5 JAZYKOVÉ ŠTRUKTÚRY A KOMPILÁTORY

V tejto kapitole sa budeme snažiť vyvinúť *kompilátor* (*prekladač*) jednoduchého programovacieho jazyka. Proces tvorby tohto kompilátora nám súčasne poslúži ako príklad systematického, dobre štruktúrovaného vývoja zložitého a veľkého programu. V tomto zmysle je prekladač vítanou aplikáciou metód štruktúrovania programov a údajov, uvedených a rozpracovaných v predchádzajúcich kapitolách. Navyše naším ďalším cieľom je všeobecný úvod do problematiky štruktúry a činnosti kompilátorov. Jej poznanie a zvládnutie umožní hlbšie pochopiť umenie programovania vo vyšších programovacích jazykoch a programátorovi zjednoduší celkový proces vývoja programovacích systémov pre špecifické ciele a aplikačné oblasti. Pretože zložitosť kompilátorovej problematiky je všeobecne známa, treba túto kapitolu pokladať za úvodnú a prehľadovú. Azda za najdôležitejšiu možno považovať skutočnosť, že štruktúra jazyka sa odzrkadľuje v štruktúre jeho kompilátora, z čoho vyplýva, že jazyková zložitosť (alebo jednoduchosť) určuje zložitosť jeho kompilátora. Začneme preto opisom skladby jazyka, a potom sa sústredíme výlučne na jednoduché štruktúry, ktorých použitie vedie k jednoduchým modulárnym prekladačom. Ukazuje sa, že takéto jazykové konštrukcie sú vhodne aplikovateľné takmer vo všetkých praktických programovacích jazykoch.

5.1 DEFINÍCIA A ŠTRUKTÚRA JAZYKA

Každý jazyk je založený na nejakom slovníku. Jeho prvky sa obyčajne nazývajú slová; v teórii formálnych jazykov im hovoríme (základné) symboly. Pre každý jazyk je charakteristické, že určité postupnosti slov

sa považujú za správne, dobre vytvorené vety tohto jazyka, pričom iné sa považujú za nesprávne alebo zle vytvorené. Na základe čoho sa dá rozhodnúť, či daná postupnosť slov je správna veta alebo nie? Je to gramatika, *syntax* alebo štruktúra jazyka. Všeobecne definujeme syntax ako množinu pravidiel alebo formúl, ktorá definuje množinu (formálne konkrétnych) viet. Oveľa dôležitejšou vlastnosťou množiny takýchto pravidiel, okrem rozhodnutia, či daná postupnosť slov je veta alebo nie, je, že každej vete pripisujú istú štruktúru, ktorá je nápomocná pri určovaní významu vety. Je teda jasné, že syntax a *sémantika* (t. j. význam) sú úzko späté. Štrukturálne definície sa preto považujú za pomocné, vzhľadom na určité vyššie zámery. To nás, samozrejme, nesmie odradiť od začiatočného štúdia výlučne štrukturálnych aspektov bez uvažovania sémantiky a interpretácie.

Zoberme si napr. túto vetu: *Mačky spia.* Slovo *mačky* je podmet a *spia* prísudok tejto vety. Táto veta patrí do jazyka, ktorého syntax môže byť definovaná nasledujúcim spôsobom:

⟨veta⟩ : : = ⟨podmet⟩ ⟨prísudok⟩
⟨podmet⟩ : : = *mačky* | *psy*
⟨prísudok⟩ : : = *spia* | *jedia*

Význam týchto troch riadkov je takýto:

1. Vetu tvorí podmet, za ktorým nasleduje prísudok.
2. Podmetom môže byť buď slovo *mačky*, alebo slovo *psy*.
3. Prísudkom môže byť buď slovo *spia*, alebo slovo *jedia*.

Hlavnou myšlienkou uvedenej syntaxe je, že vetu možno odvodiť zo začiatočného symbolu ⟨veta⟩ opakovanou aplikáciou prepisovacích pravidiel.

Formalizmus alebo zápis (notácia) použitý na vyjadrenie týchto pravidiel sa nazýva *Backusova-Naurova forma* (BNF). Prvý raz sa použila pri definovaní jazyka algol 60 [5-7]. Vetné členy ⟨veta⟩, ⟨podmet⟩ a ⟨prísudok⟩ sa nazývajú *neterminálne symboly*; slová mačky, psy, spia a jedia nazývame *terminálne symboly* a prepisovacím pravidlám sa niekedy zvykne hovoriť, že sú to *produkcie*. Symboly : : = a | nazývame *metasymboly* jazyka BNF. Ak by sme sa rozhodli používať stručnejšiu formu zápisu, neterminálne symboly označíme veľkými

písmenami a terminálne malými písmenami, mohol by náš príklad vety vyzerať takto:

Príklad 1.

$$\begin{aligned} S &::= AB \\ A &::= x/y \\ B &::= z/w \end{aligned} \tag{5.1}$$

Jazyk definovaný uvedenou syntaxou pozostáva zo štyroch viet xz, yz, xw, yw.

Na upresnenie uvedieme tieto matematické definície:

1. Nech je jazyk $L = L(T, N, P, S)$ špecifikovaný pomocou
   a) slovníka T terminálnych symbolov,
   b) množiny N neterminálnych symbolov (gramatických kategórií),
   c) množiny P prepisovacích pravidiel (syntaktických pravidiel),
   d) symbolu S (z množiny N) nazývaného *začiatočný symbol.*

2. Jazyk $L(T, N, P, S)$ je množina reťazcov terminálnych symbolov ξ, ktoré možno generovať (derivovať) zo začiatočného symbolu S podľa definície 3.

$$L = \{\xi \mid S \overset{*}{\Rightarrow} \xi \quad \text{a} \quad \xi \in T^*\} \tag{5.2}$$

(Na označenie reťazcov symbolov budeme používať grécke písmená.) T^* označuje množinu všetkých postupností symbolov zo slovníka T.

3. Reťazec σ_n možno generovať z reťazca σ_0 vtedy a len vtedy, ak existujú také reťazce $\sigma_1, \sigma_2, \ldots, \sigma_{n-1}$, že každý reťazec σ_i môže byť priamo generovaný z reťazca σ_{i-1} podľa definície 4.

$$(\sigma_0 \overset{*}{\Rightarrow} \sigma_n) \leftrightarrow ((\sigma_{i-1} \Rightarrow \sigma_i) \quad \text{pre} \quad i = 1, \ldots, n) \tag{5.3}$$

4. Reťazec η možno priamo generovať z reťazca ξ vtedy a len vtedy, ak existujú také reťazce $\alpha, \beta, \xi', \eta'$, že platí:
   a) $\xi = \alpha\xi'\beta$,
   b) $\eta = \alpha\eta'\beta$,
   c) P obsahuje prepisovacie pravidlo $\xi' ::= \eta'$.

Poznamenávame, že zápis $\alpha ::= \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ budeme používať ako skrátenú formu zápisu množiny prepisovacích pravidiel $\alpha ::= \beta_1$, $\alpha ::= \beta_2, \ldots, \alpha ::= \beta_n$.

Napríklad reťazec xz z príkladu 1 možno generovať nasledujúcou postupnosťou priamych krokov generovania: $S \Rightarrow AB \Rightarrow xB \Rightarrow xz$; teda $S \overset{*}{\Rightarrow} xz$, a pretože $xz \in T^*$, xz je veta jazyka, t. j. $xz \in L$. Všimnime si, že neterminálne symboly A a B sa objavia iba v nekoncových krokoch, zatiaľ čo koncové kroky musia viesť k reťazcu, ktorý obsahuje iba terminálne symboly. Gramatické pravidlá nazývame preto prepisovacie, lebo určujú, akým spôsobom možno nové formy generovať alebo prepisovať.

Jazyk nazývame bezkontextovým vtedy a len vtedy, ak sa dá definovať prostredníctvom bezkontextovej (nezávislej od kontextu) množiny prepisovacích pravidiel. Množina prepisovacích pravidiel je bezkontextová vtedy a len vtedy, ak všetky jej prvky majú tvar

$$A ::= \xi \qquad (A \in N, \ \xi \in (N \cup T)^*)$$

t. j. ak ľavá strana pozostáva z jednoduchého neterminálneho symbolu, ktorý možno nahradiť (prepísať) symbolom ξ bez ohľadu na kontext, v ktorom sa A vyskytuje. Ak má prepisovacie pravidlo tvar

$$\alpha A \beta ::= \alpha \xi \beta$$

hovoríme, že je kontextové a náhrada symbolu A symbolom ξ je možná len v rámci kontextu α a β. My sa sústredíme iba na bezkontextové systémy.

Príklad 2 nám názorne ukazuje, ako možno rekurzívne generovať prostredníctvom konečnej množiny prepisovacích pravidiel nekonečne veľa viet.

Príklad 2.

$$\begin{aligned} S &::= xA \\ A &::= z \mid yA \end{aligned} \tag{5.4}$$

Zo začiatočného symbolu S možno potom generovať nasledujúce vety:

xz
xyz
$xyyz$
$xyyyz$
...

5.2 ANALÝZA VETY

Úlohou jazykových prekladačov alebo procesorov je najprv rozpoznávanie viet a vetných štruktúr a potom ich generovanie. To znamená, že všetky kroky generovania, ktoré vedú k vete, sa musia po jej rozpoznaní zrekonštruovať a súčasne použiť ako návratová cesta. To je však vo všeobecnosti veľmi zložitá a niekedy dokonca nemožná úloha. Jej zložitosť silne závisí od druhu prepisovacích pravidiel použitých v definícii jazyka. Úlohou teórie syntaktickej analýzy je vyvinúť rozpoznávacie algoritmy pre jazyky so značne zložitými štrukturálnymi pravidlami. Našou úlohou však bude načrtnúť spôsob tvorby analyzátorov, ktoré budú dostatočne jednoduché a efektívne pre praktické účely. To znamená, že výpočtové úsilie potrebné na analýzu vety musí byť lineárnou funkciou dĺžky vety; v najhoršom prípade môže byť funkčná závislosť $n \cdot \log n$, pričom n je dĺžka vety. Pochopiteľne, nemôžeme sa zaoberať problémami nájdenia rozpoznávacieho algoritmu pre ľubovoľný jazyk. Budeme preto pracovať pragmaticky a v opačnom smere: najskôr budeme definovať efektívny algoritmus, a potom určíme triedu jazykov, ktoré možno pomocou neho analyzovať [5-3].

Prvým dôsledkom základnej požiadavky efektívnosti je, že výber každého kroku analýzy musí závisieť iba od súčasného stavu výpočtu a od nasledujúceho (práve načítaného) symbolu. Ďalšia a najdôležitejšia požiadavka je, aby žiadny krok nebol neskôr odvolaný. Tieto dve požiadavky sú všeobecne známe pod pojmom *metóda s predsnímaním jedného symbolu dopredu bez návratu.*

Základná metóda, ktorou sa budeme zaoberať, sa nazýva *syntaktická analýza zhora nadol.* Charakteristická je úsilím o rekonštrukciu krokov generovania (ktoré vo všeobecnosti tvoria štrukturálny strom) z ich začiatočného symbolu do konečnej vety, t. j. zhora nadol [5-5] a [5-6]. Začnime opäť príkladom 1: Máme danú vetu *Psy jedia*, o ktorej musíme rozhodnúť, či patrí do jazyka alebo nie. Podľa definície je to jedine v prípade, ak danú vetu možno (alebo nemožno) generovať zo začiatočného symbolu ⟨veta⟩. Z gramatických pravidiel vieme, že každá veta musí mať podmet, za ktorým nasleduje prísudok. Zvyšnú časť úlohy rozdelíme na dve časti: najprv rozhodneme, či sa určitá začiatočná časť vety dá (alebo nedá) generovať zo symbolu ⟨podmet⟩. Ako vidíme, symbol *psy* sa dá priamo generovať, preto ho môžeme považovať za spracovaný; zo vstupnej postupnosti ho vylúčime (t. j. načítame ďalší symbol) a prikročíme k druhej časti úlohy, a to k rozhodnutiu, či sa zvyšná časť vety dá (nedá) generovať zo symbolu ⟨prísudok⟩. Pretože sa dá generovať, môžeme výsledok analýzy považovať za správny. Celý proces analýzy si môžeme názorne zobraziť nasledujúcou schémou; v ľavej časti tejto schémy sú uvedené úlohy, ktoré treba rozriešiť, v pravej časti zvyšok vstupnej postupnosti:

⟨veta⟩	*psy jedia*
⟨podmet⟩ ⟨prísudok⟩	*psy jedia*
psy ⟨prísudok⟩	*psy jedia*
⟨prísudok⟩	*jedia*
jedia	*jedia*
—	—

Druhý príklad znázorňuje schému analýzy vety *xyyz* podľa prepisovacích pravidiel uvedených v príklade 2.

S	$xyyz$
xA	$xyyz$
A	yyz
yA	yyz
A	yz
yA	yz
A	z
z	z
—	—

Pretože proces sledovania jednotlivých krokov generovania vety sa nazýva *syntaktická analýza*, uvedený postup predstavuje *algoritmus syntaktickej analýzy*. Jednotlivé odvodzovacie kroky možno v uvedených dvoch príkladoch uskutočniť na základe prezretia ďalšieho symbolu vstupnej postupnosti. Žiaľ, ako uvidíme z nasledujúceho príkladu, nie je to vždy možné.

Príklad 3.

$$\begin{aligned} S &::= A \mid B \\ A &::= xA \mid y \\ B &::= xB \mid z \end{aligned} \tag{5.5}$$

Ak sa pokúsime o syntaktickú analýzu vety $xxxz$

S	$xxxz$
A	$xxxz$
xA	$xxxz$
A	xxz
xA	xxz
A	xz
xA	xz
A	z

zistíme, že sme uviazli. Ťažkosť vznikne už pri prvom kroku, v ktorom nedokážeme prezretím jediného ďalšieho symbolu rozhodnúť, či symbol S treba nahradiť symbolom A alebo symbolom B. Možné riešenie spočíva v pokračovaní podľa jednej z možných alternatív a v návrate po prejdenej ceste, ak ďalší postup už nie je možný. Táto metóda sa nazýva *prehľadávanie s návratom* (hovorili sme o nej v predchádzajúcej kapitole — pozn. prekl.). Pre jazyk z príkladu 3 neexistuje obmedzenie počtu nevybavených krokov. Taká situácia je rozhodne najmenej želateľná, a preto by sme sa v praxi mali vyhýbať jazykovým štruktúram spôsobujúcim prehľadávanie s návratom. V dôsledku toho sa budeme zaoberať iba takými gramatickými systémami, ktoré spĺňajú obmedzenie, že začiatočné symboly alternatívnych pravých častí prepisovacích pravidiel budú odlišné.

Tvrdenie 1.

Ak máme dané prepisovacie pravidlo

$$A ::= \xi_1 \mid \xi_2 \mid \ldots \mid \xi_n$$

tak množiny začiatočných symbolov všetkých viet, ktoré možno generovať zo symbolov ξ_i, musia byť disjunktné, t. j.

$$\text{first}(\xi_i) \cap \text{first}(\xi_j) = \emptyset \quad \text{pre všetky} \quad i \neq j$$

Množina first (ξ) je množinou všetkých terminálnych symbolov, ktoré sa môžu vyskytnúť ako prvý symbol viet odvodených zo symbolu ξ. Nech je táto množina vypočítateľná podľa týchto zákonitostí:

1. Prvý symbol argumentu je terminálny symbol:

$$\text{first}(a\xi) = \{a\}$$

2. Prvý symbol je neterminálny symbol s prepisovacím pravidlom

$$A ::= \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

Potom

$$\text{first}(A\xi) = \text{first}(\alpha_1) \cup \text{first}(\alpha_2) \cup \ldots \cup \text{first}(\alpha_n)$$

Všimnime si, že v príklade 3 platí $x \in \text{first}(A)$ a súčasne $x \in \text{first}(B)$. Na základe toho je prvým pravidlom porušené tvrdenie 1. Je však skutočne triviálne nájsť syntax jazyka uvedeného v príklade 3, ktorá by vyhovovala tvrdeniu 1. Riešenie spočíva v odložení faktorizácie dovtedy, kým sa „nevybavia" všetky symboly x. Nasledujúce prepisovacie pravidlá sú ekvivalentné s prepisovacími pravidlami (5.5) v tom zmysle, že generujú tú istú množinu viet:

$$\begin{aligned} S &::= C \mid xS \\ C &::= y \mid z \end{aligned} \tag{5.5a}$$

Žiaľ, tvrdenie 1 nás neuchráni pred ďalšou ťažkosťou. Majme takýto príklad:

Príklad 4.

$$\begin{aligned} S &::= Ax \\ A &::= x \mid \varepsilon \end{aligned} \tag{5.6}$$

Symbol ε označuje prázdny reťazec symbolov. Keď sa pokúsime analyzovať vetu x, môžeme sa dostať do „slepej uličky":

S	x
Ax	x
xx	x
x	—

Ťažkosť vzniká preto, lebo sme nepokračovali prepisovacím pravidlom $A::=x$, ale prepisovacím pravidlom $A::=\varepsilon$. Táto situácia sa nazýva problém prázdneho reťazca a vzniká iba v prípade neterminálnych symbolov, ktoré môžu generovať prázdny reťazec. Aby sme sa vyhli tejto situácii, zavedieme ďalšie tvrdenie.

Tvrdenie 2.

Pre každý symbol $A \in N$, ktorý generuje prázdny reťazec ($A \overset{*}{\Rightarrow} \varepsilon$), musí byť množina jeho začiatočných symbolov disjunktná s množinou symbolov, ktoré môžu nasledovať za ktorýmkoľvek reťazcom, generovaným zo symbolu A, t. j.

$$\text{first}(A) \cap \text{follow}(A) = \emptyset$$

Množina follow(A) sa vypočíta zohľadnením každého prepisovacieho pravidla P_i tvaru

$$X::=\xi A\eta$$

a množiny $S_i = \text{first}(\eta_i)$. Množina follow$(A)$ je zjednotením všetkých množín S_i. Ak je aspoň jeden reťazec η_i schopný generovať prázdny reťazec, musí byť množina follow(X) obsiahnutá v množine follow(A). V príklade 4 je porušené tvrdenie 2 pre symbol A, pretože

$$\text{first}(A) = \text{follow}(A) = \{x\}$$

Zaužívaným spôsobom vyjadrenia opakujúceho sa reťazca symbolov je použitie rekurzívnej definície vetnej konštrukcie. Napríklad prepisovacie pravidlo

$$A::=B\,|\,AB$$

opisuje množinu reťazcov B, BB, BBB, ... Jeho použitie je však znemožnené na základe tvrdenia 1, pretože

$$\text{first}(B) \cap \text{first}(AB) = \text{first}(B) \neq \emptyset$$

Ak nahradíme prepisovacie pravidlo mierne modifikovanou verziou

$$A::=\varepsilon\,|\,AB$$

generujúcou reťazce ε, B, BB, BBB, ..., porušíme tvrdenie 2, pretože

$$\text{first}(A) = \text{first}(B)$$

a teda

$$\text{first}(A) \cap \text{follow}(A) \neq \emptyset$$

Dve spomenuté tvrdenia nepochybne zakazujú použitie definície s ľavou rekurziou. Jednoduchou metódou, ako sa vyhnúť takýmto tvarom, je buď použitie pravej rekurzie

$$A::=\varepsilon\,|\,BA$$

alebo rozšírenie syntaxe jazyka BNF, aby umožňoval explicitné vyjadrenie opakovania; uskutočníme to tak, že zápisom $\{B\}$ označíme množinu reťazcov

$$\varepsilon,\ B,\ BB,\ BBB,\ \ldots$$

Prirodzene, musíme si byť vedomí toho, že každá takáto konštrukcia je schopná generovať prázdny reťazec. Zložené zátvorky { a } predstavujú metasymboly rozšíreného jazyka BNF.

