INFORMATIKA

Redukce posloupnosti

REDAKCE

Jak jsme si v redakčních článcích vícekrát naznačili, posouzení programu jiného autora nás může vybavit cennými zkušenostmi. Tentokrát byla zpracovávána tato úloha:

⋆⋆⋆

V textovém souboru `Vstup.dat` je uložena posloupnost celých čísel v celkovém počtu do 30 000, která jsou oddělena mezerami. Je třeba vytvořit nový textový soubor `Vystup.dat`, který bude obsahovat tatáž čísla jako vstupní soubor, ale každé jen při jeho prvním výskytu. Např. pokud vstupní soubor obsahuje posloupnost

$$7, -6, 10, -8, 7, -1, 10, 10, -8, 8, 7, 9, 9, 9, -7, 2, 2,$$

pak výstupní soubor bude obsahovat posloupnost

$$7, -6, 10, -8, -1, 8, 9, -7, 2.$$

Program na obrazovku vypíše, kolik různých čísel se v posloupnosti opakovalo (v našem příkladu se opakuje 5 čísel: 7, 10, −8, 9 a 2, takže program vypíše: 5).

⋆⋆⋆

Problém byl předložen studentům středních škol. Jeden z nich zvolil řešení s užitím lineárního spojového seznamu. Jeho program *RedPosl-A* čte hodnoty H ze vstupního souboru a dále zjišťuje, jestli se načtená hodnota vyskytla ve vstupním souboru už dříve. Jestliže ne, vytvoří se další prvek spojového seznamu, kde se položí `C^.P := false`. Jestliže se při hledání duplicity hodnoty H dojde ke členu seznamu, kde se načtená hodnota H již vyskytla, pak se u tohoto členu zkoumá hodnota P; je-li *false*, tak se změní na true a o 1 se zvětší počítadlo opakujících se prvků, v případě *true* (tj. když se táž hodnota H ze vstupního souboru čte alespoň po třetí), neprovádí se nic a pokračuje se čtením dalšího čísla ze vstupu. Spojový seznam tak obsahuje právě ty hodnoty, které jsou požadovány do výstupního souboru, a počítadlo obsahuje číslo vyžadované zadáním ke zobrazení na obrazovce.

```
program RedPosl-A;

type
  TPoint = ^Posl;
    Posl = record
      H: Integer;  {číselná hodnota členu posloupnosti}
      P: Boolean;  {zda došlo k opakování hodnoty}
      D: Tpoint    {ukazatel na další prvek}
    end;
var
  Prvni, C: TPoint;
  F: Text;
  X, Opak: Integer;

function Hledej(X: Integer): Boolean;
begin
  C := Prvni;
  while (C^.H <> X) and (C^.D <> nil) do
```

```
    C := C^.D;
  Hledej := (C^.H = X)
end;

begin
  WriteLn('Redukce posloupnosti');
  Assign(F, 'Vstup.dat');
  Reset(F);
  Prvni := nil;
  Opak := 0;
  while not Eoln(F) do
  begin
    Read(F, X);
    if Prvni = nil then
    begin
      New(Prvni);
      Prvni^.H := X;
      Prvni^.P := false;
      Prvni^.D := nil
    end
    else
      if Hledej(X) then
      begin
        if not C^.P then
        begin
          Opak := Opak + 1;
          C^.P := true
        end
      end
      else
      begin
        New(C^.D);
        C := C^.D;
        C^.H := X;
        C^.D := nil;
        C^.P := false
      end
  end;
```

```
  Close(F);
  Assign(F, 'Vystup.dat');
  Rewrite(F);
  C := Prvni;
  while C <> nil do
  begin
    Write(F, C^.H, ' ');
    C := C^.D
  end;
  Close(F);
  WriteLn('Pocet opakujicich se cisel: ', Opak);
  ReadLn
end.
```

Uvedené studentské řešení úlohy je plně funkční, z hlediska způsobu naprogramování k němu ovšem můžeme mít řadu výhrad. Pomocná funkce *Hledej* je typickou ukázkou toho, jak by se funkce v programech psát správně neměly. V dobře navrženém programu mají funkce komunikovat se svým okolím prostřednictvím parametrů, hlavička funkce tvoří její interface vůči okolí a v těle funkce pak používáme výhradně jen lokální symboly. Zde uvedená funkce *Hledej* ovšem tyto zásady strukturovaného programování hrubě porušuje. Využívá hned dva globálně deklarované ukazatele – v ukazateli *První* dostává začátek zkoumaného seznamu a v ukazateli *C* vrací výsledek hledání. Program s touto funkcí pracuje sice bezchybně, kvůli nejasné komunikaci funkce s hlavním programem je však zbytečně nepřehledný a pokud bychom tímto stylem psali rozsáhlejší programový celek nebo pokud bychom třeba chtěli použít navrženou funkci jinde v programu či dokonce přenést ji do jiného programu, mohlo by se takovéto nevhodné používání globálních proměnných v těle funkce snadno stát zdrojem zbytečných a obtížně odhalitelných chyb.

