VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DISTRIBUOVANÁ SYNTAKTICKÁ ANALÝZA

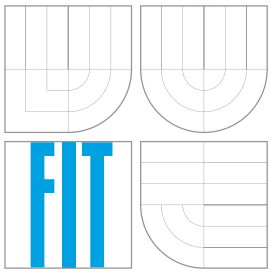
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAN LIPOWSKI

BRNO 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DISTRIBUOVANÁ SYNTAKTICKÁ ANALÝZA

DISTRIBUTED SYNTAX ANALYSIS

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE — Bc. JAN LIPOWSKI

AUTHOR

VEDOUCÍ PRÁCE — Prof. RNDr. ALEXANDER MEDUNA, CSc.

SUPERVISOR

BRNO 2007

Abstrakt

Práce prezentuje metodu distribuované syntaktické analýzy založené na vymezovacích symbolech. Je uvedena jejich definice, algoritmus výpočtu a algoritmus jejich analýzy. Na základě této analýzy je představen algoritmus, který vytváří pro vstupní gramatiku s vymezovacími symboly distribuovaný syntaktický analyzátor. Dále se práce zabývá implementací distribuovaného syntaktického analyzátoru vytvořeného touto metodou.

Klíčová slova

distribuovaná syntaktická analýza, vymezovací symbol, množina First, množina Last, množina Follow, množina Precede, množina Pair,

Abstract

The thesis presents a method of the delimiter based syntax analysis. There is introduced a definition, an algoritm of the computation and analysis of the delimiters in the thesis. Farther the thesis presents an algorithm creating a distributed parser based on the input grammar with the analysed delimiter symbols. Then there is introduced an implementation of the distributed parser created by the introduced method.

Keywords

distributed parser, delimiter, set First, set Last, set Follow, set Precede, set Pair

Citace

Jan Lipowski: Distribuovaná syntaktická analýza, diplomová práce, Brno, FIT VUT v Brně, 2007

Distribuovaná syntaktická analýza

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Prof. RNDr. Alexandra Meduny, CSc. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....

Jan Lipowski

22. května 2007

Poděkování

Rád bych poděkoval panu Prof. RNDr. Alexandru Medunovi, CSc. za věnovaný čas a trpělivost, který mi věnoval po dobu vypracování této práce.

Zadání diplomové práce/4237/2006/xlipow02

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2006/2007

Zadání diplomové práce

Řešitel: **Lipowski Jan, Bc.**
Obor: Informační systémy
Téma: **Distribuovaná syntaktická analýza**
Kategorie: Překladače

Pokyny:

1. Seznamte se detailně s pokročilými metodami syntaktické analýzy.
2. Studujte vlastnosti metod diskutovaných v předchozím bodě. Porovnejte je. Diskutujte jejich přednosti a nedostatky.
3. Navrhněte vlastní metodu syntaktické analýzy, která je založená na rozdělení zdrojového programu na vhodné části, které jsou syntakticky analyzovány různými metodami. Diskutujte přednosti a nedostatky tohoto distribuovaného přístupu.
4. Studujte užití navržené syntaktické analýzy (např. kompilátory, lingvistika, mikrobiologie). Navrhněte vhodný programovací jazyk a sestrojte jeho syntaktický analyzátor na bázi metody navržené v předchozím bodě. Testujte výsledný analyzátor.
5. Zhodnoťte dosažené výsledky a diskutujte další možný vývoj projektu.

Literatura:

- Tremblay, J. P. and Sorenson, P. G.: The Theory and Practice of Compiler Writing, McGraw-Hill, New York, 1985

Při obhajobě semestrální části diplomového projektu je požadováno:

- Body 1, 2 a 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Meduna Alexander, prof. RNDr., CSc.**, UIFS FIT VUT
Datum zadání: 28. února 2006
Datum odevzdání: 22. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Jaroslav Zendulka, CSc.
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Bc. Jan Lipowski**
Id studenta: 47526
Bytem: Č.p. 904, 739 94 Vendryně
Narozen: 06. 05. 1983, Třinec
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
diplomová práce

Název VŠKP: Distribuovaná syntaktická analýza
Vedoucí/školitel VŠKP: Meduna Alexander, prof. RNDr., CSc.
Ústav: Ústav informačních systémů
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:
tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☒ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

.....	.....
Nabyvatel	Autor

Obsah

Kapitola 1

Úvod

Syntaktická analýza je řídící a nejdůležitější komponentou překladače zdrojového kódu na cílový. Syntaktická analýza kontroluje, zda daný řetězec tokenů zdrojového textu je syntakticky správný, tzn. zda je generován gramatikou, která zdrojový text popisuje.
V případě korektního vstupu je výstupem syntaktické analýzy derivační strom.

Generování derivačního stromu je řízeno gramatickými pravidly a přistupujeme k němu obvykle dvěma způsoby. Metodou shora-dolů, nebo metodou zdola-nahoru. Podrobný popis obou metod je vysvětlen v [4] a [1]. Většina syntaktických analyzátorů využívá právě jednu z těchto metod.

Obě metody se od sebe liší svými vlastnostmi. Analyzátory implementující metodu zdola-nahoru jsou vhodné zejména pro zpracování výrazů. Analyzátory implementující metodu shora-dolů jsou vhodné pro zpracování kostry programu, tj. jejich řídících celků. Je na autorovi překladače, aby vyhodnotil vstupní gramatiku a vybral příslušnou metodu.

Vždy se dostáváme do situace, kdy překladač pracuje neefektivně. Nevhodným vstupem můžeme docílit toho, že překladač, primárně určený pro zpracování řídících konstrukcí, bude zpracovávat převážně podmínky. V tomto případě bychom po překladači žádali, aby reagoval na svůj vstup a vybral vždy efektivní metodu dynamicky.

Zde se dostáváme k jádru problému. Překladač implementující jednu metodu analýzy není schopen dynamicky tuto metodu měnit. Proto se snažíme postavit překladač tak, aby implementoval více metod, dynamicky reagoval na svůj vstup a vybral nejefektivnější metodu. Cílem této práce je vytvořit návrh takového analyzátoru, který by byl schopen splnit uvedené požadavky.

Prvním řešeným problémem je specifikace takových celků zdrojového kódu, které se vyznačují nějakou charakteristickou vlastností. Těmito celky mohou pro nás být funkce, výrazy, podmínky, třídy apod. Abychom mohli automatizovat výstavbu takového překladače, který by byl schopen tyto celky rozpoznávat, byla pro tento účel navržena teorie, díky které jsme schopni analyzovat jakoukoli vstupní gramatiku a tyto unikátní celky z ní automaticky vyjímat.

Navržená teorie se opírá o počáteční a koncové vymezovací symboly. Tyto symboly jsme schopni získat ze vstupní gramatiky výpočtem speciálních množin, jejichž definici a výpočet uvedeme v rámci práce. Dále budou tyto vymezovací symboly využity v návrhu distribuovaného překladače, kdy pomocí těchto vymezovacích symbolů budeme schopni vyjmout úseky vstupního řetězce a poslat je příslušné komponentě.

V první části práce se seznámíme s již známými principy syntaktických analyzátorů. Poté vytvoříme teoretické zázemí pro výpočet množin Follow, Precede a zejména množiny Pair, které nám pomohou definovat vymezovací symboly. Tyto množiny budou poté sloužit k analýze, který nám určí jedinečnost úseků ohraničených vymezovacích symbolů.

Druhou částí práce je vytvoření korespondujícího formálního modelu distribuovaného syntaktického analyzátoru. Tento model je založen na idei kooperujících komponent, mezi které je distribuován zdrojový kód. Každá z těchto komponent implementuje určitou metodu syntaktické analýzy.

V poslední části práce ukážeme praktické využití vytvořených algoritmů a modelů. Na navržené vstupní gramatice bude ukázán kompletní výpočet vymezovacích symbolů a jejich analýza. Poté vytvoříme příslušný model distribuovaného syntaktického analyzátoru. Závěr práce věnujeme implementaci navrženého systému a jeho testování.

Tato práce navazuje na semestrální projekt. V rámci semestrálního projektu byl vytvořen algoritmus pro vytvoření a analýzu množiny Pair 5.0.3. Diplomová práce doplňuje semestrální projekt o zbývající algoritmy a o celou praktickou část.

Kapitola 2

Metody syntaktické analýzy

V počátečních kapitolách se budeme věnovat teoretickým základům. V této kapitole se seznámíme s různými metodami syntaktické analázy. Vyzdvihneme jejich jednotlivé vlastnosti, přednosti a nedostatky. V následujícím textu bude místy místo výrazu *syntaktický analyzátor* použit ekvivalentní výraz *parser*.

Podle směru tvoření derivačního stromu rozlišujeme dvě základní metody syntaktické analýzy:

2.1 Metoda shora-dolů

Top-down parser vezme startovací symbol a pokusí se z něho vygenerovat vstupní řetězec. Vstup je na počátku popisován jedním počátečním neterminálním symbolem. Tento symbol je kořenem derivačního stromu. Poté dochází k větvení na další neterminální symboly, které jsou kořenem pro menší části zdrojového textu. Nejmenší, dále nedělitelné úseky vstupního řetězce, tokeny, odpovídají terminálům. Pokud se nám takto shora-dolů podaří vytvořit derivační strom, vstupní řetězec je přijat. Zde uvádíme několik nejznámějších top-down parserů.

LL parser *(Left to right leftmost parser)* Princip parseru spočívá v generování nejlevější derivace vstupní věty. Model LL parseru obsahuje zásobník, pro uchování právě generovaných symbolů gramatiky, a LL tabulku, která určuje pravidlo, které má přepsat nejlevější neterminál. Formální popis LL parseru se nachází v [1].

parser rekurzivním sestupem Tento parser je alternativou k LL parseru. Principem rekurzivního sestupu je analýza na základě rekurzivních funkcí. Každá z těchto funkcí reprezentuje levou stranu jednoho pravidla dané gramatiky. Tělo funkce obsahuje akce, závislé na pravé straně pravidla, které reprezentují přepis levé strany pravidla pravou. V případě, že se na pravé straně pravidla nachází neterminál, je v těle funkce volána funkce reprezentující tento neterminál. V případě terminálu je provedeno porovnání se vstupním tokenem. Při shodě tokenu a terminálu funkce vrací úspěšné porovnání. Tato

hodnota je při návratu z rekurze distribuována směrem ke kořeni, tedy rekurzivně zpět k funkci, která reprezentuje počáteční neterminál. Po ukončení syntaktické analýzy kořenová funkce obsahuje informaci o správnosti vstupního řetězce. Více v [1].

Packrat parser Packrat parser [2] vychází z backtracking parserů, kdy větná konstrukce není předvídaná (jako u prediktivního analyzátoru), ale je odhadována. V případě, že větná konstrukce není shodná s odhadovanou, systém se vrátí do určitého kořenového stavu (backtracking) a zkusí se jiná konstrukce. Díky této metodě mají backtracking parsery velkou časovou složitost.

Packrat parser zachovává veškeré výhody backtracking parserů, ale zároveň zachovává lineární složitost a to tím způsobem, že si zapamatovává návratové hodnoty volaných funkcí. Žáden výsledek není počítán 2x. Tento algoritmus má narozdíl od backtracking parseru vysokou paměťovou spotřebu.

Tímto parserem může být zpracovávána každá LL(k) a LR(k) gramatika. Struktura Packrat parseru je podobná backtracking rekursive descent parseru. Převod Packrat parseru na rekursive descent je možný pouhou změnou struktury parseru.

Tail recursive parser Tail recursive parser [9] vznikl z recursive descent parseru. Používá se zejména k parsování levě rekurzivních gramatik, protože odstranil riziko vzniku nekonečné smyčky (rekurzivní zanořování). Levou rekurzi nahrazuje jednoduchou iterací. Navíc se nezaplňuje místo na zásobníku jak je tomu u každého rekurzivního volání.

Unger's parser Unger parser [7] předpokládá na vstupu řetězec bezkontextové gramatiky. Metoda je bezesměrná. Zpracovává vstupní řetězec jako celek, resp. tokeny, které jsou brány v různém pořadí. Parser přepisuje jednotlivé neterminály pravými stranami gramatických pravidel, a kontroluje, zda je možno generovat vstupní řetězec. Pokud ne, zkouší najít vhodnější pravidlo (backtracking metodou s depth-first vyhledáváním).

2.2 Metoda zdola-nahoru

Základní myšlenkou bottom-up parserů je využívání gramatických pravidel v opačném pořadí. To znamená, že začínáme aplikovat pravidla na listy derivačního stromu, na terminály (bottom), a skončíme vygenerováním startovacího symbolu. Postupně tedy redukujeme řetězec pomocí pravých stran pravidel gramatiky tak dlouho, dokud nedojdeme k počátečnímu symbolu. K uložení současné větné formy slouží zásobník. Ten poskytuje operace *shift* a *reduce* k vkládání zásobníkových symbolů k redukci symbolů na zásobníku na jediný symbol. Volba operace je řízena tabulkou (precedenční, LR, apod.)

