Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра программного обеспечения информационных технологий

Л. А. Глухова, Е. П. Фадеева, Е. Е. Фадеева

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

ЛАБОРАТОРНЫЙ ПРАКТИКУМ
для студентов специальности I-40 01 01
«Программное обеспечение информационных технологий»
дневной формы обучения

В 4-х частях

Часть 3

Минск 2007

УДК 681.3 (075.8)
ББК 32.973-018.2 я 73
Г 55

Р е ц е н з е н т
зав. кафедрой информатики
Минского государственного высшего радиотехнического колледжа,
канд. техн. наук, доц. Ю. А. Скудняков

Глухова, Л. А.

Г 55 Основы алгоритмизации и программирования : лаб. практикум для студ. спец. I-40 01 01 «Программное обеспечение информационных технологий» дневн. формы обуч. В 4 ч. Ч. 3 / Л. А. Глухова, Е. П. Фадеева,
Е. Е. Фадеева. – Минск : БГУИР, 2007. – 52 с. : ил.
ISBN 978-985-488-184-3 (ч. 3)

Третья часть лабораторного практикума посвящена вопросам модульного построения программ с использованием процедур и функций. Рассмотрены различные механизмы передачи параметров в вызываемые подпрограммы. Даны варианты индивидуальных заданий для выполнения лабораторных работ.

УДК 681.3 (075.8)
ББК 32.973-018.2 я 73

Первая и вторая части изданы в БГУИР соответственно в 2004 и 2005 гг.

ISBN 978-985-488-184-3 (ч. 3)
ISBN 978-985-488-183-6
ISBN 985-444-616-6

университет информатики
и радиоэлектроники», 2007

СОДЕРЖАНИЕ

1. Общие сведения

При разработке программы имеют место случаи, когда один и тот же набор операторов, реализующих определенную цель, необходимо повторить многократно в нескольких местах программы. Чтобы избавиться от дублирования и тем самым упростить процесс написания и отладки программы, повторяющиеся фрагменты выделяют в отдельные модули, называемые подпрограммами. ***Подпрограмма*** – поименованная логически законченная группа операторов языка. Упоминание имени подпрограммы в тексте программы приводит к ее активизации и называется вызовом подпрограммы. В языке Pascal существуют *два вида подпрограмм* – процедуры и функции.

Процедура – это независимая поименованная часть программы, вызываемая по имени, для выполнения определенных действий. Структура процедуры аналогична структуре программы. Процедура не может выступать как операнд в выражении.

Функция – это независимая поименованная часть программы, вызываемая по имени, для выполнения определенных действий. Однако в отличие от процедуры, функция передает в точку вызова скалярное значение, а имя функции должно входить в выражение как операнд.

Все процедуры и функции языка Pascal подразделяются на *две группы*: встроенные и определенные пользователем. ***Встроенные*** (стандартные) процедуры и функции являются частью языка и могут быть вызваны по строго фиксированному имени без предварительного определения в разделе описания. Большой набор стандартных подпрограмм языка существенно облегчает работу программиста, однако иногда необходимы специфические действия, не предусмотренные стандартными процедурами и функциями. В этом случае программист может разработать свою подпрограмму. Ее называют ***пользовательской*** (определенной пользователем). Вызову пользовательской подпрограммы должно предшествовать ее описание.

2. Процедуры, определенные пользователем

Процедура пользователя представляет собой поименованную группу операторов, которые выполняются при каждом вызове процедуры по имени из любого места раздела операторов. *Описание процедуры* производится в разделе объявления подпрограмм в программе и включает описания заголовка и тела процедуры:

<Объявление_процедуры> **::=**
<Заголовок_процедуры> ; <Тело_процедуры> ; **.**

Здесь и далее по тексту для описания синтаксиса конструкций языка Pascal используется метаязык, представляющий собой один из упрощенных

вариантов формы Бэкуса–Наура. Описание используемых в работе символов метаязыка дано на стр. 38.

Тело процедуры – это локальный блок, по структуре аналогичный структуре программы и состоящий в общем случае из директив компилятора, раздела локальных описаний и раздела операторов. *Заголовок* определяет имя процедуры и необязательный список формальных параметров:

<Заголовок_процедуры> **::=** Procedure <Имя процедуры>
[(<Список_формальных_параметров>**)] .** (1)

Имя процедуры – это идентификатор, уникальный в пределах программы. Список формальных параметров определяет имя и тип каждого параметра.

Для активизации действий, предусмотренных процедурой, используется *оператор вызова процедуры*. Он состоит из имени процедуры и заключенного в круглые скобки списка фактических параметров, отделенных друг от друга запятыми. Если процедуре не передаются никакие значения, то список параметров отсутствует. Между фактическими и формальными параметрами устанавливается соответствие слева направо. При этом количество формальных и фактических параметров должно быть равно. Соответствие по типам между параметрами рассматривается в разд. 5, 6.

ПРИМЕР 1. Программа сортировки обменами («пузырьком») массива из 20 целочисленных элементов. Алгоритм сортировки описан в [2, 3, 5, 7]. Используются процедуры без параметров.

```
Program Primer1;
    Uses Crt;
    Const   n = 20;
    Var    A : array [1 .. n] of  Integer;
           i : Integer;
    Procedure Bubble;                  {Описание процедуры сортировки}
       Var     i, j, temp : Integer;                  {элементов массива}
       Begin
          For i := 2 to n do
           For j := n downto i do
            If A[j-1] > A[j] then
            Begin                      {Обмен местами двух  элементов}
                temp := A[j-1];  A[j-1] := A[j];  A[j] := temp;
            End
       End;
    Procedure PrintMas;                {Описание процедуры вывода}
       Var    i : Integer;             {элементов массива на экран}
```

```
    Begin
       For i := 1 to n do
       Write (A[i] : 4);    Writeln
    End;
Begin                                          {Вызывающая программа}
   ClrScr;                          {Стандартная процедура очистки экрана}
   For i := 1 to n do               {Заполнение массива случайными числами}
     A[i] := Random(100);
  GotoXY (20, 7);                {Стандартная процедура смещения курсора}
  Writeln (' Исходный массив ');
  PrintMas;                                    {Вызов процедуры вывода}
  Bubble;                                  {Вызов процедуры сортировки}
  GotoXY (20, 17);
  Writeln(' Отсортированный массив ');
  PrintMas;                                    {Вызов процедуры вывода}
  ReadKey
End.
```

В данной программе использованы две процедуры: процедура ***Bubble*** реализует алгоритм сортировки «пузырьком», процедура ***PrintMas*** предназначена для вывода элементов массива на экран. Обе являются процедурами без параметров. Тело каждой из них содержит раздел описания переменных (***Var***) и раздел операторов, предназначенный для выполнения сортировки элементов в случае выполнения процедуры ***Bubble*** и вывода элементов массива на экран монитора в случае выполнения процедуры ***PrintMas***.

3. Функции, определенные пользователем

Функция пользователя возвращает в точку вызова функции искомое значение и представляет собой поименованную группу операторов, которые реализуют определенные действия, выполняемые при каждом вызове функции.

Описание функции выполняется в разделе объявления подпрограмм программы и имеет вид

<Объявление_функции> **::=** <Заголовок_функции> ; <Тело_функции> ; **.**

Заголовок функции определяет ее имя, список формальных параметров и тип возвращаемого результата:

<Заголовок_функции> **::=** Function <Имя функции> **[(**<Список_ формальных_параметров>**)]** : <Идентификатор_типа_результата> **.** (2)

Возвращаемый результат может иметь любой из скалярных типов, тип *String* или тип «указатель». В качестве типа возвращаемого результата можно использовать только имя типа, но не его задание. Поэтому, если тип является нестандартным, его необходимо предварительно объявить в разделе описания типов головной программы. Например

```
Type  Tst = String [60];
Function F1: Tst;
```

Тело функции – это локальный блок, структура которого аналогична структуре программы. Отличием является то, что в разделе операторов подпрограммы-функции ее имени должно быть присвоено некоторое значение, которое воспринимается как возвращаемый функцией результат. Если в теле функции идентификатору присваивались различные значения многократно, то результатом выполнения функции будет значение последнего оператора присваивания.

ПРИМЕР 2. Программа определения суммы элементов одномерного массива. Используются процедура и функция без параметров.

```
Program Primer2;
  Uses Crt;
  Const n = 20;
  Var A: array [1 .. n] of  Integer;
      i : Integer;
  Function SumMas : Longint;          {Описание функции для подсчета}
    Var i : Integer;                          {суммы элементов массива}
        temp : Longint;
    Begin
      temp := 0;
      For i := 1 to n do
          temp := temp + A[i];
      SumMas := temp;
    End;
  Procedure PrintMas;                        {Описание процедуры вывода}
    Var  i : Integer;                        {элементов массива на экран }
    Begin
      For i := 1 to n do
        Write (A[i] : 3);
      Writeln;
    End;
  Begin                                                {Вызывающая программа}
    ClrScr;
    For i := 1 to n do
      A[i] := Random(100);     {Заполнение массива случайными числами}
```

```
    GotoXY (20, 5);
    Writeln (' Элементы массива ');
    PrintMas;
    GotoXY (13, 10);
    Writeln (' Сумма элементов массива =  ', SumMas);      {Вызов функции}
End.
```

В данной программе помимо процедуры ***PrintMas*** реализована функция ***SumMas***. В заголовке функции ***SumMas*** после имени через двоеточие записан тип возвращаемого результата ***Longint***. При вызове функции из тела головной программы в точку вызова возвращается и выводится на экран монитора число, равное сумме элементов массива. Результат выполнения программы представлен на рис. 1.

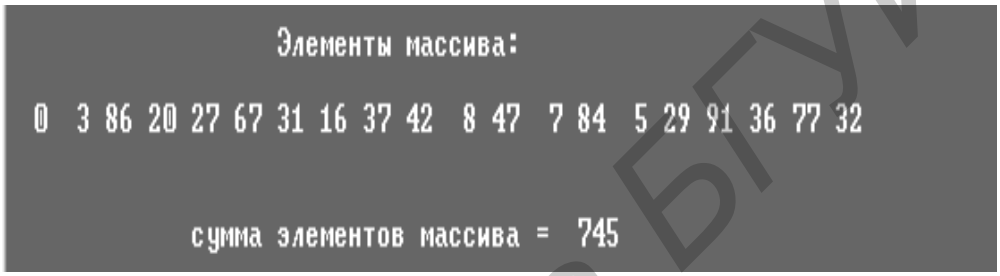

Рис. 1. Результат выполнения программы подсчета суммы элементов

4. Глобальные и локальные переменные

Как уже было отмечено, тело подпрограммы состоит из директив, раздела описаний и раздела операторов. Раздел описаний подпрограммы может включать в себя описание констант, типов, переменных, подпрограмм низшего уровня. Все идентификаторы, которые были описаны в подпрограмме, называются ***локальными***. Они недоступны в вызывающей их программе. Идентификаторы, описание которых выполнено в головной программе, называются ***глобальными***. Такие глобальные данные доступны как в головной программе, так и в любом месте вызываемой подпрограммы. Если локальные идентификаторы совпадают с глобальными, то в подпрограмме глобальные данные недоступны.

Так, в примере 2 глобальной переменной является массив *A*. Локальными для функции ***SumMas*** являются переменные ***i*** и ***temp***. Важно знать, что место под локальные данные выделяется только на время работы подпрограммы. После того как подпрограмма окончила свою работу, место, занимаемое локальными данными, освобождается, но информация из оперативной памяти не удаляется. При вызове следующей подпрограммы эта «испачканная» память может быть выделена под очередные локальные данные. Следовательно, надо тщательно следить за тем, чтобы локальным переменным в подпрограмме были присвоены начальные значения. Например,

если в примере 2 в теле функции *SumMas* опустить оператор *temp:=0*, то результат вычисления будет неверным (рис. 2).

Для того чтобы понять, почему сумма элементов массива так сильно увеличилась, следует узнать, чему было равно значение переменной *temp* до входа в цикл *For*. Необходимо внести в окно отладки *Watches* идентификатор *temp* (горячая клавиша *Ctrl-F7*) и установить контрольную точку в строке *For i := 1 to n do* (горячая клавиша *Ctrl-F8*). При запуске программы на выполнение (горячая клавиша *Ctrl-F9*) произойдет остановка в контрольной точке, т.е. до входа в тело цикла *For*. Как видно из рис. 3, в этот момент значение переменной *temp* равно 1 073 481 368. Откуда взялось это число? Переменная *temp* локальная, а значит, место для нее выделилось лишь в момент вызова функции *SumMas*. Очевидно ранее в тех же четырех байтах, которые отведены теперь под *temp*, хранилась другая информация.

