МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Московский государственный университет экономики, статистики и информатики
Московский международный институт эконометрики, информатики, финансов и права

Грибанов В.П.
Калмыкова О.В.
Сорока Р.И.

Основы алгоритмизации и программирование

Москва 2002

УДК 004.42
ББК -018*32.973
К 174

Калмыкова О.В., Грибанов В.П., Сорока Р.И. Основы алгоритмизации и программирование. /Московский международный институт эконометрики, информатики, финансов и права. - М.: 2002. - 151 с.

Учебное пособие содержит краткий теоретический материал по основам программирования. Изложение ведется применительно к реализации Турбо Паскаля 7.0. Теоретический материал иллюстрируется большим количеством примеров с объяснениями использованных конструкций.

© Калмыкова О.В. 2002г.
© Грибанов В.П. 2002 г.
© Сорока Р.И. 2002г.
© Московский международный институт эконометрики, информатики, финансов и права, 2002г.

СОДЕРЖАНИЕ

Введение 6

1. Алгоритмизация вычислительных процессов. 8

 1.1 Основные определения и понятия. 8

 1.2 Средства изображения алгоритмов. 9

 1.3 Базовые канонические структуры алгоритмов. 14

Вопросы к главе 1. 16

2. Введение в Турбо Паскаль. 17

 2.1 Общая характеристика языка Паскаль. 17

 2.2 Основные понятия языка Турбо Паскаль. 18

  2.2.1 Алфавит языка. 18

  2.2.2 Элементарные конструкции. 19

  2.2.3 Типы данных. 21

 2.3 Операторы языка Паскаль. 24

  2.3.1 Оператор присваивания. 24

  2.3.2 Оператор перехода. Пустой оператор. Составной оператор. 25

  2.3.3 Условный оператор. 26

  2.3.4 Оператор выбора. 26

  2.3.5 Операторы цикла. 28

Вопросы к главе 2. 35

3. Структурированные типы данных. 37

 3.1 Свойства множеств. 37

 3.2 Операции над множествами. 38

 3.3 Описание записи (RECORD). 42

 3.4 Оператор присоединения. 44

 3.5 Запись с вариантами. 47

Вопросы к главе 3. 48

4. Использование подпрограмм в Турбо Паскале. 49

 4.1 Структура программы на языке Паскаль. 49

 4.2 Описание и вызов процедур. 50

 4.3 Описание функции. 51

 4.4 Формальные и фактические параметры. 52

 4.5 Область действия имен. 57

 4.6 Процедуры и функции без параметров. 59

 4.7 Рекурсивные процедуры и функции. 59

4.8 Предварительно-определенные процедуры. ________________ 61
4.9 Модули.____ 62
Вопросы к главе 4. ____________________________________ 65
5. Стандартные процедуры и функции.____________________ 66
5.1 Математические функции. ______________________________ 66
5.2 Функции округления и преобразования типов. _____________ 67
5.3 Функции порядкового типа._______________________________ 68
5.4 Процедуры порядкового типа. ___________________________ 69
5.5 Строковые функции. ____________________________________ 69
5.6 Строковые процедуры. __________________________________ 70
5.7 Прочие процедуры и функции.___________________________ 71
5.8 Процедуры ввода данных._________________________________ 72
5.9 Процедуры вывода данных. ________________________________ 74
5.9.1 Особенности вывода вещественных значений. __________ 75
Вопросы к главе 5. ____________________________________ 77
6. Работа с файлами. ____________________________________ 78
6.1 Общие сведения о файлах. _______________________________ 78
6.2 Процедуры и функции для работы с файлами.______________ 79
6.3 Особенности обработки типизированных файлов.__________ 80
6.4 Особенности обработки текстовых файлов. _______________ 83
6.5 Файлы без типа. ____ 85
6.6 Проектирование программ по структурам данных __________ 86
6.7 Работа с файлами при обработке экономической информации 93
6.7.1 Постановка задачи.________________________________ 93
6.7.2 Проектирование программы.________________________ 98
6.7.3 Кодирование программы.____________________________ 99
Вопросы к главе 6. ____________________________________ 104
7. Динамическая память.________________________________ 105
7.1 Указатель.____ 105
7.2 Стандартные процедуры размещения и освобождения динамической памяти. ____ 108
7.3 Стандартные функции обработки динамической памяти._____ 110
7.4 Примеры и задачи.____ 112
7.5 Работа с динамическими массивами.______________________ 113

7.6 Организация списков. 117
7.7 Задачи включения элемента в линейный однонаправленный список без головного элемента. 125
7.8 Задачи на удаление элементов из линейного однонаправленного списка без головного элемента. 129
7.9 Стеки, деки, очереди. 135
7.10 Использование рекурсии при работе со списками. 136
7.11 Бинарные деревья. 137
7.12 Действия с бинарными деревьями. 139
7.13 Решение задач работы с бинарным деревом. 141
Вопросы к главе 7. 145
8. Основные принципы структурного программирования. 146
8.1 Понятие жизненного цикла программного продукта 146
8.2 Основные принципы структурной методологии. 147
8.3 Нисходящее проектирование. 148
8.4 Структурное кодирование. 148
8.5 Модульное программирование. 148
Вопросы к главе 8. 150
9. Список литературы 151

Введение

Учебное пособие разработано в соответствии с программой курса «Основы алгоритмизации и программирование» и предназначено для студентов специальностей «Прикладная информатика в экономике», «Прикладная информатика в менеджменте», «Прикладная информатика в юриспруденции».

Учебное пособие состоит из 8 глав.

В первой главе излагаются общие вопросы курса. Приводятся основные определения и понятия, описываются изобразительные средства представления алгоритмов, базовые канонические структуры алгоритмов.

Вторая глава посвящена введению в программирование на языке Турбо Паскаль. Описываются основные конструкции языка, начиная с алфавита языка, элементарных конструкций и типов данных и заканчивая операторами языка.

В третьей главе описываются структурированные типы данных: множества и записи, а также приемы работы с ними.

В четвертой главе излагаются вопросы использования подпрограмм в языке Турбо Паскаль. Приводится структура программы, описание и вызов процедур и функций, виды и способы передачи параметров в процедуры и функции, области действия идентификаторов в программах сложной структуры, использование рекурсивных процедур и функций. Завершается глава описанием структуры и отдельных частей модулей.

В пятой главе приводятся стандартные процедуры и функции. Описываются процедуры и функции модуля System, в котором располагается стандартная библиотека Турбо Паскаля, подключаемая по умолчанию. Рассматриваются особенности использования процедур ввода и вывода данных различных типов. Кроме того, описываются процедуры и функции модуля Crt, обеспечивающие удобную работу с экраном и клавиатурой.

Шестая глава посвящена работе с файлами. Сначала излагаются общие вопросы работы с файлами, затем особенности работы с различными типами файлов: типизированными файлами, текстовыми файлами и файлами без типа. Кроме того, описываются основные этапы проектирования программ по структурам данных, и рассматривается применение этого подхода к решению задач обработки файлов.

В седьмой главе излагаются вопросы использования динамической памяти в программах. Приводятся стандартные процедуры и функции работы с динамической памятью и их использование для обработки динамических массивов. Далее рассматриваются динамические структуры данных: списки, стеки, очереди и деревья, а также приемы работы с ними. Описываются типовые операции, выполняемые над динамическими структурами данных, и обсуждаются возможности их реализации на языке Турбо Паскаль.

Восьмая глава посвящена основным принципам структурного программирования. Рассматриваются основные этапы жизненного цикла программного обеспечения, некоторые подходы к разработке программных продуктов: нисходящее проектирование, структурное кодирование и модульное программирование.

1. Алгоритмизация вычислительных процессов.

1.1 Основные определения и понятия.

Алгоритмизация – это процесс построения алгоритма решения задачи, результатом которого является выделение этапов процесса обработки данных, формальная запись содержания этих этапов и определение порядка их выполнения.

Алгоритм – это точное предписание, определяющее вычислительный процесс, ведущий от варьируемых начальных данных к искомому результату.

Свойства алгоритма:

1) детерминированность – точность указаний, исключающая их произвольное толкование;

2) дискретность – возможность расчленения вычислительного процесса на отдельные элементарные операции, возможность выполнения которых не вызывает сомнений;

3) результативность – прекращение процесса через определенное число шагов с выдачей искомых результатов или сообщения о невозможности продолжения вычислительного процесса;

4) массовость – пригодность алгоритма для решения всех задач заданного класса.

Алгоритмический язык – набор символов и правил образования и истолкования конструкций из этих символов для записи алгоритмов.

Язык программирования – предназначен для реализации программ на ЭВМ.

Программа – это алгоритм, записанный в форме, воспринимаемой машиной. Программа содержит наряду с описанием данных команды, в какой последовательности, над какими *данными* и какие операции должна выполнять машина, а также в какой форме следует получить результат. Это обеспечивают различные *операторы*.

Данные – это факты и идеи, представленные в формализованном виде, позволяющем передавать или обрабатывать эти факты и идеи с помощью некоторого процесса.

Оператор – совокупность символов, указывающих операцию и значения, либо местонахождение ее элементов.

A:= B+C; **{A, B, C –** переменные**;}**

К:= 2; IF T< 0 THEN . . .

Переменная – это объект, который в ходе выполнения программы может менять свое значение.

Свойства переменной:

1) переменная называется неопределенной до тех пор, пока она не получит значение:

а) вводом извне;

б) занесением константы;

в) занесением значения другой, ранее определенной переменной;

2) в каждый момент времени переменная может либо иметь определенное значение, либо быть неопределенной;

3) последующее значение уничтожает (стирает) предыдущее значение. Выбор (чтение) переменной и ее использование не изменяют значение переменной.

Предметом курса являются методы и средства составления алгоритмов и программ с целью решения задач на ЭВМ. Для разработки программ используются *системы программирования*.

Система программирования – средство автоматизации программирования, включающее язык программирования, транслятор этого языка, документацию, а также средства подготовки и выполнения программ.

Транслятор – это программа, которая переводит с одного языка на другой.

Интерпретатор – это программа, которая сразу выполняет переводимые команды.

Компилятор – это программа, которая переводит конструкции алгоритмического языка в машинные коды.

1.2 Средства изображения алгоритмов.

Основными изобразительными средствами алгоритмов являются следующие способы их записи:

- словесный;
- формульно-словесный;
- блок-схемный;
- псевдокод;
- структурные диаграммы;
- языки программирования.

Словесный – содержание этапов вычислений задается на естественном языке в произвольной форме с требуемой детализацией.

Рассмотрим пример словесной записи алгоритма. Пусть задан массив чисел. Требуется проверить, все ли числа принадлежат заданному интервалу. Интервал задается границами A и B.

п.1 Берем первое число. На **п.2**.

п.2 Сравниваем: выбранное число принадлежит интервалу; если да, то на **п.3**, если нет – на **п.6**.

п.3 Все элементы массива просмотрены? Если да, то на **п.5**, если нет – то на **п.4**.

п.4 Выбираем следующий элемент. На **п.2**.

п.5 Печать сообщения: все элементы принадлежат интервалу. На **п.7**.

п.6 Печать сообщения: не все элементы принадлежат интервалу. На **п.7**.

п.7 Конец.

При этом способе отсутствует наглядность вычислительного процесса, т.к. нет достаточной формализации.

Формульно-словесный – задание инструкций с использованием математических символов и выражений в сочетании со словесными пояснениями.

Например, требуется написать алгоритм вычисления площади треугольника по трем сторонам.

п.1 – вычислить полупериметр треугольника

$p=(a+b+c)/2$. К **п.2**.

п.2 – вычислить

$$S=\sqrt{p(p-a)(p-b)(p-c)}$$

К **п.3.**

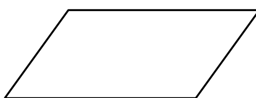
п.3 – вывести S, как искомый результат и прекратить вычисления.

При использовании этого способа может быть достигнута любая степень детализации, более наглядно, но не строго формально.

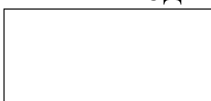
Блок-схемный – это графическое изображение логической структуры алгоритма, в котором каждый этап процесса переработки данных представляется в виде геометрических фигур (блоков), имеющих определенную конфигурацию в зависимости от характера выполняемых операций.

Блок-схемы могут быть традиционные и структурированные.

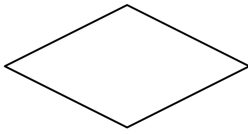
Основные символы блок-схем:



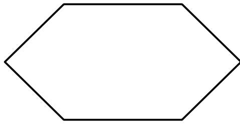
- ввод-вывод;



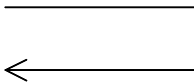
- процесс (выполнение операций или группы операций);



- решение (выбор направления);



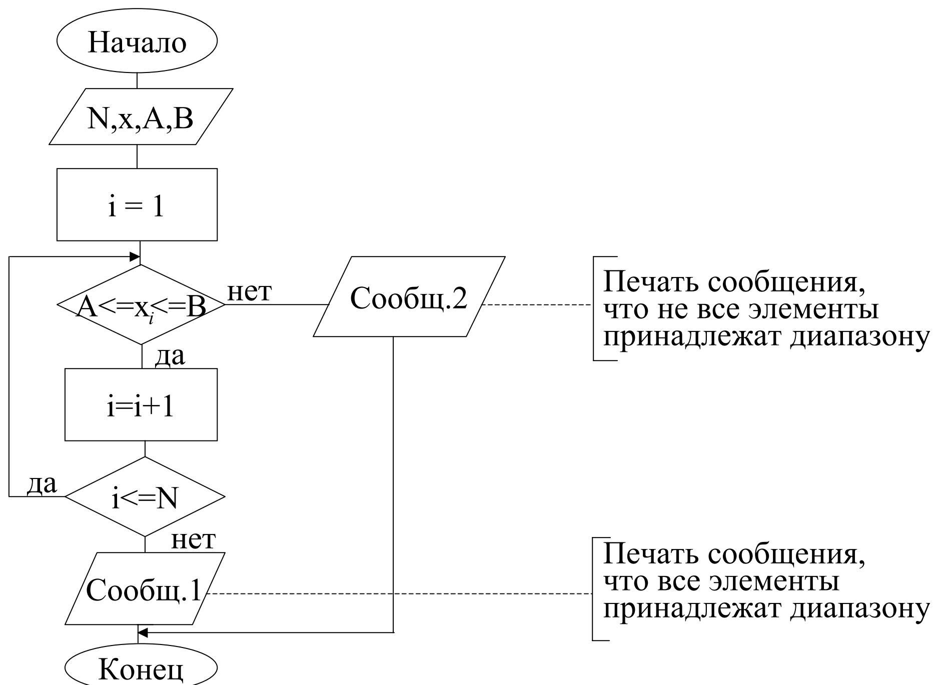
- модификация (организация цикла);



- линии потока данных

- пуск-останов (начало, конец программы).

Рассмотрим пример блок-схемы той же задачи, для которой приведен словесный алгоритм.



Псевдокод - позволяет формально изображать логику программы, не заботясь при этом о синтаксических особенностях конкретного языка программирования. Обычно представляет собой смесь операторов языка программирования и естественного языка. Является средством пред-

ставления логики программы, которое можно применять вместо блок-схемы.

Запись алгоритма в виде псевдокода:

Выбираем первый элемент ($i=1$)

IF $A > x_i$ или $x_i > B$ **THEN**

печать сообщения и переход на конец

ELSE

переход к следующему элементу($i = i + 1$)

IF массив не кончился ($i <= n$) **THEN**

переход на проверку интервала

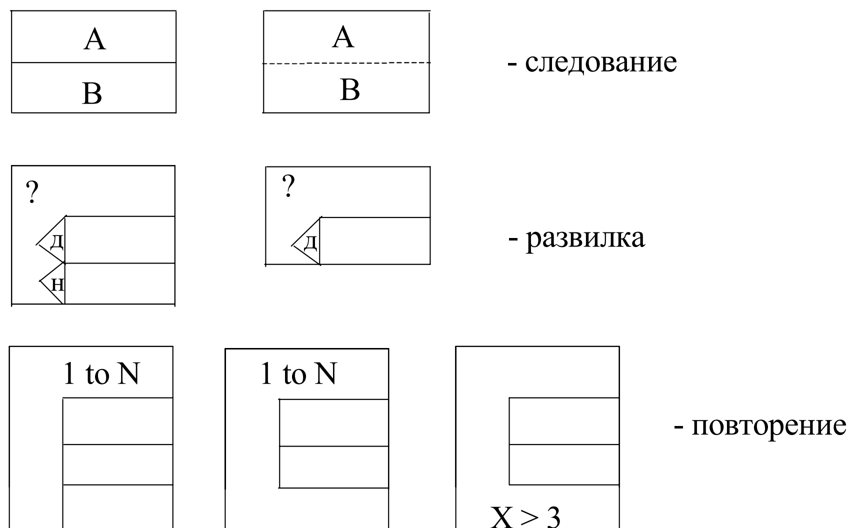
ELSE

печать сообщения, что все элементы входят в интервал

Конец

Структурные диаграммы - могут использоваться в качестве структурных блок-схем, для показа межмодульных связей, для отображения структур данных, программ и систем обработки данных. Существуют различные структурные диаграммы: диаграммы Насси-Шнейдермана, диаграммы Варнье, Джексона, МЭСИД и др.

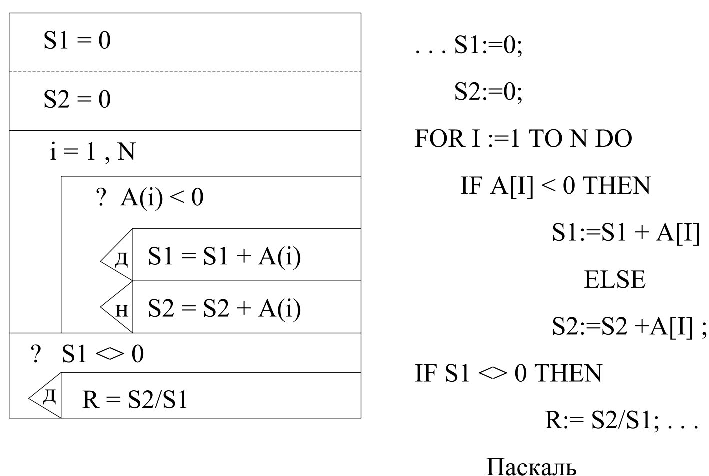
Основные элементы МЭСИД



- выбор

Рассмотрим пример использования диаграмм МЭСИД.

Задан одномерный массив из положительных и отрицательных чисел. Требуется определить частное от деления суммы положительных элементов на сумму отрицательных элементов этого массива. Справа от диаграммы приводятся соответствующие операторы языка Паскаль.



Паскаль

Языки программирования - изобразительные средства для непосредственной реализации программы на ЭВМ. *Программа* – алгоритм, записанный в форме, воспринимаемой ЭВМ.

Каждая машина имеет свой собственный язык (машинный язык) и может выполнять программы только на этом языке. Это последовательность машинных команд. Писать программы на машинном языке очень сложно и утомительно. Для повышения производительности труда программистов применяются искусственные языки программирования. При этом требуется перевод программы, написанной на таком языке, на машинный язык. Этот перевод выполняет транслятор. Наиболее часто встречающимся транслятором интерпретирующего типа является транслятор с языка Бейсик, где команды читаются, преобразуются и выполняются сразу. Итогом работы такого транслятора являются требуемые результаты.

Транслятор с Паскаля – компилирующего типа.

Текст программы на исходном языке сначала переводится в текст на машинном языке и получается так называемый объектный модуль. Затем объектный модуль должен быть обработан программой Редакто-

ром межпрограммных связей и только после этого программа будет готова к выполнению.

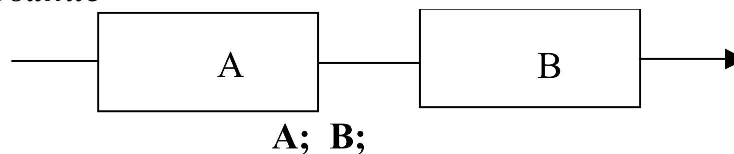
1.3 Базовые канонические структуры алгоритмов.

Доказано, что любую программу можно написать, используя комбинации трех управляющих структур:

- следования или последовательности операторов;
- развилки или условного оператора;
- повторения или оператора цикла.

Программа, составленная из канонических структур, будет называться регулярной программой, т.е. иметь 1 вход и 1 выход, каждый оператор в программе может быть достигнут при входе через ее начало (нет недостижимых операторов и бесконечных циклов). Управление в такой программе передается сверху-вниз. Снабженные комментариями, такие программы хорошо читабельны.

1) следование

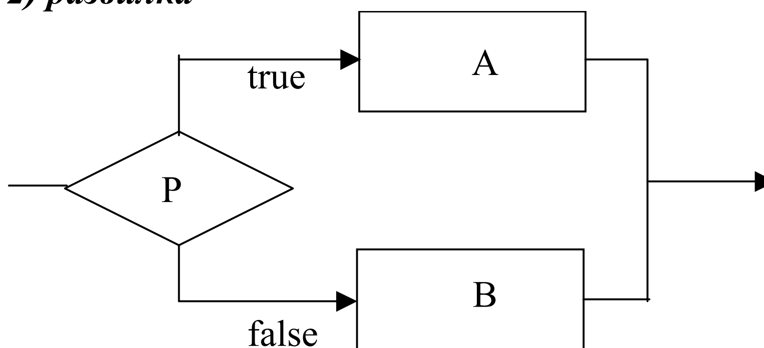


A; B;

Действия **A** и **B** могут быть:

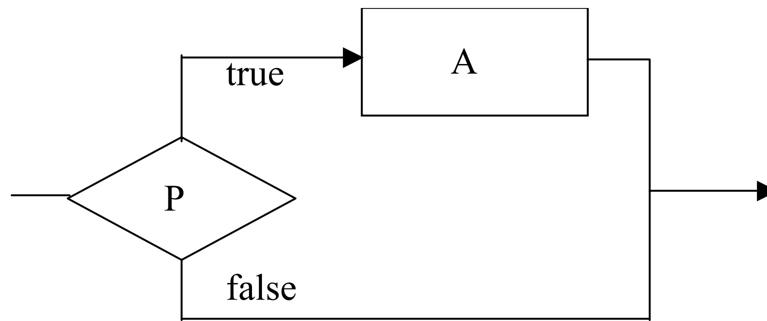
- отдельным оператором;
- вызовом с возвратом некоторой процедуры;
- другой управляющей структурой.

2) развилка



IF P then A else B;

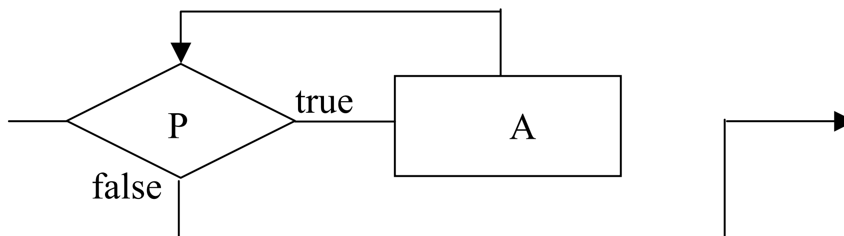
Проверка **P** представляется *предикатом*, т.е. функцией, задающей логическое выражение или условие, значением которого может быть *истина* или *ложь*. Эта структура может быть неполной, когда отсутствует действие, выполняемое при ложном значении логического выражения. Тогда структура будет следующая:



IF P then A ;

3) повторение

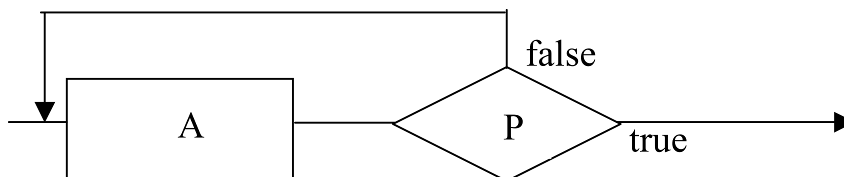
цикл – пока



While P do A ;

Действие A будет повторяться до тех пор, пока значение предиката будет оставаться истинным. Поэтому в действии A должно изменяться значение переменных, от которых зависит P. В противном случае произойдет зацикливание. Вычисление предиката производится **до** начала выполнения действия A и может случиться так, что действие A не будет выполняться ни разу.

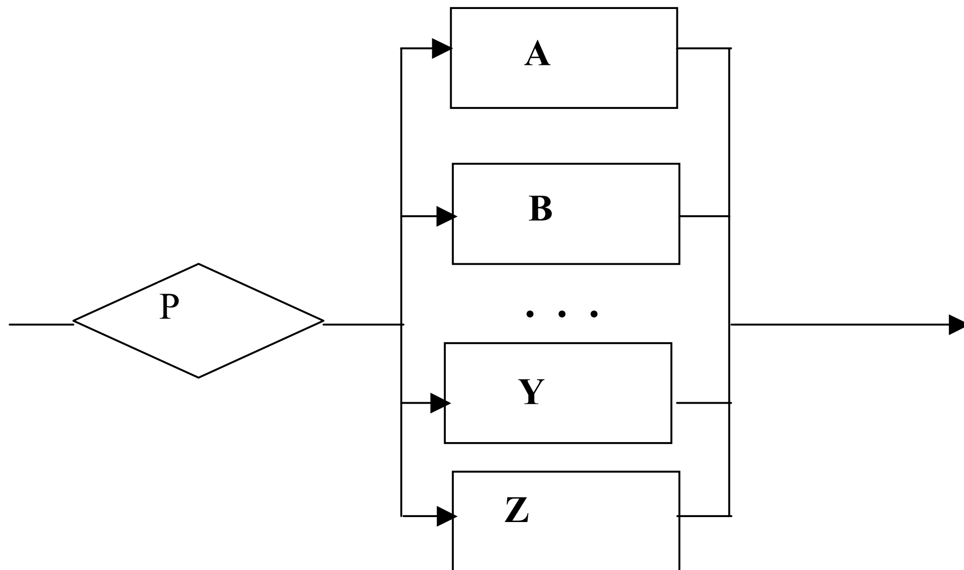
цикл – до



Repeat A until P;

Повторение типа **Repeat until** всегда выполняется хотя бы 1 раз. Действие A перестает выполняться, как только предикат становится истинным.

4) выбор – переключатель ***case*** (обобщение развилки), структура, облегчающая программирование без ущерба для ясности программы. Структура выбор полезна в том случае, когда требуется выбрать одну из нескольких альтернатив.



В зависимости от значения P выполняется одно из действий A, B, …Z. После чего происходит переход к выполнению следующей управляющей структуры.

Вопросы к главе 1.

1. Что такое данные?
2. Что такое программа?
3. Что такое алгоритм?
4. Что такое алгоритмический процесс?
5. Перечислить свойства алгоритмов.
6. Чем отличается компилятор от интерпретатора?
7. Что такое подпрограмма?
8. Дать краткую характеристику различных типов вычислительных процессов.
9. Особенности словесного способа изображения алгоритмов.
10. Особенности формульно-словесного способа изображения алгоритмов.
11. Особенности изображения алгоритмов с помощью операторных схем.
12. Особенности изображения алгоритмов с помощью структурных диаграмм.
13. Особенности блок-схемного способа изображения алгоритмов.
14. Основные символы, использующиеся при составлении блок-схем.
15. Основные элементы диаграмм МЭСИД.
16. Понятие регулярной программы.
17. Особенности использования базовых конструкций «следование» и «повторение».
18. Особенности использования базовых конструкций «развилка» и «выбор».

2. Введение в Турбо Паскаль.

2.1 Общая характеристика языка Паскаль.

Язык Паскаль был разработан Никласом Виртом первоначально для целей обучения программированию. В настоящее время он получил широкое распространение по ряду объективных причин.

Во-первых, по своей идеологии Паскаль наиболее близок к современной методике и технологии программирования. В частности, он достаточно полно отражает идеи структурного программирования, что довольно хорошо видно даже из основных управляющих структур языка.

Во-вторых, Паскаль хорошо приспособлен для применения технологии разработки программ сверху-вниз (пошаговой детализации).

В-третьих, Паскаль содержит большое разнообразие различных структур данных, что обеспечивает простоту алгоритмов, а следовательно снижение трудоемкости при разработке программ.

Основные отличия алгоритмических языков от машинных языков:

- алгоритмический язык обладает гораздо большими выразительными возможностями, т.е. его алфавит значительно шире алфавита машинного языка, что существенно повышает наглядность текста программы;

- набор операций, допустимых для использования, не зависит от набора машинных операций, а выбирается из соображений удобства формулирования алгоритмов решения задач определенного класса;

-формат предложений достаточно гибок и удобен для использования, что позволяет с помощью одного предложения задать достаточно содержательный этап обработки данных;

- требуемые операции задаются в удобном для человека виде, например, с помощью общепринятых математических обозначений;

- для задания операндов операций, используемым в алгоритме данным присваиваются уникальные имена, выбираемые программистом, и ссылка на операнды производится, в основном, по именам;

- в языке может быть предусмотрен значительно более широкий набор типов данных по сравнению с набором машинных типов данных.

Из вышеперечисленного следует, что алгоритмический язык в значительной мере является машинно-независимым.

Для описания синтаксиса алгоритмического языка используется специальный метаязык, позволяющий в компактной форме отразить все особенности конкретных конструкций алгоритмического языка. Мы воспользуемся для этих целей металингвистическими формулами Бэкуса-Наура (язык БНФ).

При описании синтаксиса языка используются некоторые его понятия: определив простейшие из них, с их помощью можно уже достаточно просто определить более сложные понятия и т.д., пока не будет определено наиболее сложное понятие - программа. С точки зрения син-

таксиса каждое определяемое понятие (но не основной символ) есть метапеременная языка БНФ, значением которой может быть любая конструкция (т.е. последовательность основных символов) из некоторого фиксированного для этого понятия набора конструкций.

Для каждого понятия языка должна существовать единственная метаформула, в левой части которой указывается определяемое понятие (метапеременная языка БНФ), а правая часть формулы тем или иным способом задает все множество значений этой метапеременной (все допустимые конструкции, которые объединяются в это понятие). Все метапеременные заключаются в специальные угловые скобки < и >, которые не принадлежат алфавиту определяемого языка, т.е. являются метасимволами, например, <выражение>, <число> и т.д. Основные же символы языка указываются непосредственно. Левая и правая части метаформулы разделяются специальным знаком **: : =**, смысл которого можно интерпретировать как «по определению есть». Обычно в качестве значений метапеременной может приниматься любая из нескольких допустимых конструкций. Все допустимые конструкции указываются в правой части формулы и разделяются метасимволом «|», смысл которого можно передать словом «или» («либо»). Кроме перечисления всех возможных значений метапеременной в правой части метаформулы может быть указано правило построения значений.

2.2 Основные понятия языка Турбо Паскаль.

2.2.1 Алфавит языка.

Язык Турбо Паскаль допускает использование прописных и строчных букв латинского алфавита, знака подчеркивания, арабских цифр и ограничителей.

<алфавит> :: = <буквы> | <цифры> | <ограничители>
<буквы> :: = **A | B | …| Z | a | b | …| z |** <знак подчеркивания>
<цифры> :: = **0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**
<знак подчеркивания> :: = **_**
<ограничители>::=<знаки операций>|<скобки> |
<зарезервированные слова>| <разделители>
<знаки операций>::= <арифметические>|<отношения>|
<логические>|<над строками>|<над множествами>
<арифметические знаки операций>::= **+ | - | * | / | div | mod**
<знаки операций отношения>::= **= | <> | > | < | >= | <=**
<логические знаки операций>::= **not | and | or | xor**
<знаки операций над строками>::= **+** |<знаки операций
отношения>
<знаки операций над множествами>::= *** | + | - | = | <> | <= |
>= | in**

<скобки>**::=** <для индексов>**|**<для выражений и функций>**|**
<операторные>**|** <для строк>
<скобки для индексов>**::= [ ] | (. .)**
<скобки для выражений и функций>**::= ( )**
<скобки операторные>**::= begin end**
<скобки для строк>**::= ' '**
<зарезервированные слова>**::= and | asm | array | begin |**
case | const| constructor | destructor | div |
do | downto | else | end | exports | file | for |
function | goto | if | implementation | in |
inherited | inline | interface | label | library |
mod | nil | not | object | of | or | packed |
procedure | program | record | repeat | set |
shl | shr | string | then | to | type | unit | until |
uses | var | while | with | xor
<разделители>**::= := | . | , | : | ; | ^ | $ | #**

Пробел также является разделителем. Может использоваться для повышения наглядности программы. Его не должно быть внутри идентификаторов, чисел и зарезервированных слов.

Комментарий - это текст, заключенный между скобками { } или **(* *)**. Может содержать буквы (в том числе и русские), цифры, специальные знаки. Используется для обозначения участков программ или пояснения команд. Может располагаться в любом месте программы.

2.2.2 Элементарные конструкции.

Из основных символов непосредственно образуются элементарные конструкции языка, которые являются минимальными неделимыми синтаксическими единицами. К ним относятся: идентификаторы, константы, переменные, указатели стандартных функций.

