**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**Учреждение образования “Витебский государственный технологический университет”**



# ИНФОРМАТИКА

## Программирование на алгоритмическом языке

**Конспект лекций для студентов специальностей 1-50-01-02 «Конструирование и технология швейных изделий», 1-50-02-01 «Конструирование и технология изделий из кожи», 1-54 01 01-04 «Метрология, стандартизация и сертификация (лёгкая промышленность)»**

**ВИТЕБСК**
**2011**

# ИНФОРМАТИКА

**Программирование на алгоритмическом языке**

**Конспект лекций**

УДК **004**
ББК **32.81**
**К 14**

Рецензент:
кандидат технических наук, доцент, декан экономического факультета УО «Витебский государственный технологический университет» Шарстнёв Владимир Леонидович

Рекомендовано к изданию редакционно-издательским советом УО «ВГТУ», протокол № 5 от 7 сентября 2010 г.

**К 14 Казаков, В. Е.**
Информатика. Программирование на алгоритмическом языке : конспект лекций / В. Е. Казаков. – Витебск : УО «ВГТУ», 2010.– 93 с.

**ISBN 978 - 985 - 481 - 221 - 2**

Конспект лекций содержит необходимый теоретический материал по дисциплине «Информатика», изложенный в краткой форме. Издание подготовлено в соответствии с типовой Программой по информатике для специальностей 1-50 01 02 «Конструирование и технология швейных изделий», 1-50 02 01 «Конструирование и технология изделий из кожи», 1-54 01 01-04 «Метрология, стандартизация и сертификация (лёгкая промышленность)», учебной программой данной дисциплины и может быть использовано студентами дневной и заочной форм обучения.

УДК **004**
ББК **32.81**

**ISBN 978 - 985 - 481 - 221 - 2**

СОДЕРЖАНИЕ

# Глава 1. Основы алгоритмического языка

## Алфавит языка

В языке Pascal используется набор символов кодовой таблицы ASCII. Язык Pascal является регистронезависимым, т. е. между большими и малыми буквами нет разницы.

## Символы и константы

Программа, написанная на Паскале, состоит из лексем.

**Лексемами** (словами) называются минимальные значимые единицы текста в программе, написанной на Паскале. Лексемы представлены такими категориями, как:

- специальные символы и зарезервированные слова,
- идентификаторы,
- неименованные константы-литералы.

Две соседних лексемы, если они обе представляют собой зарезервированное слово, идентификатор, числовую константу-литерал, должны быть отделены друг от друга одним или несколькими **разделителями,** причем разделитель представляет собой пробел или комментарий.

Разделители не могут быть частью лексем.

## Специальные символы и зарезервированные слова

**Специальные символы и зарезервированные слова** представляют собой символы или последовательности символов, имеющие одно или несколько фиксированных значений.

Примеры зарезервированных слов и специальных символов: var, for, do, program, :=, <>, >=, +, *, type, ;.

## Идентификаторы

**Идентификаторы** выступают в качестве имен констант, типов, переменных, процедур, функций, модулей, программ и полей в записях.

Правила составления идентификаторов:

- Идентификатор должен начинаться с буквы.
- После первого символа идентификатора можно использовать буквы, цифры и символы подчеркивания (значение ASCII $5F).
- Идентификатор может иметь любую длину, однако только первые 63 символа являются значимыми.

Примеры идентификаторов: x1, var_1, zyx, a_1_12, My_name.

Неправильные идентификаторы: 1x, sum$, My name, 1_var, x-1.

## Неименованные константы-литералы

**Неименованные константы** представляют собой конкретные значения (числовые, символьные или строковые), вводимые в состав выражений и операторов языка программирования Pascal.

### Числовые неименованные константы-литералы

Десятичные числа обозначают константы целого типа. Они должны принимать значения в диапазоне от -2147483648 до 2147483647.

Примеры: 23, -3555.

Целая константа в шестнадцатеричном формате имеет в качестве префикса знак доллара $. Шестнадцатеричные числа обозначают константы целочисленного типа, они должны находиться в диапазоне от $00000000 до $FFFFFFFF.

Примеры: $22, $A12B.

Константы вещественного типа в формате с плавающей запятой представляют собой числа с десятичными точками (.) в качестве разделителя целой и дробной части.

Примеры: 23.0034, -23.0.

Константы вещественного типа в формате с фиксированной запятой используют техническое обозначение E или e, которое читается как "на десять в степени". Например, 7E-2 означает $7 \cdot 10^{-2}$; 12.25E+6 или 12.25E6 оба обозначают $12.25 \cdot 10^{+6}$.

Примеры: 7E-2, 12.25E6, -12.3e+02.

### Строковые неименованные константы-литералы

Строка символов представляет собой последовательность, содержащую нуль и более символов из расширенного набора символов кода ASCII, записанную в одной строке программы и заключенную в одиночные кавычки (апострофы "’"). Строка символов, ничего не содержащая между апострофами, называется нулевой строкой. Два последовательных апострофа в строке символов обозначают один символ апостроф.

Примеры: 'TURBO', 'A = 12.25E6'.

## Комментарии

Комментарии представляют собой произвольный текст, заключённый в фигурные скобки "{}" или в составные скобки, состоящие из звездочки и круглой скобки, который игнорируется компилятором.

Примеры: {любой текст, не содержащий правую фигурную скобку},

(* любой текст, не содержащий звездочку/правую круглую скобку *).

## Блоки

**Блок** – программная единица, обладающая собственным набором описаний и последовательностью операторов, реализующих законченный алгоритм.

Блоками в языке Pascal являются:

- процедуры;
- функции;
- программы;
- модули.

При составлении программы программист использует различные объекты, каждый из которых имеет уникальный идентификатор (имя).

Объектами в наших лекциях будем называть: константы, переменные, типы, процедуры, функции. Далее мы рассмотрим сущность и способы описания каждого из них.

В программе, разрабатываемой в интегрированной среде Borland Pascal, уже имеется набор так называемых стандартных объектов, имеющих определённые идентификаторы (например: тип integer, процедура write, функция sqrt). Если программисту недостаточно стандартных объектов для решения задачи, то он может создать собственные, так называемые пользовательские объекты.

Конструкция, которая определяет идентификатор пользовательского объекта и описывает его параметры, называется **описанием.**

Структура блока:

**<Заголовок блока>**
**<Раздел описаний>**
**Begin**
**<раздел операторов>**
**End**

Все идентификаторы и метки, описанные в разделе описания, являются для блока локальными, т. е. их можно использовать в разделе операторов данного блока.

### Блок программы

**Блок программы** – синтаксическая конструкция, которая преобразуется компилятором в исполняемый файл (файл с расширением .exe).

**Program Ид_прог; { Заголовок блока}**
**{----Раздел описаний:----}**
[**uses** <список подключаемых модулей>;]

[**label** <описания меток>; ]
[**const** <описания констант>;]
[**type** <описания типов>;]
[**var** <описания переменных>;]
[<описания подпрограмм>]
**{----Раздел операторов:----}**
**begin**

**end.**

Объекты, описываемые в каждом из разделов блока программы, рассмотрим в нижеследующих пунктах.

## Подключаемые модули

**Модуль** представляет собой дополнительный набор описаний типов, переменных, констант, процедур и функций, которые расширяют возможности подключающей модуль программы.

**Синтаксис раздела uses:**

**Ид_модуль_1, Ид_модуль_2,... Ид_модуль_N;**

Пример:
uses SysUtils, Windows;

## Типы

**Тип данных** – характеристика объектов, использующихся для хранения данных (переменные, типизированные константы, результаты функций), которая определяет:

- диапазон возможных значений данных из набора;
- допустимые операции, которые можно выполнять над этими значениями;
- способ хранения этих значений в памяти.

По количеству содержащихся значений типы можно поделить на:
**- скалярные** (позволяют хранить только одно значение);
**- структурные** (позволяют хранить множество значений, структурированных определённым образом).
Среди типов по принципу описания можно выделить:
**- стандартные, или встроенные** (это уже описанные, имеющие собственный идентификатор типы, которые можно использовать при разработке программы);

**- пользовательские** (типы, которые конструирует непосредственно разработчик программы).

Пользовательские типы описываются в разделе type.

**Синтаксис раздела type:**

**Ид_типа1 = описание_типа1; Ид_типа2 = описание_типа2;... Ид_типаN = описание_типаN;**

где **описание_типа** – специальная конструкция, описывающая параметры определённого типа (для каждого типа используется собственная конструкция).



### Скалярные порядковые типы

Все возможные значения порядкового (дискретного) типа представляют собой упорядоченное конечное множество, и каждое возможное значение связано с порядковым номером, который представляет собой целое число.

### Встроенные целочисленные типы

В языке TP имеется несколько стандартных целочисленных типов, отличающихся друг от друга величиной диапазона (таблица 1).

Таблица 1 – Стандартные целочисленные типы

| Стандартный идентификатор | Диапазон | Размер занимаемой области памяти, бит |
|---|---|---|
| **byte** | 0 .. 255 | 8 |
| **word** | 0 .. 65535 | 16 |
| **shortint** | -128 .. 127 | 8 |
| **integer** | -32768 .. 32767 | 16 |
| **longint** | -2147483648 .. 2147483647 | 32 |

### Встроенный булевский тип

Объекты булевского (логического) типа могут принимать одно из двух значений, описанных встроенными идентификаторами констант False (ложно, 0) и True (истинно, 1).

Для описания объектов логического типа имеется стандартный идентификатор **boolean.**

### Встроенный символьный тип

Множеством значений этого типа являются символы, упорядоченные в соответствии с кодами расширенной кодовой таблицы ASCII.

Для описания объектов символьного типа имеется стандартный идентификатор **char.**

### Пользовательский перечисляемый тип

Перечисляемые типы определяют упорядоченные множества значений через перечисление идентификаторов констант, которые обозначают эти значения.

**Синтаксис описания перечисляемого типа (раздел type):**
**(Ид_константы1, Ид_константы2, ... Ид_константыN)**

Замечания:

- При указании идентификатора в списке перечисляемого типа он описывается как константа для блока, в котором указано описание перечисляемого типа. Типом этой константы является описанный перечисляемый тип.
- Порядковый номер перечисляемой константы определяется ее позицией в списке идентификаторов при описании. Первая перечисляемая константа в списке имеет порядковый номер 0.

Пример описания перечисляемого типа (раздел type):

```
type
 DnNed = (Pn, Wt, Sr, Ht, Pt, Sb, Ws);
```

### Пользовательский тип «отрезок»

Отрезок типа представляет собой диапазон значений одного из порядковых типов, называемого главным типом.

**Синтаксис описания типа «отрезок» (раздел type):**
**Константа_А..Константа_В**

Замечания:

- Обе константы должны иметь один и тот же порядковый тип.
- Объект типа «отрезок» имеет все свойства главного типа, однако его значение на этапе выполнения программы должно принадлежать указанному в описании интервалу.

Пример описания типа «отрезок»:

```
ChMes = 1..31;
```

## Вещественные типы

Если основной характеристикой целочисленного типа является диапазон значений, то для вещественного типа основными характеристиками являются **диапазон представления** и **точность** (Рисунок 1).

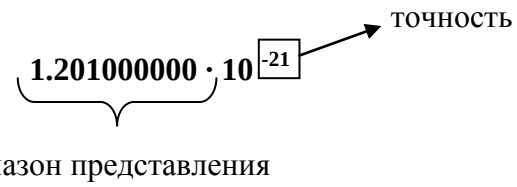



Рисунок 1 – Характеристики вещественного типа

**Диапазон представления** – количество знаков, выделяемое для представления значимых разрядов числа.

**Точность** – диапазон возможных значений степени числа 10.

В языке ТР имеется несколько стандартных вещественных типов, отличающихся друг от друга величиной точности (таблица 2).

Таблица 2 – Стандартные вещественные типы

| Стандартный идентификатор | Точность | Диапазон представления (цифр) | Размер занимаемой области памяти |
|---|---|---|---|
| Real | $2.9e^{-39}$ .. $1.7e^{38}$ | 11-12 | 6 |
| Single (с одинарной точностью) | $1.5e^{-45}$ .. $3.4e^{38}$ | 7-8 | 4 |
| Double (с двойной точностью) | $5.0e^{-324}$ .. $1.7^{308}$ | 15-16 | 8 |
| Extended (с повышенной точностью) | $1.9e^{-4951}$ .. $1.1^{4932}$ | 19-20 | 10 |

## Переменные

**Переменная** представляет собой поименованную область памяти ЭВМ определённого размера, используемую для хранения данных определённого типа.

Переменные, описанные в основном блоке (программе), называются **глобальными**. Переменные, описанные во вложенном блоке (процедуре или функции), называются **локальными.**

Примечание: размер памяти, занимаемый всеми объявленными переменными, не должен превышать 65520 байт.

**Синтаксис описания переменных (раздел var):**
**Ид_переменной1, Ид_переменной2...: описание_типа|Ид_типа; ...**

Пример:

```
var
  X,Y,Z: real;
  I,J,K: integer;
  Digit: 0..9;
```

## Выражения

**Выражение** – конструкция, предназначенная для выполнения вычислений. Выражение состоит из последовательности операций; выполнение этой последовательности операций приводит к появлению единственного значения, имеющего определённый тип.

## Операции

**Операция** – определённое действие, производимое над операндами, в результате выполнения которого появляется единственное значение, имеющее определённый тип.

Операции в языке программирования Pascal характеризуются:
– количеством операндов;
– типами операндов и результата;
– приоритетом.

**Унарными** называются операции, производимые над одним операндом.

**Синтаксис использования унарной операции:**
**Знак_операции Операнд**

Пример:
- 3.44, not false.

**Бинарными** называют операции, производимые над двумя операндами.

**Синтаксис использования бинарной операции:**
**Операнд1 Знак_операции Операнд2**

Пример:
3 + 55, x1 > 2, 5 mod 2.

**Приоритет** – целое число от 1 (высший приоритет) до 4 (низший приоритет), определяющее очерёдность выполнения операции в выражении.

Операции с высоким приоритетом выполняются раньше, чем операции, имеющие более низкий приоритет. Операции с одинаковым приоритетом при вычислении выражения выполняются в порядке слева-направо.

Выражение, заключённое в круглые скобки, имеет наивысший приоритет, т. е. вычисляется в первую очередь.

В таблице 3 представлены некоторые операции языка.

Таблица 3 – Операции языка

<table>
<tr><th rowspan="2">Операция</th><th colspan="2">Типы операндов</th><th rowspan="2">Тип результата</th><th rowspan="2">Приоритет</th><th rowspan="2">Пояснения</th></tr>
<tr><th>1</th><th>2</th></tr>
<tr><td rowspan="3">+</td><td>ц</td><td>ц</td><td>ц</td><td rowspan="3">3</td><td></td></tr>
<tr><td>в</td><td>в</td><td>в</td><td></td></tr>
<tr><td>с</td><td>с</td><td>с</td><td>объединение строк<br>(’стр’+’.1’ = ’стр.1’)</td></tr>
<tr><td rowspan="2">-</td><td>ц</td><td>ц</td><td>ц</td><td rowspan="2">3</td><td></td></tr>
<tr><td>в</td><td>в</td><td>в</td><td></td></tr>
<tr><td rowspan="2">*</td><td>ц</td><td>ц</td><td>ц</td><td rowspan="2">2</td><td></td></tr>
<tr><td>в</td><td>в</td><td>в</td><td></td></tr>
<tr><td>/</td><td>в</td><td>в</td><td>в</td><td>2</td><td></td></tr>
<tr><td>div</td><td>ц</td><td>ц</td><td>ц</td><td>2</td><td>деление нацело (5 div 2 = 2)</td></tr>
<tr><td>mod</td><td>ц</td><td>ц</td><td>ц</td><td>2</td><td>остаток от деления нацело<br>(5 mod 2 = 1)</td></tr>
<tr><td><></td><td colspan="2">*</td><td>л</td><td>4</td><td></td></tr>
<tr><td>></td><td colspan="2">*</td><td>л</td><td>4</td><td></td></tr>
<tr><td><</td><td colspan="2">*</td><td>л</td><td>4</td><td></td></tr>
<tr><td>>=</td><td colspan="2">*</td><td>л</td><td>4</td><td></td></tr>
<tr><td><=</td><td colspan="2">*</td><td>л</td><td>4</td><td></td></tr>
<tr><td>not</td><td>л</td><td>-</td><td>л</td><td>1</td><td>отрицание
<table>
<tr><th>Операнд 1</th><th>Результат</th></tr>
<tr><td>true</td><td>false</td></tr>
<tr><td>false</td><td>true</td></tr>
</table>
(not(2>1)=false)</td></tr>
</table>

Окончание таблицы 3

<table>
<tr><th rowspan="2">Операция</th><th colspan="2">Типы операндов</th><th rowspan="2">Тип результата</th><th rowspan="2">Приоритет</th><th rowspan="2">Пояснения</th></tr>
<tr><th>1</th><th>2</th></tr>
<tr><td>and</td><td>л</td><td>л</td><td>л</td><td>2</td><td>Логическое «И»
<table>
<tr><th>Операнд 1</th><th>Операнд 2</th><th>Результат</th></tr>
<tr><td>false</td><td>false</td><td>false</td></tr>
<tr><td>true</td><td>false</td><td>false</td></tr>
<tr><td>false</td><td>true</td><td>false</td></tr>
<tr><td>true</td><td>true</td><td>true</td></tr>
</table>
((2>1) and (2<4) = true,<br>(2>1) and (2<0) = false)</td></tr>
<tr><td>or</td><td>л</td><td>л</td><td>л</td><td>3</td><td>Логическое «ИЛИ»
<table>
<tr><th>Операнд 1</th><th>Операнд 2</th><th>Результат</th></tr>
<tr><td>false</td><td>false</td><td>false</td></tr>
<tr><td>true</td><td>false</td><td>true</td></tr>
<tr><td>false</td><td>true</td><td>true</td></tr>
<tr><td>true</td><td>true</td><td>true</td></tr>
</table>
((2>1) or (2<0) = true,<br>(2<1) or (2<0) = false)</td></tr>
<tr><td>=</td><td colspan="2">*</td><td>л</td><td>4</td><td></td></tr>
</table>

Обозначения в таблице: ц – целочисленный тип, в – вещественный тип, л – логический тип, с – строковый тип.

Примечание:

* - любой совместимый тип.

Пример:

Определим порядок выполнения операций в выражении:

2.5+10/(2-1)-2*3>5 mod 2

В первую очередь будет вычислено выражение в круглых скобках:

2.5+10/(**2-1**)-2*3>5 mod 2

2.5+10/1-2*3>5 mod 2

В полученном выражении операции «/», «*» и «mod» имеют одинаковый приоритет, а значит, выполняются в порядке слева-направо:

2.5+**10/1**-2*3>5 mod 2

2.5+1.0000000000E+01-**2*3**>5 mod 2

2.5+1.0000000000E+01-6>**5 mod 2**

2.5+1.0000000000E+01-6>1

Следует обратить внимание на то, что после выполнения операции «10/1» над целыми константами появилось вещественное значение 1.0000000000E+01 (см. тип результата операции /, табл. 3).

В полученном выражении операции «+», «-» имеют одинаковый приоритет, а значит, выполняются в порядке слева-направо:

**2.5+1.0000000000E+01**-6>1

**1.2500000000E+01-6**>1

Последней будет выполнена операция сравнения, имеющая 4-й приоритет.

**6.5000000000E+00>1**

Результат вычисления выражения:

**true**.

## Стандартные подпрограммы

Одним из видов операндов является **вызов функции** – конструкция, которая вызывает подпрограмму-функцию и передаёт в неё фактические аргументы. После выполнения подпрограммы в точку вызова (в то место выражения, где размещался вызов функции) передаётся полученное функцией значение.

В данном разделе приводятся заголовки подпрограмм, по которым можно определить количество и тип аргументов, а для функций – тип возвращаемого результата (для ).

### Процедуры и функции обработки порядковых типов

Процедура **Dec(var x)** – уменьшает значение переменной x до следующего порядкового значения.

Процедура **Inc(var x)** – увеличивает значение переменной x до следующего порядкового значения.

Функция **Odd(x:longint): boolean** – возвращает true если x – нечётное число и false в противном случае.

Функция **Pred(x):тип_аргумента** – возвращает предшествующее значению x порядковое значение. Тип возвращенного значения соответствует типу аргумента функции (x). Если функция применяется к первому значению в этом порядковом типе, то выдается сообщение об ошибке.

Функция **Succ(x) :тип_аргумента** – возвращает следующее за значением x порядковое значение. Тип возвращенного значения соответствует типу аргумента функции (x).Если функция применяется к последнему значению в этом порядковом типе, то выдается сообщение об ошибке.

### Функции преобразования значений

Функция **Chr(b: byte): char** – возвращает символ, заданный целым числом b.

Функция **Ord(x):longint** – возвращает порядковое число по значению перечислимого типа.

Функция **Round(x:real)longint** – используя математические правила округления, округляет значение вещественного типа до значения, имеющего длинный целый тип.

Функция **Trunc(x:real)longint** – усекает (округляет в меньшую сторону) значение x вещественного типа до значения, имеющего длинный целый тип.

Функция **Frac(x:real):real** – возвращает дробную часть аргумента x.

Функция **Int(x:real): real** – возвращает целую часть аргумента x.

### Арифметические функции

Функция **Abs(x) :тип_аргумента** – возвращает абсолютное значение аргумента x.

Функция **Arctan(x:real):real** – возвращает арктангенс аргумента x.

Функция **Cos(x:real):real** – возвращает косинус аргумента x.

Функция **Exp(x:real):real** – возвращает экспоненту аргумента x.

Функция **Ln(x:real):real** – возвращает натуральный логарифм аргумента x.

Функция **Pi:real** – возвращает значение числа пи (3,141592653897932385).

Функция **Sin(x:real):real** – возвращает синус аргумента x.

Функция **Sqr(x:real):тип_аргумента** – возвращает аргумент x в квадрате.

Функция **Sqrt(x:real):тип_аргумента** – возвращает квадратный корень аргумента x.

### Совместимость типов в операциях

Иногда для операндов в операциях, например, в операциях сравнения, требуется совместимость типов операндов. Совместимость типов, кроме того, является важной предпосылкой для **совместимости по присваиванию**.

Совместимость типов имеет место, если выполняется по крайней мере одно из следующих условий:

- оба типа являются одинаковыми (имеют одинаковые описания);
- оба типа являются вещественными типами;
- оба типа являются целочисленными;
- один тип является поддиапазоном другого;
- оба типа являются поддиапазонами одного и того же основного типа.

## Именованные константы

### Выражения-константы

**Выражение-константа** представляет собой выражение, которое может вычисляться компилятором (без выполнения программы).

Поскольку компилятор должен иметь возможность полностью вычислить выражение-константу во время компиляции, в выражениях-константах не допускается использовать:

– ссылки на переменные и типизированные константы;

– вызовы функций (за исключением стандартных функций).