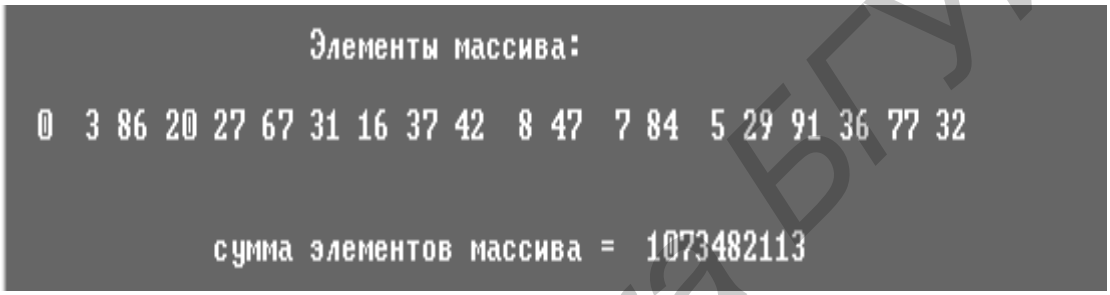

Рис. 2. Неверный результат подсчета суммы элементов массива

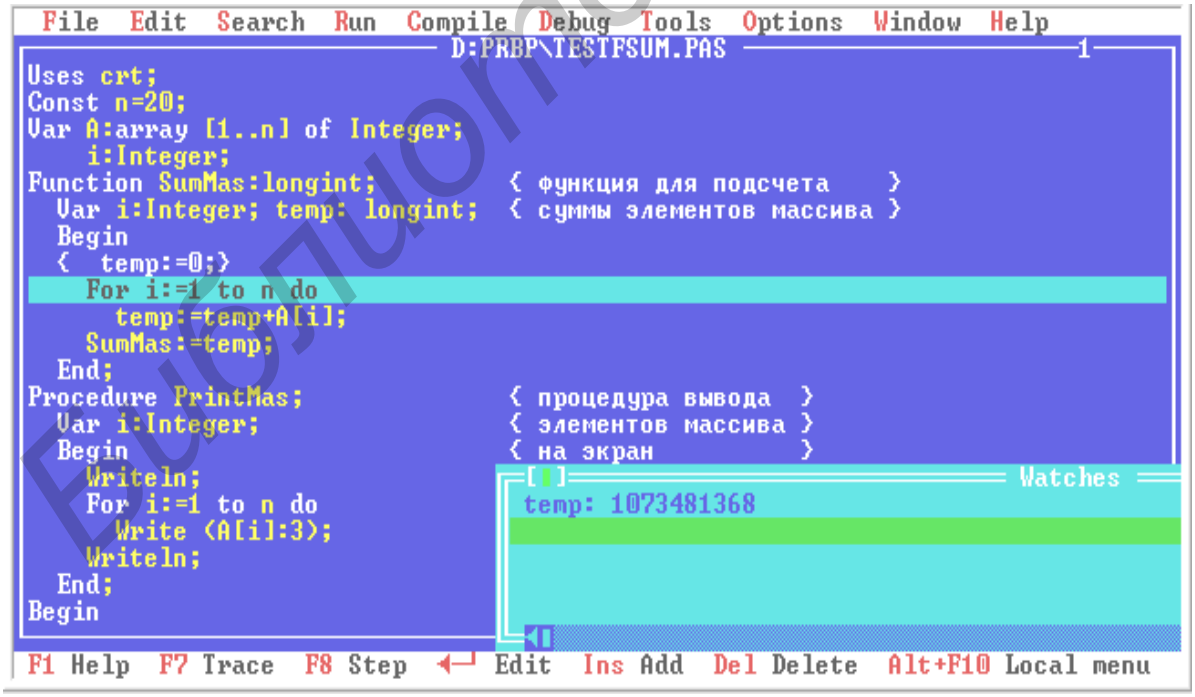

Рис. 3. Режим отладки программы

5. Параметры подпрограмм

При описании подпрограммы (процедуры или функции) за указанием ее имени может следовать необязательный список формальных параметров, заключенный в круглые скобки (конструкции (1) и (2) в разд. 2, 3). *Список формальных параметров* имеет следующий формат:

<Список_формальных_параметров> **::=**
[(<Формальный_параметр> {; <Формальный_параметр> } **)] .**

Формальный параметр имеет следующую структуру:

<Формальный_параметр> **::= [** Var **|** Const **] (** <Идентификатор>
{, <Идентификатор>} **)** [: <Тип_параметра>**] .** (3)

Формальный параметр, которому предшествует зарезервированное слово *Var* и за которым следует тип, называется *параметром-переменной*. Формальный параметр, которому предшествует зарезервированное слово *Const* и за которым следует тип, называется *параметром-константой*. Формальный параметр, перед описанием которого отсутствуют зарезервированные слова и за которым следует тип, называется *параметром-значением*. Если тип параметра не указан, то такой параметр называется *нетипизированным*. Подробнее особенности передачи различных типов параметров рассмотрены ниже.

ПРИМЕР 3. В программе сортировки элементов массива «пузырьком» (пример 1) организована процедура для обмена местами двух элементов. В эту процедуру в качестве параметров передаются индексы меняемых элементов. Используется процедура с параметрами-значениями.

```
Program Primer3;
  Uses  Crt;
  Const    n = 20;                    {Описание глобальной константы  n }
  Var   A : array [1..n] of Integer;  {Описание глобальной переменной A }
        i : Integer;
  Procedure Bubble;                    {Описание процедуры сортировки  }
    Var   i, j : Integer;               {Описание локальных переменных }
    Procedure Exchange (k, l : Integer);    {Описание процедуры обмена}
      Var   temp : Integer;                      {местами двух элементов}
      Begin
        temp := A[k];
        A[k] := A[l];
        A[l] := temp;
      End;                                          {End  Exchange }
    Begin                                       {Тело процедуры Bubble}
```

```
      For i := 2 to n do
        For j := n downto i do
          If A[j-1]  > A[j] then
            Exchange (j-1, j);        {Вызов процедуры обмена элементов }
  End;                                                    {End  Bubble}
Procedure PrintMas;                                {Описание процедуры вывода}
  Var     i : Integer;                               {элементов массива на экран}
  Begin
    Writeln;
    For i :=1 to n do
      Write (A[i] : 3);
    Writeln;
  End;                                                    {End  PrintMas }
Begin                                                {Тело головной программы}
  ClrScr;
  For i := 1 to n do
    A[i] := Random(100);
  GotoXY (20, 7);
  Writeln (  ' Исходный массив ' );
  PrintMas;                                {Вызов процедуры печати элементов}
  Bubble;                                {Вызов процедуры сортировки элементов}
  GotoXY (20, 17);
  Writeln ( ' Отсортированный массив ' );
  PrintMas;                                {Вызов процедуры печати элементов}
  ReadKey
End.
```

В данной программе в заголовке процедуры *Exchange* после имени процедуры приведен список формальных параметров (*k, l : Integer*) с указанием их типа. В точке вызова в процедуру *Exchange* передаются фактические параметры (*j-1*) и (*j*), и элементы именно с этими индексами меняются местами.

Параметры могут иметь любой тип, в том числе и структурированный. Однако необходимо внимательно следить за тем, чтобы типы фактического и соответствующего ему формального параметра совпадали в случае передачи параметра-переменной и были совместимы по присваиванию в случае передачи параметра-значения.

Подводя промежуточный итог, можно выделить *основные правила написания подпрограмм*:

– необходимо определить круг задач, возлагаемых на проектируемую подпрограмму;

– в зависимости от местоположения точек вызова и решаемых задач следует определить, процедура это будет или функция;

– необходимо определить, какие данные должны передаваться в подпрограмму в качестве параметров; это должны быть лишь те данные,

изменение которых в точке вызова делает подпрограмму универсальной, пригодной для широкого использования;

– следует определить, какие данные будут локальными, а какие целесообразнее оставить глобальными (глобальных данных должно быть как можно меньше);

– необходимо выбрать способ передачи данных (параметры-значения, параметры-константы, параметры-переменные, параметры без типа); от способа передачи данных зависят эффективность и правильность выполнения программы.

6. Способы передачи данных в подпрограмму

Если параметр определен как параметр-значение, то при вызове подпрограммы автоматически создается локальная переменная, куда копируется значение фактического параметра (переменной, константы или результата вычисления выражения). После этого фактический параметр становится недоступным из подпрограммы. Подпрограмма работает с созданными локальными переменными. Таким образом, фактические и формальные параметры при передаче параметров-значений должны быть совместимыми по присваиванию. Изменения параметра-значения в теле подпрограммы не приведут к изменениям значения переменной головной программы, соответствующей фактическому параметру (рис. 4).

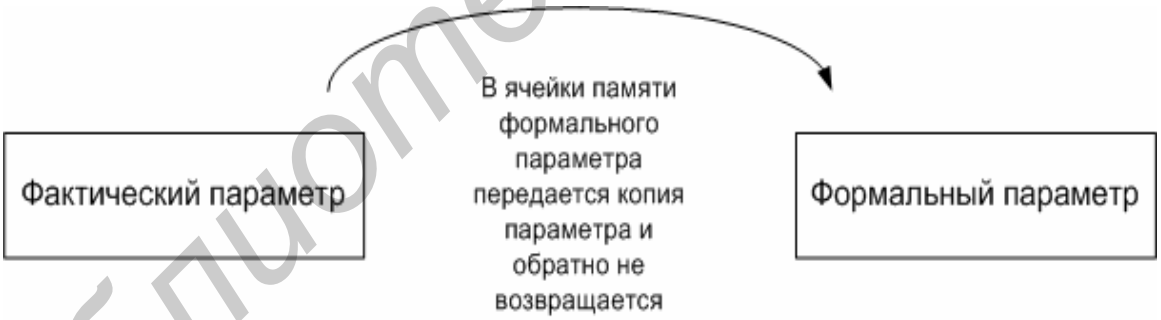

Рис. 4. Механизм передачи параметра-значения

Примером параметров-значений являются параметры *k* и *l* процедуры *Exchange*, описанной в примере 3.

Если параметр определен как параметр-константа, то при вызове подпрограммы данному параметру может соответствовать фактический параметр в виде константы, переменной нужного типа или выражения, результат вычисления которого имеет нужный тип. При передаче параметра-константы в подпрограмму передается адрес области памяти, в которой располагается фактический параметр. Однако любое изменение значения параметра-константы в подпрограмме невозможно (рис. 5).

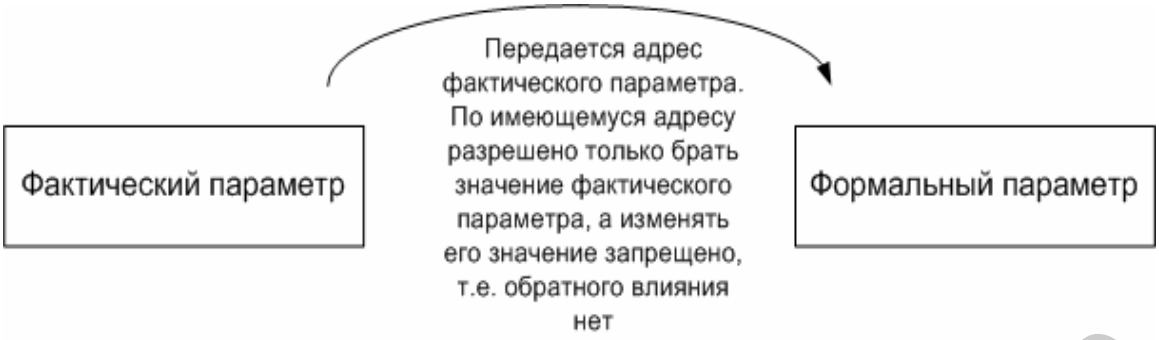

Рис. 5. Механизм передачи параметра-константы

ПРИМЕР 4. Программа вычисления суммы элементов массива, индексы которых принадлежат заданному диапазону (например с 5-го по 15-й). Диапазон для вычисления вводится с клавиатуры и передается в функцию подсчета суммы в качестве параметров-констант. За основу взята программа, рассмотренная в примере 2.

```
Program Primer4;
  Uses Crt;
  Const  n=20;
  Var  A: Array [1..n] of Integer;
       i, l, r : Integer;
  Function SumMas (Const k, n: Integer): Longint; {k, n – начальный и конечный}
    Var  i: Integer;                              {индексы элементов для суммирования}
         temp: Longint;
    Begin
      temp:=0;
      For i:=k to n do
        temp:=temp+A[i];
      SumMas:=temp;
    End;                                          {End SumMas}
  Procedure PrintMas;
    Var     i: Integer;
    Begin
      Writeln;
      For i:=1 to n do
        Write (A[i]:3);
      Writeln;
    End;                                          {End PrintMas}
  Begin                                           {Тело головной программы }
    ClrScr;
    For i:=1 to n do
      A[i]:=Random(100);
```

```
    GotoXY (20, 5);
    Writeln('Элементы масива:');
    PrintMas;
    GotoXY (3, 10);
    Writeln (' Введите левую, а затем правую границы индексов для
              суммирования');
    Readln (l, r);
    GotoXY (5, 12);
    Writeln(' Сумма элементов массива с ', l ,' по ', r, ' =  ', SumMas (l, r));
    ReadKey
End.
```

В данном примере в функцию ***SumMas*** передаются границы для суммирования *l* и *r*. Они принимаются подпрограммой в качестве параметров-констант *k* и *n*, что обеспечивает невозможность изменения границ в процессе вычисления суммы. Следует отметить, что в данном примере глобальная константа *n*, задающая размерность массива и равная 20, при вызове функции ***SumMas*** заменяется локальной, равной 12 (индекс правой границы суммирования).

