TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta strojní **Katedra aplikované kybernetiky**

doc. Ing. Vladimír V ě c h e t , CSc.

ALGORITMY A DATOVÉ STRUKTURY

Liberec 1997

PŘEDMLUVA

Předkládaný učební text je určen studentům IV. ročníku strojní fakulty, oboru Automatizované systémy řízení ve strojírenství. Vznikl upravením již rozebraných skript Programování, rekursivní algoritmy v Pascalu, vydaných v roce 1990. Tato skripta byla rozšířena především o zápisy rekursivních algoritmů v jazyku C.

Cílem předkládaného textu je podat hlubší informace o rekursivních programovacích technikách, které velmi často reprezentují mocný nástroj pro zefektivnění práce programátora. Není snahou zaměřit se na nějaký konkrétní programovací jazyk, který rekursi podporuje, nicméně u studentů oboru ASŘ ve strojírenství se znalost programovacích jazyků Pascal a C předpokládá a priori. Proto také původní obsah skript byl rozšířen o zápis alespoň některých algoritmů v jazyku C.

OBSAH

1. ÚVOD

V lit. [1] se píše, že "kdyby byly počítače existovaly ve středověku, byli by bývali programátoři upáleni na kolech jinými programátory za kacířství a je velmi pravděpodobné, že jedním z kacířstev by byla bývala víra (nebo nevíra) v rekursi". Vskutku, ještě dnes jsou názory na rekursi velmi rozporuplné. Mnozí programátoři se snaží dokazovat, že libovolnou rekursivní rutinu je lépe vyjádřit mnohem efektivněji pracující iterační rutinou. Jiní se přinejmenším na vybraných příkladech snaží dokázat pravý opak (a ne bez úspěchů).

Je však nesporné, že rekursivní způsob definování funkcí nebo procesů je v matematice mocným nástrojem úsporného vyjadřování, které pak má zcela logicky i svůj odraz v programovacích technikách. To, že se rekursivní operace provádějí málo efektivně je dáno spíše tím, že stávající kompilátory s ohledem na současné struktury počítačů nejsou schopné generovat efektivní výsledný kód. I když výsledný program nebude pracovat právě nejefektivněji, tak v mnoha případech je aplikace rekursivních metod jedinou zárukou efektivnosti práce programátora. V každém případě lze však z hlediska "počítačů budoucnosti" označit rekursivní programovací techniky za perspektivní a studenti by je měli znát.

2. REKURSIVNÍ DEFINICE A PROCESY

Všeobecně se pod pojmem rekurse rozumí způsob definování objektů nebo procesů pomocí jich samotných. Tak např. π je definováno jako poměr obvodu kruhu ku jeho průměru. To lze přirovnat k těmto instrukcím : urči obvod kruhu a jeho průměr, poděl obvod kruhu průměrem a výsledek označ π. Ale obvod kruhu je definován zase ve vztahu k π.

Ovšem nejznámějším příkladem rekursivně definované funkce je funkce faktoriál (symbol !) pro nezáporný celočíselný argument. Víme, že 0! je definováno jako 1 a pro kladné celočíselné *n* je *n*! definováno jako součin všech kladných čísel 1 až *n*. Tudíž můžeme napsat definici ve tvaru

$n! = 1$, pro $n = 0$

$n! = n * (n-1)*(n-2)* .... *1$, pro $n > 0$

Pro násobení jsme použili symbol *, přičemž tři tečky v zápisu definice je naše konvence pro označení součinu všech celých čísel od (*n*-3) do 2. Kdybychom se chtěli oprostit od této konvence, mohli bychom definovat *n*! výčtem definic pro každou hodnotu *n* zvlášť :

0! = 1 , 1! = 1 , 2! = 2*1 , 3! = 3*2*1 , atd.

Pro všechna nezáporná *n* by však takový výčet nebyl konečný. Tento nedostatek můžeme obejít tak, že zapíšeme algoritmus, který pro dané nezáporné celočíselné N vrací hodnotu N! v proměnné FACT :

```
X := N ; Y := 1 ;
while  X <> 0  do
   begin
        Y := X * Y ; X := X - 1
   end ;
FACT := Y
```

Takový algoritmus se nazývá **iterační**, jeho charakteristickým rysem je opakování nějakého procesu až k dosažení určité podmínky. Takto zapsaný algoritmus může být velmi snadno převeden do tvaru funkce v symbolickém jazyku, nicméně však nemůže sloužit jako matematická definice *n*!, vždyť u reálných počítačů jsme vždy omezeni počtem dvojkových řádů pro zobrazení celých čísel.

Jistě je známa matematická definice funkce *n*! pro nezáporná celočíselná *n* :

$n! = 1$, pro $n = 0$

$n! = n * (n-1)!$, pro $n > 0$.

Všimněme si, že 0! je definováno přímo, je vyjádřeno explicitně, kdežto pro $n > 0$ je $n!$ definováno pomocí $(n-1)!$. Takové definice, kdy objekty nebo procesy jsou definovány pomocí jednodušších případů těch samých objektů nebo procesů se nazývají **rekursivní**. Každá taková definice musí obsahovat explicitní definici pro určitou hodnotu nebo hodnoty, jinak by byla kruhová.

Tak např. chceme-li podle rekursivní definice vypočítat 4!, tak nejprve musíme vypočítat 3!, což ovšem předpokládá výpočet 2! atd. :

```
1.   4! = 4 * 3!
2.            3! = 3 * 2!
3.                     2! = 2 * 1!
4.                              1! = 1 * 0!
5.                                       0! = 1
```

Postupně jsme výpočet redukovali na stále jednodušší případ, až v řádku 5 je 0!, což je ovšem 1. Pak se můžeme vrátit zpět od řádku 5 k řádku 1 :

```
1'.  0! = 1
2'.  1! = 1 * 0! = 1 * 1 = 1
3'.  2! = 2 * 1! = 2 * 1 = 2
4'.  3! = 3 * 2! = 3 * 2 = 6
5'.  4! = 4 * 3! = 4 * 6 = 24
```

Zapíšeme nyní tento algoritmus pro výpočet $n!$ pro celá nezáporná n následovně :

```
1.   if  N = 0
2.     then  FACT := 1
3.     else  begin
4.              X := N - 1 ;
5.              "Nalezni hodnotu X! a označ ji Y" ;
6.              FACT := N * Y
7.           end
```

Klíč k celému algoritmu je však v řádku 5, který vlastně přepokládá znovu provedení téhož algoritmu, ovšem s hodnotou o 1 menší. Toto opakování se předpokládá tak dlouho, až na vstupu algoritmu bude 0 a pak tento algoritmus vrátí prostě hodnotu 1 do řádku 5. Pak se provede N*Y (N = 1 , Y = 1) a výsledek je znovu vrácen do řádku 5 atd., až se dosáhne hodnoty N!.

Poznamenejme hned na tomto místě, že jak pro $n!$, tak i pro většinu příkladů v této úvodní kapitole jsou iterační algoritmy mnohem efektivnější než rekursivní, které zde mají demonstrovat pouze principy rekurse. Později budou uvedeny příklady, které lze velmi elegantně a efektivně řešit pomocí rekursivních metod.

Druhá poznámka se týká explicitní definice pro určitou hodnotu. Pochopitelně, že platí :

$n! = (n + 1)! / (n + 1)$,

ale pokud explicitně vyjádříme pro celé $a \geq 0$ hodnotu $a!$, nejsme ovšem pro $b > a$ schopni podle takového algoritmu určit $b!$.

Jako jiný příklad lze uvést tzv. **binární vyhledávání**. Mějme nějaký seznam objektů, které jsou určitým způsobem v seznamu uspořádány. Jako příklad nám může sloužit telefonní seznam, slovníky slov apod. Naším úkolem je vyhledat v seznamu zadaný objekt, nebo konstatovat, že v seznamu není.

Pokud seznam obsahuje pouze jeden objekt, tak je úloha triviální. V opačném případě porovnáme hodnotu zadaného objektu s hodnotou objektu někde "uprostřed" seznamu. Dojde-li k rovnosti, je vyhledávání skončeno, jinak na základě provedeného porovnání se stejný proces provede buď v první nebo druhé polovině seznamu. Aplikujeme tudíž stejný proces na seznamy obsahující stále menší počet objektů, až je ev. tento počet roven 1. Vyjádřili jsme tudíž takový algoritmus rekursivně.

Nechť je tedy dáno nějaké vzestupně setříděné pole A a zapišme rekursivní algoritmus pro vyhledání prvku X v poli A od A[DOLNI] do A[HORNI]. Algoritmus umístí do proměnné BINVYH takovou hodnotu INDEX, že A[INDEX] = X, když . Když ovšem takový element v zadaném úseku pole A není, bude BINVYH = 0 (to ovšem předpokládá, že buď HORNI,DOLNI > 0, nebo HORNI, DOLNI < 0, tedy že nemůžeme uvažovat A[0]) :

```
1. if  DOLNI > HORNI
2.    then   BINVYH := 0
3.    else  begin
4.             STR := ( DOLNI + HORNI )  div  2 ;
5.             if  X = A[STR]
6.               then  BINVYH := STR
7.               else  if  X < A[STR]
8.                        then "Vyhledání X v poli A od A[DOLNI] do A[STR-1]" ;
9.                        else  "Vyhledání X v poli A od A[STR+1] do A[HORNI]" ;
10.          end
```

Poznamenejme hned, že triviální případ vyhledávání v poli o jedné složce není v tomto algoritmu vyjádřen přímo. Místo toho je zde porovnání vyhledávaného prvku s tímto prvkem pole a v případě nerovnosti by vyhledávání pokračovalo v úseku pole, jež neobsahuje žádný element a takový případ je indikován podmínkou DOLNI > HORNI a algoritmus vrací hodnotu 0.

Nechť např. pole A obsahuje tyto prvky :

1 , 3 , 4 , 5 , 17 , 18 , 31 , 33 .

Uvedený algoritmus budeme prezentovat na příkladě X = 17, DOLNI = 1 a HORNI = 8 :

ř. 1 : DOLNI > HORNI ? 1 není větší než 8, tudíž se provede část **else**

ř. 4 : STR := (1 + 8) **div** 2 = 4

ř. 5 : X = A[4] ? 17 není rovno 5, tudíž se provede část **else**

ř. 7 : X < A[4] ? 17 není menší než 5, tudíž se provede část **else** v ř. 9

ř. 9 : opakování algoritmu s hodnotami DOLNI := STR + 1 = 5 a HORNI := HORNI = = 8

ř. 1 : DOLNI > HORNI ? 5 není větší než 8, tudíž se provede část **else**

ř. 4 : STR := (5 + 8) **div** 2 = 6

ř. 5 : X = A[6] ? 17 není rovno 18, tudíž se provede část **else**

ř. 7 : X < A[6] ? 17 není menší než 18, tudíž se provede část **then**

ř. 8 : opakování algoritmu s hodnotami DOLNI := DOLNI = 5 a HORNI := STR-1 = 5

ř. 1 : DOLNI > HORNI ? 5 není větší než 5, tudíž se provede část **else**

ř. 4 : STR := (5 + 5) **div** 2 = 5

ř. 5 : X := A[5] ? 17 je rovno 17, tudíž se provede část **then**

ř. 6 : BINVYH := 5

Protože za krokem reprezentovaným příkazem **if** v ř. 7 již v algoritmu není žádný příkaz, vrátí nám tento algoritmus hodnotu 5.

Zcela přirozeným požadavkem na rekursivní algoritmy je, že musí generovat konečnou posloupnost volání sebe sama, tzn. že musí existovat "cesta ven" z této posloupnosti rekursivních odkazů. Tento požadavek je zajišťován nerekursivní částí rekursivních definic. Tak např. u popsané rekursivní definice *n*! je nerekursivní část 0! = 1. U binárního vyhledávání je nerekursivní část ve tvaru :

if DOLNI > HORNI **then** BINVYH := 0
if X = A[STR] **then** BINVYH := STR

Poznámka : V literatuře se často rekursivní definice zapisují ve tvaru podmíněných příkazů (tento tvar zápisu poprvé zavedl McCarthy). Podmíněné výrazy mají tvar :

$[B_1 \to V_1, B_2 \to V_2, ..., B_{n-1} \to V_{n-1}, V_n],$

kde B_i jsou booleovské výrazy a V_i jsou nějaké výrazy. Vyhodnocování se provádí zleva dokud se nenajde B_i , jehož hodnota je true a pak hodnota je odpovídající V_i. Pokud všechna B_i vrací false, hodnota je V_n. Tak např. pro funkci faktoriál bude :

$FAK(N) = [(N = 0) \to 1, N * FAK(N-1)]$

Cvičení

1. Napište iterační a rekursivní verzi algoritmu pro výpočet součinu A*B (užitím sčítání), kde A, B jsou nezáporná celá čísla.
2. Je dáno pole A se složkami typu integer. Napište nerekursivní algoritmus pro nalezení maximálního prvku pole A.
3. Stejně jako v předchozím příkladě je dáno pole A se složkami typu integer. Napište rekursivní algoritmus pro výpočet součtu prvků pole A od indexu D do indexu H.
4. Napište iterační verzi algoritmu pro binární vyhledávání (tj. modifikujte meze DOLNI a HORNI přímo).
5. Vztah mezi Besselovými funkcemi prvního druhu stejného argumentu, ale rozdílného řádu, lze zapsat ve tvaru :

$$J_{n+1}(x) = (2 \,.\, n / x) \,.\, J_n(x) - J_{n-1}(x) \,.$$

Napište rekursivní definici $J_n(x)$ ve tvaru zavedeném McCarthym pro případ, že pro určitý argument x známe $J_0(x)$ a $J_1(x)$.

3. MECHANISMUS REKURSE

Většina symbolických jazyků dovoluje zapsat funkce (ev. procedury), které volají sebe samotné. Takové rutiny se nazývají rekursivní. V dalším textu budou takové rutiny zapisovány v jazyku Pascal a pro některé z nich jsou odpovídající ekvivalentní zápisy v jazyku C uvedeny v příloze.

Jistě snadno přepíšeme rekursivní algoritmus pro výpočet *n*! jako rekursivní funkci. Uvažujme globální typy

```
type  KLADNACELA    = 1..maxint ;
      NEZAPCELA  =  0..maxint ;
```

a dříve uvedený rekursivní algoritmus zapíšeme následovně :

```
function  FACT ( N : NEZAPCELA ) : KLADNACELA ;
var  X : NEZAPCELA ;
     Y : KLADNACELA ;
begin
      if N = 0
        then FACT := 1
        else  begin
                    X := N - 1 ;
                    Y := FACT ( X ) ;
                    FACT := N * Y
              end
end ;
```

To, že horní mez intervalu pro NEZAPCELA bychom měli specifikovat největším přirozeným číslem, jehož faktoriál je ještě v daných podmínkách zobrazitelný, je zatím z našeho hlediska méně významné. Uvažujme však, že tato rekursivní funkce je volána v jiné rutině např. příkazem

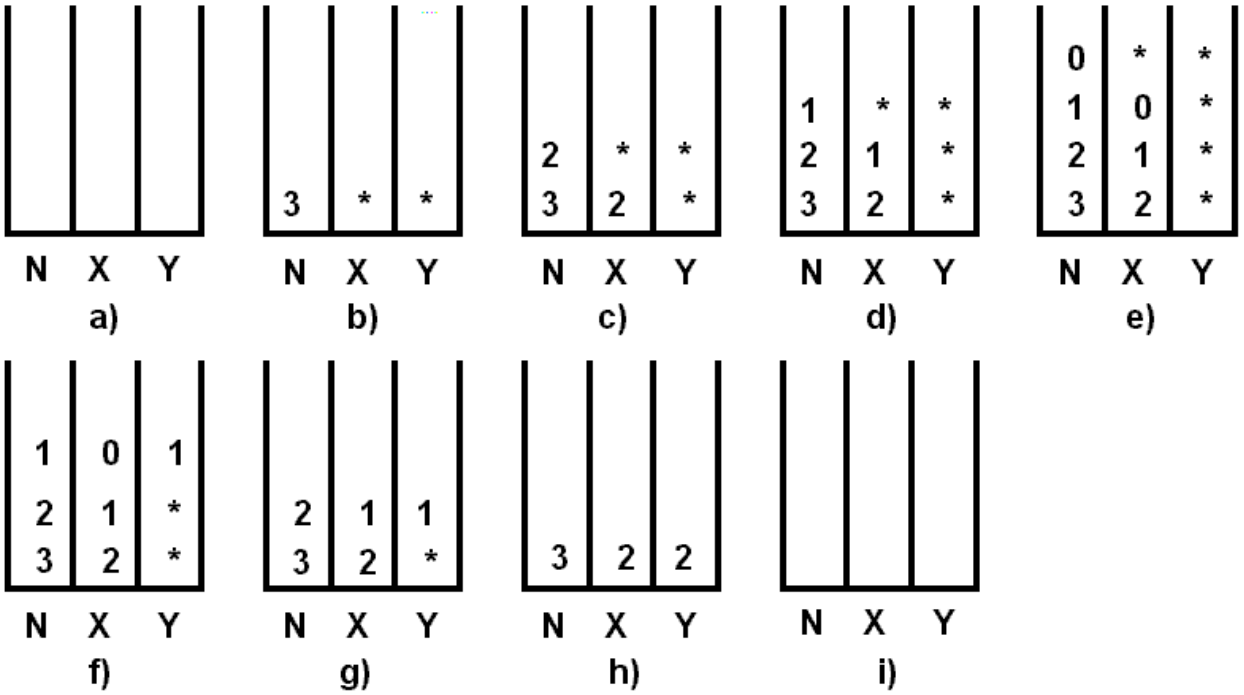
```
write ( FACT ( 3 ) )
```

.

