VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PARALELNÍ SYNTAKTICKÁ ANALÝZA ZALOŽENÁ NA NĚKOLIKA METODÁCH

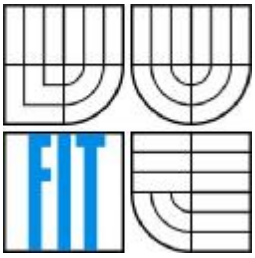
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE
AUTHOR

DOMINIK WOLF

BRNO 2009

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PARALELNÍ SYNTAKTICKÁ ANALÝZA ZALOŽENÁ NA NĚKOLIKA METODÁCH

PARALLEL PARSING BASED ON SEVERAL METHODS

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE DOMINIK WOLF
AUTHOR

VEDOUCÍ PRÁCE prof. RNDr. ALEXANDER MEDUNA CSc.
SUPERVISOR

BRNO 2009

Abstrakt

Tato práce se zabývá konstrukcí paralelního syntaktického analyzátoru založeného na několika vzájemně propojených metodách analýzy zdrojového textu. Byla použita metoda rekurzivního sestupu, která tvoří hlavní část analyzátoru, metoda precedenční analýzy, která je využita pro analýzu výrazů. V práci je postupně řešen návrh vhodného jazyka a gramatiky pro testování správné funkčnosti navrženého syntaktického analyzátoru. Dále je řešena konstrukce jednotlivých částí analyzátoru, včetně úvodní lexikální analýzy zdrojového textu.

Klíčová slova

Paralelní syntaktický analyzátor, paralelní syntaktická analýza, kombinovaná syntaktická analýza, komponentní syntaktický analyzátor, rekurzivní sestup, precedenční analýza.

Abstract

This work deals with a construction of a parallel parser which is based on several mutually interconnected methods of the source code analysis. The method of recursive descent has been used as the main part of the analyser and the method of precedence analysis to resolve phrases. The work gradually deals with the design of a suitable language and the grammar for testing of the functionality of the submitted parser. Further the construction of particular sections of the parser is being resolved including the introductory syntax analysis of the source code.

Keywords

Parallel parser, parallel syntax analysis, composite syntax analysis, composite parser, recursive descent, operator precedence analysis.

Citace

Dominik Wolf: Paralelní syntaktická analýza založená na několika metodách, bakalářská práce, Brno, FIT VUT v Brně, 2009.

Paralelní syntaktická analýza založená na několika metodách

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením prof. Alexandera Meduny.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Dominik Wolf
27. července 2009

Poděkování

Tímto bych chtěl velice poděkovat prof. Alexandru Medunovi za podporu a motivaci při tvorbě této bakalářské práce.

Obsah

Úvod

Interakce mezi lidmi je velice efektivně proveditelná pomocí jazyka, kterému všichni rozumí. Díky jazyku jsou lidé schopni vyjádřit své aktuální pocity, potřeby a myšlenky tak, že jim ostatní jsou schopni porozumět.

Při programování využívají vývojáři možnosti, jež jim poskytují programovací jazyky, které umožňují komunikaci mezi člověkem, který má problém, a počítačem, který je schopen pomoci problém vyřešit. Efektivní programovací jazyk musí přemostit díru mezi často nestrukturovaným lidským myšlením a precizně definovanými požadavky počítačové logiky. V současné době jsou využívány tzv. vysoko úrovňové jazyky, jež obsahují programátorské struktury, které nemají s fyzickou realizací počítačových systémů žádnou, nebo malou spojitost. Výhodou tohoto přístupu je snadnější pochopení a jednodušší realizace zamýšlených aplikací. Z důvodu malé, či žádné návaznosti současných přístupů k programování na architekturu počítače, je nutný překlad výsledného zdrojového kódu od programátora do jazyka, kterému je schopen porozumět počítač.

Při dnešní úrovni rozšíření informačních technologií jsou jasné tendence automatizace většiny úkonů nutných k vytvoření spustitelné aplikace tak, aby je mohl vykonávat kdokoli v co nejkratším čase. Proto byly vyvinuty aplikace typu YACC či Bizon, které podle zadaných parametrů mohou vygenerovat již hotový syntaktický analyzátor, jakožto hlavní součást moderních překladačů. Pro automatické generování lexikálních analyzátorů poslouží např. nástroj Flex.

Hlavním cílem mé práce je prozkoumat možnosti paralelního spojení několika metod syntaktické analýzy dohromady tak, aby se vzájemně vhodně doplnily a vytvořily jeden celek. Proto budu využívat jak metodu typu shora dolů, tak i metodu opačného typu, tzn. zdola nahoru. Mnou vybrané metody, rekurzivní sestup a precedenční analýza, se velimi dobře doplňují. Každá z těchto metod je velice efektivní pro určitou část zpracování kódu, a proto bude nutné mezi těmito metodami vhodně přepínat. Abych mohl aplikovat jednotlivé metody, je nutné mít jazyk, na který budou aplikovány. Proto se tato práce bude zabývat také analýzou rozkládaného jazyka.

Optimální využití obou výše zmíněných metod vyžaduje důkladné pochopení jejich funkce a zvládnutí teoretické části z oblasti formálních jazyků a překladačů. Lexikální analyzátor bude implementován manuálně, bez použití pomocných prostředků typu Flex. Výstupem samotného překladače pak bude program ve formě abstraktního syntaktického stromu, nebo jiné struktury pro tento účel vhodné. Nebude řešeno generování mezikódu a ani jeho případné optimalizace.

1 Základní pojmy

Tato kapitola obsahuje stručný přehled základních pojmů, které nám slouží k podrobnému popisu zkoumaného jazyka a jeho vlastností.

1.1 Abeceda a jazyk

Pro vymezení jazyka využiji některé základní pojmy, jako je abeceda a slovo.

Slovo je konečná posloupnost symbolů z určité abecedy, existuje i prázdné slovo. *Prázdné slovo neobsahuje žádné symboly, jeho posloupnost je prázdná [4].* V různých literaturách se můžeme setkat i s výrazem řetěz, popř. řetězec, význam těchto slov je ekvivalentní.

Abeceda je libovolná neprázdná konečná množina prvků, které nazýváme symboly abecedy [4]. V určitých případech je možné pracovat i s abecedami nekonečnými, ale zde se omezím pouze na abecedy konečné. Jako příklad abecedy lze uvést latinskou abecedu, která obsahuje 52 symbolů, které reprezentují velká a malá písmena.

Formálně lze abecedu *A* definovat takto [6]:

1. *prázdné slovo ε je slovo nad abecedou A*
2. *je-li x slovo nad A a* $a \in A$*, pak xa je slovo nad A*
3. *y je slovo nad A, když, a jen když lze y získat aplikací pravidel 1. a 2.*

Příklad: je-li $A = \{a, b, c\}$ abeceda, pak

a, *b*, *c*, *aa*, *ab*, *ac*, *acb*, *abc* – jsou některá slova nad abecedou *A*. Pořadí jednotlivých symbolů ve slově je významné, *ab* a *ba* jsou dvě rozdílná slova.

Jazyk je podmnožinou množiny všech řetězců nad abecedou *A*. Pak množinu *L* nazveme jazykem nad abecedou *A*. Jazykem tedy může být libovolná podmnožina slov nad danou abecedou [4].

Pokud budeme uvažovat abecedu jazyka Pascal, pak jazyk Pascal je nekonečná množina slov (programů) nad jeho abecedou.

Např.: *program P; begin end.*

Operace, které můžeme definovat nad jazyky, lze rozdělit do dvou základních skupin. První skupina jsou obvyklé množinové operace, jelikož je jazyk množina, můžeme tyto operace definovat. Můžeme mluvit o sjednocení, průniku, rozdílu a komplementu jednotlivých jazyků. Druhou skupinou jsou operace respektující specifikum množin tvořící jazyky, tj. skutečnost, že jejich prvky jsou slova. Pak definujeme operaci součinu a iterace řetězců.

Operace konkatenace řetězců je definována:

Nechť x a y jsou dva řetězce nad abecedou A. Konkatenace x a y je řetězec xy [6].

Operace součinu je definována pomocí konkatenace řetězců a má stejné vlastnosti jako konkatenace řetězců – je asociativní a nekomutativní. Pomocí součinu jazyka se sebou samým můžeme definovat důležitou operaci nad jazykem – iteraci jazyka.

Iterací jazyka L budeme označovat L^* a definovat [4]

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

1.2 Gramatika

Programovací jazyky musí být definovány přesně. Řádná specifikace programovacího jazyka obsahuje definice:

- Množinu symbolů (abecedu), které jsou využívány při konstrukci správných programů
- Množinu všech syntakticky správných programů
- Význam všech syntakticky správných programů

Jazyk obsahuje konečnou, nebo nekonečnou množinu vět. Konečné jazyky mohou být reprezentovány konečným výčtem všech vět, které mohou být vytvořeny. Naproti tomu pro nekonečné jazyky tento konečný výčet není možné vytvořit, ale jakákoliv specifikace významu jazyka by měla být konečná. Jeden model specifikace, která uspokojí naše požadavky specifikace, se jmenuje gramatika. Gramatika obsahuje konečnou neprázdnou množinu pravidel, které určují syntaxi jazyka. Gramatika určuje strukturu věty. Studium gramatik představuje velice zajímavou část oblasti formálních jazyků. Tuto oblast vytvořil Noam Chomsky [6], který gramatice dal matematický koncept definice ve spojení s jeho studiem přirozených jazyků.

Obecná definice gramatiky:

Generativní gramatika je čtveřice $G = (N, T, P, S)$ [6]

- *N je abeceda*
- *T je abeceda,* $N \cap T = \emptyset$
- $P \subset (N \cup T) \cdot N(N \cup T) \times (N \cup T)^*$ *je konečná množina*
- $S \in N$

N – abeceda nonterminálů, T – abeceda terminálů, P – množina pravidel, S – tzv. počáteční symbol

Pravidlo *(x, y)* P obvykle zapisujeme: $x \to y$.

Chomsky klasifikoval čtyři typy gramatik [6]:

Typ 0 – výše uvedená gramatika je gramatika typu 0, obsahuje pravidla v nejobecnějším tvaru a z toho důvodu se také nazývá gramatikou neomezenou.

Typ 1 – buď $G = \{N, T, P, S\}$ gramatika typu 0 a pro každé pravidlo $v \to w$ z P platí $|v| \leq |w|$ jedinou výjimkou může být pravidlo $S \to \varepsilon$, jehož výskyt však implikuje, že se S nevyskytuje na pravé straně žádného pravidla. Pak tuto gramatiku G nazýváme gramatika typu 1, neboli kontextová gramatika.

Typ 2 – buď $G = \{N, T, P, S\}$ gramatika typu 0 a pro každé pravidlo $v \to w$ z P implikuje $v \in N$. Pak G nazýváme gramatikou typu 2, neboli bezkontextová gramatika, protože substituci levé strany pravidla můžeme provádět bez ohledu na kontext, ve kterém je nonterminál uložen. Na rozdíl od kontextových mohou bezkontextové gramatiky obsahovat pravidla typu $A \to \varepsilon$. Každou bezkontextovou gramatiku lze dále transformovat tak, že obsahuje nejvýše jedno pravidlo s prázdným řetězcem na levé straně tvaru $S \to \varepsilon$. V takovém případě se, stejně jako u kontextových gramatik, nesmí výchozí symbol S objevit na pravé straně přepisovacího pravidla gramatiky.

Typ 3 – buď $G = \{N, T, P, S\}$ gramatika typu 0 a pro každé pravidlo z P je buď tvaru $A \to xB$, nebo $A \to x$, kde $A, B \in N, x \in \Sigma^*$. Pak G nazýváme gramatikou typu 3, neboli pravá lineární gramatika.

Máme-li gramatiku typu $i, i = 0,1,2,3$, pak jazyk, který tato gramatika generuje, nazýváme jazyk typu i. Podobně, jak tomu bylo u gramatik, pokud jde o jazyk typu 2, jedná se o bezkontextový jazyk a u jazyka typu 3 se jedná o regulární jazyk.

Nechť $L_i, i = 0,1,2,3$ značí třídu všech jazyků typu i. Pak platí $L_0 \supseteq L_1 \supseteq L_2 \supseteq L_3$ [6].

Z definice gramatiky typu i vyplývá, že každá gramatika typu 1 je zároveň gramatikou typu 0 a každá gramatika typu 3 je zároveň bezkontextovou gramatikou. Dále se budu zabývat gramatikami bezkontextovými a regulárními, jelikož tyto dva typy gramatik mají pro definici programovacích jazyků velký význam. S těmito gramatikami úzce souvisí dva typy automatů, a to konečný a zásobníkový.

L je bezkontextový jazyk, právě když jej lze definovat vhodným zásobníkovým automatem [4]

L je regulární jazyk, právě když jej lze definovat vhodným konečným automatem [4]

Tyto automaty tvoří základy prostředků, pomocí kterých definujeme hlavní části kompilátorů (jedná se hlavně o lexikální a syntaktický analyzátor).

