Od teorie formálních jazyků k jednoduchému překladači

HASHIM HABIBALLA - ROSTISLAV FOJTÍK - EVA VOLNÁ

Přírodovědecká fakulta OU, Ostrava

V dnešní době se ve výuce informatiky na střední škole bohužel často projevuje trend klesající odborné úrovně. Jde zejména o výuku orientovanou na aplikační programy bez důrazu na principy informatiky. Ty zahrnují především algoritmy a teoretické disciplíny zkoumající související pojmy jako je teorie formálních jazyků a automatů, formální logika nebo vyčíslitelnost a složitost. Tento článek navazuje na některé již uveřejněné materiály např. [6], [7], [8].

Úloha kterou přinášíme má hodně společného s typickou úlohou v informatické praxi a tou je překlad resp. interpretace nějakého zdrojového kódu (programu) do jiného jazyka (včetně optimalizací vzniklého kódu). S tímto se setkáváme prakticky každodenně (otázka je zda si to uvědomujeme). Samozřejmě nejprve informatika asi napadne úloha typu - programuji řešení zadaného úkolu v nějakém programovacím jazyce (např. Pascalu) a pak použiji překladač, který mi vyrobí kód v nějakém počítačově použitelném formátu (třeba exe soubor pro počítač typu PC pod platformou MS-DOS). Nemusí však jít jen o tento "programátorský" případ. Aniž si to mnohdy uvědomujeme, pokud prohlížíme libovolnou webovou stránku, tak náš webový prohlížeč získává z daného URL kód v jazyce HTML. Ten by samozřejmě pro člověka asi nebyl na čtení to nejlepší. Proto prohlížeč musí PŘELOŽIT a interpretovat tento kód, tak aby byl pro člověka příjemný. Jde o zobrazení různých velikostí písma, tabulek, obrázků, pozadí stránek atd. Jakkoliv to vypadá naprosto přirozeně, při zkoumání principů a metod programů pro tuto činnost zjistíme, že to není až tak triviální činnost a souvisí neoddělitelně právě s teorií formálních jazyků. I zdánlivě jednoduchá součást tohoto procesu je bez dobré znalosti pojmů a postupů, které přináší teorie formálních jazyků a automatů, takřka neřešitelný úkol.

Touto jednoduchou úlohou, se kterou se nyní důkladně seznámíme a ukážeme si její obecně použitelné řešení, je kontrola, překlad a vyhodnocení aritmetického výrazu. Úlohu si samozřejmě pro naše účely výkladu algoritmů zjednodušíme oproti klasickým programovacím jazykům, kde můžeme používat reálná čísla v různých tvarech nebo číselné proměnné. To ale nijak nesnižuje obecnost principů, které se zde naučíme. Aritmetickým výrazem budeme myslet řetězce s číslicemi $0-9$, operace sčítání a násobení $+, *$, lze používat i pomocné symboly závorek $(,)$. Na místě operandu může stát struktura obecně stejného typu jako jsme právě definovali (výraz lze vnořit do výrazu). Příkladem budiž velmi jednoduchý výraz $5+3*2$, jehož hodnotu lehce spočítáme. Uvědomíme si prioritu operátorů

a nejprve spolu vynásobíme $3 * 2 = 6$ a tento mezivýsledek pak použijeme při operaci s nejnižší prioritou $5 + 6 = 11$. Tato jednoduchost je však jen zdáním, uvědomme si, že počítač pracuje s datovými strukturami bez "lidského vidění". To zahrnuje velmi rychlé vyhodnocení hierarchie formule. Navíc tento postup předpokládá, že ve výrazu není žádná syntaktická chyba (např. chybějící párová závorka). Takovou chybu by musel člověk rovněž odhalit. Zobecnění na složitější operandy jako jsou reálná čísla není principiálně problém - stačí během analýzy nevytvářet jen znaky jako výstup, ale struktury obsahující přímo reálné číslo. Stejně tak lze navázat i jinými operacemi.

