VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

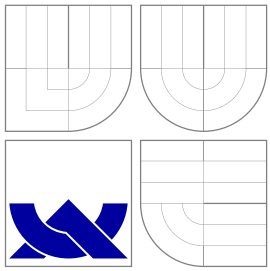
KOMBINOVANÁ SYNTAKTICKÁ ANALÝZA ZALOŽENÁ NA NĚKOLIKA METOD

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

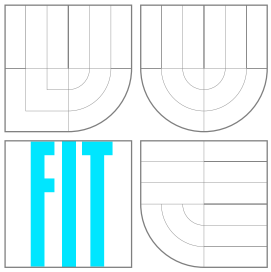
AUTOR PRÁCE
AUTHOR

LUDĚK DOLÍHAL

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

KOMBINOVANÁ SYNTAKTICKÁ ANALÝZA ZALOŽENÁ NA NĚKOLIKA METOD

COMPOSITE SYNTAX ANALYSIS BASED ON SEVERAL METHODS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — LUDĚK DOLÍHAL
AUTHOR

VEDOUCÍ PRÁCE — prof. RNDR. ALEXANDER MEDUNA, CSc
SUPERVISOR

BRNO 2007

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2006/2007

Zadání bakalářské práce

Řešitel: **Dolíhal Luděk**
Obor: Informační technologie
Téma: **Kombinovaná syntaktická analýza založená na několika metod**
Kategorie: Překladače

Pokyny:

1. Podrobně se seznamte s několika metodami syntaktické analýzy.
2. Navrhněte novou metodu syntaktické analýzy, která je založená na několika gramatikách. Diskutujte její přednosti a nedostatky.
3. Studujte užití navržené syntaktické analýzy. Navrhněte vhodný programovací jazyk a sestrojte jeho syntaktický analyzátor na bázi metody navržené v předchozím bodě. Testujte výsledný analyzátor.
4. Zhodnoťte dosažené výsledky a diskutujte další možný vývoj projektu.

Literatura:

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

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2.

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

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% 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: 1. listopadu 2006
Datum odevzdání: 15. 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

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: .....

..... Dal. Kaf .....

Nabyvatel Autor

Abstrakt

Hlavním cílem této práce je analýza tvorby komponentního syntaktického analyzátoru. Komponentním syntaktickým analyzátorem se zde myslí analyzátor, který je složen z několika vzájemně spolupracujících částí. V mém případě jsou to části dvě. V práci bych se chtěl zaměřit především na konstrukci jednotlivých částí překladače, dále na jejich vzájemnou komunikaci a spolupráci. Taktéž se pokusím obhájit, zda je vůbec potřebné a vhodné takový typ parseru vytvářet. V neposlední řadě pak bude analyzován jazyk jehož syntaktický analyzátor bude implementován zvolenou metodou.

Klíčová slova

Syntaktický analyzátor, komponentní syntaktický analyzátor, syntaktická analýza, kombinovaná syntaktická analýza, rekurzivní sestup, precedenční analýza.

Abstract

The main goal of this work is to analyze the creation of the composite parser. Composite syntactic parser is in this case a parser, which consists of more cooperating parts. In my case two parts. The work is focused on the construction of the parsers parts, on its cooperation and comunication. Then I will try to justify whether or not it is neccesary and suitable to create such a kind of parser. Last but not least I will analyse the language, whose syntactic analyser is to be implemented by the chosen method.

Keywords

Parser, Composite parser,syntax analysis, composite syntax analysis, recursive descent, operator precedence analysis.

Citace

Luděk Dolíhal: Kombinovaná syntaktická analýza založená na několika metod, bakalářská práce, Brno, FIT VUT v Brně, 2007

Kombinovaná syntaktická analýza založená na několika metod

Prohlášení

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

. . . . . .
Luděk Dolíhal
2. května 2007

Poděkování

Tímto děkuji prof. Alexandru Medunovi za podporu a motivaci v období tvorby této bakalářské práce.

Obsah

Kapitola 1

Úvod

S překladači programovacích jazyků se každý vývojář setkává dnes a denně. Jejich množství neustále narůstá stejně rychle, jako roste počet programovcích jazyků. Je tuďíž velmi pravděpodobné, že ani v budoucnu tento trh nezůstane stranou stále vyšším tempem se rozvýjejícího trhu informačních technologií. Pro mnohé však překladače nejsou jen nástrojem každodenního použití, ale tvoří, ať již přímo či nepřímo jejich pracovní náplň, nebo jsou objektem jejich zájmu případně výzkumu. V nedávné minulosti i v současnosti je jasný trend, stejně jako v ostatních odvětvích informačních technoligií, usnadnit tvorbu parserů do takové míry, aby je mohlo vytvářet více lidí, než jen úzká skupina specialistů. Existují aplikace typu YACC či Bison, které podle zadaných parametrů dokáží vygenerovat již hotový syntaktický analyzátor. Stejně jednoduše lze vygenerovat i lexikální analyzátor například pomocí nástroje Flex.