1.3 Konečné automaty, regulární výrazy a regulární gramatiky

V předchozí kapitole jsem uvedl některé prostředky pro specifikaci libovolného regulárního jazyka, zde bych je chtěl postupně probrat a ukázat jejich těsnou souvislost.

1.3.1 Konečné automaty

Nedeterministický konečný automat je pětice [6] M = (Q,Σ,δ,q_0,F) kde

1. *Q je konečná množina stavů*
2. *Σ je konečná vstupní abeceda*
3. *δ je zobrazení* $Q \times \Sigma \rightarrow 2^Q$ *(2^Q je množina podmnožin množiny Q)*
4. $q_0 \in Q$ *je počáteční stav*
5. $F \subseteq Q$ *je množina koncových stavů*

Zobrazení δ nazýváme funkcí přechodu, je-li δ: $Q \times \Sigma \rightarrow Q$*, pak automat M je deterministický konečný automat [6].*

Konečné automaty jsou pro konstrukci překladače velice důležité, neboť tvoří základ tzv. konečných převodníků, pomocí kterých je realizována první fáze překladu – lexikální analýza.

Neformálně řečeno, konečný automat je jako stroj, který sestává ze dvou částí, jedna část načítá jednotlivé znaky a druhá část vyhodnocuje další postup. Vždy, když automat začíná číst, nachází se výkonná část v počátečním stavu, po načtení jakéhokoliv symbolu je provedeno jeho vyhodnocení a výkonná část přejde do následujícího stavu. Slovo je přijatu, pokud jsou přečteny všechny symboly na pásce a zároveň je dosaženo koncového stavu. Jinak slovo přijato není.

1.3.2 Regulární výrazy

Lexikální symboly daného jazyka musí být tvořeny podle přísně daných pravidel. Např. v programovacím jazyce Pascal musí každý identifikátor začínat písmenem, řetězec znaků musí začínat a končit symbolem apostrof apod. Velmi jednoduchým nástrojem, který lze využít pro zadání těchto vlastností lexikálních symbolů jsou regulární výrazy.

Buď V abeceda. Regulární množina nad V je definována rekurzívně: [4]

1. *Ø je regulární množina nad V*
2. *{ε} je regulární množina nad V*
3. *jestliže* $a \in V$*, pak* $\{a\}$ *je regulární množina nad V*
4. *jestliže* L_1, L_2 *jsou regulární množiny nad V, pak:*
   - $L_1 \cup L_2$
   - $L_1 L_2$
   - L_1^*

   jsou regulární množiny nad V
5. *žádná jiná regulární množina než vytvořená podle 1-4 nad V neexistuje*

Buď V abeceda. Regulární výraz R nad V je definován rekurzívně: [4]

1. *Ø je regulární výraz označující regulární množinu Ø*
2. *ε je regulární výraz označující regulární množinu {ε}*
3. *jestliže* $a \in v$, *pak a je regulární výraz označující regulární množinu* $\{a\}$
4. *jestliže R_1, R_2 jsou regulární výrazy označující regulární množiny L_1, L_2 pak:*
   - $(R_1 + R_2)$ *je regulární výraz označující regulární množinu* $L_1 \cup L_2$
   - $(R_1 R_2)$ *je výraz označující regulární množinu* $L_1 L_2$
   - R_1^* *je regulární výraz označující regulární množinu* L_1^*
5. *žádné jiné regulární výrazy, než vytvořené podle 1-4 nad abecedou V neexistují*

Označuje-li regulární výraz *R* regulární množinu *L,* pak říkáme, že *L* je reprezentován regulárním výrazem *R*.

Pro každou regulární množinu lze nalézt vhodný regulární výraz, který ji reprezentuje, a také naopak platí, že každý regulární výraz reprezentuje nějakou regulární množinu.

1.3.3 Regulární gramatiky

Pro každou regulární gramatiku *G* existuje nedeterministický konečný automat *A* tak, že $L(G) = L(A)$ a naopak [4].

2 Základní části překladače

Zde představím základní principy konstrukce jednotlivých částí překladače. Přeložení zdrojového kódu na funkčně ekvivalentní spustitelný program je komplexní záležitostí většinou několika spolupracujících částí překladače. Konstrukce překladače je často jednodušší, pokud je analýza zdrojového souboru rozdělena na dvě části, kdy jedna část provádí analýzu na nejnižší úrovni jazykových konstrukcí, identifikuje např.: rezervovaná slova, jména proměnných, operátory, a druhá provádějící syntaktickou analýzu.

2.1 Lexikální analýza

Lexikální analyzátor provádí prvotní a nejnižší analýzu zdrojového textu, představuje jakéhosi prostředníka mezi zdrojovým textem programu a syntaktickým analyzátorem. Prochází celý soubor znak po znaku a rozděluje části textu na tzv. *token* – y. Tyto tokeny představují jednotlivá jména proměnných, operátory atd., které obsahuje zdrojový text programu.

Lexikální analyzátor je většinou při analýze zdrojového textu využíván jedním ze dvou možných způsobů. Jedná se o možnost, kdy analyzátor zpracuje celý vstupní soubor ještě před spuštěním syntaktického analyzátoru, následně jsou všechny tokeny uloženy do souboru nebo do velké tabulky. Další možností využití je větší propojení těchto dvou analyzátorů tak, že syntaktický analyzátor volá lexikální analyzátor v případě potřeby dalšího tokenu, tato možnost propojení analyzátorů je v praxi více rozšířená, jelikož neklade tak velké paměťové nároky. Další její výhodou je možnost napsat více různých lexikálních analyzátorů pro stejný jazyk.

Jak bylo zmíněno výše, lexikální analyzátor rozdělí zdrojový text programu na jednotlivé tokeny. Typy tokenů jsou obvykle reprezentovány unikátním celým číslem. Např. proměnná může být reprezentována číslem 1, konstanta číslem 2, operátor + číslem 3 atd. Tokeny, které jsou tvořeny řetězcem, jsou obvykle ukládány do samostatné tabulky, stejně jako jednotlivé konstanty nebo proměnné. Ne všechny typy tokenů jsou ukládány do tabulek, např. tokeny operátorů ukládány nejsou.

Aby byl lexikální analyzátor schopný navracet jednotlivé tokeny, musí být schopný rozdělit jednotlivé sekvence znaků jdoucích za sebou ve zdrojovém textu programu, které mohou reprezentovat navrhnutý a správný token. Aby toto mohl analyzátor dokázat, musí být schopný odstranit ze vstupního souboru všechny znaky, které nejsou pro syntaktický analyzátor potřebné. Jedná se o všechny komentáře, mezery, prázdné řádky atp. Lexikální analyzátor musí správně identifikovat kompletní token. Občas si mohou být tokeny velice podobné a analyzátor se musí rozhodnout, zda má načítat další znaky ze vstupního souboru, nebo již objevil kompletní a správný

token. Např. pokud analyzátor načte znak :, ale token může nabývat i hodnoty :=, musí analyzátor načíst i další znak a provést kontrolu, zda se další znak shoduje se znakem = nebo ne, pokud ne, vrátí token odpovídající hodnotě : a znak načtený navíc musí vrátit zpět. Rozpoznání tokenu se nejčastěji provádí matematickým modelem, který se nazývá *konečný automat*.

Fyzicky je většinou lexikální analyzátor realizován pomocí jedné funkce, která navrací token v definované struktuře.

2.2 Syntaktická analýza

Druhou fází překladu je syntaktická analýza, touto analýzou se transformuje vstupní text do datové struktury, většinou se jedná o tzv. *derivační strom*, jenž je vhodný pro další zpracování a zachovává hierarchii vstupních dat. Úkolem syntaktického analyzátoru je zjistit, zda je možné zdrojový kód programu vygenerovat z počátečního symbolu, který je definován v gramatice. Syntaktický analyzátor je hlavní částí moderního překladače, proto se také tato metoda nazývá syntaxí řízený překlad. Lze rozlišit dva principy funkčnosti syntaktického analyzátoru. Jedná se o běžný sekvenční princip a paralelní princip. Oba tyto principy mají možnost využívat stejné typy metod syntaktické analýzy, které se dělí na analýzu shora dolů a analýzu zdola nahoru.

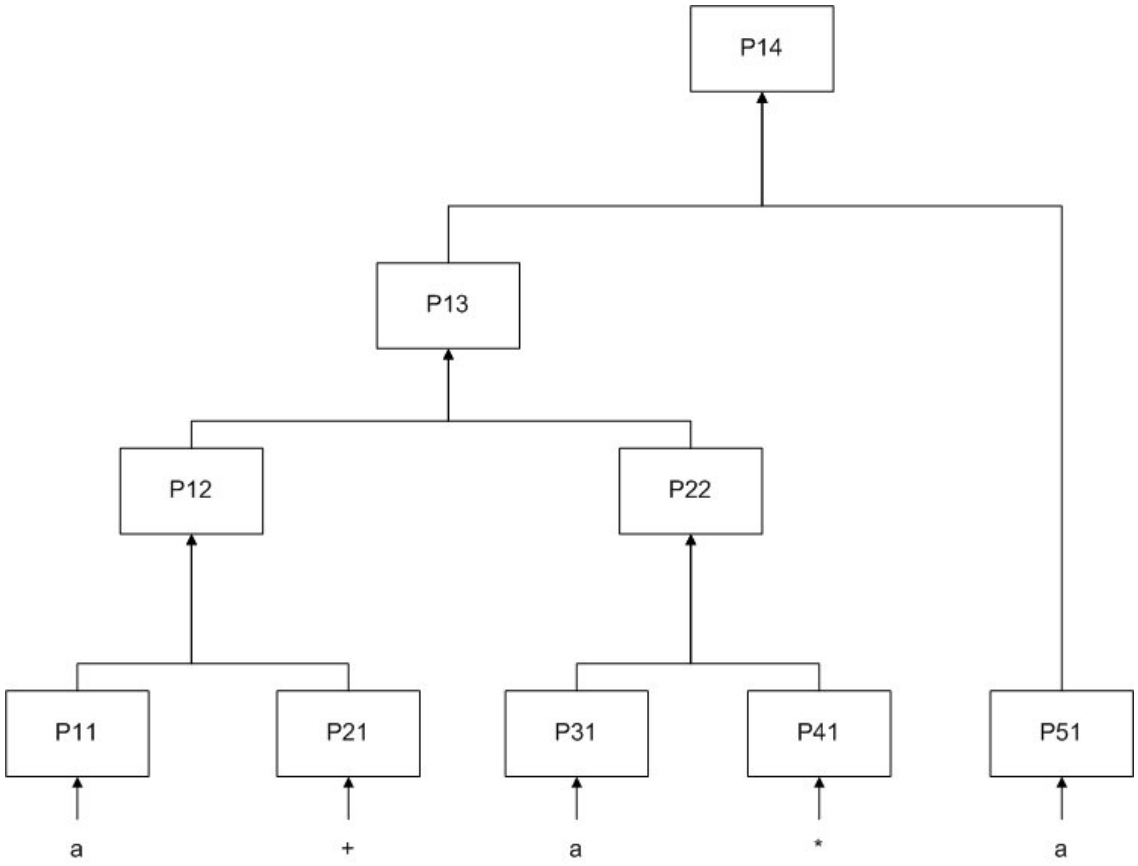
2.2.1 Paralelní syntaktická analýza

Veliký potenciál paralelní syntaktické analýzy je v úspoře času, který je nutné strávit při analýze zdrojového textu. Pokud ale jsou jednotlivé operace vysoce závislé na ostatních, pak ani lepší hardware nebo paralelizační techniky nemohou zajistit, aby analýza proběhla rychleji.

Při paralelní syntaktické analýze je nezbytné rozdělit zdrojový program na jednotlivé podřetězce, které budou následně vyhodnoceny jednotlivými procesory. Nejkratším podřetězcem je jeden symbol.

Např. pro paralelní syntaktickou analýzu zdrojového řetězce $a + a \cdot a$ využijeme síť paralelně propojených procesorů zobrazených na obrázku 2.2.1-1. Nejprve procesory *P11,P21,P31,P41,P51* současně zpracují své vstupní symboly. Následně procesory *P12,22* přijmou výsledky analýzy od procesorů *P11,P21* resp. *P31,P41* a pokračují v analýze jednotlivých částí zdrojového řetězce. Výsledky analýzy předají procesoru *P13*, který po dokončení své části analýzy předává výsledky, společně s procesorem *P51*, procesoru *P14*, který analýzu dokončí.

Tento ideální algoritmus paralelní syntaktické analýzy je založen na znalosti kontextu. Ve skutečnosti ale tyto informace má analyzátor k dispozici až po dokončení jednotlivých analýz.



Obrázek 2.2.1-1 Paralelní syntaktická analýza

2.2.2 Analýza shora dolů

Buď $G = (N, T, P, S)$ *bezkontextová gramatika, která má n pravidel očíslovaných 1,…,n a nechť* $w \in L(G)$. *Syntaktická analýza metodou shora dolů je proces, který vede k nalezení posloupnosti čísel pravidel užitých při levé derivaci věty w [4].*

Pro analýzu shora dolů je možné využít jednu ze tří metod analýzy zdrojového textu. Tyto metody jsou - metoda hrubé síly, metoda rekurzivního sestupu a metoda s omezenou délkou zásobníku.

