Jan K O Z Á K

Výzkumný ústav ekonomiky paliv a energie Praha

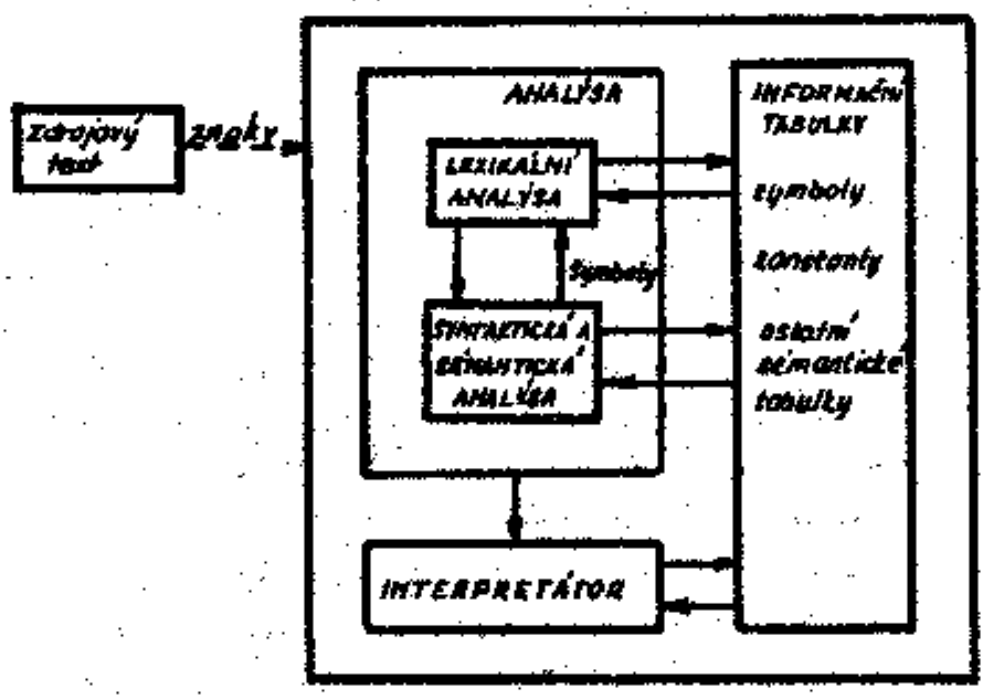
Některé aspekty řešení interaktivních systémů

Programátorovou snahou je, aby parametry nebo data byly pokud možno pevného formátu. Výhody tohoto pojetí jsou zřejmé. Programový modul pro čtení dat je jednoduchý a je také zjednodušena vstupní kontrola. V aplikacích jako je interaktivní systém člověk-počítač nebo i v dávkových obecně pojatých programech je tento přístup těžko přijatelný. Je sice možné navrhnout tyto systémy pracující s kódovanými informacemi, ovšem jejich profesionální úroveň v podmínkách, které by se měly blížit lidské komunikaci, je velmi nízká. Příčinou, proč obvykle se opouští od myšlenky implementace interaktivního jazyka je jeho náročnost, nedostatek znalostí a zkušeností z této oblasti a také zde postrádáme vhodnou domácí literaturu. Implementace komunikačního jazyka je obdobná implementaci kompilátoru pracujícího interpretačním způsobem a vede zpravidla k vyvolání množiny aplikačních routin. Příspěvek se zabývá programovým řešením kompilátoru interaktivních jazyků pro aplikace jako je vyhledávání informací, grafické systémy, statistické systémy, generátory výstupních sestav apod. V této oblasti byla vyvinuta řada metod a prací zabývajících se i optimalizací jak procesu vlastního překladu, tak i optimalizací výsledku překladu /generace optimálního kódu, optimalizace vyvolání databázových routin z hlediska četnosti výskytu informací apod/. Zde se nebudeme zabývat aplikačně závislými aspekty překladu a tím související optimalizací, ale soustředíme se především na ty metody a prostředky, které jsou poměrně jednoduché a nenáročné.

1. Struktura programového zabezpečení

Kompilátor musí provést nejdříve analýzu zdrojového textu a

potom syntésu, jejíž výsledkem je v našich uvažovaných aplikacích vhodně uspořádaná soustava tabulek využívaná dále interpretátorem.
Struktura kompilátoru je zobrazena na obr. 1.



