КОМП'ЮТЕРНІ ЗАСОБИ, МЕРЕЖІ ТА СИСТЕМИ

УДК 681.3

В.П. ЗИНЧЕНКО, Н.П. ЗИНЧЕНКО

РЕКУРСИВНЫЕ АЛГОРИТМЫ И ИХ ОСОБЕННОСТИ

Показано, что использование рекурсивной процедуры во многих случаях позволяет придавать алгоритмам компактную и наглядную форму, в частности, для вычисления значений функций, задаваемых рекурсивными соотношениями. Показаны особенности реализации рекурсии в программных средах персональных компьютеров.

© В.П. Зинченко, Н.П. Зинченко, 2003

Введение. Рекурсия (Recursion) для особенности большинства программистов остаются “таинственной незнакомкой”. Однако, именно рекурсия позволяет построить рекурсивный алгоритм решения некоторых задач, которые нагляднее и эффективнее по сравнению с интерактивным алгоритмом. Реализация рекурсии возможна в любой программной среде ПК, что и показано в работе.

Механизм рекурсии. Рекурсивная процедура (РП) – процедура в описании которой содержится явное обращение к ней самой непосредственно (прямая рекурсия) или с помощью другой процедуры (косвенная рекурсия).

Использование РП во многих случаях позволяет придавать алгоритмам компактную и наглядную форму, в частности, для вычисления значений функций, задаваемых рекурсивными соотношениями, например: вычисление факториала $n! = F(n) = \begin{cases} F(0) = 1 \\ F(n-1) = n \end{cases}$;

вычисление ряда Фибоначчи

$$F(n) = \begin{cases} 1, \text{при } n = 0 \text{ или } n = 1 \\ F(n-1) + F(n-2) \end{cases}.$$

Использование РП связано с рекурсивным входом в процесс выполнения программы в один и тот же блок до выхода из него. Число рекурсивных входов называется уровнем рекурсии. На разных уровнях рекурсии одинаковые величины, локализированные в блоке, имеют разные значения. Эта особенность РП затрудняет их реализацию.

Во многих языках программирования допускаются также рекурсивные обращения к процедурам, при котором оператор процеду-

ры в качестве фактического параметра содержит идентификатор этой же процедуры, а соответствующий формальный параметр вызывается по наименованию. Например, обращение $f(f(x))$ к процедуре рекурсивно, если параметр x вызывается по наименованию.

Рекурсивная функция (РФ) – функция, которая определяется через саму себя, например функция Акермана, которая индуктивно задана на парах неотрицательных целых чисел:

$A(0,n)=n+1;\ A(m+1,0)=A(m,1);\ A(m+1,n+1)=A(n,\ A(m+1,n)),\ m,\ n \geq 0$.

Следовательно, $A(1,n)=n+1;\ A(2,n)=2n+3;\ A(3,n)=2^{n+3}-3$.

Всякая рекурсивность этой функции используется для проверки способностей компиляторов или ПК выполнять рекурсию. Эту РФ можно рассматривать как функцию одной переменной $Ack(n)=A(n,m)$. Функция Акермана является РФ, а не примитивно РФ вследствие очень быстрого возрастания ее значений по мере увеличения m.

В общем случае одна процедура может обращаться к другой процедуре [1]. Например, основная программа вызывает процедуру *P*, которая в свою очередь вызывает процедуру *Q*. Как частный случай, вызываемой процедуры *Q* может быть сама вызывающая процедура *P* (*Q*=*P*), и тогда *P* окажется РП рис.1. Таким образом в РП есть команда вызова ее самой, т.е. передача управления на ее начало (прямая рекурсия) рис. 2.