LR parser *(Left to right rightmost parser)* [6] LR parser čte vstup zleva doprava a produkuje nejpravější derivaci. Jeho síla je velmi vysoká, ale bohužel je jeho implementace

složitá. Součástí LR parseru je zásobník pro ukládání současného stavu a LR tabulka (má dvě části: *action* a *goto*), která definuje operace, které se mají v daném stavu provést. Podle typu konstrukce LR tabulky rozdělujeme LR parsery na tyto typy:

SLR parser *(Simple LR parser)* SLR tabulka je generována stejným způsobem, jako by byla generována pro LR(0) parser, ale s tou výjimkou, že operace redukce na zásobníku je provedena pouze v případě pravidla $A \rightarrow w$ za podmínky, že následující znak na vstupní pásce je prvkem množiny $follow$ neterminálu A. Tento parser má menší sílu než LR(1) parsery. Gramatiku, kterou je schopen SLR parser parsovat, nazýváme SLR gramatika.

LALR parser *Look-ahead LR parser* LALR parsery jsou další kategorií LR parserů. Tyto parsery jsou široce používány a jsou to právě tyto parsery, které generují nástroje jako je yacc, bison apod. Jsou o něco málo silnější než SLR parsery, ale jsou stále slabší než LR(1) parsery. Jejich obrovskou výhodou je, že při své síle vytvoří stejně velkou tabulku jako SLR parser. To díky tomu, že místo $follow$ množiny zavádí *lookahead* množinu, která je více vázaná na daný kontext, což nám vymezuje konkrétnější pravidla a tím zmenšuje velikost tabulky. Algoritmus konstrukce LALR tabulky opět vychází z LR(0) tabulky.

Canonical LR parser Canonical LR parser je velice silný, ale má tu nevýhodu, že tvoří velice objemnou LR tabulku. Princip tvorby tabulky je obdobný jako u LR(0). Pravidla z množiny položek (item) obsahují navíc $follow$ prvek, tzn. prvek který očekáváme za řetězcem, který je generován z následujícího neterminálu.

$$A \rightarrow B \bullet C, a$$

tedy znamená, že jsme přečetli řetězec korespondující B, očekáváme řetězec korespondující C, který bude následován symbolem a.

GLR parser *generalized LR parser* GLR je rozšířením LR parseru pro nedeterministické a nejednoznačné gramatiky. Princip je identický s LR parserem až na fakt, že vstup je nejdříve zpracován metodou breath-first search a zjistí se všechny jeho možné interpretace. Následně jsou zjištěné gramatiky převedeny do LR tabulek. Poté se nejednoznačné úseky zpracovávají paralelně.

precedence parsing Precedenční parser [1] je nejlabší z uvedených. Jeho hlavní výhodou je však jeho jednoduchost z pohledu implementace a také kvalita a rychlost vyhodnocování výrazů. Precedenční parser využívá precedenční tabulku. Jejím základem je precedence a asociativita operátorů.

Earley parser Earley parser [8] je obecný překladač, který je schopen analyzovat jakékoli bezkontextové gramatiky. Je považován za top-down parser, ale má hodně společného s bottom-up parsery. Má kvadratickou a v horším případě až kubickou složitost (závisí

na gramatice). LR algoritmus generuje ze vstupní gramatiky množinu stavů, která reprezentuje všechny možné rozklady vstupního řetězce. Poté je tato sekvence stavů procházena a parser se snaží složit derivační strom. Pokud nejsme schopni se při analýze rozhodnou o volbě pravidla, prohledáváme stavový prostor možných stavů a hledáme alternativy k rozkladu (depth first, breadth first).

CYK parser Standardní verze CYK parseru vyžaduje na vstupu bezkontextovou gramatiku zapsanou v Chomského normální formě. CYK parser má sílu rozpoznat všechny CFG. Algoritmus CYK je velice jednoduchý a spadá spíše do oblasti dynamického programování. Pomocí CYK algoritmu byl také uznán membership problém CFG jako rozhodnutelný. Vzhledem k Earley parseru je zajímavé porovnání v [10], kde bylo zjištěno, že CYK parser je téměř vždy pomalejší než Earley a téměř vždy zabere více paměťového prostoru.

Existuje mnoho typů parserů a každý z nich má své specifické rysy. Mají různou sílu, časovou a prostorovou složitost, různé interní řídící tabulky a použité algoritmy. Nelze jednoznačně vyhodnotit, který je lepší či horší. Vždy záleží na vstupní gramatice a dalších faktorech. Obecně můžeme konstatovat, že top-down parsery jsou vhodnější k analýze řídících struktur programů a naopak bottom-up parsery jsou vhodnější k analýze složitých výrazů.

Účelem této práce je navrhnout model distribuované syntaktické analýzy umožňující využití lepších stránek obou těchto přístupů. Stěžejní částí modelu bude automatizace procesu distribuce vstupního řetězce mezi systém parserů a také formální návrh komunikace parserů .

Kapitola 3

Distribuovaná syntaktická analýza

V této kapitole se seznámíme s několika existujícími přístupy k distribuci syntaktické analýzy. V závěru kapitoly neformálně popíšeme nový model a nastíníme vývoj zbývající části této práce.

3.1 Sekvenční syntaktická analýza

Sekvenční syntaktická analýza pracuje tak, že v každém okamžiku je aktivní pouze jedna komponenta kompilátoru. Základním nedostatkem je použití pouze jedné metody (např. top-down). Tato metoda je sice vybírána na základě převažujících rysů jazyka, tak aby byla schopna analyzovat daný jazyk co nejrychleji, ale bohužel vždy budou zůstávat úseky, kde vybraná analýza je vysoce neefektivní. Tento nedostatek se snažíme odstranit zavedením distribuované syntaktické analýzy. Cílem je vytvořit paralelní kooperativní model složený ze sekvenčních parserů - komponent.

Prvním problémem se kterým se potýkáme je specifikace a extrakce celků, které budou jednotlivé komponenty zpracovávat. Na základě výhod a nevýhod přístupů bottom-up a top-down bylo zvoleno zpracování těla programu metodou top-down a jednotlivých výrazů metodou bottom-up. Nyní je na místě klást si otázku, jakým způsobem dělit vstupní řetězec tak, aby jeho jednotlivé části mohly být distribuovány mezi jednotlivé komponenty systému.

3.2 Analýza pomocí stopových prvků

Jedním z již existujících přístupů je definování tzv. stop-point prvků, které označují přepínání jednotlivých komponent systému. Pokud komponenta při svém zpracování narazí na tento stop-point, předá řízení zpracovávání další komponentě. Model s použitím stop-pointů je popsán v [5].

Za jednotlivé stop-pointy můžeme například považovat např: if, $then$. Úsek kódu mezi těmito symboly je výraz, na jehož zpracování použijeme jinou metodu, než je metoda zpracovávající tělo programu.

Tento model má určitou podobnost s CD gramatickými systémy, kdy na vygenerování tzv. komunikačního symbolu je vstupní řetězec předán další komponentě. Viz: [3]

Tato metoda však stále běží sekvenčně. Oproti jednokomponentové syntaktické analýze je díky použití optimálních metod pro jednotlivé celky rychlejší, ale stále nám pracuje pouze jedna komponenta. Jak tedy efektivněji využít volný čas ostatních komponent?

3.3 Umělé vymezovací symboly

Na základě poznatků o využití stop-point prvků autor navrhl metodu, jak je možno za běhu vyžádat zpracování určité části řetězce jinou komponentou. Mějme systém o dvou komponentách. První komponenta implementuje top-down parser, druhá komponenta bottom-up parser. Top-down parser bude řídící komponentou systému. Bottom-up komponenta bude podřízenou komponentou systému top-down parseru.

3.3.1 Princip

Princip vymezení různých celků vstupního řetězce spočívá v přidání, v rámci jazyka, jedinečných symbolů, které tyto celky ohraničují, např. výrazy. Tyto symboly budeme nazývat *vymezovací symboly*. Pokud top-down parser má na vstupní pásce počáteční vymezovací symbol, začne kopírovat svou vstupní pásku na vstupní pásku bottom-up komponenty až do doby, dokud nenarazí na ukončující vymezovací symbol. Kopírovaný úsek nahradí pseudoterminálem, který označuje pozici, kam se po ukončení bottom-up analýzy vrátí výsledek o úspěšnosti analýzy, případně generovaný intermediární kód. Mezitím je top-down parser schopen dále pokračovat ve zpracovávání struktury programu. Tento postup se opakuje tak dlouho, než řídící analyzátor nenarazí na konec vstupní pásky.

Po ukončení bottom-up analýzy je vygenerován komunikační symbol, který oznámí, že analýza identifikovaná pod daným pseudoterminálem byla ukončena a jsou provedeny akce návratu výsleků analýzy, případně intermediárního kódu.

3.3.2 Model

Řídící komponenta využívá standardní model top-down parseru (LL tabulka, zásobníkový automat). Navíc má ale tato komponanta tu vlastnost, že je schopna při přečtení počátečního vymezovacího symbolu kopírovat svůj vstup na vstup jiné komponenty. Pro tuto operaci kopírování jsou do zásobníkového automatu přidána nová pravidla. Operaci automat provádí, pokud je ve stavu kopírování (stav *copy*). Do tohoto stavu se automat dostane načtením počátečního vymezovacího symbolu. Pokud je na vstupu koncový vymezovací symbol, zásobníkový automat přejde zpět do původního stavu a pokračuje v analýze.

Podřízená komponenta(bottom-up parser) je navržena tak, aby po ukončení analýzy generovala příslušný komunikační symbol. Symbol je poslán řídící komponentě, která provede příslušné operace.

3.3.3 Zásobníkový automat s kopírovacím stavem

Zásobníkový automat s kopírovacím stavem vychází z definice nerozšířeného zásobníkového automatu a je definován následovně:

Definice: 3.3.1 Zásobníkový automat M *je sedmice*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F), \quad kde$$

- Q *je konečná množina stavových symbolů reprezentující vnitřní stavy řídící jednotky*
- Σ *je konečná vstupní abeceda; jejími prvky jsou vstupní symboly*
- Γ *je konečná abeceda zásobníkových symbolů*
- δ *je zobrazení z množiny* $Q \times (\Sigma \in \{\epsilon\}) \times \Gamma$ *do konečné množiny podmnožin* $Q \times \Gamma^* \times I$ *popisující funkci přechodů a kde* I *je n-tice* $(i_1, i_2, \ldots, i_n)$*, kde* i_n *označuje vstupní pásku komponenty n.*
- $q_0 \in Q$ *je počáteční stav řídící jednotky*
- $Z_0 \in \Gamma$ *je symbol který je na počátku uložen do zásobníku - tzn.* startovací symbol.
- $Z \subseteq Q$ *je množina koncových stavů*

3.3.4 Převod BKG na ZA s kopírovacím stavem

Algoritmus převodu BKG na ZA je narozdíl od [1] rozšířen o zmiňovaný *copy* stav, kdy automat kopíruje svůj vstup na vstup jiné komponenty. Podmínkou pro úspěšné vytvoření pravidel je existence vymezovacích symbolů, které ohraničují námi zvolený řetězec, který chceme kopírovat na vstup jiné komponenty.

Algoritmus: 3.3.1 *Převod BKG na ZA s copy stavem:*

vstup: $BKG \quad G = (N, T, P, S); \quad D = \{d_s, d_e\}$*, kde* d_s *je počáteční vymezovací symbol a* d_e *je koncový vymezovací symbol;* $E : E \in N$ *a označuje neterminál, který je počátečním symbolem pro řetězce, které chceme kopírovat na jinou vstupní pásku.*

výstup: $ZA \quad M = (Q, \Sigma, \Gamma, R, s, S, F); \quad L(G) = L(M)_\varepsilon$

1. $Q = \{s, c\}$
2. $\Sigma = T$
3. $\Gamma = N \cup T$

4. množina R:

- *pro všechna* $a \in \Sigma \wedge a \notin D$*: přidej* $asa \rightarrow s(,)$ *do* R
- *pro všechna* $A \in (N \backslash E) : A \rightarrow x \in P$ *přidej* $As \rightarrow ys(,)$ *do* R*, kde* $y = reversal(x)$ *a současně z tohoto pravidla odstraníme neterminál, který nám ohraničují omezovací symboly.*
- *přidej* $sd_s \rightarrow c$ *do* R
- *přidej* $cd_e \rightarrow s$ *do* R
- *přidej* $ca \rightarrow c(,a)$

Pravidlo $sd_s \rightarrow q$ do R vyjadřuje přechod kopírovacího modu, pravidlo $qd_e \rightarrow s$ do R vyjadřuje přechod zpět a pravidlo $qa \rightarrow q(,a)$ vyjadřuje kopírování symbolu na vstupní pásku komponenty 2.

Zde je praktická ukázka převodu BKG na ZA s *copy* stavem následovaná aplikací této metody v konkrétní analýze.