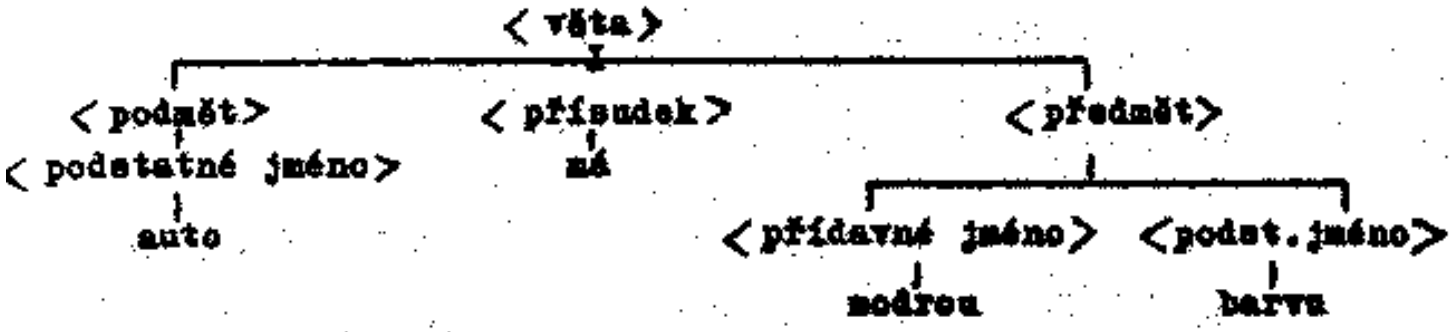
obr. 1 struktura kompilátoru

Překlad sestává ze tří částí: lexikální, syntaktické a sémantické analýsy.
Lexikální analysátor /scanner/ zjednodušuje konstrukci dalších fází překladu tím, že pracují se symboly pevné délky místo řetězců znaků proměnné délky. Lexikální analysátor vytváří symboly ze zdrojového textu jako jsou numerická čísla, identifikátory, řetězcové konstanty, rezervovaná slova apod. Může provádět další činnosti např. otisk zdrojového textu popř. ukládat symboly do tabulek. Z funkčního hlediska může být činnost lexikálního analysátoru pevně determinována, umožňuje-li to syntax, nebo je činnost lexikálního analysátoru řízena syntaktickým analysátorem.
Někdy kompilátor obsahuje i více lexikálních analysátorů např. pro různé typy vstupních médií, pro různé gramatiky apod.
Syntaktický analysátor /rozkladač/ kontroluje syntax zdrojového textu. Když syntaktický analysátor rozpozná konstrukci zdrojového jazyka, vyvolá sémantickou proceduru, která provádí

kontrolu po semantické stránce a vkládá informace do tabulek. Účelem semantické kontroly je zajistit správnost zdrojového textu, který je syntaktícky /strukturně/ správný, ale odporuje významu dané aplikace.

2. Teorie formální gramatiky

Pro konstrukci analysátorů jsou nutné aspoň minimální znalosti formální gramatiky.
Uvádíme zde stručně řadu definic nejzákladnějších pojmů a výrazů z této oblasti.
Gramatika je množina pravidel, které popisují jaké znakové posloupnosti jsou větami v jazyku.
Jazyk je množina vět.
Věta je vhodná posloupnost znaků složená z atomů /konečných slov/.
Konečné slovo je nedělitelnou atomární jednotkou.
Příkladem věty je "auto má modrou barvu". Zde jsou atomárními slovy "auto", "má", "modrou", "barvu".
Při mechanickém rozkladu abychom mohli dedukovat které řetězce jsou větami, potřebujeme nějaké formální prostředky. Na obr. 2 diagram zobrazuje strukturu neboli syntax věty a je nazvaný syntaktický strom.