Metoda hrubé síly prochází analyzovaný text od počátečního symbolu analýzy směrem k terminálním symbolům. Analýza se snaží projít od kořenů stromové struktury k jednotlivým listům stromu. Při využití úplného zásobníku se analyzátor snaží vytvořit derivační strom procházením jednotlivých větví, dokud nedosáhne správného terminálního symbolu. Nejhorší situace při provádění analýzy touto metodou nastává při analýze řetězce, který nenáleží pro aktuálně analyzovaný jazyk, jelikož před tím než je analýza prohlášena za chybnou, projde analyzátor všechny možné kombinace, které by mohly nastat.

Obecně tato metoda pracuje následovně:

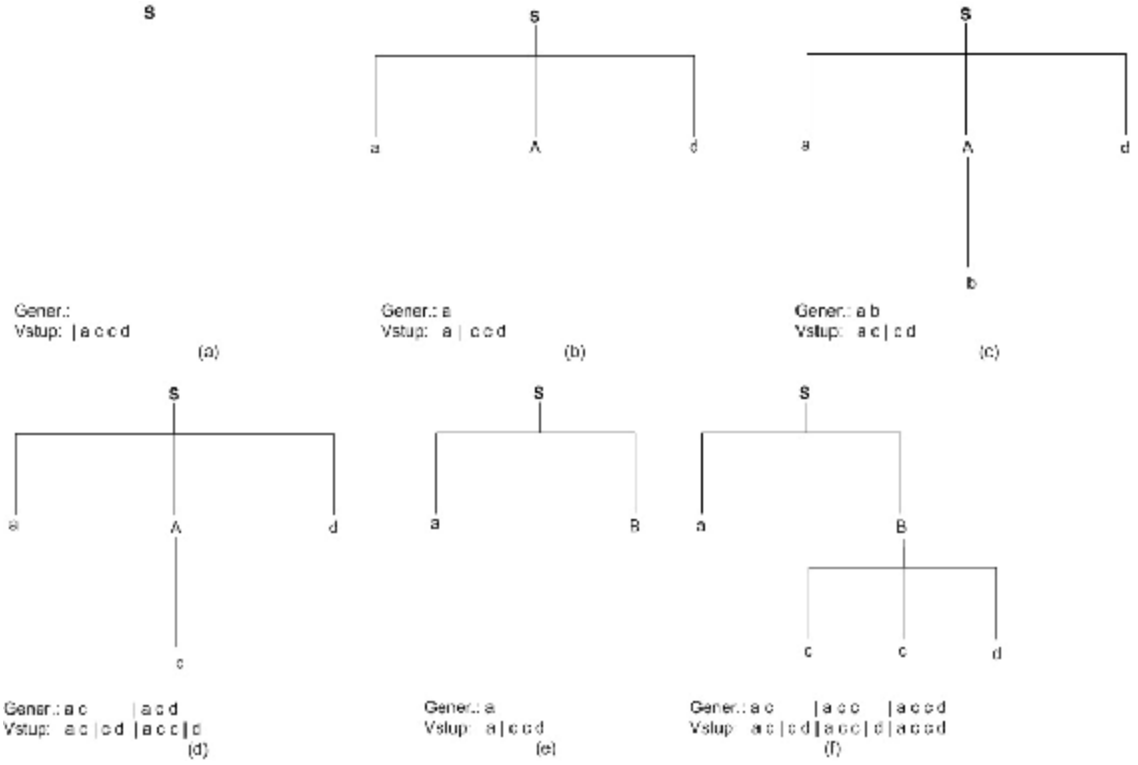
1. Po přijetí jednoho nonterminálu, který je nutné rozšířit, je na tento nonterminál aplikováno první možné pravidlo.
2. Následně je vyhodnocen rozšířený řetězec, po nalezení nejlevějšího nonterminálu je tento nově nalezený nonterminál rozšířen prvním možným pravidlem.
3. Tento proces rozšiřování a aplikace pravidel je aplikován na všechny nonterminály, dokud bude schopen analyzátor pokračovat. Analyzátor skončí pouze, pokud by nastal jeden ze dvou případů. V první případě již nejsou k dispozici žádné nonterminály, což znamená, že byl načten celý zdrojový text programu a analýza proběhla úspěšně. Ve druhém případě nastane chyba při pokusu o rozšíření podřetězce a dochází k chybě. Analyzátor postupně ruší provedená rozšiřování až do místa, kde může využít jiné pravidlo pro rozšíření, pak analýza pokračuje dále.

Tento princip je aplikován na zdrojový text tak dlouho, dokud nejsou vyčerpány všechny možnosti kombinací použití jednotlivých pravidel pro rozšíření vstupního řetězce. Po vyzkoušení všech pravidel a jejich kombinací, kdy se analyzátor stále nedostal na konec úspěšné analýzy je analýza ukončena a vstupní řetězec je označen jako neanalyzovatelný pro zadaný jazyk.
Pro předvedení principu analýzy hrubou silou využijeme tuto jednoduchou gramatiku:

$S \to aAd \mid aB \qquad A \to b \mid c \qquad B \to ccd \mid ddc$

kdy S je cíl nebo startovací symbol. Obrázek 2.2.1–1 ilustruje průběh syntaktické analýzy hrubou silou vstupního řetězce ‘accd’.

Analýza začíná inicializací, kterou můžeme vidět na obrázku 2.2.2-1a, kde je zobrazen pouze počáteční nonterminál S. Dále musíme zvolit první pravidlo, které použijeme pro rozšíření počátečního nonterminálu. Na obrázku 2.2.2-1b je patrná volba pravidla $S \to aAd$, kdy se nám v této fázi shoduje první nonterminál pravidla s prvním načteným řetězcem. Dále zvolíme pravidlo pro rozšíření dalšího nonterminálu, který se aktuálně v derivačním stromě vyskytuje. Pro nonterminál A zvolíme první možné pravidlo $A \to b$, jak můžeme vidět na obrázku 2.2.2-1c . Po načtení dalšího vstupního řetězce je patrné, že jsou tyto terminály rozdílné, tudíž je nutné vrátit poslední rozšíření zpět a zvolit pravidlo jiné. Na obrázku 2.2.2-1d je použito pravidlo $A \to c$, zde jsou aktuálně načtené terminály opět shodné, ale již při dalším kroku analýzy dochází k neshodě. Proto musí analyzátor vrátit zpět všechna rozšíření nonterminálů až do místa, odkud bude možné pokračovat dalším ještě nepoužitým pravidlem. V našem případě musel analyzátor vrátit všechna rozšíření až na začátek analýzy a použít jiné pravidlo pro rozšíření inicializačního nonterminálu. Bylo použito pravidlo $S \to aB$, kdy první načtený řetězec a první terminál si odpovídají, viz obrázek 2.2.2-1e a přecházíme na hledání pravidla pro rozšíření nonterminálu B. Na obrázku 2.2.2-1f vidíme použití pravidla $B \to ccd$ a pokračujeme v načítání vstupních řetězců, kdy můžeme vidět, že si již načtené terminály a generované terminály podle pravidel odpovídají a analýza končí úspěchem.



Obrázek 2.2.2-1 Analýza „hrubou silou“

Metoda rekurzivního sestupu poskytuje lepší možnost analýzy zdrojového textu a není tak náročná na spotřebované zdroje, ale je také méně obecná než metoda analýzy hrubou silou. Tato metoda pro svou funkci nevyužívá explicitního zásobníku, ale používá rekurzivního zanoření. Metodu rekurzivního sestupu není možné použít na všechny bezkontextové gramatiky, lze je využít pouze pro analýzu tzv. LL gramatik.

Jednoduchá LL gramatika je taková gramatika, kde levou stranu tvoří právě jeden nonterminál a každá pravá strana začíná terminálním symbolem [6]. Navíc musí platit, že např. pro pravidla $A \rightarrow ...$ jsou počáteční symboly různé.

Obecná LL gramatika nemá omezení, ale musí pro ni existovat rozkladová tabulka. Rozkladová tabulka nahrazuje přechodovou funkci. Pro zjištění, které pravidlo má být aktuálně použito, je využíván vstup a aktuální pozice analyzátoru v průběhu analýzy. Metoda rekurzivního sestupu je realizována pomocí sekvence volání procedur, kdy pro každý nonterminál je napsána samostatná procedura, která navrací hodnoty v rozsahu pravda, nepravda, podle úspěšnosti analýzy načteného řetězce. Tělo procedur je dáno pravými stranami jednotlivých pravidel pro zadaný nonterminál. Terminální symboly jsou reprezentovány testem na svůj výskyt ve vstupu následovaným čtením dalšího symbolu. Celý překlad začíná voláním funkce, která reprezentuje startovací nonterminál.

Pro příklad použití metody rekurzivního zanoření použiji následující gramatiku:

1) $S \rightarrow dSA$

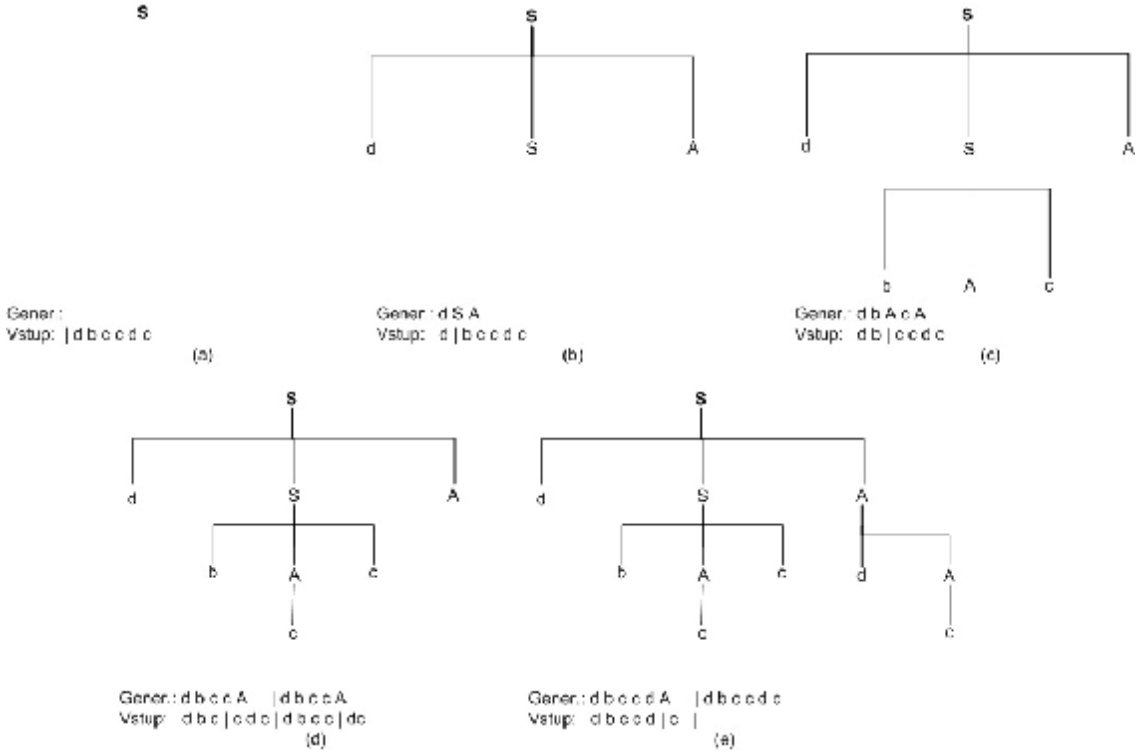
2) $S \rightarrow bAc$

3) $A \rightarrow dA$

4) $A \rightarrow c$

Vstupním řetězcem je $dbccdc$.

Analýza začíná inicializací na nonterminál *S*. Následně volíme pravidlo pro rozšíření počátečního symbolu *S*, toto pravidlo volíme s ohledem na načtený terminální symbol. V tomto případě volíme pravidlo číslo 1 a počáteční symbol rozšíříme pravou stranou pravidla. Následně hledáme nejlevější nonterminál, který budeme dále rozšiřovat. Nalezli jsme nonterminál *S* a je načten terminál *b*, proto aplikujeme pravidlo číslo 2. Aktuálně posledním nonterminálem je *A* a načteným terminálem je *c*, aplikujeme pravidlo číslo 4. V následujícím čtení odpovídá načtený terminál terminálu, který byl vygenerován pomocí pravidel gramatiky. Posledním nonterminálem je opět nonterminál A, ale načteným terminálem je nyní *d*, a proto aplikujeme pravidlo číslo 3. Následně opět rozšiřujeme nonterminál *A*, kdy načtený terminál je *c* a aplikujeme pravidlo číslo 3. Vygenerovaný řetězec odpovídá řetězci načtenému a zavolaná procedura navrátí hodnotu pravda.



