Uplatnění metod na zvolený jazyk

Při výběru mezi metodami výše popsanými se řídíme především podle typu symbolů, které jazyk obsahuje.

Výhodná bývá často kombinace těchto metod – nejdřív použijeme metodu přímého stavového programování (A) a pokud je načtený symbol identifikátor, použijeme některou z metod pro konečné jazyky pro zjištění, zda se jedná o klíčové slovo. U metod ukázaných na příkladech **??** a **??** musíme ještě přidat test znaku následujícího za symbolem, který nás zavedl do některého koncového stavu.

Budeme dále pokračovat v příkladu z kapitoly **??**. Sestavíme proceduru `Lex`, jejímž úkolem bude načíst řetězec symbolu a určit jeho typ (identifikovat). V každém koncovém stavu symbolu buď přímo stanovíme hodnotu proměnné `symbol.atrib` deklarované v kapitole **??**, nebo v případě identifikátoru budeme volat proceduru `ZpracujID`, která načtený atribut dále zpracuje a určí, zda nejde o klíčové slovo. Na konci každého symbolu se procedura zastaví a pokračuje ve vyhodnocení vstupu, když je znovu volána.

```
var
  znak:   TZnak;       // znak načtený ze souboru
  symbol: TSymbol;     // zde ukládáme načtený symbol
...
procedure Lex;   // načte jeden symbol do globální proměnné symbol
begin     // Procedura Dejznak byla už volána, načtený znak je v záznamu znak
  while (znak.rad[znak.pozice] = ' ') do Dejznak(znak);
  case znak.rad[znak.pozice] of
    'A'..'Z': begin               // identifikátor nebo klíčové slovo
      symbol.atrib := znak.rad[znak.pozice];
      Dejznak(znak);
      while (znak.rad[znak.pozice] in ['A'..'Z', '0'..'9']) do begin
        symbol.atrib := symbol.atrib + znak.rad[znak.pozice];
        Dejznak(znak);
      end;
      ZpracujID(symbol.atrib);
    end;

    '0'..'9': begin               // číslo
      symbol.atrib := znak.rad[znak.pozice];
      Dejznak(znak);
      while (znak.rad[znak.pozice] in ['0'..'9']) do begin
        symbol.atrib := symbol.atrib + znak.rad[znak.pozice];
        Dejznak(znak);
      end;
      symbol.Typ := S_NUM;
    end;

    '<': begin                    // symbol '<' nebo '<=' nebo '<>'
      Dejznak(znak);
      case znak.rad[znak.pozice] of
        '>': begin
          Dejznak(znak);
          symbol.Typ := S_NEQ;    // <>
        end;
```

```
        '=': begin
          Dejznak(znak);
          symbol.Typ := S_LQ;      // <=
        end;
      else symbol.Typ := S_LESS;   // <
    end;
    ...                            // Podobně všechny ostatní symboly
    else ...                       // Ošetření chyby
  end;
end;
```

Dále musíme odlišit klíčová slova od ostatních identifikátorů a výsledky uložit do výstupního souboru. Proceduru můžeme sestavit více způsoby. První způsob je vhodný nejvýše pro velmi jednoduchý programovací jazyk s několika klíčovými slovy, my tento způsob *nebudeme používat*:

```
procedure ZpracujID(s: string);
begin
  if      s = 'BEGIN' then symbol.Typ := S_BEGIN
  else if s = 'END'   then symbol.Typ := S_END
  else if s = 'CONST' then symbol.Typ := S_CONST
  else if s = 'VAR'   then symbol.Typ := S_VAR
  ...  // atd. pro všechna klíčová slova
  else begin
    symbol.Typ := S_ID; // Není to klíčové slovo
    symbol.atrib := s;  // Tento řádek v našem případě není nutný
  end;
end;
```

Z hlediska překladu a výsledného kódu překladače (časové složitosti překladu) je optimálnější, u jazyků s rozsáhlejší „slovní zásobou“ velmi výrazně, jiná metoda. Napíšeme proceduru jako konečný automat podle druhé nebo třetí metody z kapitoly **??**.