Различные объекты программы должны иметь имена. В качестве имен используются идентификаторы. ***Идентификатор*** - последовательность букв и цифр, а также знака подчеркивания, начинающаяся с буквы. Длина идентификатора может быть до 63 символов. Желательно выбирать значащие идентификаторы, например, MAX, MIN, SUMMA и т.д. Зарезервированные слова нельзя использовать в качестве идентификаторов.

Константы – это данные, значения которых не изменяются в процессе работы программы. Константы могут быть поименованными (с типом и без типа) и явно заданными.

Арифметические константы предназначены для представления целых и вещественных числовых данных.

Для *вещественных констант* используется форма записи с фиксированной точкой и экспоненциальная форма. В форме с фиксированной точкой целая и дробная части разделяются точкой, при экспоненциальной форме число представляется в виде мантиссы и порядка, между которыми стоит буква E.

<мантисса>E{±}<порядок>

Тип константы	Диапазон	Объем памяти	Примечания
Shortint	-128..127	1 байт	Со знаком
Byte	0..255	1 байт	Без знака
Integer	-32768..32767	2 байта	Со знаком
Word	0..65535	2 байта	Без знака
Longint	-2147483648..2147483647	4 байта	Со знаком
Single	1.5E-45..3.4E38	4 байта	7-8 значащих цифр, со знаком
Real	2.9E-39..1.7E38	6 байт	11-12 значащих цифр, со знаком
Double	5.0E-324..1.7E308	8 байт	15-16 значащих цифр, со знаком
Extended	3.4E-4932..1.1E4932	10 байт	19-20 значащих цифр, со знаком
Comp	9.2E18..9.2E18	8 байт	19-20 значащих цифр, со знаком

В Турбо Паскале определены некоторые именованные константы, использовать которые можно без объявления.

MAXINT=32767,   MAXLONGINT=2147483647.

Логические константы могут принимать одно из двух значений: True(истина) или False(ложь). В языке предусмотрены следующие логические типы констант: Boolean(1 байт), Bytebool(1 байт), Wordbool(2 байта), Longbool(4 байта).

Константы символьного типа Char представляют собой 1 символ кодового набора ASCII (американский стандартный код обмена информацией). Занимает в памяти 1 байт. Символьная константа заключается в апострофы ‘’. Существует упорядоченность символов в соответствии с их кодами. Для цифр и букв латинского алфавита коды символов удовлетворяют соотношению: ‘0’<…<’9’<‘A’<…<’Z’<’a’<…<’z’.

Константы строкового типа String представляют собой последовательность символов, заключенную в апострофы. Длиной строки называется количество символов в ней. Если среди символов константы строкового типа имеется хотя бы один апостроф, он изображается двумя апострофами. Длина строки может быть от 0 до 255 символов.

Переменная – это наименование, данное некоторому значению. Обозначается с помощью идентификатора переменной. Переменные

стандартных типов принимают значения из диапазонов значений, указанных для констант соответствующего вида.

2.2.3 Типы данных.

Каждая переменная и константа в программе на языке Паскаль имеет свой тип данных. Тип определяет набор операций, которые могут быть к ней применимы, а также тип результата выполнения этих операций. Имеются типы стандартные и типы, описанные пользователем.

Все переменные, используемые в программе должны быть описаны в разделе описания переменных **VAR**.

```
VAR <идентификатор> [,<идентификатор>,…]: <тип>;
       [<идентификатор> [,<идентификатор>,…]: <тип>;…]
```

Например,

```
VAR     A : INTEGER;
        B, C: REAL;
```

Здесь описана переменная A целого типа и две переменные B и C вещественного типа.

Тип, в свою очередь, может быть предварительно определен в разделе описания типов **TYPE**.

```
TYPE <идентификатор типа> = <тип> ;
```

Например,

```
TYPE    I = INTEGER;
        R = REAL;
```

После такого описания типов описания переменных A, B и C могут быть следующими:

```
VAR A : I;
    B, C: R;
```

Порядковый стандартный тип обозначает конечное линейное множество значений. К нему обычно относят целые типы, байтовые, символьные и логические.

Перечисляемый тип – определяет упорядоченное множество значений путем перечисления идентификаторов, выражающих эти значения как постоянные. Для каждого такого элемента выделяется один байт памяти. Используется в качестве индексов массивов и имеет порядковую нумерацию, начиная с нуля.

```
TYPE <идент-тор типа> = (<идентификатор>
                          [,<идентификатор>,…]);
```

Интервальный тип определяет некоторое подмножество значений, которые может принимать данная переменная. Задаются наименьшее и

наибольшее значения порядкового типа. Для каждого элемента выделяется один байт памяти.

```
TYPE <идентификатор типа> = <константа>. .<константа>;
```

В качестве констант могут использоваться значения любых простых типов кроме вещественных типов.

Например,

```
TYPE   GR = (DS101, DS102, DS201, DS202, DS301, DS302);
        SPEC = DS101. . DS302;
        DIGIT = 0. .9;
VAR    A : DIGIT;
       B : SPEC;
       D :  100. .200;
```

Строчный тип используется для описания строк длиной от 0 до 255 символов. Максимальная длина строки указывается в квадратных скобках. Если она не указана, максимальная длина полагается 255. Строковые переменные, как и строковые константы, занимают количество байтов, равное максимальной длине строки плюс 1 байт (нулевой), предназначенный для хранения текущей длины этой строковой переменной. Важной особенностью Турбо Паскаля является то, что к каждому символу строки возможен доступ по его номеру.

```
TYPE  <идент-тор типа> = String [<максимальная длина>];
```

Например,

```
TYPE   TSTRING = STRING[100];
       TS = STRING;
 VAR   S,S1 : TSTRING;
       S2 : STRING[20];
       SS : TS;
```

Массив – это упорядоченная совокупность однотипных переменных, обладающих одинаковыми свойствами. Отношение порядка между элементами массива задается с помощью индексирования. Каждому элементу массива ставится в соответствие один или несколько индексов. Если каждому элементу ставится в соответствие один индекс, то это - одномерный массив (вектор). При наличии двух индексов – двумерный массив (матрица), причем, обычно первый индекс обозначает номер строки, а второй – номер столбца, где находится соответствующий элемент.

```
TYPE <идентификатор типа> = ARRAY [<список типов индексов>]
                                                OF<тип>;
```

<тип индексов> :: = <простой тип>
<простой тип>:: = <идентификатор типа>|<идентификатор>
[,<идентификатор>]| <константа>. .<константа>

В качестве типа индекса может быть любой простой тип, кроме вещественных. Чаще всего используется интервальный тип индекса от целых типов.

Например,

```
TYPE T1 = ARRAY [-10 .. 20,1..30] OF BYTE;
        T2 = ARRAY [0..50] OF BOOLEAN;
        T3 = ARRAY [BYTE] OF INTEGER;
VAR A, B: T1;
          C: T2;
          Z: ARRAY[1..100] OF REAL;
     MAS: T3;
```

Здесь в разделе описания типов приводятся три различных типа массивов. T1 – это тип двумерного массива, в котором номера строк могут принимать значения от –10 до 20, а номера столбцов – от 1 до 30. Элементами массива типа T1 должны быть целые числа без знака от 0 до 255. Тип T2 определяет одномерный массив с элементами логического типа, номера элементов могут быть от 0 до 50. Тип T3 определяет одномерный массив с элементами целого типа со знаком, при этом диапазон изменения индексов массива от 0 до 255. В разделе описания переменных определяются переменные A и B типа T1, C – типа T2 и MAS – типа T3. Помимо предварительного описания типа возможно определять массивы непосредственно в разделе описания переменных, как это сделано для переменной Z.

Обращение к элементам массивов осуществляется с помощью переменных с индексами. Число индексов в переменной с индексами равно числу измерений массива. Индексы могут задаваться целыми числами, простыми переменными, арифметическими выражениями. Если массивы имеют одинаковое описание, их можно копировать **B:= A**.

Например,

```
             …
        S := S + Z [I] ;
         P := P * A [I] [J] ;
        C[6] := TRUE ;
        P := P * A[I,J] ;
        R := B [I+5,J] ;
MAS [I] := MAS [I-1] * MAS [I] ;
             …
```

Рассмотрим различные способы описания массивов.

Пусть требуется описать матрицу А, содержащую 10 строк и 50 столбцов, с элементами целого типа.

```
1)      CONST  N = 10;
               M = 50;
        TYPE TMATR = ARRAY [1..N, 1..M] OF INTEGER;
        VAR A : TMATR;

2)      TYPE TSTR = ARRAY [1..50] OF INTEGER;
             TMATR = ARRAY [1..10] OF TSTR;
        VAR A : TMATR;

3)      VAR A : ARRAY[1..10,1..50] OF INTEGER;

4)      VAR A : ARRAY[1..10] OF ARRAY[1..50] OF INTEGER;
```

2.3 Операторы языка Паскаль.

2.3.1 Оператор присваивания.

Наиболее простым и часто используемым оператором языка является ***оператор присваивания:***

<переменная> **: =** <выражение>**;**

Выражение – это формула для вычисления значения. Она образуется из операндов, соединенных знаками операций и круглыми скобками. В качестве операндов могут выступать переменные, константы, указатели функций.

Тип переменной в левой части оператора присваивания обычно должен совпадать с типом значения выражения в правой части. Возможны случаи несовпадения типов, например, когда слева переменная вещественного типа, а справа выражение целого типа. Выражения являются составной частью операторов.

В Паскале приоритеты выполнения операций следующие (в порядке убывания):

- одноместный минус;
- операция **NOT**;
- операции типа умножения ;
- операции типа сложения;
- операции сравнения (отношения).

Одноместный минус применим к операндам арифметического типа. Операция **NOT** – к операндам логических и целых типов. Если в од-

ном выражении несколько операций одного приоритета, то они выполняются, начиная слева. Приоритеты можно изменить, поставив скобки. В логических выражениях необходимы скобки во избежание конфликта типа по приоритету.

Например, если в выражении **… (X > 5) AND (Y > 10) …** не поставить скобки, то будет синтаксическая ошибка, так как приоритет операции **AND** выше приоритета операций сравнения >.

<операции типа умножения> **:: = * | / | div | mod | and**

<операции типа сложения> **:: = + | - | or | xor**

<операции сравнения> **:: = = | <> | < | > | <= | >= | in**

Операции сравнения применимы для всех стандартных простых типов. Причем в одном выражении возможно использование операндов различных типов. Результат сравнения всегда имеет логический тип.

Например, **(5 + 6) < (5 - 6) = TRUE** в результате даст **FALSE**, а **NOT(8.5 < 4)** будет равно **TRUE**.

Сравнение строк символов выполняется слева направо посимвольно. Более короткие строки дополняются пробелами справа.

2.3.2 Оператор перехода. Пустой оператор. Составной оператор.

Оператор безусловного перехода **GOTO** служит для прерывания естественного хода выполнения программы. Следующим выполняется оператор, помеченный меткой, которая использована в данном операторе перехода. Один оператор может помечаться несколькими метками.

GOTO <метка> **;**

<метка> - это целое без знака или идентификатор, обязательно описанный в разделе описания меток (**LABEL**).

Для того, чтобы пометить оператор, перед ним ставится метка, после которой записывается двоеточие.

< метка**> :[<**метка**>: …] <**оператор**>;**

Оператор **GOTO** не рекомендуется использовать при программировании, так как это существенно усложняет отладку и тестирование программы, тем более что остальных управляющих операторов языка вполне достаточно для реализации любого алгоритма.

Пустой оператор не обозначается и не вызывает никаких действий в программе, представляет собой дополнительную точку с запятой.

Если необходимо, чтобы группа операторов рассматривалась транслятором, как один оператор, эту группу операторов заключают в операторные скобки **BEGIN** и **END**. Такой оператор называется ***составным оператором***. Составной оператор может быть использован в любом месте программы, где разрешен простой оператор, но требуется выполнение группы операторов.

2.3.3 Условный оператор.

Условный оператор используется для программирования развилки, если условие сформулировано как логическое выражение.

```
IF <логическое выражение> THEN<оператор 1>
                               [ ELSE <оператор 2>] ;
<следующий оператор >;
```

Оператор выполняется таким образом: если результат вычисления логического выражения **TRUE**, то выполняется <оператор 1>, затем <следующий оператор >; если – **FALSE**, то выполняется <оператор 2>, затем <следующий оператор>. Операторы 1 и 2 могут быть простым или составным оператором. Если часть оператора, начинающаяся **ELSE**, отсутствует, то при логическом выражении равным **FALSE**, будет выполняться <следующий оператор>. При вложенности условных операторов **ELSE** всегда относится к ближайшему предшествующему **IF**. Следует избегать большой глубины вложенности условных операторов, так как при этом теряется наглядность и возможно появление ошибок.

Например,

```
… IF A > 0 THEN P := P + 1
            ELSE
               IF A < 0 THEN O := O + 1
                        ELSE  N := N + 1 ; …
… IF A > 0 THEN
            BEGIN
                S := S+ A ; K := K + 1
            END ;…
```

2.3.4 Оператор выбора.

***Оператор выбора* CASE** может быть использован вместо условного оператора, если требуется сделать выбор более, чем из двух возможностей.

```
CASE <селекторное выражение> OF
   <метка>[ ,<метка>                 ]      : <оператор 1> ;
          [ ,<метка> . . <метка>     ]
      <метка>[ ,<метка>                 ]     : <оператор 2> ;
             [ ,<метка> . . <метка>     ]
             . . . . .
   <метка> [ ,<метка>                ]      : <оператор n>;
           [ ,<метка> . . <метка>    ]
  [ ELSE  <оператор>]
END;
```

Селекторное выражение (селектор, переключатель) и метки-константы (метки варианта, метки выбора) должны иметь один и тот же простой тип (кроме вещественного). Метки-константы в отличие от меток программы не требуется описывать в разделе описания меток. Но на них нельзя ссылаться в операторе **GOTO**. Метки варианта могут быть перечисляемого и интервального типа.

<оператор 1>,<оператор 2>,<оператор n> - простой или составной оператор.

Оператор выбора выполняется следующим образом. Сначала вычисляется селекторное выражение; затем выполняется оператор, метка варианта которого равна текущему значению селектора; после этого происходит выход из оператора **CASE** на следующий оператор. Если значение селектора не совпадает ни с одной из меток варианта, будет выполнен оператор после **ELSE**. Если ветвь **ELSE** отсутствует, то управление передается следующему за **CASE** оператору.

Например,

$$Z = \begin{cases} \cos x, & \text{при } k=3 \\ \sin x, & \text{при } k=2 \\ e^{x}, & \text{при } k=1 \\ \ln x, & \text{при } k=0 \\ 0 & \text{в остальных случаях} \end{cases}$$

```
… CASE K OF
      0:  Z := LN(X) ;
      1:  Z := EXP(X) ;
      2:  Z := SIN(X) ;
      3:  Z := COS(X)
                      ELSE
          Z := 0
      END ; …
```

В этом примере результат вычисляется по одной из стандартных функций в зависимости от параметра К, который получает свое значение перед выполнением этого оператора.

В следующем примере переменная **OTVET** получает значение YES или NO в зависимости от введенного значения символьной переменной **V.** Здесь метки варианта задаются перечислением.

```
… VAR V : CHAR;
          OTVET : STRING;
              …
          CASE  V  OF
               'D', 'd', 'Д', 'д' : OTVET := 'YES';
               'N', 'n', 'H', 'н' : OTVET := 'NO'
           ELSE
                OTVET := ' '
           END; …
```

В следующем примере метки выбора заданы интервалом.

```
… VAR V : CHAR;
        OTVET : STRING;
        …
      CASE  V  OF
         'A' . . 'Z','a' . . 'z' : OTVET := 'буква';
         '0' . . '9' : OTVET := 'цифра'
      ELSE
          OTVET := 'специальный символ'
      END; …
```

2.3.5 Операторы цикла.

Операторы цикла используются для многократного повторения входящих в их состав операторов. В языке Турбо Паскаль различают операторы цикла типа арифметической прогрессии (оператор цикла со счетчиком – **FOR**) и операторы цикла итерационного типа (**WHILE** и **REPEAT**).

Оператор цикла типа арифметической прогрессии используется, если заранее известно количество повторений цикла и шаг изменения параметра цикла +1 или –1.

```
FOR<параметр цикла>:=<выраж.1>{ TO     }<выраж.2> DO
                              { DOWNTO }
                  < оператор > ;
```

< параметр цикла > - это переменная цикла любого порядкового типа (целого, символьного, перечисляемого, интервального);

TO – шаг изменения параметра цикла +1;

DOWNTO - шаг изменения параметра цикла -1;

< выражение 1> - начальное значение параметра цикла, выражение того же типа, что и параметр цикла;

< выражение 2> -конечное значение параметра цикла, выражение того же типа, что и параметр цикла;

< оператор>- тело цикла - простой или составной оператор.

При выполнении оператора **FOR** выполняются следующие действия:

- вычисляется < выражение 1> , которое присваивается параметру цикла;
- проверяется условие окончания цикла: <параметр цикла> больше <выражения 2> при использовании конструкции **TO** и <параметр цикла> меньше <выражения 2> при использовании конструкции **DOWNTO**;
- выполняется тело цикла;
- наращивается (**TO**) или уменьшается (**DOWNTO**) на единицу параметр цикла;

- все этапы, кроме первого, циклически повторяются.

При использовании оператора необходимо помнить:

- Внутри цикла **FOR** нельзя изменять начальное, текущее или конечное значения параметра цикла.
- Если в цикле с шагом +1 начальное значение больше конечного, то цикл не выполнится ни разу. Аналогично для шага -1, если начальное значение меньше конечного.
- После завершения цикла значение параметр цикла считается неопределенным, за исключением тех случаев, когда выход из цикла осуществляется оператором **GOTO** или с помощью процедуры **BREAK**.
- Телом цикла может быть другой оператор цикла.

Например, для того, чтобы вычислить значение факториала **F=N!** можно воспользоваться следующими операторами:

a)
```
… F:=1;
FOR I:=1 TO N DO
    F:=F*I; …
```

b)
```
… F:=1;
FOR I:=N DOWNTO 1 DO
    F:=F*I; …
```

В следующем примере цикл выполняется 26 раз и SIM принимает значения всех латинских букв от ’A’ до ‘Z’.

```
…
FOR  SIM:=’A’ TO ‘Z’ DO
    WRITELN( SIM);
…
```

Если телом цикла является другой цикл, то циклы называются *вложенными* или *сложными*. Цикл, содержащий в себе другой цикл, называют *внешним*. Цикл, содержащийся внутри другого цикла, называется *внутренним*. Внутренний и внешний циклы могут быть любыми из трех видов**:** **FOR**, **WHILE** или **REPEAT**. При построении вложенных циклов необходимо, чтобы все операторы внутреннего цикла полностью находились в теле внешнего цикла. Возможная глубина вложенности циклов ограничивается объемом памяти компьютера. Вначале выполняется самый внутренний цикл при фиксированных значениях параметров циклов с меньшим уровнем вложенности, затем изменяется параметр цикла следующего (за внутренним) уровня и снова выполняется самый внутренний цикл и т.д.

Пример. Вычислить значение Y , определяемое по формуле

$$Y = \sum_{i=1}^{N} \prod_{j=1}^{M} A_{ij}$$

```
PROGRAM SP;
   CONST N=10;
            M=15;
  VAR A: ARRAY [1..N,1..M] OF REAL;
         I,J: INTEGER;
         P,Y: REAL;
  BEGIN
        FOR I:=1 TO N DO
        FOR J:=1 TO M DO
             READLN(A[I,J]);
        Y:=0;
        FOR I:=1 TO N DO
              BEGIN
                P:=1;
                FOR J:=1 TO M DO
                  P:=P*A[I,J];
                Y:=Y+P
              END;
       WRITELN('Y=',Y)
   END.
```

Операторы цикла итерационного типа используются обычно в том случае, если число повторений цикла заранее неизвестно или шаг изменения параметра цикла отличен от +1 или –1.

Оператор цикла с предусловием:

WHILE **<**логическое выражение **> DO** **<**оператор**>;**

Логическое выражение вычисляется перед каждым выполнением тела цикла. Если логическое выражение принимает значение **TRUE**, то тело цикла выполняется, если значение **FALSE**, происходит выход из цикла. Тело цикла может не выполниться ни разу, если логическое выражение сразу ложно. Телом цикла является простой или составной оператор.

Любой алгоритм, реализуемый с помощью оператора FOR, может быть записан с использованием конструкции **WHILE.** Например, вычисление значения факториала **F=N!:**

```
… F:=1;
  I:=1;
  WHILE I<=N DO
   BEGIN
       F:=F*I;
       I:=I+1;
   END;   …
```

В следующем примере требуется подсчитать значение Sin (x) с использованием разложения функции в ряд:

$$Y = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots = \sum_{i=1}^{\infty} U_i$$

В сумму включить только те члены ряда, для которых выполняется условие:

$$|U_i| > \varepsilon$$

т.е. очередной член ряда должен быть больше заданной точности вычислений ε.

```
PROGRAM SINX;
VAR X,Y,E,U,Z : REAL;
             K: INTEGER;
BEGIN
     READLN(X,E);
     K:=0;
     Y:=0;
     U:=X;
     Z:=SQR(X);
     WHILE  ABS(U)>E DO
    BEGIN
         Y:=Y+U;
          K:=K+2;
          U:= -U* Z/ (K*(K+1));
     END;
     WRITELN( ‘ SIN(X)=’, SIN(X), ‘  Y=’,Y);
END.
```

Для проверки правильности работы программы в оператор вывода включена печать значения синуса, вычисленного при помощи стандартной функции. Если полученное значение отличается от рассчитанного при помощи стандартной функции не более, чем на точность, можно считать, что программа работает правильно.

Рассмотрим пример вычисления значения квадратного корня из числа X по итерационной формуле

$$Y_{i+1} = (Y_i + X/Y_i)/2 \quad \text{с точностью} \quad |Y_{i+1} - Y_i| <= \varepsilon$$

Начальное приближение Y0=A является параметром.

```
PROGRAM SQRTX;
VAR   X:  REAL;  {аргумент                     }
      EPS:  REAL;  {точность вычисления        }
       Y0:  REAL;  {предыдущее приближение}
       Y1:  REAL;  {очередное приближение    }
        A:  REAL;  {начальное приближение    }
BEGIN
      READLN( A,EPS,X);
      IF  X>0 THEN
      BEGIN
            Y0:=A;
            Y1:=(Y0+X/Y0)/2;
             WHILE ABS(Y1-Y0)>EPS DO
             BEGIN
                  Y0:=Y1;
                   Y1:=(Y0+X/Y0)/2
             END;
             WRITELN('Y1=',Y1,'  при X=',X)
      END
                 ELSE
      WRITELN('Число ', X, ' меньше нуля');
END.
```

Оператор цикла с постусловием:

```
REPEAT
       < оператор 1 >  [;
       < оператор 2 >] [;
              ...
       < оператор n >]
UNTIL   <логическое выражение>;
```

Данная конструкция оператора цикла используется, если число повторений цикла заранее неизвестно, но известно условие выхода из цикла. Управляющее циклом логическое выражение является условием выхода из цикла. Если оно принимает значение **TRUE**, то выполнение цикла прекращается. При использовании оператора **REPEAT** цикл выполняется хотя бы один раз. В отличие от других операторов цикла опе-

ратор данного вида не требует операторных скобок **BEGIN - END**, так как их роль выполняют **REPEAT - UNTIL.**

Вычисление **F=N!** с использованием конструкции **REPEAT – UNTIL** будет выглядеть следующим образом:

```
. . .
F:=1;
I:=1;
REPEAT
    F:=F*I;
    I:=I+1;
UNTIL  I>N;
 . . .
```

Рассмотрим другой пример использования этого оператора. Вводится последовательность чисел. Определить количество элементов кратных 7.

```
PROGRAM REP;
VAR  A,K: INTEGER;
       C    : CHAR;
BEGIN
     K:=0;
     REPEAT
          WRITELN('  Введите очередное число ');
          READLN( A);
          IF A MOD 7=0 THEN K:=K+1;
          WRITELN('Хотите выйти  из цикла? Д/Y' );
          READLN( C ) ;
    UNTIL  ( C='Д' ) OR ( C='Y') ;
    WRITELN('KOL=',K);
END.
```

Здесь условием выхода из цикла является ввод символов **Д** или **Y** при ответе на вопрос о выходе из цикла. Если вводится одна из этих букв, логическое выражение, записанное после **UNTIL,** становится **TRUE** и происходит выход из цикла.

В следующем примере требуется поменять местами максимальный и минимальный элементы, найденные среди элементов четных строк матрицы A(M,N).

```
PROGRAM OBMEN;
VAR A: ARRAY[1..30,1..30] OF INTEGER;
      I,J : INTEGER;
      K1,L1,K2,L2 : INTEGER;
      T,M,N: INTEGER;
BEGIN
      READLN(M,N);
      FOR I:=1 TO M DO
            FOR J:=1 TO N DO
                  READLN( A[I,J]);
      K1:=1;L1:=1;  {координаты максимального элемента}
       K2:=1;L2:=1;  {координаты минимального элемента}
       I:=2;
       WHILE I<=M DO
            BEGIN
                   FOR J:=1 TO N DO
                  IF A[K1, L1]< A[I,J] THEN
                  BEGIN
                        K1:=I;L1:=J;
                  END
                                         ELSE
                     IF A[K2, L2]> A[I,J] THEN
                     BEGIN
                           K2:=I;L2:=J;
                     END;
            I:=I+2;
       END;
       T:= A[K1, L1];
       A[K1, L1]:= A[K2, L2];
       A[K2, L2]:=T;
        FOR I:=1 TO M DO
        BEGIN
              FOR J:=1 TO N DO
                    WRITE ( A[I,J] :6);
              WRITELN ;
        END;
END.
```

Здесь используется цикл **WHILE** для индексации строк, т.к. нас интересуют только четные строки, следовательно, шаг для строк должен быть равен 2. В цикле **FOR** этого сделать нельзя.

В языке Турбо Паскаль 7.0 имеются процедуры **BREAK** и **CONTINUE** . Эти процедуры могут использоваться только внутри циклов

FOR, WHILE или **REPEAT**. Процедура **BREAK** прерывает выполнение цикла и вызывает переход к оператору, следующему за циклом (может использоваться вместо оператора **GOTO**).Процедура **CONTINUE** осуществляет переход к следующему повторению цикла с пропуском последующих операторов тела цикла.

Например, необходимо определить номер первого элемента одномерного массива, оканчивающегося на 3.

```
PROGRAM KON;
VAR A:ARRAY [1.. 30] OF INTEGER;
     FL: BOOLEAN;
     I,N: INTEGER;
BEGIN
          READLN(N);
          FOR I:=1 TO N DO
                READLN( A[I] );
          FL:=FALSE;
          FOR I:=1 TO N DO
    BEGIN
               IF A[I] MOD 10 <> 3 THEN CONTINUE;
               WRITELN('Номер первого числа на 3  ',I);
                FL:=TRUE;
                BREAK;
         END;
         IF NOT FL THEN WRITELN(' нет чисел на 3');
END.
```

Если встречается число, оканчивающееся на 3, происходит вывод сообщения об этом, флаг – переменная **FL** – становится равным **TRUE** и программа заканчивает свою работу, т.к. цикл прерывается. Если же такое число не встречается, происходит нормальное завершение цикла, переменная **FL** остается равной **FALSE** и выводится сообщение о том, что таких чисел нет.

Вопросы к главе 2.

1. Дать определение языка программирования.
2. Дать классификационную характеристику языков программирования.
3. Определить особенности языков высокого уровня.
4. Назначение и особенности машинно-ориентированных языков программирования.
5. Основные особенности языка Турбо Паскаль.
6. Алфавит языка Турбо Паскаль.

7. Особенности построения арифметических, строковых и логических выражений в Паскале.
8. Приоритет выполнения операций в выражениях различного типа.
9. Классификация типов данных.
10. Описание типизированных констант.
11. Описание нетипизированных констант.
12. Описания целых типов.
13. Описания вещественных типов.
14. Что такое идентификатор?
15. Описание интервальных типов.
16. Описание перечисляемых типов.
17. Какую структуру имеет программа на Турбо Паскале?
18. Способы написания комментариев в Турбо Паскале.
19. Что такое пустой оператор?
20. Что такое операторные скобки?
21. Что такое составной оператор?
22. Назначение и особенности использования оператора безусловного перехода.
23. Способы записи условного оператора.
24. Назначение и варианты использования оператора выбора.
25. Форма записи меток варианта.
26. Формы оператора цикла.
27. Принципы выбора типа оператора цикла.
28. Особенности записи и использования оператора цикла типа арифметической прогрессии.
29. Особенности записи и использования оператора цикла итерационного типа с предусловием.
30. Особенности записи и использования оператора цикла итерационного типа с постусловием.
31. Особенности организации вложенных циклов.

3. Структурированные типы данных.

Данные одинакового простого типа (кроме вещественного) могут объединяться в ***множество***.

В общем виде тип множество описывается:

TYPE <идентификатор типа**>= SET OF <**тип компонент**>;**

Тип компонент множества (базовый тип) обычно интервальный или перечисляемый. Значения переменной типа множества изображаются перечислением компонент, разделенных запятыми и заключенных в квадратные скобки.

Например,

```
TYPE  INTERVAL= 5..10;
      MN=SET OF INTERVAL;
VAR   PR: MN;
```

PR может принимать значения:

[5,6,7,8,9,10], [5], [6],…, [5,6], [5,7],…, [6,7,8],…, [ ],

где [ ] - пустое множество, т.к. оно не содержит выражения, указывающего базовый тип. Оно совместимо со всеми типами множеств.

В языке Турбо Паскаль на множества накладываются следующие ограничения:

- Число элементов множества не должно превышать 256.
- Элементами множества могут быть только данные простых типов (кроме вещественных).
- Элементы, входящие в состав множества должны быть определены заранее.
- Порядок элементов множества произвольный.

3.1 Свойства множеств.

1) Если все элементы одного множества совпадают с элементами другого множества, то они (множества) считаются равными.

   Множества [1..5] и [1,2,3,4,5] равны.
2) Если все элементы одного множества являются членами другого множества, то 1 множество включено во 2 множество.

   [‘C’,’E’] включено в множество [‘A’..’Z’].
3) Если нижнее граничное значение больше, чем верхнее граничное значение, то множество является пустым.

   [5..1] – пустое множество, т.е. эквивалентно [ ].

Переменным типа множество присваивают результат выражений над множествами с помощью обычного знака присваивания.

3.2 Операции над множествами.

1) * - пересечение множеств (произведение).

Пусть X=[1,2,3,4,7,9], а Y=[1,2,3,10,5], тогда при выполнении оператора Z: = X * Y; множество Z содержит элементы, которые принадлежат как множеству X, так и множеству Y, т.е. Z=[1,2,3].

2) + - объединение множеств (сумма).

Пусть X=[1,2,3,4,7,9], а Y=[1,2,3,10,5], тогда при выполнении оператора Z:= X + Y; множество Z содержит элементы, которые принадлежат либо множеству X, либо множеству Y, т.е. Z=[1,2,3,4,5,7,9,10].

3) - - разность множеств или относительное дополнение.

Пусть X=[1,2,3,4,7,9], а Y=[1,2,3,10,5], тогда при выполнении оператора Z: = X - Y; множество Z содержит элементы X, которые не принадлежат множеству Y, т.е. Z=[4,7,9].

4) = - проверка на равенство.

Если все элементы множества X являются элементами множества Y, то результатом выполнения операции будет TRUE, в противном случае – FALSE. Пусть X=[1,2,3,4,7,9], а Y=[1,2,3,10,5], тогда при выполнении оператора F: = X = Y; F= FALSE.

5) <> - проверка на неравенство.

Если какие-то элементы множества X не являются элементами множества Y или наоборот, то результатом выполнения операции будет TRUE, в противном случае – FALSE. Пусть X=[1,2,3,4,7,9], а Y=[1,2,3,10,5], тогда при выполнении оператора F:=X<>Y; F= TRUE.

6) >= - проверка на включение.

Если все элементы множества Y являются элементами множества X, то результатом выполнения операции будет TRUE, в противном случае – FALSE. Пусть X=[1,2,3,4,7,9], а Y=[1,2,3], тогда при выполнении оператора F: = X >= Y; F= TRUE.

7) <= - проверка на включение.

Если все элементы множества X являются элементами множества Y, то результатом выполнения операции будет TRUE, в противном случае – FALSE. Пусть X=[1,2,3], а Y=[1,2,3,10,5], тогда при выполнении оператора F: = X < = Y; F= TRUE.

8) IN - проверка принадлежности отдельного элемента множеству.

Слева от знака операции записывается выражение того же типа, что и базовый, а справа – множество. Если левый операнд является эле-

ментом множества Y, то результатом выполнения операции будет TRUE, в противном случае – FALSE. Пусть A=5, а Y=[1,2,3,10,5], тогда при выполнении оператора F: = A IN Y; F= TRUE.

Отношения > и < для множеств не определены.

Рассмотрим некоторые примеры, в которых используются операции над множествами.

