



*Томский межвузовский центр  
дистанционного образования*

**В.М. Зюзьков**

# **ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ**

**Учебное пособие**

**ТОМСК – 2003**

Министерство образования Российской Федерации

**ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ  
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)**

**Кафедра компьютерных систем в управлении  
и проектировании (КСУП)**

**В.М. Зюзьков**

# **ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ**

**Учебное пособие**

**2003**

Корректор: Красовская Е.Н.

**Зюзьков В.М.**

Программирование на языке высокого уровня: Учебное пособие. – Томск: Томский межвузовский центр дистанционного образования, 2003. – 217 с.

Пособие содержит курс лекций по дисциплине «Программирование на языке высокого уровня» для специальности 22.03.00 «Системы автоматизированного проектирования».

Этот же материал может использоваться при изучении дисциплины «Программирование и основы алгоритмизации» для специальности 21.01.00 «Управление и информатика в технических системах».

Содержание лекций направлено на обучение алгоритмическому мышлению и программированию на языке Паскаль. Предназначено для студентов, обучающихся при помощи всех форм обучения с использованием дистанционных образовательных технологий.

© Зюзьков В. М., 2003  
© Томский межвузовский центр  
дистанционного образования, 2003

## СОДЕРЖАНИЕ

1 Введение в информатику.....	8
1.1 Информация и ее представление.....	8
1.2 Понятие алгоритма .....	9
1.3 Вычислительные структуры .....	13
1.4 Алгоритмические языки.....	16
1.5 Способы описания синтаксиса алгоритмических языков....	17
1.6 Семантика программы: функциональная, операционная, аксиоматическая.....	21
1.7 Трансляция и выполнение.....	22
1.8 Компьютеры фон Неймана .....	24
2 Введение в язык Паскаль .....	27
2.1 Историческая справка.....	27
2.2 Алфавит языка Паскаль.....	27
2.3 Переменные .....	28
2.4 Основные понятия языка Паскаль.....	28
2.5 Правила записи текста программы .....	30
2.6 Система типов языка .....	31
2.7 Основные вычислительные структуры в Паскале .....	33
2.8 Выражения .....	38
2.9 Оператор присваивания.....	40
2.10 Ввод-вывод .....	41
2.11 Последовательное выполнение и составной оператор.....	43
2.12 Условный оператор.....	43
2.13 Оператор цикла с предусловием .....	45
2.14 Оператор цикла с постусловием.....	46
2.15 Оператор цикла с параметром .....	46
2.16 Примеры бесконечных циклов .....	48
2.17 Пустой оператор.....	48
2.18 Ограниченные типы.....	49
2.19 Функции .....	50
2.20 Примеры программ для задач без массивов.....	53
2.21 Более подробно о символьном типе.....	59
3 Технология программирования .....	64
3.1 Машинно-ориентированный оператор: оператор перехода ...	64

3.2	Диаграммы управления потоком.....	65
3.3	Структурное программирование .....	66
3.4	Решение задачи .....	68
3.5	Разработка программы .....	71
3.6	Стиль программирования.....	83
3.7	Доказательное программирование .....	89
4	Регулярные типы (массивы).....	90
4.1	Определение регулярного типа .....	90
4.2	Примеры программ для работы с массивами.....	93
4.3	Символьные массивы .....	96
5	Подпрограммы .....	97
5.1	Понятие подпрограммы.....	97
5.2	Общая структура подпрограмм .....	98
5.3	Тело подпрограммы. Области действия имен.....	100
5.4	Использование процедур и функций .....	101
5.5	Механизм параметров .....	104
5.6	Предварительное описание .....	107
5.7	Побочный эффект .....	108
5.8	Распределение памяти для переменных .....	109
6	Строковый тип.....	110
6.1	Определение строкового типа .....	110
6.2	Строковые операции.....	111
6.3	Стандартные процедуры и функции для работы со строковыми типами.....	113
7	Тестирование и отладка.....	116
7.1	Тестирование .....	116
7.2	Отладка.....	118
8	Сортировка.....	120
8.1	Постановка задачи .....	120
8.2	Сортировка простыми включениями .....	120
8.3	Сортировка простым выбором .....	121
8.4	Сортировка простым обменом («метод пузырька»).....	123
8.5	Сортировка слиянием .....	124
9	Перечислимый тип.....	126
9.1	Определение перечислимого типа .....	126
9.2	Оператор варианта .....	127

10 Множественный тип .....	130
10.1 Определение множественного типа .....	130
10.2 Операции .....	132
11 Файловые типы и ввод-вывод .....	135
11.1 Файловые переменные и типы .....	135
11.2 Установочные и завершающие операции над файлами ..	136
11.3 Операции ввода-вывода .....	137
11.4 Текстовые файлы .....	138
11.5 Примеры работы с файлами .....	140
12 Рекурсия .....	142
12.1 Понятие рекурсии .....	142
12.2 Создание рекурсивных подпрограмм .....	147
12.3 Хранение значений переменных в теле рекурсивной подпрограммы .....	152
12.4 Рекурсия и итерация. Метод накапливающего параметра .....	152
12.5 Рекурсия в своем блеске и великолепии .....	154
13 Записи .....	162
13.1 Определение комбинированных типов .....	162
13.2 Записи с вариантами .....	164
13.3 Оператор над записями with (оператор присоединения) ..	166
14 Динамические структуры данных .....	169
14.1 Ссылочный тип .....	169
14.2 Статические и динамические переменные .....	171
14.3 Линейные списки .....	173
14.4 Проблема потерянных ссылок .....	178
14.5 Циклические списки .....	180
14.6 Списки с двумя связями .....	181
14.7 Стеки и очереди .....	182
14.8 Определение деревьев .....	185
14.9 Обход дерева .....	186
14.10 Поиск по дереву с включением .....	188
14.11 Сортировка деревом .....	190
15 Модули .....	191
15.1 Модульное программирование .....	191
15.2 Стандартные модули .....	195

16 Графическое программирование на языке Паскаль .....	198
16.1 Аппаратная и программная поддержка графики .....	198
16.2 Инициализация графики.....	201
16.3 Базовые процедуры и функции.....	205
16.4 Работа с текстом.....	209
16.5 Построение графических фигур .....	211
16.6 Простые анимационные алгоритмы.....	217
Рекомендуемая литература .....	218

Для того чтобы что-то понять — надо это уже знать...  
 Те виды информации, для которых у нас нет готовых  
 когнитивных схем, мы просто не воспринимаем...  
 В пустую голову ничего поместить нельзя.

М. Холодная. Психология интеллекта:  
 парадоксы исследования

Знания достигаются не быстрым бегом, а медленной  
 ходьбой.

Томас Бабингтон Маколей, 1800–1859

## 1 ВВЕДЕНИЕ В ИНФОРМАТИКУ

### 1.1 Информация и ее представление

**Информатика** — это наука и техника, связанные с машинной обработкой, хранением и передачей информации. Цель состоит в разработке способов решения задач информационной обработки на вычислительных машинах (компьютерах), а также в разработке, организации и эксплуатации вычислительных систем.

В информатике **информация** понимается как абстрактное значение выражений, графических изображений, указаний (операторов) и высказываний.

Мы различаем в связи с информацией:

- ее представление или изображение (внешняя форма);
- ее значение (собственно «абстрактная» информация);
- ее отношение к реальному миру (связь абстрактной информации с действительностью).

#### **Информация и представление**

*Информацией* называют абстрактное содержание («содержательное значение», «семантика») какого-либо высказывания, описания, указания, сообщения или известия. Внешнюю форму изображения называют *представлением* (конкретная форма сообщения).

Виды представлений:

- условные знаки («сигналы»),



- произносимые слова («акустическое представление»),
- рисунки (графическое представление, «пиктограммы», «иконки»),
- последовательность символов («слова», «текст»), и т.д.

### **Интерпретация**

Переход (часто только воображаемый, мыслимый) от представления к абстрактной информации, т.е. к значению представления, называют *интерпретацией*.

Только в том случае, когда существуют единые, согласованные интерпретации, системы представления могут использоваться для обмена информацией.

## **1.2 Понятие алгоритма**

Под *алгоритмом* понимается способ преобразования представления информации. Слово *algorithm* произошло от имени аль-Хорезми – автора известного арабского учебника по математике.

*Алгоритм* – свод конечного числа правил, задающих последовательность выполнения операций при решении той или иной специфической задачи.

Алгоритмы типичным образом решают не только частные задачи, но и классы задач. Подлежащие решению частные задачи, выделяемые по мере надобности из рассматриваемого класса, определяются с помощью параметров. Параметры играют роль исходных данных для алгоритма.

### **Пять важных особенностей алгоритма**

- **Конечность (финитность).** Алгоритм должен всегда заканчиваться после конечного числа шагов.
- **Определенность.** Каждый шаг алгоритма должен быть точно определен.
- **Ввод.** Алгоритм имеет некоторое (быть может, равное нулю) число входных данных, т.е. величин, заданных ему до начала работы.

- **Вывод.** Алгоритм имеет одну или несколько выходных величин, т.е. величин, имеющих вполне определенное отношение ко входным данным.
- **Эффективность.** Это означает, что все операции, которые необходимо произвести в алгоритме, должны быть достаточно простыми, чтобы их можно было в принципе выполнить точно и за конечный отрезок времени с помощью карандаша и бумаги.

Алгоритм должен быть практичным и хорошим с эстетической точки зрения.

Для алгоритмов важно различать следующие аспекты:

- постановку задачи, которая должна быть разрешена с помощью алгоритма;
- специфичный способ, каким решается задача, при этом для алгоритма различают:
  - а) элементарные шаги обработки, которые имеются в распоряжении;
  - б) описание выбора отдельных подлежащих выполнению шагов.

Для алгоритмически разрешимой постановки задачи всегда имеется много различных способов ее решения, т.е. различных алгоритмов.

Примеры «почти» алгоритмов – медицинский и кулинарный рецепты.

### **Примеры неформальных описаний алгоритмов**

#### *1. Арифметические операции над десятичными числами.*

В начальной школе мы на неформальном уровне изучаем правила вычисления суммы двух чисел и умножения («вычисления в столбик»).

#### *2. Алгоритм Евклида для вычисления наибольшего делителя (НОД).*

*Постановка задачи.* Пусть даны два положительных целых числа  $a$  и  $b$ , надо найти наибольший общий делитель  $\text{НОД}(a, b)$  чисел  $a$  и  $b$ .

*2а. Первая версия алгоритма:*

- если  $a=b$ , то справедливо  $\text{НОД}(a, b)=a$ ;
- если  $a < b$ , то применяем алгоритм НОД к числам  $a$  и  $b-a$ ;
- если  $b < a$ , то применяем алгоритм НОД к числам  $a-b$  и  $b$ .

Используется математическое свойство:

для любых положительных целых чисел  $x$  и  $y$  если  $x < y$ , то  $\text{НОД}(x, y) = \text{НОД}(y-x, x)$ .

*2б. Вторая версия алгоритма:*

- 1) если  $a < b$ , то меняем местами значения (так, чтобы стало  $a \geq b$ );
- 2) делим  $a$  на  $b$ , пусть  $r$  – остаток от деления (имеем  $a \geq b > r \geq 0$ );
- 3) если  $r=0$ , то  $b$  – выход;
- 4) положить (заменить)  $a$  на  $b$ ,  $b$  на  $r$  и вернуться к шагу 2.

*3. Сортировка колоды карт.*

*Постановка задачи.* Имеется колода карт. Пусть на каждой карте зафиксировано одно натуральное число (ради простоты будем считать, что все числа попарно различны). Требуется отсортировать, т.е. упорядочить, колоду карт так, чтобы зафиксированные на картах числа образовывали монотонную (возрастающую или убывающую) последовательность.

*3а. Сортировка путем предсортировки и слияния.*

Заданная колода  $x$  сортируется с помощью следующего предписания:

- если  $x$  пуста или содержит одну карту, то  $x$  отсортирована;
- если  $x$  содержит более одной карты, то  $x$  разделить на две непустые колоды; отсортировать каждую из них и затем слить (объединить) эти колоды в одну отсортированную колоду.

*3б. Сортировка путем вставок.*

Заданная колода сортируется по убыванию с помощью следующего алгоритма, который в соответствующее место отсортированной колоды  $y$  вставляет по очереди карты, выбираемые из неотсортированной колоды  $x$  (процесс начинается с пустой колоды  $y$ ).

- Если колода  $x$  пуста, то  $y$  – искомая отсортированная колода;
- если  $x$  непуста, то из  $x$  берется любая карта и вставляется в подходящее место колоды  $y$  так, чтобы в результате колода  $y$  осталась отсортированной.

Этот алгоритм применяется к уменьшающейся колоде  $x$  и увеличивающейся колоде  $y$ .

### *3в. Сортировка путем выбора.*

Заданная колода  $x$  сортируется по убыванию по следующему алгоритму, который последовательно выбирает из  $x$  «наибольшую» карту и добавляет ее в конец колоды  $y$  (процесс начинается с пустой колоды  $y$ ).

Пусть даны две колоды  $x$  и  $y$ . Пусть колода  $y$  отсортирована по убыванию, и пусть все карты из  $y$  больше любых карт из  $x$ . Колода  $x$  всортировывается в  $y$  по следующему предписанию:

- если  $x$  пуста, то  $y$  – искомая отсортированная колода;
- если  $x$  непуста, то из  $x$  выбирается «наибольшая» карта и добавляется в конец  $y$ .

Алгоритм применяется к уменьшающейся колоде  $x$  и увеличивающейся колоде  $y$ .

### *3г. Сортировка путем перестановки.*

Заданная колода  $x$  сортируется по следующему алгоритму:

- если  $x$  содержит две соседние карты, нарушающие требуемое упорядочение, то эти карты меняются местами, после чего к полученной колоде применяется этот же алгоритм;
- если в  $x$  не встречается ни одной неупорядоченной пары соседних карт, то колода  $x$  отсортирована и тем самым является искомой колодой.

### *4. Алгоритм вычисления дроби $(a+b)/(a-b)$ .*

Сначала вычисляем (используя алгоритмы сложения и вычитания) значения выражений  $a+b$  и  $a-b$  (все равно, последовательно или одновременно), а потом образуем частное от деления полученных результатов (используя алгоритм деления).

*5. Алгоритм, распознающий, можно ли получить последовательность знаков  $a$  из последовательности знаков  $b$  посредством вычеркивания некоторых знаков.*

Если  $a$  – пустая последовательность знаков, то ответом будет «да». В противном случае нужно посмотреть, не пуста ли последовательность  $b$ . Если это так, то ответом будет «нет». Иначе нужно сравнить первый знак последовательности  $a$  с первым знаком последовательности  $b$ . Если они совпадают, то надо снова применить тот же алгоритм к остатку последовательности  $a$  и остатку последовательности  $b$ . В противном случае нужно снова применить тот же алгоритм к исходной последовательности  $a$  и остатку последовательности  $b$ .

Этот алгоритм выдает двужначный результат, – «да» или «нет», т.е. он является алгоритмом распознавания свойства «быть частью данной последовательности знаков».

Хотя описание алгоритма конечно и постоянно, количество фактически выполняемых тактов – величина переменная. Это оказывается возможным благодаря итерации (алгоритмы 2б, 3б, 3в, 3г) или рекурсии (алгоритмы 2а, 3а, 5). При словесном описании итерацию часто записывают в следующей форме «Пока выполнено определенное условие, повторяй». Рекурсия – это когда поставленная задача решается с помощью решения той же самой задачи, но с несколько измененными (более простыми) параметрами.

#### **Классические элементы описания алгоритмов**

- выполнение элементарных шагов;
- разветвление по условию;
- повторение и рекурсия.

### **1.3 Вычислительные структуры**

Установление отношений между содержащейся в представлении информацией и окружающим миром мы называем **пониманием**. Поскольку понимание является **индивидуальным**, субъективным процессом, который трудно сделать общедоступным, мы удовлетворяемся в информатике тем, что интерпрета-

цию представления как носителя информации определим путем отождествления информации с подходящими **математическими (вычислительными) структурами**. Тогда значение представления устанавливается путем отображения на эти математические (вычислительные) структуры.

Представление  $N$  вместе с сопоставленной ему информацией  $I$  называется **объектом (элементом данных)**, а во множественном числе – **данными**. Итак, объект есть пара  $(N, I)$ , при этом информацию  $I$  называют значением объекта, а представление  $N$  – обозначением объекта.

Объекты в алгоритмах играют роль предметов, над которыми производятся определенные операции. На практике классы объектов часто выделяются благодаря тому, что на них определен некоторый естественный процесс обработки информации и ее представлений. (Например, операции сложения и умножения над целыми числами.) Не всякий объект годится как операнд для той или иной операции. Множество объектов, для которых естественным образом определено некоторое количество операций, называется множеством объектов определенного **типа**. Таким образом, тип элементов данных характеризуется операциями, которые могут над ними выполняться.

Вычислительная структура состоит из одного или нескольких множеств объектов, называемых **типами**, и некоторых основных (элементарных, базовых) **операций** над этими типами, каждая с результатом одного из этих типов.

При составлении программы для решения какой-либо задачи необходимо сначала выделить подходящие вычислительные структуры, а затем решить, как эти структуры представлять в языке программирования.

## **Основные вычислительные структуры**

Рассмотрим основные вычислительные структуры и обычное представление этих структур с помощью языка Паскаль.

### **1. Вычислительная структура целых чисел**

Состоит из множества целых чисел и некоторого ряда производимых над ними арифметических операций.

Внутренние операции: сложение, вычитание, умножение и некоторые другие. Операция деления не всегда определена.

Наряду с внутренними операциями для целых чисел, определены операции сравнения: предикаты равно, не равно, больше, меньше, больше или равно, меньше или равно. Результат сравнений имеет значение «истина» или «ложь».

Типичное представление целых чисел в Паскале осуществляется с помощью типа данных *Integer*.

## **2. Вычислительная структура вещественных чисел**

Состоит из множества вещественных чисел и арифметических операций с вещественными результатами. В отличие от целых чисел, для вещественных чисел нет операций получения последующего или предыдущего числа.

Типичное представление вещественных чисел в Паскале осуществляется с помощью типа данных *Real*.

## **3. Вычислительная структура символов**

Состоит из множества символов (знаков), для которых выполняются некоторые операции, например, сравнения. Символы в языке Паскаль представляются с помощью типа *Char*.

## **4. Однородные конечные последовательности**

Состоит из множества конечных последовательностей, элементами которых служат данные одного и того же типа, например числа или символы. В Паскале данной вычислительной структуре соответствуют массивы. Конечные последовательности легко реализуются также динамически в виде списков (используется ссылочный тип). Представления с помощью строк или списков позволяют последовательностям менять свои размеры во время выполнения программы.

## **5. Неоднородные конечные последовательности**

Состоит из множества конечных последовательностей, элементы которых могут быть данными разных типов. В Паскале данной вычислительной структуре соответствуют записи.

## **6. Бесконечные последовательности**

Абстракции заранее не ограниченных последовательностей в языках программирования соответствуют файлы.

## 7. Вычислительная структура значений истинности

При выполнении алгоритма часто необходим разбор случаев (разветвление по условию), и поэтому необходимо иметь возможность формально выражать выполнение или невыполнение тех или иных условий. Поэтому вводится вычислительная структура значений истинности, состоящая из двух элементов данных «истина» и «ложь» – соответственно *True* и *False* в Паскале.

### 1.4 Алгоритмические языки

Использованный ранее неформальный способ описания алгоритмов отличается следующими недостатками:

- громоздкость и излишняя многословность;
- неоднозначность понимания.

Для представления, улучшения читаемости и для простоты представления алгоритмов, которые будут выполняться на компьютере, применяются **алгоритмические языки** (языки программирования).

Запись алгоритма на языке программирования называется **программой**.

При конструировании алгоритмического языка необходимо:

- исходить из некоторого набора вычислительных структур (структур данных);
- вводить как составные операции, так и составные (структурированные) объекты;
- выбрать форму, удобную как для человека, который описывает алгоритм, так и для человека, который должен будет читать и понимать этот алгоритм, – форму, соответствующую кругу человеческих понятий и представлений.

Три составляющие (любого) языка: алфавит, синтаксис и семантика.

**Алфавит** – набор основных символов, «букв алфавита», никакие другие символы в предложениях языка не допускаются.

**Синтаксис** – система правил, определяющая допустимые конструкции из букв алфавита. Синтаксис отвечает на вопрос, является ли последовательность символов текстом (программой) на данном языке или нет?



**Семантика** – система правил, истолковывающая отдельные конструкции языка с точки зрения процесса обработки (смысл программы).

При описании языка и при построении алгоритма используются понятия языка. Любое понятие алгоритмического языка имеет свой синтаксис и свою семантику.

Точнее:

- **синтаксические правила** показывают, как образуется данное понятие из других понятий и (или) букв алфавита;
- **семантические правила** определяют свойства данного понятия в зависимости от свойств используемых в них понятий.

## 1.5 Способы описания синтаксиса алгоритмических языков

Для описания синтаксиса языка необходим какой-то язык (**метаязык**, надъязык). Естественный язык часто используется как метаязык, но для описания языков программирования он громоздкий, нестрогий, неоднозначный.

### Язык металингвистических формул Бэкуса-Наура (язык БНФ)

Этот язык позволяет описывать формальные языки в виде некоторых формул. Описание синтаксиса иерархично: используя символы (буквы алфавита), сначала описывают простейшие понятия, потом более сложные понятия, наконец, описываются синтаксически правильные программы.

С точки зрения синтаксиса, каждое определяемое понятие является **метапеременной** языка БНФ, значением которой может быть любая конструкция из некоторого фиксированного для этого понятия набора конструкций.

Для каждого понятия используется своя **метаформула**, которая состоит из левой и правой частей.

**Левая часть** – определяемое понятие (метапеременная). **Правая часть** задает все множество значений этой метапеременной, все возможные способы конструирования этого понятия.

Метапеременные заключаются в угловые скобки, например,  $\langle \text{число} \rangle$ ,  $\langle \text{арифметическое выражение} \rangle$ . **Метасимвол** « $::=$ » (смысл которого можно передать словом «есть») разделяет левую и правую части метаформул. **Метасимвол** « $|$ » (смысл – «или») отделяет все допустимые конструкции в правой части формулы.

Пример:

$\langle \text{переменная} \rangle ::= A \mid B$

$\langle \text{выражение} \rangle ::= \langle \text{переменная} \rangle \mid \langle \text{переменная} \rangle - \langle \text{переменная} \rangle$   
 $\mid \langle \text{переменная} \rangle + \langle \text{переменная} \rangle$

Понятие  $\langle \text{выражение} \rangle$  допускает следующие синтаксические цепочки:  $A, B, A+A, A-A, B+B, B-B, A+B, B+A, A-B, B-A$ .

Чтобы с помощью конечного множества правил задавать бесконечное множество последовательностей символов, используют *рекурсивные* определения.

Пример:

$\langle \text{двоичная цифра} \rangle ::= 0 \mid 1$

$\langle \text{двоичный код} \rangle ::= \langle \text{двоичная цифра} \rangle \mid \langle \text{двоичный код} \rangle \langle \text{двоичная цифра} \rangle$

Для определения понятия в правой части используется само это понятие. Чтобы не получился порочный круг, правая часть формулы должна содержать хотя бы одно частное определение (не содержащее определяемое понятие).

**Метасимволы**  $\{$  и  $\}$  используются для задания синтаксических конструкций произвольной длины. Таким образом,  $\{\langle \text{конструкция} \rangle\}$  означает, что  $\langle \text{конструкция} \rangle$  может повторяться ноль или более раз.

Пример:

$\langle \text{двоичный код} \rangle ::= \langle \text{двоичная цифра} \rangle \{\langle \text{двоичная цифра} \rangle\}$

Пример (арифметическое выражение в БНФ). Язык арифметического выражения над целыми числами с символами операций  $+$ ,  $-$  и  $*$  образует формальный язык над символами  $(0, \dots, 9, (, ), +, -, *)$ . Этот язык определяется через следующие БНФ-правила синтаксического понятия  $\langle \text{арифм\_выраж} \rangle$ :

$\langle \text{арифм\_выраж} \rangle ::= \langle \text{число} \rangle \mid (\langle \text{арифм\_выраж} \rangle$   
 $\mid \langle \text{арифм\_выраж} \rangle \langle \text{операция} \rangle \langle \text{арифм\_выраж} \rangle$   
 $\langle \text{операция} \rangle ::= + \mid - \mid *$   
 $\langle \text{число} \rangle ::= \langle \text{цифра} \rangle \{ \langle \text{цифра} \rangle \mid 0 \} \mid 0$   
 $\langle \text{цифра} \rangle ::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Еще один пример. Пусть формальный язык «Показушка» описывается следующими формулами Бэкуса-Наура:

$\langle \text{предложение} \rangle ::= \langle \text{подлежащее} \rangle \langle \text{предикат} \rangle$   
 $\langle \text{подлежащее} \rangle ::= \langle \text{именная группа} \rangle$   
 $\langle \text{предикат} \rangle ::= \langle \text{глагол} \rangle \langle \text{именная группа} \rangle \mid \langle \text{глагол} \rangle \langle \text{наречие} \rangle$   
 $\langle \text{именная группа} \rangle ::= \langle \text{существительное} \rangle \mid \langle \text{список прилагательных} \rangle$   
 $\langle \text{список прилагательных} \rangle ::= \langle \text{прилагательное} \rangle \{ \langle \text{список прилагательных} \rangle \}$   
 $\langle \text{существительное} \rangle ::= \text{КОШКА} \mid \text{МЫШКА}$   
 $\langle \text{прилагательная} \rangle ::= \text{ЧЕРНАЯ} \mid \text{БЕЛАЯ} \mid \text{ТОЩАЯ} \mid \text{УПИТАННАЯ}$   
 $\langle \text{глагол} \rangle ::= \text{БЕЖИТ} \mid \text{ЕСТ}$   
 $\langle \text{наречие} \rangle ::= \text{БЫСТРО} \mid \text{МЕДЛЕННО}$

Следующие синтаксические цепочки символов принадлежат множеству значений понятия  $\langle \text{предложение} \rangle$ :

- 1) МЫШКА БЕЖИТ БЫСТРО;
- 2) БЕЛАЯ КОШКА ЕСТ ЧЕРНАЯ КОШКА;
- 3) УПИТАННАЯ, ТОЩАЯ, ЧЕРНАЯ, БЕЛАЯ МЫШКА ЕСТ МЕДЛЕННО.

Описание формальных языков может быть достигнуто в общем случае с помощью так называемых грамматик. БНФ-нотация представляет собой некоторую специальную форму описания для определения таких грамматик, которые называются **контекстно-свободными грамматиками**.

## Синтаксические диаграммы

**Синтаксические диаграммы** графически изображают структуру всех тех конструкций, которые могут быть значением метапеременной языка, соответствующей данному понятию.

Элементы диаграммы:

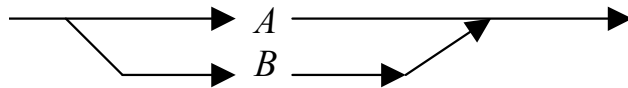
- *основные символы;*
- *понятия языка;*

- *стрелки*, указывающие, какие элементы могут следовать непосредственно за данным элементом.

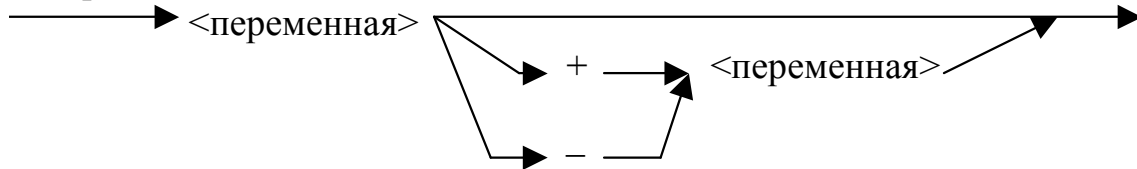
Каждый конечный путь через синтаксическую диаграмму определяет последовательность символов, которая специфицирована с помощью синтаксической диаграммы как синтаксически допустимая.

Пример:

$\langle \text{переменная} \rangle ::=$

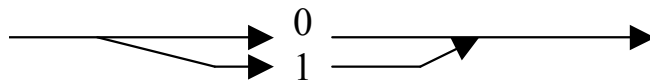


$\langle \text{выражение} \rangle ::=$

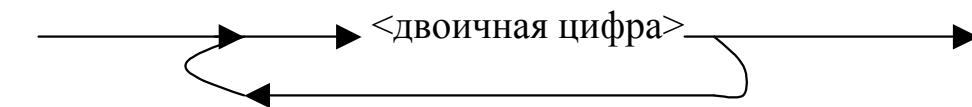


Пример определения конструкций произвольной длины:

$\langle \text{двоичная цифра} \rangle ::=$

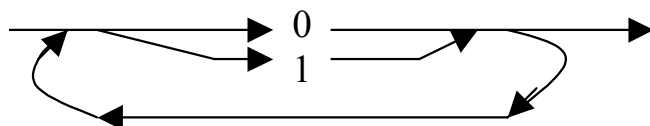


$\langle \text{двоичный код} \rangle ::=$



Другой вариант определения:

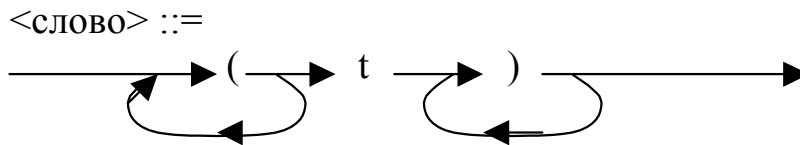
$\langle \text{двоичный код} \rangle ::=$



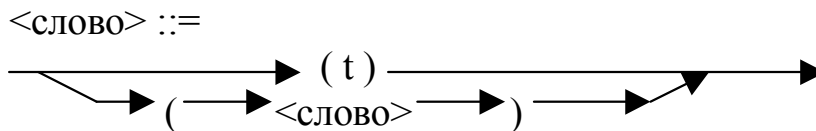
В общем случае рекурсии не избежать. Для примера опишем конструкции следующего вида  $(t)$ ,  $((t))$ ,  $((((t))))$ ,  $(((((t))))))$ , ... .

$\langle \text{слово} \rangle ::= (t) \mid (\langle \text{слово} \rangle)$

Следующее определение на языке синтаксических диаграмм неэквивалентно данному.



Правильное определение:



## 1.6 Семантика программы: функциональная, операционная, аксиоматическая

Для недвусмысленного описания семантики целесообразно выбрать математическую форму описания, т.е. сопоставление математических объектов (например, элементов из определенных множеств и функций) для конструкций языка. Существенно различаются две крайние точки зрения в связи с установлением семантики.

**Функциональная семантика.** Описание функций программы, т.е. установление отношения между входными и выходными данными (экстенциональное или наблюдаемое отношение), называют функциональной семантикой.

**Операционная семантика.** Описание последствий отдельных шагов вычислений, которые имеют место при выполнении программы, называют операционной семантикой.

Функциональная семантика программы может быть получена в общем случае через абстракцию, т.е. избегая частных, что имеет место в операционной семантике.

Принципиально семантика (т.е. значение) программы допускает описание благодаря тому, что задается специфичный способ действия при ее выполнении на конкретной ЭВМ. Однако языки программирования для того и развивались, чтобы формулирование алгоритмов можно было осуществлять единообразно для большого числа различных машин, независимо от специфич-

ных внутренних структур машины. Поэтому представляет интерес машинно-независимое описание алгоритмов в виде программ, а также машинно-независимое описание значения языка программирования. При конструировании программ для решения поставленной задачи необходимо обеспечить, чтобы окончательно составленная программа соответствовала требованиям, т.е. являлась *корректной* в смысле поставленной задачи. Эта корректность не всегда очевидна, особенно при больших структурах программ. Доказательство корректности, в принципе, может быть проведено с помощью семантики языка программирования. Впрочем, многие способы определения семантики практически мало подходят для такого доказательства. Поэтому делаются попытки привести как требования к программам, так и значение программы к логическим формулам. Тогда доказательство корректности можно достичь с помощью логического вывода. Такой подход называется *верификацией* программ. Через правила перевода программ в логические формулы неявно включается также и определение семантики.

Наряду с функциональным и операционным описанием семантики языков программирования, представляют также интерес такие способы описания, которые позволяют делать высказывания о свойствах программы. К ним можно отнести логические формулы, которые определяют («аксиоматизируют») определенные свойства конструкций языков программирования. Тогда мы говорим также об *аксиоматической* семантике. С помощью аксиом и логических правил вывода могут быть выведены и высказывания о программе.

## 1.7 Трансляция и выполнение

Компьютер – это автомат, выполняющий алгоритм. Для того чтобы решить какую-либо задачу с помощью компьютера, необходимо алгоритм ввести в память машины, затем он должен интерпретироваться (т.е. восприниматься и выполняться) аппаратным путем.

Запись алгоритма на языке, понятном машине, называется **машинной программой**, а язык – **машинным языком**. Для разных типов компьютеров машинные языки разные.

Машинные операции кодируются с помощью цифрового кода (номера операции) и информации об операндах – адрес ячейки памяти, отведенной для хранения операнда. Следовательно, машинная программа ненаглядная, трудно понимаемая для человека.

**Отличия алгоритмических языков от машинных языков:**

- большая выразительная возможность: алфавит алгоритмического языка шире алфавита машинного языка;
- набор операций не зависит от набора машинных операций и определяется удобством формулирования алгоритмов;
- одно предложение языка задает достаточно содержательный процесс обработки;
- удобное задание для человека операций, например, математическая запись;
- операнды имеют имена, выбираемые программистом (более удобно, чем адреса);
- более широкий выбор типов данных и, в целом, широкий набор вычислительных структур.

Алгоритм на алгоритмическом языке не может быть выполнен непосредственно на компьютере. Он должен быть переведен на машинный язык.

Если возможен перевод алгоритмического языка на машинный по формальным правилам, то перевод осуществляется машиной, с помощью выполнения определенной машинной программы. Такая программа называется **транслятором** с данного алгоритмического языка (на данный машинный язык).

**Обратите внимание!** Транслятор – это алгоритм, для которого входом служит текст на алгоритмическом языке, а выходом – текст на машинном языке.

Транслятор является, грубо говоря, либо компилятором либо интерпретатором. **Компилятор** читает всю программу целиком, делает ее перевод на машинный язык и помещает команды в память компьютера. После того как программа откомпилирована,

исходная программа больше не нужна. *Интерпретатор* преобразует лишь небольшой фрагмент исходной программы в машинные команды, а затем, дождавшись, когда компьютер их выполнит, переходит к обработке следующего фрагмента (пример – Бейсик).

Если мы всегда пишем правильные программы и имеем возможность работать с хорошим компилятором, то для практических целей можно условно считать компилятор и вычислительную машину единой машиной, которая может непосредственно выполнять программы. Таким образом, мы можем совершенно не задумываться о процессе компиляции. Комбинация компилятора и вычислительной машины иногда называется *виртуальной машиной*.

Нам не нужно было бы вникать в структуру виртуальной машины, если бы не возможность возникновения ошибок.

**Существуют три типа ошибок в программах:**

- **Синтаксические ошибки** – ошибки, которые можно обнаружить при компиляции.
- **Ошибки, обнаруживаемые при выполнении рабочей программы** (при некоторых входных данных программа не работает, например, деление на нуль).
- Ошибки, которые не обнаруживаются компьютером ни при компиляции, ни при выполнении программы (**неправильный алгоритм**).

## 1.8 Компьютеры фон Неймана

Для того чтобы понимать проблематику традиционных языков программирования, нам нужно сначала исследовать их интеллектуального прародителя – компьютер фон Неймана.

**Джон (Янош) фон Нейман (1903–1957)**

Сын преуспевающего будапештского банкира, он рано зарекомендовал себя вундеркиндом. В шесть лет мальчик уже свободно владел древнегреческим, а в восемь освоил основы высшей математики. С середины 1930-х годов фон Нейман жил в США. Он был человеком добродушным и по-своему экстравагантным. Одевался он, скорее, как банкир, а не как профессор. Любил кра-



сивых женщин, изысканную еду, очень увлекался автомобилями, которые разбивал примерно раз в год.

Что такое компьютер фон Неймана? Когда фон Нейман и другие задумывали его более 50 лет назад, это была изящная, практичная и объединяющая идея, которая упрощала ряд существовавших тогда инженерных и программистских задач. Хотя условия, породившие архитектуру этого компьютера, с тех пор радикально изменились, тем не менее мы по-прежнему идентифицируем понятие компьютера с этой концепцией пятидесятилетней давности.