Když volající rutina volá FACT, tak parametru N je přiřazena hodnota 3. Protože však N je různé od 0, tak lokální proměnné X je přiřazena hodnota 2 a pak je volána FACT, ovšem s argumentem 2. Tudíž se znovu opakuje vstup do bloku FACT a lokální proměnné X a Y a parametr N volaný hodnotou jsou znovu alokovány, přičemž jejich původní alokace se zachovává. Tedy pokaždé, když je rekursivně volána funkce FACT, tak se alokuje nová množina lokálních proměnných a parametrů volaných hodnotou a pouze tato nová množina může být referencována uvnitř tohoto volání FACT. Návrat z FACT do bodu předchozího volání uvolňuje

nejnovější alokaci těchto proměnných a aktivizuje předchozí alokaci těchto proměnných. Takový mechanismus je pak realizován pomocí zásobníku. Při každém vstupu do rekursivní rutiny se nová alokace jejích proměnných umístí do zásobníku a každý odkaz na lokální proměnné je přes aktuální vrchol zásobníku. Každý návrat z rekursivní rutiny znamená odběr ze zásobníku, takže aktuální vrchol lze užít pro odkaz na předchozí alokaci.

Pro náš příklad je vše ilustrováno na obr. 1. Inicializace zásobníku je na obr. 1a - zásobník je prázdný. Situaci po prvním volání ilustruje obr. 1b. Proměnné X, Y jsou alokovány, ale zatím nejsou inicializovány (to je na obr. 1b vyznačeno hvězdičkami). Protože N <> 0, tak X se přiřadí hodnota 2 a znovu se volá FACT, ale s argumentem 2 (viz obr. 1c). Dále pak na obr. 1e je zachycena situace po volání FACT s argumentem 0. V tom případě však FACT vrátí do místa volání hodnotu 1, proměnné alokované pro FACT (0) se uvolní a proměnné Y alokované při volání FACT (1) se přiřadí hodnota 1 (viz obr. 1f).



obr. 1

Pak se provede příkaz FACT := N * Y vynásobením hodnot N a Y na vrcholu zásobníku. Tato hodnota (1) se vrátí do místa volání FACT (2), uvolní se alokace při volání FACT (1) a hodnotě Y alokované při volání FACT (2) se přiřadí tato vrácená hodnota (obr. 1g). Analogicky proměnné Y alokované při volání FACT (3) se přiřadí hodnota 2 (obr. 1h). Po vynásobení N a Y na vrcholu zásobníku se uvolní alokace pro FACT (3) a získaná hodnota se vrátí do volající rutiny, kde je příkazem write (FACT (3)) zapsána do souboru output.

Jednodušeji lze rekursivní funkci FACT zapsat ve tvaru :

```
function FACT ( N : NEZAPCELA ) : KLADNACELA ;
begin
        if N < 2 then FACT := 1
                 else  FACT := N * FACT ( N - 1 )
end ;
```

Zde je ovšem rozdíl mezi užitím FACT na levé a pravé straně příkazu přiřazení v části **else** příkazu **if**. Odkaz na FACT na levé straně je odkaz na lokální proměnnou, kdežto odkaz na pravé straně je rekursivní volání funkce FACT. Tím se vyhneme explicitnímu užití lokálních proměnných X (pro hodnotu N-1) a Y (pro hodnotu FACT (X)). Ovšem při každém volání rekursivní funkce FACT je provedena alokace pro tyto dvě hodnoty stejně, jako při explicitním použití lokálních proměnných.

Jiným příkladem je **Fibonacciho posloupnost**, definovaná rekursivně následovně :

$fib(n) = fib(n-1) + fib(n-2)$, pro , $n \geq 2$

$fib(n) = n$, pro n = 0 V 1 .

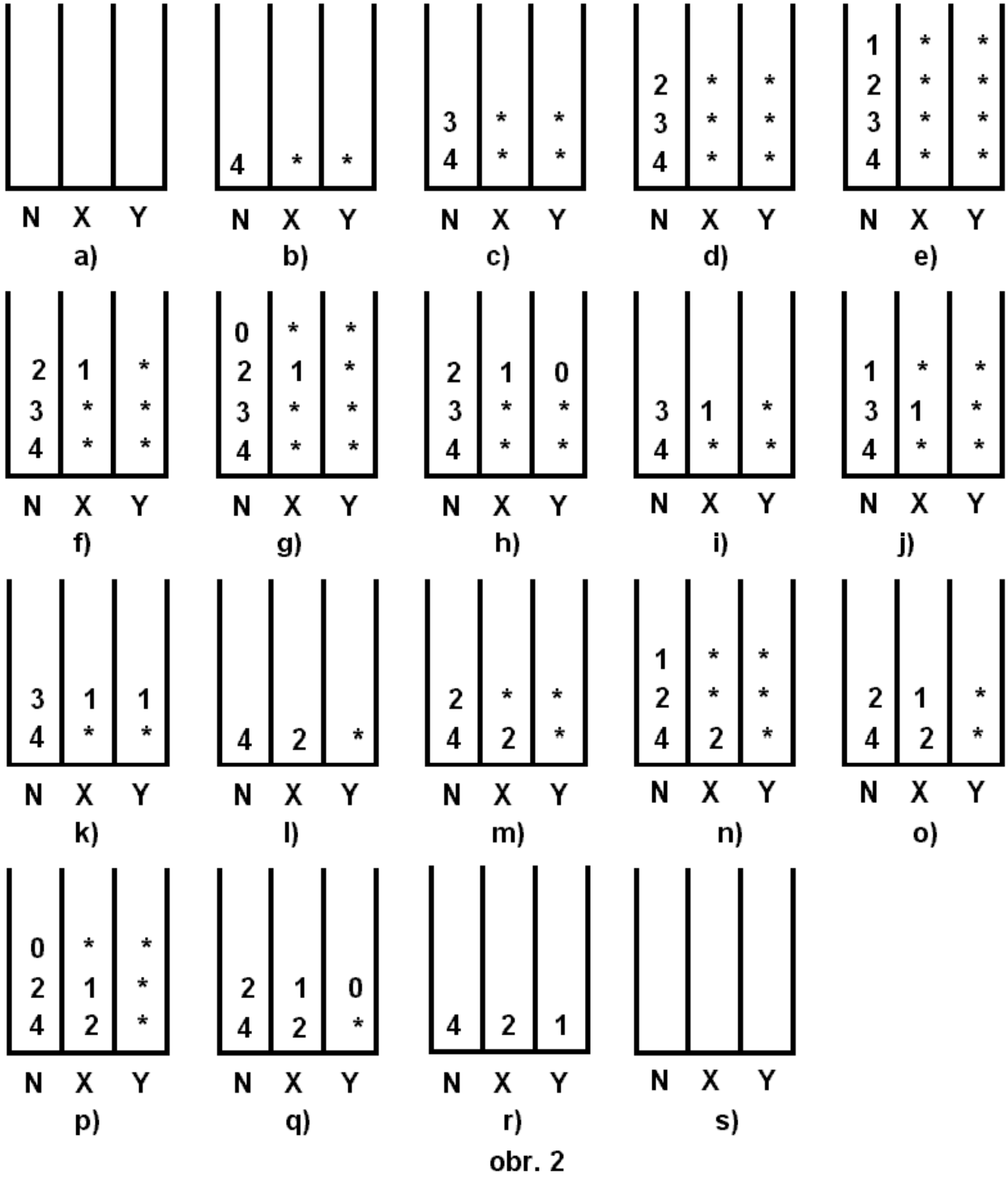
Rekursivní funkci pro určení Fibonacciho čísel můžeme zapsat následovně :

```
function  FIB (  N : NEZAPCELA ) : NEZAPCELA ;
begin
        if  N <= 1  then  FIB := N
                    else   FIB := FIB ( N - 1 ) + FIB ( N - 2 )
end ;
```

Pro snazší pochopení procesu vkládání a odebírání ze zásobníku raději explicitně užijeme lokální proměnné :

```
function  FIB ( N : NEZAPCELA ) : NEZAPCELA ;
var  X, Y : NEZAPCELA ;
begin
        if  N <= 1  then FIB := N
                    else begin  X := FIB ( N - 1 ) ; Y := FIB ( N - 2 ) ;
                                FIB := X + Y
                         end
end ;
```

Kdybychom nyní chtěli postupně ilustrovat operace nad zásobníkem, dostali bychom schéma podle obr. 2.



obr. 2

Čtenář nechť se nyní pokusí sám postupně ilustrovat operace nad zásobníkem v případě, že funkci FIB zapíšeme následovně :

```
function  FIB ( N : NEZAPCELA ) : NEZAPCELA ;
var  X, Y : NEZAPCELA ;
begin
        if  N <= 1  then FIB := N
                    else begin  X := FIB ( N - 2 ) ;
                                Y := FIB ( N - 1 ) ;
                                FIB := X + Y
                         end
end ;
```

Jistě je na první pohled patrné, jak komplikovaně je v uvedených příkladech realizován výpočet na bázi rekursivních algoritmů. To ovšem není možné generalizovat, později uvidíme, jak elegantně a efektivně lze některé úlohy řešit právě pomocí rekursivních rutin. V uvedených dvou příkladech bychom jistě použili iteračních algoritmů. Tak např. Fibonacciho posloupnost je

0 , 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 , ...

a již z definice vidíme, že každé Fibonacciho číslo $(n \geq 2)$ dostaneme sečtením dvou Fibonacciho čísel bezprostředně předcházejících hledanému v dané posloupnosti. Z toho plyne i následující zápis nerekursivní funkce :

```
function  FIB ( N : NEZAPCELA ) : NEZAPCELA ;
var X, MENSI, VETSI, : NEZAPCELA ;
begin
        if  N <= 1
          then   FIB := N
          else begin
                  MENSI := 0 ; VETSI := 1 ;
                  for I := 2 to N do
                    begin
                      X := MENSI ; MENSI := VETSI ;
                      VETSI := X + MENSI
                    end ;
                  FIB := VETSI
               end
end ;
```

Vraťme se nyní k dříve popsanému rekursivnímu algoritmu pro binární vyhledávání a zapišme rekursivní funkci, která nám pro vyhledávaný prvek X v poli se složkami typu integer od

A[DOLNI] do A[HORNI] vrátí jeho index v poli (resp. 0, pokud v daném úseku pole prvek není). Uvažujme nejprve tyto globální definice :

```
const   MAX = 500 ;
        MAXPLUS1 = 501 ;
type    POLE = array [1..MAX]  of  integer ;
        INDEX = 0..MAXPLUS1 ;
```

Rekursivní funkci pak lze zapsat následovně :

```
function BINVYH( A:POLE ; X:integer ; DOLNI, HORNI:INDEX ) : INDEX ;
var  STR : INDEX ;
begin
   if  DOLNI > HORNI
      then  BINVYH := 0
      else  begin
               STR := ( DOLNI + HORNI )  div 2 ;
               if  X = A[STR]
                 then  BINVYH := STR
                 else  if  X < A[STR]
                          then  BINVYH := BINVYH ( A, X, DOLNI, STR - 1 )
                          else   BINVYH := BINVYH ( A, X, SR + 1, HORNI )
            end
end ;
```

Parametry DOLNI, HORNI jsou intervaly z typu integer od 0 do MAXPLUS1, protože když X < A[1], tak při posledním rekursivním volání bude mít HORNI hodnotu 0, nebo naopak když X > A[MAX], tak při posledním rekursivním volání bude mít DOLNI hodnotu MAXPLUS1.

Pokaždé, když je volána rekursivní funkce BINVYH, tak se do zásobníku ukládají též všechny hodnoty předávané formálním parametrům, tudíž i hodnoty složek pole A a parametru X. Ovšem hodnoty A i X se během výpočtu nemění, a je proto zcela zbytečné "kopírovat" stále stejné hodnoty při každém volání rekursivní funkce. Řešení tohoto nedostatku je možné provést dvojím způsobem. První možnost spočívá v zavedení globálních proměnných A a X, deklarovaných v bloku, kde je funkce BINVYH jako subblok (použití externích proměnných je běžné v jazyku C). Pak deklarační část bude obsahovat deklarace :

```
var     A : POLE ;
        X : integer ;
```

```
function  BINVYH ( DOLNI, HORNI : INDEX ) : INDEX ;
```

a v operační části tohoto bloku lze zapsat např. příkaz

```
I := BINVYH ( 2, MAX-1 )
```

Především z formálních a estetických důvodů je však lepší v Pascalu zapsat funkci BINVYH jako nerekursivní, která volá interní rekursivní funkci POMVYH se 2 parametry vymezujícími úsek pole pro vyhledávání :

```
function BINVYH ( var A : POLE ; X : integer ; DOLNI, HORNI : INDEX )
                        : INDEX ;
  function POMVYH ( D, H : INDEX ) : INDEX ;
  var STR : INDEX ;
  begin  { POMVYH  }
    if  D > H
      then  POMVYH := 0
      else  begin
              STR := ( D + H )  div  2 ;
              if  X = A[STR]
                then  POMVYH := STR
                else  if  X < A[STR]
                        then  POMVYH := POMVYH ( D, STR-1 )
                        else   POMVYH := POMVYH ( STR+1, H )
            end
  end  { POMVYH } ;
begin  { BINVYH  }
  BINVYH := POMVYH ( DOLNI, HORNI )
end  {  BINVYH  } ;
```

Uvedenou rekursivní verzi snadno přepíšeme na nerekursivní tím, že meze úseku pole, ve kterých se provádí vyhledávání, vyjádříme explicitně :

```
function  BINVYH ( var A : POLE ; X : integer ; DOLNI, HORNI : INDEX )
                        : INDEX ;
var  NALEZ : boolean ;
     STR : INDEX :
begin
  NALEZ := false ;
  while  not NALEZ and  ( DOLNI <= HORNI )  do
    begin
      STR := ( DOLNI + HORNI )  div 2 ;
```

```
        if  X = A[STR]
          then  begin
                    NALEZ := true ; BINVYH := STR
                end
           else   begin
                    if  X < A[STR]
                      then  HORNI := STR - 1  else  DOLNI := STR + 1
                end
      end  ;
    if  not NALEZ then  BINVYH := 0
  end  ;
```

Až dosud jsme se zabývali případy, kdy nějaká procedura nebo funkce volala sebe samotnou. V takových případech pak mluvíme o tzv. **přímé rekursi**. K rekursi však může dojít i nepřímo, když např. nějaká procedura A volá jinou proceduru B, která volá proceduru A :

```
procedure A ( formální parametry ) ;
    begin
        .
        .
        .
        B ( argumenty ) ;
        .
        .
        .
end ;
```

```
procedure B ( formální parametry ) ;
begin
        .
        .
        .
        A ( argumenty ) ;
        .
        .
        .
end  ;
```

Procedura A tedy volá jinou proceduru B a prověřujeme-li proceduru A samotnou, tak nelze určit, že bude volat sebe samotnou nepřímo (prostřednictvím procedury B). Na **nepřímé rekursi** se může podílet více rutin, např. A volá B, B volá C, C volá D a D volá A. Pak každá z rutin A, B, C a D může nepřímo volat sebe samotnou a je tedy rekursivní. Místo nepřímá se používá i termínu **vzájemná rekurse**.