Hlavním cílem mé práce je prozkoumat možnosti rozkladu řetězce několika analyzátory, přičemž budu používat jak metodu shora dolů, tak metodu opačnou, zdola nahoru[5]. Jak uvidíme dále, mnou vybrané metody, rekurzivní sestup a precedenční analýza se vhodně doplňují. Bude tedy kladen důraz na to, aby každá z těchto metod zpracovávala tu část kódu, na které je efektivní. Vyjma tohoto se práce rovněž zabývá analýzou rozkládaného jazyka, protože parser nelze testovat bez jazyka, na němž bychom ho mohli vyzkoušet. Tato bakalářská práce navazuje na semestrální projekt, který byl vypracován v zimním semestru. V rámci projektu byly především vybrány a nastudovány vhodné metody.

Kombinace analýzy shora dolů a zdola nahoru vyžaduje důkladné pochopení obou metod, což odpovída rozdělení času mezi praktickou a teoretickou část zhruba v poměru 50:50. Lexikální analýza nebude implementována manuálně, ale pomocí programu Flex. Výstupem ze samotného překladače pak bude program ve formě abstraktního syntakticého stromu, nebo jiné vhodné struktury. Nebude řešeno ani generování mezikódu ani jeho případné optimalizace.

1.1 Poznámka k jazykovému aparátu

Vzhledem k faktu, že se v textu poměrně často vyskytují určitá slovní spojení typu ”rekurzivní sestup”, nebo ”lexikálni analyzátor”, rozhodl jsem se nahrazovat je jejich kratšími variantami. Místo lexikální analyzátor budu používat lexer, či AST místo abstraktní syntaktický strom. Na konci práce je pro úplnost uveden seznam zkratek.

1.2 Záměr práce

Jak již bylo zmíněno výše, hlavním záměrem mé práce je skloubit dvě metody syntaktické analýzy. Syntaktické analyzátory dělíme do dvou skupin a to podle směru, kterým pracují. Buď shora dolů, nebo zdola nahoru.

Top down parsing, neboli metoda shora dolů lze principielně rozdělit na tři hlavní algoritmy. Buď můžeme jít cestou brutální síly, která je sice nejobecnější, ale také nejpomalejší. Nebo můžeme použít metody rekurzivního sestupu, což je metoda, která díky rekurzi nevyžaduje zásobník, ale má exponenciální složitost a generuje hluboká zanoření. Třetí možností je použití rozkladu pomocí LL tabulky, která sice nezná problémy rekurzivního sestupu a počtem zanoření, ale je implementačně náročná. Především kvůli nutnosti vytvářet složité tabulky.

Cesta opačným směrem, tedy bottom up nám nabízí širší možnosti výběru. Buď můžeme využít precedenční analýzu, nebo některou z LR(n)[7] metod. Výhodou precedenční analýzy je její rychlost při rozkládání výrazů, na druhou stranu má řadu nevýhod. Zato LR(n) metody, ať už se jedná o LR(1), SLR(1), nebo LALR(1) jsou sice neobyčejně mocné, nicméně jejich implementace je značně náročná. N v závorce značí počet tokenů, které metoda využívá k ”výhledu” dopředu a podle nichž se rozhoduje, jaké pravidlo bude použito. Nejčastěji je n rovno 1, ale nezřídka nabývá i jiných hodnot.

Kapitola 2

Návrh

Tato část práce se zabývá konceptuálním návrhem vytvářeného syntaktického analyzátoru.