В своей простейшей форме компьютер фон Неймана состоит из трех частей: центрального процессорного устройства, памяти и соединительной шины, которая может за один шаг передавать только одно слово между процессором и памятью (и посылать некий адрес в память). Можно назвать (Дж. Бэкус) эту шину *«бутылочным горлышком» (узким местом)* фон Неймана. В памяти размещается программа и данные, с которыми эта программа работает.

Задача программы состоит в том, чтобы неким существенным образом изменить содержимое памяти; если считать, что эта задача должна быть выполнена исключительно перекачиванием через узость фон Неймана, то становится ясной причина такого названия.

Ирония ситуации состоит в том, что большую часть потока через эту узость составляют не полезные данные, а всего лишь имена данных, а также операции и данные, служащие лишь для вычисления таких имен. Прежде чем слово можно будет послать через шину, его адрес должен находиться в процессоре; поэтому он должен либо быть послан через шину из памяти, либо генерироваться посредством некоторой операции процессора. Если адрес посылается из памяти, то адрес этого адреса должен либо быть послан из памяти, либо генерироваться в процессоре, и т.д. С другой стороны, если адрес генерируется в процессоре, он должен генерироваться либо по фиксированному правилу (например, «добавить 1 к счетчику исполняемых команд»), либо по команде, которая была послана через шину, в последнем случае ее адрес нужно было прежде послать... и т.д.

Итак, программирование на традиционных языках в основном сводится к планированию и спецификации огромного потока слов через узость фон Неймана, причем большая часть этого потока состоит не из самих значащих данных, а из сведений о том, где их искать.

Обычные языки программирования в основном являются высокоуровневыми, сложными версиями компьютера фон Неймана.

## 2 ВВЕДЕНИЕ В ЯЗЫК ПАСКАЛЬ

### 2.1 Историческая справка

**Pascal** – разработан в 1968–1971 годах Никлаусом Виртом в Цюрихском Институте Информатики (Швейцария).

**Цель** – инструмент для обучения программированию как систематической дисциплине. Обнаружилась чрезвычайная эффективность при применении и надежность программирования.

Паскаль – язык, ориентированный на машину фон-неймановского типа; такие языки называются *императивными* (imperative – содержащий указание на выполнение некоторого действия) или *процедурными*.

Мы будем описывать *Турбо-Паскаль* (расширение стандарта Паскаля).

### 2.2 Алфавит языка Паскаль

**Основные символы** языка (*лексемы*) – либо отдельные литеры на клавиатуре, либо их некоторые комбинации.

```

<лексема> ::= <буква> | <цифра> | <спецсимвол>
<буква> ::= a|b|...|z|A|B|...|Z|_
<цифра> ::= 1|2|3|4|5|6|7|8|9|0
<спецсимвол> ::= <знак арифметической операции> |
                 <знак операции сравнение> |
                 <разделитель> |
                 <служебное слово>
<знак арифметической операции> ::= *|/|+|-
<знак операции сравнения> ::= =|<|>|<=|>=
<разделитель> ::= .|,|;|:|(|)|{ }|^|'|#|$|@
  
```

Служебные слова – «зарезервированные» слова – служат для определенных целей.

```

<служебные слова> ::= begin|end|var|const|if|then|else|function|for|.....
  
```

## 2.3 Переменные

Содержимое памяти (и определенных регистров) характеризует состояние фон-неймановской машины. Выполнение программ этими машинами ориентировано сугубо на изменение состояний. Соответственно этому в машине шаг за шагом выполняются определенные инструкции (команды), и с каждым шагом изменяется состояние памяти, т.е. содержимое определенных ячеек памяти. С любым элементом данных, используемым в алгоритмическом языке, связано такое понятие языка как *переменная*. Любая переменная имеет *имя* (обозначение) и *значение*. Переменная именуется свое значение – элемент данных. Значение переменной во время работы программы меняется. В языке Паскаль любая переменная используется только в рамках какой-то определенной вычислительной структуры, т.е. *значением конкретной переменной может быть только элемент данных определенного типа*.

Имя переменной в Паскале синтаксически описывается с помощью *идентификаторов*:

<идентификатор> ::= <буква> {<цифра> | <буква>}

Длина идентификатора произвольна, но компилятор с языка Турбо-Паскаль воспринимает только первые 63 символа.

## 2.4 Основные понятия языка Паскаль

**Операторы.** *Каждый оператор представляет законченную фразу языка и определяет некоторый вполне законченный этап обработки данных.*

**Основные операторы** (не содержат в своем составе других операторов):

- Оператор присваивания. Предназначен для изменения значений переменных.
- Оператор ввода (чтения). Предназначен для ввода в программу входных данных.

- Оператор вывода (записи). Предназначен для вывода из программы результатов работы.

Из заданных операторов с помощью композиций различных форм можно получать операторы, которые называются *производными* операторами. Различные формы композиции операторов позволяют задавать *последовательное выполнение, разветвление по условию и повторение*.

**Описания данных.** Программа на Паскале начинается с описания используемых переменных. Для каждой переменной указывается ее имя и тип значения.

Описание последовательности действий, которые необходимо выполнить, следует после описания всех переменных.

Пример описания переменных:

```
var
  a:integer;
  x,y,z:real;
  Sinus:real;
```

В программе на языке Паскаль любая используемая переменная, за исключением *системных (предопределенных)*, должна быть определена, причем определение переменной должно текстуально предшествовать первому ее использованию. **Область известности** («видимости») переменной ограничивается *блоком*, в котором она определена. Каждая переменная, описанная в блоке, должна упоминаться в описаниях не более одного раза. Это относится не только к переменным, а вообще ко всем идентификаторам.

Паскаль допускает введение в программы объектов, внешне похожих на переменные, но которые, в отличие от них, не могут изменять свое значение. Такие объекты называются **константами**. Можно сказать, что идентификатор константы является *синонимом* некоторого определенного значения, которое сопоставляется с этим идентификатором при описании.

Пример описания констант:

```
const
  One=1;
  HighLimit=1000;
  pi=3.14159265358;
```

Тип константы определяется по ее значению.

Константа может входить во все конструкции, в которых может присутствовать связанное с ней значение. Естественно, не допускаются ситуации, когда идентификатору константы предлагается изменить значение.

Использование в программе идентификаторов констант вместо записи конкретных значений считается *хорошим стилем программирования*, так как делает программу более «читабельной» и способствует лучшему ее пониманию, без какого бы то ни было снижения эффективности (в части быстродействия и объема занимаемой памяти). Кроме того, если некоторые важные для программы значения обозначены идентификаторами (например, границы массивов, показатели точности вычислений), то при необходимости их легко изменить, исправив описание соответствующих констант. В противном случае эти значения будут «растворены» в тексте программы и придется просматривать ее целиком, чтобы произвести нужные изменения.

Грубая схема программы:

```
const
    <описания констант>
var
    <описания переменных>
begin
    <операторы>
end.
```

## 2.5 Правила записи текста программы

Для облегчения понимания текста программы и для упрощения трансляции требуется выполнение определенных правил написания текста программы. Нам потребуются следующие понятия.

### Разделители

- **Пробел** – литера, которой соответствует пустая позиция в строке текста (на бумаге, на экране), не имеет графического изображения.

- В текстах программ допускаются фрагменты пояснительного текста – **комментарии**. Наличие комментариев не изменяет смысл программы и не влияет на ее выполнение. Комментарии представляют собой *произвольную последовательность символов* (не обязательно из алфавита языка, т.е. допускаются и русские буквы), *заключенную в фигурные скобки*. Пример: {это комментарий}. Комментарий может находиться между двумя любыми лексемами программы.
- **Конец строки** – управляющая литера. Текст программы разбивается на строки, при работе редактора в конце строк помещается невидимая литера «конец строки».

### Правила записи

1. Между двумя конструкциями языка (являющимися идентификаторами, числами или служебными словами) должен быть хотя бы один разделитель.
2. Разделители не должны быть внутри идентификаторов, чисел и служебных слов.
3. Если это не противоречит правилам 1, 2, то между двумя любыми лексемами может находиться любое число разделителей (транслятором они игнорируются).

## 2.6 Система типов языка

В Паскале используются несколько вычислительных структур. Данные в этих структурах принадлежат определенным **типам**. Элементы данных представляются переменными в программе. Поэтому любая переменная характеризуется своим типом. *Под типом в данном случае понимается множество значений, которые может принимать переменная, и, как следствие, множество операций, допустимых над данной переменной.*

Паскаль является **типизированным** и **статическим** языком. Это означает, что тип переменной определяется при ее описании и не может быть изменен. Такой подход способствует большей аккуратности и ответственности при составлении программ, делает их поддающимися автоматической (при компиля-

ции) проверке на корректность и, в конечном итоге, приводит к более высокой надежности создаваемых программ.

Паскаль имеет развитую и изощренную систему вычислительных структур (типов). На основании небольшого числа стандартных типов программист может сконструировать данные произвольной структуры и сложности, адекватно отражающие информационную природу задачи.

**Таблица 2.1 – Классификация предопределенных типов языка Паскаль**

Группа	Подгруппа	Название	Идентификатор
Простой	Скалярный	Короткий целый	ShortInt
		Байтовый	Byte
		Слово	Word
		Целый	Integer
		Длинный целый	LongInt
		Символьный	Char
		Булев	Boolean
	Вещественный	Вещественный	Real
		С ординарной точностью	Single
		С двойной точностью	Double
		С повышенной точностью	Extended
		Сложный	Comp
Составной (структурный)		Строковый	String
		Массив	Array
		Множество	Set
		Файл	File
		Запись	Record
Ссылочный		Ссылочный	Pointer
Процедурный		Процедура	Procedure
		Функция	Function
Объектный		Объектный	Object

Кроме указанных типов, программист может сам определить ограниченные, перечислимые и свои ссылочные типы.

Составные и ссылочные типы можно считать некоторыми правилами для построения более сложных типов из более про-



стых. Процедурные типы в некотором отношении расширяют понятие подпрограмм, позволяя обращаться с подпрограммами, как с переменными.

## 2.7 Основные вычислительные структуры в Паскале

Основные типы языка: целые, вещественные, символьный, булевский.

### Целые типы

Эта группа типов обозначает множества целых чисел в различных диапазонах.

Тип **integer** (целый).

Диапазон: -32768...32767.

Размер памяти: 2 байта (16 двоичных разрядов).

Над целыми значениями допустимы следующие операции:

а) арифметические бинарные (два операнда):

- сложение (знак операции +);
- умножение (знак операции \*);
- вычитание (знак операции -);
- деление (знак операции /);
- деление нацело с отбрасыванием целой части (знак операции **div**);
- взятие остатка от целочисленного деления (знак операции **mod**);

б) арифметическая унарная (одноместная) операция – изменение знака (знак операции -);

в) операции сравнения:

- равно (=);
- не равно (<>);
- больше (>);
- больше или равно (>=);
- меньше (<);
- меньше или равно (<=);

г) стандартные функции (в обычной функциональной записи):

- абсолютная величина (пример, **abs**(-4)=4);

- квадрат (пример, **sqr**(6)=36);

д) функция **odd**(<целое>) (**odd** – нечетный) – проверяет аргумент на нечетность.

Результат операции деления принадлежит вещественному типу, результаты операций сравнения и функции **odd** – булевский тип, все остальные операции выдают целый тип.

Примеры операции **div**:

```
6 div 4 =1;
-7 div -2 = 3;
34 div -45 = 0.
```

Операция **m mod n** применяется только при  $n > 0$ :

```
6 mod 4 =2;
-6 mod 4 =-2.
```

В таблице 2.2 перечислены остальные целые типы в Паскале.

Таблица 2.2

Идентификатор	Описание типа	Множество допустимых значений
Shortint	8-битный целый со знаком	-128...127
Longint	32-битный целый со знаком	-2147483648...2147483647
Byte	8-битный целый без знака	0...255
Word	16-битный целый без знака	0...65535

## Вещественные типы

Обозначают множества вещественных чисел в различных диапазонах.

Тип **real** (вещественный).

Диапазон значений:  $2.9 \cdot 10^{-39} \dots 1.7 \cdot 10^{38}$ .

Число цифр мантииссы: 11–12.

Размер памяти – 6 байт.

Вещественные значения могут изображаться в форме с фиксированной точкой и в форме с плавающей точкой. В первом случае целая и дробная части вещественного числа отделяются

друг от друга символом «.» (точка). Обе эти части должны обязательно присутствовать, например,

17.384

0.5

Следующие примеры демонстрируют неправильные формы записи чисел:

.3 (правильно 0.3)

10. (правильно 10.0)

Вещественное число в форме с плавающей точкой записывается как пара вида  $\langle \text{мантисса} \rangle E \langle \text{порядок} \rangle$ .

Такое обозначение понимается как «мантисса, умноженная на 10 в степени, равном порядку». Например,  $7E-2$  означает  $7 \cdot 10^{-2}$ ,  $12.25E+6$  или  $12.25E6$  оба обозначают  $12.25 \cdot 10^6$ . Мантисса представляется в виде целого числа или как вещественное с фиксированной точкой. Порядок обозначается целым числом; допускаются как положительные, так и отрицательные значения порядка.

Над вещественными значениями допустимы следующие операции:

- арифметические операции (сложение, умножение, вычитание, деление, изменение знака) – операнды могут быть или оба вещественными, или один из них целым, а другой вещественным; результат всегда вещественный;
- операции сравнения – результат булевский;
- стандартные функции:
  - **sqrt(x)** (квадратный корень);
  - **sin(x)** (синус);
  - **cos(x)** (косинус);
  - **arctan(x)** (арктангенс);
  - **ln(x)** (натуральный логарифм);
  - **exp(x)** (экспонента).

Для этих функций аргумент может быть вещественным или целым, результат – всегда вещественный;

- **abs(x)** (абсолютная величина числа);
- **sqr(x)** (квадрат) – тип результата совпадает с типом аргумента;

- операции округления:

- **trunc (x)** –

x – вещественное, результат – целая часть числа x, дробная часть отбрасывается и не округляется, **trunc (5.2)=5**, **trunc (-5.8)=-5**;

- **round (x)** –

x – вещественное, результат – округленное целое,

**round (x)=trunc (x+0.5)** при  $x \geq 0$ ,

**round (x)=trunc (x-0.5)** при  $x < 0$ .

В таблице 2.3 перечислены остальные вещественные типы в Паскале.

Таблица 2.3

Идентификатор	Описание типа	Диапазон
Single	4-байтовый вещественный с ординарной точностью, 7-8 значащих цифр	$-3.4 \cdot 10^{38} \dots -1.5 \cdot 10^{-45}$ , $1.5 \cdot 10^{-45} \dots 3.4 \cdot 10^{38}$
Double	8-байтовый вещественный с двойной точностью, 15-16 значащих цифр	$-1.7 \cdot 10^{308} \dots -5.0 \cdot 10^{-324}$ , $5.0 \cdot 10^{-324} \dots 1.7 \cdot 10^{308}$
Extended	10-байтовый вещественный с повышенной точностью, 19-20 значащих цифр	$-1.1 \cdot 10^{4932} \dots -1.9 \cdot 10^{-4951}$ , $1.9 \cdot 10^{-4951} \dots 1.1 \cdot 10^{4932}$
Comp	8-байтовый сложный тип	$-2^{63} \dots 2^{63}-1$ (только целые значения)

### Символьный (литерный) тип

Стандартное имя типа **char** (character).

Значениями этого типа являются элементы набора литер, определяемого реализацией языка. В Турбо-Паскале символы составляют множество **ASCII** (*American Standard Code for Information Interchange*) – 256 символов. Множество ASCII содержит заглавные и строчные латинские буквы, цифры; может содержать русские буквы. Некоторые символы не имеют изображения (например, «конец строки»).

Размер памяти, занятый символом, – 1 байт.

Множество символов *упорядочено*.

Функция **ord(c)** ( $0 \leq \text{ord}(c) \leq 255$ ) дает порядковый номер литеры *c*.

Функция **chr(i)** дает литеру (значение типа *char*) с порядковым номером  $0 \leq i \leq 255$ .

**ord(chr(i)) = i**

**chr(ord(c)) = c**

Функция **pred(c)** ( $\text{ord}(c) > 0$ ) дает предыдущую литеру.

Функция **succ(c)** ( $\text{ord}(c) < 255$ ) дает последующую литеру.

**pred(c) = chr(ord(c) - 1)**

**succ(c) = chr(ord(c) + 1)**

Графическое представление символа (за исключением символов, не имеющих графического представления) – соответствующий знак в кавычках (апострофах) '+', 'y', '!', 'л', '"'. Символ, не имеющий графического представления, можно задать с помощью записи #<порядковый номер символа>, например, #7 – гудок.

*Цифры '0', '1', ..., '9' имеют подряд идущие порядковые номера. Малые латинские буквы 'a', 'b', ..., 'z' имеют подряд идущие номера. Большие латинские буквы 'A', 'B', ..., 'Z' имеют подряд идущие порядковые номера.*

Операции сравнения над символами такие же, как для чисел. Результат – булевский (логический) тип. Эти операции над символами эквивалентны сравнениям порядковых номеров символов.

### Булевский (логический) тип

Стандартное имя типа – **boolean**.

Используется для представлений значений двузначной логики:

**true** (истина), **false** (ложь).

Размер памяти для значений – 1 байт.

Логический тип упорядочен: **true > false**.

**ord(false) = 0**

**ord(true) = 1**

**pred(true) = false**

**succ(false)=true**

Используются операции сравнения, например, **true<false = false**.

### Логические операции

Аргументы (операнды) – булевский тип, результат – булевский тип.

Служебные слова: **and**, **or**, **not**.

Операции **and** и **or** – бинарные коммутативные и ассоциативные:

- **x and y** – логическое умножение (логическое "и") или **конъюнкция** – значение **true** тогда и только тогда, когда оба операнда – **true**;
- **x or y** – логическое сложение (логическое "или") или **дизъюнкция** – значение **false** тогда и только тогда, когда оба операнда **false**.
- Операция **not x** – унарная операция – логическое отрицание: **not true = false**, **not false = true**

## 2.8 Выражения

*Выражение* представляет собой формальное правило для вычисления некоторого (нового) значения. Понятие выражения присутствует практически в любом достаточно развитом языке программирования, а синтаксические правила для построения выражений очень похожи в различных языках.

В самом общем случае можно сказать, что *выражения строятся из операндов, знаков операций и круглых скобок*. Переменные и константы сами по себе являются частным случаем выражений.

*Операнды* представляют собой «элементарные» значения; ими могут быть переменные, константы, вызовы функций.

Примеры выражений:

**a+b+c\*2**

**a/b/c**

```

7-a*(sin(x)+2)
a=b
(2>3)=(7<>10)
pred('x')<x
(not(2>3))or(1=1)
true<=false

```

Значение выражения вычисляется во время вычисления. Мы говорим о *выражениях целого, вещественного, булевского* и других типов, имея в виду тип значения, вычисляемого данным выражением. Тип выражения определяется синтаксически, исходя из типов операндов и операций.

Поэтому выражения должны быть синтаксически правильными.

Примеры неправильных выражений:

```

2**3
e==3
(2+1) )
true+false
not (a)  - недопустимо, если a не булевского типа.

```

Синтаксис выражений предусматривает определенное старшинство операций (*приоритеты*); приоритеты задают очередность выполнения операций, указанных в выражении. Круглые скобки предназначены для указания порядка (очередности) вычислений, если необходимо изменить тот порядок действий, который определен приоритетами операций.

Перечень операций в порядке убывания приоритета:

- 4) not
- 3) \*, /, div, mod, and
- 2) +, -, or
- 1) =, <>, <,>, <=, >=

*Выражение ничего не говорит, что делать с полученным результатом (значением).*

## 2.9 Оператор присваивания

В результате выполнения оператора присваивания переменная получает значение выражения.

$\langle \text{оператор присваивания} \rangle ::= \langle \text{переменная} \rangle := \langle \text{выражение} \rangle$

*Выражение в правой части должно иметь тот же тип, что и переменная.*

### Арифметический оператор присваивания

Тип выражения – **integer** или **real**. Разрешается вещественной переменной присваивать целое значение.

Пример:

```

const
    pi=3.14159;
var
    x,a,b,c,r:real;
    i:integer;
begin
    .....
    x:=0;
    i:=i+1; {пример использования предыдущего значения
переменной}
    c:=sqrt(a*a+b*b);
    x:=2*pi*r;
    {следующие присваивания - неправильны}
    z:=i+2;
    x=2*pi*r;
    i:=5/4;
    x:=a*-b/2; {два знака операции не допускаются подряд}
    .....
end.
```

### Логический оператор присваивания

В левой части оператора присваивания присутствует переменная типа **boolean**.

Примеры:



```

var
    d,b,c:boolean;
    x,y:real;
    k:integer;
begin
    .....
d:=x<2*y;
d:=true;
d:=odd(k) ;
d:=not not b;
d:=not(not(b)) ;
c:=d and (x<>y) and b;
c:=(b or d) and not b;
c:=(d or b) < not c;
{неправильно =>} c:=d or x>y;
{ правильно  c:=d or (x>y);}
    .....
end.

```

### Литерный оператор присваивания

В левой части оператора присваивания присутствует переменная типа **char**. Справа может быть только константа, переменная или функция.

Примеры:

```

var
    sym,alpha,beta:char;
begin
    .....
sym:='+' ;
alpha:=sym;
beta:=succ(sym) ;
alpha:=chr(40) ;
    .....
end.

```

## 2.10 Ввод-вывод

Для выполнения операций ввода-вывода служат четыре процедуры: **read**, **readln**, **write**, **writeln**.

Процедура чтения **read** обеспечивает ввод числовых данных, символов и некоторых других типов данных с клавиатуры, в результате чего эти данные становятся значениями переменных.

Вызов процедуры: **read(x1, x2, . . . , xN) ;**

где **x1, x2, . . . , xN** – переменные допустимых типов данных. Значения **x1, x2, . . . , xN** набираются минимум через один пробел на клавиатуре и высвечиваются на экране. После набора данных для одной процедуры **read** нажимается клавиша ввода <Enter>. Значения переменных должны вводиться в строгом соответствии с синтаксисом языка Паскаль. Если соответствие нарушено (например, **x1** имеет тип **integer**, а при вводе набирается значение типа **char**), то возникают ошибки ввода-вывода.

Если в программе имеются несколько подряд идущих процедур **read**, данные для них вводятся потоком, т.е. после набора значений переменных для одной процедуры **read** можно, не нажимая клавишу <Enter>, продолжить набор данных для следующей процедуры в той же строке и только потом нажать клавишу ввода.

Процедура чтения **readln** аналогична процедуре **read**. Единственное отличие заключается в том, что после считывания последней переменной в списке значений для одной процедуры **readln** данные для следующей процедуры ввода будут считываться с начала новой строки (т.е. ввод потоком невозможен).

Процедура записи **write** производит вывод на экран числовых данных, символов, булевских значений и строк.

Вызов процедуры

**write(y1, y2, . . . , yN) ;**

где **y1, y2, . . . , yN** – выражения вида **integer, real, char, boolean** и строки.

Процедура записи **writeln** аналогична процедуре **write**, но после вывода последнего в списке значения для текущей процедуры **writeln** происходит перевод курсора к началу следующей строки. Процедура **writeln**, записанная без параметров, вызывает перевод строки.

В процедурах вывода **write** и **writeln** имеется возможность записи выражения, определяющего *ширину поля* вывода.

Если **x** и **i** – целочисленные выражения, а **y** – вещественное выражение, то вызов **write(x:i,y:i)** печатает значения **x** и **y** – каждое в крайние правые позиции полей шириной **i**.

**Write(y:i:x)** – в крайние правые позиции поля шириной **i** символов выводится десятичное представление значения **y** в формате с фиксированной точкой, причем после десятичной точки выводится **x** цифр, представляющих дробную часть числа.

## 2.11 Последовательное выполнение и составной оператор

Для того чтобы указать в программе последовательное выполнение операторов **S1**, **S2**, ..., **SN** (т.е. после оператора **S1** должен выполняться оператор **S2** и т.д.), операторы отделяются символом «;»

```
S1;  
S2;  
.....  
SN
```

Используя служебные слова **begin** и **end**, можно из последовательности операторов сделать *составной* оператор:

```
begin  
  S1;  
  S2;  
  .....  
  SN  
end
```

Составной оператор применяется в тех случаях, когда синтаксис языка допускает использование только одного оператора, в то время как семантика программы требует задание последовательности действий.

## 2.12 Условный оператор

*Условный оператор* применяется для задания разветвления в программе, т.е. выбора действий в зависимости от истинности или ложности какого-либо условия.

Пример:

Для того чтобы **z** стало равным наибольшему из двух чисел **x** и **y**, необходимо выполнить присвоение **z:=x** либо **z:=y**.  
Две формы условного оператора: *полная и сокращенная*.

```
<полный условный оператор>::=
  if <логическое выражение> then <оператор>
                                else <оператор>
```

Примеры:

```
if x>y then z:=x
      else z:=y;
```

```
if x<0 then begin
                x:=x+h;
                y:=y-h;
                d:= not d
              end
      else begin
                x:=0;
                y:=0
              end;
```

```
if x>0 then sgn:=1
      else if x=0 then sgn:=0 else sgn:=-1;
```

```
<сокращенный условный оператор>::=
  if <логическое выражение> then <оператор>;
```

Например, оператор **x:=abs (x)** эквивалентен **if x<0 then x:=-x**.

Поменять местами значения переменных **x** и **y** так, чтобы в **x** было большее значение:

```
if y>x then begin
                t:=x;
                x:=y;
                y:=t
              end;
```

Ветви then и else могут снова содержать условные операторы, например:

```
if x>0 then sgn:=1
      else if x=0 then sgn:=0 else sgn:=-1;
```

**Неоднозначность в условном операторе следующего вида:**

```
if B1 then if B2 then S1 else S2
```

## Возможные истолкования

```
a) if B1 then begin
    if B2 then S1 else S2
end;
b) if B1 then begin if B2 then S1 end
    else S2;
```

В Паскале используется вариант а).

**Совет.** Если **if** следует после **then**, то всегда используйте **begin** и **end**.

## 2.13 Оператор цикла с предусловием

<b>while</b> <логическое выражение> <b>do</b>		заголовок цикла
<оператор>		тело цикла

Тело цикла повторяется, пока истинно *предусловие*.

Примеры:

```
{s=сумма целых чисел от 1 до n}
s:=0;i:=1;
while i<=n do
  begin
    s:=s+i;
    i:=i+1
  end
```

```
{x1:=наибольший общий делитель x и y}
x1:=x;y1:=y;
while x1<>y1 do
    if x1>y1 then x1:=x1-y1
        else y1:=y1-x1;
```

{Для данного  $M > 0$  требуется найти наименьшее целое число  $k \geq 0$ , такое что  $3^k > M$ }

```
y:=1;k:=0;
while y<=M do
begin
k:=k+1;
y:=y*3;
end;
```

## 2.14 Оператор цикла с постусловием

Число повторений заранее неизвестно, но известно условие, при выполнении которого цикл должен завершиться.

```
repeat {<оператор>} until <логическое выражение>
```

*Цикл с постусловием всегда выполняется, по крайней мере, один раз!*

Пример. Вычислить наименьшее  $n$ , для которого  $y = 1 + 1/2 + 1/3 + \dots + 1/n \geq 5$ .

```
y:=0;n:=0;
repeat
n:=n+1;
y:=y+1/n;
until y>=5;
```

## 2.15 Оператор цикла с параметром

Такие операторы обычно используются, когда число повторений цикла может быть определено перед его началом. Кроме того, **циклы с параметром** позволяют задать автоматическое изменение значения некоторой переменной и использование этого значения в последовательных итерациях.

Более конкретно данный вариант оператора цикла определяет:

- *диапазон* изменения значений управляющей переменной и, одновременно, *число повторений* оператора, содержащегося в теле цикла;

- *направление изменения* значения переменной (возрастание или убывание);
- собственно действия, выполняемые на каждой итерации (*оператор тела цикла*).

<оператор цикла с параметром>::=  
 for <переменная> := <диапазон> do <оператор>

<диапазон>::=  
 <выражение> <направление> <выражение>

<направление>::= to | downto

На использование управляющей переменной (параметра) налагаются следующие ограничения:

1. Управляющая переменная должна иметь *дискретный тип* (целый, символьный, булевский, перечислимый).
2. Начальные и конечные значения диапазона должны иметь *тот же тип*, что и параметр.
3. В теле цикла *запрещается явное изменение* значения управляющей переменной (например, оператором присваивания).
4. После завершения оператора значение параметра становится *неопределенным*.

Семантику данного оператора цикла можно представить следующим образом. Оператор

**for V := Expr1 to Expr2 do Body;**

эквивалентен оператору:

```
begin
Temp1:=Expr1;
Temp2:=Expr2;
if Temp1 <= Temp2 then
  begin
    V:=Temp1;
    Body;
    while V <> Temp2 do
      begin
        V:=succ(V) ;
        Body
      end
    end
  end
end
```

Примеры:

```
{y=x^n}
y:=1;
for i:=1 to n do y:=y*x;

{печать латинского алфавита с конца}
for c:='z' downto 'a' do write(c);
```

## 2.16 Примеры бесконечных циклов

```
y:=0;i:=1;
while i<=n do y:=y+1/i;i:=i+1;

while true do <оператор>
```

{Вычислить площадь под параболой  $f(x)=x^2$  от  $a=0$  до  $b=10$  по формуле трапеции  
 $S=h*[f(a)/2+f(a+h)+f(a+2*h)+\dots+f(b-h)+f(b)/2]$ , где  
 $h=(b-a)/n$  }

```
h:=(b-a)/n;
S:=h*(a*a+b*b)/2;
x:=a;
repeat
  x:=x+h;
  S:=S+x*x*h
until x=b-h; {Нужно x>=b-h, иначе равенство может и
не выполниться}
```

## 2.17 Пустой оператор

*Пустой оператор* является в некотором смысле парадоксальной конструкцией, так как он, во-первых, не имеет «графического» начертания, а, во-вторых, не производит никаких действий. Необходимость введения такого понятия диктуется в первую очередь синтаксическими причинами. Синтаксис языка определяет символ ';' (точка с запятой) как разделитель операторов. Таким образом, последний оператор, например, в составном опера-



торе не должен завершаться этим символом, так как после него не следует оператор, а идет служебное слово **end**:

```
begin S1;S2;...;SN end
```

Однако, если предположить, что между последним оператором **SN** и служебным словом **end** расположен пустой оператор, то становится допустимой такая форма записи:

```
begin S1;S2;...;SN; end
```

что в ряде случаев более предпочтительно.

Пустой оператор удобно использовать и в тех случаях, когда по той или иной причине необходимо организовать некоторый составной оператор, не содержащий в себе ни одного оператора, например,

```
writeln('нажмите любую клавишу');  
repeat until Keypressed;
```

**Экзотические конструкции:**

- 1) **begin S1;;;;;S2; end**
- 2) **if Cond then begin end**  
    **else while true do;**
- 3) **if Cond then;**

Все эти примеры операторов синтаксически правильны (но бесполезны).

## 2.18 Ограниченные типы

Стандартные дискретные типы (целый, символьный, булевский, перечислимый) предопределены, но пользователь может создать (определить) собственный дискретный тип.

**Ограниченный тип** определяется сужением (ограничением) допустимого диапазона значений некоторого стандартного дискретного типа. Задаются минимальное и максимальное значения диапазона.

Пример:

<b>1..10</b>	ограничение некоторого
<b>-100..100</b>	целого типа
<b>'a'..'z'</b>	ограничение символьного типа

Описание переменных:

```
var
    s:1..10;
```

Можно ввести идентификатор для нового типа. Для этого в программе вводится раздел описаний, начинающийся со служебного слова **type**:

```
type
    digits=0..9;
    sto=-100..100;
    letter='a'..'z';
var
    n:digits;
```

Схема программы в более общем виде:

<b>const</b>		
<описания констант>		
<b>type</b>		любой из этих разделов
<описание типов>		может отсутствовать
<b>var</b>		
<описания переменных>		
<b>begin</b>		
<операторы>		
<b>end.</b>		

Ограниченный тип *наследует* все свойства базового типа (в частности, набор допустимых операций).

**Активно используйте ограниченные типы:**

- наглядность программы,
- контроль ошибочных выходов значений за пределы заданного диапазона (как при трансляции, так и при выполнении).

## 2.19 Функции

Кроме стандартных (*предопределенных*) функций, программист может определить новые функции. Смысл функций заключается, в первую очередь, в том, чтобы определить алгоритм вычисления нового значения некоторого простого (или ссылочного) типа. В этом отношении функции подобны выражениям,

которые также вычисляют значения. В соответствии с этим вызов функции является одним из допустимых операндов выражения, обозначая в нем то значение, которое вычисляет («вырабатывает» или «возвращает») функция.

Новая функция (а их может быть несколько) определяется в разделе описания, как правило, после описания переменных в виде:

```
function <имя функции><формальные параметры>:<тип
значения>;
    <описания локальных констант, типов, переменных>
begin
    <операторы>
end;
```

Примеры:

1)

```
var
    m,n,k:real;

function max(x,y:real):real;
{максимум двух чисел}
begin
    if x>y then max:=x else max:=y
end;

begin
    writeln('введите три числа');
    readln(m,n,k);
    writeln('максимум=',max(max(m,n),k) )
end.
```

2)

```
type
    natural=0..maxint;
var n:natural;
function sumDigits(a:natural):natural;
{вычисляет сумму цифр числа a}
var m,b:natural;
begin
    m:=0;
    repeat
        b:=a mod 10;
```

```

        m:=m+b;
        a:=a div 10;
    until a=0;
    sumDigits:=m
end;

begin
    writeln('введите число');
    readln(n);
    writeln('Сумма цифр числа ',n,' = ',sumDigits(n));
end.

```

<формальные параметры>::=  
 (<описание параметров>{;<описание параметров>} )

<описание параметров>::=  
 <идентификатор> {,<идентификатор>} : <идентификатор типа>

*Список формальных параметров может быть пустым – функция может быть без параметров.*

В теле функции могут использоваться **локальные объекты** (в том числе и формальные параметры), а также **глобальные объекты** (т.е. типы, константы, переменные, описанные вне функции).

Чтобы функция действительно вычисляла нужное значение, необходимо присутствие в теле функции оператора присваивания (*новая форма!*) в виде:

<имя функции>:=<выражение, вычисляющее возвращаемое значение>;  
 Таких операторов может быть несколько. Хотя бы один должен работать.

При вызове функции **фактические** параметры должны *соответствовать формальным* по количеству и типу.

Понятие функции является частным случаем подпрограммы. **Подпрограмма** – это запись вспомогательных алгоритмов. Другим частным случаем подпрограмм в языке Паскаль является **процедура**.

## 2.20 Примеры программ для задач без массивов

Задача 1. Даны две целые переменные  $a$ ,  $b$ . Составить фрагмент программы, после исполнения которого значения переменных поменялись бы местами (новое значение  $a$  равно старому значению  $b$ , и наоборот).

*Решение.* Введем дополнительную целую переменную  $t$ .

```
t:=a;
a:=b;
b:=t;
```

Почему нельзя обойтись без дополнительной переменной  $t$ , написав **a:=b; b:=a**?

Решить предыдущую задачу, не используя дополнительных переменных.

*Решение.* Начальные значения  $a$  и  $b$  обозначим  $a_0$ ,  $b_0$ .

```
a:=a+b; {a=a0+b0, b=b0}
b:=a-b; {a=a0+b0, b=a0}
a:=a-b; {a=b0, b=a0}
```

Задача 2. Дано целое число  $a$  и натуральное (целое неотрицательное) число  $n$ . Вычислить  $a^n$ .

Для использования циклов полезно понятие инварианта цикла. Логическое утверждение, истинное перед выполнением и после выполнения каждого шага цикла, называется **инвариантным отношением** или просто **инвариантом** цикла. Во многих случаях полезно явно найти инвариант (и указать его в виде аннотации к циклу).

*Решение.* Введем целую переменную  $k$ , которая меняется от 0 до  $n$ , причем поддерживается такое свойство (инвариант)  $b=a^k$ .

```
k:=0; b:=1;
{b = a в степени k}
while k <> n do
  begin k:=k+1; b:=b*a; end;
```

Другое решение той же задачи.

```

k:=n; b:=1;
{a в степени n = b * (a в степени k)}
while k <> 0 do
    begin
        k:=k-1;
        b:=b*a;
    end;

```

Решить предыдущую задачу, если требуется, чтобы число действий (выполняемых операторов присваивания) было порядка  $\log n$  (т.е. не превосходило бы  $C \log n$  для некоторой константы  $C$ ;  $\log n$  – логарифм  $n$  по основанию 2).