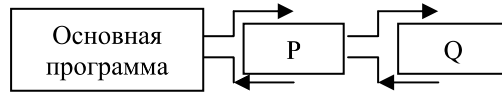

РИС. 1

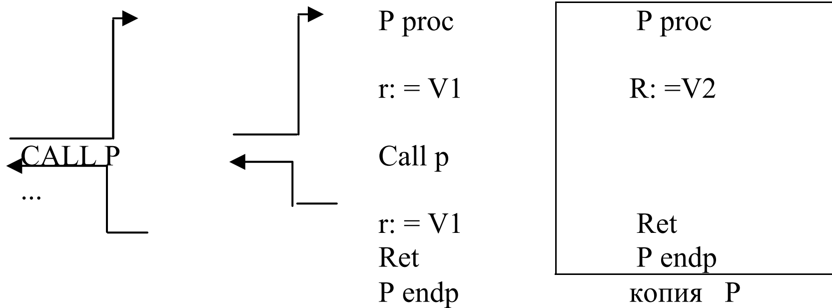

РИС. 2

В таком случае, рекурсивное обращение лучше рассматривать не как вызов самой процедуры, а как вызов ее копии: можно считать, что существует много копий такой процедуры, и при рекурсивном обращении из одного экземпляра вызываем другой экземпляр. Такая трактовка рекурсивных вызовов нагляднее и понятнее.

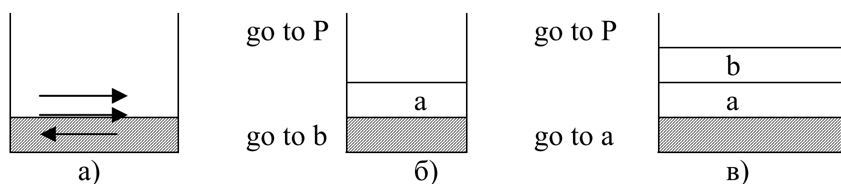

РИС. 3

Поскольку РП вызывает сама себя, то возникает вопрос о ее зацикливании и механизма выхода из рекурсии. РП не зациклится, если она описана правильно, если в ней есть нерекурсивная ветвь, т.е. такой путь вычисления, при котором рекурсивный вызов обходится, и если при каком-то обращении вычисление будет выполняться по этой ветви.

Пусть, например, при первом (из основной программы) вызове процедуры *P* внутренняя команда *CALL* сработает, а при втором (рекурсивном) вызове она уже обходится. Тогда происходит следующее: основная программа вызывает первую копию процедуры *P* и заносит в стек адрес возврата рис. 3 б. В этой копии выполняется команда *CALL P*, по которой в стек заносится адрес возврата рис.3в и передается управление в начало второй копии процедуры. Согласно нашемупредположению, команда *CALL* в этой копии не выполняется, поэтому вычисление доходит до команды, которая считывает из стека адрес возврата b и передадут по нему управление рис. 3 б. Тем самым происходит возврат в первую копию процедуры, и ее работа возобновляется. Команда из этой копии считывает из стека адрес возврата *a* рис. 3 а и передает по нему управление, в результате чего происходит возврат в основную программу. Зацикливания нет.

Из примера видно, что для правильной организации РП важно для передачи адресов возврата использовать стек. Во-первых, благодаря стеку адрес возврата не «забывает» прежний адрес возврата, что было бы, если бы адреса передавались через регистр. Во-вторых, благодаря стеку адреса возврата в нужном порядке – первым всегда берем адрес, записанный в стек последним. В МП i80x86/ iPx для реализации РП введены команды *Call* и *Ret*, которые организуют передачу адресов возврата именно через стек.

Регистры и оперативная память. Рассмотрим проблему, связанную с использованием регистров в РП. Предположим, что РП использует регистр *r*, не заботясь о сохранении его значения, и пусть она сначала этому регистру присваивает некоторую величину V_1 ($r:=V_1$), а затем использует его значение, и пусть между этими присваиваниями РП обращается сама к себе. Поскольку при рекурсивном вызове начинают работать те же самые команды, то регистру *r* снова будет что-то присвоено, но уже какое-то новое значение V_2. Это потому, что в циклах выполняются одни и те же команды, но при каждом шаге они оперируют с разными данными. Так и здесь, команда присваивает регистру *r* – та же самая, но присваиваемое значение может быть другим. Поэтому, во второй копии РП $r:=V_2$. Пусть в этой копии нет рекурсивного вызова, тогда она завер-