Pro nepřímou rekursi platí v jazyku Pascal zvláštní pravidlo, neboť procedura nebo funkce musí být všeobecně deklarována dříve, než je na ni odkazováno (v jazyku C řešíme tuto záležitost pomocí prototypů funkcí zcela běžně). V našem příkladě však A nelze deklarovat dříve než B, protože v A je odkaz na B a také B nelze deklarovat dříve než A, neboť B obsahuje odkaz na A. V jazyku Pascal se tento rozpor řeší předsunutou deklarací s direktivou **forward** :

```
procedure  A ( formální parametry ) ; forward ;
procedure  B  ( formální parametry ) ;
        {  blok  B  }   ;
procedure  A ;
        {  blok  A  } ;
```

Při zápisu bloku A nejsou formální parametry v záhlaví A předeklarovány.

Jako příklad nepřímé rekurse lze uvést **rekursivní definici algebraických výrazů**. Tato zjednodušená definice nechť je naší konvencí a čtenář nechť ji porovná s odpovídajícími syntaktickými diagramy jazyka Pascal :

1. **Výraz** je **term** následovaný **znakem plus**, za kterým následuje **term**, nebo je to **term samotný**.
2. **Term** je **faktor** následovaný **znakem hvězdička**, za kterým následuje **faktor**, nebo je to **faktor samotný**.
3. **Faktor** je buď **písmeno** nebo **výraz** uzavřený v okrouhlých **závorkách**.

Vidíme, že výraz je definován pomocí termu, term pomocí faktoru a faktor pomocí výrazu. Tudíž úplná množina definic tvoří nepřímou rekursi.

Tak např. A, B, C, ... jsou faktory. Protože term může být faktor samotný, jsou to rovněž termy, a tudíž jsou to také výrazy. Protože A je výraz, tak (A) je faktor, a tudíž jak term, tak i výraz. Ovšem např. A+B je výraz, ale není to ani term, ani faktor. Naproti tomu (A+B) je faktor, term i výraz. A*B je term, tudíž i výraz, ale není to faktor. A*B+C je výraz, nikoliv term nebo faktor. A*(B+C) je term i výraz, ale není to faktor. To byly příklady správných zápisů s ohledem na naši definici. Správné však z hlediska naší definice nejsou zápisy reprezentované těmito řetězci znaků :

A+*B , A+B+C , (A+B*)C .

Napíšeme nyní program, který přečte a vytiskne řetězec znaků a podá zprávu, zda tento řetězec reprezentuje správný či nesprávný algebraický výraz (dle naší definice). Použijeme pomocnou funkci VEMZNAK, která používá globální proměnné RETEZ, DELKA a POS. Složky pole RETEZ obsahují vstupní posloupnost znaků délky DELKA, zatímco POS ukazuje na složku pole RETEZ, kterou pro vrací funkce VEMZNAK, zatímco pro POS > DELKA vrací tato funkce prázdný znak.

Funkce VYRAZ, TERM a FAKOR jsou rekursivní a realizují výpočet podle dříve uvedené definice algebraických výrazů. Používají globální proměnné RETEZ a POS. Vrátí-li funkce VYRAZ hodnotu true, tak hodnota proměnné POS ukazuje na tu složku pole RETEZ, která s předcházejícími složkami pole RETEZ tvoří správný zápis algebraického výrazu. Aby byl celý zápis správný, musí být také POS = DELKA. To lze demonstrovat na příkladech :

RETEZ	DELKA	POS	VYRAZ
(A)	3	3	true
A*B+C	5	5	true
A+*B	4	3	false
(A+B*)C	7	6	false
A+B+C	5	3	true
A+B*CD	6	5	true

Tak, jak je zapsána procedura CTIRETEZ, nemůže být v zápisu výrazu prázdný znak. Čtenář nechť si přepíše dále uvedený program tak, že zápis každého výrazu bude ukončen znakem středník a mezery v zápisu výrazu jsou bezvýznamné.

```
program  KONTROLA ( input , output ) ;
const    MAX = 80 ;
         MAXPLUS1 = 81 ;
type     POLEZNAKU = packed array [1..MAX] of char ;
         INDEX = 0..MAX ;
var      RETEZ : POLEZNAKU ;
         DELKA, I : INDEX ;
         POS : 0..MAXPLUS1 ;
         OK : boolean ;
function VEMZNAK : char ;
begin  { VEMZNAK  }
   POS := POS + 1 ;
   if POS > DELKA then  VEMZNAK := ' ' else  VEMZNAK := RETEZ[POS]
end { VEMZNAK } ;
function FAKTOR : boolean ; forward ;
function  TERM : boolean ; forward ;
function  VYRAZ : boolean ;
var  OK : boolean ;
```

```
     C : char ;
begin  { VYRAZ  }
   { Najdi term : } OK := TERM ;
   if not OK
        then { neex. výraz  }  VYRAZ := false
        else  begin
                 { Prohlédni další symbol : }
                 C := VEMZNAK ;
                 if  C <> '+'
                    then  { Byl nalezen nejdelší výraz, vrať POS na poslední znak
                             výrazu : } begin  POS := POS - 1 ; VYRAZ := true  end
                    else   { Nalezen term následovaný znakem +, najdi jiný term }
                           begin
                              OK := TERM ;
                              if OK then VYRAZ := true else VYRAZ := false
                           end
              end
end   { VYRAZ  }  ;

function  TERM ;
var  OK : boolean ;
     C :char ;
begin    {  TERM  }
   OK := FAKTOR ;
   if  not OK
      then  TERM := false
      else  begin
                 C := VEMZNAK ;
                 if  C <> '*'
                    then begin  POS := POS - 1 ; TERM := true  end
                    else begin
                              OK := FAKTOR ;
                              if OK then TERM := true  else  TERM := false
                           end
            end
end  {  TERM  } ;
```

```
function FAKTOR ;
var  OK : boolean ;
     C : char ;
 function PISMENO ( C : char ) : boolean ;
 begin  { PISMENO  }
        if  ( C >= 'A' )  and  ( C <= 'Z' )
            then PISMENO := true  else PISMENO := false
 end  {  PISMENO  } ;
begin  { FAKTOR  }
  C := VEMZNAK ;
  if  C <> '('
     then { Kontrola, zda následuje písmeno }
          if PISMENO ( C )  then  FAKTOR := true  else  FAKTOR := false
     else  {  Faktor může být výraz  }
           begin
              OK := VYRAZ ;
              if  not OK then  FAKTOR := false
                          else  begin
                                  C := VEMZNAK ;
                                  if  C <> ')'
                                     then FAKTOR := false else FAKTOR := true
                                end
           end
end  {  FAKTOR  }  ;

procedure  CTIRETEZ ( var RETEZ : POLEZNAKU ; var DELKA : INDEX ) ;
var  I : INDEX ;
begin   {  CTIRETEZ  }
  I := 0 ;
  while  ( I < MAX ) and  ( input    <> ' ' ) do
     begin
       I := I + 1 ; RETEZ[I] := input              ; get ( input )
     end ;
  DELKA := I
end  { CTIRETEZ  } ;
```

```
begin  { KONTROLA  }
   CTIRETEZ ( RETEZ, DELKA ) ;
   for I := 1 to DELKA do write ( RETEZ[I] ) ;
   writeln ;
   POS := 0 ; OK := VYRAZ :
   write ( 'Vyraz je  ' ) ;
   if OK and ( POS = DELKA ) then write ('spravny' ) else write ( 'nespravny' ) ;
   write ( '      POS=', POS :3, '  OK=', OK ) ;
end   { KONTROLA } .
```

Poznámka : U rekursivních podprogramů je třeba velmi obezřetně používat formální parametry volané odkazem (v Pascalu). Tak např. následující zápis funkce FACT je chybný již proto, že jako argument odpovídající formálnímu parametru volanému odkazem je výraz N-1 :

```
function FACT ( var N : NEZAPCELA ) : KLADNACELA ;
begin
      if  N  <  2  then  FACT := 1
                  else   FACT := FACT ( N - 1 ) * N
end  ;
```

Následující zápis je sice formálně správný, nicméně v tomto případě nám bude funkce místo hodnoty N! vracet hodnotu 2^{N-1} :

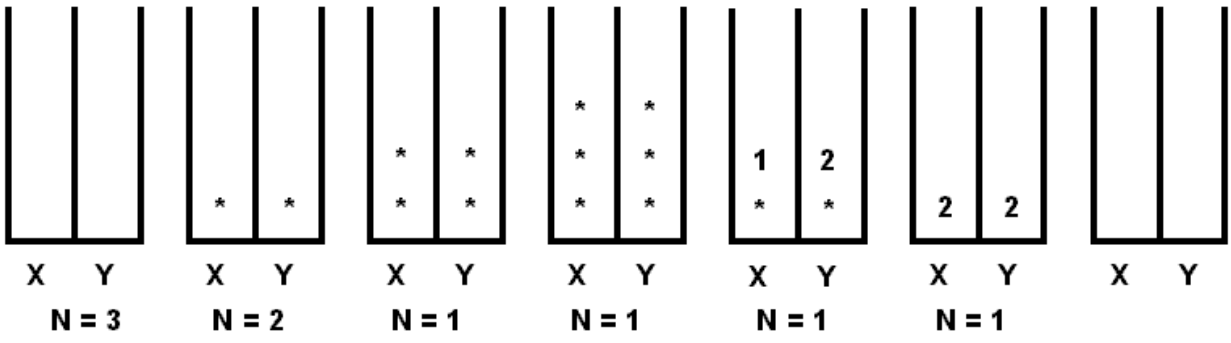
```
function  FACT ( var N : NEZAPCELA ) : KLADNACELA ;
begin
      if  N  <  2  then  FACT := 1
                 else  begin
                          N := N - 1 ; FACT := FACT ( N ) * ( N + 1 )
                       end
end  ;
```

Je třeba si uvědomit, že tato verze funkce FACT používá N v podstatě jako globální proměnnou, jejíž hodnotu mění na 1 a tudíž nějaké lokální proměnné reprezentující výraz (N+1) je po návratu z rekursivních volání přiřazována hodnota 2. Vyjádřeme explicitně lokální proměnné pro tento případ :

```
function  FCAT  ( var N : NEZAPCELA) : KLADNACELA ;
var  X, Y : KLADNACELA ;
```

```
begin
        if  N  <  2  then  FACT := 1
                    else  begin
                              N := N - 1 ;
                              X := FACT ( N ) ;
                              Y := N + 1 ;
                               FACT := X * Y
                          end
end ;
```

Pro dříve uvedený příklad write (FACT (3)) pak můžeme schematicky znázornit operace nad zásobníkem podle obr. 3.



obr. 3

Z toho je zřejmé, že bychom potřebovali odkládat do zásobníku hodnoty výrazu (N+1), čehož však můžeme dosáhnout záměnou pořadí operandů v operaci násobení. Tudíž následující verze rekursivní funkce FACT by měla správný efekt, neboť do místa volání vrací N! :

```
function  FACT ( var  N : NEZAPCELA ) : KLADNACELA ;
begin
        if  N  < 2  then  FACT := 1
                    else  begin
                              N := N -1 ; FACT := ( N + 1 ) * FACT ( N )
                          end
end  ;
```

Není ovšem vyloučeno, že takovou záměnu pořadí operandů by mohly vykonat i kompilátory ve fázi optimalizace vnitřní formy programu.

Cvičení

1. Uvažujte následující rekursivní funkci v jazyku Pascal :

```
function  FUNC ( N : NEZAPCELA ) : NEZAPCELA ;
begin
        if  N = 0  then  FUNC := 0
                   else   FUNC := N + FUNC ( N -1 )
end  ;
```

Určete, jakou hodnotu bude funkce vracet a napište stejnou funkci v iterační verzi.

2. Euklidův algoritmus pro určení největšího společného dělitele dvou celých kladných čísel lze zapsat ve tvaru :

$$HCF(m,n) = [(n > m) \rightarrow HCF(n,m), (n = 0) \rightarrow m, HCF(n, m \bmod n)]$$

Napište rekursivní funkci pro výpočet *HCF*.

3. Pro nalezení maximálního prvku pole A od indexu D do H napište nerekursivní funkci, která bude volat pomocnou interní rekursivní funkci (viz cvičení 2 v předchozí kapitole).

4. Zobecněná Fibonacciho posloupnost s argumenty f_0 a f_1 je posloupnost čísel $fib_Z(f_0,f_1,0)$, $fib_Z(f_0,f_1,1)$, $fib_Z(f_0,f_1,2)$, ..., kde

$fib_Z(f_0,f_1,n) = fib_Z(f_0,f_1,n\text{-}1) + fib_Z(f_0,f_1,n\text{-}2)$, $n > 1$,

$fib_Z(f_0,f_1,n) = f_0$, $n = 0$,

$fib_Z(f_0,f_1,n) = f_1$, $n = 1$.

Pro výpočet $fib_Z(f_0,f_1,n)$ napište funkci a to jak v rekursivní, tak i iterační verzi.

5. Ackermannova funkce je definována následovně :

$$A(m,n) = [(m = 0) \rightarrow n+1, (n = 0) \rightarrow A(m-1,1), A(m-1, A(m,n-1))].$$

a) Z definice určete ručně hodnotu $A(2,2)$ - výsledek je 7.

b) Napište rekursivní funkci pro výpočet hodnot $A(m,n)$.

c) Již pro malá m, n je rekursivní algoritmus tak pomalý, že je prakticky nepoužitelný (vyzkoušejte na $A(4,4)$). Dokážete nalézt iterační algoritmus pro určení $A(m,n)$?

6. Modifikujte v textu uvedený program KONTROLA tak, že zápis každého výrazu bude ukončen znakem středník a mezery v zápisu výrazu budou nevýznamné.

7. Ve vstupním souboru input jsou zapsány hodnoty typu integer. Napište rekursivní rutinu, která přečte tyto hodnoty ze souboru input a zapíše do souboru output tytéž hodnoty, ale v opačném pořadí.

4. POUŽITÍ REKURSE

Zřejmě není obtížné k rekursivně definovanému algoritmu vytvořit odpovídající rekursivní rutinu. Mnohem obtížnější je však nalézt taková řešení, když je specifikován pouze problém. Rekursivní definice např. funkce faktoriál je všeobecně známa, ovšem iterační řešení tohoto problému se nabízí samo a bude efektivnější. Naopak jiné nerekursivně definované problémy lze elegantně řešit jen pomocí rekursivních technik. V jejich principu je redukce nějakého "komplexního" problému na řešení téhož jednoduššího problému. Tato redukce vede k řešení problému, které je dáno a priori.

Dále uvedené příklady demonstrují výhody rekursivních technik při řešení relativně složitých problémů.

4.1 Problém Hanojských věží

Problém Hanojských věží patří mezi klasické matematické problémy. Není specifikován rekursivně, ale právě pomocí rekursivní techniky lze tento problém neobyčejně elegantně řešit.

Máme k dispozici 3 jehly A, B a C a *n* disků různého průměru, které lze na jehly nasadit. Výchozí situaci znázorňuje obr. 4a - disky jsou navlečeny na jedné jehle tak, že tvoří věž, která se směrem vzhůru zužuje. Úkolem je přemístit disky na druhou jehlu (např. C) s použitím třetí pomocné jehly (např. B), přičemž při přemístění se musí dodržet :

- ☞ v každém kroku lze přemístit pouze jeden disk (z vrcholu věže) a to vždy jen z jehly na jehlu (disky nelze odkládat mimo jehly) ,
- ☞ nelze položit disk většího průměru na disk menšího průměru.