obr. 2 syntaktický strom

Syntaktický strom poskytuje obraz struktury věty tím, že popisuje všechna gramatická pravidla použitá při tvorbě věty. Kromě atomárních slov jako jsou "auto" nebo "má" obsahuje syntaktický strom i speciální symboly podmět , přísudek aj,

které nazýváme specifikačními typy jazyka. Specifikační typy jazyka spolu s atomárními slovy tvoří metajazyk, což je jazyk, kterým popisujeme jiný jazyk. Nejdůkladnější formou metajazyka je dnes Backus-Naurův tvar /BNF/, který byl poprvé použit k popisu programovacího jazyka ALGOL 60 /Naur/. Syntaktický strom zobrazený na obr. 2 vypadá přepsán do BNF takto:

<věta>=<podmět><přísudek><předmět>
<podmět>=<podstatné jméno>
<podstatné jméno>="auto"
<přísudek>="má"
<předmět>=<přídavné jméno><podstatné jméno>
<přídavné jméno>="modrou"
<podstatné jméno>="barvu"

Z uvedené množiny pravidel můžeme generovat věty, které jsou strukturně /syntakticky/ správné, ale nemají význam např. "barvu má modrou barvu". Proto je nutná sémantická kontrola. Definice specifikačního typu jazyka může být zapsána pomocí výrazu

$$\langle a \rangle = x$$

kde levá část definice $\langle a \rangle$ je specifikační typ jazyka a pravá část je zřetězení specifikačních typů jazyka označených malými písmeny a atomárních slov.
Jestliže x a y jsou zřetězení specifikačních typů jazyka a atomárních slov, potom x přímo odvozuje y

$$x \Rightarrow y$$

v případě, že x je zřetězení $v\langle a \rangle w$
y je zřetězení $v\ q\ w$
a $\langle a \rangle = q$ je definice specifikačního jazyka v gramatice

Specifikační typ tvaru $\langle a \rangle = uvw$, kde u a w jsou prázdné /null/ vytváří derivaci tvaru $\langle a \rangle = v$. Podobně x odvozuje y

$$x \Rightarrow^{+} y$$

jestliže existuje množina derivací

$$x \Rightarrow w_1 \Rightarrow w_2 \Rightarrow w_3 \Rightarrow \ldots\ldots \Rightarrow w_n = y$$

kde $n > 0$.
Gramatika $G(\langle s \rangle)$ je definována jako konečná množina definic

specifikačních typů jazyka. ⟨s⟩ je specifikační typ jazyka, který je nazýván počátečním typem.
Výše uvedená gramatika G(⟨věta⟩) vytváří derivaci:

⟨věta⟩⇒ + auto má modrou barvu

O gramatice vytvářející derivaci x⇒ +y x w říkáme, že je samoobsažná v x.
Jestliže gramatika obsahuje specifikační typ jazyka tvaru ⟨a⟩= x⟨a⟩y, kde x je prázdný, potom o definici říkáme, že je přímo vlevo rekursivní.
Jestliže x není prázdný a y je nulový, pak definice je přímo vpravo rekursivní.
U gramatiky G(⟨s⟩) říkáme o y, že je větným tvarem G, jestliže y může být přímo odvozen z počátečního typu ⟨s⟩. Tak větný tvar je derivací obsahující atomární slova nebo specifikační typy jazyka. Věta je větný tvar obsahující pouze členy z nějaké množiny atomárních slov.
Jazyk L[G(⟨s⟩)] je množina vět, které mohou být odvozeny z počátečního typu s.
Při větném rozkladu postupně na větný tvar vytváříme derivace z počátečního typu.
Při rozkladu posloupnosti znaků y musíme pro gramatiku s počátečním typem ⟨x⟩ nalézt derivaci

⟨x⟩⇒+y

Např. při rozkladu věty auto má modrou barvu můžeme vytvářet posloupnost derivací

⟨věta⟩⇒⟨podmět⟩⟨přísudek⟩⟨předmět⟩⇒⟨podmět⟩⟨přísudek⟩
⟨přídavné jméno⟩⟨podst. jméno⟩⇒⟨podst. jméno⟩
⟨přísudek⟩⟨přídavné jméno⟩⟨podst. jméno⟩⇒auto
⟨přísudek⟩⟨přídavné jméno⟩⟨podst. jméno⟩⇒auto
⟨přísudek⟩modrou⟨podst. jméno⟩⇒auto⟨přísudek⟩
modrou barvu⇒auto má modrou barvu