шает работу и возвращает управление в первую копию РП. При этом $r:=V_2$, что нарушает работу первой копии РП, так как необходимо для нее чтобы $r:=V_1$. Это происходит потому, что копия РП не сохранила значения r. Если бы РП на входе сохраняла значение регистра r, а на выходе восстанавливала бы его, тогда при возврате из второй копии РП в регистре r сохранялось бы нужное значение V_1 и первая копия правильно продолжала бы работу. На рис. 4 приведен пример РП для вычисления чисел Фибоначчи ($n \geq 0$), где аргумент функции передается через регистр AL, а значение функции возвращается через регистр BX.

```
F   PROC
      CMP AL, 1    ; n>1
      JA F1
; нерекурсивная ветвь
      MOV BX, 1   ; n <=1 -> BX= F(n) = 1
      RET
; рекурсивная ветвь
F1:  PUSH AX       ; сохранить AX
     DEC AL        ; AL = n-1
     CALL F        ; BX = F(n-1) (AX не изменится)
     PUSH BX       ; сохранить F(n-1)
     DEC AL        ; AL = n-2
     CALL F        ; BX = F(n-2)
     POP AX        ; восстановить F(n-1), но уже в AX
     ADD BX, AX  ; BX = F(n) = F(n-2) + F(n-1)
     POP AX         ; восстановить исходное значение AX
     RET
F    ENDP
```

РИС. 4

Таким образом, если нерекурсивная процедура может сохранять, а может и не сохранять значения регистров, которые она использует, то РП просто обязана сохранять значения всех регистров в стеке, которыми она пользуется, чтобы новая копия РП сохраняла значения регистров из новой копии.

Аналогичная проблема возникает, если РП пользуется какими-либо ячейками памяти для временного хранения промежуточных результатов. В этом легко убедиться, если r рассматривать не как регистр, а как ячейку памяти. Поэтому, если РП использует значения этих ячеек, то она на входе должна сохранять прежние значения этих ячеек, а на выходе восстанавливать эти прежние значения. Для того чтобы не было лишних действий в РП лучше сразу организовать хранение данных в стеке.Таковы правила использования РП. Если их соблюдать, тогда проблем с реализацией РП не будет. Передавать параметры и результат РП можно через стек и через регистры. Это не влияет на результат рекурсии.

Рекурсия и побочный эффект. Рассмотрим РП вычисления факториала от неотрицательного целого числа рис. 5. Отметим, что в языке Паскаль нет никаких ограничений на вызовы РП, необходимо только понимать, что каждый очередной рекурсивный вызов приводит к образованию новой копии локальных объектов подпрограммы и они существуют независимо друг от друга [2, 3].

```
function fact (N: word):
integer;
  begin
     if n =1 then fact: = 1
            else
fact: = N*fact (N-1)
  end;
```

РИС. 5

Рассмотрим неожиданный эффект при РП, причиной которого является изменение значений нелокальных переменных в теле функции [4].

Если в некоторой функции имеются конструкции (например, оператор присваивания), изменяющие значения переменных, описанных в объемлющих блоках, то может возникнуть ситуация, при которой значение выражения, использующего вызов такой функции, зависит от порядка следования операндов, что является потенциальным источником трудноуловимых программных ошибок, которые и приводят к побочным эффектам. Пример побочного эффекта можно наблюдать при выполнении РП, текст которой приведен на рис. 6.

```
Program S_ EF;
Var  a, z: integer;
function Ch (x: integer): integer;
begin
{изменяем значение нелокальной переменной}
z: = z-x;
Ch: = sqr (x)
end;
begin
z: =10; a: = Ch(z); Writeln (a, z);
z: =10; a: = Ch(10)* Ch(z); Writeln (a, z);
z: =10; a: = Ch(z) )* Ch(10); Writeln (a, z);
end.
```

РИС. 6

Ее выполнение приводит к выводу таких пар значений: 100 0, 100000 -10. То есть два последних присваивания переменной *a* дают различный результат. Это происходит потому, что существует зависимость функции от глобальных по отношению к ней переменных. Отметим, что современные языки (например, Ada) содержат механизмы прямых запретов на подобные действия.