Вводится строка символов. Требуется составить программу, распознающую является ли введенная строка идентификатором.

```
VAR
    SIM:CHAR;
    FL:BOOLEAN;
    S:STRING[127];
    I,L:BYTE;
BEGIN
    READLN(S);
    L:=LENGTH(S);
    SIM:=S[1];
    IF NOT(SIM IN ['A'..'Z']) THEN FL:=FALSE
                              ELSE FL:=TRUE;
    I:=2;
    WHILE(I<=L) AND FL DO
    BEGIN
        SIM:=S[I];
        IF NOT (SIM IN ['A'..'Z','0'..'9','_']) THEN FL:=FALSE
                                                 ELSE I:=I+1
    END;
    IF FL THEN WRITELN('идентификатор')
          ELSE WRITELN('    нет      ')
END.
```

Базовым типом множества в данной программе является тип **CHAR.** Поэтому для выполнения операции проверки принадлежности элемента множеству в правой части также требуется тип **CHAR**. Для выделения символа из строки используется индексирование, т.к. выделение символа с помощью функции **COPY** дает результат типа **STRING**, что недопустимо для множеств.

В следующем примере требуется вывести на экран все простые числа из интервала от 2 до 255.

```
USES CRT;
CONST N=255;
VAR  ISX, REZ: SET OF 2..N;
        I,J: INTEGER;
BEGIN
      CLRSCR;
      REZ:=[ ];
      ISX:=[2..N];
      I:=2;
      REPEAT
            WHILE NOT(I IN ISX) DO
                 I:=SUCC(I);
            REZ:=REZ+[I];
            J:=I;
                   WHILE J<=N DO
            BEGIN
                  ISX:=ISX-[J];
                  J:=J+I
            END;
      UNTIL ISX=[ ];
      WRITELN;
      FOR I:=1 TO 255 DO
           IF I IN REZ THEN
               WRITE (‘ ‘,I,’ ‘);
      READKEY
END.
```

В основу программы положен алгоритм, известный под названием «решето Эратосфена», т.е. метод отсеивания составных чисел, при котором последовательно вычеркиваются числа, делящиеся на 2, 3, 5 и т.д.; первое число, остающееся после каждого этапа, является простым и добавляется к результату. Исходное множество в начале программы содержит все натуральные числа от 2 до 255. Результат также является множеством, к которому последовательно добавляются элементы. Условием окончания работы является пустое множество, получившееся вместо исходного.

В следующем примере вводится строка символов. Разделителями будем считать символы «;?,. <>”’». Назовем словом любую последовательность символов, ограниченную с одной или с двух сторон разделителями. Требуется удалить слова, состоящие не более, чем из 2 различных символов. Если таких слов нет, то выдать сообщение.

```
PROGRAM DELWD;
USES CRT;
CONST  R:SET OF CHAR= [' ',';',',','<','"','"','>','?','.'];
VAR S,C:STRING;
        I,NS:BYTE;
        F:BOOLEAN;
BEGIN
      CLRSCR;
      WRITE('S='); READLN(S);
       I:=1;  F:=TRUE;
      WHILE I<=LENGTH(S) DO
      BEGIN
           IF NOT(S[I] IN R) THEN
           BEGIN
                   NS:=I; C:='';
                   WHILE (I<=LENGTH(S)) AND
                                                NOT(S[I] IN R) DO
                   BEGIN
                         IF POS(S[I],C)=0 THEN C:=C+S[I];
                         INC(I);
                   END;
                   IF LENGTH(C)<=2 THEN
                   BEGIN
                          DELETE (S,NS,I-NS);
                          F:=FALSE;
                          I:=NS-1
                   END;
             END;
             I:=I+1
       END;
       IF F THEN WRITELN('NO')
             ELSE WRITELN(S);
       READKEY
 END.
```

Во вспомогательную строку C выбираются различные буквы во время прохода по слову. Если длина этой строки оказывается меньше или равна 2, слово нужно удалять. После удаления необходимо скорректировать значение переменной основного цикла для выделения очередного слова.

В следующем примере вводится строка символов. Требуется определить, содержит ли данная строка цифры и выдать сообщение об этом. Распечатать имеющиеся в строке цифры.

```
PROGRAM DIGIT;
USES CRT;
VAR
      CF:SET OF 0..9;
      S:STRING;
      I,K:BYTE;
BEGIN
      CLRSCR;
      WRITE('S='); READLN(S);
      CF:=[ ];
      I:=1;
      WHILE I<=LENGTH(S) DO
      BEGIN
            K:=ORD(S[I])-ORD('0');
            IF K IN [0..9] THEN CF:=CF+[K];
            I:=I+1
      END;
      IF CF=[ ] THEN WRITELN('NO')
                ELSE
                BEGIN
                     FOR I:=0 TO 9 DO
                          IF I IN CF THEN WRITE(I:3);
                     WRITELN
                END;
      READKEY
END.
```

Так как цифры имеют последовательные коды, значение самой цифры можно определить, вычитая из кода цифры код цифры 0, что и выполняется в программе. Цифры, обнаруженные в строке, записываются во множество, которое затем выводится на экран.

3.3 Описание записи (RECORD).

Запись – это структура данных, состоящая из фиксированного числа компонент, называемых полями. Каждое поле имеет свой идентификатор и тип. К компонентам записи возможен прямой доступ и они могут выборочно обновляться. Идентификатор в самой записи должен быть уникальным. Для обращения к отдельным полям записи указываются составные имена: имя записи, после которого ставится точка и записывается идентификатор поля. Запись можно передавать в качестве параметра процедуры или функции, но значением функции запись быть не может.

В общем виде описание типа для записи можно представить:

```
TYPE <идентификатор типа>= RECORD
    <идентификатор 11>[,< идентификатор 12>,…]: <тип 1>;
    < идентификатор 21>[,< идентификатор 22>,…]: <тип 2>;
                          . . .
       END;
```

Например,

```
TYPE  TA= RECORD
       P1 : REAL;
       P2 : CHAR;
       P3 : BYTE
      END;
VAR A: ARRAY[1..10] OF TA;
```

Здесь описан одномерный массив, каждый элемент которого представляет собой запись, имеющую структуру типа **TA.**

Запись может объявляться и непосредственно в разделе описания переменных.

```
VAR C : RECORD
       P1 : REAL;
       P2 : CHAR;
       P3 : BYTE
      END;
```

Рассмотрим пример. Дан массив записей со следующей структурой:

- шифр группы;
- номер зачетной книжки;
- код дисциплины;
- оценка.

Требуется определить средний балл студентов группы ДС101. При вводе массива последняя запись имеет шифр группы «99999».

```
PROGRAM SRBALL;
TYPE ZAP=RECORD
          SHG:STRING[5];
          NZK:INTEGER;
          KD:1..100;
          OC:2..5
          END;
VAR MAS:ARRAY[1..100] OF ZAP;
        K,N,I:BYTE;
        SUM:REAL;
BEGIN
      I:=0;
      REPEAT
              INC(I);
              READLN (MAS[I].SHG, MAS[I].NZK, MAS[I].KD,
                                                      MAS[I].OC)
       UNTIL MAS[I].SHG='99999';
       N:=I; SUM:=0; K:=0;
       FOR I:=1 TO N DO
       IF MAS[I].SHG='ДС101' THEN
       BEGIN
             SUM:=SUM+MAS[I].OC;
             INC(K)
       END;
       IF K<>0 THEN SUM:=SUM/K;
      WRITELN ('Средний балл в группе ДС-101=',SUM)
END.
```

3.4 Оператор присоединения.

При обращении к компонентам записи используются составные имена. Для сокращения имен и повышения удобства работы с записями применяется оператор присоединения **WITH**.

WITH <идентификатор переменной типа **RECORD> DO**
< оператор **>;**

Тогда в операторе при ссылке на компоненты записи имя переменной можно опускать.

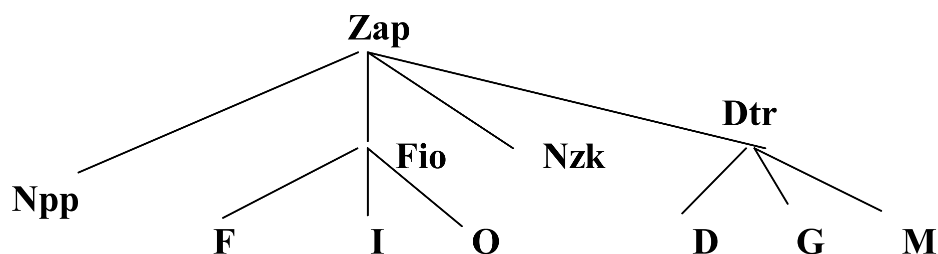
При использовании оператора присоединения фрагмент рассмотренной ранее программы будет выглядеть:

```
  . . .
I:=0;
REPEAT
       INC(I);
       WITH MAS[I] DO
       READLN(SHG,NZK,KD,OC)
 UNTIL MAS[I].SHG='99999';
 N:=I; SUM:=0; K:=0;
 FOR I:=1 TO N DO
 WITH MAS[I] DO
 IF SHG='ДС101' THEN
 BEGIN
       SUM:=SUM+OC;
       INC(K)
 END;
   . . .
```

Возможны вложенные описания записи и вложенные конструкции **WITH.** Рассмотрим пример вложенных описаний. Пусть запись о студентах содержит следующие поля:

- номер по порядку;
- ФИО (содержит в свою очередь поля – фамилия, имя, отчество),
- номер зачетной книжки;
- дата рождения (содержит поля –год, месяц, день).

Тогда в виде графа структуру записи можно изобразить следующим образом:



Программа ввода и подсчета количества введенных записей такой структуры может выглядеть:

```
USES CRT;
TYPE ZAP=RECORD
            NPP:BYTE;
            FIO:RECORD
           F,I,O:STRING[15];
            END;
            NZK:WORD;
            DTR:RECORD
               G:1970..2000;
               M:STRING[3];
               D:1..31
            END;
       END;
VAR A:ZAP;
     K,N:BYTE;
BEGIN CLRSCR;
     K:=0;
     WITH A DO
         WITH FIO DO
            WITH DTR DO
     REPEAT
          INC(K);
          WRITELN('ВВОД ');
          READLN(NPP);
          READLN(F);
          READLN(I);
          READLN(O);
          READLN(NZK);
          READLN(G);
          READLN(M);
          READLN(D);
     UNTIL D=99;
     WRITELN(K);
     READKEY
END.
```

ИЛИ

```
USES CRT;
TYPE ZAP=RECORD
          NPP:BYTE;
          FIO:RECORD
        F,I,O:STRING[15];
               END;
        NZK:WORD;
        DTR:RECORD
            G:1970..2000;
            M:STRING[3];
            D:1..31
          END;
         END;
    VAR A:ZAP;
       K,N:BYTE;
    BEGIN CLRSCR;
          K:=0;
        WITH A,FIO,DTR DO
          REPEAT
               INC(K);
        WRITELN('ВВОД ');
              READLN(NPP);
              READLN(F);
              READLN(I);
              READLN(O);
              READLN(NZK);
              READLN(G);
              READLN(M);
              READLN(D);
          UNTIL D=99;
              WRITELN(K);
              READKEY
    END.
```

3.5 Запись с вариантами.

Состав и структура записи могут динамически меняться в зависимости от значения какого-либо из своих полей, называемого полем-признаком.

В общем виде описание записи с вариантами выглядит так:

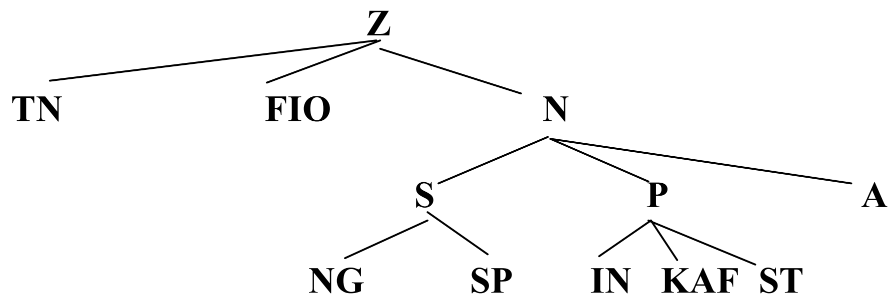
```
TYPE <идентификатор типа >= RECORD
        <идентификатор поля 1>:<тип 1>;
        <идентификатор поля 2>:<тип 2>;
                    . . .
        CASE <селектор>:<тип селектора> OF
      <метка варианта 1>:(<поле варианта 11>:<тип 11>
  [;<поле вар-та 12>:<тип 12>;<поле варианта 13>:<тип 13>;. . .]);
      <метка варианта 2>:(<поле варианта 21>:<тип 21>
  [;<поле вар-та 22>:<тип 22>;<поле варианта 23>:<тип 23>;. . .]);
      <метка варианта k>:(<поле варианта k1>:<тип k1>
  [;<поле вар-та k2>:<тип k2>;<поле варианта k3>:<тип k3>;. . .]);
                                        . . .
      <метка варианта m>:( )
         END;
```

В этом описании вариантная часть записывается после постоянной части, к которой относятся поля 1, 2 . . . , и может быть только одна в записи. Метки варианта должны иметь такой же тип, как у селектора. Если какой-либо метке варианта не соответствуют поля, то записываются пустые круглые скобки, как у метки варианта m.

Пусть требуется описать запись следующей структуры. В каждой записи имеются поля, содержащие табельный номер и фамилию. В зависимости от того, кому принадлежит запись, состав остальных полей может быть разным:

- для студентов поля: номер группы и специальность;
- для преподавателей: институт, кафедра, стаж работы;
- для сотрудников дополнительных полей нет.

В виде графа структуру записей можно изобразить:



Описание соответствующей записи структуры данных:

```
TYPE TZ=RECORD
        TN:BYTE;
        FIO:STRING;
        CASE N:CHAR OF
          'P': ( IN:BYTE; KAF:STRING; ST:BYTE );
          'S': ( NG:BYTE; SP:INTEGER );
          'A': ( )
        END;
VAR Z:TZ;
```

Вопросы к главе 3.

1. Как описываются множества?
2. Варианты использования множеств.
3. Основные операции над множествами.
4. Способы описания записи.
5. Обращение к компонентам записи.
6. Использование оператора присоединения With.
7. Назначение и общий вид записи с вариантами.

4. Использование подпрограмм в Турбо Паскале.

4.1 Структура программы на языке Паскаль.

Синтаксически программа на языке Паскаль делится на 2 части: заголовок и программный блок.

Общий вид заголовка:

PROGRAM <имя программы>**[(**<список файлов>**)];**

Заголовок программы может отсутствовать. Стандартные файлы **INPUT** (входной) и **OUTPUT** (выходной) также могут опускаться, т.к. принимаются по умолчанию.

Блок программы состоит из описательной и исполнительной частей (описательная часть предшествует исполнительной части) и включает следующие разделы:

LABEL <описание меток>**;** - раздел описания меток
CONST <описание констант>**;** - раздел описания констант
TYPE <описание типов>**;** - раздел описания типов
VAR <описание переменных>**;** - раздел описания переменных
PROCEDURE <описание процедуры**> ;** - раздел описания
FUNCTION <описание функции>**;** процедур и функций
BEGIN
<исполнительная часть программы> - раздел операторов
END.

Текст программы записывается в виде строк длиной не более 127 символов. В Турбо Паскале порядок следования разделов описаний произвольный и каждый из разделов может повторяться несколько раз или отсутствовать. Раздел операторов (начинается с **BEGIN** и заканчивается **END)** состоит из операторов, разделенных точкой с запятой. Исполнительная часть программы является обязательной, может быть единственной частью программы.

Видом работы компилятора можно управлять директивами. Их включают в исходный текст в виде комментариев со специальным синтаксисом. Если процедура или функция хранятся в виде отдельного файла, то для включения их описания в исходный текст программы компилятору задается директива **INCLUDE** следующего вида:

{$I <имя файла>**}**,

которая должна быть помещена в разделе описаний процедур и функций.

Например,

```
PROGRAM A1;
VAR ...
{$I  B1.PAS}
BEGIN
    ...
END.
```

Файл **B1.PAS** может иметь вид:

```
PROCEDURE PP;
VAR ...
BEGIN
        ...
END;
```

4.2 Описание и вызов процедур.

Для реализации многократно повторяющихся участков вычислений и для обеспечения модульности программ в языке Турбо Паскаль предусмотрена возможность использования процедур и функций.

Процедура - это поименованное сложное действие, которое представляет собой совокупность операторов, вычисляющих некоторое число результатов в зависимости от некоторого числа аргументов.

Процедура или ***функция*** (общее название - *подпрограмма*) определяется в разделе описаний основной программы или другой процедуры(функции). Процедура(функция) имеет ту же структуру, что и основная программа, т.е. состоит из заголовка, описательной части и выполняемой части.

Синтаксис *заголовка процедуры*:

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

Например:

PROCEDURE PR1 (A,B,C : INTEGER; VAR S: REAL);

- Здесь **PR1** – имя процедуры, а **A,B,C,S** – имена переменных, являющихся параметрами.

В отличие от основной программы заголовок в процедуре обязателен, но завершается процедура не точкой, а точкой с запятой. Описание процедуры выполняется с формальными параметрами.

Оператор процедуры служит для вызова процедуры из основной программы или из другой процедуры(функции).

Вызов осуществляется в следующей форме:

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

Таким образом, для приведенного заголовка процедуры можно написать такой оператор вызова:

PR1 (A,B,C,S);

Формат списка параметров в заголовке процедуры и при вызове процедуры отличаются. При вызове переменные, константы или выражения следуют через запятую, а в заголовке запись переменных напоминает объявление переменных в разделе описания переменных. Для всех элементов списка должен быть указан тип данных. Несколько переменных, отнесенных к одному типу , пишутся через запятую, а группы переменных различного типа отделяются точкой с запятой. После обращения к процедуре управление передается на выполнение следующего за вызовом процедуры оператора.

4.3 Описание функции.

Функция предназначена для вычисления какого-либо одного значения и используется в выражениях аналогично стандартным функциям.

Синтаксис *заголовка функции*:

FUNCTION < имя функции >**[(**<список формальных параметров>**)]:** <тип результата>;

Например:

FUNCTION PRF (A,B,C: INTEGER) : REAL;

Отличие описания функции от процедуры:

- результатом обращения к функции может быть одно единственное значение;
- идентификатор результата не указывается в списке формальных параметров;
- в выполняемой части функции , хотя бы один раз , имени функции должно быть присвоено значение результата (чаще всего перед выходом из функции);
- после списка формальных параметров задается тип результата;
- после обращения к функции управление передается на выполнение следующей операции данного выражения (в соответствии с приоритетом).

Для вызова функции используется *указатель функции* (имя функции со списком фактических параметров), который обязательно должен быть частью какого-либо выражения (входить в правую часть оператора присваивания, присутствовать в списке данных оператора вывода, в логическом выражении условного оператора и т.д.). Для приведенного заголовка функции вызов функции может быть осуществлен одним из следующих способов:

```
S:=PRF ( A,B,C);
Writeln ( PRF ( A,B,C));
If  PRF ( A,B,C)>20 then K=K+1;
```

4.4 Формальные и фактические параметры.

При описании процедуры (функции) в ее заголовке могут быть указаны параметры следующих видов:

- параметры-значения;
- параметры-переменные;
- параметры-константы;
- параметры-процедуры;
- параметры-функции.

При записи параметров необходимо помнить:

- число формальных и фактических параметров должно быть одинаково;
- порядок следования и тип фактических параметров должен совпадать с порядком и типом соответствующих формальных параметров;
- идентификаторы формальных и фактических параметров могут совпадать;
- формальные параметры в языке Турбо Паскаль в заголовке находятся вместе с описаниями и объявлять их в разделе описаний процедуры(функции) не требуется;
- формальные параметры должны иметь простые или ранее определенные типы.

При передаче в подпрограмму массива его тип объявляют предварительно в разделе описания типов **TYPE**.
Например.

```
TYPE  TV=ARRAY [1..30] OF  INTEGER;
        TM=ARRAY [1..20,1..20]  OF  REAL;
              ...
PROCEDURE TOP ( A:TM; VAR  B: TV ; N: INTEGER);
              ...
```

Здесь описаны два типа массивов. **TV** – для одномерного массива и **TM** для двумерного массива. Затем в списке формальных параметров для переменных **A** и **B** используются эти ранее определенные типы при описании соответственно матрицы и вектора.

Список параметров, задаваемых в заголовке процедуры или функции, обеспечивает связь подпрограммы с вызывающей программой. Через него в подпрограмму передаются исходные данные и возвращается результат (в процедуре). В языке Турбо Паскаль предусмотрены два принципиально отличающихся механизма передачи параметров : по значению и по ссылке.

Параметры-значения.

При передаче параметров по значению в стеке, в котором осуществляется выделение памяти под внутренние (локальные) переменные подпрограммы, выделяется дополнительная память, в которую копируются значения соответствующих фактических параметров. В вызывающей программе в качестве аргумента подпрограммы для параметра-значения может использоваться не только переменная, но и выражение. После завершения работы подпрограммы выделенная этим параметрам память становится недоступной, поэтому передача параметров по значению не может использоваться в подпрограммах для получения результатов.

Параметры-переменные.

При вызове по ссылке в подпрограмме память под передаваемые переменные не отводится. В подпрограмму передается не значение переменной, а ссылка на место в памяти соответствующего фактического параметра. Подпрограмма, выполняющая некоторые действия с этой переменной, в действительности производит действия с соответствующим фактическим параметром, поэтому после выполнения процедуры , изменения, выполненные над этой переменной, сохраняются. Перед записью параметров-переменных в списке формальных параметров указывается ключевое слово **VAR** (действует до " ; "). Для вычисляемых результатов могут быть использованы только параметры-переменные. Формальным параметрам-переменным не могут соответствовать в качестве фактических значений константы или выражения ,так как они не имеют адреса для передачи.

В качестве параметров-переменных могут быть использованы массивы и строки открытого типа, у которых не задаются размеры. Открытый массив представляет собой формальный параметр подпрограммы, описывающий базовый тип элементов, но не определяющий его размерность и границы. Индексация элементов в этом случае начинается с нуля. Верхняя граница открытого массива возвращается функцией **HIGH** . Такое описание возможно только для одномерных массивов. Для открытого массива в стеке создается его копия, что может вызвать переполнение стека.

Рассмотрим пример использования открытого массива. Пусть требуется подсчитать сумму элементов одномерного массива.

```
FUNCTION SUM (VAR A: ARRAY OF INTEGER):INTEGER;
      VAR  S,I : INTEGER;
      BEGIN
          S:=0;
          FOR I:=0 TO HIGH(A)  DO
          S:=S+A[I];
          SUM:=S;
      END;
```

В основной программе такой массив может быть описан даже как **Var A: array [-2 .. 3] of integer;** Фактические границы массива здесь значения не имеют. Важно только то, что количество элементов массива в данном случае равно 6.

Открытая строка может задаваться с помощью стандартного типа **OPENSTRING** и стандартного типа **STRING** с использованием директивы компилятора **{$P+}** .

Например,

```
PROCEDURE ZAP ( VAR ST : OPENSTRING; R: INTEGER );
```

или

```
{$P+}
PROCEDURE ZAP ( VAR ST : STRING; R: INTEGER );
```

В языке Турбо Паскаль можно устанавливать режим компиляции, при котором отключается контроль за совпадением длины формального и фактического параметра строки **{$V- }**. При передаче строки меньшего размера формальный параметр будет иметь ту же длину, что и параметр обращения; при передаче строки большего размера происходит усечение до максимального размера формального параметра. Контроль включается только при передаче параметров-переменных , для параметров - значений длина не контролируется.

Рассмотрим пример, в котором используются процедура и функция. Требуется написать процедуру, в которой для матрицы, содержащей M столбцов и N строк, необходимо составить вектор номеров столбцов, все элементы которых упорядочены по возрастанию или убыванию и являются простыми числами. В главной программе вводятся все входные данные, производится обращение к процедуре и осуществляется вывод полученных результатов.

```
USES CRT;
TYPE TMAS=ARRAY[1..100,1..100] OF WORD;
        TVECT=ARRAY[1..100] OF WORD;
VAR A:TMAS;
       V:TVECT;
  N,M,K:BYTE;
      I,J:BYTE;
PROCEDURE FORM(VAR X:TMAS;   {матрица}
           N,M:BYTE;     {количество строк и столбцов}
       VAR R:TVECT;  {результат - вектор}
           VAR K:BYTE);  {длина полученного вектора}
VAR I,J,Z,S:BYTE;
              F:BOOLEAN;
FUNCTION PROS(B:WORD):BOOLEAN;
{функция проверки простого числа}
VAR I:WORD;
BEGIN
       IF B<>1 THEN PROS:=TRUE
                   ELSE PROS:=FALSE;
       FOR I:=2 TO B DIV 2 DO
             IF B MOD I = 0 THEN PROS:=FALSE;
END;
BEGIN
       K:=0;
       FOR J:=1 TO M DO
       BEGIN
             Z:=0; S:=0; F:=TRUE;
             FOR I:=1 TO N-1 DO
             BEGIN
                   IF X[I,J]>X[I+1,J] THEN Z:=Z+1;
                   IF X[I,J]<X[I+1,J] THEN S:=S+1
             END;
             IF (Z = N-1) OR (S = N-1) THEN
                     BEGIN
                   FOR I:=1 TO N DO
                         IF NOT(PROS(X[I,J])) THEN F:=FALSE;
                   IF F THEN
                   BEGIN
                         K:=K+1; R[K]:=J
                   END;
             END;
     END;
END;
BEGIN
       WRITELN('Введите N и M:');
       READLN(N,M);
       WRITELN('Введите матрицу:');
       FOR I:=1 TO N DO
             FOR J:=1 TO M DO
                   READLN(A[I,J]);
       FORM(A,N,M,V,K);
       WRITELN('Результат:');
       FOR I:=1 TO K DO
             WRITE(V[I],'  ');
       READKEY
END.
```

В этом примере в процедуру передаются входные данные: двумерный массив и его размерность. Массив передается как параметр-переменная, чтобы в процедуре не выделялась память для его копии. Результаты: вектор и его размерность обязательно передаются как параметры-переменные. Функция проверки простого числа является внутренней для процедуры и недоступна из главной программы.

Параметры-константы.

Так как аргументы, передаваемые в процедуру или функцию , размещаются в стеке, то в случае передачи значением массива большого размера, может произойти переполнение стека. В языке Турбо Паскаль 7.0 введен описатель **CONST**, который может задаваться для формальных параметров подпрограмм. Аргумент, соответствующий такому параметру, передается по ссылке, подобно параметру с описателем **Var,** но в самой процедуре(функции) запрещается присваивать этому аргументу новое значение.

```
PROCEDURE  <имя процедуры> (CONST <имя константы>:
                                         <тип>; ...);
FUNCTION <имя функции> (CONST <имя константы> :
                              <тип> ; ...): <тип результата> ;
```

Параметр-константу нельзя передавать в качестве параметра в другую подпрограмму.

Параметры-процедуры и параметры-функции.

Для объявления процедурного типа используется заголовок подпрограммы, в котором опускается имя процедуры (функции).

Например:

```
TYPE
    TPR1= PROCEDURE( X,Y : REAL; VAR Z : REAL);
    TPR2= PROCEDURE ;
    TF1= FUNCTION: STRING;
    TF2=FUNCTION ( VAR S: STRING) : REAL;
```

Ниже приведен пример использования функции **FF** в качестве параметра другой функции **RR**.

```
USES CRT;
TYPE FUN=FUNCTION (X,Y: REAL): REAL;
VAR   ...
FUNCTION FF (X,Y: REAL): REAL; FAR;
          ...
BEGIN  ...   END;
FUNCTION RR  (X,Y: REAL; F : FUN): REAL; FAR;
          ...
BEGIN  ...   END;
PROCEDURE TP (VAR Z : REAL; X,Y: REAL;
                              CONST R: INTEGER);
          ...
BEGIN  ...   END ;
BEGIN
     ... Z:=RR(3 , 1 , FF);
         TP (Z,X,Y,R);
          ...
END .
```

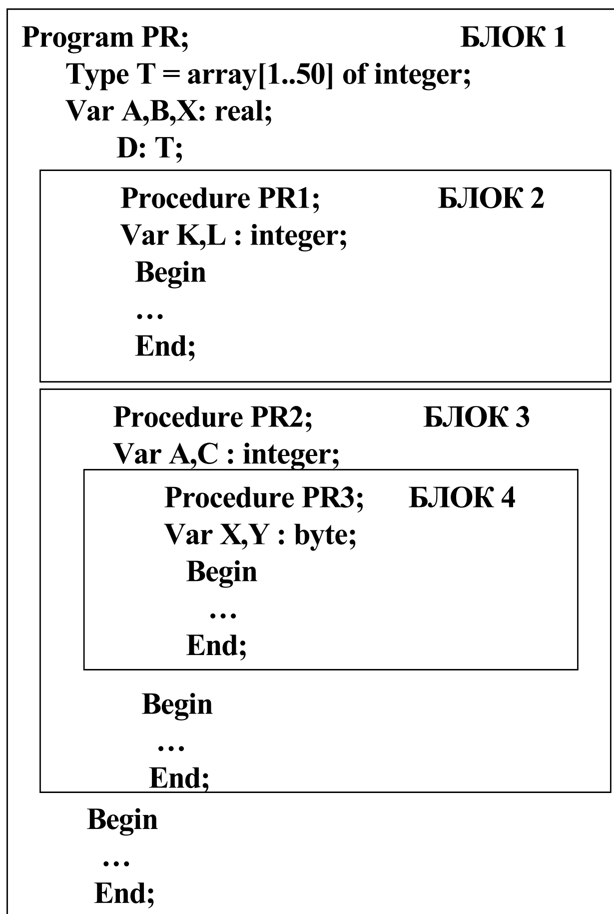
В этом примере используются:

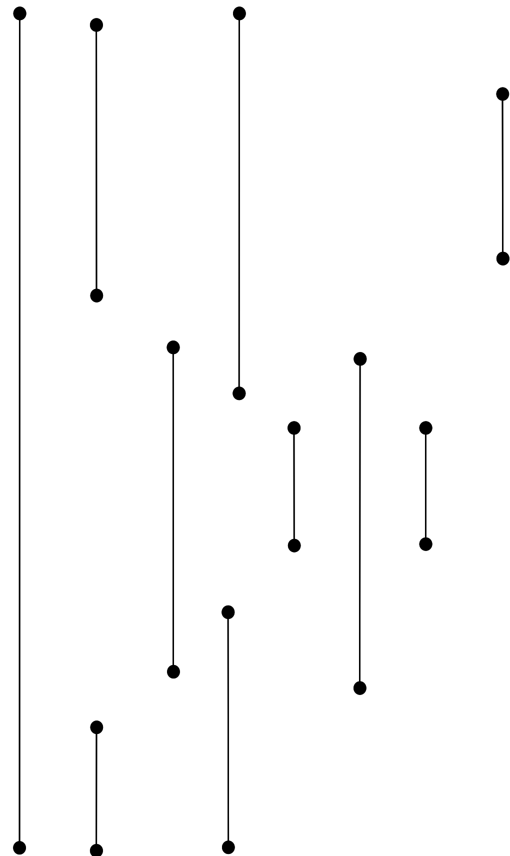
X,Y - параметры-значения;
Z - параметр-переменная;
F- параметр-функция;
R- параметр-константа.

4.5 Область действия имен.

Любая подпрограмма представляет собой блок со своей областью описаний. Она может содержать внутри этого блока описания других процедур и функций, а также обращения к ним. Объекты, описанные внутри какого-либо блока, являются по отношению к нему ***локальными*** и не доступны внешним блокам. На них можно ссылаться только внутри блока, в котором они описаны. Под объектами понимаются имена констант, типов, переменных, процедур, функций. Объекты, описанные во внешних блоках и не описанные во внутренних, являются ***глобальными*** по отношению к внутренним и доступны как во внешних блоках, так и во внутренних. При совпадении имен глобальных и локальных переменных, локальные переменные отменяют действия глобальных в пределах области своего действия.

B,T,D A A’ X X’ C Y K,L





На рисунке схематично представлены области действия отдельных идентификаторов:

Y - локальная переменная по отношению к блоку 4, недоступна в блоках 1,2,3.

K, L - локальная переменная для блока 2, недоступна в блоках 1,3,4.

C - глобальная переменная по отношению к блоку 3, недоступна в блоках 1 и 2.

B, D- глобальные переменные для блоков 2,3,4.Доступны в блоках 1,2,3,4.

T - общий тип.

Идентификатор A обозначает две разные переменные: A - областью действия которой являются блоки 1 и 2, и переменная A'- область действия которой блоки 3 и 4 . То же самое с именем X: одна переменная X - с областью действия 1,2 и 3 блоки и переменная X', которая действует только в блоке 4.

Если подпрограмма имеет параметры, то идентификаторы из списка формальных параметров являются локальными для этой процедуры

(функции) и глобальными для каждой подпрограммы в ней (если таковые имеются).

4.6 Процедуры и функции без параметров.

В случае использования процедур и функций без параметров связь данных осуществляется через *глобальные* переменные.

Подпрограммы должны быть, по возможности, независимы от основной программы, поэтому все переменные , нужные только в пределах подпрограммы, должны описываться как *локальные*. Связь основной программы с процедурой или функцией должна осуществляться, как правило, через список параметров, что придает подпрограммам необходимую гибкость. Вместе с тем, слишком большое число параметров замедляет работу программы, поэтому переменные заведомо общие в основной программе и подпрограммах целесообразно использовать как глобальные.