Nechť v našem příkladě podle obr. 4 chceme přemístit 5 disků z jehly A na jehlu C (při dodržení výše uvedených podmínek) a předpokládejme, že známe toto řešení pro 4 disky. Tím ovšem známe i řešení, jak přemístit 4 disky z vrcholu věže na jehle A na jehlu B s využitím jehly C jako pomocné jehly (výsledek znázorňuje obr. 4b). Pak ovšem můžeme disk největšího průměru přemístit z jehly A na prázdnou jehlu C (obr. 4c) a konečně s využitím znalosti řešení pro 4 disky přemístit 4 disky z jehly B na jehlu C použitím jehly A jako pomocné jehly (viz obr. 4d).

Obecně tedy řešení pro *n* disků budeme formulovat ve vztahu k řešení pro *n*-1 disků, přčemž pro *n* = 1 je řešení triviální. Rekursivní řešení problému Hanojských věží lze tudíž popsat následovně :

```
if n = 1
        then "Přemístění disku z jehly A na jehlu C"
```

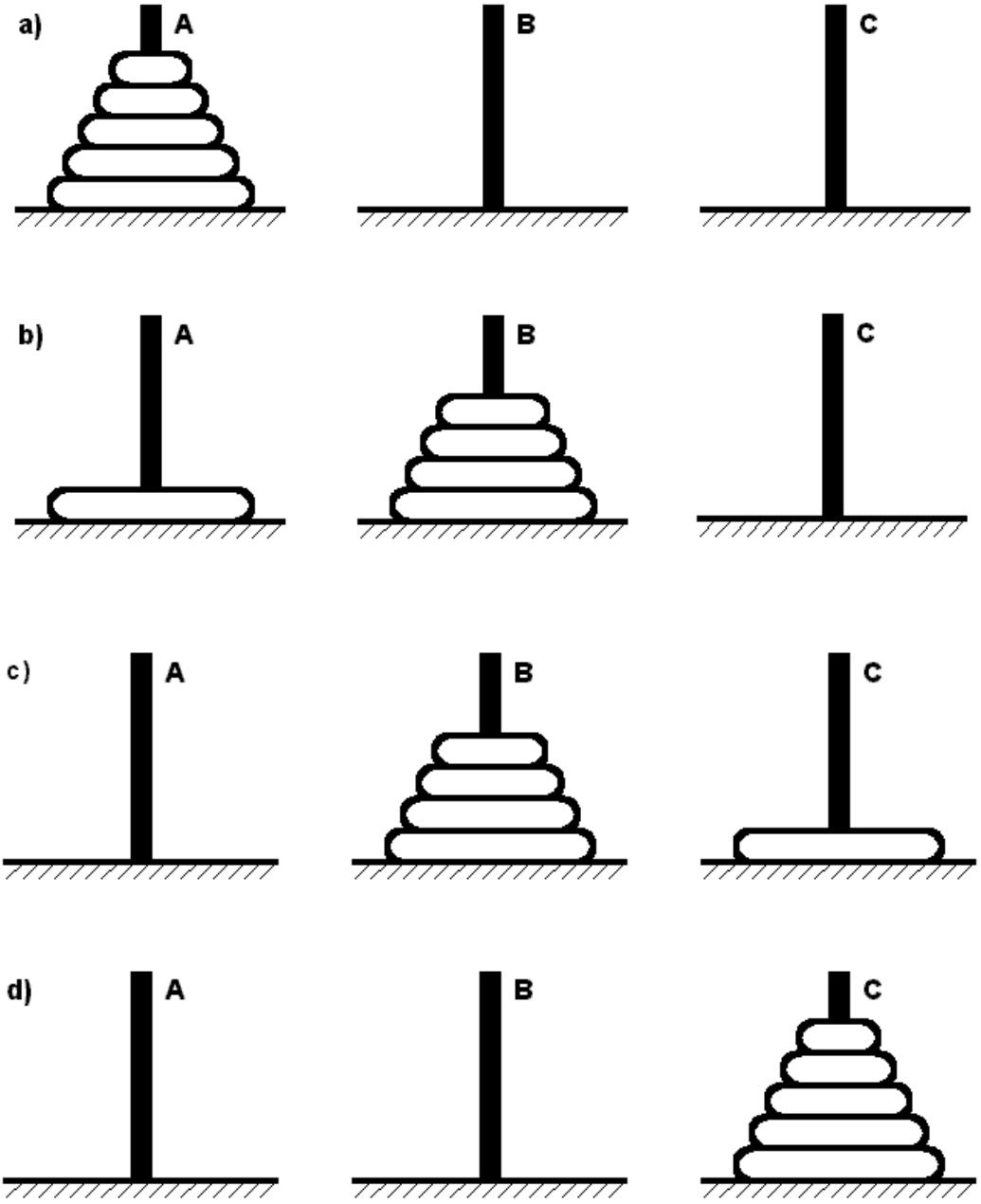
else begin

"Přemístění *n*-1 disků z jehly A na jehlu B užitím pomocné jehly C";

"Přemístění zbývajícího disku z jehly A na jehlu C" ;

"Přemístění *n*-1 disků z jehly B na jehlu C užitím pomocné jehly A";

end



obr. 4

Zřejmě již není velkým problémem zapsat uvedený algoritmus jako rekursivní podprogram. Nicméně však problém Hanojských věží musíme zadat konkrétně s ohledem na

požadované vstupy a výstupy. Mohli bychom např. požadovat, aby jednotlivé disky byly označeny barvami "bílý", "červený", modrý" atd., nebo jehly místo písmen jejich vzájemnou polohou "vlevo", "uprostřed", "vpravo" apod. To vše je však věcí konvence. Stačí, když na vstupu zadáme počet disků *n* a výstup budeme požadovaným způsobem konvertovat. Tak např. ponecháme označení jehel jako v uvedeném příkladě A, B a C a jednotlivé disky označíme čísly 1, 2, 3, ..., *n* tak, že disk nejmenšího průměru má číslo 1, zatímco disk největšího průměru číslo *n*. Napíšeme potom následující program, za nimž je patrný tvar výstupu pro *n* = 5.

```
program VEZE ( input, output ) ;
type  POCETDISKU = 1..maxint ;
var   N : POCETDISKU ;

procedure HANOJSKEVEZE ( N : POCETDISKU ;
                                 ODKUD, KAM, POMOCI : char ) ;
if  N = 1
 then  writeln ( 'Prenes disk', N:3, ' z jehly', ODKUD:2, ' na jehlu', KAM:2 ) ;
 else  begin
         HANOJSKEVEZE ( N-1, ODKUD, POMOCI, KAM ) ;
         writeln ( 'Prenes disk', N:3, ' z jehly', ODKUD:2, ' na jehlu', KAM:2 ) ;
         HANOJSKEVEZE ( N-1, POMOCI, KAM, ODKUD ) ;
       end
end ;

begin  { VEZE }
       read (N) ; HANOJSKEVEZE ( N, 'A', 'C', 'B' )
end   { VEZE } .
```

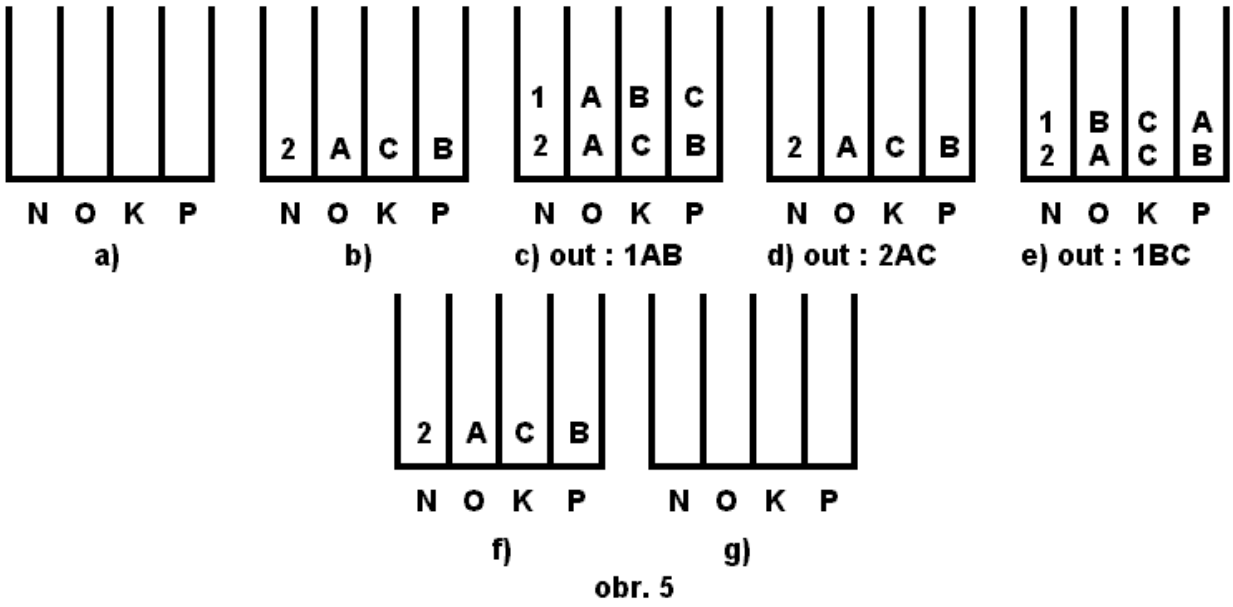
```
Prenes disk 1 z jehly A na jehlu C
Prenes disk 2 z jehly A na jehlu B
Prenes disk 1 z jehly C na jehlu B
Prenes disk 3 z jehly A na jehlu C
Prenes disk 1 z jehly B na jehlu A
Prenes disk 2 z jehly B na jehlu C
Prenes disk 1 z jehly A na jehlu C
Prenes disk 4 z jehly A na jehlu B
Prenes disk 1 z jehly C na jehlu B
Prenes disk 2 z jehly C na jehlu A
```

Prenes disk 1 z jehly B na jehlu A
Prenes disk 3 z jehly C na jehlu B
Prenes disk 1 z jehly A na jehlu C
Prenes disk 2 z jehly A na jehlu B
Prenes disk 1 z jehly C na jehlu B
Prenes disk 5 z jehly A na jehlu C
Prenes disk 1 z jehly B na jehlu A
Prenes disk 2 z jehly B na jehlu C
Prenes disk 1 z jehly A na jehlu C
Prenes disk 3 z jehly B na jehlu A
Prenes disk 1 z jehly C na jehlu B
Prenes disk 2 z jehly C na jehlu A
Prenes disk 1 z jehly B na jehlu A
Prenes disk 4 z jehly B na jehlu C
Prenes disk 1 z jehly A na jehlu C
Prenes disk 2 z jehly A na jehlu B
Prenes disk 1 z jehly C na jehlu B
Prenes disk 3 z jehly A na jehlu C
Prenes disk 1 z jehly B na jehlu A
Prenes disk 2 z jehly B na jehlu C
Prenes disk 1 z jehly A na jehlu C

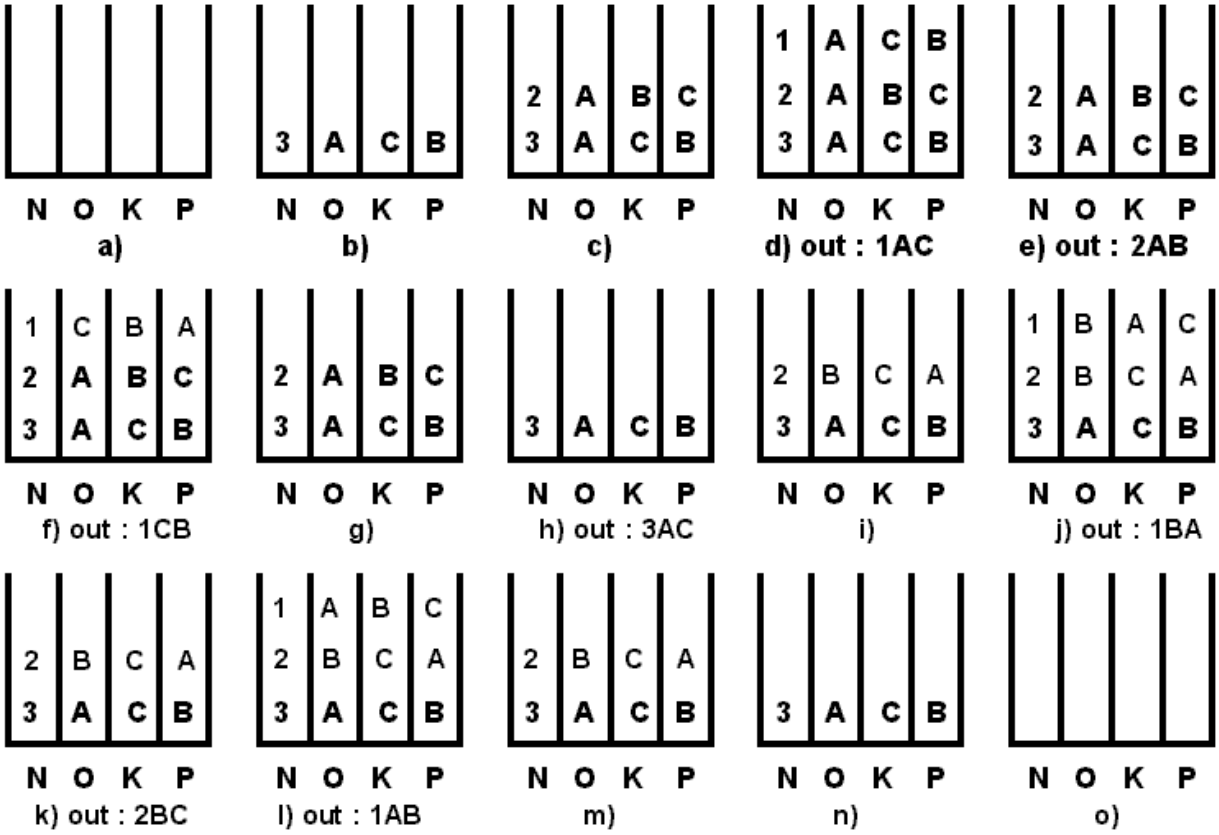
Předpokládejme nyní, že nějaká rutina volá uvedenou proceduru příkazem

HANOJSKEVEZE(2, 'A', 'C', 'B').

Průběh výpočtu pak můžeme ilustrovat podle obr. 5 (O je zkratka pro ODKUD, K pro KAM, P pro POMOCI, údaje za out: znamenají zápis integeru a dvou znaků do souboru output). Pro složitější příklad HANOJSKEVEZE(3,'A','C','B') je analogické schéma na obr.6.