Obrázek 2.2.2-2 Analýza rekurzivním sestupem

Předpokládejme, že máme následující pravidla $A \to \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$. V případě použití metody, která pro analýzu využívá neomezený zásobník, postupně vyzkoušíme analyzovat text všemi dostupnými pravidly, dokud nedospějeme k úspěchu nebo k chybě. Při analýze pomocí metody s omezenou délkou zásobníku bychom s tímto principem neuspěli.

Analýza s omezenou délkou zásobníku je obecnější než technika rekurzivního sestupu, ale je méně obecná než analýza hrubou silou, která má možnost plného využití zásobníku.

2.2.3 Analýza zdola nahoru

Při syntaktické analýze zdola nahoru začínáme derivační strom budovat od koncových uzlů a postupnými přímými redukcemi dojdeme ke kořenu (výchozímu symbolu gramatiky) [6].
Tento typ syntaktické analýzy je velmi úspěšně využíván při analýze výrazů. Precedenční analýza je pro tento účel velice výhodná.

K této metodě neodmyslitelně patří způsob analýzy zleva doprava, během něhož metoda neuchovává své jednotlivé kroky, které vedly k úspěšné nebo neúspěšné analýze vstupního výrazu. Cílem metody je nalezení předchůdců a odpovídajících vazeb k danému prvku, kdy předchůdci k danému prvku leží na cestách vedoucích do daného prvku. Pro tento účel je nutné sestrojit tabulku, podle které se budou řídit jednotlivá rozšíření aktuálně analyzované části výrazu.

V tabulce, která řídí precedenční analýzu, se vyskytují tři typy symbolů, jedná se o '=' , '<' a '>', které značí akce pro redukci a načtení dalšího vstupního symbolu. Tuto metodu je možné použít na třídu bezkontextových gramatik zvanou jednoduché precedenční gramatiky.

Mějme jednoduchou gramatiku, kde počáteční symbol je *E*:

$E \to T \mid E + T \mid E - T$
$T \to F \mid T * F \mid T / F$
$F \to P \mid F \uparrow P$
$P \to i \mid (E)$

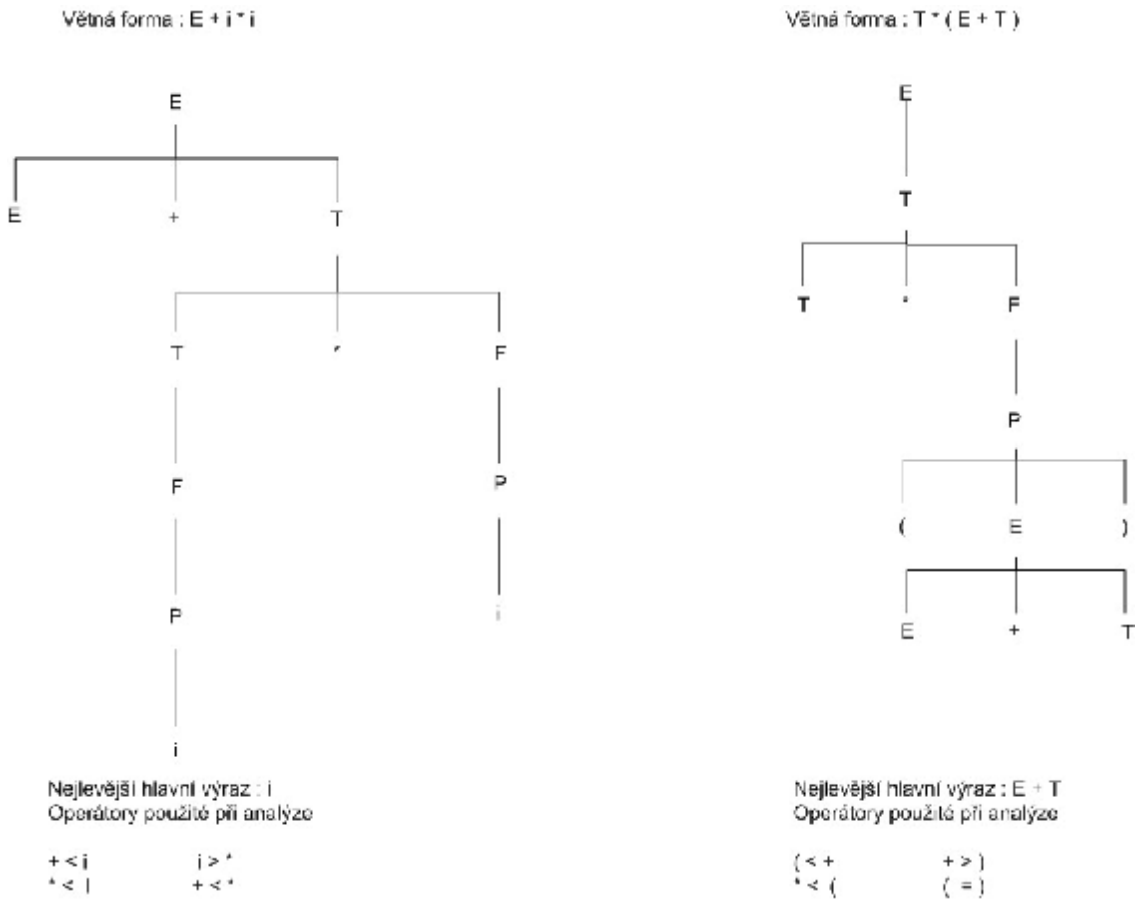
Jazyk spojen s touto gramatikou je množina aritmetických výrazů složená z operátorů pro sčítání, odečítání, násobení, dělení a umocňování. Hlavní výraz je výraz, který musí obsahovat nejméně jeden terminální symbol z gramatiky daného jazyka, který ale není jiným hlavním výrazem. Např. pro výše uvedenou jednoduchou gramatiku obsahuje výraz $P \cdot P/(i + T)$ hlavní výraz $P \cdot P$ a *i*. Všechny ostatní části výrazu jsou nonterminály, nebo obsahují jeden z těchto hlavních výrazů. V každé fázi analýzy se snažíme redukovat ten nejlevější hlavní výraz, který můžeme nalézt v načtené části vstupního výrazu, což je také důvodem, proč je tato analýza klasifikována jako metoda zleva doprava.

Předchozí omezení gramatických pravidel pro gramatiku operátorů kladou důraz na důležitost vlastností větných forem. Zvláště může ukázat, že žádná z větných forem v operátorové gramatice

nemůže obsahovat dva po sobě jdoucí nonterminály. Náhled na vztahy mezi jednotlivými operátory nám poskytují syntaktické stromy, kde každý jednotlivý sloupec na obrázku 2.2.3-1 reprezentuje větnou formu, syntaktický strom pro ni, nejlevější hlavní výraz, vztahy při precedenční analýze. Např. nejlevější hlavní výraz vstupního výrazu $E + i \cdot i$ na obrázku 2.2.3-1 je i a vztahy při analýze jsou $i > *$ a $+ < i$. Vztahem $i > *$ je myšleno, že i bude redukováno předtím, než bude redukován operátor *, podobně i v případě $+ < i$ znamená, že i bude redukováno před +. Po provedení redukce i na P se i stává nejlevějším hlavním výrazem větné formy $E + T \cdot i$, na kterou bude dále aplikován vztah $* < i$. Pokračováním analýzy dále stejným způsobem zjistíme i ostatní vztahy mezi jednotlivými dvojicemi vstupního výrazu. Tabulku obsahující všechny vztahy využitelné v průběhu analýzy testovací gramatiky obsahuje tabulka 1, kde prázdná políčka značí, že pro daný vstupní výraz neexistuje žádný vztah.

	(	i	+	-	*	/	↑	)	$
(	<	<	<	<	<	<	<	=	
i			>	>	>	>	>	>	>
+	<	<	>	>	<	<	<	>	>
-	<	<	>	>	<	<	<	>	>
*	<	<	>	>	>	>	<	>	>
/	<	<	>	>	>	>	<	>	>
↑	<	<	>	>	>	>	>	>	>
)			>	>	>	>	>	>	>
$	<	<	<	<	<	<	<		

Tabulka 1



Obrázek 2.2.3-1 Syntaktické stromy při precedenční analýze

3 Návrh jazyka

Aby mohl být syntaktický analyzátor sestrojen a řádně otestován, je nutné navrhnout jednoduchý programovací jazyk. Pro tyto účely byl vytvořen jednoduchý jazyk pascalovského typu jménem SPL, což je zkratka slov Simple Pascal Language. V následujících odstavcích bude představena lexikální a syntaktická struktura navrhnutého jazyka, částečně bude popsána i jeho sémantika, bez které by nebyl návrh kompletní. SPL patří do skupiny imperativních jazyků stejně jako např. Pascal nebo jazyk C.

3.1 Lexikální struktura

Identifikátory

Identifikátor je taková sekvence znaků, která vyhovuje regulárnímu výrazu

"$[a-zA-Z][0-9a-zA-Z]^*$" Identifikátor tedy nesmí začínat číslicí!

Čísla

SPL rozeznává tři možné způsoby zápisu čísel, jedná se o

- Celá čísla – "$[0-9]^*$"
- Desetinná čísla - "$[0-9]^*$ \ .$[0-9]^*$"
- Čísla v exponenciálním tvaru - " $[0-9]^*[e,E][0-9]^+$ "

Klíčová slova

Množina klíčových slov tvoří podmnožinu identifikátorů, jejichž použití jako identifikátorů je však zakázáno. V textových řetězcích se vyskytovat mohou.

begin
end
end.
if
then
else
read
write
do
repeat
until
true
false
var

Jako klíčová slova jsou považovány i jednotlivé typy.

int
float
string
char
bool

Operátory, oddělovače, závorky

Překladač je schopen rozeznat následující znaky jako operátory, oddělovač nebo závorky. Všechny operátory jsou binární.

- Operátory: + - * / && || !=
  < > <= >= =

- Operátor přiřazení: :=
- Oddělovače: : “
- Závorky: ( )

Komentáře

Pro komentování kódu nabízí jazyk SPL dvě možnosti

- // - komentář do konce řádku
- /* */ - vše co je uzavřeno mezi znaky /* a */ je považováno za komentář

Řetězce a znaky

Za znak, resp. řetězec je považována libovolná sekvence, která je uzavřená v uvozovkách. Uvnitř řetězců není prováděna žádná analýza, tudíž mohou obsahovat libovolné znaky.

3.2 Syntaktická a sémantická struktura

Nyní bude představena syntax jazyka SPL spolu s jeho sémantikou. Budou představeny jednotlivé řídicí konstrukce, které jazyk nabízí. Jelikož je jazyk SPL z rodiny imperativních jazyků, jednotlivé příkazy programu jsou vykonávány jeden za druhým tak, jak jsou uvedeny ve zdrojovém textu.

Proměnné

Jazyk SPL je silně staticky typovaný, což umožňuje především při sémantické analýze kontrolovat, zda nedochází např. k přiřazování nesprávných typů hodnot do jednotlivých proměnných.

Deklarace každé proměnné musí proběhnout zvlášť a musí se odehrát ještě před začátkem samotného programu. Pro deklaraci proměnné je využita tzv. Pascalovská notace, které předchází klíčové slovo *var*. Inicializace jednotlivých proměnných probíhá až v samotném programu.

Příklad deklarace: *var cislo : int; //deklarace celočíselné proměnné*

Typy

Jazyk rozlišuje následující typy:

- *int* – typ reprezentující celá čísla
- *float* – typ reprezentující desetinná čísla
- *char* – reprezentující jeden znak
- *string* – reprezentuje textový řetězec
- *bool* – reprezentuje logickou hodnotu, může nabývat pouze dvou hodnot *true/false*

Řídicí struktury

V jazyce SPL se vyskytují konstrukce s příkazy známé z ostatních klasických imperativních jazyků. Jedná se především o konstrukci *if – else*, dále pak cykly *while, repeat – until*. Za příkazem *if* a *while* následuje podmínka, za které se vykoná následující úsek kódu, cokoliv jiného než výraz je považováno za chybu.