4.7 Рекурсивные процедуры и функции.

Рекурсия- это способ организации вычислительного процесса, при котором процедура или функция в процессе выполнения входящих в ее состав операторов обращается сама к себе непосредственно, либо через другие процедуры и функции.

Рекурсия может быть ***прямой*** или ***косвенной***. При прямой рекурсии оператор вызова подпрограммы содержится непосредственно в ее исполняемой части. Любую рекурсивную процедуру (функцию) можно сделать не рекурсивной. Рекурсивное описание обычно короче и нагляднее, но требует больших затрат машинного времени (за счет повторных обращений) и памяти машины (за счет дублирования переменных).

Рекурсивная подпрограмма однократно вызывается извне. Условие полного окончания работы рекурсивной процедуры или функции должно находиться в ней самой.

Рассмотрим пример. Вычислить значение **F=M!** Рекурсивное определение значение факториала можно записать следующим образом:

при M=1 F=1
при M> 1 F= M!= M*(M-1)!, т.е. M! определяется через (M-1)!

a) Рекурсивная функция

```
PROGRAM MAIN;
VAR   M: INTEGER;
          F:REAL;
 FUNCTION FACT (N:INTEGER): REAL;
 BEGIN
       IF  N=1 THEN
           FACT:=1
                ELSE
           FACT:= N* FACT(N-1);
 END;
 BEGIN
       READLN(M);
       F:= FACT ( M );
       WRITELN (' M!=', F);
 END.
```

b) Рекурсивная процедура

```
PROGRAM MAIN;
VAR   M: INTEGER;
          F:REAL;
 PROCEDURE FACT(N:INTEGER; VAR F: REAL);
 VAR Q : REAL;
 BEGIN
       IF  N=1 THEN Q:=1
                ELSE  FACT(N-1,Q);
       F:=N*Q;
 END;
 BEGIN
       READLN(M);
       FACT ( M, F );
       WRITELN (' M!=', F);
 END.
```

В варианте а) при м=4 выполняются следующие действия: FACT:=4*FACT(3); FACT:=3*FACT(2); FACT:=2*FACT(1); FACT(1):=1. При входе в функцию в первый раз отводится память под локальную переменную, соответствующую параметру-значению; ей присваивается значение фактического параметра (M). При каждом новом обращении строятся новые локальные переменные. Так как FACT(1)=1, то выполнение функции заканчивается. После этого выполняются действия:

FACT(1):=1; FACT:=2*FACT(1);

FACT(2):=2; FACT:=3*FACT(2);

FACT(3):=3*2=6; FACT:=4*FACT(3); т.е. FACT=24.

4.8 Предварительно-определенные процедуры.

При *косвенной* рекурсии одна подпрограмма вызывает другую, которая либо сама, либо посредством других подпрограмм вызывает исходную. В этом случае используется предварительное описание процедур (функций), так как в языке Турбо Паскаль все процедуры и функции должны быть описаны до их вызова. В этом случае отдельно от текста процедуры (функции) записывается ее заголовок и к нему после символа «;» добавляется ключевое слово **FORWARD**. Сам текст подпрограммы начинается урезанным заголовком, в котором задается только ее имя, а список параметров (и тип результата для функции) отсутствует.

Например:

```
PROGRAM KOSV_R;
      VAR X,Y: INTEGER;
      PROCEDURE LR ( A : INTEGER); FORWARD;
PROCEDURE TT (B: INTEGER);
             ...
 BEGIN
             ...
                 LR (...);
             ...
 END;
PROCEDURE LR ;
                   ...
BEGIN
               ...
            TT(...) ;
               ...
END;
 BEGIN
             ...
            TT(X) ;
            LR(Y)  ;
             ...
END.
```

В этом примере одна процедура вызывает другую, а вторая вызывает первую. При такой схеме взаимодействия процедур невозможно описать обе процедуры так, чтобы обращение к ним следовало после описания, как требует Паскаль. Для разрешения противоречия используется предварительное описание одной из процедур.

4.9 Модули.

Процедуры и функции могут быть сгруппированы в отдельный *модуль*. *Модуль* (**unit**)- это программная единица, текст которой компилируется автономно (независимо от главной программы). Если модуль откомпилирован для реального режима, то результат имеет расширение TPU; модули, откомпилированные для защищенного режима, имеют расширение TPP.

Структура модуля отличается от структуры обычной программы на языке Турбо Паскаль. Модули имеют четыре основные части: заголовок, который следует за зарезервированным словом **UNIT**; описательную (интерфейсную) часть, которая начинается за зарезервированным словом **INTERFACE** (в ней помещаются объявления переменных, процедур, функций, констант и типов данных, которые должны быть доступны для других программных модулей, использующих данный модуль); исполнительную (внутреннюю) часть, которая начинается словом **IMPLEMENTATION** (в нее входит текст подпрограмм и локальные объекты, доступные только внутри данного модуля) и необязательную часть (секцию инициализации), расположенную после исполнительной части между словами **BEGIN** и **END** (при этом, если инициализация модуля не нужна, то в секции помещается лишь слово **END**).

При описании подпрограмм модуля допустимо использовать сокращенные заголовки (без параметров и указания типа результата для функции) как, например, в случае использования директивы FORWARD. Начинается модуль заголовком, состоящим из зарезервированного слова UNIT и имени модуля. Имя модуля обязательно должно совпадать с именем файла (имеющим расширение PAS), в котором он находится. Модуль имеет следующую структуру:

```
UNIT  <имя модуля>;
INTERFACE
USES  <список подключаемых модулей>;
TYPE  <описание типов, определенных в данном модуле
                          и доступных для других модулей>;
CONST  <описание констант, определенных в данном
                    модуле и доступных для других модулей >;
VAR  <описание переменных, определенных в данном
                    модуле и доступных для других модулей >;
PROCEDURE  <заголовки процедур, определенных в данном
                         модуле и доступных для других модулей >;
FUNCTION  <заголовки функций, определенных в данном
                       модуле и доступных для других модулей >;
IMPLEMENTATION
USES  <список подключаемых модулей>;
TYPE  <описание типов, определенных в данном модуле
                          и недоступных для других модулей>;
CONST  <описание констант, определенных в данном
                    модуле и недоступных для других модулей >;
VAR  <описание переменных, определенных в данном
                    модуле и недоступных для других модулей >;
PROCEDURE  <реализация процедур, определенных в
                данном модуле и доступных для других модулей >;
FUNCTION  <реализация функций, определенных в данном
                        модуле и доступных для других модулей >;
PROCEDURE  <заголовки и реализация процедур,
        определенных в данном модуле и недоступных для
                                  других модулей >;
FUNCTION  <заголовки и реализация функций,
        определенных в данном модуле и недоступных для
                                  других модулей >;
BEGIN <это слово необходимо, если имеются операторы
                    секции инициализации>
               <Необязательная часть модуля>
END.
```

Интерфейсная и реализационная части могут быть пустыми, но присутствовать должны обязательно. При подключении модуля вначале выполняются операторы секции инициализации (если они имеются), а затем операторы основного блока главной программы, в которую включен данный модуль.

Рассмотрим пример. Требуется написать главную программу, в которой вводится размер вектора и его элементы и вызывается процеду-

ра сортировки одномерного массива целых чисел в порядке возрастания. Длина массива не превышает 100. Процедуру оформить в виде модуля.

```
USES CRT,MODSORT;
VAR A:MAS;
      I:BYTE;
      N:BYTE;
 BEGIN
      WRITELN('ВВОД ИСХОДНЫХ ДАННЫХ:');
      READLN(N);
      FOR I:=1 TO N DO
            READLN(A[I]);
      SORT(A,N);
      FOR I:=1 TO N DO
            WRITELN(A[I]);
      READKEY
END.
```

Первым предложением программы является **Uses,** в котором подключается стандартный модуль **Crt** и модуль **Modsort**, где находится процедура сортировки.

Кроме того, тип, с которым описывается массив, отсутствует в главной программе, т.к. он присутствует в модуле.

```
UNIT MODSORT;
INTERFACE
TYPE MAS=ARRAY[1..100] OF INTEGER;
PROCEDURE SORT(VAR A:MAS; N:BYTE);
IMPLEMENTATION
PROCEDURE SORT;
VAR I,J:BYTE;
     X:INTEGER;
BEGIN
     FOR J:=1 TO N-1 DO
           FOR I:=1 TO N-J DO
                 IF A[I]>A[I+1] THEN
                 BEGIN
                       X:=A[I]; A[I]:=A[I+1]; A[I+1]:=X
                 END;
END;
END.
```

В интерфейсной части модуля описан тип **mas** и заголовок процедуры сортировки. При подключении этого модуля с помощью предло-

жения **Uses** в любой программе становятся доступными рассматриваемые тип и процедура. Это продемонстрировано в главной программе.

Вопросы к главе 4.

1. Назначение процедур и функций.
2. Возможность подключения процедур и функций с помощью опции компилятора.
3. Описание заголовка процедуры.
4. Описание заголовка функции.
5. Описание процедуры.
6. Как осуществляется вызов процедуры?
7. Особенности описания функции.
8. Особенности вызова функции.
9. Понятие глобальных и локальных переменных.
10. Область действия имен в программах сложной структуры.
11. Особенности использования формальных и фактических параметров.
12. Как осуществляется передача информации в процедурах без параметров?
13. Особенности использования рекурсивных процедур и функций.
14. С какой целью и как описываются предварительно определенные процедуры и функции?
15. Назначение модулей.
16. Особенности описания модулей.

5. Стандартные процедуры и функции.

В языке программирования Турбо Паскаль все используемые процедуры и функции объединены в стандартные модули. К основным модулям можно отнести следующие:

- ***SYSTEM*** – включает стандартные процедуры и функции языка; при использовании этого модуля его не требуется подключать с помощью **USES**, так как он подключен по умолчанию;
- ***DOS*** – содержит процедуры и функции для взаимодействия с MS DOS;
- ***CRT*** - объединяет процедуры и функции для работы с экраном в текстовом режиме и клавиатурой;
- ***GRAPH*** - состоит из процедур и функций для работы с экраном в графическом режиме;
- ***OVERLAY*** - обеспечивает работу с оверлеями (то есть частями программы), используется для обработки программ большого объема при недостатке оперативной памяти;
- ***PRINTER*** – модуль для работы с принтером.

Процедуры и функции модуля ***SYSTEM*** наиболее часто встречаются в программах. Рассмотрим некоторые из них.

5.1 Математические функции.

Имя функции	Назначение функции	Тип результата
Abs(X)	Абсолютное значение (модуль) аргумента Abs(-3.5)=3.5	Совпадает с типом X
ArcTan(X)	Арктангенс аргумента в радианах ArcTan(1)=7.8539816340E-01	Real
Cos(X)	Косинус аргумента в радианах Cos(PI/3)=5.0000000000E-01	Real
Exp(X)	Экспонента аргумента (E в степени X) Exp(1)=2.7182818285E+00	Real
Ln(X)	Натуральный логарифм Ln(10)=2.3025850930E+00	Real
PI	Значение числа π PI=3.1415926536E+00 (более точно 3.1415926535897932385)	Real
Random	Случайное число из диапазона от 0 до 1	Real

Random(X)	Случайное число из диапазона от 0 до X	Word
Sin(X)	Синус аргумента в радианах Sin(PI/3)=8.6602540378E-01	Real
Sqr(X)	Квадрат аргумента Sqr(-12)=144	Совпадает с типом X
Sqrt(X)	Квадратный корень аргумента Sqrt(841)=2.9000000000E+01	Real

При необходимости вычисления некоторых математических функций, для которых не существует стандартных функций в языке Турбо Паскаль, их выражают через имеющиеся стандартные функции. Например:

tg(X)=**Sin(X)/Cos(X)**
lg(X)=**Ln(X)/Ln(10)**
X^n=**Exp(n*Ln(X))**

Перед использованием функций **Random** или **Random(X)** обычно предварительно выполняют процедуру **Randomize** (процедура без параметров), которая обеспечивает несовпадение последовательностей случайных чисел, генерируемых функцией **Random** или **Random(X)**.

5.2 Функции округления и преобразования типов.

Имя функции	Тип аргумента	Тип результата	Назначение функции
Chr(X)	Целый Chr(66)=’B’ Chr(Ord(‘M’)=’M’	Char	Преобразование ASCII-кода в символ (0-255)
Frac(X)	Real Frac(-12.34)=-.34	Real	Дробная часть вещественного числа X-Int(X)
Int(X)	Real Int(-12.34)=-12	Real	Целая часть вещественного числа
High(X)	Порядковый, массив, строка, открытый массив	Совпадает с аргументом	Получение максимального значения номера элемента
Low(X)	Порядковый, массив, строка, открытый массив	Совпадает с аргументом	Получение минимального значения номера элемента

Ord(X)	Порядковый Ord(‘A’)=65 Ord(Chr(86))=86	LongInt	Возвращает число, соответствую-щее символу X в ASCII- таблице
Round(X)	Real Round(-1.2)=-1 Round(-1.5)=-2 Round(1.2)=1 Round(1.5)=2	LongInt	Округление до ближайшего цело-го
Trunc(X)	Real Trunc(-1.2)=-1 Trunc(-1,5)=-1 Trunc(1.2)=1 Trunc(1.5)=1	LongInt	Отбрасывание дробной части

5.3 Функции порядкового типа.

Имя функ-ции	Назначение функции
Odd(X)	Проверяет, является ли аргумент нечетным числом Odd(0)=false Odd(1)=true Odd(2)=false Odd(-1)=true
Pred(X)	Возвращает предшествующее значение аргумента Pred(10)=9 Pred(-10)=-11
Succ(X)	Возвращает последующее значение аргумента Succ(10)=11 Succ(-10)=-9

5.4 Процедуры порядкового типа.

Имя процедуры	Назначение процедуры
Dec(X [,dx])	Уменьшает значение переменной X на величину dx (если параметр dx не задан, то на –1) k:=5; Dec(k)=4 Dec(k,2)=3 Dec(k,-2)=7
Inc(X [,dx])	Увеличивает значение переменной X на величину dx (если параметр dx не задан, то на +1) k:=5; Inc (k)=6 Inc (k,2)=7 Inc (k,-2)=3

5.5 Строковые функции.

Имя функции	Назначение функции
Concat(<строка1>,<строка2>**,..)**	Сцепление строк Concat(‘A’,’BC’,’_1’)=’ABC_1’
Copy(<строка>,<позиция>,<количество>**)**	Копирование части строки с заданной позиции Copy (‘INFORMATION’,3,5) = ’FORMA’
Length(<строка>**)**	Определение текущей длины строки Length('строка’)=6
Pos(<подстрока>,<строка>**)**	Определение позиции первого вхождения подстроки в строку Pos(‘е’,‘Определение’)= 4 Pos(‘к’,‘Определение’)= 0

Для функции Concat общая длина полученной строки не должна превышать 256 байт. Знак «+» для строковых данных также означает операцию конкатенации.

Для функции **Copy**, если позиция в строке превышает длину этой строки, то результатом будет пустая строка. Если <позиция>+ <количество> больше длины строки, то копируются лишь последние символы. Если же <позиция> не принадлежит интервалу [1,255], то возникает ошибка при выполнении программы.

5.6 Строковые процедуры.

Имя процедуры	Назначение процедуры
Delete(<строка>,<позиция>,<количество>**)**	Удаление части строки с заданной позиции 1) S:='abcdefgh'; Delete(S,2,4); Результат: S='afgh' 2) S:='abcdefgh'; Delete(S,2,10); Результат: S='a'
Insert(<подстрока>,<строка>,<позиция>**)**	Вставка подстроки в строку с заданной позиции S:='abcdefgh'; Insert('XXL',S,3); Результат: S='abXXLcdefgh'
Str(<число>,<строка>**)**	Преобразование числового значения в строку символов 1) Str(567,A); Результат: A='567' 2))B:=567; {B:integer} Str(B:5,A); Результат: A='_ _567' 3) B:=5.67E+3; {B:real} Str(B:8:0,A); Результат: A='_ _ _ _5670'
Val (<строка>,<число>,<код>**)**	Преобразование строки в числовое значение (если нет ошибки, то < код>=0) 1)A:='135'; Val(A,R,Code); Результат: R=135; Code=0 2)A:='_135'; Val(A,R,Code); Результат: R=не определено; Code=1 3)A:='2.5E+4'; Val(A,R,Code); Результат: R=25000; Code=0

Для процедуры **Delete**, если <позиция> больше длины строки, то строка не меняется. Если <позиция> + <количество> больше длины строки, то удаляется конец строки с указанной позиции. Если же <позиция> не принадлежит интервалу [1,255], то возникает ошибка при выполнении программы.

Для процедуры **Insert**, если в результате вставки получается строка, длина которой превышает максимальную длину исходной строки, то последние символы вставляемой строки не добавляются. Если <позиция> превышает фактическую длину исходной строки, определяемой функцией **Length**, то результатом является сцепленная строка.

Для процедуры **Val** в строке не разрешаются предшествующие преобразуемому числу и последующие за числом пробелы.

5.7 Прочие процедуры и функции.

Имя функции	Модуль	Назначение процедуры или функции
Keypressed	Crt	Функция. Возвращает значение True, если на клавиатуре была нажата клавиша и False в противном случае
ReadKey	Crt	Функция. Приостанавливает выполнение программы до нажатия на любую клавишу
SizeOf(X**)**	System	Функция. Возвращает число байт, занимаемых аргументом
WhereX	Crt	Функция. Возвращает горизонтальную координату текущей позиции курсора относительно текущего окна
WhereY	Crt	Функция. Возвращает вертикальную координату текущей позиции курсора относительно текущего окна
ClrScr	Crt	Процедура. Очищает экран
Delay (X)	Crt	Процедура. Приостанавливает работу программы на X миллисекунд
Exit	System	Процедура. Преждевременное завершение процедуры, функции или основной программы
FillChar(X,COUNT,Value**)**	System	Процедура. Заполняет заданное количество COUNT последовательных байт переменной X значением Value
GetDate(<год>,<месяц>, <число>,<день недели>**)**	Dos	Процедура. Возвращает текущую дату

GotoXY(X,Y**)**	Crt	Процедура. Перемещает курсор в нужное место экрана
Window(X1,Y1,X2,Y2**)**	Crt	Процедура. Определяет текстовое окно на экране(X1,Y1- координаты верхнего левого угла; X2,Y2- координаты правого нижнего угла)

5.8 Процедуры ввода данных.

Ввод данных в языке Турбо Паскаль выполняется стандартными процедурами (операторами) **READ** или **READLN**, вывод - процедурами **WRITE** или **WRITELN**. Процедуры **READ** и **READLN** используют для ввода символов (тип данных **CHAR**), строк (тип данных **STRING**) или числовых данных (тип данных **INTEGER**, **BYTE**, **REAL** и др.).

Вызов процедуры **READ** имеет следующий вид:

READ ([<имя файла**>,]<**список переменных**>);**

Для процедуры **READLN** соответственно:

READLN ([<имя файла**>,]<**список переменных**>);**

При отсутствии <имени файла> считывание данных производится из стандартного файла **INPUT**; при этом стандартными устройствами считаются клавиатура или дисплей, связанные с файлом INPUT.

Каждому оператору ввода соответствует свой поток данных, в котором перечисляются значения констант, присваиваемые переменным, указанным в списке переменных. Присваивание значений из входного потока происходит слева направо в соответствии с порядком следования переменных в списке переменных.

Необходимо помнить:

- Типы переменных и констант должны совпадать (исключение составляет только тип данных **REAL**, для которых можно при вводе указывать переменные и константы типа **INTEGER**).
- Вводимые *числовые данные* должны быть разделены одним или несколькими пробелами; нельзя отделять пробелами знак от числа и цифру от цифры одного и того же числа.
- Если вводится строка (тип данных **STRING**), то **READ** считывает столько символов, сколько допустимо по максимальной длине, заданной в описании **VAR**.
- При вводе последовательности символов (тип данных **CHAR** или **STRING**) пробел воспринимается как символ.

Например.

А) Ввод числовых данных:

```
VAR B,A,D: REAL;
     K:INTEGER;                Входной поток данных:
       . . .                   2.5  –4.95  20  1.25E2
     READ(A,D);                   После ввода:
     READ(K,B);                  A=2.5; D=-4.95; K=20; B=125
```

Б) Ввод числовых и строковых данных.

```
VAR   A: REAL;
      B:INTEGER;
      C1,C2,C3: CHAR;           Входной поток данных:
      D: STRING[5];            2.5 10 KLMKOШКА
      . . .                          После ввода:
      READ(A,B,C1,C2,C3,D);          A=2.5; B=10; C1=' ';
                                      C2='K';C3='L';
                                      D='MKOШК'
```

Из примера видно, что ввод смешанных данных (и числовых и строковых) из одного входного потока осуществляется не совсем верно. Если же во входном потоке данных после 0 не поставить пробел, то это приводит к ошибке ввода данных (ERROR 106). Поэтому рекомендуется вводить числовые значения отдельно от символов или строк.

При выполнении оператора **READ** конец строки ввода (нажатие клавиши <Enter>) приравнивается к вводу пробела, разделяющего элементы данных во входном потоке; в этом случае нет перехода к следующей строке данных во входном потоке, а входные данные последовательно считываются в соответствующие переменные.

Например, для одних и тех же операторов ввода входной поток может быть разным:

```
READ(A,B,C);        Входной поток:2 9 5 3 7  или
READ(D,E);          Входной поток 1 строка:2 9 5
                                   2 строка 3 7
```

Отличие оператора **READ** от **READLN** состоит в том, что после считывания последней переменной при **READLN** остаток строки ввода игнорируется. Следующий за ним оператор **READ** или **READLN** считывает данные с начала новой строки, т.е. оператор **READLN** реагирует на конец строки (нажатие клавиши <Enter>) и в случае его обнаружения происходит переход к следующей строке данных во входном потоке данных. Возможно использование оператора **READLN** без параметров; в этом случае обеспечивается безусловный переход к следующей строке данных во входном потоке.

Например, рассмотрим ввод элементов двумерного массива разными способами:

1)
```
FOR I:=1 TO 2 DO
  BEGIN
       FOR J:=1 TO 3 DO
            READ (A[I,J]);
       READLN
  END;
```

Входной поток: 3 5 1
-4 7 9

2)
```
FOR I:=1 TO 2 DO
       FOR J:=1 TO3 DO
            READ (A[I,J]);
```

Входной поток: 3 5 1
-4 7 9

3)
```
FOR I:=1 TO 2 DO
       FOR J:=1 TO 3 DO
            READLN(A[I,J]);
```

Входной поток: 3
5
1
-4
7
9

В следующем примере наглядно показано, что при использовании **READLN** данные, находящиеся в конце строки, игнорируются.

```
VAR A,B,C,D:INTEGER;
        …
     READLN(A,B,C);
     READLN(D);
```

Входной поток:
1 строка: 10 20 30 40 50
2 строка: 60

Результат:
A=10;B=20;C=30;D=60

5.9 Процедуры вывода данных.

Процедура (оператор) **WRITE** предназначена для вывода выражений следующих типов: Integer, Byte, Real, Char, String, Boolean и др.

WRITE ([< имя файла или устройства **>,]**
<список выражений**>);**

Если <имя файла> отсутствует, то вывод осуществляется в стандартный файл **OUTPUT** (на экран дисплея). Если указано < имя файла >, этот файл должен быть описан или подготовлен заранее.

Для вывода на печать используется логическое устройство **LST**; при этом должен быть подключен стандартный модуль **PRINTER** (т.е. в начале программы должно быть предложение **Uses Printer**;).

Оператор **WRITE** выводит значения выражений из списка на текущую строку до ее заполнения. Если за ним следует оператор вывода, а текущая строка еще не заполнена, то вывод осуществляется на ту же строку.

Например:

```
X:=5; Y:=10;
WRITE ('X=', X);
WRITE (' Y=',Y);
```

На печать:X=5 Y=10

При выводе на печать для величин стандартного типа отводится определенное число позиций, зависящее от реализации Паскаля. Существует возможность задавать ширину поля (число позиций) для выводимой величины.

Оператор вывода с форматом:

WRITE ([< имя файла или устройства **>,]**$R_1:N_1,R_2:N_2,\ldots, R_m:N_m$**);**

Здесь - $R_1,R_2,\ldots,R_m$- выводимые переменные;

$N_1,N_2,\ldots,N_m$- ширина поля.

Если ширина поля избыточна, то слева помещаются пробелы. Если же ширины поля не хватает для вывода, то заданное значение для ширины поля игнорируется и выводится реальное число.

5.9.1 Особенности вывода вещественных значений.

Если описать переменную вещественного типа, то возможны следующие варианты вывода этой переменной:

1) **Write(R);** Вывод осуществляется в нормализованном виде (экспоненциальная форма):

$$\{\ \ |-\}\ d.dddddddddd\ E\{+|-\}\ dd$$

2) **Write(R:N) ;** Вывод в нормализованном виде с выравниванием по правой границе поля длиной N. Минимальное значение N равно 8. При задании меньшего размера ширины поля компилятор добавляет недостающие разряды до 8.

3) **Write(R:N:M) ;** Вывод в форме с фиксированной точкой и с M десятичными знаками после точки(0<=M<=24).

Например,

```
VAR B,D:REAL;
        C:INTEGER;
        A:STRING[10];
                . . .
       A:='КНИГА';
       B:=1253E-5;
       C:=12;
       D:=1253E2;
WRITE(LST,'B=',B:10:3,' C=',C:8,' A=',A:7,' B1=',B:8,' D=',D:6);
```

На печать будет выведено (здесь _ означает символ пробел, который на экране не виден):
B=_ _ _ _ _ 0.013_C=_ _ _ _ _ _12_A=_ _книга_B1=_1.3E-02_D=_1.3E+05

Процедура **WRITELN** имеет аналогичный вид:

WRITELN ([<имя файла или устройства**>,]<**список выражений**>);**

При вызове этой процедуры завершается формирование текущей строки файла. Следующий оператор **WRITE** или **WRITELN** формирует новую строку. Можно использовать **WRITELN** без параметров.

Например, при совместном использовании операторов **WRITE** и **WRITELN** можно регулировать вывод по строкам:

```
VAR A,B,C,D,E:INTEGER;
BEGIN
     A:=1;  B:=2;  C:=3;  D:=4;  E:=5;
     WRITELN (' A=',A,' B=',B);
     WRITE(' C=',C);
     WRITELN(' D=',D,' E=',E);
END.
```

На экран дисплея результат будет выведен в двух строках:
_A=1_B=2
_C=3_D=4_E=5

Вывод матрицы A (M,N) целых чисел на экран в виде прямоугольной таблицы можно реализовать следующими операторами:

```
          . . .
FOR I:=1 TO M DO
BEGIN
    FOR J:=1 TO N DO
       WRITE(A[I,J]:5);
     WRITELN
END;
```

Вывод матрицы A(M,N) вещественных чисел на принтер в виде таблицы с одним разрядом после запятой представлен следующей программой:

```
USES PRINTER;
VAR A:ARRAY[1..10,1..10]OF REAL;
     M,N:INTEGER;
BEGIN
     READLN(M,N);
      FOR I:=1 TO M DO
           FOR J:=1 TO N DO
                READ(A[I,J]);
     FOR I:=1 TO M DO
     BEGIN
          FOR J:=1 TO N DO
             WRITE(LST,A[I,J]:6:1);
          WRITELN(LST)
     END;
     READKEY
END.
```

Вопросы к главе 5.

1. Общая классификация стандартных процедур и функций.
2. Назначение основных стандартных модулей.
3. Особенности математических функций.
4. Особенности использования процедур для работы со строковыми данными.
5. Особенности использования функций для работы со строковыми данными.
6. Особенности использования экранно-ориентированных процедур.
7. Основные особенности процедур ввода данных.
8. Основные особенности процедур вывода данных.
9. Особенности вывода вещественных значений .
10. Особенности ввода одномерных и двумерных массивов.
11. Особенности вывода одномерных и двумерных массивов

6. Работа с файлами.

6.1 Общие сведения о файлах.

При обработке на компьютере информация может храниться на внешних носителях в виде файлов. ***Файл*** на носителе – это поименованная совокупность логически связанных между собой данных (записей), имеющая определенную организацию и общее назначение.

Физическая запись – это совокупность данных, передаваемых в том или обратном направлении при одном обращении к внешнему носителю (т.е. минимальная единица обмена данными между внешней и оперативной памятью). Физическая запись состоит из логических записей.

Логическая запись – единица данных, используемая в операторах чтения и записи файлов. Логические записи объединяются в физическую запись для уменьшения числа обращений к внешнему устройству.

Для обращения к записям файла на внешнем носителе используется понятие логического файла. Логический файл или файл в программе – это совокупность данных, состоящая из логических записей, объединенных общим назначением.

Для связи файла в программе и файла на внешнем носителе используется процедура **ASSIGN**, где указывается имя файла в программе и имя файла на внешнем носителе.

Число записей файла произвольно, но в каждый момент времени доступна только одна запись. Длиной файла называют количество записанных компонент. Файл, не содержащий записей, называется пустым.

Каждая переменная файлового типа должна быть описана в разделе описания переменных **VAR .** Не допускается использование таких переменных в выражениях и операторах присваивания. Тип компонент файла может быть любым кроме файлового.

В Турбо Паскале предварительно определен следующий стандартный тип:

```
TYPE TEXT = FILE OF CHAR;
```

В системе программирования Паскаль различаются 3 вида файлов:

- файлы с типом записей (типизированные файлы);
- текстовые файлы со строками неопределенной длины;
- файлы без типа для передачи данных блоками записей.

При работе с файлами необходимо придерживаться следующих общих правил:

- все имена файлов могут быть указаны в заголовке программы;
- текстовые файлы должны быть описаны с типом **TEXT;**
- каждый файл в программе должен быть закреплен за конкретным файлом на носителе процедурой **ASSIGN;**

- открытие существующего файла для чтения, корректировки или дозаписи производится процедурой **RESET,** открытие создаваемого файла для записи – процедурой **REWRITE**;
- по окончании работы с файлом он должен быть закрыт процедурой **CLOSE**.

6.2 Процедуры и функции для работы с файлами.

ASSIGN (<имя файла>**,**<имя файла на носителе>**) –** процедура устанавливает связь между именем файловой переменной и именем файла на носителе. Здесь <имя файла> это файловая переменная, т.е. правильный идентификатор, объявленный в программе как переменная файлового типа. <Имя файла на носителе> – текстовое выражение, содержащее имя файла или имя логического устройства. Перед именем файла на носителе может ставиться путь к файлу – имя диска и(или) имя текущего каталога и имена каталогов вышестоящих уровней.

RESET(<имя файла>**) –** процедура открытия существующего файла для чтения при последовательном доступе и для чтения и записи при прямом доступе. Указатель файла при этом устанавливается на первую запись (с 0 номером).

REWRITE(<имя файла>**) –** процедура открытия создаваемого файла для записи. Если файл с таким именем уже существовал, то он стирается. Указатель файла устанавливается на первую запись.

READ(<имя файла>, <переменные>) – процедура чтения очередных компонент файла в переменные, тип которых должен совпадать с типом компонент файла. Указатель файла при этом передвигается на количество прочитанных компонент.

WRITE(<имя файла>, <переменные>) – процедура записи содержимого переменных в файл согласно положению указателя. Указатель автоматически сдвигается на число записанных компонент**.**

SEEK(<имя файла>**,** <номер компоненты>**) –** процедура установки текущего указателя для чтения или записи требуемой компоненты файла. Используется для организации прямого доступа к записям файла.

CLOSE(<имя файла>**) –** процедура закрытия файла. Обязательно должна использоваться после создания файла, иначе может произойти потеря данных.

ERASE(<имя файла>**)** – процедура уничтожения файла. Открытый файл прежде должен быть закрыт.

RENAME(<старое имя файла>**,** <новое имя файла>**) –** процедура для переименования файла. Используется после закрытия файла.

IORESULT – функция возврата условного признака последней операции ввода-вывода. Если операция завершилась успешно, функция возвращает нуль. Функция становится доступной только при отключенном автоконтроле ошибок ввода-вывода. Директива компилятора {$I-} отключает, а {$I+} – включает автоконтроль ошибок. Если автоконтроль

отключен и операция ввода-вывода привела к возникновению ошибки, устанавливается флаг ошибки и все последующие обращения к вводу-выводу блокируются, пока не будет вызвана функция **IORESULT.**

FILEPOS(<имя файла>**)** – функция определения номера текущей записи файла.

FILESIZE(<имя файла>**)** – функция определения общего количества записей файла.

EOF(<имя файла>**)** – функция определения признака конца файла. Получает значение **TRUE** при чтении последней записи файла.

EOLN(<имя файла>**)** – функция обнаружения конца строки в текстовом файле. Имеет значение **TRUE,** если найден конец строки.

6.3 Особенности обработки типизированных файлов.

Файл с типом (типизированный файл) состоит из последовательности записей одинаковой длины и одинакового внутреннего формата. Записи следуют непрерывно друг за другом. Первые 4 байта первого сектора файла содержат количество и длину записи. К файлам с такой организацией можно обращаться последовательно и выборочно (с прямым доступом).