Z uvedeného dôvodu a z transformácie prepisovacích pravidiel (5.5) na (5.5a) sa môže zdať, že „trik" transformácií gramatík by mohol byť všeliekom na všetky problémy týkajúce sa syntaktickej analýzy. Musíme však pamätať na to, že vetná štruktúra je pomocným prostriedkom pri určovaní vetného významu; teda vysvetlenia významu vetnej konštrukcie sú obyčajne vyjadrené pomocou významov vetných komponentov. Uvažujme napr. o gramatike, kde sa jazykové výrazy skladajú z operandov a, b, c a zo znamienka mínus znamenajúceho odčítanie:

$$S::=A\,|\,S-A$$
$$A::=a\,|\,b\,|\,c$$

V súlade s touto gramatikou má veta $a-b-c$ štruktúru, ktorá sa dá vyjadriť pomocou zátvoriek takto: $((a-b)-c)$. Keby sme však túto gramatiku transformovali na syntakticky ekvivalentný tvar bez ľavej rekurzie

$$S::=A\,|\,A-S$$
$$A::=a\,|\,b\,|\,c$$

tá istá veta by nadobudla inú štruktúru, ktorá sa dá vyjadriť v tvare $(a - (b - c))$. Ak uvážime zvyčajný význam odčítania, zistíme, že uvedené dva tvary sú sémanticky neekvivalentné.

Aké ponaučenie teda pre nás z uvedeného vyplýva? Predovšetkým také, že keď definujeme zmysluplný jazyk, musíme mať pri návrhu jeho syntaxe na zreteli vždy aj jeho sémantické štruktúry, pretože syntax musí zodpovedať sémantike.

5.3 KONŠTRUKCIA SYNTAKTICKÉHO GRAFU

V predchádzajúcom článku sme uviedli algoritmus syntaktickej analýzy zhora nadol vhodnej pre gramatiky, ktoré spĺňajú obmedzujúce tvrdenia 1 a 2. Teraz sa budeme snažiť transformovať tento algoritmus na konkrétny program. Existujú dve podstatne rozdielne techniky, ktoré sa dajú použiť. Prvá spočíva v návrhu všeobecného programu syntaktickej analýzy zhora nadol, platného pre všetky možné gramatiky (spĺňajúce tvrdenia 1 a 2). V tomto prípade musia byť jednotlivé gramatiky reprezentované nejakou štruktúrou údajov, s ktorou je daný program schopný pracovať. Takýto všeobecný syntaktický analyzátor je v istom zmysle riadený štruktúrou údajov; program je teda riadený tabuľkou. Druhá technika spočíva vo vyvinutí programu syntaktickej analýzy zhora nadol, ktorý je špecifický pre daný jazyk, a v jeho systematickej konštrukcii v súlade s množinou prepisovacích pravidiel zobrazujúcich danú syntax do postupnosti príkazov, t. j. do programu. Obidve techniky majú svoje výhody i nevýhody. Pri vývoji kompilátora daného programovacieho jazyka sa vyžaduje vysoký stupeň flexibility a parametrizácie všeobecného syntaktického analyzátora, pričom v prípade špecifického syntaktického analyzátora dospejeme k oveľa efektívnejším a ľahšie ovládateľným systémom. Preto sa uprednostňuje tento druhý spôsob. V oboch prípadoch je výhodné reprezentovať danú syntax pomocou syntaktického (derivačného) grafu, ktorý zobrazuje tok riadenia v rámci syntaktickej analýzy vety.

Charakteristickou vlastnosťou analýzy zhora nadol je, že jej cieľ je známy už na začiatku. Cieľom je rozpoznať vetu, t. j. reťazec symbolov

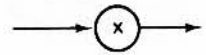
generovateľný zo začiatočného symbolu. Aplikácia prepisovacieho pravidla, t. j. nahradenie jednoduchého symbolu reťazcom symbolov, zodpovedá rozdeleniu jednoduchého cieľa na reťazec podcieľov, sledovaných v špecifickom poradí. Metóda zhora nadol sa potom nazýva aj cieľovo orientovaná syntaktická analýza. Pri konštrukcii syntaktického analyzátora sa dá jednoducho využiť zrejmá súvislosť medzi neterminálnymi symbolmi a cieľmi: zostrojíme podprogram syntaktickej analýzy pre každý neterminálny symbol. Cieľom každého z týchto podprogramov je rozpoznanie podvety, generovateľnej z jej zodpovedajúceho neterminálneho symbolu. Pretože však chceme zostrojiť graf reprezentujúci celý syntaktický analyzátor, bude potrebné, aby každý neterminálny symbol bol zobrazený do podgrafu. To nás vedie k nasledujúcim zásadám konštrukcie syntaktického grafu.

A1. Každý neterminálny symbol A so zodpovedajúcou množinou pravidiel

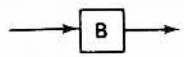
$$A ::= \xi_1 \mid \xi_2 \mid \ldots \mid \xi_n$$

sa zobrazí do syntaktického grafu A, ktorého štruktúra sa určuje na základe pravej strany prepisovacieho pravidla podľa zásad A2 až A6.

A2. Každý výskyt terminálneho symbolu x v ξ_i zodpovedá príkazu na rozpoznanie tohto symbolu a načítanie ďalšieho symbolu zo vstupnej vety. V grafe sa to zobrazuje týmto spôsobom (symbol x sa uvádza v krúžku, do ktorého smeruje a z ktorého vychádza orientovaná hrana):



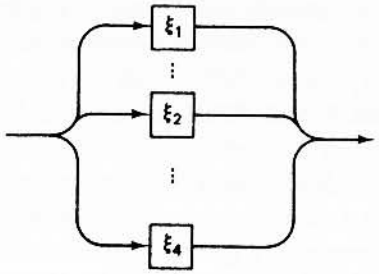
A3. Každý výskyt neterminálneho symbolu B v ξ_i zodpovedá volaniu programu pre analýzu B. V grafe to zobrazujeme takto:



A4. Prepisovacie pravidlo tvaru

$$A ::= \xi_1 \mid \ldots \mid \xi_n$$

sa zobrazí na graf,



v ktorom sa každý reťazec $\boxed{\xi_i}$ získa aplikáciou konštrukčných zásad A2 až A6 na reťazec ξ_i.

A5. Reťazec ξ tvaru

$$\xi = \alpha_1 \alpha_2 \ldots \alpha_m$$

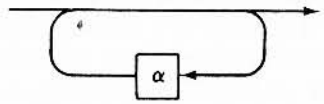
sa zobrazí na graf

v ktorom sa každá $\boxed{\alpha_i}$ získa aplikáciou konštrukčných zásad A2 až A6 na symbol α_i.

A6. Reťazec ξ tvaru

$$\xi = \{\alpha\}$$

sa zobrazí na graf



v ktorom sa $\boxed{\alpha}$ získa aplikáciou konštrukčných zásad A2 až A6 na symbol α.

Príklad 5.

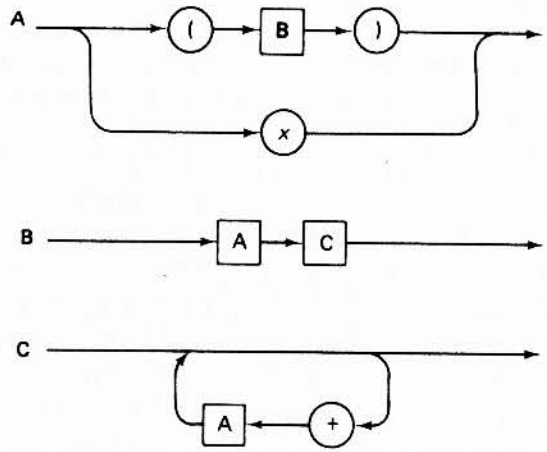
$$\begin{aligned} A &::= x \mid (B) \\ B &::= AC \\ C &::= \{+A\} \end{aligned} \tag{5.7}$$

Symboly $+$, x, (,) predstavujú terminálne symboly. Zložené zátvorky {, } patria do rozšírenej syntaxe BNF, teda sú to metasymboly. Jazyk

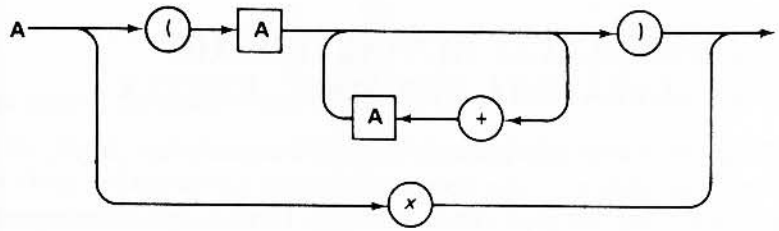
generovateľný z A pozostáva z výrazov, ktoré sa skladajú z operandov x, operátora $+$ a zátvoriek. Príkladmi viet sú

$$\begin{gathered} x \\ (x) \\ (x + x) \\ ((x)) \\ \vdots \end{gathered}$$

Grafy, ktoré získame aplikáciou šiestich konštrukčných zásad, sú znázornené na *obr. 5.1*. Všimnime si, že vhodnou substitúciou C na B a B na A môžeme tento systém grafov nahradiť jediným grafom (*obr. 5.2*).



Obr. 5.1. Syntaktické grafy k príkladu 5



Obr. 5.2. Redukovaný syntaktický graf k príkladu 5

Syntaktický graf je ekvivalentnou reprezentáciou gramatiky jazyka. Môžeme ho použiť namiesto množiny prepisovacích pravidiel vyjadrených v jazyku BNF. Je to veľmi vhodná forma a v mnohých (ak nie vo všetkých) prípadoch uprednostňovaná pred BNF. Určite poskytuje jasnejší a stručnejší obraz o jazykovej štruktúre a navyše umožňuje oveľa rýchlejšie pochopiť proces syntaktickej analýzy. Graf je pre tvorcov jazyka vhodná forma v procese jeho návrhu. Príklady syntaktických špecifikácií úplných jazykov sú uvedené v článku 5.7 (pre jazyk PL/0) a v prílohe B (pre jazyk pascal).

Obmedzujúce tvrdenia 1 a 2 sme zaviedli preto, aby sme umožnili deterministickú syntaktickú analýzu s predsnímaním jedného symbolu dopredu. Ako sa tieto tvrdenia prejavia v syntaktickom grafe? So zreteľom na prehľadnosť a zrozumiteľnosť grafovej reprezentácie takto:

1. Tvrdenie 1 sa interpretuje ako požiadavka vetvenia grafu: výber vhodnej vetvy musí byť uskutočniteľný na základe prezretia najbližšieho ďalšieho symbolu vety. To znamená, že každá vetva musí začínať iným symbolom.

2. Tvrdenie 2 sa interpretuje ako požiadavka na prechod grafom: ak je možné nejakým grafom A prejsť bez akéhokoľvek načítania vstupného symbolu, musí byť táto „prázdna vetva" charakterizovaná množinou všetkých symbolov, ktoré môžu nasledovať za A. (To bude ovplyvňovať rozhodnutie, ktoré treba robiť pri vstupe do tejto vetvy.)

Nie je ťažké overiť, či systém grafov spĺňa uvedené dve prispôsobené pravidlá, a to aj bez toho, že by sme použili BNF reprezentáciu gramatiky. Pomôžeme si tým, že pre každý graf A určíme množiny first(A) a follow(A). Aplikácia tvrdení 1 a 2 je potom bezprostredná. Systém grafov, ktorý spĺňa uvedené dve tvrdenia, nazývame *deterministickým syntaktickým grafom.*

5.4 KONŠTRUKCIA SYNTAKTICKÉHO ANALYZÁTORA PRE DANÚ SYNTAX

Program, ktorý akceptuje a syntakticky analyzuje jazyk, je bez ťažkostí odvoditeľný z jeho deterministického syntaktického grafu (ak, samozrejme, takýto graf vôbec existuje). Graf v zásade reprezentuje

vývojový diagram programu. Pri vývoji takéhoto programu sa však odporúča postupovať presne podľa danej množiny pravidiel podobných pravidlám, ktoré usmerňujú prechod od BNF ku grafovej reprezentácii syntaxe jazyka. (Tieto pravidlá sú uvedené v ďalšom.) Sú použiteľné v špecifických prípadoch, ktoré môžu byť reprezentované hlavným programom, do ktorého sú začlenené procedúry zodpovedajúce rôznym podcieľom a procedúra umožňujúca prístup k ďalšiemu symbolu.

Pre jednoduchosť predpokladajme, že veta, ktorú máme analyzovať, je reprezentovaná súborom input a terminálnymi symbolmi sú jednotlivé znaky. Požadujeme, aby existovala premenná typu znak reprezentujúca ďalší načítaný symbol. Načítanie ďalšieho symbolu môžeme vyjadriť príkazom

read (*ch*)

Hlavný program bude potom pozostávať zo začiatočného príkazu načítania prvého symbolu, nasledovaného príkazom, ktorý zahájí celý proces syntaktickej analýzy (čo je naším hlavným cieľom). Jednotlivé procedúry zodpovedajúce príslušným cieľom analýzy alebo grafom získame na základe nasledujúcich zásad. Označme príkaz, ktorý získame prekladom grafu S, symbolom $T(S)$.

Zásady pre preklad grafu do programu:

B1. Vhodnými substitúciami zredukovať systém grafov na čo najväčší možný počet individuálnych grafov.

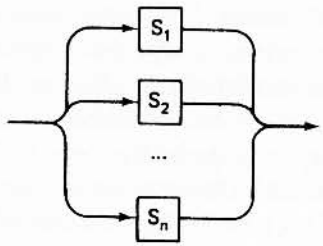
B2. Preložiť každý graf do deklarácie procedúry podľa zásad B3 až B7.

B3. Postupnosť prvkov

sa preloží do zloženého príkazu

begin $T(S_1)$; $T(S_2)$; ...; $T(S_n)$ **end**

B4. Výber prvkov



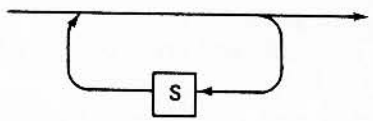
sa preloží buď na príkaz výberu, alebo na podmienený príkaz

```
case ch of          if ch in L1 then T(S1) else
L1: T(S1);          if ch in L2 then T(S2) else
L2: T(S2);          :
:                   if ch in Ln then T(Sn) else
Ln: T(Sn)           error
end
```

kde L_i označuje množinu začiatočných symbolov konštrukcie S_i (L_i = = first (S_i)).

Poznámka: Ak L_i pozostáva z jednoduchého symbolu a, tak možno výraz *ch* **in** L_i vyjadriť ako $ch = a$.

B5. Cyklus v tvare



sa preloží na príkaz

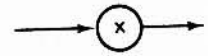
while *ch* **in** L **do** $T(S)$

kde $T(S)$ je výsledok prekladu S podľa zásad B3 až B7 a L je množina L = first (S) (pozri predchádzajúcu poznámku).

B6. Prvok grafu, ukazujúci na iný graf A

sa preloží na príkaz vyvolania procedúry A.

B7. Prvok grafu, ukazujúci na terminálny symbol x,



sa preloží na príkaz

if $ch = x$ **then** read (*ch*) **else** error

kde error predstavuje procedúru, ktorá sa vyvoláva v prípade výskytu chybnej syntaktickej konštrukcie.

Použitie uvedených zásad si môžeme ukázať na príklade prekladu redukovaného grafu (príklad 5, *obr. 5.2*) na program syntaktického analyzátora (program 5.1).

PROGRAM 5.1. *Program syntaktického analyzátora pre gramatiku z príkladu 5*

```
program Analyzátor (input, output);
  var ch: char;
  procedure A;
  begin if ch = 'x' then read (ch) else
        if ch = '(' then
        begin read (ch); A;
          while ch = '+' do
            begin read (ch); A
            end;
          if ch = ')' then read (ch) else error
        end else error
  end;
begin read (ch); A
end.
```

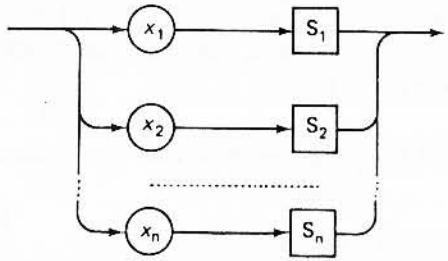
V uvedenom preklade sme využili niektoré zaužívané programovacie zásady, ktoré zjednodušujú program. Preklad literálu by napr. mohol vyústiť do takejto podoby:

```
if ch = 'x' then
  if ch = 'x' then read (ch) else error
  else ...
```

Použitím spomenutých programovacích zásad sme získali jednoduchšiu formu uvedenú v programe. Príkazy čítania read v piatom a siedmom riadku programu sú výsledkom podobných redukcií.

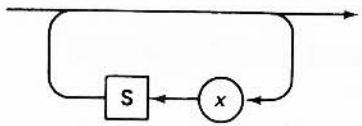
Zdá sa, že by bolo rozumné zistiť všetky prípady, kde vo všeobecnosti sú takéto redukcie možné, a tieto potom vyjadriť priamo pomocou grafov. Nasledujúce dve dodatočné zásady znázorňujú dva relevantné prípady:

B4a.



if $ch = 'x_1'$ **then begin** read (ch); $T(S_1)$ **end else**
if $ch = 'x_2'$ **then begin** read (ch); $T(S_2)$ **end else**
...
if $ch = 'x_n'$ **then begin** read (ch); $T(S_n)$ **end else** error

B5a.



while $ch = 'x'$ **do**
  begin read (ch); $T(S)$ **end**

Navyše často sa vyskytujúcu konštrukciu

read (ch); $T(S)$;
while B **do**
  begin read (ch); $T(S)$ **end**

môžeme vyjadriť v zjednodušenom tvare

repeat read (ch); $T(S)$ **until** B (5.8)

Ako ste si už iste všimli, nezaoberali sme sa doteraz podrobnejšie procedúrou error. Pretože teraz nás zaujíma iba to, či je daný vstupný reťazec syntakticky správny alebo nie, môžeme túto procedúru pokladať za ukončenie realizácie programu.

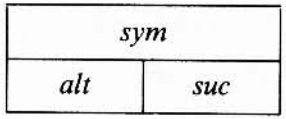
Prirodzene, v praxi sa používajú oveľa dômyselnejšie prostriedky spracúvania syntaktických chýb. Tieto budú predmetom článku 5.9.

5.5 KONŠTRUKCIA PROGRAMU SYNTAKTICKEJ ANALÝZY RIADENÉHO TABUĽKOU

Namiesto zostrojenia špecifického programu syntaktickej analýzy pre každý jazyk a jeho syntax, podľa pravidiel uvedených v predchádzajúcej kapitole, je možné vytvoriť jednoduchý, všeobecný program syntaktickej analýzy. Gramatiky jednotlivých jazykov predstavujú začiatočné údaje pre takýto všeobecný analyzátor a poskytnú sa mu ešte pred samotnými vetami, ktoré sa majú analyzovať. Všeobecný program sa dôsledne riadi pravidlami metódy jednoduchej syntaktickej analýzy zhora nadol. Je priamočiary, ak je príslušný syntaktický graf deterministický, t.j. zodpovedajúca gramatika umožňuje analýzu viet s predsnímaním jedného symbolu dopredu bez návratu.

