.

КРАТКАЯ СПРАВКА

ОБЗОР СТАНДАРТНЫХ ПРОЦЕДУР, ФУНКЦИЙ И СИНТАКСИСА

Стандартные процедуры и стандартные функции перечислены в алфавитном порядке. Справа приводятся номера страниц, на которых рассматривается соответствующая процедура или функция. Обзор синтаксиса построен по принципу сверху вниз.

СТАНДАРТНЫЕ ПРОЦЕДУРЫ

ЕСЛИ ИМЯ ФАЙЛА НЕ УКАЗАНО, ТО ПОДРАЗУМЕВАЕТСЯ INPUT *ИЛИ* OUTPUT

DISPOSE(*имя указателя*) • возврат ненужной записи в кучу 149

GET(*имя файла*) • продвижение окна указанного входного файла 135

NEW(*имя указателя*) • создание новой пустой записи 149

PACK(*имя массива1*, *индекс массива1*, *имя массива2*) • упаковка содержимого одного массива и запись в другой 98

PAGE(*имя файла*) • запись в указанный выходной файл литеры конца страницы (если таковая распознается при печати) 126

PUT(*имя файла*) • продвижение окна указанного выходного файла 135

READ(*имя файла*, *переменная*) • чтение из указаного файла; элементы файла типа TEXT разделяются *пробелами* или *признаками новой строки* 127

READLN [(*имя файла*, *переменная*) | (*имя файла*)] • то же, что READ, но только для файлов типа TEXT: когда прочитан последний параметр, переход на следующую строку ввода 127

RESET(*имя файла*) • подготовка указанного файла для чтения 124

(*никогда не делайте* reset *для* INPUT *и* rewrite *для* OUTPUT)

REWRITE(*имя файла*) • подготовка указанного файла для записи 124

UNPACK(*имя массива2*, *имя массива1*, *индекс массива1*) • обратная для PACK 98

WRITE(*имя файла*, *выражение* : *ширина* : *точность*) 126

WRITELN [(*имя файла*, *выражение* : *ширина* : *точность*) | (*имя файла*)] 126

• Параметры *ширина* и *точность* – выражения целого типа. Параметр *точность* используется только для печати *выражений* типа REAL.