Příklad: 3.3.1 *Převod BKG na ZA s copy stavem*
Vstup:

- $BKG \quad G = (N, T, P, S)$
  - $N = \{< PROG >, < EXP >\}$
  - $T = \{id, =, +, ;, [,]\}$
  - $P = \{< PROG > \rightarrow id = [< EXP >]; \}$
  - $S =< PROG >$
- $D = \{[,]\}$

Výsledek:
$ZA \quad M = (Q, \Sigma, \Gamma, R, s, S, F) :$

- $Q = \{s, c\}$
- $\Sigma = \{id, =, +, ;, [,]\}$
- $\Gamma = \{< PROG >, < EXP >, id, =, +, ;, [,]\}$
- $S =< PROG >$
- $F = \{\emptyset\}$

- $R = \{$
  $id \quad s \quad id \rightarrow s;$
  $= s = \rightarrow s;$
  $+s+ \rightarrow s;$
  $; s; \rightarrow s;$
  $< PROG > s \rightarrow;][= id \quad s;$
  $[s[\rightarrow c;$
  $]c] \rightarrow s;$
  $ca \rightarrow c(, a);$
  $\}$

Příklad: 3.3.2 *Příklad analýzy vstupní pásky ZA definovaným v 3.3.1. Cílem příkladu je ukázat využití kopírovací funkce ZA. Mějme systém o dvou komponentách. Definovaný ZA je součástí první komponenty, která implementuje metodu top-down. V případě, že narazí na počáteční vymezovací symbol, začne kopírovat symboly ze svého vstupu na vstupní pásku druhé komponenty. Po načtení koncového vymezovacího symbolu přejde zpět do původního stavu a pokračuje v analýze.*

Zásobník komp. 1	vstup komp. 1	pravidlo	stav	vstup komp. 2
$\$< PROG >$	x=[a+b];	$< PROG > \rightarrow;][= id$	s	-
$\$;][=$id	x=[a+b];	$idsid \rightarrow s$	s	-
$\$;][=$	=[a+b];	$= s = \rightarrow s$	s	-
$\$;][$	[a+b];	$[s[\rightarrow c$	s	-
$\$;]$	a+b];	$ca \rightarrow c(, a)$	c	-
$\$;]$	+b];	$ca \rightarrow c(, a)$	c	a
$\$;]$	b];	$ca \rightarrow c(, a)$	c	a+
$\$;]$	];	$]s] \rightarrow s$	c	a+b
$\$;$	;	$; s; \rightarrow s$	s	a+b
$\$$			s	a+b

Tabulka 3.1: Výsledek analýzy pomocí *copy* stavu

Zde vidíme, že po ukončení analýzy byl řetězec, který je výrazem, úspěšně nakopírován na vstup druhé komponenty, která implementuje např. metodu bottom-up. Splnili jsme tedy první krok a to kopírování vstupu jedné komponenty na pásku druhé

Tento model má jednu nevýhodu. Vstupní gramatika byla navržena tak, aby vymezovací symboly byly unikátní, tzn. tyto symboly se nikde jinde nevyskytují v jiném kontextu, případně nejsou ani součástí kopírovaného výrazu. Je to tedy nepřirozený umělý zásah do gramatiky jazyka. Pouze málokdy se stane, že se ve vstupní gramatice tyto unikátní symboly

nacházejí a jsou viditelné na první pohled. Vyhledání a anylýza vymezovacích symbolů s sebou přináší řadu nepříjemností. Řadu z nich budeme řešit v následujících částech práce. Budeme se zabývat zabývat obecným řešením, kdy budeme vyhledávat
a analyzovat vymezovací symboly, které jsou součástí jazyka a budeme diskutovat vhodnost jejich použití.

Kapitola 4

Obecné vymezovací symboly

V minulé kapitole jsme ukázali algoritmus, kterým můžeme v průběhu analýzy kopírovat úseky vstupního řetězce na vstupní pásku komponenty, která nad tímto řetězcem provede syntaktickou analýzu. Takto jsme schopni distribuovat syntaktickou analýzu mezi více komponent s tím, že každá komponenta bude provádět nad daným řetězcem nejefektivnější analýzu.

Idea minulé kapitoly byla opřena o určitý ideální stav, kdy symboly ohraničující úsek řetězce, který chceme kopírovat, byly přidány tak, aby byly unikátní.

Bohužel většina zkoumaných jazyků se nebude nacházet v tomto ideálním stavu. Tato a další kapitoly se budou zabývat obecným mechanismem, jak algoritmicky analyzovat vymezovací symboly od jejich extrakce z gramatiky jazyka až po vyhodnocení vhodnosti použití.

Cílem bude ukázat, že jsme algoritmicky schopni zjistit, zda námi zvolený úsek vstupního řetězce (výraz, či jiný logický celek zdrojového kódu), je vhodný pro distribuci mezi jednotlivými komponentami systému.

Druhým výsledkem bude zjištění, jak gramatiku navrhnout tak, aby byla vhodná pro distribuovaný překlad, tzn. jakých rysů jazyka se vyvarovat aby vstupní gramatika byla distribuovatelná.

Celý tento princip stojí na množině Pair, která obsahuje dvojice množin vymezovacích symbolů. Výpočet množiny Pair stojí na výpočtu množin Empty, First, Last, Follow a Precede. Definujme nyní všechny algoritmy, které budeme potřebovat k výpočtu těchto množin a k výpočtu Pair.

4.1 množina $Empty(X)$

Definice: 4.1.1 *Nechť $G = (N, T, P, S)$ je BKG. $Empty(x) = \{\varepsilon\}$ pokud $x \Rightarrow^* \varepsilon$; jinak $Empty(x) = \emptyset$, kde $x \in (N \cup T)^*$.*

Množina empty tedy obsahuje prvek ε pokud x derivuje ε. Jinak je prázdná.

Algoritmus: 4.1.1 *Postup tvorby množiny Empty(X) pro všechny symboly $X \in N \cup T$ vstupní gramatiky $G = (N, T, P, S)$.*

- *pro každé $a \in T$: $Empty(a) = \emptyset$*
- *pro každé $A \in N$:*
  if $A \to \varepsilon$
  then $Empty(A) = \varepsilon$
  else $Empty(A) = \emptyset$
- *while(je možno měnit množinu Empty())*
  if $A \to (X_1 X_2 \dots X_n \in P$ a $Empty(X_i) = \{\varepsilon\}$ pro $i = 1, \dots, n$
  then $Empty(A) = \varepsilon$.

4.2 množina $First(X)$ a $Last(X)$

Při výpočtu množiny Last(X) vyjdeme ze známého algoritmu výpočtu množiny First(X). Jak název napovídá, množina Last je přesným opakem množiny First.

Algoritmus: 4.2.1 *Výpočet množiny First(X) pro každé $X \in N \cup T$ vstupní gramatiky $G = (N, T, P, S)$:*

- *pro každé $a \in T$: $First(a) = \{a\}$*
- *pro každé $A \in N$: $First(A) = \emptyset$*
- *while(je možno měnit množinu $First()$):*
  if $A \to X_1 X_2 \dots X_{k-1} X_k \dots X_n \in P$ then
  - *Přidej všechny symboly z $First(X_1)$ do $First(A)$*
  - *if $Empty(X_i) = \{\varepsilon\}$ pro $i = 1, \dots, k-1$, kde $k \leq n$*
    then přidej všechny symboly z $First(X_k)$ do First(A)

Algoritmus: 4.2.2 *Výpočet množiny Last(X) pro každé $X \in N \cup T$ vstupní gramatiky $G = (N, T, P, S)$:*

- *pro každé $a \in T$: $Last(a) = \{a\}$*
- *pro každé $A \in N$: $Last(A) = \emptyset$*
- *while(je možno měnit množinu $Last()$):*
  if $A \to X_1 X_2 \dots X_k X_{k+1} \dots X_n \in P$ then

– *Přidej všechny symboly z* $Last(X_n)$ *do* $Last(A)$

– *if* $Empty(X_i) = \{\varepsilon\}$ *pro* $i = k+1, \ldots, n$, *kde* $k \leq n$
then přidej všechny symboly z $Last(X_k)$ *do Last(A)*

4.3 množina $First(X_1, X_2, \ldots, X_n)$ a $Last(X_1, X_2, \ldots, X_n)$

Opět budeme při výpočtu množiny $Last(X_1, X_2, \ldots, X_n)$ vycházet z algoritmu výpočtu množiny $First(X_1, X_2, \ldots, X_n)$.

Algoritmus: 4.3.1 *Výpočet množiny* $First(X_1, X_2, \ldots, X_n)$ *pro vstupní gramatiku* $G = (N, T, P, S)$. *Předpokládejme, že máme definováno* $First(X)aEmpty(X)$ *pro každé* $X \in N \cup T$, *a také* $x = X_1, X_2, \ldots, X_n$, *kde* $x \in (N \cup T)^+$

- $First(X_1, X_2, \ldots, X_n) = First(X_1)$
- *while(je možno měnit množinu* $First(X_1, X_2, \ldots, X_n)$*):*
if $Empty(X_i) = \{\varepsilon\}$ *pro* $i = 1, \ldots, k-1$, *kde* $k \leq n$
then přidej všechny symboly z $First(X_k)$ *do* $First(X_1, X_2, \ldots, X_n)$

Algoritmus: 4.3.2 *Výpočet množiny* $Last(X_1, X_2, \ldots, X_n)$ *pro vstupní gramatiku* $G = (N, T, P, S)$. *Předpokládejme, že máme definováno* $Last(X)aEmpty(X)$ *pro každé* $X \in N \cup T$, *a také* $x = X_1, X_2, \ldots, X_n$, *kde* $x \in (N \cup T)^+$

- $Last(X_1, X_2, \ldots, X_n) = Last(X_n)$
- *while(je možno měnit množinu* $Last(X_1, X_2, \ldots, X_n)$*):*
if $Empty(X_i) = \{\varepsilon\}$ *pro* $i = k+1, \ldots, n$, *kde* $k \leq n$
then přidej všechny symboly z $Last(X_k)$ *do* $Last(X_1, X_2, \ldots, X_n)$

4.4 množina $Empty(X_1, X_2, \ldots, X_n)$

Výpočet množiny $Empty(X_1, X_2, \ldots, X_n)$ jsme schopni vypočítat z těchto údajů: $G = (N, T, P, S)$, $Empty(X)$ pro všechna $X \in N \cup T$ a $x = X_1, X_2, \ldots, X_n$, kde $x \in (N \cup T)^+$.

Algoritmus: 4.4.1 *Výpočet množiny* $Empty(X_1, X_2, \ldots, X_n)$.

- *if* $Empty(X_i) = \{\varepsilon\}$ *pro* $i = 1, \ldots, n$
then $Empty(X_1, X_2, \ldots, X_n) = \{\varepsilon\}$
else $Empty(X_1, X_2, \ldots, X_n) = \emptyset$

4.5 množina $Follow(X)$ a množina $Precede(X)$

Množina Follow(X) je množina všech terminálů, které se mohou vyskytovat vpravo od X ve větné formě. Tato množina nám pomůže nalézt koncový vymezovací symbol daného neterminálu. Na základě algoritmu výpočtu množiny Follow(X) je definován algoritmus výpočtu množiny Precede(X). Množina Precede(X) je množina všech terminálů, které se mohou vyskytovat vlevo od X ve větné formě. Tento algoritmus nám tedy pomůže nalézt symboly, které zleva ohraničují námi zvolený neterminál, tedy počáteční omezovací symboly.

Definice: 4.5.1 *Nechť $G = (N, T, P, S)$ je BKG. Pro všechna $A \in N$ definujeme množinu $Follow(A)$: $Follow(A) = \{a : a \in T, S \Rightarrow^* xAay, x, y \in (N \cup T)^*\} \cup \{\$: S \Rightarrow^* xA,$ $x \in (N \cup T)^*\}$*

Definice: 4.5.2 *Nechť $G = (N, T, P, S)$ je BKG. Pro všechna $A \in N$ definujeme množinu $Precede(A)$: $Precede(A) = \{a : a \in T, S \Rightarrow^* xaAy, x, y \in (N \cup T)^*\} \cup \{\dot{} S \Rightarrow^* Ax,$ $x \in (N \cup T)^*\}$*

Algoritmus: 4.5.1 *Algoritmus výpočtu množiny $Follow(A)$ pro všechna $A \in N$ gramatiky $G = (N, T, P, S)$ definujeme následovně:*

- $Follow(S) = \{\$\}$
- *while(je možno měnit množinu Follow):*
  - *if $A \to xBy \quad \in P$ then*
    - *if $y \neq \{\varepsilon\}$ then*
      přidej všechny symboly z First(y) do Follow(B)
    - *if $Empty(y) = \{\varepsilon\}$ then*
      přidej všechny symboly z Follow(A) do Follow(B)

Algoritmus: 4.5.2 *Algoritmus výpočtu množiny $Precede(A)$ pro všechna $A \in N$ gramatiky $G = (N, T, P, S)$ definujeme následovně:*

- $Precede(S) = \{\hat{}\}$
- *while(je možno měnit množinu Precede):*
  - *if $A \to xBy \quad \in P$ then*
    - *if $x \neq \{\varepsilon\}$ then*
      přidej všechny symboly z Last(x) do Precede(B)
    - *if $Empty(x) = \{\varepsilon\}$ then*
      přidej všechny symboly z Precede(A) do Precede(B)

V této kapitole jsme definovali algoritmy nezbytné pro výpočet množiny Pair(). Algoritmus výpočtu množiny Pair() bude popsán v následující kapitole. Tato množina obsahuje pro každý neterminál množinu dvojic symbolů, které daný neterminál mohou ohraničovat. Dále si ukážeme, jakým způsobem je třeba tuto množinu analyzovat a jak ověřujeme vhodnost gramatiky pro paralelní syntaktické zpracování.

Kapitola 5

Množina $Pair(X)$

Množinu Pair() počítáme pro všechny neterminály pravé strany každého pravidla. Pouze tak zajistíme, že neztratíme závislost počátečního vymezovacího symbolu na koncovém. Množina obsahuje dvojice množin symbolů, které ve vstupním řetězci ohraničují řetězec generovaný z tohoto neterminálu.