При последовательном доступе записи располагаются на внешнем носителе последовательно в порядке их поступления, т.е. чтение или запись I+1 компоненты возможно только после I –ой компоненты.

При прямом доступе предполагается, что данные располагаются в определенных областях, имеющих последовательные номера, начиная с нуля. Вычисляя значение указателя, фиксирующего номер записи, можно обеспечить прямой доступ к нужной записи, используя процедуру позиционирования **SEEK**.

Общий вид описания типизированного файла:

TYPE < идентификатор типа **>= FILE OF <** тип компонент **>;**

Например,

```
1) TYPE  T = FILE  OF REAL;
     VAR   F: T;

2)   VAR  F: FILE  OF REAL;

3) TYPE ST= RECORD
               A: STRING[10];
                B: INTEGER;
               C: REAL;
               D: BYTE
            END;
     VAR  DAN: FILE OF ST;
```

В первом варианте тип файла описан в разделе описания типов, а затем в разделе описания переменных файловая переменная получает этот тип, во втором варианте тип предварительно не описывается. В третьем варианте предварительно описывается тип записи файла, а в разделе описания переменных этот тип используется для указания типа отдельной записи.

Процедуры чтения и записи для файлов с типом **READ** и **WRITE**. Кроме того, используются процедуры и функции **ASSIGN, RESET, REWRITE, SEEK, CLOSE, FILEPOS, FILESIZE, EOF**. Процедура **TRUNCATE** обрезает файл на заданной файловой позиции.

Пусть требуется создать файл из записей, данные которых вводятся с клавиатуры. После создания файла содержимое файла вывести на экран.

Структура записи файла следующая:

- фамилия;
- табельный номер;
- заработная плата.

```
TYPE TZ=RECORD
           FIO:STRING[10];
           TN:INTEGER;
           ZP:REAL
        END;
VAR  ZAP:TZ;
      FOUT:FILE OF TZ;
      FL:BOOLEAN;
      NAME:STRING;
BEGIN
      REPEAT
         WRITELN('ИМЯ ФАЙЛА ');
         READLN(NAME);
         ASSIGN (FOUT,NAME);
       {$I-} RESET(FOUT); {$I+}
         IF IORESULT=0 THEN
         BEGIN
               WRITELN('ФАЙЛ ',NAME,' УЖЕ ЕСТЬ');
               CLOSE(FOUT);
               FL:=FALSE
         END
                        ELSE
         BEGIN
               REWRITE(FOUT);
               FL:=TRUE
         END
```

```
        UNTIL FL;
        WITH ZAP DO
        REPEAT
           WRITELN('ВВОД  FIO,TN,ZP');
           READLN(INPUT,FIO,TN,ZP);
           WRITE(FOUT,ZAP);
        UNTIL EOF(INPUT);
        CLOSE(FOUT);
        RESET(FOUT);
         WITH ZAP DO
         REPEAT
             READ(FOUT,ZAP);
             WRITELN(FIO:15,TN:9,ZP:8:2);
        UNTIL EOF(FOUT);
        CLOSE(FOUT)
   END.
```

В начале программы выполняется ввод имени файла до тех пор, пока не будет введено имя несуществующего файла, т.к. в противном случае старый файл будет уничтожен и данные будут утеряны. После ввода нового имени флаг **FL** становится равным **TRUE** и цикл ввода имени файла прекращается. После этого начинается непосредственно цикл ввода данных с клавиатуры из файла **INPUT** . Признак конца ввода **Ctrl+Z**. Стандартное имя файла ввода с клавиатуры **INPUT** можно опустить и в операторе чтения из файла **READLN**, и в функции проверки конца ввода **EOF**. После создания файла и ввода всех данных файл закрывается процедурой **CLOSE.** Затем созданный файл открывается для чтения, данные из него читаются и выводятся на экран в виде таблицы.

Файл может расширяться путем включения последующих элементов за последним существующим элементом файла. Для этого используется процедура позиционирования:

SEEK(<имя файла**>, FILESIZE(<**имя файла**>)).**

В следующей программе, используя прямой доступ к записям файла, созданного в предыдущей программе, требуется подсчитать сумму зарплаты рабочих, чьи табельные номера вводятся с клавиатуры (предполагается, что при создании файла были использованы табельные номера в интервале 101-999 и запись с номером 101 занимает первое место в файле, за ней следует запись с табельным номером 102 и т.д.).

```
TYPE TZ=RECORD
         FIO:STRING[10];
         TN:INTEGER;
         ZP:REAL
      END;
VAR  ZAP:TZ;
     FOUT:FILE OF TZ;
     TN1,TN2,N:INTEGER;
     S:REAL;
     NAME:STRING;
BEGIN
     WRITELN('ИМЯ ФАЙЛА ');
     READLN(NAME);
      ASSIGN (FOUT,NAME);
      RESET(FOUT);
     S:=0;
     REPEAT
        READLN(TN2);
        TN1:=TN2-101;    {формирование указателя записи}
         SEEK(FOUT,TN1);
         READ(FOUT,ZAP);
         S:=S+ZAP.ZP;
      UNTIL EOF;
      WRITELN('S= ',S);
      CLOSE(FOUT)
END.
```

В этой программе процесс обработки заканчивается, когда прекращается ввод табельных номеров, подлежащих обработке, т.е. Ctrl+Z для файла ввода с клавиатуры.

6.4 Особенности обработки текстовых файлов.

Текстовые файлы состоят из символов. Каждый текстовый файл разделяется на строки неопределенной длины, которые заканчиваются символом конец строки. Весь файл заканчивается символом конец файла.

К текстовым файлам возможен только последовательный доступ. С текстовыми файлами работают различные редакторы текстов. Текстовые файлы имеют стандартный тип **TEXT**.

```
VAR < имя файла>: TEXT;
```

Посимвольные операции ввода-вывода выполняются для текстовых файлов процедурами **READ** и **WRITE.** Строки обрабатываются специальными процедурами для текстовых файлов **READLN** и **WRITELN.** Кроме того, для текстовых файлов применяются процедуры **ASSIGN, RESET, REWRITE, CLOSE, EOF, EOLN.** Процедура **APPEND** открывает существующий текстовый файл для добавления записей. Для текстовых файлов нельзя использовать процедуры и функции **SEEK, FILEPOS, FILESIZE,** т.к. элементы имеют разную длину.

INPUT и **OUTPUT** стандартные текстовые файлы для ввода с клавиатуры и вывода на экран.

Рассмотрим программу, реализующую следующую задачу. Дан текстовый файл F1.Необходимо заменить во всех записях этого файла код группы ДС-101 на ДС-201. Скорректированные записи поместить в файл F2.

```
VAR F1,F2: TEXT;
        POLE:STRING;
        NAME:STRING[12];
        PZ: INTEGER;
BEGIN   WRITE('ВВОД ИМЕНИ ВХОДНОГО ФАЙЛА:');
         READLN(NAME);
          ASSIGN(F1,NAME);
         WRITE('ВВОД ИМЕНИ ВЫХОДНОГО ФАЙЛА:');
          READLN(NAME);
          ASSIGN(F2,NAME);
          RESET(F1);  REWRITE(F2);
          WHILE NOT EOF(F1) DO
         BEGIN
                READLN(F1,POLE);
                WHILE POS('ДС-101', POLE) <> 0 DO
               BEGIN
                     PZ:= POS('ДС-101', POLE);
                     DELETE(POLE,PZ+3,1);
                     INSERT('2',POLE,PZ+3);
                 END;
              WRITELN(F2,POLE)
         END;
         CLOSE(F1);
         CLOSE(F2);
END.
```

Здесь читаются последовательно строки из входного файла и в каждой строке в номере группы заменяется символ 1 на 2. Скорректированные строки выводятся в новый файл.

6.5 Файлы без типа.

Любой файл может быть представлен в виде последовательности символов кода ASCII. Турбо Паскаль позволяет рассматривать файл с любой организацией как бы состоящим из блоков по 128 байт.

Файлы без типа используются обычно при копировании файлов, когда не важна внутренняя структура записи файла. Если длина сегмента на диске 1024 байта, то количество блоков в группе равно 8 при длине блока 128 символов.

Обмен информацией происходит непосредственно между программой и файлом без использования буферной памяти. Адресация блоков производится по их номерам. Блоки в этом случае являются компонентами файла. Использование файлов без типа приводит к экономии памяти.

Для работы с такими файлами предусмотрены специальные процедуры, позволяющие производить обмен группами блоков по 128 символов.

BLOCKREAD(<имя файла>,<переменная>,<число компонент>
[,<факт.число>**]);**

- для чтения блока из файла.

BLOCKWRITE(<имя файла>,<переменная>,<число компонент>
[,<факт.число>**]);**

- для записи блока в файл.

Здесь:

<имя файла> - имя файла без типа;

<переменная>- имя переменной для чтения или записи;

<число компонент> **-** количество передаваемых компонент за один раз;

<фактическое число> **-** количество фактически переданных записей длиной 128 байт.

Файл для блочного ввода-вывода описывается с типом **FILE**. Для файла без типа нельзя использовать процедуры **READ** и **WRITE**.

VAR < имя файла **> : FILE;**

При открытии файла без типа можно указать длину записи файла в байтах. Она указывается вторым параметром при обращении к процедуре **RESET** или **REWRITE,** в качестве которого используется выражение типа **WORD.** Если длина записи не указана, она принимается равной 128 байтам.

Рассмотрим пример блочного ввода-вывода. Пусть требуется скопировать данные из файла **FROMF** в файл **TOF**.

```
VAR
      FROMF, TOF: FILE;
      NR, NWR: WORD;
      NAME:STRING[12];
      BUF: ARRAY[1..2048] OF CHAR;
BEGIN
      WRITE('ИМЯ ВХ.ФАЙЛА');
      READLN(NAME);
      ASSIGN(FROMF, NAME);
      WRITE('ИМЯ ВЫХ.ФАЙЛА ');
      READLN(NAME);
      ASSIGN(TOF,NAME);
      RESET(FROMF, 1);
      REWRITE(TOF, 1);
      REPEAT
           BLOCKREAD(FROMF, BUF, SIZEOF(BUF), NR);
           BLOCKWRITE(TOF, BUF, NR, NWR);
      UNTIL (NR = 0) OR (NWR <> NR);
      CLOSE(FROMF);
      CLOSE(TOF);
 END.
```

В примере программы при открытии файла без типа в вызове процедуры **RESET** указана длина записи равная 1. В этом случае при копировании файла не будут записаны лишние символы в новый файл.

6.6 Проектирование программ по структурам данных

Проектирование программ по структурам данных можно считать одним из самых зрелых и продвинутых направлений в индустриальном отношении.

Использование диаграмм Варнье, Джексона и МЭСИД для проектирования программ уже упоминалось ранее, а здесь остановимся несколько подробнее на использовании подхода МЭСИД для задач, относимых обычно к классу экономических.

Все эти подходы основываются на создании диаграмм структур выходных и входных данных, определении несоответствий в них и путей устранения этих несоответствий через диаграмму структуры программы, составление списков операций и написание псевдокода (текста) программы.

Одним из принципов в этих методологиях является упреждающее чтение записей. В языках программирования, в которых при N записях

в последовательном файле об окончании записей становится известно при N + 1 чтении, такое программирование является более удобным. В этом случае чтение первой записи выполняется по началу программы, а чтение остальных записей осуществляется в блоках, где производится их обработка. Для языков, в которых признак окончания файла устанавливается для последней прочитанной записи, то есть требуется N чтений при N записях, упомянутое неудобство устраняется чтением первой записи по началу программы и использованием дополнительно условного оператора для установки флага завершения файла в блоках, где производится их обработка. Другим способом является написание специальной подпрограммы, к которой производится N+1 обращение с получением флага завершенности файла и записи в целом или только необходимых полей. При использовании ряда языков четвертого поколения (4 GL) эта проблема решается автоматически.

Одним из вариантов в подходе МЭСИД является следующая последовательность шагов с преобладающим направлением от выхода к входу и от физического к логическому:

1. Составление диаграммы структуры выходных данных.
2. Составление диаграммы структуры входных данных.
3. Составление диаграммы структуры программы с матрицей операций.
4. Заполнение матрицы операций.
5. Составление текста программы на языке программирования.

В основе составления спецификации помимо графических средств лежит активный контроль речевой деятельности, направленный на выделение сходных подмножеств и их синхронизацию и/или отношений 1:1, 1:M, M:N.

Пусть имеется условный макет отчета (сводки) о начислении заработной платы рабочим-сдельщикам (рис. 1), в котором для каждого табельного номера (рабочего, исполнителя) выводится одна строка. Кроме итогов по табельным номерам выводятся итоги для участков, цехов и общий итог. Заголовок и «шапка» выводятся один раз в начале отчета. Обрабатываются все записи входного набора данных с последовательной организацией, упорядоченного по шифру цеха, шифру участка и табельному номеру. Кроме упомянутых элементов каждая запись включает шифр детали, номер операции, единицу измерения, нормированное время на единицу, расценку и количество.

Сводка о начислении заработной платы.

Шифр цеха	Шифр участка	Табельный номер	Начислено
______	______	______	______
______	______	______	______
______	______	______	______
			По участку
______	______	______	______
______	______	______	______
			По участку
			По цеху
______	______	______	______
______	______	______	______
______	______	______	______
			По участку
			По цеху
			Всего

Рис.1 Макет выходного документа.

Сначала произведем описание структуры выходных данных. Глядя на макет отчета, можно видеть, что самая частая строка отчета относится к табельному номеру (рабочему, исполнителю). Полями (элементами) всех этих строк являются шифр цеха, шифр участка, табельный номер и начислено. Низшим уровнем в иерархии является рабочий, а шифр цеха и шифр участка повторяются для разных рабочих в пределах участков и цехов. Таким образом, можно выделить (рис.2а) конструкцию повторения «табельный номер», в тело которой входят элементы шифр цеха, шифр участка, табельный номер и начислено.

Зададим вопросы: «Что предшествует этой конструкции, что следует за ней, во что она входит?». Отвечая на эти вопросы, можно получить диаграмму (рис. 2б), затем диаграмму (рис.2в) и, наконец, диаграмму (рис.2г) которую можно прочитать примерно следующим образом:

1. Печать отчета начинается с вывода заголовка и шапки.
2. Производится вывод информации по цехам, по участкам, по табельным номерам.
3. При завершении участка выводится итог по участку.
4. При завершении цеха выводится итог по цеху.
5. При завершении файла (всей информации) выводится общий итог.

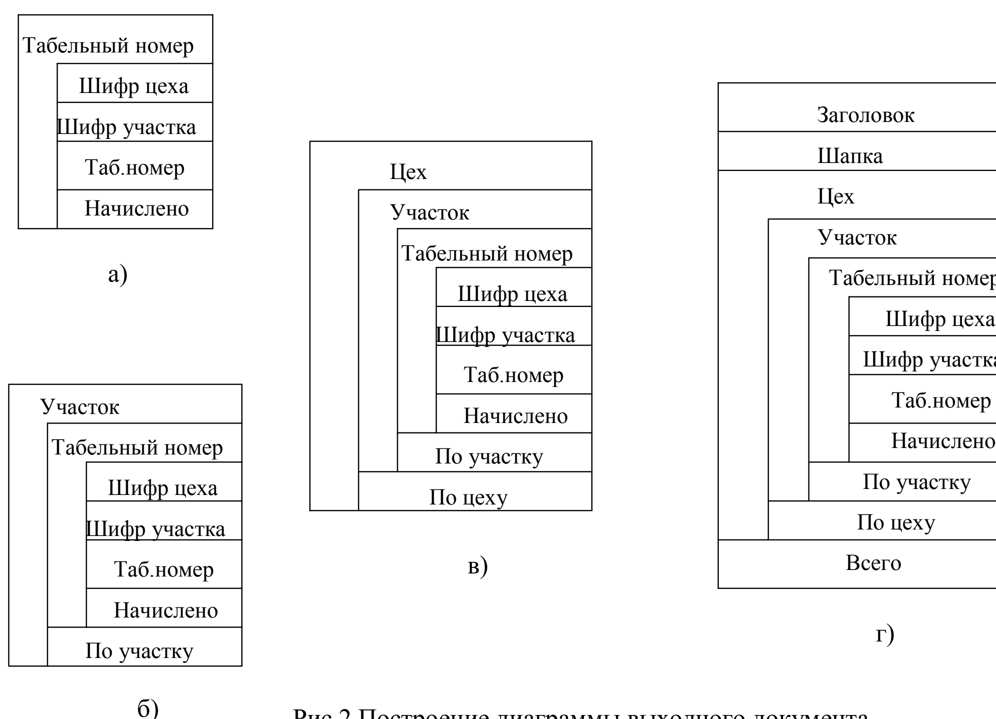


Рис.2 Построение диаграммы выходного документа.

Построение диаграммы выходных данных настраивает на более внимательный анализ структуры входных данных. Возвращаясь к описанию входных данных, можно отметить, что упорядоченность последовательного файла по шифру цеха, шифру участка и табельному номеру соответствует порядку выделения подмножеств выходных данных. Отметим, однако, что упорядоченность по табельному номеру служит не только, и не столько, удобству чтения информации отчета, а формированию итоговой строки отчета для табельного номера, то есть одному полю «начислено» может соответствовать несколько записей входного файла об операциях, выполненных над определенными деталями. Кроме того, запись входного файла может содержать разнообразные поля, которые могут не иметь непосредственного отношения к формируемому отчету. В рассматриваемой задаче можно предположить, что для вычисления «начислено» будут использоваться значения «расценки» и «количества» *(в случае неуверенности необходимо согласовать с постановщиком задачи*). Таким образом, диаграмму входных данных (рис. 3) можно прочитать следующим образом:

1. Файл состоит из подмножеств записей, соответствующих цехам.
2. Эти подмножества состоят из подмножеств записей, соответствующих участкам.
3. Эти подмножества состоят из подмножеств записей, соответствующих табельным номерам.
4. Эти подмножества состоят из подмножеств записей, соответствующих обработанным деталям.

5. Каждая такая запись содержит информацию об обработанной детали, включая данные, необходимые для вычислений и идентификации подмножеств.

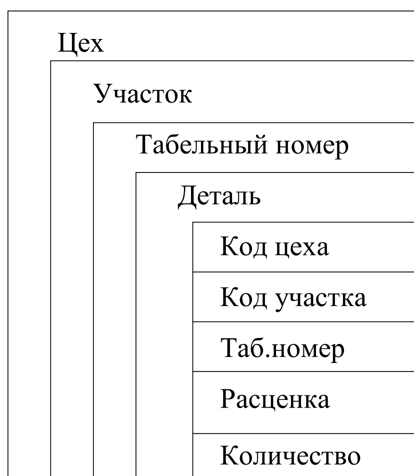


Рис.3 Диаграмма входных данных

Сравнив иерархии в диаграммах выходных и входных данных, мы не видим существенных противоречий (например, различий в упорядоченности), что позволяет перейти к составлению диаграммы структуры программы. За основу этой структуры берется структура входных данных, каждое повторение и развилка которой наращиваются соответствующими началами и окончаниями (рис. 4). Диаграмма структуры программы является достаточно компактной, и ее можно дополнить матрицей операций, определяющей порядок действий программы на уровне псевдокода. Столбцы матрицы соответствуют категориям операций:

1. Чтение с прямым доступом.
2. Подготовка переходов.
3. Подготовка вычислений и вычисления.
4. Подготовка вывода, вывод и очистка.
5. Чтение с последовательным доступом.
6. Переходы.

Категория «Чтение с прямым доступом» предназначена главным образом для извлечения данных из файлов прямого доступа для расширения состава полей записей последовательного файла. В нашем примере при необходимости работы с наименованиями цеха и участка после последовательного чтения основного файла мы могли бы задать чтение соответствующих справочников по началу цеха и по началу участка.

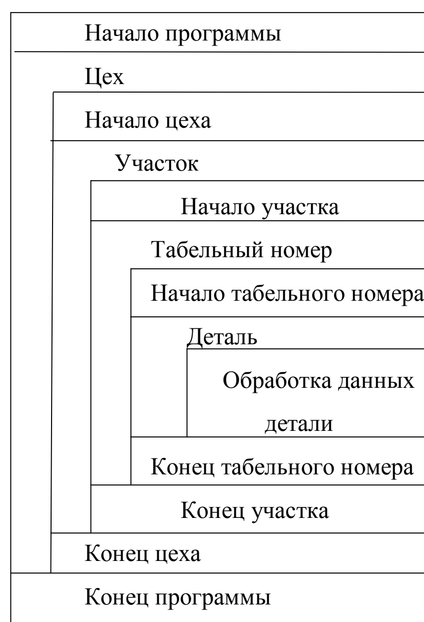


Рис.4 Структура программы

Категория «Подготовка переходов» связана в значительной степени с определением однородных в определенном смысле подмножеств записей входного файла. Так, например, печать итога по участку должна производиться, когда начинается другой участок или когда заканчивается файл. С этой целью определяется значение флажка окончания файла и фиксируется код участка.

Категория «Подготовка вычислений и вычисления» связана, в основном, с обнулением счетчиков и их наращиваем.

Категория «Подготовка вывода, вывод и очистка» связана главным образом с выводом значений идентификации подмножества по его началу или/и окончанию. Так, при детальной печати распечатывается каждая запись входного файла, а при итоговой печати выводятся итоги для подмножеств и идентификаторы. В нашем примере на нижнем уровне выводится строка для табельного номера, когда это подмножество завершено и уже известны идентификаторы для следующего подмножества. По это причине необходимо запоминать эти идентификаторы на соответствующем уровне для последующего вывода. Этот процесс похож на подготовку переходов, и в ряде случаев поля подготовки переходов можно использовать и для вывода.

Существует еще вид отчетов с индикацией группы, когда идентификаторы подмножеств и сопутствующие элементы выводятся в более детальной строке только при смене подмножества, а после этого в детальной строке остаются пробелы. В этом случае в поле(я) вывода по началу подмножества помещаются его идентификаторы, а после первого же вывода поля вывода заполняются пробелами.

Категория «Чтение с последовательным доступом» обычно связана с основным файлом, исчерпание которого приводит к завершению программы.

Категория «Переходы» в явном виде, в отличие от метода Варнье, может не использоваться, поскольку в диаграмме структуры программы имеются строки для соответствующих операторов управления.

Возникает, однако, вопрос, как связать эти диаграммы структур данных и программы на уровне полей. Допишем с этой целью на диаграммах достаточно короткие наименования полей записей и дополнительных переменных (рис.5).

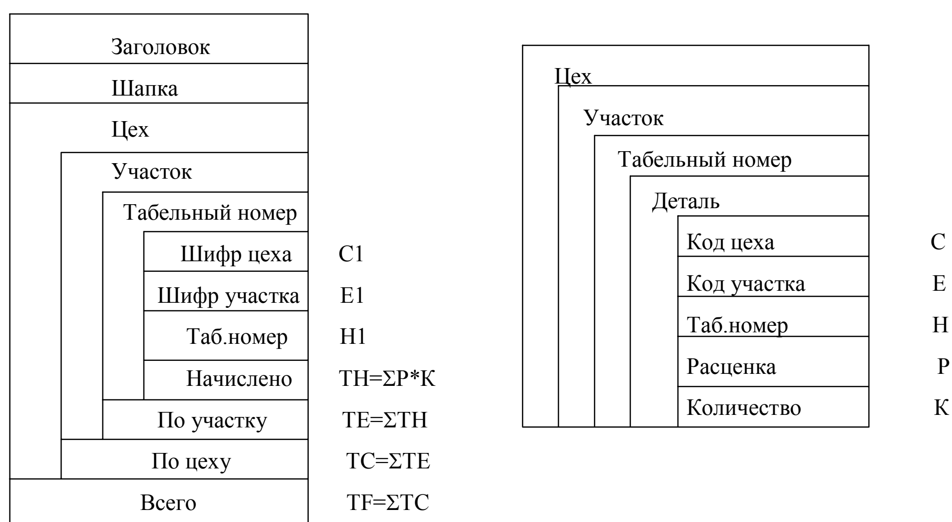


Рис.5 Диаграммы выходных и входных данных с разметкой идентификаторов полей и вычислительных формул.

Запись операций в матрице (рис.6) должна производиться с учетом различия частот выполнения блоков программы. Так, задавая вопросы, как часто и когда должно выполняться то или иное действие, можно определить его место в матрице. Например, операция ввода должна выполняться для каждой детали и еще один раз по началу программы, операция вывода заголовка и шапки должна выполняться один раз по началу программы и т.д.

Структура программы	Подготовка переходов	Чтение с прямым доступом	Вычисления	Вывод	Чтение с послед. доступом
Начало программы	F=true		TF = 0	заголовок, шапка	читать С,Е,Н,Р,К
Цех	пока F				
Начало цеха	С1=С		ТС =0		
Участок	пока F и C1 = C				
Начало участка	Е1=Е		ТЕ = 0		
Табельный номер	пока F и C1 = C и E1 = E				
Начало т.н.	Н1=Н		ТН= 0		
Деталь	пока F и C1 = C и E1 = E и H1 = H				
Обработка детали			ТН=ТН+Р*К		читать С,Е,Н,Р,К
Конец т.ном.			ТЕ=ТЕ+ТН	С1,Е1,Н1,ТН	
Конец участка			ТС=ТС+ТЕ	ТЕ	
Конец цеха			TF=TF+ТС	ТС	
Конец программы				TF	

Примечание: при конце файла F = false

Рис.6 Структура программы с матрицей операций.

6.7 Работа с файлами при обработке экономической информации

6.7.1 Постановка задачи.

Пусть для системы обработки данных требуется написать программу одновременного формирования трех ведомостей о начислении заработной платы для рабочих-сдельщиков на основе пооперационных данных, находящихся в файле «Наряд», и постоянных справочных файлах «Рабочий», «Подразделение» и «Операция».

На рис.7 показан фрагмент диаграммы потоков данных, из которого видно, что требуется получить три ведомости, названные здесь 1, 2 и 3. При этом входными данными для обработки являются четыре файла, а выходными данными – три ведомости.

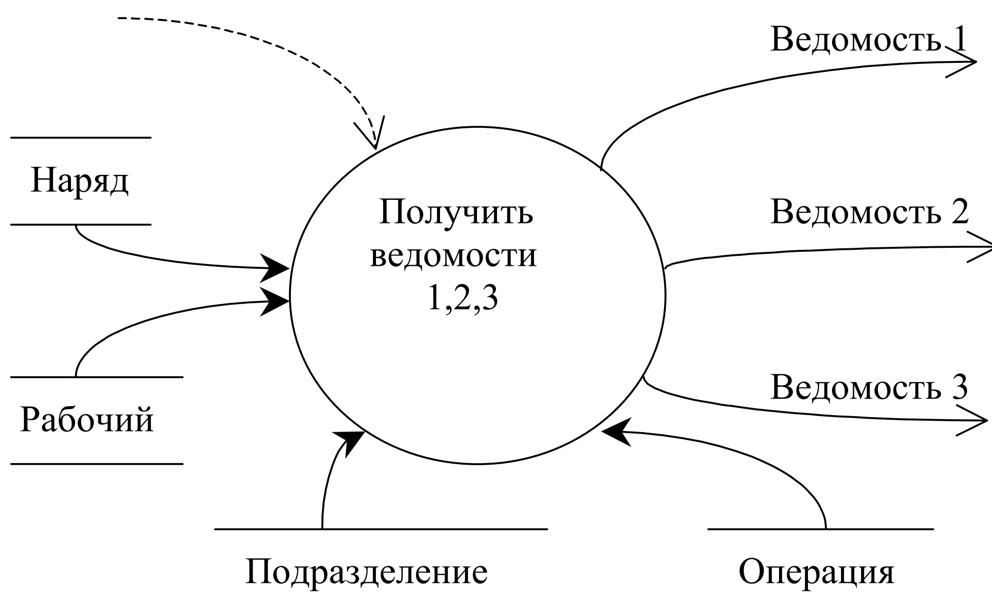


Рис.7 Фрагмент диаграммы потоков данных.

На рис.8 представлены макеты выходных документов. Здесь видно, какие заголовки и показатели выводятся в каждой ведомости, какие итоги должны подводиться.

Рассмотрим структуры записей входных файлов. На рис. 9 приводятся структуры записей каждого входного файла. Кроме рассмотренных полей каждый файл может содержать и другие поля, которые не используются при решении данной задачи. Для обозначения наличия таких полей в структуре каждого файла имеется одно поле, не используемое для формирования ведомостей. При описании структуры записей в программе, естественно, необходимо описывать полную

Ведомость 1

Цех	Участок	Таб.номер	Начислено
01	01	0001	130
01	01	0003	40
		по участку	170
01	02	0002	10
01	02	0004	10
		по участку	20
		по цеху	190
04	02	0005	10
04	02	0007	10
		по участку	20
		по цеху	20
		всего	210

Ведомость 2

Цех 01 Механический

Участок	Т.ном.	ФИО	Начислено
01	0001	Иванов	130
	0003	Петров	40
		по участку	170
02	0002	Белов	10
	0004	Сидоров	10
		по участку	20
		по цеху	190

Цех 04 Прессо-кузовной

Участок	Т.ном.	ФИО	Начислено
02	0005	Буров	10
	0007	Кузин	10
		по участку	20
		по цеху	20
		всего	210

Ведомость 3

Цех	Название	Начислено
01	Механический	190
04	Прессо-кузовной	20

Рис.8 Макеты отчетов.

Структура записи файла «Наряд»

Название	Идентификатор	Длина	Тип
Код цеха	A	2	символьный
Код участка	B	2	символьный
Код бригады	H	3	символьный
Таб.номер	C	4	символьный
Код операции	E	10	символьный
Количество	K	2	цифровой

Длина записи - 23 байта
Идентификатор файла - FN
Имя файла на диске - NAR
Упорядоченность файла: по цеху, участку, табельному номеру.

Структура записи файла «Рабочий»

Название	Идентификатор	Длина	Тип
Таб.номер	C	4	символьный
Фамилия	M	30	символьный
Год рождения	G	2	цифровой

Длина записи - 36 байт
Идентификатор файла - FR
Имя файла на диске - RAB
Упорядоченность файла: по табельному номеру.
Примечание: табельный номер соответствует номеру записи +1.

Структура записи файла «Подразделение»

Название	Идентификатор	Длина	Тип
Код цеха	A	2	символьный
Название цеха	X	25	символьный
Телефон	PH	7	символьный

Длина записи - 34 байта
Идентификатор файла - FP
Имя файла на диске - PODR
Упорядоченность файла: по цеху.
Примечание: код цеха соответствует номеру записи +1.

Структура записи файла «Операция»

Название	Идентификатор	Длина	Тип
Код операции	E	10	символьный
Расценка	P	2	цифровой
Единица измерения	EI	2	символьный

Длина записи - 14 байт
Идентификатор файла - FO
Имя файла на диске - OPER

Рис.9 Структуры записей входных файлов.

структуру, а не выборочно только те поля, которые задействованы при решении конкретной задачи.

Отношения между файлами для данной задачи могут быть представлены в виде диаграммы доступа на рис.10 или в виде информационных таблиц на рис.11.

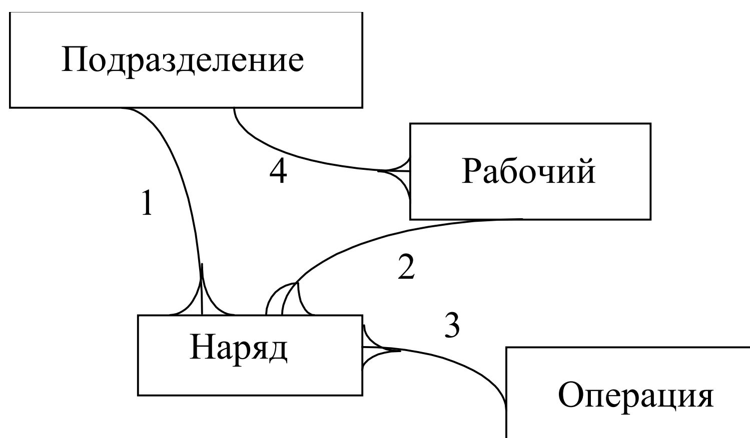


Рис.10 Диаграмма доступа.

На диаграмме доступа линия 1 показывает соотношение записей файла «Подразделение» и файла «Наряд». Т.о. видно, что для одного подразделения в файле «Наряд» может быть много записей, т.е. это соотношение «один-ко-многим». Линия 2 показывает аналогичное соотношение записей файла «Рабочий» и файла «Наряд», т.е. для одного рабочего может быть несколько записей в файле «Наряд», если конкретный рабочий за отчетный период выполнял работу по нескольким нарядам. Линия 4 показывает аналогичное соотношение записей файла «Подразделение» и файла «Рабочий», т.к. в одном подразделении работает несколько рабочих. Линия 3 показывает соотношение записей файла «Наряд» и файла «Операция». Это соотношение отражает тот факт, что одна и та же операция может встречаться во многих нарядах, но в одном наряде может быть только одна операция.

В информационных таблицах на рис.11 показан пример входных данных, на основе которых получены отчеты на рис.8. Здесь же обозначены соотношения между записями, содержащимися в разных таблицах.