СТАНДАРТНЫЕ ФУНКЦИИ

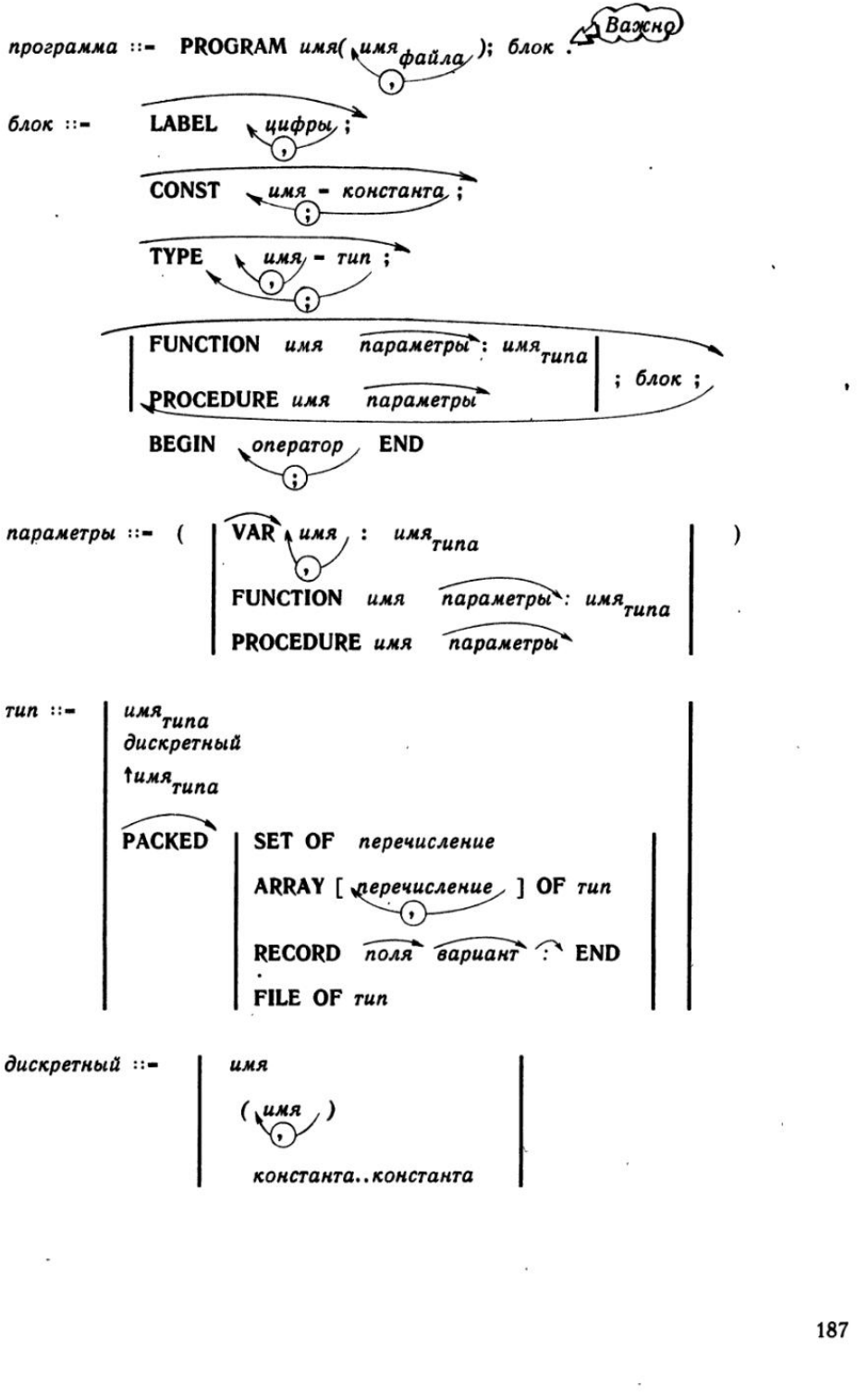
В ВАШЕМ ПАСКАЛЕ, ВЕРОЯТНО, БОЛЬШЕ ФУНКЦИЙ, ЧЕМ ЗДЕСЬ ОПИСАНО

Буква *i* означает выражение, которое приводится к целому значению; буква *r* – выражение, которое приводится к вещественному значению; буква *m* означает параметр, имеющий порядковое значение: например, целое, литеру или элемент перечисляемого типа.