Примеры выражений-констант:

(2000-20)/2, Chr(255), ’ошибка №’+’2’;

### Нетипизированные константы

**Нетипизированные константы** фактически является числом, закрепленным за определённым именем. Нетипизированным константам не выделяется область памяти для хранения значения. Прежде чем начать процесс компиляции, в тексте программы выполняется замена идентификаторов нетипизированных констант на их значения. Следовательно, внутри раздела операторов блока нельзя использовать операторы, изменяющие значение нетипизированных констант.

**Синтаксис описания нетипизированных констант (раздел const):**

**Ид_конст1 = выражение_константа1; ...**

**Ид_констN = выражение_константаN;**

Примеры описания нетипизированных констант:

```
const
Min = 10;
Max = 100;
Mid = (Max-Min)/2;
Beta = Chr(255);
```

### Типизированные константы

**Типизированные константы** фактически являются переменными, которые проинициализированы перед началом работы программы.

**Инициализацией** объектов, использующихся для хранения данных, называют присваивание им начального значения.

В отличие от нетипизированных констант, типизированные константы можно использовать точно так же, как переменные. Типизированные константы не могут использоваться в описании других констант или типов.

Надо отметить, что типизированные константы инициализируются только один раз – в начале выполнения программы. Таким образом, при каждом новом входе в процедуру или функцию локально описанные типизированные константы заново не инициализируются.

**Синтаксис описания типизированных констант (раздел const)**:
**Ид_конст1:описание_типа1|Ид_типа1 = выражение_константа1; ...**
**Ид_констN:описание_типаN|Ид_типаN = выражение_константаN;**

Примеры описания типизированных констант:
```
const
F  : real = -0.1;
IntF : integer = round()
Hider : string[7] = ’Section’;
```

## Операторы

**Оператор** – конструкция языка программирования, определяющая законченное алгоритмическое действие.

**Синтаксис раздела операторов:**
**Опер1;Опер2;...**
**...**
**ОперN;**

Порядок выполнения операторов слева-направо, сверху-вниз.

Операторы подразделяют на **простые** и **структурные**.
**Простые операторы** не содержат в своей структуре других операторов и описывают только определённое алгоритмическое действие.
**Структурные операторы** включают в себя другие операторы и управляют их работой.

### Оператор присваивания

**Оператор присваивания** – простой оператор, который выполняет замену текущего значения переменной новым значением, которое определяется результатом вычисления выражения. Также оператор присваивания может использоваться для задания значения, возвращаемого функцией (см. раздел Подпрограммы-функции).

**Синтаксис оператора присваивания:**
**Ид_переменной|Ид_функции := выражение;**

Выражение должно быть совместимо по присваиванию с типом переменной или типом значения, возвращаемого функцией в качестве результата.

### Совместимость в операциях присваивания

Совместимость по присваиванию необходима, если имеет место присваивание значения, например, в операторе присваивания или при передаче значений-параметров в подпрограмму.

Объект хранения данных, имеющий тип T1, является совместимым по присваиванию со значением, имеющим тип T2 (то есть допустимо T1:=T2), если выполняется одно из следующих условий:

- T1 и T2 имеют тождественные типы (правила определения тождественности см. в пункте Параметры-переменные);
- T1 и T2 являются совместимыми порядковыми типами, и значения типа T2 попадают в диапазон возможных значений T1;
- T1 и T2 являются вещественными типами, и значения типа T2 попадают в диапазон возможных значений T1;
- T1 является вещественным типом, а T2 является целочисленным типом;
- T1 и T2 являются строковыми типами;
- T1 является строковым типом, а T2 является символьным типом.

Если в операторе необходима совместимость по присваиванию, а ни одно из условий предыдущего списка не выполнено, то на этапе компиляции выдается сообщение об ошибке.

### Операторы вызова процедуры

**Оператор вызова процедуры** простой оператор, который инициализирует формальные параметры процедуры фактическими (также этот процесс называют передачей параметров в процедуру), и передаёт ей управление (см. пункт Параметры).

**Синтаксис оператора вызова процедуры:**
**Ид_процедуры ( Фактич_парам1, Фактич_парам2,... );**

### Стандартные процедуры

### Процедуры вывода

**Write( Зн1 [,Зн2,..., Знn] )** – записывает одно или более значений из одной или более переменных в текстовый файл.

Каждые параметры **Зн** являются выражениями, значения которых должны быть выведены на экран. Каждое выводимое выражение должно быть символьного, целого, вещественного, строкового, или булевского типа.

**Writeln( Зн1, [,Зн2,..., Знп])** – данная процедура является расширением процедуры Write. После выполнения процедуры Write процедура Writeln переводит курсор на следующую строку.

Пример использования процедур вывода:
Рассмотрим выполнение процедур вывода следующей программы:

```
program Wrt;
var
r:real; i:integer;
s:string[10];
begin
i:=12;
r:=100;
s:= 'r=';
write(s,r);
writeln('i=',i);
write(i+1,r+i);
writeln();
write(i,', ',i);
end.
```

При выполнении программы на экране будут появляться выводимые процедурами вывода значения (таблица 4).

Таблица 4 – Пошаговое выполнение процедур вывода

| **Выполняемый оператор** | **Экран программы** |
|---|---|
| write(s,r); | r=1.0000000000E+02 |
| writeln('i=',i); | r=1.0000000000E+02i=12 |
| write(i+1,r+i); | r=1.0000000000E+02i=12<br>13 1.1200000000 E+02 |
| writeln(); | r=1.0000000000E+02i=12<br>13 1.1200000000 E+02 |
| write(i,', ',i); | r=1.0000000000E+02i=12<br>13 1.1200000000 E+02<br>12, 12 |

**Процедуры ввода**

**Read( Пр1, [,Пр2,..., Прn ] )** – считывает одно или более значений из текстового файла в одну или более переменных.

Каждый параметр **Пр№** является переменной символьного, строкового, целого или вещественного типа.

Процедура останавливает выполнение программы до тех пор, пока пользователь не введёт с клавиатуры требуемое количество значений и нажмёт <Enter>. Числовые значения при вводе должны разделяться пробелом (<Space>) или окончанием строки (клавиша <Enter>). При считывании строковых и символьных значений символ пробел считается частью вводимой строки, а не разделительным символом.

**Readln( Пр1, [,Пр2,..., Прn] )** – данная процедура является расширением процедуры Read. После выполнения процедуры Read процедура Readln делает пропуск всех введённых с клавиатуры символов до начала следующей строки.

**Пример использования процедур ввода:**

Рассмотрим по шагам выполнение следующей программы:

```
program Rd;
var
r:real; i,i2:integer; c: char; s:string[3];
begin
read(c,r,i,i2);
read(i);
readln(s,r);
read(i);
end.
```

При выполнении программы пользователь должен будет вводить определённые значения, которые будут сохраняться в соответствующие переменные (таблица 5).

Таблица 5 – Пошаговое выполнение процедур ввода

| **Выполняемый оператор** | **Вводимое пользователем с клавиатуры** | **Состояние переменных** | | | | |
|---|---|---|---|---|---|---|
| | | **r** | **i** | **i2** | **c** | **s** |
| read(c,r,i,i2); | 234.5 3 22<Enter> | 34.5 | 22 | 0 | ’2’ | ’’ |
| | 33 101<Enter> | 34.5 | 22 | 33 | ’2’ | ’’ |
| read(i); | программа не останавливается для ввода данных | 34.5 | 101 | 33 | ’2’ | ’’ |
| readln(s,r); | st2.25 2.3 13<Enter> | 0.25 | 101 | 33 | ’2’ | ’st2’ |

Окончание таблицы 5

| Выполняемый оператор | Вводимое пользователем с клавиатуры | Состояние переменных |
|---|---|---|
| read(i); | 2.5<Enter> | Ошибка: invalid numeric format (попытка записать вещественное число в переменную целого типа) |

### Прочие стандартные процедуры

Процедура **Exit** – позволяет немедленно выйти из текущего блока (процедуры, функции или программы).

Процедура **Pandomize** – инициализирует генератор случайных чисел.

Функция **Random [ ( Range: Word)]: word** – возвращает случайное число, равномерно распределённое в диапазоне от 0 до значения **Range**. Если генератор случайных чисел не инициализирован, то при последовательном обращении к данной функции будет возвращаться псевдослучайная последовательность значений (т. е. при каждом новом запуске программы эта последовательность не изменится).

Рассмотренного в предыдущих разделах достаточно, чтобы составить полноценную программу решения задачи.

В качестве примера разработаем программу для решения следующей задачи:

Даны катеты прямоугольного треугольника *a* и *b*. Найти его гипотенузу *c* и периметр P.

Для решения задачи необходимо припомнить формулы вычисления гипотенузы *c* и периметра P:

$$c = \sqrt{a^2 + b^2},$$

$$P = c + b + a.$$

Текст программы на языке Pascal выглядит следующим образом:

```
program Lin;
var
a,b,c:real;
P:real;
begin
writeln('введите длины двух сторон треугольника');
readln(a,b);
c:=sqrt(sqr(a)+sqr(b));
P:=a+b+c;
writeln('гипотенуза-',c,' периметр-',P);
end.
```

### Составной оператор

Составные операторы задают последовательный порядок выполнения операторов, являющихся их элементами (как и в разделе операторов: слева-направо, сверху-вниз). Составные операторы обрабатываются, как один оператор, что имеет решающее значение там, где синтаксис языка Pascal допускает использование только одного оператора.

**Синтаксис составного оператора:**
**Begin**
**Опер1;Опер2;...**
**...**
**ОперN;**
**End;**

### Оператор if

Оператор позволяет организовать ветвление в программе.

**Синтаксис оператора if:**

**if лог_выражение then оператор1 [else оператор2];**

Если результатом выражения является значение **False** и присутствует ключевое слово **else**, то выполнятся оператор, следующий за ключевым словом **else** (**оператор2**). Если ключевое слово **else** отсутствует, то управление передаётся на следующий оператор, минуя оператор, следующий за ключевым словом **then** (**оператор1**).

Замечания:

- Результат вычисления **лог_выражение** должен, иметь логический тип (**boolean**). Если результатом выражения является истинное значение (**True**), то выполняется оператор, следующий за ключевым словом **then** (**оператор1**).
- Следует обратить особое внимание на то, что после ключевых слов **then** и **else** согласно синтаксису может находиться только один оператор.

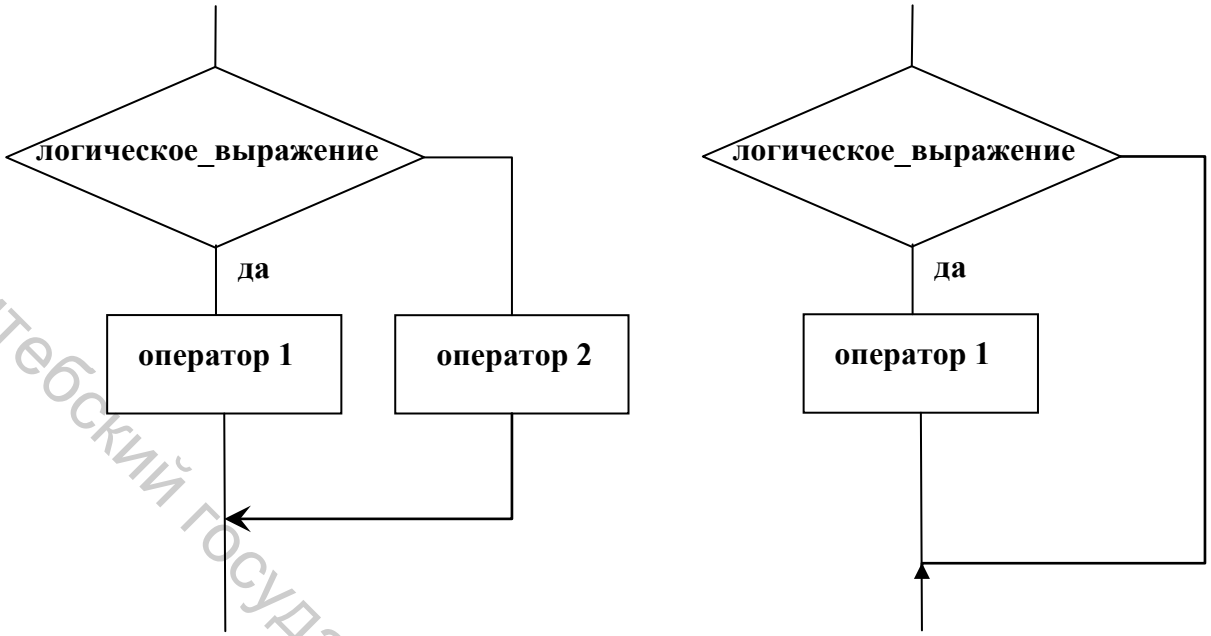



Рисунок 2 – Блок-схема оператора ветвления:
a) полная форма (с ветвью else); b) сокращённая форма (без ветви else)

В качестве примера использования оператора if приведём решение следующей задачи: Даны три числа. Найти и вывести максимальное из них и среднеарифметическое двух оставшихся.

Для трёх различных входных значений количество разветвлений алгоритма также равно трём. После ввода данных (рис. 3, блок 2) в блоке 3 проверяется условие максимальности значения в переменной a. Если условие верно, т. е. из трёх значений a является максимальным, то управление передаётся блокам 5 и 6. В противном случае остаётся два варианта: либо максимальным является значение в переменной b, либо – в переменной c. Блок 4 выполняет такую проверку и, в зависимости от результата, передаёт управление либо на блок 9 и далее на блок 10, либо на блоки 7 и 8.

В каждой из веток алгоритма вычисляется среднеарифметическое значение, которое и выводиться перед окончанием алгоритма в блоке 11.

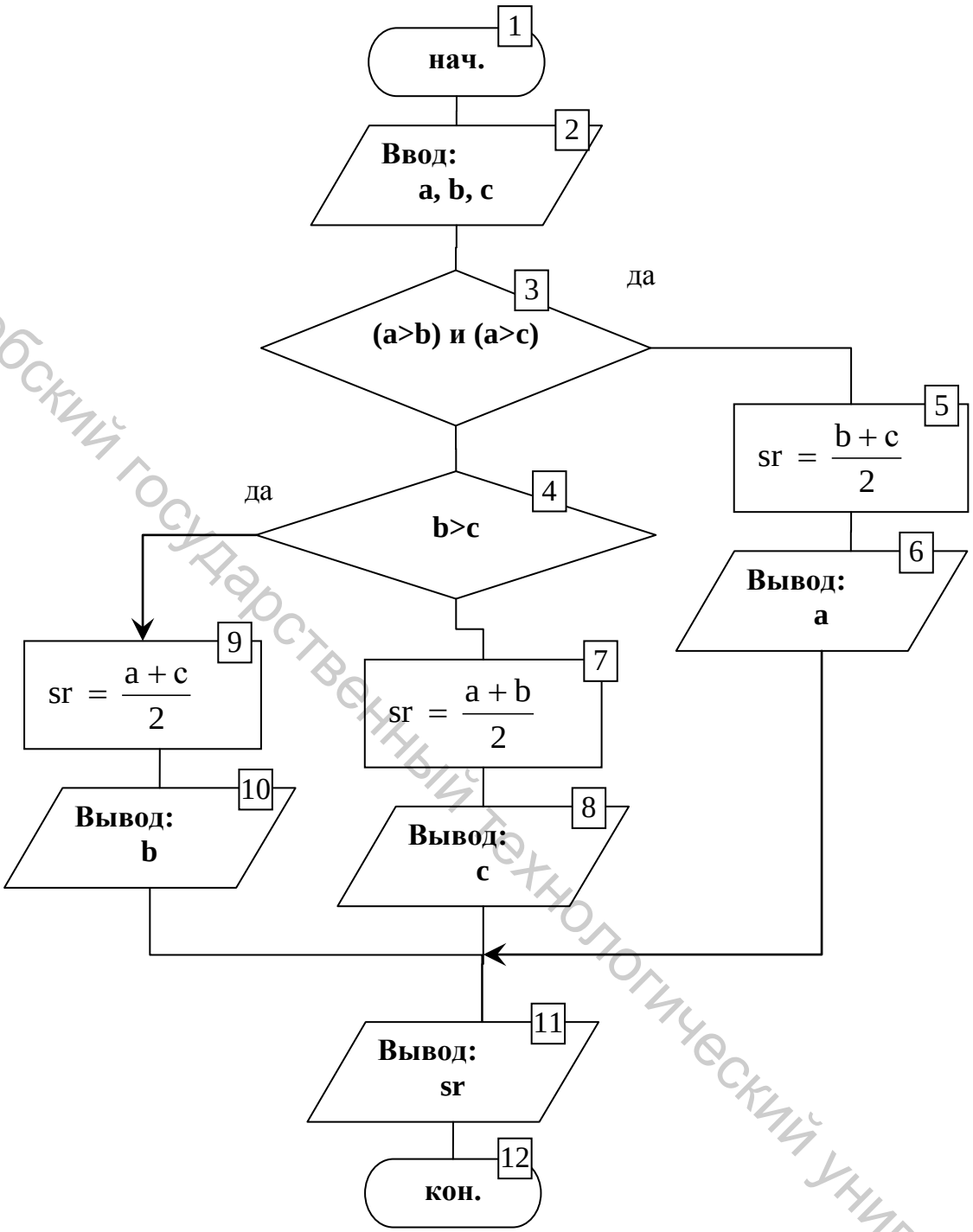



Рисунок 3 – Блок-схема разветвлённого алгоритма

Реализация алгоритма в программе на языке Pascal:

```
program vetv;
var
a,b,c:real;
sr:real;
begin
```

```
writeln('введите три числа');
readln(a,b,c);
if (a>b) and (a>c) then
      begin
       sr:=(b+c)/2;
       writeln ('максимальное - ',a);
      end
       else
          if b>c then
             begin
              sr:=(a+c)/2;
              writeln ('максимальное - ',b);
             end
              else
                begin
                 sr:=(a+b)/2;
                 writeln ('максимальное - ',c);
                end;
writeln('среднеарифметическое - ', sr);
end.
```

Следует обратить внимание на то, что два оператора, находящиеся в ветке else или then, должны быть заключены в составной оператор (begin...end).

### Оператор варианта (case)

Для организации каскада ветвлений может быть использован оператор выбора.

**Синтаксис оператора case:**

```
Case выражение_селектор of
Выражение_перечисления1: оператор1;
Выражение_перечисления2: оператор2;
...
[else операторN; операторN+1;]
 end;
```

Оператор варианта приводит к выполнению оператора, которому предшествует **выражение_перечисления**, одному из значений которого равно значение **выражения_селектора**. Если такого **выражения_перечисления** не существует и присутствует ветвь else, то выполняется оператор, следующий за ключевым словом else. Если же ветвь else отсутствует, то никакой оператор не выполняется.

**Выражение селектор** должно иметь порядковый тип, и значения верхней и нижней границы этого типа должны лежать в диапазоне от -32768 до 32767. Таким образом, длинный целый тип является недопустимым типом переключателя.

Все **выражения перечисления** должны быть уникальными и иметь порядковый тип, совместимый с типом **выражения селектора**.

В качестве примера использования оператора **case** приведём программу, анализирующую введённый пользователем символ.

Текст программы на языке Pascal выглядит следующим образом:

```
program ff;
var
ch:char;
begin
read(ch);
case Ch of
'A'..'Z', 'a'..'z': WriteLn('Буква');
'0'..'9':  WriteLn('Цифра');
'+', '-', '*', '/': WriteLn('Знак операции');
else  WriteLn('спец. символ');
end;
end.
```

### Операторы цикла

Общим для всех операторов цикла является наличие у них **тела** и **заголовка**. **Тело цикла** составляют операторы, предназначенные для циклического повторения. **Заголовок цикла** определяет условие окончания или количество повторений операторов, составляющих **тело** цикла.

### Оператор цикла с постусловием (repeat)

Оператор **repeat** позволяет организовать цикл, **тело** которого расположено перед **заголовком.**

```
Синтаксис оператора Repeat:
Repeat
Оператор1;оператор2;
...
операторN;
Until логическое_выражение;
```

Операторы, заключенные между ключевыми словами **repeat** и **until**, выполняются последовательно до тех пор, пока результат **логического_выражения** не примет значение **True**.

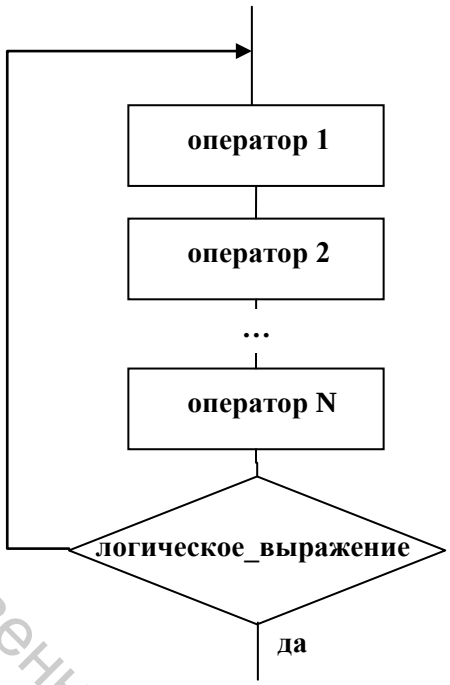



Рисунок 4 – Блок-схема цикла с постусловием

В качестве примера использования цикла с постусловием **repeat** приведем программу, которая по заданному пользователем значению x выводит на экран результат вычисления формулы:

$$y = \frac{\sqrt{x+5}}{x} + 1.$$

Цикл **repeat** в данном случае используется для контроля вводимого пользователем значения. Условие окончания цикла (**(x>-5)and(x<>0)**) составлено таким образом, что если пользователь введёт значение, равное 0 или меньшее -5, то цикл выполнится снова, и пользователь будет вынужден повторно ввести значение x.

```
program rpt;
var
x,y:real;
begin
 repeat
 writex('x->');
 readln(x);
 until (x>-5)and(x<>0);
 y:=sqrt(x+5)/x+1;
```

```
  writeln('x=',x);
  writeln('y=',y);
end.
```

### Оператор цикла с предусловием (while)

Оператор **while** позволяет организовать цикл, **тело** которого расположено после **заголовка.**

**Синтаксис оператора While:**
**While логическое_выражение do**
**оператор;**


Рисунок 5 – Блок-схема цикла с предусловием

Единственный оператор, составляющий **тело цикла,** выполняется повторно до тех пор, пока **логическое_выражение** принимает значение **True**.

В качестве примера использования цикла с предусловием **while** приведем программу, которая выполняет те же функции, что и программа, приведённая в предыдущем пункте (**Пример использования оператора repeat**).