Рассмотрим РП, текст которой приведен на рис. 7. Вызов Pr (1) означает, что Pr вызывает себя каждый раз с помощью Pr (2), Pr (3) …, до тех пор, пока условие zvp не отменит новый вызов. При каждом вызове выполняется оператор an1, после чего порядок выполнения операторов прерывается новым вызовом Pr (i+1). Чтобы для каждого вызова был отработан и оператор an2, все локальные переменные процедуры сохраняются в стеке. Локальные параметры помещаются в стек один за другим и выбираются из стека в обратной последовательности по правилу LIFO (Last In First Out) [5].

```
Procedure  Pr ( i: integer );
  Begin
    an1;
      if zvp then Pr ( i+1 );
    an2;
  End;
```

РИС. 7

В Паскале можно пользоваться именами лишь тогда, когда в тексте программы этому предшествует их описание. РП является единственным исключением из этого правила, так как ее имя можно использовать сразу же, не закончив его описание.

Рассмотрим пример бесконечной РП [6], с помощью которой можно установить, насколько велик стек (см. рис. 8). При использование директивы {$S+} при переполнении стека получим сообщение об ошибке, а при использовании директивы {$S-} – нет (зависание). РП будет прервана с выдачей сообщения об ошибке: “Error 202: stack overflow error“.

Стек связан с динамической областью памяти. Использовав директиву {$M} можно управлять размером стека.

РП не должна восприниматься как некий программистский трюк. Это скорее некий принцип, метод. Если в программе нужно выполнить что-то повторно, можно действовать двумя способами: с помощью последовательного присоединения (или итерации в форме цикла); с помощью вложения одной операции в другую (рекурсии).

```
Program Stack_Test ;
procedure Pr (i : integer);
begin
if i mod 1024 = 0 then Writeln (i: 6);
pr (i+1);
end;
Begin
Pr (1);
End.
```

РИС. 8

Так в примере (рис. 9) один раз счет от 1 до *n* ведется с помощью цикла, а второй – с помощью рекурсии. При этом хорошо видно, как заполняется, а затем освобождается стек. В процедуре REK операция writeln (i:30) выполняется перед рекурсивным вызовом, после чего writeln (i:3) освобождает стек. Поскольку рекурсия выполняется от *n* до 1, вывод по команде writeln(i:30) выполняется в обратной последовательности *n*, *n–1*,…,1, а вывод по команде writeln(i:3) – в прямой последовательности 1, 2,… , *n* (согласно принципу LIFO).

```
Program   SCHL_and_REK;
    Var   n : integer ;
     Procedure   REK (i : integer );
        Begin
         Writeln (i:30);
          If i < 1 then REK(i – 1);
          Writeln (i : 3) ;
        End;  {Рекурсия}
Procedure SCHL ( i : integer );
        Var k : integer ;
         Begin
           k: = 1 ;
           While k <= i do Begin
                  Write (k:3) ;
                   k: = k+1;
                              End ;
          End ;  {Цикл}
  Begin
Write (‘Введите n : ‘); Readln (n);
Writeln (‘Пока :  ‘); SCHL (n);
Writeln (‘Рекурсия : ‘); REK (n) ;
  End .
```

РИС. 9

Этот пример показывает принципиальное различие между итерацией и рекурсией: итерации необходим цикл и локальная переменная k как переменная цикла. Рекурсии ничего этого не требуется.

Рассмотрим пример РП convert, которая переводит десятичное число z в восьмеричную систему путем деления его на 8 и выдачи остатка в обратной последовательности рис. 10.

```
      Program O_D ;
      Var z : integer ;
         Procedure convert( z : integer );
           Begin   {рекурсивный вызов}
           If z > 1 then convert( z div 8);
           Write ( z mod 8 : 1 ) ;
         End ;
      Begin
        Writeln (‘Введите число:‘); Readln(z);
        Writeln (‘Десятеричное число: z : 6 ‘);
        Write (‘Восьмеричное число:‘); Convert (z);
End .
```

РИС. 10

Следующий пример рис. 11 позволяет вычислить *n*–й элемент ряда Фибоначчи как интерактивно, начиная с *x[1]* до *x[n]*, так и рекурсивно. Причем, РП (функция) вызывает себя дважды и глубина рекурсии индицируется. Перед каждым рекурсивным вызовом выводится ASCII – символ с номером 8 (BackSpace), а после вызова вновь стирается. Тем самым можно наблюдать за работой программы, поскольку программа за счет Delay (300) приостанавливается на 0.3 секунды.

