Rekurzia

Na Zem prišiel mimozemšťan Ignác. Počas svojho pobytu už pochopil, že každý človek má nejakých rodičov. Ako by ste mu vysvetlili, čo znamená slovo *predok*?

Najjednoduchšie vysvetlenie vyzerá asi takto: "Predkovia človeka sú jeho rodičia a tiež všetci ich predkovia."

Všimnite si, že slovo *predok* sme definovali pomocou neho samého. Napriek tomu je takáto definícia úplne v poriadku – Ignác si z nej vie postupne odvodiť, že predkovia sú aj rodičia rodičov, aj rodičia rodičov rodičov, a tak ďalej. Takémuto spôsobu definovania pojmov hovoríme rekurzia[1].

Cvičenie 1.0.1 *Tehla váži jedno kilo a pol tehly. Koľko váži tehla?*

1.0.1 Rekurzia v matematike

V podobnom duchu ako pri definícii predkov s obľubou používame rekurziu v matematike. Napríklad takto môžeme definovať, čo sú to prirodzené čísla: "1 je prirodzené číslo. Ak x je prirodzené číslo, tak aj $x + 1$ je prirodzené číslo. A nič iné nie je prirodzené číslo."

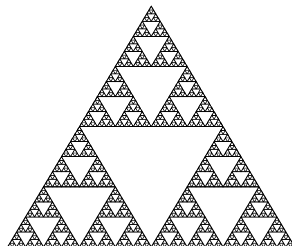
Iným príkladom je definovanie jednoduchých aritmetických výrazov s premennými x a y, sčítaním a násobením:

1. "x" je výraz, aj "y" je výraz.
2. Ak "U" a "V" sú výrazy, tak aj "$U + V$" a "$U \times V$" sú výrazy.
3. Ak "U" je výraz, tak aj "(U)" je výraz.
4. Ak sa niečo použitím týchto pravidiel nedá vyrobiť, tak to nie je výraz.

Všimnite si, že rekurzívna definícia objektu sa skladá z dvoch častí: vymenujeme nejaké základné objekty, a následne uvedieme pravidlá, ako z nich vyrábať nové, zložitejšie objekty.

Cvičenie 1.0.2 *Naša definícia aritmetických výrazov má jednu zlú vlastnosť – spôsob "výroby" výrazu nemusí zodpovedať poradiu, v akom by sme ho vyhodnotili.*
Napríklad môžeme ukázať, že $x \times y + x$ je výraz, nasledovne: Podľa pravidla 1 vieme, že x aj y sú výrazy. Podľa pravidla 2 vieme, že $y + x$ je výraz. A opäť podľa pravidla 2 vieme výraz $x \times y + x$ vyrobiť vložením symbolu násobenia medzi výrazy x a $y + x$.
Keby sme však výsledný výraz išli vyhodnotiť (podľa priorít operátorov + a ×), výsledok nedostaneme vynásobením x a $y + x$, ale sčítaním $x \times y$ a x.
Nájdite upravenú definíciu aritmetických výrazov, v ktorej bude spôsob "výroby" výrazu vždy zodpovedať poradiu, v akom daný výraz vyhodnocujeme.

Jedným zo známych použití rekurzie sú *fraktály* – geometrické objekty, ktoré sa skladajú zo zmenšených kópií seba samého. Známym príkladom je Sierpińského trojuholník (obrázok 1.1).



Obr. 1.1: Sierpińského trojuholník sa skladá z troch menších kópií seba.

[1] Z latinského "recurrere" – bežať späť.