*Решение.*

```

k:=n; b:=1; c:=a;
{a в степени n = b * (c в степени k)}
while k <> 0 do
    if k mod 2 = 0 then begin
        k:= k div 2;
        c:= c * c;
    end
    else begin
        k:=k-1;
        b := b*c;
    end;

```

Каждый второй раз (не реже) будет выполняться первый вариант оператора выбора (если  $k$  нечетно, то после вычитания 1 становится четным), так что за два цикла величина  $k$  уменьшается, по крайней мере, вдвое.

Задача 3. Дано натуральное  $n$ . Вычислить  $1/0!+1/1!+1/2!+\dots+1/n!$  ( $x!$  обозначает произведение всех натуральных чисел от 1 до  $x$ ,  $0! = 1$ ). Требуется, чтобы количество операций (выполненных команд присваивания) было порядка  $n$  (не более  $C \cdot n$  для некоторой константы  $C$ ).

*Решение.* Инвариант:  $Sum=1/0!+1/1!+\dots+1/k!$ .  $Last=1/k!$  (важно не вычислять заново каждый раз  $k!$ ).

```

Sum:= 0; Last:=1;k:=0;
While k=<n do
  Begin
    Sum:= Sum+Last;
    k:=k+1;
    Last:=Last/k;
  End

```

Задача 4. *Вычисление значений функции.* Следующая программа предназначена для вывода таблицы значений функции  $F(x) = x/(1+x)$  на экран.

```

var
  x: Real;
  k: word;

function F(x: Real): Real;
begin
  F:= x/(1.0+x);
end;

begin
  x:=0.0;
  Writeln('Таблица значений функции  F(x) = x/(1.0+x) ');
  Writeln;
  Writeln('x':10, 'F(x) ':20);
  Writeln;

  for k:= 0 to 50 do
    begin
      Writeln(x:10:4, F(x):20:10);
      x:= x+0.1;
      if k mod 10 = 9 then Readln;
    end;
  Writeln;
  Write('Нажмите <Enter:>');
  Readln;
end.

```

Обратите внимание на то, что программа приостанавливает вывод через каждые 10 строк и ожидает нажатия клавиши Enter.

Задача 5. *Извлечение корня из действительных чисел.* Известно много различных методов извлечения корня. В программировании удобнее всего пользоваться универсальным методом, т.е. таким методом, в соответствии с которым одни и те же, хотя бы и более продолжительные вычисления выполняются при любых исходных данных. Таким методом является метод итерации. Корень с натуральным основанием

$$y = \sqrt[m]{x}, \quad x \geq 0,$$

где  $m$  – натуральное число, извлекается по формуле

$$y_n = \frac{1}{m}((m-1)y_{n-1} + \frac{x}{y_{n-1}^{m-1}}).$$

Поясним, каким образом по этой формуле можно получить корень  $y = \sqrt[m]{x}$ . Сначала примем, что корень равен произвольному числу (например,  $y_0 = 1$ ; доказано, что получающееся в результате значение корня не зависит от того, какое значение  $y_0$  мы избрали в качестве исходного), вычисляем новое значение  $y_1$ , затем опять новое значение корня  $y_2$  и т.д. Каждое новое значение  $y_n$  оказывается все ближе к подлинному значению корня. Когда разность между значением  $y_n$  и предыдущим значением  $y_{n-1}$  становится весьма малой – меньшей, нежели допустимая погрешность, принимаем, что полученное значение  $y_n$  является корнем и вычисление заканчивается.

Приведем ряд примеров, показывающих, как извлекается кубический корень из 125 при различных исходных значениях:

$y_0$	125.0	1.0	10.0
$y_1$	83.3	42.3	7.1
$y_2$	55.6	28.2	5.6
$y_3$	37.1	18.9	5.1
$y_4$	24.7	12.7	5.0
$y_5$	16.6	8.7	5.0



Написав  $yy$  вместо  $y_n$  и  $y$  вместо  $y_{n-1}$  (где  $n=1,2,3,\dots$ ), составим следующий фрагмент программы извлечения корня  $m$ -ой степени из неотрицательного числа  $x$ :

```
yy:=1.0; {исходное значение корня}
repeat
    y:=yy;
    yy:=y*(m-1)/m +x/power(y, m-1)/m
until abs(yy-y)<epsilon;
root:=yy;
```

Здесь  $\text{power}(y, m)$  – функция, которая возводит число  $y$  в  $m$ -ую степень, а  $\epsilon$  – допустимая погрешность при извлечении корня.

Этот, правильный на первый взгляд, фрагмент программы практически не очень хорош. Извлекая корень более высокой степени, можно получить очень большой результат функции  $\text{power}$ . Например, если мы будем извлекать корень  $\sqrt[100]{10000.0}$ , т.е. выполним программу при  $m=100$  и  $x=10000.0$ , то при первом прохождении цикла получим  $yy \approx 100$ . При повторном прохождении цикла необходимо будет это число возвести в 99-ую степень. Хотя интервал действительных чисел в вычислительной машине очень велик, однако в данном случае мы получим значение, не попадающее в этот интервал, т.е. произойдет переполнение. Это выглядит неестественным: если число, из которого мы извлекаем корень, находится в допустимом интервале, то и значение корня будет находиться в том же самом интервале. Поэтому для пользователя данной программы будет совершенно непонятно, почему машина прекратила вычисления вследствие переполнения, тогда как исходное число не очень велико.

Чтобы избежать переполнения, не будем возводить число  $yy$  в степень; вместо этого будем много раз делить число  $x$  на  $yy$ .

Составим функцию извлечения корня  $m$ -ой степени ( $m$  – натуральное число) из неотрицательного числа  $x$  с точностью до 0.00001:

```

Function root(x:real; m:integer):real;
  {корень m-ой степени из x}
  const epsilon=0.00001;
  var y, yy, w, z, mm: real;
      k: integer;
begin
  yy:=1.0; {исходное значение корня}
  mm:=(m-1)/m; z:= x/m;
  repeat
    y:=yy;
    k:=1;
    w:=z;
    while (k< m) and (w >= epsilon) do
      begin
        w:=w/yy;
        k:=k+1;
      end;
    yy:=y*mm+w;
  until abs(yy-y)< epsilon;
  root:=yy
end;

```

Обратим внимание на два момента.

1. Формулу извлечения корня мы разбили на две части. Одну часть действия мы вынесли из цикла, а другую часть оставили в цикле. Это было сделано из соображений экономии. Значения (часть значений), которые не изменяются при прохождении цикла, достаточно вычислить один раз, до начала цикла. В цикле остаются только те значения, которые изменяются в процессе прохождения цикла. Такое преобразование вычислений называется *чисткой цикла*.

Когда требуется извлечь корень более высокой степени, то при вычислении  $w:=w/yy$  частное может оказаться весьма малым, еще прежде чем деление будет произведено  $m-1$  раз. Поэтому цикл может быть закончен раньше, т.е. когда частное станет меньшим, нежели *epsilon*. Это сокращает время работы машины. Однако здесь есть еще более существенный момент. Когда получаются очень малые действительные числа, может возникнуть странное на вид явление – переполнение в сторону малых чисел. Ведь, чем меньше число, тем больше абсолютная величина его

порядка (например, 1E–60, 1E–70, 1E–89 и т.д.). В результате число, выражающее порядок, может не уместиться в отведенное для него место. Прерывая цикл раньше, мы избегаем этого явления.

## 2.21 Более подробно о символьном типе

Таблица кодов ASCII использует 8 двоичных разрядов и состоит из двух частей. Первая, в которую входят символы с кодами 0–127, является универсальной, а вторая (коды 128–255) предназначена для специальных символов и букв национальных алфавитов (в том числе и русского).

Универсальная таблица (коды 0–127), кроме обычных символов (латинских букв, цифр и др.), имеет специальные *управляющие* символы. Они представляют собой команды, вывод которых на стандартное выходное устройство приводит к выполнению определенных действий. Эти символы имеют мнемонические двух- или трехбуквенные сокращения, пришедшие к нам из эры телеграфа. К управляющему символу можно обратиться по его ASCII-коду или по Ctrl-последовательности. Последняя представляет собой код, порожденный одновременным нажатием клавиши Ctrl и какой-либо другой клавиши.

Предположим, например, что в программе имеется описание **Var ch: Char;**

Тогда операторы

```
ch := chr(7);
```

```
ch := #7;
```

```
ch := ^G;
```

присваивают символьной переменной **ch** одно и то же символьное значение. Здесь **^G** обозначает Ctrl-последовательность Ctrl+G (управляющий символ BEL – звуковой сигнал). Знак **#** и следующая за ним целая константа без знака обозначают код символа.

Из 32 управляющих символов вам, скорее всего, могут понадобиться те, которые сведены в таблицу 2.4.

Таблица 2.4

Код	Ctrl-последовательность	Использование функции Chr	Мнемоническое обозначение	Действие
#7	^G	Chr(7)	BEL	Звуковой сигнал динамика
#8	^H	Chr(8)	BS	Возврат курсора на одну позицию
#9	^I	Chr(9)	HT	Горизонтальная табуляция
#10	^J	Chr(10)	LF	Перевод строки
#12	^L	Chr(12)	FF	Прогон строки
#13	^M	Ctrl(13)	CR	Возврат каретки
#26	^Z	Ctrl(26)	SUB	Конец файла
#27	^[	Ctrl(27)	ESC	Символ Escape

До сих пор мы занимались вопросами вывода символов на экран. Обратимся теперь к их вводу с клавиатуры. При нажатии на символьную клавишу естественно ожидать увидеть на экране в месте расположения курсора тот символ, который был введен.

```

var
  ch: Char;
begin
  Writeln('Введите символ: '); Readln(ch);
  Writeln('Введен символ: ',ch);
  Writeln('Нажмите <Enter>: '); Readln;
end.

```

Эта программа использует оператор **Readln** для того, чтобы считать с клавиатуры один символ и вывести в следующей строке то, что было считано. В данном случае при вводе символа вы увидите его на экране еще до того, как будет нажата клавиша Enter. Нажмем клавишу с символом «a», а затем Enter, и программа сообщит, что она считала символ «a». Запустим программу, в ответ на приглашение «Введите символ» нажмем любую символьную клавишу и будем держать ее, не отпуская. На

экране при этом будут отображаться символы, и происходить это будет до тех пор, пока количество символов не достигнет 127. После этого нажатие клавиш уже не будет приводить к отображению новых символов, только динамик компьютера будет недовольно пощелкивать. Это сигнал о том, что полностью заполнен *буфер клавиатуры* – рабочая область памяти, в которой временно могут храниться до 127 символов, введенных с клавиатуры.

Можно ли прочитать значение нажатой клавиши так, чтобы она не отображалась при этом на экране? Да, можно. Для этого надо подключить к программе дополнительный модуль CRT и использовать из этого модуля в программе функцию **ReadKey**, которая именно это и делает. (Что такое модули и как их использовать – будет рассмотрено в одной из последних лекций, а сейчас достаточно знать, что доступ к модулю CRT обеспечивается строчкой программы «uses CRT».) При вводе символа функция **ReadKey** не сдвигает курсор и поэтому дает возможность вместо введенного символа вывести любой другой. Следующая программа, используя эту функцию, заменяет каждую строчную букву заглавной.

```
uses CRT;
var
  ch: Char;
begin
  Writeln('Вводите строчные латинские буквы ',
          'или z для того, чтобы выйти из программы');
  repeat
    ch:=ReadKey;
    Write(UpCase(ch));
  until ch = 'z';
end.
```

Функция **UpCase** преобразует маленькую латинскую букву в соответствующую большую, а любой другой символ оставляет без изменения.

Реакция на нажатие специальных клавиш при чтении с клавиатуры отличается от реакции на нажатие алфавитно-цифровых клавиш. Чтобы понять, как это происходит, познакомимся с тем, как работает клавиатура. При нажатии клавиши формируются два

кода — код символа и код сканирования (называемый также расширенным кодом). Код сканирования важен для тех клавиш, которым не сопоставлены обычные символы. Это, например, клавиши перемещения курсора, функциональные клавиши и т.д. Расширенный код является двухбайтовым. При считывании функцией **ReadKey** специальной клавиши (например, F1) сначала возвращается код символа, равный 0, а второй вызов **ReadKey** возвращает значение кода сканирования.

Таблица 2.5

Нажатые клавиши	Код сканирования	Нажатые клавиши	Код сканирования
<b>Ctrl+@</b> или <b>Ctrl+3</b>	3	<b>PgDn</b>	81
<b>Shift+Tab</b>	15	<b>Ins</b>	82
<b>Alt+1</b> .. <b>Alt+=</b>	120..131	<b>Del</b>	83
<b>Alt+Q</b> .. <b>Alt+P</b>	16..25	<b>F1..F10</b>	59..68
<b>Alt+A</b> .. <b>Alt+L</b>	30..38	<b>Shift+F1</b> .. <b>Shift+F10</b>	84..93
<b>Alt+Z</b> .. <b>Alt+M</b>	44..50	<b>Ctrl+F1</b> .. <b>Ctrl+F10</b>	94..103
<b>Alt+Enter</b>	28	<b>Alt+F1</b> .. <b>Alt+F10</b>	104..113
<b>Home</b>	71	<b>Ctrl+Prt Sc</b>	114
↑	72	<b>Ctrl+Home</b>	119
<b>PgUp</b>	73	<b>Ctrl+PgUp</b>	132
←	75	<b>Ctrl+←</b>	115
→	77	<b>Ctrl+→</b>	116
<b>End</b>	79	<b>Ctrl+End</b>	117
↓	80	<b>Ctrl+PgDn</b>	118

Следующий фрагмент программы предназначен для определения кода сканирования, соответствующего нажатой клавиши.

```
uses CRT;
var
  ch: Char;
  .....
  Write('Нажмите клавишу');
```

```
ch := ReadKey;  
Writeln;  
if ch <> #0 then Writeln('Обычная клавиша, Ord(ch)  
                        = ', Ord(ch))  
    else begin  
        Write('Символьный код: #0. ');  
        ch:= ReadKey;  
        Writeln('Код сканирования: ', Ord(ch));  
    end;
```

### 3 ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

#### 3.1 Машинно-ориентированный оператор: оператор перехода

Для программ в императивных (процедурных) языках при их выполнении характерно строго последовательное выполнение содержащихся в них операторов. Какой оператор в зависимости от обстоятельств должен выполняться в качестве следующего, управляется через специальные условия. Переход от одного оператора к другому в процессе выполнения программы называется *течением программы* или *управлением потоком*. В языке для управления течением программы используются, прежде всего, условные операторы и циклы. В машинном языке для управления потоком используется *команда перехода*. С помощью этой команды и реализуются условные операторы и циклы. Но в языке Паскаль существует и прямой аналог команды перехода – *оператор перехода*.

Отдельные операторы можно пометить с помощью *меток*. Благодаря этому может быть использован специальный оператор **goto m** (называемый *переходом*), так что в результате выполнения этого оператора выполнение программы продолжается с оператора, помеченного меткой **m**.

Все метки, используемые в программе (блоке), должны быть описаны в специальном разделе **label**. Метка представляет собой идентификатор или целое число от 1 до 9999.

Пример:

```
label
    m, 1, metka, 56;
```

Оператор, помеченный меткой, имеет вид

```
<метка>:<оператор>;
```

Циклы можно реализовать, используя условный оператор и оператор перехода. Например, цикл

```
while B do S;
```

эквивалентен операторам:

```
2: if not B then goto 1;
   S;
   goto 2;
1;
```



*Не допускаются переходы вовне подпрограмм или внутрь их. Переход внутрь структурного оператора может вызвать непредсказуемые эффекты, хотя компилятор не выдает сообщение об ошибках.*

### 3.2 Диаграммы управления потоком

Графическое представление управления потоком выполняемых команд может быть дано с помощью так называемых **диаграмм управления потоком (блок-схем)**. С математической точки зрения блок-схема есть **ориентированный граф** с различными типами узлов, которые помечены определенными выражениями или операторами.

Блок-схема для императивной программы может быть систематически выведена из программы.

Условный оператор **if C then S1 else S2** выражается с помощью следующей блок-схемы, изображенной на рис. 3.1

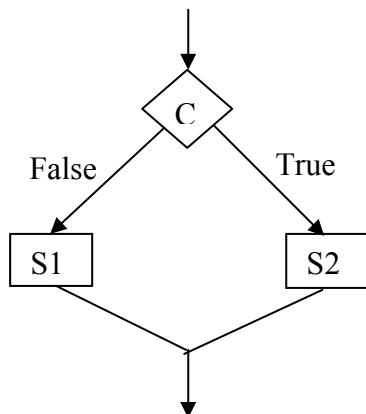


Рис. 3.1

Операторы присваивания (и вызовы подпрограмм) записываются в прямоугольниках. Булевские выражения, управляющие потоком, записываются в ромбах.

Последовательной композиции **S1 ; S2** соответствует соединение блок-схем для **S1** и соответственно для **S2** с помощью стрелки от **S1** до **S2** (рис. 3.2).

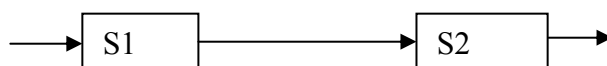


Рис. 3.2

Оператору цикла **while B do S** соответствует блок-схема рисунка 3.3.

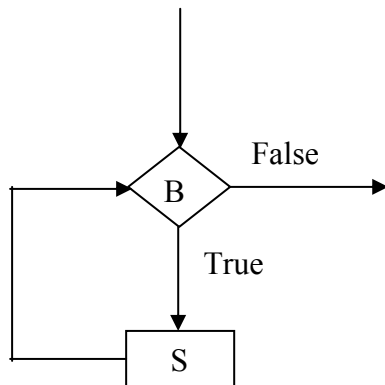


Рис. 3.3

Блок-схемы особенно часто применялись в ранние годы программирования, так как с их помощью надеялись получить наглядное представление о структуре программы, точнее, о структуре потока программы. Однако и сейчас блок-схемы достаточно широко используются в очень прагматических областях программирования.

Решающим для сегодняшнего, лишь спорадического, использования блок-схем является их фундаментальный недостаток: они содержат слишком мало структур и, в особенности, не содержат никаких графических изображений потоков данных. При более сложных программах выясняется, что преимущества наглядного представления управления потоком тотчас теряются.

### 3.3 Структурное программирование

Рассмотрим меры, повышающие надежность программ и облегчающие их сопровождение.

Не существует формального аппарата для получения нужного алгоритма, следовательно, разработка алгоритма (а значит, и программы) ведется методом *«проб и ошибок»*. После создания (и во время создания) алгоритма необходима проверка алгоритма на правильность и анализ эффективности.

Идеи структурного программирования появились после осознания факта – *программы пишутся для людей, на машине они только обрабатываются.*

Трансляция и выполнение программы трудностей практически не вызывает. Проверку правильности программы, модификацию (исправления и изменения) приходится выполнять человеку.

Суть структурного программирования – ***программирование, ориентированное на общение с людьми, а не с машиной.*** Это и означает, что запись программы должна быть максимально удобной для ее восприятия и понимания людьми (в том числе и людьми, не являющимися ее авторами).

**Теорема.** *Любой алгоритм можно описать, используя только последовательное выполнение операторов, условные операторы и циклы.*

Оператор перехода теоретически избыточен и практически вреден. Использование операторов перехода подвергается серьезной критике, так как поощряет создание малопонятных и трудно модифицируемых программ, которые вызывают большие сложности при отладке и сопровождении. Естественный способ чтения и прослеживания работы программы – «сверху вниз и слева направо», поэтому ***программируйте без goto!***

Для лучшего понимания программы предлагаются следующие особенности написания:

- на строке не более одного оператора;
- делайте отступы – они выделяют блоки, поясняют структуру программы.

Рекомендации:

а)

```
S1;
S2;
S3;
.....
```

б)

```
    If  B  then S1
        else S2;
```

в)

```
while  B  do
    S;
```

г)

```
repeat
  S
until B;
```

д)

```
for i:=1 to n do
  S;
```

### 3.4 Решение задачи

Большинство процессов разработки программного обеспечения – это процессы решения некоторых задач. Удачные решения сложных проблем проектирования обычно находят те студенты, которые сознательно или подсознательно, применяют общие правила решения задач.

Понимание задачи	
<i>Начальное понимание задачи.</i>	Что является входными данными (или аргументами)? Что является выходными данными (или результатами)?
<i>Имя программы или функции.</i>	Что является спецификацией задачи?
<i>Какие использовать вычислительные структуры и, следовательно, какие использовать типы?</i>	Является ли постановка задачи удовлетворительной? Достаточная, или излишняя, или противоречивая? Есть ли специальные условия на входные и выходные значения?
	Можно ли задачу разбить на части? Может быть, следует начертить блок-схему или использовать псевдокод?

Составьте план решения	
<i>При разработке программы Вы должны думать о связях между входом и выходом.</i>	Встречались ли Вы с этой задачей ранее? Хотя бы в слегка измененной форме?
	Знаете ли Вы какую-нибудь задачу,

*Если нет непосредственной связи, то Вы могли бы подумать о вспомогательных задачах, которые помогли бы найти решение.*

*Вы должны придумать для себя план решения задачи.*

связанную с этой? Знаете ли Вы какие-нибудь программы или функции, которые могли бы быть полезны?

Рассмотрите спецификацию задачи. Попробуйте найти знакомую задачу с той же или похожей формулировкой.

Вы обнаружили задачу, связанную с вашей и уже решенную. Можете ли Вы использовать это? Можете ли Вы использовать ее результаты? Можете ли Вы использовать ее методы? Не следует ли Вам внести некоторые вспомогательные части в вашу программу?

Если Вы не можете решить предложенную задачу, то попробуйте решить задачу, связанную с искомой задачей. Можете ли Вы представить подобную задачу, но более легкую для решения? А более общую задачу? А, может, более специализированную? Или аналогичную?

Можете ли Вы решить часть задачи? Можете ли Вы получить что-нибудь полезное из входных данных? Какая информация помогла бы Вам вычислить выходные данные? Как можно изменить вход и выход, так чтобы они были как можно близки друг к другу?

Использовали ли Вы все входные данные? Использовали ли Вы специальные условия на входные данные? Взяли ли Вы в расчет все, что используется в спецификации задачи?

## Выполнение плана

*Переведите проект вашей программы в код на конкретном языке программирования.*

*Подумайте, как вы можете записать программу на конкретном языке. Следует ли Вам воспользоваться разбором случаев? Должны ли Вы написать программу в виде последовательных шагов? Или Вам необходимо воспользоваться итерацией или рекурсией?*

*Вам необходимо учитывать программы, которые Вы уже писали до этого, и знать встроенные функции языка или библиотеки.*

При написании программы будьте уверены, что Вы следуете своему плану. Уверены ли Вы, что каждый шаг делает именно то, что он должен?

Вы должны писать программу по этапам. Думайте, на какие различные случаи можно разбить задачу; в частности, есть ли различные случаи для входных величин? Можете ли Вы получать результаты отдельно по частям и как вместе собрать эти части, чтобы получить окончательный результат?

Вы можете свести решение вашей задачи первоначально к решению с «меньшими» входными данными и воспользоваться полученным решением для решения исходной задачи, т.е. воспользоваться рекурсией.

Ваш проект может потребовать у Вас решить более общую задачу или более специальную. Напишите решения для этих случаев. Они могут послужить путеводной нитью к нужному решению или могут быть использованы в решении.

Посмотрите на программы, которые вы писали прежде. Можно ли их использовать? Или модифицировать? Могут ли они помочь вам построить искомое решение?

Взгляд назад – анализ решения	
Проверьте Ваше решение: как его можно улучшить?	<p>Можете ли Вы проверить работу программы для какого-то множества аргументов?</p> <p>Подумайте, как Вы написали бы эту программу, если бы начали решать задачу снова?</p> <p>Подумайте, как Вы можете использовать эту программу или этот метод для построения другой программы?</p>

### 3.5 Разработка программы

При оценке мастерства программиста необходимо различать:

- умение пользоваться алгоритмическим языком, т.е. **кодирование** (технический аспект);
- умение программировать (проектировать программу).

Типичное распределение стоимости затрат при создании программного продукта:

Программирование (кодирование)	– 8%
Проектирование	– 17%
Тестирование	– 25%
Сопровождение	– 50%

#### Этапы разработки:

- создание алгоритма;
- проектирование программы;
- кодирование;
- тестирование.

С любого этапа возможен возврат к любому более раннему этапу.

#### Метод пошаговой детализации

Этот метод также называется методом разработки «сверху-вниз» или методом «нисходящего проектирования». Он считается

наиболее эффективным при разработке программ и состоит из следующих этапов.

**А.** В решаемой задаче выделяется небольшое число (3–5) подзадач (более простые самостоятельные задачи). В проектируемой программе намечается соответствующее число блоков (подпрограмм) для решения подзадач.

Для блоков определяются:

- назначение;
- порядок выполнения;
- связь блоков по обрабатываемым данным.

Важно определение функционального назначения блока – что является входом, а что – выходом, что он делает; как делает – не уточняется.

**Б.** Общая схема программы составлена – проверяем правильность.

**В.** Если отдельные подзадачи достаточно сложны, повторяем процедуру детализации; выделяем новые подзадачи в данной подзадаче.

**Г.** Кодлируем.

Наиболее типичные крупные блоки:

1. Задание исходных данных. Контроль правильности исходных данных.
2. Решение поставленной задачи.
3. Выдача результатов.

#### **Особенности пошаговой детализации**

- На каждом шаге детализации (уточнения) принимается небольшое число решений.
- Детализация блока проходит локально, независимо от остальной программы. Проверка правильности блока может проходить независимо от правильности всей программы.
- Проектирование программы не связано с тем языком, на котором ведется кодирование.
- Блоки должны быть достаточно независимыми друг от друга.



Критерии проектирования (выбора) блоков:

- 1) сложность взаимодействия блока с другими блоками должна быть меньше сложности его внутренней структуры;
- 2) хороший блок снаружи проще, чем внутри;
- 3) хороший блок проще использовать, чем построить.

При нисходящем проектировании сначала отлаживается главная программа, в которой вместо еще не написанных подпрограмм стоят «заглушки» («муляж» подпрограммы – вместо работы печатает только сообщение о своей работе). По мере детализации «заглушки» заменяются написанными подпрограммами, тем самым проверяется работа подпрограмм.

*Восходящее проектирование* («снизу – вверх») – сначала создаются подпрограммы, а только потом – сама программа. Практически при нисходящем проектировании может использоваться частично восходящее проектирование.

### **Способы фиксирования результатов проектирования**

А. Блок-схема.

Сначала описание алгоритма можно представлять, используя диаграмму управления потоком, в укрупненном блочном виде. Блок обычно неэлементарен, не сводится к одному оператору (его размеры не ограничены и выбираются произвольно). Другие блоки связаны с данным блоком только через точки входа и выхода. Поэтому если блок правильно решает задачу, т.е. всегда дает нужный результат, то его внутренняя структура несущественна для остальной части алгоритма. Отсутствие детального описания внутренней структуры блока не мешает пониманию того, как работает алгоритм в целом; важно лишь, чтобы было четко определено, какие блоки запускают данный блок в работу, где лежит его исходная информация, где будет записан результат и куда переходить после окончания его работы.

Потом работа крупных блоков уточняется и заменяется соответствующими блок-схемами.

После того как получена достаточно подробная блок-схема программы, пишется код программы.

Недостатки блок-схем при проектировании:

- трудоемкость вычерчивания и перечерчивания при модификациях;
- потеря наглядности при детализации.

Б. Специальный язык – псевдокод.

Правила обработки данных и условия не формализуются – запись на естественном языке. В качестве языка проектирования можно использовать сам Паскаль, задавая содержание еще не детализированных блоков в виде комментариев Паскаля. При детализации блоков комментарии следует оставлять.

*При хороших комментариях структур данных, алгоритма и сложных мест в структурированной программе нет необходимости ни в какой другой документации!*

### Пример описания на псевдокоде алгоритма

{Алгоритм, распознающий, можно ли получить последовательность знаков *a* из последовательности знаков *b* посредством вычеркивания некоторых знаков.}

```

if {a - пустая последовательность знаков} then
    {ответ= "да"} else
if {b - пустая последовательность} then {ответ=
    "нет"}
else begin
    {сравнить первый знак последовательности a
    с первым знаком последовательности b}
    if {знаки совпадают}
        then {надо снова применить тот же алго-
            ритм к остатку
            последовательности a и остатку
            последовательности b}
        else {нужно снова применить тот же алго-
            ритм к исходной
            последовательности a и остатку
            последовательности b}
end;
```

### Пример разработки

**Гипотеза Гольдбаха** (1742 г., Гольдбах – академик С.-Петербургской академии наук): *любое четное число больше двух представимо в виде суммы двух простых чисел.*

**Задача.** Дано натуральное  $m$ . Проверить гипотезу Гольдбаха для всех четных чисел, меньших  $m$ .

При проектировании программ будем использовать метод пошаговой детализации. В каждой версии программы внесенные изменения по сравнению с предыдущей версией выделяются курсивом.

- **Вариант 1.**

```
var
    i,m: integer;
begin
    writeln('Введите натуральное m');
    readln(m);
    i:=2;
    while i<m do
        begin
            {проверяем гипотезу Гольдбаха для четного
              числа i}
            i:=i+2
        end;
    end.
```

- **Вариант 2.** Вводим функцию **gold** в виде «заглушки» (пока она еще абсолютно ничего не делает).

```
var
    i,m: integer;
function gold(n:integer):boolean;
    {n удовлетворяет гипотезе Гольдбаха <=> значение
    функции - истина}
begin
    end;

begin
    writeln('Введите натуральное m');
    readln(m);
```

```

i:=2;
while i<m do
  begin
    {проверяем гипотезу Гольдбаха для четного числа i}
    if not gold(i)
      then writeln(i, ' - не удовлетворяет
        гипотезе Гольдбаха');
    i:=i+2
  end;
end.

```

- Вариант 3. Чтобы проверить взаимодействие «заглушки» с основной программой, присваиваем функции **gold** постоянно ложное значение.

```

var
  i,m: integer;

function gold(n:integer):boolean;
{n удовлетворяет гипотезе Гольдбаха <=> значение
функции - истина}
begin
  {заглушка}
  gold:=false
end;

begin
  writeln('Введите натуральное m');
  readln(m);
  i:=2;
  while i<m do
    begin
      {проверяем гипотезу Гольдбаха для четного
        числа i}
      if not gold(i)
        then writeln(i, '- не удовлетворяет
          гипотезе Гольдбаха');
      i:=i+2
    end;
  end.

```

- Вариант 4. Только число 4 представляется в виде суммы двух четных простых чисел ( $4 = 2+2$ ). Все большие четные числа, если удовлетворяют гипотезе Гольдбаха, то представляются только в виде суммы двух нечетных простых чисел. Поэтому уточняем функцию `gold`: ищем простые числа только среди нечетных и первое проверяемое четное число полагаем равным 6.

```

var
    i,m: integer;

function gold(n:integer):boolean;
{n удовлетворяет гипотезе Гольдбаха <=> значение
функции - истина}
var j: integer;
begin
gold:=false;
for j:=3 to n-3 do
    if {j и n-j - простые числа} then gold:=true
end;

begin
writeln('Введите натуральное m>=6');
readln(m);
i:=6;
while i<m do
    begin
        {проверяем гипотезу Гольдбаха для четного
        числа i}
        if not gold(i)
            then writeln(i, ' - не удовлетворяет
            гипотезе Гольдбаха');
        i:=i+2
    end;
end.

```

В таком виде программу даже нельзя откомпилировать: в функции **gold** в условном операторе отсутствует логическое выражение.

- Вариант 5. Определяем функцию **prime** в виде «заглушки».

```

var
    i,m: integer;
function prime(n:integer):boolean;
{n - простое число <=> значение функции - истина}
begin
    prime:=true
end;
function gold(n:integer):boolean;
{n удовлетворяет гипотезе Гольдбаха <=> значение
функции - истина}
var j: integer;
begin
gold:=false;
for j:=3 to n-3 do
    if prime(j) and prime(n-j) then
        begin
            writeln(n, '=', j,
                '+', n-j);
            gold:=true
        end
    end;

begin
writeln('Введите натуральное m>=6');
readln(m);
i:=6;
while i<m do
    begin
        {проверяем гипотезу Гольдбаха для четного
        числа i}
        if not gold(i)
            then writeln(i, ' - не удовлетворяет
                гипотезе Гольдбаха');
        i:=i+2
    end;
end.

```

- Вариант 6. «Заглушку» в функции **prime** заменяем полноценным кодом.

```
var i,m: integer;
```

```
function prime(n:integer):boolean;
{n - простое число <=> значение функции - истина}
var
```

```
    k: integer;
```

```
begin
```

```
    prime:=true;
```

```
    for k:=2 to n-1 do
```

```
        if n mod k=0 then prime:=false
```

```
end;
```

```
function gold(n:integer):boolean;
```

```
{n удовлетворяет гипотезе Гольдбаха <=> значение функции - истина}
```

```
var j: integer;
```

```
begin
```

```
gold:=false;
```

```
for j:=3 to n-3 do
```

```
    if prime(j) and prime(n-j) then
```

```
        begin
```

```
            writeln(n, '=', j,
```

```
            '+', n-j);
```

```
            gold:=true
```

```
        end
```

```
end;
```

```
begin
```

```
writeln('Введите натуральное m>=6');
```

```
readln(m);
```

```
i:=6;
```

```
while i<m do
```

```
    begin
```

```
        {проверяем гипотезу Гольдбаха для четного  
        числа i}
```

```
        if not gold(i)
```

```
            then writeln(i, ' - не удовлетворяет ги-  
            потезе Гольдбаха');
```

```
        i:=i+2
```

```
    end;
```

```
end.
```

Уже создана работоспособная программа, решающая проблему Гольдбаха. Но эта программа неэффективна. Следующие версии программы постепенно увеличивают ее эффективность.

- Вариант 7. Оптимизируем функцию **prime**. Достаточно проверять наличие делителей только среди чисел, не превышающих половины  $n$ .

```
function prime(n:integer):boolean;
{n - простое число <=> значение функции - истина}
var
    k: integer;
begin
    prime:=true;
    for k:=2 to n div 2 do
        if n mod k=0 then prime:=false
    end;
```

- Вариант 8. Оптимизируем функцию **prime** в большей степени. Оказывается, что достаточно проверять наличие делителей только среди чисел, не превышающих корня квадратного из  $n$  (действительно, если  $n = a \cdot b$ , то одно из чисел  $a$  или  $b$  не больше  $n^{1/2}$ ).

```
function prime(n:integer):boolean;
{n - простое число <=> значение функции - истина}
var
    k: integer;
begin
    prime:=true;
    for k:=2 to trunc(sqrt(n)) do
        if n mod k=0 then prime:=false
    end;
```

- Вариант 9. Еще раз оптимизируем функцию **prime**: прекращаем перебирать потенциальные делители  $n$ , как только найдем делитель.



```

function prime(n:integer):boolean;
{n - простое число <=> значение функции - истина}
var
    k: integer; p: boolean;
begin
    p:=true; k:=2;
    while p and (k<= trunc(sqrt(n))) do
        if n mod k=0 then p:=false else k:=k+1;
    prime:=p
end;

```

- Вариант 10. Оптимизируем функцию **gold** – прекращаем выполнять цикл, как только обнаружим, что число удовлетворяет гипотезе Гольдбаха.

```

function gold(n:integer):boolean;
{n удовлетворяет гипотезе Гольдбаха <=> значение
функции - истина}
var j: integer; g:boolean;
begin
    g:=false;
    j:=3;
    while (j<=n div 2) and not g do
        if prime(j) and prime(n-j) then
            begin
                writeln(n, '=', j,
                    '+', n-j);
                g:=true
            end
        else j:=j+2;
    gold:=g
end;

```

- Вариант 11. Пишем окончательную версию функции **prime**: учитываем, что  $n$  – нечетное число. Заметим, что в этой версии функция **prime** будет правильно работать только для нечетных чисел, а в предыдущих версиях такого ограничения не было.

```

function prime(n:integer):boolean;
{n - простое нечетное число <=> значение функции - истина}
var
    k: integer; p: boolean;
begin
    p:=true;
    k:=3;
    while p and (k<= trunc(sqrt(n))) do
        if n mod k=0 then p:=false else k:=k+2;
    prime:=p
end;

```

- Вариант 12. Окончательная версия программы: оптимизируем главную программу; убираем печать из функции **gold** и вставляем печать в главную программу.

```

function prime(n:integer):boolean;
{n - простое нечетное число <=> значение функции - истина}
var
    k: integer; p: boolean;
begin
    p:=true;
    k:=3;
    while p and (k<= trunc(sqrt(n))) do
        if n mod k=0 then p:=false else k:=k+2;
    prime:=p
end;
function gold(n:integer):boolean;
{n удовлетворяет гипотезе Гольдбаха <=> значение функции - истина}
var j: integer; g:boolean;
begin
    g:=false;
    j:=3;
    while (j<=n div 2) and not g do
        if prime(j) and prime(n-j) then g:=true
        else j:=j+2;
    gold:=g
end;

var m,i,h :integer;

```

```

begin
writeln('Введите натуральное m>=6');
readln(m);
h:=0;
i:=6;
while (i<m) and (h=0) do
    begin
        {проверяем гипотезу Гольдбаха для четного
         числа i}
        if not gold(i) then h:=i else i:=i+2;
    end;
if h=0 then writeln('Гипотеза Гольдбаха верна для всех
                    чисел <',m)
else writeln('Гипотеза Гольдбаха не выполнена для ',h)
end.