2.1 Syntaktický analyzátor a jeho úloha

Syntaktický analyzátor je typicky nejsložitější částí překladače. Jelikož se v dnešní době nejčastěji používá syntaxí řízený překlad, je kladen velký důraz na to, aby byl parser co nejrychlejší. Syntaxí řízený překlad znamená, že právě syntaktický analyzátor řídí činnost ostatních částí překladače. Například si volá lexer, když potřebuje další lexém, aby veděl jak má dále pokračovat v překladu a podobně. Vstupem parseru jsou obvykle lexémy, které produkuje lexikální analyzátor a jeho výstupem je pak AST. Vygenerovaný AST je strukturou, nad kterou můžeme provádět libovolné činnosti. Můžeme ji například optimalizovat, což je činnost, nad kterou můžeme strávit léta. Nad AST také obvykle pracuje sémantický analyzátor, či generátor mezikódu. Sémantika jazyka je sice v této práci zmíněna, avšak úkolem bylo vytvořit syntaktický analyzátor. Je jí tedy dáno méně místa než by si bezpochyby zasloužila a ani v programové části projektu není nijak řešena.

Vezmeme-li nejjednoduší případ, pak je syntaktický analyzátor jednotný algoritmus provádějící syntaktickou analýzu jedním směrem a v jednom průchodu vygeneruje AST, který reprezentuje vstupní program. Existují také víceprůchodové parsery, které vygenerují kompletní AST až po několika průchodech vstupního řetězce. Některé parsery, jako například parser jazyka Java, pak negenerují AST pro cílovou platformu, ale pro virtuální stroj, jehož hlavním cílem je odstínit případné odlišnosti jednotlivých platforem a dosáhnout tak lepší přenositelnosti kódu. V jistých případech pak dokonce parser ani nemusí vytvářet AST. Může totiž být rovnou interpretem daného jazyka a kód přímo vykonávat, jak tomu bylo například v případě projektu do ifj potažmo jazyka ifj05.

Mnou kostruovaný parser se ve většině vlastností neliší od běžných parserů. Jeho vstupem jsou lexémy, avšak výstupem není AST. Důvod je prostý. Nebudou totiž prováděny žádné další akce, pro které by bylo nutné AST vytvářet. Jeho výstupem programu bude pouze jednosměrně svázaný lineární seznam pravidel, která byla aplikována na vstupní soubor.Tato pravidla pak budou vypsána v případě, že zdrojový kód je validním vstupem kompilátoru, nebo seznam chybových hlášení a pravidla, která se podařilo rozpoznat v případě, že narazíme na chybu ve vstupním souboru. Zvláštnost parseru je především v tom, že sestává za dvou částí.

2.2 Kombinovaný parser

Proč vlastně vytvářet kombinovaný parser? Je samozřejmě možné využít některou z konvenčních metod, ať už z rodiny top-down, nebo bottom-up. Podívejme se tedy blíže na jednotlivé možnosti.

Bottom up parsing

Precedenční analýza Precedenční analýza je stejně jako většina metod zdola nahoru založena na tabulce. V angličtině je celý název této metody operator precedence parser. Jak je zřejmé z názvu, jedná se tedy o tabulku precedence operátorů. Algoritmus pro rozklad výrazu pomocí precedenční analýzy vychází z algoritmu pro převod infixové notace na notaci obrácenou polskou [5]. Devízou těchto parserů je jejich snadná implementace a pro drtivou většinu jazyků se velikost tabulky drží v rozumných mezích. Je tedy možné je konstruovat ručně. Další silnou stránkou této metody je, že dokáže velmi svižně rozkládat výrazy, což se určitě bude hodit. Mezi hlavní nevýhody této metody patří její "slabost". Pomocí této techniky se totiž dá rozložit pouze podmnožina LR(1) jazyků. To je také hlavním důvodem, proč tato metoda není v současnoti samostatně používána. Jak ovšem uvidíme dále, je vhodným doplňkem k jiným metodám.

LR analyzátory Tyto parsery umí rozkládat ve své podstatě všechny nejednoznačné bezkontextové gramatiky [5]. Procházejí řetězec zleva doprava a vytváří obrácenou nejpravější derivaci řetězce. Stejně tomu je i u precedenční analýzy. Metoda LR(k) se rozhoduje na základě obsahu parsovacího zásobníku a K sybolů výhledu. Tato metoda je také založena na tabulce. V tomto případě má tabulka dvě části. Akční a přechodovou. Konstrukce této tabulky je o dost složitější než v případě precedenční analýzy.

Top down parsing

Rekurzivní sestup Jednou z nejoblíbenějších metod top-down je právě rekurzivní sestup. Je to metoda, která nevyžaduje použití zásobníku, neboť právě rekurze nám zásobník "supluje". Při použití této metody je téměř vždy velmi dobře vidět v jaké fázi překladu se právě nacházíme. Tato metoda je velmi vhodná pro rozkládání příkazů a v neposlední řadě je implementace této metody vcelku intuitivní. Faktem ovšem zůstává, že chceme-li použít rekurzivního sestupu, musíme mít LL(k) gramatiku. Úprava gramatiky na gramatiku LL se však velmi často neobejde bez vytýkání, či jiného zbavování se levé rekurze. Za další stinnou stránku lze bezesporu považovat exponenciální složitost tohoto postupu, která se naplno projeví při rozkladu výrazů. V takovém případě není vzácností vygenerovat velmi hluboká zanoření na pěti či šesti úrovních závorek.

