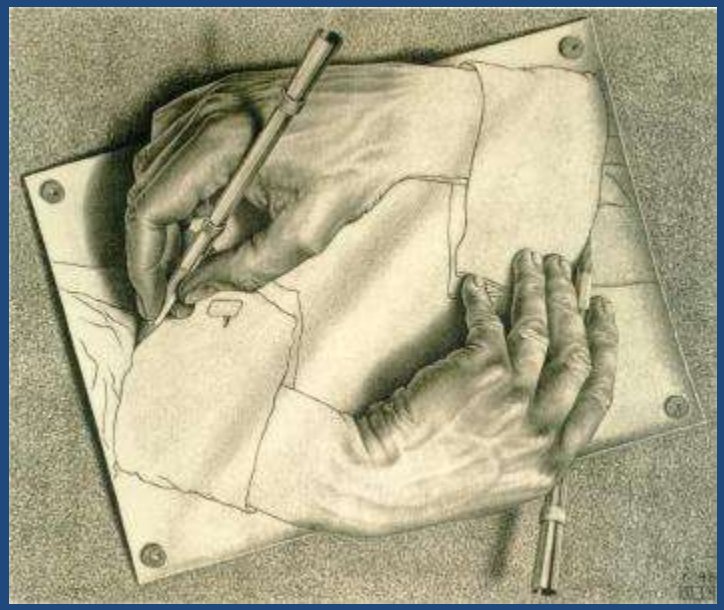


РЕКУРСИЯ

Чтобы понять рекурсию, нужно сначала понять рекурсию.

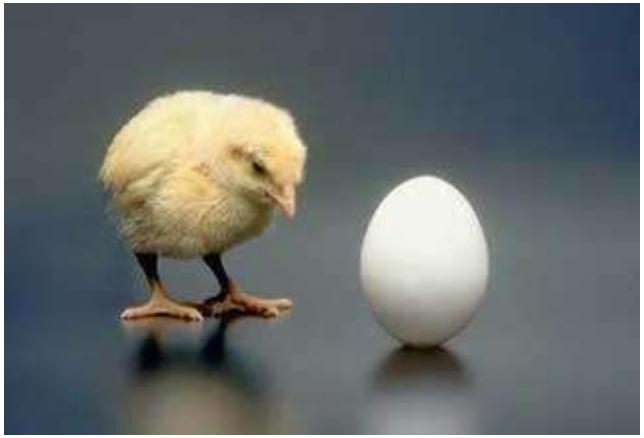
Из программистского фольклора

Содержание

- Понятие рекурсии
- Примеры рекурсивных алгоритмов
- Реализация механизма рекурсии
- Сравнение рекурсии и итерации
- Рекурсивные алгоритмы с заглядыванием вперед
- Рекурсивные алгоритмы с возвратом

Рекурсия

- Рекурсия – определение некоторого понятия через самое себя.



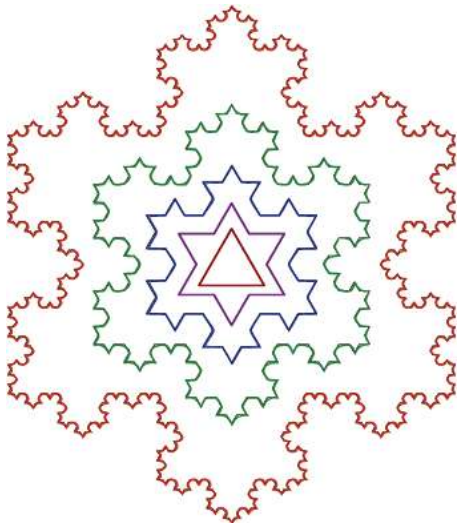
Рекурсия вокруг нас: литература

- У попа была собака, он ее любил.
  Она съела кусок мяса, он ее убил.
  В землю закопал и на камне написал:
  "У попа была собака …
  "У попа была собака …
  "У попа была собака …
- С. Лем
  - СЕПУЛЬКИ — важный элемент цивилизации ардритов (*см.*) с планеты Энтеропия (*см.*). См. СЕПУЛЬКАРИИ.
  - СЕПУЛЬКАРИИ — устройства для сепуления (*см.*).
  - СЕПУЛЕНИЕ — занятие ардритов (*см.*) с планеты Энтеропия (*см.*). См. СЕПУЛЬКИ.
  - СЕПУЛЬКИ очень похожи на МУРКВИ, а своей цветовой гаммой напоминают мягкие ПЧМЫ.

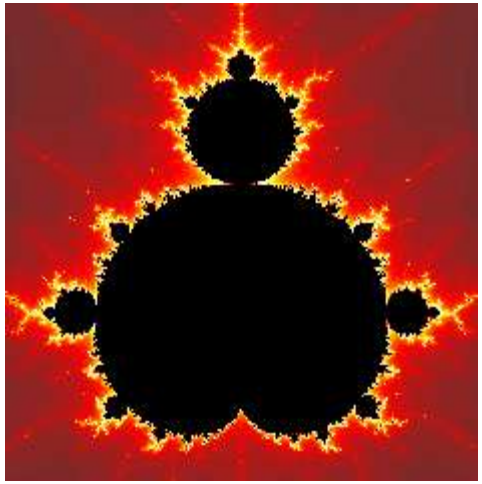
Рекурсия вокруг нас: математика

- Принцип математической индукции
- Фракталы (самоподобные множества нецелой размерности)

Треугольники Серпинского



Кривые Коха



Множество Мандельброта

Рекурсия вокруг нас: физика

- Взаимное отражение двух расположенных друг напротив друга зеркал (бесконечная рекурсия)
- Эффект положительной обратной связи у электронных схем усиления: сигнал с выхода попадает на вход, усиливается, снова попадает на вход схемы и снова усиливается (бесконечная рекурсия)

Рекурсия вокруг нас: головоломки

- В одном из буддийских монастырей монахи уже тысячу лет занимаются перекладыванием колец.
  Они располагают тремя пирамидами, на которых надеты кольца разных размеров.
- В начальном состоянии 64 кольца были надеты на первую пирамиду и упорядочены по размеру. Монахи должны переложить все кольца с первой пирамиды на вторую, выполняя единственное условие - кольцо нельзя положить на кольцо меньшего размера. При перекладывании можно использовать все три пирамиды.
- Монахи перекладывают одно кольцо за одну секунду. Как только они закончат свою работу, наступит конец света.

