Лекция 3

Рекурсия и избавление от нее

3.1 Рекурсивные процедуры

Рекурсивной называется процедура, вызывающая себя. Вообще, вызов (любой) процедуры происходит так: во время исполнения программы с шага s происходит переход на адрес начала процедуры; вычисления продолжаются (заметим, что при рекурсивном вызове «одни и те же» локальные переменные в вызываемой и вызывающей копиях процедуры имеют разные значения); затем происходит возврат на шаг $s+1$.

Пример 3.1 (числа Фибоначчи).
function f (i : integer) : integer;
begin
  if $i = 0$ then $f := 0$
  else if $i = 1$ then $f := 1$
  else $f := f(i-1) + f(i-2)$;
end; □

Пример 3.2 (задача о рюкзаке). Имеется N предметов и рюкзак объема U. Даны объемы предметов v_i и стоимости g_i. Требуется найти набор предметов максимальной стоимости, помещающийся в рюкзак.

function knapsack (U, N : integer, массивы целых чисел v и g) : набор целых чисел;
var $optG$: integer $= -1$;
  $nextG$: integer;
  opt : набор целых чисел = пустой;
  $next$: набор целых чисел;
  $temp$: набор целых чисел;
begin
  for $k := 1$ to N do
    *(*решаем, предмет с каким наименьшим номером берем*)*
    if $v[k] \leqslant U$ then
    begin
      (*вычисляем оптимальный набор для предметов с номерами, большими k,*)
        (*при условии, что предмет с номером k мы взяли и $v[k]$ он отнял*)

$temp$:= knapsack($U - v[k], N - k,$ части v и g при $i \geqslant k+1$);
перенумеровать числа в $temp$, учитывая, что рекурсивный вызов работал не
со всеми предметами, и у него была своя нумерация предметов;
$next := \{k\} \cup temp$;
$nextG$:= стоимость($next$);
if ($nextG > optG$) then begin $opt := next$; $optG := nextG$; end;
end;
$knapsack := opt$;
end; □

Пример 3.3 (проверка правильности выражения).
выражение ≡ сумма `;`
сумма ≡ терм | терм `+` сумма
терм ≡ буква | `(` сумма `)`

Например, выражением является

`a+(b+(c+d)+e);`

Читаем входной поток функцией getnext : char. Процедура getback возвращает символ во входной поток (чтобы в следующий раз был прочтен тот же символ), но применить ее можно лишь один раз (вернее, повторное применение не приводит к повторному откату). Наша задача — прочесть одно выражение до конца и выдать true, если оно корректно; либо прочесть до замеченной ошибки и выдать false.

```
function expression : boolean;
begin
    if (not sum) then expression:=false
    else if (not getnext = ';') then expression:=false
    else expression:=true;
end;

function sum : boolean;          (* рекурсивно читает длинную сумму до конца *)
begin
    if (not term) then sum:=false
    else if (not getnext = '+') begin getback; sum:=true; end;
    else if (not sum) sum:=false;
    else sum:=true;
end;

function term : boolean;
begin
    if (getnext in ['a'..'z']) term:=true
    else if (not getnext = '(') term:=false
    else if (not sum) term:=false
    else if (not getnext = ')') term:=false
    else term:=true;
end;
```
□

3.2 Реализация рекурсии в компьютере: стек

Стек. Вспомним, что стек — это абстрактная структура данных, имеющая операции PUSH, POP, и, если повезет с реализацией, `[·]`. Стандартной реализацией стека является

- непрерывный фрагмент оперативной памяти (его можно рассматривать как массив),
- счетчик: верхушка стека.

Ясно, что в этом случае реализовать операцию `[·]` можно эффективно, причем нумеровать элементы стека можно как снизу, так и сверху.

Реализация рекурсии.

Вызов процедуры:

- PUSH адрес возврата.
- PUSH параметры.
- JUMP процедура.
- Передвинуть счетчик (PUSH 0) на размер памяти, необходимый для хранения локальных переменных.

Возврат:

- Передвинуть счетчик (POP, …) на размер памяти, в которой хранились локальные переменные и параметры.
- POP адрес возврата и JUMP туда.

Передача результата — зависит от реализации.

3.3 Избавление от рекурсии

Способ 1: при помощи стека.

Реализовать стек в массиве. Решение годится и для РАМ.

Способ 2: динамическое программирование.

Пример 3.4 (числа Фибоначчи).
type intarray = массив[1] целых чисел;
function g (j : integer, p : intarray) : integer;
(*В p хранятся числа Фибоначчи с номерами от 0 до $j-1$,*)
(*мы туда дописываем число номер j.*)
begin
  if $j = 0$ then $g := 0$
  else if $j = 1$ then $g := 1$
  else $g := p[j-1] + p[j-2]$
end;

[1]Напомним, что массив — это структура данных, для которой есть операция `[·]`.

```
function f (i : integer) : integer;
var a : intarray;
begin
    for j := 1 to i do a[j] := g(j, a);
    f := a[i];
end;
```

□

Пример 3.5 (задача о рюкзаке).
Для всех k от 1 до N и p от 1 до общей стоимости всех предметов последовательно найдем наименьший по объему набор предметов общей стоимостью p, использующий лишь предметы с 1 по k (при этом можно пользоваться уже найденными оптимальными наборами для $k' < k$ и $p' < p$).

Чуть более формально,

```
function knapsackvalue(...):integer;
var k, p : integer;
    W : array[1..N, 1..∑_{i=1}^{N} g[i]] of integer;
(* В W[k, p] подсчитывается минимальный объем, достаточный для того, чтобы
набрать стоимость p при помощи первых k предметов.*)
begin
    for k := 1 to N do
        for p := 1 to ∑_{i=1}^{N} g[i] do
            W[k, p] = min(v[k] + W[k − 1, p − g[k]], W[k − 1, p]);

    p := 0;
    while W[k, p + 1] ≤ U do
        p := p + 1;

    knapsackvalue := p;
end;
```

Чтобы дать еще более формальное решение, следует вставить проверку на выход за границы массива и положить

$$W[\ldots, \leq 0] = 0; \quad W[0, > 0] = +\infty.$$

□

Упражнение 3.1. Доработать последний алгоритм так, чтобы он выдавал не только стоимость выбранных предметов, но и весь набор, и написать полную и корректную программу на Паскале. □