Příklad *if – else*:

```
if i=1 then i:=i+1;
else begin
        i:=i-1;
        r:=2*i;
    end;
```

Příklad *while* a *repeat – until*:

```
while a > 0 then
        begin
            a:=a-1;
        end;

repeat
    a:=a+5;
until a<1000;
```

4 Reprezentace jazyka

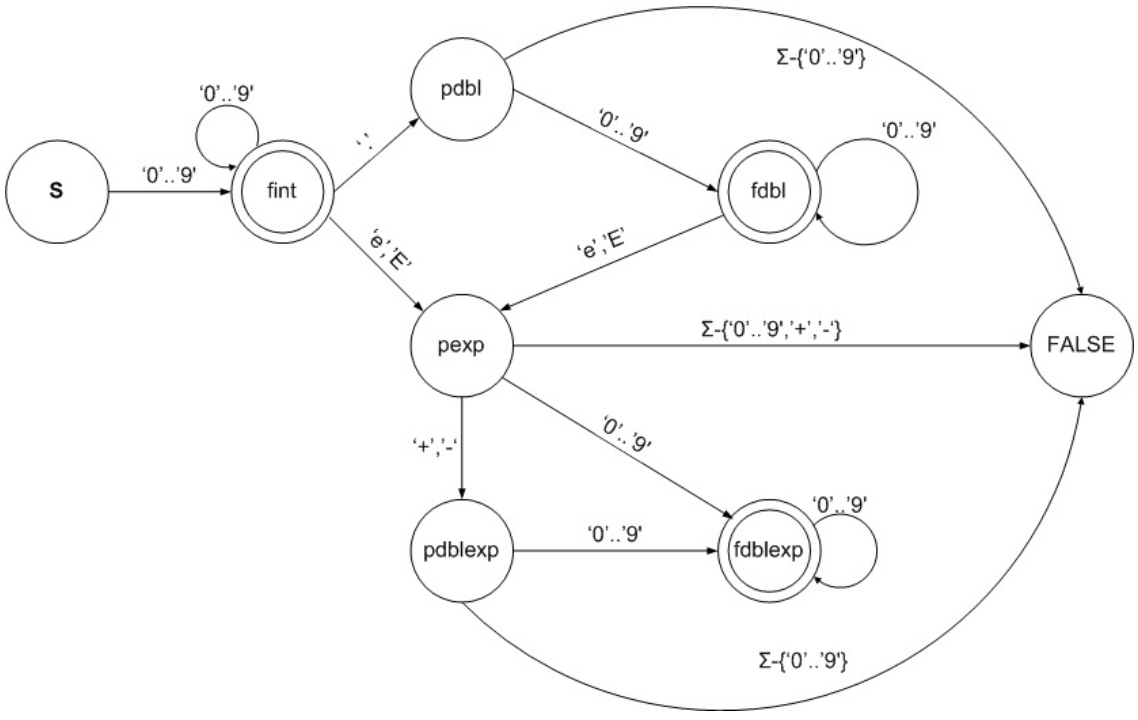
Tato kapitola se zabývá reprezentací zdrojového kódu v jednotlivých částech analyzátoru. Bude zde představena reprezentace vstupních symbolů pro lexikální analyzátor a také reprezentace dat pro syntaktický analyzátor.

4.1 Lexikální analýza

Lexikální analýzu zdrojového textu je možno vykonat dvěma způsoby. Pro tuto práci byl zvolen způsob, kdy lexikální analyzátor nejprve zanalyzuje celý zdrojový text a teprve po té začne

syntaktická analýza. Lexikální analyzátor čte postupně vstupní symboly zleva doprava, po načtení určitého, pro daný jazyk, správného řetězce je načítání ze vstupního souboru přerušeno a je načten tzv. *lexém*. Lexikální analyzátor musí určit typ lexému a tento lexém převést do tvaru, který je schopen zpracovat syntaktický analyzátor, což je tzv. *token*. Tento token je uložen a lexikální analyzátor pokračuje dále v načítání vstupních symbolů, dokud nepřečte celý zdrojový text.

Na obrázku 2.2.3-1 je možné vidět část grafu lexikálního analyzátoru, která se stará o analýzu číselných hodnot. Kompletní grafy lexikálního analyzátoru jsou uvedeny v příloze A. Po načtení číselného symbolu je proveden přechod z počátečního stavu *S* do následujícího stavu *fint*, kde se, pokud se na vstupu nachází další číselné symboly, pokračuje ve čtení, nebo může také načítání skončit. Pokud by se na vstupu dále objevil symbol tečky, přechází lexikální analyzátor do dalšího stavu – *pdbl*, ze kterého je možno přejít do koncového stavu *fdbl* po načtení dalšího číselného symbolu. Pokud lexikální analyzátor načetl cokoliv jiného než číselný symbol, přejde do stavu *FALSE* a analýza končí chybou. Ve stavu *fdbl* analyzátor akceptuje načtení dalších číselných symbolů, popř. může být načítání vstupních symbolů ukončeno a bude navrácen token s desetinným číslem, nebo po načtení symbolu pro *e*, nebo *E* bude analyzátor pokračovat do stavu *pexp* a očekávat další číselné symboly, pokud budou načteny, analyzátor přejde do stavu *fdblexp*, kde může skončit a navrátit token s desetinným číslem v exponenciálním tvaru, nebo může načítat další číselné symboly. Pokud ale ve stavu *pexp* je načten jakýkoliv jiný symbol než číselný, popř. symboly + nebo -, přechází analyzátor do stavu *FALSE* a končí neúspěchem. Pokud je ve stavu *fint* načten symbol pro *e*, nebo *E,* přejde analyzátor do stavu *pexp* a analýza pokračuje výše uvedeným způsobem.



Obrázek 2.2.3-1 Lexikální analyzátor – analýza čísel

4.1.1 Struktura tokenu

Token je tvořen strukturou, ve které je jednoznačně určen typ vstupního řetězce, tzn. rozlišuje, jedná-li se o proměnnou, operátor či část řídicí struktury jazyka. V některých případech je nutné uchovat i hodnotu, která je nutná pro další zpracování, hodnota může být typu čísla, řetězce nebo logické hodnoty. Je možné konstatovat, že lexikální analyzátor je schopen rozlišit několik základních typů lexému a komentáře:

- Proměnné
- Operátory
- Literály
- Komentáře

V případě načtení identifikátoru je okamžitě provedena kontrola, zda se nejedná o klíčové slovo jazyka SPL. Pokud ano, je nutné nastavit výstupnímu tokenu odpovídající typ, aby bylo jasné, o které klíčové slovo se jedná.

4.2 Syntaktická analýza

Syntaktický analyzátor tvoří základní část dnešních překladačů, proto lze také mluvit o syntaxí řízeném překladu. Pro syntaktickou analýzu zdrojového textu byla zvolena kombinace dvou metod. Analýzu hlavní části zdrojového textu, jako např. analýzu podmínek a cyklů, provádí metoda rekurzivního sestupu, která ale neprovádí analýzu výrazů, jež se ve zdrojovém textu objevují. Analýzu výrazů provádí metoda precedenční analýzy. Obě tyto metody pracují zároveň, čímž je možné dosáhnout vyššího výkonu při syntaktické analýze zdrojového textu, než kdyby byla použita pouze jedna z nich. Gramatika jazyka SPL, podle které je syntaktická analýza prováděna, je uvedena v příloze B.

4.2.1 Analýza shora dolů

Jak již bylo zmíněno hlavní metoda syntaktické analýzy je metoda rekurzivního sestupu, tato metoda potřebuje pro svou činnost gramatiku *LL1*. Ze startovacího nonterminálu začne pomocí přepisovacích pravidel generovat příslušnou větu programovacího jazyka. Pro určení pravidla, které bude použito při derivaci, je využíván předcházející token. Tato metoda řídí analýzu celého zdrojového textu.

První gramatické pravidlo využívané při překladu je následující: $FILE \rightarrow VARbeginBODYend.$ Určuje strukturu celého programu zapsaného v jazyce SPL, který musí začínat zápisem všech proměnných využívaných v programu, teprve pak následuje samotný program. Pravidla pro zápis jednotlivých proměnných jsou následující: $VAR \rightarrow \text{var}\, identifier : TYPE; VAR$ a $VAR \rightarrow \varepsilon$.

Pomocí těchto dvou pravidel je možné nadefinovat všechny proměnné, které jsou potřeba, typy jednotlivých proměnných mohou být následující: *int, float, string, bool*.

Dále následuje samotný program uvedený klíčovým slovem *begin* a ukončen klíčovým slovem *end.*, mezi těmito klíčovými slovy se nachází funkční kód programu. Jedná se o řídicí struktury programu, např. příkaz podmínky: $COMMAND \rightarrow if\ EXPRESSION\ then\ BODY\ ELSE$, kdy po načtení příslušného tokenu označujícího, že se dále bude nacházet podmínkový příkaz, tudíž analyzátor načte výsledek výrazu, který byl zpracován precedenčním analyzátorem a pokračuje dále v analýze. Nonterminál *BODY* může být rozšířen na jediný příkaz, nebo také na blok příkazů, které se vykonají v případě platnosti podmínky. Dále následuje nonterminál *ELSE*, který ale může být přepsán na ε pomocí pravidla $ELSE \rightarrow \varepsilon$, nebo může být nahrazen jedním, či blokem příkazů, které se vykonají v případě neplatnosti úvodní podmínky.

Metoda rekurzivního sestupu provádí analýzu všech těchto řídicích struktur a zbylých příkazů, které může podle definované gramatiky obsahovat jazyk SPL.

4.2.2 Analýza zdola nahoru

Druhou použitou metodou je precedenční syntaktická analýza, pro činnost této metody postačuje bezkontextová gramatika, levá rekurze zde není překážkou, spíše naopak. Díky ní je gramatika ucelená a přehledná. Tato metoda je při analýze zdrojového textu využívána pro analýzu výrazů. Metoda analyzuje jednotlivé výrazy a výsledek analýzy je uložen pro pozdější využití. Metoda precedenční syntaktické analýzy postupuje tak, že postupně načítá vstupní tokeny z výrazu a pomocí opačně aplikovaných přepisovacích pravidel se pokouší celý výraz zredukovat na příslušný nonterminál.

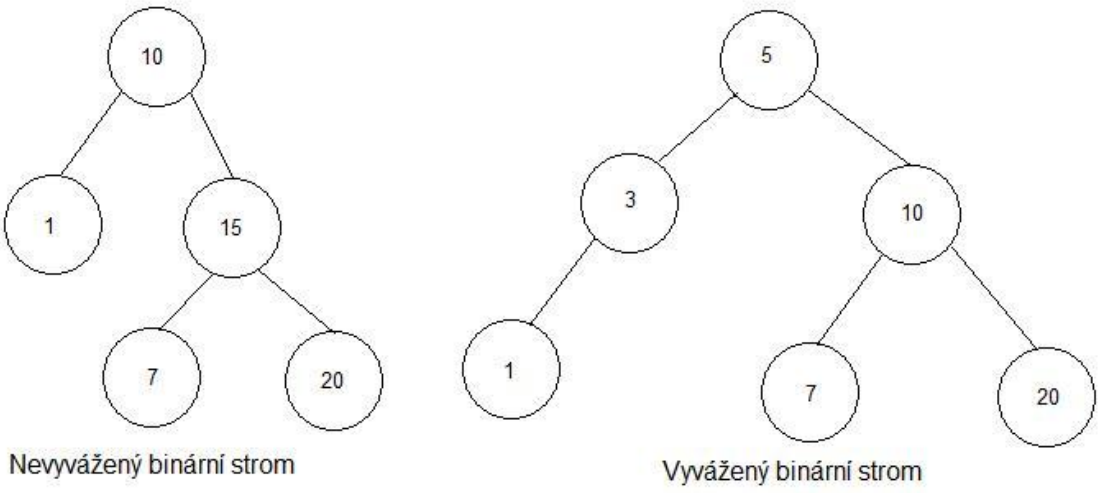
Pro správnou funkci je třeba sestavit korektní precedenční tabulku, pomocí které je analýza schopna vyhodnotit, kdy je potřeba aktuálně načtený vstup na zásobníku redukovat a kdy je potřeba načíst další vstupní symbol. Kompletní precedenční tabulka jazyka SPL je uvedena v příloze C.

Např. pokud je vstupní řetězec následující $i + i$, je po inicializaci metody zjištěn první token, což je *i* na zásobníku se nachází inicializační token *$*, při projití precedenční tabulky zjistíme, že je nutné vstupní token načíst a uložit na zásobník. Dalším tokenem v řadě je symbol +, na vrcholu zásobníku se nachází token *i*, dle tabulky dochází k redukci terminálu *i* na nonterminál. Nejpravějším terminálem na zásobníku je opět inicializační token, další token je stále +, nyní je token načten na zásobník. Na vstupu se nachází token *i* na vrcholu zásobníku je token +, dle tabulky dochází k načtení. Nyní byl přečten celý výraz, nejpravějším terminálem je *i*, řetězec výrazu již neobsahuje žádný token, dle tabulky dochází k redukci. Posledním terminálem na zásobníku, mimo inicializační, je token +, což podle tabulky vede k další redukci, čímž jsme zredukovali celý řetězec tokenů daného výrazu a analýza končí úspěchem.