Algoritmus vychází z výpočtu množin Follow a Precede

Algoritmus: 5.0.3 *Algoritmus pro výpočet množiny Pair():*
Množina Pair(X) je definovaná pro každý neterminál $X \in N$ a označuje množinu dvojic vymezovacích symbolů: $Pair = \{D(P_1); D(P_2); \ldots; D(P)i)\}$, kde $D(P_i) = \{d_s, d_e\}$, kde d_s je množina počátečních vymezovacích symbolů a d_e je množina koncových vymezovacích symbolů a $i = 1 \ldots num(P)$. DP_i je definována pro každé pravidlo.

Potom množina Pair(B) se získá následovně:

- *pro všechna pravidla $A \rightarrow xBy$ definujeme $D(A \rightarrow xBy) = \{d_s, d_e\}$:*
  - *if $x \neq \{\varepsilon\}$ then*
    přidej všechny symboly z Last(x) do d_s
  - *if $Empty(x) = \{\varepsilon\}$ then*
    přidej všechny symboly z Precede(A) do d_s
  - *if $y \neq \{\varepsilon\}$ then*
    přidej všechny symboly z First(y) do d_e
  - *if $Empty(y) = \{\varepsilon\}$ then*
    přidej všechny symboly z Follow(A) do d_e

Příklad: 5.0.1 *Mějme gramatiku G definovanou následovně:*
BKG $G = (N, T, P, S)$, *kde:*

- $N = \{< PROG >, < EXP >\}$
- $T = \{id, =, +, ;, [,]\}$
- $P = \{$
  $< PROG > \rightarrow main(id)\{< STLIST >\}$
  $< STLIST > \rightarrow < STAT >; < STLIST >$
  $< STLIST > \rightarrow \varepsilon$
  $< STAT > \rightarrow id =< ITEM >$
  $< STAT > \rightarrow write(id)$
  $< STAT > \rightarrow if(< EXPR >)then\{< STLIST >\}$
  $< ITEM > \rightarrow read(id)$
  $< ITEM > \rightarrow < EXPR >$
  $< EXPR > \rightarrow < EXPR > + < T >$
  $< EXPR > \rightarrow < T >$
  $< T > \rightarrow (< EXPR >)$
  $< T > \rightarrow id$
  $\}$
- $S =< PROG >$

Pro tuto gramatiku vypočítáme pro každý neterminál množinu Pair(). Množinu Pair() počítáme postupně pro každé pravidlo a proto se nám neztratí závislosti mezi počátečními a koncovými omezovacími symboly. Výsledek výpočtu množin Empty, First a Last je v tabulce 5.1.

X	Empty(X)	First()	Last()
< PROG >	$\emptyset$	{main}	{}}
< STLIST >	ε	{write, id, if}	{; }
< STAT >	$\emptyset$	{write, id, if}	{), }, id}
< ITEM >	$\emptyset$	{read, id, (}	{id,)}
< EXPR >	$\emptyset$	{id, (}	{), id}
< T >	$\emptyset$	{id, (}	{id,)}

Tabulka 5.1: Výsledek výpočtu množin Empty, First a Last

Pro každý neterminál pravé strany každého pravidla zde vidíme jejich příslušné množiny počátečních vymezovacích symbolů a koncových.

Pravidlo	< PROG >	< STLIST >	< STAT >	< ITEM >	< EXPR >	< T >
(1)	{^}, {$}	{{}, {}}				
(2)		{;}, {}}	{;{}, {;}			
(3)						
(4)				{=}, {;}		
(5)						
(6)		{{}, {}}			{(}, {)}	
(7)						
(8)					{=}, {;}	
(9)					{(=}, {+}	{+}{); +}
(10)					{(}, {)}	
(11)						
(12)						

Tabulka 5.2: Výsledek výpočtu množin Empty, First a Last

Nyní musíme tyto omezovací symboly vyhodnotit a ověřit, zda daný neterminál je vhodný k distribuci. Zvolme neterminál $< EXP >$, který nám označuje výrazy. Vidíme, že tento neterminál může být ve zdrojovém textu vymezen dvěmi dvojicemi vymezovacích symbolů: $\{\{=\}, \{;\}\}$ a $\{\{(\}, \{)\}\}$. Zde nám bohužel plynou dva závěry. Tím prvním je, že symboly $\{\{=\}, \{;\}\}$ mohou taktéž ohraničovat volání funkce $read(id)$, což nám přináší nejednoznačnost, kdy bychom kopírovali řetězec na vstup druhé komponenty v domnění, že se jedná o výraz.

Druhým nešťastným závěrem je, že jsme doplatili na nevhodně navrženou gramatiku. Jak nám ukazuje tabulka, dvojice symbolů $\{\{(\}, \{)\}\}$ vypadá, jako by byla jedinečná, tzn. že ohraničuje pouze neterminál $< EXP >$. Pokud si dobře prohlédneme pravidla gramatiky G vidíme, že jsou symboly $\{\{(\}, \{)\}\}$ použity např. v pravidle
$< PROG > \rightarrow main(id)\{< STLIST >\}$. Pokud bychom začali se symbolem (kopírovat řetězec, který po něm následuje, kopírovali bychom argumenty funkce main, což je opět nežádoucí chování. Vzhledem ke gramatice vidíme další nevýhodu plynoucí z této tabulky. Z neterminálu $< EXPR >$ jsme schopni generovat symboly '(' a ')'. Jak ale vidíme, tyto symboly nám také řetězce generované z neterminálu $< EXPR >$ ohraničují. Jak bychom tedy poznali, kdy se jedná o ohraničující symbol a kdy je ')' ještě součástí výrazu?

Zde vidíme, že takto zobrazené výsledky nepostačují a že bychom jich potřebovali více. Proto nyní na vstupní gramatiku klademe omezení 5.0.1. Tato podmínka nám zajistí, že se každý terminál nutně objeví v množině follow() nebo Precede() a budeme tak schopni následné automatické analýzy.

Podmínka: 5.0.1 *Nechť G = (N,T,P,S) je vstupní gramatika taková, že všechna pravidla mají tvar $N \rightarrow w$, kde $w \in (N \cup T)^*$ a nechť neexistuje rozklad $w = xaby$ takový, kde $x, y \in (N \cup T)^*$ a $a, b \in T$. Tedy nechť se nikde na pravé straně pravidla nevyskytují dva terminální symboly vedle sebe.*

Vstupní gramatiku přepracujeme tak, aby vyhovovala 5.0.1. Pro tuto upravenou gramatiku 5.3 vypočítáme opět množinu Pair 5.4. Seznam počátečních vymezovacích symbolů a koncových vymezovacích symbolů budeme v následujících tabulkách oddělovat znakem '-'.

(1)	$< PROG > \rightarrow main < FARG >< BODY >$	(10)	$< STAT > \rightarrow write < FARG >$
(2)	$< BODY > \rightarrow \{< STLIST >\}$	(11)	$< STAT > \rightarrow if < COND > then < BODY >$
(3)	$< FARG > \rightarrow (< PAR >)$	(12)	$< COND > \rightarrow (< EXPR >)$
(4)	$< PAR > \rightarrow id$	(13)	$< ITEM > \rightarrow read < FARG >$
(5)	$< PAR > \rightarrow \varepsilon$	(14)	$< ITEM > \rightarrow < EXPR >$
(6)	$< STLIST > \rightarrow < STAT >; < STLIST >$	(15)	$< EXPR > \rightarrow < EXPR > + < T >$
(7)	$< STLIST > \rightarrow \varepsilon$	(16)	$< EXPR > \rightarrow < T >$
(8)	$< STAT > \rightarrow id < A >$	(17)	$< T > \rightarrow (< EXPR >)$
(9)	$< A > \rightarrow = < ITEM >$	(18)	$< T > \rightarrow id$

Tabulka 5.3: Upravená vstupní gramatika

P	PROG	BODY	FARG	PAR	STLIST	A	STAT	COND	ITEM	EXPR	T
(1)	^ – $	) – $	main-}								
(2)					{-}						
(3)				(-)							
(4)											
(5)											
(6)					;-}		{-;				
(7)											
(8)						id-;					
(9)									=-;		
(10)			write-;								
(11)		then-;						if-then			
(12)										(-)	
(13)			read-;								
(14)										=-;	
(15)										(=-+	+-);
(16)											(=-);+
(17)										(-)	
(18)											

Tabulka 5.4: Přepočítaná množina Pair

Nyní jsme spočítali všechny možné vymezující symboly. Je velice těžké ad-hoc určit vhodnost neterminálu pro distribuci. Je potřeba mít formální nástroj kterým tento proces můžeme automatizovat. Pokud vstupní gramatika splňuje dříve popsanou omezovací podmínku, následující podmínka nám zaručuje jedinečnost vymezovacích symbolů.

Podmínka: 5.0.2 *Nechť G = (N,T,P,S) je BKG a její pravidla splňují podmínku 5.0.1. Poté platí, že:*

- *počáteční vymezovací symbol neterminálu E, kde $E \in N$, je unikátní, pokud pro všechny neterminály N gramatiky G, které mají tento počáteční symbol ve své Pair() množině, pro všechny jeho možné derivace platí: $N \Rightarrow^* Ew$, kde $w \in (N \cup T)^*$.*
- *koncový vymezovací symbol neterminálu E, kde $E \in N$, je unikátní, pokud pro všechny neterminály N gramatiky G, které mají tento koncový symbol ve své Pair() množině, pro všechny jeho možné derivace platí: $N \Rightarrow^* wE$, kde $w \in (N \cup T)^*$.*

Podle podmínky 5.0.2 se vymezovací symboly neterminálu E se mohou vyskytovat v Pair množině jiných neterminálů pouze tehdy, jde-li z tohoto neterminálu vyderivovat Ew případně wE. Tzn. z neterminálu, který obsahuje naše omezovací symboly musíme přímo derivovat náš neterminál. Pokud se stane, že neterminál generuje jiný řetězec, nejedná se o unikátní omezovací symboly a nemůžeme je tímto použít.

Analyzujme některé vymezovací symboly:

Příklad: 5.0.2 *Pro náš příklad zvolme neterminál $< EXPR >$, který generuje výrazy. Soustřeďme se nyní na* externí rozlišitelnost*, tj. na vymezovací symboly, které ohraničují $< EXPR >$ jako kořen výrazů (kontext pravidel (12) a (14)). Externí rozlišitelnost určuje, že symboly ohraničují vždy žádaný neterminál, tzn. pokud narazíme na tyto symboly tak vždy budou např. ohraničovat výraz.*

Jak vidíme máme zde dva páry vymezovacích symbolů $\{(,)\}$ a $\{=,;\}$. Důkaz nejedinečnosti těchto symbolů:

- *$\{(,)\}$ se vyskytují jako vymezovací symboly neterminálu $< PAR >$ (pravidlo (3)), tzn. symbol (se vyskytuje v množině Precede neterminálu $< PAR >$ a) se vyskytuje v množině Follow neterminálu $< PAR >$. Nyní musíme ve všech možných derivacích získat z neterminálu $< PAR >$ řetězec E (průnik Ew a wE),*
  aby bylo možno, aby $< PAR >$ obsahoval zároveň oba symboly. Bohužel jsme schopni generovat pravidlem $< PAR > \rightarrow$ id řetězec id, čímž získáváme nejednoznačnost. To takovou, že nám nyní symboly $\{(,)\}$ ohraničují jak parametr funkce (id) tak (výraz). Tyto symboly tedy nejsou jedinečné.
- *dalšími vymezovacími symboly zde jsou $\{=,;\}$. Stejně jako v předchozím případě zjistíme, že nám také vymezují neterminál $< ITEM >$. Nyní kontrolujeme, zda je splněna podmínka 5.0.2. Pravidlo (14) podmínku splňuje. Pravidlo (13) nám generuje*

řetězec read $< FARG >$, což nám podmínku porušuje. Ani tyto omezovací symboly nejsou unikátní.

Jak vidíme, v obou případech příkladu 5.0.2 je porušena externí rozlišitelnost. V obou případech nám tyto symboly ohraničovaly i jiné řetězce než pouze výrazy. Nemůžeme tedy tyto symboly využít. V příkladu 5.0.3 ještě zavedeme pojem interní rozlišitelnosti.

Příklad: 5.0.3 *Interní rozlišitelnost určuje, že ze zvoleného neterminálu nejsme schopni generovat ohraničující symboly. Tzn. že v případě, že nám např. daný výraz ohraničují závorky, tak tyto závorky nesmíme vygenerovat ze zvoleného kořenového neterminálu. Pokud bychom totiž byli schopni generovat, v případě že na tento neterminál narazíme, nejsme schopni určit, zda jsme narazili na ukončující vymezovací symbol, nebo zda daný symbol je ještě součástí výrazu.*

Jako příklad uveďme symboly $\{(,)\}$. Pravidlo (16) zavádí, že neterminál $< T >$ může být těmito symboly ohraničen. $< T >$ je generován z $< EXPR >$, řešíme tedy interní rozlišitelnost. Vidíme že $< T >$ generuje např. řetězce $(< EXPR >)$, čímž opět porušujeme podmínku 5.0.2.

Nyní byla vstupní gramatika upravena tak, aby předešla všem kolidujícím stavům a vyhověla tak interní i externí rozlišitelnosti. Byla změněna tato pravidla:

(12) $< COND > \rightarrow [< EXPR >]$
(14) $< ITEM > \rightarrow [< EXPR >]$

Tato pravidla nám generují tyto Pair() množiny uvedené v tabulce 5.5.

Jak vidíme, nemáme zde žádné kolidující stavy. Tuto gramatiku jsme tedy schopni použít jako modelovou gramatiku pro distribuovanou syntaktickou analýzu jejíž model bude předveden v následující kapitole.