Если параметр определен как параметр-переменная, то при вызове подпрограммы в нее передается адрес фактического параметра. Поэтому фактическим параметром в данном случае может быть только переменная того же типа, что и тип формального параметра. Любое изменение в теле подпрограммы формального параметра ведет к изменению в головной программе фактического параметра. Поэтому результаты работы подпрограммы могут быть переданы в вызывающую программу, используя параметры-переменные (рис. 6).

Рис. 6. Механизм передачи параметра-переменной

ПРИМЕР 5. Программа сортировки элементов массива методом выбора. Алгоритм сортировки описан в [2, 3, 5, 7]. В качестве параметров в процедуру ***Select*** передаются размерность массива (*n*) и сам массив. Используются процедуры с параметрами-переменными и параметрами-константами.

```
Program Primer5;
    Uses   Crt;
    Const    n = 20;
    Type    TM = Array [1..n] of Integer;                    {Задание типа массива}
    Var    A: TM;
           i: Integer;
    Procedure PrintMas;
       Var   i: Integer;
       Begin
          Writeln;
          For i := 1 to n do
             Write (A[i]: 3);
          Writeln;
       End;                                                          {End PrintMas }
    Procedure Swap (Var a, b: Integer);          {a и b – параметры-переменные}
       Var   temp: Integer;                                {temp – локальная переменная}
       Begin
          temp := a;
          a := b;
          b := temp;
       End;                                                          {End Swap}
    Procedure Select (Var A: TM; Const n: Integer);   {A – параметр-переменная}
       Var      i, j, min: Integer;
       Begin
          For i:=1 to n-1 do
          Begin
             min := i;
             For j := i+1 to n do
                If A[j] < A[min] then min := j;
             Swap (A[i], A[min]);
          End;
          GotoXY (10, 9);
          Writeln (' Отсортированный массив, вывод из процедуры ');
          PrintMas;
       End;                                                          {End Select}
    Begin                                                          {Вызывающая программа}
       ClrScr;
       For i := 1 to n do
          A[i] := Random(100);
       GotoXY (20, 5);
       Writeln (' Исходный массив ');
       PrintMas;
```

```
    Select (A, n);
    GotoXY (3,13);
    Writeln (' Вывод массива из головной программы после выполнения
              сортировки ');
    PrintMas;
    ReadKey
End.
```

В процедуре *Select* массив *A* объявлен как параметр-переменная. Поэтому в подпрограмму передается адрес фактического массива *A*, что экономит оперативную память. Выполнение программы иллюстрирует рис. 7.

Рис. 7. Результат выполнения сортировки массива методом выбора

Следует заметить, что при передаче в подпрограмму в качестве параметра переменной нестандартного типа возможны ошибки. Например, ошибка возникает в следующем случае:

```
Const     n = 20;
Var     A : Array [1 .. n] of Integer;
Procedure Select (Var A : Array [1 .. n] of Integer);
```

Тип формального параметра процедуры должен совпадать с типом фактического. Однако в данном случае в заголовке процедуры объявлены два новых типа – тип диапазон и тип массив. Напомним, что если в нескольких местах программы объявлены одинаковые типы, то в языке Pascal они считаются разными. Поэтому если в заголовке процедуры используются нестандартные типы, то предварительно нужно создать пользовательский тип в разделе *Type*, а затем объявить глобальные переменные и формальные параметры процедуры этого типа:

```
Const     n = 20;
Type    Mas = Array [1 .. n] of Integer;
Var     A : Mas;
Procedure Select (Var A : Mas);
```

Кроме того, нередко возникают ошибки, связанные с неправильно выбранным способом передачи данных. Например, если в процедуру *Swap* примера 5 параметры будут передаваться как параметры-значения

```
Procedure Swap (a, b: Integer);
```

то сортировки элементов массива не произойдет (рис. 8). При вызове процедуры *Swap* будут созданы локальные переменные, куда занесутся значения фактических параметров *A[i]*, *A[min]*. Поэтому замена данных произойдет между локальными переменными. Такие действия не повлекут замены местами фактических элементов массива в вызывающей подпрограмме *Select.*

Рис. 8. Результат неверно работающей программы сортировки

Следовательно, в процедуру *Swap* данные можно передавать только как параметры-переменные (пример 5).

Рассмотрим, что произойдет, если в процедуру *Select* передать массив как параметр-значение:

```
Procedure Select (A : TM; Const n : Integer);
```

На первый взгляд массив вновь оказался неотсортированным (результат работы программы аналогичен представленному на рис. 8). Однако при пошаговом выполнении в окне отладки можно убедиться, что копия массива *A*, переданная в процедуру *Select*, сортируется успешно. Ошибка в данном случае заключается в том, что процедура *PrintMas* (вывод элементов массива на экран) описана в разделе глобальных объявлений программы *Primer5.* Поэтому в теле данной процедуры доступен глобальный массив *A*, а этот массив сортировке не подвергается – сортируется лишь его копия. Для получения реальной картины в процедуру *PrintMas* необходимо передавать имя выводимого на экран массива в качестве параметра-переменной. Результат работы исправленной программы представлен на рис. 9.

Рис. 9. Сортировка локального массива в процедуре

7. Директивы компилятора

Директивы позволяют управлять работой компилятора и определенным образом корректировать его действия [9, 10]. При описании подпрограммы сразу за заголовком может следовать одна из стандартных директив компилятора. С учетом вышесказанного тело подпрограммы в общем виде имеет вид

<Тело_подпрограммы> **::= [** Far **|** Interrupt **|** Forward **|** Inline**|** Assembler **|** External **]** ; <Раздел_объявлений> <Раздел_операторов>; **.**

Директива ***Assembler*** отменяет стандартную последовательность машинных инструкций, выполняемых при входе в процедуру и выходе из нее. Тело подпрограммы в этом случае должно реализоваться с помощью команд встроенного Ассемблера.

Директива ***External*** предназначена для объединения отдельно скомпилированных процедур и функций, написанных на языке Ассемблера, и для описания процедур, импортируемых из динамически компилируемых библиотек.

Директива ***Far*** определяет, что компилятор должен создавать код подпрограммы, рассчитанный на дальнюю модель вызова.

В соответствии с архитектурой микропроцессора персонального компьютера в программах могут использоваться две модели памяти: ближняя и дальняя. Модель памяти определяет возможность вызова процедуры из различных частей программы. Если используется ближняя модель (директива *Near*), вызов возможен только в пределах 64 Кбайт (в пределах одного сегмента кода, который выделяется основной программе и каждому используемому в ней модулю). При дальней модели (директива *Far*) вызов возможен из любого сегмента. Ближняя модель экономит 1 байт и несколько микросекунд на каждом вызове подпрограммы, поэтому стандартный режим компиляции предполагает эту модель памяти. Однако при вызове параметров процедурного типа, а также в оверлейных модулях соответствующие подпрограммы должны

компилироваться с расчетом на универсальную – дальнюю – модель памяти, одинаково пригодную при любом расположении процедуры и вызывающей ее программы в памяти. По умолчанию все подпрограммы, объявленные в интерфейсной части модулей *Unit*, генерируются с расчетом на дальнюю модель вызова, а все остальные подпрограммы – на ближнюю модель.

Аналогом данной директивы является директива компилятора ***{$f+}*** (данная директива устанавливается перед заголовком процедуры и отключается (***{$f-}***) после ее тела).

Директива ***Forward*** используется при опережающем описании (разд. 11). Эта директива сообщает компилятору, что описание текущей подпрограммы следует ниже по тексту программы. Однако описание должно размещаться в данном программном модуле.

Директива ***Inline*** указывает на то, что тело подпрограммы реализуется непосредственно в машинном коде.

Директива ***Interrupt*** используется при создании подпрограмм обработки прерываний. В заголовке процедуры прерывания в качестве параметров должны быть описаны имена регистров микропроцессора.

Любая директива распространяется на текущую подпрограмму. Если для следующей подпрограммы потребовалось выполнение тех же действий компилятора, то директиву необходимо вновь указать в теле подпрограммы.

8. Процедурный тип

Основное назначение процедурного типа – предоставить возможность передачи имени функции или процедуры в качестве фактического параметра в другие процедуры или функции.

При объявлении процедурного типа используется заголовок процедуры (или функции) без указания ее имени, например:

```
Const  n = 20;
Type   TMas = Array [1 .. n] of Integer;
       TProc = Procedure  (Var A : TMas);
       TFunc = Function (x: Real ) : Real;
```

Существуют два процедурных типа: *тип-процедура* и *тип-функция* [9, 10]. Представители обоих типов описаны выше (*TProc, TFunc*).

Если подпрограмма должна передаваться в качестве фактического параметра в другую подпрограмму, она должна удовлетворять следующим ***требованиям***:

1) компилироваться в состоянии *Far*;
2) не должна быть стандартной процедурой или функцией;
3) не должна быть вложенной;
4) не должна быть подпрограммой типа *Inline*;
5) не должна быть подпрограммой прерывания (*Interrupt*);

6) ее заголовок должен соответствовать типу процедурной переменной, используемой в качестве формального параметра вызывающей подпрограммы.

На физическом уровне при присваивании процедурной переменной имени подпрограммы в данную переменную заносится *адрес подпрограммы*.

ПРИМЕР 6. Программа вычисления с точностью *Eps* = 0.001 корней трех уравнений методом деления отрезка пополам (метод дихотомии). Уравнения и отрезки для определения заданы следующим образом:

$x^2 + 2x - 11 = 0$ на отрезке [0; 3];

$e^x - 2x^2 + 4x - 2 = 0$ на отрезке [0,2; 0,3];

$x + ln(x + 0{,}5) - 0{,}5 = 0$ на отрезке [0; 2].

```
Program Primer6;
  Uses  Crt;
  Type  TFunc = Function (x: Real): Real;       {TFunc – тип-функция}
  Const  Eps = 0.001;
  Function F_1 (x: Real): Real; Far ;                {Дальний вызов}
    Begin
       F_1 := Sqr(x) + 2*x - 11;   {Возвращаемое значение функции}
    End;
  Function F_2 (x: Real):Real; Far ;                 {Дальний вызов}
    Begin
       F_2 := Exp(x) - 2*Sqr(x) + 4*x - 2;
    End;
  Function F_3 (x: Real):Real; Far ;                 {Дальний вызов}
    Begin
       F_3 := x + Ln(x + 0.5) - 0.5;
    End;
  Function FDihot (f: TFunc; a, b: Real): Real; {f – формальный параметр}
    Var fa, fc, c: Real;                              {процедурного типа}
    Begin
      Repeat
        fa:=f(a);                          {Вызов вычисляющей процедуры}
        c:=(a+b)/2;
        fc:=f(c);                          {Вызов вычисляющей процедуры}
        If fa*fc < 0 then b:=c
        Else
        Begin
           a:=c;  fa:=fc
        End
      Until  fc<=eps;
      FDihot:=c;
    End;
```

```
Begin                    {Раздел операторов вызывающей программы}
  ClrScr;
  GotoXY (5, 4);
  Write (' Для функции F_1 на отрезке 0 - 3 корень= ', FDihot (F_1, 0, 3): 8: 3);
  GotoXY (5, 6);
  Write ('Для функции F_2 на отрезке 0.2 - 0.3 корень= ', FDihot (F_2, 0.2, 0.3):
         8: 3);
  GotoXY (5, 8);
  Write ('Для функции F_3 на отрезке 0 - 2 корень= ', FDihot (F_3, 0, 2):8:3);
  ReadKey
End.
```

В данной программе объявлен тип *TFunc = Function (x: Real): Real*, описывающий функцию с параметром-значением и возвращаемым результатом типа *Real*. Функции *F_1*, *F_2*, *F_3* имеют заголовок той же структуры и директиву *Far*, а значит, могут передаваться в качестве фактического параметра, если формальный параметр имеет тип *TFunc*. Функция *FDihot* в списке формальных параметров содержит переменную *f* типа *TFunc*. При вызове функции *FDihot* данному параметру *f* ставится в соответствие имя одной из функций: *F_1*, *F_2*, *F_3*. При выполнении функции *FDihot* происходит вызов той функции, чье имя указано в качестве фактического параметра функции *FDihot*.

