

# Тема 6.

# Подпрограммы в языке Паскаль

# Содержание

---

1. Структура, назначение и применение подпрограмм.
2. Параметры и аргументы, области действия имен.
3. Процедуры
4. Функции
5. Рекурсивные процедуры и функции
6. Примеры решения задач

# 1. Понятие подпрограммы

---

Алгоритм решения задачи проектируется путем декомпозиции всей задачи в отдельные подзадачи. Обычно подзадачи реализуются в виде подпрограмм.

Весьма поэтичное объяснение понятия *подпрограмма* дал В.Ф.Очков:  
"*Подпрограмма* - это припев песни, который поют несколько раз, а в текстах песен печатают только один раз".

В самом деле, если есть необходимость многократно совершать одни и те же действия, то вполне логично описать их единожды, а потом лишь ставить на них ссылку.

Именно такой смысл имеет использование ***подпрограмм***.

---

*Подпрограмма* - это в первую очередь программа. Со всеми полагающимися полноценной программе атрибутами:

- именем,
- разделами описания меток (*label*),
- разделами описания констант (*const*),
- разделами описания типов (*type*),
- разделами описания переменных (*var*)
- и даже со своими (вложенными) функциями и процедурами.

# Определение подпрограммы

---

Т.о., **подпрограмма** - это последовательность операторов, которые определены и записаны только в одном месте программы, однако их можно вызвать для выполнения из одной или нескольких точек программы.

Программа, содержащая подпрограммы, называется **главной** (головной).

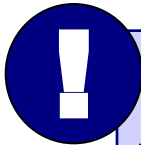
Каждая подпрограмма определяется уникальным именем.

# Виды подпрограмм

---

В языке Pascal имеется два вида *подпрограмм*:

- *процедуры*
- *и функции.*



Процедура может возвращать в качестве ответа несколько значений, а функция – только одно.

Описывая их общие черты, мы будем употреблять обобщенный термин "*подпрограмма*".

Если же в тексте встретятся слова "*процедура*" или "*функция*", то это будет означать, что излагаемая информация свойственна только одному конкретному виду *подпрограмм*:

- либо только *процедурам*,
- либо только *функциям*.

---

При использовании процедур или функций программа должна содержать текст процедуры или функции и обращение к процедуре или функции.

Работа с процедурами и функциями состоит из двух частей:

- 1) **описания процедуры** или **функции** в разделе описаний процедур и функций программы;
- 2) **вызова ее на исполнение** (передача управления компьютером) с одновременной передачей исходных данных, необходимых для работы процедуры или функции.

По окончании работы процедуры или функции управление возвращается за **точку вызова** (к следующему после вызова оператору головной программы).



## **2. Объявление и описание подпрограммы**

# Объявление функции

---

**Функция** – это подпрограмма, определяющая одно – единственное скалярное или ссылочное значение, используемое при вычислении выражения.

Функции объявляются следующим образом:

```
function <имя_функции> (<список_параметров>) :  
    <тип_результата>;
```

В отличие от констант и переменных, объявление подпрограммы может быть оторвано от ее описания. В этом случае после объявления нужно указать ключевое слово ***forward***:

```
function <имя_функции> (<список_параметров>) :  
    <тип_результата>; forward;
```

# Объявление процедуры

---

Процедуры следует объявлять так:

```
procedure <имя_процедуры> (<список_параметров>);
```

Если объявление *процедуры* оторвано от ее описания, нужно поставить после него ключевое слово ***forward***:

```
procedure <имя_процедуры> (<список_параметров>);  
    forward;
```

# Описание подпрограммы

---

Описание подпрограммы должно идти после ее объявления. Оно осуществляется по следующей схеме (единой для процедур и функций):

```
uses <имена_подключаемых_модулей>;  
label <список_меток>;  
const <имя_константы> = <значение_константы>;  
type <имя_типа> = <определение_типа>;  
var <имя_переменной> : <тип_переменной>;  
procedure <имя_процедуры>  
    <описание_процедуры>  
function <имя_функции>  
    <описание_функции>;  
begin  
    {начало тела подпрограммы}  
        <операторы>  
end;    (* конец тела подпрограммы *)
```

# Описание подпрограммы

---

Если объявление подпрограммы было оторвано от ее описания, то описание начинается дополнительной строкой с указанием только имени подпрограммы:

```
function <имя_подпрограммы>;
```

ИЛИ

```
procedure <имя_подпрограммы>;
```

# Описание подпрограммы

---

Описания двух различных подпрограмм не могут пересекаться: каждый блок должен быть логически законченным.