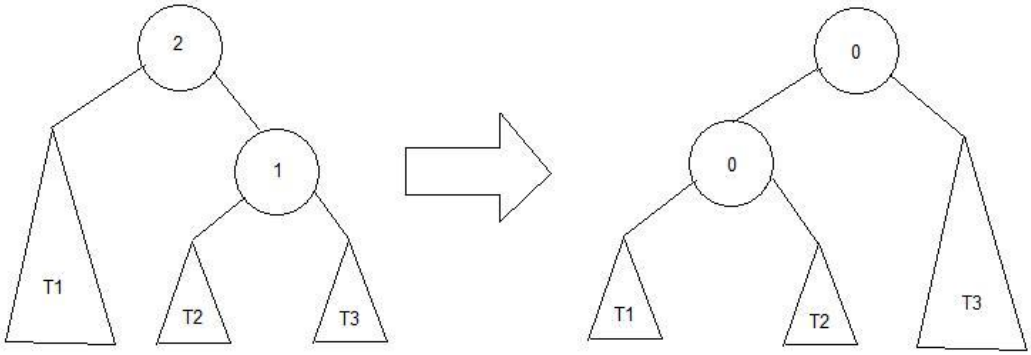
4.2.3 Tabulka symbolů

Tabulka symbolů je pro jakýkoliv překladač nepostradatelná. Tato tabulka je struktura, která v sobě uchovává informaci o všech proměnných, které byly deklarovány v daném vstupním programu. Pro tabulku symbolů byla použita abstraktní datová struktura jménem AVL strom. AVL strom je pojmenován podle svých vynálezců G.M. Adelson-Velsky a E.M. Landis, kteří jej poprvé publikovali roku 1962. V podstatě se jedná o samo vyrovnávací binární vyhledávací strom. V této struktuře se výška synovských podstromů může lišit maximálně o jedničku, tudíž může být tato struktura nazvána výškově vyrovnanou. Vyhledávání, vkládání či mazání z takovéto struktury má logaritmickou složitost [7].

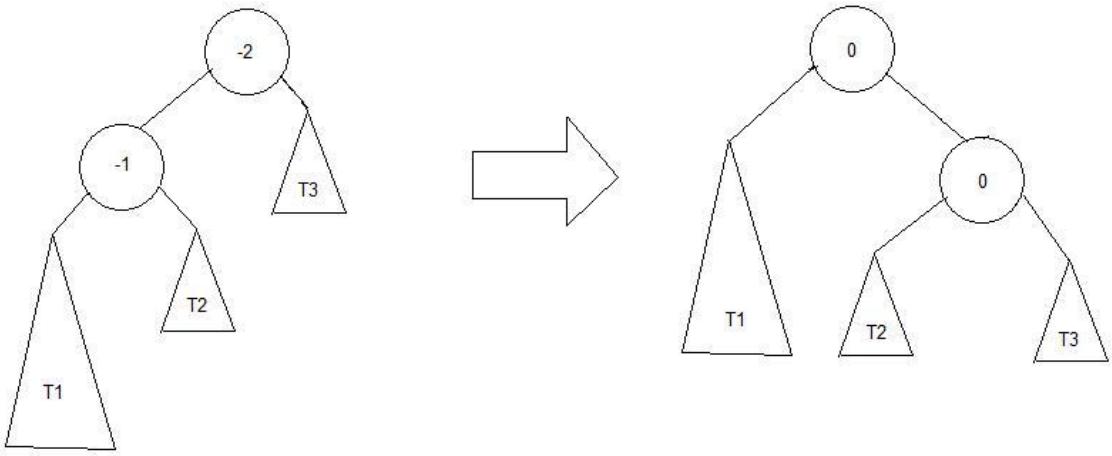
Následující obrázky ukazují, jak vypadá binární strom a následně AVL strom, kdy jsou předvedeny dvě jednoduché operace nad AVL stromem. Jedná se o jednoduchou levou a pravou rotaci, existují i složitější rotace, ale ty zde nebudou uvedeny.



Obrázek 4.2.3-1 Binární stromy



Obrázek 4.2.3-2 Jednoduchá levá rotace



Obrázek 4.2.3-3 Jednoduchá pravá rotace

Z výše uvedeného popisu jednoduchých rotací v AVL stromu vyplývá, jaké informace musí obsahovat struktura jednoho uzlu, který bude AVL strom obsahovat. Jelikož se jedná o binární vyhledávací strom, je nutné uchovávat ukazatele na levý i pravý podstrom, hodnotu, podle které bude možné v tomto stromu vyhledávat, kvůli nutnosti vyvažování stromu musí být uložena také výška jednotlivých podstromů a také samotná data pro jednotlivé proměnné.

Definice struktury jednoho uzlu:

```
typedef struct tNode{                  //  uzel stromu
  tIdent Cont;              // obsah uzlu
  int Test;                 // poznámková proměnná
  int Vyvazenost;             // hodnota vyváženosti prvku
  struct tNode * LPtr;      // ukazatel na levý podstrom
  struct tNode * RPtr;        // ukazatel na pravý podstrom
} *tNodePtr;
```

5 Implementace překladače

Po návrhu všech struktur, které bude překladač využívat, přichází na řadu samotná implementace překladače. Lexikální analyzátor bude implementován pomocí funkce, která při spuštění analyzátoru načte všechny lexémy na vstupu a převede je na odpovídající řetěz tokenů. Pro syntaktickou analýzu byly zvoleny dvě metody a to metoda rekurzivního sestupu a metoda precedenční analýzy. Po spuštění analyzátoru a načtení celého zdrojového textu budou následně spuštěny obě metody analýzy současně, přičemž každá z metod se zaměří na analýzu jiné části zdrojového textu.

5.1 Lexikální analyzátor

Lexikální analyzátor musí po zavolání načíst všechny vstupní lexémy, převést je na odpovídající tokeny. Tyto tokeny uložit do seznamu v takovém pořadí, ve kterém byly načteny. Pokud analyzátor při postupném načítání tokenů narazí na část vstupního textu, kde se nachází výraz, na místo v seznamu tokenů, kam náleží tento výraz, vloží tzv. pseudotoken, který bude označovat, že se na tomto místě nacházel výraz, který je zpracován odděleně. Výraz by měl následovat v případě, že byl načten token označující *if*, *while*, *until* nebo *:=*. Následně by měl být ukončen jedním z těchto tokenů: *;*, *then* nebo *do*. Tokeny tohoto výrazu jsou uloženy do dalšího seznamu, který obsahuje jednotlivé výrazy v pořadí, v jakém se nacházely ve zdrojovém textu. Dalším úkolem lexikálního analyzátoru je aktualizovat čítač řádků, který je využíván při hlášení chyb ve zdrojovém textu programu. Samotné načítání lexému probíhá dle přísně daných pravidel, které byly nadefinovány výše.

Lexikální analyzátor je implementován pomocí funkce, která má jako vstupní parametr ukazatel na token, do kterého bude ukládat aktuálně načtený token, a navrací hodnotu typu int, která ukazuje, zda analýza skončila chybou či nikoliv. Definice funkce lexikálního analyzátoru: *int getNextToken(tToken *token);*. Tuto funkci využívá pro svou činnost funkce, která zajistí načtení a uložení kompletního zdrojového textu ve formě tokenů.

Celý analyzátor obsahuje jeden cyklus, který skončí v případě přečtení celého vstupního souboru a *switch*, který rozlišuje jednotlivé stavy, ve kterých se aktuálně lexikální analyzátor nachází. Jednotlivé stavy jsou reprezentovány celým číslem, každý z těchto stavů obsahuje sérii podmínkových příkazů, dle kterých se rozhodne, zda je aktuálně načtený řetězec správný či nikoliv.

Pokud je např. na vstupu lexikálního analyzátoru znak pro menšítko (<), přechází analyzátor z úvodního stavu *S* do stavu *fmen*, kde musí otestovat další symbol na vstupu, zda se náhodou nejedná o symbol =., v případě, že je tento symbol načten, přechází analyzátor do stavu *fmrov*, kde nastaví token na hodnotu odpovídající vstupnímu symbolu menší nebo rovno a ukončí analýzu. Pokud se tento symbol na vstupu nenacházel, nastaví se token na hodnotu odpovídající tokenu rovná se a analýza se opět ukončí.

5.2 Syntaktický analyzátor – shora dolů

Pro syntaktický analyzátor shora dolů byla zvolena metoda rekurzivního sestupu, tato metoda tvoří hlavní část celého překladače. Pro každý nonterminál, který definuje gramatika jazyka SPL, je vytvořena funkce, ve které je ručně implementován jeho rozklad. Každá z těchto funkcí pak navrací celočíselnou hodnotu odpovídající buď úspěšné analýze nebo chybě. V každé funkci jsou porovnávány terminály s tokeny, které analyzátor získává ze seznamu tokenů, které byly načteny po spuštění. Pokud posloupnost vstupních tokenů odpovídá definici daného nonterminálu v gramatice, je analýza úspěšná.

Např. funkce pro analýzu deklarace proměnných před samotným programem *int Var(tToken *mtoken)* po svém zavolání ví, že již byl načten terminál *var*, který uvozuje deklaraci proměnné, dále si od pomocné funkce *int getNextTokenLex(tToken **token)* vyzvedne další token, který následuje v seznamu všech tokenu vstupního souboru. Tento token musí odpovídat identifikátoru, neboli musí být načteno jméno proměnné, kterou chceme deklarovat. Toto jméno se nesmí shodovat s definovanými klíčovými slovy jazyka SPL. Pokud byl načten identifikátor, pokračuje se v načítání dalšího tokenu, který musí odpovídat *':'*. Dalším tokenem v pořadí musí být typ deklarované proměnné, dle definice jazyka může proměnná nabývat 5 typů: *int, float, string, char* a *bool*, pokud by byl uveden jiný typ dochází k chybě. Posledním terminálem, který musí následovat, je ukončující *;*, který musí být zapsán na konci každého příkazu. Po analýze celé deklarace proměnné je nutné ještě prověřit, zda již nebyla zadána proměnná stejného jména, pokud není, je vygenerována nová struktura pro aktuálně zadanou proměnnou jazyka SPL.

Po dokončení analýzy deklarací proměnných následuje analýza samotného programu, který je uvozen klíčovým slovem *begin* a ukončen *end.*. „Tělo“ programu začíná analyzovat funkce *int MainBody()*, která spouští další jednotlivé funkce pro analýzu definovaných nonterminálů, které budou načteny ze zdrojového souboru. Tato funkce probíhá, dokud nejsou načteny všechny tokeny ze seznamu.

Příkazy, které se mohou v těle programu objevit, jsou: *if*, *read*, *write*, *while*, *repeat* a příkaz přiřazení hodnoty do proměnné. Funkce, která realizuje analýzu příkazu *if*, je mírně odlišná od ostatních, jelikož příkaz *if* může, ale nemusí, obsahovat i alternativní část kódu, která se vykoná v případě, že je podmínka vyhodnocena negativně.

Funkce *int Else()* je volána až v případě, že hlavní podmínkový příkaz obsahuje i pokračování pro případ negativního vyhodnocení podmínky v příkazu *if,* jinak tato funkce volána není. Potom, co analyzátor obdrží token, který určuje, že se bude vyhodnocovat podmínkový příkaz, je zavolána funkce *int If(tToken *tELSE)*, analýza začne zpracováním výsledků z analýzy výrazu. Dále za výrazem se musí nacházet token, který označuje *THEN*, jakožto začátek místa, od kterého následuje definice, co se bude vykonávat v případě kladného vyhodnocení podmínky. Může následovat jeden nebo více příkazů. V případě, že bude následovat více příkazů, je nutné, aby tyto příkazy byly

ohraničeny terminály *begin* a *else;*, v opačném případě nebudou další příkazy zahrnuty do podmínkového příkazu *if*. Následné zavolání funkcí pro analýzu, ať už následuje jeden nebo více příkazů, provádí funkce *int Body(tToken *token)*. Po ukončení analýzy funkce *body* je ukončena i funkce *If* a je nutné vyhodnotit token, který následuje po bloku příkazu *if*. Pokud se jedná o terminál *else,* bude zavolána funkce *int Else()*, která zajistí vyhodnocení následujících příkazů. Blok příkazu *else* může, stejně jako *if*, obsahovat jeden nebo více příkazů, jejichž vyhodnocení zajistí funkce *Body*.

5.3 Syntaktický analyzátor – zdola nahoru

Syntaktický analyzátor zdola nahoru je implementován pomocí metody precedenční analýzy. Ke své funkčnosti potřebuje tento analyzátor precedenční tabulku, která na základě aktuálního symbolu na zásobníku a aktuálního vstupního symbolu zvolí akci, která bude provedena. Analyzátor pracuje ve dvou režimech, buď načítá symboly na zásobník, nebo provádí jejich redukci dle gramatiky.

Mezi těmito režimy se postupně přepíná v průběhu analýzy. K jejich přepnutí dochází v okamžiku, kdy symbol, který je na zásobníku musí být redukován dříve než symbol, který je aktuálně na vstupu. Redukce na zásobníku probíhá tak dlouho, dokud tento stav platí. Ve chvíli kdy tento stav přestane platit, dojde k přepnutí stavu zpět do stavu načítání symbolů na zásobník. Redukce symbolů na zásobníku je postupně prováděna dokud na zásobníku nezůstane pouze startovací symbol gramatiky, pak je analýza hotova.

Mezi jednotlivými symboly gramatiky musí být zavedena relace precedence, která může nabývat jedné ze tří respektive čtyř hodnot, pokud se prázdné políčko bude považovat za stav.

