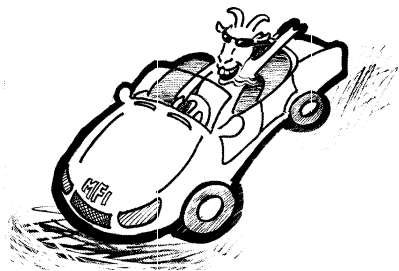


# INFORMATIKA

## Úloha Montyho Halla

JIŘÍ MAZUREK

Gymnázium Orlová



Na počátku 90. let 20. století byla ve Spojených státech velmi populární show Montyho Halla. V programu jmenovaného konferenciéra se mohli diváci zúčastnit soutěže o automobil, a úkol, který přitom museli splnit, byl zdánlivě jednoduchý: ve studiu se nacházely troje zavřené dveře. Za jedněmi z nich se skrýval automobil, za dalšími dvěma kozy (živá zvířata).

Soutěžící si měl vybrat „správné“ dveře (tedy ty skrývající auto) a postavit se před ně. Jakmile tak dotyčný učinil, otevřel Monty Hall některé ze zbývajících dvou dveří a ukázal, že je za nimi koza. Načež učinil soutěžícímu „dábelskou“ nabídku: „Chcete si ponechat výhru za dveřmi, které jste si vybral, nebo svou volbu změníte a vyberete si cenu za posledními (zavřenými) dveřmi?“

Tak a teď se rozhodněte i vy: je lepší svou volbu přehodnotit? Nebo je lepší trvat na původním výběru? A není to vlastně úplně jedno?

Se stejnou otázkou se obrátil jeden ze čtenářů časopisu Parade na Marilyn vos Savant, ženu s údajně nejvyšším IQ na světě. Paní vos Savant odpověděla, že lepší je svou volbu změnit, protože v tom případě je pravděpodobnost výhry 2:3.

Po tomto článku se rozpoutala obrovská diskuse na internetu i mezi

vědci a matematiky (počet diskutujících přesáhl číslo 10 000!). Většina příspěvků tvrdila, že se vos Savant mýlí, a že pravděpodobnosti výhry jsou u obou voleb 50 % na 50 % (je tedy jedno, jestli svou volbu změníme nebo ne).

Jeden z dopisů např. konstatoval, že: „... **všeobecný nedostatek matematického povědomí u veřejnosti mi působí velké starosti. Buďte prosím nápomocna a přiznejte svůj omyl...**“, nebo: „*Jsem zděšený, že ani poté, co Vás opravili přinejmenším tři matematikové, stále nechápete svou chybu...*“

Paní vos Savant ovšem měla pravdu! Jak je to možné?

Zkusme se na problém podívat znovu: soutěžící vybírá ze tří dveří, a jen jedny dveře ukrývají výhru (auto). Pravděpodobnost výhry je tedy rovna 1:3.

Soutěžící má ovšem na výběr dvě možnosti, jak postupovat dále:

**1. Soutěžící trvá na své volbě:** pravděpodobnost výhry auta je 1:3, protože situace se ničím neliší od případu, *kdy by soutěžící žádnou volbu neměl* (a prostě by tipoval jednu ze tří možností). Dá se to ukázat i následovně: pokud si soutěžící vybral první kozu, dostane první kozu. Pokud si vybral druhou kozu, dostane druhou kozu. Pokud si vybral auto, dostane auto. Pořád máme **jednu** příznivou možnost ze tří možností.

**2. Soutěžící svou volbu změní:** jestliže si původně vybral první kozu, dostane auto. Jestliže si vybral druhou kozu, dostane auto. Jestliže si vybral auto, dostane kozu (první nebo druhou). Vidíme, že ze tří možností *dvě* vedou k výhře automobilu, a tedy pravděpodobnost výhry je 2/3!

**Závěr:** Při trvání na své původní volbě soutěžící vyhraje jen v případě, že si vybral automobil. Při změně volby vítězí soutěžící v případě, že si vybral první nebo druhou kozu (tyto možnosti jsou dvě!), a prohraje jen tehdy, pokud si na počátku vybral auto.