Этот пример демонстрирует прежде всего различия между итерацией и рекурсией. Итерации необходим цикл и вспомогательные величины; итерация сравнительно ненаглядна (функция fibit). Рекурсия обходится без вспомогательных величин и обычно проще для понимания, что подтверждает такая запись:

```
If  (n = 1) or ( n = 2 ) then fibrek:=1
                              else fibrek:=fibrek(n – 1) + fibrek(n – 2).
```

Итерация требует меньше места в памяти и процессорного времени, чем рекурсия, которой необходимы затраты на управление стеком.

```
Program Fibonacci ;
Uses Crt ;
Var n , result : integer ;
  Function Fibit ( n : integer ): integer ;
   Var a, b, c, i : integer ;
   Begin
    a: = 1; b: = 1;
    if ( n = 1) or  ( n = 2)  then  fibit: = 1
           else  begin
           for i : = 3 to n do
                 begin
                 c: = a + b; a: = b; b: = c;
                 end ;
                 fibit : = c ;
                 end ;
   end;
  Function Fibrek ( n : integer ) : integer ;
   Begin
    Write (‘ – ‘) ;  delay (300) ;
     if  (n = 1) or (n = 2) then fibrek : = 1
    else fibrek: = fibrek (n – 1) + fibrek (n - 2) ;
     Write (chr (8) , ‘   ‘ , chr (8)) ; Delay (300);
   End ;
Begin
  ClrScr ;
   Write (‘n = ‘) ; Readln (n) ;
   Writeln (‘Интерактивно’, fibit (n) :5 );
   Writeln (‘ Рекурсивно :  ‘ ) ;
   Write (‘ 000000 …  ! … # … ! … # … ‘) ;
   Writeln (‘Глубина рекурсии: ‘) ;
   Result : = fibrek (n) ;
   Writeln ;
   Write (result) ;
End .
```

РИС. 11

Итак, если для некоторой задачи возможны два решения, предпочтение следует оказать итерации. Правда, для многих задач рекурсивная формулировка совершенно прозрачна, в то время как построить итерацию оказывается весьма сложно. Например, для двоичных деревьев (рис. 12) в РП вычисления простых чисел. В этом случае имеем косвенную рекурсию, что демонстрируется использованием директивы forward.

Следующая программа выдает простые числа от 1 до n, для чего используются функции next и prim, которые вызываются рекурсивно. Одновременно это является примером применения директивы forward.

В Паскале все процедуры и функции рекурсивны. При рекурсивном вызове процедур или функций используется стековая организация хранения параметров и адресов возврата. В момент ухода в блок в стеке запоминается адрес возврата, слово состояния программы PSW, значение фактических параметров. Локальные переменные тоже помещаются в стек. Такой механизм хранения информации очень удобен для вызова процедурами (функциями) самих себя.

```
program prim;
  var n, i : integer; c: char;
  function next (i: integer): integer; forward;
{прямая ссылка вперед на next}
  function prim (i: integer ): boolean;
    {prim — true, если j простое число, a false в противном случае}
  var k: integer;
   begin
     k: = 2;
     while (k*k < =j) and (j mod k <> 0) do  k: = next (k);
{k изменяется от 2 до корня из j (простые числа), при этом проверяется
деление j на одно из таких простых чисел}
if j mod k = 0 then prim: = false else prim:= true;
  end;
 function next ; {параметры уже стоят в ссылке вперед,
                   next вычисляет следующее за j простое число}
var l: integer;
begin
 l: = i +1;
 while not (prim (l)) do l:= l + 1;  {next вызывает prim}
 next : = 1
end;
   begin
writeln (‘Введите положительное число n :’) ;
readln ( n ) ;
for i : = 2 to n do
  begin
if prim ( i) then writeln ( i : 14 ) else writeln ( i : 4 ) ;
if i mod 23 = 0 then begin write (‘< RET>‘: 60 ); read (c, c); end;
end
end.
```