```

### 3.6 Стил ь программирования

В процессе разработки каждая программа не один раз подвергается исправлениям, и при этом человеку каждый раз приходится читать ее, разбираться в структуре, искать места, в которых могут возникнуть затруднения. Поэтому, чем легче они написаны, тем проще в них разобраться.

Начинающие программисты часто считают, что им нет необходимости специально заботиться о внятности программы: ведь они сами ее придумали и все в ней знают, зачем тратить время на какое-то оформление? Увы, это заблуждение, хотя и очень распространенное.

Как бы хорошо программист не понимал свою программу, есть предел, после которого невозможно удержать все детали. Если работа ведется с продолжительными перерывами, вспомнить подробности бывает не так-то просто. Довольно частая жизненная ситуация – программу начинает один человек, а доделывает другой.

Опытные программисты всегда уделяют внимание тому, как выглядит их программа. Красивую программу будет легко читать, понимать и исправлять, а значит, у нее больше шансов оказаться правильной и полезной.

Правила оформления программы обычно называют *стилем программирования*. Одни и те же действия почти всегда можно выполнить по-разному. У каждого программиста есть свои привычки и пристрастия, которые и образуют его стиль. Со временем у него обязательно вырабатывается свой неповторимый почерк, свой способ использования инструментов, по которому его трудно опознать, как по подписи на бумаге.

Стиль программирования может быть хорошим и плохим. Хороший стиль программирования способствует повышению надежности программ (*надежная программа* – программа, не содержащая ошибок).

Вот несколько рекомендаций, которые позволяют понять, что такое хороший стиль.

### **Кто ясно мыслит, тот ясно излагает!**

Прежде чем писать программу, надо четко уяснить, какая поставлена задача и как ее решить. Если в мыслях нет ясности – не помогут никакие приемы и ухищрения.

### **Не забывайте о комментариях!**

Все языки программирования позволяют включать в программу комментарии – пояснительный текст, который нужен только для человека. Компьютер не читает комментарии, он просто не замечает их. Комментарии – важнейший элемент стиля, это очень мощный инструмент, позволяющий сделать программу действительно понятной.

- У программы должно быть предисловие – большой вступительный комментарий, который может занимать несколько страниц текста. Он включает следующие разделы:
  - имя автора программы (творчество не должно быть анонимным!);
  - название программы и ее назначение;
  - по возможности подробное описание решаемой задачи;
  - описание исходных данных и требуемых результатов задачи;
  - описание основной идеи решения;
  - ссылки на литературу и другие источники, связанные с задачей и ее решением;

- дата написания первоначальной версии программы и история последующих изменений.
- Каждое описание переменных, констант должно сопровождаться комментарием, поясняющим их содержательную суть.
- Для каждой подпрограммы рекомендуется давать небольшой вступительный комментарий – для чего предназначена данная конкретная подпрограмма.
- Программы порой основаны на довольно сложных алгоритмах, понять которые по тексту не так-то легко. В этом случае хорошие комментарии, поясняющие смысл алгоритма в целом, могут оказаться очень полезными.
- Не доверять комментариям (этот совет предназначен не для тех, кто пишет, а для тех, кто будет читать программу).

Примеры плохих комментариев: «счетчик», «дополнительная переменная», «вещественное число», «строка». Примеры хороших комментариев: «количество найденных совпадений», «самое большое число в списке».

Вот типичный фрагмент программы:

**k:=k+1; { увеличим k на 1 }**

Комментарий абсолютно ничего не прибавляет к тексту программы: читатель и так видит, что k увеличивается на 1 (а если не видит, то здесь ему не поможет даже самый подробный комментарий). Желательно описать, почему происходит это увеличение. Например, «нашли очередное совпадение» или «переходим к следующему элементу».

Необходимо помнить, что компьютер всегда делает то, что написано в программе, а не в комментариях. Хороший комментарий рассказывает о замысле программиста, а анализ самой программы – о реализации задуманного.

Но, с другой стороны, подробный комментарий в иных случаях помогает заметить, что выполняются не те действия, о которых было заявлено.

**Понимание переменных – ключ к пониманию программы.**

Любая программа обрабатывает какую-то информацию, представленную в виде переменных. Поэтому необходимо в пер-

вую очередь понять смысл переменных, которые в ней используются.

Надо тщательно выбирать имена переменных. Следует избегать однобуквенных имен и конструкций типа «буква и цифра» – в них легко запутаться. Надо давать близкие имена родственным данным, но нельзя называть похожими именами данные, которые ничего общего не имеют. Как назовешь корабль, так он и поплывет!

- Используйте осмысленные имена. Плохо: x, xx, xxx, x1, ux и т.д.
- Избегайте похожих имен (X1, XI и Xl – три разных окончания: цифра один, буквы «ай» и «эль»; X0 и XO).
- Если в идентификаторах используете цифры, помещайте их в конце.
- Никогда не используйте в качестве определяемых идентификаторов стандартные идентификаторы.
- Не создавайте лишние промежуточные переменные.

### **Разделяй и властвуй!**

Разбивайте программу на подпрограммы. Объем подпрограммы должен быть таким, чтобы ее было несложно охватить как единое целое, понять сразу всю логику ее работы. Обычно подпрограмма без описаний и комментариев должна целиком помещаться на экране. Если экрана не хватает, есть смысл подумать о дальнейшем разбиении.

Текст должен быть удобным для чтения.

Для компьютера чаще всего неважно, как ваша программа разбита на строки и где в ней поставлены пробелы. Компилятор должен отделить лексемы друг от друга, а остальное ему безразлично.

Человек же, наоборот, обращает особое внимание на то, как оформлен текст, ведь от этого зависит его восприятие.

- Надо «выделять» алгоритмическую структуру.

Опытные программисты всегда пишут программу «лесенкой». Каждый раз, когда в ней используется алгоритмическая структура (цикл, ветвление), текст сдвигается вправо. Такая запись позволяет сразу увидеть, где начинается и где

заканчивается конструкция, какие операторы входят в нее, а какие нет.

- Смещение означает подчинение.

Данное правило – пояснительное расширение предыдущего. Вообще, всегда, когда надо показать, что один фрагмент подчинен другому, подчиненный фрагмент записывается со смещением вправо. Тем самым поясняют, что это не самостоятельное действие, а часть предыдущего.

- Не надо экономить на пробелах.

Они облегчают чтение и понимание программы. Если в одной строке записываются несколько операторов, необходимо поставить пробелы между ними. Во многих случаях желательно использовать пробелы при знаках действий.

### **Надо избегать волшебных чисел.**

Числа в программе не стоит использовать в явном виде. Когда в тексте программы встречается такое число (что-нибудь вроде 179), читателю часто бывает совершенно непонятно, что оно означает и откуда взялось.

Тяжело приходится программистам, когда возникает необходимость изменить такое число. Надо сделать кропотливую и не очень приятную работу – не пропустить все вхождения данного числа и не делать замену там, где данное число имеет другой логический смысл (например, число 100 может означать в одном месте количество элементов массива, а в другом месте – температуру кипения воды).

Используйте описание констант.

Конечно, такие числа, как 0 или 1, не стоит делать константами – это только усложнит программу.

### **Шаг за шагом.**

Известно, что проще всего разобраться в линейной программе, в которой все действия выполняются последовательно, одно за другим, как записаны.

Это не означает, что надо избегать циклов и ветвлений – без них невозможно построить ни один сложный алгоритм. Но лучше, чтобы конструкции были короткими, тогда каждая из них может рассматриваться как отдельное действие в общей линейной последовательности.

- Без goto.

Оператор goto делает программу чуть-чуть короче, нарушая последовательность выполнения. Однако в участках, куда совершается переход, может таиться опасность. Именно там чаще всего происходят ошибки. Причина очевидна: в одно и то же место программы можно попасть из разных мест с различной предысторией. Уследить за соблюдением всех необходимых условий становится очень трудно, отсюда и сложности.

- Не надо прерывать циклы.

Еще один способ нарушения последовательного выполнения – досрочное выполнение цикла. В языке Паскаль для этого есть специальная процедура break.

Цикл – самая сложная для понимания алгоритмическая структура. Обычно предполагается, что заголовок цикла полностью описывает условие его продолжения и завершения. Досрочное прекращение нарушает это правило, поэтому им лучше не пользоваться.

- Блок должен заканчиваться естественно.

Не надо вставлять в середину блока (программы или подпрограммы) операторы, ведущие к выходу из блока (exit) или прекращению работы (halt). Завершение должно быть естественным – подпрограмма обязана вернуть управление в точку вызова. Единственное исключение из этого правила – обнаружение фатальной ошибки, которая делает дальнейшую работу программы бессмысленной.

### **Использование логических величин.**

Начинающиеся программисты часто не понимают сути логических величин и создают громоздкие выражения там, где можно написать коротко и ясно.

Например, вместо **If a = true then ...** следует писать **If a then ...**. Логическая величина – это условие, не нуждающееся ни в каком дополнительном сравнении.

Еще пример: выражение **if a = true then a := false else a := true** преобразуем в **a := not a**.



### **Изучите и активно используйте возможности языка.**

В результате программы становятся короче и исключаются определенные ошибки. *Избегайте трюков*, например, использование параметров цикла после окончания цикла `for` (неизвестно, какое значение имеет параметр цикла после его окончания).

### **Не гонитесь за микроэффективностью.**

Настоящая эффективность программы достигается за счет изменения алгоритма.

- Игнорируете все предложения по повышению эффективности, пока программа не будет правильной.
- Не жертвуйте легкостью чтения ради эффективности.
- Добивайтесь эффективности на основе измерений, а не догадок.

## **3.7 Доказательное программирование**

*Доказательное программирование* – синтез программ на основе спецификаций пред- и постусловий задачи, сопровождаемый логическим рассуждением, которое подтверждает правило каждого шага синтеза.

Тезис: *"Невозможно программировать иначе, нежели по дисциплине, подсказываемой логическими правилами синтеза программ"*.

С доказательным программированием можно познакомиться по следующим книгам:

Интеллектуальное откровение

– Э. Дейкстра Дисциплина программирования, М., Мир, 1976 г.  
Апостольское деяние, направленное на то, чтобы превратить учение одиночки в мировую религию

– Д. Грис Наука программирования, М., Мир, 1984 г.

## 4 РЕГУЛЯРНЫЕ ТИПЫ (МАССИВЫ)

### 4.1 Определение регулярного типа

Рассмотренные простые типы определяют различные множества атомарных (неразделимых) значений.

Составные типы задают множества «сложных» значений; каждое значение из такого множества образует некоторую структуру (совокупность) нескольких значений другого типа.

Мы вводим вычислительную структуру, использующую абстрактное понятие «конечная последовательность».

*Каждое значение регулярного типа (массива) состоит из фиксированного числа элементов одного и того же базового типа (т.е. значение содержит фиксированное число однотипных компонент).*

Способ образования позволяет обозначать значения этих типов одним групповым именем, а доступ к отдельным элементам массивов посредством указания этого группового имени и порядкового номера (индекса) необходимого элемента.

Для корректного определения регулярного типа необходимо задать две характеристики:

- *тип элементов массива,*
- *количество и «способ нумерации» элементов.*

**type**

**<идентификатор - тип массива>= array[T1] of T2;**

**T1 - тип индекса:**

**ограниченный,**

**литерный,**

**перечислимый (см. дальше);**

**T2 - тип элементов массива (любой тип) .**

Примеры:

```
const n=20;
```

```
type
```

```
vector=array[1..100] of real;
```

```
m=array[char] of boolean;
```

```
matrix=array[1..n] of array[1..n] of integer;
```

```
var
```

```

v:vector;
SymTab:m;
arr1,arr2:matrix;
s:array['a'..'z'] of boolean;

```

Мы можем «нумеровать» элементы массива не только целыми числами, но и значениями произвольного дискретного типа (типа, на котором действуют функции **ord**, **succ**, **pred**).

В том случае, когда определен тип массива, элементами которого являются снова массивы, можно использовать эквивалентное определение *двухмерного* массива.

Следующие два определения эквивалентны:

```

v:array[1..10] of array[1..20] of integer;
v:array[1..10,1..20] of integer;

```

Число индексов (*размерность массива*) не ограничена.

Синтаксис:

```

<регулярный тип> ::=
    ---->array-->[----><тип индекса>---->]---->of--
><тип>---->
                                |               |
                                <-----, ---<---

```

Единственное возможное действие над массивом в целом – это присваивание:

```

v1:=v2;

```

Типы обоих массивов в данном случае должны совпадать.

Доступ к элементам массива:

```

<идентификатор массива>[<индекс>]
или
<идентификатор массива>[<список индексов через
запятую>]

```

В качестве индексов – произвольные выражения, тип которых должен совпадать с типом индекса.

Примеры:

```
v[1]
v[(i+2)*6]
```

Пусть

```
v2:array[1..10] of array[5..20] of integer
```

– двухмерный массив, тогда  $v2[k]$  –  $k$ -ый массив в группе из 10 массивов целых чисел,  $v2[k][5]$  – 5-ый элемент этого массива, тот же элемент получаем и так  $v[k, 5]$ .

Элемент массива считается **переменной**:

- он может получать значения (например, в операторе присваивания);
- он может участвовать в выражениях там, где может присутствовать переменная данного типа.

Ассортимент операций над элементами массива **полностью определяется** типом этих элементов.

Примеры:

```
v2[i,j]:=v2[i,j-1]+1;
SymTab['z']:=not SymTab['a'];
```

**Ошибки в работе с массивами:**

*Выход индекса за допустимые пределы*

```
var
  v:array[0..10] of real;
....
v[11]:=0.5;    <==== транслятор обнаружит ошибку

i:=11;
v[i]:=0.5;    <==== эта ошибка может быть обнаружена
только во время исполнения и то, если осуществляется
контроль диапазона, иначе это может привести к
непредсказуемым последствиям
```

## 4.2 Примеры программ для работы с массивами

Задача 1. Ввести последовательность из  $n$  чисел и распечатать ее с конца.

```
var
  i:integer;
  s:array[1..100] of integer;
begin
  writeln('Введите количество чисел');
  readln(n);
  writeln('Введите ',n,' чисел');
  for i:=1 to n do
    read(s[i]); {ввод потоком}

  writeln;
  writeln('Вывод:');
  for i:=n downto 1 do write(s[i],' ');
end.
```

Задача 2. Ввести матрицу построчно.

```
const n=4; m=5;
var s:array[1..n,1..m] of integer;
    i,j:integer;
.....
writeln('введите матрицу ',n,' - строк', m,' - столбцов);
for i:=1 to n do
  begin
    for j:=1 to m do
      read(s[i,j]); {Ввод потоком}

    writeln
  end;
```

Задача 3. Напечатать матрицу построчно.

```
for i:=1 to n do
  begin
    for j:=1 to m do
      write(s[i,j],' ');

    writeln
  end;
```

Задача 4. Вычислить максимальный элемент матрицы и его индексы.

```
const n=4;m=5;
type matrix=array[1..n,1..m] of real;
var a:matrix;
    max:real; in,jm:integer;
    i,j:integer;
.....
in:=1;jm:=1;max:=a[1,1];
for i:=1 to n do
for j:=1 to m do
    if a[i,j]>max then begin
                                max:=a[i,j];
                                in:=i;
                                jm:=j;
                                end;
```

Задача 5. В матрице  $a$  изменить порядок строк на противоположный.

```
const n=4;m=5;
type
    stroka=array[1..m] of real;
    matrix=array[1..n] of stroka;
var a:matrix);
    i:1..n;
    x:stroka;
.....
for i:=1 to n div 2 do
    begin
        x:=a[i];
        a[i]:=a[n+1-i];
        a[n+1-i]:=x
    end;
```

Задача 6. Коэффициенты многочлена лежат в массиве  $a: \text{array}[0.. n] \text{ of integer}$  ( $n$  – натуральное число, степень многочлена). Вычислить значение этого многочлена в точке  $x$ , т.е. найти  $a_n x^n + \dots + a_1 x + a_0$ .  
*Решение.* (Описываемый алгоритм называется схемой Горнера.)

```

k:=0; y:=a[n];
{инвариант: 0<=k<=n, y = a[n] x^k+ a[n-1] x^(k-1)+
...+ a[n-k] x^0}
while k <>n do
  begin
    k:=k+1;
    y:=y*x +a[n-k]
  end;

```

Задача 7. Произведение многочленов. В массивах **a: array[0..k] of integer** и **b: array[0.. n] of integer** хранятся коэффициенты двух многочленов степеней  $k$  и  $n$ . Поместить в массив **c: array[0.. m] of integer** коэффициенты их произведения. (Числа  $k, n, m$  – натуральные,  $m=n+k$ ; элемент массива с индексом  $i$  содержит коэффициент при степени  $i$ .)

*Решение.* Воспользуемся математическим свойством

$$c_t = \sum_{i+j=t} a_i b_j.$$

```

for i:=0 to m do
  c[i]:=0;
for i:=0 to k do
  for j:=0 to n do
    c[i+j]:=c[i+j]+a[i]*b[j];

```

Задача 8. Двоичный поиск.

Даны последовательность  $x_1 \leq x_2 \leq \dots \leq x_n$  и число  $a$ . Выяснить, содержится ли  $a$  в этой последовательности, т.е. существует ли такое  $i$ ,  $1 \leq i \leq n$ , что  $x_i = a$ . (Количество действий должно быть порядка  $\log n$ .)

*Решение.* (Предполагаем, что  $n>0$ .)

```

left:=1; right:=n+1;
{right>left, если a есть вообще, т.е. и среди
x[left]..x[right-1]}
while right-left <> 1 do
  begin
    m:=left+(right-left) div 2;
    {left < m < right}
    if x[m]<=a then left:=m
      else right:=m;
  end;

```

(Обратите внимание, что и в случае  $x[m]=a$  инвариант не нарушается.)

Каждый раз  $right-left$  уменьшается приблизительно вдвое, откуда и вытекает требуемая оценка числа действий.

Замечание:  $left+(right-left) \text{ div } 2 = (right+left) \text{ div } 2$ .

### 4.3 Символьные массивы

*Символьные массивы* – одномерные массивы, состоящие из элементов символьного типа и индекс элементов принадлежит ограниченному целому типу.

Например:

```
var s:array[1..26] of char;
```

*Дополнительное средство для работы* – изображение конкретного значения символьного массива в виде строки

```
s:='Пример символьного массива';
```

Это изображение может использоваться в операторах присваивания, в описаниях констант и в процедуре **write**.



## 5 ПОДПРОГРАММЫ

### 5.1 Понятие подпрограммы

Представление программы как совокупности (иерархии) относительно обособленных фрагментов со строгими и явно определенными интерфейсами *способствует* большей ясности и модифицируемости программы, легко проверяемой, и, как следствие, *влечет* повышение качества и эффективности программы.

Современный подход к разработке программ поощряет явное оформление в виде подпрограммы любого достаточно самостоятельного и законченного программного фрагмента.

**Подпрограмма** – обособленная именованная часть программы со своим собственным *локальным контекстом имен*.

**Вызов подпрограммы** – выполнение действий, заданных в подпрограмме, может быть произведен в некоторой точке программы посредством указания имени этой подпрограммы.

Понятие подпрограммы – элементарное средство **повышения «уровня»** языка.

Сосредоточив в одном месте программы подробное описание некоторых «технических» аспектов, мы получаем следующую возможность: в остальной программе достаточно указывать имена этих действий («операций», введенных подпрограммами), не конкретизируя семантику.

Подпрограмма содержит описание некоторой совокупности локальных объектов: констант, типов, переменных и т.п. Эти объекты предназначены для организации действий в подпрограмме и *имеют смысл* («доступны», «видимы») только внутри подпрограммы.

Подпрограмма может быть разработана для различных случаев применения. Следовательно, перед выполнением подпрограммы можно передать ей некоторую информацию из точки вызова («настроив» ее соответствующим образом). Это осуществляется с помощью понятия *параметров подпрограммы*, и приводит к следующему:

- к большей гибкости и универсальности подпрограммы,

- к уменьшению взаимной зависимости подпрограммы и точки вызова.

Примеры подпрограмм – функции и стандартные процедуры чтения и записи.

## 5.2 Общая структура подпрограмм

Структура подпрограммы почти буквально повторяет структуру всей Паскаль-программы – «часть подобна целому», «регулярный характер построения языка».

При описании подпрограммы в общем случае необходимо задать три основных компоненты:

- интерфейс подпрограммы, т.е. информацию, необходимую для ее вызова (активизации);
- локальный контекст подпрограммы: совокупность описаний (рабочих) объектов, с которыми осуществляются действия;
- собственно действия (операторы), составляющие смысл подпрограммы.

Интерфейс подпрограммы или, иными словами, та информация, которая «представляет» подпрограмму и достаточна для корректного ее вызова, сосредоточена в заголовке.

Для уточнения двух остальных компонент подпрограммы нам понадобится определение такой синтаксической категории, как блок.

Синтаксически любая программа выглядит следующим образом

```
<необязательный заголовок программы><блок>.
<необязательный заголовок программы>::=
    program <идентификатор>;
<блок>::=
    <описания типов, переменных, подпрограмм и т.п.>
    begin <операторы> end
```

Описание локальных объектов и операторы (алгоритм) подпрограммы составляют внутреннюю ее часть и, как правило, имеют синтаксис блока. Можно сказать, что *заголовок* содержит

информацию о том, ЧТО делает подпрограмма, а *тело подпрограммы* (блок) описывает, КАК она это делает.

Два вида подпрограмм – **процедуры и функции** – одинаковый смысл и аналогичная структура, но различие в назначении и способе их использования.

**Функции** служат для определения алгоритма вычисления нового значения некоторого простого типа, и вызов функции должен быть операндом в выражении.

**Процедуры** служат для задания совокупности действий, направленных на изменение внешней по отношению к ним программной обстановки и, как следствие, определение новых значений переменных в программе. Вызов процедуры играет роль оператора.

## Синтаксические диаграммы

<описание процедуры> ::=

---><заголовок процедуры>-->;--><тело процедуры>-->;-->

<описание функции> ::=

---><заголовок функции>-->;--><тело функции>-->;-->

<заголовок процедуры> ::=

--->procedure--><идентификатор>--><формальных параметров>--->  
|----->-----|

<заголовок функции> ::=

function--><идентификатор>--><формальных параметров> -->:<тип>  
|----->-----|

### 5.3 Тело подпрограммы. Области действия имен

Пример:

```

var a,b:integer;
procedure out(a,c:real);
  var b:real;
      e,f:integer;
      procedure into;
        type digits='0'..'9';
        var c:digits;
        begin
          {Здесь доступны только:
           целые e и f, формальный параметр a процедуры out,
           вещественная переменная b, переменная c типа digits}
          .....
        end;
      begin
        {Здесь доступны только:
         вещественная переменная b, формальные вещественные параметры
         a и c, целые переменные e и f}
        .....
      end;
begin
{Здесь доступны только целые переменные a и b}
.....
end.

```

Тело подпрограммы есть блок (в примере схематично представлены три вложенных блока).

Имена объектов, описанных в блоке подпрограммы, считаются известными в пределах данного блока. Это же относится и к именам формальных параметров.

Формальные параметры и объекты, описанные в блоке, образуют *собственный локальный контекст* данного блока.

В описании блока могут содержаться описания процедур и функций, следовательно, возможны *вложенные подпрограммы*. Правила для доступа к объектам, описанным в разных блоках (**правила видимости**):

- Имена объектов, описанных в блоке, считаются известными в пределах данного блока, включая и все вложенные блоки.
- Имена объектов, описанных в блоке, должны быть уникальными в пределах данного блока и могут совпадать с именами объектов в других блоках.

- Если в некотором блоке описан объект, имя которого совпадает с именем объекта, описанного в объемлющем блоке, то это последнее имя становится недоступным в данном блоке. Говорят, что имя, описанное в блоке, **экранирует (закрывает, делает недоступным)** одноименные объекты из блоков, объемлющих данный.

**Локальный контекст имен** – локальные объекты блока вместе с видимыми в нем объектами объемлющих блоков.

**Глобальный контекст имен** – совокупность объектов, описанных в самом внешнем блоке.

## 5.4 Использование процедур и функций

Разбиение программ на процедуры и функции повышает качество и эффективность программирования. Важно владеть различными приемами использования подпрограмм.

**Задача.** Дан четырехугольник с вершинами  $A, B, C$  и  $D$ . Известны длины сторон  $AB, AD, BC, CD$  и длина диагонали  $BD$ . Требуется найти площадь четырехугольника.

Приведем различные примеры подпрограмм для решения этой задачи.

- Приведенная ниже подпрограмма без параметров не выдерживает никакой критики, единственное ее достоинство в том, что с помощью процедуры **str1** мы явно выделяем похожие действия при вычислении площади треугольника.

```
var
  AB, BC, CD, AD, BD: real;
  S1, S, a, b, c, p: real;
procedure str1;
  {Вычисление площади треугольника со сторонами a, b
и c. Длины этих сторон заданы в виде значений гло-
бальных переменных.}
  begin
    p := (a+b+c) / 2;
    S := sqrt(p * (p-a) * (p-b) * (p-c))
  end;
begin
  read(AB, BC, CD, AD, BD);
```

```

a:=AB;b:=AD;c:=BD;
str1;
S1:=S;
a:=BC;b:=CD;c:=BD;
str1;
S1:=S1+S;
writeln('площадь=',S1)
end.

```

- Приведенная ниже процедуры для вычисления площади треугольника (теперь она называется **str2**) основывается на наблюдении, что переменная **p** в главной программе не используется и поэтому может быть сделана локальной. По-прежнему предложенная процедура обладает существенным недостатком – слишком сложно ее вызывать.

```

procedure str2;
{Вычисление площади треугольника со сторонами a, b
и c. Длины этих сторон заданы в виде значений гло-
бальных переменных.}
var p:real;
begin
p:=(a+b+c)/2;
S:=sqrt(p*(p-a)*(p-b)*(p-c))
end;

```

Из описания переменных в главной программе можно удалить переменную **p**, но можно и оставить (в последнем случае она не будет иметь какой-нибудь пользы).

- Процедура с параметрами значительно облегчает ее вызов.

```

var
AB,BC,CD,AD,BD:real;
S1,S2:real;
procedure str3(a,b,c:real;var S:real);
{Вычисление площади S треугольника со сторонами a,
b и c}
{a,b,c,S - формальные параметры}
var p:real;
begin

```

```

    p:=(a+b+c)/2;
    S:=sqrt(p*(p-a)*(p-b)*(p-c))
    end;
begin
    read(AB,BC,CD,AD,BD);
    str3(AB,AD,BD,S1); {AB,AD,BD,S1 - фактические
    параметры}
    str3(BC,CD,BD,S2); {BC,CD,BD,S2 - фактические
    параметры}
    writeln('площадь=',S1+S2)
end.

```

- Выходной параметр **S** у процедуры **str3** имеет вещественный тип, поэтому лучшим решением является использование не процедуры, а функции.

```

var
    AB,BC,CD,AD,BD:real;
function str4(a,b,c:real):real;
{функция вычисляет площадь треугольника со
    сторонами a, b и c}
var p:real;
    begin
        p:=(a+b+c)/2;
        S:=sqrt(p*(p-a)*(p-b)*(p-c))
        end;
begin
    read(AB,BC,CD,AD,BD);

writeln('площадь=',str4(AB,AD,BD)+str4(BC,CD,BD));
end.

```

## 5.5 Механизм параметров

```

<список формальных параметров> ::=
  ---> (---> <описание параметров> --->) --->
      |
      |-----<--- ; ---<----->

<описание параметров> ::=
  -----> -----<идентификатор> -----> : --> <идентификатор типа> -->
      |           |           |           |
      --> var --> <----- , ---<----->

```

Имеются два различных способа передачи фактических параметров в подпрограммы. Если в описании формального параметра ему предшествует **var**, то этот способ называется *передачей параметра по ссылке*, а сам параметр называется *параметром-переменной*. Если **var** отсутствует, то такой параметр называется *параметром-значением*, а способ – *передачей параметра по значению*.

### Параметры-значения

Формальный параметр в этом случае считается обычной локальной переменной подпрограммы. Фактический параметр может быть произвольным выражением того же типа, что и формальный параметр.

Внутри подпрограммы возможны произвольные действия с данным формальным параметром, но любые изменения его значения никак *не отражаются* на значениях переменных вне подпрограммы.

Пример:

```

procedure SumSquare (x,y:real);
begin
  x:=x*x;
  y:=y*y
  writeln('Сумма квадратов = ',x+y)
end;

var
  a,b:real;x,y:real;
begin

```



```

SumSquare (2,3) ;           =====>    13
a:=0;b:=1;
SumSquare (a,b) ;           =====>     1
x:=2;y:=3;
SumSquare (x,y) ;           =====>    13
writeln (x, '  ',y) ;       =====>    2 3
SumSquare (x*y,x+y) ;       =====>    61
end.

```

### Параметры-переменные

Параметры, передаваемые по ссылке, *используются для передачи некоторых значений, вычисленных в подпрограмме, наружу в программу.*

Пример – сумма и разность квадратов двух чисел:

```

procedure Draft1 (x,y:real) ;
var
  Sum,Sub:real;
begin
  Sum:=x*x+y*y;
  Sub:=x*x-y*y
end;

```

Локальные переменные **Sub**, **Sum** известны только в пределах текущего блока, после выполнения процедуры исчезают.

```

procedure Draft2 (x,y:real;Sum,Sub:real) ;
begin
  Sum:=x*x+y*y; Sub:=x*x-y*y
end;

```

Что произойдет?

```

var a,b:real;
    SumAB,SubAB:real;
begin
  a:=2;b:=3;
  Draft2 (a,b,SumAB,SubAB) ;
  . . . . .
end.

```

Присваивание любых значений **Sum** и **Sub** не приведет к получению этих значений переменными **SumAB** и **SubAB**.

*При передаче параметра по ссылке изменение в теле подпрограммы формального параметра приводит к аналогичному изменению фактического параметра.*

```

procedure SumSub(x,y:real; var Sum,Sub:real) ;
begin
    Sum:=x*x+y*y;
    Sub:=x*x-y*y
end;

```

Здесь формальные параметры **Sum** и **Sub** считаются **синонимами** соответствующих фактических параметров (указывают на одно и то же место в памяти).

Фактические параметры при передаче по ссылке должны быть только *переменными* того же типа, что и формальные параметры.

Пример:

```

procedure swap(var x,y:real) ;
{обмен значениями}
var t:real;
begin
    t:=x;
    x:=y;
    y:=t
end;
.....
swap(A,B) ;

```

Пример:

```

var
    a,b:integer;
procedure h(x:integer; var y:integer) ;
    begin
        x:=x+1; y:=y+1;
        writeln(x, ' ', y) ;
    end;

begin
    a:=0; b:=0;
    h(a,b) ;

```

=====> печать: 1 1

```
writeln(a, ' ', b)      =====> печать: 0 1
end.
```

Умелое использование процедур с параметрами-переменными приводит к прекрасным результатам. В качестве примера приведем решение следующей задачи.

*Пусть процедура  $\text{reduction}(a,b,p,q)$  от целых параметров ( $b \neq 0$ ) приводит дробь  $a/b$  к несократимому виду  $p/q$ . Описать данную процедуру и использовать ее для приведения дроби  $1+1/2+1/3+\dots+1/20$  к несократимому виду.*

```
procedure reduction(a,b:integer;var p,q:integer);
var
  a1,b1,f:integer;

begin
  a1:=abs(a);b1:=abs(b);
  while (a1<>0) and (b1<>0) do
    if a1>b1 then a1:=a1 mod b1
      else b1:=b1 mod a1;
  if a1=0 then f:=b1 else f:=a1;
  p:=a div f; q:=b div f;
end;
```

```
.....
{следующий фрагмент главной программы приводит дробь
1+1/2+1/3+...+1/20 к несократимому виду}
c:=0; d:=1; {c/d=0}
for i:=1 to 20 do
  reduction(c*i+d,i*d,c,d); {c/d+1/i=(ci+d)/(di)}
```

## 5.6 Предварительное описание

Порядок описания подпрограмм накладывает ограничение на вызов процедур.

```
program p;
  <описание процедуры А>      {здесь не может быть
                                обращения к процедуре В}
  <описание процедуры В>      {здесь может быть обращение
                                к процедуре А}
  <тело главной программы>
```

Но возможно предварительное описание:

```
program p;
  procedure B<список формальных параметров>; forward;
  <описание процедуры A> {здесь может быть
                           обращения к процедуре B}
  <описание процедуры B>
<тело главной программы>
```

**Предварительное описание** состоит из заголовка процедуры, потом вместо тела пишется идентификатор **forward**. Потом обязательно полное (определяющее) описание. Предварительное описание необходимо при взаимной рекурсии.

## 5.7 Побочный эффект

В императивных языках (в том числе и в Паскале) функции могут ссылаться на глобальные данные и разрешается применять деструктивное (разрушающее) присваивание, что может привести к изменению значения функции при повторном ее вызове. Такие динамические изменения в величине данных часто именуются *побочными эффектами*. Благодаря им значение функции может изменяться, даже если ее аргументы и остаются без изменения всякий раз, когда к ней обращаются.

Пример побочного эффекта:

```
var flag: boolean;

function f (n:integer):integer;
begin
  if flag then f:=n
                else f:= 2*n;
  flag:=not flag
end;
```

```

begin
    flag:=true;
    writeln(f(1)+f(2));      {будет напечатано 5}
    writeln(f(2)+f(1));      {будет напечатано 4}
end.

```

В функции **f** имеется оператор присваивания, изменяющий значение переменной **flag**, описанной в объемлющем блоке, и поэтому значение суммы, использующей вызов функции **f**, зависит от порядка следования слагаемых. Функция **f** не является математической («чистой») функцией. Операции, подобные этой, в математике не разрешены, поскольку математические рассуждения базируются на идее равенства и возможности замены одного выражения другим, означающим то же самое, т.е. определяющим ту же величину.

Побочный эффект приводит к трудноуловимым программным ошибкам и, следовательно, крайне нежелателен.

## 5.8 Распределение памяти для переменных

Все глобальные переменные распределяются в памяти **статически**. Локальные переменные размещаются в памяти **динамически**, при активизации подпрограммы, их содержащей; после выполнения подпрограммы память, отведенная для локальных переменных, освобождается.

В некоторый момент выполнения программы в памяти могут присутствовать несколько групп локальных переменных, соответствующих цепочке вызванных и незавершенных подпрограмм.

В случае очередного вызова новая область памяти связывается с последней группой в цепочке; при завершении подпрограммы ее локальные переменные удаляются из хвоста цепочки. Этот принцип отведения и освобождения памяти соответствует понятию **стека**. Слишком длинная цепочка вызовов – возникает ошибка «*переполнения стека*».

## 6 СТРОКОВЫЙ ТИП

### 6.1 Определение строкового типа

Строковый тип обобщает понятие символьных массивов, позволяя динамически менять длину строки.

***Строковый тип** данных определяет множество символьных цепочек произвольной длины от нуля символов до заданного их числа.*

<описание строкового типа>::=

string | string[<максимальная длина строки>]

Максимальная длина строки, если не указана, то подразумевается равной 255, иначе может быть любое целое от 0 до 255.

Пример:

```
type
  line=string[80];
var
  myLine:line;
  myLineShort:string[10];
```

Описание

```
Var St: string[80];
```

резервирует для **St** 81 байт памяти. Если учесть, что на каждый символ отводится один байт, возникает естественный вопрос, а откуда взялся еще один байт, 81-й, а точнее, нулевой байт? Дело в том, что после того как переменной **St** присвоено какое-то значение, нулевой байт содержит фактическую длину строки **St**. Длина строки в байтах при этом равна значению кода символа, находящегося в **St[0]**.