Рекурсия вокруг нас: головоломки

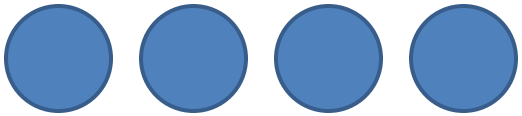
- Рекурсивный алгоритм решения задачи "Ханойские башни"
  передвинуть нужное количество *дисков* из пирамиды *источник* на пирамиду *задание*, используя *запасную* пирамиду
  - Если количество *дисков* равно одному, тогда **передвинуть** диск из *источника* в *задание*
  - В противном случае:
    - рекурсивно **передвинуть** все *диски*, кроме последнего, из *источника* в *запас*, используя *задание* как *запас*
    - **передвинуть** оставшийся *диск* из *источника* в *задание*
    - **передвинуть** все *диски* из *запаса* в *задание*, используя *источник* как *запас*

Рекурсивные алгоритмы

- *Рекурсивный алгоритм* – алгоритм, определяемый через себя.
  - *Прямая рекурсия*: один из шагов алгоритма *A* формулируется как обращение к *A*.
  - *Косвенная (взаимная) рекурсия*: один из шагов алгоритма *A* формулируется как обращение к алгоритму *B*, в котором один из шагов формулируется как обращение к алгоритму *A*.

Рекурсивные данные

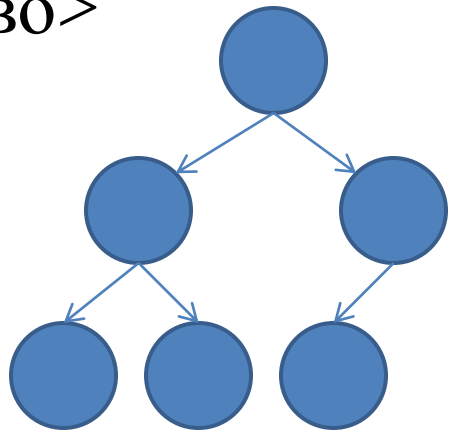
- <список>::=<элемент>|<элемент><список>
  <элемент>::=  | <пусто>



- <дерево>::= <корень> | <пусто>

  <левое поддерево> <правое поддерево>

  <левое поддерево>::=<дерево>
  <правое поддерево>::=<дерево>
  <корень>::= 



Пример: факториал

$$f(n) = \begin{cases} 1, & n \le 1 \\ n * f(n-1), & n > 1 \end{cases}$$

```
function Factorial(n: Integer): Integer;
begin
        if (n=0) or (n=1) then
                Result:=1
        else
                Result:=n*Factorial(n-1);
end;
```

Пример: числа Фибоначчи

$$f(n) = \begin{cases} 1, & n \le 2 \\ f(n-1) + f(n-2), & n > 2 \end{cases}$$

```
function Fib(n: Integer): Integer;
begin
        if (n=1) or (n=2) then
                Result:=1
        else
                Result:=Fib(n-
1)+Fib(n-2);
end;
```

Пример: НОД

$$GCD(n,m) = \begin{cases} n, & m = 0 \\ GCD(m, n \bmod m), & m \neq 0 \end{cases}$$

```
function GCD(n, m: Integer): Integer;
begin
        if m=0 then
                Result:=n
        else
                Result:=GCD(m, n mod m);
end;
```

Пример: НОД


```
function GCD(n, m: Integer): Integer;
begin
  while (n<>0) and (m<>0) do
    if n>=m then
      n:=n mod m
    else
      m:=m mod n;
  Result:=n +m;
end;
```

Пример: биномиальные коэф-ты

- Биномиальный коэффициент *C(n,m)* – число сочетаний из *n* по *m*, т.е. число способов сформировать *m*-элементное множество из множества, в котором *n* элементов.
- Если *m*=*n* или *m*=0, то *C(n,m)*=1.
- Разобьем сочетания *C(n,m)* на два подмножества по вхождению в них *n*-й элемент.
  - Тогда в подмножестве, в которое не входит *n*-й элемент, будет *C(n*-1,*m)* сочетаний.
  - В подмножестве, в которое входит *n*-й элемент, будет *C(n*-1,*m*-1*)* сочетаний: *n*-й элемент фиксирован, и можно делать выборки из *n*-1 по *m*-1 элементов.
- Таким образом, ***C(n,m)=C(n*-1,*m)+C(n*-1,*m*-1*)***

Структура рекурсивного алгоритма


```
procedure Recursion(n: TParam; var Res: TOutput);
begin
        if n=SimplestValue then
                Res:=SimplestRes
        else begin
                Simplify(n);
                Recursion(n, Res);
                Transform(Res);
        end;
end;
```

База рекурсии (условие окончания рекурсивных вызовов)

Тело рекурсии (рекурсивные вызовы подпрограммы)

Структура рекурсивного алгоритма


```
function Recursion(n: TParam): TOutput;
begin
        if n=SimplestValue then
                Result:=SimplestRes
        else
                Result:=Transform(Recursion(Simplify(n)));
end;
```

База рекурсии (условие окончания рекурсивных вызовов)