Цикл **while** продолжает работать, пока логическое выражение будет иметь значение **true**, в отличие от цикла **repeat**, который продолжает работать, пока логическое выражение будет иметь значение **false**. Поэтому потребовалось изменить логическое выражение (условие окончания цикла): **(x<-5) or (x=0).**

Также следует обратить внимание на оператор **x:=-10**, который используется для того, чтобы цикл выполнился в первый раз. Вместо значения

**10** можно использовать любое другое значение, позволяющее получить значение **true** при вычислении выражения **(x<-5) or (x=0)** (например: 11, 100, -3 и т. д.).

```
program rpt;
var
x,y:real;
begin
 x:=-10;
 while (x<-5) or (x=0) do
 begin
  write('x->');
  readln(x);
 end;
 y:=sqrt(x+5)/x+1;
 writeln('x=',x);
 writeln('y=',y);
end.
```

### Оператор цикла с параметром (For)

Оператор цикла **For** позволяет выполнить тело цикла определённое количество раз.

**Синтаксис оператора For:**
**For Ид_переменной := выражение1 to|downto выражение2 do**
**оператор;**


Рисунок 6 – Блок-схема цикла с параметром

**Оператор**, который содержится в теле цикла for, выполняется один раз для каждого значения в диапазоне между начальным и конечным значением.

**Управляющая переменная (Ид_переменной**) должна иметь порядковый тип.

Значения **выражения1** (начальное значение) и **выражения2** (конечное значения) определяются один раз. Эти значения сохраняются на протяжении всего выполнения оператора for.

В результате вычисления **выражения1** и **выражения2** должны быть получены значения, тип которых совместим по присваиванию с **управляющей переменной.**

Когда в операторе цикла используется ключевое слово to, значение управляющей переменной увеличивается при каждом повторении цикла на единицу. Если в начале работы цикла начальное значение превышает конечное значение, то содержащийся в теле оператора for оператор не выполнятся.

Когда в операторе цикла используется ключевое слово downto, значение управляющей переменной уменьшается при каждом повторении на единицу. Если в начале работы такого цикла начальное значение меньше, чем конечное значение, то содержащийся в теле оператора цикла оператор не выполняется.

После выполнения оператора for значение управляющей переменной становится неопределенным.

Приведём эквивалентную схему оператора:

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

из которой следуют все вышеприведённые замечания.

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

Следует обратить внимание на следующее:

Условием выхода из цикла является ложное значение выражения **V <> Temp2,** и если оператор, содержащийся в теле оператора for, изменяет значение

управляющей переменной, то может возникнуть ситуация, когда значение переменной **V** никогда не станет равным значению **Temp2,** и цикл не сможет корректно завершиться.

## Рекуррентные вычисления

Общая формулировка задач рекуррентного вычисления суммы или произведения конечного количества элементов выглядит следующим образом:

$$S = \sum_{i=1}^{n} A_i \text{ , или } \quad S = \prod_{i=1}^{n} A_i ,$$

где $A_i = F(A_{i-1})$, т. е. последующее значение элемента $A$ вычисляется на основе предыдущего его значения.

Кроме того, и вычисление суммы, или произведения, можно считать рекуррентным, поскольку следующее значение суммы, или произведения, вычисляется как:

$$S_i = S_{i-1} + A_i = F(S_{i-1}) .$$

Общую методологию решения подобных задач рассмотрим на следующем примере. Необходимо вычислить значение $S$:

$$S = \sum_{n=2}^{k} \left( (-1)^{n+2} \frac{(x+1)^{n+2}}{(2n-1)!} \sin(\frac{\pi}{n}) \right) \qquad (1)$$

Вначале, для упрощения рекуррентной формулы вычисления элемента суммы, разобьём его на несколько частей (рис. 7).

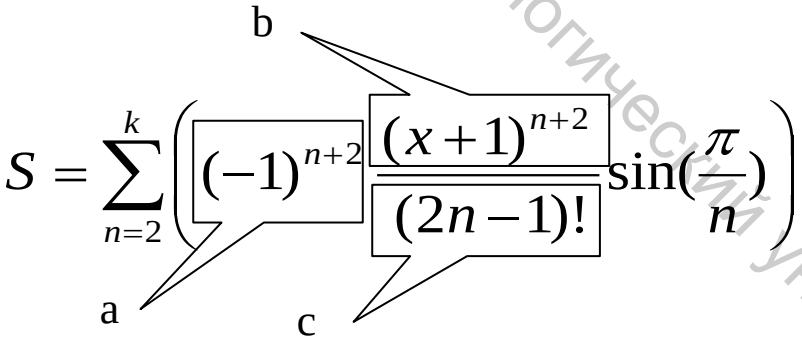



Рисунок 7 – Разбиение элемента суммы формулы 1

Элемент $\sin(\frac{\pi}{n})$ можно вычислить без использования рекуррентной формулы, поэтому для него отдельной переменной не выделено.

Затем сведём в таблицу значения каждой из частей формулы при разных значениях переменной *n* (см. таблицу 6).

Таблица 6 – Изменения частей элемента суммы формулы 1

| n | a | b | c |
|---|---|---|---|
| **2** | 1 | $(x+1)^4$ | 3!=1*2*3 |
| **3** | -1 | $(x+1)^5$ | 5!=1*2*3*4*5 |
| **4** | 1 | $(x+1)^6$ | 7!=1*2*3*4*5*6*7 |

Теперь, когда появилась возможность сравнить предыдущее значение части элемента суммы с его последующим значением, можно составить рекуррентные формулы:

$$a_i = a_{i-1} \cdot (-1)$$
$$b_i = b_{i-1} \cdot (x+1)$$
$$c_i = c_{i-1} \cdot (2n-1) \cdot (2n-2)$$

Рассмотрим подробнее рекуррентное выражение для переменной *c*. Если обратиться к таблице 6, то можно заметить, что последующее значение переменной *c* получается домножением предыдущего значения на два числа. При *n = 3* необходимо домножить на 4 и 5, при *n = 4* необходимо домножить на 6 и 7.

Эти два числа определённым образом зависят от числа *n*. Второе из них соответствует формуле элемента суммы, связанного с переменной *c - 2n-1* (см. рис. 7). Тогда первое число можно получить, отняв от числа *2n-1* единицу (поскольку это предыдущее целое число). Откуда и получаем значение *2n-2*.

После составления рекуррентных формул можем перейти к составлению алгоритма и программы.

Следует обратить внимание на блок 3 (рис. 8), в котором инициализируются переменные *a, b, c*. В качестве начальных значений используем значения из первой строки таблицы 6 (при *n = 2*).

Теперь обратимся к блоку 4. Почему начальное и конечное значения цикла выбраны именно таким образом?

Дело в том, что в блоке 5 вначале выполняется вычисление следующего значения суммы (переменная *S*), а только затем, с помощью составленных ранее рекуррентных выражений, вычисляются следующие значения переменных *a, b* и *c*.

Новые значения переменных *a, b* и *c* должны вычисляться при *n = 3* (следующее значение), а значение *S* при *n = 2*. И если цикл начнётся со значения *n = 3*, то следующие значения переменных *a, b* и *c* будут вычислены правильно, а выражение для вычисления *S* нужно будет изменить, уменьшив на единицу все вхождения в него переменной *n* (сравните формулы вычисления переменной *S* на рисунке 7 и в блоке 5 на рисунке 8).

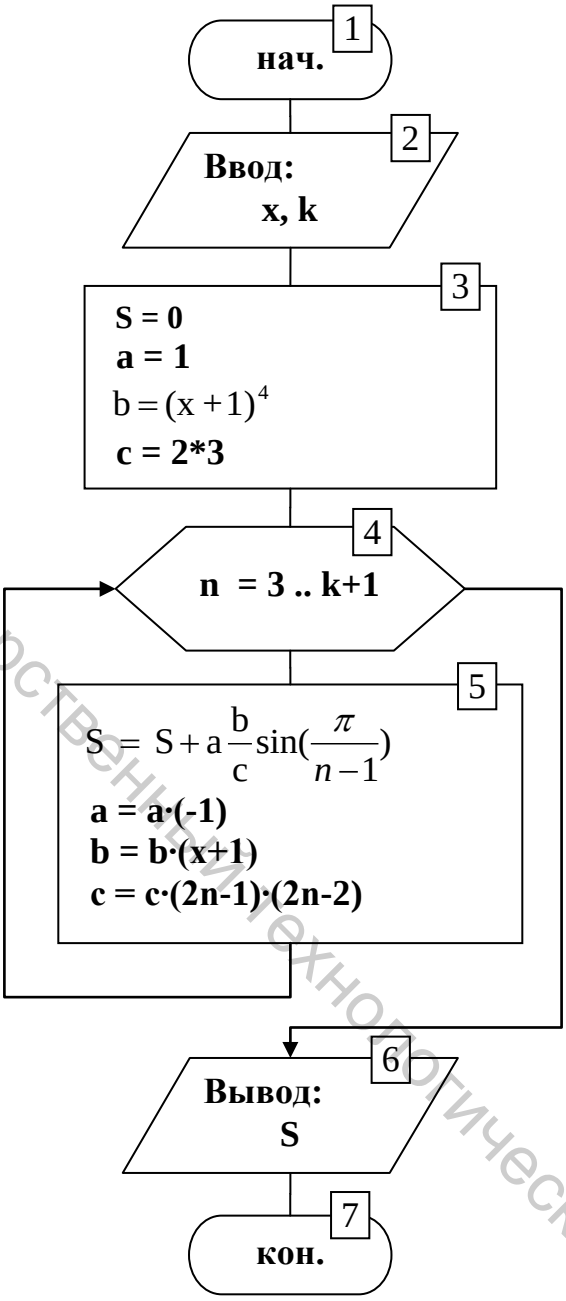



Рисунок 8 – Блок-схема алгоритма вычисления суммы конечного количества элементов

Кроме того, чтобы сохранить требуемое количество повторений тела цикла *(k-2)*, увеличим конечное значение счетчика цикла на единицу. Таким образом, количество повторений будет равно: *3–(k+1) = k-2*.

Теперь можно перейти к реализации алгоритма в исходном тексте программы на языке Pascal.

```
program K_S;
var x,s,a,c:real;
   n,b:integer;
begin
writeln('введите x и количество слагаемых');
readln(x, k);
s:=0;
a:=1;
b:=sqr(sqr((x+1)));
c:=2*3;
for n:=3 to k+1 do
begin
s:=s+a*(b/c)*sin(Pi/(n-1));
a:=a*(-1);
b:=b*(x+1);
c:=c*(2*n-1)*(2*n-2);
end;
writeln('S=',s:8:5);
end.
```



**Вычисление бесконечных сумм**

Общая формулировка задач рекуррентного вычисления суммы бесконечного количества элементов:

$$S = \sum_{i=1}^{\infty} A_i ,$$

где $A_i = F(A_{i-1})$, т. е. последующее значение элемента А вычисляется на основе предыдущего его значения. Вычисление продолжается до тех пор, пока разница между предыдущим и последующим значением суммы не станет меньше, чем заданное пользователем значение точности:

$$\xi < | S_{i-1} - S_i | .$$

В остальном постановка этой задачи аналогична постановке задачи нахождения суммы конечного количества элементов.

Общую методологию решения подобных задач рассмотрим на следующем примере:

Необходимо вычислить значение $S$:

$$S = \sum_{n=1}^{\infty}\left((-1)^{2n+1}\frac{n\cdot x^{3-2n}}{(n+3)!}\right) \qquad (2)$$

Вначале, как и в случае с задачей вычисления суммы конечного количества элементов, разобьём исходную формулу суммы на несколько частей (рис. 9).

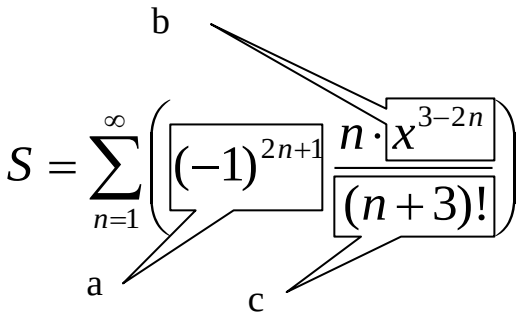



Рисунок 9 – Разбиение элемента суммы формулы 2

Элемент *n* можно вычислить без использования рекуррентной формулы, поэтому для него отдельной переменной не выделено.

Затем сведём в таблицу значения каждой из частей формулы при разных значениях переменной *n* (см. таблицу 7).

Таблица 7 – Изменения частей элемента формулы 2

| **n** | **a** | **b** | **c** |
|---|---|---|---|
| **1** | -1 | $x^{1}$ | 4! = 1*2*3*4 |
| **2** | -1 | $x^{-1}$ | 5! = 1*2*3*4*5 |
| **3** | -1 | $x^{-3}$ | 6! = 1*2*3*4*5*6 |

Из таблицы 7 видно, что значение переменной *a* неизменно, поэтому от её использования можно отказаться. Составим рекуррентные формулы для оставшихся переменных:

$$b_i = b_{i-1}\cdot\frac{1}{x^2}$$

$$c_i = c_{i-1}\cdot(n+3)$$

Рассмотрим подробнее рекуррентное выражение для переменной *c*. Если обратиться к таблице 7, то можно заметить, что последующее значение переменной *c* получается домножением предыдущего значения на одно число. При *n = 2* необходимо домножить на 5, при *n = 3* необходимо домножить на 6.

Это число зависит от числа *n*, оно соответствует формуле элемента суммы, связанного с переменной *c – n+3* (см. рис. 9).

После составления рекуррентных формул можем перейти к составлению алгоритма и программы.

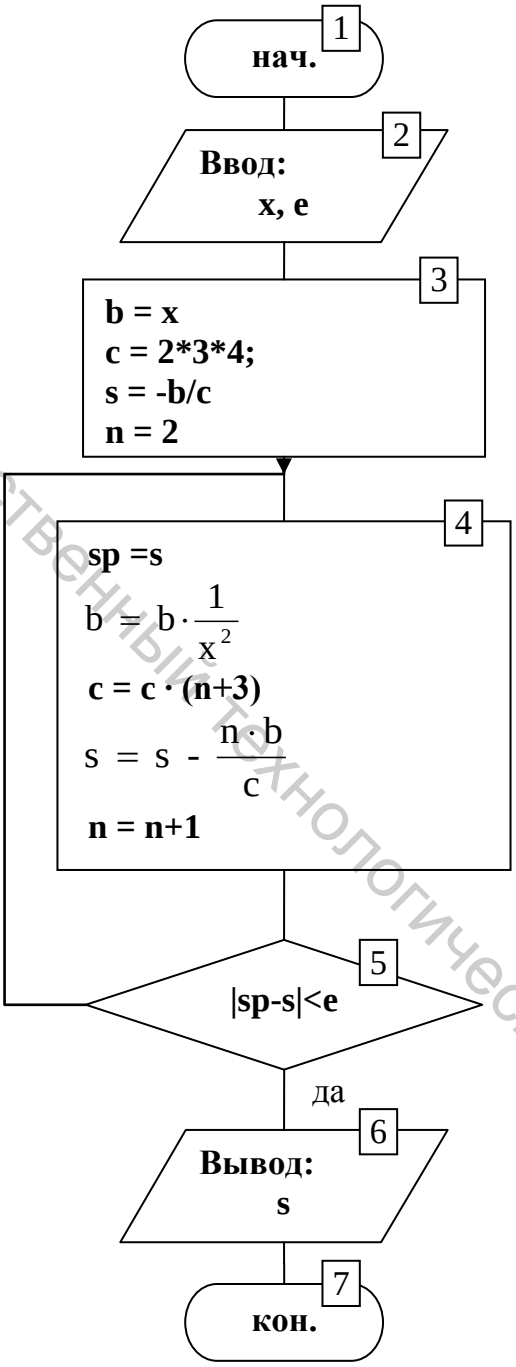



Рисунок 10 – Блок-схема алгоритма вычисления суммы конечного количества элементов

Блок 3 (рис. 10) предназначен для инициализации переменных *a* и *b*. В качестве начальных значений используем значения из первой строки таблицы 7

(при *n = 2*). Затем в этом же блоке вычисляется и сохраняется в переменной *s* первый элемент суммы, вычисляемый при *n = 1*.

Цикл repeat не использует переменную счётчик, поэтому вводим её самостоятельно (переменная *n*). Инициализируется эта переменная также в блоке 3. Её начальное значение - 2, т. к. первый элемент суммы уже вычислен и сохранён до начала цикла.

В теле цикла (блок 4) сначала сохраняем значение переменной *s* в переменной *sp*, затем вычисляем новые значения переменных *a* и *b*, добавляем к переменной *s* новый элемент суммы и увеличиваем значение счётчика *n*.

Теперь в переменной *s* находится последующее значение суммы $S_i$, а в переменной *sp* – предыдущее $S_{i-1}$.Заголовок цикла repeat (блок 5) проверяет условие окончания цикла на основе этих значений и при истинности условия передаёт управление на блок вывода результатов 6.

Теперь можно перейти к реализации алгоритма в исходном тексте программы на языке Pascal.

```
program B_S;
var x,s,c,e,sp:real;
   n,b:integer;
begin
writeln('введите x и точность');
readln(x, e);
b:=x;
c:=2*3*4;
s:=-b/c;
n:=2;
repeat
sp:=s;
b:=b*(1/(x*x));
c:=c*(n+3);
s:=s+-(n*b)/c;
n:=n+1;
until abs(sp-s)<e;
writeln('S=',s:8:5);
end.
```

## Структурные типы

**Структурный тип** позволяет содержать более одного значения и характеризуется методом структурирования и типами своих компонентов.

Для структурных типов характерно использование понятия «**структура**» как описания представления пользователя о таком объекте. Например,

двумерный массив пользователю удобно представлять в виде матрицы, состоящей из строк и столбцов, хотя на самом деле в памяти ЭВМ такой массив размещается вовсе не в виде таблицы (для ЭВМ с её линейной организацией памяти понятия «строка» и «столбец» весьма условны).

Структурные типы языка Pascal:
- массив;
- множественный тип;
- файловый тип;
- запись.

Все структурные типы в языке Pascal являются пользовательскими.

## Массивы

### Описания массивов

**Массив** – проиндексированное конечное множество элементов одинакового типа.

**Синтаксис описания одномерного массива:**
**Array [индексный_тип] of тип_элемента**

Где **индексный_тип** (индексирующий тип) любой из перечисляемых типов (например: **byte** или **«отрезок»**), множество значений которого являются индексами для элементов массива.

Например: имеется описание переменной одномерного массива:

```
var
A:array[3..8] of real;
```

Переменная A является массивом, содержащим 6 элементов типа **real**. Элементы массива проиндексированы (пронумерованы) целыми числами от 3 до 8. В описании массива 3..8 является описанием типа **«отрезок»**.

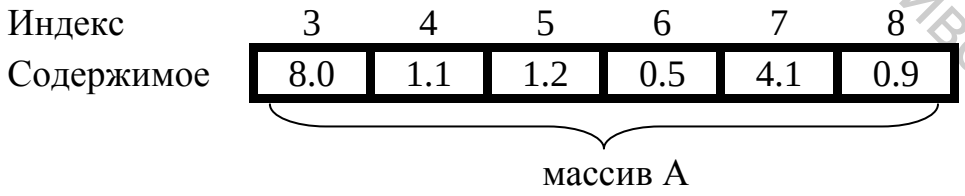



Рисунок 11 – Структура одномерного массива

Таким образом, элемент с индексом 5 массива **A** (см. рис. 11) содержит значение 1.2, а элемент с индексом 7 – значение 4.1.

Многомерные массивы используют один индексирующий тип для каждой размерности. Число размерностей является неограниченным.

**Синтаксис описания многомерного массива:**
**Array [индексный_тип1, индексный_тип2,... ] of тип_элемента**

Какой из индексов считать номером строки, а какой номером столбца, не имеет значения.

Совет: определите для себя, какой из индексов будет соответствовать номеру строки, а какой – номеру столбца, и придерживайтесь этого правила в дальнейшем.

Например, имеется описание переменной двумерного массива:
**var**
**B:array[1..4, 2..4] of byte;**
Такой массив можно представить в виде матрицы (рис. 12).

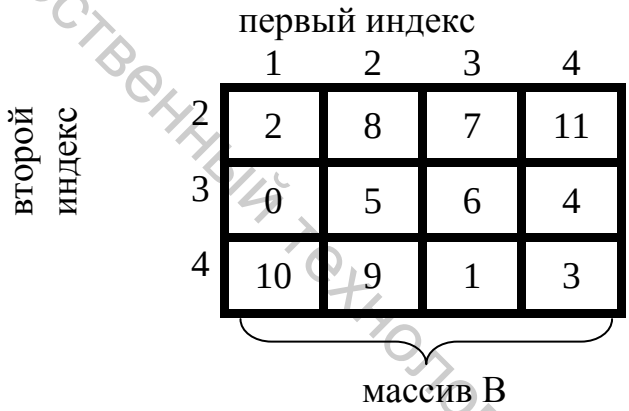



Рисунок 12 – Структура двумерного массива

Таким образом, элемент массива B (см. рис. 12), первый индекс которого равен 2, а второй – 3 ($B_{2,3}$), содержит значение 5, а элемент, первый индекс которого равен 4, а второй – 2 ($B_{4,2}$), содержит значение 11.

Если тип элемента в массиве также является массивом, то результат можно рассматривать как массив массивов или как один многомерный массив.

Например, массив описанный как:

**array[0..5] of array[1..10] of real**

интерпретируется компилятором точно так же, как массив с описанием:

**array[0..5,1..10] of real.**

**Типизированные константы массивы**

Описание константы одномерного массива содержит значения элементов, заключенные в скобки и разделенные запятыми.

**Синтаксис выражения-константы одномерного массива:**
**(Знач1, Знач2, ...)**

**Знач1, знач2... – выражения-константы,** тип результата которых должен соответствовать типу элементов константы массива.

Пример описания константы одномерного массива:
```
const
C: array[1..3] of  byte = (23,200,250);
```

При описании константы многомерного массива константы каждой размерности заключаются в отдельные скобки и разделяются запятыми. Расположенные в середине константы соответствуют самым правым размерностям.

**Синтаксис выражения-константы двумерного массива:**
**((Знач11, Знач12, ...), (Знач21, Знач22, ...), ...)**

**Знач11, знач12... – выражения-константы,** тип результата которых должен соответствовать типу элементов константы массива.

Пример описания константы двумерного массива:
```
const
M2: array[1..3,1..2] of integer = ((0,1),(2,3),(4,5));
```

Таким образом, если принять, что первый индекс массива M2 соответствует номеру столбца, а второй – номеру строки, то массив M2 можно представить так, как показано на рисунке 13.