Например, присваивание

```
St := 'abcdefgh';
```

дает такой результат:

```
St[0] = Chr(8) .
St[1] = 'a' .
...
St[8] = 'h' .
```

Фактическая длина строковой переменной `St` равна `Ord(St[0])`. Значение 255 для верхнего предела длины строки объясняется тем, что одиночный байт может принимать 256 различных значений от 0 до 255.

Изображение строки такое же, как для символьных массивов, может использоваться

- в операторах присваивания,
- как фактические параметры,
- в описании констант.

*Символьный массив, в отличие от строки, может быть очень большим!*

## 6.2 Строковые операции

Операция **конкатенации** (сцепления) применяется для соединения (сцепления) нескольких строк в одну результирующую строку. Эта операция является бинарной ассоциативной операцией и обозначается знаком `+`. Например, выражение `'Т' + 'у' + 'р' + 'бо' + 'паскаль'` дает `'Турбо паскаль'`.

Длина результирующей строки не должна превышать 255.

Пример:

```
myLine:='Короткая строка'; {длина 15}
myLine:=myLine+' стала длинее'; {длина 28}
myLine:=' '; {длина 0}
```

Операции **сравнения** `=, <>, >, <, >=, <=`

При сравнении строк используется *лексикографический* порядок:

а) сравнение строк производится с первых символов (слева направо) до первого несовпадающего символа, и та строка считается больше, в которой первый несовпадающий символ больше;

б) если строки имеют различную длину и одна строка полностью входит во вторую, то более короткая строка меньше, чем более длинная.

Пример:

```
'abcd' >= 'abc'
'abcde' > 'aacde' > 'aacdd'
```

Элементы строки нумеруются целыми числами, начиная с 1.

### Доступ к отдельным элементам строки

`myLine[<выражения типа 1..255>]`

Элемент строки принадлежит типу `char`.

**Выражения строкового типа** – выражения, в которых операндами служат строковые данные:

- строковые константы,
- строковые переменные,
- функции со строковым значением,
- литерные выражения.

Строковые выражения могут использоваться там, где используется строковый тип. *В операторах присваивания, если значение переменной после выполнения присваивания превышает по длине максимально допустимую величину, то все лишние символы отбрасываются!*

		значение
<code>a:string[6]</code>	<code>a:='группа 1';</code>	<code>'группа'</code>
<code>a:string[8]</code>	<code>a:='группа 1';</code>	<code>'группа 1'</code>
<code>a:string[2]</code>	<code>a:='группа 1';</code>	<code>'гр'</code>

Стандартная функция **Length(<выражение строкового типа>)**. Значение функции есть текущая длина строки – фактического параметра.

Пример:

```
var myLine:string;
    i:integer;
    ....
for i:=1 to Length(myLine) do
    myLine[i]:=chr(ord(myline[i])+1);
```

*Нет полной идентичности между строковыми типами и символьными массивами, поэтому возможна ошибка при работе с элементами строки без учета ее текущей длины.*

Пример:

```
var
    str:string[26];
    i:integer;
begin
```



```

str:='A';
for i:=1 to 26 do
  str[i]:=chr(ord('A')+i-1);
writeln(str)
end.

```

Предполагается, что данная программа сформирует строку из 26 букв латинского алфавита, но печать **writeln(str)** дает просто A! Объяснение: присваивание значений элементам строки не влияет на ее текущую длину (здесь текущая длина = 1).

Правильная программа:

```

var
  str:string[26];
  i:integer;
begin
  str:='';
  for i:=1 to 26 do
    str:=str+chr(ord('A')+i-1);
  writeln(str)
end.

```

### 6.3 Стандартные процедуры и функции для работы со строковыми типами

Процедура

**delete (var St:string; Poz:integer; N:integer)**

производит удаление N символов строки St, начиная с позиции Poz. Если Poz>255, то осуществляется программное прерывание.

```

St:='abcdef';
delete(St,4,2);
writeln(St); ====> abcf
St:='река Волга';
delete(St,1,5);
writeln(St); ====> Волга

```

Процедура

**insert(Source:string;var S:string;Index:integer)**

осуществляет вставку строки Source в строку S, начиная с позиции Index.

```
S:='Золотойключик' ;  
insert(' ',S,8) ;  
writeln(S) ; =====> Золотой ключик
```

Функция

**copy(S:string;Index:integer;Count:integer):string**

Функция выделяет из строки S подстроку длиной Count символов, начиная с позиции Index.

Если Index > length(S), то возвращается пустая строка.

Если Count+Index > length(S), то возвращается конец строки.

Если Index > 255, то диагностируется ошибка при выполнении.

```
copy('abcdefg',2,3)='bcd'  
copy('abcdefg',4,10)='defg'
```

Функция

**concat(S1,S2,...,SN:string):string**

производит соединение последовательности строк, эта функция равносильна операции конкатенации +.

Функция

**pos(Substr:string;S:string):integer**

обнаруживает первое появление в строке S подстроки Substr. Результат равен номеру той позиции, где начинается подстрока Substr. Если подстроки нет, то результат равен 0.

```
pos('de','abcdef')=4  
pos('z','abcdef')=0
```

Имеются две процедуры преобразования числовых значений в строковые и наоборот: Str и Val. Процедура Str при обращении к ней вида

```
Str(num, strnum) ;
```

где num – значение числового типа, а strnum – переменная строкового типа, присваивает переменной strnum строковое значение, представляющее собой символьное изображение значения переменной num. В первом параметре функции Str можно использо-

вать спецификаторы формата (такие же, как для оператора write). Процедура Val выполняет обратное преобразование. Обращение к ней имеет вид:

```
Val(strnum, num, errcode);
```

Третий параметр в этой процедуре равен нулю при успешном выполнении преобразования. В том случае, когда первый параметр содержит символы, недопустимые при записи числа, значение параметра errcode равно номеру позиции с ошибочно заданным символом.

Пример использования процедур Str и Val дается ниже:

```
var
  i, errcode: Integer;
  S: String;
begin
  Str(2000,S);
  Writeln('Строковое значение ',S);
  Readln;
  Val(S, i, errcode);
  If errcode <> 0 then
    Writeln('Ошибка ввода в позиции: ', errcode)
  else
    Writeln('Числовое значение = ', i);
  Readln;
End.
```

В этом примере вначале выполняется преобразование целочисленного значения 2000 в строковое (обращение к процедуре Str), а затем, наоборот, строка символов «2000» преобразуется в значение типа Integer (процедура Val).

## 7 ТЕСТИРОВАНИЕ И ОТЛАДКА

### 7.1 Тестирование

*Тестирование* – процесс выполнения программы с намерением найти ошибки. Если ваша цель – показать отсутствие ошибок, вы их найдете не слишком много. Если же ваша цель – показать наличие ошибок, вы найдете значительную их часть.

Два крайних подхода к стратегии тестирования:

- **программа – "черный ящик"**, идеальная стратегия – проверить всевозможные комбинации значений на входе и выходе;
- **главное – логика программы**, идеальная стратегия – проверить каждый путь, каждую ветвь алгоритма.

Искусство тестирования заключается в искусстве отбора тестов с максимальной отдачей.

#### Миф о тестировании путей

Выполнение всех путей не гарантирует соответствия программы с спецификациям, потому что

- сделана правильная программа, но для других целей;
- отсутствуют необходимые пути (например, нет контроля входных данных);
- программа чувствительна к данным (например, для проверки на равенство  $A$ ,  $B$  и  $C$  используется соотношение  $(A+B+C)/3=A$ ).

В соответствии с проектированием возможно *восходящее* и *нисходящее* тестирование.

#### 10 аксиом тестирования

1. Хорош тот тест, для которого высока вероятность обнаружения ошибки, а не тот, что демонстрирует правильную работу программы.
2. Одна из самых сложных проблем при тестировании – решить, когда нужно закончить.
3. Невозможно тестировать свою собственную программу.
4. Необходимая часть всякого теста – описание выходных данных или результатов.

5. Избегайте невоспроизводимых тестов, не тестируйте «с лету».
6. Готовьте тесты как для правильных, так и для неправильных входных данных.
7. Детально изучайте результаты каждого теста.
8. По мере того как число ошибок, обнаруженных в программе увеличивается, растет также и относительная вероятность существования в ней необнаруженных ошибок.
9. Поручайте тестирование самым способным программистам.
10. Никогда не изменяйте программу, чтобы облегчить ее тестирование.

Процесс разработки тестов:

1. Исходя из спецификаций (описания программы как «черного ящика») подготовьте тест для каждой ситуации допустимых и недопустимых входных данных.

2. Проверьте текст программы, чтобы убедиться, что все условные переходы будут выполнены в каждом направлении. Если необходимо, добавьте соответствующие тесты.

3. Убедитесь по тексту, что тесты охватывают достаточно много возможных путей. Например, каждый цикл должен быть выполнен 0, 1 и максимальное число раз.

4. Проверьте по тексту программы ее чувствительность к отдельным особым значениям входных данных.

Пример набора тестов для программы, решающей квадратное уравнение  $ax^2 + bx + c = 0$ :

- $a=0, b=0, c=0$  (уравнение  $0=0$  не может быть разрешено относительно  $x$ );
- $a=0, b=0, c=10$  (уравнение  $0=10$  – тест на ошибочные входные данные);
- $a=0, b=5, c=17$  ( $5x+17=0$  – не квадратное уравнение, может быть деление на 0);
- $a=6, b=1, c=-2$  («нормальный» тест);
- $a=3, b=7, c=0$  («нормальный» тест, один из корней равен 0);
- $a=3, b=2, c=5$  (комплексные корни);
- $a=7, b=0, c=0$  (может ли извлекаться корень из 0?).

## 7.2 Отладка

*Отладка* – деятельность, направленная на установление точной природы известной ошибки, а затем на исправление этой ошибки.

Результаты тестирования являются исходными данными для отладки. Последовательность событий:

симптом (ошибки) → причина (ошибки) → исправление (ошибки).

Как искать ошибку?

1. Поймите ошибку. Какие данные правильные, какие – неправильные?
2. Постройте гипотезу о причине ошибки.
3. Проверьте гипотезу (для этого пропустите новый тест).
4. При подтверждении гипотезы следует внести исправление, а потом повторить все тесты (не внесли ли вы новую ошибку – вероятность этого от 0,2 до 0,5).

### Наилучший способ отладки

Просто читать программу и изо всех сил стараться вникнуть в алгоритм, хотя это и требует усердия и собранности.

Можно пользоваться автоматизированными средствами отладки: в интегрированной среде Паскаля имеется отладчик *debug*, который предлагает средства отслеживания и трассировки программы (в том числе печать переменных, счетчиков, критических значений). Эффективность отладчика новичками часто переоценивается.

Парадоксы программной ошибки:

- ошибка – не всегда результат недостатков;
- ошибка не всегда может быть выявлена;
- выявленная ошибка не всегда может быть описана;
- описанная ошибка не всегда может быть понята;
- понятая ошибка не всегда может быть исправлена;
- исправленная ошибка не всегда может улучшить качество программы.

Известна некоторая статистика о программных ошибках.

- Язык С++ дает на 25 % больше ошибок, чем традиционные Си или Паскаль.
- Исправление ошибок в объектно-ориентированных программах на С++ требует в 2–3 раза большего времени. Наследование порождает в 6 раз больше ошибок, чем его отсутствие.
- 33% всех ошибок случаются реже, чем 1 раз за 5000 лет работы системы.
- Каждый Кбайт кода содержит 0,5–2 ошибки.

## 8 СОРТИРОВКА

### 8.1 Постановка задачи

Что такое сортировка?

Пусть даны элементы  $a_1, a_2, \dots, a_n$  и функция упорядочения  $f(a_i)$ , которая возвращает целое или вещественное число. *Сортировка* означает перестановку этих элементов в таком порядке  $a_{k1}, a_{k2}, \dots, a_{kn}$ , что  $f(a_{k1}) \leq f(a_{k2}) \leq \dots \leq f(a_{kn})$  (сортировка по неубыванию) или  $f(a_{k1}) \geq f(a_{k2}) \geq \dots \geq f(a_{kn})$  (сортировка по невозрастанию).

**Что можно сортировать?**

- Числа:  $f(a_i) = a_i$ .
- Литеры:  $f(a_i) = \text{ord}(a_i)$ .
- Значения перечислимого типа:  $f(a_i) = \text{ord}(a_i)$ .
- Массивы и строки – использовать лексикографический порядок.
- Записи.

Мы будем сортировать массивы, в частности, массив `a`.

```
type index=1..n;
var a:array[index] of element;
```

В качестве элементов массива рассматриваются любые данные, для которых можно ввести отношение порядка (естественное или какое-то особенное). При реализации алгоритма сортировки для какого-то особенного порядка удобно в качестве отношения «больше» между элементами использовать значение специально введенной для этого булевской функции **greater** ( $x, y$ ) =  $f(x) < f(y)$ .

К алгоритмам сортировки предъявляется **требование экономии памяти**: т.е. переупорядочивание элементов нужно выполнять на том же самом месте, не использовать вспомогательные массивы.

### 8.2 Сортировка простыми включениями

Этот метод обычно используют игроки в карты.

- Элементы (карты) условно разделяются на готовую последовательность  $a_1, a_2, \dots, a_{i-1}$  и входную последовательность  $a_i, \dots, a_n$ .



- На каждом шаге, начиная с  $i=2$  и увеличивая  $i$  на 1, берут  $i$ -ый элемент входной последовательности и передают в главную последовательность, вставляя его на подходящее место.

Алгоритм

```
for i:=2 to n do
  begin
    x:=a[i];
    {вставить x на подходящее место в
     a[1], a[2], ..., a[i-1] или оставить на месте}
  end
```

В следующем примере каждой строке соответствует состояние массива после очередного шага цикла (выделен шрифтом выби-  
раемый элемент):

```
44 55 12 42 94 18 06 67
44 55 12 42 94 18 06 67
12 44 55 42 94 18 06 67
12 42 44 55 94 18 06 67
12 42 44 55 94 18 06 67
12 18 42 44 55 94 06 67
06 12 18 42 44 55 94 67
06 12 18 42 44 55 67 94
```

Для этого алгоритма число сравнений в среднем равно  $n^2/4$  (в худшем случае равно  $n^2/2$ ). Число присваиваний (пересылок) – та-  
кое же.

### 8.3 Сортировка простым выбором

Метод:

- Выбирается наименьший элемент.
- Он меняется с первым элементом  $a[1]$ .
- Эти операции повторяются с оставшимися  $n-1$  элементами, потом с  $n-2$  элементами, пока не останется только один наи-  
большой элемент.

В следующем примере каждой строке соответствует состоя-  
ние массива после очередного шага цикла (выделен шрифтом  
выбираемый элемент):

```

44 55 12 42 94 18 06 67
06 55 12 42 94 18 44 67
06 12 55 42 94 18 44 67
06 12 18 42 94 55 44 67
06 12 18 42 94 55 44 67
06 12 18 42 44 55 94 67
06 12 18 42 44 55 94 67
06 12 18 42 44 55 67 94

```

Алгоритм:

```

for i:=1 to n-1 do
  begin
    {присвоить k индекс наименьшего элемента
     из a[i], a[i+1], ..., a[n]}
    {поменять местами a[i] и a[k]}
  end

```

Для этого алгоритма число пересылок в среднем равно  $n \ln n$ , наихудшее —  $n^2/4$ .

Число сравнений равно  $(n^2 - n)/2$ .

Рассмотрим конкретное применение алгоритма простого выбора.

Задача. Дана вещественно значная матрица  $a$  с  $n$  строками и  $m$  столбцами. Определить для каждой строки  $a_i$  функцию  $f(a_i) = a_{i1} - a_{i2} + a_{i3} - \dots + (-1)^{m+1} a_{im}$ . Переставить строки матрицы  $a$  по неубыванию значений функции  $f$ .

```

const n=3; m=4;
type
  index=1..n;
  element = array[1..m] of real;
  mas = array[index] of element;

function f(s:element):real;
var
  i:integer;
  y:real;
begin
  y:=0;
  for i:=1 to m do

```

```

        if i mod 2 = 0 then y:=y-s[i]
            else y:=y+s[i];
    f:=y
end;

procedure sort(var a:mas);
var i,j,k:index;
    x:element;
begin
    for i:=1 to n-1 do
        begin
            k:=i;
            x:=a[i];
            for j:=i+1 to n do
                if f(a[j])< f(x) then begin k:=j;
                    x:=a[j] end;

            a[k]:=a[i];
            a[i]:=x
            end;
        end;

var
    a:mas;

...
begin
    {ввод матрицы a}
    sort(a);
    {печать матрицы a}
end.

```

## 8.4 Сортировка простым обменом («метод пузырька»)

Метод:

- Будем представлять массив в виде столбца элементов – первый элемент массива является самым верхним.
- Сравниваем и обмениваем два соседних элемента до тех пор, пока не будут рассортированы все элементы. На каждом шаге, двигаясь снизу вверх, меняем все подходящие пары. «Легкие» элементы «всплывают» вверх после каждого прохода.

```

44 06 06 06 06 06
55 44 12 12 12 12
12 55 44 18 18 18
42 12 55 44 42 42
94 42 18 55 44 44
18 94 42 42 55 55
06 18 94 67 67 67
67 67 67 94 94 94

```

Алгоритм:

```

for i:=2 to n do
for j:=n downto i do
  {Если a[j] и a[j-1] не в нужном порядке, то их меня-
  ем местами}

```

Число перемещений в среднем равно  $3n^2/4$ , в худшем случае –  $3n^2/2$ . Число сравнений равно  $(n^2-n)/2$ .

## 8.5 Сортировка слиянием

*Слияние* обозначает объединение двух или более упорядоченных последовательностей в одну упорядоченную последовательность. Можно, например, слить две последовательности – (503, 703, 765) и (087, 512, 677), получив (087, 503, 512, 677, 703, 765).

Простой способ сделать это – сравнить два наименьших элемента, вывести наименьший из них (или оба – если они равны) и повторить эту процедуру.

Начав с

```

503, 703, 765
087, 512, 677

```

получим

```

503, 703, 765
                    => 087
512, 677

```

затем

$$\begin{array}{rcl} 703,765 & & \\ & \Rightarrow & 087,503 \\ 512,677 & & \end{array}$$

и т.д. Необходимо позаботиться о действиях на случай, когда исчерпается одна из последовательностей.

Общий объем работы, выполняемый алгоритмом слияния, пропорционален сумме длин исходных последовательностей.

## 9 ПЕРЕЧИСЛИМЫЙ ТИП

### 9.1 Определение перечислимого типа

Скалярные стандартные и ограниченные типы – интуитивно понятная трактовка типа как множества традиционных (целых, вещественных или символьных) значений из определенного диапазона.

*Перечислимые типы* вводят некоторое простое обобщение такой трактовки посредством абстрагирования от «физической» природы значений.

Иными словами, можно определить новый тип путем *явного перечисления всех его возможных значений*, причем *каждое такое значение будет определяться только именем*.

Пример.

Переменная программы, представляющая состояния светофора, – введение перечислимого типа из трех значений **red**, **green**, **yellow**.

Синтаксис:

<перечислимый тип>::=

--->(-----><идентификатор>----->)----->  
           |  |  
           <-----> , <----->

Примеры:

**type**

```
color=(red,yellow,green) ;
week=(monday, tuesday, wednesday, thursday,
      friday, saturday, sunday) ;
```

**var**

```
v:color;
d:(left,up,right,down) ;
```

Boolean также является перечислимым типом:

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

Имена из списка перечислимого типа считаются **константами** соответствующего перечислимого типа. Эти идентификаторы должны быть уникальными. Недопустимо описание двух и более перечислимых типов с совпадающими константами.

Значения перечислимого типа **упорядочены** (в соответствии с описанием), порядковый номер начинается с 0. Применимы функции **ord**, **pred**, **succ**. К значениям перечислимого типа применяются **операции сравнения** (сравниваются порядковые номера). Переменные перечислимого типа могут использоваться в качестве параметра цикла:

```
var
    d:week;
.....
for d:=monday to sunday do S;
```

Допускается образование ограниченных типов из перечислимых по обычным правилам.

Пример:

```
type
workweek=monday..friday;
```

Значения перечислимого типа нельзя использовать для непосредственного ввода и вывода с помощью операторов **read** и **write**.

## 9.2 Оператор варианта

Оператор варианта – обобщение условного оператора.

Пример:

```
case v of
    red:   write('красный');
    yellow:write('желтый');
    green: write('зеленый');
end;
writeln(' цвет');
```





```
type
month=(jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,
dec);
var
  m:month;
  d:28..31;
```

По значению m присвоить переменной d число дней в месяце:

```
case m of
  apr,jun,sep,nov: d:=30;
  feb: d:=29 else d:=31
end;
```

## 10 МНОЖЕСТВЕННЫЙ ТИП

### 10.1 Определение множественного типа

Чтобы ввести в язык Паскаль вычислительную структуру множеств, используют **множественный** тип. Значения множественного типа, так же как и массивы, строятся из нескольких значений одного (базового) типа. Однако, в отличие от массивов, значение множественного типа может содержать *любое* количество *различных* элементов базового типа – от нуля элементов (пустое множество) до всех возможных значений базового типа. Иными словами, *возможными значениями переменных множественного типа являются все подмножества значений базового типа*.

Множественный тип задается с помощью двух служебных слов – `set` и `of` – и следующего за ним базового типа. Например:

```
type
digits = set of 1..5;
var
s:digits;
```

Переменная `s` в качестве значений может принимать следующие множества целых чисел:

пустое множество,  $\{1\}$ , ...,  $\{5\}$ ,  $\{1,2\}$ , ...,  $\{4,5\}$ ,  $\{1,3,5\}$ , ...,  $\{4,5,3,2\}$ , ...,  $\{1,2,3,4,5\}$  (всего 32 различных множества).

Обратите внимание:

- Все значения базового типа в множестве должны быть *различными*.
- Порядок «расположения» элементов в множестве никак *не фиксируется*.

Базовый тип множества может быть:

- символьным,
- перечислимым,
- ограниченным.

Базовый тип должен содержать **не более 256 значений**. Если базовый тип – ограниченный целый, то значения должны быть в диапазоне от 0 до 255.

Пример:

```
type
  elemColor = (red, yellow, blue);
  color = set of elemColor;
```

В Паскале допускаются явные изображения значений множественного типа:

- пустое множество изображается [],
- в общем случае изображение строится из списка элементов множества, разделенных запятыми, и весь список заключается в квадратные скобки.

Например:

```
[1, 2, 5]
[red, yellow]
```

В качестве элементов в изображении множества допускаются выражения, тип которых должен совпадать с базовым типом. Кроме того, можно указывать диапазоны значений. Так например, следующие два множества равны: [1,3..5] и [1,3,4,5].

Следующие изображения представляют одно и то же множество:

```
[1..3], [1, 2, 3], [1, 2, 3, 1], [3, 3, 1, 1, 2, 2, 2, 3].
```

Примеры:

```
type
  Setofchar = set of char;
  digits = set of 0..100;
var
  mychars: setofchar;
  mydig1, mydig2: digits;
  x, y: 0..100;
  .....
  mychars := ['a'..'z', ' ' .. '9', '_'];
  mydig1 := [];
  mydig2 := mydig1;
  mydig1 := [x..x+10, 0, y-1, y+1];
```

## 10.2 Операции

- **Теоретико-множественные операции**

Множества языка Паскаль обладают свойствами математических множеств. В частности, над ними можно выполнять те же операции.

**Объединение множеств** – бинарная, коммутативная и ассоциативная операция.

$C := A + B;$

**Пересечение множеств** – бинарная, коммутативная и ассоциативная операция.

$C := A * B;$

**Разность множеств** – бинарная операция.

$C := A - B;$

Примеры:

```
[1,3]+[2,4] = [1..4]
[1..10]*[5..15] = [5..10]
[1,2]*[3,4] = []
[1..10] - [5..15] = [1..4]
```

Как добавить элемент во множество? Для этого можно использовать объединение множеств. Пример:

```
mydig1:=mydig1+[5];
```

Альтернативой оператору  $S := S + [x]$  является оператор **Include(S,x)**. Имеется и обратная процедура **Exclude** – исключение элемента из множества. У этой процедуры два параметра: первый указывает множество, а второй – исключаемый элемент.

- **Проверка принадлежности множеству**

$x \text{ in } A$  ( $x$  принадлежит  $A$ ?) – бинарная булевская операция:

```
2 in [1..10,21] = true,
```

```
5 in [1,2,x,10] = true тогда и только тогда, когда x=5.
```

**Использование постоянных множеств для проверок:**

`(ch='a') or (ch='b') or (ch='x') or (ch='y')`

эквивалентно

`ch in ['a', 'b', 'x', 'y'];`

`('0'<c) and (c<'9')` эквивалентно `c in ['0'..'9']`.

- Проверка на равенство, неравенство и включения множеств

**Равенство множеств** – бинарная булевская операция **A=B**.

**Неравенство множеств** – бинарная булевская операция **A<>B**.

**Множество A входит во множество B** – бинарная булевская операция **A<=B (B>=A)** .

Операции **<** и **>** для множеств не используются.

Примеры:

Значение выражения `[1,2,3]=[1,2]` равно **false**.

Значение выражения `[1,2,3]>=[1,2]` равно **true**.

Значение выражения `[s]<=[1..10]` равно **true**, тогда и только тогда, когда  $1 \leq s \leq 10$ .

**Задача.** *Состоят ли строки `s[1]` и `s[2]` из одних и тех же символов?*

```
a[1]:=[];a[2]:=[];
for i:=1 to 2 do
  for j:=1 to length(s[i]) do
    a[i]:=a[i]+[s[i][j]];
writeln(a[1]=a[2]);
```

**Задача.** Используя решето Эратосфена, найти простые числа  $\leq 255$ .

Решето Эратосфена – первый эффективный алгоритм для генерации простых чисел:

1. Выпишем числа 2, 3, 4, 5, 6,...
2. Отметим первый элемент в последовательности `p` как простое число.

3. Удалим из списка все числа, кратные  $p$ .

4. Вернемся к шагу 2.

Эти операции дают следующее:

2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,...

– так, что 2 – простое число.

Убираем числа, кратные 2:

3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,...

– так, что 3 – простое число.

Убираем теперь числа, кратные 3:

5,7,11,13,17,19,23,25,29,31,35,37,41,43,...

– так, что 5 – простое число.

Теперь удаляем все числа, кратные 5:

7,11,13,17,19,23,29,31,37,41,43,47,49,53,...

и т.д.

```
const n=255; {максимальное количество перебираемых
натуральных чисел}
var
    S,          {исходное множество}
    Primes: set of 2..n; {результатирующее множество}
    next,j:integer;
begin
    S:=[2..n]; {все числа в заданном диапазоне}
    Primes:=[];
    next:=2; {начинаем с минимального простого числа}
    repeat
        {поиск очередного простого числа}
        while not (next in S) do
            next:=next+1; {ищем в S наименьшее число}
        Primes:=Primes+[next]; {помещаем его в Primes}
        j:=next;
        while j<=n do {удаляем из S все числа, кратные
            next}
            begin
                S:=S-[j];
                j:=j+next
            end;
        until S=[]; {повторяем цикл до исчезновения S}
        writeln('Простые числа < 256:');
        for j:=2 to n do
            if j in Primes then write(j:5)
    end.
```

## 11 ФАЙЛОВЫЕ ТИПЫ И ВВОД-ВЫВОД

В языке Паскаль под **файлом** понимается область памяти на внешнем запоминающем устройстве, способная хранить некоторую совокупность информации. Можно как поместить определенные данные в эту область внешней памяти, так и извлечь их из нее. Эти действия имеют общее название – **ввод-вывод**.

### 11.1 Файловые переменные и типы

Для организации ввода-вывода могут быть определены специальные **переменные файловых типов**, которые считаются *представителями файлов* в программе.

Использование переменных файловых типов предполагает интерпретацию файла как потенциально бесконечного списка значений одного и того же (базового) типа.

Пример:

```
var f:file of integer;
```

Под именем *f* определен список неопределенного количества целых чисел, расположенных на некотором внешнем устройстве (например, магнитном диске).

С каждой переменной файлового типа также связано понятие текущего указателя файла. **Текущий указатель** – скрытая переменная (т.е. неявно описанная вместе с файловой переменной), *которая обозначает («указывает») некоторый конкретный элемент файла*.

Как правило, все действия с файлами (чтение из файла, запись в файл) производятся поэлементно, причем в этих действиях участвует тот элемент файла, который обозначается текущим указателем. В результате совершения операций текущий указатель может перемещаться, настраиваясь на тот или иной элемент файла.

Заметим, что один и тот же внешний файл в различных программах (или даже в различных частях одной и той же программы) может интерпретироваться по-разному, например как последовательность целых чисел или как последовательность некоторых массивов, и т.д.

<файловый тип>::= file of <тип>

Тип элементов может быть любым, за исключением файлового.

Пример:

```
type      sequence=file of char;
var F1,F2:sequence;  inputData:file of real;
```

## 11.2 Установочные и завершающие операции над файлами

Стандартные процедуры: **assign**, **reset**, **rewrite**, **close**.

Процедура **assign** предназначена для установления связи между конкретным физическим файлом на магнитном носителе и переменной файлового типа, которая будет являться представителем этого файла в программе.

`assign(<идентификатор – файловая переменная>,<строковое выражение>);`

Значение второго параметра – литеральное имя файла. Имя файла строится по правилам, принятым в операционной системе MS-DOS для именования файлов.

Пример:

```
assign(f, 'd:\mydir\myfile.dta');
```

После выполнения данного вызова файловая переменная **f** будет связана с файлом **myfile.dta** в каталоге **mydir** диска **d**.

Второй параметр процедуры **assign** может быть строкой, содержащей условное обозначение "*псевдофайлов*" MS-DOS, т.е. файлов, связанных с конкретным физическим устройством:

- **con** – консоль, т.е. для случая вывода информация помещается на экране дисплея, а в случае ввода информация воспринимается с клавиатуры;
- **prn** – вывод на принтер;
- **kbd** – ввод с клавиатуры без «эха»;
- **nul** – фиктивное (несуществующее) устройство. Может использоваться для вывода информации «в никуда», когда в про-



грамме почему-либо нужно указать имя выходного файла, а информация, записываемая в него, не требуется.

Процедуры **reset** и **rewrite** имеют один параметр – файловую переменную и предназначены для открытия файлов.

При этом файловая переменная, указываемая в качестве параметра, должна быть уже связана с конкретным дисковым файлом с помощью процедуры **assign**.

*Открытие файла* – поиск файла на внешнем носителе, образование специальных системных буферов для обменов с ним и установка текущего указателя файла на его начало.

**Reset** используется, когда файл уже существует. **Rewrite** используется, и когда файл еще не существует, а если существует, то он очищается.

Процедура **close** имеет один параметр – файловую переменную и завершает действия с файлом – ликвидируются внутренние буферы, образованные при открытии этого файла.

После этого файловую переменную можно связать посредством процедуры **assign** с каким-либо другим дисковым файлом.

Заметим, что при окончании работы всей программы происходит автоматическое закрытие всех файлов, открытых в программе.

### 11.3 Операции ввода-вывода

Процедуры **write** и **read**, в отличие от многих других процедур, могут вызываться с *различным числом параметров, и эти параметры могут иметь различные типы*.

Процедура **read** предназначена для **чтения значений** из файла в программу. Первым параметром должно быть имя файловой переменной, к которой была применена одна из операций открытия (**reset** или **rewrite**). Далее должны следовать переменные, в которые будут помещаться читаемые из файла значения. Тип этих переменных должен совпадать с базовым типом файла из первого параметра.

Выполнение процедуры `read` происходит следующим образом. Начиная с текущей позиции указателя файла, будут последовательно читаться значения, содержащиеся в файле. Каждое прочитанное значение будет присваиваться очередной переменной из тех, которые указаны в вызове процедуры. После каждого акта чтения указатель файла будет смещаться на следующую позицию.

Если указатель файла указывает на «конец файла», то чтение невозможно. Функция `eof(<файловая переменная>)` – булевская функция, равна истине, если имеется ситуация «конец файла».

Процедура `write` позволяет записывать в файл информацию из программы. Первым параметром этой процедуры должна быть файловая переменная, открытая процедурой `reset` или `rewrite`. Далее должен идти список переменных, тип которых совпадает с базовым типом файла из первого параметра.

Порядок выполнения процедуры `write`:

- Значение очередной переменной будет помещено в файл в место, отмеченное текущим указателем.
- После этого текущий указатель будет передвинут на одну позицию и действия повторяются для следующей переменной из списка параметров.

## 11.4 Текстовые файлы

**Текстовые файлы** – файлы, у которых базовый тип есть `char`.

Представителем текстового файла в программе является переменная файлового типа, которая должна быть описана с указанием стандартного типа `text`:

```
var
  textInf: text;
```

Структура текстовых файлов отличается от структуры обычных файлов (линейная последовательность элементов одного типа) тем, что содержимое текстового файла рассматривается как *последовательность символьных строк переменной длины, разделенных специальной комбинацией, называемой «конец стро-*

ки». Как правило, эта комбинация строится из управляющего кода «перевод каретки» (символ #13), за которым, возможно, следует управляющий код «перевод строки» (символ #10). Текстовый файл завершается специальным кодом «конец файла» (символ #26).

Открытие текстового файла для чтения выполняет процедура `reset`. Открытие текстового файла для записи выполняет процедура `rewrite`.

Логическая функция `eofn(<файловая переменная>)` возвращает `true`, если текущая строка исчерпана, и `false` в противном случае.

*В процедуре `write` для текстового файла все параметры, начиная со второго, могут быть не только переменными, но и выражениями следующих типов: `integer`, `real`, `char`, `boolean` и `string`.*

Текстовый файл по определению содержит символьную информацию, поэтому при записи значения других типов (`integer`, `real`) будут преобразовываться в символьное представление и в таком виде записываться в очередную строку текстового файла. Аналогично при чтении из текстового файла очередная часть текущей строки будет пониматься как символьное представление значения, тип которого определяется типом очередной переменной из процедуры `read`.

Помимо процедур `read` и `write`, для текстовых файлов имеются две их модификации – процедуры `readln` и `writeln`. Эти процедуры осуществляют те же действия, что и соответствующие процедуры `read` и `write`, но после операций чтения и записи производят переход к следующей строке текстового файла.

*С текстовым файлом можно использовать текстовый редактор для создания или изменения файла.*

В Паскале имеются две стандартных файловых переменных текстового типа – **Input** и **Output**. Стандартная файловая переменная `Input` представляет собой доступный только для чтения файл, связанный со стандартным файлом ввода операционной системы (клавиатура). Вторая стандартная файловая переменная `Output` – это доступный только для записи файл, связанный со стандартным файлом вывода (дисплей). Перед началом выполнения программы DOS эти файлы автоматически открываются. Имя

файла в процедурах read и write не указывается, если работа ведется со стандартным файлом.

## 11.5 Примеры работы с файлами

1. Запись в файл квадратов целых чисел.

```
var f:file of integer;
.....
assign(f,'...');
rewrite(f);
for i:=1 to n do
  begin
    k:=i*i;
    write(f,k);
  end;
close(f);
```

2. Компонентами файла f являются массивы a1, a2 , ... из 10 действительных чисел. Вывести наибольшие элементы всех этих массивов.

```
type
  mas = array[1..10] of real;
  fmas = file of mas;
var a:mas; f:fmas; r:real; i:integer;
begin
  assign(f,'...'); reset(f);
  while not eof(f) do
    begin
      read(f,a); r:=a[1];
      for i:=2 to 10 do if a[i]>r then r:=a[i];
      writeln(r)
    end;
  close(f)
end.
```

3. Описать логическую функцию eq(t1,t2), проверяющую текстовые файлы t1 и t2 на равенство.

```
function eq(var t1,t2:text):boolean;
var
```

```

    c1,c2:char;
    ok:boolean;
begin
    reset(t1);reset(t2);ok:=true;
    while not eof(t1) and not eof(t2) and ok do
        begin
            read(t1,c1);read(t2,c2);ok:=c1=c2;
        end;
    eq:=ok and eof(t1) and eof(t2);
end;

```

4. Программа, которая печатает свой файл с текстом.

```

    var s:string[126];
        f:text;
begin
    assign(f,'name.pas');
    reset(f);
    while not eof(f) do
        begin readln(f,s); write(s) end;
    close(f)
end.

```

«... на столе появился маленький Витька Корнеев, точная копия настоящего, но величиной с руку. Он щелкнул маленькими пальчиками и создал микродубля еще меньшего размера. Тот тоже щелкнул пальцами. Появился дубль величиной с авторучку. Потом величиной со спичечный коробок. Потом с наперсток».