Тело рекурсии (рекурсивные вызовы подпрограммы)

Пример: умножение

$$a*b = \underbrace{a+a+a+\ldots+a}_{b\_раз} = a + \underbrace{a+a+\ldots+a}_{b-1\_раз} = a + a*(b-1)$$

```
function Mult(a, b: Integer): Integer;
begin
    if b=0 then
        Result:=0
    else
        if b<0 then
            Result:=-Mult(a,-b)
        else
            Result:=a+Mult(a,b-1);
end;
```

Пример: возведение в степень

$$a^b = \underbrace{a*a*a*...*a}_{b\_раз} = a*\underbrace{a*a*...*a}_{b-1\_раз} = a*a^{b-1}$$

```
function Power(a,b: Integer): Real;
begin
        if b=0 then
                Result:=1
        else
                if b<0 then
                        Result:=1/Power(a,-b)
                else
                        Result:=a*Power(a,b-1);
                        { или Result:=Mult(a,Power(a,b-1)); ☺ }
end;
```

Пример: метод половинного деления

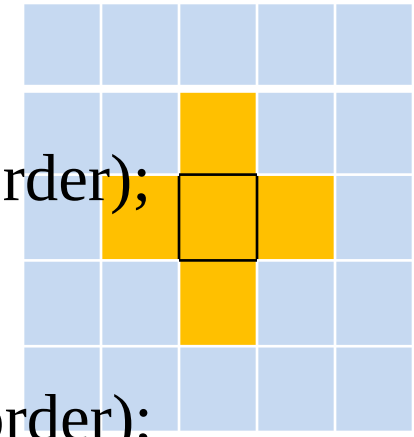
- Корень уравнения $f(x)=0$ на отрезке $[a;b]$ с точностью $Eps>0$.
- База рекурсии
  - Пусть $[u;v] \subset [a;b]$. Если $f(u)=0$, то u – корень. Если $f(v)=0$, то v – корень.
  - Если $|v-u|<Eps$, то v – корень.
- Тело рекурсии
  - Найдем середину отрезка $c=(u+v)/2$.
  - Корень находится на том из двух отрезков $[u;c]$ и $[c;v]$, где $f(x)$ имеет на границах разные знаки.

Пример: метод половинного деления


```
function FindRoot(u, v, Eps: Real; f: TFunc): Real;
var c: Real;
begin
    if (f(u)=0) or (v-u<=Eps) then
        Result:=u
    else
        if f(v)=0 then
            Result:=v
        else begin
            c:=(u+v)/2;
            if f(u)*f(c)>=0  then
                Result:=FindRoot(c, v, Eps, f)
            else
                Result:=FindRoot(u, c, Eps, f);
        end;
end;
```

Пример: закраска области


```
procedure Fill(x, y: Integer; Color, Border: TColor);
var c: TColor;
begin
        c:=GetPixelColor(x,y);
        if (c<>Border) and (c<>Color) then begin
                PutPixel(x,y,Color);
                if x-1>=MinX then Fill(x-1,y,Color,Border);
                if x+1<=MaxX then
Fill(x+1,y,Color,Border);
                if y-1>=MinY then Fill(x,y-1,Color,Border);
                if y+1<=MaxY then
Fill(x,y+1,Color,Border);
        end;
end;
```



Реализация механизма рекурсии

- При вызове подпрограммы ее локальные переменные и фактические параметры помещаются в *сегмент стека*.
- *Стек* – линейный список, в котором чтение и запись данных возможны только с одного конца (дисциплина LIFO – Last In First Out, последним включен, первым извлечен).
- *Прямой ход рекурсии*. Рекурсивный вызов подпрограммы оставляет в сегменте стека данные, помещенные туда на предыдущем вызове подпрограммы, и записывает на вершину стека данные текущего вызова.
- *Обратный ход рекурсии*. При достижении базы рекурсии рекурсивные вызовы прекращаются и начинается серия завершений вызовов с извлечением из стека сохраненных значений и использованием их для продолжения вычислений. Последним завершается первый вызов.

Реализация механизма рекурсии


```
function Factorial(n: Integer): Integer;
begin
        if (n=0) or (n=1) then Result:=1
        else
                Result:=n*Factorial(n-1);
end;
```

Factorial(5)

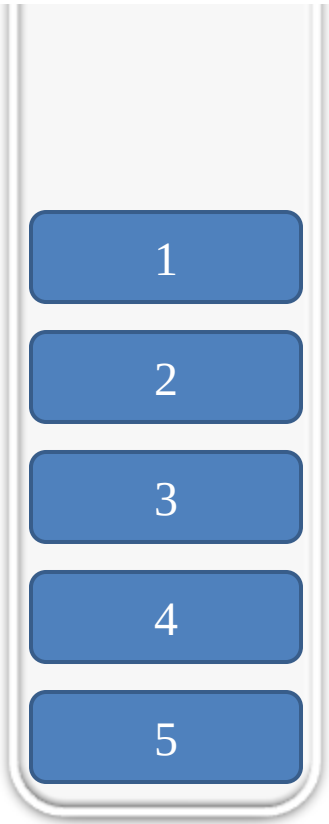
5*Factorial(4)

5*4*Factorial(3)

5*4*3*Factorial(2)

5*4*3*2*Factorial(1)

Прямой ход



Реализация механизма рекурсии


```
function Factorial(n: Integer): Integer;
begin
        if (n=0) or (n=1) then Result:=1
        else
                Result:=n*Factorial(n-1);
end;
```

Factorial(5) **5*24=120**

5*Factorial(4) **4*6=24**

5*4*Factorial(3) **3*2=6**

5*4*3*Factorial(2) **2*1=2**

5*4*3*2*Factorial(1) **1**

Обратный ход

Пример: обращение строки


```
Program ReverseString;
procedure Reverse;
var Ch: Char;
begin
    Read(Ch);
    if Ch<>" then Reverse;
    Write(Ch);
end;
begin
    Reverse;
end.
```