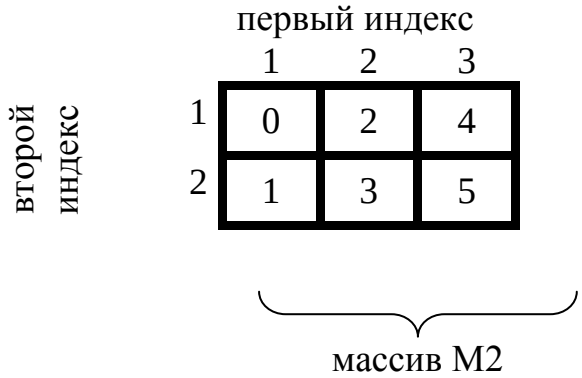



Рисунок 13 – Двумерный массив-константа

**Индексы**

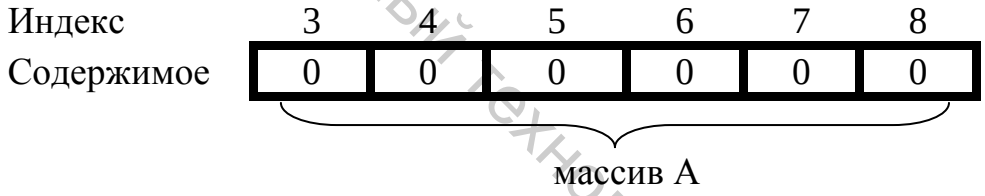
Конкретный элемент массива обозначается с помощью идентификатора переменной массива, за которым указывается индекс, определяющий номер элемента.

**Синтаксис индекса:**
**Ид_переменной_массива[выражение]**

Пример конструкций, используемых для доступа к элементам одномерных массивов:
A[2], Mass2[i+1], B_2[j*2+2].

Такие конструкции могут быть использованы при написании текста программы так же, как и идентификаторы обычных переменных, имеющих тип элемента массива.

Значение, полученное в результате вычисления **выражения,** должно быть совместимо по присваиванию с индексирующим типом, указанным в описании массива.




После выполнения оператора: A[5]:=2.4
В элемент массива A с индексом 5 будет записано значение 2.4.

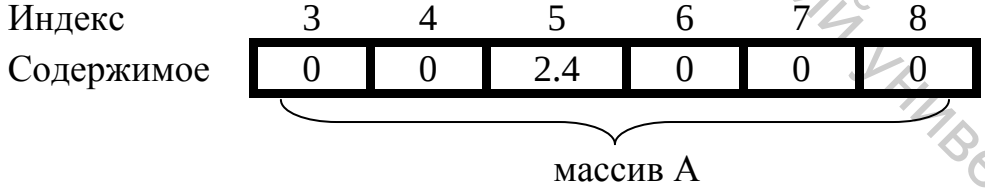



Рисунок 14 – Доступ к элементам одномерного массива

В случае доступа к элементу многомерного массива нужно использовать несколько, указанных через запятую, выражений внутри индекса.

**Синтаксис многомерного составного индекса:**
**Ид_переменной_массива[выражение1, выражение1, ...]**

Пример конструкций, используемых для доступа к элементам двумерных массивов:

B[2,2], Mass22[i+1,j-1], C_2d[2*i,j*2+2].

Например, имеем массив B (рисунок 15 а), после выполнения оператора: B[2,3]:= 1, в элемент массива B с первым индексом (номером столбца), равным 2 и со вторым индексом (номером строки), равным 3, будет записано значение 1 (рисунок 15 б).

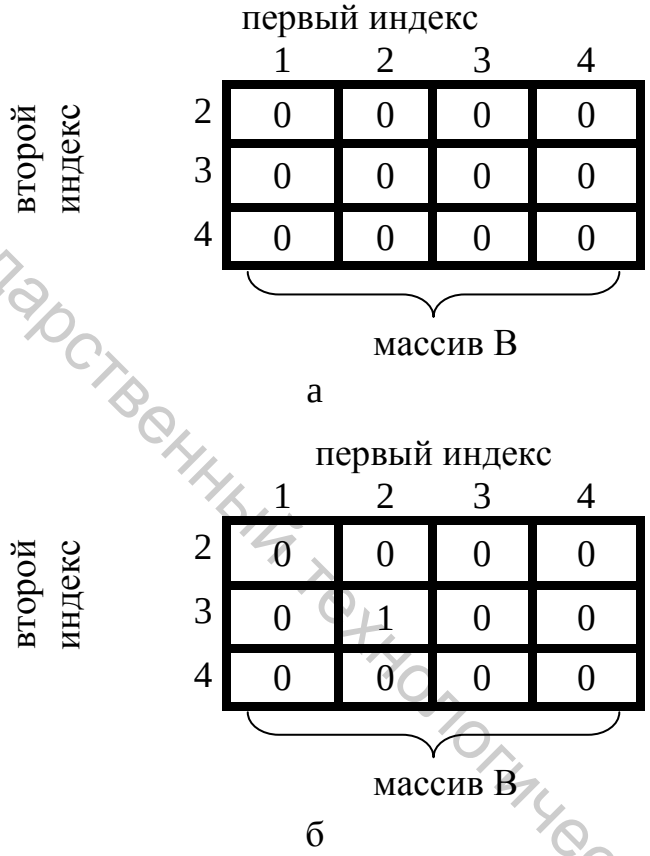



Рисунок 15 – Доступ к элементам двумерного массива

## Строковые типы

Значением строкового типа является последовательность символов, что фактически является одномерным массивом, состоящим из элементов символьного типа (**char**).

**Синтаксис описания строкового типа**:

**String [N]**

**N** – это **выражение-константа,** результат вычисления которой определяет максимально возможное количество символов в строке. Реальное количество символов в строке может изменяться в процессе работы программы, например,

после выполнения оператора присваивания. Текущее количество символов в строке можно выяснить с помощью функции **Length**.

**Пример описания переменных строкового типа:**

```
const
n = 6;
var
s: string[10];
s1, s2: string [n];
S3: string[n+2];
```

К символам в строке можно получить доступ как к компонентам массива – с помощью индекса, который фактически определяет позицию символа в строке.

Первый символ в строке имеет индекс 1, второй – 2 и т. д. Последний символ имеет индекс **N.**

Например, имеется пустая строка s1 (рисунок 16 а). Фактическая длина строки s1 равна 0.

После выполнения оператора: s1:=’end .’ элементы строки s будут заполнены так как показано на рисунке 16 б. Теперь фактическая длина строки s1 равна 5.

После выполнения оператора: s1[4]:=’-’; в строке s1 вместо находившегося ранее на позиции с номером 4 символа пробел (#13) появится символ ’-’ (рисунок 16 в), фактическая длина строки не изменится.

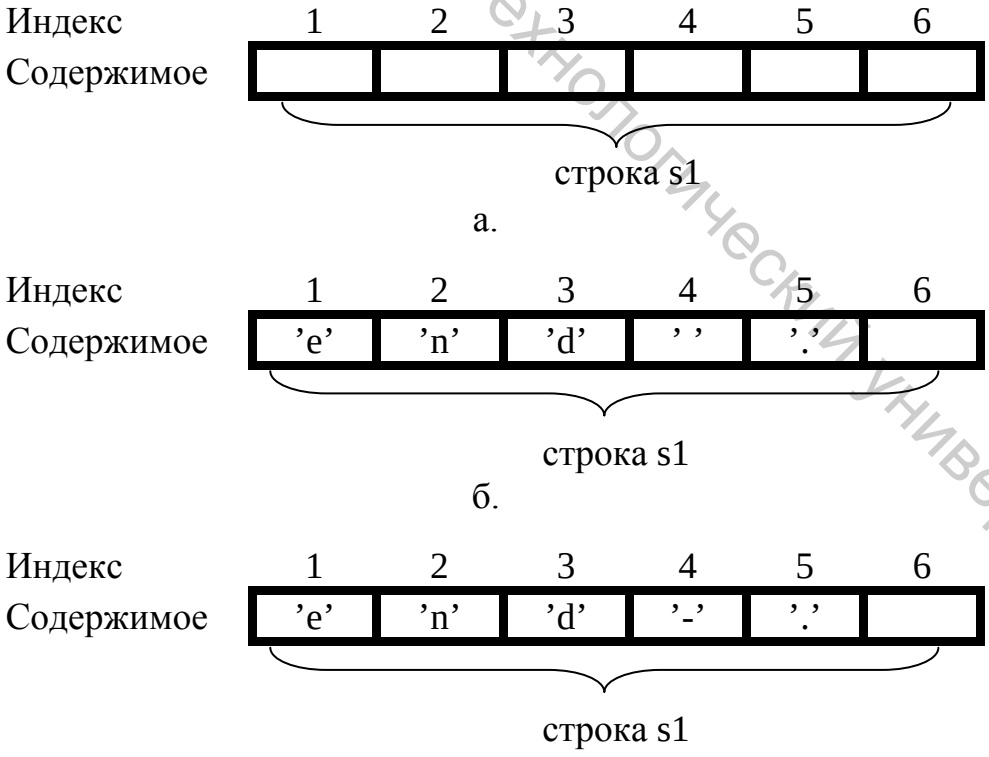



Рисунок 16 – Доступ к символам строки

### Стандартные подпрограммы обработки строк

Функция **Concat(s1, [s2, ..., sn] ; string** – возвращает строку, которая объединяет в себе последовательно строки **s1, s2, ... , sn**.

Функция **Copy(s : string; нач_инд : integer; колич:integer)** – возвращает часть строки **s**, начиная с символа с индексом **нач_инд** длиной в **колич** символов.

Процедура **Delete(var s : string; нач_инд: integer; колич: integer)** – удаляет из строки **s**, часть символов, начиная с символа с индексом **нач_инд** длиной в **колич** символов.

Процедура **Insert( вст_стр: string; var нов_стр: string; нач_инд : integer)** – добавляет строку **вст_стр** в строку **нов_стр,** начиная с символа с индексом **нач_инд**.

Функция **Length(s : string): integer** – возвращает фактическую длину строки **s**.

Функция **Pos(поиск_стр, s: string):byte** – возвращает позицию вхождения строки **поиск_стр** в строку **s**. Если строка **поиск_стр** в строке **s** не найдена, то возвращается значение 0.

Процедура **Str(x, var s :string)** – преобразует численное значение **x** в его строковое представление и помещает полученное строковое значение в переменную **s**.

Процедура **Val(s : string; var v; var код :integer)** – преобразует строковое значение **s** в его численное представление и помещает полученное значение в переменную **v** (переменная должна быть целого или вещественного типа). В переменной **код** возвращается индекс первого символа в строке **s,** который невозможно преобразовать в число. Если переменная **код** после выполнения процедуры равна 0, то все символы строки **s** были успешно преобразованы.

Пример использования подпрограмм обработки строк:

Имеется строка вида 'SonyEricsson 700i - 180.33$ (+чехол)'. Необходимо представить цену товара в рублях.

```
program StrEdit;
var S,ZstrD,ZstrR: String;
nach, kon:integer;
kod:integer;
Znum:real;
begin
 Writeln('Введите строку');
 readln(S);
 nach:= Pos('-', S)+2;
 kon:= Pos('$', S);
 ZstrD:= copy(S, nach, kon-nach);
```

```
 val(ZstrD, Znum,kod);
 Znum:=Znum*2140;
 Str(Znum:5:2,ZstrR);
 delete(S, nach, kon-nach+1);
 insert(concat(ZstrR,'руб.'),s,nach);
 writeln(s);
end.
```

## Алгоритмы обработки одномерных массивов

Одномерные массивы – одни из наиболее часто используемых структур хранения технических данных (например, значения какого-либо параметра, изменяющегося во времени).

### Анализ элементов массива

Обычно алгоритмы данной группы осуществляют последовательный перебор и накопление сведений об элементах массива с последующим вычислением характеристик всей последовательности значений.

К задачам подобного рода можно отнести: вычисления суммы и среднеарифметического значений элементов, нахождение количества определённых элементов, вычисление статистических характеристик (отклонений, медиан, дисперсий) и т.п.

Некоторые из этих задач решаются с помощью нескольких последовательных переборов всех элементов массива.

Рассмотрим в качестве примера программу, которая вычисляет процент положительных значений среди ненулевых элементов массива.

Алгоритм решения задачи представлен на рисунке 17.

На рисунке 17 блоки 2, 3, 4 используются для ввода исходных данных. Блок 3 организует цикл с параметром i, который, изменяясь от 1 n (n – количество элементов в массиве), перебирает все индексы элементов массива. Блок 4 в каждом новом проходе цикла заполняет последовательно каждый из элементов массива.

Блок 5 инициализирует служебные переменные: chNO используется для подсчёта количества ненулевых элементов, chP используется для подсчёта количества положительных элементов.

Далее блок 6 организует перебор всех элементов массива, аналогично блоку 3. В этом цикле каждый элемент проверяется на положительность и отличность от 0. Если элемент положительный (блок 7), то увеличивается на 1 значение переменной chP (блок 8). Затем снова выполняется проверка: если

элемент не равен нулю (блок 9), то увеличивается на 1 значение переменной chNO (блок 10).

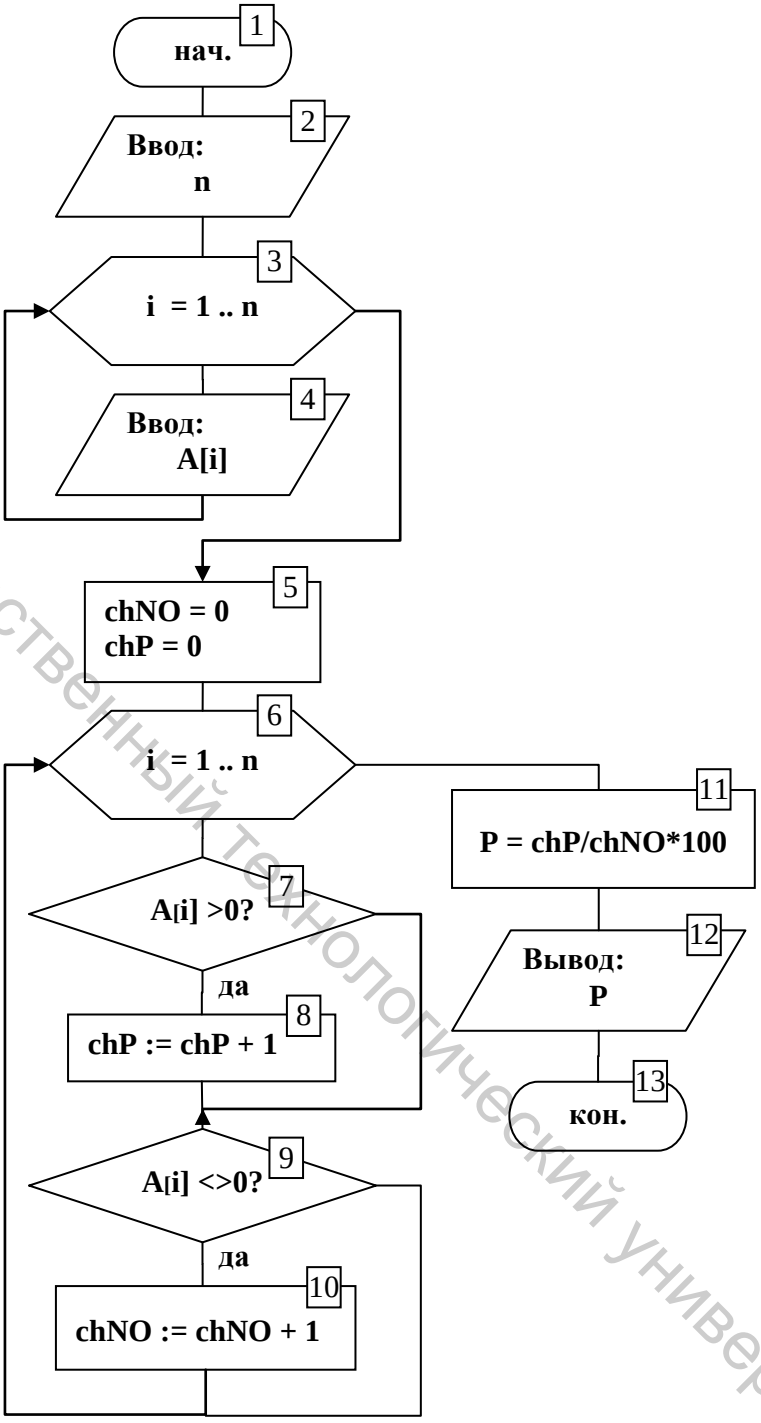



Рисунок 17 – Блок-схема алгоритма вычисления процента положительных значений среди ненулевых элементов массива

После окончания цикла вычисляем требуемый процент (блок 11) и выводим полученное значение (блок 12).

В таблице 8 приведена трассировка части алгоритма (рис. 17). Каждая строка трассировочной таблицы соответствует выполнению определённого алгоритмического блока. В колонках представлены значения определённых переменных, или выражений, необходимых для пояснения алгоритма. Для удобства восприятия жирным в таблице выделены значения, которые изменились во время выполнения соответствующего блока.

Таблица 8 – Трассировка алгоритма подсчёта количества положительных и ненулевых элементов

| № блока | i | chNO | chP | A[i] | A | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | 1 | 2 | 3 | 4 | 5 |
| ... | | | | | | | | | |
| 6 | 1 | 0 | 0 | -2 | -2 | 2 | 0 | 4 | -1 |
| 7 | 1 | 0 | 0 | -2 | -2 | 2 | 0 | 4 | -1 |
| 9 | 1 | 0 | 0 | -2 | -2 | 2 | 0 | 4 | -1 |
| 10 | 1 | **1** | 0 | -2 | -2 | 2 | 0 | 4 | -1 |
| 6 | **2** | 1 | 0 | **2** | -2 | 2 | 0 | 4 | -1 |
| 7 | 2 | 1 | 0 | 2 | -2 | 2 | 0 | 4 | -1 |
| 8 | 2 | 1 | **1** | 2 | -2 | 2 | 0 | 4 | -1 |
| 9 | 2 | 1 | 1 | 2 | -2 | 2 | 0 | 4 | -1 |
| 10 | 2 | **2** | 1 | 2 | -2 | 2 | 0 | 4 | -1 |
| 6 | **3** | 2 | 1 | **0** | -2 | 2 | 0 | 4 | -1 |
| 7 | 3 | 2 | 1 | 0 | -2 | 2 | 0 | 4 | -1 |
| 9 | 3 | 2 | 1 | 0 | -2 | 2 | 0 | 4 | -1 |
| 10 | 3 | 2 | 1 | 0 | -2 | 2 | 0 | 4 | -1 |
| 6 | **4** | 2 | 1 | **4** | -2 | 2 | 0 | 4 | -1 |
| 7 | 4 | 2 | 1 | 4 | -2 | 2 | 0 | 4 | -1 |
| 8 | 4 | 2 | **2** | 4 | -2 | 2 | 0 | 4 | -1 |
| 9 | 4 | 2 | 2 | 4 | -2 | 2 | 0 | 4 | -1 |
| 10 | 4 | **3** | 2 | 4 | -2 | 2 | 0 | 4 | -1 |
| 6 | **5** | 3 | 2 | **-1** | -2 | 2 | 0 | 4 | -1 |
| 7 | 5 | 3 | 2 | -1 | -2 | 2 | 0 | 4 | -1 |
| 9 | 5 | 3 | 2 | -1 | -2 | 2 | 0 | 4 | -1 |
| 10 | 5 | **4** | 2 | -1 | -2 | 2 | 0 | 4 | -1 |
| 6 | 5 | 4 | 2 | -1 | -2 | 2 | 0 | 4 | -1 |
| 11 | 5 | 4 | 2 | -1 | -2 | 2 | 0 | 4 | -1 |
| ... | | | | | | | | | |

Реализация алгоритма в программу выглядит следующим образом:

```
program Proz;
 var
 A: array [1..10] of real ;
 chNO,chP,n,i:integer;
 begin
 writeln('введите размер массива');
 readln(n);
 for i:=1 to n do
       readln(A[i]);
 chNO := 0;
 chP := 0;
 for i:=1 to n do
       begin
        if A[i]>0 then
            chP := chP + 1;
        if A[i]<>0 then
            chNO := chNO + 1;
       end;
writeln('процент положительных значений -', chP/chNO*100);
end.
```

Рассмотрим ещё один пример: необходимо найти все чётные элементы в массиве A и поместить их в массив B.

Алгоритм решения задачи представлен на рисунке 18.

В целом рассматриваемый алгоритм (рис. 18) схож с алгоритмом поиска определённых элементов. Разница состоит в том, что значения найденных элементов не накапливаются для последующей обработки, а перемещаются в новый массив.

Для индексов найденных элементов используется независимый счётчик j. В начале перебора ему присваивается значение 1 (блок 5). При нахождении в исходном массиве первого чётного элемента выполняется его запись в массив B под номером j (j = 1), и значение счётчика j увеличивается на единицу и становится равным 2 (блоки 7, 8). Соответственно, следующий найденный элемент будет помещён в массив B под номером 2.

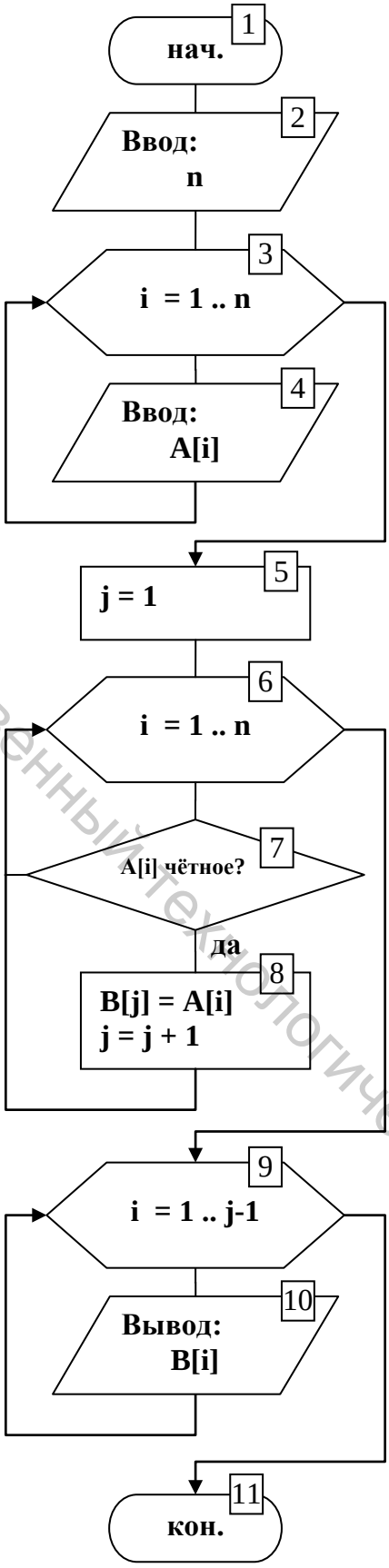



Рисунок 18 – Блок-схема алгоритма поиска
чётных элементов

В таблице 9 приведена трассировка части алгоритма (рис. 18). Каждая строка трассировочной таблицы соответствует состоянию переменных после выполнения определённого алгоритмического блока. В колонках представлены значения определённых переменных, или выражений, необходимых для пояснения алгоритма. Для удобства восприятия жирным в таблице выделены значения, которые изменились во время выполнения соответствующего блока.