Gramatika, o ktorej predpokladáme, že je reprezentovaná v tvare deterministickej množiny syntaktických grafov, sa teda prekladá do príslušnej štruktúry údajov, a nie do programovej štruktúry [5-2]. Prirodzený spôsob reprezentácie grafu spočíva v zavedení vrcholu pre každý symbol a v pospájaní takýchto vrcholov smerníkmi. Preto tabuľka už nie je iba jednoduchým poľom. Pravidlá usmerňujúce preklad sú uvedené ďalej a sú samozrejmé. Vrcholy grafu sú reprezentované štruktúrou záznam s dvoma variantmi: jedným pre terminálne symboly a druhým pre neterminálne symboly. Prvý variant je identifikovateľný tým symbolom, ktorý zastupuje, druhý variant smerníkom na štruktúru údajov reprezentujúcu príslušný neterminálny symbol. Obidva varianty obsahujú dva smerníky, z ktorých prvý ukazuje na nasledujúci symbol (t.j. na nasledovníka) a druhý na zoznam možných alternatív.

Výsledné definície typov vrchol a smerník zobrazuje schéma (5.9). Pri zobrazovaní grafovej reprezentácie bude vrchol vyzerať takto:



Ako sa ukazuje, budeme potrebovať aj prvok, reprezentujúci prázdnu postupnosť, t. j. prázdny symbol. Budeme ho označovať terminálnym prvkom, ktorý nazývame *prázdny*.

```
type smerník = ↑vrchol;
     vrchol = record suc, alt: smerník;
          case terminál: boolean of
               true: (tsym: char);
               false: (nsym: psmerník)
          end
```
(5.9)

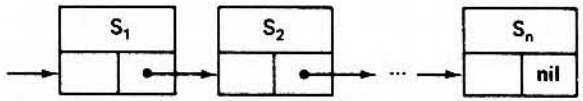
Zásady prekladu z grafov do štruktúr údajov sú analogické ako zásady B1 až B7.

Zásady prekladu grafov do štruktúr údajov:

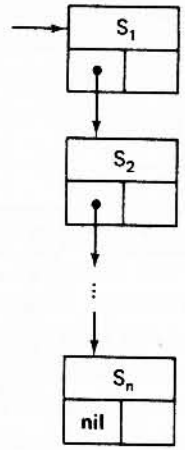
C1. Vhodnými substitúciami zredukuj systém grafov na čo najväčší možný počet individuálnych grafov.

C2. Prelož každý graf do štruktúry údajov podľa zásad C3 až C5.

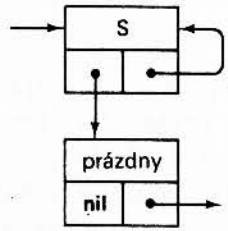
C3. Postupnosť prvkov (pozri obrázok pre zásadu B3) sa preloží na nasledujúci zoznam vrcholov:



C4. Zoznam alternatív (pozri obrázok pre zásadu B4) sa preloží na štruktúru:



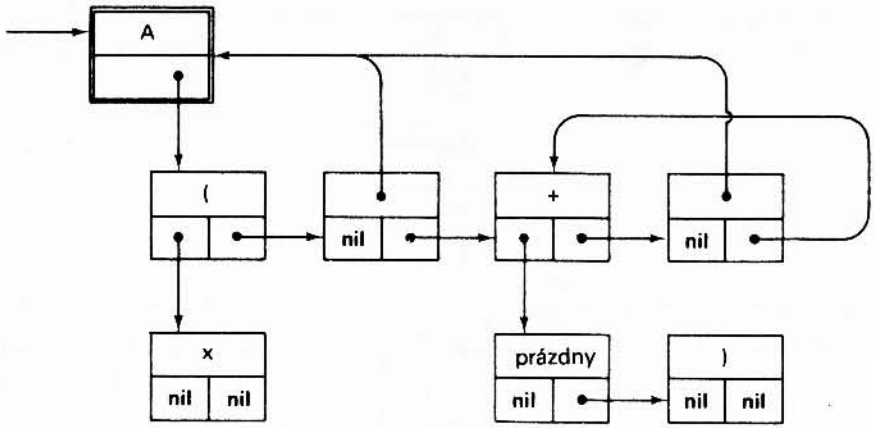
C5. Cyklus (pozri obrázok pre zásadu B5) sa preloží na štruktúru:



Ako príklad nám môže poslúžiť štruktúra na *obr. 5.3* reprezentujúca graf, ktorý zodpovedá syntaxi z príkladu 5 (*obr. 5.2*).

Štruktúra údajov je identifikovateľná prostredníctvom vrcholu nazývaného *čelo*, ktorý obsahuje meno neterminálneho symbolu (*cieľ*) reprezentovaného danou štruktúrou. Tento vrchol je v podstate nepotrebný, pretože smerník ukazujúci naň môže vlastne ukazovať priamo na začiatok príslušnej štruktúry. Môžeme ho však využiť v tom zmysle, že by obsahoval názov patričnej štruktúry.

```
type čsmerník = ↑čelo;
     čelo = record vstup: smerník;
                   sym: char
          end
```
(5.10)



Obr. 5.3. Štruktúra údajov reprezentujúca graf z *obr. 5.2*

Program na analýzu vety reprezentovanej reťazcom znakov vstupného súboru obsahuje príkaz cyklu, ktorý opisuje prechod od jedného vrcholu k ďalšiemu. Program je vyjadrený procedúrou opisujúcou interpretáciu grafu; ak sa pri nej narazí na vrchol, reprezentujúci neterminálny symbol, vykoná sa najprv interpretácia tohto grafu a až potom sa dovŕši interpretácia rozpracovaného grafu. Vidíme, že interpretačná procedúra je volaná rekurzívne. Ak sa bežný symbol (*sym*) vstupného súboru zhoduje so symbolom v momentálnom vrchole štruktúry údajov, tak ďalší krok analýzy je určený zložkou *suc*, v opačnom prípade zložkou *alt*.

```
procedure analyzátor (cieľ: čsmerník; var zhodný: boolean);
  var s: smerník;
begin s: = cieľ↑.vstup;
  repeat
    if s↑.terminál then
    begin if s↑.tsym = sym then
        begin zhodný: = true; getsym
        end                                                    (5.11)
```



```
        else zhodný: = (s↑.tsym = prázdny)
    end
    else analyzátor (s↑.nsym, zhodný);
    if zhodný then s: = s↑.suc else s: = s↑.alt
  until s = nil
  end
```

Tento program má tú vlastnosť, že len čo sa určí nový podcieľ analýzy G, hneď ho začne „sledovať" bez ohľadu na to, či sa momentálny symbol nachádza v množine začiatočných symbolov first(G) alebo nie. To znamená, že zodpovedajúci syntaktický graf musí byť zbavený výberov rôznych alternatívnych neterminálnych symbolov. Presnejšie, ak je neterminálny symbol schopný generovať prázdny reťazec, žiadna z jeho pravých častí nesmie začínať neterminálnym symbolom.

Z uvedenej schémy (5.11) sa dajú odvodiť oveľa dômyselnejšie syntaktické analyzátory riadené tabuľkou, pracujúce s menej obmedzenými triedami gramatík. Nepatrnými úpravami algoritmu sa dá dosiahnuť aj prehľadávanie s návratom, ale za cenu zmenšenia celkovej efektívnosti.

Grafová reprezentácia syntaxe má jednu závažnú nevýhodu: počítače nedokážu priamo čítať grafy. Preto štruktúry údajov, ktoré riadia syntaktický analyzátor, sa musia vytvoriť predtým, než sa zaháji proces samotnej analýzy. V tomto zmysle je reprezentácia gramatík pomocou BNF ideálnou formou vstupu pre všeobecný program syntaktickej analýzy. Ďalší článok je preto venovaný návrhu programu, ktorý číta postupnosť prepisovacích pravidiel a prekladá ich podľa zásad B1 až B6 do vnútorných štruktúr, s ktorými dokáže analyzátor (5.11) pracovať [5-8].

5.6 PREKLADAČ Z BNF DO ŠTRUKTÚR ÚDAJOV

Prekladač, ktorý na vstupe pripúšťa prepisovacie pravidlá BNF a tieto mení na inú reprezentáciu, je skutočným príkladom programu, ktorého vstupné údaje predstavujú vety nejakého jazyka. Je naozaj rozumné pokladať BNF za jazyk charakteristický svojou syntaxou,

ktorá môže byť opäť vyjadrená prostredníctvom prepisovacích pravidiel BNF. V dôsledku toho môže tento prekladač slúžiť ako ďalší príklad konštrukcie analyzátora, ktorý je navyše rozšírený na procesor svojho vstupu. Budeme preto postupovať takýmto spôsobom:

Krok 1. Budeme definovať syntax metajazyka nazývaného EBNF (rozšírený jazyk BNF).

Krok 2. Zostrojíme analyzátor EBNF podľa zásad uvedených v článku 5.4.

Krok 3. V kombinácii so syntaktickým analyzátorom riadeným tabuľkou rozšírime tento analyzátor na prekladač.

Nech je metajazyk — jazyk syntaktických prepisovacích pravidiel — opísaný týmito pravidlami:

$$\begin{aligned} &\langle \text{pravidlo} \rangle :: = \langle \text{symbol} \rangle = \langle \text{výraz} \rangle \\ &\langle \text{výraz} \rangle \quad :: = \langle \text{term} \rangle \{, \langle \text{term} \rangle\} \\ &\langle \text{term} \rangle \quad :: = \langle \text{faktor} \rangle \{\langle \text{faktor} \rangle\} \\ &\langle \text{faktor} \rangle \quad :: = \langle \text{symbol} \rangle \mid [\langle \text{term} \rangle] \end{aligned} \tag{5.12}$$

Všimnime si, že symboly, ktoré sú odlišné oproti obvyklým metasymbolom BNF, sme použili na označenie práve týchto symbolov v prepisovacích pravidlách vstupného jazyka. Existujú na to dva dôvody:

1. Rozlíšiť metasymboly od jazykových symbolov v (5.12).
2. Použiť všeobecne dostupné znaky počítačového systému, najmä jednoduchého znaku = namiesto :: =.

Tab. 5.1 zobrazuje zodpovedajúce si symboly obvyklého jazyka BNF a nášho rozšíreného vstupného jazyka EBNF. Každé prepisovacie pravidlo je navyše ukončené explicitnou bodkou.

Metajazykové a jazykové symboly *Tabuľka 5.1*

BNF	Vstupný EBNF
:: =	=
|	,
{	[
}	]

Použitím tohto vstupného jazyka na opis syntaxe príkladu 5 (5.7) dostávame

$$\begin{aligned} &A = x, (B). \\ &B = AC. \\ &C = [+A]. \end{aligned} \tag{5.13}$$

Pri zjednodušovaní vytváraného prekladača budeme požadovať, aby terminálne symboly boli jednoduchými písmenami a každé prepisovacie pravidlo bolo napísané na osobitnom riadku. To nám umožní používať medzery vo vstupnom texte (čím sa tento text stane čitateľnejším). Medzery však prekladač ignoruje. V dôsledku toho sa musí príkaz read (*ch*) v zásade B7 nahradiť volaním procedúry, ktorá načíta najbližší relevantný znak. Táto činnosť prináleží lexikálnemu analyzátoru, ktorého úlohou je určiť ďalší symbol — v súlade s definovanými jazykovými pravidlami — zo vstupného reťazca znakov. Doteraz sme uvažovali o tom, že symboly sú identické so znakmi; to je však len špeciálny prípad a v praxi dosť zriedkavý.

Posledná zásada, ktorá sa bude týkať vstupu BNF, bude požadovať, aby neterminálne symboly boli reprezentované písmenami *A* až *H* a terminálne symboly písmenami *I* až *Z*. Túto zásadu však použijeme iba preto, že je výhodná, inak nemá žiadne hlbšie opodstatnenie. Jej uplatnením nepotrebujeme napr. vytvárať slovníky terminálnych a neterminálnych symbolov pred vlastným zoznamom produkcií.

Program 5.2 predstavuje syntaktický analyzátor jazyka, definovaného produkciami (5.12), ktorý sme získali na základe zásad B1 až B7 konštrukcie analyzátora a overením, či definície (5.12) spĺňajú obmedzenia 1 a 2. Poznamenávame, že lexikálny analyzátor predstavuje procedúra getsym.

PROGRAM 5.2. *Syntaktický analyzátor jazyka (5.12)*

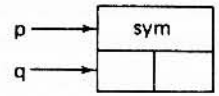
```
program Analyzátor (input, output);
label 99;
const prázdny = '*';
var sym: char;
procedure getsym;
begin
```

```
  repeat read(sym); write(sym) until sym ≠ ' '
end {getsym};
  procedure error;
  begin writeln;
      writeln ('NESPRÁVNY VSTUP'); goto 99
  end {error};
  procedure term;
      procedure factor;
      begin
        if sym in ['A'..'Z', prázdny] then getsym else
        if sym = '[' then
        begin getsym; term;
          if sym = ']' then getsym else error
        end else error
      end {factor};
    begin factor;
        while sym in ['A'..'Z', '[', prázdny] do factor
    end {term};
    procedure výraz;
    begin term;
        while sym = ',' do
        begin getsym; term
        end
    end {výraz};
      begin {hlavný program}
        while ¬ eof(input) do
        begin getsym;
          if sym in ['A'..'Z'] then getsym else error;
          if sym = '=' then getsym else error;
          výraz;
          if sym ≠ '.' then error;
          writeln; readln;
        end;
      99: end.
```

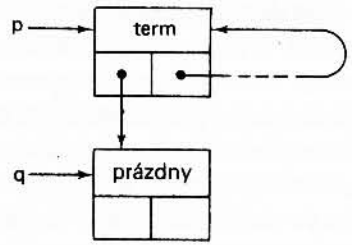
Tretí krok tvorby prekladača sa týka konštrukcie požadovanej štruktúry údajov, ktorá reprezentuje práve prečítané prepisovacie pravidlá BNF a umožňuje ich interpretáciu prostredníctvom procedúry (5.11). Žiaľ, tento krok sa nedá až tak formalizovať, ako to bolo v prípade druhého kroku, týkajúceho sa tvorby analyzátora EBNF. Pretože nám chýba formalizmus, uvedieme ešte raz (vo forme obrázku) štruktúry, ktoré sú potrebné na reprezentáciu každej jazykovej konštrukcie. Tieto štruktúry sa potom odovzdávajú vo forme výstupných parametrov zodpovedajúcim procedúram analyzátora (vylepšených na procedúry prekladača). Pochopiteľne, že návratom nie sú samotné štruktúry, ale iba smerníky p, q, r na tieto štruktúry.

Faktory:

1. ⟨symbol⟩

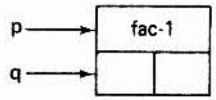


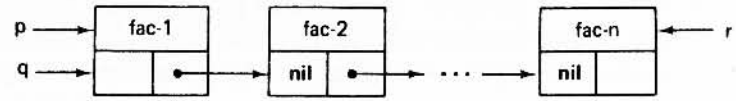
2. [⟨term⟩]



Termy:

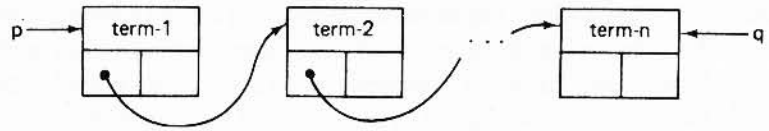
⟨faktor − 1⟩ ... ⟨faktor − n⟩





Výrazy:

⟨term − 1⟩⟨term − 2⟩ ... ⟨term − n⟩



Zrejme, že generovanie nových prvkov štruktúry údajov je úlohou procedúry faktor; úlohou zvyšných dvoch procedúr je pospájať ich do lineárneho zoznamu, v ktorom term používa na zreťazenie zložku *suc* a výraz zložku *alt*. Podrobnosti sú vidieť v programe 5.3.

Spôsob spracúvania neterminálnych symbolov vyžaduje ďalšie vysvetlenie. Neterminálny symbol sa môže vyskytnúť ako faktor predtým, než sa zjaví ako ľavá časť prepisovacieho pravidla. Na vyhľadanie symbolu *sym* v lineárnom zozname všetkých neterminálnych symbolov sa používa procedúra nájdi (*sym*, *h*). Ak sa nájde v zozname príslušný symbol, priradí sa smerník naň parametru *h*; v opačnom prípade sa symbol *sym* pridá do zoznamu. Procedúra nájdi využíva techniku zarážky, o ktorej sme písali vo štvrtej kapitole.

Program 5.3 sa skladá z troch častí, z ktorých každá zodpovedá príslušnej vstupnej sekcii. Prvá časť sa týka spracovania prepisovacích pravidiel a ich transformácie do príslušných štruktúr údajov. V druhej časti sa číta a identifikuje symbol definovaný ako začiatočný, generujúci vety jazyka. (Predchádza mu znak **$**, ktorým sú oddelené časti 1 a 2 vstupných údajov.) Tretia časť predstavuje vlastný program syntaktickej analýzy (5.11), ktorý riadený štruktúrou údajov, vygenerovanou v prvej časti, číta a analyzuje vstupné vety.

Je pozoruhodné, že program 5.3 vznikol iba pridaním ďalších príkazov do nezmeneného programu 5.2. Program 5.2, ktorého cieľom je výlučne rozpoznanie správne vytvorených viet, sa dá použiť aj v širšom kontexte, t. j. v programe, ktorý nielen rozpoznáva, ale aj spracúva a prekladá správne vytvorené vety. Takáto metóda konštrukcie jazykových procesorov, t. j. postupné zjemňovanie alebo (lepšie povedané) postupné rozširovanie (obohacovanie), sa veľmi odporúča. Umožňuje tvorcovi kompilátora jazyka venovať sa v každej vývojovej etape výlučne vybranému okruhu problémov spracovania jazyka (pričom sa abstrahuje od nepodstatných detailov). Tým sa súčasne podstatne zjednoduší proces overovania správnosti prekladacieho programu, alebo sa

aspoň udrží značný stupeň dôvery v úspešný priebeh vývoja prekladacieho programu. Tento príklad prekladača možno považovať za jednoduchý, pretože jeho vývoj pozostával iba z dvoch krokov. Pochopiteľne, zložitejšie jazyky a zložitejšie preklady vyžadujú podstatne väčší počet jednotlivých rozširovacích krokov. V článkoch 5.8 až 5.11 sa budeme zaoberať veľmi podobným vývojom prekladača, ktorý však vyžaduje tri kroky.

Z vývoja programu 5.3 vidíme, že syntaxou riadený preklad, alebo skôr štruktúrou riadený preklad, poskytuje oveľa väčší stupeň voľnosti a flexibility ako program pre špecifický syntaktický analyzátor. Táto dodatočná flexibilita, aj keď sa vo všeobecnosti nepožaduje, je základnou charakteristikou kompilátora rozšíriteľných jazykov. Rozšíriteľný jazyk sa dá rozšíriť o ďalšie syntaktické konštrukcie viac-menej podľa uváženia programátora. Podobne ako vstup pre program 5.3 bude vstup kompilátora rozšíriteľného jazyka obsahovať sekciu definujúcu jazykové rozšírenia použité v nasledujúcom programe. Náročnejšia schéma dokonca umožňuje zmeny v jazyku počas procesu kompilácie, a to na základe vkladania sekcií s novými jazykovými špecifikáciami do prekladaného programu.