Рекурсия и итерация

- *Рекурсия*
  - решает задачу методом "от сложного к простому"; алгоритм – описание сложного объекта через более простой (простые) объект того же типа
  - запись алгоритма, как правило, компактна, интуитивно понятна, красива
  - исполнение алгоритма связано с большими накладными расходами и, как правило, более медленное, чем итеративное
- *Итерация*
  - решает задачу методом "от простого к сложному"; алгоритм – описание процесса построения сложного объекта из более простых объектов
  - запись алгоритма не всегда компактна, интуитивно понятна
  - исполнение итеративного алгоритма, не связано с большими накладными расходами и, как правило, более быстрое, чем рекурсивное

Пример: факториал

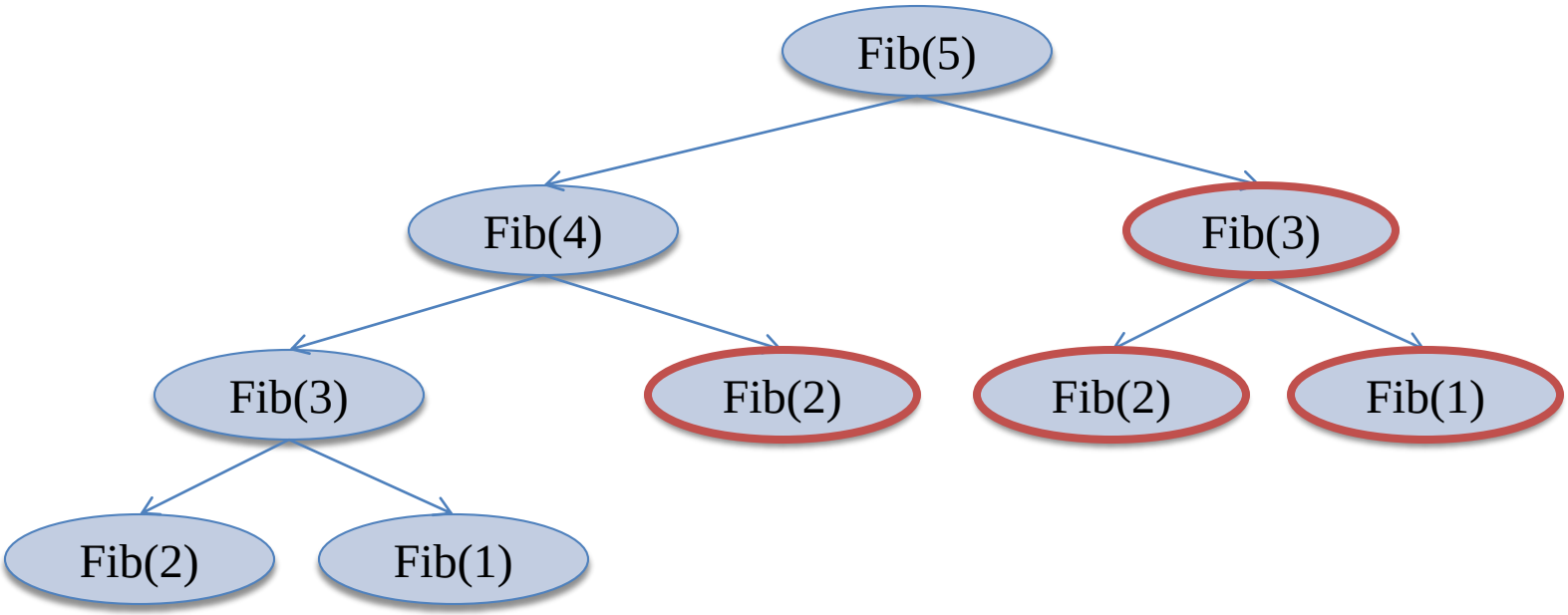

```
function Factorial(n: Integer): Integer;
begin
        if (n=0) or (n=1) then
                Result:=1
        else

        Result:=n*Factorial(n-1);
end;
```

```
function Factorial(n: Integer): Integer;
var i, Res: Integer;
begin
        Res:=1;
        for i:=2 to n do
                Res:=i*Res;
        Result:=Res;
end;
```