Na základě zde uvedených algoritmů a omezení jsme schopni definovat jazyk, který je vhodný pro model distribuované syntaktické analýzy uvedený v této práci.

Definice: 5.0.3 *Nechť G = (N,T,P,S) je vstupní gramatika. Gramatika je vhodná pro distribuovanou syntaktickou analýzu, pokud existuje alespoň jedno $A \in N$ takové, že jeho vymezovací symboly splňují podmínku 5.0.2.*

V této kapitole jsme ukázali, že jsme schopni pro každý neterminál spočítat jeho vymezovací symboly, schopni automaticky určit, zda je konkrétní neterminál vhodný pro distribuci a obecně jsme schopni definovat množinu jazyků, které jsou vhodné pro model distribuované syntaktické analýzy uvedené v této práci.

P	PROG	BODY	FARG	PAR	STLIST	A	STAT	COND	ITEM	EXPR	T
Empty	∅	∅	∅	ε	ε	∅	∅	∅	∅	∅	∅
First	main	{	(	id	id,write,if	=	if,write,if	[	read,[	id,(	id,(
Last	}	}	)	id	;	],)	),}	]	],)	id,)	id,)
(1)	^ – $	) – $	main-}								
(2)					{-}						
(3)				(-)							
(4)											
(5)											
(6)					;-}		{-;				
(7)											
(8)						id-;					
(9)									=-;		
(10)			write-;								
(11)		then-;						if-then			
(12)										[-]	
(13)			read-;								
(14)										[-]	
(15)										(=-+	+-);
(16)											(=-);+
(17)										(-)	
(18)											

Tabulka 5.5: Množina Pair

Kapitola 6

Neunikátní vymezovací symboly

V předchozí kapitole jsme se zabývali unikátními vymezovacími symboly. Nyní opouštíme navrženou teorii a zamyslíme se nad možností existence neunikátních vymezovacích symbolů.

Unikátní vymezovací symboly vyžadují základní úpravy pravidel zásobníkových automatů. Jde o přidání pravidel, kdy při načtení počátečního vymezovacího symbolu zásobníkový automat přejde do copy stavu a při načtení ukončujícího vymezovacího symbolu přejde zpět do původního stavu. Unikátní vymezovací symboly splňují podmínku interní i externí rozlišitelnosti 5.0.2. Syntaktickou analýzu jsme všach schopni založit i na neunikátních symbolech.

6.1 Externě nerozlišitelné vymezovací symboly

V tomto případě řešíme situaci, kdy námi vybrané vymezovací symboly jsou interně rozlišitelné, ale nejsou externě rozlišitelné. Je to tedy situace, kdy nám například symboly = a ; vymezují jak např. výraz, tak volání funkce. Tuto situaci jsme schopni řešit tím, že se podíváme na následující symbol. Pokud jsme v situaci, kdy tento symbol jednoznačně rozliší gramatiky popisující řetězce mezi vymezovacími symboly, jsme schopni na základě tohoto symbolu rozhodnout, zda má zásobníkový automat přejít do kopírovacího stavu, či nikoli.

V praxi by to tedy znamenalo, že pokud narazíme na počáteční vymezovací symbol, podíváme se na následující symbol. Pokud tímto symbolem může začínat námi žádaný výraz, přejdeme do kopírovacího stavu a následující symboly kopírujeme na vstup dané komponenty. Pokud tímto symbolem začíná jiný řetězec než námi žádaný, vrátíme tento symbol na vstupní pásku a pokračujeme v analýze standardním způsobem. To, zda daný řetězec může začínat daným symbolem budeme zjišťovat z množin First a Last, případně Follow a Precede.

Jednoznačnou rozhodnost mezi gramatikami nám zaručí disjunktní množiny $First$ pro počáteční neterminály těchto gramatik.

Kapitola 7

Model distribuované syntaktické analýzy

Idea tohoto modelu vychází z principu necentralizovaných PC gramatických systémů, kdy více komponent spolu produkuje jeden řetězec. Jednotlivé komponenty spolu komunikují a posílají si generované výsledky.

Základní model distribuované syntaktické analýzy je totožný s modelem který byl uveden zde 3.3.1. Rozdíl je pouze ve vstupní gramatice. Jak vidíme, analýzou jedinečnosti omezovacích symbolů jsme se dostali zpět k našemu příkladu z úvodu této práce. Tento model bude v rámci dalšího zkoumání distribuované syntaktické analýzy implementován a testován.

7.1 Návrh

Jádro modelu spočívá v tom, že každá komponenta má svůj vlastní zásobníkový automat a zpracovává svůj vlastní řetězec. Tento řetězec dostane od řídící komponenty. Řídící komponenta má oproti klasickému systému navíc schopnost kopírovat kusy svého vstupu na vstupy jiných komponent. Tato vlastnost je závislá na existenci tzv. vymezovacích symbolů. Pokud komponenta ukončí analýzu svého vstupu, oznámí řídící komponentě že analýza je ukončena s daným výsledkem. Řídící komponenta tento výsledek zařadí na místo odkud pro danou komponentu kopírovala vstup.

Model uvedený zde 3.3.1 se opírá o unikátní vymezovací symboly. Úpravou tohoto modelu získáme model, kdy omezovací symboly nemusí být jedinečné. Řídící komponenta musí mít ale vlastnost podívat se na symbol za řídící komponentou a až na základě tohoto symbolu přejde do kopírovacího stavu. Tento model snižuje požadavky na vstupní gramatiku, avšak zvyšuje složitost systému. Otázkou tedy je, kdy je ještě výhodnější distribuovaný přístup, a kdy je míra režie distribuce již větší a tím méně výhodná.

7.2 Algoritmus převodu BKG na model distribuované SA

V 7.2.1 uvádíme algoritmus, kterým jsme schopni algoritmicky převést BKG na systém zásobníkových automatů, které distribuovaně zpracují syntaktickou analýzu.

Tento algoritmus je jedním z nejvýznamnějších výsledků této práce. Díky němu jsme schopni mechanicky analyzovat vstupní gramatiku a vybrat vhodné celky k distribuované analýze.

Algoritmus: 7.2.1 *Nechť G = (N,T,P,S) je BKG vstupní gramatika.*

- *Upravme pravidla vstupní gramatiky tak, aby splňovala podmínku 5.0.1*
- *Pro všechny neterminály N vypočítejme množinu Pair podle 5.0.3*
- *Vyberme ty neterminály, které splňují podmínku 5.0.2*
- *Tyto neterminály budeme nyní brát jako počáteční symboly nových gramatik. Nové gramatiky vytvoříme tím způsobem, že všechna pravidla korespondující k těmto novým neterminálům vyjmeme z původní gramatiky a vložíme do nově vytvářené gramatiky.*
- *Každá z takto nově vytvořených gramatik bude mít příslušnou komponentu nového distribuovaného systému.*
- *Torzo původní gramatiky bude tvořit řídící komponentu systému. Pro tuto komponentu vytvoříme zásobníkový automat podle algoritmu 3.3.1.*
- *Distribuovaný systém syntaktické analýzy je hotov.*

Kapitola 8

Praktická ukázka výpočtu

Předchozí příklady ukazovaly pouze hotové výsledky a jejich analýzy. V této kapitole uvedeme podrobný postup převodu vstupní gramatiky na distribuovaný překladač. Algoritmicky vypočteme vymezovací symboly, analyzujeme je a na základě analýzy provedeme návrh distribuovaného systému syntaktické analýzy. Zároveň ukážeme základní rozdíly mezi návrhem jednoduchého syntaktického analyzátoru a distribuovaného. Algoritmus převodu byl naznačen zde: 7.2.1.

Vytvořený model bude následně implementován a testován z hlediska časových nároků a výsledky budou porovnávány s jednoduchým modelem syntaktické analýzy.

8.1 Vstupní jazyk

Vstupní jazyk byl vytvořen tak, aby přinášel rozsáhlé řídící konstrukce, do kterých budou zakomponovány výrazy. Při analýze ukážeme veškeré aspekty výpočtu, zejména odlišné případy externích a interních rozlišitelností.

Vstupní gramatika je schopna popsat následující konstrukce:

```
my id;
my id = <ITEM>;
```

Dva způsoby deklarace proměnné. Druhý případ naplňuje proměnnou výrazem.

```
read id;
print id;
id = <ITEM>;
x = eval id;
break;
```

Tři vestavěné metody sloužící k načtení hodnoty, zápisu hodnoty a vyhodnocení výrazu. Rovněž jsme schopni do proměnné přiřadit $< ITEM >$, což reprezentuje celočíselnou hodnotu, řetězec, proměnnou, případně vyhodnocený výraz.

```
switch <E>
{
    case id:
        <COMM_LIST>
        break;
    case id:
        <COMM_LIST>
        break;
    case id:
        <COMM_LIST>
        break;
    default:
        <COMM_LIST>
        break;
}
```

Toto je případ rozsáhlé syntaktické konstrukce. Významově je totožná s obdobnou konstrukcí v jazycích C/C++/Java apod.

```
if <E> then
{
   <COMM_LIST>
}
elif <E> then
{
   <COMM_LIST>
}
else
{
   <COMM_LIST>
}
```

Zde je příklad větvení toku řízení. Konstrukce je mírně odlišná od konstrukcí v C/C++/Java. Zde pouze splňuje účel složitějšího syntaktického objektu.

Sémantická stránka popisovaného jazyka pro nás nemá zatím význam. Proto ji zde zcela vynecháváme.

Zde uvedený jazyk není příliš složitý. Zcela nám však slouží našemu účelu. Nebudeme se ztrácet ve výpočtu jednotlivých množin a v analýze vhodnosti symbolů. Navíc budeme schopni vytvořit rozsáhlé testovací konstrukce, které ukážou silné, či slabé stránky navrženého distribuovaného systému.

8.2 Gramatika popisující vstupní jazyk

Gramatiku popisující výše uvedený jazyk jsme navrhli tak, aby splňovala podmínku 5.0.1. Definovaná gramatika by mohla mít méně pravidel, ale podmínka nám říká, že každý terminál se musí objevit alespoň v jedné množině follow() nebo Precede(). Proto vstupní gramatika vypadá následovně:

$BKG \quad G = (N, T, P, S)$, kde:

- $N = \{\}$
- $T = \{\}$
- $P = \{$

(1) $< \text{COMM} - \text{LIST} > \rightarrow < \text{COMM} >; < \text{COMM} - \text{LIST} >$
(2) $< \text{COMM} - \text{LIST} > \rightarrow \varepsilon$
(3) $< \text{COMM} > \rightarrow \text{my} < \text{ID} >< \text{INIT} >$
(4) $< \text{COMM} > \rightarrow \text{read} < \text{ID} >$
(5) $< \text{COMM} > \rightarrow \text{print} < \text{ID} >$
(6) $< \text{COMM} > \rightarrow < \text{ID} >=< \text{ASSIGN} >$
(7) $< \text{COMM} > \rightarrow \text{break}$
(8) $< \text{COMM} > \rightarrow \text{switch} < \text{E} > \{< \text{SWITCH} - \text{BODY} >\}$
(9) $< \text{COMM} > \rightarrow < \text{IF} >< \text{ELIF} - \text{LIST} >< \text{ELSE} >$
(10) $< \text{INIT} > \rightarrow =< \text{ITEM} >$
(11) $< \text{INIT} > \rightarrow \varepsilon$
(12) $< \text{ASSIGN} > \rightarrow < \text{ITEM} >$
(13) $< \text{ASSIGN} > \rightarrow \text{eval} < \text{ID} >$
(14) $< \text{ITEM} > \rightarrow < \text{ID} >$
(15) $< \text{ITEM} > \rightarrow \text{string}$
(16) $< \text{ITEM} > \rightarrow \text{int}$
(17) $< \text{ITEM} > \rightarrow < \text{EVAL} - \text{E} >$
(18) $< \text{ID} > \rightarrow \text{id}$
(19) $< \text{EVAL} - \text{E} > \rightarrow \text{`} < \text{E} > \text{`}$
(20) $< \text{SWITCH} - \text{BODY} > \rightarrow < \text{CASE} - \text{LIST} >< \text{CASE} - \text{DEFAULT} >$
(21) $< \text{CASE} - \text{LIST} > \rightarrow < \text{CASE} >< \text{CASE} - \text{LIST} >$
(22) $< \text{CASE} - \text{LIST} > \rightarrow \varepsilon$
(23) $< \text{CASE} > \rightarrow \text{case} < \text{ID} >:< \text{COMM} - \text{LIST} >$
(24) $< \text{CASE} - \text{DEFAULT} > \rightarrow < \text{DEFAULT} >:< \text{COMM} - \text{LIST} >$
(25) $< \text{DEFAULT} > \rightarrow \text{default}$
(26) $< \text{IF} > \rightarrow \text{if} < \text{E} > \text{then} < \text{IF} - \text{BODY} >$
(27) $< \text{ELIF} - \text{LIST} > \rightarrow < \text{ELIF} >< \text{ELIF} - \text{LIST} >$
(28) $< \text{ELIF} - \text{LIST} > \rightarrow \varepsilon$
(29) $< \text{ELIF} > \rightarrow \text{elif} < \text{E} > \text{then} < \text{IF} - \text{BODY} >$

(30) $< \text{ELSE} > \rightarrow \varepsilon$

(31) $< \text{ELSE} > \rightarrow \text{else} < \text{IF} - \text{BODY} >$

(32) $< \text{IF} - \text{BODY} > \rightarrow \{< \text{COMM} - \text{LIST} >\}$

(33) $< \text{E} > \rightarrow < \text{E} > \& < \text{E} >$

(34) $< \text{E} > \rightarrow < \text{E} > | < \text{E} >$

(35) $< \text{E} > \rightarrow ! < \text{E} >$

(36) $< \text{E} > \rightarrow (< \text{E} >)$

(37) $< \text{E} > \rightarrow \text{i}$

(38) $< \text{E} > \rightarrow \#$

}

- $S =< PROG >$

V následujících výpočtech bude z důvodu přehlednosti výpočtu vynechán výpočet pro gramatiku generující podmínky. Tento výpočet uvedeme na konci kapitoly.