Aj keď sa takéto myšlienky zdajú byť príťažlivé, snahy realizovať takéto kompilátory boli málo úspešné. Dôvodom je skutočnosť, že syntaktická analýza vety je len malou časťou celého prekladacieho procesu a dá sa najjednoduchšie formalizovať, a tým aj ľahko reprezentovať prostredníctvom systematizovanej tabuľkovej štruktúry. Náročnejšou časťou na formalizáciu je význam jazyka, t. j. vstup alebo výsledok prekladu. Tento problém doteraz nebol uspokojivo vyriešený, čím sa vysvetľuje to, prečo zvyknú byť tvorcovia kompilátorov nadšení rozšíriteľnými jazykmi skôr, ako dospejú k ich prvému dohotoveniu. Zvyšnú časť tejto kapitoly venujeme vývoju jednoduchého kompilátora pre jeden špecifický malý programovací jazyk.

PROGRAM 5.3. *Prekladač jazyka (5.12)*

```
program Prekladač (input, output);
label 99;
const prázdny = '*';
type smerník = ↑vrchol;
```

```
    čsmerník = ↑čelo;
    vrchol = record suc, alt: smerník;
                 case terminál: boolean of
                   true: (tsym: char);
                   false: (nsym: čsmerník)
               end;
    čelo = record sym: char;
                  vstup: smerník;
                  suc: čsmerník
           end;
var zoznam, zarážka, h: čsmerník;
  p: smerník;
  sym: char;
  ok: boolean;
 procedure getsym;
 begin
   repeat read(sym); write(sym) until sym ≠ ' '
 end {getsym};
 procedure nájdi (s: char; var h: čsmerník);
   {vyhľadá neterminálny symbol s v zozname; ak v ňom nie je,
    pridá ho doňho}
   var h1: čsmerník;
 begin h1 := zoznam; zarážka↑.sym := s;
   while h1↑.sym ≠ s do h1 := h1↑.suc;
   if h1 = zarážka then
     begin {pridaj symbol} new(zarážka);
       h1↑.suc := zarážka; h1↑.vstup := nil
     end;
   h := h1
 end {nájdi};
 procedure error;
 begin writeln;
   writeln ('NESPRÁVNA SYNTAX'); goto 99
 end {error};
 procedure term (var p, q, r: smerník);
   var a, b, c: smerník;
```



```
 procedure factor (var p, q: smerník);
     var a, b: smerník; h: čsmerník;
 begin if sym in ['A'..'H', prázdny] then
   begin {symbol} new(a);
     if sym in ['A'..'H'] then
     begin {neterminál} nájdi(sym, h);
       a↑.terminál := false; a↑.nsym := h
     end else
     begin {terminál}
       a↑.terminál := true; a↑.tsym := sym
     end;
     p := a; q := a; getsym
   end else
   if sym = '[' then
   begin getsym; term(p, a, b); b↑.suc := p;
     new(b); b↑.terminál := true; b↑.tsym := prázdny;
     a↑.alt := b; q := b;
     if sym ']' then getsym else error
   end else error
 end {factor};
begin factor(p, a); q := a;
 while sym in ['A'..'Z', '[', prázdny] do
 begin factor(a↑.suc, b); b↑.alt := nil; a := b
 end;
 r := a
end {term};
procedure výraz (var p, q: smerník);
 var a, b, c: smerník;
begin term(p, a, c); c↑.suc := nil;
 while sym = ',' do
 begin getsym;
   term(a↑.alt, b, c); c↑.suc := nil; a := b
 end;
 q := a
end {výraz};
procedure analýza (cieľ: čsmerník; var zhodný: boolean);
```

```
    var s: smerník;
begin s: = cieľ↑.vstup;
    repeat
      if s↑.terminál then
      begin if s↑.tsym = sym then
        begin zhodný: = true; getsym
        end
        else zhodný: = (s↑.tsym = prázdny)
      end
      else analýza (s↑.nsym, zhodný);
      if zhodný then s: = s↑.suc else s: = s↑.alt
    until s = nil
end {analýza};
begin {pravidlá}
    getsym; new(zarážka); zoznam: = zarážka;
    while sym ≠ '$' do
    begin nájdi(sym, h);
      getsym; if sym = '=' then getsym else error;
      výraz(h↑.vstup, p); p↑.alt: = nil;
      if sym ≠ '.' then error;
      writeln; readln; getsym
    end;
    h: = zoznam; ok: = true;
    {kontrola, či sú všetky symboly definované}
    while h ≠ zarážka do
    begin if h↑.vstup = nil then
      begin writeln('NEDEFINOVANÝ SYMBOL', h↑.sym);
        ok: = false
      end;
      h: = h↑.suc
    end;
    if ¬ ok then goto 99;
{cieľový symbol}
    getsym; nájdi(sym, h); readln; writeln;
{vety}
    while ¬ eof(input) do
      begin write(' '); getsym; analýza(h, ok);
        if ok ∧ (sym = '.') then writeln('SPRÁVNA')
                            else writeln('NESPRÁVNA');
        readln
      end;
99: end.
```


5.7 PROGRAMOVACÍ JAZYK PL/0

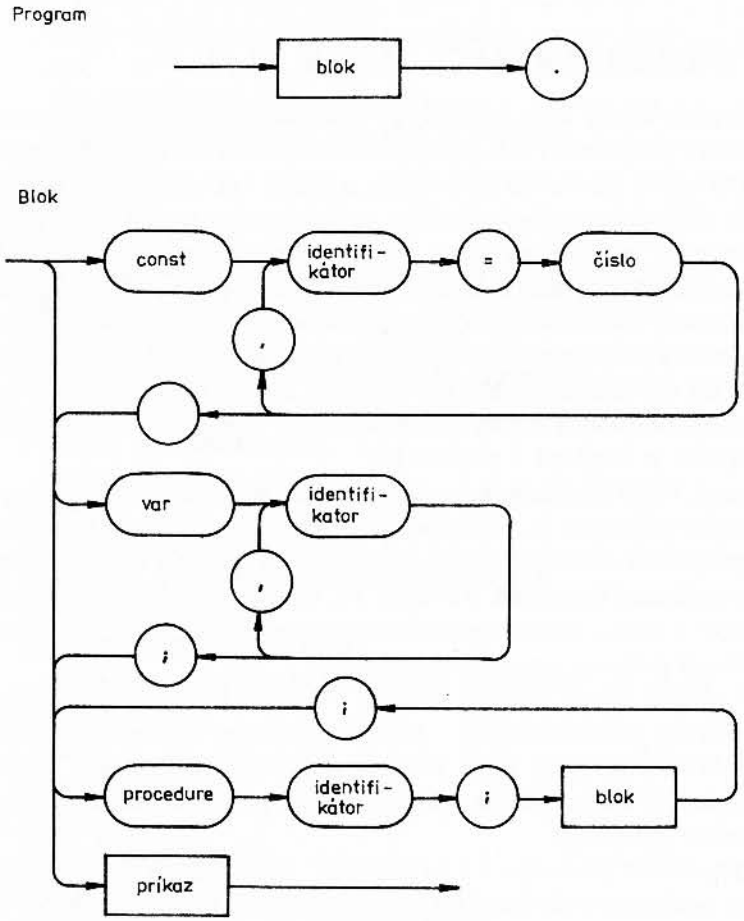
Zvyšné články tejto kapitoly sú venované vývoju kompilátora jazyka, ktorý nazývame PL/0. Nevyhnutnosť primerane malého kompilátora, vzhľadom na rozsah tejto knihy, a snaha vysvetliť všetky najzákladnejšie pojmy kompilácie jazykov vyššej úrovne predstavujú okrajové podmienky návrhu takéhoto jazyka. Nepochybne by sa dal vybrať úplne jednoduchý jazyk, ako aj veľmi zložitý jazyk; PL/0 predstavuje kompromis medzi dostatočnou jednoduchosťou, vzhľadom na zrozumiteľnosť vysvetlenia, a dostatočnou zložitosťou, vzhľadom na celkovú hodnotu vytváraného jazyka. Podstatne zložitejším jazykom je pascal, ktorého kompilátor bol vyvinutý pomocou tých istých techník a ktorého syntax je uvedená v prílohe B.

Jazyk PL/0 obsahuje kompletne všetky programové štruktúry. Základným príkazom je, pochopiteľne, priraďovací príkaz. Štruktúrovanie programu — sekvenčnosť, podmienený výpočet a cyklus — dosiahneme príkazmi **begin**/**end**, **if** a **while**. PL/0 podporuje koncepciu podprogramov, a preto umožňuje deklarovanie procedúr, resp. ich aktiváciu pomocou príkazu vyvolania procedúry.

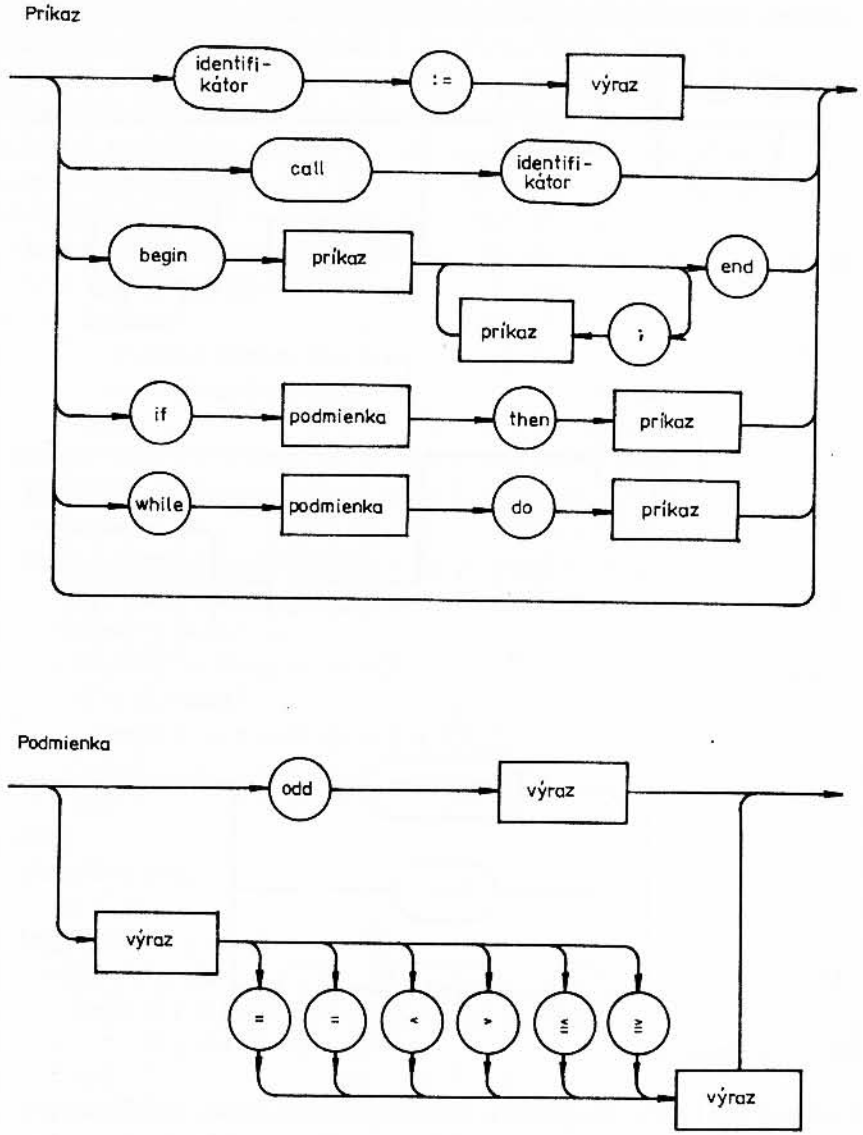
V oblasti typov údajov sa však PL/0 nekompromisne pridržiava požiadavky jednoduchosti — jediným možným typom údajov sú celé čísla. Pomocou tohto typu môžeme deklarovať celočíselné konštanty a premenné. Pochopiteľne, PL/0 dovoľuje používať bežné aritmetické a relačné operátory.

Prítomnosť procedúr, t. j. viac-menej „samostatných" úsekov programu, poskytuje príležitosť na zavedenie pojmu lokalita objektov (konštánt, premenných a procedúr). PL/0 vyžaduje, aby deklarácie všetkých objektov boli uvedené v záhlaví každej procedúry, čím sa tieto objekty stávajú lokálnymi v rámci tejto procedúry.

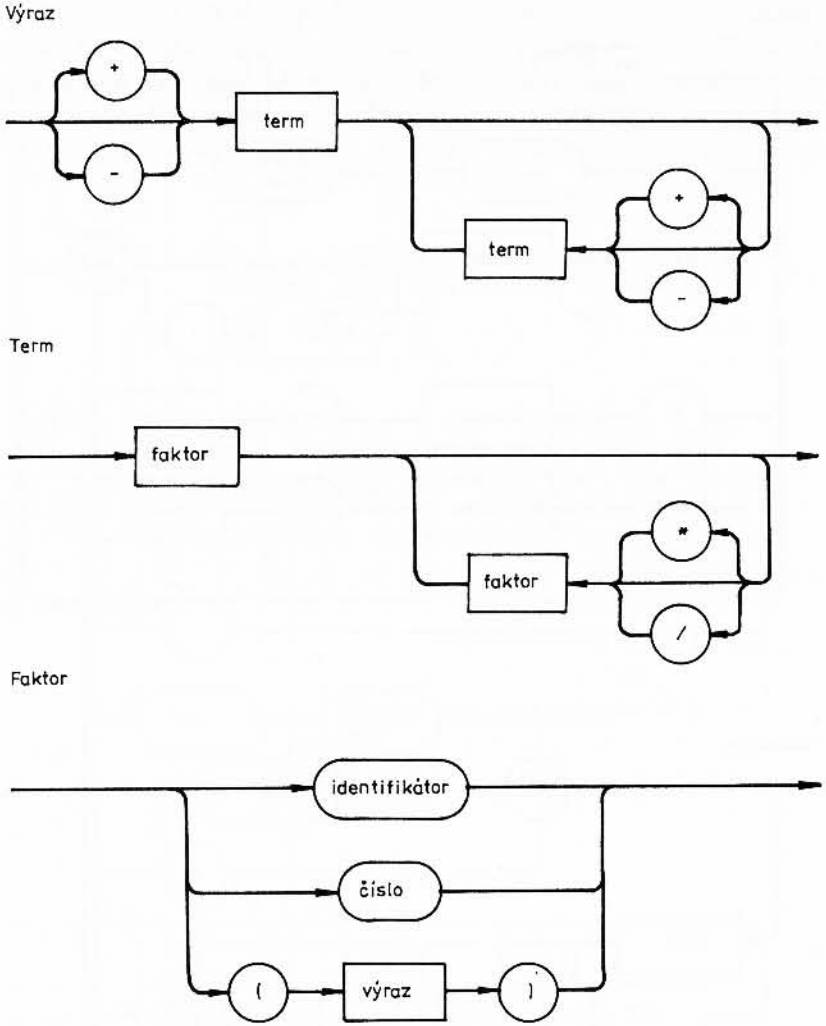
Tento stručný úvod a prehľad predstavujú nevyhnutné minimum znalostí potrebných na porozumenie syntaxe jazyka PL/0. Táto syntax je zobrazená na *obr. 5.4* formou siedmich diagramov. Transformáciu týchto diagramov do množiny ekvivalentných prepisovacích pravidiel BNF ponechávame na čitateľa. *Obr. 5.4* je presvedčivým príkladom



Obr. 5.4. Syntax jazyka PL/0



Obr. 5.4. Pokračovanie



Obr. 5.4. Pokračovanie

výraznej sily týchto diagramov, umožňujúcich stručnú, prehľadnú zrozumiteľnú formuláciu syntaxe celého programovacieho jazyka.

Účelom nasledujúceho programu je preukázať niektoré charakteristické vlastnosti minijazyka PL/0. Program obsahuje dobre známe algo-

ritmy na násobenie, delenie a nájdenie najväčšieho spoločného deliteľa (gcd) dvoch prirodzených čísel.

```
const m = 7; n = 85;
var x, y, z, q, r;
procedure násobenie;
  var a, b;
begin a: = x; b: = y; z: = 0;                                    (5.14)
    while b > 0 do
    begin
      if odd b then z: = z + a;
      a: = 2 * a; b: = b/2;
    end
end;
procedure delenie;
  var w;
begin r: = x; q: = 0; w: = y;
  while w ≦ r do w: = 2 * w;                                     (5.15)
  while w > y do
    begin q: = 2 * q; w: = w/2;
    if w ≦ r then
      begin r: = r - w; q: = q + 1
      end
    end
end;
procedure gcd;
  var f, g;
begin f: = x; g: = y;
  while f ≠ g do                                                 (5.16)
    begin if f < g then g: = g - f;
          if g < f then f: = f - g;
    end;
  z: = f
end;
begin
  x: = m; y: = n; call násobenie;
```

```
    x: = 25; y: = 3; call delenie;
    x: = 84; y: = 36; call gcd;
end.
```

5.8 SYNTAKTICKÝ ANALYZÁTOR JAZYKA PL/0

Vytvorenie syntaktického analyzátora bude prvý krok procesu tvorby kompilátora jazyka PL/0. Dá sa uskutočniť presne podľa zásad tvorby syntaktického analyzátora B1 až B7, ktoré boli opísané v článku 5.4. Túto metódu však možno použiť iba v tom prípade, ak sú zodpovedajúcou syntaxou splnené obmedzujúce tvrdenia 1 a 2. Musíme preto overiť, či zodpovedajúce syntaktické grafy vyhovujú uvedeným podmienkam.

Tvrdenie 1 určuje, že každá vetva vychádzajúca z určitého bodu vetvenia musí viesť k jednoznačnému začiatočnému symbolu. Toto

Začiatočné a nasledujúce symboly jazyka PL/0 *Tabuľka 5.2*

Neterminálny symbol S	Začiatočné symboly $L(S)$	Nasledujúce symboly $F(S)$
Blok	**const var** **procedure** identifikátor **if call begin while**	. ;
Príkaz	identifikátor **call** **begin if while**	. ; **end**
Podmienka	**odd** + − (identifikátor číslo	**then do**
Výraz	+ − (identifikátor číslo	. ;) *R* **end then do**
Term	identifikátor číslo (	. ;) *R* + − **end then do**
Faktor	identifikátor číslo (	. ;) *R* + − * / **end** **then do**

tvrdenie sa dá veľmi jednoducho overiť na príslušných syntaktických diagramoch znázornených na *obr. 5.4*. Tvrdenie 2 sa vzťahuje na všetky grafy, ktoré možno prejsť bez prečítania akéhokoľvek symbolu. Jediným takýmto grafom v syntaxi PL/0 je graf zobrazujúci príkazy jazyka. Tvrdenie 2 vyžaduje, aby všetky prvé symboly, ktoré môžu nasledovať za príkazom, boli disjunktné so začiatočným symbolom príkazov. Vzhľadom na to, že neskôr bude užitočné poznať množiny začiatočných a nasledujúcich symbolov pre všetky grafy, stanovíme si tieto množiny pre všetkých sedem neterminálnych symbolov (grafov) syntaxe jazyka PL/0 (okrem programu). *Tab. 5.2* predstavuje záruku disjunktnosti množín začiatočných a nasledujúcich symbolov príkazov. Tým je overená aplikácia zásad konštrukcie syntaktického analyzátora B1 až B7.

Pozorný čitateľ si iste všimol, že základnými symbolmi jazyka PL/0 už nie sú iba jednoduché znaky, ako to bolo v predchádzajúcich príkladoch, ale postupnosti znakov, napr. BEGIN alebo : =. Podobne ako v programe 5.3 využijeme lexikálny analyzátor, ktorý bude mať na starosti reprezentačné alebo lexikálne aspekty vstupného reťazca symbolov. V kompilátore PL/0 je lexikálny analyzátor reprezentovaný procedúrou getsym, ktorej hlavnou úlohou je vyprodukovať ďalší symbol. Lexikálny analyzátor slúži na tieto ciele:

1. Ignoruje oddeľovače (medzery).
2. Rozpoznáva kľúčové slová, akými sú napr. BEGIN, END atď.
3. Rozpoznáva ostatné slová (ako sú identifikátory). Momentálny identifikátor je priradený globálnej premennej *id*.
4. Rozpoznáva reťazce číslic ako čísla. Bežná hodnota čísla je priradená globálnej premennej *num*.
5. Rozpoznáva dvojice zvláštnych znakov, ako je napr. : =.