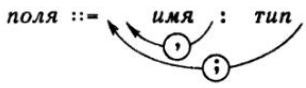
ABS(*i*)	• *абсолютная величина*: ABS(-6) возвращает 6 (целое)	46
ABS(*r*)	• *абсолютная величина*: ABS(-6.5) возвращает 6.5 (веществ.)	46
ARCTAN(*r*)	• *арктангенс*: ARCTAN(1.0) возвращает 0.785398 (π/4)	47
CHR(*i*)	• *литера*: в кодировке ASCII CHR(65) возвращает 'A'	51
COS(*r*)	• *косинус*: COS(3.141593/3) возвращает 0.5	47
EOF (*имя$_{файла}$*)	• возвращает TRUE, если окно – в конце файла (процедура READ не сработает еще раз)	49
EOLN (*имя$_{файла}$*)	• возвращает TRUE, если последующий вызов процедуры READ прочитает пробел, означающий конец строки	49
EXP(*r*)	• *экспонента* или *натуральный антилогарифм*: EXP(1) возвращает 2.7182818 (т.е. e^1)	46
LN(*r*)	• *натуральный логарифм*: LN(2.7182818) возвращает 1 (т.е. *ln*(e))	46
ODD(*i*)	• *нечетность*: ODD(-3) возвращает TRUE, ODD(0) – FALSE	49
ORD(*m*)	• *порядковое значение*: в коде ASCII ORD('A') возвращает число 65, ORD(TRUE) – 1, ORD(FALSE) – 0	50
PRED(*m*)	• *предшественник*: PRED('B') возвращает 'A', PRED(6) возвращает 5, PRED(TRUE) возвращает FALSE	51
ROUND(*r*)	• *округление до ближайшего целого*: ROUND(3.5) возвращает число 4, ROUND(-3.8) – число -4	48
SIN(*r*)	• *синус*: SIN(3.141593/6) возвращает 0.5	47
SQR(*i*)	• *квадрат*: SQR(-3) возвращает 9 (целое)	46
SQR(*r*)	• *квадрат*: SQR(-3.0) возвращает 9.0 (вещественное)	46
SQRT(*r*)	• *квадратный корень*: SQRT(81) возвращает 9.0 (вещественное)	46
SUCC(*m*)	• *следующий*: SUCC('A') возвращает 'B', SUCC(5) возвращает число 6, SUCC(FALSE) возвращает TRUE	51
TRUNC(*r*)	• *отбрасывает дробную часть*: TRUNC(-3.8) возвращает число -3 (целое)	48

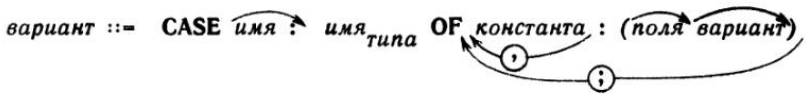
СИНТАКСИС

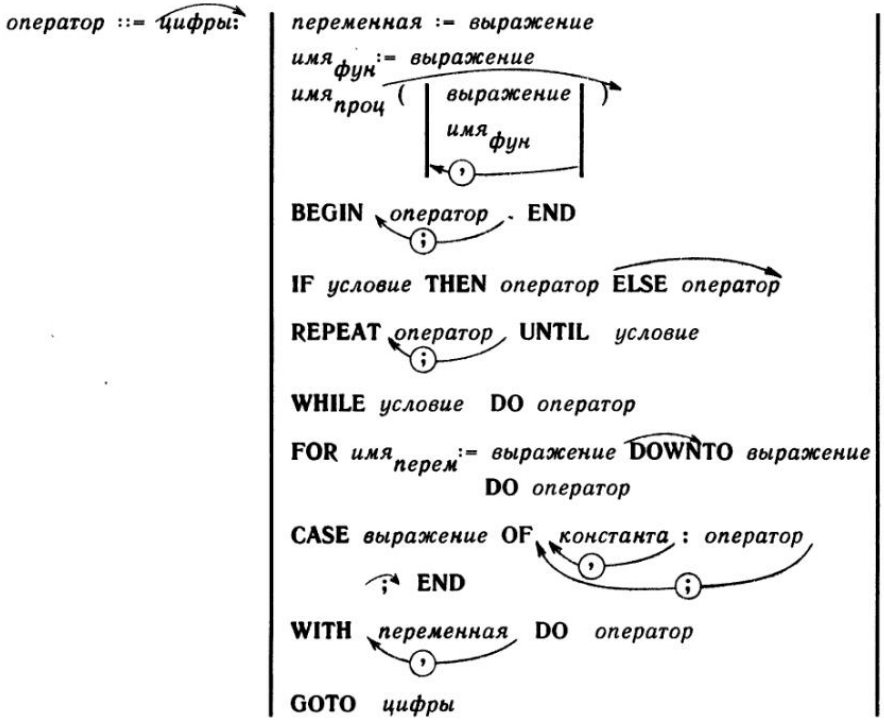
ОБЗОР СВЕРХУ ВНИЗ. СИСТЕМА ОБОЗНАЧЕНИЙ ПРИВЕДЕНА В ГЛАВЕ 3



СИНТАКСИС (ПРОДОЛЖЕНИЕ ОБЗОРА)