Однако внутри любой подпрограммы (она ведь тоже является программой) могут быть описаны другие процедуры или функции - **вложенные**. На них распространяются все те же правила объявления и описания подпрограмм.

# Описание подпрограммы: пример

---

```
procedure err(c:byte; s:string);  
var zz: byte;  
begin  
    if c=0 then writeln(s)  
    else writeln('Ошибка!')  
end;
```

# Список параметров

---

В заголовке подпрограммы (в ее объявлении) указывается список **формальных параметров** переменных, которые принимают значения, передаваемые в подпрограмму извне во время ее вызова.

Для краткости мы далее будем опускать слово "формальный".

Поскольку внутри подпрограммы параметры рассматриваются как переменные с начальным значением, то имена локальных переменных, описываемые в разделе **var** (внутреннем для подпрограммы), не могут совпадать с именами параметров этой же подпрограммы.

Подробнее о локальных и глобальных переменных будет рассказано далее.



# Список параметров

---

Список параметров может и вовсе отсутствовать:

```
procedure procl;  
function func1: boolean;
```

В этом случае подпрограмма не получает никаких переменных "извне".

Однако отсутствие параметров и, как следствие, передаваемых извне значений, вовсе не означает, что при каждом вызове подпрограмма будет выполнять абсолютно одинаковые действия.

Поскольку глобальные переменные видны изнутри любой подпрограммы, их значения могут неявно изменять внутреннее состояние подпрограмм. Это очень нежелательный эффект.

# Список параметров

Если же параметры имеются, то каждый из них описывается по следующему шаблону:

```
[<способ_подстановки>]<имя_параметра>:<тип>;
```

О возможных способах подстановки значений в параметры (<пустой>, *var*, *const*) рассказано в разделе "Способы подстановки аргументов".

Если способ подстановки и тип нескольких параметров совпадают, описание этих параметров можно объединить:

```
[<способ_подстановки>]<имя1>, . . . , <имяN>: <тип>;
```

Пример описания всех трех способов подстановки:

```
function func2(a,b:byte;  
    var x,y,z:real; const c:char)real;
```

# Список параметров

---

В заголовке подпрограммы можно указывать только простые (не составные) типы данных. Следовательно, попытка записать

```
procedure proc2(a: array[1..100] of char);
```

вызовет ошибку уже на этапе компиляции.

Для того чтобы обойти это ограничение, составной тип данных нужно описать в разделе **type**, а при объявлении подпрограммы воспользоваться именем этого типа:

```
type arr = array[1..100] of char;  
procedure proc2(a: arr);  
function func2(var x: string): arr;
```

# Возвращаемые значения

---

Основное различие между *функциями* и *процедурами* состоит в *количестве возвращаемых ими значений*.

Любая функция, завершив свою работу, должна вернуть основной программе (или другой вызвавшей ее подпрограмме) ровно одно значение, причем его тип нужно явным образом указать уже при объявлении функции.

Для возвращения результата применяется специальная "переменная", имеющая имя, совпадающее с именем самой функции. Оператор присваивания значения этой "переменной" обязательно должен встречаться в теле функции хотя бы один раз.

# Возвращаемые значения

---

Например:

```
function min(a, b: integer): integer;  
begin  
  if a>b then min:= b  
  else min:= a  
end;
```

# Возвращаемые значения

---

В отличие от функций, процедуры вообще не возвращают (явным образом) никаких значений.

О том, как все-таки получить результаты работы процедуры, вы узнаете из пункта "Параметр-переменная".

# Вызов подпрограмм

---

Любая подпрограмма может быть вызвана не только из основного тела программы, но и из любой другой подпрограммы, объявленной позже нее.