9. Нетипизированные параметры-переменные

Нетипизированными могут быть только параметры-переменные. Если в заголовке подпрограммы тип формального параметра-переменной не указан (конструкция (3) в разд. 5), то ему может соответствовать фактический параметр любого типа.

Существует несколько способов приведения *нетипизированного* параметра к необходимому при выполнении подпрограммы типу.

Первый способ заключается в том, что внутри подпрограммы с помощью зарезервированного слова *Absolute* объявляется локальная переменная, наложенная на переданный фактический параметр. Синтаксическая диаграмма описания налагаемой переменной имеет вид

<Описание_налагаемой_переменной> **::=** <Имя_переменной> **:**
<Тип_переменной> Absolute <Имя_параметра_без_типа>**; .**

ПРИМЕР 7. Сортировка методом выбора одномерных массивов любой размерности. Элементы массивов могут иметь различный тип, являющийся поддиапазоном типа *Byte*. Использование первого способа обеспечения совместимости типов.

```
Program Primer7;
```

```
Uses   Crt;
Const    n=20;    m=10;
Type    TM = Array [1..n] of Byte;
         TS = Array [1..m] of Char;
Var    A: TM;
       S: TS;
       i, h: Integer;
Procedure Select (Var D; Const n: Integer);          {D – параметр без типа}
  {n – параметр, определяющий размер памяти, занимаемой аргументом D}
   Var i, j: Integer; min, temp: Byte;
       MA: Array [1..Maxint] of Byte Absolute D;     {Наложение переменной}
   Begin                                  {MA типа массив на параметр без типа D}
      For i:=1 to n – 1 do
      Begin
         min:=i;
         For j:=i+1 to n do
            If MA[j] < MA[min] then min:=j;
         temp:=MA[i];
         MA[i]:=MA[min];
         MA[min]:=temp;
      End;
   End;
Begin                                  {Раздел операторов вызывающей программы}
   ClrScr;
   For i:=1 to n do
      A[i]:=Random(100);
   h:=3;
   GotoXY (20, h);
   Writeln (' Исходный массив ');
    For i := 1 to n do
       Write (A[i]: 3);
   Select (A, n);              {Передача в процедуру в качестве фактического}
   Inc (h, 4);                                  {параметра массива целых чисел}
   GotoXY (3, h);
   Writeln (' Вывод массива после выполнения сортировки ');
   For i:=1 to n do
     Write (A[i]: 3);
   Inc (h, 4);
   GotoXY(3,h);
   Writeln ('Введите элементы массива символов как строку');
   For i:=1 to m do
   Read (S[i]);
   Select (S, m);              {Передача в процедуру в качестве фактического}
   Inc (h, 4);                                  {параметра массива символов}
```

```
    GotoXY (3, h);
    Writeln (' Вывод массива после выполнения сортировки ');

    For i:=1 to m do
        Write (S[i]);
    ReadKey
End.
```

В процедуре *Select* используется *нетипизированный* параметр *D*. В теле процедуры объявляется переменная-массив *MA: Array [1..2*maxint] of byte Absolute D*, максимально возможная длина которого ограничена удвоенным значением константы *Maxint* (2·32 767 = 65 534). При вызове процедуры *Select* переменная *MA* накладывается на конкретную область памяти, занимаемую фактическим параметром. В первом случае это массив натуральных чисел *A*, во втором – массив символов *S*. При этом *MA* – фиктивная переменная, размер которой никак не влияет на объем используемой памяти. Конкретный объем обрабатываемой памяти ограничивается вторым параметром *n* процедуры *Select.*

Второй способ приведения *нетипизированного* параметра к необходимому типу заключается в объявлении в разделе описания подпрограммы нужного типа. Затем в разделе операторов данный тип ставится в соответствие параметру-переменной без типа с помощью присваивания типа переменной.

ПРИМЕР 8. Сортировка методом выбора одномерных массивов любой размерности. Элементы массивов могут иметь различный тип, являющийся поддиапазоном типа *Byte*. Использование второго способа обеспечения совместимости типов.

```
Program Primer8;
  Uses  Crt;
  Const   n = 20;   m = 10;
  Type   TM = Array [1..n] of Byte;
         TS = Array [1..m] of Char;
  Var   A: TM;
        S: TS;
        i, h: Integer;
  Procedure Select (Var D; Const n: Integer);          {D – параметр без типа}
    {n – параметр, определяющий размер памяти, занимаемой аргументом D}
     Type   TA = Array [1..Maxint] of Byte;          {Тип – массив байтов для}
     Var  i, j: Integer;  {последующего присваивания его параметру без типа D}
          min, temp: Byte;
     Begin
       For i := 1 to n – 1 do
       Begin
```

```
            min := i;
            For j := i+1 to n do
               If  TA(D)[j] < TA(D)[min] then min := j;  {Приведение параметра D}
            temp := TA(D)[i];                            {к типу TA присваиванием типа}
            TA(D)[i] := TA(D)[min];
            TA(D)[min] := temp;
          End;
      End;
   Begin                                {Раздел операторов вызывающей программы}
      ClrScr;
      For i:=1 to n do
         A[i] := Random(100);
      h := 3;
      GotoXY (20, h);
      Writeln (' Исходный массив ');
      For i := 1 to n do
        Write (A[i]: 3);
      Select (A, n);                {Передача в процедуру в качестве фактического}
      Inc (h, 4);                                  {параметра массива целых чисел}
      GotoXY (3, h);
      Writeln (' Вывод массива после выполнения сортировки ');
      For i := 1 to n do
          Write (A[i]: 3);
      Inc (h, 4 );
      GotoXY(3, h);
      Writeln ('Введите элементы массива символов как строку');
      For i:=1 to m do
         Read(S[i]);
      Select (S, m);       {Передача в процедуру в качестве фактического}
      Inc (h, 4);                                  {параметра массива символов}
      GotoXY(3, h);
      Writeln ('Вывод массива после выполнения сортировки ');
      For i:=1 to m do
        Write (S[i]);
      ReadKey
   End.
```

В данной реализации в процедуре *Select* описывается тип *TA = Array [1..Maxint] of Byte;* , который впоследствии присваивается *нетипизированному* параметру *D:*

TA(D).

10. Рекурсия

Рекурсия – это способ организации вычислительного процесса, при котором подпрограмма в ходе выполнения производит вызов из раздела операторов самой себя. Согласно [3] рекурсивным называется обьект частично состоящий или определяемый с помощью самого себя. Рекурсия, как правило, применяется в тех случаях, когда основную задачу можно разбить на подзадачи той же структуры, что и первоначальная задача.

Основным достоинством рекурсивных программ по сравнению с нерекурсивными является наглядность и компактность. Однако некоторые рекурсивные программы выполняются дольше чем итерационные и требуют значительных затрат оперативной памяти. Это связано с тем, что каждый раз при вызове подпрограммы выделяется новый блок памяти и начинает работать, по сути, новая подпрограмма. Необходимо помнить, что если в рекурсивной подпрограмме существуют локальные данные (константы, типы данных, переменные, подпрограммы), то при каждом следующем вызове такой подпрограммы порождается новый набор локальных данных, значения которых отличаются от данных с теми же именами на предыдущем шаге рекурсии. Если рекурсивная подпрограмма содержит правила изменения глобальных данных, то при каждом вызове подпрограммы произойдет новое уточнение этих данных.

Каждый набор локальных данных и текущие значения фактических параметров помещаются в программный стек в виде экземпляра памяти. *Программным стеком* называется определенным образом организованная область оперативной памяти, в которую помещаются локальные данные и фактические параметры на время выполнения рекурсивной подпрограммы и в которой доступен экземпляр памяти, созданный последним.

Из-за многократного выделения памяти под локальные данные рекурсивной подпрограммы может произойти переполнение программного стека. Следовательно, особое внимание надо уделять подбору оптимального количества локальных данных, выбору типа соответствующего параметра или переменной и способа передачи данных в подпрограмму. По умолчанию размер стека равен 16 Кбайт. С помощью директивы компилятора *$M* его можно изменить [9, 10].

Особое внимание необходимо уделить корректному выходу из рекурсивной подпрограммы. Рекурсивный вызов подпрограммы должен управляться некоторым условием, которое в определенный момент перестает выполняться. Пока условие истинно, рекурсия повторяется (осуществляется *рекурсивный спуск*), но как только условие становится ложным, начинается последовательный *рекурсивный возврат* из всех созданных копий подпрограммы.

По месторасположению условия входа в рекурсию различают *три типа* рекурсивных подпрограмм [9]:

тип 1 – с выполнением действий на рекурсивном спуске;

тип 2 – с выполнением действий на рекурсивном возврате;

тип 3 – с выполнением действий на рекурсивном спуске и возврате.

Данным типам подпрограмм соответствует следующая структура раздела операторов:

<Раздел операторов_рекурсивной_подпрограммы_типа_1> **::=** Begin <Действия _подпрограммы> If <Условие_входа> Then <Идентификатор_ подпрограммы> **[(**<Список_фактических_параметров>**)]** ; End **.**

<Раздел операторов_рекурсивной_подпрограммы_типа_2> **::=** Begin If <Условие_входа> Then <Идентификатор_подпрограммы> **[(**<Список_фактических_ параметров >**)]** ; <Действия_подпрограммы> End **.**

< Раздел операторов_рекурсивной_подпрограммы_типа_3> **::=** Begin <Действия _подпрограммы> If <Условие_входа> Then <Идентификатор_подпрограммы> **[(**<Список_фактических_параметров >**)]** ; <Действия_ подпрограммы> End **.**

Для решения многих задач безразлично, каким способом организована рекурсивная подпрограмма. Однако есть задачи, при решении которых необходимо сознательно выбирать тип организации рекурсивных подпрограмм.

ПРИМЕР 9. Организация рекурсивных процедур. Программа вывода массива в прямом и обратном порядке.

```
Program Primer9;
  Uses Crt;
  Const n = 20;
  Type Tmas = Array [1..n] of Integer;
  Var M: Tmas; i: Integer;
  Procedure Print_1(Var A: Tmas; k: Integer);  {Вывод массива в прямом}
    Var i: Integer;                                          {порядке}
    Begin
      Write (A[k]: 3);
      If k<n then Print_1(A, k+1);          {Рекурсивный вызов процедуры}
    End;                                                  {End  Print_1}
  Procedure Print_2 (Var A: Tmas; k: Integer); {Вывод массива в обратном }
    Var i: Integer;                                          порядке}
    Begin
      If k<n then Print_2(A, k+1);          {Рекурсивный вызов процедуры}
      Write (A[k]: 3);
    End;                                                  {End  Print_2 }
  Begin                          {Раздел операторов вызывающей программы}
    ClrScr;
```

```
    For i:=1 to n do
       M[i]:=Random (100);
    GotoXY (10, 5);
    Writeln ('Вывод массива в прямом порядке');
    Print_1(M, 1);
    GotoXY (10, 8);
    Writeln ('Вывод массива в обратном порядке');

    Print_2(M, 1);
    ReadKey
  End.
```

Процедура *Print_1* является процедурой с выполнением действий на рекурсивном спуске. Вначале происходит печать очередного элемента, а затем проверка на необходимость вызова процедуры с параметром (*k +1*). Процедура *Print_2* является процедурой с выполнением действий на рекурсивном возврате. Рекурсивный вызов процедуры продолжается до тех пор, пока будет выполняться условие $k < n$. Как только условие окажется ложным, произойдет печать элемента массива с текущим индексом и возврат на предыдущий шаг рекурсии, где будет произведена печать элемента с индексом, на единицу меньшим, и так далее до возврата в точку вызова в головной программе.

Рекурсивные алгоритмы особенно подходят для задач, где обрабатываемые данные определяются в терминах рекурсии. Однако это не означает, что рекурсивное определение данных гарантирует эффективность использования рекурсивных алгоритмов для решения таких задач [3].

Ярким примером, когда рекурсию использовать неэффективно, является программа, рассмотренная в примере 10.