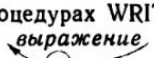


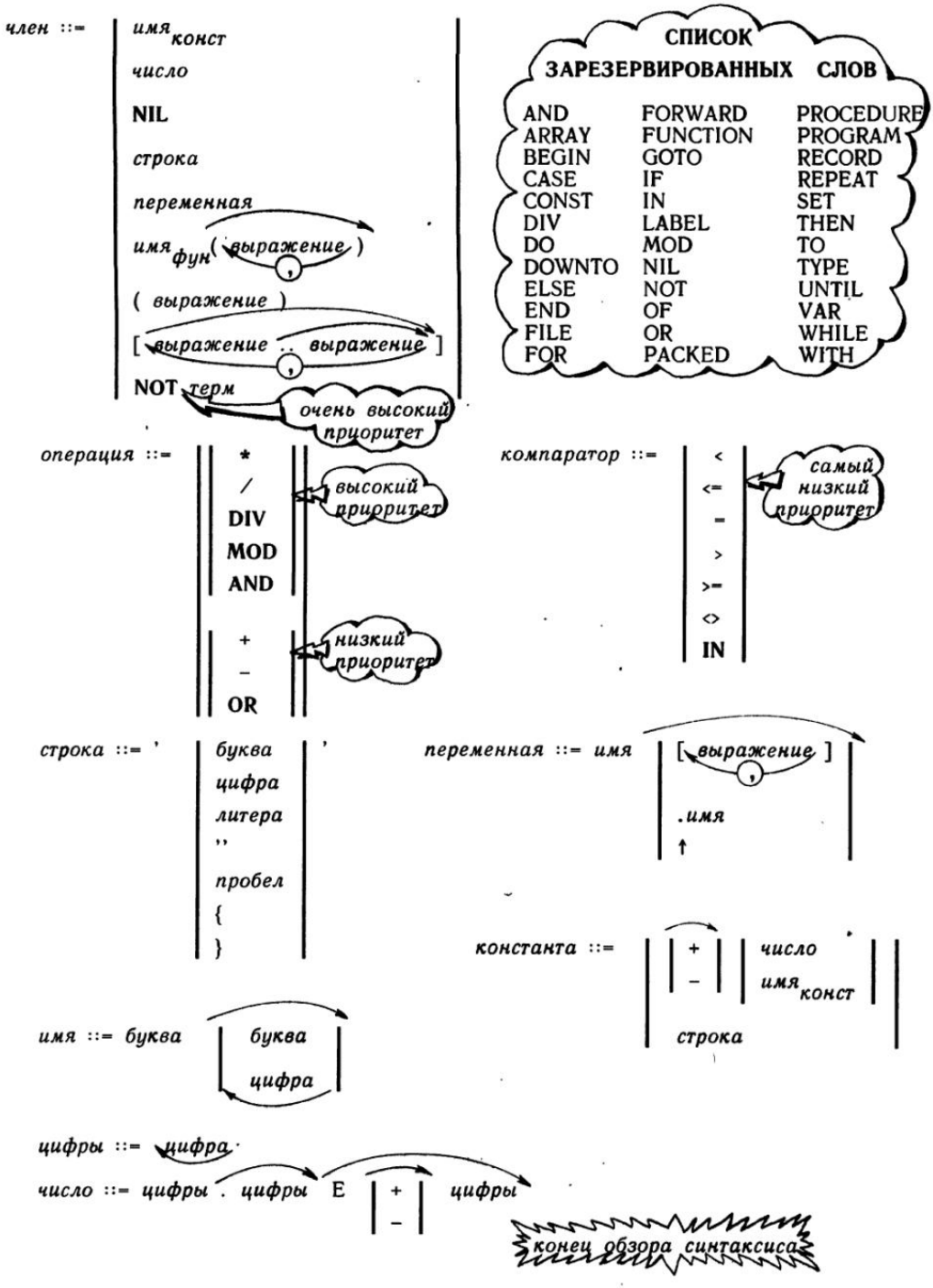








в процедурах WRITE и WRITELN *выражения* могут записываться как 


член ::=
имя конст
число
NIL
строка
переменная
имя фун (выражение ,)
(выражение)
[выражение .. выражение ,]
NOT терм
очень высокий приоритет
СПИСОК ЗАРЕЗЕРВИРОВАННЫХ СЛОВ
AND FORWARD PROCEDURE
ARRAY FUNCTION PROGRAM
BEGIN GOTO RECORD
CASE IF REPEAT
CONST IN SET
DIV LABEL THEN
DO MOD TO
DOWNTO NIL TYPE
ELSE NOT UNTIL
END OF VAR
FILE OR WHILE
FOR PACKED WITH
операция ::=
* / DIV MOD AND
высокий приоритет
+ - OR
низкий приоритет
компаратор ::=
< <= = > >= <> IN
самый низкий приоритет
строка ::= ' буква цифра литера '' пробел { } '
переменная ::= имя [выражение ,] .имя ↑
константа ::= + - число имя конст строка
имя ::= буква буква цифра
цифры ::= цифра
число ::= цифры . цифры E + - цифры
конец обзора синтаксиса

УКАЗАТЕЛЬ

КРАТКУЮ СПРАВКУ О СИНТАКСИСЕ, СТАНДАРТНЫХ ПРОЦЕДУРАХ И ФУНКЦИЯХ СМ. НА С. 185–189

А

аргументы 46
астра (пример) 162, 163

Б

базовый тип 84, 90
безопасное чтение (пример) 128–133
блок-схемы 27, 54, 55
буквы 34
буферизация 142
былая слава (пример) 29
быстрая сортировка (пример) 96, 97

В

возврат 148, 171, 183
выражения 24
 – логические 43

Г

геометрические фигуры (пример) 27

Д

двоичные деревья 164–167
