Rekurzívne algoritmy

Hovoríme, že objekt je *rekurzívny*, ak sa čiastočne skladá, alebo je definovaný pomocou seba samého. S rekurziou sa nestretávame iba v matematike, ale aj v bežných situáciách denného života. Kto by sa napr. ešte nestretol s reklamným obrázkom, ktorý obsahuje sám seba?

Rekurzia je silným nástrojom najmä pri matematických definíciách. Známe sú príklady prirodzených čísel a niektorých funkcií:

1. Prirodzené čísla:
a) 1 je prirodzené číslo,
b) nasledovníkom prirodzeného čísla je prirodzené číslo.
2. Funkcia *n*-faktoriál n! (pre nezáporné celé čísla):
a) 0! = 1,
b) ak n>0, tak n! = n . (n - 1)!.

Sila rekurzie očividne spočíva v možnosti definovať nekonečnú množinu objektov konečným príkazom. Podobným spôsobom možno nekonečný počet výpočtov opísať pomocou konečného rekurzívneho programu, dokonca aj vtedy, keď tento program neobsahuje explicitné opakovania. Prirodzene, rekurzívne algoritmy sú najvhodnejšie pri riešení problémov a pri výpočtoch funkcií alebo spracúvaní takých štruktúr údajov, ktoré už samy osebe sú definované rekurzívnym spôsobom.

Dôležitým a účinným prostriedkom na vyjadrenie rekurzie v programoch je procedúra (alebo podprogram), ktorej identifikátor slúži a rekurzívnu aktiváciu jej tela. Ak procedúra *P* obsahuje priamy odkaz na seba, tak o nej hovoríme, že je *priamo rekurzívna*; ak *P* obsahuje odkaz na inú procedúru *Q*, ktorá obsahuje odkaz na procedúru *P*, tak *P* je *nepriamo rekurzívna*. Použitie rekurzie teda nemusí byť okamžite zrejmé z textu programu.

Býva zvykom združovať množinu lokálnych objektov s procedúrou, t.j. množinu premenných, konštánt, typov a procedúr, ktoré sú definované lokálne v rámci tejto procedúry a ktoré neexitujú, resp. nemajú zmysel mimo tejto procedúry. Každým rekurzívnym volaním takejto procedúry vznikne nová množina jej lokálnych premenných. Tieto premenné majú síce tie isté identifikátory, aké mali pri predošlom volaní procedúry, ale ich hodnoty sú odlišné. Konfliktom pri pomenúvaní sa prechádza na základe pravidla viditeľnosti (rozsahu) identifikátorov; identifikátory sa vzťahujú vždy na najposlednejšie vytvorenú množinu premenných. Rovnaké pravidlo platí pre parametre procedúr, ktoré sú na základe definície zviazané s procedúrou.

Podobne, ako príkazy cyklov, aj rekurzívne procedúry dávajú možnosť nekonečných výpočtov. Preto je potrebné zaoberať sa otázkou ukončenia výpočtu. Základnou požiadavkou je, aby sa rekurzívne volanie procedúry *P* riadilo podmienkou *B*, ktorá môže byť za určitých okolností nesplnená.

Jedným z vhodných spôsobov zaručenia ukončenia rekurzie je použitie (konštantného) parametra, povedzme *n*, procedúry *P*, ktorú potom voláme s *n - 1*, ako hodnotou parametra. Ukončenie rekurzívnych volaní je potom zaručené prostredníctvom podmienky *n>0* namiesto podmienky *B*.

V praktických aplikáciách sa odporúča poukázať nielen na konečnosť hĺbky rekurzie, ale aj na malú hodnotu tejto hĺbky. Dôvod je prostý: každým volaním

rekurzívnej procedúry *P* sa vyžaduje pridelenie určitého množstva pamäti potrebnej na lokálne premenné procedúry *P*. Okrem týchto združených lokálnych premenných potrebujeme nejakú pamäť na uchovanie momentálneho stavu výpočtu, v ktorom sa nachádza rekurzívna procedúra. Tento moment je dôležitý vtedy, keď sa skončí výpočet posledného vyvolania procedúry a je potrebné obnoviť jej predchádzajúci stav, v ktorom sa nachádzala pred posledným vyvolaním.

Rekurzívne algoritmy sú výhodné najmä vtedy, keď problém, ktorý sa rieši, alebo údaje, s ktorými sa má manipulovať, sú definované rekurzívne. To však neznamená, že rekurzívne definície automaticky zaručujú, že použitie rekurzívneho algoritmu bude najlepším spôsobom riešenia daného problému. Skutočne, objasnenie pojmu rekurzívny algoritmus prostredníctvom nevhodných príkladov bolo najväčším dôvodom vzniku všeobecných obáv a antipatií proti použitiu rekurzie pri programovaní. Súčasne vznikla domienka, že rekurzia je neefektívna.
Napríklad výpočet n!:

❖ rekurzívny algoritmus:
```
function F(I : integer) : integer;
begin
    if I>0 then F:=I*F(I - 1) else F:=1;
end;
```

❖ nerekurzívny algoritmus:
```
I := 0; F := 1;
while I < n do begin I := I + 1; F := I * F; end;
```