Další metody top down Další možnosti syntaktické analýzy shora dolů jsou například LL analyzátory with partial backup nebo with no backup [5]. Tyto metody jsou bezesporu velmi zajímavé a mocné. Jejich podrobnější popis by vydal na několik stránek. Nicméně v této práci si vystačíme s konstatováním, že tyto možnosti existují a jejich případné rozebrání necháme na pozdější dobu.

Kombinovaný parser Použití kombinovaného parseru není dnes ničím neobvyklým. Spousta jazyků používá parsery složené z více částí. Jako příklad nám může posloužit překladač jazyka Perl. Ten používá dokonce dva RD parsery a mezi nimi je vložen precedenční analyzátor. Zatím co RD parsery se starají o zachycení struktury příkazů, pomocí precedenční analýzy jsou rozkládány výrazy. Právě toto je dnes nejčastější použití precedenčního analyzátoru. Cílem mé práce je použít

vlastně shodný princip až na to, že v mém případě budou pouze dvě úrovně rozkladu. Bude kladen důraz na to, aby si parser udržel co možná nejvíce předností z obou metod a zároveň jsme se zbavili hlubokých rekurzí a omezili exponenciální časovou složitost čistého RD parseru. Jak je tedy vidět, tak kombinace právě těchto metod se ukázala více než vhodná. Obě metody se navzájem dobře doplňují. A to je i hlavním důvodem, proč vytvářet kombinované parsery. Abychom dosáhli lepších výsledků než při použití jedné metody. V některých případech dokonce nejsme schopni rozložit jazyk jedinou metodou, a proto se používá kombinovaných parserů. Kombinované parsery jsou nejčastěji složeny ze dvou částí, ale vyjímkou není ani více spolupracujících částí, jak ostatně dokládá již zmiňovaný překladač Perlu.

2.3 Komponenty syntaktického analyzátoru

Je samozřejmé, že komponenty parseru spolu musí vzájemně komunikovat. Vztah jednotlivých komponent velmi připomíná vztah klient server [3] známý především z prostředí sítí. Jelikož hlavní metodou rozkladu je metoda rekurzivního sestupu, je právě ona klientem, který žádá server (precedenční analýzu) o rozklad výrazu. Precedenční analýza v roli serveru přečte a rozloží výraz. Zpátky pak vrací buď token, který zůstal na zásobníku po provedení minimalizace, nebo chybové hlášení. Byly vytvořeny dvojice stopových prvků, při kterých se ujímá řízení jedna nebo druhá komponenta parseru. Jednotlivé dvojice budou rozebrány později.

Kapitola 3

Návrh jazyka

Aby bylo možné požadovaný syntaktický analyzátor sestrojit a testovat, byl navržen jednoduchý programovací jazyk. Jazyk je podobný jiným školním jazykům [4]. Jeho jméno je Simpl, což je zkratka slov SIMple Programing Language. Podle mého názoru se vyplatí jazyk pojmenovat už jen kvůli tomu, aby se na něj dalo jednoduše odkazovat. V následujících odstavcích bude probrána jak jeho lexikální a syntaktická struktura, tak částečně i jeho sémantika, která sice není součástí zadání, ale bez ní by to nebylo úplné. Simpl patří do skupiny imperativních jazyků podobně jako Pascal nebo C.

3.1 Lexikální struktura

Začněme jak jinak než lexikální strukturou jazyka.

Identifikátory

Identifikátor jako takový je libovolná sekvence znaků vyhovující regulárnímu výrazu `''[A-Za-z][A-Za-z0-9]*''`. Identifikátor tedy nesmí začínat číslicí!

Klíčová slova

Jako každý jiní jazyk i jazyk Simpl má svá klíčová slova. Množina klíčových slov tvoří podmnožinu identifikátorů. Jejich použití jako identifikátorů je však zakázáno. V řetězci se vyskytnout samozřejmě mohou.

```
if    then    else    for    repeat    until

true    false     read     print    while    do    .
```

Dále jsou za klíčová slova považovány i všechny typy.Takže ještě musíme přidat:

```
int    float    string    char    bool    void
```

Čísla

Simpl rozeznává dva druhy čísel. Jsou to čísla celá, vyhovující zápisu `''[0-9]*''`, dále pak čísla desetinná, která odpovídají regulárnímu výrazu `''[0-9]*\.[0-9]*''`. Je povolena i "vědecká" forma zápisu. Například oblíbený zápis v C typu 1.6 e08 zde taktéž projde.