8.3 Výpočet množiny $Empty(N)$

Množinu Empty vypočteme na základě algoritmu 4.1.1. Pravidlo říká, že pokud jsme z daného neterminálu schopni vygenerovat prázdný řetězec, obsahuje jeho množina Empty symbol ε.

Při analýze pravidel v prvním kroku najdeme ta pravidla, která na pravé straně mají symbol ε. Vidíme, že takovým pravidlem je například pravidlo (30). Množina Empty neterminálu $< ELSE >$ obsahuje symbol ε. Obdobně postupujeme pro všechna ostatní pravidla. V další iteraci hledáma ta pravidla, která na své pravé straně obsahují řetězec takových symbolů, které ve své Empty množině obsahují ε a mohou být tedy všechny generovat prázdný řetězec. Proto neterminál na levé straně tohoto pravidla musí ve své množině Empty obsahovat ε symbol. Vidíme, že tento případ zde nenastává. Výpočet množiny Empty 8.1 je u konce.

8.4 Výpočet množin $First(N)$ a $Last(N)$

Nyní přicházejí na řadu množiny First a Last. Algoritmy jsou popsány 4.2.1 a 4.2.2. Tyto množiny opět vypočítáváme iterativně tak dlouho, dokud jsme schopni tyto množiny měnit. Významově množina First obsahuje ty symboly, kterými může začínat řetězec generovaný z neterminálu na levé straně pravidla. Naopak množina Last obsahuje ty symboly, kterými mohou tyto generované věty končit. Do množiny First daného neterminálu na levé straně pravidla tedy přidáváme ty symboly, které jsou v množině First prvního symbolu pravé strany pravidla. Pokud jsme schopni z tohoto symbolu generovat prázdný řetězec (jeho množina Empty obsahuje ε) přidáme do této množiny navíc symboly množiny First následujícího symbolu. Tento cyklus opakujeme. Obdobně je to u výpočtu množiny Last.

N	Empty(N)
< COMM − LIST >	ε
< COMM >	$\emptyset$
< INIT >	ε
< ASSIGN >	$\emptyset$
< ITEM >	$\emptyset$
< ID >	$\emptyset$
< EVAL − E >	$\emptyset$
< SWITCH − BODY >	$\emptyset$
< CASE − LIST >	ε
< CASE >	$\emptyset$
< CASE − DEFAULT >	$\emptyset$
< DEFAULT >	$\emptyset$
< IF >	$\emptyset$
< ELIF − LIST >	ε
< ELIF >	$\emptyset$
< ELSE >	ε
< IF − BODY >	$\emptyset$
< E >	...

Tabulka 8.1: Výpočet množiny Empty

8.4.1 $First(N)$

Jako příklad uvedeme výpočet neterminálu $< CASE\_LIST >$. Pokud v první iteraci řešíme pouze pravé strany pravidel začínající terminálem, nemáme nic, co bychom přidali
do množiny First. V případě neterminálu $< CASE >$ v tomto kroku přídáme *case*. Ve druhém kroku již do množiny First($< CASE\_LIST >$) přidáváme množinu First($< CASE >$)
a proto množina First($< CASE\_LIST >$) také obsahuje prvek *case*. Totožně pokračujeme u výpočtu množiny Last s tím rozdílem, že neanalyzujeme pravé strany pravidel od prvního symbolu, ale od posledního.

Výpočet na naší vstupní gramatice proběhl ve třech iteracích. Poté jsme již nebyli schopni tyto množiny měnit. Zde uvádíme výsledek výpočtu množin First a Last. Tabulka 8.2 ukazuje výsledek výpočtu množiny First.

8.4.2 $Last(N)$

Množinu Last jsme vypočítali ve třech iteracích. V nulté iteraci, která v tabulce není uvedena, jsme všechny množiny Last vyplnili $\emptyset$. V tabulce 8.3 je zobrazen průběh výpočtu množiny Last().

N	Empty(N)	0. iterace First(N)	1. iterace First(N)	2. iterace First(N)
< COMM − LIST >	ε	$\emptyset$		switch, break, print, read, my,
< COMM >	$\emptyset$	$\emptyset$	switch, break, print, read, my,	switch, break, print, read, my,
< INIT >	ε	$\emptyset$	eval,	eval, id, string, int, ‘
< ASSIGN >	$\emptyset$	$\emptyset$	=	=,
< ITEM >	$\emptyset$	$\emptyset$	int, string,	id, string, int, ‘,
< ID >	$\emptyset$	$\emptyset$	id	id,
< EVAL − E >	$\emptyset$	$\emptyset$	‘	‘,
< SWITCH − BODY >	$\emptyset$	$\emptyset$		case, default,
< CASE − LIST >	ε	$\emptyset$		case,
< CASE >	$\emptyset$	$\emptyset$	case	case,
< CASE − DEFAULT >	$\emptyset$	$\emptyset$		default,
< DEFAULT >	$\emptyset$	$\emptyset$	default	default,
< IF >	$\emptyset$	$\emptyset$	if	if,
< ELIF − LIST >	ε	$\emptyset$		elif,
< ELIF >	$\emptyset$	$\emptyset$	elif	elif,
< ELSE >	ε	$\emptyset$	else	else,
< IF − BODY >	$\emptyset$	$\emptyset$	{	{
< E >	...	$\emptyset$		...

Tabulka 8.2: Výpočet množiny First

N	Empty(N)	1. iterace Last(N)	2. iterace Last(N)	3. iterace Last(N)
< COMM − LIST >	ε		;,	;,
< COMM >	$\emptyset$	break,	{, break, id, string, int, ‘,	{, break, id, string, int, ‘,
< INIT >	ε		id, string, int, ‘,	id, string, int, ‘,
< ASSIGN >	$\emptyset$		id, string, int, ‘,	id, string, int, ‘,
< ITEM >	$\emptyset$	int, string,	id, string, int, ‘,	id, string, int, ‘,
< ID >	$\emptyset$	id	id,	id,
< EVAL − E >	$\emptyset$	‘	‘,	‘,
< SWITCH − BODY >	$\emptyset$		:,	:,;,
< CASE − LIST >	ε		:	:,;,
< CASE >	$\emptyset$		:	:,;,
< CASE − DEFAULT >	$\emptyset$		:	:,;,
< DEFAULT >	$\emptyset$	default	default	}
< IF >	$\emptyset$		}	}
< ELIF − LIST >	ε		}	}
< ELIF >	$\emptyset$		}	}
< ELSE >	ε		}	}
< IF − BODY >	$\emptyset$	{	{	}
< E >	...	...	...	...

Tabulka 8.3: Výpočet množiny Last

8.5 Výpočet množin $Follow(N)$ a $Precede(N)$

Množina Follow(X) je množina všech terminálů, které se mohou vyskytovat vpravo od X ve větné formě. Z množiny Follow budeme vycházet při hledání koncového vymezovacího symbolu. Množina Precede(X) je množina všech terminálů, které se mohou nacházet vlevo od X ve větné formě. Tato množina poslouží jako východisko výpočtu množiny počátečních vymezovacích symbolů.

Množinu Follow vypočítáme podle algoritmu 4.5.1. Jako příklad uvedeme výpočet pro neterminál CASE a CASE-LIST. Nejprve vezměme pravidlo (20). Do Follow(CASE-LIST) přidáme symboly z First(CASE-DEFAULT). Jde o symbol 'default'. Dále vidíme, že Empty(CASE-DEFAULT)=$\emptyset$, což znamená že je výpočet u konce. Nyní vezměme pravidlo (21). Do Follow(CASE) přidáme First(CASE-LIST), což je symbol 'case'. Dále vidíme, že Empty(CASE-LIST) = ε. Proto do Follow(CASE) přidáme Follow(CASE-LIST), tedy symbol 'default'. Neterminál CASE již není na žádné pravé straně pravidla a proto je výpočet u konce.

Množinu Precede počítáme podle algoritmu 4.5.2. Jako příklad uvedeme výpočet neterminálu CASE-LIST. CASE-LIST se nám na pravé straně vyskytuje v pravidlech (20) a (21). Podle pravidla (20) přidáme do Precede(CASE-LIST) symboly z Precede(SWITCH-BODY). Tato množina obsahuje symbol '{' podle pravidla (8). Dále podle pravidla (21) přidáme do Precede(CASE-LIST) symboly z množiny Last(CASE), což jsou symboly ';' a ':'. Precede(CASE-LIST) tedy obsahuje symboly: ';',':' a '{'.

V tabulce 8.4 je výsledek výpočtu obou těchto množin.

N	Follow(N)	Precede(N)
< COMM − LIST >	$, case, default, {	;,,:,{
< COMM >	;	; ;
< INIT >	;	=
< ASSIGN >	;	id
< ITEM >	;	=
< ID >	=, ;, :	my, read, print, ; ;, eval, =, case
< EVAL − E >	;	=
< SWITCH − BODY >	}	{
< CASE − LIST >	default,	{, :, ;
< CASE >	case, default	{, :, ;
< CASE − DEFAULT >	}	{, :, ;
< DEFAULT >	:	{, :, ;
< IF >	elif, ;,	;,
< ELIF − LIST >	else, ;,	}
< ELIF >	elif, else, ;	}
< ELSE >	;	}
< IF − BODY >	elif, ;, else	then, else,
< E >	{, ‘, then	switch, ‘, if, elif

Tabulka 8.4: Výpočet množin Follow a Precede

8.6 Výpočet množiny $Pair(N)$

Nyní se konečně dostáváme k jádru naší problematiky. Výpočtu množiny Pair. Tento výpočet provádíme postupně pro každé pravidlo zvlášť, abychom neztratili závislosti mezi množinami Precede a Follow a tímto také mezi počátečními vymezovacími symboly a koncovými vymezovacími symboly.

Postupně pro každé pravidlo a na všechny neterminály na jeho na pravé straně aplikujeme algoritmus 5.0.3. Zde je kompletní výpočet:

- (1) $< COMM - LIST > \rightarrow < COMM >; < COMM - LIST >$

N	d_s	d_e
$< COMM >$	^, ;, :, {	;,
$< COMM - LIST >$	;,	$, case, default, {

- (2) $< COMM - LIST > \rightarrow \varepsilon$

- (3) $< COMM > \rightarrow$ my $< ID >< INIT >$

N	d_s	d_e
$< ID >$	my,	=,;,:,
$< INIT >$	id	;

- (4) $< COMM > \rightarrow$ read $< ID >$

N	d_s	d_e
$< ID >$	read,	;,

- (5) $< COMM > \rightarrow$ print $< ID >$

N	d_s	d_e
$< ID >$	print,	;,

- (6) $< COMM > \rightarrow < ID >=< ASSIGN >$

N	d_s	d_e
$< ID >$	; ;,	=,
$< ASSIGN >$	=,	;,

- (7) $< COMM > \rightarrow$ break

- (8) $< COMM > \rightarrow$ switch $< E > \{< SWITCH - BODY >\}$

N	d_s	d_e
$< E >$	switch,	{,
$< SWITCH - BODY >$	{,	},

- (9) $< \text{COMM} > \rightarrow < \text{IF} >< \text{ELIF} - \text{LIST} >< \text{ELSE} >$

N	d_s	d_e
$< \text{IF} >$	$\wedge$,;,	elif, else, ;,
$< \text{ELIF} - \text{LIST} >$	},	else, ;,
$< \text{ELSE} >$	},	;,

- (10) $< \text{INIT} > \rightarrow =< \text{ITEM} >$

N	d_s	d_e
$< \text{ITEM} >$	=,	;,

- (11) $< \text{INIT} > \rightarrow \varepsilon$

- (12) $< \text{ASSIGN} > \rightarrow < \text{ITEM} >$

N	d_s	d_e
$< \text{ITEM} >$	=,	;,

- (13) $< \text{ASSIGN} > \rightarrow \text{eval} < \text{ID} >$

N	d_s	d_e
$< \text{ID} >$	eval,	;,

- (14) $< \text{ITEM} > \rightarrow < \text{ID} >$

N	d_s	d_e
$< \text{ITEM} >$	=,	;,

- (15) $< \text{ITEM} > \rightarrow$ string

- (16) $< \text{ITEM} > \rightarrow$ int

- (17) $< \text{ITEM} > \rightarrow < \text{EVAL} - \text{E} >$

N	d_s	d_e
$< \text{EVAL} - \text{E} >$	=,	;,

- (18) $< \text{ID} > \rightarrow$ id

- (19) $< \text{EVAL} - \text{E} > \rightarrow$ ‘ $< \text{E} >$ ‘

N	d_s	d_e
$< \text{E} >$	‘,	‘,

- (20) $< \text{SWITCH} - \text{BODY} > \rightarrow < \text{CASE} - \text{LIST} >< \text{CASE} - \text{DEFAULT} >$