При вызове в подпрограмму передаются **фактические параметры** или **аргументы** (в круглых скобках после имени подпрограммы, разделенные запятыми):

**<имя\_подпрограммы> (<список\_аргументов>)**

**Аргументами** могут быть переменные, константы и выражения, включающие в себя **вызовы функций**.

# Вызов подпрограмм

---



*Количество и типы передаваемых в подпрограмму аргументов должны соответствовать количеству и типам ее параметров.*

Кроме того, тип каждого аргумента должен обязательно учитывать *способ подстановки, указанный для соответствующего параметра.*

Если у подпрограммы вообще нет объявленных параметров, то при вызове список передаваемых аргументов будет отсутствовать вместе с обрамляющими его скобками.



# Вызов подпрограмм



*Вызов функции не может быть самостоятельным оператором, потому что возвращаемое значение нужно куда-то записывать.*

Вызов функции может стать равноправным участником арифметического выражения.

Например:

```
c := min(a, a*2);  
if min(z, min(x, y)) = 0 then ...;
```


# Вызов подпрограмм

---

Процедура же ничего не возвращает явным образом, поэтому ее вызов является отдельным оператором в программе.

Например:

```
err (res, 'Привет!');
```

 После того как вызванная подпрограмма завершит свою работу, *управление передается оператору, следующему за оператором, вызвавшим эту подпрограмму.*

# **3. Способы подстановки аргументов**

---

Как уже упоминалось выше, при вызове подпрограммы подстановка значений аргументов в параметры производится в соответствии с правилами, указанными в атрибуте **<способ\_подстановки>**.

Рассмотрим три различных значения этого атрибута:

- **<пустой>**;
- **var**;
- **const**.

# Параметр-значение: описание

---

В списке параметров подпрограммы перед **параметром-значением** служебное слово отсутствует.

Например, функция **func3** имеет три параметра-значения:

```
function func3(x:real; k:integer;  
              flag:boolean) :real;
```

При вызове подпрограммы параметру-значению может соответствовать аргумент, являющийся

- выражением,
- переменной
- или константой.

Например:

```
dlna := func3(shirina/2, min(a shl 1,  
                          ord('y')), true)+0.5;
```

Для типов данных здесь не обязательно строгое совпадение (эквивалентность), достаточно и совместимости по присваиванию (см. Тему 2).

## Параметр-значение: механизм передачи значения

---

В области памяти, выделяемой для работы вызываемой подпрограммы, создается переменная с именем **<имя\_подпрограммы>.<имя\_параметра>**, и в эту переменную записывается значение переданного в соответствующий параметр аргумента.

Дальнейшие действия, производимые подпрограммой, выполняются именно над этой новой переменной. Значение же входного аргумента не затрагивается.

Следовательно, после окончания работы подпрограммы, когда весь ее временный контекст будет уничтожен, значение аргумента останется точно таким же, каким оно было на момент вызова подпрограммы.

# Параметр-значение: механизм передачи значения

---

В качестве примера рассмотрим последовательность действий, выполняемых при передаче аргументов

$1+a/2$ ,  $a$  и  $true$

в описанную выше функцию  $func3$ .

Пусть  $a$  - переменная, имеющая тип  $byte$ , тогда значение выражения  $1+a/2$  будет иметь тип  $real$ , а  $true$  и вовсе является константой (неименованной).

# Параметр-значение: механизм передачи значения

---

Итак, при вызове *func3(1+a/2, a, true)* будут выполнены следующие действия:

- 1) создать временные переменные *func3.x*, *func3.k*, *func3.flag*;
- 2) вычислить значение выражения  $1+a/2$  и записать его в переменную *func3.x*;
- 3) записать в переменную *func3.k* значение переменной *a*;
- 4) записать в переменную *func3.flag* значение константы *true*;
- 5) произвести действия, описанные в теле функции;
- 6) уничтожить все временные переменные, в том числе *func3.x*, *func3.k*, *func3.flag*.