Takovéto výrazy jsou nedílnou součástí každého vyššího procedurálního programovacího jazyka (řešení všech problémů obvykle stavíme na správně formulovaných podmínkách a aritmetických výpočtech). Samozřejmě úloha má ještě důležitou část, která je již za hranicí tohoto článku - jsou jím různé optimalizace kódu, např. u logických podmínek lze vyhodnotit, zda formule není platná či nesplnitelná, případně ji značně zmenšit. Ale jak tuto činnost překladače - programy - realizují? Uvažujeme-li jako lidé s inteligencí a schopností číst text nejen lineárně (po znacích), pak v našem mozku probíhají pochody přímo na počítači realizovatelné velmi těžko. Na našem výrazu si ihned uvědomíme, v jakém pořadí probíhá vyhodnocení jednotlivých operací. Nejvyšší prioritu mají číslice a závorky, pak postupujeme ve vyhodnocování podvýrazů podle struktury výrazu a priorit operátorů (nejdříve násobení a pak sčítání). Bohužel dnešní počítače a klasické procedurální programovací jazyky takovou inteligencí nedisponují a uvědomme si například, že výraz musíme zadat jako nějaký řetězec znaků, který pak můžeme číst po znacích. Je také dobré si uvědomit, že aby program (překladač) měl pro praktickou práci smysl, musí být efektivní - mít dobrou časovou složitost [7]. Určitě bychom nechtěli mít překladač, který řetězec prochází opakovaně a stále se vrací k tomu, co už jednou přečetl (takový algoritmus byl představen již dříve v MFI, i přes jeho nespornou jednoduchost je však pro profesionální programy nepřijatelný). V obecném případě může být program o desítkách nebo stovkách tisíc znaků a tudíž opakované procházení v závislosti na jeho velikosti je velmi neefektivní.

Sami si představte, že program v Pascalu vám překladač kompiluje hodiny, dny nebo měsíce, aby nakonec oznámil, že v něm máte na konci syntaktickou chybu. Takový přístup by vás asi napadl, budete-li se snažit vyhodnocovat výraz nějakým triviálním algoritmem. Tedy postupně vyhodnocovat podvýrazy a stále znovu procházet řetězec. To je však z hlediska optimality zcela nepřijatelné - my se přece snažíme ušetřit každou operaci a tedy opakované procházení je zcela proti tomuto cíli. Budeme tedy muset zřejmě zapojit nějaké "know-how" a v našem případě to budou právě prostředky a algoritmy, které nám dává teorie formálních jazyků a překladačů.

1 Definice jazyka pomocí gramatiky - Backusovy-Naurovy formy

Aby byl schopen počítač pracovat s aritmetickým výrazem (resp. s jakýmkoliv jiným syntaktickým elementem v libovolném zdrojovém kódu), musíme nějakým vhodným formálním prostředkem zapsat definici tohoto výrazu. Tyto výrazy vlastně tvoří specifický

jazyk a tedy teoretická informatika nám dává prostředek ve formálních gramatikách, o kterých jsme si již v MFI mohli přečíst článek [6]. Pro definice struktur programovacích jazyků se velmi dobře hodí takzvaná Backusova-Naurova forma. Je podobná bezkontextovým gramatikám (má terminální symboly, neterminální symboly, pravidla pro přepis neterminálů). Navíc zjednodušuje zápis obvyklých technik jako je opakování nějakého řetězce (např. přičítání podvýrazů v aritmetických výrazech se může dělat opakovaně). U bezkontextových gramatik bychom toto museli řešit krkolomně pomocí jakési "rekurze". Bezkontextová gramatika (BKG) je jedním z možných způsobu zápisu syntaxe jazyků. Syntaxí zde rozumíme jejich jazykovou strukturu. Umožňuje totiž vyjádřit většinu technik, které například u programovacích jazyků používáme. Jde o alternativu několika možností, opakování stejného jazykového výrazu a hlavně vnořování celých rozvětvených struktur mezi sebou. Poslední zmiňovaná technika je právě onou technikou, kterou neumíme vyjádřit pomocí jazyků regulárních, ale teprve pomocí bezkontextových jazyků. Zkusme si představit velmi omezenou část nějakého programovacího jazyka - například strukturu aritmetického výrazu. Principiálně je většina jiných struktur velmi podobných (např. sekvence příkazů je analogická opakovanému sčítání podvýrazů!).