N	d_s	d_e
$< \text{CASE} - \text{LIST} >$	{,	default,
$< \text{CASE} - \text{DEFAULT} >$	{,:,;,	},

- (21) $< \text{CASE} - \text{LIST} > \rightarrow < \text{CASE} >< \text{CASE} - \text{LIST} >$

N	d_s	d_e
$< \text{CASE} >$	{,:,;,	case, default,
$< \text{CASE} - \text{LIST} >$	:,;,	default,

- (22) $< \text{CASE} - \text{LIST} > \rightarrow \varepsilon$

- (23) $< \text{CASE} > \rightarrow$ case $< \text{ID} >:< \text{COMM} - \text{LIST} >$

N	d_s	d_e
$< \text{ID} >$	case,	:,
$< \text{COMM} - \text{LIST} >$	:,	case, default,

- (24) $< \text{CASE} - \text{DEFAULT} > \rightarrow < \text{DEFAULT} >:< \text{COMM} - \text{LIST} >$

N	d_s	d_e
$< \text{DEFAULT} >$	{, ;, :,	:,
$< \text{COMM} - \text{LIST} >$	:,	},

- (25) $< \text{DEFAULT} > \rightarrow$ default

- (26) $< \text{IF} > \rightarrow$ if $< \text{E} >$ then $< \text{IF} - \text{BODY} >$

N	d_s	d_e
$< \text{E} >$	if,	then,
$< \text{IF} - \text{BODY} >$	then,	elif,

- (27) $< \text{ELIF} - \text{LIST} > \rightarrow < \text{ELIF} >< \text{ELIF} - \text{LIST} >$

N	d_s	d_e
$< \text{ELIF} >$	},	elif, else, :,
$< \text{ELIF} - \text{LIST} >$	},	else, ;,

- (28) $< \text{ELIF} - \text{LIST} > \rightarrow \varepsilon$

- (29) $< \text{ELIF} > \rightarrow$ elif $< \text{E} >$ then $< \text{IF} - \text{BODY} >$

N	d_s	d_e
$< \text{E} >$	elif,	then,
$< \text{IF} - \text{BODY} >$	then,	elif, else, ;,

- (30) $< \text{ELSE} > \rightarrow \varepsilon$

- (31) $< \text{ELSE} > \rightarrow \text{else} < \text{IF} - \text{BODY} >$

N	d_s	d_e
$< \text{IF} - \text{BODY} >$	else,	;,

- (32) $< \text{IF} - \text{BODY} > \rightarrow \{< \text{COMM} - \text{LIST} >\}$

N	d_s	d_e
$< \text{COMM} - \text{LIST} >$	{,	},

- (33) $< \text{E} > \rightarrow$ ...

8.7 Gramatika podmínek

V předchozích kapitolách jsme se seznámili s výpočtem množiny Pair. Tuto množinu je třeba spočítat pro celou vstupní gramatiku a na jejím základě poté tuto gramatiku analyzovat a vybrat vhodné celky k distribuci.

My jsme se vybrali trochu jinou cestou. Tuto množinu jsme spočítali pouze pro část gramatiky. Gramatika již byla uměle navržena tak, abychom dospěli k výsledku, že řetězce generované z neterminálu $< E >$ jsou vždy ve vstupním řetězci vymezeny jedinečnými symboly a tedy že tyto řetězce jsou ideální pro distribuovanou syntaktickou analýzu.
Z tohoto důvodu jsme si dovolili z důvodu zjednodušení výkladu vynechat tuto část gramatiky. Nyní vše napravíme a doplníme výpočty množin Empty, First, Last, Follow a Precede pro řetězce generované z neterminálu $< E >$.

Pro zopakování uvádíme vstupní gramatiku výrazů:

- (33) $< \text{E} > \rightarrow < \text{E} > \& < \text{E} >$
  (34) $< \text{E} > \rightarrow < \text{E} > | < \text{E} >$
  (35) $< \text{E} > \rightarrow \, ! < \text{E} >$
  (36) $< \text{E} > \rightarrow \, (< \text{E} >)$
  (37) $< \text{E} > \rightarrow$ i
  (38) $< \text{E} > \rightarrow \, \#$

Výsledky výpočtu množiny Empty, First, Last, Follow, Precede je v tabulce 8.5. Výsledek výpočtu Pair je v 8.6

Set	Result
Empty(E)	∅
First(E)	#, i, (, !
Last(E)	#, i,)
Follow(E)	), |, &
Precede(E)	(, !, |, &

Tabulka 8.5: Výpočet množin Empty, Fist, Last, Follow a Precede pro gramatiku podmínek

pravidlo	N	d_s	d_e
(1)	$< E >$	(, !, |, &	&
(1)	$< E >$	&	), |, &
(2)	$< E >$	(, !, |, &	|,
(2)	$< E >$	|,	), |, &
(3)	$< E >$	!,	), |, &
(4)	$< E >$	(,	),

Tabulka 8.6: Výpočet množiny Pair pro gramatiku podmínkek

8.7.1 Analýza množiny Pair

Nyní již máme pro celou gramatiku vypočítanou množinu Pair. Úplná množina Pair je zobrazena v 8.7.

V ní je seznam neterminálů vstupní gramatiky a jejich počáteční a koncové vymezovací symboly. Našim cílem je najít takovou část gramatiky, který by byla vhodná k distribuovanému zpracování. Budeme hledat unikátní počáteční a koncové vymezovací symboly.

Jak jsme již zdůraznili dříve, gramatika byla navržena tak, abychom z této gramatiky byli schopni vyjmout ta pravidla, která generují podmínky. Tato analýza je podrobně rozebrána v 5 a proto se zde budeme soustředit pouze na analýzu neterminálu $< E >$, který je počátečním neterminálem gramatiky podmínek. Pokud chceme být schopni tento neterminál distribuovaně zpracovávat, musí být tzv. externě rozlišitelný. Musí ho tedy vymezovat unikátní vymezovací symboly.

Vezměme nejprve gramatiku podmínek jako celek. To znamená, že vyloučíme ty dvojice množin symbolů, které obsahují symboly, které jsme schopni z neterminálu $< E >$ generovat. Tyto symboly by totiž kolidovaly v interní rozlišitelnosti. Vidíme ale, že se žáden z těchto symbolů nevyskytuje mezi vymezovacími symboly a proto tyto symboly můžeme zanedbat a podmínka interní rozlišitelnosti je splněna.

Zůstávají nám tyto dvojice symbolů: $(switch, \{)$, $(`, `)$, $(elif, then)$, $(if, then)$. Zjišťujeme tedy, jestli jsou jejich počáteční symboly ($switch$, `, $elif$, if) unikátní. Z tabulky vidíme, že tyto symboly se nikde dále jako počáteční vymezovací. Můžeme tedy prohlásit, že pokud syntaktický analyzátor narazí na tento symbol, za tímto symbolem bude vždy následovat úsek kódu označovaný jako podmínka. Tento úsek bude končit příslušným koncovým vy-

N	d_s	d_e
< COMM − LIST >	;,	$, case, default, {
< COMM − LIST >	{,	},
< COMM − LIST >	:,	},
< COMM − LIST >	:,	case, default,
< COMM >	; ;, :, {	;,
< INIT >	id	;
< ASSIGN >	=,	;,
< ITEM >	=,	;,
< ITEM >	=,	;,
< ITEM >	=,	;,
< ID >	my,	=,;,:,
< ID >	read,	;,
< ID >	print,	;,
< ID >	; ;,	=,
< ID >	eval,	;,
< ID >	case,	:,
< EVAL − E >	=,	;,
< SWITCH − BODY >	{,	},
< CASE − LIST >	{,	default,
< CASE − LIST >	:,;,	default,
< CASE >	{,:,;,	case, default,
< CASE − DEFAULT >	{,:,;,	},
< DEFAULT >	{, ;, :,	:,
< IF >	;;,	elif, else, ;,
< ELIF − LIST >	},	else, ;,
< ELIF − LIST >	},	else, ;,
< ELIF >	},	elif, else, :,
< ELSE >	},	;,
< IF − BODY >	else,	;,
< IF − BODY >	then,	elif,
< IF − BODY >	then,	elif, else, ;,
< E >	switch,	{,
< E >	‘,	‘,
< E >	elif,	then,
< E >	if,	then,

Tabulka 8.7: Kompletní množina Pair

mezovacím symbolem. Neterminál $< E >$ je tedy externě rozlišitelný.

Analýzou jsme zjistili, že neterminál $< E >$ je vymezen unikátními vymezovacími symboly. Jsme tedy schopni navrhnout syntaktický analyzátor tak, že řetězce generované z neterminálu $< E >$ budeme zpracovávat distribuovaně.

8.8 Návrh distribuovaného analyzátoru

Předchozí analýza nám vstupní gramatiku rozdělila na dvě části. Na gramatiku podmínek, jejímž počátečním symbolem je neterminál $< E >$ a na zbývající část gramatiky, který generuje řídící struktury programu.

Tím, že máme formální model distribuovaného syntaktického analyzátoru, můžeme zvolit pro každou vstupní gramatiku nejvhodnější metodu analýzy. Pro gramatiku řídících struktuz zvolíme metodu top-down, pro gramatiku podmínek metodu bottom-up.

Lexikální analyzátor je společný pro oba syntaktické analyzátory. O inicializaci a obsluhu lexikálního analyzátoru se stará syntaktický analyzátor implementující metodu top-down, tedy řídící analyzátor. Po načtení počátečního vymezovacího symbolu top-down analyzátor kopíruje vstup na vstup bottom-up analyzátoru tak dlouho, dokud nenarazí na ukončovací vymezovací symbol. Poté top-down analyzátor odstraní ze zásobníku symbol označující počáteční neterminál gramatiky podmínek a pokračuje v analýze. Bottom-up analyzátor po ukončení analýzy vrací výsledek zpět top-down analyzátoru.

8.8.1 Řídící gramatika - metoda shora-dolů

Gramatiky řídících struktur se týkají pravidla 1-33 v 8.2. Pro tuto gramatiku budeme navrhovat analyzátor tak, jak je popsán v [1]. Analyzátor je řízen LL-tabulkou 8.8

Novou součástí kompilátoru je rozšíření o copy stav a kopírovací pravidla, která jsou uvedena v tabulce 8.9.

8.8.2 Gramatika podmínek - metoda zdola-nahoru

Gramatiky podmínek se týkají pravidla 34-38. Pro tuto gramatiku navrhneme precedenční analyzátor. Tvorba analyzátoru je popsána v [1]. Precedenční tabulka analyzátoru je zobrazena v 8.10:

Analyzátor bude analyzovat vstup, který mu zašle řídící analyzátor implementující metodu top-down. Po ukončení analýzy bottom-up analyzátor vrátí výsledek zpět řídícímu analyzátoru.

	if	switch	break	id	print	read	my	case	default	}	=	;	str	int	‘	eval	elif	else	{	$
< COMM – LIST >	1	1	1	1	1	1	1	2	2	2										2
< COMM >	9	8	7	6	5	4	3	2	2	2										
< INIT >											10	11								
< ASSIGN >				12									12	12	12	13				
< ITEM >				14									15	16	17					
< ID >				18																
< EVAL – E >															19					
< SWITCH – BODY >								20	20											
< CASE – LIST >								21	22											
< CASE >								23												
< CASE – DEFAULT >									24											
< DEFAULT >									25											
< IF >	26																			
< ELIF – LIST >												28					27	28		
< ELIF >																	29			
< ELSE >												30						31		
< IF – BODY >																			32	

Tabulka 8.8: LL-tabulka pro Top-down parser

přechod do copy stavu	*přechod z copy stavu*	*samotné kopírování*
switch S switch $\rightarrow$ C_1	{ C_1 { $\rightarrow$ S	C_1a $\rightarrow$ $C_1(, a)$
if S if $\rightarrow$ C_2	then C_2 then $\rightarrow$ S	C_2a $\rightarrow$ $C_2(, a)$
elif S elif $\rightarrow$ C_3	then C_3 then $\rightarrow$ S	C_3a $\rightarrow$ $C_3(, a)$
‘ S ‘ $\rightarrow$ C_4	‘ C_4 ‘ $\rightarrow$ S	C_4a $\rightarrow$ $C_4(, a)$

Tabulka 8.9: Kopírovací pravidla

	i	#	&	|	!	(	)	$
i			>	>			>	>
#			>	>			>	>
&	<	<	>	>	<	<	>	>
|	<	<	<	>	<	<	>	>
!	<	<	>	>	<	<	>	>
(	<	<	<	<	<	<	=	
)			>	>			>	>
$	<	<	<	<	<	<		

Tabulka 8.10: Precedenční tabulka

Kapitola 9

Implementace

Navržený distribuovaný syntaktický analyzátor byl implementován v jazyce C++. V této kapitole se seznámíme s jednotlivými implementovanými třídami a s jejich významem v rámci analyzátoru.

Analyzátor využívá dvě struktury, z nichž jedna definuje zásobníkový symbol a druhá token vrácený lexikálním analyzátorem.

- definice tokenu

```
typedef struct T_Token
{
    int type;
    string name;
    int    value;
};
```

- definice zásobníkového symbolu

```
typedef struct T_PdSymbol
{
    PD_SYMBOLS symbol;
    T_Token * token;
};
```

`PD_SYMBOLS` je výčtový typ definující všechny vstupní lexémy.