*Lepší je tedy svou volbu změnit.*

Monty Hallova úloha se hodí ke zpestření hodiny matematiky, proto ji vřele doporučuji. Možná budete sami překvapeni, jak k řešení přistoupí žáci. Na adrese wikipedie (uvedené níže) se pak může případný zájemce dozvědět například to, jak by se úloha řešila, pokud by bylo dveří více a pokud by měli soutěžící i Monty Hall více možností pro své rozhodování; nebo že existuje velmi podobný *Problém tří karet...*

(Autorkou úvodní ilustrace je Mgr. Jaroslava Čermáková z Hlinska v Čechách. Je autorkou úvodních ilustrací v informatické části MFI také v č. 1 a 2. Tato informace tam omylem nebyla uvedena, a proto se autorce omlouváme.)

## L i t e r a t u r a

- [1] *Haddon, Mark*: Podivný případ se psem, Argo 2003.
- [2] [http://en.wikipedia.org/wiki/Monty\\_Hall\\_problem](http://en.wikipedia.org/wiki/Monty_Hall_problem)

# K úloze Montyho Halla

STANISLAV TRÁVNÍČEK

Přírodovědecká fakulta UP, Olomouc

Krátce ještě doplním výklad o úloze Montyho Halla z [1]. I když pomineme pro úlohu nepodstatné okolnosti, jak nás i s nimi věrně seznámil Jiří Mazurek, pořád tu zůstává úloha hodná pozornosti zejména tím, že nabízí řešení "pro laickou veřejnost" překvapivé a snad i málo přesvědčivé. Vysvětlení v článku [1] je sice jistě dostatečné, ale přesto si asi mnozí rádi potvrdí výsledek i jinak.

Existuje totiž jednoduchá metoda, která v mnoha případech je velmi přesvědčivá – a to provést simulaci požadovaných akcí a výsledky posoudit z hlediska statistického. Před mnoha lety jsme v MFI (viz [2]) takto vyšetřovali pravděpodobnost toho, že ze tří kousků špejle dvakrát náhodně zlomené lze sestavit trojúhelník.

Podobně zde můžeme pozvat počítač, ať simuluje danou úlohu a vyjádří se pak ke správnosti výsledku předkládaného v [1], tj. že při změně dveří je pravděpodobnost získání auta  $2/3$ . Nabízíme čtenáři takový program; je sestaven tak, aby celou situaci simuloval co nejnázorněji. Program je bohatě komentován, takže už další naše komentáře nepotřebuje.

```

program MONTY;
uses Crt;

var ZaDvermi: array [0..2] of Integer; {Dvere 0, 1, 2,
                                         za nimi auto = 1, koza = 0}
    N, K, CetnostZustal, CetnostZmenil: LongInt;
    R, C, Q, I: Integer;

begin
  ClrScr;
  Randomize;
  WriteLn('Monty Hall');
  Write('Pocet akci: ');
  ReadLn(N);
  CetnostZustal := 0; {vynulovani pocitadel ziskanych aut,}
  CetnostZmenil := 0; {kdyz by u zvolenych dveri zustal}
                      {nebo je zmenil}

  K := 0;
  repeat
    {zacatek cyklu akci}
    Inc(K);
    for I := 0 to 2 do
      ZaDvermi[I] := 0; {nulování předchozí akce}
    R := Random(3);
    ZaDvermi[R] := 1;   {za dvermi R=0v1v2 je auto}
    C := Random(3);     {pred dvermi C=0v1v2 stojí clovek}
    repeat
      Q := Random(3)    {otevreme dvere Q ≠ C s kozou}
    until (Q <> C) and (ZaDvermi[Q] = 0);
    {C se nyní rozhodl, jestli zustane nebo dvere zmeni}
    for I := 0 to 2 do {zjisteme, co je za vsemi dvermi}
      if I <> Q then    {otevrene však vynechame}
        if I = C then  {zustal-li by clovek na puvodnim
                        miste C a je-li tam auto, zapocita
                        se mu}
          Inc(CetnostZustal, ZaDvermi[I])
        else
          {jestlize by dvere zmenil a je-li za
            nimi auto, zapocita se mu}

```

```

        Inc(CetnostZmenil, ZaDvermi[I])
until K >= N;
WriteLn('Pocet aut, kdyby u dveri zustal: ',
        CetnostZustal);
WriteLn('Pocet aut, kdyby dvere zmenil : ',
        CetnostZmenil);
ReadLn
end.