Existujú oveľa zložitejšie rekurzívne schémy, ktoré možno transformovať na iteratívny tvar. Príkladom môže byť výpočet Fibonacciho čísel definovaných rekurentným vzťahom $fib_{n+1}=fib_n+fib_{n-1}$ pre n>0. Navyše platí $fib_1=1$, $fib_0=0$. Priame a naivné riešenie môže viesť k tomuto programu:

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

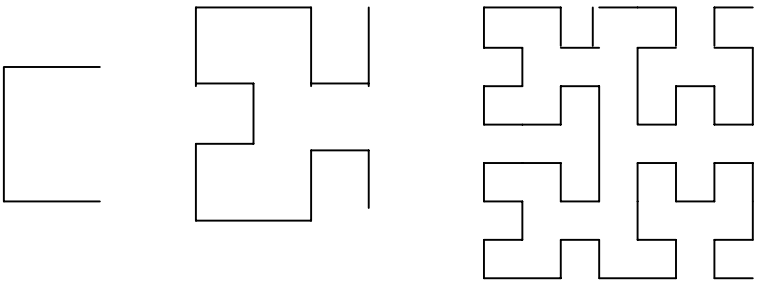
Výpočet hodnoty fib_n prostredníctvom vyvolania funkcie Fib(n) spôsobí rekurzívne volanie tejto funkcie. Otázkou je, koľkokrát sa táto funkcia volá. Poznamenávame, že každé volanie s hodnotou parametra *n*>1 má za následok dve ďalšie volania, t.j. celkový počet volaní rastie exponenciálne. Takýto program je zjavne nepraktický. Je jasné, že Fibonacciho čísla môžeme vypočítať i prostredníctvom iteratívnej schémy algoritmu, v ktorej sa navyše vyhneme vypočítavaniu rovnakých hodnôt pomocou zavedenia dvoch pomocných premenných x, y takých, že $x=fib_i$ a $y=fibi_{-1}$.

```
i:=1;x:=1;y:=0;
while i<n do
     begin
          z:=x;i:=i+1;
          x:=x+y;y:=z;
     end;
```

Poznamenávame, že tri priraďovacie príkazy, ktoré priraďujú hodnoty premenným x, y, z, možno nahradiť dvoma príkazmi, pri ktorých nepotrebujeme pomocnú premennú z: x:=x+y; y:=x-y.

Z uvedených úvah pre nás vyplýva ponaučenie: pokiaľ je možné daný problém vyriešiť pomocou iterácie, vyhýbajme sa rekurzii.

To, samozrejme, neznamená, že by sme sa mali za každú cenu snažiť zbaviť rekurzie. Existuje predsa mnoho vhodných aplikácií rekurzie, ako dokumentujú nasledujúci príklad.

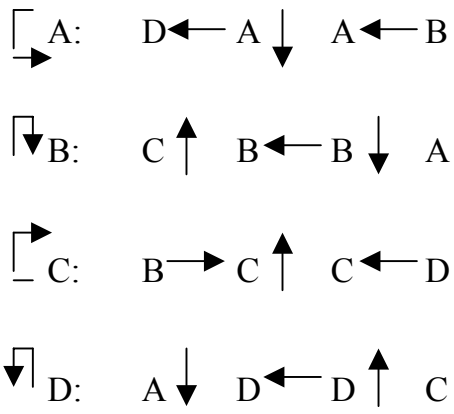


H_1 H_2 H_3

Obrázok nám súčasne dokumentuje vznik krivnky H_{i+1} z krivky H_i: krivku H_{i+1} dostaneme kompozíciou štyroch kriviek H_i polovičnej veľkosti, ich vhodnou rotáciou a spojením pomocou troch priamok. Krivku H_1 môžeme pritom považovať za kompozíciu štyroch prázdnych kriviek H_0 spojených tromi priamkami. Krivka H_i sa nazýva *Hilbertova krivka i-tého rádu* podľa svojho objaviteľa D. Hilberta (1891).

Predpokladajme, že našimi základnými kresliacimi prostriedkami sú dve súradnicové premenné *x* a *y*, procedúra *nastavpero* (nastaví kresliace pero do polohy určenej súradnicami *x* a *y*) a procedúra *kresli* (posunie pero z jeho momentálnej polohy do polohy určenej súradnicami *x* a *y*).

Pretože každá krivka H_i pozostáva zo štyroch polovičných kópií krivky H_{i-1}, je prirodzené vyjadriť procedúru pre nakreslenie krivky H_i ako kompozíciu štyroch častí, z ktorých každá nakreslí krivku H_{i-1} v primeranej mierke a pootočení. Ak označíme tieto štyri časti symbolmi A, B, C, D a procedúry, kresliace spájacie priamky pomocou šípok v zodpovedajúcom smere, tak získame nasledujúcu rekurzívnu schému:



Ak označíme jednotkovú dĺžku priamky symbolom *h*, tak procedúra zodpovedajúca schéme A sa dá výstižne vyjadriť pomocou rekurzívneho volania podobne navrhnutých procedúr B a D a seba samej.

```
procedure A(i:integer);
begin    if i>0 then
     begin D(i-1);x:=x-h;kresli;
          A(i-1);y:=y-h;kresli;
          A(i-1);x:=x+h;kresli;
          B(i-1);
     end;
end;
```

Vyžadujeme, aby maximálna šírka strany $h_0=2^k$ pre ľubovoľné $k \leq n$.

Podobne vytvoríme procedúry B, C, D.