Třída `Lex` implementuje lexikální analyzátor. Ten čte vstupní řetězec a rozpoznává v něm tokeny. Jádrem lexikálního analyzátoru je konečný automat, který koresponduje s regulárními výrazy definující lexikální jednotky.

- `T_Token * getNextToken()`
  tato metoda vrátí další token ze vstupního řetězce

Třída `SymbolTable` udržuje tabulku symbolů a poskytuje operace pro práci s touto tabulkou. Tabulka symbolů je implementována jako datová struktura:

```
struct T_SymbolInfo
{
    int length;
};

typedef map< string, T_SymbolInfo > T_SymbolTable;
```

SymbolTable dále poskytuje tyto metody:

- `int installID( string lexema )`
  metoda uloží lexému do tabulky symbolů
- `T_SymbolTableIter getID( string lexema )`
  metoda vrátí token identifikovaný danou lexémou

`PushdownAutomaton` je třída reprezentující zásobníkový automat. Ten je implementován jako:

```
typedef vector<T_PdSymbol*> T_PDautomaton;
```

Pro operace nad zásobníkovým automatem `PushdownAutomaton` poskytuje tyto metody:

- `push( T_PdSymbol* symbol )` Na vrchol zásobníku je vložen nový zásobníkový symbol.
- `pop()` Z vrcholu zásobníku je odebrán zásobníkový symbol.
- `getTopmostTerm()` Tato metoda vrátí nejvrchnější neterminál. Zde je nutno poznamenat, že nejvrchnější nemusí nutně znamenat vrchol zásobníku. Zásobník totiž může obsahovat i neterminální symboly.
- `switchTop( vector<T_PdSymbol*> oldTop, vector<T_PdSymbol*> newTop )`
  Řetězec na vrcholu zásobníku oldTop je nahrazen novým řetězcem symbolů newTop.
- `switchSymbol( PD_SYMBOLS symbol, vector<T_PdSymbol*> newTop )` První nalezený výskyt daného symbolu je nahrazen řetězcem symbolů newTop.
- `compareTop( vector<T_PdSymbol*> top)` Metoda zjistí, zda se na vrcholu zásobníku nachází daný řetězec.

- `getSymbolsToReduce()` Tato metoda vrátí řetězec symbolů, který se nachází mezi vrcholem zásobníku a zásobníkovým symbolem $<$.
- `top()` Ze zásobníku je odstraněn symbol, který se nachází na vrcholu zásobníku.

Třída `PredictiveParser` je jádrem distribuovaného syntaktického analyzátoru. Zde je implementována metoda top-down analýzy. Tato třída zároveň zodpovědna za komunikaci s precedenčním analyzátorem, který je implementován třídou `PrecedenceParser`.

`PredictiveParser` analyzátor řídí své chování podle těchto tabulek:

- `ruleTable`: je tabulka pravidel vstupní gramatiky.
- `LLtable`: je tabulka LL-tabulka, která obsahuje identifikátory pravidel v závislosti na zásobníkovém symbolu a vstupním tokenu.

V konstruktoru `PredictiveParser::PredictiveParser()` jsou vytvořeny instance již zmíněných tříd: `SymbolTable`, `PushdownAutomaton` a `Lex`. Navíc je zde vytvořena instance třídy `PrecedenceParser`.

V metodě `runParsing` je implementován algoritmus, který řídí jak top-down analýzu, tak distribuci zdrojového kódu precedenčnímu analyzátoru. Pro ilustraci uvádíme neformální popis tohoto algoritmu:

```
* push($) & push(S) na zásobník & state = ANALYSIS
* repeat
  * nechť X je vrchol zásobníku a a je aktuální token
  * if( state == ANALYSIS):
      * case X of:
          * X == $:
            * if( a == $ ) then SUCCESS
                           else ERROR
          * terminal(X):
            * if( X == a ) then
                             * pop(X)
                             * if ( a == počáteční vymezovací symbol )
                                 state = COPY
                             * a = lex->getNextToken()
                           else
                             * ERROR
          * neterminal(X):
            * r = LLtable[X][a]
            * if( r == X->x ) then
                               * switchSymbol(X, reversal(x))
                               * zapiš na výstup r
```

```
                                    else
                                      * ERROR
    * if( state == COPY ):
        * if( a == ukončovací vymezovací symbol )
            * state = ANALYSIS
            * push_back(a)
            * precedenceParser->runAnalysis()
          else
            * kopíruj a na vstup precedenčního analyzátoru
  until SUCCESS or ERROR
```

Třída `PrecedenceParser` zajišťuje syntaktickou analýzu podmínek pomocí bottom-up metody. Svůj vstup získává od analyzátoru `PredictiveParser`.

Precedenční analyzátor je řízen opět tabulkou pravidel `ruleTable` a precedenční tabulkou `precedenceTable`. Nezávisle na top-down analyzátoru má precedenční analyzátor vytvořeny instance tříd `SymbolTable` a `PushdownAutomaton`.

Činnost analyzátoru je opět implementována v metodě `runParsing`. I zde uvedeme algoritmus, kterým precedenční analyzátor řídí svou činnost.

```
* push($) na zásobník
* repeat
    * a je aktuální znak na vstupu
    * b je terminál nejblíže vrcholu
    * case precedenceTable[b,a] of:
        * '=':
              * push(a)
              * a = lex->getNextToken()
        * '<':
              * switchSymbol(b, b<)
              * push(a)
              * a = lex->getNextToken()
        * '>':
              * if <y je na vrcholu zásobníku a existuje r: A->y
                  * switchTop(<y,A)
                  * zapiš na výstup r
                else ERROR
        * empty: ERROR
  until a == $ and b == $
```

Oba parsery spolu komunikují skrz sdílenou strukturu, která symbolizuje vstup `PrecedenceParseru`. Tato struktura je chráněna zámky-mutexy, takže vždy může přistu-

povat k této struktuře pouze jeden parser. Nad těmito zámky byl navíc postaven systém, který zajiťuje, že se oba parsery v přístupu k této struktuře střídají.

V průběhu analýzy `PredictiveParser` cyklicky kontroluje, zda `PrecedenceParser` nehlásí syntaktickou chybu. Pokud dojde k syntaktické chybě, překlad se ukončí. Pokud `PrecedenceParser` úspěšně ukončil syntaktickou analýzu, `PredictiveParser` má možnost převzít zpět generovaný intermediární kód.

Kapitola 10

Testování

Aplikace byla testována na testovacích zdrojových souborech, z nichž každý se vyznačoval speciální vlastností. Jednalo se o extrémní aspekty zdrojového kódu. Tyto výsledky byly porovnány s prediktivním analyzátorem, který zpracovával i gramatiku podmínek. Tento analyzátor vznikl pro testovací účely modifikací prediktivního analyzátoru navrženého v této práci.

Veškeré testovací soubory jsou přiloženy k této práci na CD. Jedná se o tyto soubory:

- *test3* obsahuje obsahuje velký počet zanořených struktur se zcela minimálním počtem krátkých podmínek.
- *test4* obsahuje velké množství extrémně dlouhých výrazů.
- *test5* obsahuje menší množství extrémně dlouhých výrazů vzhledem k řídícím strukturám.
- *test6* obsahuje vyvážený počet podmínek k řídícím strukturám.

Časové výsledky jednotlivých testů jsou shrnuty v tabulce 10.1. Čísla zde uvedená jsou časovým intervalem, po kterou analyzátor běžel.

input	sekvenční	paralelní
input3	13281	13460
input4	60619	62010
input5	20408	19099
input6	15464	12971

Tabulka 10.1: Výsledky testů

Z tabulky 10.1 vidíme, že distribuovaný syntaktický analyzátor se nejvíce uplatnil ve zdrojovém kódu, kde byly obě části zdrojového kódu vyváženy, tedy *input6*. Obě vlákna tedy běžela paralelně téměř po celou dobu činnosti.

Naopak paralelní analyzátor selhal v případě, kdy vstupní zdrojový kód obsahoval velké množství extrémně dlouhých výrazů (*input4*). Výpočet se prodlužoval o kopírování vstupu jedné komponenty druhé. Také v případě testu *input3* paralelní analyzátor měl horší výsledky. Zde docházelo k případu, kdy byly přenášeny krátké podmínky. Režie přenosu tak přesáhla efektivitu výpočtu.

Kapitola 11

Závěr

Celá tato práce se zabývala algoritmickým převodem a analýzou vstupní gramatiky na příslušný distribuovaný syntaktický analyzátor. Cílem bylo vytvořit takový model analyzátoru, který by byl schopen distribuovat zdrojový kód mezi více komponent, z nichž každá implementuje určitou metodu syntaktické analýzy, a tak byl schopen efektivního výpočtu.

V první části práce jsme uvedli ideální model distribuovaného syntaktického analyzátoru. Tento model jsme formálně definovali a předvedli jsme si jeho funkčnost. Příslušná vstupní gramatika byla vytvořena uměle tak, aby vyhovovala podmínkám modelu, tzn. aby obsahovala unikátní vymezovací symboly.

V další části práce jsme se dále zaměřili na obecné teoretické řešení. Vytvořili jsme algoritmy, díky kterým jsme schopni automaticky získat vymezovací symboly z jakékoli BKG gramatiky. Dále byly vytvořeny algoritmy, díky kterým můžeme automaticky vyhodnotit vhodnost těchto symbolů pro distribuovanou syntaktickou analýzu. Na základě získaných poznatků byly určeny podmínky, kdy je vstupní gramatika vhodná pro distribuované zpracování a kdy nikoli.

Touto částí jsme si vytvořili teoretické zázemí, díky kterému jsme schopni vyznačovat ve zdrojovém textu unikátní logické celky, které jsou schopny distribuovaného zpracování. Poté jsme tuto teorii upravili a nastínili jsme, jak obejít podmínku unikátnosti vymezovacích symbolů. Byl rozšířen základní model distribuované syntaktické analýzy o další schopnosti a tímto byly sníženy nároky kladené na vstupní gramatiku.

Poslední část práce ukázala praktické využití. Na navržené gramatice bylo prakticky ukázáno, jak pro vstupní gramatiku vytvořit podle navrženého algoritmu distribuovaný syntaktický analyzátor. Tento analyzátor byl následně porovnáván s nedistribuovaným prediktivním analyzátorem a výsledky byly analyzovány v závěru práce.

Tato práce má velký význam zejména v teoretické oblasti. Díky zde navrženým algoritmům jsme schopni automaticky analyzovat vstupní gramatiku a vytvořit pro ní distribuovaný syntaktický analyzátor. Jedním z výsledných produktů této teorie by mohl být prostředek, který analyzuje a případně převede BKG na SA.

Tato teorie může být základem pro další výzkum a to zejména v těchto oblastech:

1. V uvedeném systému není řešena sémantická stránka. Obvykle je syntaktická analýza spojena s analýzou sémantickou. Další součástí práce by tedy bylo upravit sémantickou analýzu tak, aby mohla být součástí tohoto distribuovaného řešení.

2. Další možností je vytvořit takový model, který by optimalizoval distribuci mezi jednotlivými kompomentami systému. Šlo by o automatické zjišťování meze, kdy režie přenosu zdrojového textu je vyšší než jeho efektivní analýza.

3. Jednou z možností je také formálně definovat model, který řeší externí nerozlišitelnost vymezovacích symbolů.

4. Formálně specifikovat třídu jazyků, které splňují v této práci uvedené podmínky nutné pro distribuovanou syntaktickou analýzu.

5. Je dokázáno, že PC gramatické systémy jsou schopny generovat složitější jazyky, než jaké jsou schopny generovat jejich komponenty samostatně. Součástí další práce by bylo možné také zkoumání síly této distribuované syntaktické analýzy, tedy specifikace třídy jazyků, kterou jsou schopny tyto analyzátory přijímat.

Závěr práce popisuje implementaci navrženého modelu. Z testování jsme zjistili, že k urychlení analýzy zdrojového textu dojde zejména v případě, pokud zdrojový soubor obsahuje vyvážený počet výrazů k řídícím strukturám.

Literatura

[1] Meduna A. *Automata and Language*. Springer, 2000. London.

[2] Ford B. Packrat parsing: Simple, powerful, lazy, linear time. `http://pdos.csail.mit.edu/papers/packrat-parsing:icfp02.pdf`, [cit. 2007-05-21].

[3] Csuhaj-Varjú E., Dassow J., and Kelemen J. *Grammar systems*. Gordon and Breach, 1994. London.

[4] Češka M. and Rábová Z. *Gramatiky a jazyky*. Skriptum FIT VUT Brno, 1992. Brno.

[5] Smrček J. *Syntax analysis based on multiple methods*. VUT FIT, 2006. Brno.

[6] Fegaras L. Bottom-up parsing. `http://lambda.uta.edu/cse5317/notes/node16.html`. [cit. 2007-05-21].

[7] Kwiatkowski L. Reconciling ungers parser as a top-down parser for cf grammars for experimental purposes. `http://www.cs.vu.nl/ steven/thesis-lukasz-kwiatkowski.pdf`, [cit. 2007-05-21].

[8] McLean P. and Horspool N. A faster earley parser. `www.csr.uvic.ca/ nigelh/Publications/fastEarley.pdf`, [cit. 2007-05-21].

[9] Smith T. Recursive descent, tail recursion, & the dreaded double divide. `http://www.ddj.com/dept/architect/184406384`, [cit. 2007-05-21].

[10] Li Te and Alagappan Devi. *A Comparison of CYK and Earley parsing algorithms*. Arizona state University, 2006. Arizona.