Druhou nešikovností v zápisu programu je skutečnost, že se v hlavním cyklu opakovaně pro každé přečtené číslo testuje, zda je první nebo ne. Tyto testy naprosto zbytečně zdržují výpočet. Vhodnější by bylo ještě před zahájením cyklu přečíst ze vstupu první číslo a uložit ho do prvního prvku spojového seznamu, následně pak v cyklu číst ze vstupu zbývající čísla a pro ně již nemusíme netestovat, zda jsou první (neboť již víme, že nejsou).

Program B deklaruje pole A. Pracuje tak, že v cyklu vždy načte číslo, uloží je do pole A a zjišťuje, kolikrát byla právě načtená hodnota již dříve v poli A uložena. Jestliže dvakrát, přechází program na nové čtení, při němž se poslední prvek pole přepíše novou hodnotou ze vstupního souboru (tedy žádná hodnota není v poli A uložena více než dvakrát). Jestliže se právě načtená hodnota dosud nevyskytla, pošle ji program do výstupního souboru, jestli již je v poli uložena právě jednou, zvětší se o 1 počítadlo opakujících se hodnot. V obou těchto případech se zvýší index pole o 1 a pokračuje se v cyklu čtením další hodnoty ze vstupního souboru.

```
program RedPosl-B;

var
  F1, F2: Text;
  I, N, KolikSeOpakuje, Stejnych: Integer;
  A: array [1..30000] of Integer;

begin
  WriteLn('Redukce posloupnosti');
  Assign(F1, 'Vstup.dat');
  Reset(F1);
  Assign(F2, 'Vystup.dat');
  Rewrite(F2);
  N := 1;
  KolikSeOpakuje := 0;
  while not Eoln(F1) do
  begin
    Read(F1, A[N]);
    Stejnych := 0;
    I := 1;
    repeat
      if ((N > 1) and (A[N] = A[I]))
        then Inc(Stejnych);
      Inc(I)
    until ((I > N-1) or (Stejnych = 2));
    if Stejnych < 2 then
    begin
```

```
      case Stejnych of
        0: Write(F2, A[N], ' ');
        1: Inc(KolikSeOpakuje)
      end;
      Inc(N)
    end
  end;
  Close(F1);
  Close(F2);
  WriteLn('Pocet opakujicich se cisel: ', KolikSeOpakuje);
  ReadLn
end.
```

Rovněž v tomto druhém programu se jeho autor dopustil určitých nešikovností. Je hloupé a nevhodné v cyklu repeat opakovaně testovat podmínku `N>1`, když se v tomto cyklu hodnota proměnné N vůbec nemění. Také další podmínka `Stejnych<2` v zápisu programu je zbytečná, když po ní následuje pouze příkaz `case Stejnych of`. Případ, kdy proměnná *Stejnych* nabyla hodnoty 2, by bylo hezčí ošetřit jako třetí větev v tomto příkazu case.

Je nyní na čtenářích (resp. na studentech), aby posoudili výhody a nevýhody obou programů nebo aby si vytvořili svůj program řešící zadanou úlohu.

Oba předložené programy mají stejnou časovou složitost $O(N^2)$. Program A má vtipně řešený problém, jak zaznamenat první výskyt opakující se hodnoty a jak při zjišťování dalších výskytů stejné hodnoty se vyhnout zbytečnému testování. Na druhé straně se může zdát, že lineární spojový seznam je tu vlastně zbytečný, protože se další prvky připojují jen na jeho konci. Program B se také chce vyhnout zbytečnému testování a volí kompromis, že každou hodnotu ukládá nejvýše dvakrát. Zajímavé je sledovat práci programů v extrémních případech – jednak, když jsou ve vstupním souboru takřka všechny hodnoty navzájem různé, a jednak, když je v něm málo různých hodnot, ale velmi často se opakují.