Таблица 9 – Трассировка алгоритма копирования чётных элементов из массива A в массив B

| № блока | A[i] | i | A | | | | | j | B | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 1 | 2 | 3 | 4 | 5 | | 1 | 2 | 3 | 4 | 5 |
| ... | | | | | | | | | | | | | |
| 5 | - | - | 3 | 6 | 8 | 3 | 2 | 1 | | | | | |
| 6 | 3 | 1 | 3 | 6 | 8 | 3 | 2 | 1 | | | | | |
| 7 | 3 | 1 | 3 | 6 | 8 | 3 | 2 | 1 | | | | | |
| 6 | **2** | **2** | 3 | 6 | 8 | 3 | 2 | 1 | | | | | |
| 7 | 2 | 2 | 3 | 6 | 8 | 3 | 2 | 1 | | | | | |
| 8 | 6 | 2 | 3 | 6 | 8 | 3 | 2 | **2** | **6** | | | | |
| 6 | **8** | **3** | 3 | 6 | 8 | 3 | 2 | 2 | 6 | | | | |
| 7 | 8 | 3 | 3 | 6 | 8 | 3 | 2 | 2 | 6 | | | | |
| 8 | 8 | 3 | 3 | 6 | 8 | 3 | 2 | **3** | 6 | **8** | | | |
| 6 | **3** | **4** | 3 | 6 | 8 | 3 | 2 | 3 | 6 | 8 | | | |
| 7 | 3 | 4 | 3 | 6 | 8 | 3 | 2 | 3 | 6 | 8 | | | |
| 6 | **2** | **5** | 3 | 6 | 8 | 3 | 2 | 3 | 6 | 8 | | | |
| 7 | 2 | 5 | 3 | 6 | 8 | 3 | 2 | 3 | 6 | 8 | | | |
| 8 | 2 | 5 | 3 | 6 | 8 | 3 | 2 | 4 | 6 | 8 | **2** | | |
| 6 | 2 | 5 | 3 | 6 | 8 | 3 | 2 | 4 | 6 | 8 | 2 | | |
| ... | | | | | | | | | | | | | |

Реализация алгоритма в программу выглядит следующим образом:

```
program A_Bmas;
var
B,A:array[1..100] of integer;
n,i,j:integer;
begin
 write('n>');
 read(n);
 for i:=1 to n do
 begin
  write('A[',i,']→');
```

```
 read(A[i]);
end;
j:=1;
for i:=1 to n do
 begin
  if A[i] mod 2 = 0 then
     begin
      B[j]:=A[i];
      j:=j+1;
     end;
 end;
for i:=1 to j-1 do
 writeln(B[i]);
end.
```

**Поиск определённых элементов**

Общая формулировка задач этой группы может звучать так: в одномерном массиве найти определённый элемент, или элементы, удовлетворяющие определённому условию. Найденные элементы могут быть сохранены в отдельный массив для дальнейшей обработки.

Для алгоритмов данной группы не характерно использование обычного последовательного перебора всех элементов массива. Последовательный перебор может быть использован, например, для поиска элементов массива, если их особенность не зависит от расположения в массиве, или от других элементов (например, чётность, или кратность 3).

К задачам подобного рода можно отнести: поиск максимального и минимального элементов, поиск локальных экстремумов и т.п.

В качестве примера рассмотрим задачу поиска максимального значения, которая часто выступает в качестве составной части при решении других задач.

Основная идея алгоритма заключается в последовательном парном сравнении всех элементов массива.

Алгоритм решения задачи представлен на рисунке 19.

Переменная max используется для хранения текущего максимального значения. Вначале в качестве текущего максимального значения принимается значение первого элемента массива (блок 5).

Затем организуется перебор элементов массива, начиная со второго (блок 6). В теле цикла текущий максимальный элемент сравнивается с очередным элементом массива, и большее из этих значений сохраняется в качестве нового текущего максимального в переменной max (блоки 7, 8). Таким образом, при первом проходе цикла выбирается максимальное значение из двух первых элементов. На втором проходе полученное значение сравнивается с третьим элементом, и т. д. После окончания цикла в переменной max будет находиться максимальное из всех значений элементов массива.

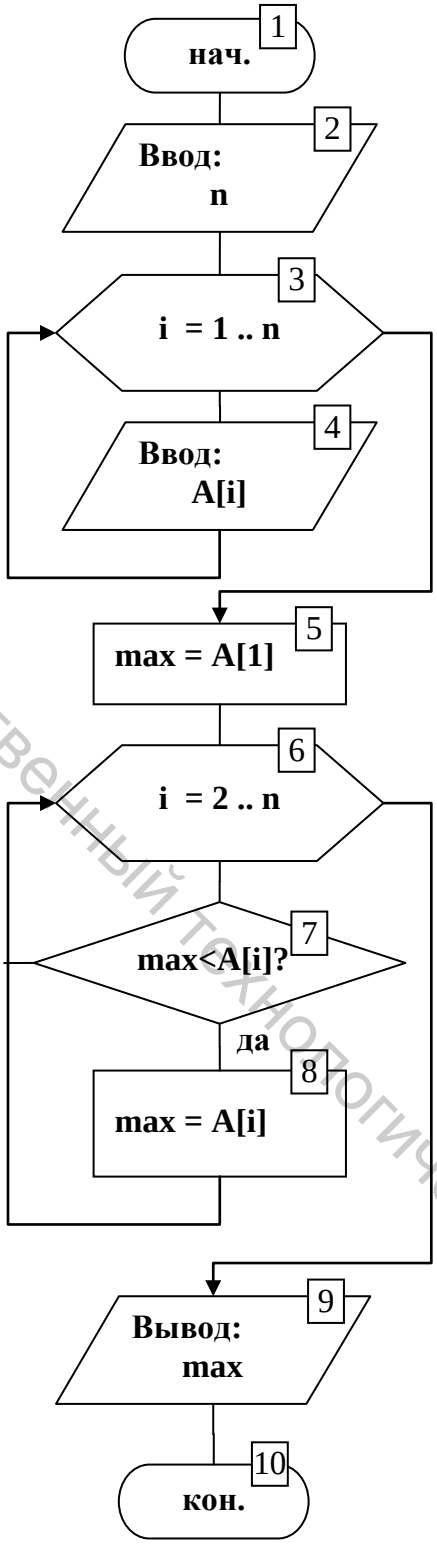





Рисунок 19 – Блок-схема алгоритма поиска максимального элемента

В таблице 10 представлена трассировка основной части алгоритма, представленного на рисунке 19.

Таблица 10 – Трассировка алгоритма поиска максимального элемента

| № блока | i | A[i] | max | A | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | 1 | 2 | 3 | 4 | 5 |
| ... | | | | | | | | |
| 5 | - | - | **2** | 2 | 1 | 5 | 8 | 2 |
| 6 | **2** | **1** | 2 | 2 | 1 | 5 | 8 | 2 |
| 7 | 2 | 1 | 2 | 2 | 1 | 5 | 8 | 2 |
| 6 | **3** | **5** | 2 | 2 | 1 | 5 | 8 | 2 |
| 7 | 3 | 5 | 2 | 2 | 1 | 5 | 8 | 2 |
| 8 | 3 | 5 | **5** | 2 | 1 | 5 | 8 | 2 |
| 6 | **4** | **8** | 5 | 2 | 1 | 5 | 8 | 2 |
| 7 | 4 | 8 | 5 | 2 | 1 | 5 | 8 | 2 |
| 8 | 4 | 8 | **8** | 2 | 1 | 5 | 8 | 2 |
| 6 | **5** | **2** | 8 | 2 | 1 | 5 | 8 | 2 |
| 7 | 5 | 2 | 8 | 2 | 1 | 5 | 8 | 2 |
| 6 | 5 | 2 | 8 | 2 | 1 | 5 | 8 | 2 |
| ... | | | | | | | | |

Реализация алгоритма в программу выглядит следующим образом:

```
program Max;
var
A:array[1..100] of real;
n,i:integer;
max:real;
begin
 write('n>');
 read(n);
 for i:=1 to n do
 begin
  write('A[',i,']>');
  readln(A[i]);
 end;
 max:=A[1];
 for i:=2 to n do
  begin
   if max<A[i] then
     max:=A[i];
  end;
  writeln('Max=',max:4:2);
end.
```

## Последовательная сортировка одномерных массивов

Сортировку массива можно отнести к алгоритмам преобразования массивов. К таким алгоритмам также можно отнести алгоритмы, осуществляющие различного рода перестановки элементов массива, а также вставку и удаление элементов.

Сортировка представляет собой процесс упорядочения элементов в массиве в порядке возрастания или убывания их значений.

Основная идея последовательной сортировки заключается в следующем. Вначале выбирается минимальный элемент из всего массива, найденный элемент перемещается в начало массива (в позицию с индексом, равным 1). Затем снова проводится поиск минимального элемента, но поиск начинается с позиции с индексом 2, так что элемент с индексом 1 при поиске не учитывается. Таким образом, будет найден второй по величине элемент массива. Затем это элемент перемещается в позицию с индексом 2. Теперь необработанная часть массива, в которой будет произведён поиск минимального значения, начинается с элемента с индексом 3. Эта последовательность операций выполняется до тех пор, пока в необработанной части массива не останется один элемент.

Рассмотрим алгоритм, представленный на рисунке 20.

Переменная k служит для указания вершины необработанной части массива, т.е. места, куда нужно поместить очередной минимальный элемент из этой необработанной части. В начале работы k = 1, следовательно, поиск минимального элемента будет проходить по всему массиву.

Тело основного цикла (блоки 5, 6, 7, 8) содержит алгоритм поиска минимального элемента. Его отличает от алгоритма, представленного на рисунке 19, следующее: осуществляется поиск не максимального, а минимального элемента (поскольку сортировка производится по возрастанию элементов), поиск осуществляется не по всему массиву, а только в его части (элементы с индексами от k+1 до n), помимо самого минимального значения выясняется и его положение в массиве (переменная Nmin). Далее в блоке 9 происходит перестановка элементов массива, так что найденный минимальный элемент оказывается на вершине необработанной части массива, под индексом k.

После этого увеличивается на единицу значение переменной k, и весь алгоритм повторяется для части элементов массива, начинающихся с индекса 2. Перемещённый в первую позицию минимальный элемент в последующей обработке массива не используется.

Так повторяется до тех пор, пока в необработанной части массива не остаётся один элемент.

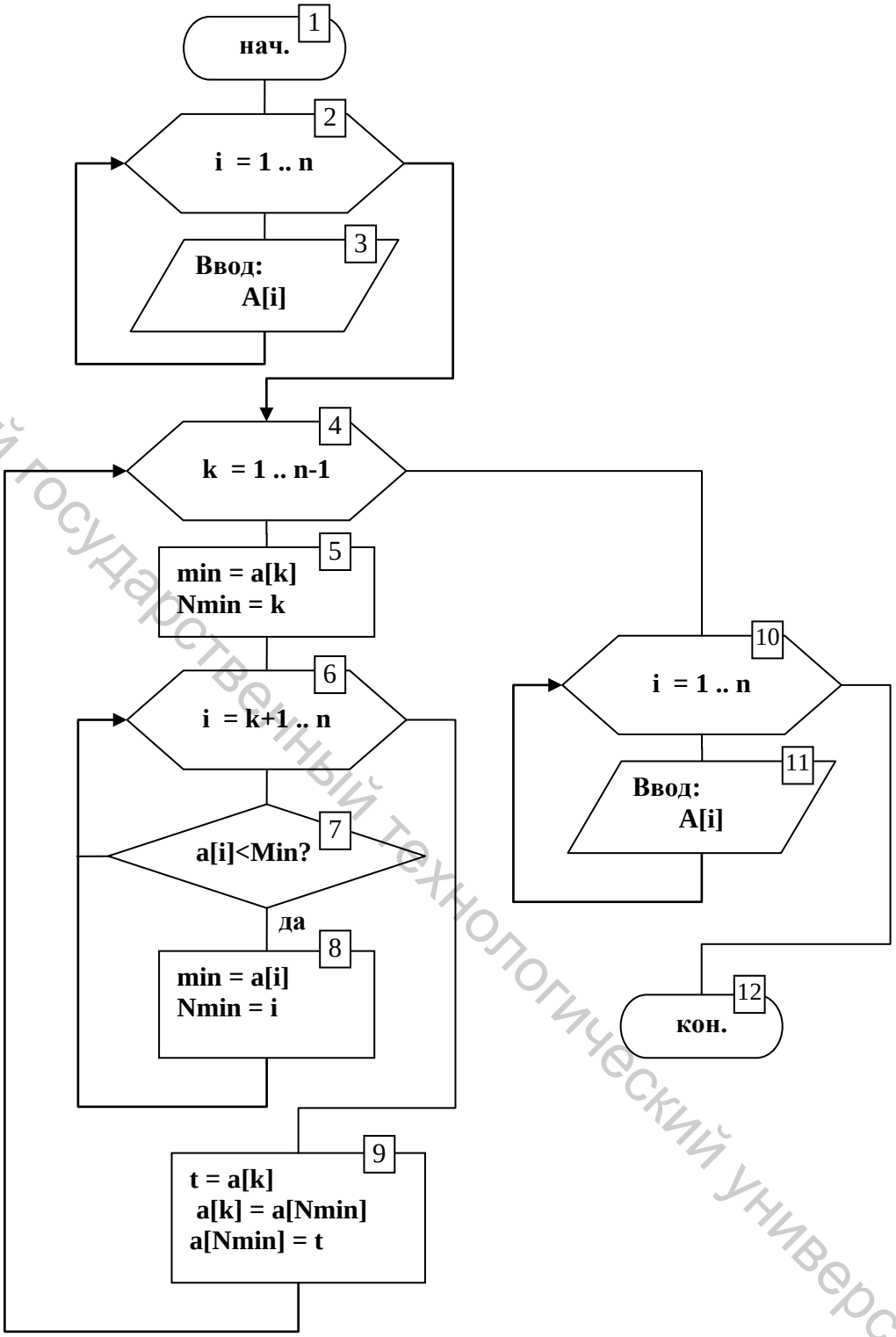





Рисунок 20 – Блок-схема алгоритма
последовательной сортировки

Приведём трассировку части представленного алгоритма (таблица 11). Вложенный цикл (блоки 6, 7, 8) предназначен для нахождения минимального элемента массива, среди элементов с индексами от k+1 до n. Это алгоритм

рассматривался ранее, поэтому в трассировке блоки 6, 7, 8 будут отображены одной строкой, элемент, находящийся на вершине необработанной части массива A, помечен серой заливкой ячейки.

Таблица 11 – Трассировка алгоритма последовательной сортировки одномерного массива

| № блока | k | A[k] | Nmin | min | A[Nmin] | A | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | 1 | 2 | 3 | 4 | 5 |
| ... | | | | | | | | | | |
| 4 | **1** | **22** | - | - | - | 22 | 33 | 11 | 55 | 44 |
| 5 | 1 | 22 | **1** | **22** | **22** | 22 | 33 | 11 | 55 | 44 |
| 6,7,8 | 1 | 22 | **3** | **11** | **11** | 22 | 33 | 11 | 55 | 44 |
| 9 | 1 | 22 | 3 | 11 | 11 | **11** | 33 | **22** | 55 | 44 |
| 4 | **2** | **33** | - | - | - | 11 | 33 | 22 | 55 | 44 |
| 5 | 2 | 33 | 2 | 33 | 33 | 11 | 33 | 22 | 55 | 44 |
| 6,7,8 | 2 | 33 | **3** | **22** | **22** | 11 | 33 | 22 | 55 | 44 |
| 9 | 2 | 33 | 3 | 22 | 22 | 11 | **22** | **33** | 55 | 44 |
| 4 | **3** | **33** | - | - | - | 11 | 22 | 33 | 55 | 44 |
| 5 | 3 | 33 | **3** | **33** | **33** | 11 | 22 | 33 | 55 | 44 |
| 6,7,8 | 3 | 33 | 3 | 33 | 33 | 11 | 22 | 33 | 55 | 44 |
| 9 | 3 | 33 | 3 | 33 | 33 | 11 | 22 | 33 | 55 | 44 |
| 4 | **4** | **55** | - | - | - | 11 | 22 | 33 | 55 | 44 |
| 5 | 4 | 55 | 4 | 55 | 55 | 11 | 22 | 33 | 55 | 44 |
| 6,7,8 | 4 | 55 | **5** | **44** | **44** | 11 | 22 | 33 | 55 | 44 |
| 9 | 4 | 55 | 5 | 44 | 44 | 11 | 22 | 33 | **44** | **55** |
| 4 | 4 | 55 | 5 | 44 | 44 | 11 | 22 | 33 | 44 | 55 |
| 10 | | | | | | | | | | |
| ... | | | | | | | | | | |

Реализация алгоритма в программу выглядит следующим образом:

```
program sort1;
const n=5;
var a:array [1..n] of real;
   k,m,i,Nmin:integer;
t,min:real;
Begin
 for i:=1 to n do {ввод массива}
   read(a[i]);
 for k:=1 to n-1 do
  begin
```

```
  min:=a[k];
  Nmin:=k;
   for i:=k+1 to n do
     if a[i]<Min then
      begin
       min:=a[i];
       Nmin:=i;
      end;
        t:=a[k];
        a[k]:=a[Nmin];
        a[Nmin]:=t;
  end;
 for i:=1 to n do
    write(a[i],' ');
End.
```

## Алгоритмы обработки двумерных массивов

Двумерные массивы в качестве структур для хранения данных используются в алгоритмах решения математических задач, связанных с матричными операциями, для моделирования различных планов (местности или зданий), для исследования процессов с двумя входными параметрами и т. п.

В некоторых из этих алгоритмов при обработке строк или столбцов можно использовать, как составные части, алгоритмы обработки одномерных массивов.

### Анализ элементов массива

Рассмотрим в качестве примера программу, которая вычисляет среднеарифметическое элементов каждой строки двумерного массива.

Первая часть алгоритма, представленного на рисунке 21 (блоки 2, 3, 4), предназначена для ввода элементов массива. Счётчик внешнего цикла (блок 2) задаёт номер столбца, после этого внутренний цикл (блок 3) обеспечивает перебор всех элементов текущего столбца. Таким образом, эта часть алгоритма обеспечивает ввод массива по столбцам.

Далее, в обрабатывающей части алгоритма, внешний цикл (блок 5) задаёт значение счётчика j, который определяет номер обрабатываемой строки. Внутри данного цикла находится алгоритм вычисления суммы элементов одномерного массива (блоки 6, 7, 8). После этого значение переменной S, в которой сохранено значение суммы элементов строки, будет поделено на

количество столбцов массива (блок 9) и полученное среднеарифметическое будет выведено на экран (блок 10).

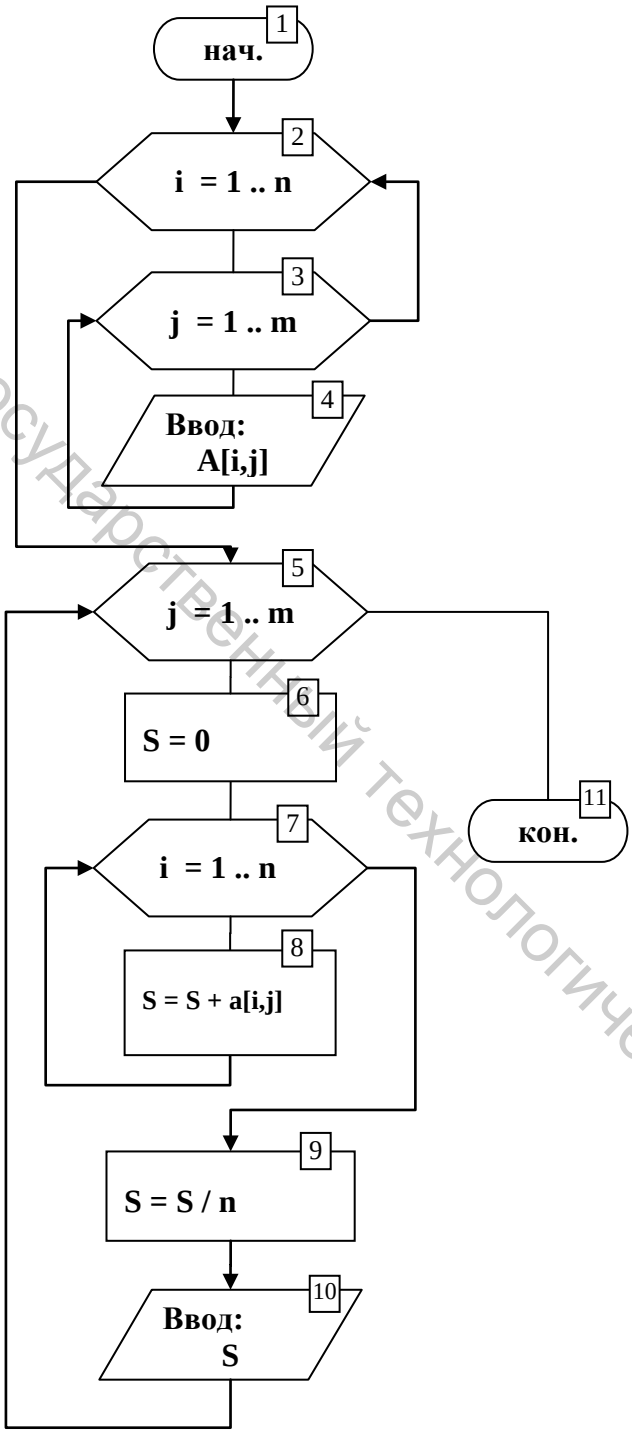



Рисунок 21 – Блок-схема алгоритма вычисления среднеарифметического значения элементов каждой строки двумерного массива

Приведём трассировку обрабатывающей части представленного алгоритма (таблица 12).