Уже видно, что значения аргументов не изменятся.

```
function func3(x:real; k:integer;  
              flag:boolean):real;
```



# Параметр-переменная: описание

---

В списке параметров подпрограммы перед параметром-переменной ставится служебное слово **var**.

Например, процедура **proc3** имеет три *параметра-переменные* и один *параметр-значение*:

```
procedure proc3 (var x,y:real; var k:integer;  
                 flag:boolean);
```

При вызове подпрограммы параметру-переменной может соответствовать только аргумент-переменная; константы и выражения запрещены.

Кроме того, тип аргумента и тип параметра-переменной должны быть эквивалентными (см. Тему 2).

# Параметр-переменная: механизм передачи значения

---

В отличие от параметра-значения, для параметра-переменной не создается копии при вызове подпрограммы. Вместо этого в работе подпрограммы участвует та самая переменная, которая послужила аргументом.

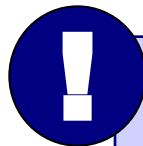
Понятно, что если ее значение изменится в процессе работы подпрограммы, то это изменение сохранится и после того, как будет уничтожен контекст подпрограммы.

# Параметр-переменная: механизм передачи значения

---

Понятно опять же и *ограничение на аргументы*, которые должны соответствовать *параметрам-переменным*:

- ни константа, ни выражение не смогут сохранить изменения, внесенные в процессе работы подпрограммы.



Т.о., параметры-переменные служат теми посредниками, которые позволяют получать результаты работы процедур, а также увеличивать количество результатов, возвращаемых функциями.

**Замечание:** Для экономии памяти в параметр-переменную можно передавать и такую переменную, изменять значение которой не требуется. Если нужно передать в качестве аргумента массив, то лучше не создавать его копию, как это будет сделано при использовании параметра-значения, а использовать параметр-переменную.

# Параметр-константа: описание

---

В списке параметров подпрограммы перед *параметром-константой* ставится служебное слово **const**.

Например, процедура **proc4** имеет один *параметр-переменную* и один *параметр-константу*:

```
procedure proc4 (var k:integer;  
                 const flag:boolean);
```

При вызове подпрограммы параметру-константе может соответствовать аргумент, являющийся

- выражением,
- переменной
- или константой.

# Параметр-константа: описание

---

Во время выполнения подпрограммы соответствующая переменная считается обычной константой.

Ограничением является то, что при вызове другой подпрограммы из тела текущей, параметр-константа не может быть подставлен в качестве аргумента в параметр-переменную.

Для типов данных здесь не обязательно строгое совпадение (эквивалентность), достаточно и совместимости по присваиванию (см. Тему 2).

# Параметр-константа: механизм передачи значения

---

В отличие от параметра-переменной, для параметра-константы создается копия при вызове подпрограммы.

## 4. Области действия имен

# Разграничение контекстов

---

**Глобальные объекты** - это типы данных, константы и переменные, объявленные в начале программы до объявления любых подпрограмм.

Эти объекты будут видны во всей программе, в том числе и во всех ее подпрограммах.

Глобальные объекты существуют на протяжении всего времени работы программы.

**Локальные объекты** объявляются внутри какой-нибудь подпрограммы и "видны" только этой подпрограмме и тем подпрограммам, которые были объявлены как внутренние для нее.

Локальные объекты не существуют, пока не вызвана подпрограмма, в которой они объявлены, а также после завершения ее работы.



```
program prog;
  var a:byte;
  procedure pr1 (p:byte);
    var b:byte;           {первый уровень вложенности}
    function f (pp:byte);
      var c:byte;       {второй уровень вложенности}
      begin
        {здесь "видны" переменные a, b, c, p, pp}
      end; {для f }
    begin
      {здесь "видны" переменные a, b, p}
    end; {для pr1 }
    var g:byte;
    procedure pr2;
      var d:byte;           {первый уровень вложенности}
    begin
      {здесь видны переменные a, d, g}
    end; {для pr2 }
  begin
    {тело программы; здесь "видны" переменные a, g}
  end. {для prog }
```

# Побочный эффект

---

Поскольку глобальные переменные видны в контекстах всех блоков, то их значение может быть изменено изнутри любой подпрограммы.