```

A teď můžete provést volbou počtu akci, třeba 3 000 000, a za malou chvíli vám počítač sdělí, že asi milionkrát by vyhrál ten, co by setrval u své původní volby dveří, ale dvamilionkrát ten, který by přešel ke zbývajícím neotevřeným dveřím, že tedy pravděpodobnost výhry auta při změně dveří je zřejmě opravdu  $2/3$ .

## L i t e r a t u r a

- [1] *Mazurek, J.*: Úloha Montyho Halla. MFI 17 (2007/08), č. 3, str. xxx.
- [2] *Trávníček, S.*: Geometrická pravděpodobnost a kalkulátor. MFI 10 (2000/01), č. 9, str. 552 – 561.

# 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 číslíce 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 MFL, 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 bu-

dou 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:

Sestrojíme 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:

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

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:

$$\begin{aligned}\langle \text{aritmetickyvyraz}+ \rangle &::= \langle \text{aritmetickyvyraz}^* \rangle \{ + \langle \text{aritmetickyvyraz}^* \rangle \} \\ \langle \text{aritmetickyvyraz}^* \rangle &::= \langle \text{operand}/\text{podvyraz} \rangle \{ * \langle \text{operand}/\text{podvyraz} \rangle \} \\ \langle \text{operand}/\text{podvyraz} \rangle &::= (\langle \text{aritmetickyvyraz}+ \rangle) | x\end{aligned}$$

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:

$$\langle \text{podminenyprikaz} \rangle ::= \text{if } \langle \text{booleovskyyvyraz} \rangle \text{ then } \langle \text{prikaz} \rangle [\text{else } \langle \text{prikaz} \rangle]$$

## 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 o 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 výhodou 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:

$$\begin{aligned}\langle V y r a z \rangle &::= \langle T e r m \rangle \{ + \langle T e r m \rangle \} \\ \langle T e r m \rangle &::= \langle F a k t o r \rangle \{ * \langle F a k t o r \rangle \} \\ \langle F a k t o r \rangle &::= (\langle V y r a z \rangle) | 0 | 1 | 2 | \dots | 9\end{aligned}$$

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 zpracováváný 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 načítání 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 nepřipustný 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:=Uppcase(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 (sčítání), 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;
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;
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 pro

vede volání počátečního neterminálu a odhalí některé chyby dodatečně po přečtení výrazu, např. že sice skončila nejvyšší procedura Vyráz, ale ještě ve vstupu jsou nějaké znaky.

```
begin
  err:=0;           {inicializace prom.}
  infixpos:=0;
  postfix:='';
  Getchar;
  Vyráz;           {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  $5 + 3 * 2$ .

| Infix       | Akt. znak | Postfix   | Akt. procedura | Návrat    |
|-------------|-----------|-----------|----------------|-----------|
| $5 + 3 * 2$ | 5         |           | Výraz          |           |
| $5 + 3 * 2$ | 5         |           | Term           |           |
| $+ 3 * 2$   | +         | 5         | Faktor         | Term      |
| $+ 3 * 2$   | +         | 5         | Term           | Výraz     |
| $3 * 2$     | 3         | 5         | Výraz (+)      |           |
| $3 * 2$     | 3         | 5         | Term           |           |
| $* 2$       | *         | 5 3       | Faktor         | Term      |
| 2           | 2         | 5 3       | Term (*)       |           |
|             |           | 5 3 2     | Faktor         | Term (*)  |
|             |           | 5 3 2 *   | Term (*)       | Výraz (+) |
|             |           | 5 3 2 * + | Vyraz (+)      |           |

(Pokračování)

## Literatura

- [1] Česka, 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. – Kmeř, T.: Vyčíslitelnost a složitost. In Matematika-fyzika-informatika. 02-03/2005-06(15).
- [8] Habiballa, H. – Pavlíšková, L. – Kmeř, 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.