А. и Б. Стругацкие

«Понедельник начинается в субботу»

## 12 РЕКУРСИЯ

### 12.1 Понятие рекурсии

Объект называется *рекурсивным*, если он содержит сам себя или определен с помощью самого себя.

Рекурсия встречается не только в математике, но и в обычной жизни. Каждый сталкивается с рекурсией, когда стоит с зеркальцем перед большим зеркалом. Рекурсия встречается обычно и в природе: деревья имеют рекурсивное строение (ветки образуются из других веток), реки образуются из впадающих в них рек. Клетки делятся рекурсивно. Продолжение жизни связано с рекурсивным процессом. Молекулы ДНК и вирусы размножаются, копируя себя, живые существа имеют потомство, которое, в свою очередь, тоже имеет потомство и т.д. Рекурсия распространена и в языке, и в поведении так же, как и в способах рассуждения и познания. Рекурсия в языке, например, может быть в структуре или в содержании:

«Петя сказал, что Вася сказал, что...»

«Знаю, что знаю, но не помню.»

«Сделать; заставить сделать; заставить, чтобы заставили сделать;...»

«Замени *x* этим предложением.»

«Запомни и передай это сообщение.»

...

Литературным примером может служить «рассказ в рассказе», как-то: известное стихотворение «У попа была собака,...», роман Яна Потоцкого «Рукопись, найденная в Сарагосе», рассказ Хулио Кортасара «Непрерывность парков» или рассказ Виктора

Пелевина «Водонапорная башня» (и по форме – состоит из одного предложения и по содержанию). Музыкальные формы и действия также могут быть рекурсивными во многих отношениях (например, канон или fuga, в которых мелодия сопровождается той же мелодией с задержкой, и другие). Целенаправленное поведение и решение проблем так же являются рекурсивными процессами.

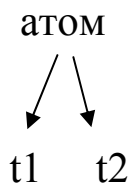
Рекурсия является особенно мощным средством в математических определениях. Известны примеры рекурсивных определений натуральных чисел, древовидных структур и некоторых функций.

*Натуральные числа:*

- 1) 1 есть натуральное число;
- 2) целое число, следующее за натуральным, есть натуральное число.

*Древовидные структуры:*

- 1) атомарный объект есть дерево;
- 2) если  $t_1$  и  $t_2$  – деревья, то



есть дерево (нарисованное сверху вниз).

*Функция факториал  $n!$  для неотрицательных целых чисел:*

- 1)  $0! = 1$ ;
- 2) если  $n > 0$ , то  $n! = n (n-1)!$ .

Очевидно, что мощность рекурсии связана с тем, что она позволяет определить бесконечное множество объектов с помощью конечного высказывания.

Рекурсия в программировании – один из важнейших принципов построения подпрограмм. Если процедура  $P$  содержит явное обращение к самой себе, то она называется *прямо рекурсивной*; если  $P$  содержит обращение к процедуре  $Q$ , которая

содержит (прямо или косвенно) обращение к  $P$ , то  $P$  называется *косвенно рекурсивной*. Поэтому использование рекурсии не всегда сразу видно из текста программы.

С процедурой принято связывать некоторое множество локальных объектов, т.е. переменных, констант, типов и процедур, которые определены локально в этой процедуре, а вне ее не существуют или не имеют смысла. Каждый раз, когда такая процедура рекурсивно вызывается, для нее создается новое множество локальных переменных. Хотя они имеют те же имена, что и соответствующие элементы множества локальных переменных, созданного при предыдущем обращении к этой же процедуре, их значения различны. Следующие правила области действия идентификаторов позволяют исключить какой-либо конфликт при использовании имен: *идентификаторы всегда ссылаются на множество переменных, созданное последним*. То же правило относится к параметрам процедуры.

Подобно операторам цикла, рекурсивные процедуры могут приводить к бесконечным вычислениям. Поэтому для того чтобы работа процедуры когда-либо завершилась, необходимо, чтобы рекурсивное обращение к процедуре подчинялось некоторому логическому условию, которое в какой-то момент перестает выполняться.

Для начала приведем два элементарных примера прямой рекурсии.

*Пример 1.* Подпрограмма для вычисления факториала неотрицательного числа:

```
function fac(n:integer{n>=0}):integer;
begin
  if n=0 then fac:=1
  else fac:=n*fac(n-1)
end
```

Отметим сразу же, что рекурсивную функцию можно заменить рекурсивной процедурой. Для данного случая определим процедуру:



```

procedure f(var y:integer;n:integer) ;
var x:integer;
    begin
        if n=0 then y:=1
        else begin f(x,n-1);y:=n*x end
    end

```

Первый параметр в процедуре  $f$  в результате работы получает значение равное  $n!$ .

*Пример 2.* Подпрограмма для вычисления наибольшего общего делителя двух положительных целых чисел:

```

function gcd(m,n:integer) ;integer;
    begin
        if m=n then gcd:=m else
        if m<n then gcd:=gcd(n-m,m)
        else gcd:=gcd(m-n,n)
    end

```

Идею косвенной рекурсии хорошо иллюстрирует гравюра Мориса Эшера, на которой изображены две руки, взаимно рисующие друг друга. Косвенная рекурсия может быть задана в системе подпрограмм, которые определяются «бок о бок» и взаимно опираются друг на друга (рис. 1).

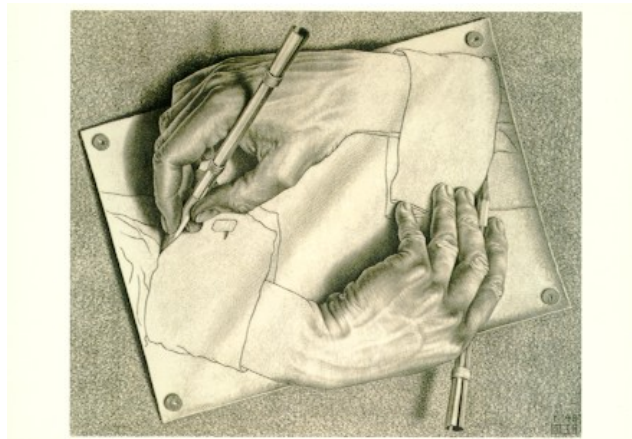


Рис. 1 – Рисующие руки

*Пример 3.* Пара подпрограмм ( $iseven$ ,  $isodd$ ) для определения четности или нечетности данного натурального числа  $n$ .

```

function iseven(n:integer):boolean;
  {is iven - быть четным}
  begin
    if n=1 then iseven:=false else iseven:=isodd(n-1)
  end;

function isodd(n:integer):boolean;
  {is odd - быть нечетным}
  begin
    if n=1 then isodd:=true else isodd:=iseven(n-1)
  end;

```

Подставляя, скажем, *isodd* в *iseven*, можно получить непосредственную рекурсию для *iseven*:

```

function iseven(n:integer):boolean;
  begin
    if n=1 then iseven:=false else
      if n=2 then iseven:=true
      else iseven:=iseven(n-2)
    end;

```

*Пример 4.* Примером иерархической рекурсивной системы подпрограмм может служить пара (*gcd1*, *moda*), где *moda* – подпрограмма, заменяющая операцию *mod* (с положительным делителем), а *gcd1* – подпрограмма для вычисления наибольшего общего делителя двух неотрицательных целых чисел:

```

function moda(m,n:integer):integer;
  {m>=0 и n>0}
  begin
    if m<n then moda:=m
    else moda:=moda(m-n,n)
  end;

function gcd1(m,n:integer):integer;
  {m,n >=0}
  begin
    if n=0 then gcd1:=m
    else gcd1:=gcd1(n,moda(m,n))
  end;

```

Простое исключение *moda* с тем, чтобы, скажем, получить *gcd* из примера 2, здесь уже невозможно.

## 12.2 Создание рекурсивных подпрограмм

### Органически рекурсивные определения

Нет готовых рецептов, как для данной произвольной проблемы получить (рекурсивный) алгоритм ее решения. Однако некоторые рекомендации имеются.

Многие рекурсивные подпрограммы являются точным «слепок» с соответствующего определения. Это верно, скажем, в отношении подпрограммы вычисления факториала.

Для рекурсивного определения полиномов Лежандра:

$$P_n(x) = \begin{cases} 1, & \text{если } n=0, \\ x, & \text{если } n=1, \\ ((2n-1)xP_{n-1}(x) - (n-1)P_{n-2}(x))/n, & \end{cases}$$

сразу получаем рекурсивную программу:

```
function p(n:integer;x:real):real;
begin
  if n=0 then p:=1 else
    if n=1 then p:=x
    else p:=((2*n-1)*x*p(n-1,x) - (n-1)*p(n-2,x))/n
end
```

### Извлечение рекурсии из постановки задачи

В большинстве случаев, однако, рассматриваемая задача не является алгоритмически сформулированной. Поэтому пытаются прийти к (рекурсивному) алгоритму, сводя общую задачу к «более простым» задачам того же рода. Точнее говоря, речь идет о получении достаточного числа (условных) уравнений, определяющих искомую функцию. Иногда непосредственно ясно, как это сделать, но чаще всего требуется интуиция. Например, для решения задачи нахождения наибольшего общего делителя двух (положительных) чисел можно привлечь следующий математический результат: если  $a$  – большее из двух чисел  $a$  и  $b$ , то пары  $a, b$

и  $a-b$ ,  $b$  имеют одни и те же делители, а значит, один и тот же наибольший общий делитель. Отсюда непосредственно следует алгоритм *gcd* нахождения наибольшего общего делителя.

В следующем примере рекурсивно определим сложение целых, опираясь на операции перехода к следующему и предыдущему числам. В силу законов ассоциативности и коммутативности имеем  $(a+1)+(b-1)=a+b$ . Поэтому операцию сложения можно ввести так:

```
function plus(a,b:integer):integer;
begin
  if b=0 then plus:=a else
    if b>0 then plus:=plus(succ(a),pred(b))
    else plus:=plus(pred(a),succ(b))
  end
```

В качестве еще одного примера покажем, как можно рекурсивно определить возведение в степень (причем алгоритм получается более эффективный, чем с использованием  $n$ -кратного умножения).

Алгоритм:

$$\begin{aligned} x^0 &= 1, \\ x^{2n} &= (x \times x)^n, \\ x^{2n+1} &= x \times (x \times x)^n. \end{aligned}$$

Получаем естественную программу:

```
function power(x:real; n:integer):real;
{предполагаем, что n>0}
begin
  if n=0 then power:=1 else
    if n mod 2 = 0 then power:= power(x*x, n div 2)
    else power := x * power(x*x, n div 2)
  end;
```

### Вложение

Если не удастся извлечь рекурсию из самой постановки задачи, то часто оказывается полезным обобщить задачу (например, введя дополнительные параметры); подходящее обобщение по-

звояет усмотреть возможность рекурсии, а, возвращаясь к частному случаю, мы получаем алгоритм для первоначальной задачи.

Классический пример применения такого приема вложения дал Маккарти в 1962 г. Исходная задача была следующей: *isprim(n)*: «Установи, является ли заданное положительное натуральное число  $n$  простым».

При этом предполагается, что известно определение простого числа: «Натуральное число  $n$  называется простым, если оно больше 1 и не делится ни на одно число, большее или равное 2 и меньшее  $n$ ».

Удачным обобщением будет такая задача (в которой вводится один дополнительный параметр): *ispr(n,m)*: «Установи, верно ли, что заданное натуральное число  $n$  не делится ни на одно число, большее или равное  $m$  и меньшее  $n$ ». (Не уменьшая общности, можно считать, что  $2 \leq m \leq n$ .)

Но для этой новой задачи ясно, что *ispr(n,m)* истинно, во-первых, если  $m=n$ , и, во-вторых, если истинно *ispr(n,m+1)* и  $n$  не делится на  $m$ , и мы приходим к подпрограмме:

```
function ispr(n,m:integer):boolean;
  {2 <= m <= n}
  begin
    if m=n then ispr:=true
    else ispr:=(n mod m <>0) and ispr(n,m+1)
  end
```

В качестве частного случая получаем нужную нам подпрограмму:

```
function isprim(n:integer):boolean;
  {n>=1}
  begin
    if n=1 then isprim:=false
    else isprim:=ispr(n,2)
  end
```

### Использование характеристических свойств

Иногда бывает целесообразно переформулировать характеристическое свойство задачи так, чтобы из него можно было извлечь какой-либо (другой) алгоритм. Так в задаче: «для заданного

натурального числа  $n \geq 1$  определить (единственное) натуральное число  $ld(n)$ , такое что

$$2^{ld(n)-1} \leq n \leq 2^{ld(n)} \gg$$

необходимо выполнение предиката  $P(ld(n), n)$ , где

$$P(a, n) = 2^{a-1} \leq n \leq 2^a.$$

Но для этого характеристического предиката  $P(a, n)$  имеет место очевидная рекурсия

$$P(a, n) = \begin{cases} a=1, & \text{при } n=1; \\ P(a-1, n \operatorname{div} 2), & \text{при } n>1, \end{cases}$$

а значит, и  $ld(n)$  допускает рекурсивное определение

$$ld(n) = \begin{cases} 1 & \text{при } n=1; \\ ld(n \operatorname{div} 2) + 1 & \text{при } n>1, \end{cases}$$

из которого получаем алгоритм:

```
function ld(n:integer):integer;
  {n>=1}
  begin
    if n=1 then ld:=1
    else ld:=ld(n div 2)+1
  end
```

### Разделяй и властвуй

Для решения той или иной задачи ее часто разбивают на части, находят их решения и затем из них получают решение всей задачи. Этот прием, особенно если его применять рекурсивно, часто приводит к эквивалентному решению задачи, подзадачи которой представляют собой ее меньшие версии. Проиллюстрируем эту технику на следующем примере.

Рассмотрим задачу о нахождении наибольшего и наименьшего элементов множества  $S$ , содержащего  $n$  элементов. Представим множество в виде массива **x:array[1..n] of integer**.

Очевидный путь поиска наибольшего и наименьшего элементов состоит в том, чтобы искать их по отдельности. Более оптимальный алгоритм по числу сравнений получается при одновременном нахождении наибольшего и наименьшего элементов.

Применяя прием «разделяй и властвуй», мы разбиваем множество  $S$  на два подмножества  $S_1$  и  $S_2$  по  $n/2$  элементов в каждом. После этого рекурсивно находим наибольший и наименьшие элементы в каждой из двух половин. Наибольший и наименьший элементы множества  $S$  можно определить, произведя еще два сравнения – наибольших элементов в  $S_1$  и  $S_2$  и наименьших элементов в них.

Обозначим через *max* и *min* очевидные подпрограммы для нахождения наибольшего и наименьшего элементов из двух чисел:

```
function max(a,b:integer):integer;
```

```
function min(a,b:integer):integer;
```

Части множества  $S$ , получающиеся в результате разбиения, в нашем представлении можно задавать парой индексов  $(i, j)$ , указывающих нижнюю и верхнюю границы подмассива.

В этих обозначениях легко написать следующую рекурсивную процедуру.

```
procedure maxmin(i,j:integer;var
maxel,minel:integer);
  var
    max1,min1,max2,min2:integer;
    k:integer;
  begin
    if abs(i-j)<=1 then begin
      maxel:=max(x[i],x[j]);
      minel:=min(x[i],x[j])
    end

    else begin
      k:=(i+j) div 2;
      maxmin(i,k,max1,min1);
      maxmin(k,j,max2,min2);
      maxel:=max(max1,max2);
      minel:=min(min1,min2)
    end
  end
```

Для нахождения наибольшего  $M1$  и наименьшего  $M2$  элементов множества  $S$  достаточно в главной программе обратиться к процедуре: **maxmin(1, n, M1, M2)** .

В нашем примере задача разбивалась на подзадачи равных размеров (точнее, почти равных размеров; идеальным был бы случай, когда  $n$  было бы степенью двойки). Это не было случайностью. Поддержание равновесия (балансировка) – основной руководящий принцип при разработке хорошего алгоритма. Разбиение задачи на подзадачи неравных размеров менее эффективно.

### 12.3 Хранение значений переменных в теле рекурсивной подпрограммы

*Задача.* Дана последовательность целых чисел, вводимая с клавиатуры, за которой следует 0. Напечатать эти числа в обратном порядке, не используя массивы.

Рекурсия позволяет запомнить значения и позже использовать их в обратном порядке.

```
procedure print;
    var a:integer;
begin
    read(a);
    if a<>0 then begin print;writeln(a) end
end
```

### 12.4 Рекурсия и итерация. Метод накапливающего параметра

Известно, что любую рекурсивную программу можно преобразовать в эквивалентную программу, не имеющую рекурсивных вызовов. И наоборот, для каждой программы, содержащей циклические вычисления (итерацию), существует эквивалентная рекурсивная программа, не имеющая циклов. Таким образом, рекурсия и итерация являются *взаимозаменяемыми методами* программирования. Отметим, что имеются языки программирования, в которых отсутствует итерация – ее заменяет рекурсия. К таким



языкам относятся «чистые» функциональные языки (например, Haskell) и Пролог.

Как и в итеративном случае, для решения конкретной задачи возможны различные рекурсивные программы, некоторые из них могут быть неэффективны.

Рассмотрим числа Фибоначчи. Пусть  $f(n)$  –  $n$ -ое число Фибоначчи, определяемое с помощью рекуррентного соотношения  $f(n+1)=f(n)+f(n-1)$ , для  $n>0$  и  $f(1)=1, f(0)=0$ .

При непосредственном, «лобовом подходе» мы получаем программу:

```
function f(n:integer):integer;
  begin
    if n=0 then f:=0 else
      if n=1 then f:=1
      else f:=f(n-1)+f(n-2)
    end
```

Эта программа неэффективна, так как каждое обращение к функции  $f$  при  $n>1$  приводит к двум дальнейшим обращениям, т.е. общее число обращений растет экспоненциально.

Однако очевидно, что числа Фибоначчи можно вычислять по итеративной схеме, при которой использование вспомогательных переменных  $x=f(i)$  и  $y=f(i-1)$  позволяет избежать повторного вычисления одних и тех же значений:

```
  {вычисление x=f(n) при n>0}
  i:=1;x:=1;y:=0;
  while i<n do
    begin
      z:=x;i:=i+1;
      x:=x+y;y:=z
    end
```

Но, мы можем определить эффективную рекурсивную функцию для вычисления чисел Фибоначчи, используя *метод накапливающего параметра*. Для ясности изложения запрограммируем сначала с помощью этого метода вычисление факториала:

```
function fac(n:integer):integer;
```

```

function facr(n,y:integer):integer;
begin
  if n=0 then facr:=y else facr:=facr(n-1,y*n)
end;
begin
  fac:=facr(n,1)
end

```

Параметр *y* в данном случае называется *накапливающим*. Для вычисления чисел Фибоначчи нам понадобятся два накапливающих параметра.

```

function fib(n:integer):integer;
  function fiba(n,f1,f2:integer):integer;
  begin
    if n<2 then fiba:=f2 else fiba:=fiba(n-1,
                                          f2,f1+f2)
    end;
  begin
    fib:=fiba(n,0,1)
  end;

```

## 12.5 Рекурсия в своем блеске и великолепии

### Ханойские башни

После примеров предыдущих разделов может сложиться мнение, что использование рекурсии излишне и предыдущие задачи можно было бы легко решить без рекурсии с помощью циклов. Для тривиальных задач это действительно так. В этом разделе рекурсия продемонстрирует всю свою мощь, красоту и необходимость.

В конце 19 столетия в Европе появилась игра под названием «Ханойские башни». Популярности игры содействовали рассказы о том, что этой игрой заняты служители храма брахманов и что завершение игры будет означать конец света. Предметы, которыми пользовались монахи, будто бы состояли из медной платформы с укрепленными на ней тремя алмазными иглами с насаженными 64 золотыми дисками. Цель игры – перенести башню с левой иглы на правую, причем за один раз можно переносить

только одно кольцо, кроме того, запрещается помещать большее кольцо над меньшим (рис. 2).

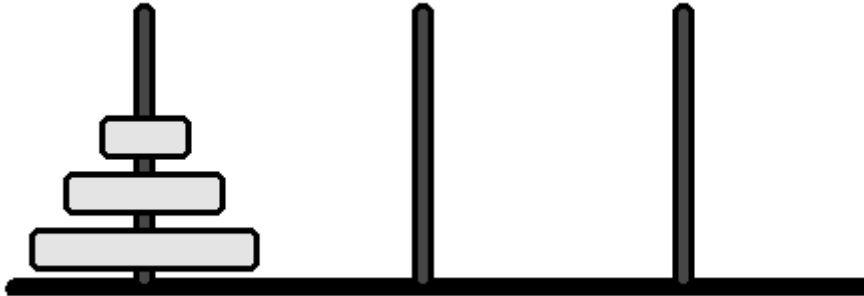


Рис. 2 – Ханойские башни

Предположим, что иглы пронумерованы числами 1, 2 и 3 и что монахам надо перенести 64 диска с иглы 1 на иглу 3. Обозначим поставленную задачу таким образом:

*перенести башню(64,1,3)*

Наша цель будет заключаться в разработке алгоритма, который подскажет монахам последовательность корректных перемещений, в результате чего задача будет решена. К простому решению приводит догадка о том, что основное внимание нужно обратить на нижний диск иглы 1, а не на верхний. Задача «перенести башню(64,1,3)» после этого оказывается эквивалентной следующей последовательности подзадач:

*перенести башню(63,1,2);*

*перенести диск с иглы 1 на иглу 3;*

*перенести башню(63,2,3)*

Это маленький, но значительный шаг к решению. Маленький, поскольку нам все еще надо дважды переносить по 63 диска. Значительный, потому что мы можем повторить подобный анализ столько раз, сколько будет необходимо. Например, задача

*перенести башню(63,1,2)*

может быть выражена как

*перенести башню(62,1,3);*

*перенести диск с иглы 1 на иглу 2;*

*перенести башню(62,3,2)*

Для построения общего алгоритма нам надо указывать, какой иглой можно пользоваться как временным хранилищем. Это можно сделать, расширив нашу запись так, чтобы

*перенести башню( $n, a, b, c$ )*

значило бы

*перенести  $n$  дисков с иглы  $a$  на иглу  $b$ , используя иглу  $c$  для временной башни*

Мы можем утверждать, что задача

*перенести башню( $n, a, b, c$ )*

может быть решена за три шага:

*перенести башню( $n-1, a, c, b$ );*

*перенести диск с  $a$  на  $b$ ;*

*перенести башню( $n-1, c, b, a$ ).*

Этот алгоритм не имеет смысла для случая  $n < 1$ , поэтому добавляем правило:

*ничего не делать, если  $n < 1$*

Процедуру для действия «перенести диск» запишем в следующем виде:

```
procedure transferDisk (from{откуда} , where{куда} : integer) ;
begin
    writeln (from, '--->' , where)
end
```

Рекурсивную процедуру для действия «перенести башню» получаем в виде:

```
procedure transferTower (n, from, where, work : integer) ;
begin
    if n > 0 then begin
        transferTower (n-1, from, work, where) ;
        transferDisk (from, where) ;
        transferTower (n-1, work, where, from)
    end
end
```

Пусть главная программа содержит операторы:

```
read(disk) ;  
transferTower(disk, 1, 3, 2)
```

Тогда будем иметь

Ввод :

3

Вывод :

1--->3

1--->2

3--->2

1--->3

2--->1

2--->3

1--->3

Математический анализ алгоритма показывает, что время, нужное программе для переноса башни, состоящей из  $n$  дисков, пропорционально  $2$  в степени  $n$ . Современные вычислительные машины, решая задачу Ханойские башни, могут вычислить (но не напечатать) одно перемещение за одну миллионную долю секунды. Даже при такой скорости для расчета переноса башни из 64 дисков им потребуется около миллиона лет.

### **Поиск маршрута – алгоритм с возвратом**

Особенно интересный раздел программирования – это задачи из области «искусственного интеллекта». Здесь нужно строить алгоритмы, которые находят решения определенной задачи не по фиксированным правилам вычисления, а методом проб и ошибок. Обычно процесс проб и ошибок разделяется на отдельные подзадачи. Часто эти подзадачи наиболее естественно описываются с помощью рекурсии. Процесс проб и ошибок можно рассматривать в общем виде как поисковый процесс, который постепенно строит и просматривает (а также обрезает) дерево подзадач.

*Задача.* Имеется  $n$  населенных пунктов, пронумерованных от 1 до  $n$ . Некоторые пары пунктов соединены дорогами. Определить, можно ли попасть по этим дорогам из 1-го пункта в  $n$ -ый, и напечатать маршрут.

Информация о дорогах задается в виде матрицы  $A$  размером  $n \times n$ . Пункты  $i$  и  $j$  соединены дорогой, если  $A[i, j]=1$  (и  $A[j, i]=1$ ), иначе  $A[i, j]=0$ . Считаем также, что  $A[i, i]=0$ .

```

type
  path=array[1..n] of 0..n; {представление маршрута;
  ненулевые элементы массива - пройденные населенные
  пункты в маршруте}

procedure add(var p:path;k:integer);
{добавление пункта в путь}
  var i:integer;
  begin
    i:=1;
    while (i<n) and (p[i]<>0) do i:=i+1;
      p[i]:=k;
  end;

function good(i:integer;p:path):boolean;
{функция проверяет, принадлежит ли пункт i маршруту
p}
  var j:integer;
  begin
    good:=true;
    for j:=1 to n do   if p[j]=i then good:=false
  end;

procedure print (p:path);
{печать маршрута p}
var i: integer;
begin
  for i:=1 to n do
    if p[i]<>0 then write(p[i], ' ')
end;

procedure search(p:path;x:integer);
{x - населенный пункт, p - пройденный путь}
  var i:integer;
  begin
    if x=n then begin add(p,x);print(p) end
    else for i:=1 to n do
      if (A[x,i] =1) and good(i,p)
        then

```

```

begin add(p,x) ; search(p,i) end
end;

var A:array[1..n,1..n] of 0..1;
    p:path;
    i:integer;
begin
  {ввод матрицы A}
  for i:=1 to n do p[i]:=0;
  search(p,1)
end.

```

### Быстрая сортировка

Рекурсия дает лучший из известных до сего времени метод сортировки массивов. Он обладает столь блестящими характеристиками, что его изобретатель К. Хоар окрестил его быстрой сортировкой.

Пусть дан массив целых чисел:

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

Попробуем рассмотреть следующий алгоритм: выберем случайным какой-то элемент массива (назовем его  $x$ ), просмотрим массив, двигаясь слева направо, пока не найдем элемент  $a[i] > x$ , а затем просмотрим его справа налево, пока не найдем элемент  $a[j] < x$ . Теперь поменяем местами эти два элемента и продолжим процесс «просмотра с обменом», пока два просмотра не встретятся где-то в середине массива. В результате массив разделится на две части: левую – с элементами, меньшими, чем  $x$ , и правую – с элементами, большими  $x$ . Теперь запишем этот алгоритм в виде процедуры.

```

procedure partition;
  var w,x:integer;
  begin
    i:=1;j:=n;
    выбор случайного элемента x;
    repeat
      while a[i]<x do i:=i+1;
      while x<a[j] do j:=j-1;
      if i<=j then

```

```

begin
  w:=a[i];a[i]:=a[j];a[j]:=w;
  i:=i+1;j:=j-1
end
until i>j
end

```

Заметим, что отношения  $>$  и  $<$  заменены на  $\geq$  и  $\leq$ , отрицания которых в операторе цикла с предусловием – это  $<$  и  $>$ . При такой замене  $x$  действует как барьер для обоих просмотров. Если, например, в качестве  $x$  выбрать средний элемент, равный 42, из массива

44 55 12 42 94 6 18 67,

то для того чтобы разделить массив, потребуется два обмена: 18 на 44 и 6 на 55

18 6 12 42 94 55 44 67,

конечные значения индексов равны  $i=5$  и  $j=3$ . Элементы  $a[1], \dots, a[i-1]$  меньше или равны  $x=42$ , элементы  $a[j+1], \dots, a[n]$  больше или равны  $x$ . Следовательно, мы получили два подмассива:

$a[k] \leq x$  для  $k=1, \dots, i-1$ ,

$a[k] \geq x$  для  $k=j+1, \dots, n$

и, следовательно,

$a[k]=x$  для  $k=j+1, \dots, i-1$ .

Теперь нам пора вспомнить, что наша цель – не только разделить массив на большие и меньшие, но также рассортировать его. Однако от деления до сортировки всего небольшой шаг: разделив массив, нужно сделать то же самое с обеими полученными частями, затем с частями этих частей и т.д., пока каждая часть не будет содержать только один элемент. Этот метод представим программой:

```

procedure quiksort;
  {быстрая сортировка массива a[i] целых чисел}
type
  index:1..n;

procedure sort(l,r:index);
  var i,j:index;
      w,x:integer;

```



```

begin
  i:=1;j:=r;
  x:=a[(1+r) div 2];
  repeat
    while a[i]<x do i:=i+1;
    while x<a[j] do j:=j-1;
    if i<=j then
      begin
        w:=a[i];a[i]:=a[j];a[j]:=w;
        i:=i+1;j:=j-1
      end
    until i>j;

    if l<j then sort(1,j);
    if i<r then sort(i,r)
  end;{sort}

begin
  sort(1,n)
end;{quicksort}

```

Анализ алгоритма показывает, что общее число сравнений и обменов пропорционально  $n \log n$ .

«... Витькины дубли на столе продолжали работать. Самый маленький был уже ростом с муравья».

А. и Б. Стругацкие  
«Понедельник начинается в субботу»

## 13 ЗАПИСИ

### 13.1 Определение комбинированных типов

*Запись (комбинированный тип)* есть абстракция конечной последовательности элементов, но, в отличие от массивов, объединяет значения различных типов.

Пример. Для адекватного представления информации о некотором студенте необходима структура данных, состоящая из следующих элементов:

- фамилия, имя, отчество – строки;
- пол – перечислимый тип (два значения);
- индекс специальности – целое.

*В отличие от массивов, компоненты записи обозначаются идентификаторами.*

```
type
  person = record
    name, secondName, surName: string;
    sex : (male, female);
    speciality: integer
  end;
```

Элементы записи называются *полями*. Каждое описание поля выглядит как описание простой переменной.

**Структура записи:**

- фиксированное число полей;
- каждое поле имеет имя (уникальное в пределах записи, но может совпадать с любым идентификатором вне этой записи);
- каждое поле имеет произвольный тип.

Описание переменных комбинированного типа:

```
var
  Sasha, Masha: person;
```

Доступ к полям записи осуществляется с помощью *селектора записи* – конструкции вида

<переменная типа записи>. <идентификатор поля>

Использование:

```
Sasha.name:='Александр';
Masha.name:='Мария';
Sasha.sex:=male;
Masha.sex:=female;
Masha.Speciality:=Sasha.Speciality;
```

Можно создавать сложные структуры данных: массив записей или запись, в состав которых входят записи.

```
var
    group:array[1..20] of person;
Доступ к полям записей, составляющих этот массив:
    group[i].sex:=female;
    if group[j].name='Борис' then
writeln(group[j].surName);
```

Или, например, пусть требуется хранить информацию о дате рождения некой персоны.

```
type
    date=record
month: (jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov,
dec);
        day:1..31;
        year:1900..2005
    end;
    person=record
        name, secondName, surName:string;
        sex: (male, female);
        speciality:integer;
        birthDay:date
    end;
```

Доступ к полям из элемента birthDay осуществляется с помощью повторного селектора записи:

```
Sasha.birthDay.year:=1970;
Masha.birthDay.month:=feb;
```

Пример: определение комплексных чисел с помощью записи.

```

type
  complex = record
    re,im:real
  end;

procedure addc(c1,c2:complex;var c3:complex);
  {сложение комплексных чисел}
begin
  c3.re:=c1.re+c2.re;
  c3.im:=c1.im+c2.im;
end;

```

### 13.2 Записи с вариантами

Довольно часто бывает необходимо в пределах одной записи иметь различную информацию в зависимости от конкретного значения некоторого поля. Например, если человек мужчина, то в запись входят время прохождения очередных воинских сборов и информация о том, курит он или нет; если же это женщина, то информация о цвете глаз.

Возможный способ – ввести два типа PersonMale и PersonFemale, но лучший способ следующий.

Комбинированный тип, помимо фиксированного списка полей, может содержать *вариантную часть*, т.е. разные переменные, относящиеся к одному и тому же комбинированному типу, могут иметь отличающуюся структуру:

- различное число компонент;
- разные типы компонент.

Пример:

```

type
  personSex = (male,female);
  person = record
    name,secondName,surName:string;
    speciality: integer;
    birthDay: date ;
    case sex: personSex of {sex - дискриминант
- поле - признак выбора варианта}
      male: (army:date;smoking:boolean); {альтернатива}

```

```
female: (eyesColor: (blue, brown, gray, green)) {альтернатива}
end;
```

### Запомните:

- Начало вариантной части отличается служебным словом case.
- После определения поля – признака выбора варианта (дискриминанта) – служебное слово of.
- Вариантная часть вместе с записью заканчивается служебным словом end.
- В записи может быть только одна вариантная часть, и она должна находиться в конце записи.
- Альтернативы вариантной части помечаются допустимыми значениями поля-дискриминанта.
- Тип дискриминанта должен задаваться только идентификатором.
- Для некоторых значений дискриминанта вариант может отсутствовать, тогда после двоеточия ставится () – пустой список.

Пример использования записи с вариантной частью:

```
if groop[i].sex=male then groop[i].smoking:=true
else groop[i].eyesColor:=gray;
```

*Любой вариант может содержать вариантную часть, которая также должна располагаться в конце списка полей данного варианта.*

Заметим, что

1) для переменной комбинированного типа отводится фиксированный объем памяти, и если в записи есть варианты, то этот объем определяется по самому большому варианту;

2) Паскаль не содержит никаких средств контроля за правильностью работы с вариантами записей.

Возможно ошибочное присвоение (компилятором не распознается):

```
Masha.sex:=female;
Masha.smoking:=true;
```

Считается, что за соответствием текущего дискриминанта доступу к полям записи должен следить программист.

### Синтаксические диаграммы

<комбинированный тип>::=

```

---->record-----> <список полей> ---->end----->
      |----->----->|

```

<список полей>::=

```

--->-<фиксированная часть>-->;--->-<вариантная часть>---->;----->
|----->----->| |----->----->|

```

<фиксированная часть>::=

```

<-----> , <----->
|               |
-----> <идентификатор> ----->:-----> <тип> ----->----->
|               |
<-----><-----> ; <-----><----->

```

<вариантная часть>::=

```

-----> case --> <дискриминант> --> of -----> <альтернатива> ----->
                        |               |
                        <-----> ; <----->

```

<дискриминант>::=

```

-----> < идентификатор поля> ----> <идентификатор типа> ----->

```

<альтернатива>::=

```

-----> <константа> ----> : ----> ( ----> <список полей> ----> ) ----->
|               |               |               |
<-----> , <----->         ---->----->----->

```

### 13.3 Оператор над записями with (оператор присоединения)

Следующий пример показывает, что в некоторых случаях использование селектора записи приводит к громоздким выражениям:

```

var myDate:date;
.....
{ переход к следующему месяцу}
if myDate.month = dec then
    begin
        myDate.month := jan;
        myDate.year := myDate.year+1
    end
else myDate.month := succ(myDate.month) ;

```

Для более краткой записи действий над полями записи используется оператор **with**. **Оператор присоединения (with)** «выносит» общий для всех составных имен идентификатор записи в отдельный заголовок, после которого поля записи указываются *только их идентификаторами*.

```

with myDate do
    if month=dec then begin
        month:=jan;
        year:=year+1
    end
else month:=succ(month) ;

```

В общем виде:

```

with <переменная записи> do
    begin
        .....
    end;

```

Примеры:

1. Комбинированный тип для представления экзаменационной ведомости (предмет, номер группы, дата экзамена, 25 строчек с полями: фамилия студента, номер его зачетной книжки, оценка за экзамен).

```

type
    ведомость=record
        предмет:string;
        номергруппы:integer;
        дата: record

```

```

        число:1..31;
        месяц:1..12;
        год:integer
    end;
    студенты: array[1..25] of
        record
            фамилия:string;
            номерзачкнижки:integer;
            оценка:2..5
        end
    end;
end;
```

Использование русского языка в идентификаторах связано только с повышением ясности определения.