Příklad 0.1

Sestavíme gramatiku, podle ní tabulku přechodů a program, který bude rozpoznávat tento jazyk:
$L = \{$begin, end, const, var, if, then, else, print$\}$

$S \to bA_1$ $A_1 \to eA_2$ $A_2 \to gA_3$ $A_3 \to iA_4$ $A_4 \to n$

$S \to eA_5$ $A_5 \to nA_6$ $A_6 \to d$

$S \to cA_7$ $A_7 \to oA_8$ $A_8 \to nA_9$ $A_9 \to sA_{10}$ $A_{10} \to t$

$S \to vA_{11}$ $A_{11} \to aA_{12}$ $A_{12} \to r$

$S \to iA_{13}$ $A_{13} \to f$

$S \to tA_{14}$ $A_{14} \to hA_{15}$ $A_{15} \to eA_{16}$ $A_{16} \to n$

$A_5 \to lA_{17}$ $A_{17} \to sA_{18}$ $A_{18} \to e$

$S \to pA_{19}$ $A_{19} \to rA_{20}$ $A_{20} \to iA_{21}$ $A_{21} \to nA_{22}$ $A_{22} \to t$

Automat bude mít stavy 0 . . . 22 přejaté z gramatiky, dále přidáme tyto stavy:

```
const
  k_chyba = 23;    k_const = 26;    k_then  = 29;
  k_begin = 24;    k_var   = 27;    k_else  = 30;
  k_end   = 25;    k_if    = 28;    k_print = 31;
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	B	E	G	I	N	D	C	O	S	T	V	A	R	F	H	L	P
⇝ 0	1	5		13			7			14	11						19
1		2															
2			3														
3				4													
4					24												
5					6											17	
6						25											
7								8									
8					9												
9									10								
10										26							
11												12					
12													27				
13														28			
14															15		
15		16															
16					29												
17									18								
18		30															
19													20				
20				21													
21					22												
22										31							

Tabulka 1: Tabulka přechodů pro klíčová slova zvoleného jazyka

Navrhneme deterministickou tabulku přechodů (je v tabulce 1, bez řádků pro chybový a koncové stavy) a přepíšeme do datové struktury. Pokračujeme:

```
const
  PocetZnaku = 17; // Počet znaků, ze kterých se skládají klíčová slova
var
  tab: array [0..22, 1..PocetZnaku] of byte;

procedure NactiTabulku;
var
  i, j: byte;
begin
  for i := 0 to 22 do
    for j := 1 to PocetZnaku do tab[i,j] := k_chybovy;
  tab[ 0, 1] :=  1;   tab[ 0, 2] :=  5;   tab[ 0, 4] := 13;
```

```
  tab[ 0, 7] :=  7;   tab[ 0,10] := 14;   tab[ 0,11] := 11;
  tab[ 0,17] := 19;   tab[ 1, 2] :=  2;   tab[ 2, 3] :=  3;
  tab[ 3, 4] :=  4;   tab[ 4, 5] := 24;   tab[ 5, 5] :=  6;
  tab[ 5,16] := 17;   tab[ 6, 6] := 25;   tab[ 7, 8] :=  8;
  tab[ 8, 5] :=  9;   tab[ 9, 9] := 10;   tab[10,10] := 26;
  tab[11,12] := 12;   tab[12,13] := 27;   tab[13,14] := 28;
  tab[14,15] := 15;   tab[15, 2] := 16;   tab[16, 5] := 29;
  tab[17, 9] := 18;   tab[18, 2] := 30;   tab[19,13] := 20;
  tab[20, 4] := 21;   tab[21, 5] := 22;   tab[22,10] := 31;
end;

function DejCisloZnaku(zn: char): byte;
// Pokud zn nepatří do abecedy, nad kterou jsou vytvořena klíčová slova,
// funkce vrátí hodnotu 0. Jinak vrací index znaku.
const
  Index: string [PocetZnaku] = 'BEGINDCOSTVARFHLP';
var
  i, v: byte;
begin
  v := 0;             // číslo 0 náleží nedefinovanému znaku
  i := 1;
  while (i <= PocetZnaku) do begin
    if (zn = Index [i]) then begin
      v := i;         // nalezen index (číslo) znaku v seznamu
      break;
    end;
    inc(i);
  end;
  DejCisloZnaku := v;
end;