Počas analýzy vstupnej postupnosti znakov používa procedúra getsym lokálnu procedúru getch, ktorej úlohou je načítanie ďalšieho znaku. Okrem tejto hlavnej úlohy vykonáva procedúra getch aj tieto ďalšie činnosti:

1. Rozpoznáva a potláča informáciu o ukončení vstupného riadku.
2. Kopíruje vstup na výstupný súbor, čím vytvára protokol zdrojového programu.
3. Na začiatok každého riadku vypíše jeho poradové číslo.

Lexikálny analyzátor svojou činnosťou zabezpečuje predsnímanie

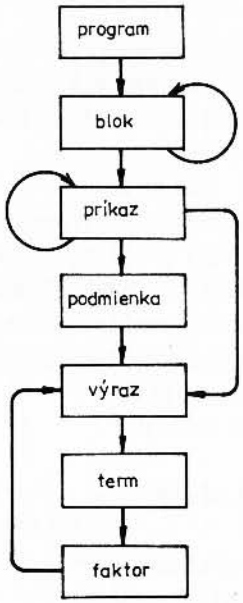
jedného symbolu dopredu. Pomocná procedúra getch navyše reprezentuje pozretie sa na jeden znak dopredu. Celkový počet pohľadov dopredu tohto kompilátora bude potom jeden symbol a jeden znak.

Podrobnosti týchto procedúr nájdeme v programe 5.4, ktorý predstavuje úplný syntaktický analyzátor jazyka PL/0. Tento analyzátor je už navyše rozšírený v tom zmysle, že všetky deklarované identifikátory konštánt, premenných a procedúr ukladá do tabuľky. Výskyt identifikátora v nejakom príkaze potom spôsobí prehľadanie tejto tabuľky za účelom zistenia, či príslušný identifikátor bol (alebo nebol) správne deklarovaný. Ak sa príslušná deklarácia nenachádza v tabuľke symbolov, došlo k syntaktickej chybe, pretože ide o formálnu chybu pri zostavovaní programu (z dôvodu použitia nedefinovaného symbolu).

Skutočnosť, že sa takáto chyba dá odhaliť iba na základe informácií z tabuľky, je dôsledkom kontextovej závislosti jazyka, vyjadrenej pravidlom, že všetky identifikátory musia byť deklarované v príslušnom kontexte. Z tohto hľadiska sú prakticky všetky programovacie jazyky kontextovo závislé; napriek tomu je bezkontextová syntax pre tieto jazyky najvýhodnejším modelom a je veľkou pomôckou pri systematickej tvorbe ich analyzátorov. Jednoduchou úpravou sa potom dokážeme vysporiadať aj s tými niekoľkými kontextovo závislými prvkami jazyka, ako to dokazuje zavedenie tabuľky identifikátorov v uvedenom syntaktickom analyzátore.

Predtým ako sa pustíme do tvorby jednotlivých procedúr syntaktického analyzátora, bude užitočné, ak uvedieme, ako príslušné syntaktické grafy navzájom závisia. Na tento účel zostrojíme diagram závislosti, ktorý zobrazuje závislosti medzi jednotlivými grafmi, t. j. pre každý graf G udáva všetky grafy $G_1, \ldots, G_n$, pomocou ktorých je graf G definovaný. Obdobne zobrazuje tie procedúry, ktoré môžu byť vyvolávané z iných procedúr. Diagram závislosti jazyka PL/0 znázorňuje *obr. 5.5.*

Cykly na *obr. 5.5* znamenajú výskyty rekurzie. Preto je dôležité, aby jazyk, v ktorom má byť implementovaný kompilátor PL/0, umožňoval použitie rekurzie. Diagram závislosti navyše predstavuje užitočnú pomôcku pri návrhu hierarchickej organizácie programu syntaktickej analýzy. Napríklad všetky procedúry analyzátora môžu byť obsiahnuté (deklarované ako lokálne) v procedúre, ktorá analyzuje konštrukciu ⟨program⟩ (a je preto hlavnou programovou časťou syntaktického



Obr. 5.5. Diagram závislosti pre jazyk PL/0

analyzátora). Podobne všetky procedúry aktivované pri analýze bloku môžu byť definované lokálne v procedúre, ktorej cieľom analýzy je ⟨blok⟩. Pochopiteľne, všetky takéto procedúry volajú lexikálny analyzátor reprezentovaný procedúrou getsym, ktorá zasa vyvoláva procedúru getch.

PROGRAM 5.4. *Syntaktický analyzátor jazyka PL/0*

```
program PL0 (input, output);
{kompilátor jazyka PL/0 — syntaktická analýza}
label 99;
const
    norw = 11;     {počet kľúčových slov}
    txmax = 100;   {veľkosť tabuľky identifikátorov}
    nmax = 14;     {maximálny počet číslic v čísle}
    al = 10;       {dĺžka identifikátorov}
```

```
type symbol =
  (nul, identifikátor, číslo, plus, mínus, krát, deleno, oddsym, eql, neq,
   lss, leq, gtr, geq, ľzátvorka, pzátvorka, čiarka, bodkočiarka, bodka,
   nadobudne, beginsym, endsym, ifsym, thensym, whilesym, dosym,
   callsym, constsym, varsym, procsym);
   alfa = packed array [1..al] of char;
   objekt = (konštanta, premenná, procedúra);
var ch: char;          {posledný prečítaný znak}
    sym: symbol;       {posledný prečítaný symbol}
    id: alfa;          {posledný prečítaný identifikátor}
    num: integer;      {posledné prečítané číslo}
    cc: integer;       {počet znakov}
    ll: integer;       {dĺžka riadku}
    kk: integer;
    riadok: array [1..81] of char;
    a: alfa;
    slovo: array [1..norw] of alfa;
    wsym: array [1..norw] of symbol;
    ssym: array [char] of symbol;
    tab: array [0..txmax] of
           record meno: alfa;
                  druh: objekt
           end;
  procedure error (n: integer);
  begin writeln(' ': cc, '↑', n: 2); goto 99
  end {error};
  procedure getsym;
    var i, j, k: integer;
    procedure getch;
    begin if cc = ll then
      begin if eof(input) then
          begin write('NEÚPLNÝ PROGRAM'); goto 99
          end;
        ll: = 0; cc: = 0; write(' ');
        while ¬ eoln(input) do
          begin ll: = ll + 1; read(ch); write(ch); riadok[ll]: = ch
          end;
        writeln; ll: = ll + 1; read(riadok[ll])
      end;
      cc: = cc + 1; ch: = riadok[cc]
    end {getch};
begin {getsym}
while ch = ' ' do getch;
  if ch in ['A'..'Z'] then
  begin {identifikátor alebo kľúčové slovo}
    k: = 0;
    repeat if k < al then
      begin k: = k + 1; a[k]: = ch
      end;
      getch
    until ¬ (ch in ['A'..'Z', '0'..'9']);
    if k ≧ kk then kk: = k else
      repeat a[kk]: = ' '; kk: = kk - 1
      until kk = k;
    id: = a; i: = 1; j: = norw;
    repeat k: = (i + j) div 2;
      if id ≦ slovo[k] then j: = k - 1;
      if id ≧ slovo[k] then i: = k + 1
    until i > j;
    if i - 1 > j then sym: = wsym[k] else sym: = identifikátor
  end else
  if ch in ['0'..'9'] then
  begin {číslo} k: = 0; num: = 0; sym: = číslo;
    repeat num: = 10 * num + (ord(ch) + ord('0'));
      k: = k + 1; getch
    until ¬ (ch in ['0'..'9']);
    if k > nmax then error (30)
  end else
  if ch = ':' then
  begin getch;
    if ch = '=' then
    begin sym: = nadobudne; getch
```

```
      end else sym := nul;
    end else
    begin sym := ssym[ch]; getch
    end
end {getsym};
procedure blok(tx: integer);
  procedure vstup(k: objekt);
  begin {zaradenie objektu do tabuľky}
    tx := tx + 1;
    with tab[tx] do
    begin meno := id; druh := k;
    end
  end {vstup};
  function pozícia (id: alfa): integer;
    var i: integer;
  begin {vyhľadanie identifikátora id v tabuľke}
    tab[0].meno := id; i := tx;
    while tab[i].meno ≠ id do i := i - 1;
    pozícia := i
  end {pozícia};
procedure constdeklarácia;
begin if sym = identifikátor then
  begin getsym;
    if sym = eql then
    begin getsym;
      if sym = číslo then
        begin vstup(konštanta); getsym
        end
      else error (2)
    end else error (3)
  end else error (4)
end {constdeklarácia};
procedure vardeklarácia;
begin if sym = identifikátor then
  begin vstup(premenná); getsym
  end else error (4)
```



```
end {vardeklarácia};
procedure príkaz;
  var i: integer;
  procedure výraz;
    procedure term;
      procedure factor;
        var i: integer;
      begin
      if sym = identifikátor then
      begin i := pozícia(id);
        if i = 0 then error (11) else
        if tab[i].druh = procedúra then error (21);
        getsym
      end else
      if sym = číslo then
      begin getsym
      end else
      if sym = ľzátvorka then
      begin getsym; výraz;
        if sym = pzátvorka then getsym else error (22)
      end
      else error (23)
      end {factor};
  begin {term} factor;
    while sym in [krát, deleno] do
      begin getsym; factor
      end
  end {term};
begin {výraz}
  if sym in [plus, mínus] then
  begin getsym; term
  end
end {výraz};
procedure podmienka;
begin
  if sym = oddsym then
```

```
    begin getsym; výraz
    end else
    begin výraz;
      if ¬ (sym in [eql, neq, lss, leq, gtr, geq]) then
        error (20) else
      begin getsym; výraz
      end
    end
  end {podmienka};
  begin {príkaz}
  if sym = identifikátor then
  begin i: = pozícia (id);
    if i = 0 then error (11) else
    if tab[i].druh ≠ premenná then error (12);
    getsym; if sym = nadobudne then getsym else error (13);
    výraz
  end else
  if sym = callsym then
  begin getsym;
    if sym ≠ identifikátor then error (14) else
    begin i: = pozícia (id);
      if i = 0 then error (11) else
      if tab[i].druh ≠ procedúra then error (15);
      getsym
    end
    end else
    if sym = ifsym then
    begin getsym; podmienka;
      if sym = thensym then getsym else error (16);
      príkaz;
    end else
    if sym = beginsym then
    begin getsym; príkaz;
      while sym = bodkočiarka do
        begin getsym; príkaz
        end;
```



```
      if sym = endsym then getsym else error (17)
    end else
    if sym = whilesym then
    begin getsym; podmienka;
      if sym = dosym then getsym else error (18);
      príkaz
    end
  end {príkaz};
begin {blok}
  if sym = constsym then
  begin getsym; constdeklarácia;
    while sym = čiarka do
      begin getsym; constdeklarácia
      end;
    if sym = bodkočiarka then getsym else error (5)
  end;
  if sym = varsym then
  begin getsym; vardeklarácia;
    while sym = čiarka do
      begin getsym; vardeklarácia
      end;
    if sym = bodkočiarka then getsym else error (5)
  end;
  while sym = procsym do
  begin getsym;
    if sym = identifikátor then
      begin vstup (procedúra); getsym
      end
    else error (4);
    if sym = bodkočiarka then getsym else error (5);
    blok (tx);
    if sym = bodkočiarka then getsym error (5);
  end;
  príkaz
end {blok};
begin {hlavný program}
```

```
  for ch := 'A' to ';' do ssym[ch] := nul;
  slovo  [1]: = 'BEGIN';      slovo  [2]: = 'CALL';
  slovo  [3]: = 'CONST';      slovo  [4]: = 'DO';
  slovo  [5]: = 'END';        slovo  [6]: = 'IF';
  slovo  [7]: = 'ODD';        slovo  [8]: = 'PROCEDURE';
  slovo  [9]: = 'THEN';       slovo [10]: = 'VAR';
  slovo [11]: = 'WHILE';
  wsym   [1]: = beginsym;     wsym   [2]: = callsym;
  wsym   [3]: = constsym;     wsym   [4]: = dosym;
  wsym   [5]: = endsym;       wsym   [6]: = ifsym;
  wsym   [7]: = oddsym;       wsym   [8]: = procsym;
  wsym   [9]: = thensym;      wsym  [10]: = varsym;
  wsym  [11]: = whilesym;
  ssym ['+']: = plus;         ssym ['-']: = mínus;
  ssym ['*']: = krát;         ssym ['/']: = deleno;
  ssym ['(']: = ľzátvorka;    ssym [')']: = pzátvorka;
  ssym ['=']: = eql;          ssym [',']: = čiarka;
  ssym ['.']: = bodka;        ssym ['≠']: = neq;
  ssym ['<']: = lss;          ssym ['>']: = gtr;
  ssym ['≦']: = leq;          ssym ['≧']: = geq;
  ssym [';']: = bodkočiarka;
  page(output);
  cc: = 0; ll: = 0; ch: = ' '; kk: = al; getsym;
  blok(0);
  if sym ≠ bodka then error (9);
99: writeln
end.
```

5.9 ZOTAVENIE SA ZO SYNTAKTICKÝCH CHÝB

Zistiť, či daný vstupný reťazec symbolov patrí do jazyka alebo nie, to bola doteraz jediná úloha syntaktického analyzátora. Vedľajšou úlohou syntaktickej analýzy bolo určiť štruktúru vety. Keď sa však vyskytla nesprávna syntaktická konštrukcia, ktorú kompilátor dokázal

odhaliť, bol cieľ syntaktickej analýzy prakticky splnený a program mohol byť ukončený. Takéto správanie kompilátora by však v praxi ťažko obstálo. Prakticky použiteľný kompilátor musí vyprodukovať primeranú diagnostiku chyby a pokračovať v analytickom procese za účelom objavenia ďalších chýb. Pokračovať v analýze je možné buď na základe určitého predpokladu o povahe chyby a úmysle autora nesprávneho programu, alebo preskočením určitej časti vstupného reťazca. Niekedy sa pokračovanie analytického procesu uskutočňuje na základe obidvoch uvedených možností. Voľba správneho predpokladu je však dosť zložitá a doteraz bezúspešne formalizovaná, pretože formalizácie syntaxe a syntaktickej analýzy neposkytujú možnosť vziať do úvahy viaceré faktory, ktoré silne ovplyvňujú ľudskú myseľ. Bežnou chybou býva napr. zanedbanie interpunkčných symbolov, akým je bodkočiarka (a to nielen pri programovaní). Ale oveľa zriedkavejšie sa stane, že niekto zabudne napísať operátor + v aritmetickom výraze. Bodkočiarka aj symbol + sú pre syntaktický analyzátor terminálnymi symbolmi bez ďalšieho rozlíšenia. Pre programátora nemá bodkočiarka prakticky žiadny význam a na konci riadku sa mu javí dokonca ako nepotrebná, pričom pre aritmetický operátor je nenahraditeľná. Existuje, pochopiteľne, oveľa viac takýchto úvah, ktoré treba uvážiť pri návrhu primeraného systému zotavovania sa zo syntaktických chýb. Všetky závisia od konkrétneho jazyka, preto ich nemožno zovšeobecniť pre všetky bezkontextové jazyky.

Predsa však existujú niektoré pravidlá a pokyny, ktoré by sa mali požadovať a dodržiavať, pretože majú platnosť aj mimo rámca jednoduchých jazykov, akým je aj náš jazyk PL/0. Ich charakteristickou črtou je, že sa týkajú jednak začiatočnej koncepcie jazyka, ako aj návrhu mechanizmu syntaktického analyzátora na zotavenie sa z chýb. Predovšetkým je jasné, že realizácia výkonného a najmä citlivého mechanizmu zotavenia sa z chýb je možná iba v prípade jednoduchej štruktúry jazyka. Odporúča sa, aby jazyk, ktorého kompilátor obsahuje mechanizmus zotavenia sa z chýb, pracujúci na základe princípu ignorovania určitej časti vstupnej postupnosti pri výskyte syntaktickej chyby, obsahoval také kľúčové slová, ktoré pravdepodobne nebudú nesprávne použité, a tým poslúžia obnoveniu chodu syntaktického analyzátora. Jazyk PL/0 sa riadi týmto pravidlom: každý štruktúrova-

ný príkaz začína jednoznačným kľúčovým slovom, akým je **begin**, **if** alebo **while**, čo platí takisto pre deklarácie, ktoré začínajú slovami **var**, **const** alebo **procedure**. Toto pravidlo budeme nazývať preto *pravidlom kľúčového slova.*

Druhé pravidlo sa už priamo týka konštrukcie syntaktického analyzátora. Syntaktická analýza metódou zhora nadol je charakteristická tým, že ciele sa rozdeľujú na podciele a analyzátory cieľov vyvolávajú analyzátory podcieľov za účelom splnenia týchto podcieľov. Druhé pravidlo špecifikuje, čo má urobiť syntaktický analyzátor v prípade zistenia syntaktickej chyby. Rozhodne by nemal iba ohlásiť zistenú chybu svojmu nadriadenému analyzátoru a prestať v procese analýzy. Požaduje sa, aby analyzátor pokračoval vo svojej analýze textu až do okamihu, keď už znova môže nasledovať prijateľná syntaktická analýza. Tejto stratégii sa zvykne hovoriť *pravidlo úniku z chybového stavu.* Jeho praktickým programátorským dôsledkom je skutočnosť, že ukončenie činnosti analyzátora môže nastať iba v jeho regulárnom mieste ukončenia. Možná presná interpretácia uvedeného pravidla pozostáva z preskočenia vstupného textu pri zistení chybnej syntaktickej konštrukcie až po prvý taký symbol, ktorý môže korektne nasledovať za rozanalyzovanou vetnou konštrukciou. To znamená, že každý analyzátor pri svojej aktivácii pozná množinu svojich nasledujúcich symbolov.

V prvom zjemňovacom (alebo obohacovacom) kroku zabezpečíme, aby každá procedúra analyzátora obsahovala explicitný parameter fsys, ktorého hodnota bude určovať množinu možných nasledujúcich symbolov. Na koniec každej procedúry zaradíme explicitný test, ktorým sa overí, či ďalší symbol vstupného textu patrí do množiny nasledujúcich symbolov (ak už splnenie tejto podmienky nevyplýva priamo z logiky programu).