Příklad:

Sestrojme BKG pro jazyk složený z aritmetických výrazů s operandem x, operacemi +,* a umožňující vnořovat další podvýrazy stejného typu pomocí symbolů závorek (,). Kupříkladu se může jednat o výraz: $x * (x + x + x)$

Gramatiku sestrojíme hierarchicky - tedy aby byla rozlišena priorita operátorů a využijeme "rekurzivní" vlastnosti přepisu neterminálu, abychom docílili možnosti generovat opakovaně sčítání a násobení.

$G = (\{S, A, B\}, \{x, *, +, (,)\}, S, P)$

$P : S \rightarrow^1 A + S, S \rightarrow^2 A$ (opakovaný přepis na S nám umožní generovat libovolně mnoho sčítání struktury A)
$A \rightarrow^3 B * A, A \rightarrow^4 B$ (opakovaný přepis na A nám umožní generovat libovolně mnoho násobení struktury B)
$B \rightarrow^5 (S), B \rightarrow^6 x$ (rekurzivním přepisem na S můžeme vnořit libovolně mnoho podvýrazů zcela stejné struktury jako výraz sám do závorek anebo ukončit generování operandem x) (Pozn.: index u symbolu určuje pomocné číslo pravidla)

V této gramatice pak lze snadno generovat například výše uvedený výraz $x*(x+x+x)$:

$S \Rightarrow^2 A \Rightarrow^3 B * A \Rightarrow^6 x * A \Rightarrow^4 x * B \Rightarrow^5 x * (S) \Rightarrow^1 x * (A + S) \Rightarrow^4 x * (B + S) \Rightarrow^6 x * (x + S) \Rightarrow^1 x * (x + A + S) \Rightarrow^4 x * (x + B + S) \Rightarrow^6 x * (x + x + S) \Rightarrow^2 x * (x + x + A) \Rightarrow^4 x * (x + x + B) \Rightarrow^6 x * (x + x + x)$

(Pozn.: index u symbolu $\Rightarrow^i$ určuje pomocné číslo pravidla)

Zmiňovaným a hlavně v praxi ještě více využívaným způsobem zápisu syntaxe jazyka je takzvaná Backusova-Naurova forma (BNF). Jde o zápis podobný bezkontextové gramatice, ale přitom bližší spíše programátorům, resp. praxi.

BNF obsahuje podobně jako BKG neterminály, které se uvádějí do úhlových závorek a přepisují skrze symbol := na řetězce terminálních a neterminálních symbolů. Jde tedy o pravidla tvaru:

$\langle X \rangle ::= \alpha_1 | \ldots | \alpha_n$

Pro přehlednější zápis je však ještě lepší modifikace BNF zvaná EBNF (Extended BNF) - rozšířená BNF, která zjednodušuje zápis opakovaně používaných, příp. podmíněně vyskytujících se výrazů. Umožňuje následující zápisy:

$\{\alpha\}$ - znamená, že výraz se vyskytuje v libovolném počtu (ekvivalent operace iterace)
$\{\alpha\}_n^m$ - znamená, že výraz se vyskytuje v počtu nejméně n a nejvýše m (ekvivalent operace mocniny od n do m)
$[\alpha]$ - znamená, že výraz se může a nemusí na daném místě vyskytnout - je to ekvivalentní zápisu $\{\alpha\}_0^1$

Příklad:

Gramatika z předchozího příkladu by v BNF mohla být zapsána například takto:

$\langle aritmetickyvyraz+ \rangle ::= \langle aritmetickyvyraz* \rangle \{+ \langle aritmetickyvyraz* \rangle\}$
$\langle aritmetickyvyraz* \rangle ::= \langle operand/podvyraz \rangle \{* \langle operand/podvyraz \rangle\}$
$\langle operand/podvyraz \rangle ::= (\langle aritmetickyvyraz+ \rangle) | x$

BNF umožňuje přehledný zápis a navíc i jednoduchý přechod k některým typům syntaktických analyzátorů. S pomocí BNF je zapsána například celá gramatika jazyka Pascal v učebnici [11]. Příkladem může být deklarace podmíněného příkazu:

$\langle podminenyprikaz \rangle ::= if \langle booleovskyvyraz \rangle then \langle prikaz \rangle [else \langle prikaz \rangle]$

2 Metoda rekurzivního sestupu pro syntaktickou analýzu

Prvním důležitým úkolem při překladu z nějakého zdrojového jazyka do cílového je především syntaktická analýza - tedy kontrola, zda je text ve zdrojovém jazyce správně zapsán. V nejjednodušším případě pouhé kontroly typu ANO/NE (program je správně/není správně syntakticky zapsán) se jedná o takzvanou syntaktickou analýzu (dále budeme zkracovat SA). V anglicky psaných zdrojích se setkáte spíše s jednoslovným označením "parsing". O daném postupu - algoritmu jak tuto SA provést, pak hovoříme jako syntaktickém analyzátoru (anglicky "parser").