Таблица 12 – Трассировка алгоритма вычисления среднеарифметического значения элементов каждой строки двумерного массива

| № блока | j | i | A[i,j] | S | A |
|---|---|---|---|---|---|
| 5 | **1** | - | - | - | |
| 6 | 1 | - | - | **0** | |
| 7 | 1 | **1** | **2** | 0 | |
| 8 | 1 | 1 | 2 | **2** | |
| 7 | 1 | **2** | **4** | 2 | |
| 8 | 1 | 2 | 4 | **6** | |
| 7 | 1 | **3** | **6** | 6 | |
| 8 | 1 | 3 | 6 | **12** | |
| 7 | 1 | 3 | 6 | 12 | |
| 9, 10 | 1 | 3 | 6 | **3** | |
| 5 | **2** | - | - | 3 | |
| 6 | 2 | - | - | **0** | |
| 7 | 2 | **1** | **3** | 0 | |
| 8 | 2 | 1 | 3 | **3** | |
| 7 | 2 | **2** | 3 | 3 | |
| 8 | 2 | 2 | 3 | **6** | |
| 7 | 2 | **3** | **9** | 6 | |
| 8 | 2 | 3 | 9 | **15** | |
| 7 | 2 | 3 | 9 | 15 | |
| 9, 10 | 2 | 3 | 9 | **5** | |
| 5 | **3** | - | - | 5 | |
| 6 | 3 | - | - | **0** | |
| 7 | 3 | **1** | **1** | 0 | |
| 8 | 3 | 1 | 1 | **1** | |
| 7 | 3 | **2** | **0** | 1 | |
| 8 | 3 | 2 | 0 | **1** | |
| 7 | 3 | **3** | **2** | 1 | |
| 8 | 3 | 3 | 2 | **3** | |
| 7 | 3 | 3 | 2 | 3 | |
| 9, 10 | 3 | 3 | 2 | **1** | |
| 5 | 3 | - | - | 1 | |
| 11 | - | - | - | 1 | |

| индексы | 1 | 2 | 3 |
|---|---|---|---|
| 1 | **2** | **4** | **6** |
| 2 | **3** | **3** | **9** |
| 3 | **1** | **0** | **2** |

| индексы | 1 | 2 | 3 |
|---|---|---|---|
| 1 | **2** | **4** | **6** |
| 2 | **3** | **3** | **9** |
| 3 | **1** | **0** | **2** |

| индексы | 1 | 2 | 3 |
|---|---|---|---|
| 1 | **2** | **4** | **6** |
| 2 | **3** | **3** | **9** |
| 3 | **1** | **0** | **2** |



Реализация алгоритма в программу выглядит следующим образом:

```
program sort2;
uses crt;
const n=5; m=4;
```

```
var a:array [1..n,1..m] of real;
   j,i:integer;
   s:real;
Begin
 for i:=1 to n do
  for j:=1 to m do
   begin
    write('A[' , i , ' , ' , j , ' ] = ' );
    read(a[i, j]);
   end;
 for j:=1 to m do
  begin
   s:=0;
   for i:=1 to n do
        s:=s+a[i,j];
   s:=s/n;
   writeln('среднеарифметическое в строке №',j,' = ', s);
  end;
End.
```

## Записи

**Запись** – конечное множество поименованных элементов произвольного типа.

Элементы, из которых состоит запись, называют **полями.**

**Синтаксис описания типа запись:**

```
Record
Ид_поля1, Ид_поля2...: описание_типа|ид_типа;
...
End;
```

Запись удобно представлять в виде строки в таблице. Таким образом, множество записей представляет собой непосредственно таблицу. Средствами языка Pascal таблица может быть организована как одномерный массив с элементом типа **запись**.

Пример описаний структур данных типа запись:

```
type
{описание нового пользовательского типа PhRec}
      PhRec = record
        No: integer;
        Name : string [20];
        Phone : string [18];
      end;
```

```
var
{описание двух переменных типа PhRec}
  Zap1, Zap2: PhRec;
{описание массива из 20-ти элементов типа PhRec}
   Tab: array [1..20] of PhRec;
```

В данном примере переменные Zap1 и Zap2 представляются отдельными строками, а переменная Tab – таблицей, состоящей из 20-ти строк (рис. 22).

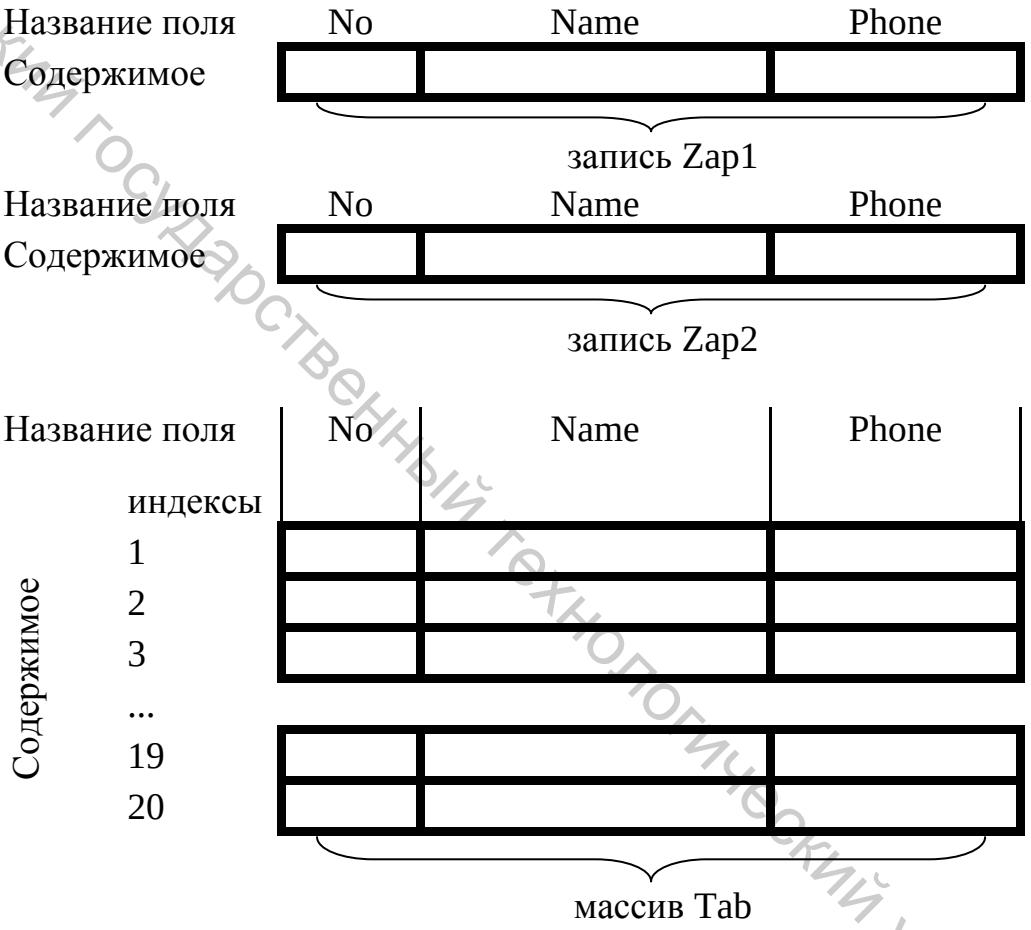



Рисунок 22 – Структура записей и массива записей

**Записи и десигнаторы полей**

Конкретное поле переменной-записи обозначается с помощью ссылки на переменную-запись, после которой указывается идентификатор поля.

**Синтаксис конструкции доступа к полю записи:**
**Ид_переменной_записи.Ид_поля**

Приведем несколько примеров использования десигнаторов полей:

```
{заполнение полей переменной Zap1}
Zap1.No := 1 ;
Zap1.Name := 'Иванов П.С.';
Zap1.Phone := '478556';
{запись полной строки в таблицу Tab}
Tab[1] = Zap;
```

Такие конструкции могут быть использованы при написании текста программы так же, как и идентификаторы обычных переменных, имеющих тип поля записи.

### Типизированные константы типа запись

Описание константы записи содержит заключенные в скобки и разделенные символом ; конструкции, определяющие значения каждого из полей.

Поля должны указываться в том же порядке, как они следуют в описании типа запись. Если запись содержит поля файлового типа, то для этого типа записи нельзя описать константу.

**Синтаксис выражения-константы двумерного массива:**
**(Ид_поля1: Знач1; Ид_поля2: Знач2;...)**

Пример описания константы записи:

```
type Point = record
    x,y: real;
   end;
Month = (Jan,Feb,Mar,Apr,May,Jun,Jly,Aug,Sep,Oct,Nov,Dec);
Date  = record
    d: 1..31;
    m: Month;
    y: 1900..1999;
   end;
const Origon : Point = (x: 0.0; y: 0.0);
SomeDay : Date = (d: 2; m: Dec; y: 1960);
```

### Блоки подпрограмм

Блоки подпрограмм позволяют включать в основной программный блок дополнительные алгоритмические блоки, которые могут быть запущены на исполнение из основной программы любое количество раз.

Блоки подпрограмм описывают самостоятельный законченный алгоритм.

В языке Pascal имеется два типа блока подпрограмм:

- процедура;
- функция.

### Подпрограммы-процедуры

В заголовке процедуры определяется её идентификатор и формальные параметры.

**Синтаксис описания блока процедуры:**

**{заголовок процедуры}**
**Procedure Ид_процедуры [(список_формальных_параметров)];**
**{раздел описаний}**
[**const** <описания констант>;]
[**type** <описания типов>;]
[**var** <описания переменных>;]
[<описания подпрограмм>]
**begin**
**{раздел операторов}**
**end;**

Запуск процедуры осуществляется с помощью оператора вызова процедуры:

**Синтаксис вызова процедуры:**
**Ид_процедуры [(список_фактических_параметров)];**

Если в содержащемся в процедуре операторе внутри модуля процедуры используется идентификатор процедуры, то процедура будет выполняться рекурсивно (будет при выполнении обращаться сама к себе). Такой вызов подпрограммы называют рекурсивным.

### Подпрограммы-функции

В заголовке функции определяется идентификатор функции, формальные параметры (если они имеются) и тип результата функции.

**{заголовок функции}**
**function Ид_функции (список_формальных_параметров): тип_результата;**
**{раздел описаний}**
[**const** <описания констант>;]

[**type** <описания типов>;]
[**var** <описания переменных>;]
[<описания подпрограмм>]

**begin**
**{раздел операторов}**
**end;**

Функция активизируется при вычислении выражения с функцией. Вызов функции в выражении имеет вид:

**Ид_функции [(список_фактических_параметров)];**

В модуле должен содержаться, по крайней мере, один оператор присваивания, в котором идентификатору функции присваивается значение. Результатом функции (тем значением, которое возвращается в точку вызова) является последнее присвоенное идентификатору функции значение.

Если такой оператор присваивания отсутствует или он не был выполнен, то значение, возвращаемое функцией, не определено.

**Синтаксис оператора возврата значения из функции:**

**Ид_функции := выражение**

Выражение должно быть совместимо по присваиванию с типом результата функции (см. раздел Совместимость в операциях присваивания стр. 19).

### Правила видимости идентификаторов

Наличие идентификатора какого-либо объекта языка (переменой, константы, типа, процедуры или функции) в разделе описаний означает определение идентификатора. Каждый раз, когда идентификатор встречаются в программе, он должен находиться в области действия этого описания.

Область действия идентификатора распространяется от их места его описания в тексте до конца текущего блока, включая все блоки, входящие в текущий блок. Так переменная i, описанная в основной программе (рис. 23), будет видна и в самой программе, и в процедурах Blok2 и Blok3.

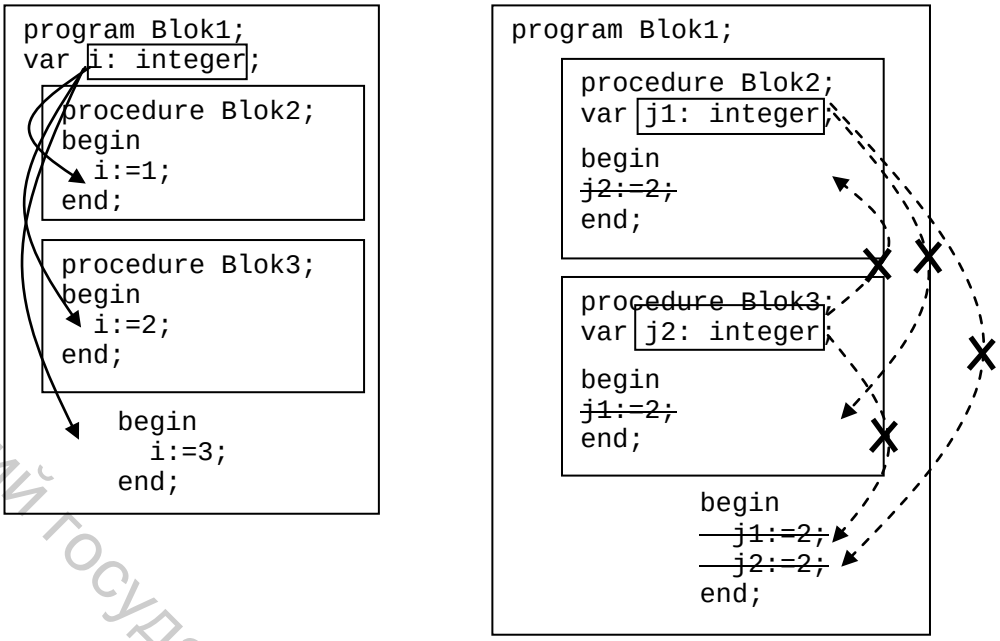



Рисунок 23 – Области видимости описаний

Описание идентификатора должно предшествовать любому вхождению идентификатора в текст программы.

Рассмотрим следующую ситуацию. Допустим, что в программе Blok1 вложен блок процедуры Blok2. Если в обоих блоках имеются описания переменных с одинаковым идентификатором, например, k, то в блоке Blok1 имеется доступ только к идентификатору k, который в нем описан, и аналогично внутри блока Blok2 имеется доступ только к идентификатору k, описание которого он содержит (рис. 24).

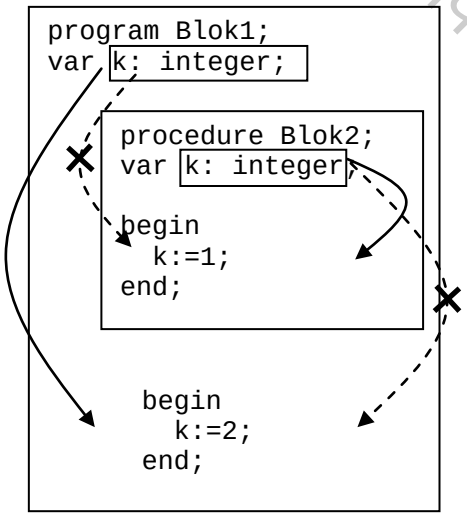



Рисунок 24 – Переопределение идентификатора во внутреннем блоке

**Параметры**

В описании процедуры или функции задается список формальных параметров. Каждый **формальный параметр**, описанный в списке формальных параметров, является локальной переменной по отношению к описываемой процедуре или функции, и в разделе операторов его можно использовать как обычную переменную.

**Фактический параметр** представляет собой определённое значение или переменную, с помощью которой инициализируется соответствующий формальный параметр.

**Параметры-значения**

Формальный параметр-значение обрабатывается как локальная по отношению к процедуре или функции переменная, за исключением того, что он получает свое начальное значение из соответствующего фактического параметра при активизации процедуры или функции. Изменения, которые претерпевает формальный параметр-значение, не влияют на значение фактического параметра.

**Синтаксис описания параметров-значений в списке формальных параметров подпрограммы:**

**Ид_парам1, Ид_парам2,...:тип_параметров;**

Соответствующее фактическое значение параметра-значения должно быть выражением. Фактический параметр должен иметь тип, совместимый по присваиванию с типом формального параметра-значения.

**Параметры-переменные**

Соответствующий фактический параметр в операторе вызова процедуры или функции должен быть ссылкой на переменную. При вызове процедуры или функции формальный параметр-переменная замещается переменной, указанной в вызове. Любые изменения в значении формального параметра-переменной отражаются на переменной, переданной в качестве фактического параметра.

**Синтаксис описания параметров-переменных в списке формальных параметров подпрограммы:**

**var Ид_парам11, Ид_парам12,...:тип_параметров;**

Тип фактического параметра должен быть **тождественным** типу формального параметра-переменной.

Параметр-переменная используется, когда значение должно передаваться из процедуры или функции вызывающей программе.

**Тождественность типов**

Тождественность типов требуется только для переменных фактических и формальных параметров-переменных при вызове процедур и функций.

Две переменных, T1 и T2, имеют тождественный тип, если выполняется одно из следующих условий:

T1 и T2 описаны совместно;

при описании T1 и T2 используется один и тот же идентификатор типа;

Примеры тождественных и нетождественных описаний:

```
type
 Mass = array [1..10] of real;
var
T1, T2 : array [1..10] of integer;
M1 : array [1..10] of integer;
M2 : array [1..10] of integer;
A1:Mass;  A2:Mass;
```

В примере переменные T1 и T2 тождественны, т. к. описаны совместно; переменные A1 и A2 тождественны, т. к. используют для описания один и тот же идентификатор Mass; переменные M1 и M2 нетождественны, т. к. конструкция **array [1..10] of integer** является описанием типа.

В качестве примера рассмотрим следующую задачу: Задано три числа: *n, m, k*. Необходимо вычислить три значения:

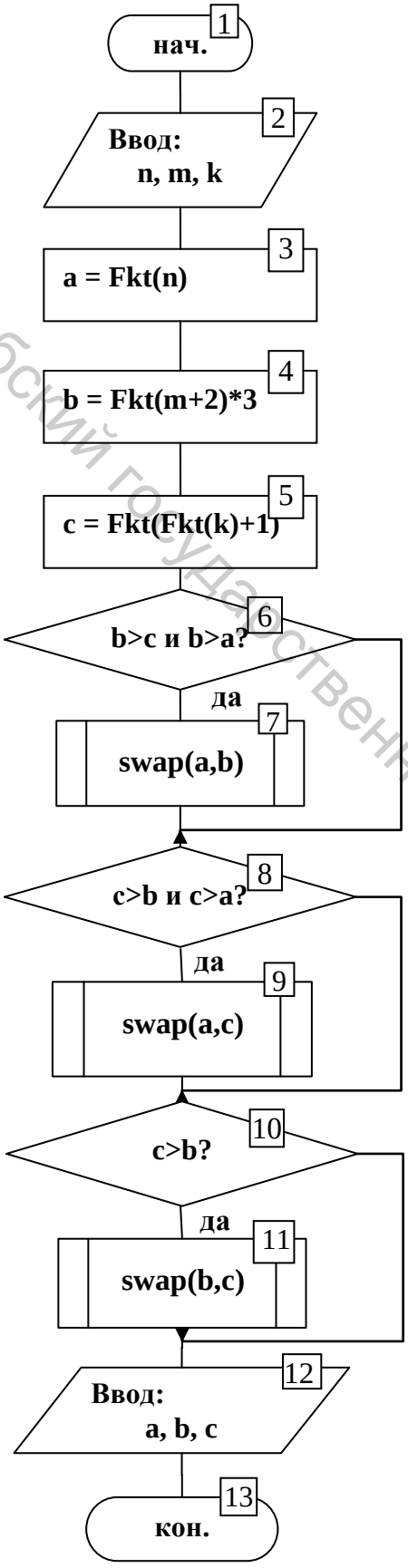
$$a = n!$$

$$b = (m+2)!*3$$

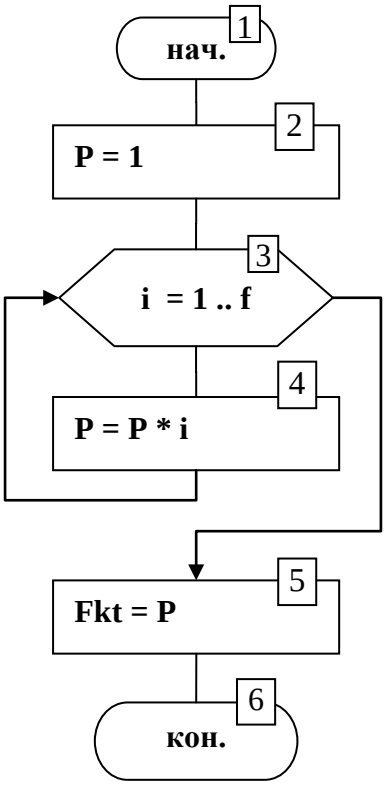
$$c = (k!+1)!$$

и поменять полученные значения так, чтобы в переменной *a* оказалось наибольшее из трёх, в *b* – среднее и в *c* – наименьшее, т.е. выстроить их в порядке убывания.

Алгоритм основной программы достаточно прост (рис. 25). После ввода исходных значений (блок 2) производятся вычисления значений *a, b, c* по заданным выражениям (блоки 3, 4, 5). Далее следует алгоритм сортировки массива из трёх значений. Блоки 6, 7, 8 и 9 предназначены для нахождения максимального из трёх значений. Если переменная *b* содержит максимальное из трёх значений (блок 6), то первая переменная (a) обменивается значением с переменной *b* (блок 7). Если максимальное значение находиться в переменной *c* (блок 8), то первая переменная (a) обменивается значением с переменной *c* (блок 9). Если максимальное значение находиться в переменной *a*, то блоки 7 и 8 останутся не выполненными.




**Fkt (f)**




**Swap(s1, s2 )**

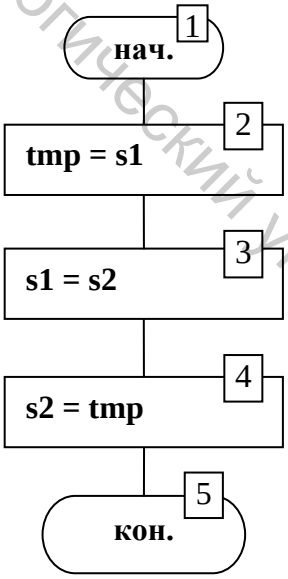





Рисунок 25 – Блок-схема алгоритма упорядочивания значений трёх элементов по убыванию

Таким образом, перед выполнением блока 10 максимальное из трёх значений будет находиться в переменной *a*.

Осталось поместить наибольшее из значений в переменных *b* и *c* в переменную *b* (блоки 10, 11).

Алгоритм функции Fkt(f), предназначенной для вычисления факториала заданного значения, по сути, является алгоритмом вычисления произведения целых чисел от 1 до f. Формальный параметр f является параметром-значением, т. е. при вызове функции в переменную f до начала работы подпрограммы будет записано значение фактического параметра – значения, указанного в вызове функции из основной программы.

Процедура Swap(s1, s2) выполняет перестановку значений двух переменных. Формальные параметры s1 и s2 являются параметрами-переменными. При вызове процедуры swap в качестве фактических параметров должны использоваться переменные. Любые изменения в значении переменных s1 и s2 в подпрограмме отражаются на переменных, переданных в качестве фактического параметра при вызове процедуры.

Рассмотрим подробнее, каким образом передаются параметры в вызываемые подпрограммы (табл. 13).

После запуска программы пользователь ввёл три значения: 2, 1, 2, которые были записаны соответственно в переменные n, m, k (строка 1).

**Первый вызов функции Fkt (строка 2):**

В основной программе (рис. 25, блок 3) при вызове функции передаётся значение переменной n (n = 2). Функция присваивает это значение параметру f. Таким образом, в начале работы алгоритма функции в переменной f находится значение 2.

По окончании работы функция возвращает основной программе значение факториала числа 2 (строка 10). Оператор в блоке 3 на рисунке 24 получает вычисленное значение факториала и записывает его в переменную a.