Boli by sme však veľmi krátkozrakí, keby sme sa za každých okolností snažili preskakovať vstupný text až po ďalší výskyt jedného z nasledujúcich symbolov. Veď programátor môže omylom vynechať presne jeden symbol (povedzme bodkočiarku); v takomto prípade by preskočenie textu až po ďalší nasledujúci symbol mohlo znamenať katastrofu. Preto rozšírime množiny symbolov, určujúcich ukončenie preskočenia vstupného textu, o kľúčové slová jazyka, ktoré špecifikujú začiatok syntaktickej konštrukcie tak, aby sa nedal symbol prehliadnuť. Symboly, ktoré sa formou parametrov posielajú analyzujúcim procedúram, budeme nazývať stop-symboly (namiesto pojmu nasledujúce symboly). Množina stop-symbolov je inicializovaná rôznymi kľúčovými symbolmi a pri prechode hierarchiou podcieľov analýzy postupne dopĺňaná prípustnými nasledujúcimi symbolmi. Vzhľadom na flexibilitu zavedieme všeobecnú procedúru test, ktorá overí, či sa symbol nachádza (alebo nenachádza) v množine stop-symbolov. Táto procedúra (5.17) má tri parametre:

1. Množinu $s1$ prípustných nasledujúcich symbolov; ak sa momentálny symbol nenachádza v tejto množine, dôjde k výskytu chyby.
2. Množinu $s2$ prídavných stop-symbolov, ktorých prítomnosť je rozhodne chybou, ale ktoré neslobodno v žiadnom prípade ignorovať a preskočiť.
3. Číslo n určujúce príslušnú chybovú diagnostiku.

```
procedure test (s1, s2: symset; n: integer);
begin if ¬ (sym in s1) then
    begin error(n); s1 := s1 + s2;
        while ¬ (sym in s1) do getsym
    end
end
```
(5.17)

Procedúru 5.17 možno s výhodou použiť i na vstupe do analyzujúcich procedúr, a to na zistenie, či momentálny symbol patrí do množiny začiatočných symbolov. To sa odporúča vo všetkých prípadoch, v ktorých sa analyzujúca procedúra X vyvoláva nepodmienene, ako napr. v príkaze

if $sym = a_1$ **then** S_1 **else**
⋮
if $sym = a_n$ **then** S_n **else** X

čo je výsledkom prekladu prepisovacieho pravidla

$$A ::= a_1 S_1 | \ldots | a_n S_n | X \tag{5.18}$$

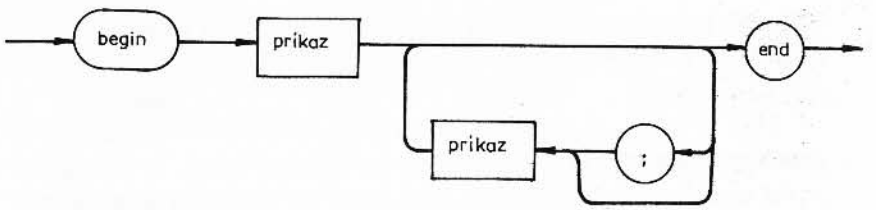
V týchto prípadoch musí byť parametrom $s1$ množina začiatočných symbolov procedúry X, pričom sa za parameter $s2$ vyberie množina nasledujúcich symbolov A (*tab. 5.2*). Podrobnosti tejto procedúry mož-

no nájsť v programe 5.5, ktorý je rozšírenou verziou programu 5.4. Pre čitateľovo pohodlie uvádzame opäť celý program syntaktického analyzátora okrem inicializácie globálnych premenných a procedúry getsym, ktoré ostali nezmenené.

Doterajšia schéma zotavovania sa zo syntaktických chýb je charakteristická tým, že proces analýzy sa obnovuje ignorovaním jedného alebo viacerých symbolov vstupného textu. To je však nevyhovujúca stratégia vo všetkých tých prípadoch, v ktorých bola chyba spôsobená vynechaním nejakého symbolu. Skúsenosť potvrdzuje, že takéto chyby sa prakticky výlučne vzťahujú na symboly, ktoré majú iba syntaktickú funkciu a nereprezentujú nijakú činnosť. Príkladom je bodkočiarka v jazyku PL/0. Skutočnosť, že sa množiny nasledujúcich symbolov rozšíria o určité kľúčové slová, spôsobuje, že syntaktický analyzátor predčasne ukončuje preskakovanie vstupných symbolov a vyzerá to tak, ako keby bol chýbajúci symbol pridaný do uvedenej množiny. Tento fakt možno vidieť v časti programu (5.19), ktorá analyzuje zložené príkazy. Chýbajúce bodkočiarky sa skutočne pridávajú pred kľúčové slová. Množina, nazývaná statbegsys predstavuje množinu začiatočných symbolov konštrukcie „príkaz".

```
if sym = beginsym then
begin getsym;
      príkaz ([bodkočiarka, endsym] + fsys);
      while sym in [bodkočiarka] + statbegsys do
        begin                                              (5.19)
          if sym = bodkočiarka then getsym else error;
          príkaz ([bodkočiarka, endsym] + fsys)
        end;
      if sym = endsym then getsym else error
end
```

Stupeň úspešnosti, s akým tento program diagnostikuje syntaktické chyby a zotavuje sa z nich, možno odhadnúť pomocou programu (5.20) napísaného v jazyku PL/0. Protokol zdrojového textu programu (5.20) predstavuje výstup programu 5.5. *Tab. 5.3* obsahuje množinu možných diagnostických správ, zodpovedajúcich číslam chýb v programe 5.5.



Obr. 5.6. Pozmenená syntax zloženého príkazu

Zoznam chybových správ kompilátora jazyka PL/0 *Tabuľka 5.3*

1. Namiesto symbolu : = treba použiť symbol =.
2. Za symbolom = musí nasledovať číslo.
3. Za identifikátorom musí nasledovať symbol =.
4. Za **const**, **var**, **procedure** musí nasledovať identifikátor.
5. Chýbajúca bodkočiarka alebo čiarka.
6. Za deklaráciou procedúry nasleduje nesprávny symbol.
7. Očakáva sa príkaz.
8. Nesprávny symbol za príkazovou časťou bloku.
9. Očakáva sa symbol bodka.
10. Medzi dvoma príkazmi chýba bodkočiarka.
11. Nedeklarovaný identifikátor.
12. Neprípustné priradenie konštante alebo procedúre.
13. Očakáva sa operátor priradenia : =.
14. Za príkazom **call** musí nasledovať identifikátor.
15. Nesprávne použitie konštanty alebo premennej.
16. Očakáva sa symbol **then**.
17. Očakáva sa symbol bodkočiarka alebo **end**.
18. Očakáva sa symbol **do**.
19. Za príkazom nasleduje nesprávny symbol.
20. Očakáva sa relačný operátor.
21. Výraz nesmie obsahovať identifikátor procedúry.
22. Chýbajúca pravá zátvorka.
23. Za faktorom nasleduje nesprávny symbol.
24. Nesprávny začiatok výrazu.
30. Číslo je väčšie ako maximálne zobraziteľné číslo v počítači.

Nasledujúci program (5.20) vznikol úmyselným „vyrobením" niektorých syntaktických chýb v programoch (5.14) až (5.16).

Neúplný program:

```
const m = 7, n = 85
var x, y, z, q, r;
    ↑5
procedure násobenie;
  var a, b
begin a: = u; b: = y; z: = 0
↑5
        ↑11
  while b > 0 do
    ↑10
  begin
  if odd b do z: = z + a;
                    ↑16
                    ↑19
  a: = 2a; b: = b/2;
       ↑23
  end
end;
procedure delenie
  var w;
  ↑5
  const dva = 2, tri: = 3;
    ↑7
                    ↑1
begin r = x; q: = 0; w: = y;
          ↑13                                    (5.20)
          ↑24
  while w ≦ r do w: = dva * w;
  while w > y
    begin q: = (2 * q; w: = w/2);
      ↑18
                ↑22
                          ↑23
    if w ≦ r then
```



```
        begin r: = r − w; q: = q + 1
                    ↑23
        end
      end
end;
procedure gcd;
  var f, g;
begin f: = x; g: = y
  while f ≠ g do
      ↑17
    begin if f < g then g: = g − f;
          if g < f then f: = f − g;
  z: = f
end;
begin
  x: = m; y: = n; call násobenie;
  x: = 25; y: = 3; call delenie;
  x: = 84; y: = 36; call gcd;
  call x; x: = gcd; gcd = x
    ↑15
          ↑21
                ↑12
                   ↑13
                   ↑24
end.
    ↑17
    ↑5
    ↑7
```

Treba si uvedomiť, že žiadna schéma, ktorá primerane efektívne prekladá správne vytvorené vety, nebude schopná takisto efektívne spracúvať všetky možné nesprávne syntaktické konštrukcie. A prečo by aj mala! Každá schéma implementovaná s primeraným úsilím zlyhá, t. j. bude neprimerane spracúvať niektoré nesprávne konštrukcie. Charakteristickou vlastnosťou dobrého kompilátora je, že:

1. Žiadna vstupná postupnosť nespôsobí jeho haváriu.

2. Odhalí a označí všetky konštrukcie, ktoré sú podľa definície jazyka neprípustné.

3. Chyby, ktoré sa objavujú pomerne často a sú čisto programátorské (zapríčinené prehliadnutím alebo nedorozumením), správne diagnostikuje a nepripustí, aby spôsobili vznik ďalších, falošných chybových správ.

Uvedená schéma pracuje uspokojivo, aj keď by sa dalo ešte čo-to vylepšiť. Cenná je rozhodne tá skutočnosť, že bola vytvorená systematickým spôsobom podľa malého počtu základných pravidiel. Tieto pravidlá boli obohatené iba o niektoré vybrané parametre, získané na základe heuristiky a zo skúseností z praktického používania programovacieho jazyka.

PROGRAM 5.5. *Syntaktický analyzátor jazyka PL/0 so systémom zotavenia sa z chýb*

```
program PL/0 (input, output);
{kompilátor jazyka PL/0, ktorého syntaktický analyzátor obsahuje
 systém zotavenia sa zo syntaktických chýb}
label 99;
const norw = 11;      {počet kľúčových slov}
      txmax = 100;    {veľkosť tabuľky identifikátorov}
      nmax = 14;      {maximálny počet číslic v čísle}
      al = 10;        {dĺžka identifikátorov}
type symbol =
 (nul, identifikátor, číslo, plus, mínus, krát, deleno, oddsym, eql, neq,
  lss, leq, gtr, geq, ľzátvorka, pzátvorka, čiarka, bodkočiarka, bodka,
  nadobudne, beginsym, endsym, ifsym, thensym, whilesym, dosym,
  callsym, constsym, varsym, procsym);
  alfa = packed array [1..al] of char;
  objekt = (konštanta, premenná, procedúra);
  symset = set of symbol;
var ch: char;         {posledný prečítaný znak}
    sym: symbol;      {posledný prečítaný symbol}
    id: alfa;         {posledný prečítaný identifikátor}
    num: integer;     {posledné prečítané číslo}
```



```
    cc: integer;      {počet znakov}
    ll: integer;      {dĺžka riadku}
    kk: integer;
    riadok: array [1..81] of char;
    a: alfa;
    slovo: array [1..norw] of alfa;
    declbegsys, statbegsys, facbegsys: symset;
    wsym: array [1..norw] of symbol;
    ssym: array [char] of symbol;
    tab: array [0..txmax] of
           record meno: alfa;
                  druh: objekt
           end;
procedure error (n: integer);
begin writeln (' ': cc, '↑', n: 2);
end {error};
procedure test (s1, s2: symset; n: integer);
begin if ¬ (sym in s1) then
    begin error (n); s1 := s1 + s2;
        while ¬ (sym in s1) do getsym
    end
end {test};
procedure blok (tx: integer);
  procedure vstup (k: objekt);
  begin {zaradenie objektu do tabuľky}
    tx := tx + 1;
    with tab[tx] do
    begin meno := id; druh := k;
    end
  end {vstup};
  function pozícia (id: alfa): integer;
    var i: integer;
  begin {vyhľadanie identifikátora id v tabuľke}
    tab[0].meno := id; i := tx;
    while tab[i].meno ≠ id do i := i - 1;
    pozícia := i
```

```
  end {pozícia};
  procedure constdeklarácia;
  begin if sym = identifikátor then
    begin getsym;
      if sym in [eql, nadobudne] then
      begin if sym = nadobudne then error (1);
        getsym;
        if sym = číslo then
        begin vstup (konštanta); getsym
        end
        else error (2)
      end else error (3)
    end else error (4)
  end {constdeklarácia};
  procedure vardeklarácia;
  begin if sym = identifikátor then
      begin vstup (premenná); getsym
      end else error (4)
end {vardeklarácia};
procedure príkaz (fsys: symset);
var i: integer;
procedure výraz (fsys: symset);
  procedure term (fsys: symset);
  procedure factor (fsys: symset);
    var i: integer;
  begin test (facbegsys, fsys, 24);
      while sym in facbegsys do
      begin
        if sym = identifikátor then
        begin i: = pozícia (id);
          if i = 0 then error (11) else
          if tab[i].druh = procedúra then error (21);
          getsym
        end else
        if sym = číslo then
        begin getsym;
```



```
        end else
        if sym = ľzátvorka then
        begin getsym; výraz ([pzátvorka] + fsys);
          if sym = pzátvorka then getsym else error (22)
        end;
        test (fsys, [ľzátvorka], 23)
      end
  end {factor};
  begin {term} factor (fsys + [krát, deleno]);
    while sym in [krát, deleno] do
      begin getsym; factor (fsys + [krát, deleno])
      end
  end {term};
begin {výraz}
  if sym in [plus, mínus] then
    begin getsym; term (fsys + [plus, mínus])
    end else term (fsys + [plus, mínus]);
  while sym in [plus, mínus] do
    begin getsym; term (fsys + [plus, mínus])
    end
end {výraz};
procedure podmienka (fsys: symset);
begin
  if sym = oddsym then
  begin getsym; výraz (fsys);
  end else
  begin výraz ([eql, neq, lss, gtr, leq, geq] + fsys);
    if ¬ (sym in [eql, neq, lss, leq, gtr, geq]) then
      error (20) else
    begin getsym; výraz (fsys)
    end
  end
end {podmienka};
begin {príkaz}
  if sym = identifikátor then
  begin i: = pozícia (id);
```

```
    if i = 0 then error (11) else
    if tab[i].druh ≠ premenná then error (12);
    getsym; if sym = nadobudne then getsym else error (13);
    výraz (fsys);
  end else
  if sym = callsym then
  begin getsym;
    if sym ≠ identifikátor then error (14) else
      begin i: = pozícia (id);
        if i = 0 then error (11) else
        if tab[i].druh ≠ procedúra then error (15);
        getsym
      end
  end else
  if sym = ifsym then
  begin getsym; podmienka ([thensym, dosym] + fsys);
    if sym = thensym then getsym else error (16);
    príkaz (fsys)
  end else
  if sym = beginsym then
  begin getsym; príkaz ([bodkočiarka, endsym] + fsys);
    while sym in [bodkočiarka] + statbegsys do
    begin
      if sym = bodkočiarka then getsym else error (10);
      príkaz ([bodkočiarka, endsym] + fsys)
    end;
    if sym = endsym then getsym else error (17)
  end else
  if sym = whilesym then
  begin getsym; podmienka ([dosym] + fsys);
    if sym = dosym then getsym else error (18);
    príkaz (fsys);
  end;
  test (fsys, [ ], 19)
end {príkaz};
begin {blok}
```



```
repeat
  if sym = constsym then
  begin getsym;
    repeat constdeklarácia;
      while sym = čiarka do
        begin getsym; constdeklarácia
        end;
      if sym = bodkočiarka then getsym else error (5)
    until sym ≠ identifikátor
  end;
  if sym = varsym then
  begin getsym;
    repeat vardeklarácia;
      while sym = čiarka do
        begin getsym; vardeklarácia
        end;
      if sym = bodkočiarka then getsym else error (5)
    until sym ≠ identifikátor;
  end;
  while sym = procsym do
  begin getsym;
    if sym = identifikátor then
      begin vstup (procedúra); getsym
      end
    else error (4);
    if sym = bodkočiarka then getsym else error (5);
    blok (tx, [bodkočiarka] + fsys);
    if sym = bodkočiarka then
      begin getsym; test (statbegsys + [identifikátor, procsym],
                          fsys, 6)
      end
    else error (5)
  test (statbegsys + [identifikátor], declbegsys, 7)
until ¬ (sym in declbegsys);
príkaz ([bodkočiarka, endsym] + fsys);
test (fsys, [ ], 8);
```

```
end {blok};
begin {hlavný program}
   ..... Inicializácia (pozri program 5.4) .....
   cc: = 0; ll: = 0; ch: = ' '; kk: = al; getsym;
blok (0, [bodka] + declbegsys + statbegsys);
if sym ≠ bodka then error (9);
99: writeln
end.
```

5.10 PROCESOR JAZYKA PL/0

Je skutočne pozoruhodné, že doterajší vývoj kompilátora jazyka PL/0 prebiehal bez akýchkoľvek vedomostí o počítači, pre ktorý má generovať cieľový kód. Ale prečo by mala štruktúra cieľového počítača ovplyvniť schému syntaktickej analýzy a mechanizmus zotavenia sa zo syntaktických chýb! Naozaj by nemala. Naopak vhodná schéma generovania kódu pre ľubovoľný počítač by mala byť odvodená z existujúceho syntaktického analyzátora metódou postupného zjemňovania programu. Pretože práve uvedená myšlienka je naším najbližším cieľom, musíme vybrať procesor, pre ktorý budeme generovať kód.

Vzhľadom na to, že chceme, aby opis kompilátora bol primerane jednoduchý a zbavený akýchkoľvek špeciálnych úvah prameniacich z rôznych vlastností reálneho, existujúceho procesora, zvolíme si počítač podľa našich predstáv, špeciálne prispôsobený potrebám jazyka PL/0. Pretože takýto procesor v skutočnosti fyzicky neexistuje, je to hypotetický procesor a nazývame ho počítačom PL/0.

Cieľom tohto článku nie je podrobne vysvetliť dôvody, prečo sme zvolili práve takúto architektúru počítača, ale poskytnúť neformálny opis, pozostávajúci z intuitívneho úvodu a podrobnej definície procesora vo forme algoritmu. Takáto formalizácia nám môže poslúžiť ako príklad presného a podrobného algoritmického opisu skutočného procesora. Algoritmus interpretuje inštrukcie jazyka PL/0 sekvenčným spôsobom a nazýva sa *interpret*.

Počítač PL/0 sa skladá z dvoch druhov pamätí: inštrukčného registra a troch adresových registrov. Pamäť pre program nazývaná *kód*

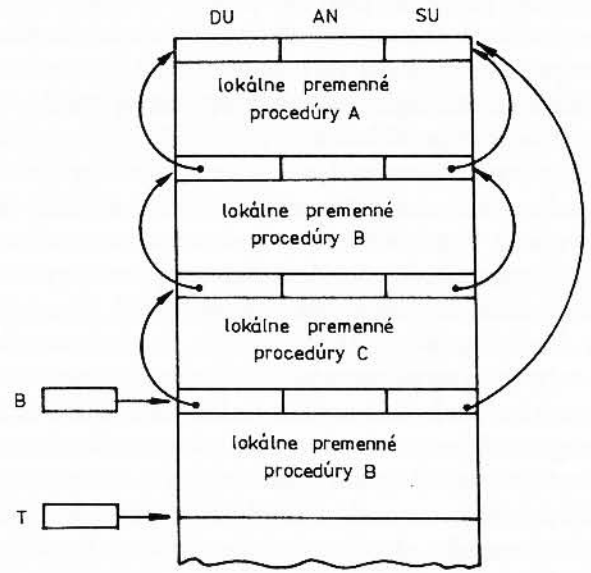
napĺňa kompilátor a v priebehu interpretovania kódu zostáva jej obsah nezmenený. Možno ju pokladať za pamäť, z ktorej sa dá iba čítať. Pamäť *S* pre údaje je organizovaná ako zásobník a všetky aritmetické operátory operujú s dvoma prvkami, ktoré sú na vrchu tohto zásobníka, pričom výsledok operácie nahradí ich operandy. Prvok na vrchu tohto zásobníka je prístupný (adresovateľný) prostredníctvom registra *T*, nazývaného *register vrchu zásobníka*. Práve interpretovaná inštrukcia sa nachádza v registri inštrukcií *I*. Register adresy programu, označený symbolom *P*, obsahuje adresu ďalšej inštrukcie, ktorá sa bude interpretovať.