Velice oblíbenou, jednoduchou a přehlednou metodou vedoucí přímo k hotovému a dobře čitelnému analyzátoru v libovolném strukturovaném programovacím jazyce je metoda rekurzivního sestupu (Recursive Descent Parsing) [2]. Metoda je založena na principu analýzy "shora dolů", tedy se snažíme hledat odvození slova podle dané gramatiky od počátečního neterminálu v hierarchii směrem "dolů".

Metoda rekurzivního sestupu spočívá v konstrukci procedur strukturovaného programovacího jazyka přesně dle Backusovy-Naurovy formy (BNF), kde je každému neterminálu přiřazena jedna procedura a je volána procedura GetChar (načítající vždy následující symbol slova) před každým terminálem. Výskyt neterminálu v pravidle je v proceduře nahrazen

voláním příslušné procedury (i rekurzivním). Iterace a podmínka v Backus-Naurově formě je nahrazena jednoduše jejich programátorskými protějšky. Vyžaduje se, aby jazyk byl definován tzv. LL(k) gramatikou (detailní podmínky v [4], pro účely tohoto článku budou takovéto gramatiky vytvořeny). Z hlediska časové složitosti je použitelná pro jednoduché gramatiky z praxe a zejména je její výhodnou vysoká čitelnost kódu ve vztahu k výchozí gramatice. Navíc existuje modifikace tzv. packrat parser, která pro omezenou třídu takzvaných parsing expression grammars pracuje v **lineární čase**. Lineární časová složitost je oproti naivním metodám analýzy, které jsou typicky kvadratické složitosti, o třídu lepší. Také je tento postup flexibilní, protože umožňuje kdykoliv změnit a přidat syntaktický element bez nutnosti měnit celý kód, ale pouze dotčenou část gramatiky. Například změna struktury číslo z celého čísla na reálné znamená pouze změnu procedury reprezentující tento element. Podívejme se nyní na příklad gramatiky pro generování aritmetických výrazů se sčítáním, násobením, číslicemi a vnořenými závorkovanými strukturami. Vyhodnocování takového výrazu strojově je pak velmi jednoduché pomocí tzv. postfixové notace (reverzní polská notace). V tomto způsobu zápisu platí, že operátory se vyskytují až za jeho operandy. Je to tedy odlišný způsob oproti notaci používané člověkem, která se nazývá infixní. V infixní formě se zapisuje první operand, operátor a teprve pak druhý operand. Pro vyhodnocení postfixové notace nám postačí jen datová struktura typu zásobník a nijak to neovlivní **lineární** časovou složitost celého procesu včetně syntaktické analýzy.

Příklad:

Nejprve sestrojme mírně modifikovanou gramatiku (oproti příkladu z minulé kapitoly) v Backusově-Naurově formě:

$\langle Vyraz\rangle ::= \langle Term\rangle\{+\langle Term\rangle\}$
$\langle Term\rangle ::= \langle Faktor\rangle\{*\langle Faktor\rangle\}$
$\langle Faktor\rangle ::= (\langle Vyraz\rangle)|0|1|2||9$

Nyní se schématicky pokusíme ukázat (nejde o zcela hotový kód, ale o jeho fragmenty po částech, které byly dohromady úspěšně testovány), jak bychom sestrojili SA metodou rekurzivního sestupu pro tuto gramatiku v jazyce Pascal. Tento kód, pak umožňuje nejen SA, ale i detekci možných chyb. Nejprve sestrojíme proceduru, která zapouzdřuje celou činnost SA. Její hlavička může vypadat například takto:

```
program Preklad;

var ch:char;              {aktuální zpracovávaný znak}

    infixProg, postfixProg:string;     {globální prom. pro uchování výrazu}
    errProg, posProg, infixpos:word;
    value:integer;                   {globální prom. čísla chyby, hodnota výrazu}

procedure SyntaktickaAnalyza(infix:string;var err,pos:word;
```

```
                                  var postfix:string);
  {procedura analyzuje aritmetický výraz infix, postfix obsahuje
   postfixovou notaci vhodnou pro vyhodnocení zásobníkem
   err obsahuje číslo chyby, pos obsahuje pozici, kde analýza skončila}

procedure Term;forward;
procedure Faktor;forward;
```