procedure ZpracujID(s: string);
var
  stav:   byte;         // aktuální stav automatu
  pozice: byte;         // pozice v testovaném řetězci s
  delka:  byte;         // délka řetězce s
  znak:   byte;         // číslo znaku podle seznamu znaků klíčových slov
begin
  stav   := 0;
  pozice := 1;
  delka  := length(s);
  while (pozice <= delka) and (stav < k_chyba) do begin
    znak := DejCisloZnaku(s[i]);
    if (znak = 0) then stav := chybovy
                  else stav := tab[stav,znak];
  end;
  case stav of
    k_begin: symbol.Typ := S_BEGIN;
    k_end:   symbol.Typ := S_END;
    k_const: symbol.Typ := S_CONST;
    k_var:   symbol.Typ := S_VAR;
    k_if:    symbol.Typ := S_IF;
    k_then:  symbol.Typ := S_THEN;
    k_else:  symbol.Typ := S_ELSE;
    k_print: symbol.Typ := S_PRINT;
```

```
    else begin
      symbol.Typ := S_ID;
      symbol.atrib := s;
    end;
  end;
end;

procedure InitLex;
// Tato procedura je volána pouze jednou za celý překlad
begin
  ...             // Otevření vstupního souboru pro čtení, zpřístupnění
                  // přes proměnnou zdroj (textový soubor).
  NactiTabulku;   // Načteme tabulku přechodů do proměnné tab.
  DejZnak;        // Načteme do proměnné znak první znak souboru,
                  // v proceduře NactiSymbol se s tím počítá
end;
```

První způsob implementace používající prosté porovnávání řetězců je určitě velmi jednoduchý, rychlý a intuitivní. Časová složitost výpočtu[1] je však (zejména pro jazyky s větším množstvím klíčových slov) podstatně vyšší, než je únosné. Důvodem je vícenásobné procházení testovaného řetězce – v nejhorším případě, tedy když nejde o klíčové slovo, je alespoň začátek řetězce procházen při každém uvedeném porovnávání. Proto má smysl takto postupovat pouze u jazyků, které mají jen velmi málo klíčových slov a jsou postaveny především na jiných typech symbolů.

U druhého způsobu je časová složitost obecně mnohem nižší (každý znak slova je zpracováván nejvýše jednou), narůstá však prostorová složitost[2], protože v paměti je uložena celá tabulka přechodů automatu. V dnešní době však vyšší prostorová složitost již tolik nevadí, a i kdyby, dá se řešit například použitím technik pro zachycení řídké matice (většina prvků tabulky má tutéž hodnotu, chybový stav). Můžeme samozřejmě postupovat také metodou pro konečné jazyky s nižší prostorovou složitostí, která je ukázaná na příkladu **??** na straně **??**.

Datové typy konstantních hodnot

Do této chvíle jsme pracovali pouze s programovacími jazyky, které měly jediný datový typ – celé nezáporné číslo. V praxi se však používají jazyky přijímající obvykle celá čísla (bez znaménka nebo se znaménkem), reálná čísla, znaky, řetězce, pole, záznamy, pointery, výčtové typy atd. Lexikální analyzátor se obvykle takovými rozlišeními nemusí zabývat, pokud ovšem nejde o konstanty.

Čísla můžeme nechat v znakové podobě tak, jak byla ve zdrojovém textu, nebo je předat dál v binárním tvaru ve vhodné reprezentaci (pak pro atribut nepoužijeme řetězec, ale variantní záznam, příp. v C union, kde jednotlivé možnosti budou odpovídat zvolenému datovému typu konstanty). Tuto reprezentaci volíme podle toho, co nám nabízí programovací jazyk, ve kterém překladač píšeme, obvykle například u celých čísel máme na výběr mezi těmito možnostmi:

[1] *Časová složitost* znamená náročnost výpočtu algoritmu z hlediska doby jeho trvání v závislosti na délce vstupu. Vyšší časovou složitost má ten algoritmus, jehož provedení v běžném případě trvá déle.

[2] Jestliže máme dva algoritmy A_1 a A_2 a řekneme, že A_1 má vyšší *prostorovou složitost*, znamená to, že při výpočtu algoritmu A_1 je pro běžné vstupy použito více paměťového prostoru než při výpočtu algoritmu A_2.

- celé číslo se znaménkem na 2 B (integer)[3], rozmezí $-32\,768 \ldots 32\,767$,
- celé číslo bez znaménka na 2 B (word), rozmezí $0 \ldots 65\,535$,
- celé číslo se znaménkem na 1 B (short), rozmezí $-128 \ldots 127$,
- celé číslo bez znaménka na 1 B (byte, char), rozmezí $0 \ldots 255$.

Vzhledem k tomu, že znaménko „−“ můžeme chápat jako zvláštní symbol, volíme spíše datové typy, které znaménko nepoužívají, ale díky tomu na stejně velkém paměťovém místě nabízejí větší rozsah pro kladné číslo. Pro racionální čísla s plovoucí desetinnou čárkou můžeme volit vždy tentýž datový typ nebo rozhodovat obdobně jako u celých čísel.

V takovém jazyce byl napsán úsek programu:

```
CONST
  a = 224;
  b = - 224;
  c = - 5;
  d = 10000;
...
  prom := 25 * b + 8224;
```

Vyskytuje se zde celkem šest celočíselných konstant, u kterých je nutné určit datový typ. Toto rozlišení může provádět sémantický analyzátor nebo je lze přenechat lexikálnímu. V lexikálním analyzátoru postupujeme takto:

1. načteme řetězec s číslicemi (nebo průběžně načítáme),
2. převedeme řetězec na číslo (vytvoříme „meziprodukt“ představující nejuniverzálnější reprezentaci – použijeme datový typ zabírající nejvíce místa v paměti),
3. porovnáváme načtené číslo s mezními hodnotami a podle toho určujeme přesný datový typ,
4. pokud má lexikální analyzátor přístup k informaci, zda jde o kladné nebo záporné číslo, můžeme tento fakt zohlednit při výběru datového typu, ovšem to se týká spíše definice pojmenované konstanty než výskytu konstanty ve výrazu.

V našem případě tedy bude výsledek takový (jsou uvedeny pouze číselné symboly, nikoliv ostatní včetně symbolu pro znaménko „−“):

```
S_NUM_BYTE   224
S_NUM_BYTE   224
S_NUM_BYTE   5
S_NUM_WORD   10000
S_NUM_BYTE   25
S_NUM_WORD   8224
```

[3] Skutečné množství paměti pro integer závisí na operačním systému – 2 B platí pro 16-bitový OS, 32-bitové operační systémy (momentálně nejpoužívanější) používají 4 B, v 64-bitových systémech zabírá integer 8 B, a od toho se odvíjí také rozmezí hodnot.

Reprezentace konstantního *řetězce* není problém, pouze vzhledem k optimalizaci prostorové složitosti volíme vhodnou délku řetězce. Řetězec je obvykle ohraničen speciálními znaky (jednoduché nebo dvojité uvozovky), takže lexikální analyzátor po nalezení prvního takového znaku pokračuje v načítání, dokud nenajde druhý, uzavírací znak řetězce. Uvozovací znaky nejsou symboly, z hlediska překladače jde pouze o pomocné znaky, které mu říkají, kde řetězec začíná a kde končí.

Dále můžeme ošetřit případ, kdy uživatel velmi dlouhý řetězec rozdělí na více menších řetězců a každý umístí na nový řádek (to umožňuje například programovací jazyk C). Pokud konstantní řetězce ukládáme do dostatečně rozsáhlých hodnot symbolů (jestliže je výstupem dynamická struktura nebo soubor, lze délku hodnot typu řetězec určovat též dynamicky), můžeme všechny tyto konstantní řetězce spojit do jediného. Pokud programovací jazyk umožňuje sčítání řetězců, lze jednotlivé řetězce načíst zvlášť a spojit je explicitně operátorem sčítání (není to obvyklý postup) nebo se lexikální analyzátor nemusí vůbec namáhat řešením těchto situací a výsledkem je prostě posloupnost řetězců, kterou zpracuje syntaktický analyzátor.

U konstantních *polí* a *záznamů* záleží na zvolené vnitřní reprezentaci jazyka a předepsaném tvaru definice těchto konstant. Obvyklé je zadávat pole jako výčet prvků oddělených čárkou a záznam jako posloupnost vnitřních proměnných a jejich hodnot, takže lexikální analyzátor tyto konstanty jako celek nemusí zpracovávat a předává je dál v rozloženém tvaru.