- = - oba symboly budou redukovány současně, načtení na zásobník
- < - načítání pravé strany pravidla, načítání na zásobník
- > - byla nalezena hranice pravé strany pravidla, obsah zásobníku je nutné redukovat před tím, než bude symbol ze vstupu načten na zásobník
- ‘ ’ – prázdné místo, pro aktuální symbol na zásobníku a aktuální symbol na vstupu není definována žádná relace

Algoritmus pro redukci symbolů na zásobníku je následující:

push $ na zásobník
repeat
a je aktuální token
b je symbol na vrcholu zásobníku
case tabulka[b,a] of
= push(a) & read next a from jmout
< nahraď b za b< na zásobníku & push(a) & read next a from input

> if <y na zásobníku na vrcholu and r: $A \rightarrow y \in P$ then přepiš <y na A

& zapiš r na výstup else ERROR

‘ ’ ERROR

until a=$ and b=$

Na dně zásobníku se nachází symbol *$*, který také symbolizuje konec vstupu, relace pro tyto symboly je $\$ < n$ a $\$ > n$ pro všechna $n \in T$.

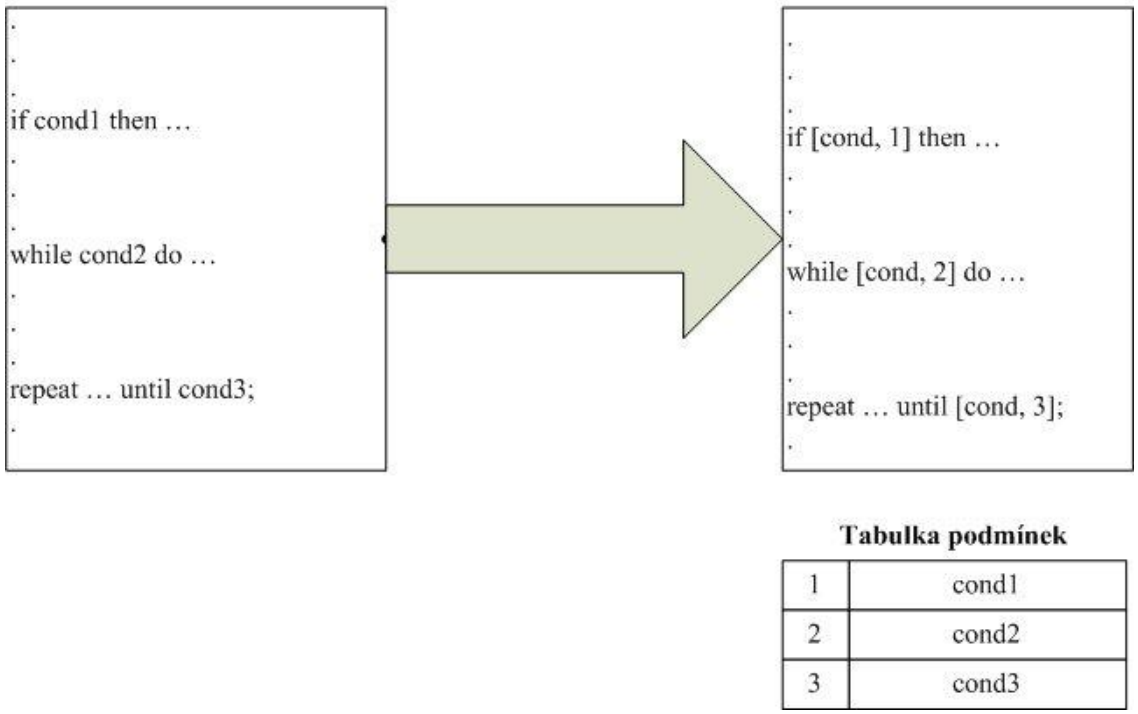
Díky této podmínce je zajištěno, že všechny symboly na zásobníku budou redukovány v případě, že již nejsou na vstupu žádné symboly a hranice se nachází nejpozději na dně zásobníku. Prázdná místa v precedenční tabulce značí syntaktickou chybu a lze je jednoduše využít pro chybová hlášení, stejně tak i situace, kdy nemáme žádné, popř. více pravidel. Precedenční tabulka pro analyzátor jazyka SPL je uvedena v příloze C.

Základní funkcí, která je zavolána v případě, že je potřeba analyzovat výraz, je funkce *int Vyraz(tIdent ** vysled,tToken *ztoken)*, která inicializuje zásobníkovou strukturu, do které se budou ukládat jednotlivé načtené symboly, inicializačním symbolem, který bude po celou dobu na dně zásobníku je symbol *$*. Dále začíná již samotná analýza zavoláním funkce pro získání následujícího tokenu ze seznamu. Po získání toho tokenu je provedena kontrola, zda se tento symbol může nacházet ve výrazu, pokud se jedná o proměnnou, je potřebná kontrola, zda tato proměnná byla deklarována, dále pokud je načten literál, je nutné převést jeho typ na vnitřní reprezentaci typů pro literály.

V další části analyzátoru dochází k výběru pravidla z precedenční tabulky. Po zjištění pravidla, je známa akce, která se na zásobníku má aktuálně provést, zda se jedná o další načítání symbolu, redukci nebo nastala syntaktická chyba, jelikož pro tuto kombinaci vstupu a aktuálního vrcholu zásobníku pravidlo neexistuje.

5.4 Paralelní syntaktická analýza

Syntaktická analýza začíná pracovat po kompletním načtení vstupního souboru lexikálním analyzátorem, který jednotlivé lexémy převedl na odpovídající tokeny. V seznamu těchto tokenů se také mohou objevit tzv. pseudotokeny, které označují místo, kde se ve zdrojovém textu nacházel výraz, viz obrázek 4.2.3-1. Tokeny tohoto výrazu byly uloženy do speciálního seznamu, čímž došlo k rozdělení zdrojového souboru na různé části, které budou následně analyzovány paralelně.



Tabulka podmínek

1	cond1
2	cond2
3	cond3

Obrázek 4.2.3-1 Vyjmutí výrazů pro paralelní analýzu

Hlavní analyzátor analyzuje postupně kostru programu a současně s ním je spuštěn další proces, který prochází seznam, obsahující vyjmuté výrazy a pro každý výraz spustí zvláštní proces, jenž provede jeho analýzu. Po dokončení analýzy čeká proces, která analýzu prováděl na zavolání, aby vypsal výsledky své analýzy.

Pokud při v průběhu své analýzy narazí hlavní analyzátor na pseudotoken, označující výraz, zavolá příslušný proces, který předá výsledek analýzy, a pokračuje dál v analýze kostry programu, dokud neprojde všechny tokeny programu.

5.5 Zpracování chyb

Výskyt chyb se nevyhýbá ani lexikálním analýze. Nejjednodušším příkladem může být výskyt znaku, který ve vstupním souboru nemá co dělat. V takovém případě je token určený pro navrácení syntaktickému analyzátoru nastaven na chybový stav a řízení je předáno zpět syntaktické analýze. Ta pak pokračuje dále v analýze zdrojového textu od prvního možného místa. V případě výskytu jiných chyb je postup stejný, navrátí se chybový token a syntaktická analýza si nechá načítat další tokeny a snaží se pokračovat v analýze zdrojového textu.

Syntaktická analýza shora dolů se také snaží pokračovat dále v analýze zdrojového textu co nejdříve, ovšem pokračování v analýze může být problematické v případě velkého zanoření v jednotlivých funkcích, které slouží k analýze.

V případě chybového stavu, který nastane v průběhu analýzy zdola nahoru, je dočten celý výraz do konce a analýza končí s chybou. Řízení je předáno zpět analyzátoru shora dolů a analýza pokračuje dále s příznakem chyby.

6 Uživatelská příručka

Tato kapitola seznamuje s používáním demonstračního programu a s obsahem přiloženého média.

Přiložené médium obsahuje několik archivů:

- *spl_paralel.tgz* – obsahuje zdrojové kódy paralelního analyzátoru
- *spl.tgz* – obsahuje zdrojové kódy sekvenčního analyzátoru pro porovnání výsledků
- *test.tgz* – obsahuje vzorové příklady pro testování analyzátoru
- *dokumentace.tgz* – obsahuje programovou dokumentaci vygenerovanou programem Doxygen

Pro úspěšné přeložení a spuštění analyzátoru je nutným softwarovým vybavením překladač *gcc* a program *make*. Archiv se zdrojovými kódy (*spl_paralel.tgz* a *spl.tgz*) je nutné rozbalit vhodným nástrojem, doporučuji extrahování archivu provést do nového adresáře. Pro samotný překlad extrahovaných zdrojových kódů je určen soubor *makefile*.

Pokud byly předchozí kroky provedeny s úspěchem, obsahuje adresář se zdrojovými kódy také nový binární soubor *spl_paralel/spl*, který je možné spustit. Tento spustitelný soubor vyžaduje při svém spuštění jediný parametr, a to cestu k souboru, který obsahuje zdrojový kód v jazyce SPL. Po spuštění programu s platným parametrem dojde k analýze zdrojového textu obsaženého v zadaném souboru. Pokud při spuštění nebyl zadán parametr, nebo je tento parametr neplatný, bude zobrazena jednoduchá nápověda, která vypíše základní informace o způsobu práce s analyzátorem.

Pro otestování funkčnosti analyzátoru je možné využít některý z dodaných testovacích programů napsaných v jazyce SPL. Tyto zdrojové texty se nacházejí v archivu *test.tgz*. Testovací programy je vhodné rozbalit do stejného adresáře, kde se nachází spustitelná verze analyzátoru. Spuštění analyzátoru vypadá následovně: *spl_paralel <jmeno souboru se zdrojovym textem>* . Výstupem analyzátoru je seznam eventuálních chyb a pravidel použitých při analýze zdrojového textu.

Práce byla přeložena a testována na Ubuntu 8.04 a školním serveru Merlin.

Závěr

Cílem bakalářské práce bylo popsat výstavbu paralelního syntaktického analyzátoru, který při své analýze zdrojového textu kombinuje různé metody analýzy. Konstrukce syntaktického analyzátoru je v počátku spojena s teoretickým návrhem lexikálního analyzátoru, který je schopen identifikovat jednotlivá slova na vstupu a uložit je k dalšímu zpracování pro syntaktický analyzátor.

Lexikální analýza zdrojového textu vyžaduje definici abecedy znaků, které budou analyzátorem akceptovány a budou z nich také složeny lexémy dané věty jazyka. Nad definovanou abecedou je dále nutné navrhnout vhodnou gramatiku. Navržená struktura lexémů pak následně dává předpoklad pro úspěšnou konstrukci deterministického konečného automatu, jenž podle načítaných vstupních znaků dokáže rozpoznat lexémy ve zdrojovém textu, a také dokáže rozhodnout, zda je možné, aby daný lexém jazyk akceptoval.

Tento lexém získané ze vstupní věty zdrojového textu je základem pro syntaktickou analýzu. Díky rozpoznanému lexému je možné rozvíjet derivační strom daným směrem až do poslední větve. Syntaktická správnost je důležitá k tomu, aby bylo možné správně sestavit program.

V případě, že jsou lexémy v daném vstupním programu nesprávně sestavena, nebo tento lexém v daném jazyce SPL vůbec neexistuje, nastává výjimka, která způsobí vypsání chybového hlášení na obrazovku. Analyzátor se pokusí dále pokračovat v analýze zdrojového textu od nejbližšího možného místa.

Pro konstrukci syntaktického analyzátoru byly zvoleny dvě metody. Jedná se o metodu rekurzivního zanoření a metodu precedenční analýzy. Paralelním spojením těchto dvou metod analýzy je možné získat výkonný syntaktický analyzátor, který je možné, vždy po mírných úpravách, využít pro většinu běžně používaných typů programovacích jazyků. V této práci byl za účelem otestování možností spojení těchto metod analýzy navržen nový jazyk SPL, který se svou konstrukcí podobá známému programovacímu jazyku Pascal.

Bylo dosaženo efektivního propojení jednotlivých částí analyzátoru. Při propojení použitých metod syntaktické analýzy bylo využito hlavních předností každé z nich a dle těchto vlastností jsou aplikovány pouze na ty části zdrojového textu, kde je to výhodné.

V dalším vývoji projektu by bylo vhodné přepsat analyzátor pomocí modernějšího programovacího jazyka, nejlépe objektově orientovaného, např. Java, C++. Rovněž sestrojit plnohodnotný sémantický analyzátor zdrojového textu a přidat generování mezikódu. V neposlední řadě také implementovat lepší metodu zotavování se z chyb při analýze.

Literatura

[1] Aho, V. A.; Lam, S. M; Sethi, R.; Ullman, D. J.: *Compilers: Principles, Techniques, and Tools.* Addison Wesley 2006, ISBN 0-321-48681-1, 1000 s.

[2] Bořivoj M.: *Tvorba překladačů*. ČVUT-FEL Praha, 118 s.

[3] Herout, P.: *Učebnice jazyka C*. 4. přeprac. vyd. České Budějovice : Kopp, 2004, ISBN 80-7232-220-6, 271 s.

[4] Meduna, A.; Lukáš, R.: *Formální jazyky a překladače IFJ Studijní opora 2006*. FIT VUT Brno, 152 s.