Tento rozklad není ve všech případech úplně definovaný. Např. u věty auto má modré auto by nebylo jasné, jak by se auto mělo redukovat při dvou specifikačních typech ⟨podst. jméno⟩. Z toho důvodu se definuje kanonický rozklad, při kterém se vždy nahrazuje specifikační typ jazyka, který je ve větném

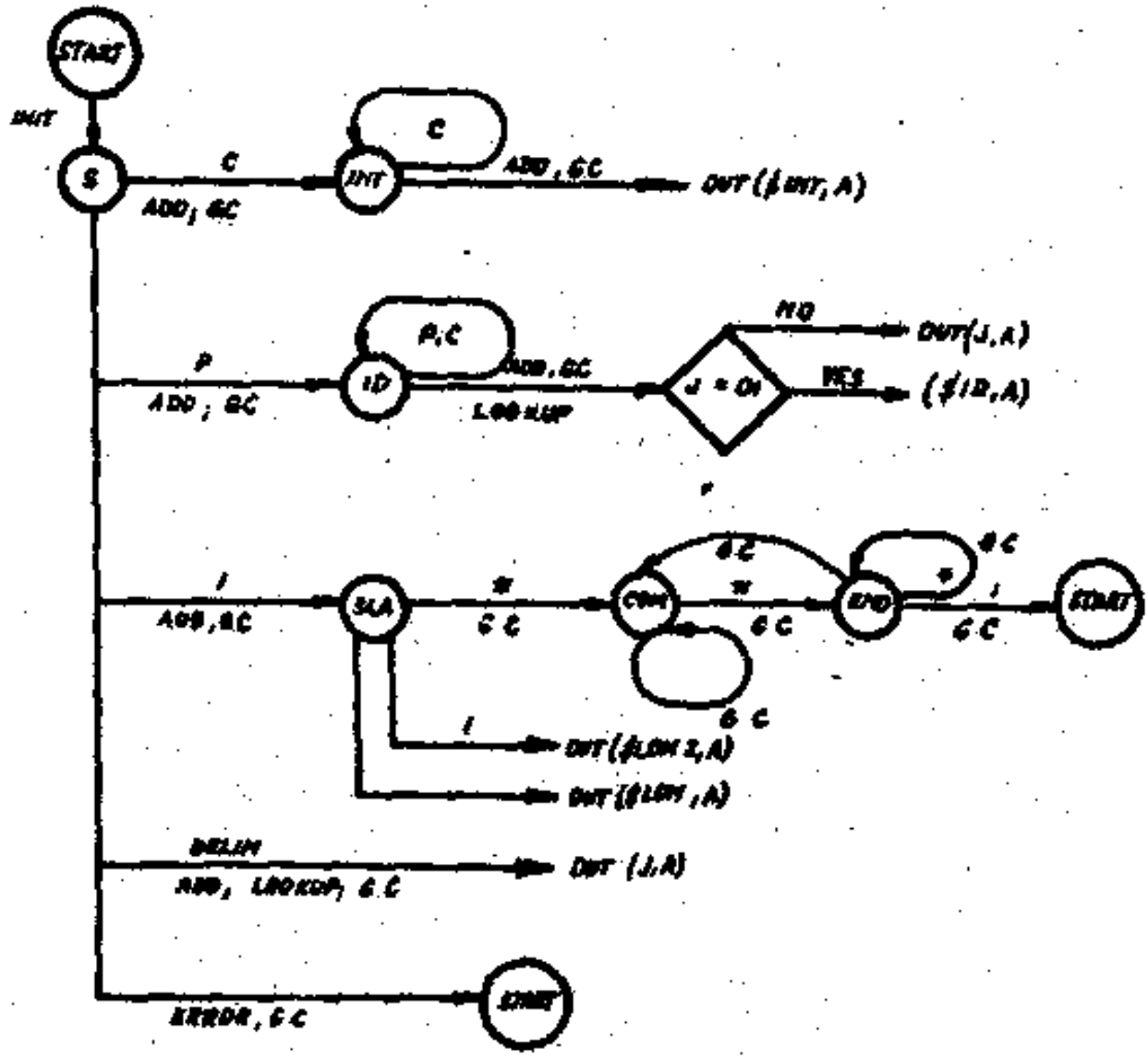
tvaru umístěn nejvíce vlevo. Tak pro větný tvar u⟨a⟩w, kde u obsahuje pouze atomární slova, se nahrazuje specifikační typ jazyka ⟨a⟩ pravou částí jeho definice, tj. pro ⟨a⟩= z odvozuje se další větný tvar usw.
O jazyku říkáme, že je jednoznačný, jestliže každá jeho věta má pouze jeden kanonický rozklad.