**Второй вызов функции Fkt (строка 11):**

Блок 4 (рис. 25) основной программы во второй раз вызывает функцию Fkt, но в этот раз передаёт в неё значение выражения m+2, равное трём. Полученное значение снова присваивается параметру f и снова исполняется алгоритм функции, который на этот раз возвращает значение 6 (факториал числа 3).

Оператор в блоке 4 на рисунке 24 получает это значение, заканчивает вычисление выражения 6*3 и записывает полученное значение 18 в переменную b (строка 21).

**Третий и четвёртый вызовы функции Fkt (строки 22-31):**

Первой из выражения Fkt(Fkt(k)+1)оператора в блоке 5 (рис. 25) вызывается вложенная функция Fkt(k). Фактический параметр, передаваемый в функцию – k (k = 2). Переменная f получает значение 2 (строка 22).

После окончания работы Функция возвращает 2 (факториал числа 2). Оператор в блоке 5 на рисунке 25 получает это значение, заканчивает вычисление выражения 2 + 1 и снова вызывает функцию Fkt (строка 26). На этот раз осуществляется внешний вызов Fkt(Fkt(k)+1). Результат работы

внутреннего вызова равен 2, выражение Fkt(k)+1 даёт значение 3, которое и передаётся в функцию Fkt при её четвёртом вызове из программы (строка 27).

После окончания работы Функция возвращает 6 (факториал числа 3). Оператор в блоке 5 на рисунке 24 получает вычисленное значение факториала, записывает его в переменную c и заканчивает свою работу.

**Первый вызов процедуры Swap (строка 33):**

Блок 7 (рис. 25) основной программы вызывает процедуру Swap для обмена значениями между переменными a и b. Эти переменные передаются в процедуру в качестве фактических параметров. Внутри процедуры переменные a и b становятся переменными s1 и s2. Любые изменения в значении s1 и s2 отражаются на переменных a и b соответственно (строки 35, 36).

**Второй вызов процедуры Swap (строка 39):**

Второй раз процедура Swap вызывается из блока 11 (рис. 25). В этот раз передаются в процедуру в качестве фактических параметров переменные b и c. Внутри процедуры переменная b становиться переменной s1, а переменная c – переменной s2. После окончания работы процедуры значения в переменных b и c поменяются местами.

Таблица 13 – Трассировка алгоритма упорядочивания значений трёх элементов по убыванию

| **№ п/п** | **№ блока** | **n** | **m** | **k** | **a** | **b** | **c** | **Подпрограммы** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 1 | **2** | **1** | **2** | - | - | - | | | | |
| **2** | 3 | 2 | 1 | 2 | - | - | - | Fkt(n) -> f:=n | | | |
| | | | | | | | | **№ блока** | **f** | **P** | **i** |
| **3** | 3 | 2 | 1 | 2 | - | - | - | 2 | **2** | **1** | - |
| **4** | 3 | 2 | 1 | 2 | - | - | - | 3 | 2 | 1 | **1** |
| **5** | 3 | 2 | 1 | 2 | - | - | - | 4 | 2 | 1 | 1 |
| **6** | 3 | 2 | 1 | 2 | - | - | - | 3 | 2 | 1 | **2** |
| **7** | 3 | 2 | 1 | 2 | - | - | - | 4 | 2 | **2** | 2 |
| **8** | 3 | 2 | 1 | 2 | - | - | - | 3 | 2 | 2 | 2 |
| **9** | 3 | 2 | 1 | 2 | - | - | - | 5 | 2 | 2 | 2 |
| **10** | 3 | 2 | 1 | 2 | **2** | - | - | P = 2 -> Fkt(n)= 2 | | | |
| **11** | 4 | 2 | 1 | 2 | 2 | - | - | Fkt(m+2) -> f:=m+2 | | | |
| | | | | | | | | **№ блока** | **f** | **P** | **i** |
| **12** | 4 | 2 | 1 | 2 | 2 | - | - | 2 | **3** | **1** | - |
| **13** | 4 | 2 | 1 | 2 | 2 | - | - | 3 | 3 | 1 | **1** |
| **14** | 4 | 2 | 1 | 2 | 2 | - | - | 4 | 3 | 1 | 1 |
| **15** | 4 | 2 | 1 | 2 | 2 | - | - | 3 | 3 | 1 | **2** |
| **16** | 4 | 2 | 1 | 2 | 2 | - | - | 4 | 3 | **2** | 2 |
| **17** | 4 | 2 | 1 | 2 | 2 | - | - | 3 | 3 | 2 | **3** |

Окончание таблицы 13

| № п/п | № блока | n | m | k | a | b | c | Подпрограммы | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **18** | 4 | 2 | 1 | 2 | 2 | - | - | 4 | 3 | **6** | 3 |
| **19** | 4 | 2 | 1 | 2 | 2 | - | - | 3 | 3 | 6 | 3 |
| **20** | 4 | 2 | 1 | 2 | 2 | - | - | 5 | 3 | 6 | 3 |
| **21** | 4 | 2 | 1 | 2 | 2 | **18** | - | P = 6 -> Fkt(m+2)=6 | | | |
| **22** | 5 | 2 | 1 | 2 | 2 | 18 | - | Fkt(k) -> f = k | | | |
| | | | | | | | | **№ блока** | **f** | **P** | **i** |
| **23** | 5 | 2 | 1 | 2 | 2 | 18 | - | 2 | **2** | **1** | - |
| **24** | 5 | 2 | 1 | 2 | 2 | 18 | - | ... | | | |
| **25** | 5 | 2 | 1 | 2 | 2 | 18 | - | 5 | 2 | **2** | **2** |
| **26** | 5 | 2 | 1 | 2 | 2 | 18 | - | P = 2 -> Fkt(k)= 2 | | | |
| **27** | 5 | 2 | 1 | 2 | 2 | 18 | - | Fkt(Fkt(k)+1) -> f:=3 | | | |
| | | | | | | | | **№ блока** | **f** | **P** | **i** |
| **28** | 5 | 2 | 1 | 2 | 2 | 18 | - | 2 | **3** | **1** | - |
| **29** | 5 | 2 | 1 | 2 | 2 | 18 | - | ... | | | |
| **30** | 5 | 2 | 1 | 2 | 2 | 18 | - | 5 | 3 | **6** | **3** |
| **31** | 5 | 2 | 1 | 2 | 2 | 18 | **6** | P = 6 -> Fkt(Fkt(k)+1)=6 | | | |
| **32** | 6 | 2 | 1 | 2 | 2 | 18 | 6 | | | | |
| **33** | 7 | 2 | 1 | 2 | 2 | 18 | 6 | Swap(a,b) -> s1 ≡ a  s2 ≡ b | | | |
| | | | | | | | | **№ блока** | **s1(a)** | **s2(b)** | **tmp** |
| **34** | 7 | 2 | 1 | 2 | 2 | 18 | 6 | 2 | 2 | 18 | **2** |
| **35** | 7 | 2 | 1 | 2 | **18** | 18 | 6 | 3 | **18** | 18 | 2 |
| **36** | 7 | 2 | 1 | 2 | 18 | **2** | 6 | 4 | 18 | **2** | 2 |
| **37** | 8 | 2 | 1 | 2 | 18 | 2 | 6 | | | | |
| **38** | 10 | 2 | 1 | 2 | 18 | 2 | 6 | | | | |
| **39** | 11 | 2 | 1 | 2 | 18 | 2 | 6 | Swap(b,c) -> s1 ≡ b  s2 ≡ c | | | |
| | | | | | | | | **№ блока** | **s1(b)** | **s2(c)** | **tmp** |
| **40** | 11 | 2 | 1 | 2 | 18 | 2 | 6 | 2 | 2 | 6 | 2 |
| **41** | 11 | 2 | 1 | 2 | 18 | 6 | 6 | 3 | 6 | 6 | 2 |
| **42** | 11 | 2 | 1 | 2 | 18 | 6 | 2 | 4 | 6 | 2 | 2 |
| **43** | 12 | 2 | 1 | 2 | 18 | 6 | 2 | | | | |
| **44** | 13 | 2 | 1 | 2 | 18 | 6 | 2 | | | | |

Реализация алгоритма в программу выглядит следующим образом:

```
program PrFun;
uses crt;
var n,m,k:integer;
 a,b,c:real;
 function Fkt (f:integer):integer;
 var i,P:integer;
 begin
  P:=1;
 for i:=2 to f do
   P:=P*i;
 Fkt:=P;
end;
 procedure Swap(var s1, s2 : real);
 var tmp:real;
 begin
   tmp:=s1;
   s1:=s2;
   s2:=tmp;
 end;
Begin
 writeln('vv  ddwsdvcsdf');
 readln(n,m,k);
 a:=Fkt(n);
 b:=Fkt(m+2)*3;
 c:=Fkt(Fkt(k)+1);
 if (b>c) and (b>a)  then swap(a,b);
 if (c>b) and (c>a)  then swap(a,c);
 if (c>b)  then swap(b,c);
 writeln('a=', a,'b=', b,'c=',c);
End.
```

## Итоги главы

Итак, мы рассмотрели синтаксические и семантические основы языка программирования Turbo Pascal, а также привели несколько типовых вычислительных алгоритмов. Следующий шаг на пути освоения программирования – применение полученных знаний для решения прикладных задач, разработки прикладных программ.

В следующей главе мы рассмотрим, как язык программирования и типовые алгоритмы применяются для реализации численных методов решения математических задач.

# Глава 2. Численные методы

**Вычислительная математика** — раздел математики, включающий круг вопросов, связанных с производством вычислений и использованием компьютеров. В более узком понимании вычислительная математика — **теория численных методов** решения типовых математических задач.

К задачам вычислительной математики относят:

- решение систем линейных уравнений;
- нахождение собственных значений и векторов матрицы;
- нахождение сингулярных значений и векторов матрицы;
- решение нелинейных алгебраических уравнений;
- решение систем нелинейных алгебраических уравнений;
- решение дифференциальных уравнений (как обыкновенных дифференциальных уравнений, так и уравнений с частными производными);
- решение систем дифференциальных уравнений;
- решение интегральных уравнений;
- задачи аппроксимации;
- задачи интерполяции;
- задачи экстраполяции.

Основное отличие вычислительной математики заключается в том, что при решении вычислительных задач человек оперирует машинными числами, которые являются дискретной проекцией вещественных чисел. Поэтому важную роль в вычислительной математике играют оценки точности получаемых результатов. Именно поэтому, например, для решения линейной системы алгебраических уравнений очень редко используется вычисление обратной матрицы, так как этот метод может привести к ошибочному решению в случае с сингулярной матрицей, а очень распространенный в линейной алгебре метод, основанный на вычислении определителя матрицы и ее дополнения, требует гораздо больше арифметических операций, чем любой устойчивый метод решения линейной системы уравнений.

## Решение нелинейных уравнений

Если алгебраическое или трансцендентное уравнение достаточно сложно, то его сравнительно редко удается решить аналитическими методами. Более того, в некоторых случаях коэффициенты уравнения известны лишь приближенно, и сама задача о точном определении корней теряет смысл. Поэтому важное значение приобретают способы приближенного нахождения корней уравнения и оценки их точности. Пусть дано уравнение:

$$f(x) = 0,$$

где функция *f(x)* определена и непрерывна в некотором конечном или бесконечном интервале $a \le x \le b$. Всякое значение ξ, обращающее функцию *f(x)* в нуль, т.е. такое, что *f(ξ) = 0*, называется корнем уравнения. Будем предполагать, что уравнение имеет лишь изолированные корни, т.е. для каждого из них существует окрестность Δ, не содержащая другие корни (рис. 26). Решить уравнение численными методами – это значит определить, имеет ли оно корни, сколько их и найти корни с заранее заданной точностью. Для решения уравнений вида разработано много различных итерационных методов. Сущность этих методов заключается в следующем.

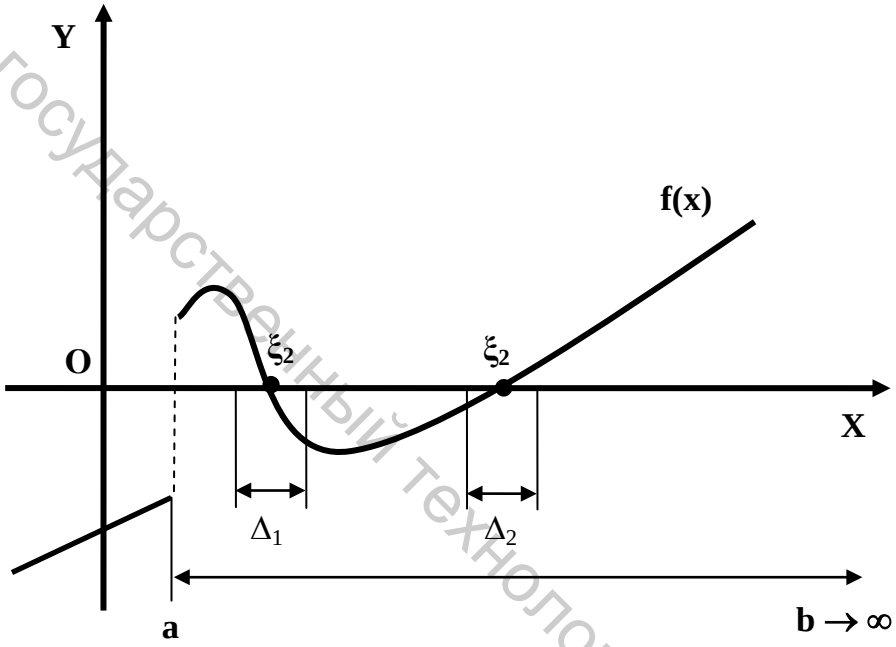



Рисунок 26 – Решение уравнения

Пусть известна достаточно малая область Δ, в которой содержится единственный корень ξ уравнения. В этой области выбирается точка x0 – начальное приближение – и строится последовательность точек x1,x2,…xn,…, сходящаяся к ξ, с помощью некоторого рекуррентного соотношения:

$$x_k = \varphi_k(x_0, x_1, \ldots, x_{k-1}).$$

Рекуррентное вычисление повторяется до тех пор, пока абсолютное значение разницы между двумя последовательными значениями x не станет меньше некоторого значения e, называемого точностью:

$$| x_i - x_{i-1} | < e.$$

Выбирая различными способами функции φk, которые зависят от функции f и номера k, можно получить различные методы.

### Численное решение нелинейных уравнений методом итерации

Заменим уравнение

$$f(x) = 0$$

равносильным ему уравнением

$$f_1(x) = x.$$

И исходное и полученное уравнения имеют одинаковый корень ξ (рис. 27) и называются эквивалентными.

Выберем любым способом $x_0$, которое затем подставим в левую часть уравнения:

$$f_1(x_0) = x_1.$$

Полученное значение $x_1$ снова подставим в левую часть и получим:

$$f_1(x_1) = x_2.$$

Продолжая этот процесс, получим последовательность чисел $x_1$ , $x_2$ , ..., $x_n$, которая может либо сходиться, т.е. иметь предел, либо расходиться, т.е. не иметь предела. Тогда в соответствии с полученным результатом в первом случае этот предел является корнем уравнения ($x \to \xi$), во втором случае сделаем вывод о невозможности получения решения данным способом.

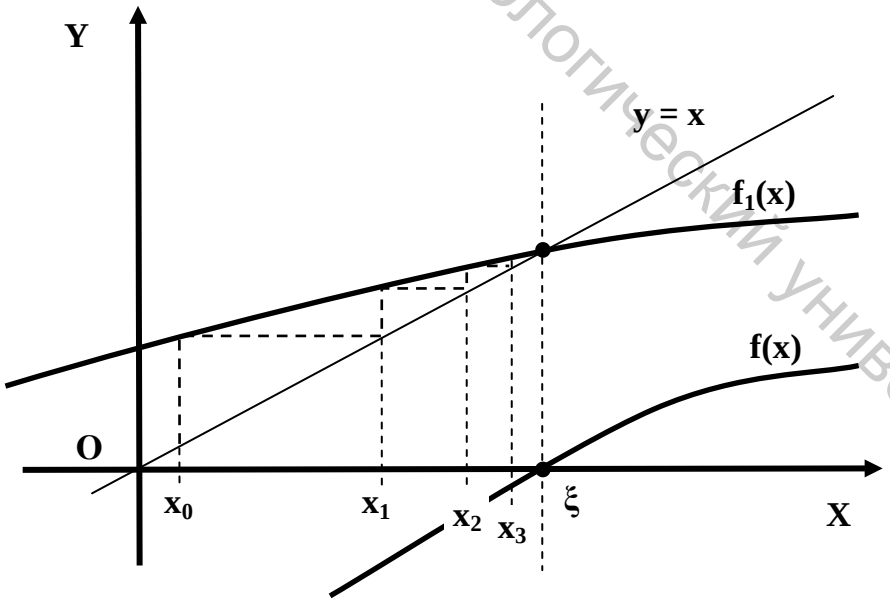



Рисунок 27 – Геометрическая интерпретация итерационного метода решения уравнения

Алгоритм реализации данного метода будет схож с алгоритмом рекуррентного вычисления бесконечной суммы.

Приведём программу, реализующую итерационный метод уточнения корня уравнения:

$$x^2 - x - 2 = 0.$$

```
program Iter;
VAR
xp, e, x0, x :REAL;
Function f(x:real): real;
begin
f:= x*x-2;
end;
begin
writeln('введите начальное приближение x и точность');
   read(x0,e);
  x:=x0;
  REPEAT
  xp:=x;
  x:=f(x);
  UNTIL ABS(x-xp)<e;
  writeln('корень уравнения =', x);
end.
```

В данной программе используется подпрограмма функция f, которая вычисляет левую часть преобразованного уравнения:

$$f_1(x) = x.$$

Переменная xp используется в программе для сохранения предыдущего значения x.

### Численное решение нелинейных уравнений методом бисекции

Метод состоит в построении последовательности вложенных отрезков, на концах которых F(x) имеет разные знаки. Каждый последующий отрезок получают делением пополам предыдущего. Этот процесс построения последовательности вложенных отрезков позволяет найти нуль функции (F(x) = 0) с любой заданной точностью.

Опишем подробно один шаг итерации (рис. 28). Пусть на k-м шаге найден отрезок $[a_k , b_k]$, на концах которого F(x) меняет знак. Заметим, что обязательно $[a_k, b_k] \in [a, b]$.

1. Разделим теперь отрезок $[a_k, b_k]$ пополам и выделим $F(c_k)$, где $c_k$ – середина $[a_k , b_k]$.
2. Здесь возможны два случая:
   - первый, когда $F(c_k) = 0$, тогда мы говорим, что корень найден;
   - второй, когда $F(c_k) \neq 0$, тогда сравниваем знак $F(c_k)$ с $F(a_k)$ и $F(b_k)$ и из двух половин $[ak, c_k]$ и $[c_k, bk]$ выбираем ту, на концах которой функция меняет свой знак. Таким образом, $a_{k+1} = a_k$ , $b_{k+1} = c_k$, если $F(c_k)F(a_k) < 0$ (рис. 27), и $a_{k+1} = c_k$ , $b_{k+1} = b_k$, если $F(c_k)F(b_k) < 0$.

При заданной точности e деление пополам продолжают до тех пор, пока длина отрезка не станет меньше 2e, тогда координата середины последнего найденного отрезка и есть значение корня требуемой точности.

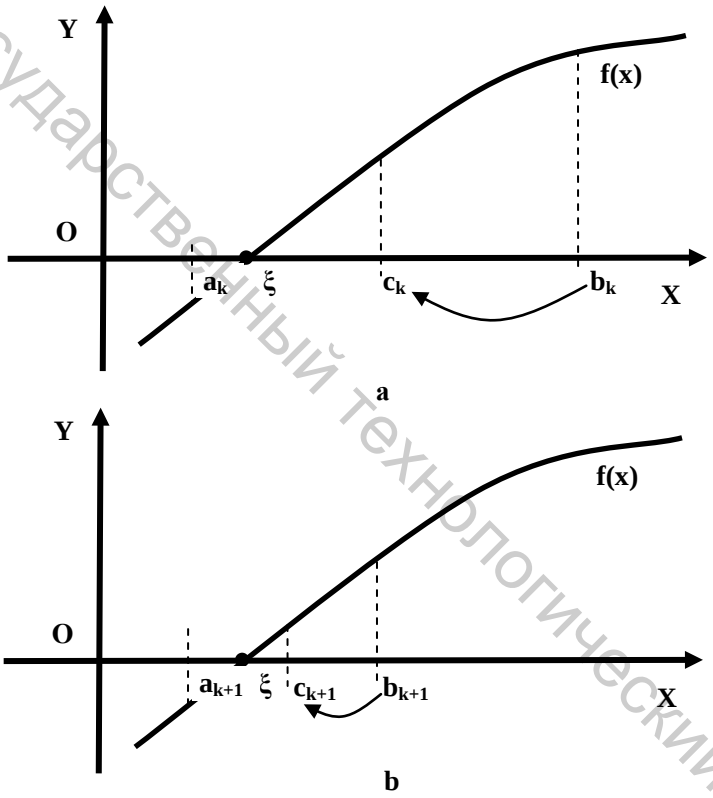



Рисунок 28 – Геометрическая интерпретация решения уравнения методом бисекции
a) k–й шаг; b) k+1-й шаг.

Приведём программу, реализующую итерационный метод уточнения корня уравнения:

$$x^2 - x - 2 = 0.$$

```
program Bisect;
VAR
A,B,E :REAL; X : REAL;
```

```
Function f(x:real): real;
begin
f:= x*x-x-2;
end;
begin
writeln('введите отрезок локализации корня [a,b] и точность');
   read(a,b,e);
  k:=0;
  REPEAT
   X := (A+B)*0.5;
  IF f(X)*f(B)<0 THEN
   a:=x
     else
   b:=x
  UNTIL ABS(B-A)<e;
  writeln('корень уравнения =', (A+B)*0.5);
END.
```

В данной программе используется подпрограмма функция f, которая вычисляет левую часть исходного уравнения:

$$f(x) = 0.$$

Переменная x используется в программе для сохранения значения середины отрезка [a, b].

### Численное решение нелинейных уравнений методом Ньютона

Положим:

$$\xi = x_n + h_n,$$

где ξ – корень уравнения f(x) = 0, $h_n$ считаем малой величиной. Отсюда, применяя формулу Тейлора, получим:

$$f(\xi) = f(x_n + h_n) \approx f(x_n) + h_n f'(x_n).$$

Следовательно, $h_n = -f(x_n) / f'(x_n)$. Подставив это выражение, получим:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \qquad n = 0, 1, 2, \ldots$$

Геометрическая интерпретация метода представлена на рисунке 29.

Через точку $A_0(x,y)$ с координатами $x = x_0$, $y = f(x_0)$ проводим касательную. По формуле

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

находим точку пересечения касательной с осью OX (точка $x_1$).