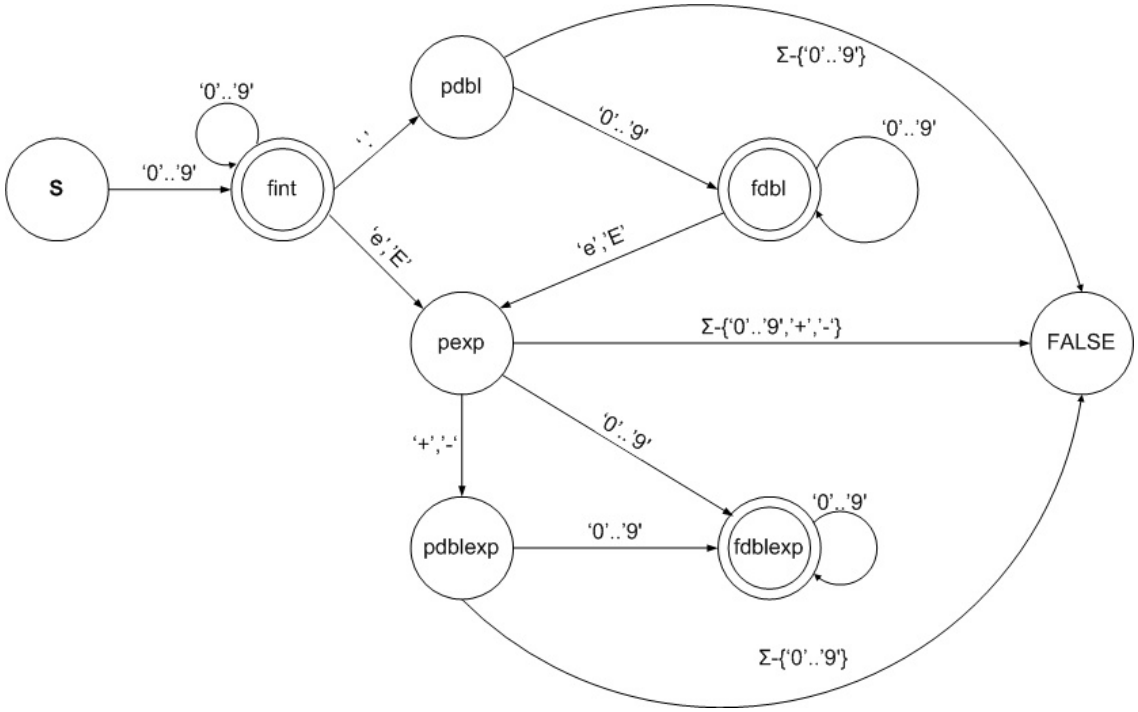
[5] Růžička M.: *Návrh algoritmu pro sémantické akce při výstavbě interpretu metodou rekurzivního sestupu*. Brno, 2006, diplomová práce, PEF MZLU v Brně.

[6] Tremblay, J.; Sorenson, P.: *The Theory and Practise of Compiler Writing*. McGraw-Hill, 1985, ISBN 0-070-65161-2, 892 s.

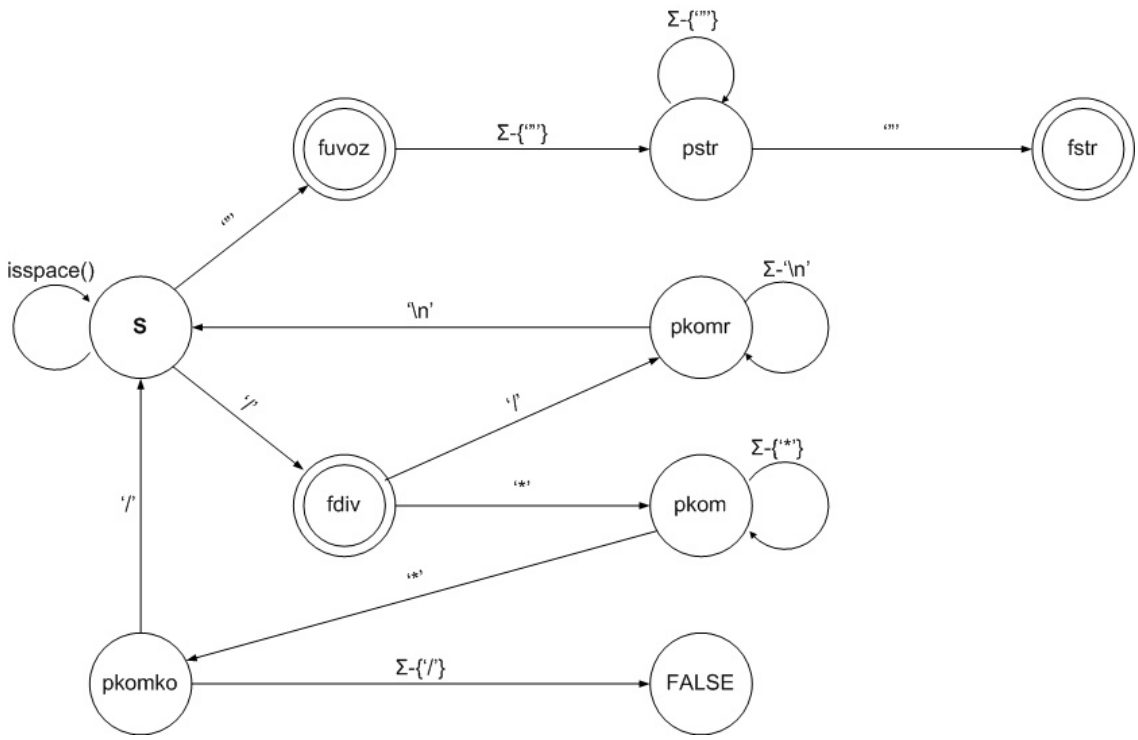
[7] Wikipedia: AVL tree —Wikipedia, The Free Encyclopedia. 2009, [Online; accessed 5-May-2009]. URL http://en.wikipedia.org/wiki/AVL-tree

Seznam příloh

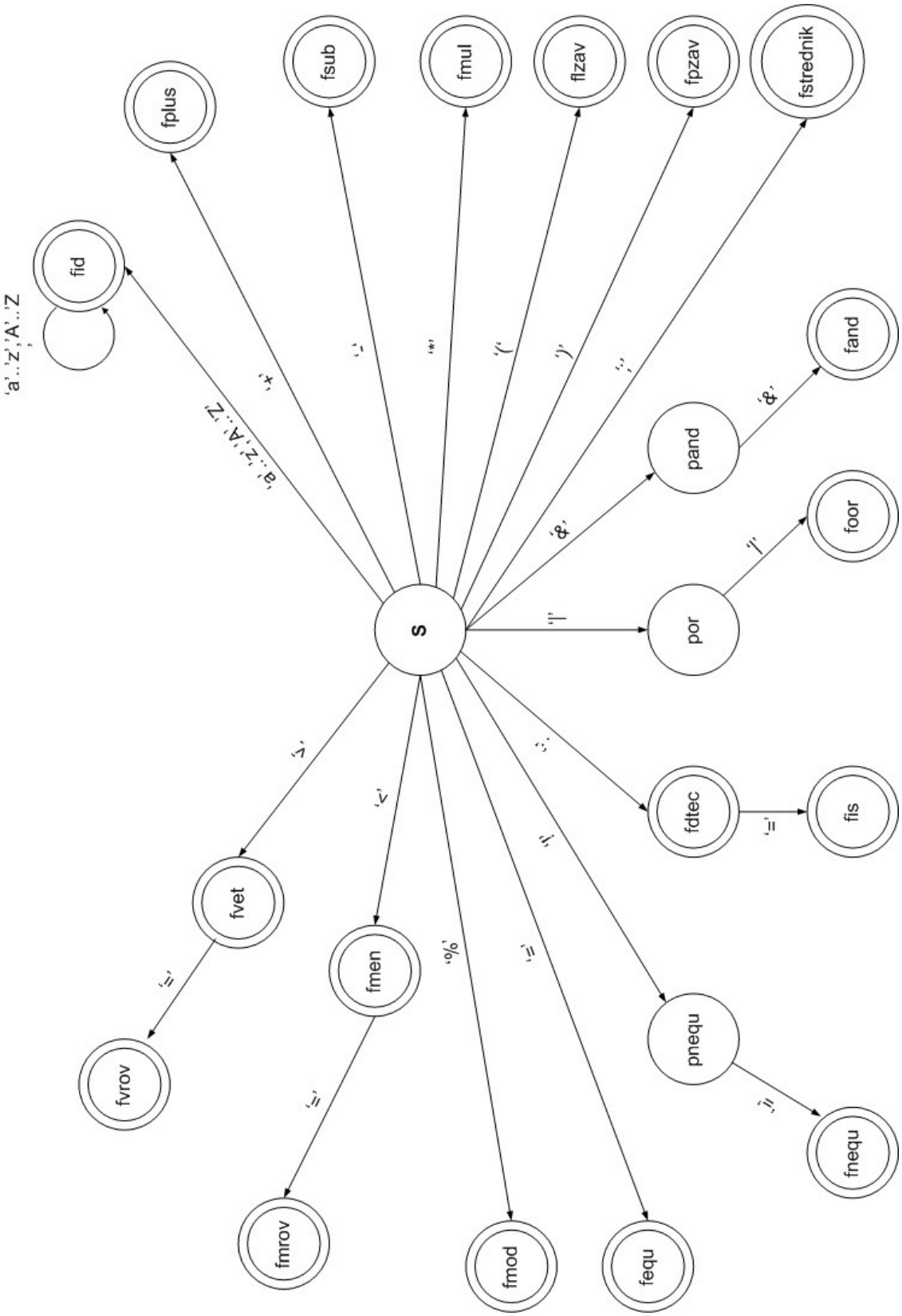
A. Grafy lexikálního analyzátoru



Lexikální analýza číselných literálů



Lexikální analýza komentářů a textových řetezců



Lexikální analýza operátorů a identifikátorů

B. Gramatika jazyka SPL

1. FILE -> VAR begin BODY end.
2. VAR -> var identifier : TYPE; VAR
3. VAR -> ε
4. BODY -> COMMAND BODY
5. BODY -> ε
6. COMMAND -> identifier := EXPRESSION;
7. COMMAND -> begin BODY end;
8. COMMAND -> if EXPRESSION then BODY ELSE
9. ELSE -> else COMMAND
10. ELSE -> ε
11. COMMAND -> while EXPRESSION do COMMAND
12. COMMAND -> repeat COMMAND until EXPRESSION;
13. COMMAND -> read identifier;
14. COMMAND -> write WRITE
15. WRITE -> identifier
16. WRITE -> (“string”)
17. EXPRESSION -> E EXPRESSION1
18. EXPRESSION1 -> ε
19. EXPRESSION1 -> = E
20. EXPRESSION1 -> < E
21. EXPRESSION1 -> > E
22. EXPRESSION1 -> <= E
23. EXPRESSION1 -> >= E
24. EXPRESSION1 -> && E
25. EXPRESSION1 -> || E
26. EXPRESSION1 -> != E
27. E -> T E1
28. E1 -> ε
29. E1 -> + T E1
30. E1 -> - T E1
31. T -> F T1
32. T1 -> ε
33. T1 -> * F T1
34. T1 -> / F T1
35. T1 -> % F T1
36. F -> TYPE
37. F -> (EXPRESSION)
38. TYPE -> int
39. TYPE -> float
40. TYPE -> string
41. TYPE -> bool
42. TYPE -> identifier

C. Precedenční tabulka jazyka SPL

	+	-	*	/	(	i	)	&&
+	>	>	<	<	<	<	>	>
-	>	>	<	<	<	<	>	>
*	>	>	>	>	<	<	>	>
/	>	>	>	>	<	<	>	>
(	<	<	<	<	<	<	=	<
i	>	>	>	>			>	>
)	>	>	>	>			>	>
&&	>	>	<	<	<	<	>	>
||	>	>	<	<	<	<	>	>
!=	>	>	<	<	<	<	>	>
<	>	>	<	<	<	<	>	>
<=	>	>	<	<	<	<	>	>
>	>	>	<	<	<	<	>	>
>=	>	>	<	<	<	<	>	>
=	>	>	<	<	<	<	>	>
$	<	<	<	<	<	<		<

	||	!=	<	<=	>	>=	=	$
+	>	>	>	>	>	>	>	>
-	>	>	>	>	>	>	>	>
*	>	>	>	>	>	>	>	>
/	>	>	>	>	>	>	>	>
(	<	<	<	<	<	<	<	
i	>	>	>	>	>	>	>	>
)	>	>	>	>	>	>	>	>
&&	>	>	>	>	>	>	>	>
||	>	>	>	>	>	>	>	>
!=	>	>	<	>	>	>	>	>
<	>	>	>	>	>	>	>	>
<=	>	>	>	>	>	>	>	>
>	>	>	>	>	>	>	>	>
>=	>	>	>	>	>	>	>	>
=	>	>	<	>	>	>	>	>
$	<	<	<	<	<	<	<	