3. Konstrukce lexikálního analysátoru

Konstrukce lexikálního analysátoru vychází ze stroje konečných stavů. Při řešení nejdříve musíme definovat symboly a jejich vnitřní ekvivalenty.
Předpokládejme, že operátory v jazyku jsou /,+,-,*,(a //, rezervovaná slova jsou BEGIN, ABS, END a dalšími symboly jsou identifikátory a celá čísla. Je požadavek, aby v textu mohly být zapsány poznámky v PL/1 tvaru, tj. /* ... */.
Vyvolaný lexikální analysátor vrací rodičovskému segmentu dvě hodnoty:

vnitřní zobrazení symbolu /označeno dále SYM/
hodnotu symbolu /označeno dále SEM/

Po definici tabulky vnitřního zobrazení se doporučuje nakreslit stavový diagram zobrazující způsob, jakým jsou symboly rozkládány. Stavový diagram pro výše uvedený příklad je na obr. 3.
Nejdříve je provedena inicialisace spočívající v přiřazení nulové hodnoty symbolu A a ve čtení nenulového znaku.
Ze stavu S dostáváme se do dalších stavů podle hodnoty znaku / C = číslice, P = písmena, DELIM = delimitační znaky ,+-*(/.
Příkaz AND provádí zřetězení načteného znaku k symbolu a procedura GC provádí čtení dalšího znaku. Při výstupu z lexikálního analysátoru obsahuje vyrovnávací paměť GC vždy další znak.
Procedure LOOKUP používá některou techniku k hledání v tabulce rezervovaných slov, popř. v tabulce delimitačních znaků.
Při výstupu obsahují proměnné ŽINT, ŽID, ŽLOM, ŽLOM2 vnitřní zobrazení symbolů integer, identifikátor, lomítko, resp. dvojnásobné lomítko.



obr. 5 stavový diagram lexikálního analysátoru

Přepsání do programovacího jazyka, např. PL/1 je už jednoduché:

```
SCANNER:PROC(SYM,SEM);
DCL SYM BIN FIXED(15),
    (SEM,A) CHAR(30) VARYING,
    LAB(5) LABEL INIT(#INT,#ID,#SLA,#DELIM,#OTHERS);
START:
 CALL GCHBL;          /* získání neprázdného znaku */
 A="";                /* nulování symbolu */
 GO TO LAB(CLASS); /* CLASS je globální proměnná s procedur */
                      /* GCHBL resp. GO */
```

```
¢INT:
 DO WHILE(CLASS=1);     /* CLASS=1 je integer */
  A=A||¢CHAR;           /* CHAR je čtený znak z GCNBL, GC */
  CALL GC;
 END;
 SYM=¢INT;              /* ¢INT je vnitřní zobrazení integer */
 GO TO ¢END;
¢ID:
 DO WHILE(CLASS¬=2);    /* písmena nebo číslice */
  A=A||¢CHAR;
  CALL GC;
 END;
 SYM=¢ID;               /* ¢ID je vnitřní zobrazení identif */
 CALL LOOKUP(A,J);      /* hledání v tabulce */
 IF J>0 THEN SYM=J;
 GO TO ¢END;
¢SLA:                   /* CLASS=3 je lomítko */
 A=¢CHAR;
 CALL GC;
 IF ¢CHAR='*'           /* je indikovaná poznámka */
    THEN DO;
         L1:CALL GC;
         L2:IF ¢CHAR¬='*' THEN GO TO L1; /* čti do dalšího zn. */
            CALL GC;
            IF ¢CHAR = '/' THEN GO TO L2; /* musí následovat / */
            CALL GC;
            GO TO START;   /* ignorace poznámky */
          END;
 IF ¢CHAR='/'
    THEN DO;
            A=A||¢CHAR;     /*  // */
            SYM=¢LOM2;
            CALL GC;
          END;
    ELSE SYM=¢LOM;          /* / */
```

```
 GO TO ¬END;
¬DELIM:
 A=¬CHAR;
 CALL LOOKUP(A,J);  /* hledání delim. znaku v tabulce */
 SYN=J;
 GO TO ¬END;
¬OTHERS:                /* nedovolený znak */
 CALL GO;               /* ignorace */
 CALL ERROR;            /* chybová routina */
 GO TO START;
¬END:
 SEM=A;
END SCANNER;
```