Находим точку $A_1(x_1,y)$, проводим касательную через точку $A_1$ для нахождения $x_2$.

Этот процесс продолжаем до тех пор, пока разница $|x_{i+1}-x_i|$ не станет меньше, чем значение ε, т.е. пока не будет найдено значение корня с заданной точностью ε.

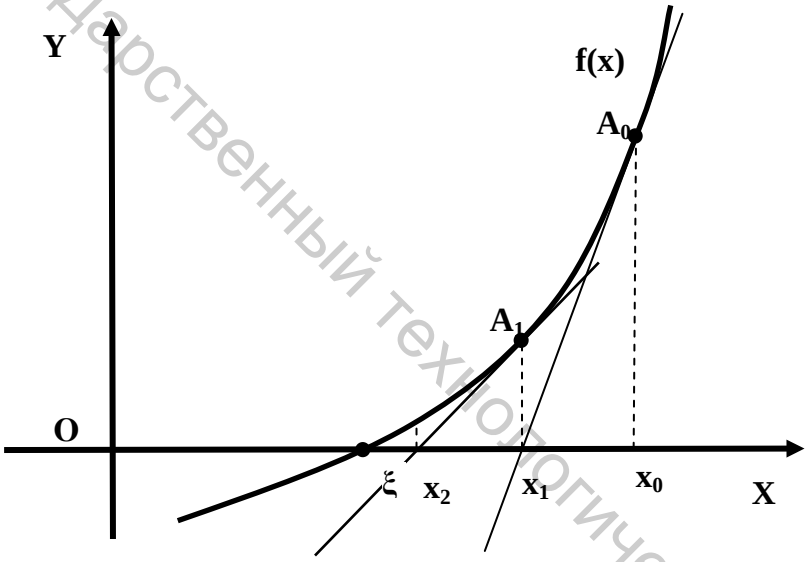



Рисунок 29 – Геометрическая интерпретация решения уравнения методом Ньютона

Приведём программу, реализующую итерационный метод уточнения корня уравнения:

$$x^2 - x - 2 = 0.$$

```
program Newton;
VAR  xp,E,x0 :REAL; X: REAL;
Function f(x:real): real;
begin
f:= x*x-x-2;
end;
Function df(x:real): real;
```

```
begin
df:= 2*x-1;
end;
begin
writeln('введите начальное приближение x и точность');
   read(x0,e);
  x:=x0;
  REPEAT
  xp:=x;
  x:=x-f(x)/df(x);
  UNTIL ABS(x-xp)<E;
  writeln('корень уравнения =', x);
END.
```

В данной программе используется подпрограмма функция f, которая вычисляет левую часть исходного уравнения:

$$f(x) = 0$$

а также подпрограмма функция df, которая вычисляет значение дифференциала:

$$f'(x).$$

Переменная xp используется в программе для сохранения предыдущего значения x.

## Решение систем линейных уравнений

К решению систем линейных алгебраических уравнений с большим числом неизвестных приводят многие практически важные задачи. Разработано множество методов, предназначенных для решения систем с матрицей того или иного вида. Все эти методы можно разделить на два больших класса: прямые (или точные) и итерационные.

**Прямые методы** дают решение системы за конечное число арифметических операций. Если коэффициенты системы не содержат погрешностей, а арифметические операции выполняются точно (без округления), то решение получается точным. Недостатки прямых методов: иногда требуется выполнение неприемлемо большого числа арифметических и логических операций; погрешности округления могут очень сильно исказить результат.

**Итерационные методы** позволяют получить приближенное решение с любой заданной точностью ε. Они дают искомое решение как предел последовательности приближений и различаются способами построения таких последовательностей. Недостатком этих методов является то, что построенные последовательности нередко оказываются расходящимися, т.е. конечного

предела таких последовательностей может не существовать. Поэтому оказывается необходимым исследование сходимости.

### Численное решение систем линейных уравнений методом Гаусса

Система линейных алгебраических уравнений выглядит следующим образом:

$$\begin{cases} a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,n}x_n = b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \ldots + a_{2,n}x_n = b_2 \\ \quad \ldots \qquad \ldots \qquad \ldots \qquad \ldots \\ a_{n,1}x_1 + a_{n,2}x_2 + \ldots + a_{1,n}x_n = b_n \end{cases}$$

в матричной форме такая система выглядит так:

$$A \cdot \bar{x} = \bar{b},$$

где $A$ – матица коэффициентов системы уравнений:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \ldots & a_{1,n} \\ a_{2,1} & a_{2,2} & \ldots & a_{2,n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{n,1} & a_{n,2} & \ldots & a_{1,n} \end{bmatrix},$$

$\bar{b}$ – вектор свободных членов (правых частей) уравнений, $\bar{x}$ – вектор неизвестных:

$$\bar{b} = \begin{bmatrix} b_1 \\ b_2 \\ \ldots \\ b_n \end{bmatrix}, \qquad \bar{x} = \begin{bmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{bmatrix}$$

Метод Гаусса (метод последовательного исключения неизвестных) состоит из прямого и обратного хода.

В результате прямого хода матрицы системы за $m$ преобразований приводится к треугольному виду:

$$A^m = \begin{bmatrix} 1 & a^m_{1,2} & \dots & a^m_{1,n} \\ 0 & 1 & \dots & a^m_{2,n} \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 \end{bmatrix}.$$

Количество преобразований исходной матрицы ($m$) в общем случае равно количеству уравнений в системе ($n$).

На каждом $k$-ом шаге преобразования выбирается ведущая строка ($k$) и ведущий элемент $a_{kk}$.

Формулы прямого хода метода Гаусса на $k$-ом шаге преобразования имеют следующий вид: для ведущей строки:

$$a^{(k)}_{kj} = \frac{a^{(k-1)}_{kj}}{a^{(k-1)}_{kk}}, \quad k = 1,2,\dots,n; \quad j = k, k+1,\dots,n+1;$$

для строк, лежащих под ведущей строкой:

$$a^{(k)}_{ij} = a^{(k-1)}_{ij} - a^{(k-1)}_{ik} a^{(k-1)}_{kj}, \qquad i = k+1,\dots,n.$$

Элемент $a^{(k-1)}_{kk}$ называется ведущим.

Обратный ход используется для вычисления значений неизвестных.

Система уравнений с треугольной матрицей А имеет следующий вид:

$$\begin{cases} x_1 + a^m_{1,2} x_2 + \dots + a^m_{1,n} x_n = b^m_1 \\ 0x_1 + 1x_2 + \dots + a^m_{2,n} x_n = b^m_2 \\ \quad \dots \qquad \dots \qquad \dots \qquad \dots \\ 0x_1 + 0x_2 + \dots + x_n = b^m_n \end{cases}$$

Очевидно, что

$$x_n = b^m_n.$$

Формула обратного хода метода Гаусса (нахождения остальных неизвестных в обратном порядке) имеет вид:

$$x_i = b^m_i - \sum_{j=i+1}^{n} a^m_{ij} x_j, \quad i = n-1, n-2, \dots, 1.$$

При разработке программы необходимо учитывать то, что принципы индексации, принятые в математике и используемые при разработке

программы, могут различаться: коэффициенту уравнения $a_{ij}$ соответствует элемент массива A[j,i] (см. пункт«Описания массивов» стр. 38).

Приведём программу, использующую метод Гаусса для решения системы линейных уравнений (обратите внимание на комментарии, поясняющие назначение ключевых фрагментов программы):

```
program gauss;
const n = 3;
var A:array[1..n,1..n]of real;
 x,b:array[1..n] of real;
 i,j,n,p,ii,jj:integer;  aii,akk:real;
begin
{ввод исходных данных}
writeln('введите матрицу коэффициентов');
for j:=1 to n do
begin
writeln('уравнение №', j)
for i:=1 to n do
begin
writeln(' a', i, ', ', j, ' =') ;
read(a[i,j]);
end;
writeln('свободный коэффициент');
read(b[j]);
end;
{прямой ход метода}
for i:=1 to n do  {цикл отсчитывает шаги прямого хода метода}
begin
aii:=a[i,i]; {сохранения значения ведущего элемента}
{преобразование ведущей строки}
for j:=i to n do
a[j,i]:=a[j,i]/aii;
b[i]:=b[i]/aii;
{преобразование строк, под ведущей}
for p:=i+1 to n do
 begin
akk:=a[i,p];
 for j:=1 to n do
 a[j,p]:= a[j,p]-a[j,i]*akk;
 b[p]:=b[p]-b[i]*akk;
 end;
end;
```

Следует обратить внимание на то, что данная программа, как в прочем и любые другие программы, состоит из отдельных частей. Каждая из этих частей выполняет определённую задачу, поэтому каждая из них может рассматриваться как отдельная программа. Такой подход упрощает разработку и отладку программы.



## Численное интегрирование

Задача численного интегрирования функции заключается в вычислении значений определенного интеграла, когда известен ряд значений подынтегральной функции. Численное вычисление однократного интеграла называется механической квадратурой, двойного – механической кубатурой. Соответствующие формулы называются квадратурными и кубатурными.

Обычный прием механической квадратуры заключается в том, что данную функцию f(x) на рассматриваемом отрезке [a, b] заменяют интерполирующей или другой аппроксимирующей функцией $\varphi(x)$ простого вида (например, полиномом), а затем приближенно полагают

$$\int_a^b f(x)dx \approx \int_a^b \varphi(x)dx .$$

Функция $\varphi(x)$ такова, что интеграл от неё вычисляется напрямую с помощью формулы.

## Численное интегрирование методом прямоугольников

Простейшим методом численного интегрирования является метод прямоугольников, который в ряде случаев оказывается наиболее эффективным.

Известны три разновидности метода прямоугольников: это методы левых, правых и средних прямоугольников. Все они основаны на аппроксимации подынтегральной функции f(x) прямой, проходящей через точку $f(x_i)$, $f(x_{i+1})$ или $f(x_i+\Delta/2)$ соответственно.

Таким образом, площадь подынтегральной кривой заменяется площадью прямоугольника:

левого прямоугольника:

$$S = \Delta \cdot f(x_i);$$

правого прямоугольника:

$$S = \Delta \cdot f(x_{i+1});$$

среднего прямоугольника:

$$S = \Delta \cdot f(x_i + \Delta/2).$$

С учетом представления на элементарном отрезке составные формулы вычисления интегралов могут быть записаны так:

левых прямоугольников:

$$I = \sum_{i=0}^{n-1} \Delta \cdot f(x_i);$$

правых прямоугольников:

$$I = \sum_{i=1}^{n} \Delta \cdot f(x_{i+1});$$

средних прямоугольников:

$$I = \sum_{i=0}^{n-1} \Delta \cdot f(x_i + \Delta/2).$$

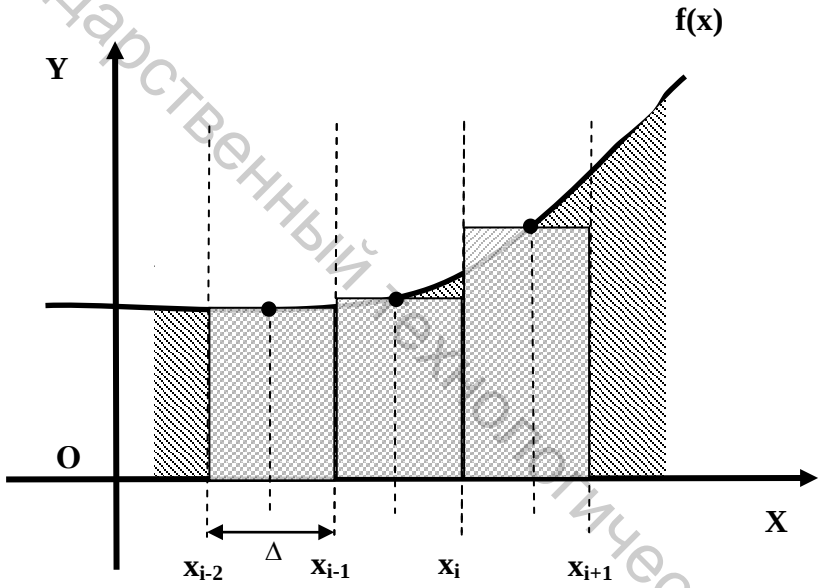



Рисунок 30 – Геометрическая интерпретация численного интегрирования методом центральных прямоугольников

Приведём программу, реализующую вычисление определённого интеграла методом центральных прямоугольников с заданным количеством разбиений. В качестве подынтегральной будем использовать функцию:

$$f(x) = \frac{1}{x}.$$

```
program rect;
function f(x: real): real;
begin
  f:=1/x
end;
```

```
var
a,b,e: real;
i: integer;
x,s,dx: real;
n: integer;
begin
  writeln('[a,b],n');
  readln(a,b,n);
 {вычисление длины отрезка – основания прямоугольника (дельта)}
  dx:=(b-a)/n;
   for i:=0 to n-1 do
     begin
{вычисление абсциссы середины очередного отрезка разбиения}
       x:=a+dx*i+dx/2;
       s:=s+dx*f(x);
     end;
 writeln('int=',s);
end.
```

В данной программе используется подпрограмма функция f, которая вычисляет подынтегральную функцию $f(x)$.

### Численное интегрирование методом Симпсона с заданной точностью

Принцип метода Симпсона состоит в замене подынтегральной функции f(x) интерполяционным многочленом Ньютона второй степени. Тогда для каждого элементарного отрезка [xi,xi+1] имеем следующее значение площади подынтегральной кривой:

$$S = h \cdot \left( f(x_i) + 4 f(x_i + \frac{\Delta}{2}) + f(x_{i+1}) \right) / 6.$$

Для всего отрезка интегрирования [a,b] формулой Симпсона:

$$I = \sum_{i=0}^{N} h \cdot \left( f(x_i) + 4 f(x_i + \frac{\Delta}{2}) + f(x_{i+1}) \right) / 6 =$$

$$= \frac{b-a}{6N} \cdot \left( \left( f(x_0) + f(x_n) \right) + 4 \sum_{i=0}^{N} f(x_i + \frac{\Delta}{2}) + 2 \sum_{i=1}^{N-1} f(x_{i+1}) \right).$$

Данное выражение называется формулой Симсона. Оно относится к формулам повышенной точности и является точной для многочленов второй и третьей степени.

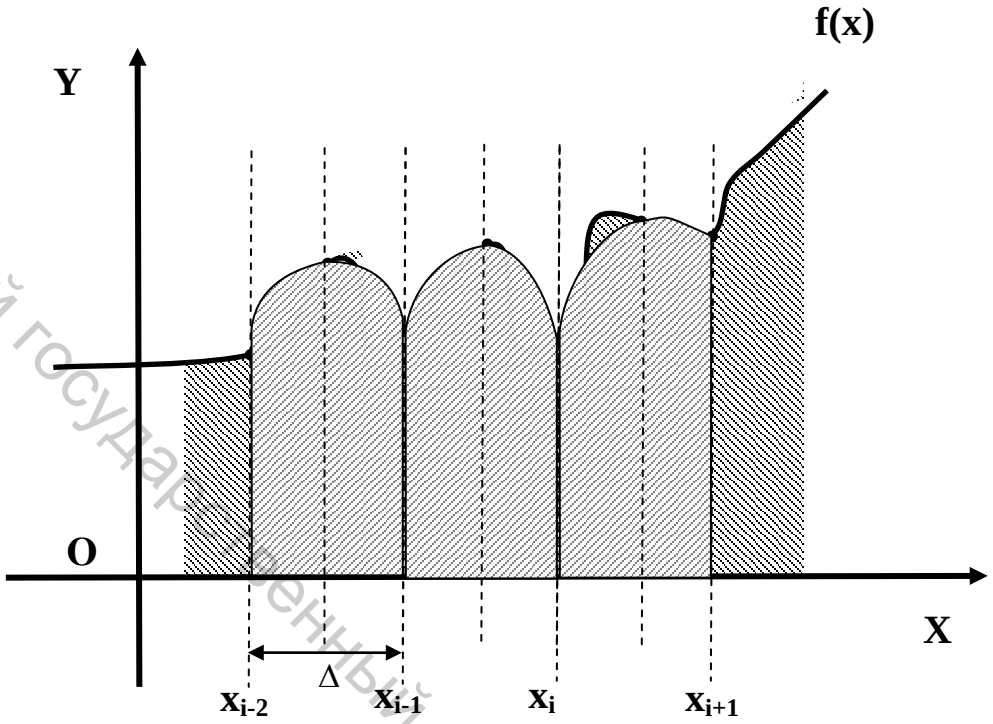



Рисунок 31 – Геометрическая интерпретация численного интегрирования методом Симпсона

Приведём программу, реализующую вычисление определённого интеграла методом Симпсона с заданной точностью**.** В качестве подынтегральной будем использовать функцию:

$$f(x) = \frac{1}{x}.$$

Рассмотренные формулы численного интегрирования требуют чёткого указания количества разбиений отрезка интегрирования. Однако классическое использование численного метода предполагает вычисление значения (корня, интеграла и т.д.) с заданной точностью.

Точность любой формулы численного интегрирования зависит от величины отрезка разбиения Δ.

Будем вычислять значение интеграла при разных значениях Δ ($\Delta_1$, $\Delta_2$, $\Delta_3$,...), где $\Delta_{i+1} = 2\Delta_i$. Как только разница между значением интеграла, вычисленного при $\Delta_i$ и интеграла, вычисленного при $\Delta_{i+1}$, станет меньше, чем значение ε, будем считать, что интеграл вычислен с заданной точностью ε.

Данный метод интегрирования с заданной точностью прост в реализации, однако он требует значительных избыточных вычислений, что приводит к повышению затрат времени на вычисление.

```
program simp;
function f(x: real): real;
begin
  f:=1/x
end;
 var
a,b,e: real;
i: integer;
xa,xab,xb,dx,s1,s: real;
n: integer;
begin
  writeln('[a,b],e');
  readln(a,b,e);
{вычисление интеграла с количеством разбиений равным 1, т. е. одной
фигурой с основанием равным [a,b]}
  n:=1;
  dx:=(b-a)/n;
  s:=dx*(f(a)+4*f(a+dx/2)+f(b))/6;
  repeat
    n:=n*2; {удвоение количества отрезков разбиения}
    s1:=s;
    s:=0;
{вычисление длины отрезка – основания прямоугольника (дельта)
при новом количестве разбиений}
    dx:=(b-a)/n;
{суммирование площадей  - нахождение интеграла при заданном
количестве разбиений}
    for i:=0 to n-1 do
      begin
  xa:=a+dx*(i);
        xb:=xa+dx;
        xab:=xa+dx/2;
        s:=s+dx*(f(xa)+4*f(xab)+f(xb))/6;
      end;
 until abs(s-s1)<=abs(e);
 writeln('int=',s);
end.
```

В данной программе используется подпрограмма функция f, которая вычисляет подынтегральную функцию $f(x)$.

Переменная s1 используется для сохранения значения интеграла, вычисленного при вдвое меньшем количестве разбиений.

## Итоги главы

Алгоритмы реализации численных методов являются очень важной составной частью широкого круга программных продуктов, например: систем автоматизированного проектирования, научного проргаммного обеспечения, графических редакторов, современных компьютерных игр и т. п.

В данной главе мы привели базовые алгориты, реализующие основные схемы численных методов. Алгоритмы, применяемые в современном программном обеспечении, являются более сложными, поскольку предусматривают дополнительные возможности, такие как определение сходимости, автоматический выбор начального приближения, повышение скорости расчётов и т. п.

## Рекомендуемая литература


1. Математика и информатика. [Электронный ресурс] – 2007. – Режим доступа: http://do.rksi.ru/library/courses/mathinf/l
2. Borland Pascal. Руководство пользователя [Электронный ресурс] – 2000. – Режим доступа: http://citforum.ru/programming/bp70_ug/index.shtml
3. Вальвачев, А. Н. Программирование на языке Паскаль для персональных ЭВМ ЭС / А. Н. Вальвачев, В. С. Крисевич. – Минск : ВШ, 1991. – 224 с.
4. Основы алгоритмизации и программирования [Электронный ресурс] – 2007. – Режим доступа: http://256bit.ru/informat/index11.htm
5. Терентьев, В. П. TURBO PASCAL и объектно-ориентированное программирование. Часть 1 / В. П. Терентьев, Е. Ю. Вардомацкая, Д. В. Черненко. – Витебск : ВГТУ, 1999.
6. Терентьев, В. П. TURBO PASCAL и объектно-ориентированное программирование. Часть 2 / В. П. Терентьев, Е. Ю. Вардомацкая, Е. А. Калиновская, Т. Н. Окишева, Т. П. Стасеня. – Витебск : ВГТУ, 2002.
7. Офицеров, Д. А. Программирование на персональных ЭВМ / Д. А. Офицеров [ и др.]. – Минск : ВШ, 1993.
8. Епанешников, А. М. Программирование в среде TURBO PASCAL 7.0. / А. М. Епанешников, В. Н. Епанешников. – Москва : "Диалог" МИФИ, 2000.
9. Шарстнев, В. Л. Методические указания и задания к типовым расчетам по предметам цикла «Информатика» / В. Л. Шарстнев [ и др.]. – Витебск : УО «ВГТУ», 2002.
10. Демидович, Б. П. Основы вычислительной математики / Б. П. Демидович, И. А. Марон. – Минск : Наука, 1989.
11. Вычислительная техника и программирование / под ред. А. В. Петрова. – Минск : "ВШ", 1990. – 479 с.
12. Комягин, В. Н. Программирование в Excel 5 и Excel 7 на языке Visual Basic / В. Н. Комягин. – Москва : Радио и связь, 1996. – 320 с.
13. Турчак, Л. И. Основы численных методов / Л. И. Турчак. – Минск : Наука, 1987.
14. Мудров, А. Е.Численные методы решения для ПЭВМ на языках Бейсик, Фортран, Паскаль / А. Е. Мудров. – Томск : МП «Раско», 1991. – 272 с.
15. Информационный сервер для программистов. Исходники со всего света [Электронный ресурс] – 2007. – Режим доступа: http://pascal.sources

Учебное издание

**Казаков Вадим Евгеньевич**



**ИНФОРМАТИКА.**
**Программирование на алгоритмическом языке**

Конспект лекций

Редактор В. П. Терентьев
Технический редактор И. В. Соколов
Корректор Е. М. Богачева
Компьютерная верстка В. Е. Казаков

---

Подписано к печати _____. Формат 60x90 1/16 Бумага офсетная № 1. Гарнитура «Таймс». Уч.-изд. листов _____. Усл. печ. листов _____. Тираж _____ экз. Зак. № _____.

Учреждение образования «Витебский государственный технологический университет» 210035, г. Витебск, Московский пр., 72.

Отпечатано на ризографе учреждения образования «Витебский государственный технологический университет».
Лицензия № 02330/0494384 от 16 марта 2009 г.