динамическая память 146, 147

З

заголовок программы 20, 21, 38
заем (пример) 25
записи 110–119
зарезервированные слова 21, 32
 – –, список 189

И

имена
 – полей 110, 111
 –, синтаксис 35
 –, эквивалентность 91, 98
индексы 90
интерактивный режим 140–143
интервальный тип 83

К

клавиатура 15
код ASCII 50, 99, 125
кольца связанные 160–163
команды 16, 17
компаратор 34, 45
 – IN 45
компараторы для множеств 85
компиляция 14–16
компоненты записей 110
 – массивов 90
конец строки 123, 140, 141
 – файла 143
конкатенация 174
константа NIL 147
константы 13, 22, 23
 – строковые 99
константы-указатели 147
кратчайший путь (пример) 156–159

Л

литерный тип (CHAR) 23
литеры 23
логические
 – выражения 24, 26, 43, 45
 – значения 26
логический тип (BOOLEAN) 23

М

маляр (пример) 12–14
массив
 – литер 99
 – упакованный 98
массивы-параметры
 – настраиваемые 106, 107
многоугольник (пример) 92
множества 84, 85
м-у-у-у (пример) 87

О

обозначения 33
обратная польская нотация 152–154
обратный сленг (пример) 178–179
объявление
 – констант 20–23, 38, 80
 – меток 37, 55
окно файла 123, 134, 141
оператор
 – AND 42
 – DIV 42
 – MOD 42
 – NOT 36
 – OR 42
операторы 20, 37
операции 24, 34, 42, 43
 – отношения 34, 45
определение процедуры (PROCEDURE) 69
 – функции (FUNCTION) 64, 65
отложенный ввод 140, 143
отступы 21
очереди 150, 151