Výše uvedený lexikální analysátor je deterministický. Je ovšem možno navrhnout nedeterministický analysátor /používá se v generátorech kompilátorů/ nebo jeho činnost doplnit zahrnutím maker apod. V každém případě by měl být lexikální analysátor jednoduchý a dostatečně efektivní.

4. Syntaktická analýza

V zásadě existují dvě metody syntaktické analysy: shora-dolu /top-down/ nebo zdola-nahoru /bottom-up/.
Metody jsou nazvány podle způsobu jimiž pracují a vytvářejí syntaxový strom.
Nebudeme se zde zabývat přehledem ani vhodností jednotlivých metod a jejich variant - reference např. /Gries, Hopgood/.
Zaměříme se na techniku shora-dolu nazvanou rekursivní sestup, která je dostatečně jednoduchá pro přehledný zápis algoritmu syntaxu.
Syntaktický analysátor má jednu proceduru pro každý specifikační typ jazyka. Jestliže definice specifikačního typu je rekursivní, musíme navrhnout rekursivní proceduru, kterou lze dobře implementovat pomocí progr. jazyků umožňujících jejich vytváření /např. PL/1, ALGOL 60, ALGOL 68/. Tato technika neumožňuje přímo implementaci těch definic, které

jsou přímo vlevo rekurzivní, protože by došlo k zacyklování algoritmu. Proto k obejití problému levé rekurze /resp. samovolatelnosti/ musíme upravit gramatiku pomocí faktoru opakování { }.

Pro ilustraci aplikujeme tuto techniku pro gramatiku přiřazovacího příkazu umožňující podmínku, která je popsána tímto BNF zápisem:

⟨příkaz⟩=⟨proměnná⟩=⟨výraz⟩
| IF ⟨výraz⟩ THEN ⟨příkaz⟩
| IF ⟨výraz⟩ THEN ⟨příkaz⟩ ELSE ⟨příkaz⟩
⟨proměnná⟩= identifikátor | identifikátor(výraz)
⟨výraz⟩= ⟨termín⟩ |⟨výraz⟩+⟨termín⟩
⟨termín⟩=⟨faktor⟩ |⟨termín⟩*⟨faktor⟩
⟨faktor⟩=⟨proměnná⟩ |(⟨výraz⟩)

Pro odstranění zpětného dotazu je nutno určit vždy jeden cíl a zároveň je nutné odstranit levou rekurzi. Proto přepíšeme výše uvedený BNF zápis do tvaru:

⟨příkaz⟩=⟨proměnná⟩=⟨výraz⟩
| IF ⟨výraz⟩ THEN ⟨příkaz⟩ [ELSE ⟨příkaz⟩] {0 1}
⟨proměnná⟩= identifikátor [(⟨příkaz⟩)] { 0 1 }
⟨výraz⟩=⟨termín⟩ [+⟨termín⟩] { 0 100 }
⟨termín⟩=⟨faktor⟩ [*⟨faktor⟩] { 0 100 }
⟨faktor⟩=⟨proměnná⟩|(⟨výraz⟩)

Opakující se skupina je uzavřena do hranatých závorek a za ní je uveden faktor opakování s dolní a horní mezí. V našem případě dovolujeme možnost výskytu 100 operací součtu resp. násobení v příkaze.