Этот эффект называется **побочным**, а его использование очень нежелательно, потому что может стать источником непонятных ошибок в программе.

Чтобы избежать побочного эффекта, необходимо строго следить за тем, чтобы подпрограммы изменяли только свои локальные переменные (в том числе и параметры-переменные).

# Совпадение имен

---

Вообще говоря, совпадения глобальных и локальных имен допустимы, поскольку к каждому локальному имени неявно приписано имя той подпрограммы, в которой оно объявлено.

Таким образом, в приведенном выше примере (см. слайд 41) фигурируют переменные

*a*, *g*, *pr1.p*, *pr1.b*, *pr1.f.pp*, *pr1.f.c*, *pr2.d*.

# Совпадение имен

---

Если имеются глобальная и локальная переменные с одинаковым именем, то изнутри подпрограммы к глобальной переменной можно обратиться, приписав к ней спереди имя программы:

```
<имя_программы>.<имя_глобальной_переменной>
```

Например, (локальной переменной здесь присваивается значение глобальной):

```
a := prog.a;
```

# 5. Примеры решения задач

# Пример 1

---

Вычислить  $s = 1 + x/1! + x^2/2! + x^3/3! + \dots + x^n/n!$  .

**Решение.** Вычисление степени, факториала и суммы оформим в виде функций.

```
program task1;  
var x : real; {аргумент}  
    n : integer; {количество слагаемых в сумме}  
    y : real; {сумма}  
{функция вычисления числителя}  
function a(x:real; i:integer) :real;  
var j : integer; {счетчик умножений}  
begin  
    a:=1;  
    for j:=1 to i do a:= a*x  
end;
```

# Пример 1

{функция вычисления знаменателя}

```
function b(i:integer) :real;  
  var j : integer; {счетчик умножений}  
  begin  
    b:=1;  
    for j:=1 to i do b:= b*j  
end;
```

{функция вычисления суммы}

```
function y(x:real; n:integer) :real;  
  var i : integer; {номер очередного слагаемого}  
  begin  
    y:=0;  
    for i:=1 to n do y:= y + a(x, i)/b(i)  
end;
```

# Пример 1

---

{основная программа}

*begin*

*write ('введите аргумент - x, и количество  
слагаемых - n');*

*readln(x, n);*

*write ('Сумма = ', y(x, n))*

*end.*



## Пример 2

---

Найти наименьший член последовательности  $a(n)=n^2-7n+1$ , где  $n$  изменяется от 1 до  $m$ .

**Решение.** Для поиска минимального элемента используем функцию, находящую минимальный элемент из двух параметров.

```
program task2;  
  var m: integer; {кол-во элементов посл-сти}  
    n: integer; {номер текущего элемента посл-сти}  
    k: real;      {минимальный элемент посл-сти}  
function min(p1, p2 :real) :real;  
  begin  
    if p1 < p2 then min:=p1  
    else min:=p2  
end;
```

## Пример 2

```
begin  
  write('Введите m - количество элементов  
    последовательности');  
  readln(m);  
  k := 1*1-7*1+1;  
  for n:=2 to m do  
    k := min(k, n*n-7*n+1);  
  write('Минимальный элемент  
последовательности равен ', k)  
end.
```

## Пример 3

---

Написать набор процедур для работы с обыкновенными дробями, обеспечив их сложение, вычитание, умножение, деление.

**Решение.** Обыкновенную дробь будем изображать двумя целыми числами:

- первое число будет представлять числитель дроби,
- а второе - знаменатель.

В процессе вычислений требуется сокращать дроби на их наибольший общий делитель (НОД), для вычисления которого используется алгоритм Евклида. Если одно из чисел равно нулю, то НОД берем равным 1.