(Autorkou úvodní ilustrace je Mgr. Jaroslava Čermáková z Hlinska v Čechách.)

Od teorie formálních jazyků k jednoduchému překladači

HASHIM HABIBALLA – ROSTISLAV FOJTÍK – EVA VOLNÁ

Přírodovědecká fakulta OU, Ostrava

(*Pokračování*)

3. Vyhodnocení postfixové notace výrazů

Rozeberme nyní jádro vyhodnocení výrazu v infixní formě. Toto jádro provádí kompilaci aritmetického výrazu v infixové (tedy přirozené) notaci do notace postfixové. Ta je vhodná pro zpracování pomocí počítače, např. vyhodnocení pomocí zásobníku. Aritmetický výraz v infixové (přirozené) notaci 5 + 3 * 2 lze pomocí této procedury přeložit na výraz v postfixové notaci 5 3 2 * +. Ta je konstruována tak, že místo tvaru, kde je operátor mezi operandy, má operátor až za oběma operandy.

Tento výraz lze pak pomocí zásobníku vyhodnotit tak, že čteme jednotlivé symboly a provádíme s nimi tyto dvě operace:

1. Je-li čtený symbol operandem, pak ulož operand na zásobník.
2. Je-li čtený symbol operátorem, pak vyber ze zásobníku posledních n operandů (kde n je arita operátoru; např. pro + je $n = 2$). Proveď operaci dle operátoru s vybranými operandy a výsledek ulož na zásobník.

Pro výraz 5 + 3 * 2 vezměme jeho postfixovou notaci 5 3 2 * + a vyhodnoťme jej s pomocí zásobníku.

Postfix	Akt. znak	Zásobník	Vybírané sym.	Operace
5 3 2 * +	5			
3 2 * +	3	5		
2 * +	2	3 5		
* +	*	2 3 5	2 3	2 * 3 = 6
+	+	6 5	6 5	6 + 5 = 11
		11		

Obsah zásobníku po přečtení slova je roven hodnotě výrazu. Postup by samozřejmě bylo možno zobecnit na složitější čísla nebo proměnné, ale vyžadovalo by to složitější struktury zásobníku.

Náš kód už tedy zbývá jen obohatit o proceduru provádějící vyhodnocení výrazu v postfixové notaci.

```
function VyhodnotPostfix(var postfix:string):integer;
{vyhodnotí postfixovou formu výrazu zásobníkem / vrátí číslo}
var Stack:array[1..1000] of integer;
    {zásobník čísel jako statické pole}
    headindex, i, val1, val2 :integer;
    {index vrcholu zásobníku, val1, val2 - operandy
     z vrcholu zás.}
begin
  headindex := 0; {nastav vrchol zásobníku na nulu}
  for i:= 1 to length(postfix) do {projdi celý postfix}
  begin
    ch := postfix[i]; {do pomocné proměnné dej aktuální znak}
    if (ch in ['0'..'9']) then  {je-li to číslice}
    begin
      Inc(headindex);  {vlož do zásobníku}
      Stack[headindex]:= ord(ch) - 48; {příslušné číslo}
    end
    else    {je-li to operátor}
    begin
      case ch of
        '+': begin
               val2 := Stack[headindex];    {vrchol -- druhý
                                             operand}
               val1 := Stack[headindex - 1];{o pozici níže
                                              - první op.}
               Dec(headindex); {odeber operandy a vlož
                                místo nich výsledek}
               Stack[headindex] := val1 + val2;
             end;
        '*': begin
               val2 := Stack[headindex];
               val1 := Stack[headindex - 1];
```

```
                Dec(headindex);
                Stack[headindex] := val1 * val2;
              end;
      end;
    end;
  end;
  VyhodnotPostfix := Stack[headindex]; {na konci je na
                                        vrcholu výsledek}
end;
```

Pozn. Čtenář možná považuje za nevhodné použití globální proměnné *ch*, se kterou se pracuje zejména v syntaktické analýze. Máme proto ale jeden pádný argument. Procedury analýzy se volají rekurzivně navzájem a to mnohdy do velké hloubky vnoření (v závislosti na struktuře výrazu). Pokud bychom použili lokální proměnné resp. parametry procedur, znamenalo by to alokaci paměti pro tyto proměnné. Takových alokací bychom udělali velmi mnoho podle počtu volání procedur pro neterminály. To by algoritmus zatížilo časově i prostorově.