Používají se proměnné infixpos (pozice aktuálně čteného znaku ze vstupu), ch (aktuální znak). Analyzátor dále obsahuje nezbytný lexikální analyzátor pro načítaní jednotlivých symbolů (v našem zjednodušeném případě jde o jednoznakové symboly). Lexikální analýza je realizována procedurou GetChar, která ukládá znak do proměnné ch a případně provede detekci chybové situace err=2, pokud načteme zcela nepřípustný znak. V rámci SA je pak otázka jen přidat několik vhodně umístěných přiřazení do výstupní proměnné postfix. Ta obsahuje do postfixové notace přeložený výraz, který lze velmi jednoduše algoritmicky vyhodnotit.

```
procedure Getchar;       {čte znak z infixu do proměnné ch}
begin
  if err=0 then
    begin
      Inc(infixpos);
      if infixpos<=Length(infix) then ch:=infix[infixpos] else ch:=#0;
      ch:=Upcase(ch);
      if not((ch in ['0'..'9'])or(ch in ['(',')','*','+',#0])) then err:=2;
         {ošetření nežádoucích znaků}
    end;
end;
```

Jádrem analyzátoru jsou jednotlivé rekurzivní procedury Výraz (sčítaní), Term (násobení), Faktor (číslice, vnořený závorkovaný výraz). Výraz přesně podle BNF buď volá podřízený Faktor nebo čte terminální symboly.

```
procedure Vyraz;               {výraz s nižší prioritou}
begin
  if err=0 then
    begin
      Term;
      while (ch='+') do
        begin
                 Getchar;       {sčítání}
                 Term;
         postfix := postfix + '+';
```

```
        end;
    end;
end;

procedure Term;               {výraz s vyšší prioritou}
begin
  if err=0 then
    begin
      Faktor;
      while (ch='*') do
        begin
                 Getchar;           {násobení}
                 Faktor;
         postfix := postfix + '*';
        end;
    end;
end;
```

A dále musíme sestrojit proceduru pro Faktor, která bude mít vzhledem k jinému charakteru přepisovaného řetězce i jiný kód.

```
procedure Faktor;   {synt. analýza operandu}
begin
  if err=0 then
    begin
      case ch of
      '0'..'9':
          begin
            postfix:=postfix+ch;                  {anal. číslic}
            Getchar;
          end;
      '(':begin
            Getchar;
                        {analýza výrazu se závorkou}
            Vyraz;
            if (ch<>')')and(err=0) then err:=4   {chyba- není ukončen závorkou}
            else if err=0 then
              begin
                Getchar;
              end;
          end;
      else if err=0 then err:=5;    {nebyl detekován ani vyraz,ani číslice}
      end;
```

```
    end;
end;
```

Faktor tedy rozlišuje dvě situace - buď jde o číslici nebo jde o výraz začínající závorkou a ukončený opačnou závorkou. Logicky tedy můžeme odhalit další dvě chyby (err=4, když chybí závorka, err=5, když není detekován ani výraz ani číslice). Postfixová notace se generuje postupně - každá číslice se vloží okamžitě po načtení, znak operátoru se přidá, až po zpracování podstromu. Nebyl by problém se zcela vyhnout tvorbě postfixu a stejným rekurzivním principem přímo vyčíslovat postupně hodnotu výrazu (procedury Faktor, Term a Výraz by pak měly návratovou hodnotu rovnou výsledku po aplikaci všech operací na dané úrovni podvýrazu).