2. Комбинированный тип для представления игральной карты.

```

type
    масть = (пики, трефы, бубны, червы);
    достоинство = (шесть, семь, восемь, девять,
        десять, валет, дама, король, туз);
    карта = record
        m:масть;
        d:достоинство
    end;
```

Логическая функция kill(k1, k2, km) проверяет, «бьет» ли карта k1 карту k2 с учетом того, что масть km является козырной.

```

function
kill(k1,k2:карта;km:масть):boolean;
begin
    if k1.m = k2.m then kill:= k1.d > k2.d
        else kill:= k1.m = km
    end;
```



## 14 ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

### 14.1 Ссылочный тип

Характерная особенность рекурсивных структур данных, которая отличает их от основных структур (массивов, записей, множеств), – их способность изменять размер. Поэтому для рекурсивно определенных структур невозможно установить фиксированный размер памяти, и поэтому транслятор не может приписать компонентам такой переменной определенные адреса. Для решения этой проблемы чаще всего применяется метод *динамического распределения памяти*, т.е. выделение памяти для отдельных компонент в тот момент, когда они появляются во время выполнения программы, а не во время трансляции. В этом случае транслятор выделяет фиксированный объем памяти для хранения адреса динамически размещаемой компоненты, а не самой компоненты.

К сожалению, при программировании на языке Паскаль, программист вынужден сам решать вопросы динамического распределения памяти, используя ссылочный тип данных.

*Ссылочный тип* определяется в следующем виде:

```
type
  <имя ссылочного типа> = ^<имя базового типа>;
```

Примеры:

```
type
  mas=array[1..10] of integer;
  Ptr= ^integer;
  link = ^mas;
  linkchar = ^char;
  tie = ^real;
```

Описание переменных ссылочного типа:

```
var
  p:ptr;
  v:link;
  a:^real;
```

Значение ссылочного типа (неформально) – адрес в памяти, где располагается конкретное значение базового типа. Есть специальный указатель, который «никуда не указывает». В адресном пространстве оперативной памяти выделяется один адрес, в котором заведомо не может быть размещена никакая переменная. На это место в памяти и ссылается такой *пустой или «нулевой»* указатель, который обозначается служебным словом **nil**.

Указатель **nil** считается константой, принадлежащей любому ссылочному типу. Иными словами, это значение можно присваивать любому указательному типу.

### Операции над значениями ссылочного типа

- Сравнение на равенство (=), сравнение на неравенство (<>) – ссылаются ли два указателя на одно и то же место в памяти?

Примеры:

```
var p1,p2:ptr;
    .....
    sign:=p1=p2;
    if p1<>nil then ....
```

- Разыменование – доступ к переменной по указателю.

Для того чтобы по указателю на переменную получить доступ к самой этой переменной, необходимо после переменной-указателя поставить знак '^'; **p1^** есть «переменная», на которую ссылается **p1**. Такая конструкция может находиться в любом контексте, в котором допустимо нахождение самой указываемой переменной. Если **p1** имеет тип **^integer**, то **p1^** имеет тип **integer**.

Разыменование некорректно, если ссылочная переменная имеет значение **nil**.

Поэтому

```
p1:=nil;
p1^:=2;
```

некорректно и приводит к трудно распознаваемым ошибкам.

Пример:

```
var p1,p2:^integer;
```

Пусть переменные  $p1^{\wedge}$  и  $p2^{\wedge}$  уже имеют значения 1 и 2 соответственно (как это сделать – смотрите ниже). Тогда различные результаты следующих присваиваний изображены на рис. 3.

```
p1:=p2;  
p1^:=p2^;
```

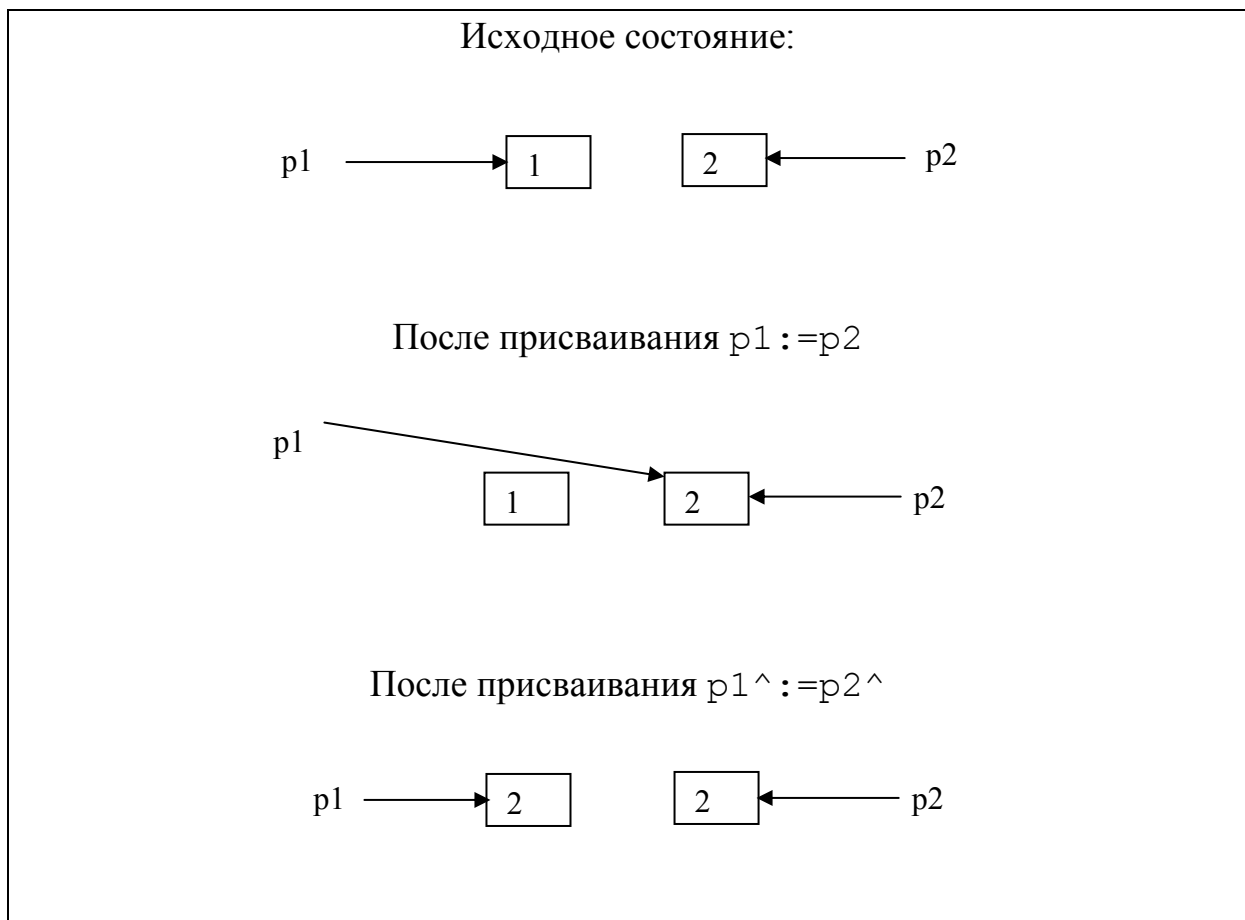


Рис. 3 – Различия в присваиваниях

## 14.2 Статические и динамические переменные

Память для локальных переменных отводится при вызове подпрограммы, при выходе память освобождается. Память для глобальных переменных отводится в начале выполнения подпрограммы. Таким образом, память отводится под переменные с учетом статической структуры программы, поэтому такие переменные называются *статическими*.

*Динамические* переменные – это те, которые образуются и уничтожаются в произвольный момент выполнения программы.

Средство доступа к статическим переменным есть идентификатор. Динамические переменные, количество которых и место расположения в памяти неизвестны, невозможно обозначить идентификаторами. Средство доступа к динамическим переменным – указатель на место их текущего расположения в памяти.

### Создание и уничтожение динамических переменных

Процедура `new (<переменная ссылочного типа>)` предназначена для создания динамической переменной:

- в динамической области памяти отводится место для хранения переменной, тип которой совпадает с базовым типом указателя-переменной;
- переменной, переданной в параметре, присваивается указатель на отведенную область памяти.

Пример:

```
type
  mas=array[1..10] of integer;  link=^mas;
var   t:link;
.....
new(t);
```

Отводится память, достаточная для хранения массива типа **mas**. Переменной **t** присваивается адрес этой памяти. Теперь возможен доступ к элементам массива, например:

```
t^[i]:=0;
t^[i]:=t^[j];
```

Переменная **t** является статической, место в памяти для ее значения выделено при трансляции. Переменная **t^** – динамическая, она появляется только при выполнении процедуры **new(t)**.

Процедура `dispose (<переменная ссылочного типа>)` служит для освобождения памяти, отведенной с помощью процедуры `new` с той же переменной.

Рис. 4 иллюстрирует применение процедур **new** и **dispose** (переменные **p1** и **p2** имеют тип **^integer**):

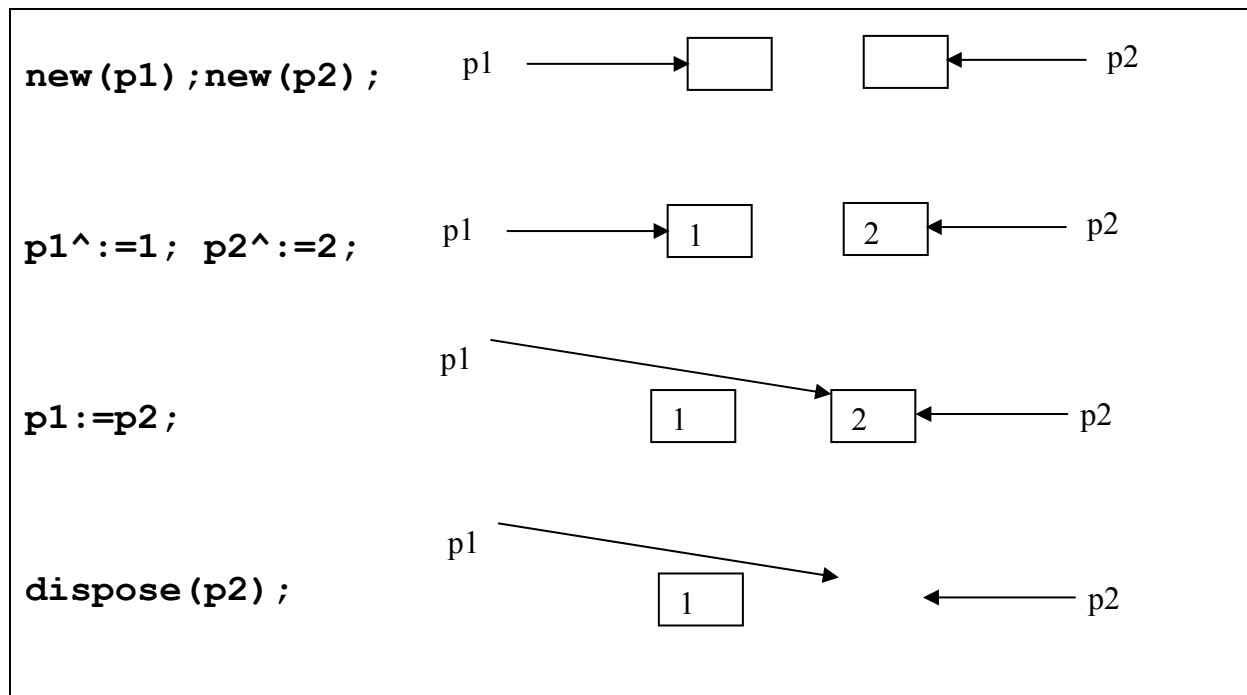


Рис. 4 – Результат выполнения фрагмента

### 14.3 Линейные списки

Рассмотрим создание и обработку структур данных, компоненты которых связаны явными ссылками. Особое значение придается структурам простой формы; приемы работы с более сложными структурами можно получить из способов работы с основными видами структур: *линейными списками и деревьями*.

Самый простой способ соединить, или связать, множество элементов – это расположить их линейно в списке. В этом случае каждый элемент содержит только одну ссылку, связывающую его со следующим элементом списка.

Пусть тип `link` описан следующим образом:

```

type
  link = ^node;
  node = record
    info:string;
    next:link
  end;
var
  s,p:link;
```

Мы можем создать с помощью этого типа список, изображенный на рис. 5:

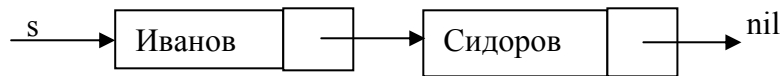


Рис. 5 – Список элементов типа **link**

Переменная – ссылка *s* – указывает на первую компоненту списка. По-видимому, самое простое действие, которое можно выполнить со списком, показанным на рисунке 5, это вставить в его начало некоторый элемент (рис. 6).

```

new(p) ;
p^.info:='Петров' ;
p^.next:=s ;
s:=p ;
  
```



Рис. 6 – Вставить элемент в начало списка

### Формирование списка

Операция *включения элемента* в начало списка определяет, как можно построить список: начиная с пустого списка, последовательно добавляя элементы в его начало.

Пусть число связываемых элементов равно *n*. Тогда формировать список можно следующим образом:

```

s:=nil; {начали с пустого списка}
while n>0 do
  begin
    new(p) ; p^.next:=s ;
    read(p^.info) ;
    s:=p ; n:=n-1
  end ;
  
```

Это самый простой способ построения списка. Но при этом полученный порядок элементов обратен порядку их «поступления». В некоторых случаях это нежелательно; следовательно, новые элементы должны добавляться в конец списка.

### Добавление в конец списка

```

procedure add (var s:link; m:string);
{s указывает на голову списка}
var t,p:link;
begin
  if s=nil then begin
    new(s);
    s^.info:=m;
    s^.next:=nil
  end
else begin
  t:=s;
  while t^.next <> nil do t:=t^.next;
  new(p);
  t^.next:=p;
  p^.info:=m;
  p^.next:=nil;
end;
end;
var
  s:link; m:string; i:integer;
begin
  s:=nil;
  for i:=1 to 10 do begin
    read(m);
    add(s,m)
  end;
end.

```

Хотя конец списка легко найти проходом по списку, такой непосредственный подход требует затрат, которые просто избежать, используя вторую ссылку  $q$ , которая всегда указывает на последний элемент.

## Рекурсивная процедура добавления элемента в список

```

procedure addrec(var s:link;m:string) ;
  begin
    if s=nil then begin
      new(s) ; s^.info:=m; s^.next:=nil
    end
    else addrec(s^.next,m)
  end;

```

## Включение в список

Предположим, что элемент, на который указывает ссылка  $q$ , нужно включить в список после элемента, на который указывает ссылка  $p$ . Необходимые присваивания значений ссылкам:

```

q^.next:=p^.next;
p^.next:=q;

```

Если требуется включение перед элементом, указанным  $p^{\wedge}$ , а не после него, то кажется, что однонаправленная цепочка связей создает трудность, поскольку нет «прохода» к элементам, предшествующим данному. Однако простой «трюк» позволяет решить эту проблему:

```

k:=q^.info;
q^:=p^;
p^.info:=k;
p^.next:=q;

```

«Трюк» состоит в том, что новая компонента в действительности вставляется после  $p^{\wedge}$ , но затем происходит обмен значениями между новым элементом и  $p^{\wedge}$ .

## Удаление из списка

```

procedure del(var t:link;m:string) ;
  {из списка t убирается элемент m}
var
  p1,p:link;

```



```

begin
  if t^.info=m then begin
    p:=t^.next;
    dispose(t);
    t:=p
  end

  else begin
    p:=t;
    while (p^.info<>m) and (p^.next<>nil) do
      begin p1:=p; p:=p^.next end;
    if p^.info=m then begin
      p1^.next=p^.next;
      dispose(p)
    end
  end
end;

```

### Рекурсивная процедура удаления элемента из списка

```

procedure delrec(var s:link;m:string);
var t:link;
begin
  if s<>nil then begin
    if s^.info=m then begin
      t:=s^.next;
      dispose(s);
      s:=t
    end
    else delrec(s^.next,m)
  end
end;

```

### Просмотр списка

Перейдем к основной операции прохода по списку. Предположим, что операция **F(x)** должна выполняться с каждым элементом списка, первый элемент которого есть **p^**. Эту задачу можно выразить следующим образом:

```

while список, на который указывает p, не пуст do
begin
  выполнить операцию F;

```

```

перейти к следующему элементу
end

```

Подробнее это действие описывается оператором

```

while p<>nil do
  begin  F(p^); p:=p^.next end

```

Из определения оператора цикла с предусловием и списковой структуры следует, что **F** будет выполнено для всех элементов списка и ни для каких других.

Очень частая операция – *поиск* в списке элемента с заданным ключом *x*. Поиск ведется строго последовательно. Он заканчивается, либо когда элемент найден, либо когда достигнут конец списка. Снова предположим, что начало списка обозначено ссылкой *p*. Соответствующий цикл следующий

```

while (p<>nil) and (p^.info<>x) do
  p:=p^.next;

```

#### 14.4 Проблема потерянных ссылок

Работа с динамическими переменными через указатели требует большей тщательности и аккуратности при проектировании программ. В частности, следует стремиться освобождать выделенные области сразу же после того, как необходимость в них отпадает, иначе «засорение» памяти ненужными динамическими переменными может привести к быстрому ее исчерпанию.

Кроме того, необходимо учитывать еще одну проблему, связанную с противоречием между стековым принципом размещения статических переменных и произвольным характером создания и уничтожения динамических переменных. Рассмотрим следующий схематический пример программы:

```

type
  PPerson=^Person;
  Person= record
    .....
  end;

```

```

    procedure GetPerson;
    var
        P: PPerson;
    begin
        new(P)
    end;

begin
    Writeln(MemAvail);
    GetPerson;
    Writeln(MemAvail)
end.

```

Вызов `New` в процедуре **GetPerson** приводит к отведению памяти для динамической переменной типа `Person`. Указатель на эту переменную присваивается переменной `P`. Рассмотрим ситуацию, возникающую после выхода из процедуры `GetPerson`. По правилам блочности все локальные переменные подпрограммы перестают существовать после ее завершения. В нашем случае исчезает локальная переменная **P**. Но, с другой стороны, область памяти, отведенная в процессе работы **GetPerson**, продолжает существовать, так как освободить ее можно только явно, посредством процедуры **Dispose**. Таким образом, после выхода из **GetPerson** отсутствует какой бы то ни было доступ к динамической переменной, так как единственная «ниточка», связывающая ее с программой (указатель `P`), оказалась потерянной при завершении **GetPerson**. Вывод на печать общего объема свободной памяти до и после работы **GetPerson** подтверждает потерю определенной области.

Турбо-паскаль, как и многие другие языки программирования, не имеет встроенных средств борьбы с засорением памяти неиспользуемыми динамическими переменными. Во всяком случае, нужно придерживаться правила, согласно которому при выходе из блока необходимо или освободить все созданные в нем динамические переменные, или сохранить каким-то образом ссылки на них (например, присвоив эти ссылки глобальным переменным).

К описанной проблеме примыкает коллизия другого рода, заключающаяся в ситуации, когда некоторая область памяти ос-

вобождена, а в программе остался указатель на эту область. Например, пусть ссылка  $p$  указывает на элемент списка и был выполнен оператор  $p := \text{nil}$  или  $\text{dispose}(p)$ . Несмотря на это, можно (неправильно) использовать далее в программе выражение  $p^{\wedge}.\text{next}$ , но его значение непредсказуемо.

## 14.5 Циклические списки

*Циклически связанный* список (сокращенно – *циклический список*) обладает той особенностью, что связь его последнего узла не равна  $\text{nil}$ , а идет назад к первому узлу списка (рис. 7). В этом случае можно получить доступ к любому элементу, находящемуся в списке, отправляясь от любой заданной точки; одновременно мы достигаем также полной симметрии, и теперь нам уже не приходится различать в списке «последний» или «первый» узел.

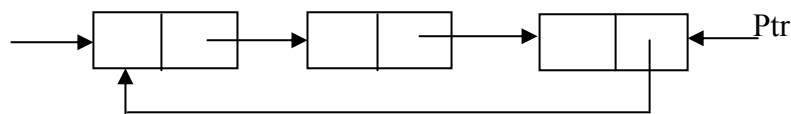


Рис. 7 – Пример циклического списка

Предположим, в узлах имеются два поля **info** и **next**. Переменная связи **Ptr** указывает на самый правый узел списка, а **Ptr<sup>^</sup>.next** является адресом самого левого узла.

Следующие операции относятся к числу наиболее важных:

a) включить **Y** слева:

```
new(p) ; p^.info := Y ;
p^.next := Ptr^.next ; Ptr^.next := p ;
```

b) включить **Y** справа: сначала включить **Y** слева, а затем  $\text{Ptr} := p$ ;

c) присвоить **Y** значение из левого узла и исключить узел:

```
p := Ptr^.next ; Y := p^.info ;
Ptr^.next := p^.next ; dispose(p) ;
```

Внимательный читатель заметит, что мы допустили серьезную ошибку в операциях (a), (b) и (c). Мы не учли, что список может быть пустым (**P<sub>tr</sub>=nil**). Для этого необходимо ввести очевидные изменения.

Циклические списки можно использовать не только для представления структур, которым свойственна цикличность, но также для представления линейных структур; циклический список с одним указателем на последний узел, по существу, эквивалентен простому линейному списку с двумя указателями на начало и конец. В связи с этим наблюдением возникает естественный вопрос: *как найти конец списка, имея в виду круговую симметрию?* Пустой связи **nil**, которая отмечает конец, не существует. Проблема решается так: если мы выполняем некоторые операции, двигаясь по списку от одного узла к следующему, то мы должны остановиться, когда мы вернулись к исходному месту (предполагая, конечно, что исходное место все еще присутствует в списке).

## 14.6 Списки с двумя связями

Чтобы достичь еще большей гибкости в работе с линейными списками, мы можем включить в каждый узел две связи, указывающие на элементы, находящиеся по обе стороны от данного узла (рис 8).

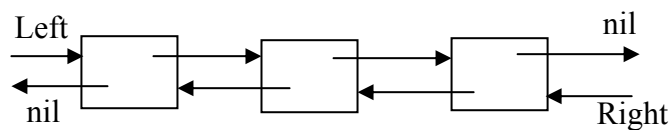


Рис. 8 – Список с двумя связями

Здесь **Left** и **Right** – переменные, указывающие на левый и правый концы списка. В каждом узле списка имеются две связи, назовем их, например, **Llink** и **Rlink**. Списки с двумя связями занимают обычно больше пространства памяти, чем односвязные. Дополнительные операции, которые можно теперь эффективно выполнять, дают часто более чем достаточную

компенсацию за повышенные требования к памяти. Очевидное достоинство состоит в том, что при просмотре списков с двумя связями можно продвигаться как вперед, так и назад. Еще одним новым важным преимуществом является возможность исключения узла, на который указывает ссылка **X**, из списка, в котором он находится, если задано только значение **X**:

```
(X^.Llink) ^.Rlink:=X^.Rlink;  
(X^.Rlink) ^.Llink:=X^.Llink;  
dispose (X) ;
```

Совершенно аналогично в список с двумя связями можно легко включить узел, соседний с узлом **X**, как слева, так и справа. Операторы

```
new (P) ; P^.Rlink:=X^.Rlink; P^.Llink:=X;  
(X^.Rlink) ^.Llink:=P; X^.Rlink:=P;
```

выполняют такое включение справа от узла **X**. Меняя местами левые и правые связи, мы получаем соответствующий алгоритм для включения слева.

## 14.7 Стеки и очереди

Очень часто встречаются линейные списки, в которых включение, исключение или доступ к значениям почти всегда производятся в первом или последнем узлах, и мы дадим им специальные названия.

*Стек* — линейный список, в котором все включения и исключения (и обычно всякий доступ) делаются в одном конце списка (рис. 9).

*Очередь* — линейный список, в котором все включения производятся на одном конце списка, а все исключения (и обычно всякий доступ) делаются на другом его конце (рис. 10).

Из стека мы всегда исключаем «младший» элемент из имеющихся в списке, т.е. тот, который был включен позже других («last-in-first-out» — «последним пришел — первым ушел»). Для очереди справедливо в точности противоположное правило: ис-

ключается всегда самый «старший» элемент; узлы покидают список в том порядке, в котором они в него вошли («first-in-first-out» – «первым пришел – первым ушел»).

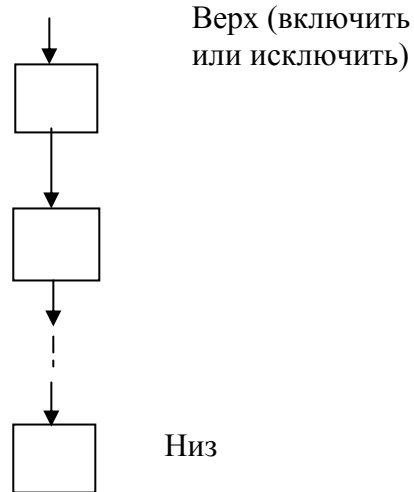


Рис. 9 – Стек

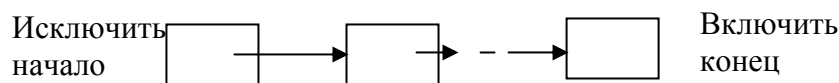


Рис. 10 – Очередь

Стеки очень часто встречается на практике. При решении задач наш мозг ведет себя, как «стек»: одна проблема приводит к другой, а та, в свою очередь, к следующей; мы накапливаем в стеке эти задачи и подзадачи и удаляем их по мере того, как они решаются. Аналогично процесс входов в подпрограммы и выходов из них при выполнении машинной программы подобен процессу функционирования стека. Стеки особенно полезны при обработке языков, имеющих архитектуру вложений. К ним относятся языки программирования, арифметические выражения. Вообще, стеки чаще всего возникают в связи с алгоритмами, имеющими явно или неявно рекурсивный характер.

## Использование стеков для перевода рекурсивных программ к нерекурсивному виду

Вспомним рекурсивную программу о Ханойских башнях.

```

procedure Tower(i,m,n,p:integer) ;
  {перекладываем i верхних колец с иглы m на иглу n,
  используя иглу p как вспомогательную}
begin
  if i=1 then writeln('сделать ход' , m'->' ,n)
    else begin
      Tower(i-1,m,p,n) ;
      writeln('сделать ход' , m'->' ,n) ;
      Tower(i-1,p,n,m)
    end
end;

```

Видно, что задача «переложить  $i$  верхних дисков с иглы  $m$  на иглу  $n$ » сводится к трем задачам того же типа: двум задачам с  $i-1$  дисками и к одной задаче с единственным диском. Занимаясь этими задачами, важно не забыть, что еще осталось делать. Для этой цели заведем стек со следующим типом элементов:

```

type
solve = record
    i,m,n,p:integer
end;

```

Каждая такая запись  $\langle i, m, n, p \rangle$  интерпретируется как заказ «переложить  $i$  верхних диска с иглы  $m$  на иглу  $n$ , используя иглу  $p$  как вспомогательную». Заказы упорядочены в соответствии с требуемым порядком выполнения: самый срочный – вершина стека. Стек можно реализовать с помощью линейного списка.

Получаем такую программу:

```

procedure Tower(i,m,n,p:integer) ;
  {перекладываем i верхних колец с иглы m на иглу n,
  используя иглу p как вспомогательную}
  begin
    {сделать стек заказов пустым}
    {положить в стек запись  $\langle i, m, n, p \rangle$ }
  end;

```



```

while {стек непуст} do
  begin
    {удалить верхний элемент, переложив его в <j,x, y,z>}
    if j=1 then writeln('сделать ход', x, '->', y)
      else begin
        {положить в стек записи:
        <j-1,z,y,x>
        <1,x,y,z>
        <j-1,x,z,y>}
        end
      end
  end;

```

Заметим, что первой в стек кладется запись, которую надо выполнять последней.

## 14.8 Определение деревьев

Определим формально *дерево* как конечное множество  $T$ , состоящее из одного или более узлов, удовлетворяющих следующим свойствам:

- Имеется один специальный узел, называемый *корнем* данного дерева.
- Остальные узлы (исключая корень) содержатся в  $m \geq 0$  попарно не пересекающихся множествах  $T_1, \dots, T_m$ , каждое из которых в свою очередь является деревом. Деревья  $T_1, \dots, T_m$  называются поддеревьями данного дерева.

Из нашего определения следует, что каждый узел дерева является корнем некоторого поддерева, которое содержится в этом дереве. Число поддеревьев данного узла называется *степенью* этого узла. Узел с нулевой степенью называется *концевым узлом* или *листом*. Дерево называется *бинарным*, если каждый узел имеет самое большое два поддерева.

В качестве примера рассмотрим бинарные деревья с базовым типом `integer`.

*Бинарное дерево либо пусто, либо состоит из узла, содержащего целое число, и левого и правого поддеревьев, являющихся бинарными деревьями.*

Соответствующее определение типа бинарное дерево, использующее ссылки, следующее:

```

type
  tree = ^node; {ссылка на узел дерева}
  node = record    {узел дерева}
    info:integer;
    left,right:tree
  end

```

На рис. 11 изображено одно из бинарных деревьев. Узел с числом 7 называется корнем дерева. Дерево обычно изображают корнем вверх. Пустые деревья не обозначены. Узлы, имеющие только пустые поддеревья, являются *листьями* (в данном дереве это узлы –15,2,100 и 10).

Каждый узел поддерева имеет свой *уровень*, который определяется рекурсивно:

- уровень корня бинарного дерева равен 0;
- если уровень узла равен  $n$ , то уровни корней левого и правого поддеревьев равны  $n+1$ .

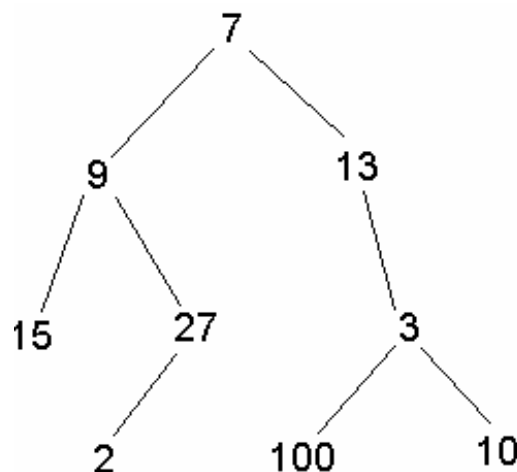


Рис. 11 – Пример бинарного дерева

## 14.9 Обход дерева

До конца раздела (если не оговорено противное) рассматриваются только бинарные деревья, поэтому прилагательное бинарное будем опускать.

Имеется много задач, которые можно выполнять на древовидной структуре: распространенная задача – выполнение заданной операции  $P$  с каждым элементом дерева. Здесь  $P$  рассматривается как параметр более общей задачи посещения всех узлов, или, как это обычно называют, обхода дерева.

Если рассматривать эту задачу как единый последовательный процесс, то отдельные узлы посещаются в некотором определенном порядке и могут считаться расположенными линейно. В самом деле, описание многих алгоритмов существенно упрощается, если можно говорить о переходе к следующему элементу дерева, имея в виду некоторое упорядочение.

Существуют три принципа упорядочения, которые естественно вытекают из структуры деревьев. Так же как и саму древовидную структуру, их удобно выразить с помощью рекурсии. Пусть  $R$  обозначает корень, а  $A$  и  $B$  – левое и правое поддеревья, тогда можно определить такие три упорядочения:

- сверху вниз:  $R, A, B$  (посетить корень до поддеревьев);
- слева направо:  $A, R, B$ ;
- снизу вверх:  $A, B, R$  (посетить корень после поддеревьев).

Обходя дерево из рис. 11 и выписывая числа, находящиеся в узлах, в том порядке, в котором они встречаются, мы получаем следующие последовательности:

- сверху вниз: 7 9 15 27 2 13 3 100 10;
- слева направо: 15 9 2 27 7 13 100 3 10;
- снизу вверх: 15 2 27 9 100 10 3 13 7.

Теперь выразим эти три метода обхода как три конкретные программы с явным параметром  $t$ , означающим дерево, с которым они имеют дело, и неявным параметром  $P$ , означающим операцию, которую нужно выполнить с каждым узлом. Эти три метода легко сформулировать в виде рекурсивных процедур; они вновь служат примером того, что действия с рекурсивно определенными структурами данных лучше всего описываются рекурсивными алгоритмами.

```

procedure preorder(t:tree) ;
begin
    if t <> nil then

```

```

        begin  P(t); preorder(t^.left);
                preorder(t^.right) end
end;

procedure postorder(t:tree);
begin
    if t<>nil then
        begin postorder(t^.left);
                postorder(t^.right); P(t) end
    end;

procedure inorder(t:tree);
begin
    if t<>nil then
        begin inorder(t^.left); P(t);
                inorder(t^.right)
        end
    end;
end;

```

## 14.10 Поиск по дереву с включением

Бинарные деревья часто используются для представления множества данных, элементы которых ищутся по уникальному, только им присущему ключу. Если дерево организовано таким образом, что для каждого узла  $t$  все ключи в левом поддереве меньше ключа  $t$ , а ключи в правом поддереве больше ключа  $t$ , то это дерево называется деревом поиска. В дереве поиска можно найти место каждого ключа, двигаясь, начиная с корня, и переходя на левое или правое поддерево каждого узла в зависимости от значения его ключа.

Возможности техники динамического размещения переменных с доступом к ним через ссылки демонстрируются в задачах, в которых сама структура дерева изменяется, т.е. дерево растет и/или уменьшается во время выполнения программы.

Хорошим примером этого является задача построения частотного словаря. В этой задаче задана последовательность слов и нужно установить число появлений каждого слова. Это означает, что, начиная с пустого дерева, каждое слово ищется в дереве. Если оно найдено, увеличивается его счетчик появлений, если нет —

в дерево вставляется новое слово (с начальным значением счетчика, равным 1).

Для простоты будем использовать целые числа в качестве слов.

Предлагаются следующие описания типов:

```

type
  tree = ^word;
  word = record
    key:integer;
    count:integer;
    left,right:tree
  end;

```

Пусть у нас есть исходный файл ключей **f**, а переменная **root** указывает на корень дерева поиска, мы можем записать программу следующим образом:

```

  root:=nil;
reset(f);
while not eof(f) do
  begin
    read(f,x);
    search(x,root)
  end

```

Процесс поиска с включением представлен в виде рекурсивной процедуры **search**:

```

procedure search(x:integer;var p:tree);
begin
  if p=nil then
    begin {слова нет в дереве; включить его}
      new(p);
      with p^ do
        begin
          key:=x;count:=1;left:=nil;right:=nil end
        end else
      if x<p^.key then search(x,p^.left) else
      if x>p^.key then search(x,p^.right)
      else p^.count:=p^.count+1
    end
  end

```

Для входной последовательности

8 9 11 15 7 3 2 1 5 6 4 10

программа строит бинарное дерево поиска (рис. 12).

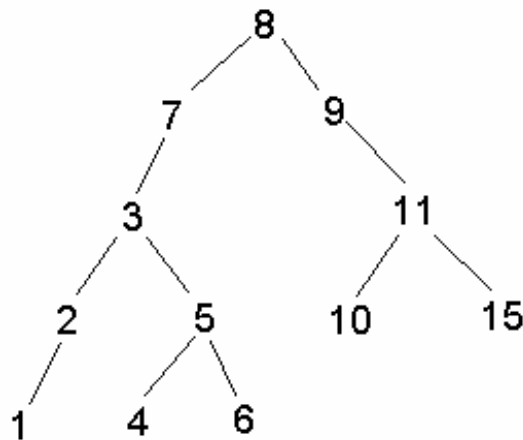


Рис. 12 – Упорядоченное бинарное дерево

### 14.11 Сортировка деревом

Хотя задача предыдущего алгоритма – поиск, его можно применить и для сортировки. Действительно, после того как входной файл ключей запишется в дерево поиска, мы можем получить отсортированную последовательность, обходя дерево методом слева направо и печатая в данном порядке ключи:

```

procedure print(t:tree);
begin
  if t<>nil then
    begin
      print(t^.left);
      write(t^.key:5);
      print(t^.right)
    end
  end

```

Обращение к этой процедуре с параметром, равным корню построенного дерева, выдаст отсортированную последовательность 1 2 3 4 5 6 7 8 9 10 11 15.

## 15 МОДУЛИ

### 15.1 Модульное программирование

Понятие модуля или, в общем случае, модульного программирования, возникло на определенном этапе развития программирования, и было обусловлено, в первую очередь, возрастающими объемами программ, их увеличивающейся внутренней сложностью и коллективным характером разработок. Модули – независимо хранимые и разрабатываемые, независимо компилируемые и тестируемые программные единицы со строго определенными интерфейсами, которые могут объединяться с главной программой.

Модуль – совокупность программных ресурсов (констант, типов, переменных, подпрограмм), предназначенных для использования другими модулями и программами.

Сам модуль не выполняется, используются только его объекты.