4. Závěr

Na poměrně detailním textu jsem si ukázali, jak lze zapisovat syntaxi jazyka (výrazů), analyzovat je, překládat do jiných forem a také vyhodnocovat. Chtěli jsme ukázat, že zdánlivě složité úkoly lze dělat přehledně, ilustrativně a hlavně efektivně z hlediska časové a prostorové složitosti. Ukázali jsme, že teorie formálních jazyků a automatů je nám blíž než bychom čekali. Úloha sestrojit jednoduchý analyzátor nebo překladač bezkontextového jazyka může potkat každého informatika – vždyť strukturované vstupy jsou součástí mnoha informačních systémů včetně různých databázových aplikací. Lze to hezky ilustrovat na příkladu ze života. Jeden z našich studentů, který rozhodně nebyl nadšenec do teoretické informatiky, nás asi rok po státních zkouškách navštívil. Uvedl, že nejdůležitější znalostí, kterou využil z naší školy v softwarové firmě, nebyla žádná moderní technologie. Byla to znalost tvorby analyzátorů bezkontextových jazyků! Jeho úkolem bylo přijímat na vstupu jistý strukturovaný vstup informací zaslaných podle síťového protokolu a překládat jej do jiného formátu. Metoda rekurzivního sestupu není sice tou nejefektivnější. Spíše se používají přímo automatizované nástroje na tvorbu kódu analyzátoru. Má ale výhodu v přehlednosti, jednoduchosti a kód lze lehce upravit. Pokud

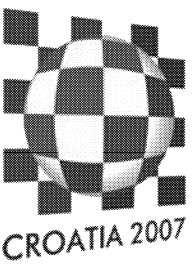
bychom chtěli třeba místo číslic používat v aritmetických výrazech nějakou složitější strukturu, není problém změnit pouze tuto malou část gramatiky a kódu a zbytek se nezmění. A možná nejdůležitější výhoda je, že si jej programujete zcela sami – tudíž máte nad kódem plnou kontrolu.

Doufáme, že tento exkurz do světa překladačů je pro čtenáře poučný nejen teoreticky, ale i programátorsky. Tvorba složitějších překladačů samozřejmě obsahuje další důležité prvky, které zde nezmiňujeme (např. tabulky symbolů).

Literatura je uvedena v 1. části příspěvku.

ZPRÁVY

19. mezinárodní olympiáda v informatice



Ve dnech 15. – 22. 7. 2007 se konal v chorvatském hlavním městě Zagreb 19. ročník Mezinárodní olympiády v informatice (IOI 2007 – *International Olympiad in Informatics*). Soutěže se zúčastnilo 285 studentů ze 77 zemí celého světa. Většina zemí využila možnosti vyslat na IOI maximální povolený počet čtyři soutěžící, z několika nových účastnických zemí přijela reprezentace menší.

Reprezentační družstvo České republiky bylo sestaveno na základě výsledků dosažených soutěžícími v ústředním kole 56. ročníku Matematické olympiády – kategorie P (programování). Naše družstvo pro IOI 2007 mělo následující složení:

Pavel Klavík, absolvent gymnázia J. Ressla v Chrudimi, *Miroslav Klimoš*, student gymnázia M. Koperníka v Bílovci, *Josef Pihera*, absolvent gymnázia ve Strakonicích, *Roman Smrž*, student gymnázia E. Krásnohorské v Praze.

Vedoucími české delegace byli jmenováni *doc. RNDr. Pavel Töpfer, CSc.* a *Bc. Petr Škoda*, oba z Matematicko-fyzikální fakulty Univerzity Karlovy v Praze. Vedle této oficiální šestičlenné delegace se 19. mezinárodní olympiády v informatice zúčastnil z České republiky ještě *Mgr. Martin Mareš*, rovněž pracovník MFF UK v Praze, a to z titulu své funkce člena Mezinárodního vědeckého výboru IOI.

Naši studenti měli možnost připravit se na soutěž nejen samostatným studiem, ale zúčastnili se i dvou přípravných akcí. V červnu jsme pro vybrané reprezentanty z České republiky, Polska a Slovenska uspořádali na Matematicko-fyzikální fakultě UK v Praze tradiční týdenní společné česko-polsko-slovenské přípravné soustředění před IOI (CPSPC – *Czech-Polish-Slovak Preparation Camp*). Na začátku července jsme pak využili skutečnosti, že jsme letos byli pořadateli 14. ročníku Středoevropské olympiády v infor-