N O K P
a)
2 A C B
N O K P
b)
1 A B C
2 A C B
N O K P
c) out : 1AB
2 A C B
N O K P
d) out : 2AC
1 B C A
2 A C B
N O K P
e) out : 1BC
2 A C B
N O K P
f)
N O K P
g)

obr. 5


N O K P
a)
3 A C B
N O K P
b)
2 A B C
3 A C B
N O K P
c)
1 A C B
2 A B C
3 A C B
N O K P
d) out : 1AC
2 A B C
3 A C B
N O K P
e) out : 2AB
1 C B A
2 A B C
3 A C B
N O K P
f) out : 1CB
2 A B C
3 A C B
N O K P
g)
3 A C B
N O K P
h) out : 3AC
2 B C A
3 A C B
N O K P
i)
1 B A C
2 B C A
3 A C B
N O K P
j) out : 1BA
2 B C A
3 A C B
N O K P
k) out : 2BC
1 A B C
2 B C A
3 A C B
N O K P
l) out : 1AB
2 B C A
3 A C B
N O K P
m)
3 A C B
N O K P
n)
N O K P
o)

obr. 6

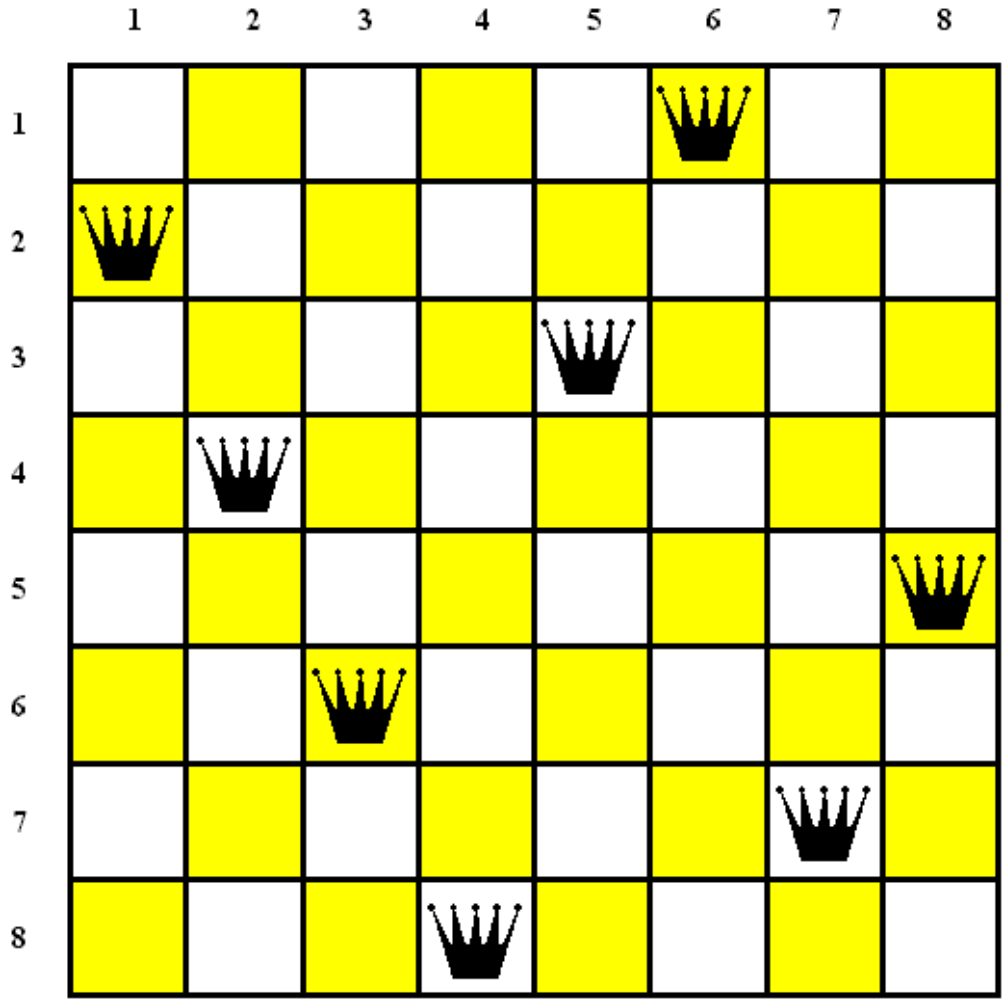
Je též pravda, že přenášet na výstup nějaké označení disků není vůbec nutné. Stačí, když víme z jaké jehly se má disk přemístit a tím je pak i dáno, o jaký disk jde (výsledkem řešení v předchozích krocích, resp. inicializovaným stavem). Tak např. přiřaďme jehlám místo písmen A, B a C čísla 1, 2 a 3, ale výstup konvertujme tak, že označení jehel bude "vlevo", "uprostřed" a "vpravo". Označení disků nebudeme na výstup přenášet. Dále je pak uvedena jedna z možností úpravy programu VEZE :

```
program  VEZE ( input, output ) ;
type  POCETDISKU = 1..maxint ;
      JEHLA = 1..3 ;
var   N : POCETDISKU ;

procedure  HANOJSKEVEZE ( N : POCETDISKU;
                          ODKUD, KAM, POMOCI : JEHLA ) ;

    procedure PRENES1DISK  ;
        procedure  TISKJEHLY ( JAKE : JEHLA ) ;
        begin  { TISKJEHLY }
          case  JAKE  of
              1 : write ( '"vlevo"':11 ) ;
              2 : write ( '"uprostred"' ) ;
              3 : write ( '"vpravo"':11 )
          end
        end  { TISKJEHLY  ) ;
    begin  { PRENES1DISK }
        write ( 'Prenes disk z jehly, ktera je  ' ) ;
        TISKJEHLY ( ODKUD ) ;
        write ( ' na jehlu, ktera je' ) ;
        TISKJEHLY ( KAM ) ;
        writeln
    end  { PRENES1DISK } ;
begin  { HANOJSKEVEZE }
    if  N = 1
        then  PRENES1DISK
        else  begin
              HANOJSKEVEZE ( N - 1, ODKUD, POMOCI, KAM ) ;
              PRENES1DISK ;
```

```
                    HANOJSKEVEZE ( N - 1, POMOCI, KAM, ODKUD )
                end
    end  { HANOJSKEVEZE } ;
    begin  { VEZE }
            read ( N ) ; HANOJSKEVEZE ( N, 1, 3, 2 )
    end  { VEZE } .
```

4.2 Problém osmi dam

Problém osmi dam je definován následujícím způsobem. Na šachovnici je třeba rozmístit osm dam tak, aby žádná dáma neohrožovala nějakou jinou dámu. Jedno z možných řešení je uvedeno na obr. 7.



obr. 7

Podle pravidel šachové hry dáma ohrožuje figury, které se nacházejí na stejném řádku, sloupci a diagonálách šachovnice. Dá se ukázat, že celkem existuje 92 možných řešení daného problému, pokud však zohledníme symetrie řešení, tak existuje 12 zásadně rozdílných řešení. Např. z obr. 7 lze snadno odvodit další symetrická řešení. Na rozdíl od běžně používaného označování šachových polí (např. sloupce *a* až *h*) je zde užito "nestandardního" označení, které bude vyhovovat z hlediska dalších potřeb výpočtu.

Dále uvedený algoritmus patří do skupiny algoritmů **prohledávání s návratem** a v hrubých rysech jej lze zapsat takto :

```
1.  begin  { Umístění i-té dámy }
2.     "Inicializace pozic pro umístění i-té dámy" ;
3.     repeat
4.        "Vyber další pozici" ;
5 .       if "Vybraná pozice přijatelná"
6.           then begin "Umísti dámu na vybranou pozici"
7.                      if "Nejsou umístěny všechny dámy"
8.                         then begin "Umístění (i+1) dámy"
9.                                    if "Umístění (i+1) dámy bylo neúspěšné"
10.                                      then "Zruš umístění i-té dámy"
11.                                end
12.                end
13.    until "Všechny pozice jsou vyčerpány" or "Umístění (i+1) dámy bylo úspěšné"
14  end { algoritmu }
```

Rekursivní volání algoritmu je v řádku 8.

V každém sloupci (ale i řádku a diagonále) může být pouze jedna dáma. Proto volbu pozice *i*-té dámy lze omezit na *i*-tý sloupec, přičemž druhý index *j* bude označovat řádek a bude sloužit pro výběr jedné z osmi možných pozic. Abychom si usnadnili kontrolu, zda vybraná pozice je přijatelná (řádek 5), tak použijeme tyto deklarace :

```
var   X : array [1..8] of integer ;
      R : array [1..8] of boolean ;
      D1: array [2..16] of boolean ;
      D2: array [-7..7] of boolean ;
```

kde znamená :

X[I]	-	pozici dámy v *i*-tém sloupci ,
R[J]	-	true, není-li v *j*-tém řádku žádná dáma,
D1[K]	-	true, není-li žádná dáma na *k*-té diagonále s orientací / ,
D2[K]	-	true, není-li žádná dáma na *k*-té diagonále s orientací \ ,

Na všech diagonálách D1 jsou konstantní součty souřadnic $i + j$, zatímco na všech diagonálách D2 jsou konstantní rozdíly souřadnic $i - j$. Tak např. podle obr. 7 je výsledný vektor řešení $\vec{X}$ = (2, 4, 6, 8, 3, 1, 7, 5), D1[10] = true (tedy na této diagonále není žádná dáma), D2[-5] = false, protože X[6] = 1 atd. Pak logický výraz "Vybraná pozice přijatelná" zapíšeme

R[J] **and** D1[I+J] **and** D2[I-J] ,

příkaz "Umísti dámu na vybranou pozici" sekvencí příkazů

X[I] := J ; R[J] := false ; D1[I+J] := false ; D2[I-J] := false

a příkaz "Zruš místění *i*-té dámy" sekvencí příkazů

R[J] := true ; D1[I+J] := true ; D2[I-J] := true .

Logické výrazy "Umístění (*i*+1) dámy bylo úspěšné", "Umístění (*i*+1) dámy bylo neúspěšné" nahradíme hodnotami typu boolean, které nám bude rekursivní rutina vracet do místa volání. Procedura OSMDAM zapíše do výstupního souboru output vygenerované řešení (v našem případě vektor $\vec{X}$ = (1, 5, 8, 6, 3, 7, 2, 4)).

```
procedure OSMDAM ;
var     I : integer ;   U :  boolean ;
        X : array [1..8] of integer ;
        R : array [1..8] of boolean ;
        D1: array [2..16] of boolean ;
        D2: array [-7..7] of boolean ;
 procedure UMISTI ( I : integer ; var U : boolean ) ;
 var  J : integer ;
 begin  {  UMISTI  }
        J := 0 ;
        repeat
           J := J + 1 ; U := false ;
           if  R[J]  and  D1[I+J]  and  D2[I-J]
              then begin
                        X[I] := J ; R[J] := false ; D1[I+J] := false ; D2[I-J] := false ;
                        if  I < 8
                           then begin
                                   UMISTI ( I+1, U ) ;
                                   if  not U
                                      then begin
                                              R[J] := true ; D1[I+J] := true ;
```

```
                                        D2[I-J] := true
                                    end
                                end
                            else  U := true
                        end
                until ( J = 8 )  or U
            end  {  UMISTI }  ;

        begin   {  OSMDAM  }
                for  I := 1  to 8 do R[I] := true ;
                for  I := 2  to 16 do D1[I] := true ;
                for  I := -7  to 7 do D2[I] := true ;
                UMISTI ( 1, U ) ;
                {  if  U  then  }  for  I := 1  to 8 do write ( X[I] : 4 )
        end  {  OSMDAM  }  ;
```

Poznamenejme, že výsledné řešení záleží na inicializaci a systematickém výběru dalších pozic (viz řádky 2 a 4 algoritmu). Kdybychom uvedenou rekursivní proceduru napsali ve tvaru :

```
procedure  UMISTI  ( I : integer ;  var  U : boolean ) ;
var  J : integer ;
begin
        J := 9 ;
        repeat
                J := J - 1 ;
                .....................................
        until  ( J = 1 )  or  U
end  ;
```

tak výsledné řešení by bylo $\vec{X}$ = (8, 4, 1, 3, 6, 2, 7, 5). Toto symetrické řešení lze ostatně předpokládat předem.

Uvedenou rutinu snadno zobecníme tak, aby generovala všechna možná řešení problému osmi dam. Systematicky budeme prohledávat všechny pozice a každé dosažené řešení nějakým způsobem uchováme. Dále uvedený program užívající rekursivní proceduru zapíše do výstupního souboru output všechna možná řešení problému osmi dam :

```
program  OSMDAM ( output ) ;
    var     I   : integer ;
            X  : array [1..8]  of  integer ;
            R  :  array [1..8]  of  boolean ;
            D1:  array [2..16]  of  boolean ;
            D2:  array [-7..7]  of  boolean ;
    procedure  ZAPIS  ;
    var  I : integer ;
    begin   {  ZAPIS  }
       for  I  :=  1  to  8  do  write ( X[I] : 4 ) ;
       writeln
    end   {  ZAPIS  } ;

    procedure  UMISTI ( I : integer ) ;
    var  J : integer ;
    begin   {  UMISTI  }
       for  J := 1  to  8 do
          if  R[J] and  D1[I+J]  and  D2[I-J]
            then  begin
                     X[I] := J ; R[J] := false ; D1[I+J] := false ; D2[I-J] := false ;
                     if  I < 8 then  UMISTI ( I + 1 )
                              else  ZAPIS  ;
                     R[J] := true ; D1[I+J] := true ; D2[I-J] := true
                  end
    end  {  UMISTI  }  ;

    begin   {  OSMDAM  }
            for  I  :=  1  to  8  do  R[I] := true ;
            for  I  :=  2  to  16  do  D1[I] := true ;
            for  I  :=  -7  to  7  do  D2[I] := true ;
            UMISTI ( 1 )
    end   {  OSMDAM  } .
```

V této verzi je rekursivní procedura UMISTI dokonce jednodušší než ve verzi pro vyhledání jednoho řešení.

4.3 Konverze prefixového zápisu výrazu na postfixový

Uvažujme součet A a B. Máme tím na mysli aplikaci operátoru "+" na operandy A a B, což zapíšeme ve tvaru A + B. Takovému tvaru zápisu se říká **infixový** :

operand$_1$ **operátor** **operand$_2$** .

Naproti tomuto běžnému tvaru zápisu výrazů má **prefixový zápis** tvar :

operátor **operand$_1$** **operand$_2$**

a **postfixový zápis** má tvar

operand$_1$ **operand$_2$** **operátor** .

Tedy infixovému zápisu A+B bude odpovídat prefixový zápis téhož výrazu ve tvaru +AB a postfixový zápis AB+. Předpony pre-, in-, post- vypovídají tedy o relativní poloze operátoru vzhledem ke dvěma operandům.

Zkoumejme nyní složitější případ infixového zápisu ve tvaru A+B*C. V tom případě je třeba znát prioritu operátorů. Nadále budeme využívat priority operátorů tak, jak je zavedena v jazyku Pascal (v uváděných triviálních případech i v jazyku C), a tudíž výraz A+B*C je interpretován jako A+(B*C). Při konverzi tohoto infixového zápisu výrazu např. do postfixového se postupně provádí konverze částí výrazu podle pořadí jejich vyhodnocování :

infixový zápis :	A+B*C
	A+(B*C)
	A+(BC*)
	A(BC*)+
postfixový zápis :	ABC*+
infixový zápis :	(A+B)*C
	(AB+)*C
	(AB+)C*
postfixový zápis :	AB+C*

Analogicky postupujeme při konverzi infixového na prefixový zápis :

infixový zápis :	A+B*C
	A+(B*C)
	A+(*BC)
	+A(*BC)
prefixový zápis :	+A*BC
infixový zápis :	(A+B)*C

(+AB)*C

*(+AB)C

prefixový zápis : *+ABC

Všimněme si hned té skutečnosti, že prefixové i postfixové notace jsou metody zápisu matematických výrazů bez použití závorek, např. :

infix	prefix	postfix
A*(B+C)	*A+BC	ABC+*
(A+B)*(C-D)	*+AB-CD	AB+CD-*
(A+B)*(C+D-E)*F	**+AB-+CDEF	AB+CD+E-*F*

Mají-li operátory stejnou prioritu, postupujeme zleva doprava. Dále řešení omezíme těmito podmínkami :

1. výraz neobsahuje konstanty ,
2. jako proměnné budou ve výrazu vystupovat pouze jednotlivá písmena ,
3. budeme uvažovat pouze binární operátory sčítání, odečítání, násobení a dělení.

Pak můžeme definice prefixových a postfixových výrazů vyjádřit rekursivně takto:

- ► prefixový výraz je buď písmeno, nebo operátor následovaný dvěma prefixovými výrazy,
- ► postfixový výraz je buď písmeno, nebo operátor, kterému předcházejí dva postfixové výrazy.

Nyní se tedy zabývejme problémem, jak provést konverzi prefixového výrazu na postfixový. V triviálním případě, kdy prefixový výraz je pouze písmeno, tak jeho postfixový ekvivalent je totožný. Např. A je prefixový i postfixový zápis téhož výrazu. Nechť prefixový výraz je reprezentován nějakým řetězcem délky větší než 1. Protože uvažujeme pouze binární operátory, tak každý takový prefixový řetězec musí obsahovat operátor, první a druhý operand. Předpokládejme, že jsme schopni identifikovat první i druhý operand, které musí být nutně kratší než originální řetězec. Pak konverzi prefixového řetězce provedeme tak, že nejprve provedeme konverzi prvního operandu na postfixový zápis, poté konverzi druhého operandu na postfixový zápis a nakonec takto vzniklého řetězce zapíšeme příslušný operátor. Dostáváme tudíž následující rekursivní algoritmus :

1. Když prefixový řetězec je písmeno, tak postfixový řetězec je totéž písmeno.
2. Nechť OP je první operátor prefixového řetězce.