Samozřejmě, že chybové detekce by mohly odhalit ještě další problematické konstrukce, např. skončení nejvyššího volání procedury Výraz před přečtením posledního znaku apod. Vlastní tělo procedury pro SA, pak provede volání počátečního neterminálu a odhalí některé chyby dodatečně po přečtení výrazu, např. že sice skončila nejvyšší procedura Vyraz, ale ještě ve vstupu jsou nějaké znaky.

```
begin
  err:=0;           {inicializace prom.}
  infixpos:=0;
  postfix:='';
  Getchar;
  Vyraz;            {zavolání syntaktické analýzy výrazu}
  if (ch=')') and (err<>5) then
    begin
      err:=6;
      pos:=0;
    end;     {ošetření přebytečné pravé závorky}
  if (ch<>#0) and (err=0) then err:=1;  {ošetření konce kompilace-není konec
                                          infixu}
  pos:=infixpos;               {nastavení návratových hodnot}
end;
```

Nakonec můžeme v hlavním programu tuto proceduru použít.

```
begin
  infixProg := '5+3*(2+5)*2'; {zadej si vlastní výraz}
  Writeln('Vstupni infixni vyraz:', infixProg);
  SyntaktickaAnalyza(infixProg, errProg, posProg, postfixProg);
  Writeln('Doslo k chybe:', errProg);
  if errProg<>0 then exit;
  Writeln('Postfixova notace:', postfixProg);
  value := VyhodnotPostfix(postfixProg);
```

```
  Writeln('Hodnota vyrazu:', value);
  readln;
end.
```

Ilustrujme nyní průběh výpočtu procedury Výraz na výrazu $5 + 3 * 2$.

Infixová notace	**Aktuální znak**	**Aktuální procedura**	**Návrat do procedury**
5 + 3 * 2	5	Výraz	
5 + 3 * 2	5	Term	
+ 3 * 2	+	Faktor	Term
+ 3 * 2	+	Term	Výraz
3 * 2	3	Výraz (+)	
3 * 2	3	Term	
* 2	*	Faktor	Term
2	2	Term (*)	
		Faktor	Term (*)
		Term (*)	Výraz (+)
		Vyraz (+)	

Obrázek 1: Průběh rekurzivního sestupu

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.

Nepřečtená část	Aktuální znak	Zásobník	Vybírané symboly	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		

Obrázek 2: Vyhodnocení výrazu s pomocí zásobníku

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

[1] ČEŠKA, M., RÁBOVÁ, Z. Gramatiky a jazyky. VUT Brno, 1992 http://www.fit.vutbr.cz/study/courses/TI1/public/gj-1.3.pdf (2003).

[2] DVOŘÁK, S. Dekompozice a rekurzivní algoritmy. Grada Praha, 1992.

[3] HABIBALLA, H. Regulární a bezkontextové jazyky I. Ostravská univerzita, Ostrava, 2003. http://www1.osu.cz/home/habibal/kurzy/xrab1.pdf

[4] HABIBALLA, H. Regulární a bezkontextové jazyky II. Ostravská univerzita, Ostrava, 2005. http://www1.osu.cz/home/habibal/kurzy/rabj2.pdf

[5] HABIBALLA, H. Překladače. Ostravská univerzita, Ostrava, 2006. http://www1.osu.cz/home/habibal/kurzy/xprek.pdf

[6] HABIBALLA, H. Formální jazyky a automaty. In Matematika-fyzika-informatika. 05-06/2004, roč.13.

[7] HABIBALLA, H.; KMEŤ, T. Vyčíslitelnost a složitost. In Matematika-fyzika-informatika. 02-03/2005-06(15).

[8] HABIBALLA, H.; PAVLISKOVÁ, L.; KMEŤ, T. Logika. In Matematika-fyzika-informatika. 07-09/2005-06(15).

[9] CHYTIL, M. Automaty a gramatiky. Matematický seminář SNTL Praha, 1984.

[10] JANČAR, P. Teorie jazyků a automatů. VŠB TU Ostrava, http://www.cs.vsb.cz/jancar/ (2003).

[11] JINOCH, J., MULLER, K., VOGEL, J. Programování v jazyce Pascal. SNTL 1988, Praha.

Kontaktní adresa:

Hashim Habiballa
katedra informatiky a počítačů
Přírodovědecká fakulta Ostravské Univerzity
30.dubna 22, 701 03 Ostrava 1,
tel.: +420596160241, e-mail: habiballa@volny.cz, www: http://www.volny.cz/habiballa/