Každá procedúra v jazyku PL/0 môže obsahovať lokálne premenné. Pretože procedúry môžu byť volané rekurzívne, nemôže byť pamäť pre lokálne premenné vyhradená pred skutočným volaním takejto procedúry. Preto je potrebné, aby sa údajové segmenty jednotlivých procedúr ukladali postupne do zásobníkovej pamäti *S*. Pretože sa volanie procedúr riadi výhradne stratégiou prvý-dnu-posledný-von, zásobník je vhodným prostriedkom prideľovania pamäti. Každá procedúra vlastní niektoré svoje interné informácie, menovite adresu v programe, odkiaľ bola volaná (adresu návratu) a adresu údajového segmentu procedúry, z ktorej bola volaná. Tieto dve adresy sú potrebné na úspešné pokračovanie realizácie programu po ukončení činnosti procedúry. Považujeme ich za vnútorné alebo implicitné lokálne premenné, ktorých pamäť je vyhradená v údajovom segmente procedúry, a nazývame ich adresa návratu *AN* a dynamický ukazovateľ *DU*. Začiatok dynamického ukazovateľa, t. j. adresa posledného vytvoreného údajového segmentu, sa nachádza v registri *B* nazývanom *register bázovej adresy*.

Pretože skutočné prideľovanie pamäti sa vykonáva až v čase behu (interpretácie) programu, nemôže kompilátor generovať cieľový kód s absolútnymi adresami. Jediné, čo dokáže urobiť, je vypočítať umiestnenia premenných v rámci údajového segmentu, t. j. určiť ich relatívne adresy. Interpret musí potom k týmto adresám pripočítať posunutie príslušného údajového segmentu vzhľadom na bázovú adresu. Ak je niektorá premenná lokálna v práve interpretovanej procedúre, tak sa bázová adresa nachádza v registri *B*. V opačnom prípade je potrebné túto adresu zistiť zostupným prechodom reťaze údajových segmentov. Kompilátor však môže poznať iba statickú hĺbku prístupovej cesty,

zatiaľ čo reťaz dynamických väzieb udržiava dynamickú históriu volaní procedúry. Žiaľ, tieto dve prístupové cesty nie sú tie isté.

Predpokladajme napr., že procedúra *A* volá procedúru *B*, ktorá je deklarovaná lokálne v procedúre *A*, procedúra *B* volá procedúru *C* deklarovanú lokálne v procedúre *B* a procedúra *C* volá procedúru *B* (rekurzívne). Hovoríme, že procedúra *A* je deklarovaná na úrovni 1, procedúra *B* na úrovni 2 a procedúra *C* na úrovni 3 (*obr. 5.7*). Ak chceme v procedúre *B* adresovať premennú a deklarovanú v procedúre *A*, tak kompilátor vie, že medzi procedúrami *A* a *B* existuje úrovňový rozdiel 1. Zostup o jeden krok pozdĺž reťaze dynamických väzieb však spôsobí sprístupnenie lokálnych premenných procedúry *C*!



Obr. 5.7. Zásobník počítača PL/0

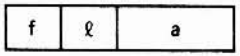
Z toho vyplýva, že budeme potrebovať ďalšiu reťaz väzieb, pomocou ktorej pospájame údajové segmenty tak, aby sa kompilátor pri generovaní kódu dokázal orientovať v každej situácii. Toto spojenie nazveme statický ukazovateľ a označíme ho *SU*.

Generované adresy potom budú pozostávať z dvojíc čísel, z ktorých prvé bude určovať rozdiel statických úrovní a druhé relatívne posunutie v rámci údajového segmentu. Predpokladáme, že každé pamäťové miesto môže obsahovať buď adresu, alebo celé číslo.

Množina inštrukcií počítača PL/0 je prispôsobená požiadavkám jazyka PL/0. Obsahuje tieto inštrukcie:

1. Inštrukcia na uloženie čísel (literálov) do zásobníka (LIT).
2. Inštrukcia na uloženie premenných na vrch zásobníka (LOD).
3. Inštrukcia na zápis objektu z vrchu zásobníka do pamäti, zodpovedajúca priraďovacím príkazom (STO).
4. Inštrukcia na vyvolanie procedúry, zodpovedajúca príkazu **call** (CAL).
5. Inštrukcia na vyhradenie pamäti v zásobníku zväčšením hodnoty smerníka, ukazujúceho na vrch zásobníka *T* (INT).
6. Inštrukcie na nepodmienené a podmienené odovzdanie riadenia (skokové inštrukcie), používané v príkazoch **if** a **while** (JMP, JPC).
7. Množina aritmetických a relačných operátorov (OPR).

Formát inštrukcií (pozri *obr. 5.8*) pozostáva z troch prvkov: kódu operácie *f* a parametrov (jedného alebo dvoch). V prípade operátorov určuje parameter a identitu operátora, v ostatných prípadoch buď číslo (ide o inštrukcie LIT a INT), alebo adresu v programe (JMP, JPC, CAL), alebo adresu údajov (LOD, STO).



Obr. 5.8. Formát inštrukcie

Podrobnosti o činnosti počítača PL/0 možno vidieť v procedúre, nazvanej interpret, ktorá je časťou programu 5.6. Tento program je kombináciou úplného kompilátora s interpretom a slúži na preklad a výpočet programov napísaných v jazyku PL/0. Čitateľ sa môže v rámci cvičenia pokúsiť o modifikáciu tohto programu s cieľom generovať cieľový kód pre existujúci počítač. Výsledné rozšírenie kompilátora môžeme považovať za mieru vhodnosti vybraného počítača pre uvedenú úlohu.

Nepochybujeme o tom, že by sa uvedený počítač PL/0 dal vylepšiť

dômyselnejšou organizáciou s efektívnejšími operáciami. Jedným z možných kandidátov na vylepšenie je mechanizmus adresovania. Uvedené riešenie sme zvolili pre jeho prirodzenú jednoduchosť a preto, že z neho sú odvodené všetky možné vylepšenia.

5.11 GENEROVANIE CIEĽOVÉHO KÓDU

Keď chce kompilátor zostaviť (vygenerovať) inštrukciu, musí poznať, jej operačný kód a parameter, ktorým môže byť buď literál (číslo), alebo adresa. Počas spracúvania deklarácií konštánt, premenných a procedúr spája kompilátor tieto hodnoty s príslušnými identifikátormi. Na tieto účely je tabuľka identifikátorov rozšírená o atribúty, ktoré sú spojené s každým identifikátorom. Ak príslušný identifikátor označuje konštantu, jeho atribútom je hodnota tejto konštanty; ak identifikátor označuje premennú, jeho atribútom je jej adresa, pozostávajúca z posunutia a úrovne; ak identifikátor označuje procedúru, tak je jeho atribútom adresa vstupného bodu procedúry a jej úroveň. Zodpovedajúce rozšírenie deklarácie premennej *tab* je uvedené v programe 5.6. Je to pozoruhodný príklad postupného zjemňovania (alebo obohacovania) deklarácie údajov, postupujúceho súčasne so zjemňovaním príkazovej časti programu.

Zatiaľ čo hodnoty konštánt poskytuje samotný text programu, adresy premenných musí vypočítať kompilátor. Pretože jazyk PL/0 je dostatočne jednoduchý, zvolila sa stratégia sekvenčného prideľovania pamäti pre premenné a kód. Spracovaním deklarácie každej premennej sa zvýši hodnota indexu prideľovania pamäti pre údaje o jednotku (pretože každá premenná zaberá podľa definície počítača PL/0 presne jedno pamäťové miesto). Uvedený index, označíme ho *dx*, je inicializovaný v okamihu zahájenia kompilácie procedúry, čím je súčasne dokumentovaná skutočnosť, že údajový segment procedúry je na začiatku jej kompilácie prázdny. (V skutočnosti je indexu *dx* priradená hodnota 3, pretože každý údajový segment obsahuje aspoň tri interné premenné *AN*, *DU*, *SU* (pozri predchádzajúci článok).) Príslušné výpočty atribútov identifikátorov sú zahrnuté v procedúre vstup, ktorá pridáva každý nový identifikátor do tabuľky.

Generovanie skutočného kódu pomocou uvedených informácií

o operandoch je už jednoduché. Vzhľadom na primeranú zásobníkovú organizáciu počítača PL/0 je vzťah medzi operandmi a operátormi zdrojového jazyka a inštrukciami cieľového kódu v pomere jedna ku jednej. Úlohou kompilátora je v podstate iba vhodné preskupenie do postfixovej formy. Pod postfixovou formou rozumieme také usporiadanie operandov a operátorov, v rámci ktorého operátory vždy nasledujú za svojimi operandmi. Zatiaľ čo pri bežnej, *infixovej forme* sa operátory nachádzajú medzi operandmi. *Postfixová forma* sa niekedy nazýva aj *Poľská forma* (podľa svojho autora LUKASCIEWICZA) alebo bezzátvorková forma, pretože zátvorky sú zbytočné. Niektoré vzťahy medzi infixovou a postfixovou formou výrazov ukazuje *tab. 5.4* (pozri aj odsek 4.4.2).

Výrazy v infixovej a postfixovej forme *Tabuľka 5.4*

Infixová forma	Postfixová forma
$x + y$	$xy +$
$(x - y) + z$	$xy - z +$
$x - (y + z)$	$xyz +$
$x * (y + z) * w$	$xyz + * w *$

Veľmi jednoduchá technika tejto transformácie je realizovaná v procedúrach výraz a term v programe 5.6. Je to v podstate iba záležitosť pozdržania momentu generovania aritmetického operátora. Čitateľ by si mal overiť, či príslušné procedúry syntaktického analyzátora dbajú aj o patričnú interpretáciu všeobecných pravidiel priority medzi rôznymi operátormi.

Preklad podmienených príkazov a príkazov cyklu je o niečo zložitejšia záležitosť. V týchto prípadoch je potrebné vygenerovať inštrukcie skokov, ktorých cieľová adresa niekedy nie je známa. Ak trváme na dôslednom sekvenčnom generovaní inštrukcií v tvare výstupného súboru, tak potrebujeme schému dvojprechodového kompilátora. Cieľom činnosti druhého prechodu je doplniť neúplné inštrukcie skokov o ich cieľové adresy. Alternatívne riešenie, aplikované v našom kompilátore, spočíva v umiestnení inštrukcií do nejakého poľa, t. j. do priamo prístupnej pamäti. V tomto poli sa potom príslušné inštrukcie dopĺňajú

chýbajúcimi adresami (len čo sú známe). Táto operácia sa nazýva *fixovanie inštrukcie* (z anglického termínu fixup — pozn. prekl.).

Jedinou prídavnou operáciou pri generovaní takéhoto skoku na neznámu adresu (skok dopredu) je uchovanie jej adresy, t. j. jej indexu do pamäti programu. Táto adresa sa potom pri fixovaní použije na sprístupnenie príslušnej neúplnej inštrukcie. Podrobnosti možno nájsť opäť v programe 5.6 (pozri procedúry spracúvajúce príkazy **if** a **while**). Schémy generovania kódu pre príkazy **if** a **while** sú (*L*1 a *L*2 sú adresy inštrukcií):

if *C* **then** *S*	**while** *C* **do** *S*
kód pre podmienku *C*	*L*1: kód pre *C*
JPC *L*1	JPC *L*2
	kód pre *S*
kód pre príkaz *S*	JMP *L*1 pre *S*
*L*1: ...	*L*2: ...

Pre jednoduchosť programu zavedieme pomocnú premennú *gen*. Jej úlohou je vyprodukovať hotovú inštrukciu podľa jej troch parametrov. Automaticky zvyšuje hodnotu indexu *cx* (nazvaného počítadlo adries), ktorý určuje adresu najbližšie vygenerovanej inštrukcie. Nasledujúci príklad dokumentuje v mnemonickom tvare vygenerovaný kód pre procedúru násobenia (procedúra 5.14). Poznámky na pravej strane inštrukcií sú uvedené iba kvôli zrozumiteľnosti a dokumentácii.

Vygenerovaný kód pre procedúru 5.14 napísanú v jazyku PL/0

```
 2  INT   0,5   vyhradenie miesta pre ukazovatele a lokálne premenné
 3  LOD   1,3   x
 4  STO   0,3   a
 5  LOD   1,4   y
 6  STO   0,4   b
 7  LIT   0,0   0
 8  STO   1,5   z
 9  LOD   0,4   b
10  LIT   0,0   0
11  OPR   0,12  >
12  JPC   0,29
13  LOD   0,4   b
14  OPR   0,7   odd
15  JPC   0,20
16  LOD   1,5   z
17  LOD   0,3   a
18  OPR   0,2   +
19  STO   1,5   z
20  LIT   0,2   2
21  LOD   0,3   a
22  OPR   0,4   *
23  STO   0,3   a
24  LOD   0,4   b
25  LIT   0,2   2
26  OPR   0,5   /
27  STO   0,4   b
28  JMP   0,9
29  OPR   0,0   návrat
```

Mnohé úlohy súvisiace s kompiláciou programovacích jazykov sú oveľa zložitejšie ako tie, ktoré sme uviedli pri kompilátore jazyka PL/0 pre počítač PL/0 [5-4]. Väčšina z nich ťažko dospeje k elegantnej organizácii. Čitateľ, ktorý by sa pokúsil rozšíriť uvedený kompilátor jazyka buď za účelom zvýšenia výrazových možností jazyka, alebo pre konvenčnejší počítač, čoskoro spozná pravdivosť tohto tvrdenia. Záverom môžeme konštatovať, že základný prístup k tvorbe zložitého programu, uvedeného v tejto knihe, zostáva v platnosti, ba dokonca ešte i získava na svojej hodnote, ak bude riešená úloha zložitejšia a intelektuálne náročnejšia. Dôkazom úspešnosti použitia spomínaného prístupu môžu byť konštrukcie veľkých kompilátorov [5-1] a [5-9].