Zápis přepsaný do PL/1 vypadá takto:

```
PRIKAZ:PROC RECURSIVE;
 IF SEM='IF'
    THEN DO; CALL SCANNER;
             CALL VYRAZ;
             IF SEM='THEN'
                THEN CALL ERROR;
                ELSE DO; CALL SCANNER;
                         CALL PRIKAZ;
```

```
                        IF SEM='ELSE'
                           THEN DO; CALL SCANNER;
                                    CALL PRIKAZ;
                                END;
                     END;
          END;
     ELSE DO; CALL PROMENNA;
              IF SEM<>':='
                 THEN CALL ERROR;
                 ELSE DO; CALL SCANNER;
                          CALL VYRAZ;
                      END;
          END;
END PRIKAZ;
PROMENNA:PROC RECURSIVE;
 IF SYM<>IDENT
    THEN CALL ERROR;
    ELSE DO; CALL SCANNER;
             IF SEM='('
                THEN DO; CALL SCANNER;
                         CALL VYRAZ;
                         IF SEM<>')' THEN CALL ERROR;
                                     ELSE CALL SCANNER;
                      END;
          END;
END PROMENNA;
VYRAZ:PROC RECURSIVE;
 CALL TERMIN;
 DO WHILE (SEM='+');
  CALL SCANNER;
  CALL TERMIN;
 END;
END VYRAZ;
TERMIN:PROC RECURSIVE;
 CALL FAKTOR;
 DO WHILE (SEM='*');
  CALL SCANNER;
```

```
  CALL FAKTOR;
 END;
END TERMIN;
FAKTOR:PROC RECURSIVE;
 IF SYM= '('
    THEN DO; CALL SCANNER;
             CALL VYRAZ;
             IF SYM ¬= ')' THEN CALL ERROR; ELSE CALL SCANNER;
         END;
    ELSE CALL PROMENNA;
END FAKTOR;
```

Před prvním vyvoláním procedury PRIKAZ je nutno vyvolat subroutinu SCANNER, která vkládá do SYM první symbol zdrojového textu, jenž bude zpracován.

5. Semantická analýza

Semantické routiny navazují na jednotlivá pravidla gramatiky. Jejich úkolem může být např. ukládání symbolů do tabulek, diagnosa nevhodných kombinací identifikátorů, provádění přímo interpretačních kroků apod.
Tyto činnosti zahrnují široký rozsah programátorských technik, výborná reference např. /Knuth/.
Gramatika aplikačních jazyků často dovoluje použití aritmetických výrazů, které jsou vysoce rekursivní a vyžadují použití pracovních proměnných v průběhu výpočtu. Zde budeme ilustrovat návrh semantických routin na konversi aritmetických výrazů do vnitřního tvaru.
Vhodným tvarem pro jednoduché binární operace jsou čtveřice

operátor, operand-1, operand-2, výsledek

Tak výraz A*B+C*D je zobrazen množinou čtveřic:

```
*, A, B, T1
*, C, D, T2
+,T1,T2, T3
```

Čtveřice jsou zapsány ve stejném pořadí, v jakém budou interpretovány. Unární operace mají druhý operand prázdný. Čtveřice mohou obsahovat i jiné operátory než jsou aritmetické,

např. podmíněné nebo nepodmíněné skoky, vyvolání externích procedur, konverse čísel apod.
Čtveřice nejsou také jediným vnitřním tvarem do kterého jsou konvertovány algoritmicky orientované jazyky /dalšími jsou např. polská notace, trojice, pětice, stromy atd/.
Nyní si ukážeme, jakým způsobem jsou sémantické routiny začleněny do syntaktického analysátoru.
Předpokládejme, že máme implementovat interpretátor pro výrazy, jejichž gramatika je popsána následujícně:

⟨výsledek⟩=⟨výraz⟩
⟨výraz⟩=[-]{0 1}⟨termín⟩[(+|-)⟨termín⟩]{0 100}
⟨termín⟩=⟨faktor⟩[(*|/)⟨faktor⟩]{0 100}
⟨faktor⟩=identifikátor|(⟨výraz⟩)

Tuto gramatiku je možno pomocí techniky rekursivního sestupu rozepsat do těchto procedur / pro úsporu jsou uvedeny pouze dvě/:

```
VYSLEDEK:PROC;
 CALL VYRAZ;
END VYSLEDEK;
TERMIN:PROC RECURSIVE;
 CALL FAKTOR;
 DO WHILE(SEM='*' |SEM='/');
  CALL SCANNER;
  CALL FAKTOR;
 END;
END TERMIN;
```