Разработаем также отдельные процедуры для ввода и вывода обыкновенных дробей

## Пример 3

```
Var  x, y, {числитель и знаменатель дроби }
     p, q, {числитель и знаменатель дроби }
     s, t:integer; {числитель и знаменатель дроби }
     { Ввод обыкновенной дроби }
procedure wwod(var a, b:integer);
begin
  writeln;
  write('Введите целые: числитель и
        знаменатель обыкновенной дроби ');
  readln(a,b)
end;
     { Вывод результата }
procedure wywod(a, b:integer);
begin
  write(a, '/', b);  writeln
end;
```

## Пример 3

```
{ Вычисление НОД(x,y) }  
function nod(x, y:integer):integer;  
begin  
  if (x=0) or (y=0) then nod:=1  
  else begin  
    while x<>y do begin  
      while x>y do x:=x-y;  
      while y>x do y:=y-x  
    end;  
    nod:=x  
  end  
end;
```

## Пример 3

{ Сокращение дроби }

```
procedure sokr(var c, d:integer);  
var r:integer;  
begin  
  r:=nod(c, d);  
  c:=c div r; d:=d div r  
end;
```

{ Сложение двух дробей }

```
procedure sum(a, b, c, d:integer;  
  var e, f:integer);  
var r:integer;  
begin  
  e:=a*d+b*c; f:=b*d;  
  sokr(e, f)  
end;
```

# Пример 3

{ Вычитание двух дробей }

```
procedure raz(a, b, c, d:integer;  
             var e, f:integer);  
var r:integer;  
begin  
  e:=a*d-b*c;      f:=b*d;  
  sokr(e,f)  
end;
```

{ Умножение двух дробей }

```
procedure mult(a, b, c, d:integer;  
              var e, f:integer);  
var r:integer;  
begin  
  e:=a*c;          f:=b*d;  
  sokr(e,f)  
end;
```

## Пример 3

{ Деление двух дробей }

```
procedure del(a, b, c, d:integer;  
    var e, f:integer);  
var r:integer;  
begin  
    e:=a*d;    f:=b*c;  
    sokr(e, f)  
end;
```

```
begin  
    write('Введите первую дробь ');  
    wwod(x,y);  
    write('Введите вторую дробь ');  
    wwod(p,q);
```



## Пример 3

```
write('Сумма равна ');  
sum(x, y, p, q, s, t);  
wywod(s, t);  
  
write('Разность равна ');  
raz(x, y, p, q, s, t);  
wywod(s, t);  
  
write('Произведение равно ');  
mult(x, y, p, q, s, t);  
wywod(s, t);  
  
write('Частное равно ');  
del(x, y, p, q, s, t);  
wywod(s, t);  
end.
```

# **6. Рекурсивные процедуры и функции**

---

В общем случае **рекурсией** называется ситуация, когда какой-то алгоритм вызывает себя прямо или через другие алгоритмы в качестве вспомогательного. Сам алгоритм при этом называется **рекурсивным**.

Понятно, что без конца такие вызовы продолжаться не могут, так как в противном случае получится бесконечный цикл.

Поэтому при построении рекурсивного алгоритма предусматриваются случаи, когда результат вычисляется явно (непосредственно) без самовывоза.

---

Таким образом, любая рекурсия обязательно должна содержать два условия:

- 1) Вычисление результата через другие значения (для простейших случаев). Выполнение этого условия не должно повлечь за собой нового рекурсивного вызова.
- 2) Вычисление значения с помощью самовывоза функции (рекурсивный вызов).

# Механизм работы рекурсии

---

Чтобы понять, как будет выполняться эта программа, вспомним, что на время выполнения вспомогательного алгоритма основной алгоритм приостанавливается.

При вызове новой копии рекурсивного алгоритма вновь выделяется место для всех переменных, объявляемых в нем, причем переменные других копий будут недоступны.

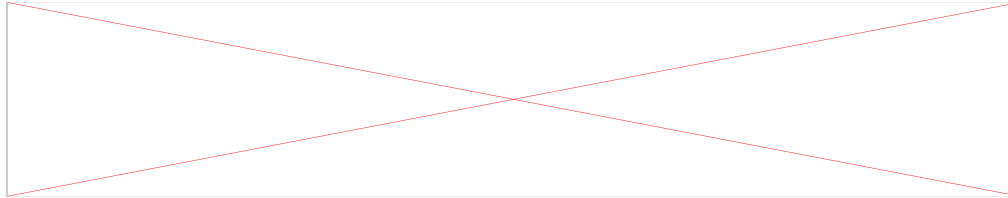
При удалении копии рекурсивного алгоритма из памяти удаляются и все его переменные.