РИС. 12

Рассмотрим реализацию РП “Ханойские башни” рис. 13, где необходимо перенести башню с левой иглы на правую, причем за один раз можно переносить только одно кольцо и при этом можно насаживать только кольцо с меньшим диаметром на кольцо с большим диаметром [7]. Итак, для обращения к процедуре переноса башни с именем M_T следует записать: M_T(3, l, r, h);, что означает перенести башню из трех дисков с левой иглы на правую. В качестве рабочей используется средняя игла.

```
Program Towers;
type  position = (L, C, R);
var  N: integer;
    Procedure M_D (F, T: Position);
    Procedure W_P (p: Position);
       Begin
          Case p of
             L : Write (’1‘);
             R : Write (’3‘);
             C : Write (’2‘);
          End ;
      End;  {W_P}
      Begin  {M_D}
      W_P (F);
      Write (’-‘);
      W_P (T);
      Writeln;
       End;     {M_D}
  Procedure M_T (h: integer; F, T, W: position);
     Begin    {M_T}
          If h >0 then
       begin
         M_T (h –1, F, W, T);
         M_D (F, Tol);
         M_T (h –1, F, W, T);
         End;
         End;  {M_T}
  Begin
     Readln (N);
     M_T (N, R, L, C);
  End.
```

РИС. 13

Рассмотрим РП вычисления значения функции $y = x^n$, где $n \geq 0$. Очевидно, что $x^n = \begin{cases} 1, если\ n = 0 \\ x \cdot x^{n-1}, если\ n > 0 \end{cases}$, поэтому решение такое, как представлено на рис. 14.

```
Function Power (x: real; Ninteger): real;
  Begin
If N = 0 then Power : =1;
  else Power : = x*Power (x, N-1)
  end;
```

РИС. 14

Взаимные рекурсии. В языке Паскаль используется ключевое слово Forward при взаимно – рекурсивном вызове РП (процедур и функций). На практике встречаются функции вида: *f(x)=p(x,g(x))* и *g(x)=t(x,f(x))*. Взаимное расположение процедур реализации вычислений функций f и g может быть организовано в следующем виде, как показано на рис. 15.

```
Procedure G (x: t); forward;
  Procedure F (y, t);
    Begin  {t}
      …
     G(Y) {вызов процедуры  G}
      ...
  End;
 Procedure G;
   Begin  {G}
     ...
     F(x); { вызов процедуры F}
      ...
    End;
```

РИС. 15

Слово Forward является директивой, означающей, что сама процедура с именем G, будет представлена ниже. При этом в заголовке процедуры G, список формальных параметров опускается, так как этот список был произведен выше перед словом Forward.

Выводы. В результате выполненной работы показаны особенности рекурсивных процедур и их особенности при реализации прикладных программ на языке Паскаль и Assembler в программных средах персональных компьютеров.

1. *Пильщиков В.Н.* Программирование на языке ассемблера IBM PC. – М.: Диалог-МИФИ, 1994. – 288 с.
2. *Новичков В.С., Парфилова Н.И., Пылкин А.Н.* Язык Паскаль. – М.: Высшая шк., 1990. – 223 с.
3. *Тумасонис В., Дагене В., Григас Г.* Паскаль. Руководство для программиста: справ. – М.: Радио и связь, 1992 – 192 с.
4. *Зуев Е.А.* Язык программирования Turbo Pascal 6.0. – М.: Унитех, 1992. – 298 с.
5. *Бородич Ю.С., Вальвачев А.Н., Кузьмич А.И.* Паскаль для персональных компьютеров. Минск: Высш. шк.: БФ ГИТМП "НИКА", 1991. – 365 с.
6. *Херель Р.* Турбо Паскаль. – Вологда: МП «МНК», 1991. – 342 с.
7. *Довгань С.И., Литвинов Б.Ю., Сбитнев А.И.* Персональные ЭВМ: Турбо Паскаль V6.0: Учебное пособие. – К.: Информсистема сервис, 1993. – 426 с.

Получено 01. 07. 2002