Ресурсы модуля делятся на две части:

- Интерфейс – то, что используется в других модулях и программах. Сюда входит описание объектов, доступных (видимых) из других программ.
- Реализация – то, что используется только в данном модуле. Эта часть содержит описание объектов, недоступных (невидимых, скрытых) другим программам.

Модуль в общем виде имеет следующее представление.

```
unit Unitname;
{имя модуля - произвольный идентификатор; этот модуль
должен храниться в файле с тем же именем, в данном
случае, в файле с именем unitname.pas}
interface
{описание видимых объектов}
implementation
{описание скрытых объектов}
begin
{здесь могут присутствовать операторы, инициализирую-
щие объекты модуля; установка значений переменных}
end.
```

Следующий пример модуля содержит только интерфейсную часть.

```
unit Calendar;
interface
  type
    Days = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
    WorkingDays = Mon.. Fri;
    Months = (jan, feb, mar, apr, may, jun, jul,
              aug, sep, oct, nov, dec);
    Summer = jun.. aug;
    Autumn = sep.. nov;
    Spring = mar.. may;
    DayNo = 1.. 31;
    YearNo = 1900.. 2000;
    Date = record
              Day: DayNo;
              Month : Months;
              Year : YearNo;
            end;
implementation
begin
end.
```

Если какая-то функция из модуля будет использоваться в других модулях или программах, то ее определение разбивается на две части. В интерфейс модуля помещается заголовок функции, а в реализацию – ее тело.

Следующий пример модуля реализует действия над комплексными числами.

```
unit Comp;
interface
  type
    Complex = record
              Re, Im: real;
            end;

  procedure InitC (var C: Complex; R, I: real);
    {создание комплексного числа}
```



```

    procedure AddC (C1, C2: Complex; var C3: Complex);
    {сложение целых чисел}
.....
implementation
    procedure InitC (var C: Complex; R, I: real);
    begin
        with C do begin Re:=R; Im:=I end
    end;

    procedure AddC (C1, C2: Complex; var C3: Complex);
    begin
        with C3 do begin
            Re := C1.Re + C2.Re;
            Im  := C1.Im + C2.Im
        end;
    end;
.....
begin
end.

```

Модификация реализации при прежнем интерфейсе никак не отражается на программах, использующих модули.

Прежде чем модуль можно использовать в программах, его необходимо откомпилировать. Если модуль содержится в файле `name.pas`, то в результате компиляции образуется дисковый файл `name.tpu`.

Если программа использует объекты из модулей `u1`, `u2`, `u3`, то первой строкой в программе (после возможного заголовка программы) должна быть директива для компилятора

```
uses u1, u2, u3;
```

Если модуль использует другие модули, то это задается с помощью такой же директивы.

```

unit u;
interface
    uses u1, u2, u3;

```

Пример:

```
uses Comp;
var
  C1,C2,C3: Complex;
begin
  InitC(C1,1,2); InitC(C2,3,-5);
  AddC(C1,C2,C3);
  ...
end.
```

Наиболее часто используемые модули (так называемые стандартные модули) хранятся в библиотеке – файле turbo.tpl. С помощью служебной программы Trimover пользователь может включить в библиотеку и какой-то собственный модуль.

Приведем пример полезного модуля, реализующего операции со стеком.

```
unit StackOps; { операции над стеком целых чисел}
interface
  procedure Push (Elem: integer); {добавление
                                   элемента в стек}
  function Pop: integer; {извлечение элемента из
                         стека}
  function Empty: boolean; {проверка стека на
                           пустоту}
  function Full: boolean; {проверка стека на
                           переполнение}
implementation
  {Стек реализован в виде линейного массива целых эле-
  ментов. Переменная Top отмечает позицию стека над
  вершиной.}
  const   Max=100;
  var
    Stack: array[1..Max] of integer;
    Top: integer;

  procedure Push (Elem: integer);
  begin
    if Top > Max then begin
      writeln('стек переполнен');
      Exit; end;
```

```

    Stack[Top] := Elem;
    inc(Top);
end;

function Pop: integer;
begin
    Pop:=0;
    If Top =1 then begin writeln('стек пуст'); Exit;
                    end;

    dec(Top);
    Pop:= Stack[Top]
end;

function Empty: boolean;
begin Empty := Top=1; end;

function Full: boolean;
begin Full:= Top > Max end;

begin
    Top:=1;{ первоначально стек пуст}
end.

```

## 15.2 Стандартные модули

В Турбо Паскале 7.0 существуют 10 стандартных модулей. Опишем кратко два наиболее часто используемых модуля.

### Модуль System

Модуль System является основной библиотекой, куда входят все predetermined процедуры и функции стандарта языка Паскаль, а также подпрограммы общего назначения (управление вводом-выводом, работа со строками, статической и динамической памятью и т.д.). Модуль System автоматически подключается к любой программе, и его не следует упоминать в разделе объявления используемых модулей uses.

Опишем некоторые функции из этого модуля.

Процедура Halt (ExitCode: word) прекращает выполнение программы, вызывает в случае необходимости подпрограмму завершения и осуществляет выход в операционную систему.

ExitCode – код завершения (при отсутствии этого параметра код завершения считается равным 0).

Процедура Exit осуществляет немедленный выход из текущего блока (подпрограммы); в основной программе Exit действует подобно Halt.

Процедура Randomize инициализирует генератор случайных чисел в зависимости от показания системных часов.

Функция Random формирует случайное число: вызов без аргумента Random выдает случайное вещественное число  $X$  в полуинтервале  $0.0 \leq X < 1.0$ ; вызов с аргументом Random(N: word) выдает целое число  $X$  в полуинтервале  $0 \leq X < N$ .

### **Модуль Crt**

Модуль Crt обеспечивает возможности для доступа к экрану дисплея в текстовом режиме, средства чтения информации с клавиатуры и простейшее управление звуком.

Экран разбивается на 25 строк по 80 позиций. Номера строк и позиций начинаются с 1. В каждой позиции помещается один символ. Для каждого элемента экрана можно задать цвет фона и цвет символа, символ можно сделать мерцающим. Атрибуты символа размещаются в одном байте.

Цвета с кодами от 0 до 7 можно использовать для символов и для фона. Цвета с кодами от 8 до 15 можно использовать только для символов. Константа Blink = 128 используется для задания мигания символа.

- Процедура TextBackGround(Color : Byte) задает цвет фона.
- Процедура TextColor(Color: Byte) задает цвет символа. Вызов TextColor(Color+Blink) устанавливает мигание символов.
- Процедура ClrScr очищает экран.
- Процедура GotoXY(X,Y: Byte) перемещает курсор к элементу экрана с заданными координатами.
- Функции WhereX и WhereY выдают в качестве значения координаты X и Y текущего положения курсора.
- Процедура Delay(Ms:word) задает задержку выполнения программы на Ms миллисекунд.
- Процедура Sound(Hz:word) запускает источник звука с частотой Hz герц.

- Процедура NoSound выключает источник звука.
- Функция KeyPressed:boolean анализирует нажатие клавиши клавиатуры (за исключением вспомогательных клавиш: Shift, Alt, NumLock и т.п.). Если клавиша нажата, то результат равен True.
- Функция ReadKey: char считывает символ с клавиатуры и освобождает буфер клавиатуры от считанного символа.

Пример:

```
repeat  
.....  
until KeyPressed;  
ch:= ReadKey;
```

## 16 ГРАФИЧЕСКОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ПАСКАЛЬ

### 16.1 Аппаратная и программная поддержка графики

#### Адаптер и монитор

Аппаратная поддержка графики персонального компьютера обеспечивается двумя основными модулями: видеоадаптером и телевизионным монитором.

**Телевизионный монитор** (или просто **экран**) – это устройство, на котором появляется вводимый текст или графическое изображение, он работает так же как, как и обычный телевизор. Экран 25 раз в секунду формируется заново. Так как человеческий глаз не способен уловить такое быстрое мелькание кадров, создается иллюзия неподвижного изображения на экране монитора. Изображение на экране строится из небольших точек (пикселей), объединяющихся в телевизионные строки. Соответственно минимальной единицей управления является пиксель.

Число точек на одной телевизионной линии и число самих телевизионных линий различны для разных типов видеоадаптеров. При работе в графическом режиме появляется возможность управлять цветом любого пикселя.

Конструктивно **видеоадаптеры** – это весьма сложные электронные устройства, управляемые собственным микропроцессором. Видеоадаптер конструктивно представляет собой электронную плату, вставляемую в один из разъемов внутри системного блока компьютера.

В самом общем виде видеоадаптер состоит из двух основных частей – **контроллера электронно-лучевой трубки** и **видеопамяти (видеобуфера)**.

#### Типы видеоадаптеров

Для компьютеров IBM PC и совместимых с ними имеется несколько стандартов видеоадаптеров:

- монохроматический адаптер дисплея (Monochrome Display Adapter – MDA);
- цветной графический адаптер (Color Graphics Adapter – CGA);

- усовершенствованный графический адаптер (Enhanced Graphics Adapter – EGA);
- виртуальный графический массив (Virtual Graphics Array – VGA);
- монохроматический адаптер Hercules (Hercules Graphics Card – HGC);
- SVGA

### **Видеобуфер**

Каждая телевизионная точка, или пиксель (PixEL – Picture Element – элемент изображения), имеет закрепленную за ней группу битов (1 или больше), задающих цвет пикселя. В совокупности эти биты образуют видеобуфер (видеопамять, память адаптера дисплея). Основное назначение видеобуфера – хранение образа экрана.

В зависимости от количества битов, выделяемых на один пиксель, можно закодировать 2, 4, 16, 256 цветов.

Видеопамять адаптеров разделена на несколько областей фиксированного размера – видеостраницы.

### **Состав графических средств**

Чтобы сделать процесс графического программирования более эффективным, фирма Borland Int. разработала:

- библиотеку Graph;
- набор драйверов для работы с разными типами мониторов;
- набор шрифтов для вывода на экран текста.

### **Модуль Graph**

Для формирования графических изображений в языке Turbo Pascal предназначен стандартный библиотечный модуль Graph. В нем содержатся 79 графических процедур, функций, десятки стандартных констант и типов данных. Все они составляют единый комплекс средств, позволяющих разрабатывать профессиональные программные продукты.

Подключение модуля Graph к пользовательской программе осуществляется стандартным способом – с помощью зарезервированного слова `uses`:

**`uses graph;`**

С этого момента все графические средства доступны пользователю.

### Драйверы

Графические драйверы разработаны практически для всех существующих видеоадаптеров (см. табл. 16.1). Они находятся в так называемых bgi-файлах и активизируются при инициализации графического режима.

Таблица 16.1 – Графические драйверы

Драйвер	Адаптер
cga.bgi	CGA
egavga.bgi	EGA, VGA
herc.bgi	Hercules

### Шрифты

Вывод текста в графических режимах может осуществляться самыми различными как стандартными, так и пользовательскими шрифтами. По умолчанию после инициализации графического режима устанавливается шрифт DefaultFont. Каждый его символ формируется в матрице 8×8 бит.

Таблица 16.2 – Стандартные шрифты

Шрифт	Файл
TriplexFont	Trip.chr
SmallFont	litt.chr
SanSerifFont	sans.chr
GothicFont	goth.chr

Стандартный набор включает шрифты, приведенные в табл. 16.2. Они размещены в отдельных файлах, имеющих расширение chr. Активизация нужного шрифта осуществляется специальной



процедурой. Высота и ширина символов каждого шрифта могут изменяться с помощью специальных средств.

## 16.2 Инициализация графики

### Видеорежимы

Прежде чем работать с графикой, необходимо установить наиболее подходящий для имеющегося монитора видеорежим. Каждый драйвер может поддерживать от одного до трех видеорежимов. Тип драйвера и режим могут быть заданы как число или как символическая константа типа. Константы, определяющие видеорежим, приведены в табл. 16.3 вместе с информацией о выбираемом режиме и типе видеоадаптера, который может такой видеорежим поддерживать.

Таблица 16.3 – Видеорежимы

Драйвер	Режим	Разрешение	Файл
CGA (1)	CGAC0, CGAHi	320×200 (640×200)	cga.bgi
EGA (3)	EGALo, EGAHi	640×200 (640×350)	egavga.bgi
VGA (9)	VGALo, VGABi	640×200 (640×480)	egavga.bgi
HERC	HERCMONOHi	720×348	herc.bgi

### Инициализация видеорежима

С момента подключения модуля Graph программисту доступны все находящиеся в нем подпрограммы. В первую очередь вызывается процедура InitGraph, которая устанавливает один из возможных графических режимов. Формат процедуры:

```
InitGraph(DriverVar, ModeVar, 'C:\Tp\Bgi');
```

Целочисленные переменные DriverVar и ModeVar задают драйвер и режим в соответствии со значениями, приведенными в табл. 16.3. Например:

```
DriverVar:=VGA; ModeVar:=VGALo;
```

Первый параметр может задаваться как по имени, так и числом (в табл. 16.3 оно указано в скобках рядом с именем драйвера).

Для новичков, которые могут не знать типа дисплея своего компьютера, имеется встроенная константа `Detect` (она имеет нулевое значение). Если это значение присвоено параметру `DriverVar`:

```
DriverVar:=Detect;
```

то `InitGraph` автоматически инициализирует нужный драйвер и устанавливает наиболее подходящий для дисплея режим.

Третий параметр – маршрут к файлу `*.bgi`. Пустая строка означает, что графический драйвер находится в том же каталоге, что и программа.

Пример:

```
Uses Graph;  
var  
    DriverVar, ModeVar: integer;  
begin  
    DriverVar:=Detect;  
    InitGraph(DriverVar, ModeVar, '');
```

### **Закрытие видеорежима**

Когда все запланированные графические работы выполнены, необходимо выйти из графического режима. Это делается с помощью процедуры `CloseGraph`, не имеющей параметров.

В процессе выполнения этой процедуры освобождается память, распределенная под драйверы графики, файлы шрифтов и промежуточные данные, и восстанавливается режим работы адаптера в то состояние, в котором он находился до выполнения инициализации системы.

### **Установка видеостраницы**

Память видеобуфера подразделяется на несколько частей – так называемых видеостраниц. Их количество зависит от текущего режима и типа адаптера. Более одной страницы имеют адаптеры EGA, VGA и Hercules. Нумерация страниц начинается с 0.

В каждый отдельный момент на экране может быть отображена только одна страница, она называется **видимой**. По умолчанию видима страница с номером 0. Страничная организация позволяет с помощью графических процедур и функций формировать изображение на любой из страниц. Страница, на которой в данный момент формируется изображение, называется **активной**.

Таблица 16.4 – Режимы и видеостраницы

Драйвер	Но- мер	Режим	Но- мер	Цвет	Страницы
EGA	3	EGALo	0	16	4
EGA	3	EGAHi	1	16	2
VGA	9	VGALo	0	16	4
VGA	9	VGAMed	1	16	4
HERC	7	HERCMONOH	0	2	2

Процедура SetActivePage(Page: word) устанавливает активную страницу для построения изображения. Например:

**SetActivePage (1) ;**

Построение изображения может производиться незаметно для смотрящего на экран (в этом случае активная страница не совпадает с видимой). Например, страница может формироваться «подкачкой» данных с диска или с помощью любых процедур Turbo Pascal. Сформировав страницу, ее можно показать на экране с помощью процедуры

**SetVisualPage (Page: word) ,**

где Page – номер видимой страницы. Например,

```
SetActivePage(0); {показ страницы 0 на экране}
OutText('Страница 0'); {строка появляется на экране}
SetActivePage(1); {активная страница}
OutText('Страница 1'); {формирование изображения на
странице 1, но на экране ее нет!}
Readln;
```

```
SetVisualPage(1); {показ страницы 1, строка на экра-  
не}
```

### Обработка ошибок

Графическая программа, как и любая другая, может содержать ошибки. Программист должен предусмотреть все возможное для их своевременного обнаружения и нейтрализации. Для этого имеются две функции: `GraphResult` и `GraphErrorMsg`.

`GraphResult` возвращает значение 0, если последняя графическая операция выполнялась без ошибок, или число в диапазоне  $-15 \dots -1$ , если ошибка была. Некоторые ошибки и их коды приведены в табл. 16.5.

Таблица 16.5 – Коды ошибок

Константа	Значение	Описание
<code>grOk</code>	0	Нет ошибок
<code>grNoInitGraph</code>	-1	Графика не инициализирована (используйте <code>InitGraph</code> )
<code>grNotDetected</code>	-2	Графическое устройство не обнаружено
<code>grFileNotFound</code>	-3	Файл драйвера устройства не найден

В качестве примера рассмотрим следующий фрагмент:

```
uses Graph;  
Var ErrorNumber:integer;  
Begin  
...  
ErrorNumber := GraphResult;
```

В переменной `ErrorNumber` содержится код ошибки. Можно пользоваться как кодом ошибки, так и соответствующей ему константой, например:

```
If ErrorNumber <> grOk then writeln('Обнаружена  
ошибка');
```

GraphErrorMsg возвращает строку сообщения об ошибке, соответствующую коду ошибки. Например, процедура

```
writeln(GraphErrorMsg(ErrorNumber));
```

выведет строку "No error", так как в нашем примере графический режим установлен правильно. Инициализацию графического режима и проверку возможных ошибок удобно осуществлять в отдельной процедуре:

```
procedure Init;
{Процедура инициализации и анализ системных ошибок,
DriverVar и ModeVar описаны в основной программе.}
Begin
    DriverVar:= Detect;
    InitGraph(DriverVar,ModeVar, '');
    ErrorCode := GraphResult;
    If ErrorCode <> grOk then
        begin
            writeln('Графическая системная ошибка',
                GraphErrorMsg(ErrorCode)); Halt(1)
        end
end;
```

## 16.3 Базовые процедуры и функции

### Система координат

Для построения изображения на экране используется **система координат**. Отчет начинается от верхнего левого угла экрана, который имеет координаты (0,0). Значение x (столбец) увеличивается слева направо, значение y (строка) увеличивается сверху вниз. Так в режиме VGAHi адаптера VGA экранные координаты каждого из четырех углов экрана и точки в центре экрана будут представлены следующим образом:

- левый верхний угол – (0,0);
- левый нижний угол – (0,349);
- правый верхний угол – (639,0);
- правый нижний угол – (639,349);
- центр – (320, 174).

### Текущий указатель

Чтобы построить изображение, необходимо указывать, по крайней мере, точку начала ввода. В графическом режиме видимого курсора нет, но есть невидимый **текущий указатель** CP (Current Pointer). Фактически это тот же курсор, но он невидим.

В графическом режиме для перемещения CP имеется ряд процедур и функций. В первую очередь это MoveTo и MoveRel. Процедура MoveTo(x,y) перемещает текущий указатель в точку с координатами x и y. Процедура MoveRel(dx,dy) перемещает CP на dx точек по горизонтали и на dy точек по вертикали.

В ряде программ выполняется постоянный контроль местоположения текущего указателя. Для этого используются функции GetX и GetY, которые возвращают соответственно значение координаты x и координаты y указателя CP. Например:

```
var
    xpos,ypos:integer;
...
xpos := GetX;
ypos := GetY;
```

В процессе управления CP может возникнуть ситуация, когда его координаты выйдут за допустимые пределы. В таких ситуациях используются функции GetMaxX:integer и GetMaxY:integer, которые возвращают соответственно максимально возможные для установленного режима значения координат x и y. Например:

```
x := 6000 div 10;
y := 2000 div 2;
if (x>GetMaxX) or (y>GetMaxY) then writeln('Вне
экрана');
```

Определение центра экрана осуществляется очень просто:

```
xc := GetMaxX div 2;
yc := GetMaxY div 2;
```

Такой способ избавляет пользователя от настройки на конкретный тип монитора и расширяет область применения программы.

### Очистка экрана

Чтобы стереть изображения на экране, т.е. очистить его, используется не имеющая параметров процедура `ClearDevice`. С момента ее выполнения все установки по цвету, фону и т.д. аннулируются и указатель `CP` переходит в точку с координатами (0,0).

### Вывод точки

Какие бы изображения не выводились на экран, все они построены из точек. Имея средство построения точки определенного цвета в нужном месте экрана, теоретически можно создать любое изображение вплоть до картины. В библиотеке `Graph` вывод точки осуществляется процедурой

**`PutPixel(x,y:integer; Color:word),`**

где `x` и `y` – экранные координаты расположения точки, `Color` – ее цвет. Возможные значения `Color` приведены в табл. 16.6. Например, оператор

**`for i:=0 to 59 do PutPixel(i,0,Red)`**

выведет в первую строку экрана 60 красных точек.

Таблица 16.6 – Цветовая шкала

Цвет	Код	Цвет	Код
Black	0	DarkGray	8
Blue	1	LightBlue	9
Green	2	LightGreen	10
Cyan	3	LightCyan	11
Red	4	LightRed	12
Magenta	5	LightMagenta	13
Brown	6	Yellow	14
LightGray	7	White	15

Чтобы узнать цвет точки в конкретной позиции экрана, используется функция `GetPixel(x,y:integer):word`.

### **Вывод линии**

Из точек строятся линии (отрезки прямых). Это делает процедура `Line(x1,y1,x2,y2)`, где `x1` и `y1` – координаты начала, `x2` и `y2` – координаты конца линии. В процедуре `Line` нет параметра для установки цвета. В этом и других аналогичных случаях цвет задается процедурой `SetColor(Color)`, где `Color` – цвет, значение которого берется из табл. 16.6. Например,

```
SetColor(Cyan) ;  
Line(1,1,600,1) ;
```

Для черчения линий применяются еще две процедуры: `LineTo` и `LineRel`. Процедура `LineTo(x,y)` строит линию из точки текущего указателя в точку с координатами (`x`, `y`). Процедура `LineRel(dx,dy)` проводит линию от точки текущего расположения указателя в точку (`CPx+dx`, `Сру+dy`), где `CPx` и `Сру` – текущие координаты `CP`.

Можно вычерчивать линии самого различного стиля: тонкие, широкие, штриховые, пунктирные и т.д. Установка стиля производится процедурой

```
SetLineStyle (LineStyle : word; Pattern: word;  
Thickness: word) ;
```

Параметр `LineStyle` устанавливает стиль линии, возможные значения которого приведены в табл. 16.7; `Pattern` – образец, `Thickness` – толщина линии, определяемая константами, указанными в табл. 16.8. Если применяется один из стандартных стилей, значение `Pattern` равно 0. Например:

```
SetLineStyle(DottedLn,0,NormWidth) ;  
Line(1,1,600,1) ;
```



Если пользователь хочет активизировать свой собственный стиль, то значение `LineStyle` равно 4. В этом случае `Pattern` – двухбайтовое число.

Таблица 16.7 – Стиль линии

Константа	Значение	Описание
<code>SolidLn</code>	0	Непрерывная линия
<code>DottedLn</code>	1	Линия из точек
<code>CenterLn</code>	2	Линия из точек и тире
<code>DashedLn</code>	3	Штриховая линия
<code>UserBitLn</code>	4	Тип пользователя

Таблица 16.8 – Толщина линии

Константа	Значение	Описание
<code>NormWidth</code>	1	Нормальная толщина (1 пиксель)
<code>ThickWidth</code>	3	Жирная линия (3 пикселя)

## 16.4 Работа с текстом

### Вывод текста

Для вывода текста на экран используются процедуры `OutText` и `OutTextXY`. Процедура

**`OutText(TextString : string)`**

выводит строку текста, начиная с текущего положения СР.

Явный недостаток этой процедуры – нельзя указать произвольную точку начала вывода. Его можно устранить с помощью `MoveTo`, но лучше воспользоваться процедурой

**`OutTextXY(x, y, Text) ,`**

где `x`, `y` – координаты точки начала вывода текста, `Text` – константа или переменная типа `string`.

### Вывод численных значений

Для начинающих проблемой является вывод численных данных, ибо в Graph нет предназначенных для этого процедур. Выход прост: сначала преобразовать число в строку с помощью процедуры Str, а затем посредством '+' подключить ее к выводимой OutTextXY строке. Например:

```

Max := 34.56;
Str (Max:6:2, Smax) ;
{результат преобразования находится в Smax}
OutTextXY (400, 40, 'Максимум =' + Smax) ;
{+ - конкатенация}

```

### Установка шрифта

Список имеющихся шрифтов приведен в табл. 16.9. Установить нужный шрифт можно процедурой

```

SetTextStyle (Font:word;Direction:word;CharSize:word) ;

```

где Font – выбранный шрифт, Direction – направление (горизонтальное или вертикальное), CharSize – размер выводимых символов. Возможные значения первых двух параметров представлены в табл. 16.9 и табл. 16.10. При организации вертикального вывода необходимо учитывать, что если программист не установит точку начала вывода с помощью MoveTo, то текст начинается с нижней строки экрана и продолжается вверх. Величину выводимых символов можно устанавливать с помощью коэффициента CharSize. Если CharSize = 1, то символ строится в матрице 8×8, если CharSize = 2, то используется матрица 16×16 и т.д. до 10-кратного увеличения.

Таблица 16.9 – Шрифты

Константа	Значение	Описание
DefaultFont	0	8×8-битовый шрифт
TriplexFont	1	Штриховые шрифты
SmallFont	2	Малый шрифт
SansSerifFont	3	Сансериф
GothicFont	4	Готический

Таблица 16.10 – Константы ориентации

Константа	Значение	Описание
HoizDir	0	Слева направо
VertDir	1	Снизу вверх

## 16.5 Построение графических фигур

### Построение прямоугольников

Для построения прямоугольных фигур имеется несколько процедур. Первая из них – процедура вычерчивания прямоугольника:

```
Rectangle (X1, Y1, X2, Y2: integer) ;
```

где X1, Y1 – координаты левого верхнего угла, X2, Y2 – координаты правого нижнего угла прямоугольника. Это очень полезная процедура, с ее помощью, в частности, можно легко построить любую диаграмму для визуального анализа данных. Область внутри прямоугольника не закрашена и совпадает по цвету с фоном. В качестве примера приведем фрагмент, который выводит на экран 100 вычерченных разным цветом динамически меняющихся по высоте прямоугольников:

```
for i:=1 to 100 do
  begin
    Random(16); {установка цвета}
    Rectangle (200, Random(300), 250, 300) ;
    {i-ый прямоугольник}
    Delay(50); {задержка}
    ClearDevice {очистка экрана}
  end;
```

Более эффектные для восприятия прямоугольники можно строить с помощью процедуры

```
Bar (x1, y1, x2, y2: integer) ;
```

которая рисует закрашенный столбец. Цвет закрашки устанавливается с помощью `SetFillStyle`. Пример использования:

```
SetFillStyle(1,3) ;  
Bar(10,10,50,100) ;
```

Функция `SetFillStyle(pattern:word;color:word)` определяет стиль заполнения. Значения `pattern` приведены в табл. 16.11 и могут быть представлены константой или цифрой, `color` берется из шкалы цветов.

Таблица 16.11 – Стиль заполнения

Константа	Значение	Стиль
<code>EmptyFill</code>	0	Заполнение цветом фона
<code>SolidFill</code>	1	Однородное заполнение цветом
<code>LineFill</code>	2	Заполнение символами "--", цвет – <code>color</code>
<code>LtSlashFill</code>	3	Заполнение символами "/" нормальной толщины, цвет – <code>color</code>
<code>SlashFill</code>	4	Заполнение символами "/" удвоенной толщины, цвет – <code>color</code>
<code>BkSlashFill</code>	5	Заполнение символами "\" удвоенной толщины, цвет – <code>color</code>
<code>LtBkSlashFill</code>	6	Заполнение символами "\" нормальной толщины, цвет – <code>color</code>
<code>HatchFill</code>	7	Заполнение вертикально-горизонтальной штриховкой тонкими линиями
<code>XhatchFill</code>	8	Заполнение штриховкой крест-накрест по диагонали «редкими» тонкими линиями, цвет – <code>color</code>
<code>InterLeaveFill</code>	9	Заполнение штриховкой крест-накрест по диагонали «частыми» тонкими линиями, цвет – <code>color</code>
<code>WideDotFill</code>	10	Заполнение «редкими» точками
<code>CloseDotFill</code>	11	Заполнение «частыми» точками
<code>UserFill</code>	12	Заполнение по определенной пользователем маске заполнения, цвет – <code>color</code>

Еще одна весьма эффектная процедура:

**Bar3D(x1,y1,x2,y2:integer; Depth: word; Top: boolean)**

вычерчивает трехмерный закрашенный прямоугольник. При этом используются тип и цвет закрашки, установленные с помощью процедуры SetFillStyle. Параметр Depth представляет собой число пикселей, задающих глубину трехмерного контура. Чаще всего его значение равно четверти ширины прямоугольника:

**Depth := (x2-x1) div 4;**

параметр Top определяет, строить над прямоугольником вершину (Top=true) или нет (Top=false).

### **Построение многоугольников**

Процедура DrawPoly позволяет строить любые многоугольники линией текущего цвета, стиля и толщины. Она имеет следующий формат:

**DrawPoly(numPoints: word; var PolyPoints);**

Параметр polyPoints является нетипизированным параметром, который содержит координаты каждого пересечения в многоугольнике. Параметр NumPoints задает число координат в PolyPoints. Необходимо помнить, что для вычерчивания замкнутой фигуры с  $n$  вершинами нужно передать при обращении к процедуре DrawPoly  $n+1$  координату, где координата вершины с номером  $n$  будет равна координате вершины с номером 1. Проиллюстрируем сказанное следующей программой:

```
{Программа вычерчиваем в центре экрана треугольник
красной линией}
user Crt,Graph;
var
  DriverVar, ModeVar :integer;
  pp : array[1..4] of PointType;
  {встроенный тип PointType определен как
   record x,y:integer end}
```

```

    xM, yM, xMaxD4, yMaxD4: word;
begin
    DriverVar := Detect;
    InitGraph(DriverVar, ModeVar, '');
    xM := GetMaxX; yM := GetMaxY; xMaxD4 := xM div 4;
    yMaxD4 := yM div 4;
    {определение координат вершин}
    pp[1].x := xMaxD4;
    pp[1].y := yMaxD4;
    pp[2].x := xM - xMaxD4;
    pp[2].y := xMaxD4;
    pp[3].x := xM div 2;
    pp[3].y := yM - yMaxD4;
    pp[4] := pp[1];
    SetColor(LightRed); {цвет для вычерчивания}
    DrawPoly(4, pp); {4 - количество пересечений + 1}
    Readln;
    CloseGraph
end.

```

В результате работы программы на экране появится красный треугольник на черном фоне (рис. 13)

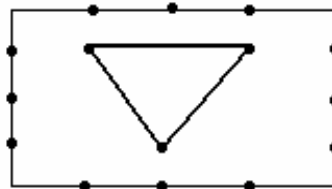


Рис. 13 – Треугольник

Возникает естественное желание его закрасить, т.е. изменить фон внутри треугольника. Это можно сделать с помощью процедуры

```
FillPoly(NumPoints: word; var PolyPoints);
```

Значения параметров те же, что и в процедуре DrayPoly. Действие тоже аналогично, но фон внутри многоугольника закрашивается. В качестве примера нарисует в левой верхней части экрана четырехугольную звезду зеленого цвета (рис. 14):

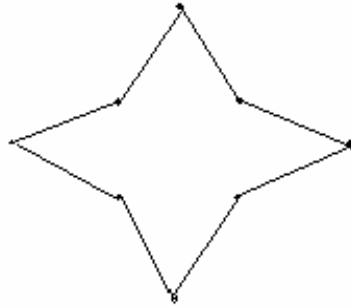


Рис. 14 – Четырехугольная звезда

```

{программа вычерчивает четырехугольную звезду}
user Crt, Graph;
const
    star: array[1..18] of integer = (75,0,100,50,150,75,
                                      100,100,75,150,
                                      50,100,0,75,
                                      50,50,75,0);

var
    DriverVar, ModeVar: integer;
begin
    DriverVar := Detect;
    InitGraph(DriverVar, ModeVar, '');
    SetFillStyle(1,Green);
    FillPoly(9,Star); {9 - количество пересечений + 1}
    Readln;
    CloseGraph
end.

```

### Построение дуг и окружностей

Для задания углов используется полярная система координат (рис. 15):

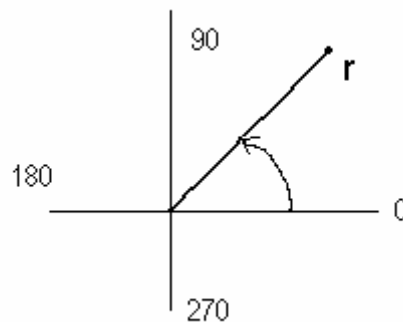


Рис. 15 – Полярная система координат

Процедура вычерчивания окружности текущим цветом имеет следующий формат:

```
Circle(x,y, Radius: word) ;
```

где  $x, y$  – центр окружности, а  $Radius$  – ее радиус. Например, следующий фрагмент обеспечивает вывод ярко-зеленой окружности с радиусом 50 пикселей и центром в точке 450, 100:

```
SetColor(LightGreen) ;  
Circle(450,100,50) ;
```

В ряде случаев, в частности для создания псевдообъемных фигур, используются дуги. Их можно вычертить с помощью процедуры

```
Arc(x,y: integer; StAngle, EndAngle, Radius:  
word) ;
```

где  $x, y$  – центр окружности,  $StAngle$  и  $EndAngle$  – начальный и конечный углы,  $Radius$  – радиус. Цвет для вычерчивания устанавливается процедурой `SetColor`. Очевидно, что если  $StAngle=0$  и  $EndAngle = 360$ , то вычерчивается полная окружность.

Для построения эллиптических дуг предназначена процедура

```
Ellipse(X,Y: integer; StAngle, EndAngle: word; xR,  
yR: word) ;
```

где  $x, y$  – центр эллипса в дисплейных координатах,  $xR$  и  $yR$  – горизонтальная и вертикальная оси. Дуга эллипса вычерчивается от начального угла  $StAngle$  до конечного угла  $EndAngle$  текущим цветом. Значения  $StAngle = 0$  и  $EndAngle = 360$  приведут к вычерчиванию полного эллипса. Пример построения эллипса, выведенного ярко-голубым цветом:

```
SetColor(LightCyan) ;  
Ellipse(100,100,0,360,30,50) ;
```



Обратите внимание, что фон внутри эллипса совпадает с фоном экрана. Чтобы создать закрашенный эллипс (в частности, закрашенный круг), используется специальная процедура

```
FillEllipse(x,y: integer; xR,yR: word);
```

где  $x$ ,  $y$  – центр эллипса в дисплейных координатах,  $xR$  и  $yR$  – горизонтальная и вертикальная оси. Заполнитель устанавливается процедурой `SetFillStyle`.

```
SetFillStyle(wideDotFill, Green);  
{установка стиля заполнения}  
SetColor(LightRed); {цвет для вычерчивания эллипса}  
FillEllipse(300,150,50,50);
```

В этом фрагменте эллипс вычерчивается ярко-красной кривой и заполняется редкими точками зеленого цвета.

## 16.6 Простые анимационные алгоритмы

Программы, которые строят, перемещают, изменяют форму различных изображений на экране в соответствии с заранее разработанным сценарием, называются *анимационными*.

Самый простой метод анимации заключается в следующем. Сначала выводится рисунок любым цветом. Через определенный период времени (используйте процедуру `Delay` из модуля `Crt`) тот же рисунок формируется цветом, совпадающим с фоном, что вызывает исчезновение изображения. Затем рисунок выводится в другом месте тем же цветом, что и первая картинка, и т.д.

Движение элемента изображения (спрайта) в этом алгоритме вызывает мелькание экрана и носит несколько прерывистый характер. Избавиться от мелькания и заставить элемент изображения двигаться более плавно можно, если организовать вывод на разные страницы видеопамяти. В этом случае применяется следующий алгоритм:

- вывести изображение на страницу 0 (она видима по умолчанию);
- сформировать новое изображение на невидимой странице 1;
- сделать видимой страницу 1;
- сформировать новое изображение на невидимой странице 0 и т.д.

## РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Бондарев В.М., Рублинецкий В.И., Качко Е.Г. Основы программирования. – Харьков: Фолио; Ростов н/Д: Феникс, 1997. – 368 с.
2. Вирт Н. Алгоритмы + структуры данных = программы. – М., Мир, 1985 – 406 с.
3. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. – М.: МЦНМО, 2001. – 960 с.
4. Немнюгин С. А. Turbo Pascal. – СПб.: Питер, 2001. – 496 с.
5. Немнюгин С.А. Turbo Pascal: Практикум. – СПб.: Питер, 2001. – 256 с.
6. Фаронов В.В. Турбо Паскаль 7.0: Практика программирования. – М.: Нолидж, 2000. – 416 с.
7. Шень А. Программирование: теоремы и задачи. – М.: МЦНМО, 1995.