ПРИМЕР 10. Программа вычисления 5-го числа Фибоначчи (в общем *n-го* числа). Числа Фибоначчи определяются рекурсивным соотношением [3]

$$fib_{n+1} = fib_n + fib_{n-1}, \text{ для } n > 1 \text{ и } fib_0 = 0,\ fib_1 = 1.$$

```
Program Primer10;
  Uses Crt;
  Function Fib (k: Integer): Integer;
    Begin
      If k = 0 then Fib := 0
      else
        If k = 1 then Fib := 1
        else
          Fib := Fib (k – 1)+Fib (k – 2);        {Рекурсивный вызов функции}
    End;
  Begin
```

```
    ClrScr;
    Read (n);
    Writeln ('Число Фибоначчи = ', Fib (5);
    ReadKey
End.
```

При каждом обращении к функции *Fib* для *k >1* дважды происходит активизация подпрограммы (на рис. 10 представлена схема вызовов рекурсивной подпрограммы для *k* = 5), а значит, общее число вызовов растет экспоненциально с ростом значения параметра *k*. В подобных случаях затраты времени и памяти не оправданы. Такой рекурсивный процесс следует заменить итерационным.

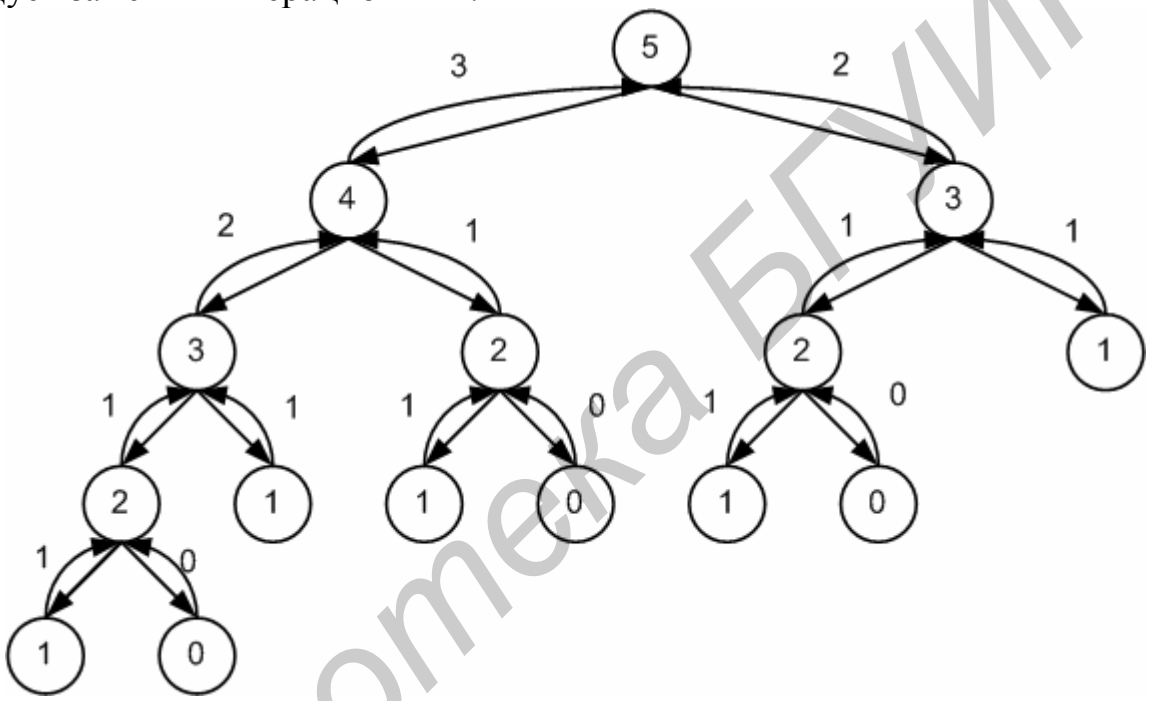

Рис. 10. Дерево обращения к функции *Fib*

Типичным является применение рекурсивных вычислений в задачах синтаксического анализа текста.

ПРИМЕР 11. Программа ввода формулы из стандартного входного файла и вычисления ее значения. Формула имеет вид

<Формула> **::=** <Цифра> **|** **(** (<Формула> <Знак> <Формула>) **)** **.**
<Знак> **::=** **+** **|** **–** **|** ***** **.**
<Цифра> **::=** 0 **|** 1 **|** 2 **|** 3 **|** 4 **|** 5 **|** 6 **|** 7 **|** 8 **|** 9 **.**

```
Program Primer11;
  Function F: Integer;          {F читает из начала входного файла символы,}
      {образующие законченную формулу, и вычисляет ее значение как целое}
```

```
Var c, op: Char;    {в op заносится <Знак>, в c – очередной символ из файла}
    x, y: Integer;
Begin
  Read (c);
  If (c>='0') and (c<='9') then
      F := Ord (c) – Ord ('0')                    {Цифра есть формула}
  else
     Begin                                        {Началась формула вида (x op y)}
        x := F;                                   {Рекурсивный вызов функции}
        Read (op);
        y := F;                                   {Рекурсивный вызов функции}
        Case op of
            '+':  F:=x+y;
            '–':  F:=x-y;
            '*':  F:=x*y
        End;
        Read (c)                                  {Пропуск ')'}
     End
End;                                              {End F}
Begin
 Writeln (F)
End.
```

ПРИМЕР 12. Программа вычисления значения $f = x^n$ с использованием рекурсивной функции. Вычисление функции f реализовано в виде

$$f = \begin{cases} 1, при \_\_\_\_ n = 0 \\ \frac{1}{x^{|n|}}, при \_\_\_\_ n < 0 \\ x * x^{n-1}, при \_\_\_\_ n > 0 \end{cases}.$$

```
{$S+}  {Директива контроля переполнения стека}
Program Primer12;
  Uses  Crt;
  Var   n: Integer; x: Real;
  Function Fexp (a: Real; st: Integer): Real;
    Begin
      If  st = 0 then Fexp := 1              {Точка останова рекурсивного спуска}
      else
        If st < 0  then Fexp := 1 / Fexp (a, abs (st)) {Рекурсивный вызов функции}
        else
          Fexp := a * Fexp (a, st – 1);      {Рекурсивный вызов функции}
     End;                                    {End Fexp}
```

```
Begin
   ClrScr;
   GotoXY (10, 5);
   Writeln ('Введите число :');
   Readln (x);
   GotoXY (10, 9);
   Write ('Введите степень :');
   Readln (n);
   GotoXY (10, 13);
   Write ('Число ',x,' в степени ',n,' = ', Fexp (x, n));
   ReadKey
End.
```

Fexp является функцией с выполнением действий на рекурсивном возврате. Если начальное значение степени – отрицательное число, функция вызывается повторно с параметром степени ***st***, взятым по абсолютному значению. При последующих вызовах параметр ***st*** проверяется на равенство нулю. Если результат сравнения ложен, функция вызывается вновь с параметром ***(st – 1)***.
Если же ***st*** достигает значения 0, возвращаемому значению функции ***Fexp*** присваивается значение 1 и происходит возврат на предыдущий уровень рекурсии.

Если для вычисляемого результата достаточно типа ***Real***, то данная программа является вполне работоспособной. Однако если ожидаемое значение не вмещается в диапазон этого типа и необходимо выбрать другой вещественный тип, требующий подключения сопроцессора, то с большой долей вероятности при запуске программы возникнет ошибка, связанная с переполнением стека. Связано это с тем, что возвращаемое значение вещественной функции возвращается через аппаратный стек сопроцессора, рассчитанный только на 8 уровней. Чтобы избежать ошибки, для разгрузки стека сопроцессора необходимо ввести локальную переменную того же типа, который имеет функция. В этой буферной переменной будет сохраняться возвращаемый функцией результат.

ПРИМЕР 13. Более эффективная реализация задачи, приведенной в примере 12.

```
{$S+}  {$N+}              {Включение директив контроля  переполнения стека и}
                          {использования сопроцессора математических операций}
Program Primer13;
Uses   Crt;
Var   n: Integer; x: Real;
Function Fexp ( a: Real; st: Integer): Extended;
Var  f: Extended;         {Локальная переменная, вводится для разгрузки стека}
   Begin
```

```
    If st = 0  then Fexp := 1
    else
        If st < 0  then
        Begin
            f := Fexp (a, abs (st));        {Сохранение результата в буферную}
            Fexp:=1/f;                                          {переменную}
        End
       else
        Begin
           f := Fexp (a, st – 1);
            Fexp := f * a;
        End;
  End;
Begin
    ClrScr;
    GotoXY (10, 5);
    Writeln ('Введите число :');
    Readln (x);
    GotoXY (10, 9);
    Writeln ('Введите степень :');
    Readln (n);
    GotoXY (10, 13);
    Writeln ('Число ', x, ' в степени  ', n, ' = ', Fexp (x, n));
    ReadKey
End.
```

11. Взаимная рекурсия

В ряде задач вычисления должны быть организованы таким образом, что одна подпрограмма вызывает другую, а та в свою очередь обращается к первой. Такая организация вычислений называется ***взаимной*** (или ***косвенной***) ***рекурсией***. Для ее реализации используется опережающее и определяющее описание одной из подпрограмм. Вторая подпрограмма описывается обычным образом.

Опережающее описание состоит из заголовка подпрограммы и следующей за ним директивы *Forward*:

<Опережающее_описание> **::=** <Заголовок_подпрограммы> **;** Forward **; .**

Определяющее описание подпрограммы содержит ее заголовок без параметров и возвращаемого значения и ее тело:

<Определяющее описание> **::= (** Procedure **|** Function **)** <Имя_подпрограммы> **;**
<Тело_подпрограммы> **; .**

ПРИМЕР 14. Использование явной и взаимной рекурсии. Программа вычисления функции:

$$y = \sum_{i=0}^{10} (x_i^2 - p_i^2),$$

где $x_i = 0.5(x_{i-1} + p_{i-1})$, $p_i = x_{i-1} \cdot p_{i-1}$, $p_0 = 2$, $x_0 = 1$.

```
Program Primer14;
  Uses Crt;
  Function Fp (i: Integer):Real; Forward; {Опережающее описание функции Fp}
  Function Fx (i: Integer): Real;
    Begin
      If i = 0  then Fx := 1
      else  Fx := 0.5*(Fx (i – 1)+Fp (i – 1));          {Явная и взаимная рекурсия}
    End;
  Function Fp;                                          {Определяющее описание функции Fp}
    Begin
      If i = 0  then Fp := 2
      else  Fp := Fx (i – 1)*Fp (i – 1);                {Явная и взаимная рекурсия}
    End;
  Function Sum (n: Integer): Real;
    Var i: Integer;
        tempx, tempp, tsum: Real;
     Begin
      tsum := 0;
      For i := 0 to n do
      Begin
        tempx := Fx (i);                                {Первый вызов функции Fx}
        tempp := Fp (i);                                {Первый вызов функции Fp}
        tsum := tsum + tempx * tempx – tempp * tempp;
      End;
      Sum := tsum
    End;
Begin
  ClrScr;
  Writeln(' Результат вычисления = ', Sum (10): 10: 5);
  ReadKey
End.
```

Из условия видно, что значение *x* вычисляется с использованием значения *p*. И наоборот, значение переменной *p* вычисляется с использованием значения *x*, полученного на предыдущем шаге. Функция *Fp* описывается ниже

точки ее вызова в функции *Fx*. Следовательно, необходимо использовать опережающее описание функции *Fp*.

Данная задача наглядно описывает явную и взаимную рекурсию. Однако следует отметить, что, как и в случае с числами Фибоначчи (пример 10), применение в этой программе рекурсивного алгоритма является неоправданным, поскольку требует существенных дополнительных затрат памяти и времени выполнения.

12. Способы структурной организации подпрограмм

Существуют три способа структурной организации подпрограмм по отношению к вызывающей их программе:

1 – в виде внутренних подпрограмм вызывающей программы;

2 – в виде библиотечных модулей пользователя *Unit*;

2 – в виде внешних файлов (директива *Include*);

Первый способ подробно рассмотрен в предыдущих разделах работы. При данном способе подпрограмма описывается в разделе объявлений содержащей ее программы или подпрограммы.

ПРИМЕР 15. Программа поиска максимального элемента в каждой строке двумерного массива. Использование внутренней подпрограммы.

```
Program Primer15;
  Uses Crt;
  Type
    Massive=Array[1..10,1..6] of Integer;
 Var
    m:Massive;
    i,j,k,num, MyMax:Integer;
    Function FindMax(Var m1:Massive; n:Integer):Integer; {Функция определения}
     Var                          {максимального значения в n-й строке массива}
       Max, Count: Integer;
     Begin
       Max:=m1[n,1];
       Count:=2;
       While Count <= 6 do
       Begin
          if m1[n,Count]>Max then
             Max:=m1[n,Count];
          Inc(Count);
       End;
       FindMax:=Max;
     End;                                                    {End FindMax}
  Begin
```

```
  ClrScr;
  For i:=1 to 10 do
  Begin
    For j:=1 to 6 do
    Begin
      m[j,i] :=Random(9);                    {Заполнение массива}
    End;
  End;
  For i:=1 to 10 do
  Begin
    For j:=1 to 6 do
    Begin
      Write(' ',m[i,j]);
    End;
    Writeln
  End;
  MyMax:=FindMax(m,4);     {Переменной MyMax присваивается результат}
  Writeln('Max is: ', MyMax);                {работы функции FindMax}
   ReadKey;
End.
```

Организация подпрограмм в виде библиотечных модулей

При данном способе организации процедур и функций они описываются в библиотечном пользовательском модуле *Unit*. В этом случае в предложении использования *Uses* вызывающей программы указывается имя модуля, содержащего необходимые подпрограммы. После этого программа может вызывать подпрограммы модуля, как если бы они были описаны в ней самой.