Nyní do těchto procedur zabudujeme sémantické routiny pro tvorbu čtveřic. Čtveřice budou generovány routinou ENTER W,X,Y,Z , kde jednotlivé parametry představují pole čtveřice. Abychom produkovali čtveřice v pořadí jejich interpretace, správně bychom museli vybudovat syntaktický strom nebo si pomoci sémantickým zásobníkem. Avšak rekursivní procedury vždy používají zásobník, i když ho explicitně nedefinujeme. Proto použijeme v procedurách lokální proměnné /zásobníky/ a sémantické informace budeme předávat pomocí formálních parametrů. Zde sémantickou informací je jméno proměnné nebo pracovní pro-

měnné. Jestliže je rozkládán specifikační typ jazyka, je s ním spojeno jméno proměnné, které vrací příslušná procedura.
Po zabudování sémantických příkazů mají procedury následující tvar:

```
VYSLEDEK:PROC(X);
 DCL (X,Y) CHAR(30) VARYING;
 CALL VYRAZ(Y);
 CALL ENTER('+',Y,0,X);
END VYSLEDEK;
TERMIN:PROC(X) RECURSIVE;
 DCL (X,Y,Z) CHAR(30) VARYING, OP CHAR(1);
 CALL FAKTOR(Y);
 OP=SEM;
 DO WHILE(OP='*' | OP='/');
  CALL SCANNER;
  CALL FAKTOR(Z);
  J=J+1;
  CALL ENTER(OP,Y,Z,T(J));
  Y=T(J);
  OP=SEM;
 END;
 X=Y;
END TERMIN;
```

6. Systémy pro zápis překladače

Existuje ještě další cesta při implementaci komunikačního jazyka a to použití systému pro zápis překladače /dále TWS/.
TWS jsou programovací prostředky, které pomocí popisu programovacího jazyka metajazykem generují překladač /kompilátor nebo interpretátor/. TWS buď v dávkovém režimu produkuje nový program nebo routiny, které jsou začleněny do uživatelského programu.
Takovým příkladem je interaktivní systém pro návrh komunikačních jazyků LANG-PAK /Heindel/.
Popis jazyka je proveden rozšířenou formou BNF v obdobném tvaru uvedeném v kapitole 4 s tím rozdílem, že metajazyk je doplněn o elementy umožňující definici obecného integeru, obecného

reálného čísla, člena z množiny konečných slov, znakového řetězce, který je následován určitou posloupností symbolů a dále je metajazyk rozšířen o instrukce řídící rozkladač.
Metajazyk obsahuje také skokové instrukce do semantických routin, které jsou vyvolávány v průběhu rozkladu zdrojového textu.
Uživatel musí připravit popis jazyka zapsaný v metajazyku a pochopitelně také semantické routiny.
Zvláštností systému je, že je důsledně interaktivní a uživatel může v průběhu výpočtu měnit gramatiku a odlaďovat tak svůj komunikační jazyk.
Výhodou těchto systémů je jejich flexibilita při změně komunikačního jazyka, která se prakticky promítá pouze do sémantické části.
Nevýhodou je menší účinnost programu získaného pomocí TWS, určité časové nároky na prostudování a ovládnutí metajazyka i celého systému a dále u routin začleněných do uživatelského programu i větší paměťové nároky ve srovnání s kódem lexikálního a syntaktického analysátoru přímo naprogramovaného.

Závěr

Jsou nastíněny některé techniky implementace komunikačního jazyka. S růstem požadavků na vytváření systémů člověk-počítač porostou i nároky na vytváření těch programových routin, které dosud byly doménou pouze profesionálního implementátora kompilačních systémů.
Existuje však už řada technik a programových podpůrných prostředků, které umožňují aplikačnímu programátorovi i tyto úlohy v rozumné době zvládnout.

Literatura

Naur P.: Revised report on the algorithmic language ALGOL 60, CACM, Vol.6, No 1 /January 1963/

Grice D.: Compiler construction for digital computers, Wiley & Sons 1971

Hopgood F.: Metódy kompilovania, preklad ALFA 1973

Knuth D.: The art of programming, Addison-Wesley 1973

Heindel L., Roberto J.: LANG-PAK - An interactive language design system, Elsevier 1975