Пример: числа Фибоначчи

Оператор Result:=Fib(n-1)+Fib(n-2) порождает "лишние" рекурсивные вызовы

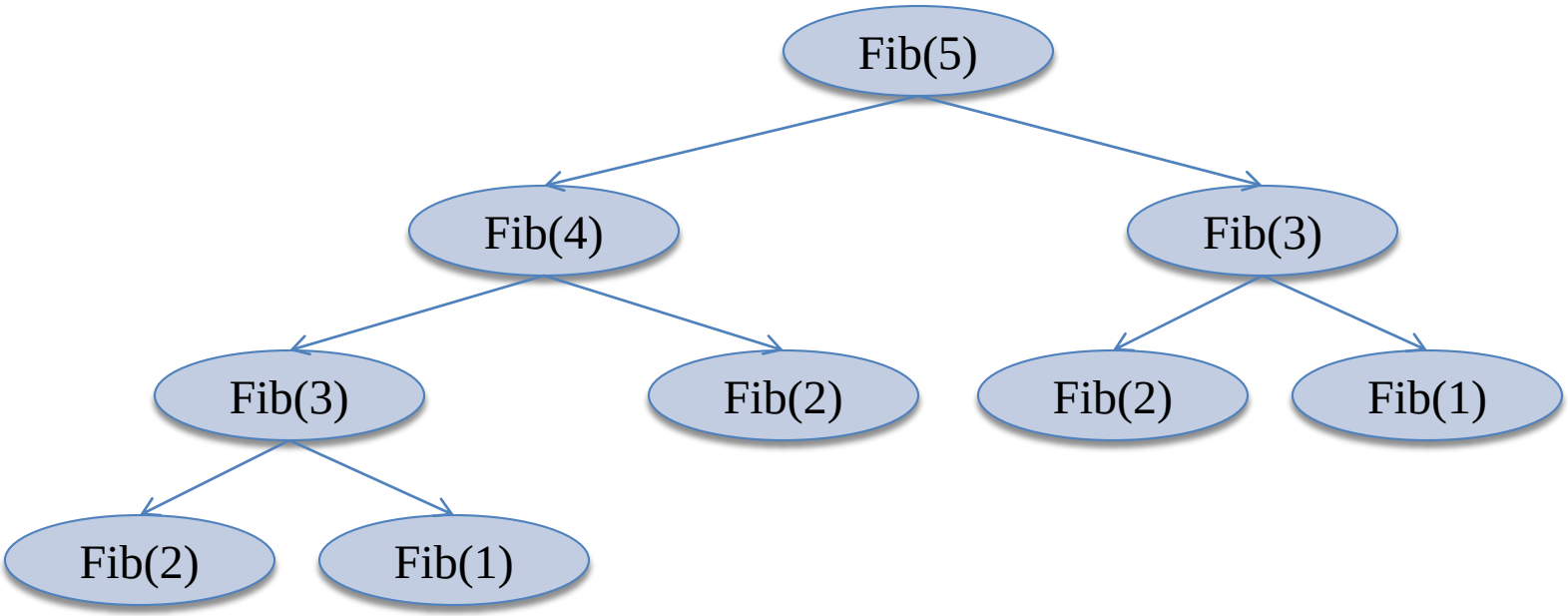


Пример: числа Фибоначчи

Операторы

```
F1:=Fib(n-1); F2:=Fib(n-2);
Result:=F1+F2;
```

также порождают "лишние" рекурсивные вызовы (но в другом порядке)



Пример: числа Фибоначчи


```
function Fib(n: Integer): Integer;
begin
    if (n=1) or (n=2) then
        Result:=1
    else
        Result:=Fib(n-1)+Fib(n-2);
end;
```

$$Fib(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

```
function Fib(n: Integer): Integer;
var i, n1, n2, Res: Integer;
begin
    N1:=1; N2:=1;
    Result:=1;
    for i:=3 to n do begin
        Res:=n1+n2;
        n2:=n1;
        n1:=Res;
    end;
    Result:=Res;
end;
```

Пример: НОД


```
function GCD(n: Integer): Integer;
begin
  if m=0 then
    Result:=n
  else
    Result:=GCD(m, n mod m);
end;
```

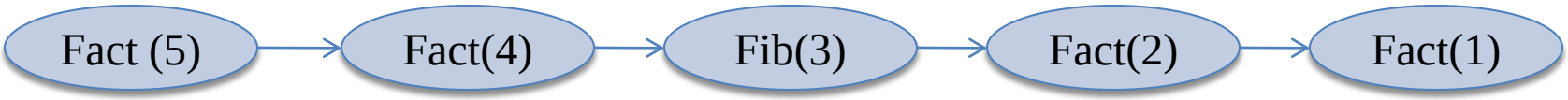
```
function GCD(n: Integer): Integer;
begin
  while (n<>0) and (m<>0) do
    if n>=m then
      n:=n mod m
    else
      m:=m mod n;
  Result:=n +m;
end;
```

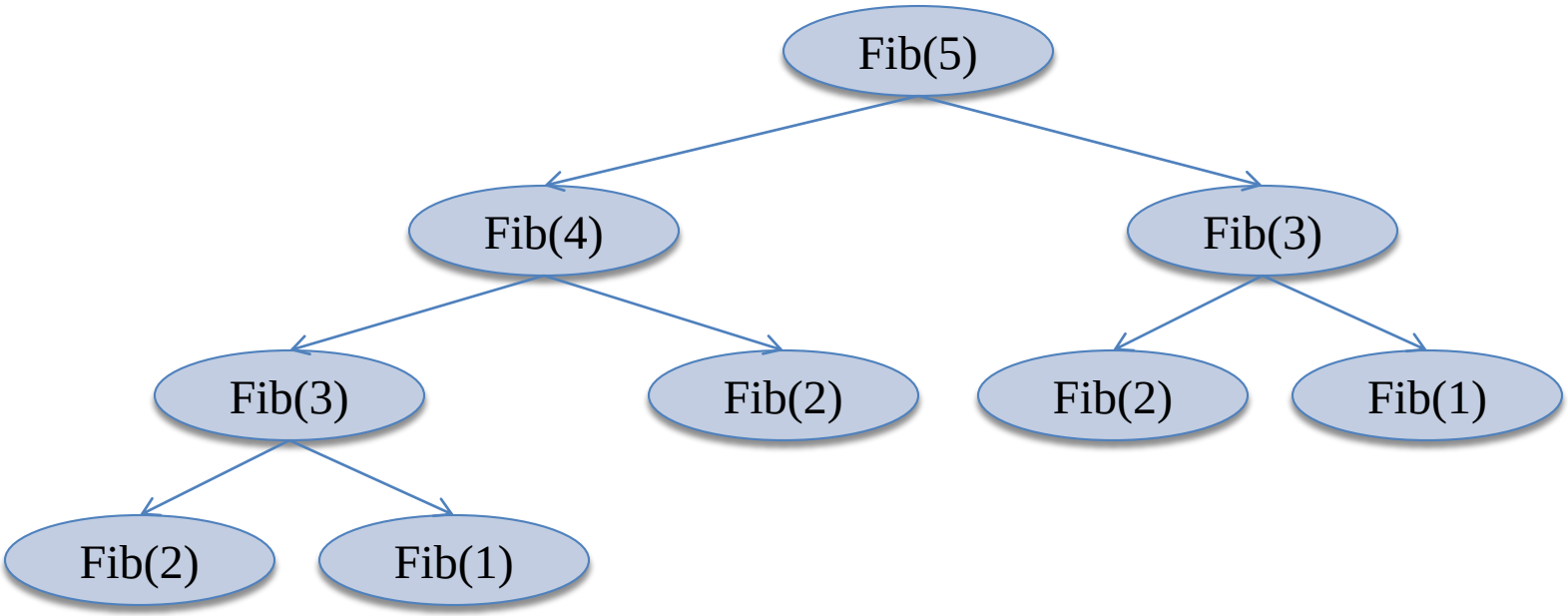
Пример: биномиальные коэф-ты


```
function C(n, m: Integer): Integer;
begin
    if (m=0) or (m=n) then
        Result:=1
    else
        Result:=C(n-1,m)+
            C(n-1,m-1);
end;
```