В отличие от программы модуль не может быть запущен на выполнение самостоятельно, он может только участвовать в построении программы или другого модуля.

Достоинства использования модулей *Unit*:

– наличие модулей позволяет использовать модульное программирование, то есть представлять программу в виде модулей и при необходимости корректировать отдельные модули, а не всю программу в целом;

– модули компилируются независимо друг от друга и от использующей их программы; при подключении модуля к другой программе он не компилируется заново. Это позволяет сократить время компиляции больших программ;

– наличие модулей позволяет создавать большие программы с суммарным размером исполнимого кода большим чем 64 Кбайт.

ПРИМЕР 16. Программа поиска максимального элемента в каждой строке двумерного массива. Использование модуля для хранения подпрограммы.

```
Program Primer16;                          {Тело основной программы}
  Uses Crt, Mylib;                         {Подключение библиотеки Mylib}
  Var
     i,j :Integer;
   Begin
      ClrScr;
      For i:=1 to 10 do
      Begin
        For j:=1 to 6 do
        Begin
          m[j,i] :=Random(9);                    {Заполнение массива }
        End;
      End;
      For i:=1 to 10 do
      Begin
        For j:=1 to 6 do
        Begin
          Write(' ',m[i,j]);
        End;
        Writeln;
    End;
    MyMax:=FindMax(m,4);      {Переменной MyMax присваивается результат}
   Writeln('Max is: ',MyMax);                {работы функции FindMax}
    ReadKey;
 End.
{*********************************************************}
Unit MyLib; {Отдельно транслируемая пользовательская библиотека  MyLib}
 Interface                                          {Секция  Interface}
  Type
     Massive=Array[1..10,1..6] of Integer;
     Function FindMax(Var m1:Massive; n:Integer):Integer;
 Implementation                                   {Секция Implementation}
  Function FindMax;                                {Функция определения}
  Var                          {максимального значения в n-й строке массива}
    Max, Count:Integer;
  Begin
     Max:=m1[n,1];
     Count:=2;
     While Count <= 6 do
     Begin
```

```
      If m1[n,Count]>Max then
          Max:=m1[n,Count];
      Inc(Count);
    End;
    FindMax:=Max
  End                                                    {End FindMax}
End.
```

Организация подпрограмм в виде отдельных файлов

При данном способе подключения процедур и функций к программе используется директива компилятора *Include* («Включение в программу внешнего текстового файла»):

{$I <Имя_файла>}.

Данная директива сообщает компилятору о необходимости включить в компиляцию названный файл. Таким образом, отлаженные законченные подпрограммы могут быть записаны в отдельные файлы и включены в основную программу по месту директивы их включения. Это замедляет процесс компиляции, но сокращает исходный код программы (она может превратиться в цепочку подключаемых файлов) и упрощает ее отладку.

Примеры включения исходных файлов:

{$I F1.pas} (или эквивалентно {$I F1})
{$I C:\Dir1\Proc1.ini}

По умолчанию расширением <Имени_файла> является **.pas*.

Включаемый файл должен удовлетворять *условиям*:

1) при его включении на место директивы ***{$I …}*** должен вписаться в структуру и смысл программы без ошибок;

2) должен содержать законченный смысловой фрагмент, то есть подпрограмма должна храниться целиком в одном файле;

3) включаемый файл не может быть указан в середине раздела операторов.

Включаемые файлы сами могут содержать директивы ***{$I …}***. Максимальный уровень такой вложенности равен восьми.

К *недостаткам* такого подключения к программе внешнего файла по сравнению с использованием библиотечных модулей можно отнести следующее:

а) подключаемые файлы каждый раз компилируются заново. Это увеличивает время компиляции;

б) размер программы не может превышать 64 Кбайт.

ПРИМЕР 17. Программа поиска максимального элемента в каждой строке двумерного массива. Использование внешнего файла для хранения подпрограммы.

```
Program Primer17;                                   {Тело основной программы}
  Uses Crt;
  Type Massive=Array[1..10,1..6] of Integer;
  Var  m:Massive;
       i,j,k,num,MyMax:Integer;
  {$I d:\work\link1.pas}     {Директива вставки фрагмента исходного текста}
   Begin                                            {с указанием пути его поиска}
     ClrScr;
     For i:=1 to 10 do
     Begin
        For j:=1 to 6 do
        Begin
           m[j,i] :=Random(9)                       {Заполнение массива}
        End;
     End;
     For i:=1 to 10 do
     Begin
        For j:=1 to 6 do
        Begin
          Write(' ',m[i,j]);
        End;
        Writeln;
     End;
     MyMax:=FindMax(m,4);      {Переменной MyMax присваивается результат}
     Writeln('Max is: ',MyMax);                     {работы функции FindMax}
     ReadKey;
 End.
{************************************************************}
Function FindMax(Var m1:Massive; n:Integer):Integer;     {Функция определения}
 Var                                {максимального значения в n-ой строке массива}
  Max, Count : Integer;               {Исходный текст подпрограммы хранится в }
  Begin                                                          {d:\work\link1.pas}
     Max:=m1[n,1];
     Count:=2;
     While Count <= 6 do
     Begin
        If m1[n,Count]>Max then
           Max:=m1[n,Count];
        Inc(Count)
      End;
```

```
  FindMax:=Max
End;                                                    {End FindMax}
```

Задание №1
Подпрограммы с параметрами процедурного типа

В соответствии с заданными преподавателем методами численных вычислений (прил. 1) и вариантом индивидуального задания (прил. 2) разработать программу для вычисления интеграла.

Для вычисления подынтегральных функций и непосредственно интеграла использовать функции и процедуры с процедурными параметрами. Интегралы вычислять с точностью $e = 10^{-2}, 10^{-3}$. Проверку правильности расчетов заданных интегралов выполнить в среде *MathCad* [8].

Построить графики поведения подынтегральных функций на отрезке интегрирования, используя графический режим. Результаты расчетов свести в таблицу вида

	1-й метод				2-й метод			
	$e = 10^{-2}$		$e = 10^{-3}$		$e = 10^{-2}$		$e = 10^{-3}$	
	Значение	N	Значение	N	Значение	N	Значение	N
1-й интеграл								
2-й интеграл								
3-й интеграл								
4-й интеграл								

В данной таблице «значение» – значение вычисленного интеграла; N – число отрезков разбиения, при котором достигнута заданная точность.

Задание №2
Рекурсивные подпрограммы

В приведенных ниже заданиях используются следующие метасимволы упрощенной формы метаязыка Бэкуса–Наура:

::= – обозначает «по определению есть»;

[ ] – необязательная часть конструкции;
{ } – повторение конструкции, заключенной в них 0, 1 или более раз;
| – выбор, альтернатива;
. – конец конструкции;
( ) – выделенная часть конструкции в альтернативе.

Следует обратить внимание на то, что элемент ( ) используется не как символ метаязыка Бэкуса–Наура, а как часть описываемого понятия.

1. Ввести текст. Определить, соответствует ли он понятию «*Простое_ выражение»:*

<*Простое_ выражение*> ::= <*Простой_ идентификатор*>**|(** (<*Простое_ выражение*> < *Знак_операции*> < *Простое_выражение*>) **).**
<*Простой_ идентификатор*>::= < *Буква* >**.**
<*Знак_ операции*>::= + **|** - **|** * **.**

Примеры правильной записи «*Простого_ выражения»:* a; (a + b); (a + (b – c)) .

2. Ввести текст. Определить, соответствует ли он понятию «*Вещественное_число»:*

<*Вещественное_число*> ::= (<*Целое_число. Целое_без_знака*>**) |**
(<*Целое_число*> **.** <*Целое_без_знака*> *E* <*Целое_число*>**) | (** < *Целое_число*> *E* <*Целое_число*>**).**
<*Целое_без_знака*>::= <*Цифра*> {<*Цифра*>} **.**
<*Целое_число*>::= [+ **|** - **]** <*Целое_без_знака*> **.**

Примеры правильной записи «*Вещественного_числа»:* 1.24;12.2E5; +14.81E – 2.

3. Ввести текст. Определить, соответствует ли он понятию «*Простое_логическое»* выражение*:*

<*Простое_логическое*> ::= *True* **|** *False* **|** <*Простой_идентификатор*>**|**
(*Not* <*Простое_логическое*>**) | (** (<*Простое_логическое*><*Знак_операции*> <*Простое_логическое*>) **).**
<*Простой_идентификатор*> :: = <*Буква*> **.**
<*Знак_операции*>::= *And* **|** *Or* **.**

Примеры правильной записи «*Простого_логического»* выражения*:*
True; (*True And a*); ((*True Or False) Or d*).

4. Написать программу, которая по заданному «*Простому_логическому»* выражению (определение понятия содержится в формулировке задачи №4), не

содержащему вхождений простых идентификаторов, вычисляет и выводит на экран значение этого выражения.

5. Ввести текст. Определить, соответствует ли он понятию «*Константное_ выражение»:*

<*Константное_выражение*> ::= (<*Цифра*> {<*Цифра*>}**) |**
(<*Константное_выражение*><*Знак_операции*><*Константное_выражение* > **) .**

<*Знак_операции*>::= + **|** – **|** * **.**

Примеры правильной записи «*Константного_ выражения»:* 24; (634 + 45); ((7 – 121) * 32).

6. Написать программу, которая по заданному «*Константному_выражению»* (определение понятия содержится в формулировке задачи №5) вычисляет и выводит на экран значение этого выражения.

7. Ввести текст. Определить, соответствует ли он понятию «*Сумма»*:

<*Сумма*>::=<*Целое*> **|(**<*Сумма*><*Знак_операции*><*Целое*>**) .**
<*Целое*>::= <*Цифра*> {<*Цифра*>} **.**
<*Знак_операции*>::= + **|** - **.**

Примеры правильной записи «*Суммы»*: 24; 34 + 45; 7 – 121+ 32.

8. Написать программу, которая по заданной «*Сумме»* (определение понятия содержится в формулировке задачи №7) вычисляет и выводит значение этой суммы.

9. Ввести текст. Определить, соответствует ли он понятию «*Список_параметров»*:

<*Список_параметров*>::= <*Параметр*> {, <*Параметр*>} **.**
<*Параметр*>::= (<*Имя*>= <*Цифра*> {<*Цифра*>} **) |**
(<*Имя*>=(<*Список_параметров*>) **) >.**
<*Имя*>::=<*Буква*> {<*Буква*>} **.**

Примеры правильной записи «*Списка_параметров»*: a = 13, b = (d = 34, asd = 66, g = 2).

10. Ввести текст. Определить, соответствует ли он понятию «*Список_списков»*:

<*Список_списков*>::= <*Список*>{; <*Список*>} **.**
<*Список*>::= <*Элемент*> {, <*Элемент*>} **.**

<***Элемент***>::=<*Буква*> {<*Буква*>} .

Примеры правильной записи «*Списка_списков*»: abc; b, c, d; asd; h.

11. Для заданного целого *N* вычислить значение суммы:

$$\sum_{i_1=1}^{N}\sum_{i_2=1}^{N}\sum_{i_3=1}^{N}\ldots\sum_{i_n=1}^{N}\left(\frac{1}{i_1+i_2+\ldots+i_n}\right)$$

12. Во входном файле задан текст, за которым следует точка. Проверить, является ли этот текст правильной записью «*Формулы*» следующего вида:

<***Формула***>::=<*Цифра*> | ((<*Формула*><*Знак*><*Формула*>)) .
<***Знак***>::=+ | - |* .
<***Цифра***>::=*0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* .

Примеры правильной записи «*Формулы*»: 2. ; (3 + 5). ; ((4 + 6)*(6 – 2)).

13. Во входном файле записано без ошибок логическое выражение следующего вида:

<***Логическое_ выражение***>::=*True* | *False* | (<*Операция*>(<*Операнды*>)) .
<***Операция***>::=*Not* | *And* | *Or* .
<***Операнды***>::= <*Операнд*>{, <*Операнд*>} .
<***Операнд***>::=<*Логическое_ выражение*> .