3. Nalezni první operand OPND1 řetězce, převeď jej na postfixový tvar, který označ POST1.
4. Nalezni druhý operand OPND2 řetězce, převeď jej na postfixový tvar, který označ POST2.
5. Spoj POST1, POST2 a OP.

Výchozí prefixový a výsledný postfixový výraz budou reprezentovány řetězcem znaků, použijeme proto tyto definice :

```
const   MAX = 80 ;
type    RETEZ = record ;
            CH : packed array [1..MAX] of char ;
            DELKA : 0..MAX
        end ;
```

Položka DELKA reprezentuje aktuální délku řetězce, jehož znaky jsou v poli CH. Dále použijeme dva intervaly z typu integer :

```
type    TPOS = 1..MAX  ;
        TDELKA = 0..MAX ;
```

Procedura SPOJENI provádí spojení dvou řetězců a má zahlaví :

```
procedure SPOJENI ( RET1, RET2 : RETEZ ; var RET3 : RETEZ )
```

Je-li např. RET1 předávána hodnota 'ABCDE' a RET2 hodnota 'XYZ', tak procedura SPOJENI vrací v RET3 hodnotu 'ABCDEXYZ'. Procedura VYJMI vrací v RET2 podřetězec RET1 délky *j* znaků, počínaje *i*-tým znakem RET1 a má záhlaví :

```
procedure VYJMI ( RET1 : RETEZ ; I, J : integer ; var RET2 : RETEZ )
```

Pokud v poli RET1 od *i*-té pozice je méně znaků než *j* (počítaje do RET1.DELKA), je RET2 doplněn prázdnými znaky.

Funkce PISMENO vrací hodnotu typu boolean, a to true, když argumentem je písmeno, jinak vrací false.

Funkce NALEZNI identifikuje první a druhý operand k prefixovému operátoru. Předává se jí hodnota řetězce a ukazatel na nějakou složku tohoto řetězce a funkce vrací hodnotu typu integer reprezentující délku nejdelšího prefixového výrazu obsaženého v předaném řetězci počínaje udanou složkou tohoto řetězce. Má záhlaví :

```
function NALEZNI ( RET : RETEZ ; POS : TPOS ) : TDELKA
```

Tak např. NALEZNI ('A+CD', 1) vrací 1, protože 'A' je nejdelší prefixový řetězec začínající první složkou předaného řetězce. Jiné příklady :

```
NALEZNI ( '+*ABCD+GH', 1 )      →     5
NALEZNI ( 'A+CD' , 2 )          →     3
```

Pokud takový prefixový řetězec nelze nalézt, vrací funkce NALEZNI hodnotu 0. Tak např. NALEZNI ('*+AB', 1) nebo NALEZNI ('+*A-C*D', 6) vrací hodnotu 0. Rekursivní funkce NALEZNI pracuje na následujícím principu. Libovolný podřetězec originálního řetězce může obsahovat nanejvýše jeden správný prefixový výraz. To je evidentní v případě, kdy délka řetězce je rovna 1. Každý takový podřetězec může začínat buď písmenem nebo znakem operátoru. Začíná-li písmenem, tak to je tím jediným prefixovým zápisem v podřetězci. Začíná-li operátorem, tak podřetězec, který z daného podřetězce vznikne vyjmutím operátoru může obsahovat také nanejvýše jeden správný prefixový zápis, který tedy musí být prvním operandem k inicializovanému operátoru. Vyjmutím prvního operandu z tohoto nového podřetězce (kratšího než originální) vznikne ještě kratší nový podřetězec, který také může obsahovat nanejvýše jeden správný prefixový zápis, který je druhým operandem k inicializovanému operátoru atd.

Vycházejíc ze všeho dříve řečeného, můžeme zapsat následující program, ve kterém požadovanou konverzi zajišťuje procedura KONVERZE :

```
program PREFIXNAPOSTFIX ( input, output ) ;
const   MAX = 80 ;
type    TDELKA = 0..MAX ;
        TPOS =  1..MAX ;
        RETEZ = record
            CH : packed array [TPOS] of char ;
            DELKA : TDELKA
        end ;
var     PREFIX, POSFIX : RETEZ ;
        I : integer ;
procedure SPOJENI ( RET1, RET2 : RETEZ ; var RET3 : RETEZ ) ;
var     DEL1, DEL2, DEL3, I : integer ;
begin   {  SPOJENI  }
        DEL1 := RET1.DELKA; DEL2 := RET2.DELKA; DEL3 := DEL1+DEL2;
        if  DEL3 > MAX
          then  writeln ('***Chyba - prilis dlouhe retezce***' )
          else  begin
                    RET3.CH := RET1.CH ;
                    for I := DEL1 + 1 to DEL3 do
                          RET3.CH[I] := RET2.CH[I-DEL1] ;
                    RET3.DELKA := DEL3
                end
```

```
end  {  SPOJENI  }  ;

procedure  VYJMI ( RET1 : RETEZ ; I, J : integer ; var RET2 : RETEZ ) ;
var  K, L : integer ;
begin  {  VYJMI  }
       if  I + J - 1 < RET1.DELKA  then L := I + J - 1 else  L := RET1.DELKA ;
       for  K := I to L do RET2.CH[K-I+1] := RET1.CH[K] ;
       for  K := L - I + 2  to J  do RET2.CH[K] := ' ' ;
       RET2.DELKA := J
end  {  VYJMI  } ;

procedure  KONVERZE ( PREFIX : RETEZ ; var  POSTFIX : RETEZ ) ;
var    POST1, POST2, OPND1, OPND2, RETOP, POM : RETEZ ;
       OP : char ;
       M, N : TDELKA ;
 function  PISMENO ( C : char ) : boolean ;  {  pro kód ASCII }
 begin  {  PISMENO  }
       if (C >= 'A') and (C <= 'Z') then PISMENO:=true  else  PISMENO:=false ;
 end  { PISMENO  }  ;

 function NALEZNI ( RET : RETEZ ; POS : TPOS ) : TDELKA ;
 var   M, N : TDELKA ;
       PRVNI : char ;
 begin  {  NALEZNI  }
   if  POS > RET.DELKA
       then NALEZNI := 0
       else  begin
                PRVNI :=RET.CH[POS] ;
                if  PISMENO ( PRVNI )
                  then  NALEZNI := 1
                  else  begin
                           M := NALEZNI ( RET, POS+1 ) ;
                           N := NALEZNI ( RET, POS+M+1 ) ;
                           if  (M=0)  or  ( N=0) then NALEZNI := 0
                                                 else  NALEZNI := M+N+1
                        end
```

```
                end
        end { NALEZNI } ;

    begin { KONVERZE }
        if PREFIX.DELKA = 1
            then { Kontrola, zda jde o písmeno }
                if PISMENO ( PREFIX.CH[1] )
                  then POSTFIX := PREFIX
                  else writeln ( '*** Chybny prefixovy vyraz ***' )
            else begin
                  OP := PREFIX.CH[1] ;
                  M := NLAEZNI (PREFIX, 2) ; N := NALEZNI (PREFIX, M + 2) ;
                  if not ( OP in ['+','-','*','/'] ) or ( M = 0 ) or ( N = 0 ) or
                    (M+N+1 <> PREFIX.DELKA )
                    then writeln ( '*** Chybny prefixovy vyraz ***' )
                    else begin
                          VYJMI ( PREFIX, 2, M, OPND1 ) ;
                          VYJMI ( PREFIX, M + 2, N, OPND2 ) ;
                          KONVERZE ( OPND1, POST1 ) ;
                          KONVERZE ( OPND2, POST2 ) ;
                      SPOJENI ( POST1, POST2, POM ) ;
                      RETOP.CH[1] := OP ;
                      RETOP.DELKA := 1 ;
                      SPOJENI ( POM, RETOP, POSTFIX ) ;
                  end
            end
    end { KONVERZE } ;

    begin { PREFIXNAPOSTFIX }
        readln ( PREFIX.DELKA ) ;
        for I := 1 to PREFIX.DELKA do read ( PREFIX.CH[I] ) ;
        KONVERZE ( PREFIX, POSTFIX ) ;
        for I := 1 to POSTFIX.DELKA do write ( POSTFIX.CH[I] )
    end { PREFIXNAPOSTFIX } .
```

Cvičení

1. Předpokládejme, že ke specifikaci problému Hanojských věží uvedené v textu přidáme další podmínku : přemisťovaný disk lze položit pouze na disk průměru o jednotku velikosti větší (tzn., že disk 1 lze položit pouze na disk 2, disk 2 na disk 3 atd.), nebo na prázdnou jehlu. Pro jaká *n* bude v textu uvedený algoritmus funkční vzhledem k této nové podmínce ? Proč obecně tento algoritmus není funkční vzhledem k přídavné podmínce ?

2. Dokažte, že když se Hanojská věž skládá z *n* disků, tak celkový počet přemístění jednotlivých disků generovaný uvedeným algoritmem je $2^n - 1$. Lze nalézt jinou metodu, která by generovala menší počet přemístění ?

3. Problém bludiště budeme formulovat následovně. Nechť je dána nějaká matice BLUD typu (10, 10), jejíž prvky jsou buď 0 nebo 1. Osoba nalézající se na poli BLUD[1,1] musí nalézt cestu do BLUD[10,10], přičemž pohyb je možný jen do přilehlého pole v tom samém řádku nebo sloupci matice BLUD (diagonální pohyb je nepřípustný). Nepřípustný je rovněž pohyb do pole obsahujícího 1, hodnoty BLUD[1,1] a BLUD[10,10] jsou 0. Napište rutinu, která podle předané hodnoty BLUD buď vrátí informaci, že žádná taková cesta neexistuje, nebo vrátí seznam polí reprezentujících cestu od BLUD[1,1] do BLUD[10,10].

Návod : Použijeme algoritmu prohledávání s návratem (viz problém osmi dam), který v prvním přiblížení můžeme pro řešení tohoto problému zapsat následovně :

```
procedure DALSIKROK ;
begin
 "Inicializace seznamů kroků" ;
 repeat
   "Vyber dalšího kandidáta ze seznamu kroků" ;
   if "Vybraný krok přijatelný"
      then begin
             "Zaznamenej vybraný krok" ;
             if "Ještě nejsme na koncovém poli"
                then begin
                       DALSIKROK ;
                       if "Další krok byl neúspěšný"
                          then "Vymaž předchozí krok"
                     end
```

```
                end
        until "Krok byl úspěšný" or "Nejsou další kandidáti"
    end ;
```

4. Napište rekursivní funkci, které jako se argument předá prefixový řetězec obsahující pouze binární operátory a číslice (jako znaky 0 až 9) a která vrátí hodnotu prefixového výrazu reprezentovaného tímto výstupním řetězcem. Např. +23 se vyhodnotí jako 5, *2+34 jako 14.

5. SIMULACE REKURSIVNÍCH ALGORITMŮ

Některé programovací jazyky (např. FORTRAN, COBOL) neumožňují zápis rekursivních algoritmů. Problémy, které lze snadno řešit pomocí rekursivních technik (např. problém Hanojských věží), je třeba v takových jazycích řešit simulací rekursivního algoritmu pomocí jednodušších operací. Pak je to tedy problém konverze rekursivního řešení na nerekursivní. Přitom je třeba si též uvědomit, že generovaný rekursivní kód má obvykle větší požadavky na paměť a čas provedení. To je rozhodující zvláště u programů, jejichž frekvence užívání je značná, a tudíž se vyplatí konverze rekursivního na nerekursivní algoritmus.

Předpokládejme nejprve jednoduchý případ, kdy nějaká rutina RUT deklarovaná např.

```
procedure RUT ( A : ..... ) ;
        { blok RUT } ;
```

je volána příkazem RUT(X). Říkáme, že X je **argument** (**skutečný parametr**) a A je **parametr** (**formální parametr**) volané rutiny.

Při volání podprogramů (procedur nebo funkcí) se provedou tyto akce :

1. **Předání hodnot argumentů**

Pro každý parametr volaný hodnotou platí, že odpovídající argument se zkopíruje do lokálního datové pole volaného podprogramu a každá změna parametru se provádí v tomto lokálním poli a originální argument zůstává nezměněn.

2. **Alokace a inicializace lokálních proměnných**

Alokace se provádí nejen pro lokální proměnné, které jsou explicitně deklarovány ve volaném podprogramu, ale provádějí se též tzv. přechodné alokace, které se musí vytvářet v průběhu výpočtu. Tak např. pro výpočet X+Y+Z se taková alokace musí vytvořit pro X+Y a pak pro úplný výraz. Stejně tak pro příkaz např. X := FACT (N) se provádí dočasná alokace pro FACT (N) před přiřazením této hodnoty X.

3. **Předání řízení podprogramu**

Aby podprogram mohl správně vrátit řízení volající rutině, musí být v podprogramu známa **adresa návratu**, tj. kam má volaná rutina vrátit řízení po ukončení své činnosti. Tato adresa návratu je však známa ve volající rutině a aby mohla být známa též ve volané rutině, tak se jí předá jako argument. Tudíž kromě explicitních argumentů specifikovaných programátorem se volané rutině předávají i implicitní argumenty, která volaná rutina potřebuje pro výpočet a vrácení řízení. Nejdůležitější informací tohoto druhu je právě adresa návratu, kterou si volaná rutina ukládá do svého pole dat. Když jsou tyto akce provedeny, tak volaná rutina přebírá řízení.

Návrat do volající rutiny předpokládá též provedení 3 akcí. Před uvolněním lokálního pole dat volané rutiny (které obsahuje i adresu návratu) se musí na později přístupné místo uložit adresa návratu. Poté je možné uvolnit lokální pole dat volané rutiny (všechny lokální proměnné i dočasné alokace volané rutiny, včetně adresy návratu). Pak se realizuje skok na adresu návratu, která byla předtím "odložena", a tudíž řízení se předá volající rutině v bodě, který bezprostředně následuje instrukci, kterou bylo volání jiné rutiny inicializováno. Pokud je volanou rutinou funkce, tak vrácená hodnota se umístí do registru, kde je přístupná volající rutině.

Stejný mechanismus platí i pro rekursivní rutiny. Pokaždé, kdy rekursivní rutina volá sama sebe, tak je alokováno celé nové datové pole pro toto rekursivní volání, obsahující všechny parametry, lokální proměnné, dočasně alokovaná data a adresu návratu. Toto pole dat však není připojeno rutině jako celku, nýbrž jen jejímu rekursivnímu volání. Každé takové volání je příčinou nové alokace pole dat a každý odkaz na toto pole dat se řeší v nejnovější alokaci tohoto pole. Každý návrat z rutiny uvolní toto aktuální pole dat a tím se stane aktuálním pole dat alokované bezprostředně před uvolněnou alokací. Takový mechanismus přímo vybízí k využití zásobníku.

Simulace funkce faktoriál

Mechanismus rekursivního volání této funkce byl popsán dříve, přičemž pro reprezentaci postupných alokací lokálních proměnných byl užit zásobník. Můžeme uvažovat zvláštní zásobník pro každou lokální proměnnou, nebo také jeden zásobník pro celé datové pole, které je třeba alokovat při volání. Každé volání rekursivní rutiny má za následek alokaci nového pole dat, ve kterém jsou parametry inicializovány hodnotami odpovídajících argumentů a adresa návratu v tomto poli dat je inicializována adresou následující instrukci volání. Každý odkaz na lokální proměnné nebo parametry se řeší v tomto poli dat na vrcholu zásobníku. Návrat do rekursivně volané rutiny znamená, že se nejprve "odloží" adresa návratu ze zatím aktuálního pole dat, které se poté uvolní, a je proveden skok na adresu návratu. Volající rutině je dostupná ev. vrácená hodnota a při dalším běhu řeší všechny odkazy ve svém vlastním poli dat, které je nyní na vrcholu zásobníku.