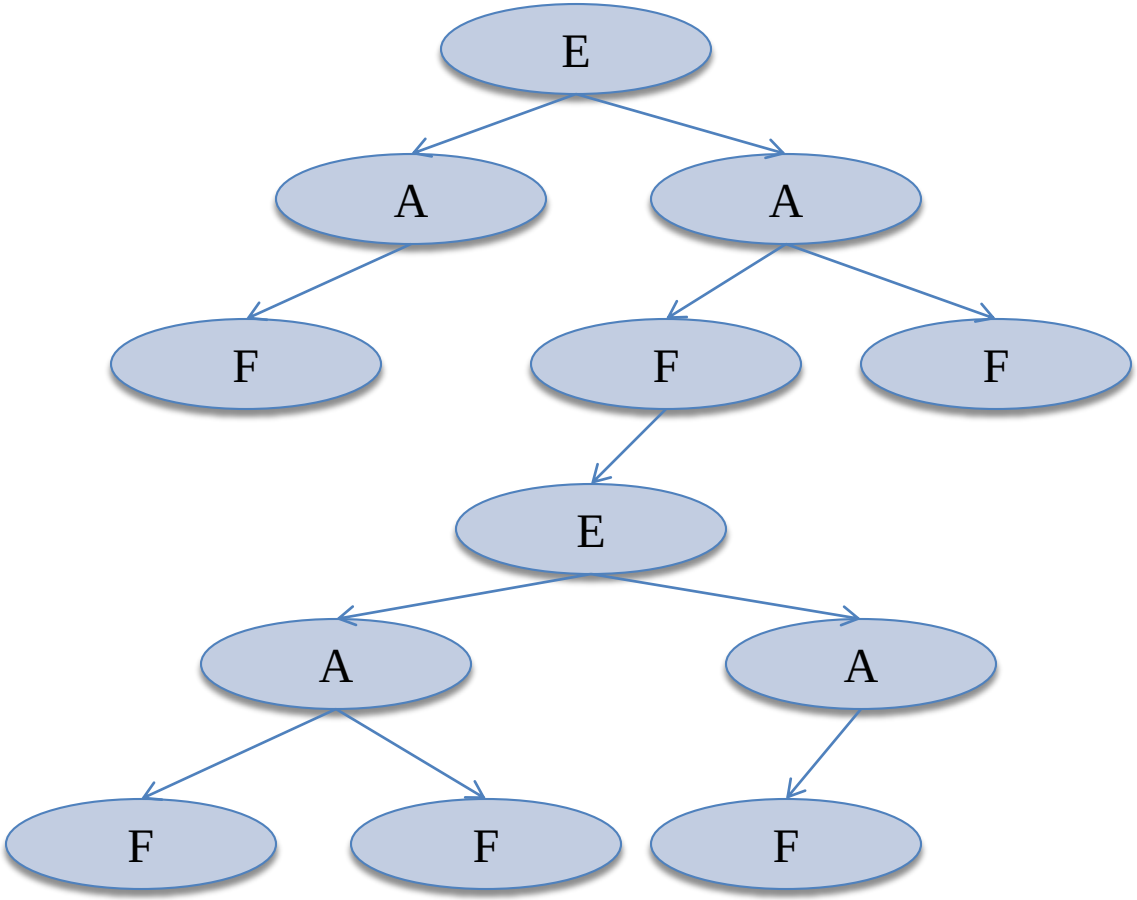
```
function C(n, m: Integer): Integer;
var i, k, l, p, q: Integer;
begin
    k:=max(n-m,m);
    l:=min(n-m,m);
    p:=1; q:=1;
    for i:=k+1 to n do p:=p*i;
    for i:=2 to 1 do q:=q*i;
    Result:=p div q;
end;
```