Úkoly ke kapitole 2

1. Vytvořte regulární gramatiku jazyka celých nezáporných čísel.
2. Podle gramatiky, kterou jste sestrojili v úkolu 1, vytvořte diagram deterministického konečného automatu.
3. Vytvořte regulární gramatiku jazyka reálných nezáporných čísel, celá a reálná část čísla jsou odděleny desetinnou tečkou, která je nepovinná (pak jde o celé číslo), před tečkou nemusí být žádná číslice, za tečkou musí být alespoň jedna číslice. Podle této regulární gramatiky vytvořte diagram deterministického konečného automatu.
4. Sestrojte regulární gramatiku a podle ní *deterministický* konečný automat reprezentovaný tabulkou přechodů pro jazyk $L_1 = \{\text{is, then, this}\}$.

   Automat má rozpoznávat jednotlivá slova jazyka, bude mít pro každé slovo jiný koncový stav. Gramatiku vytvořte tak, aby bylo možné konstruovat automat přímo jako deterministický, bez nutnosti další transformace.
5. Naprogramujte konečný automat z úkolu 4 některou z metod z této kapitoly nebo jejich kombinací (metody jsou popsány v podkapitole **??** od strany **??**, možnost kombinace metod v podkapitole od strany 1).
6. Sestrojte regulární gramatiku a podle ní deterministický konečný automat pro tyto jazyky:
   - $L_2 = \{\text{if, then, else, elif, end}\}$ (automat reprezentovaný tabulkou symbolů)

- $L_3 = \{\text{jdi, stop, doprava, doleva}\}$ (automat reprezentovaný tabulkou symbolů)
- $L_4 = \{\text{read, write, var}\} \cup \{a, \ldots, z\}^+$ (tři klíčová slova a názvy proměnných obsahující pouze malá písmena, alespoň jedno)
- $L_5 = \{\text{if, write, } <, >, <=, >=, <>\} \cup \{0, \ldots, 9\}^+$ (dvě klíčová slova, relační operátory, celá čísla)
- $L_6 = \{\text{line, oval, rect, ,, [,]}\} \cup \{0, \ldots, 9\}^+$ (tři klíčová slova, čárka, hranaté závorky, celá čísla)
- $L_7 = \{+, -, *, /, :=, (,)\} \cup \{0, \ldots, 9\}^+ \cup (\{a, \ldots, z\} \cdot \{a, \ldots, z, 0, \ldots, 9\}^*)$ (matematické výrazy s běžnými aritmetickými operátory a operátorem přiřazení, závorkami, celými čísly a proměnnými – název proměnné začíná písmenem, pak mohou následovat písmena nebo číslice)
- $L_8 = L_6 \cup L_7$ (v parametrech příkazů z jazyka L_6 mohou být běžné matematické výrazy včetně použití proměnných, hodnotu proměnných lze určit přiřazovacím příkazem)

7. Vyberte si kterýkoliv z jazyků L_2–L_7 z předchozího úkolu a naprogramujte jeho lexikální analýzu některou z metod uvedených v této kapitole (nebo jejich kombinací).
8. Naprogramujte lexikální analýzu jazyka L_8 z předchozího úkolu kombinací metod podle podkapitoly (strana 1).