Предполагается, что формирование значений «начислено» в выходных документах производится путем умножения количества выполненных операций на расценку и накопления полученных произведений. Предполагается также, что контроль входных данных, включая обращение к справочникам, был выполнен ранее.

«Наряд»

Код ц.	Код уч.	Таб.ном.	Код опер.	Кол-во
01	01	0001	IB090	20
01	01	0001	IA010	10
01	01	0001	2X100	30
01	01	0003	2X100	10
01	01	0003	IB090	10
01	02	0002	IB090	10
01	02	0004	IB090	10
04	02	0005	IB090	10
04	02	0007	2X200	10

«Рабочий»

Таб.номер	Фамилия
0001	Иванов
0002	Белов
0003	Петров
0004	Сидоров
0005	Буров
0006	Шуров
0007	Кузин

«Подразделение»

Код	Название
01	механический
02	сборочный
03	гальванический
04	прессо-кузовной

«Операция»

Код	Расценка
IA010	2
IA101	4
IA120	5
IB090	1
2X100	3
2X200	1

Рис.11 Входные данные.

6.7.2 Проектирование программы.

Для проектирования программы будем использовать подход МЭСИД. Процесс проектирования начинается с составления диаграмм структур выходных данных. Они представлены на рис.12. После составления необходимо убедиться в их структурной и информационной непротиворечивости. На базе этих диаграмм определяется «идеальный вход», обеспечивающий получение всех выходных документов. Структура «идеального входа» может быть получена при обработке последовательного файла «Наряд» путем прямого доступа к файлам-справочникам «Рабочий» и «Подразделение» и последовательного доступа к файлу-справочнику «Операция». Диаграммы структур входных данных показаны на рис.13. В этих диаграммах представлены только те поля записей, которые необходимы для получения «идеального входа». Диаграммы размечаются идентификаторами полей записей, полей управления, полей печати и вычислительными формулами. Убедившись в полноте и непротиворечивости постановки задачи на логическом уровне, можно перейти к составлению диаграммы структуры программы

и матрицы операций (рис.14), выделив в ней отдельные столбцы для каждого выходного документа, и затем к кодированию программы.

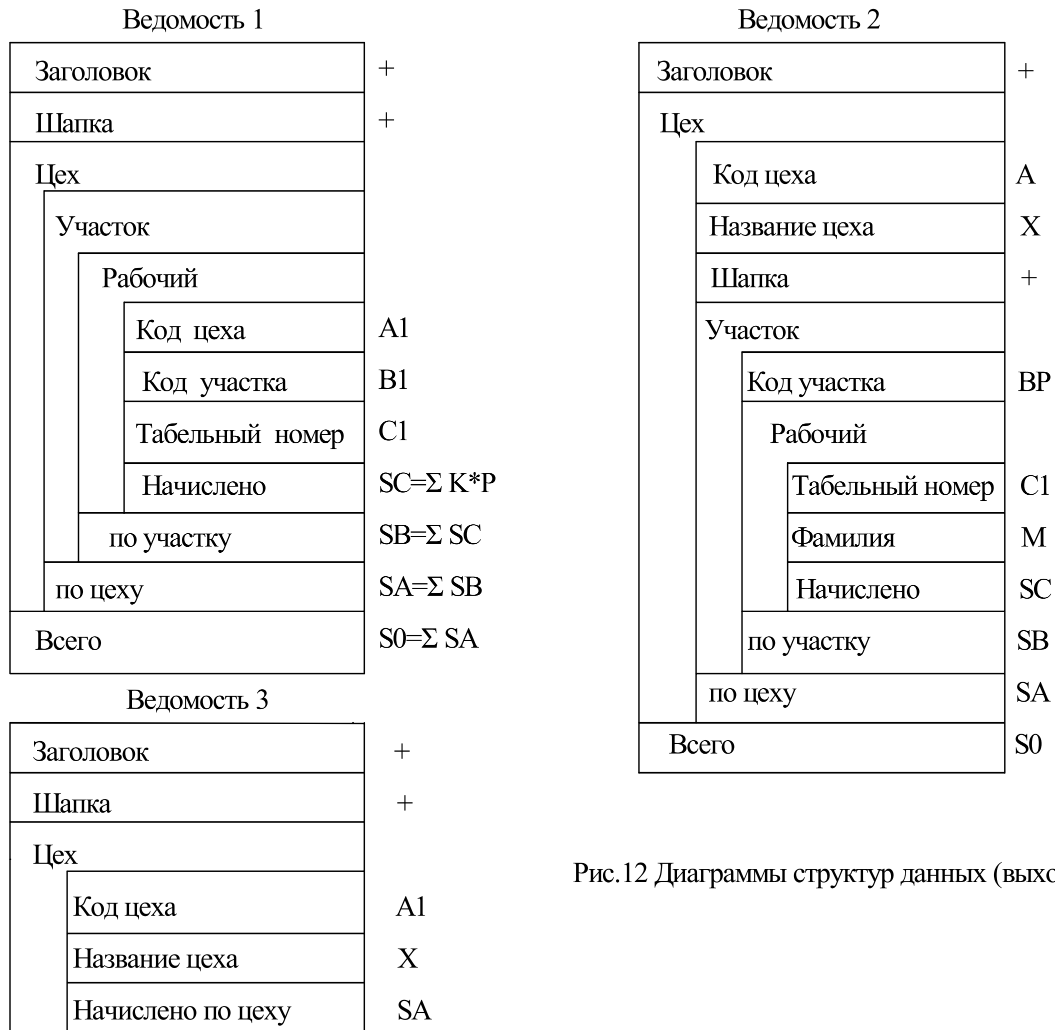


Рис.12 Диаграммы структур данных (выходных).

6.7.3 Кодирование программы.

Текст программы помимо тела, соответствующего матрице операций, должен содержать объявления файлов и структур данных. Для решения данной задачи используются не все поля записей основного и справочных файлов, однако необходимо отвести память для размещения входных записей. Должны быть объявлены и выходные файлы. Кроме того, должны быть объявлены поля управления, печати, итогов и т.п. Некоторые из этих переменных были определены при разметке диаграмм структур данных и заполнении матрицы операций, другие же могут определиться при детализации действий из матрицы операций, например, в связи с формированием ключей для обращения к файлам справочникам.

Текст программы приводится на рис.15. Таким образом из рассмотренного примера видно, что использование языка Паскаль в сочета-

нии с методологиями проектирования программ по структурам данных может обеспечить высокую эффективность реализации процедур обработки экономической информации.

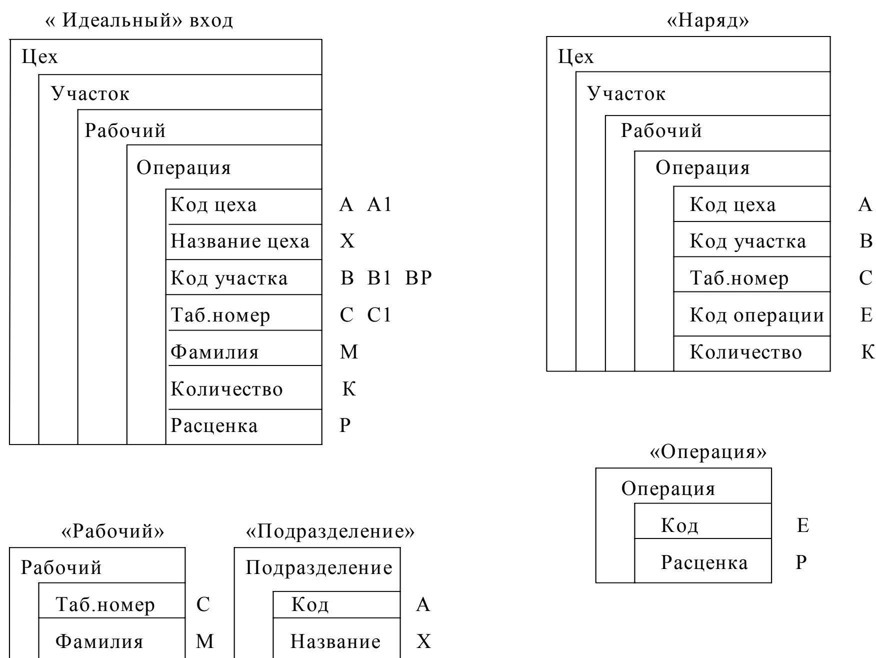


Рис.13 Диаграммы структур входных данных.

Структура программы	Подготовка переходов	Чтение с прямым доступом	Вычисления	Вывод			Чтение с послед. доступом
				Ведомость 1	Ведомость 2	Ведомость 3	
Начало программы	F=true	Открыть справочники	S0 = 0	Открыть, заголовок, шапка	Открыть, заголовок	Открыть заголовок, шапка	Открыть ‘Наряд’ , читать A, B, C, E, K
Цех	пока F						
Начало цеха	A1=A	Читать X для A	SA =0		A,X, шапка		
Участок	пока F и A1 = A						
Начало участка	B1=B		SB = 0		BP = B		
Таб.номер	пока F и A1 = A и B1 = B						
Начало таб.номера	C	Читать M для C	SC=0				
Операция	пока F и A1 = A и B1 = B и C1 = C						
Обработка операции		Читать P для E	SC=SC+P*K				Читать A, B, C, E, K
Конец аб.номера			SB=SB+SC	A1,B1,C1.SC	BP,C1,M,SC BP = ‘ ‘		
Конец участка			SA=SA+SB	SB	SB		
Конец цеха			S0=S0+SA	SA	SA	A1,X,SA	
Конец программы		Закрыть		S0, Закрыть	S0, Закрыть	Закрыть	Закрыть

Примечание: при конце файла ‘Наряд’ F = false

Рис.14 Структура программы и матрица операций.

```
type tn=record
       a:string[2];
       b:string[2];
       h:string[3];
       c:string[4];
       e:string[10];
       k:integer
           end;
     tr=record
       c:string[4];
       m:string[30];
       g:byte
           end;
     top=record
       e:string[10];
       p:word;
       ei:string[2]
           end;
     tp=record
       a:string[2];
       x:string[25];
       ph:string[7];
           end;
  var fn:file of tn;  zn:tn;
      fo:file of top; zo:top;
      fr:file of tr;  zr:tr;
      fp:file of tp;  zp:tp;
      v1,v2,v3:text;
      s0,sa,sb,sc,kod,chi:integer;
      f:boolean;
      a1,b1,bp:string[2];
      c1:string[4];
      nd:string;
      ch:char;
  begin
     write('имя осн.файла=');   readln(nd);
              assign(fn,nd);
     write('имя файла раб.='); readln(nd);
              assign(fr,nd);
     write('имя файла опер.='); readln(nd);
              assign(fo,nd);
     write('имя файла подр.='); readln(nd);
              assign(fp,nd);
     assign(v1,'v1');    rewrite(v1);
     assign(v2,'v2');    rewrite(v2);
```

```
assign(v3,'v3');    rewrite(v3);
writeln(v1,'          Ведомость 1');
writeln(v1,'  Цех   Участок   Таб.номер   ',
                                       Начислено');
writeln(v2,'          Ведомость 2');
writeln(v3,'          Ведомость 3');
writeln(v3,'    Цех      Название            ',
                                       Начислено');
s0:=0; f:=true;
reset(fn); read(fn,zn);
reset(fp); reset(fr);
while f do
begin
   a1:=zn.a; sa:=0;
   zp.x:=' '; val(a1,chi,kod);
   seek(fp,chi-1); read(fp,zp);
   writeln(v2,'Цех ',a1:3,'  ',zp.x:25);
   writeln(v2,'Участок Т.ном.  ФИО',
         '                     Начислено');
   while f and(a1=zn.a) do
   begin
      b1:=zn.b; bp:=b1; sb:=0;
      while f and(a1=zn.a)and(b1=zn.b) do
      begin
         c1:=zn.c; sc:=0;
         zr.m:=' '; val(c1,chi,kod);
         seek(fr,chi-1); read(fr,zr);
         while f and(a1=zn.a)and(b1=zn.b)and
                                  (c1=zn.c) do
         begin
            reset(fo); zo.p:=0;
            repeat
               read(fo,zo);
            until  eof(fo) or (zn.e=zo.e);
            close(fo);
            sc:=sc+zo.p*zn.k;
            if not eof(fn) then read(fn,zn)
                                else f:=false;
         end;
         writeln(v1,a1:5,b1:8,c1:10,sc:10);
         writeln(v2,bp:5,c1:7,'  ',zr.m:30,
                                            sc:3);
         bp:='';
         sb:=sb+sc;
      end;
```

```
          writeln(v1,'                  по участку',
                                                sb:5);
          writeln(v2,'                                ',
                              'по участку',  sb:5);
          sa:=sa+sb;
       end;
       writeln(v1,'                        по цеху',
                                                sa:5);
       writeln(v2,'                                  ',
                               ' по цеху', sa:5);
       writeln(v3, a1:6,zp.x:30,sa:5);
       s0:=s0+sa;
    end;
    writeln(v1,'                          ',
                                   'всего',s0:5);
    writeln(v2,'                                  ',
                                  ' всего',s0:5);
    close(v1);  close(v2);  close(v3);  close(fn);
end.
```

Рис.15 Текст программы.

Вопросы к главе 6.

1. Как можно описать файлы ?
2. Какие типы файлов существуют в Турбо Паскале ?
3. Как организовать прямой доступ к типизированным файлам ?
4. Особенности работы с типизированными файлами.
5. Особенности работы с текстовыми файлами.
6. Особенности работы с нетипизированными файлами.
7. Основные стандартные процедуры и функции для работы с типизированными файлами.
8. Основные стандартные процедуры и функции для работы с нетипизированными файлами.
9. Основные стандартные процедуры и функции для работы с текстовыми файлами.
10. Общий алгоритм создания файла.
11. Общий алгоритм обработки файла.
12. Какова последовательность шагов проектирования программ по структурам данных?

7. Динамическая память.

В предшествующих разделах использовались переменные, память под которые выделялась статически, то есть на стадии компиляции. Эти области памяти (для переменных из раздела **VAR** данного блока) существуют до конца работы блока, даже если эти переменные уже не нужны. При этом память нередко используется неэффективно, достаточно вспомнить «настройку» массива на фактическое количество элементов, а также, если, например, объявлено несколько массивов большого объема статической памяти, а в каждый конкретный момент используются не все.

Исправить положение можно, применив специальный механизм распределения памяти. Турбо Паскаль предоставляет возможность выделять и освобождать память в процессе выполнения программы, *динамически.*

Можно отметить следующие достоинства динамической памяти:

- экономичность и эффективность ее использования;

- возможность динамического изменения числа элементов в связанных структурах, например, списках (в статической памяти число элементов фиксировано для каждой компиляции);

- статические переменные существуют только в течение жизни блока, в котором они объявлены, а динамические - и после выхода из блока до окончания программы. Переменная, размещаемая динамически, не объявляется в разделе **VAR** и не имеет имени в программе («невидимка»). Компилятор не планирует выделение места в памяти под такие переменные.

7.1 Указатель.

Обращение к участку динамической памяти в программе осуществляется с помощью специальной ссылочной переменной, которая называется указателем (ссылкой).

Переменная типа «указатель» содержит адрес размещения участка динамической памяти, с которой связан этот указатель. Компилятор отводит под переменную типа «указатель» четыре байта статической памяти. Обычно указатель, связанный с определенным типом данных, называется типизированным. Однако он может быть и не типизированным, то есть совместимым с указателями любого типа данных. В этом случае указатель называется свободным (несвязанным).

Формат описания типа «указатель» следующий:

TYPE <идентификатор указателя>**=^**<тип>;

Примеры объявления типов «указатель» и переменных типа «указатель».

```
TYPE
        { правильные объявления типов}
    P1=^WORD; { p1 - идентификатор типа «указатель» на данные типа WORD.}
    P2=^CHAR;   { p2 - идентификатор типа «указатель» на данные типа CHAR}
    P4=ARRAY[1..10] OF ^REAL; {p4 - идентификатор типа «указатель» на массив указателей, ссылающихся на данные типа REAL}
        { неправильные объявления типов}
    P5=^ARRAY[1..10] OF REAL;
    P6=^STRING[25];
    P7=^RECORD
            FIELD1 : STRING [15];
            FIELD2 : REAL;
      END;
```

В формате объявления типа «указатель» должен быть указан идентификатор типа, поэтому стандартные идентификаторы (**INTEGER, REAL** и т.д.) можно указывать непосредственно в описаниях типа «указатель». Ошибки в описаниях типов **P5, P6** и **P7** будут отмечены компилятором из-за того, что, в таких случаях надо прежде описать идентификатор типа, а затем использовать его в других описаниях.

Следующие описания будут правильными:

```
TYPE
      ...
      MAS = ARRAY[1..10] OF REAL;
      ST = STRING[25];
      REC = RECORD
                FIELD1 : STRING [15];
                FIELD2 : REAL;
            END;
  VAR
      P5 : ^MAS;
      P6 : ^ST;
      P7 : ^REC;
      ...
```

Указатель может находиться в одном из трех состояний, а именно:

1) еще не инициализирован;
2) содержит адрес размещения;
3) содержит значение предопределенной константы **NIL**; такой указатель называется пустым, то есть не указывает ни на какую переменную. Указатель со значением **NIL** содержит 0 в каждом из четырех байтов.

Указатели можно сравнивать с другими указателями (=, <>), присваивать им адрес или значение другого указателя, передавать как параметр. Указатель нельзя отпечатать или вывести на экран.

Обращение к выделенной динамический памяти кодируется следующим образом:

<идентификатор указателя>^

Рассмотрим пример обращения к переменным, размещенным в динамической памяти:

```
TYPE
   SYM=^CHAR;
   ZAP=RECORD
        FIELD1, FIELD2: REAL;
      END;
   M=ARRAY[0..9] OF WORD;
VAR
   CH : SYM;
   REC :  ^ZAP;
   MAS :  ^M;
     ...
   CH^:='*';    {обращение к динамической переменной типа
CHAR, запись в эту область символа звездочка}
     ...
   READLN (REC^.FIELD1);    {обращение к полю FIELD1 динамической записи, ввод в него данных с клавиатуры }
     ...
   WRITELN ( MAS[5]^);  {обращение к элементу MAS[5] динамического массива, вывод на экран значения указанного элемента}
     ...
```

Фактически можно говорить, что **CH^, REC^.FIELD1** и **MAS[5]^** исполняют роль имён динамических объектов в программе, адреса которых хранятся в указателях **CH, REC** и **MAS** соответственно.

Следует отметить, что обращение к переменным типа **POINTER** (указателям, которые не указывают ни на какой определенный тип и совместимы со всеми другими типами указателей) приводит к ошибке.

Например.

```
VAR
   P:POINTER;
     ...
   P^:=1; {ошибка!}
```

7.2 Стандартные процедуры размещения и освобождения динамической памяти.

При выполнении программы наступает момент, когда необходимо использовать динамическую память, т.е. выделить её в нужных видах, разместить там какие-то данные, поработать с ними, а после того, как в данных отпадет необходимость - освободить выделенную память.

Динамическая память может быть выделена двумя способами:

1. С помощью стандартной процедуры **NEW**:

New (P);

где **P** - переменная типа «типизированный указатель».

Эта процедура создает новую динамическую переменную (выделяет под нее участок памяти) и устанавливает на нее указатель **P** (в **P** записывается адрес выделенного участка памяти). Размер и структура выделяемого участка памяти задается размером памяти для того типа данных, с которым связан указатель **P**. Доступ к значению созданной переменной можно получить с помощью **P^.**

2. С помощью стандартной процедуры **GETMEM.**

GetMem (P,size);

где **P** - переменная типа «указатель» требуемого типа.
size - целочисленное выражение размера запрашиваемой памяти в байтах.

Эта процедура создает новую динамическую переменную требуемого размера и свойства, а также помещает адрес этой созданной переменной в переменную **P** типа «указатель». Доступ к значению созданной переменной можно получить с помощью **P^.**

Например:

```
TYPE
   REC =RECORD
         FIELD1:STRING[30];
         FIELD2:INTEGER;
       END;
   PTR_REC = ^ REC;
VAR
     P : PTR_REC;
BEGIN
     GETMEM(P, SIZEOF (REC)); { Выделение памяти, адрес выделенного участка фиксируется в P; размер этой памяти в байтах определяет и возвращает стандартная функция SizeOf , примененная к описанному типу данных; однако, зная размеры внутреннего представления используемых полей, можно было бы подсчи-
```

тать размер памяти «вручную» и записать в виде константы вместо **SizeOf (Rec)** }

...

{использование памяти}

...

FREEMEM(P, SIZEOF(REC)); {освобождение уже ненужной памяти}

...

Динамическая память может быть освобождена четырьмя способами.

1. Автоматически по завершении всей программы.
2. С помощью стандартной процедуры **DISPOSE.**

Dispose (P);

где **P** - переменная типа «указатель» (типизированный).

В результате работы процедуры **DISPOSE(P)** участок памяти, связанный с указателем **P**, помечается как свободный для возможных дальнейших размещений. При этом физической чистки указателя **P** и связанной с ними памяти не происходит, поэтому, даже удалив этот экземпляр записи, можно все же получить значения ее полей, однако использовать это обстоятельство не рекомендуется.

*Ввиду различия в способах реализации процедуру **DISPOSE** не следует использовать совместно с процедурами **MARK** и **RELEASE**.*

3. С помощью стандартной процедуры **FREEMEM**.

FreeMem (P, size);

где **P** - переменная типа «указатель»,

size - целочисленное выражение размера памяти в байтах для освобождения.

Эта процедура помечает память размером, равным значению выражения **SIZE**, связанную с указателем **P**, как свободную (см. пример для **GETMEM).**

4. С помощью стандартных процедур **MARK** и **RELEASE**.

Mark (P);

Release (P);

где **P** - переменная типа «указатель»;

MARK - запоминает состояние динамической области в переменной-указателе **P**;

RELEASE - освобождает всю динамическую память, которая выделена процедурами **NEW** или **GETMEM** после запоминания текущего значения указателя **P** процедурой **MARK**.

Обращения к **MARK** и **RELEASE** нельзя чередовать с обращениями к **DISPOSE** и **FREEMEM** ввиду различий в их реализации.

Например:

```
VAR
   P:POINTER;
   P1, P2, P3:^INTEGER;
BEGIN
   NEW(P1);
   P1^ := 10;
   MARK(P); {пометка динамической области}
   NEW(P2);
   P2^ :=  25;
   NEW(P3);
   P3^ :=  P2^ + P1^;
   WRITELN ( P3^);
   RELEASE(P); {память, связанная с P2^ и P3^, освобождена, а
P1^ может использоваться}
END.
```

7.3 Стандартные функции обработки динамической памяти.

В процессе выполнения программы может возникнуть необходимость наблюдения за состоянием динамической области. Цель такого наблюдения - оценка возможности очередного выделения динамической области требуемого размера. Для этих целей Турбо Паскаль предоставляет две функции (без параметров).

MaxAvail;

Эта функция возвращает размер в байтах наибольшего свободного в данный момент участка в динамической области. По этому размеру можно судить о том, какую наибольшую динамическую память можно выделить.

Тип возвращаемого значения - **longint**.

```
TYPE ZAP=RECORD
       FIELD1: STRING [20];
       FIELD2: REAL;
      END;
VAR P: POINTER;
BEGIN
   ...
   IF MAXAVAIL <SIZEOF(ZAP)
   THEN
     WRITELN ('НЕ ХВАТАЕТ ПАМЯТИ!')
   ELSE
     GETMEM(P, SIZEOF(ZAP));
   ...
```

Вторая функция:

MemAvail;

Эта функция возвращает общее число свободных байтов динамической памяти, то есть суммируются размеры всех свободных участков и объем свободной динамической области. Тип возвращаемого значения - **longint**.

```
...
WRITELN( 'Доступно', MEMAVAIL, ' байтов' );
WRITELN('Наибольший свободный участок=', MAXAVAIL, 'байтов' );
...
```

Это решение основано на следующем обстоятельстве. Динамическая область размещается в специально выделяемой области, которая носит название «куча» (**HEAP**). Куча занимает всю или часть свободной памяти, оставшейся после загрузки программы. Размер кучи можно установить с помощью директивы компилятора **M**:

```
{$M <стек>, <минимум кучи>, <максимум кучи>}
```

где <стек> - специфицирует размер сегмента стека в байтах. По умолчанию размер стека 16 384 байт, а максимальный размер стека 65 538 байт;

<минимум кучи> - специфицирует минимально требуемый размер кучи в байтах; по умолчанию минимальный размер 0 байт;

<максимум кучи> - специфицирует максимальное значение памяти в байтах для размещения кучи; по умолчанию оно равно 655 360 байт, что в большинстве случаев выделяет в куче всю доступную память; это значение должно быть не меньше наименьшего размера кучи.

Все значения задаются в десятичной или шестнадцатеричной формах. Например, следующие две директивы эквивалентны:

```
{$M 16384,0,655360}
{$M $4000, $0, $A000}
```

Если указанный минимальный объем памяти недоступен, то программа выполняться не будет.

Управление размещением в динамической памяти осуществляет администратор кучи, являющийся одной из управляющих программ модуля ***System***.

7.4 Примеры и задачи.

Рассмотрим пример размещения и освобождения разнотипных динамических переменных в куче.

```
TYPE
   ST1=STRING[7];
   ST2=STRING[3];
VAR I,I1,I2,I3,I4:^INTEGER;
   R^:REAL;
   S1:^ST1;
   S2:^ST2;
BEGIN
   NEW(I);
   I1^:=1;
   NEW(I2);
   I2^:=2;
   NEW(I3);
   I3^=3;
   NEW(I4);
   I4^:=4; (*1*)
   DISPOSE (I2);{освобождается второе размещение}
   NEW (I); {память нужного размера (в данном случае два байта)
         выделяется на первом свободном месте от начала кучи,
         достаточном для размещения данной переменной; в этом
         примере - это участок, который занимала переменная I2^,
         ее адрес остался в указателе I2 }
   I^:=5; (*2*)
   DISPOSE(I3); {освобождается третье размещение}
   NEW(R); {память под переменную типа REAL выделяется в
         вершине кучи, так как размер дырки с адресом I3 (2 бай-
         та) мал для размещения переменной типа REAL, для ко-
         торой необходимо 6 байт }
   R^:=6; (*3*)
   WRITELN (R^); { ВЫВОД: 6.0000000000E+00}
END.
```

В следующем примере используется массив указателей.

```
USES CRT;
VAR
   R: ARRAY [1..10] OF  ^REAL;
   I:1..10;
BEGIN
   RANDOMIZE; {инициализация генератора случайных чисел}
   FOR I:=1 TO 10 DO
   BEGIN
```

```
        NEW(R[I]);
        R[I]^:=RANDOM; {генерация случайных вещественных чисел
в диапазоне 0 <= r[i]^ < 1}
        WRITELN(R[I]^);{Вывод случайных чисел в экспоненциальной форме}
    END;
END.
```

7.5 Работа с динамическими массивами.

При работе с массивами практически всегда возникает задача настройки программы на фактическое количество элементов массива. В зависимости от применяемых средств решение этой задачи бывает различным.

Первый вариант - использование констант для задания размерности массива**.**

```
PROGRAM FIRST;

CONST
   N : INTEGER = 10;
      { либо N = 10; }
VAR
   A : ARRAY [ 1..N ] OF REAL;
   I : INTEGER;
BEGIN
    FOR I := 1 TO N DO
    BEGIN
         WRITELN (' Введите ', I , ' -ый элемент массива ');
         READLN ( A [ I ] )
   END;
{ И далее все циклы работы с массивом используют N}
```

Такой способ требует перекомпиляции программы при каждом изменении числа обрабатываемых элементов.

Второй вариант - программист планирует некоторое условно максимальное (теоретическое) количество элементов, которое и используется при объявлении массива. При выполнении программа запрашивает у пользователя фактическое количество элементов массива, которое должно быть не более теоретического. На это значение и настраиваются все циклы работы с массивом.

```
PROGRAM SECOND;
VAR
   A : ARRAY [ 1..25 ] OF REAL;
   I, NF : INTEGER;
BEGIN
  WRITELN ('Введите фактическое число элементов',
                                              ' массива <=  25 ');
  READLN ( NF );
  FOR I := 1 TO NF DO
  BEGIN
       WRITELN ('Введите ', I , ' -ый элемент массива ');
       READLN ( A [ I ] )
  END;
{ И далее все циклы работы с массивом используют NF}
```

Этот вариант более гибок и технологичен по сравнению с предыдущим, так как не требуется постоянная перекомпиляция программы, но очень нерационально расходуется память, ведь ее объем для массива всегда выделяется по указанному максимуму. Используется же только часть ее

Вариант третий - в нужный момент времени надо выделить динамическую память в требуемом объеме, а после того, как она станет не нужна, освободить ее.

```
PROGRAM DYNAM_MEMORY;
TYPE
   MAS = ARRAY [ 1..2 ] OF  < требуемый_тип_элемента >;
   MS = ^ MAS;
VAR
   A : MS;
   I, NF : INTEGER;
BEGIN
   WRITELN ('Введите фактическое число элементов масси-
ва');
   READLN ( NF );
   GETMEM ( A, SIZEOF ( < требуемый_тип_элемента>)*NF);
   FOR I := 1 TO NF DO
   BEGIN
        WRITELN ('Введите ', I , ' -ый элемент массива ');
        READLN ( A^ [ I ] )
   END;
   { И далее все циклы работы с массивом используют NF}
   . . . . .
   FREEMEM (A, NF*SIZEOF (< требуемый_тип_элемента>));
END.
```

Рассмотрим пример использования динамического одномерного массива, который используется как двумерный массив. После ввода реальной размерности массива и выделения памяти для обращения к элементу двумерного массива адрес его рассчитывается, исходя из фактической длины строки и положения элемента в строке (при заполнении матрицы по строкам). Требуется найти максимальный элемент в матрице и его координаты.

```
USES CRT;
TYPE T1=ARRAY[1..1] OF INTEGER;
VAR
        A:^T1;
        N,M,I,J,K,P:INTEGER;
        MAX:INTEGER;
BEGIN
        CLRSCR;
        WRITE('N=');   READLN (N);
        WRITE('M=');   READLN (M);
        GETMEM (A,SIZEOF(INTEGER)*N*M);
        FOR I:=1 TO N*M DO
                READ(A^[ I ]);
        MAX:=A^[1];  K:=1;  P:=1;
        FOR I:=1 TO N DO
                FOR J:=1 TO M DO
                        IF A^[(I-1)*M+J] > MAX THEN
                        BEGIN
                                MAX:=A^[(I-1)*M+J];
                                K:=I;  P:=J
                        END;
        WRITE('строка=',K:2,'  столбец=',P:2);
        FREEMEM(A,2*N*M);
        READKEY;
END.
```

В следующем примере для хранения двумерного массива используется одномерный массив указателей на столбцы. В задаче требуется найти столбцы матрицы, в которых находятся минимальный и максимальный элементы матрицы и если это разные столбцы, то поменять их местами.

```
USES CRT;
TYPE
      VK=^T1;
      T1=ARRAY[1..1] OF INTEGER;
      MT=^T2;
      T2=ARRAY[1..1] OF VK;
VAR
      A:MT;
      M,N,I,J,K,L:INTEGER;
      MAX,MIN:INTEGER;
      R:POINTER;
BEGIN
       CLRSCR;
       READLN (N,M);
       {выделение памяти под указатели столбцов матрицы}
       GETMEM(A,SIZEOF (POINTER)*M);
       {выделение памяти под элементы столбцов}
       FOR J:=1 TO M DO
              GETMEM (A^[J],SIZEOF(INTEGER)*N);
       FOR I:=1 TO N DO
             FOR J:=1 TO M DO
                    READ(A^[ J ]^[ I ]);
       FOR I:=1 TO N DO
       BEGIN
             FOR J:=1 TO M DO
                   WRITE (A^[ J ]^[ I ]:4);
             WRITELN
       END;
       MAX:=A^[1]^[1];  K:=1;  MIN:=MAX;  L:=1;
       FOR J:=1 TO M DO
              FOR I:=1 TO N DO
                    IF A^[ J ]^[ I ]<MIN THEN
                    BEGIN
                           MIN:=A^[J]^[I];  L:=J;
                    END
                                                ELSE
                    IF A^[ J ]^[ I ]>MAX THEN
                    BEGIN
                           MAX:=A^[J]^[I];  K:=J;
                    END;
{для обмена столбцов достаточно поменять указатели на столбцы}
       IF K<>L THEN
       BEGIN
             R:=A^[K];  A^[K]:=A^[L];  A^[L]:=R
       END;
       FOR I:=1 TO N DO
       BEGIN
             FOR J:=1 TO M DO
                    WRITE(A^[J]^[I]:3,' ');
             WRITELN
       END;
       FOR I:=1 TO M DO
             FREEMEM (A^[ I ],N*SIZEOF(INTEGER));
       FREEMEM (A,M*SIZEOF(POINTER))
END.
```

7.6 Организация списков.

Преимущества динамической памяти становятся особенно очевидными при организации динамических структур, элементы которых связаны через адреса (стеки, очереди, деревья, сети и т.д.). Основой моделирования таких структур являются списки.