Řetězce a znaky

Za znak respektive řetězec znaků se považuje jejich libovolná sekvence uzavřená v uvozovkách. Jak již bylo naznačeno výše, uvnitř takto uvedených výrazů se neprovádí žadná analýza, tudíž může obsahovat libovolné znaky.

Operátory,oddělovačě,závorky.

Jazyk rozeznává následující znaky jako operátory respektive oddělovače, či závorky. Všechny operátory jsou binární! V budoucnu by mohly být, například v rámci diplomové práce, dodělány i unární operátory. Nyní si vystačíme s tvrzením, že libovolný unární operátor lze vyjádřit pomocí binárních operátorů.

Operátory	+	-	*
	/	%	==
	!=	>	<
	>=	=<	&& a ||
Operátor přiřazení	:=	=	
Oddělovače	,	:	
Závorky	(,)	{ , }	

Komentáře

Ani jazyk jako je Simpl se nemůže obejít bez komentářů. Za komentář se považuje vše co je uzavřeno mezi znaky /* */ /*zde může být komentář libovolné délky*/. Tyto komentáře mohou být i víceřádkové a na program jako takový nemají žádný vliv. Pokud je tato sekvence znaků použita v řetězci, tak samozřejmě není považována za komentář.

3.2 Syntaktická a sémantická struktura

Po krátkém seznámení se s lexikální strukturou jazyka bude nyní přiblížena jeho syntax spolu se sémantikou. Budou představeny různé programové konstrukce od přiřazení přes řídící struktury až po funkce.
Syntaktická analýza je jádrem mé práce, proto jí bude věnován největší prostor v rámci specifikace jazyka. Jazyk Simpl lze zařadit do rodiny imperativních jazyků. Jednotlivé příkazy jsou tedy vykonávány jeden za druhým tak, jak jsou uvedeny ve zdrojovém kódu.

Proměnné

Jazyk Simpl je silně staticky typovaný. Silné statické typování dovoluje především při sémantické analýze testovat, zda například nepřiřazujeme do proměnné typu int hodnotu char. Toho s výhodou využijeme budeme-li v budoucnu psát sémantický analyzátor. V současné fázi projektu je to nepodstatné.
Pro dekalaci nové proměnné je možné využít buď Pascalovskou notaci *i,j : integer;*, nebo C notaci ve tvaru *int i,j;*. Zvolil jsem C zápis ve výše uvedené podobě. Ovšem s tím rozdílem, že každá proměnná je uvedena zvlášť. Správný zápis tedy bude vypadat *int i; int j;* Nově deklarovaná proměnná samozřejmě může být na témže řádku ihned inicializována , nebo může být ponechána bez

inicializace. Vzhledem k tomu, že známe její typ, lze ji velmi snadno kontrolovat. Neinicializovaná proměnná by se však měla vyskytnout pouze na levé straně operátoru přiřazení.

Příklad deklarace proměnné

*float x;/*deklarace proměnné v plovoucí čárce*/*
*float y:=50,6;/*deklarace další proměnné společně s inicializací*/*
*x:=x+6;/*chyba protože nevím co je v x*/*
*x:=y;/*ted se to dozvíme ale pozdě */*

Pokud jsme již deklarovali proměnnou a přiřadily jí jednou typ, není možné ji znovu deklarovat, respektive změnit její typ. To ovšem platí pouze pro proměnné v jednom lexikálním bloku.

float y:=50,6;
*int y:=5;/*konstrukce tohoto typu je tedy přirozeně považována za chybnou*/*

Lexikální bloky

Jazyk Simpl dovoluje deklaraci jak lokálních tak globálních proměnných. S tím souvisí existence lexikálních bloků. Téměř všechny imperativní jazyky znají tuto konstrukci. Procedury a funkce, stejně jako cykly obsahují lexikální blok implicitně v sobě. Tato skutečnost je vyjádřena i faktem, že jazyk(např. C) si vynucuje přítomnost složených závorek jak v cyklech, tak i v konstrukcích typu if či v procedurách a funkcích. Při deklaraci lexikálního bloku dochází vlastně k překrývání proměnných, kdy proměnná deklarovaná na vyšší úrovni je překryta proměnnou definovanou v lexikálním bloku, ve kterém se právě nacházíme. V praxi se tato skutečnost řeší vytvářením nových instancí tabulky symbolů. V Simplu je lexikální blok vyznačen pomocí složených závorek stejně jak v jazyce C.