Pro simulaci tedy použijeme zásobníku :

```
const MAX = 50 ;
type  ZASOBNIK = record
          VRCHOL : 0..MAX ;
          CLEN : array [1..MAX] of  ZONADAT
        end ;
```

Přitom typ ZONADAT je nejlepší definovat jako záznam, jehož položky budou nést všechny informace přenášené do pole dat při volání rutiny. Otázkou však je, jak manipulovat v

takovém případě s adresou návratu. To lze obejít způsobem, který bude vysvětlen na příkladě funkce faktoriál :

```
function  FACT ( N : NEZAPCELA ) : KLADNACELA ;
var       X : NEZAPCELA ;
          Y : KLADNACELA ;
begin   { FACT }
          if  N = 0  then  FACT := 1
                    else  begin
                              X := N - 1 ;
                              Y := FACT ( X ) ;
                              FACT := N * Y
                          end
end   { FACT } ;
```

Pole dat této rutiny, tj. zóna či oblast paměti, kde jsou obsažena data, nad kterými rutina operuje, bude obsahovat parametr N, lokální proměnné X a Y, dočasné alokace nejsou zapotřebí, ale obsahuje též adresu návratu. V tomto případě existují dva možné body návratu : přiřazení hodnoty FACT (X) lokální proměnné Y a rutina volající FACT. Předpokládejme, že v simulačním programu jsou deklarována dvě návěští :

```
label  1, 2 ;
```

jimiž jsou označeny příkazy :

```
2:      Y := VYSLEDEK
1:      FACT := VYSLEDEK
```

Předpokládejme, že obsahem proměnné VYSLEDEK je hodnota, která má být vrácena do místa volání FACT. Adresu návratu lze potom uložit jako integer I (buď 1 nebo 2) a efekt návratu z rekursivního volání realizovat příkazem :

```
case  I  of
  1  :  goto  1 ;
  2  :  goto  2
end
```

Potom když I = 1, tak řízení se vrátí do jiné rutiny, která volala FACT a když I = 2, tak návrat je simulován přiřazením vrácené hodnoty proměnné Y v předchozím volání FACT.

Pro náš příklad použijeme následující definice a deklarace :

```
const   MAX =50 ;
type    ZONADAT = record
```

```
        PARAM : NEZAPCELA ;
        X : NEZAPCELA ;
        Y : KLADNACELA ;
        ADRNAV : 1..2
    end  ;
    ZASOBNIK = record
        VRCHOL : 0..MAX ;
        CLEN : array [1..MAX]  of  ZONADAT
    end  ;
var     Z : ZASOBNIK ;
```

Identifikátor PARAM byl zvolen záměrně proto, aby se zabránilo nedorozuměním v souvislosti s parametrem N simulační rutiny. Dále deklarujme proměnné

```
var     AKTUALZONA : ZONADAT ;
        VYSLEDEK : KLADNACELA ;
```

V proměnné AKTUALZONA budou hodnoty pole dat, na které lze odkazovat v aktuálním běhu rekursivní rutiny a hodnotami VYSLEDEK budou hodnoty vrácené rekursivní rutinou.

Každé volání (návrat) rekursivní rutiny má za následek vložení (odebrání) pole dat do (ze) zásobníku. Tyto operace nad zásobníkem nechť provádějí rutiny :

```
procedure PUSH ( var Z : ZASOBNIK ; var ZONA : ZONADAT ) ;
procedure  POP ( var Z :ZASOBNIK ; var ZONA : ZONADAT ) ;
```

Pak návrat z FACT budeme simulovat pomocí příkazů :

```
VYSLEDEK := hodnota vrácená funkcí ;
I := AKTUALZONA.ADRNAV ;
POP ( Z, AKTUALZONA ) ;
case  I  of
    1 :  goto  1 ;
    2 :  goto  2
end ;
```

Rekursivní volání FACT simulujme vložením aktuální zóny dat do zásobníku a reinicializací položek AKTUALZONA.PARAM a AKTUALZONA.ADRNAV :

```
PUSH ( Z, AKTUALZONA ) ;
AKTUALZONA.PARAM := Z.CLEN[Z.VRCHOL].X ;
AKTUALZONA.ADRNAV := 2 ;
goto 10  ;  { 10 je návěští pro start simulační rutiny  }
```

Nyní již můžeme zapsat simulační rutinu :

```
function SIMFACT ( N : NEZAPCELA ) : KLADNACELA ;
label  1, 2, 10 ;
const  MAX = 100 ;
type   ZONADAT = record
          PARAM, X : NEZAPCELA ;
          Y : KLADNACELA ;
          ADRNAV : 1..2 ;
       end ;
       ZASOBNIK = record
          VRCHOL : 0..MAX ;
          CLEN : array [1..MAX]  of  ZONADAT
       end ;
var    I : 1..2 ;
       VYSLEDEK : KLADNACELA ;
       AKTUALZONA : ZONADAT ;
       Z : ZASOBNIK ;
procedure  PUSH ( var Z : ZASOBNIK ; var ZONA : ZONADAT ) ;
   begin  { PUSH }
     with  Z  do
       if VRCHOL = MAX
         then writeln ('*** Preplneni zasobniku ***' )
         else  begin
                  VRCHOL := VRCHOL + 1 ; CLEN[VRCHOL] := ZONA
               end
   end  { PUSH } ;
   procedure  POP ( var Z : ZASOBNIK ; var  ZONA : ZONADAT ) ;
   begin  { POP }
     with  Z  do
       if  VRCHOL = 0
         then  writeln ( '*** Podteceni zasobniku ***' )
         else  begin
                  ZONA := CLEN[VRCHOL] ; VRCHOL := VRCHOL - 1
               end
   end  { POP } ;
```

```
begin  { SIMFACT }
 { Simuluj začátek vložením prázdného datového pole, aby při operaci POP při
   návratu do volající rutiny nenastalo podtečení zásobníku }
   Z.VRCHOL := 0 ;
   AKTUALZONA.PARAM := 0 ; AKTUALZONA.X :=  0 ;
   AKTUALZONA.Y := 1 ; AKTUALZONA.ADRNAV :=  1 ;
   PUSH ( Z, AKTUIALZONA ) ;
   { Inicializace předaných parametrů }
   AKTUALZONA.PARAM := N ; AKTUALZONA.ADRNAV := 1 ;
10 :  { Začátek simulační rutiny }
   if  AKTUALZONA.PARAM = 0
    then  begin
                VYSLEDEK := 1 ; I := AKTUALZONA.ADRNAV ;
                POP ( Z, AKTUALZONA ) ;
                case  I  of
                   1  :  goto  1 ;
                   2  :  goto 2
                end
          end ;
   AKZUALZONA.X := AKTUALZONA.PARAM - 1 ;
   PUSH ( Z, AKTUALZONA ) ;
   AKTUALZONA.PARAM := Z.CLEN[Z.VRCHOL].X ;
   AKTUALZONA.ADRNAV := 2 ;
   goto 10 ;
2:  { Simulace rekursivního volání }
   AKTUALZONA.Y := VYSLEDEK ;
   VYSLEDEK := AKTUALZONA.PARAM * AKTUALZONA.Y ;
   I := AKTUALZONA.ADRNAV ;  POP ( Z, AKTUALZONA, ) ;
   case  I  of
    1  :  goto 1 ;
    2  :  goto 2
   end ;
1 :  { Návrat do volající rutiny }
   SIMFACT := VYSLEDEK
end  { SIMFACT } ;
```

Kdyby nějaká rekursivní funkce FUN obsahovala např. příkaz

X := A * FUN (B) + C * FUN (D)

tak před rekursivním voláním FUN (D) by bylo nutné uchovat v datovém poli zásobníku i hodnotu A * FUN (B). U vyšetřované funkce FACT takové "přechodné" alokace nejsou nutné.

Na první pohled je patrná složitost simulační rutiny SIMFACT oproti známému iteračnímu algoritmu. Pokusme se tedy o zjednodušení. Předně ne všechny lokální proměnné datového pole je třeba vkládat do zásobníku, ale jen ty proměnné, jejichž hodnota v bodě inicializace rekursivního volání musí být znovu použita po návratu z tohoto rekursivního volání. V našem příkladě se to z parametrů a lokálních proměnných N, X a Y týká pouze parametru N. Když např. při rekursivním volání FACT (0) má N hodnotu 1, tak po návratu z tohoto rekursivního volání se musí hodnoty N = 1 užít v příkazu FACT := N*Y. Ovšem Y není v bodě volání FACT (X) definováno, takže nemusí být tato hodnota vkládána do zásobníku. Naopak X je definováno v bodě volání, ale hodnota X není potřeba po návratu z FACT a tudíž se též nemusí vkládat do zásobníku. Stejný efekt by tedy měla i funkce, která by používala X a Y jako globální proměnné.

Nyní zkoumejme, zda je třeba vkládat do zásobníku adresu návratu. Je zřejmé, že adresa návratu z rekursivního volání se nemění a zůstává uvnitř FACT. Jestliže na počátku nevložíme do zásobníku prázdné datové pole, tak návrat do volající rutiny lze indikovat pomocí pokusu o odběr z prázdného zásobníku. Takže se můžeme obejít bez vkládání adresy návratu do zásobníku, musíme však upravit proceduru POP :

```
function  SIMFACT ( N : NEZAPCELA ) : KLADNACELA ;
label  1, 2, 10 ;
const   MAX = 40 ;
type    ZASOBNIK = record
            VRCHOL : 0..MAX ;
            CLEN : array [1..MAX]  of  NEZAPCELA
        end ;
var     Z : ZASOBNIK ;
        AKTUALPARAM, X : NEZAPCELA ;
        Y, VYSLEDEK : KLADNACELA ;
        PODTEC : boolean ;
procedure  PUSH ( var Z :ZASOBNIK ; X : NEZAPCELA ) ;
begin
   with  Z  do
      if  VRCHOL = MAX then  { Chyba }
```

```
                        else  begin
                                VRCHOL := VRCHO L+ 1 ;
                                CLEN[VRCHOL] := X
                              end
end ;

procedure  POPTEST ( var Z : ZASOBNIK ; var  X : NEZAPCELA ;
                     var PODTEC : boolean ) ;
begin
  with  Z  do
    if VRCHOL = 0 then  PODTEC := true
                  else  begin
                          PODTEC := false ;
                          X := CLEN[VRCHOL] ;
                          VRCHOL := VRCHOL - 1
                        end
end ;
begin  {  SIMFACT  }
  Z.VRCHOL := 0 ;   {  Inicializace zásobníku  }
  AKTUALPARAM := N ;
10 :
  if  AKTUALPARAM = 0
    then  begin
            VYSLEDEK := 1 ; POPTEST ( Z, AKTUALPARAM, PODTEC ) ;
            if  PODTEC  then  goto 1  else  goto 2 ;
          end  ;
  X := AKTUALPARAM - 1 ; PUSH  ( Z, AKTUALPARAM ) ;
  AKTUALPARAM := X ;  goto 10 ;
2:
  Y := VYSLEDEK ; VYSLEDEK := AKTUALPARAM * Y ;
  POPTEST ( Z, AKTUALPARAM, PODTEC ) ;
  if  PODTEC  then  goto 1  else  goto 2 ;
1:
  SIMFACT := VYSLEDEK
end  {  SIMFACT  } ;
```

Uvedená verze simulační rutiny je sice o něco jednodušší, nicméně příkazy skoku působí v pascalském programu rušivě. Pokusme se o jejich eliminaci. Především příkazy

```
POPTEST ( Y, AKTUALPARAM, PODTEC ) ;
if PODTEC then goto 1 else goto 2
```

se vyskytují dvakrát pro AKTUALPARAM = 0 a pro AKTUALPARAM <> 0 a lze je sloučit. Kromě toho hodnoty přiřazované proměnným AKTUALPARAM a X jsou odvozeny od sebe navzájem, a tudíž lze použít jen jednu proměnnou X. Totéž lze říci o proměnných VYSLEDEK a Y, a proto použijme jen Y. Dostaneme pak následující verzi :

```
function  SIMFACT ( N : NEZAPCELA ) : KLADNACELA ;
label   2, 10 ;
const   MAX = 40 ;
type    ZASOBNIK = record
            VRCHOL : 0..MAX ;
            CLEN : array [1..MAX]  of  NEZAPCELA
        end ;
var     Z : ZASOBNIK ;
        X : NEZAPCELA ;
        Y : KLADNACELA ;
procedure  PUSH ( var Z : ZASOBNIK ; X : NEZAPCELA ) ;
        {  blok  PUSH  } ;
procedure POPTEST ( var Z : ZASOBNIK ; var X : NEZAPCELA ;
                            var PODTEC : boolean ) ;
        {  blok  POPTEST  } ;
begin  {  SIMFACT  }
    Z.VRCHOL := 0 ;  X := N ;
10 :   if  X = 0 then  Y := 1
                else  begin
                          PUSH ( Z, X ) ; X := X - 1 ; goto 10
                      end  ;
2:     POPTEST ( Z, X, PODTEC ) ;
       if  PODTEC then  SIMFACT := Y
                   else  begin
                             Y := X * Y ; goto 2
                         end
end   {  SIMFACT  } ;
```

Nyní vidíme, že rutina obsahuje dvě smyčky :

1. smyčka začínající návěštím 10, která se provádí pro X <> 0, jinak se provede Y:=1 a následuje smyčka uvozená návěštím 2,
2. smyčka uvozená návěštím 2, která se provádí až k podtečení zásobníku, kdy rutina vrací řízení do volající rutiny.

Obě smyčky zapíšeme pomocí příkazů **while** :

```
while X <> 0 do
        begin PUSH ( Z, X ) ; X := X - 1 end ;
Y := 1 ; POPTEST ( Z, X, PODTEC ) ;
while not PODTEC do
        begin Y := X * Y ; POPTEST ( Z, X, PODTEC ) end ;
SIMFACT := Y
```

Je pochopitelné, že když X inicializujeme hodnotou N, tak po vykonání prvního příkazu **while** bude zásobník obsahovat hodnoty 1, 2, ... , N (v posloupnosti od vrcholu zásobníku). Takže víme, co bude zásobník obsahovat, a tudíž tyto hodnoty můžeme použít přímo. Když tedy eliminujeme zásobník, tak dostaneme výsledný zápis rutiny ve tvaru :

```
function SIMFACT ( N : NEZAPCELA ) : KLADNACELA ;
var     X : NEZAPCELA ;
        Y : KLADNACELA ;
begin
        Y := 1 ;
        for X := 1 to N do Y : = Y * X ;
        SIMFACT := Y
end ;
```

A to je již běžná pascalská implementace iterační verze výpočtu funkce faktoriál.

Z právě uvedeného je možné si udělat představu o efektivnosti rekursivních rutin. Všeobecně je možné říci, že nerekursivní verze programu bude mnohem efektivnější jak s ohledem na čas provedení programu, tak i s ohledem na požadavky na paměť než její rekursivní verze.

V uvedeném příkladě funkce faktoriál byla postupně prováděna zjednodušení např. eliminací lokálních proměnných, aby je nebylo třeba ukládat do zásobníku. Ovšem stávající kompilátory nejsou (zatím?) schopné generovat taková zjednodušení, a proto je výsledný kód neefektivní. Na druhé straně však jistě bylo vidět, že rekursivní řešení je v mnoha případech

daleko přirozenější a logičtější cestou řešení mnoha problémů. Je zde tedy rozpor mezi efektivností práce programu a programátora. Takové rozpory je třeba řešit podle frekvence používání výsledného produktu - programu. Jestliže je rekursivní řešení snadné, ale frekvence používání výsledného programu veliká, tak lze postupovat tak, že nerekursivní simulační verzi rekursivní rutiny budeme zefektivňovat eliminací redundantních operací.

Cvičení

1. Napište simulační nerekursivní verzi k uvedené funkci FIB :

```
function FIB ( N : NEZAPCELA ) : NEZAPCELA ;
begin
    if N <= 1 then FIB := N else FIB := FIB ( N - 1 ) + FIB ( N - 2 )
end ;
```

Dále se pokuste konvertovat získanou verzi na známou iterační metodu výpočtu Fibonacciho čísel.

2. Napište simulační nerekursivní rutinu k rekursivní rutině pro řešení problému Hanojských věží.

LITERATURA

[1] BARRON, D, W. : Rekurzívne metódy v programovaní. Bratislava, ALFA 1973.

[2] LINES, M. V. : Pascal as Second Language. New Jersey, Prentice - Hall 1980.

[3] TENENBAUM, A. M. - AUGENSTEIN, M. J. : Data Sturctures Using Pascal. New Jersey, Prentice - Hall 1980.