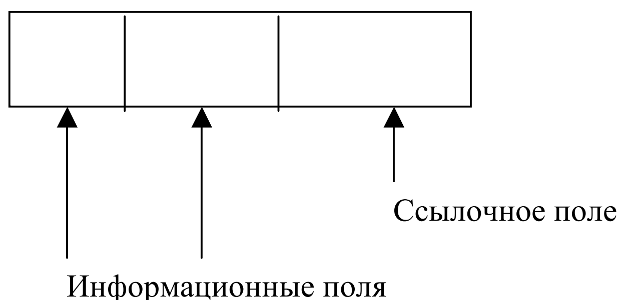
Список - это конечное множество динамических элементов, размещающихся в разных областях памяти и объединенных в логически упорядоченную последовательность с помощью специальных указателей (адресов связи).

Список - структура данных, в которой каждый элемент имеет информационное поле (поля) и ссылку (ссылки), то есть адрес (адреса), на другой элемент (элементы) списка. Список - это так называемая линейная структура данных, с помощью которой задаются одномерные отношения.

Каждый элемент списка содержит информационную и ссылочную части. Порядок расположения информационных и ссылочных полей в элементе при его описании - по выбору программиста, то есть фактически произволен. Информационная часть в общем случае может быть неоднородной, то есть содержать поля с информацией различных типов. Ссылки однотипны, но число их может быть различным в зависимости от типа списка. В связи с этим для описания элемента списка подходит только тип «запись», так как только этот тип данных может иметь разнотипные поля. Например, для однонаправленного списка элемент должен содержать как минимум два поля: одно поле типа «указатель», другое - для хранения данных пользователя. Для двунаправленного - три поля, два из которых должны быть типа «указатель».

Структура элемента линейного однонаправленного списка представлена на рисунке 16.

(а) Структура элемента однонаправленного списка



(б) Структура элемента двунаправленного списка

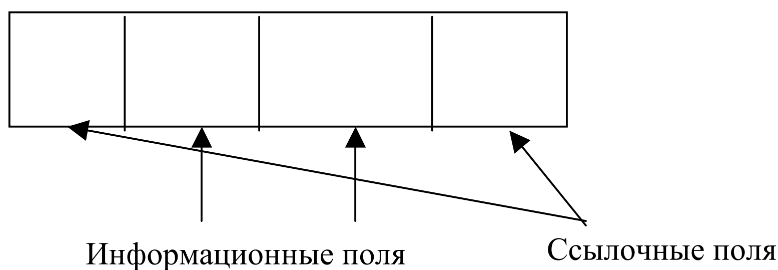


Рис 16. Структуры элементов списка (условно)

Следует рассмотреть разницу в порядке обработки элементов массива и списка.

Элементы массива располагаются в памяти в определенном постоянном порядке - подряд, друг за другом, что закрепляется их номерами. Каждый элемент массива имеет свое место, которое не может быть изменено, хотя значение элемента может изменяться. Порядок обработки элементов определяется использованием их номеров, индексов.

В отличие от элементов массива элементы списка могут располагаться в памяти в свободном порядке, не подряд. Порядок их обработки определяется ссылками, то есть в общем случае очередной элемент своей ссылкой указывает на тот элемент, который должен быть обработан следующим. Последний по порядку элемент содержит в ссылочной части признак, свидетельствующий о необходимости прекращения обработки элементов списка, указывающий как бы конец списка.

В зависимости от числа ссылок список называется одно-, двунаправленным и т.д.

В однонаправленном списке каждый элемент содержит ссылку на последующий элемент. Если последний элемент списка содержит «нулевую» ссылку, то есть содержит значение предопределенной константы **NIL** и, следовательно, не ссылается ни на какой другой элемент, такой список называется линейным.

Для доступа к первому элементу списка, а за ним - и к последующим элементам необходимо иметь адрес первого элемента списка. Этот адрес обычно записывается в специальное поле - указатель на первый элемент, дадим ему специальное, «говорящее» имя - **FIRST**. Если значение **FIRST** равно **NIL**, это значит, что список пуст, он не содержит ни одного элемента. Оператор **FIRST := NIL;** должен быть первым оператором в программе работы со списками. Он выполняет инициализацию указателя первого элемента списка, иначе говоря, показывает, что список пуст. Всякое другое значение будет означать адрес первого элемента списка (не путать с неинициализированным состоянием указателя).

Структура линейного однонаправленного списка показана на рисунке 17.

Если последний элемент содержит ссылку на первый элемент списка, то такой список называется кольцевым, циклическим. Изменения в списке при этом минимальны - добавляется ссылка с последнего на первый элемент списка: в адресной части последнего элемента значение **Nil** заменяется на адрес первого элемента списка (см. рис. 18).

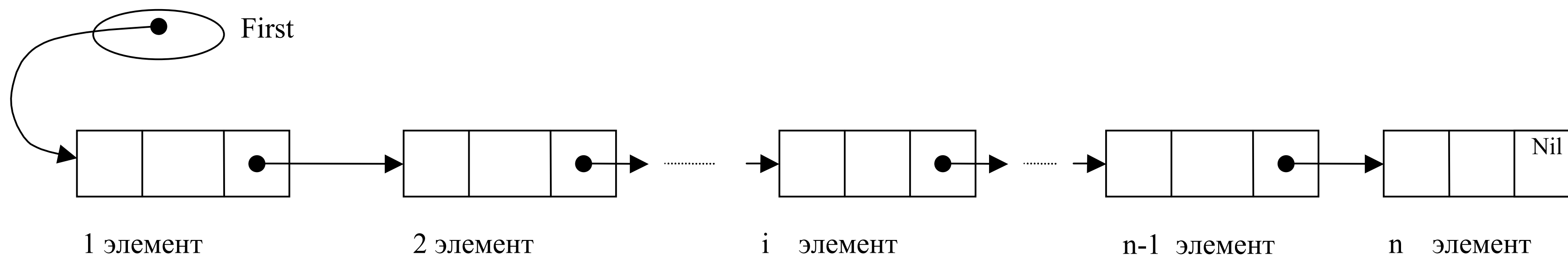


Рис. 17 Структура линейного однонаправленного списка

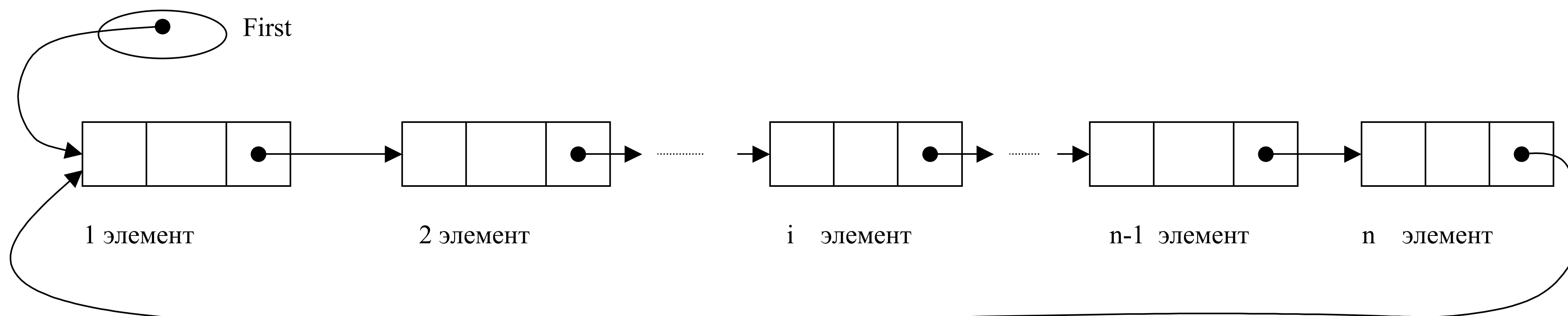


Рис.18 Структура кольцевого (циклического) однонаправленного списка

При обработке однонаправленного списка могут возникать трудности, связанные с тем, что по списку с такой организацией можно двигаться только в одном направлении, как правило, начиная с первого элемента. Обработка списка в обратном направлении сильно затруднена. Для устранения этого недостатка служит двунаправленный список, каждый элемент которого содержит ссылки на последующий и предыдущий элементы (для линейных списков - кроме первого и последнего элементов). Структура элемента представлена на рис. 1б.

Такая организация списка позволяет от каждого элемента двигаться по списку как в прямом, так и в обратном направлениях. Наиболее удобной при этом является та организация ссылок, при которой движение, перебор элементов в обратном направлении является строго противоположным перебору элементов в прямом направлении. В этом случае список называется симметричным. Например, в прямом направлении элементы линейного списка пронумерованы и выбираются так: 1, 2, 3, 4, 5. Строго говоря, перебирать элементы в обратном направлении можно по-разному, соответствующим способом организуя ссылки, например: 4, 1, 5, 3, 2. Симметричным же будет называться список, реализующий перебор элементов в таком порядке: 5, 4, 3, 2, 1.

Следует заметить, что «обратный» список, так же, как и прямой, является просто линейным однонаправленным списком, который заканчивается элементом со ссылкой, имеющей значение **NIL**. Для удобства работы со списком в обратном направлении и в соответствии с идеологией однонаправленного списка нужен доступ к первому в обратном направлении элементу. Такой доступ осуществляется с помощью указателя **LAST** на этот первый в обратном направлении элемент. Структура линейного двунаправленного симметричного списка дана на рис. 19 .

Как указывалось ранее, замкнутый, циклический, кольцевой список организован таким образом, что в адресную часть конечного элемента вместо константы **NIL** помещается адрес начального элемента (список замыкается на себя). В симметричном кольцевом списке такое положение характерно для обоих - прямого и обратного - списков, следовательно, можно построить циклический двунаправленный список (см. рис. 20) .

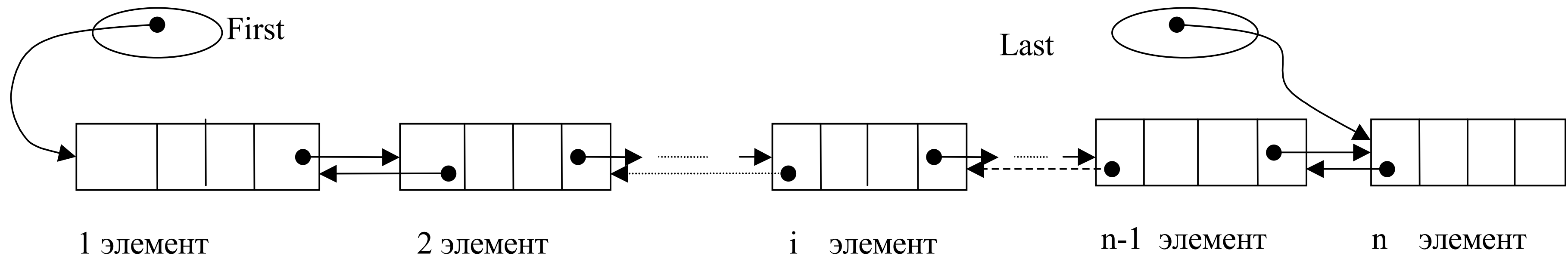


Рис. 19 Структура линейного симметричного двунаправленного списка

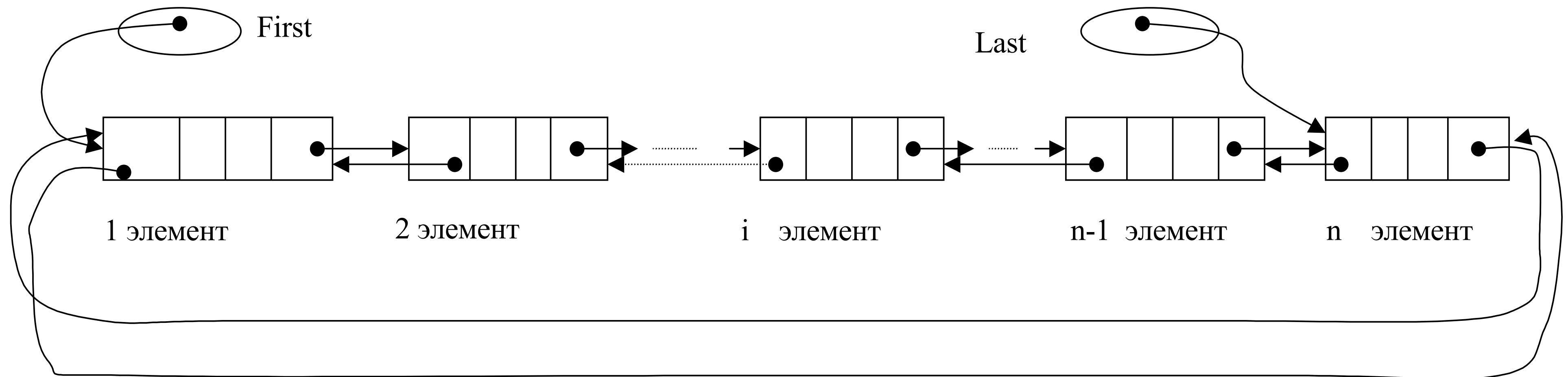


Рис. 20 Структура циклического (кольцевого) симметричного списка

Описать элемент однонаправленного списка (см. рис 1) можно следующим образом:

```
TYPE
POINT=^ZAP;
ZAP=RECORD
        INF1 :  INTEGER;  { первое информационное поле  }
        INF2 : STRING;    { второе информационное поле  }
        NEXT : POINT;    {ссылочное поле  }
   END;
```

Из этого описания видно, что имеет место рекурсивная ссылка: для описания типа **POINT** используется тип **ZAP**, а при описании типа **ZAP** используется тип **POINT**. По соглашениям Паскаля в этом случае сначала описывается тип «указатель», а затем уже тип связанной с ним переменной. Правила Паскаля только при описании ссылок допускают использование идентификатора (**ZAP**) до его описания. Во всех остальных случаях, прежде чем упомянуть идентификатор, необходимо его определить.

В случае двунаправленного списка в описании должно появиться еще одно ссылочное поле того же типа, например,

```
TYPE
   POINT=^ZAP;
   ZAP=RECORD
      INF1 :  INTEGER; { первое информационное поле  }
      INF2 : STRING;  { второе информационное поле  }
      NEXT:POINT; {ссылочное поле на следующий элемент}
      PREV:POINT; {ссылочное поле на предыдущий элемент}
    END;
```

Как уже отмечалось, последовательность обработки элементов списка задается системой ссылок. Отсюда следует важный факт: все действия над элементами списка, приводящие к изменению порядка обработки элементов списка - вставка, удаление, перестановка - сводятся к действиям со ссылками. Сами же элементы не меняют своего физического положения в памяти.

При работе со списками любых видов нельзя обойтись без указателя на первый элемент. Не менее полезными, хотя и не всегда обязательными, могут стать адрес последнего элемента списка и количество элементов. Эти данные могут существовать по отдельности, однако их

можно объединить в единую структуру типа «запись» (из-за разнотипности полей: два поля указателей на элементы и числовое поле для количества элементов). Эта структура и будет представлять так называемый головной элемент списка. Следуя идеологии динамических структур данных, головной элемент списка также необходимо сделать динамическим, выделяя под него память при создании списка и освобождая после окончания работы. Использование данных головного элемента делает работу со списком более удобной, но требует определенных действий по его обслуживанию.

Рассмотрим основные процедуры работы с линейным однонаправленным списком без головного элемента. Действия оформлены в виде процедур или функций в соответствии с основными требованиями модульного программирования (см. соответствующий раздел пособия).

Приведем фрагменты разделов **TYPE** и **VAR**, необходимые для дальнейшей работы с линейным однонаправленным списком без головного элемента.

```
TYPE
   EL = ^ZAP;
   ZAP=RECORD
          INF1 :  INTEGER; { первое информационное поле  }
          INF2 : STRING;   { второе информационное поле  }
          NEXT : EL;       {ссылочное поле  }
     END;
     VAR
     FIRST,          { указатель на первый элемент списка }
     P, Q , T : EL;     { рабочие указатели, с помощью которых
                  будет выполняться работа с элементами   списка }
```

7.7 Задачи включения элемента в линейный однонаправленный список без головного элемента.

Графическое представление выполняемых действий дано на рис. 21

Формирование пустого списка.

```
PROCEDURE CREATE_EMPTY_LIST ( VAR  FIRST : EL);
BEGIN
        FIRST = NIL;
END;
```

Формирование очередного элемента списка.

```
PROCEDURE CREATE_NEW_ELEM(VAR P: EL);
BEGIN
       NEW (P);
       WRITELN ('введите значение первого информационного поля: ');
       READLN ( P^.INF1 );
       WRITELN ('введите значение второго информационного поля: ');
       READLN ( P^.INF2 );
       P^.NEXT := NIL;
       { все поля элемента должны быть инициализированы }
END;
```

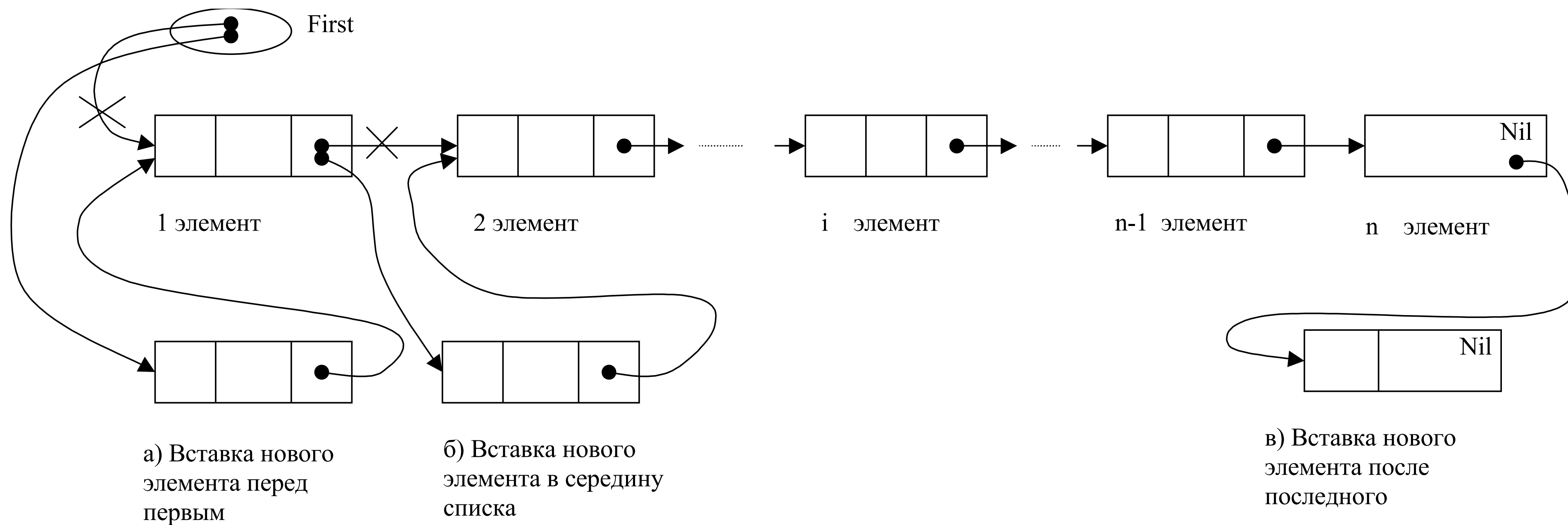


Рис. 21 Линейный однонаправленный список без головного элемента. Включение элементов в начало (а) , в середину (б), в конец (в) списка

Подсчет числа элементов списка.

```
FUNCTION COUNT_EL(FIRST:EL):INTEGER;
 VAR
     K : INTEGER;
     Q : EL;
BEGIN
     IF FIRST = NIL THEN
        K:=0    { список пуст }
                     ELSE
        BEGIN        {список существует}
            K:=1;   {в списке есть хотя бы один элемент}
            Q:=FIRST;
     {перебор элементов списка начинается с  первого}
           WHILE Q^.NEXT <> NIL DO
          BEGIN
            K:=K+1;
            Q:=Q^.NEXT;
          {переход к следующему элементу списка}
          END;
      END;
      COUNT_EL:=K;
END;
```

ПРИМЕЧАНИЕ: аналогично может быть написана процедура, например, печати списка, поиска адреса последнего элемента и др.

Вставка элемента в начало списка.

```
PROCEDURE INS_BEG_LIST(P : EL; {адрес включаемого элемента}
                             VAR FIRST : EL);
BEGIN
    IF FIRST = NIL THEN
      BEGIN
          FIRST := P;
          P^.NEXT := NIL { можно не делать, так как уже сделано при
                              формировании этого элемента}
      END
                 ELSE
      BEGIN
          FIRST:=P;{ включаемый элемент становится первым }
          P^.NEXT:=FIRST;{ссылка на бывший первым элемент}
      END;
END;
```

Включение элемента в конец списка.

```
PROCEDURE INS_END_LIST(P : EL; VAR FIRST : EL);
BEGIN
      IF FIRST = NIL THEN
         FIRST:=P
                      ELSE
         BEGIN
               Q:=FIRST; {цикл поиска адреса последнего элемента}
               WHILE Q^.NEXT <> NIL DO
                    Q:=Q^.NEXT;
              Q^.NEXT:=P;{ссылка с бывшего последнего
                              на включаемый элемент}
               P^.NEXT:=NIL; {не обязательно}
          END;
END;
```

Включение в середину (после i-ого элемента).

```
PROCEDURE INS_AFTER_I ( FIRST : EL; P : EL; I : INTEGER);
VAR
      T, Q : EL;
      K ,N : INTEGER;
BEGIN
      N := COUNT_EL(FIRST); {определение числа элементов списка}
      IF (I < 1 )  OR ( I > N )THEN
          BEGIN
              WRITELN ('i задано некорректно');
              EXIT;
         END
                             ELSE
          BEGIN
              IF I = 1 THEN
                   BEGIN
                       T := FIRST;{адрес 1 элемента}
                       Q := T^.NEXT; {адрес 2 элемента}
                       T^.NEXT := P;
                        P^.NEXT := Q;
                  END
                      ELSE
                  IF I = N THEN
                       BEGIN { см. случай вставки после последнего
                                                  элемента}
                              . . .
                        END
```

```
        ELSE {вставка в «середину» списка}
      BEGIN
        T := FIRST;
        K := 1;
        WHILE ( K < I ) DO
          BEGIN {поиск адреса i-го элемента}
            K := K + 1;
            T := T^.NEXT;
        END;
        Q := T^.NEXT;
       {найдены адреса i-го (t) и  i+1 -го (q) элементов }
        T^.NEXT := P;
        P^.NEXT := Q;
       {элемент с адресом p вставлен}
      END;
    END;
END;
```

ПРИМЕЧАНИЕ: аналогично рассуждая и применяя графическое представление действия, можно решить задачу включения элемента перед i-ым. Строго говоря, такая задача не эквивалентна задаче включения элемента после (i-1)-го.

7.8 Задачи на удаление элементов из линейного однонаправленного списка без головного элемента.

Графическое представление выполняемых действий дано на рисунке 22.

Удаление элемента из начала списка.

ПРИМЕЧАНИЕ: перед выполнением операции удаления элемента или списка желательно запрашивать у пользователя подтверждение удаления.

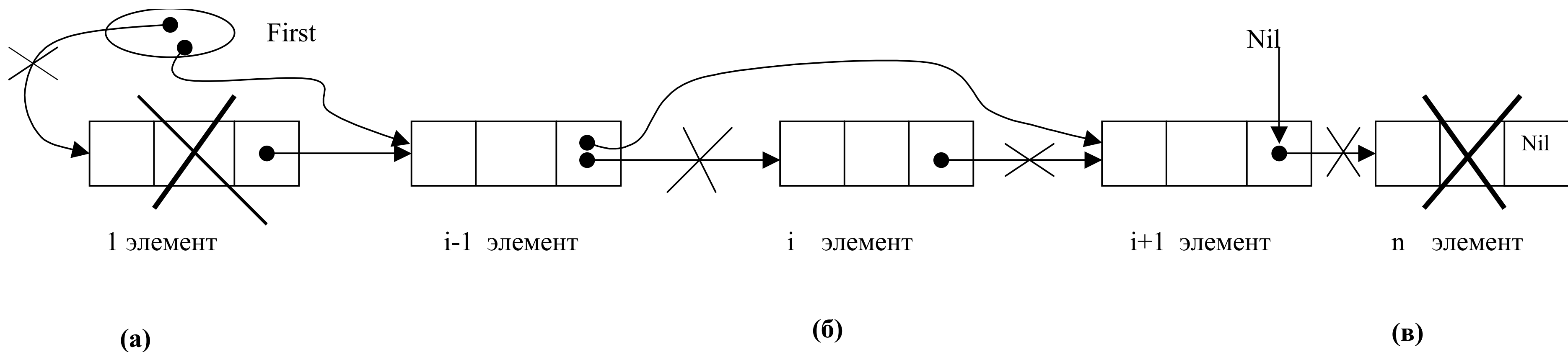


Рис. 22 Линейный однонаправленный список без головного элемента. Удаление элемента из начала (а) , середины (б), конца (в) списка.

Примечание: зачеркивание элемента крестиком на схеме означает его удаление из памяти (применение Dispose(p))

```
PROCEDURE DEL_BEG_LIST ( VAR FIRST : EL);
VAR
        P : EL;
   ANSWER : STRING;
BEGIN
      IF FIRST <> NIL THEN
      BEGIN  { список не пуст }
            WRITELN ('Вы хотите удалить первый элемент?(да/нет)  ');
            READLN ( ANSWER );
            IF ANSWER = 'ДА' THEN
            BEGIN
                  P:=FIRST;
                  IF P^.NEXT = NIL THEN {в списке один элемент }
                  BEGIN
                          DISPOSE (P); {уничтожение элемента}
                          FIRST:=NIL;  {список стал пустым }
                   END
                                      ELSE
                   BEGIN
                          P := FIRST;{адрес удаляемого элемента }
                          FIRST:=FIRST^.NEXT;
                        {адрес нового первого элемента}
                         DISPOSE(P);
                        {удаление бывшего первого элемента }
                  END;
           END
       END
                              ELSE
WRITELN (' список пуст, удаление первого элемента невозможно');
END;
```

Удаление элемента из конца списка.

Нужен запрос на удаление

```
PROCEDURE DEL_END_LIST( VAR FIRST :EL);
BEGIN
      IF FIRST < > NIL THEN
      BEGIN {список не пуст}
           IF  FIST^.NEXT = NIL THEN
            BEGIN {в списке - единственный элемент }
                 P := FIRST;
                 DISPOSE (P);
                 FIRST := NIL;
            END
                                          ELSE
             BEGIN {в списке больше одного элемента }
                   Q := FIRST;
                    T := FIRST;
         {цикл поиска адреса последнего элемента}
                    WHILE Q^.NEXT < > NIL DO
                    BEGIN
                          T := Q;{запоминание адреса текущего элемента}
                          Q:=Q^.NEXT;{переход к следующему элементу}
                    END;
         {после окончания цикла T - адрес предпоследнего,
                                  а Q - адрес последнего элемента списка}
                    DISPOSE  (Q); {удаление последнего элемента}
                    T^.NEXT := NIL; {предпоследний элемент стал
                                                        последним}
            END
       END
                          ELSE
   WRITELN ('список пуст, удаление элемента невозможно  ');
END;
```

ПРИМЕЧАНИЕ. После исключения элемента из списка этот элемент может не удаляться из памяти, а через список параметров передан на какую-либо обработку, если этого требует алгоритм обработки данных.

Удаление элемента из середины списка (i-ого элемента).

```
PROCEDURE DEL_I_ELEM ( FIRST : EL; I : INTEGER);
VAR
      T, Q, R : EL;
      K ,N : INTEGER;
BEGIN
      N := COUNT_EL(FIRST); {определение числа элементов списка}
      IF (I < 1 )  OR ( I > N ) THEN
      BEGIN
           WRITELN ('i задано некорректно');
           EXIT;
       END
                                    ELSE
       BEGIN
      {нужно добавить подтверждение удаления }
              IF I = 1 THEN
               BEGIN {удаляется 1 элемент}
                   T := FIRST;
                   FIRST:= FIRST^.NEXT;
                   DISPOSE ( T);
              END
                     ELSE
               IF I = N THEN
               BEGIN { см. случай удаления последнего элемента}
                               . . .
              END
                       ELSE {удаление из «середины» списка}
              BEGIN
                   T := FIRST;
                   Q := NIL;
                   K := 1;
                   WHILE ( K < I ) DO
                    BEGIN {поиск адресов (i-1)-го и i-го элементов}
                         K := K + 1;
                               Q := T;
                        T := T^.NEXT;
                    END;
                    R := T^.NEXT;
              {найдены адреса i-го (t), (i-1)-го (q) и (i+1)-го (r) элементов }
                    Q^.NEXT := R;
                    DISPOSE ( T );    {удален i-ый элемент }
                END;
       END;
END;
```

Удаление всего списка с освобождением памяти.

```
PROCEDURE DELETE_LIST(VAR FIRST : EL);
VAR
      P, Q : EL;
      ANSWER : STRING;
BEGIN
      IF FIRST <> NIL THEN
      BEGIN  { список не пуст }
            WRITELN ( ' Вы хотите удалить весь список ? (да/нет)' );
            READLN ( ANSWER );
            IF ANSWER = 'ДА' THEN
            BEGIN
                  Q:=FIRST;
                  P:=NIL;
                  WHILE ( Q <> NIL ) DO
                  BEGIN
                        P:=Q;
                        Q:=Q^.NEXT;
                        DISPOSE(P);
                  END;
                  FIRST:=NIL;
            END;
      END
                        ELSE
      WRITELN ('список  пуст  ');
END;
```

Задачи на замену элементов в линейном однонаправленном списке без головного элемента.

Операция замены элемента в списке практически представляет собой комбинацию удаления и вставки элемента. Читателю дается возможность, используя представленные ранее графические приемы и примеры программ, самому написать процедуры замены элементов. Перед выполнением операции замены элемента желательно запрашивать у пользователя подтверждение замены.

Действуя аналогично, можно построить графические схемы и программы задач действий с двунаправленными списками.

7.9 Стеки, деки, очереди.

Одной из важных концепций в программировании является концепция стека. Стеком называется упорядоченный набор элементов, в котором добавление новых элементов и удаление существующих выполняется только с одного его конца, который называется вершиной стека. Бытовой иллюстрацией стека может служить стопка книг. Добавить к ней очередную книгу можно, положив новую поверх остальных. Верхняя книга (элемент стека) и есть вершина стека. Просматривать книги можно только по одной, снимая их с вершины. Чтобы получить книгу из середины стека, надо задать критерий отбора и удалять элементы-книги из стека до тех пор, пока не будет найдена нужная книга. Элементы из стека могут удаляться, пока он не станет пустым. Таким образом, над стеком выполняются следующие операции:

1) добавление в стек нового элемента;
2) определение пуст ли стек;
3) доступ к последнему включенному элементу, вершине стека;
4) исключение из стека последнего включенного элемента.

Отсюда ясно виден принцип работы со стеком: «пришел последним - ушел первым» (last in - first out, LIFO).

Такая структура данных очень хорошо реализуется с помощью списка. Тип списка при этом может быть выбран в соответствии с потребностями алгоритма. Например, для стека может подойти линейный однонаправленный список без головного элемента со вставкой и исключением элементов в начале списка (это и будет вершиной стека).

Другим специальным видом использования списка является очередь. Существуют различные разновидности очередей, здесь будет рассмотрена простая бесприоритетная очередь. При этом добавление элементов производится в конец очереди, а выборка и удаление элементов - из начала. Принцип доступа к очереди – «первым пришел - первым ушел» (first in - first out, FIFO). Принцип обработки как для стека, так и для очереди определяет набор соответствующих процедур. Для реализации очереди необходим список, для которого известны адрес первого и адрес последнего элементов. Таким образом, над очередью выполняются следующие операции:

1) добавление в конец очереди нового элемента;
2) определение пуста ли очередь;
3) доступ к первому элементу очереди;
4) исключение из очереди первого элемента.

Эти операции могут быть взяты из стандартного набора действий со списком.

Кроме рассмотренных очереди и стека есть ещё и третий вариант структуры данных - дек, очередь с двойным доступом, или, как ещё его

называют, - двухконечный стек. Для дека добавление элементов, доступ к «вершине» и удаление элемента возможны с обоих концов списка.

7.10 Использование рекурсии при работе со списками.

Рекурсия является одним из удобнейших средств при работе с линейными списками. Она позволяет сократить код программы и сделать алгоритмы обхода узлов деревьев и списков более понятными.

По определению понятия, рекурсивная процедура - это процедура, в теле которой есть обращение к самой себе. Для того, чтобы процесс рекурсии не стал бесконечным и не вызвал переполнение стека, в каждой рекурсивной процедуре должен быть определен ***ограничитель рекурсии***, блокирующий дальнейшее «размножение» тела процедуры.

Хотя рекурсии в списках не являются настолько очевидным решением, как в деревьях, все же они позволяют оптимизировать обработку линейных списков. В списках возможны два варианта прохода: из начала в конец и из конца в начало. Эти методы реализуются очень легко в случае с двунаправленными списками.