Rekurzívne môžeme definovať aj funkcie a postupnosti. Bezkonkurenčne najznámejším príkladom sú Fibonacciho čísla – postupnosť definovaná nasledujúcim predpisom:

$$F_0 = 0, \qquad F_1 = 1, \qquad (\forall n \geq 0)\ F_{n+2} = F_{n+1} + F_n$$

Fibonacciho čísla sa v prírode vyskytujú až nečakane často (počty okvetných lístkov na kvetoch, počty predkov včely v generáciách, špirály na šiškách a kvetoch slnečnice). Rovnako často sa zvyknú zjaviť aj v rôznych oblastiach matematiky. Už sme sa s nimi stretli napríklad pri odhade veľkosti AVL stromov.

Cvičenie 1.0.3 *Uvažujme nasledujúci, veľmi neefektívny spôsob, ako vypočítať Fibonacciho číslo. Zoberieme si riadkovaný papier. Do prvého riadku napíšeme číslo, ktoré chceme vypočítať. Každý ďalší riadok dostaneme tak, že všetky Fibonacciho čísla, ktoré sú v predchádzajúcom riadku, rozpíšeme podľa definície.*
Pre F_3 *by teda náš papier vyzeral takto:*
F_3
F_2, F_1
F_1, F_0, *1* $\leftarrow$ *tu* F_1 *a* F_0 *vznikli z* F_2, *a 1 vznikla z* F_1.
1, 0
Hodnotu Fibonacciho čísla, ktoré počítame, dostaneme tak, že sčítame všetky prirodzené čísla na našom papieri. V našom prípade dostávame $F_3 = 1 + 1 + 0 = 2$. *Dokážte, že tento postup funguje pre ľubovoľné* F_n.
Označme P_n *celkový počet čísel (Fibonacciho aj prirodzených dokopy), ktoré budú napísané na papieri po vypočítaní* F_n. *Napíšte rekurzívny vzťah, pomocou ktorého sa budú dať hodnoty* P_i *počítať. Nájdite vzťah medzi hodnotami* P_i *a Fibonacciho číslami.*

1.0.2 Rekurzia v programovaní

V programovaní je rekurzia cenným nástrojom – podobne ako v matematike, umožňuje nám jednoduchšie a zrozumiteľnejšie sformulovať niektoré postupy.

Rozdiel bude v tom, že v programovaní sa na rekurziu väčšinou pozeráme opačne. Kým v matematike rekurzia popisovala, ako z jednoduchých objektov vyrábať zložitejšie, v programovaní budeme využívať opačný smer úvahy: Rekurzívne budeme popisovať, ako rozdeliť veľký problém na menšie, ľahšie riešiteľné problémy.

V tejto knihe sme už takýmto spôsobom rekurziu dokonca aj použili. Napríklad pri definovaní in-order zápisu binárneho stromu.

V programovaní sa pojem *rekurzia* zvykne často používať v konkrétnom, mierne špecifickejšom význame: Popisujeme ním situáciu, kedy nejaká funkcia volá samú seba. Príklad takejto funkcie uvádzame ako Program 1.

```
function Fibonacci(x : longint) : longint;
begin
   if x=0 then result := 0;
   if x=1 then result := 1;
   if x>1 then result := Fibonacci(x-1) + Fibonacci(x-2);
end;
```

Program 1: Rekurzívna funkcia počítajúca Fibonacciho číslo F_x.

Nie je ťažké uvedomiť si, že v podstate ide len o jeden špeciálny prípad *rekurzie* v matematickom zmysle – v niektorých "zložitých" prípadoch vypočítame návratovú hodnotu funkcie pomocou jej iných, "jednoduchšie spočítateľných" návratových hodnôt.