[4] WITH, N. : Algoritmy a štruktúry údajov. Bratislava, ALFA 1987.

PŘÍLOHA - rekurse v jazyku C

Rovněž jazyk C podporuje rekursi. Funkce v jazyku C mohou být používány rekursivně - mohou volat samy sebe buď přímo nebo nepřímo. Proto jsou zde uvedeny zápisy některých rekursivních algoritmů, které byly dříve zapisovány v jazyku Pascal, v jazyku C.

Funkce faktoriál

```
/* Toto je zcela korektní a funguje  */
int fact ( int n )  {

        if ( n < 2 )        return ( 1 ) ;
        else                return ( n * fact ( n - 1 ) ) ;
}

/* Toto funguje též správně, ale proč tak složitě? */

int fact1 ( int *n ) {
        int x, y ;

        if ( *n < 2 )  return ( 1 ) ;
        else     {
             x = *n - 1 ; y = fact1 ( &x ) ; return ( *n * y ) ;
        }
}

/* Tento zápis vypadá formálně správně, ale nedává správný výsledek */

int fact2 ( int *n )  {
        int x, y ;

        if ( *n < 2 )   return ( 1 ) ;
        else   {
```

```
            *n = *n - 1 ; x = fact2 ( n ) ; y = *n + 1 ; return ( x * y ) ;
        }
}
/* Co se stane, když v předchozí verzi zaměníme pořadí operací ?
   Výsledek bude správný  !!!
 */

int fact3 ( int *n ) {
        int x, y ;

        if ( *n < 2 )   return ( 1 ) ;
        else   {
            *n = *n -1 ; y = *n + 1 ; x = fact3 ( n ) ; return ( x * y ) ;
        }
}
```

Binární vyhledávání

Nechť je dáno nějaké vzestupně setříděné pole *a* s *n* složkami. Zapíšeme nyní v pseudonotaci blízké jazyku C rekursivní algoritmus (vývojové schéma) pro vyhledání prvku *x* v poli *a* od *a*[*d*] do *a*[*h*] (počínaje složkou s indexem *d* do indexu *h*,). Algoritmus umístí do proměnné *vrat* takovou hodnotu *i*, že *a*[*i*] = *x*, nebo hodnotu -1, pokud takový prvek v zadaném úseku pole *a* není.

```
1.  if ( d > h )
2.          vrat = - 1 ;
3.  else
4.          s = ( d + h ) / 2 ;
5.          if ( x == a[s] )
6.                  vrat = s ;
7.          else
8.                  if ( x < a[s] )
9.                          "Vyhledej x v poli a od a[d] do a[s-1]" ;
10.                 else
11.                         "Vyhledej x v poli a od a[s+1] do a[h]" ;
```

Odpovídající funkce v jazyku C může vypadat následovně :

```
int binvyh ( int *a, int n, int d, int h , int x )  {
        int s, vrat ;

        if ( d < 0 || h > n-1 || d > h ) vrat = -1 ;
        else {
         s  = ( d + h ) / 2 ;
         if ( x == a[s] )  vrat = s ;
         else
           if ( x < a[s] )   vrat = binvyh ( a, n, d, s-1, x ) ;
           else              vrat = binvyh ( a, n, s+1, h, x ) ;
        }
        return ( vrat ) ;
}
```

Pokud tomu nebrání jiné okolnosti, je lépe použít pole *a* jako externí proměnnou a provést ev. kontrolu argumentů funkce před jejím voláním. Pak by stejná funkce mohla vypadat následovně :

```
int binvyh ( int d, int h , int x )  {
        int s, vrat ;

        if ( d > h ) vrat = -1 ;
        else {
         s  = ( d + h ) / 2 ;
         if ( x == a[s] )  vrat = s ;
         else
           if ( x < a[s] )   vrat = binvyh ( d, s-1, x ) ;
           else              vrat = binvyh ( s+1, h, x ) ;
        }
        return ( vrat ) ;
}
```

Nepřímá rekurse (rekursivní definice algebraických výrazů)

```
/*  V ý r a z    je  t e r m  následovaný znakem +, za kterým následuje  t e r m, nebo
 *  je to  t e r m samotný
 *  T e r m  je  f a k t o r  následovaný znakem *, za   kterým následuje  f a k t o r,
 *   nebo je to  f a k t o r   samotný
 *  F a k t o r  je buď písmeno, nebo je to  v ý r a z  uzavřený v okrouhlých
 *  závorkách.
 *
 * ---------------------------------------------------------------------------------------------------------------
 *  Napište program, který podá zprávu, zda daný řetezec obsahuje správný zápis
 *  výrazu dle naší definice, resp. polohu posledního znaku správného zápisu výra-
 *  zu (viz  následující příklady ).
 *
 *  (A)    -> delka =      3 ,       posice =       3 ,      TRUE
 *  A*B+C          ->      5 ,                      5 ,      TRUE
 *  A+*B           ->      4 ,                      3 ,      FALSE
 *  (A+B*)C        ->      7 ,                      6 ,      FALSE
 *  A+B+C          ->      5 ,                      3 ,      TRUE
 *  A+B*CD         ->      6 ,                      5 ,      TRUE
 *  (A+B*C         ->      6 ,                      7 ,      FALSE
 * ---------------------------------------------------------------------------------------------------------------
 *  Sami přepište pak uvedený program tak, že zápis každého výrazu bude ukončen
 *  znakem středník, a mezery či nové řádky v zápisu jsou bezvýznamné !
 */

#include <stdio.h>
#include <strings.h>

#define  MAX  100

char *retez ;          /* Max délka vstupního výrazu 100 znaků */
int   akt_pos;         /* Ukazatel na aktuální složku v poli retez */

/* Vrat do místa volání znak na aktuální posici v poli retez a zvětši aktuální posici o 1 */
char cti_znak ( void ) {
```

```
        if ( akt_pos > 99 )         return ( 32 ) ;
        else                        return ( retez[akt_pos++] ) ;
}

/* Nutné prototypy */
char faktor ( void ) ;
char term ( void ) ;
char vyraz ( void )  {
        char nalez ;

        nalez = term () ;
        if ( !nalez )       /* neex. výraz */          return ( 0 ) ;
        else   /* vezmi další znak výrazu */
                if ( cti_znak()  != '+' )  {
                        /* Byl nalezen nejdelší výraz, vrať polohu posledního znaku
                           tohoto výrazu   */
                        akt_pos-- ; return ( 1 ) ;
                }
                else {  /* Byl nalezen term následovaný znakem +, najdi druhý term */
                        nalez = term () ;
                        if ( nalez )  return ( 1 ) ;
                        else          return ( 0 ) ;
                }
}

char term ( void )  {
        char nalez ;

        nalez = faktor () ;
        if ( !nalez )  return ( 0 ) ;
        else   if ( cti_znak() != '*' )  {
                        akt_pos-- ; return ( 1 ) ;
                }
                else  {
                        nalez = faktor () ;
                        if ( nalez )  return ( 1 ) ;
```

```
                    else        return ( 0 ) ;
                }
}

char faktor ( void ) {
        char nalez, c ;

        c = cti_znak () ;
        if (  c != '(' )  {
                if ( c > 64 && c < 91  ||  c > 96 && c < 123     )   /* písmeno */
                        return ( 1 ) ;
                else      return ( 0 ) ;
        }
        else {    /* faktor může být výraz */
                nalez = vyraz () ;
                if ( !nalez )  return ( 0 ) ;
                else {
                        if ( cti_znak () != ')' )  return ( 0 ) ;
                        else                       return ( 1 ) ;
                }
        }
}

main ()  {

        retez = ( char *) malloc ( MAX * sizeof ( char ) ) ;
        for ( ;; )  {
                gets ( retez ) ; akt_pos = 0 ;
                if ( *retez == '\0' ) break ;
                printf ( "Vyraz je spravny : " ) ;
                if ( vyraz () )      printf ( "T" ) ;
                else                 printf ( "F" ) ;
                printf ( "   ( delka : %d , posice : %d )\n",  strlen ( retez ) , akt_pos ) ;
        }
}
```

Problém osmi dam

```
/*      Nalezni jedno řešení problému osmi dam    */

#include <stdio.h>


char x[8] ;              /* Výsledné pole řešení je x */
char r[8] ;              /* Zda se dáma nachází na i- tém řádku (i=0..7) */
char d1[15], d2[15] ;    /* Zda se dáma nachází na vedlejší diagonále a na hlavní diagonále */

void umisteni ( char i , char *uspech )  {
        int j = 0 ;

        do   {
                *uspech = 0 ;
                if ( r[j] && d1[i+j] && d2[i-j+7] )  {
                        x[i] = j ; r[j] = 0 ; d1[i+j] = 0 ; d2[i-j+7] = 0 ;
                        if ( i < 7 )  {
                                umisteni ( i+1, uspech ) ;
                                if ( !*uspech )  {
                                        r[j] = 1 ; d1[i+j] =1 ; d2[i-j+7] = 1 ;
                                }
                        }
                        else  *uspech = 1 ;
                }
                j++ ;
        } while ( j < 8 && !*uspech ) ;
}


main ()  {

        char  i, uspech ;

        /* Inicializace --> */
```

```
        for ( i=0 ; i<15; i++ )  {
                if ( i < 8 ) r[i] = 1;
                d1[i] = 1 ; d2[i] = 1 ;
        }

        umisteni ( 0, &uspech ) ;
        for ( i=0 ; i<8; i++ ) printf ( "%3d", x[i]+1 ) ;
        printf ( "\n" ) ;
}

/*      Nalezni všechna řešení problému osmi dam        */

#include <stdio.h>
int pocet_zapisu = 0 ;
/* Zapiš jedno nalezené řešení na standardní výstup */

void zapis ( char *x )  {
        int i ;

        for ( i=0; i<8; i++ ) printf ( "%4d", x[i]+1 ) ;
        printf ( "\n" ) ;
        if ( pocet_zapisu % 10 == 0 ) getchar() ;
}

void umisteni ( char i , char *x, char *r, char *d1, char *d2 )  {
        char j ;

        for ( j=0; j<8 ; j++ )  {
                if ( r[j] && d1[i+j] && d2[i-j+7] ) {
                        x[i] = j ; r[j] = 0 ; d1[i+j] = 0 ; d2[i-j+7] = 0 ;
                        if ( i < 7 )  umisteni ( i+1, x, r, d1, d2 ) ;
                        else  { pocet_zapisu++ ; zapis ( x ) ; }
                        r[j] = 1 ; d1[i+j] = 1 ; d2[i-j+7] = 1 ;
                }
        }
}
```

```
void *osm_dam ( void ) {
        char i ;
        /* Vektor řešení */
        char *x = (char *) malloc ( 8 * sizeof ( char ) ) ;
        /* Zda se dáma nachází na i- tém řádku (i=0..7) */
        char *r = (char *) malloc ( 8* sizeof ( char ) ) ;
        /* Zda se dáma nachází na diagonále zleva doprava  a na diagonále zprava doleva */
        char *d1 = ( char * ) malloc ( 15 * sizeof ( char ) ) ;
        char *d2 = ( char * ) malloc ( 15 * sizeof ( char ) ) ;

        /* Inicializace */
        for ( i=0 ; i<15; i++ )  {
                if ( i < 8 ) r[i] = 1;
                d1[i] = 1 ; d2[i] = 1 ;
        }
        umisteni ( 0, x, r, d1, d2 ) ;
        free ( x ) ; free ( r ) ; free ( d1 ) ; free ( d2 ) ;
}

main ()  {
        osm_dam () ;
}
```

Konverse prefixového zápisu výrazu na postfixový

```
/*   Použitím rekurse řešte úlohu konverse prefixového výrazu na jeho postfixovy ekvivalent !
 * ------------------------------------------------------------------------------------------------------------
 *   Omezení : Uvažujme pouze binární operátory +, -, *, / a znaky A až Z jako  operandy.Jiné
 *                znaky jsou v prefixovém zápisu nepřípustné (včetně mezer)
 * ------------------------------------------------------------------------------------------------------------
 */
#include <stdio.h>
#include <strings.h>

#define MAX 101         /* Max. délka prefixového zápisu je 100 */
```

```
/* ------------------------------------------------------------------------------------------------------------------
 * Funkce "max_delka" vrací hodnotu typu integer representující délku nejdelšího prefixového
 *  výrazu obsaženého v předaném řetezci "prefix", počínaje znakem na posici  "pos" v řetězci
 *  "prefix"  - rekursivní funkce
 *  Příklady :
 *      max_delka ( "A+CD", 0 )                 -> 1
 *      max_delka ( "+*ABCD+GH", 0 )            -> 5,  neboť +*ABC je nejdelší prefix v řetězci
 *      max_delka ( "A+CD", 1 )          -> 3
 *      max_delka ( "*+AB", 0 )          -> 0
 * ------------------------------------------------------------------------------------------------------------------
 */

int max_delka ( char *prefix , int pos )   {
        int m, n ;
        int max_pos = strlen ( prefix ) - 1 ;

        if ( pos > max_pos )    return ( 0 ) ;
        else     if ( *(prefix+pos) > 64 && *(prefix+pos) < 91 )
                        return ( 1 ) ;
                else {
                        m = max_delka ( prefix, pos+1 ) ;
                        n = max_delka ( prefix, pos+m+1 ) ;
                        if ( m == 0  || n == 0 )
                                return ( 0 ) ;
                        else     return ( m + n + 1 ) ;
                }
        }



/* ------------------------------------------------------------------------------------------------------------------
 *  Rekursivní funkce "konverse" vytváří ze vstupního řetězce "prefix" výstupní řetězec "postfix"
 *  jako postfixový ekvivalent originálního prefixového zápisu výrazu.
 * ------------------------------------------------------------------------------------------------------------------
 */
```

```
int  konverse ( char *prefix, char *postfix )  {
        int delka = strlen ( prefix ) ;
        int m, n, i ;
        char *op        = (char *) malloc ( 2 * sizeof ( char ) ) ;
        char *opnd1     = (char *) malloc ( (delka-1) * sizeof ( char ) ) ;
        char *opnd2     = (char *) malloc ( (delka-1) * sizeof ( char ) ) ;
        char *post2     = (char *) malloc ( (delka-1) * sizeof ( char ) ) ;

        op[1] = '\0' ; strcpy ( postfix , "" ) ;
        if ( delka == 1 )
                if ( *prefix > 64 && *prefix < 91 )  {
                        strcpy ( postfix, prefix ) ; return ( 1 ) ;
                }
                else  {
                        strcpy ( postfix, "" ) ; return ( 0 ) ;
                }
        op[0] = *prefix ;
        m = max_delka ( prefix, 1 ) ;
        n = max_delka ( prefix, m+1 ) ;
        if ( op[0] != '+' && op[0] != '-' && op[0] != '*' &&
            op[0] != '/' || m == 0 || n == 0 || m+n+1 != delka  ) {
                strcpy ( postfix, "" ) ; return ( 0 ) ;
        }

        /* Vyjmi první prefixový výraz, označ jej opnd1 */
        for ( i=0 ; i < m ; i++ ) *(opnd1+i) = *(prefix+i+1 ) ;
        *(opnd1+m) = '\0' ;

        /* Vyjmi druhý prefixový výraz, označ jej opnd2  */
        for ( i=0 ; i < n ; i++ ) *(opnd2+i) = *(prefix+i+m+1) ;
        *(opnd2+n) = '\0' ;

        /* Konvertuj opnd1 a opnd2 na postfixový zápis */
        konverse ( opnd1, postfix ) ;
        konverse ( opnd2, post2 ) ;
```

```
        /* Spojení */
        strcat ( postfix, post2 ) ; strcat ( postfix, op ) ;
        return ( 1 ) ;
}

main ()  {

        char *prefix  = (char *) malloc ( 20*sizeof ( char ) ) ;
        char *postfix = (char *) malloc ( 20*sizeof ( char ) ) ;
        int pos ;

        for ( ; ; )  {
                gets( prefix ) ;
                if ( prefix[0] == '\0' ) break ;
                if ( konverse ( prefix, postfix ) )  printf ( "%s\n", postfix ) ;
                else                                 printf ( "Chyba v konversi\n" ) ;
        }
}
```