Количество рекурсивных вызовов





Количество рекурсивных вызовов



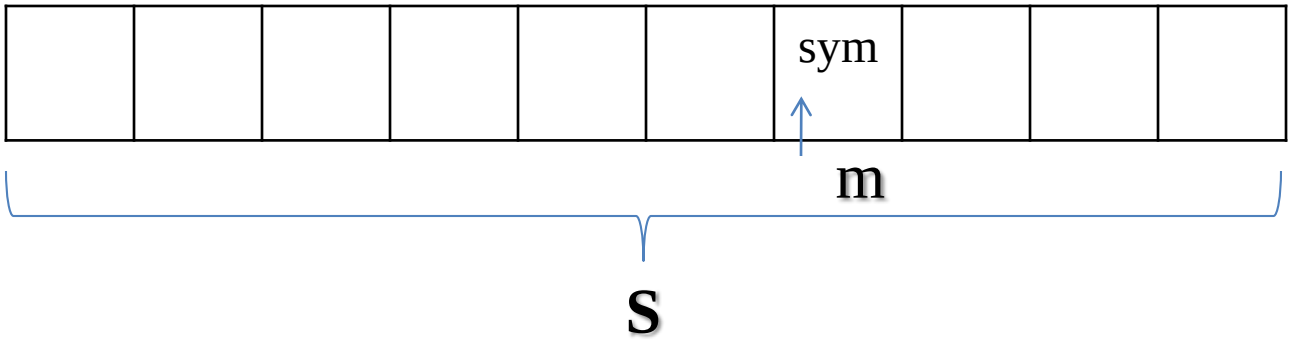
Распознавание символьной цепочки

- Разработать распознаватель символьной цепочки вида <*выражение*>.
  - <*выражение*> ::= <слагаемое> [<знак аддитивный><слагаемое>]
  - <слагаемое> ::= <множитель> [<знак мультипликативный><множитель>]
  - <множитель> ::= <переменная>|<число>| (<выражение>)
  - <знак аддитивный> ::= + | -
  - <знак мультипликативный>::= * | /
  - <переменная> ::= a | b | … | z
  - <число> ::= 0 | 1 | … | 9
- Примеры цепочек: a+b+c, (c-d)*a+n/z

Распознаватель цепочки

Распознавание цепочки: глобальные данные


```
var
    S: String; { распознаваемая строка }
    Sym: Char; { текущий символ }
    m: Integer; { маркер (номер текущего символа) }
    IsError: Boolean; { флаг недопустимости цепочки }
```



Распознавание цепочки: вспомогательные подпрограммы


```
{ Инициализировать }
procedure Init;
begin
    ReadLn(S);
    m:=1;
    Sym:=S[m];
    IsError:=TRUE;
end;
```

```
{ Передвинуть маркер }
procedure Next;
begin
    m:=m+1;
    Sym:=S[m];
end;
```

Распознавание цепочки: вспомогательные подпрограммы


```
{ Текущий символ – буква? }
function IsLetter: Boolean;
begin
    Result:=Sym in ['a'..'z'];
end;

{ Текущий символ – цифра? }
function IsDigit: Boolean;
begin
    Result:=Sym in ['0'..'9'];
end;
```

```
{ Текущий символ – "("? }
function IsLeftPar: Boolean;
begin
    Result:=Sym='(';
end;

{ Текущий символ – ")"? }
function IsRightPar: Boolean;
begin
    Result:=Sym=')';
end;
```

Распознавание цепочки: вспомогательные подпрограммы


```
{ Достигнут конец строки? }
function EOS: Boolean;
begin
    Result:=m>Length(S);
end;

{ Возникла ошибка }
procedure RaiseError;
begin
    IsError:=TRUE;
end;
```

Распознавание цепочки: проверка цепочки


```
procedure Recognize; { Проверка цепочки }
begin
    if S<>'' then
        Expression; { Проверка выражения }
    if (not IsError) and (S<>'') and EOS then
        WriteLn('Цепочка синтаксически правильна')
    else
        WriteLn('Цепочка содержит ошибки');
end;
```

Распознавание цепочки: проверка выражения


```
procedure Expression;  { Проверка выражения }
begin
    if IsLetter or IsDigit or IsLeftPar then begin
        Addend; { Проверка слагаемого }
        while IsPlusSign or IsMinesSign do begin
            Next;
            Addend;  { Проверка слагаемого }
        end;
    end
    else
        RaiseError;
end;
```

Распознавание цепочки: проверка слагаемого


```
procedure Addend;  { Проверка слагаемого }
begin
    if IsLetter or IsDigit or IsLeftPar then begin
        Factor; { Проверка множителя }
        while IsMulSign or IsDivSign do begin
            Next;
            Factor;  { Проверка множителя }
        end;
    end
    else
        RaiseError;
end;
```