Cvičenie 1.0.4 *Funkcia z Programu 1 nepredstavuje práve najlepší možný spôsob, ako počítať Fibonacciho čísla. Už na vypočítanie hodnoty* $F_{30} = 832\,040$ *potrebujeme dokopy zavolať funkciu* `Fibonacci` *až* $2\,692\,537$-*krát.*
Označme C_n *počet zavolaní funkcie* `Fibonacci` *pri počítaní hodnoty* F_n. *Nájdite rekurzívny vzťah, pomocou ktorého môžeme počítať hodnoty* C_i. *Vysvetlite, ako súvisí toto cvičenie s cvičením 1.0.3.*

1.0.3 Memoizácia

Existuje jeden veľmi dôležitý rozdiel medzi funkciami v matematike a funkciami v programovaní. Ak matematickú funkciu vyhodnotíme teraz, dostaneme ten istý výsledok, ako keby sme ju vyhodnotili pred týždňom. Obvod kruhu s polomerom r bude aj o sto rokov stále rovný $2\pi r$. O funkciách v programoch toto nemusí byť pravda.

Problém je v tom, že v mnohých programovacích jazykoch (vrátane C/C++ a Pascalu) sa môže funkcia pozerať na aktuálne hodnoty niektorých globálnych premenných, prípadne ich aj meniť, môže generovať náhodné čísla, či napríklad prečítať časť vstupu z klávesnice. Tým sa celá situácia komplikuje – aj keď spustíme dvakrát tú istú funkciu s tými istými parametrami, môžeme dostať rôzne výsledky.[2] V niektorých situáciách (napr. keď voláme metódu objektu, ktorá pristupuje k jeho súkromným premenným) je takéto správanie dokonca žiadané.

Napriek tomu sa často stane, že v programe potrebujeme často počítať hodnoty nejakej funkcie, ktorá je funkciou aj v matematickom zmysle. Ako si pri tom môžeme ušetriť prácu?

Organizátorom olympiády v informatike v USA[3] sa podarilo zhrnúť celý proces návrhu efektívnych algoritmov do dvoch bodov:

1. *Don't do anything stupid.* ("Nerob nič hlúpe," teda nepočítaj veci, ktoré na získanie výsledku nepotrebuješ.)

2. *Don't do anything twice.* ("Nerob nič dvakrát," teda nepočítaj znova veci, ktoré si už niekedy spočítal.)

V našom prípade sa môžeme držať bodu 2. Akonáhle raz spočítame hodnotu našej funkcie pre nejaké parametre, zapamätáme si výsledok. Keby sme náhodou niekedy neskôr potrebovali znova vypočítať hodnotu funkcie pre tieto isté parametre, nebudeme všetko počítať odznova, ale rovno vrátime zapamätanú hodnotu. Túto optimalizačnú metódu voláme *memoizácia*.

Ako veľmi nám môže memoizácia pomôcť? Ukážeme si jednoduchý príklad. Pozrime sa na Program 2 – náš starý kamarát, funkcia počítajúca Fibonacciho čísla, tentokrát v novom kabáte. Akonáhle spočítame novú hodnotu, zapamätáme si ju v poli `F`.

Koľko sme tým ušetrili? Pôvodná funkcia potrebovala na spočítanie F_{30} až 2 692 537 volaní. Akonáhle použijeme memoizáciu, zmenší sa tento počet na 59. (Vo všeobecnosti, na spočítanie F_N pre konkrétne $N > 1$ potrebujeme $2N - 1$ volaní funkcie `Fibonacci`.)

```
const NEVIEM = -1;
      MAX = 100;
var F : array[0..MAX] of longint;
    N,i : longint;

function Fibonacci(x : longint) : longint;
begin
   if F[x]=NEVIEM then begin
      if x=0 then F[x]:=0;
      if x=1 then F[x]:=1;
      if x>1 then F[x]:=Fibonacci(x-1) + Fibonacci(x-2);
   end;
   result := F[x];
end;

begin
   read(N);
   for i:=0 to N do F[i]:=NEVIEM;
   writeln( Fibonacci(N) );
end.
```

Program 2: Počítanie Fibonacciho čísla F_N s memoizáciou.

[2] Existujú však programovacie jazyky, kde sú funkcie naozaj funkciami v matematickom zmysle.
[3] USA Computing Olympiad, http://www.usaco.org/

1.0.4 Dynamické programovanie

TODO