Ввести это выражение и вычислить его значение. Например результатом вычисления выражения *And(or(false,not(false)), true, not(true))* будет *false.*

У операций *And* и *Or* может быть любое число операндов, у *Not* – только один операнд.

14. Имеется *N* населенных пунктов, пронумерованных от 1 до *N* (*N* = 10). Некоторые пары пунктов соединены дорогами. Определить, можно ли по дорогам попасть из пункта 1 в *N*-й пункт. Информация о дорогах задается в виде последовательности пар чисел i и j (i < j), указывающих что i-й и j-й пункты соединены дорогой. Признак конца этой последовательности – пара нулей.

15. Дано *N* различных натуральных чисел (*N* = 5). Напечатать все перестановки этих чисел.

16. Составить программу обхода шахматным конем шахматной доски, побывав на каждом поле по одному разу. Рассмотреть все способы обхода.

17. Составить программу расстановки 8 ферзей на шахматной доске 8 х 8 так, чтобы они не били друг друга. Рассмотреть все способы расстановки.

18. Расставить на шахматной доске как можно больше ферзей таким образом, чтобы при снятии любого из них появлялось ровно одно неатакованное поле.

Литература

1. ГОСТ 19.701-90. Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения.

2. Ахо, А. Структуры данных и алгоритмы / А. Ахо, Д. Хопкрофт, Д. Ульман; пер. с англ. – М. : Вильямс, 2003.

3. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт. – М. : Мир, 1989.

4. Основы алгоритмизации и программирования : лаб. практикум по курсу для студ. спец. «Программное обеспечение информационных технологий» дневн. формы обуч. В 4 ч. Ч.1 / Л. А. Глухова [и др.]. – Минск : БГУИР, 2004.

5. Глухова, Л. А. Основы алгоритмизации и программирования : лаб. практикум для студ. спец. «Программное обеспечение информационных технологий» дневн. формы обуч. В 4 ч. Ч.2 / Л. А. Глухова, Е. П. Фадеева, Е. Е. Фадеева. – Минск : БГУИР, 2005.

6. Демидович, Б. П. Основы вычислительной математики / Б. П. Демидович, И. А. Марон. – М. : Гос. изд-во физико-математической литературы, 1960.

7. Кнут, Д. Искусство программирования для ЭВМ. В 3 т. Т. 2 / Д. Кнут. – М. : Мир, 1978.

8. MathCad 6.0 Plus. Финансовые, инженерные и научные расчеты в среде Windows. 2-е изд. – М. : Филинь, 1997.

9. Марченко, А. И. Программирование в среде Turbo Pascal 7.0: учебн. пособие / А. И. Марченко, Л. А. Марченко. – М. : Бином Универсал, Киев : Юниор, 1997.

10. Фаронов, В. В. Turbo Pascal 7.0. Начальный курс : учебн. пособие / В. В. Фаронов. – М. : Холидж, 1999.

ПРИЛОЖЕНИЕ 1

Численное интегрирование

При вычислении определенного интеграла функции *f(x)* на промежутке от *a* до *b*, где *f(x)* – непрерывная на данном отрезке функция, можно воспользоваться формулой Ньютона–Лейбница:

$$\int_a^b f(x)dx = F(b) - F(a),$$

где *F(a), F(b)* – значения первообразной интегрируемой функции *f(x)*, вычисленные в точках $x = a$ и $x = b$ соответственно. Однако на практике не всегда удается найти первообразную в аналитической форме, но значение определенного интеграла всегда можно довести до числового ответа. Таким образом, под задачей численного интегрирования понимают приближенное вычисление значения интеграла при условии, что известны отдельные значения подынтегральной функции. При вычислении определенного интеграла следует помнить *геометрический смысл* вычисления определенного интеграла: значение определенного интеграла равно площади фигуры, ограниченной графиком функции, осью *0x* и прямыми $x = a$ и $x = b$. Для приближенного интегрирования существует много численных методов. Рассмотрим некоторые из них [6].

Метод левых прямоугольников

Отрезок интегрирования разбивается на n равных частей (рис. П1.1) с шагом

$$h = \frac{(b-a)}{n}.$$

Для вычисления интеграла по формуле левых прямоугольников используется формула

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

где $x_i = a + i \cdot h$, $i = 0 \ldots n-1$.

Метод правых прямоугольников

Отрезок интегрирования разбивается на n равных частей (рис. П1.2) с шагом

$$h = \frac{(b-a)}{n}.$$

Для вычисления интеграла по формуле правых прямоугольников используется формула

$$I = h \cdot \sum_{i=1}^{n} f(x_i),$$

где $x_i = a + i \cdot h$, $i = 1 \ldots n$.

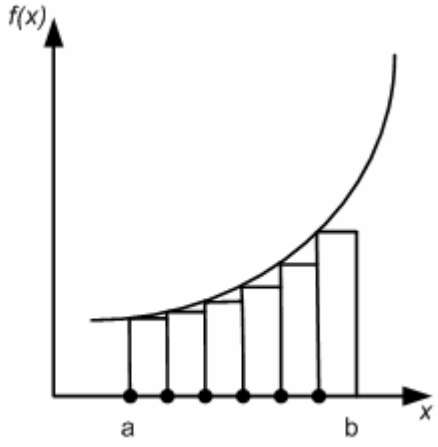

Рис. П1.1. Иллюстрация к методу левых прямоугольников

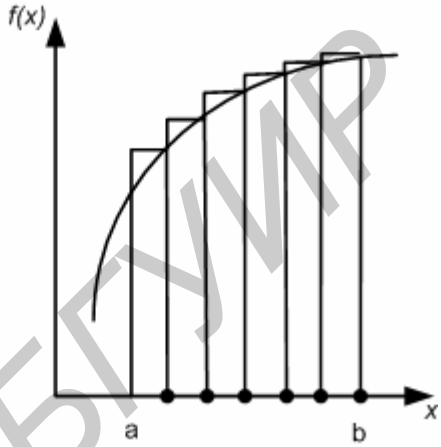

Рис. П1.2. Иллюстрация к методу правых прямоугольников

Метод центральных прямоугольников

Отрезок интегрирования разбивается на n равных частей (рис. П1.3) с шагом

$$h = \frac{(b-a)}{n}.$$

Для вычисления интеграла по формуле центральных прямоугольников используется формула

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

где $x_i = a + i \cdot h + \frac{h}{2}$, $i = 0 \ldots n-1$.

Метод трапеций

Отрезок интегрирования разбивается на n равных частей (рис. П1.4) с шагом

$$h = \frac{(b-a)}{n}.$$

Для вычисления интеграла по формуле трапеций используется формула

$$I = h \cdot (\frac{f(x_0) + f(x_n)}{2} + f(x_1) + f(x_2) + \mathbf{K} + f(x_{n-1})),$$

где $x_i = a + i \cdot h$, $i = 0 \ldots n$.

Окончание прил. 1

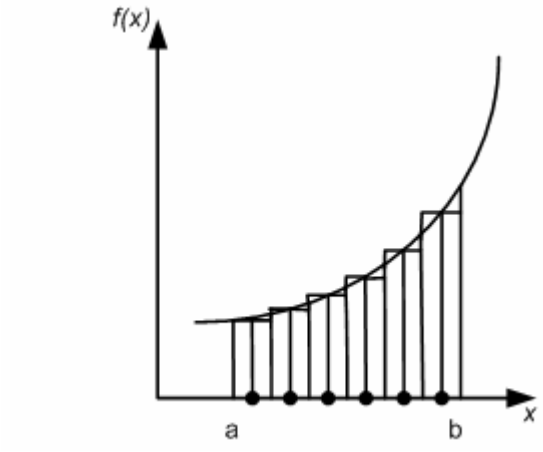

Рис. П1.3. Иллюстрация к методу центральных прямоугольников

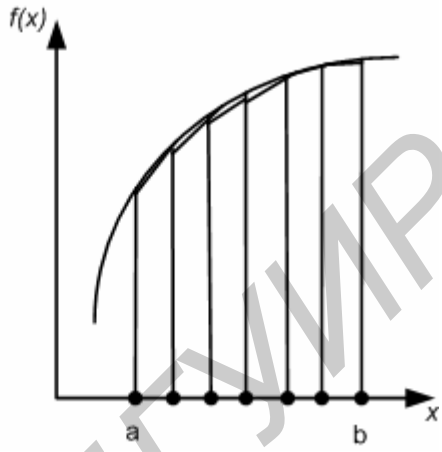

Рис. П1.4. Иллюстрация к методу трапеций

Вычисление интеграла с заданной точностью

Вычисление интеграла с заданной точностью e предполагает, что для различного количества n интервалов разбиения, например n = 5, 10, 15, 20,…, вычисляется площадь по одному из вышеописанных численных методов.

Пусть имеется ряд значений интеграла $I_0, I_1, I_2, I_3, ..., I_i, I_{i+1} \ldots$. Тогда в качестве результата расчета можно полагать I_{i+1}, если после $(i+1)$-й расчетной итерации выполняется следующее условие: $|I_{i+1} - I_i| \le e$.

В качестве начального приближения интеграла (I_0) можно взять значение интеграла, рассчитанное по следующей формуле:

$$I_0 = f(a) \cdot (b - a);$$

для метода левых прямоугольников для n = 1 (рис. П1.5.).

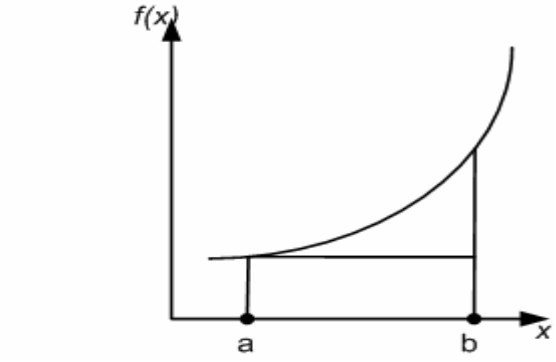

Рис. П1.5. Иллюстрация к методу левых прямоугольников для $n = 1$

Для других методов несложно получить аналогичные формулы.

ПРИЛОЖЕНИЕ 2

Варианты индивидуальных заданий

№ задания	Интегралы	
1	2	
1	$\int_{0.6}^{1.4} \frac{\sqrt{x^2+5}}{2x+\sqrt{x^2+0.5}}dx$	$\int_{0.2}^{0.8} \frac{\sin(2x+0.5)}{2+\cos(x^2+1)}dx$
	$\int_{0.8}^{1.6} \frac{dx}{\sqrt{2x^2+1}}$	$\int_{1.2}^{2} \frac{\lg(x+2)}{x}dx$
2	$\int_{0.4}^{1.2} \frac{\sqrt{0.5x+2}}{\sqrt{2x^2+1}+0.8}dx$	$\int_{0.3}^{0.9} \frac{\cos(0.8x+1.2)}{1.5+\sin(x^2+0.6)}dx$
	$\int_{1.2}^{2.7} \frac{dx}{\sqrt{x^2+3.2}}$	$\int_{1.6}^{2.4} (x+1)\sin(x)dx$
3	$\int_{0.8}^{1.8} \frac{\sqrt{0.8x^2+1}}{x+\sqrt{1.5x^2+2}}dx$	$\int_{0.4}^{1.0} \frac{\sin(x+1.4)}{0.8+\cos(2x^2+0.5)}dx$
	$\int_{1}^{2} \frac{dx}{\sqrt{2x^2+1.3}}$	$\int_{0.2}^{1} \frac{tg(x^2)}{x^2+1}dx$
4	$\int_{1.0}^{2.2} \frac{\sqrt{1.5x+0.6}}{1.6+\sqrt{0.8x^2+2}}dx$	$\int_{0.6}^{1.0} \frac{\cos(0.6x^2+0.4)}{1.4+\sin^2(x+0.7)}dx$
	$\int_{0.2}^{1.2} \frac{dx}{\sqrt{x^2+1}}$	$\int_{0.6}^{1.4} \frac{\cos(x)}{x+1}dx$
5	$\int_{1.2}^{2.0} \frac{\sqrt{2x^2+1.6}}{2x+\sqrt{0.5x^2+3}}dx$	$\int_{0.5}^{1.3} \frac{\sin(0.5x+0.4)}{1.2+\cos(x^2+0.4)}dx$

	$\int_{0.8}^{1.4} \frac{dx}{\sqrt{2x^2+3}}$	$\int_{0.4}^{1.2} \sqrt{x}\cos(x^2)dx$