Рекурсию в линейных списках демонстрирует следующий пример: подсчет числа элементов в линейном однонаправленном списке.

```
PROCEDURE COUNT_EL (VAR Q: EL
                    VAR COUNT:INTEGER);
BEGIN
    IF Q< >NIL  { ограничитель рекурсии } THEN
    BEGIN
        INC(COUNT);
        COUNT_EL( Q^.NEXT , COUNT);
    END
END;
```

При входе в процедуру **count = 0** ,а **q=first (**указатель на первый элемент списка**).** Далее рекурсивная процедура работает так. Анализируется значение указателя, переданного в процедуру. Если он не равен Nil (список не закончился), то счетчик числа элементов увеличивается на 1. Далее происходит очередной вызов рекурсивной процедуры уже с адресом следующего элемента списка, а «текущая» рекурсивная процедура приостанавливается до окончания вызванной процедуры. Вызванная процедура работает точно так же: считает, вызывает процедуру и переходит в состояние ожидания. Формируется как бы последовательность из процедур, каждая из которых ожидает завершения вызванной процедуры. Этот процесс продолжается до тех пор, пока очередное значение адреса не станет равным Nil (признак окончания списка). В последней вызванной рекурсивной процедуре уже не происходит очередного вызова, так как не соблюдается условие q<>nil, срабатывает «ограничитель

рекурсии». В результате процедура завершается без выполнения каких-либо действий, а управление возвращается в «предпоследнюю», вызывающую процедуру. Точкой возврата будет оператор, стоящий за вызовом процедуры, в данном тексте - End, и «предпоследняя» процедура завершает свою работу, возвращая управление в вызвавшую её процедуру. Начинается процесс «сворачивания» цепочки ожидающих завершения процедур. Счетчик count, вызывавшийся по ссылке, сохраняет накопленное значение после завершения всей цепочки вызванных рекурсивных процедур.

Если из текста рассмотренной процедуры убрать использование счетчика, то получится некий базисный вариант рекурсивной процедуры, который можно применять при решении других задач обработки списка: распечатать содержимое списка; определить, есть ли в списке элемент с заданным порядковым номером или определенным значением информационного поля; уничтожить список с освобождением памяти и др.

7.11 Бинарные деревья.

Кроме линейных структур существуют и нелинейные, при помощи которых задаются иерархические связи данных. Для этого используются графы, а среди них сетевые и древовидные структуры. Рассмотрим один вид деревьев - бинарное дерево.

Бинарное (двоичное) дерево - это конечное множество элементов, которое либо пусто, либо содержит один элемент, называемый корнем дерева, а остальные элементы множества делятся на два непересекающихся подмножества, каждое из которых само является бинарным деревом. Эти подмножества называются правым и левым поддеревьями исходного дерева.

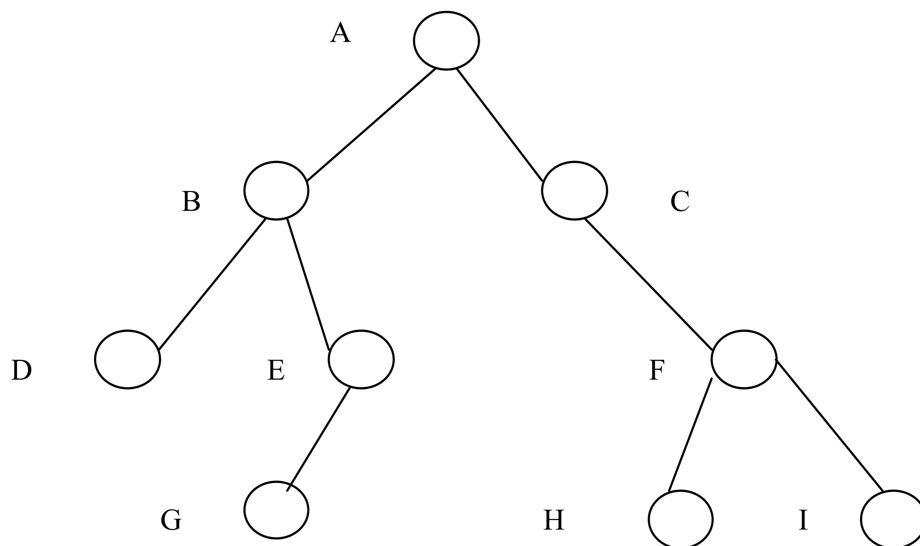


Рис. 23 Двоичное дерево

На рис. 23 показан наиболее часто встречающийся способ представления бинарного дерева. Оно состоит из девяти узлов. Корнем дерева является узел А. Левое поддерево имеет корень В, а правое поддерево - корень С. Они соединяются соответствующими ветвями, исходящими из А. Отсутствие ветви означает пустое поддерево. Например, у поддерева с корнем С нет левого поддерева, оно пусто. Пусто и правое поддерево с корнем Е. Бинарные поддеревья с корнями D, G, H и I имеют пустые левые и правые поддеревья. Узел, имеющий пустые правое и левое поддеревья, называется листом. Если каждый узел бинарного дерева, не являющийся листом, имеет непустые правое и левое поддеревья, то дерево называется строго бинарным

Уровень узла в бинарном дереве определяется следующим образом: уровень корня всегда равен нулю, а далее номера уровней при движении по дереву от корня увеличиваются на 1 по отношению к своему непосредственному предку. Глубина бинарного дерева - это максимальный уровень листа дерева, иначе говоря, длина самого длинного пути от корня к листу дерева. Узлы дерева могут быть пронумерованы по следующей схеме (см. рис. 24)

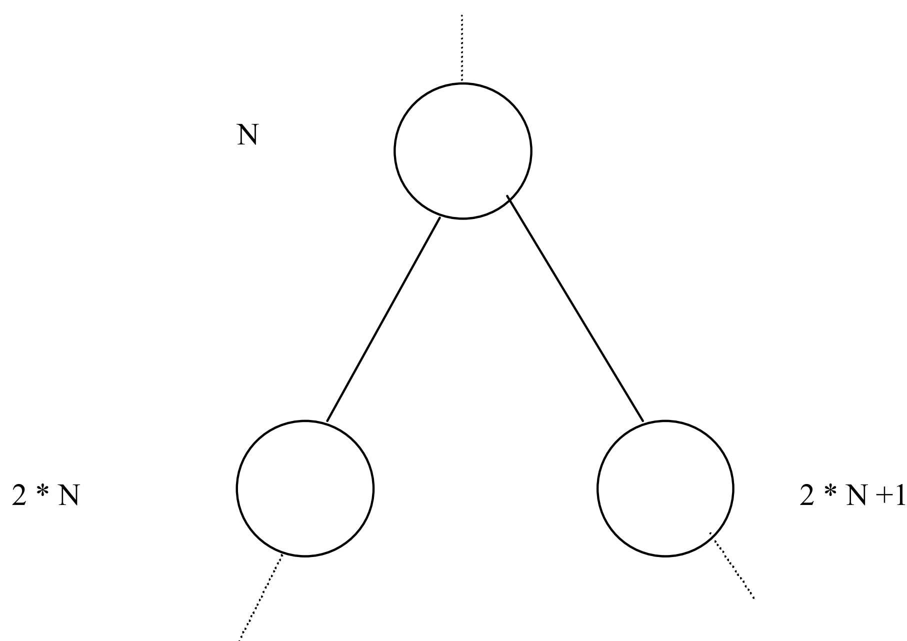


Рис. 24 Схема нумерации узлов двоичного дерева

Номер корня всегда равен 1, левый потомок получает номер 2, правый - номер 3. Левый потомок узла 2 должен получить номер 4, а правый - 5, левый потомок узла 3 получит номер 6, правый - 7 и т.д. Несуществующие узлы не нумеруются, что, однако, не нарушает указанного порядка, так как их номера не используются. При такой системе нумерации в дереве каждый узел получает уникальный номер.

Полное бинарное дерево уровня n - это дерево, в котором каждый узел уровня n является листом и каждый узел уровня меньше n имеет непустые правое и левое поддеревья.

Почти полное бинарное дерево определяется как бинарное дерево, для которого существует неотрицательное целое k такое, что:

1) каждый лист в дереве имеет уровень k или k+1;
2) если узел дерева имеет правого потомка уровня k+1, тогда все его левые потомки, являющиеся листами, также имеют уровень k+1.

7.12 Действия с бинарными деревьями.

Рассматривая действия над деревьями, можно сказать, что для построения дерева необходимо формировать узлы, и, определив предварительно место включения, включать их в дерево. Количество узлов определяется необходимостью. Алгоритм включения должен быть известен и постоянен. Узлы дерева могут быть использованы для хранения какой-либо информации.

Далее необходимо осуществлять поиск заданного узла в дереве. Это можно организовать, например, последовательно обходя узлы дерева, причем каждый узел должен быть просмотрен только один раз.

Может возникнуть задача и уничтожения дерева в тот момент, когда необходимость в нем (в информации, записанной в его элементах) отпадает. В ряде случаев может потребоваться уничтожение поддерева.

Для того, чтобы совокупность узлов образовала дерево, необходимо каким-то образом формировать и использовать связи узлов со своими предками и потомками. Все это очень напоминает действия над элементами списка.

Построение бинарного дерева.

Важным понятием древовидной структуры является понятие двоичного дерева поиска. Правило построения двоичного дерева поиска: элементы, у которых значение некоторого признака меньше, чем у корня, всегда включаются слева от некоторого поддерева, а элементы со значениями, большими, чем у корня - справа. Этот принцип используется и при формировании двоичного дерева, и при поиске в нем элементов. Таким образом, при поиске элемента с некоторым значением признака происходит спуск по дереву, начиная от корня, причем выбор ветви следующего шага - направо или налево согласно значению искомого признака - происходит в каждом очередном узле на этом пути. При поиске элемента результатом будет либо найденный узел с заданным значением признака, либо поиск закончится листом с «нулевой» ссылкой, а требуемый элемент отсутствует на проделанном по дереву пути. Если поиск был проделан для включения очередного узла в дерево, то в ре-

зультате будет найден узел с пустой ссылкой (пустыми ссылками), к которому справа или слева в соответствии со значением признака и будет присоединен новый узел.

Рассмотрим пример формирования двоичного дерева. Предположим, что нужно сформировать двоичное дерево, узлы (элементы) которого имеют следующие значения признака: 20, 10, 35, 15, 17, 27, 24, 8, 30. В этом же порядке они и будут поступать для включения в двоичное дерево. Первым узлом в дереве (корнем) станет узел со значением 20. Обратить внимание: поиск места подключения очередного элемента всегда начинается с корня. К корню слева подключается элемент 10. К корню справа подключается элемент 35. Далее элемент 15 подключается справа к 10, проходя путь: корень 20 - налево - элемент 10 - направо - подключение, так как дальше пути нет. Процесс продолжается до исчерпания включаемых элементов. Результат представлен на рис. 25.

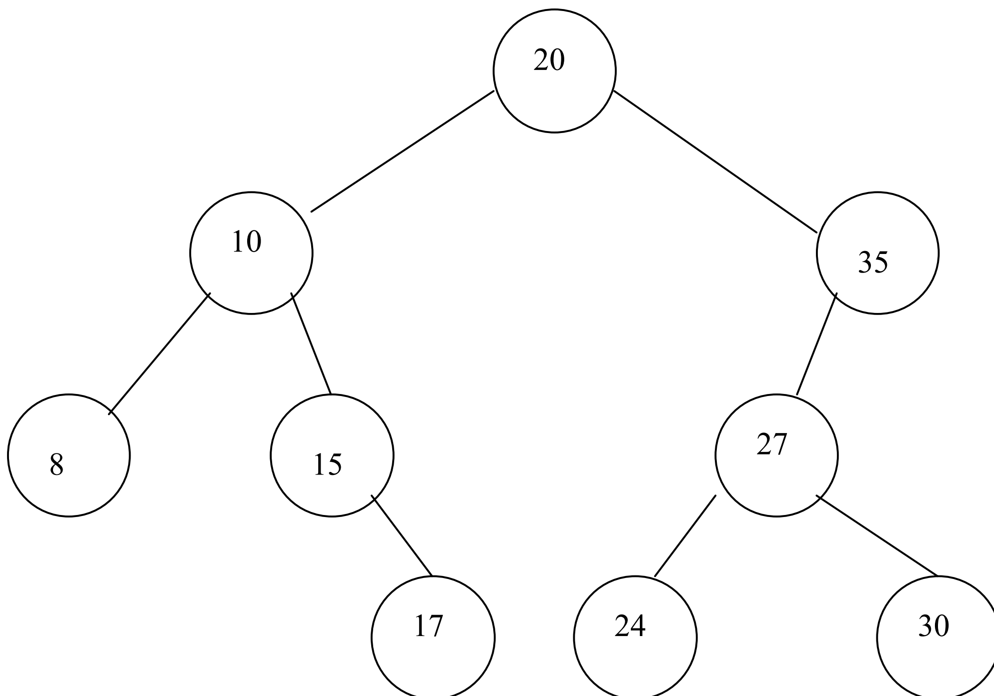


Рис. 25 Построение бинарного дерева.

Значения элементов дерева: 20, 10, 35, 15, 17, 27, 24, 8, 30

При создании двоичного дерева отдельно решается вопрос с возможностью включения элементов с дублирующими признаками. Алгоритм двоичного поиска позволяет при включении корректно расположить узлы в дереве, однако при поиске уже включенных элементов возникают сложности, так как при стандартном варианте поиска будет выбираться только один из дублей. Для обнаружения всех дублей должен быть применен алгоритм такого обхода дерева, при котором каждый узел дерева будет выбран один раз.

7.13 Решение задач работы с бинарным деревом.

Элемент дерева используется для хранения какой-либо информации, следовательно, он должен содержать информационные поля, возможно разнотипные. Элемент двоичного дерева связан в общем случае с двумя прямыми потомками, а при необходимости может быть добавлена и третья связь - с непосредственным предком. Отсюда следует, что по структуре элемент дерева (узел) похож на элемент списка и может быть описан так же. Как и в списке, в дереве должна существовать возможность доступа к его «первому» элементу - корню дерева. Она реализуется через необходимую принадлежность дерева - поле ROOT, в котором записывается ссылка на корневой элемент.

Приведем пример описания полей и элементов, необходимых для построения дерева.

```
TYPE
    ND = ^ NODE;
    NODE = RECORD
            INF1 : INTEGER;
            INF2 :  STRING ;
            LEFT :  ND;
            RIGHT : ND;
        END;
VAR
    ROOT, P,Q : ND;
```

Приведенный пример описания показывает, что описание элемента списка и узла дерева по сути ничем не отличаются друг от друга. Различия в технологии действий тоже невелики - основные действия выполняются над ссылками, адресами узлов. Основные различия - в алгоритмах.

При работе с двоичным деревом возможны следующие основные задачи:

1) создание элемента, узла дерева,
2) включение его в дерево по алгоритму двоичного поиска,
3) нахождение в дереве узла с заданным значением ключевого признака,
4) определение максимальной глубины дерева,
5) определение количества узлов дерева,
6) определение количества листьев дерева,
7) ряд других задач.

Приведем примеры процедур, реализующих основные задачи работы с бинарным деревом.

```
{создание элемента дерева}
PROCEDURE CREATE_EL_T(VAR Q:ND; NF1:INTEGER;
                                        INF2:STRING);
BEGIN
   NEW(Q);
      Q^.INF1:=INF1;
      Q^.INF2:=INF2;
      {значения полей передаются в качестве параметров}
      Q^.RIGHT:=NIL;
      Q^.LEFT:=NIL;
END;

PROCEDURE INSERT_EL ( P : ND; {адрес включаемого элемента}
                                VAR ROOT : ND);
VAR
   Q, T : ND;
BEGIN
     IF ROOT = NIL THEN
        ROOT := P {элемент стал корнем}
                   ELSE
        BEGIN { поиск по дереву }
             T := ROOT;
             Q := ROOT;
             WHILE ( T < > NIL ) DO
             BEGIN
                  IF P^.INF1 < T^.INF1  THEN
               BEGIN
                      Q := T;{ запоминание текущего адреса}
                       T := T^.LEFT; {уход по левой ветви}
                  END
                                          ELSE
               IF P^.INF1 > T^.INF1 THEN
               BEGIN
                       Q := T;{ запоминание текущего адреса}
                       T := T^.RIGHT; {уход по правой ветви}
                  END
                                          ELSE
               BEGIN
                  WRITELN ('найден дубль включаемого элемента');
                   EXIT; {завершение работы процедуры}
                  END
             END;
            {после выхода из цикла в q - адрес элемента, к которому
            должен быть подключен новый элемент}
             IF P^.INF1 < Q^.INF1  THEN
                 Q^.LEFT := P {подключение слева }
                                   ELSE
                 Q^.RIGHT := P; {подключение справа}
        END;
```

ПРИМЕЧАНИЕ: элемент с дублирующим ключевым признаком в дерево не включается.

Данный алгоритм движения по дереву может быть положен в основу задачи определения максимального уровня (глубины) двоичного дерева, определения, есть ли в дереве элемент с заданным значением ключевого признака и т.д., то есть таких задач, решение которых основывается на алгоритме двоичного поиска по дереву.

Однако не все задачи могут быть решены с применением двоичного поиска, например, подсчет общего числа узлов дерева. Для этого требуется алгоритм, позволяющий однократно посещать каждый узел дерева.

При посещении любого узла возможно однократное выполнение следующих трех действий:

1) обработать узел (конкретный набор действий при этом не важен). Обозначим это действие через О (обработка);
2) перейти по левой ссылке (обозначение - Л);
3) перейти по правой ссылке (обозначение - П).

Можно организовать обход узлов двоичного дерева, однократно выполняя над каждым узлом эту последовательность действий. Действия могут быть скомбинированы в произвольном порядке, но он должен быть постоянным в конкретной задаче обхода дерева.

На примере дерева на рис. 10 проиллюстрируем варианты обхода дерева.

1) Обход вида ОЛП. Такой обход называется «в прямом порядке», «в глубину». Он даст следующий порядок посещения узлов:
   20, 10, 8, 15, 17, 35, 27, 24, 30
2) Обход вида ЛОП. Он называется «симметричным» и даст следующий порядок посещения узлов:
   8, 10, 15, 17, 20, 24, 27, 30, 35
3) Обход вида ЛПО. Он называется «в обратном порядке» и даст следующий порядок посещения узлов:
   8, 17, 15, 10, 24, 30, 27, 35, 20

Если рассматривать задачи, требующие сплошного обхода дерева, то для части из них порядок обхода, в целом, не важен, например, подсчет числа узлов дерева, числа листьев/не листьев, элементов, обладающих заданной информацией и т.д. Однако такая задача, как уничтожение бинарного дерева с освобождением памяти, требует использования только обхода «в обратном порядке».

Рассмотрим средства, с помощью которых можно обеспечить варианты обхода дерева.

При работе с бинарным деревом с точки зрения программирования оптимальным вариантом построения программы является использование рекурсии. Базисный вариант рекурсивной процедуры обхода бинарного дерева очень прост.

```
{ обход дерева по варианту ЛОП }
PROCEDURE RECURS_TREE ( Q : ND );
BEGIN
        IF Q <> NIL THEN
         BEGIN
                RECURS_TREE( Q^.LEFT );{уход по левой ветви-Л}
                WORK ( Q ); { процедура обработки дерева-О}
                RECURS_TREE( Q^.RIGHT );{уход по правой ветви-П}
         END;
END;
```

Рекурсия в этой программе действует точно так же, как и в рекурсивных процедурах работы со списками: создается цепочка процедур, каждая из которых рекурсивно обращается к себе и затем ожидает завершения вызванной процедуры. Потенциально бесконечный процесс рекурсивного вызова останавливается с помощью «ограничителя рекурсии», в данном случае им становится нарушение условия (q <> nil), когда при обходе обнаруживается «нулевая» ссылка вместо реального адреса. При этом начинается последовательное завершение вызванных процедур с возвратом управления в вызывающую. Способ обхода меняется с изменением порядка обращений к процедурам.

Для практической проработки действия механизма рекурсии при реализации вариантов обхода дерева можно воспользоваться уже построенным деревом с рис.10.

Пример использования рекурсивной процедуры при решении задачи подсчета листьев двоичного дерева.

```
PROCEDURE LEAFS_COUNT( Q : ND; VAR K : INTEGER );
BEGIN
      IF Q <> NIL THEN
      BEGIN
            LEAFS_COUNT( Q^.LEFT, K );
            IF (Q^.LEFT = NIL) AND (Q^.RIGHT = NIL) THEN
                  K := K +1;
            LEAFS_COUNT( Q^.RIGHT, K );
```

```
    END;
END;

{удаление дерева с освобождением памяти}
PROCEDURE DEL_TREE(Q : ND );
BEGIN
    IF Q<>NIL THEN
    BEGIN
        DEL_TREE (Q^.LEFT);
        DEL_TREE (Q^.RIGHT);
        DISPOSE(Q)
    END
END;
```

В заключение следует сказать о том, что рекурсивный обход дерева применим в большинстве задач, однако необходимо все же различать варианты эффективного применения двоичного поиска и сплошного обхода.

Вопросы к главе 7.

1. Особенности использования статической и динамической памяти.
2. Описание динамических переменных.
3. Использование указателей и ссылочных переменных.
4. Основные процедуры и функции для выделения и освобождения памяти на логическом уровне.
5. Основные процедуры и функции для выделения и освобождения памяти на физическом уровне.
6. Особенности использования динамических переменных.
7. Особенности создания и обработки очередей.
8. Особенности создания и обработки стеков и деков.
9. Особенности создания и обработки однонаправленных списков.
10. Особенности создания и обработки двунаправленных списков.
11. Особенности создания и обработки кольцевых списков.
12. Особенности создания и обработки списков с головными элементами.
13. Особенности создания и обработки мультисписков.
14. Использование рекурсии при работе со списками.
15. Понятия дерева, двоичного дерева поиска.
16. Нерекурсивные способы создания и обработки двоичных деревьев.
17. Рекурсивные способы создания и обработки двоичных деревьев.

8. Основные принципы структурного программирования.
8.1 Понятие жизненного цикла программного продукта

Программное изделие проходит в своем развитии целый ряд этапов, начиная от возникновения потребности в программном продукте и заканчивая снятием программы с эксплуатации. Рассмотрение полного жизненного цикла программного продукта в данном пособии не является необходимым, поэтому на рисунке приведены не все, а только основные этапы жизненного цикла программного изделия

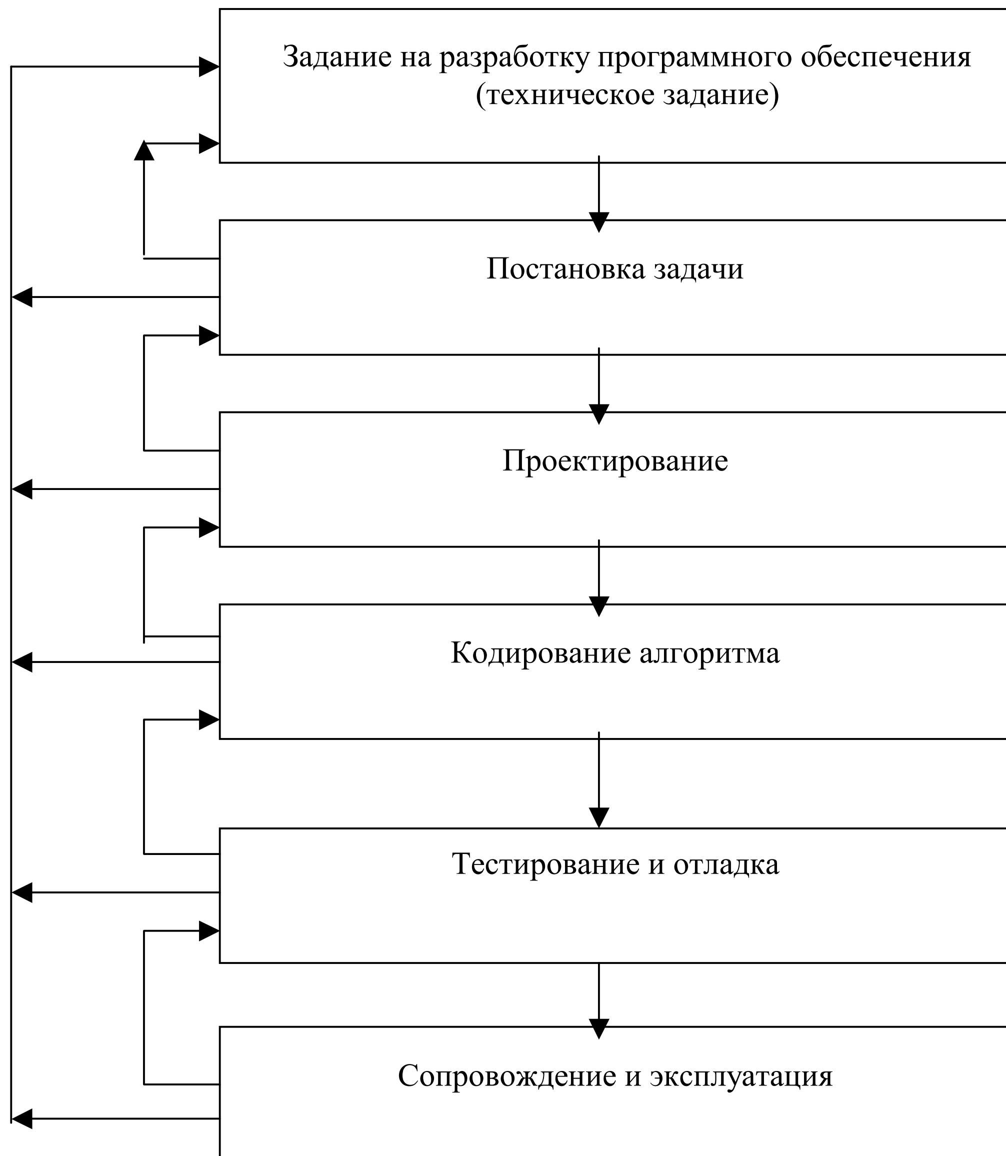


Рис. 26 Основные этапы разработки программного обеспечения.

Приведенные этапы являются главными при разработке программ и программных комплексов. В зависимости от величины разрабатываемого программного комплекса роль каждого этапа и объём работ по нему будут различными.

В настоящее время производство программ поставлено на промышленный уровень, поэтому значительную роль при этом играет использование такой технологии программирования, которая обеспечила бы создание высококачественного программного продукта.

Технология программирования - это система методов, способов и приемов обработки и выдачи информации. Одной из распространенных методик создания программной продукции в настоящее время является структурное программирование.

Цели структурного программирования:

1) повысить надежность программ; для этого нужно, чтобы программа легко поддавалась тестированию и не создавала проблем при отладке. Достигается это хорошим структурированием программы при ее проектировании;
2) повысить эффективность программ; она может быть достигнута при структурировании программы, при разбиении ее на модули так, чтобы можно было бы легко находить и корректировать ошибки, а также чтобы текст любого модуля с целью повышения эффективности его работы можно было переделать независимо от других;
3) уменьшить время и стоимость программной разработки. Достижимо при повышении производительности труда программиста;
4) улучшить читабельность программ; это значит, что необходимо избегать использования языковых конструкций с неочевидной семантикой, стремиться к локализации действия управляющих конструкций и использования структур данных, разрабатывать программу так, чтобы ее можно было бы читать от начала до конца без управляющих переходов на другую страницу;

8.2 Основные принципы структурной методологии.

Принцип абстракции.

Этот принцип позволяет разработчику рассматривать программу в нужный момент без лишней детализации. Детализация увеличивается при переходе от верхнего уровня абстракции к нижнему.

Принцип формальности.

Он предполагает строгий методический подход к программированию, придает творческому процессу определенную строгость и дисциплину.

Принцип модульности.

В соответствии с этим принципом программа разделяется на отдельные законченные фрагменты, модули, которые просты по управлению и допускают независимую отладку и тестирование. В результате отдельные ветви программы могут создаваться разными группами программистов.

Принцип иерархического упорядочения.

Взаимосвязь между частями программы должна носить иерархический, подчиненный характер. Это, кстати, следует и из принципа нисходящего проектирования.

8.3 Нисходящее проектирование.

Нисходящее проектирование строится на вышеперечисленных принципах. При нисходящем проектировании происходит анализ задачи с целью определения возможности разбиения ее на ряд подзадач. Затем каждая из полученных подзадач также анализируется для возможного разбиения на подзадачи. Процесс для очередной подзадачи заканчивается, когда подзадачу невозможно или нецелесообразно разбивать на подзадачи далее. Результат этот процесса, зафиксированный в графической форме, является основой для построения структурной схемы программы, которая показывает, во-первых, что делает вся программа в целом и ее отдельные части, а, во-вторых, отображает взаимосвязь подзадач друг с другом.

На основе структурной схемы программы выполняется реализация подзадач в виде отдельных модулей.

После разбиения программного комплекса на программные модули и подготовки спецификаций на каждый программный модуль начинается работа по проектированию алгоритмов, реализующих спецификацию каждого модуля.

8.4 Структурное кодирование.

Структурное кодирование - это метод кодирования (программирования), предусматривающий создание понятных, простых и удобочитаемых программных модулей и программных комплексов на требуемых языках программирования.

Для кодирования программных модулей используются унифицированные (базовые) структуры. Доказано, что любая программа может быть составлена с применением только трёх канонических структур.
Программные комплексы и программные модули, закодированные в соответствии с правилами структурного программирования, называются структурированными.

8.5 Модульное программирование.

Модульное программирование - это организация программы как совокупности небольших независимых блоков, модулей, структура и поведение которых подчиняется определенным правилам. Следует заметить, что понятие «модуль» не совпадает в данном случае с понятием «модуль» (unit в смысле «библиотека») языка Паскаль. Это должна быть

простая, замкнутая (независимая) программная единица (процедура или функция), обозримая, реализующая только одну функцию. Для написания одного модуля должно быть достаточно минимальных знаний о тексте других, как вызывающих, так и вызываемых.

Программа, разработанная в соответствии с принципами структурного программирования, должна удовлетворять следующим требованиям:

- программа должна разделяться на независимые части, называемые модулями;

- модуль - это независимый блок, код (текст) которого физически и логически отделен от кода других модулей;

- модуль выполняет только одну логическую функцию, иначе говоря, должен решать самостоятельную задачу своего уровня по принципу: один программный модуль - одна функция;

- работа программного модуля не должна зависеть:
  - ⇒ от входных данных;
  - ⇒ от того, какому программному модулю предназначены его выходные данные;
  - ⇒ от предыстории вызовов программного модуля;

- размер программного модуля желательно ограничивать одной-двумя страницами исходного листинга (50-100 строк исходного кода);

- модуль должен иметь только одну входную и одну выходную точку;

- взаимосвязи между модулями устанавливаются по иерархической структуре;

- каждый модуль должен начинаться с комментария, объясняющего его назначение, назначение переменных, передаваемых в модуль и из него, модулей, которые его вызывают, и модулей, которые вызываются из него;

- при создании модуля можно использовать только стандартные управляющие конструкции: выбор, цикл, блок (последовательность операторов);

- оператор безусловного перехода или вообще не используется в модуле, или применяется в исключительных случаях только для перехода на выходную точку модуля;

- в тексте модуля необходимо использовать комментарии, в особенности в сложных местах алгоритма;

- идентификаторы переменных и модулей должны быть смысловыми, «говорящими»;

- в одной строке стоит записывать не более одного оператора. Если для записи оператора требуется больше, чем одна строка, то все последующие операторы записываются с отступами;

- желательно не допускать вложенности операторов IF более, чем трех уровней;

- следует избегать использования языковых конструкций с неочевидной семантикой и программистских «трюков».

В заключение следует напомнить, что все эти вместе взятые меры направлены на повышение качества разрабатываемого программного обеспечения.

Вопросы к главе 8.

1. Понятие жизненного цикла программного продукта.
2. Основные этапы разработки программного обеспечения.
3. Дать определение технологии программирования.
4. Цели структурного программирования.
5. Основные принципы структурной методологии.
6. Использование нисходящего проектирования.
7. Дать определение структурному кодированию.
8. Принципы структурного кодирования.
9. Особенности модульного программирования.

9. Список литературы

1. Вирт Н. Алгоритмы и структуры данных. М., Мир, 1989.
2. Вирт Н. Алгоритмы + структуры данных = программы. М., Мир,1985.
3. Дайитбегов Д.М., Черноусов Е.А. Основы алгоритмизации и алгоритмические языки. Учебник. М., Финансы и статистика, 1992.
4. Джонс Ж.,Харроу К. Решение задач в системе Турбо Паскаль. М., Финансы и статистика, 1989.
5. Епанешников А.М., Епанешников В.А. Программирование в среде Turbo Pascal 7.0. М., Диалог-МИФИ, 1995.
6. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М.. МЦНМО, 1999.
7. Лэнгсам Й., Огенстайн М., Тэненбаум А Структуры данных для персональных ЭВМ, М., Мир, 1989.
8. Семашко Г.Л., Салтыков А.И. Программирование на языке Паскаль. М., Наука, 1988
9. Турбо Паскаль 7.0. К., Торгово-издательское бюро BHV, 1996
10. Фаронов В.В. Turbo Pascal 7.0. Начальный курс. Учебное пособие. М., Нолидж,1998.