PROGRAM 5.6. *Kompilátor jazyka PL/0*

```
program PL/0 (input, output);
{kompilátor jazyka PL/0 s generátorom cieľového kódu}
label 99;
const norw = 11;        {počet kľúčových slov}
```

```
    txmax = 100;   {veľkosť tabuľky identifikátorov}
    nmax = 14;     {maximálny počet číslic v čísle}
    al = 10;       {dĺžka identifikátorov}
    amax = 2047;   {najvyššia adresa}
    levmax = 200;  {maximálna hĺbka vložených blokov}
    cxmax = 200;   {veľkosť priestoru pre kód}
type symbol =
  (nul, identifikátor, číslo, plus, mínus, krát, deleno, oddsym, eql, neq,
   lss, leq, gtr, geq, ľzátvorka, pzátvorka, čiarka, bodkočiarka, bodka,
   nadobudne, beginsym, endsym, ifsym, thensym, whilesym, dosym,
   callsym, constsym, varsym, procsym);
   alfa = packed array [1..al] of char;
   objekt = (konštanta, premenná, procedúra);
   symset = set of symbol;
   fct = (lit, opr, lod, sto, cal, int, jmp, jpc); {inštrukcie}
   inštrukcia = packed record
                  f: fct;          {kód inštrukcie}
                  l: 0..levmax;    {úroveň}
                  a: 0..amax;      {adresa posunutia}
                end;
{LIT  0, a: ulož konštantu a do zásobníka
 OPR  0, a: vykonaj inštrukciu a
 LOD  l, a: ulož premenné l, a na vrchol zásobníka
 STO  l, a: zapíš premennú l z vrchu zásobníka do pamäti
 CAL  l, a: volaj procedúru a z úrovne l
 INT  0, a: zvýš obsah registra t o hodnotu a
 JMP  0, a: vykonaj skok na adresu a
 JPC  0, a: vykonaj podmienený skok na adresu a}
var ch: char;          {posledný prečítaný znak}
    sym: symbol;       {posledný prečítaný symbol}
    id: alfa;          {posledný prečítaný identifikátor}
    num: integer;      {posledné prečítané číslo}
    cc: integer;       {počet znakov}
    ll: integer;       {dĺžka riadku}
    kk, err: integer;
    cx: integer;       {počítadlo adries}
```



```
    riadok: array [1..81] of char;
    a: alfa;
    kód: array [0..cxmax] of inštrukcia;
    slovo: array [1..norw] of symbol;
    ssym: array [char] of symbol;
    wsym: array [1..norw] of symbol;
    mnemo: array [fct] of
                  packed array [1..5] of char;
    declbegsys, statbegsys, facbegsys: symset;
          tab: array [0..txmax] of
                  record meno: alfa;
                    case druh: objekt of
                      konštanta: (val: integer);
                      premenná, procedúra: (úroveň, adr: integer)
                  end;
procedure error (n: integer);
begin
  writeln ('****', ' ': cc - 1, '↑', n: 2); err := err + 1
end {error};
procedure getsym;
  var i, j, k: integer;
  procedure getch;
  begin if cc = ll then
    begin if eof (input) then
            begin write ('NEÚPLNÝ PROGRAM'); goto 99
            end;
      ll := 0; cc := 0; write (cx: 5, ' ');
      while ¬ eoln (input) do
        begin ll := ll + 1; read (ch); write (ch); riadok [ll] := ch
        end;
        writeln; ll := ll + 1; read (riadok [ll])
    end;
    cc := cc + 1; ch := riadok [cc]
  end {getch};
begin {getsym}
  while ch = ' ' do getch;
```

```
  if ch in ['A'..'Z'] then
  begin {identifikátor alebo kľúčové slovo}
  k:=0;
    repeat if k<al then
      begin k:=k+1; a[k]:=ch
      end;
      getch
    until ¬(ch in ['A'..'Z', '0'..'9']);
    if k≧kk then kk:=k else
      repeat a[kk]:=' '; kk:=kk-1
      until kk=k;
    id:=a; i:=1; j:=norw;
    repeat k:=(i+j) div 2;
      if id≦slovo[k] then j:=k-1;
      if id≧slovo[k] then i:=k+1
    until i>j;
    if i-1>j then sym:=wsym[k] else sym:=identifikátor
  end else
  if ch in ['0'..'9'] then
  begin {číslo} k:=0; num:=0; sym:=číslo;
    repeat num:=10*num+(ord(ch)-ord('0'));
      k:=k+1; getch
    until ¬(ch in ['0'..'9']);
    if k>nmax then error (30)
  end else
  if ch=':' then
  begin getch;
    if ch='=' then
    begin sym:=nadobudne; getch
    end else sym:=nul;
  end else
  begin sym:=ssym[ch]; getch
  end
end {getsym};
procedure gen(x: fct; y, z: integer);
begin if cx>cxmax then
```



```
    begin write ('PROGRAM JE PRÍLIŠ VEĽKÝ'); goto 99
    end;
    with kód [cx] do
      begin f:=x; l:=y; a:=z
      end;
    cx:=cx+1
end {gen};
procedure test (s1, s2: symset; n: integer);
begin if ¬(sym in s1) then
    begin error (n); s1:=s1+s2;
      while ¬(sym in s1) do getsym
    end
end {test};
procedure blok (lev, tx: integer; fsys: symset);
  var dx: integer;    {index prideľovania pamäti pre údaje}
      tx0: integer;   {začiatočný index do tabuľky identifikátorov}
      cx0: integer;   {začiatočná hodnota počítadla adries}
  procedure vstup (k: objekt);
  begin {zaradenie objektu do tabuľky}
    tx:=tx+1;
    with tab[tx] do
    begin meno:=id; druh:=k;
      case k of
        konštanta: begin if num>amax then
                            begin error (31); num:=0 end;
                          val:=num
                   end;
        premenná: begin úroveň:=lev; adr:=dx; dx:=dx+1;
                  end;
        procedúra: úroveň:=lev
        end
      end
    end {vstup};
    function pozícia (id: alfa): integer;
      var i: integer;
    begin {vyhľadanie identifikátora id v tabuľke}
```

```
      tab[0].meno := id; i := tx;
      while tab[i].meno ≠ id do i := i - 1;
      pozícia := i
    end {pozícia};
    procedure constdeklarácia;
    begin if sym = identifikátor then
      begin getsym;
      if sym in [eql, nadobudne] then
      begin if sym = nadobudne then error (1);
        getsym;
          if sym = číslo then
          begin vstup (konštanta); getsym
          end
        else error (2)
      end else error (3)
    end else error (4)
  end {constdeklarácia};
procedure vardeklarácia;
begin if sym = identifikátor then
    begin vstup (premenná); getsym
    end else error (4)
end {vardeklarácia};
procedure zobrazkód;
  var i: integer;
begin {vytlačenie vygenerovaného kódu pre tento blok}
  for i := cx0 to cx - 1 do
    with kód[i] do
      writeln (i, mnemo [f]: 5, l:3, a:5)
end {zobrazkód};
procedure príkaz (fsys: symset);
  var i, cx1, cx2: integer;
  procedure výraz (fsys: symset);
    var addop: symbol;
    procedure term (fsys: symset);
      var mulop: symbol;
      procedure factor (fsys: symset);
```



```
      var i: integer;
    begin test (facbegsys, fsys, 24);
      while sym in facbegsys do
      begin
        if sym = identifikátor then
        begin i := pozícia (id);
          if i = 0 then error (11) else
          with tab[i] do
          case druh of
          konštanta: gen (lit, 0, val);
          premenná: gen (lod, lev-úroveň, adr);
          procedúra: error (21)
          end;
          getsym
        end else
        if sym = číslo then
        begin if num > amax then
            begin error (30); num := 0
            end;
          gen (lit, 0, num); getsym
        end else
        if sym = ľzátvorka then
        begin getsym; výraz ([pzátvorka] + fsys);
          if sym = pzátvorka then getsym else error (22)
        end;
        test (fsys, [ľzátvorka], 23)
      end
    end {factor};
  begin {term} factor (fsys + [krát, deleno]);
    while sym in [krát, deleno] do
      begin mulop := sym, getsym; factor (fsys + [krát, deleno]);
      if mulop = krát then gen (opr, 0, 4) else gen (opr, 0, 5)
      end
  end {term};
begin {výraz}
  if sym in [plus, mínus] then
```

```
    begin addop: = sym; getsym; term (fsys + [plus, minus]);
      if addop = minus then gen (opr, 0, 1)
    end else term (fsys + [plus, minus]);
  while sym in [plus, minus] do
    begin addop: = sym; getsym; term (fsys + [plus, minus]);
      if addop = plus then gen (opr, 0, 2) else gen (opr, 0, 3)
    end
end {výraz};
procedure podmienka (fsys: symset);
  var relop: symbol;
begin
  if sym = oddsym then
    begin getsym; výraz (fsys); gen (opr, 0, 6)
    end else
    begin výraz ([eql, neq, lss, gtr, leq, geq] + fsys);
      if ¬ (sym in [eql, neq, lss, leq, gtr, geq]) then
        error (20) else
      begin relop: = sym; getsym; výraz (fsys);
        case relop of
        eql: gen (opr, 0, 8);
        neq: gen (opr, 0, 9);
        lss: gen (opr, 0, 10);
        geq: gen (opr, 0, 11);
        gtr: gen (opr, 0, 12);
        leq: gen (opr, 0, 13);
        end
      end
  end
end {podmienka};
begin {príkaz}
  if sym = identifikátor then
  begin i: = pozícia (id);
    if i = 0 then error (11) else
    if tab[i].druh ≠ premenná then
      begin {priradenie hodnoty objektu, ktorý nie je premennou}
          error (12); i: = 0
```



```
    end;
  getsym; if sym = nadobudne then getsym else error (13);
  výraz (fsys);
  if i ≠ 0 then
    with tab[i] do gen (sto, lev-úroveň, adr)
end else
if sym = callsym then
begin getsym;
  if sym ≠ identifikátor then error (14) else
    begin i: = pozícia (id);
      if i: = 0 then error (11) else
      with tab[i] do
        if druh = procedúra then gen (cal, lev-úroveň, adr)
        else error (15);
      getsym
    end
end else
if sym = ifsym then
begin getsym; podmienka ([thensym, dosym] + fsys);
  if sym = thensym then getsym else error (16);
  cx1: = cx; gen (jpc, 0, 0);
  príkaz (fsys); kód [cx1].a: = cx
end else
if sym = beginsym then
begin getsym; príkaz ([bodkočiarka, endsym] + fsys);
  while sym in [bodkočiarka] + statbegsys do
  begin
    if sym = bodkočiarka then getsym else error (10);
    príkaz ([bodkočiarka, endsym] + fsys)
  end;
  if sym = endsym then getsym else error (17)
end else
if sym = whilesym then
begin cx1: = cx; getsym; podmienka ([dosym] + fsys);
  cx2: = cx; gen (jpc, 0, 0);
  if sym = dosym then getsym else error (18);
```

```
      príkaz (fsys); gen (jmp, 0, cx1); kód [cx2].a := cx
    end;
    test (fsys, [ ], 19)
    end {príkaz};
begin {blok}
    dx := 3; tx0 := tx; tab[tx].adr := cx; gen (jmp, 0, 0);
    if lev > levmax then error (32);
    repeat
      if sym = constsym then
      begin getsym;
        repeat constdeklarácia;
          while sym = čiarka do
            begin getsym; constdeklarácia
            end;
        if sym = bodkočiarka then getsym else error (5)
        until sym ≠ identifikátor
      end;
      if sym = varsym then
      begin getsym;
        repeat vardeklarácia;
          while sym = čiarka do
            begin getsym; vardeklarácia
            end;
          if sym = bodkočiarka then getsym else error (5)
        until sym ≠ identifikátor;
      end;
      while sym = procsym do
      begin getsym;
        if sym = identifikátor then
          begin vstup (procedúra); getsym
          end
        else error (4);
        if sym = bodkočiarka then getsym else error (5);
        blok (lev + 1, tx, [bodkočiarka] + fsys);
        if sym = bodkočiarka then
```



```
          begin getsym; test (statbegsys + [identifikátor,
                              procsym], fsys, 6)
          end
        else error (5)
      end;
      test (statbegsys + [identifikátor], declbegsys, 7)
    until ¬ (sym in declbegsys);
    kód [tab[tx0]. adr].a := cx;
    with tab[tx0] do
      begin adr := cx; {začiatočná adresa kódu}
      end;
    cx0 := cx; gen (int, 0, dx);
    príkaz ([bodkočiarka, endsym] + fsys);
    gen (opr, 0, 0); {návrat}
    test (fsys, [ ], 8);
    zobrazkód;
end {blok};
procedure interpret;
    const maxzásobník = 500;
    var p, b, t: integer; {programový, bázový a zásobníkový register}
        i: inštrukcia; {inštrukčný register}
        s: array [1 .. maxzásobník] of integer; {pamäť pre údaje}
    function báza (l: integer): integer;
      var b1: integer;
    begin b1 := b; {vyhľadanie bázy na l-tej úrovni}
      while l > 0 do
        begin b1 := s[b1]; l := l - 1
        end;
      báza := b1
    end {báza};
begin writeln ('START PL/0');
    t := 0; b := 1; p := 0;
    s[1] := 0; s[2] := 0; s[3] := 0;
    repeat i := kód [p]; p := p + 1;
      with i do
```

```
case f of
lit:  begin t:=t+1; s[t]:=a
      end;
opr: case a of {operátor}
       0: begin {návrat}
            t:=b-1; p:=s[t+3]; b:=s[t+2];
          end;
       1: s[t]:=-s[t];
       2: begin t:=t-1; s[t]:=s[t]+s[t+1]
          end;
       3: begin t:=t-1; s[t]:=s[t]-s[t+1]
       4: begin t:=t-1; s[t]:=s[t]*s[t+1]
          end;
       5: begin t:=t-1; s[t]:=s[t] div s[t+1]
          end;
       6: s[t]:=ord(odd s[t]));
       8: begin t:=t-1; s[t]:=ord(s[t]=s[t+1])
          end;
       9: begin t:=t-1; s[t]:=ord(s[t]≠s[t+1])
          end;
      10: begin t:=t-1; s[t]:=ord(s[t] s[t+1])
          end;
      11: begin t:=t-1; s[t]:=ord(s[t]=s[t+1])
          end;
      12: begin t:=t-1; s[t]:=ord(s[t] s[t+1])
          end;
      13: begin t:=t-1; s[t]:=ord(s[t]=s[t+1])
          end;
      end;
lod:  begin t:=t+1; s[t]:=s[báza(l)+a]
      end;
sto:  begin s[báza(l)+a]:=s[t]; writeln(s[t]; t:=t-1)
      end;
cal:  begin {vytvorenie nového bloku}
        s[t+1]:=báza(l); s[t+2]:=:=b; s[t+3]:=p;
        b:=t+1; p:=a
```



```
        end;
    int:  t:=t+a;
    jmp: p:=a;
    jpo:  begin if s[t]=0 then p:=a; t:=t-1
          end
      end {with, case}
  until p=0;
  write ('END PL/0');
end {interpret};
begin {hlavný program}
  for ch:='A' to ';' do ssym [ch]:=nul;
  slovo [1]:='BEGIN';          slovo [2]:='CALL';
  slovo [3]:='CONST';          slovo [4]:='DO';
  slovo [5]:='END';            slovo [6]:='IF';
  slovo [7]:='ODD';            slovo [8]:='PROCEDURE';
  slovo [9]:='THEN';           slovo [10]:='VAR';
  slovo [11]:='WHILE';
  wsym [1]:=beginsym;          wsym [2]:=callsym;
  wsym [3]:=constsym;          wsym [4]:=dosym;
  wsym [5]:=endsym;            wsym [6]:=ifsym;
  wsym [7]:=oddsym;            wsym [8]:=procsym;
  wsym [9]:=thensym;           wsym [10]:=varsym;
  wsym [11]:=whilesym;
  ssym ['+']:=plus;            ssym ['-']:=mínus;
  ssym ['*']:=krát;            ssym ['/']:=deleno;
  ssym ['(']:=ľzátvorka;       ssym [')']:=pzátvorka;
  ssym ['=']:=eql;             ssym [',']:=čiarka;
  ssym ['.']:=bodka;           ssym ['≠']:=neq;
  ssym ['<']:=lss;             ssym ['>']:=gtr;
  ssym ['≦']:=leq;             ssym ['≧']:=geq;
  ssym [';']:=bodkočiarka;
  mnemo [lit]:='LIT';          mnemo [opr]:='OPR';
  mnemo [lod]:='LOD';          mnemo [sto]:='STO';
  mnemo [cal]:='CAL';          mnemo [int]:='INT';
  mnemo [jmp]:='JMP';          mnemo [jpc]:='JPC';
  declbegsys:=[constsym, varsym, procsym];
```

```
  statbegsys: = [beginsym, callsym, ifsym, whilesym];
  facbegsys: = [identifikátor, číslo, ľzátvorka];
  page (výstup); err: = 0;
  cc: = 0; cx: = 0; ll: = 0; ch: = ' '; kk: = al; getsym;
  blok (0, 0, [bodka] + declbegsys + statbegsys);
  if sym ≠ bodka then error (9);
  if err = 0 then interpret else write ('CHYBY V PL/0 PROGRAME');
99: writeln
end.
```

Cvičenia

5.1. Nech je daná nasledujúca syntax:

$$S::=A$$
$$A::=B\,|\,\textbf{if}\ A\ \textbf{then}\ A\ \textbf{else}\ A$$
$$B::=C\,|\,B+C\,|\,+C$$
$$C::=D\,|\,C*D\,|\,*D$$
$$D::=x\,|\,(A)\,|\,-D$$

Ktoré z uvedených symbolov sú terminálne a ktoré neterminálne? Určte množiny najľavejších a nasledujúcich symbolov $L(X)$ a $F(X)$ pre každý neterminálny symbol X. Zostrojte postupnosť krokov syntaktickej analýzy pre nasledujúce vety:

$x+x$
$(x+x)*(+-x)$
$(x*-+x)$
if $x+x$ **then** $x*x$ **else** $-x$
if x **then if** $-x$ **then** x **else** $x+x$ **else** $x*x$
if $-x$ **then** x **else if** x **then** $x+x$ **else** x

5.2. Rozhodnite, či gramatika z cvičenia 5.1 spĺňa obmedzujúce tvrdenia 1 a 2 pre syntaktickú analýzu zhora nadol s predsnímaním jedného symbolu dopredu. Ak nie, nájdite ekvivalentnú syntax, ktorá uvedeným tvrdeniam vyhovuje. Vyjadrite túto syntax prostredníctvom syntaktického grafu a štruktúry údajov použitých v programe 5.3.

5.3. Zopakujte cvičenie 5.2 pre nasledujúcu syntax:

$$S::=A$$
$$A::=B\,|\,\textbf{if}\ C\ \textbf{then}\ A\,|\,\textbf{if}\ C\ \textbf{then}\ A\ \textbf{else}\ A$$
$$B::=D=C$$
$$C::=\textbf{if}\ C\ \textbf{then}\ C\ \textbf{else}\ C\,|\,D$$

Návod: Zistite, ktoré konštrukcie treba vylúčiť alebo nahradiť, aby sa dala aplikovať syntaktická analýza zhora nadol s predsnímaním jedného symbolu dopredu.

5.4. Uvažujte o probléme syntaktickej analýzy zhora nadol pre túto syntax:

$$S::=A$$
$$A::=B+A\,|\,DC$$
$$B::=D\,|\,D*B$$
$$D::=x\,|\,(C)$$
$$C::=+x\,|\,-x$$

Zistite maximálny počet symbolov, ktoré treba prezrieť dopredu, aby bolo možné analyzovať vety podľa tejto syntaxe.

5.5. Vykonajte transformáciu definície jazyka PL/0 vyjadrenej syntaktickými grafmi (*obr. 5.4*) do ekvivalentnej množiny prepisovacích pravidiel BNF.

5.6. Napíšte program, ktorý pre každý neterminálny symbol S z danej množiny prepisovacích pravidiel určí množiny začiatočných a nasledujúcich symbolov $L(S)$ a $F(S)$.
Návod: Použite časť programu 5.3 na vytvorenie vnútornej reprezentácie syntaxe v tvare štruktúry údajov. Potom vhodne manipulujte s takto pospájanou štruktúrou.

5.7. Rozšírte jazyk PL/0 a jeho kompilátor o nasledujúce konštrukcie:

a) Podmienený príkaz v tvare:

⟨príkaz⟩ ::= **if** ⟨podmienka⟩ **then** ⟨príkaz⟩
else ⟨príkaz⟩

b) Príkaz cyklu v tvare:

⟨príkaz⟩ ::= **repeat** ⟨príkaz⟩ {; ⟨príkaz⟩
until ⟨podmienka⟩

Zistite, či uvedené rozšírenie jazyka spôsobí nejaké ťažkosti, ktoré by mohli viesť k zmene formy alebo interpretácie daných konštrukcií jazyka PL/0. Množinu inštrukcií počítača PL/0 však nesmiete rozšíriť o žiadne nové inštrukcie.

5.8. Rozšírte jazyk PL/0 a jeho kompilátor o možnosť používania parametrov procedúr. Zvážte dve možné alternatívy riešenia a na vašu realizáciu vyberte jednu z nich.

a) Parametre posielané hodnotou. Skutočnými parametrami pri vyvolaní procedúr sú výrazy, ktorých hodnoty sa priradia lokálnym premenným procedúr. Tieto lokálne premenné sú reprezentované formálnymi parametrami uvedenými v záhlaví deklarácií procedúr.

b) Parametre posielané referenciou. V tomto prípade sú skutočnými parametrami premenné, ktoré pri vyvolaní procedúr nahradia formálne parametre. Hodnotou parametra je teda adresa skutočného parametra, a nie jeho hodnota, ktorá sa zapíše na miesto formálneho parametra. Skutočné parametre sú potom prístupné nepriamo cez tieto adresy. Vidíme, že pomocou takýchto parametrov môžeme pristupovať k premenným mimo procedúr, v dôsledku čoho môžeme nasledujúcim spôsobom zmeniť pravidlo o rozsahu platnosti objektu: v každej procedúre sú priamo prístupné iba lokálne premenné; nelokálne premenné sú prístupné výlučne prostredníctvom parametrov.

5.9. Rozšírte jazyk PL/0 a jeho kompilátor o štruktúru pole. Predpokladajte, že rozsah indexov poľa *a* je určený v rámci jeho deklarácie nasledujúcim spôsobom:

var *a* (dolná : horná)

5.10. Upravte kompilátor jazyka PL/0 takým spôsobom, aby generoval kód pre váš počítač.
Návod: Generujte program v jazyku symbolických inštrukcií, aby ste sa vyhli problémom, ktoré by mohli nastať pri nedodržaní zásad spojovacieho a zavádzacieho programu vášho počítača. V prvom kroku sa vyhnite pokusom o optimalizáciu kódu. (Jedným z kandidátov na optimalizáciu je napr. mechanizmus prideľovania registrov.) Možné optimalizácie by sa mohli zaradiť do kompilátora až v priebehu štvrtého kroku jeho zjemňovania.

5.11. Rozšírte program 5.5 na program nazývaný „zúhľadňovač tlače". Cieľom tohto programu je prečítať text programu v jazyku PL/0 a vytlačiť ho v tvare, ktorý pekne zobrazuje textovú štruktúru vhodným oddelením riadkov a viacúrovňovým zarovnávaním. Na základe syntaktickej štruktúry jazyka PL/0 najprv definujte presné pravidlá oddeľovania riadkov a viacúrovňového zarovnávania, a potom ich implementujte prostredníctvom upravených príkazov pre zápis, ktoré vhodne zaraďte do programu 5.5. (Príkazy pre zápis musia byť potom, pochopiteľne, vyňaté z lexikálneho analyzátora jazyka PL/0.)

Zoznam použitej literatúry

5-1. AMMANN, U.: The Method of Structured Programming Applied to the Development of a Compiler. International Computing Symposium 1973, A. Günther a kol eds. Amsterdam: North-Holland Publishing Co. (1974), s. 93—99.

5-2. COHEN, D. J. — GOTLIEB, C. C.: A List Structure From of Grammars for Syntactic Analisis. Comp. Surveys, 2, No. 1 (1970), s. 65—82.

5-3. FLOYD, R. W.: The Syntax of Programming Languages — A Survey. IEEE Trans., EC-13 (1964), s. 346—353.

5-4. GRIES, D.: Compiler Construction for Digital Computers New York: Wiley (1971). (Slovenský preklad: Kompilátory číslicových počítačov. Bratislava, Alfa/SNTL 1981.)

5-5. KNUTH, D. E.: Top-down Syntax Analisis. Acta Informatica, 1, No. 2 (1971), s. 79—110.

5-6. LEWIS, P. M. — STEARNS, R. E.: Syntax-directed Transduction. J. ACM, 15, No. 3 (1968), s. 465—488.

5-7. NAUR, P. ed.: Report on the Algorithmic Language ALGOL 60. ACM, 6, No. 1 (1963), s. 1—17.

5-8. SCHOORE, D. V.: META II, A Syntax-oriented Compiler Writing Language. Proc. ACM Natl. Conf., 19 (1964), D 1.3.1 — 11.

5-9. WIRTH, N.: The Design of a PASCAL Compiler. Software-Practice and Experience, 1, No. 4 1971, s. 309—333.