Распознавание цепочки: проверка множителя

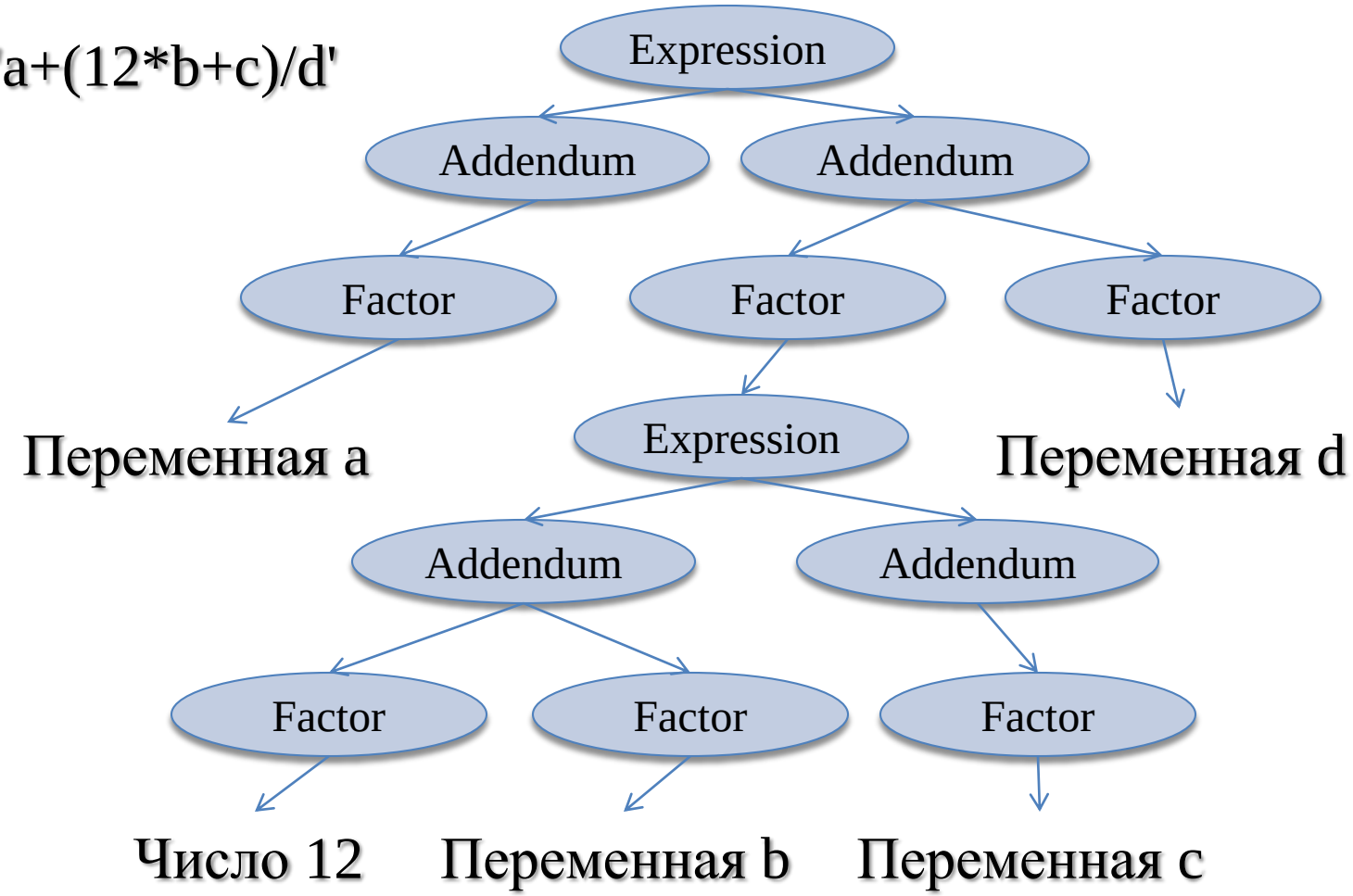

```
procedure Factor;  { Проверка множителя }
begin
    if IsLetter then Next else
        if IsDigit then Number else
            if IsLeftPar then begin
                Next;
                Expression; { Проверка выражения }
                if IsRightPar then Next
            end
            else RaiseError;
end;
```

Распознавание цепочки: тело программы


```
begin
    Init;
    Recognize;
end;
```

Количество рекурсивных вызовов

S='a+(12*b+c)/d'



Рекурсивные алгоритмы с возвратом

- Поиск с возвратом – метод перебора всех возможных решений для нахождения множества *полных решений*. *Частичное решение* определяет некоторое подмножество полных решений.
- При поиске частичного решения многократно делается попытка расширить *текущее частичное решение*. Если расширение невозможно, то на текущем шаге поиска происходит возврат к предыдущему (более короткому) текущему частичному решению и делается попытка расширить его другим способом.
- Задачи на поиск (перебор) с возвратом:
  - автоматизация процесса доказательства,
  - реализация стратегий в компьютерных играх,
  - поиск расстановок на шахматной доске
  - поиск путей в лабиринте.

Поиск пути в лабиринте

- Правила поиска пути от входа к выходу:
  - из текущего квадрата перейти в еще не исследованный соседний квадрат
  - если соседние квадраты уже исследованы и выход не найден, то вернуться на один квадрат назад по пройденному пути

0	0	0	1
0	1	0	0
1	0	1	0
0	1	0	0
0	0	0	1

Поиск пути в лабиринте


```
function SearchPath(iEntry, jEntry, iExit, jExit: Integer): Boolean;
var sp: Boolean;
begin
    if (iEntry=iExit) and (jEntry=jExit) then Result:=TRUE
    else begin
        sp:=FALSE;
        if IsDownPossible(iEntry,jEntry) then begin
            sp:=SearchPath(iEntry+1,jEntry);
            if sp then Path[iEntry+jEntry-1]:=DOWN;
        end;
        if not sp and IsRightPossible(iEntry,jEntry) then begin
            sp:=SearchPath(iEntry,jEntry+1);
            if sp then Path[iEntry+jEntry-1]:=RIGHT;
        end;
        Result:=sp;
    end;
end;
```

0	0	0	1	0
1	0	0	0	0
1	1	0	1	0
0	0	0	1	1
1	0	0	0	0

ВПРАВО
ВНИЗ
ВПРАВО
ВНИЗ
ВНИЗ
ВНИЗ
ВПРАВО
ВПРАВО

Обход конем шахматной доски


```
function Knight(move,col,row: Integer): Boolean;
var colNew,rowNew: Integer; K: Boolean;
begin
    repeat
        K:=FALSE;
        GetNextMove(row,col,rowNew,colNew);
        if Board[rowNew,colNew]=0 then begin
            Board[rowNew,colNew]:=move;
            if move=N*N then tempRes:=TRUE  else begin
                K:=Knight(move+1,colNew,rowNew);
                if not K then then Board[colNew,rowNew]:=0
            end;
        end
    until K or NoMoreMoves;
    Knight:=K;
end;
```

	1		8	
2				7
3				6
	4		5	

25	18	7	12	3
8	13	4	17	6
21	24	19	2	11
14	9	22	5	16
23	20	15	10	**1**

Заключение

- Рекурсивный алгоритм – алгоритм, определяемый через себя прямо или косвенно.
- Для создания корректного рекурсивного алгоритма необходимо определить базу и шаг рекурсии.
- Реализация механизма рекурсии: сегмент стека, прямой и обратный ход рекурсии
- Рекурсивные алгоритмы более красивы, чем итерационные, однако их выполнение связано с дополнительными накладными расходами.
- Одно из применений рекурсии – рекурсивные алгоритмы с заглядыванием вперед и рекурсивный поиск с возвратом.