Активизируется предыдущая копия рекурсивного алгоритма, становятся доступными ее переменные.

## Пример 4

---

Широко известно рекурсивное определение факториала:



**Решение.** В первой строке определения явно указано, как вычислить факториал, если аргумент равен нулю или единице.

В любом другом случае для вычисления  $n!$  необходимо вычислить предыдущее значение  $(n-1)!$  и умножить его на  $n$ .

Уменьшающееся значение гарантирует, что в конце концов возникнет необходимость найти  $1!$  или  $0!$ , которые вычисляются непосредственно.

## Пример 4

```
program task4;  
  var n : integer; { исходное значение }  
  function fact(i:integer):integer;  
  begin  
    if (i=1) or (i=0) then fact:=1  
    else fact:=fact(i-1)*i  
  end;  
begin  
  write('Введите нужное значение n ');  
  readln(n);  
  writeln('Факториал ', n, ' равен ', fact(n))  
end.
```

## Пример 4

---

Пусть необходимо вычислить  $4!$  Основной алгоритм: вводится  $n=4$ , вызов  $\text{fact}(4)$ .

Основной алгоритм приостанавливается, вызывается и работает  $\text{fact}(4)$ :  $4 \neq 1$  и  $4 \neq 0$ , поэтому  $\text{fact} := \text{fact}(3) * 4$ .

Работа функции приостанавливается, вызывается и работает  $\text{fact}(3)$ :  $3 \neq 1$  и  $3 \neq 0$ , поэтому  $\text{fact} := \text{fact}(2) * 3$ . Заметьте, что в данный момент в памяти компьютера две копии функции  $\text{fact}$ .

Вызывается и работает  $\text{fact}(2)$ :  $2 \neq 1$  и  $2 \neq 0$ , поэтому  $\text{fact} := \text{fact}(1) * 2$ . В памяти компьютера уже три копии функции  $\text{fact}$  и вызывается четвертая.

Вызывается и работает  $\text{fact}(1)$ :  $1 = 1$ , поэтому  $\text{fact}(1) = 1$ . Работа этой функции завершена, продолжает работу  $\text{fact}(2)$ .

$\text{fact}(2) := \text{fact}(1) * 2 = 1 * 2 = 2$ . Работа этой функции также завершена, и продолжает работу функция  $\text{fact}(3)$ .

$\text{fact}(3) := \text{fact}(2) * 3 = 2 * 3 = 6$ . Завершается работа и этой функции, и продолжает работу функция  $\text{fact}(4)$ .

$\text{fact}(4) := \text{fact}(3) * 4 = 6 * 4 = 24$ .

Сейчас управление передается в основную программу и печатается ответ: «Факториал 4 равен 24».



## Пример 5

---

Написать рекурсивную функцию, вычисляющую указанное число Фибоначчи.

**Решение.** Последовательность Фибоначчи задается следующими соотношениями:

$$a(0)=a(1)=1, a(i)=a(i-1)+a(i-2), \text{ где } i>1,$$

которые легко записать на Паскале в виде рекурсивной функции.

## Пример 5

---

```
function fib(n:integer):integer;  
begin  
  if n=0 then fib:=1  
  else if n=1 then fib:=1  
    else fib:= fib(n-1)+fib(n-2)  
end;
```

## Пример 6

---

Написать рекурсивную процедуру, переводящую целое число из десятичной системы счисления в восьмеричную.

```
procedure convert(z:integer);  
begin  
  if z>1 then convert(z div 8);  
  write(z mod 8:1)  
end;
```

# Пример 7

---

Написать рекурсивную функцию для поиска максимального элемента в одномерном массиве.

```
type mas= array [1..n] of real;  
...  
function m(a:mas; i:integer):real;  
begin  
    if i=1 then m:=a[1]  
    else if a[i]>m(a, i-1) then m:=a[i]  
        else m:=m(a, i-1)  
end;
```