П

пакетный режим 16, 140
параметры
 –, синтаксис 38
 –, тип 80
 – фактические 64, 65
 – формальные 64, 65
параметры-переменные 68, 69
параметры-значения 68, 69
параметры-функции 73
пересечение множеств 85
персональные записи (пример) 112–115
площадь бака (пример) 12–14
площадь многоугольника 92
побочные эффекты 76
поля вывода 26, 29, 126
порядковые значения 23, 39, 40, 80, 92, 99, 126
правила видимости 77
приоритет 24, 34
присваивание 24
 – целиком 91, 111
проблема буферизации 142
 – конца файла 143
провода (пример) 93
программа
 – исходная 16
 – объектная 16
 –, расположение 21, 74
 –, синтаксис 38
процедура
 – DISPOSE 149
 – GET 135
 – GRAB (пример) 130–133
 – NEW 149
 – PACK 98
 – PAGE 126
 – PUT 135
 – READ 127
 – READLN 127
 – RESET 124
 – REWRITE 124
процедуры, сводка 185
пунктуация 14, 20, 21

Р

размер чисел 44
редактор текстовый 14, 15, 125
рекурсия 67, 75, 94–97, 104, 150, 165, 170, 171

С

связанные структуры 148, 160, 161
сжатие (пример) 136
символ 34
синтаксис
 – выражений 36
 –, обозначения 33
 – оператора 37
 –, определение 34
 – программы 38
 –, сводка 187–189
 – составных операторов 35
 – типа 39
 –, элементы 34
синус (пример) 29
системы счисления(пример) 102–104
случайные числа (пример) 70–71
снова заем (пример) 72
сортировка
 – быстрая 96–97
 – двоичным деревом 164–167
 – методом пузырька (пример) 94, 95
 – обезьянья (пример) 166, 167
 – связанного кольца 162, 163
ссылки вперед 74, 147
стеки 150, 151
строки 99
 –, сравнение 99, 174
 –, утилиты 170–177
 –, хеширование 180–183
структура
 – BEGIN..END 20
 – CASE..OF с вариантами 119
 – CASE..OF управляющая 55
 – FOR..TO..DO 57
 – GOTO 37, 55
 – IF-THEN-ELSE 56
 – REPEAT..UNTIL 58
 – WHILE..DO 58

Т

таблица состояний 61, 129
текст программы 16
тип
 – записей 111
 – констант 80
 – INTEGER 12, 23
 – REAL 12, 23
точка с запятой 20
точность 44

У

указатели 146, 147
умножение матриц (пример) 105
упакованный тип 91, 111, 135
упаковки процедуры 98
условия 24, 26, 36, 56

Ф

файл
 – стандартный INPUT 122–125
 – – OUTPUT 122–125
файлы 122–137
 – временные 131
 – двоичные 134, 135
 –, открытие 124

–, свойства 137
– стандартные 122–127
– текстовые 123, 125–127
фильтр (пример) 59
фильтр2 (пример) 86
фокус (пример) 100–101
функции
– арифметические 46
– логические 49
– над дискретными типами 50, 51
–, определение 64, 65
– преобразования 48
–, примеры 66
–, сводка 186
– тригонометрические 47
функция
– ABS 46
– ARCTAN 47
– CHR 51
– COS 47
– EOF 49
– EOLN 49
– EXP 46
– LN 46
– ODD 49
– ORD 50
– PRED 51
– ROUND 48
– SIN 47
– SQR 46
– SQRT 46
– SUCC 51

Х

хеширование 180
хеширование (пример) 182

Ц

цепи 146, 148, 155–157
циклы 28, 54, 57, 58
цифры 34

Я

ЯДЖЭЛОП (пример) 154

4 р. 60 к.

ISBN 5-03-001292-3 (русск.)
ISBN 0-521-33695-3 (англ.)