Продолжение прил. 2

1	2	
6	$\int_{1.3}^{2.5} \frac{\sqrt{x^2+0.6}}{1.4+\sqrt{0.8x^2+1.3}}dx$	$\int_{0.4}^{0.8} \frac{\cos(x^2+0.6)}{0.7+\sin(0.8x+1)}dx$
	$\int_{0.4}^{1.2} \frac{dx}{\sqrt{2+0.5x^2}}$	$\int_{0.8}^{1.2} \frac{\sin(2x)}{x^2}dx$
7	$\int_{1.2}^{2.6} \frac{\sqrt{0.4x+1.7}}{1.5x+\sqrt{x^2+1.3}}dx$	$\int_{0.3}^{1.5} \frac{\sin(0.3x+1.2)}{1.3+\cos^2(0.5x+1)}dx$
	$\int_{1.4}^{2.1} \frac{dx}{\sqrt{3x^2-1}}$	$\int_{0.8}^{1.6} \frac{\lg(x^2+1)}{x}dx$
8	$\int_{0.8}^{1.6} \frac{\sqrt{0.3x^2+2.3}}{1.8+\sqrt{2x+1.6}}dx$	$\int_{0.5}^{1.8} \frac{\cos(x^2+0.6)}{1.2+\sin(0.7x+0.2)}dx$
	$\int_{1.2}^{2.4} \frac{dx}{\sqrt{0.5+x^2}}$	$\int_{0.4}^{1.2} \frac{\cos(x)}{x+2}dx$
9	$\int_{1.2}^{2} \frac{\sqrt{0.6x+1.7}}{2.1x+\sqrt{0.7x^2+1}}dx$	$\int_{0.4}^{1.2} \frac{\sin(1.5x+0.3)}{2.3+\cos(0.4x^2+1)}dx$
	$\int_{0.4}^{1.2} \frac{dx}{\sqrt{3+x^2}}$	$\int_{0.4}^{1.2} (2x+0.5)\sin(x)dx$
10	$\int_{0.8}^{2.4} \frac{\sqrt{0.4x^2+1.5}}{2.5+\sqrt{2x+0.8}}dx$	$\int_{0.4}^{1.2} \frac{\cos(x^2+0.8)}{1.5+\sin(0.6x+0.5)}dx$

	$\int_{0.6}^{1.5} \frac{dx}{\sqrt{1+2x^2}}$	$\int_{0.4}^{0.8} \frac{tg(x^2+0.5)}{1+2x^2}dx$

Продолжение прил. 2

1	2	
11	$\int_{1.2}^{2.8} \frac{\sqrt{1.2x+0.7}}{1.4x+\sqrt{1.3x^2+0.5}}dx$	$\int_{0.5}^{1.3} \frac{\sin(0.7x+0.4)}{2.2+\cos(0.3x^2+0.7)}dx$
	$\int_{2}^{3.5} \frac{dx}{\sqrt{x^2-1}}$	$\int_{0.18}^{0.98} \frac{\sin(x)}{x+1}dx$
12	$\int_{0.6}^{2.4} \frac{\sqrt{1.1x^2+0.9}}{1.6+\sqrt{0.8x^2+1.4}}dx$	$\int_{0.4}^{1.2} \frac{\cos(0.8x^2+1)}{1.4+\sin(0.3x+0.5)}dx$
	$\int_{0.5}^{1.3} \frac{dx}{\sqrt{x^2+2}}$	$\int_{0.2}^{1.8} \sqrt{x+1}\cos(x^2)dx$
13	$\int_{0.7}^{2.1} \frac{\sqrt{0.6x+1.5}}{2x+\sqrt{x^2+3}}dx$	$\int_{0.2}^{1} \frac{\sin(0.8x^2+0.3)}{0.7+\cos(1.2x+0.3)}dx$
	$\int_{1.2}^{2.6} \frac{dx}{\sqrt{x^2+0.6}}$	$\int_{1.4}^{3} x^2\lg(x)dx$
14	$\int_{0.8}^{2.4} \frac{\sqrt{1.5x^2+2.3}}{3+\sqrt{0.3x+1}}dx$	$\int_{0.3}^{1.1} \frac{\cos(0.3x+0.5)}{1.8+\sin(x^2+0.8)}dx$
	$\int_{1.4}^{2.2} \frac{dx}{\sqrt{3x^2+1}}$	$\int_{1.4}^{2.2} \frac{\lg(x^2+2)}{x+1}dx$
15	$\int_{1.9}^{2.6} \frac{\sqrt{2x+1.7}}{2.4x+\sqrt{1.2x^2+0.6}}dx$	$\int_{0.3}^{1.1} \frac{\sin(0.6x^2+0.3)}{2.4+\cos(x+0.5)}dx$

	$\int_{2}^{3.5} \frac{dx}{\sqrt{x^2-1}}$	$\int_{0.4}^{1.2} \frac{\cos(x^2)}{x+1}dx$

Продолжение прил. 2

1	2	
16	$\int_{0.5}^{1.9} \frac{\sqrt{0.7x^2+2.3}}{3.2+\sqrt{0.8x+1.4}}dx$	$\int_{0.4}^{1.2} \frac{\cos(0.4x+0.6)}{0.8+\sin^2(x+0.5)}dx$
	$\int_{1.6}^{2.2} \frac{dx}{\sqrt{x^2+2.5}}$	$\int_{0.8}^{1.6} (x^2+1)\sin(x-0.5)dx$
17	$\int_{1}^{2.6} \frac{\sqrt{0.4x+3}}{0.7x+\sqrt{2x^2+0.5}}dx$	$\int_{0.4}^{1.8} \frac{\sin(0.2x^2+0.7)}{1.4+\cos(0.5x+0.2)}dx$
	$\int_{0.8}^{1.8} \frac{dx}{\sqrt{x^2+4}}$	$\int_{0.6}^{1.4} x^2\cos(x)dx$
18	$\int_{0.7}^{2.1} \frac{\sqrt{1.7x^2+0.5}}{1.4+\sqrt{1.2x+1.3}}dx$	$\int_{0.2}^{1} \frac{\cos(0.3x+0.8)}{0.9+2\sin(0.4x+0.3)}dx$
	$\int_{1.2}^{2} \frac{dx}{\sqrt{x^2+1.2}}$	$\int_{1.2}^{2} \frac{\lg(x^2+3)}{2x}dx$
19	$\int_{0.6}^{2.2} \frac{\sqrt{1.5x+1}}{1.2x+\sqrt{x^2+1.8}}dx$	$\int_{0.3}^{1.1} \frac{\sin(0.8x+0.3)}{1.2+\cos(x^2+0.4)}dx$
	$\int_{1.4}^{2} \frac{dx}{\sqrt{2x^2+0.7}}$	$\int_{2.5}^{3.3} \frac{\lg(x^2+0.8)}{x-1}dx$
20	$\int_{1.2}^{3} \frac{\sqrt{2x^2+0.7}}{1.5+\sqrt{0.8x+1}}dx$	$\int_{0.5}^{1.3} \frac{\cos(x^2+0.2)}{1.3+\sin(2x+0.4)}dx$

	$\int_{3.2}^{4} \frac{dx}{\sqrt{0.5x^2+1}}$	$\int_{0.5}^{1.2} \frac{tg(x^2)}{x+1}dx$

Продолжение прил. 2

1	2	
21	$\int_{1.3}^{2.7} \frac{\sqrt{1.3x^2+0.8}}{1.7x+\sqrt{2x+0.5}}dx$	$\int_{0.4}^{1.2} \frac{\sin(0.6x+0.5)}{1.5+\cos(x^2+0.4)}dx$
	$\int_{0.8}^{1.7} \frac{dx}{\sqrt{2x^2+0.3}}$	$\int_{1.3}^{2.1} \frac{\sin(x^2-1)}{2\sqrt{x}}dx$
22	$\int_{0.6}^{1.4} \frac{\sqrt{x^2+0.5}}{2x+\sqrt{x^2+2.5}}dx$	$\int_{0.2}^{0.8} \frac{\cos(x^2+1)}{2+\sin(2x+0.5)}dx$
	$\int_{1.2}^{2.0} \frac{dx}{\sqrt{0.5x^2+1.5}}$	$\int_{0.2}^{1.0} (x+1)\cos(x^2)dx$
23	$\int_{0.4}^{1.2} \frac{\sqrt{2x^2+1}}{0.8x+\sqrt{0.5x+2}}dx$	$\int_{0.3}^{0.9} \frac{\sin(x^2+0.6)}{1.5+\cos(0.8x+1.2)}dx$
	$\int_{2.1}^{3.6} \frac{dx}{\sqrt{x^2-3}}$	$\int_{0.8}^{1.2} \frac{\sin(x^2-0.4)}{x+2}dx$
24	$\int_{0.8}^{1.8} \frac{\sqrt{1.5x^2+2}}{x+\sqrt{0.8x^2+1}}dx$	$\int_{0.4}^{1.0} \frac{\cos(2x^2+0.5)}{0.8+\sin(x+1.4)}dx$
	$\int_{1.3}^{2.5} \frac{dx}{\sqrt{0.2x^2+1}}$	$\int_{0.15}^{0.63} \sqrt{(x+1)}\lg(x+3)dx$

25	$\int_{1}^{2.2} \frac{\sqrt{0.8x^2+2}}{1.6+\sqrt{1.5x+0.6}}dx$	$\int_{0.6}^{1.0} \frac{\sin(x+0.7)}{1.4+\cos(0.6x+0.4)}dx$
	$\int_{0.6}^{1.4} \frac{dx}{\sqrt{12x^2+0.5}}$	$\int_{1.2}^{2.8} \frac{\lg(1+x^2)}{2x-1}dx$

Окончание прил. 2

1	2	
26	$\int_{1.2}^{2.0} \frac{\sqrt{0.5x^2+3}}{2x+\sqrt{2x^2+1.6}}dx$	$\int_{0.5}^{1.3} \frac{\cos(x^2+0.4)}{1.2+\sin(0.5x+0.4)}dx$
	$\int_{1.3}^{2.1} \frac{dx}{\sqrt{3x^2-0.4}}$	$\int_{0.6}^{0.72} (\sqrt{x}+1)tg(2x)dx$
27	$\int_{1.3}^{2.5} \frac{\sqrt{0.8x^2+1.3}}{1.4+\sqrt{x^2+0.6}}dx$	$\int_{0.4}^{0.8} \frac{\sin(0.8x+1)}{0.7+\cos(x^2+0.6)}dx$
	$\int_{1.4}^{2.6} \frac{dx}{\sqrt{1.5x^2+0.7}}$	$\int_{0.8}^{1.2} \frac{\cos(x)}{x^2+1}dx$
28	$\int_{1.2}^{2.6} \frac{\sqrt{x^2+1.3}}{1.5x+\sqrt{0.4x+1.7}}dx$	$\int_{0.3}^{1.5} \frac{\cos(0.5x^2+1)}{1.3+\sin(0.3x+1.2)}dx$
	$\int_{0.15}^{0.5} \frac{dx}{\sqrt{2x^2+1.6}}$	$\int_{1.2}^{2.8} (\frac{x}{2}+1)\sin(\frac{x}{2})dx$
29	$\int_{0.8}^{1.6} \frac{\sqrt{2x+1.6}}{1.8+\sqrt{0.3x^2+2.3}}dx$	$\int_{0.5}^{1.1} \frac{\cos(0.7x+0.2)}{1.2+\sin(x^2+0.6)}dx$
	$\int_{2.3}^{2.5} \frac{dx}{\sqrt{x^2-4}}$	$\int_{0.8}^{1.6} \frac{\lg(x^2+1)}{x+1}dx$

30	$\int_{1.2}^{2} \frac{\sqrt{0.7x^2+1}}{2.1x+\sqrt{0.6x+1.7}}dx$	$\int_{0.4}^{1.2} \frac{\cos(0.4x^2+1)}{2.3+\sin(1.5x+0.3)}dx$
	$\int_{0.32}^{0.66} \frac{dx}{\sqrt{x^2+2.3}}$	$\int_{1.6}^{3.2} \frac{x}{2}\lg(\frac{x^2}{2})dx$

Св. план 2007, поз. 62

Учебное издание

Глухова Лилия Александровна
Фадеева Елена Павловна
Фадеева Елена Евгеньевна

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

ЛАБОРАТОРНЫЙ ПРАКТИКУМ
для студентов специальности I-40 01 01
«Программное обеспечение информационных технологий»
дневной формы обучения

В 4-х частях

Часть 3

Редактор Н. В. Гриневич
Корректор М. В. Тезина

Подписано в печать 22.05.2007. Формат 60x84 1/16. Бумага офсетная.
Гарнитура «Таймс». Печать ризографическая. Усл. печ. л. 3,14.
Уч.-изд. л. 2,0. Тираж 100 экз. Заказ 117.

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0056964 от 01.04.2004. ЛП №02330/0131666 от 30.04.2004.
220013, Минск, П. Бровки, 6