Příklad lexikálního bloku

*string x:="ahoj";/*deklarace proměnné string*/*
*{/*zacatek bloku*/*
*int x:=50;/*nove x s hodnotou 50*/*
*print(x);/*vytiskne 50 */*
*}/*konec bloku */*
*print(x);/*vytiskne ahoj */*

Typy

S problematikou proměnných a lexikálních bloků velmi úzce souvisí problematika typů. Jazyk Simpl rozlišuje následující datové typy:

- ♢ int, typ reprezentující celá čísla
- ♢ float, datový typ pro desetinná čísla
- ♢ char, reprezentující jeden znak uzavřený do uvozovek

- ♢ string, řetězec znaků uzavřených do uvozovek
- ♢ bool, reprezentující logickou hodnotu. Může nabývat pouze hodnot truth a false.
- ♢ void, datový typ pro proceduru

Řídící struktury

V jazyce Simpl se vyskytují konstrukce a příkazy známé z klasických imperativních programovacích jazyků. Je to především konstrukce typu *if - else*, dále pak cykly *while,repeat until* a počítaný cyklus *for*. Za slovy while respektive until následuje podmínka. Výskyt čehokoli jiného je považován za chybu. Uvnitř cyklu for není možné deklarovat proměnnou tak, jak to známe například z jazyka C. Dále je nutné, aby byl každý výraz v cyklu for uzavřen středníkem. U cyklu repeat a while nejsou závorky nutné. Ovšem u cyklu for je minimálně jedna úroveň závorek povinná.

Příklad if a for

```
if(n > 1){...}/*složené závorky jsou povinné*/
else if(n=0){...}/*jednotlivé podmínky je možné vnořovat*/
else{
for(i:=0;i<10;i:=i+1;) /*příklad for cyklu*/
{
....
}
}
```

Příklad while a repeat until

```
while(a > 0) /*příklad while cyklu*/
{
....
}

repeat /*příklad repeat until cyklu*/
{
read(a);/*závorky jsou nutné*/
print (a*a);/*závorky nejsou podmínkou*/
}
until(a<0)
```

Ještě mi chybí doplnit dva repsektive čtyři důležité příkazy. Tím prvním je příkaz *return*, který se zde používá pro navrácení hodhoty z funkce. Jeho příklad bude uveden o pár řádků níže u popisu funkce. Další dva příkazy jsou také životně důležité a žádný vyšší programovací jazyk se bez nich neobejde. Jde o *read a print*.Načítat lze vždy jen jednu proměnnou uzavřenou do závorek, tisknout lze celý výraz se závorkami nebo bez. Příklad je uveden výše. Tím úplně posledním je pak operátor přiřazení, jehož použití již bylo presentováno a nijak se nevymyká zažitým zvyklostem.

Procedury a funkce

Simpl samozřejmě dovoluje deklaraci funkcí a procedur. Procedury a funkce se zapisují ve tvaru

```
typ_fce id(seznam parametrů)
{
}
```

Přičemž procedura se od funkce liší pouze tím, že její typ je void. Jak již bylo řečeno, tak hodnota se vrací z funkce pomocí příkazu *return*. Funkce má svůj speciální přiřazovač. Chceme-li tedy přiřadit do proměnné výsledek funkce, použijeme = na místo := používaného ve všech ostatních případech. Lze samozřejmě deklarovat i prototyp funkce a funkci případně proceduru používat ještě před tím, než bude nadefinována.

```
int increment(int s)
{
return(s+1);
}
int i=increment(2);/*zde pouzijeme operator= pro prirazeni vysledku fce*/
```

Prototyp funkce by pak vypadal násedovně

```
int increment(int s);
```

Tímto bych ukončil kapitolu zabývající se návrhem jazyka. Myslím, že jazyk byl na poměry bakalářské opráce navržen celkem velkoryse. Další kapitola se už bude zabývat reprezentací programu.

Kapitola 4

Reprezentace jazyka

V této kapitole se budu blíže zabývat tím, jak je zdrojový kód v jednotlivých částech programu reprezentován. Budeme se především věnovat lexikálním datům a poté syntaktickým datům.

4.1 Lexikální reprezentace

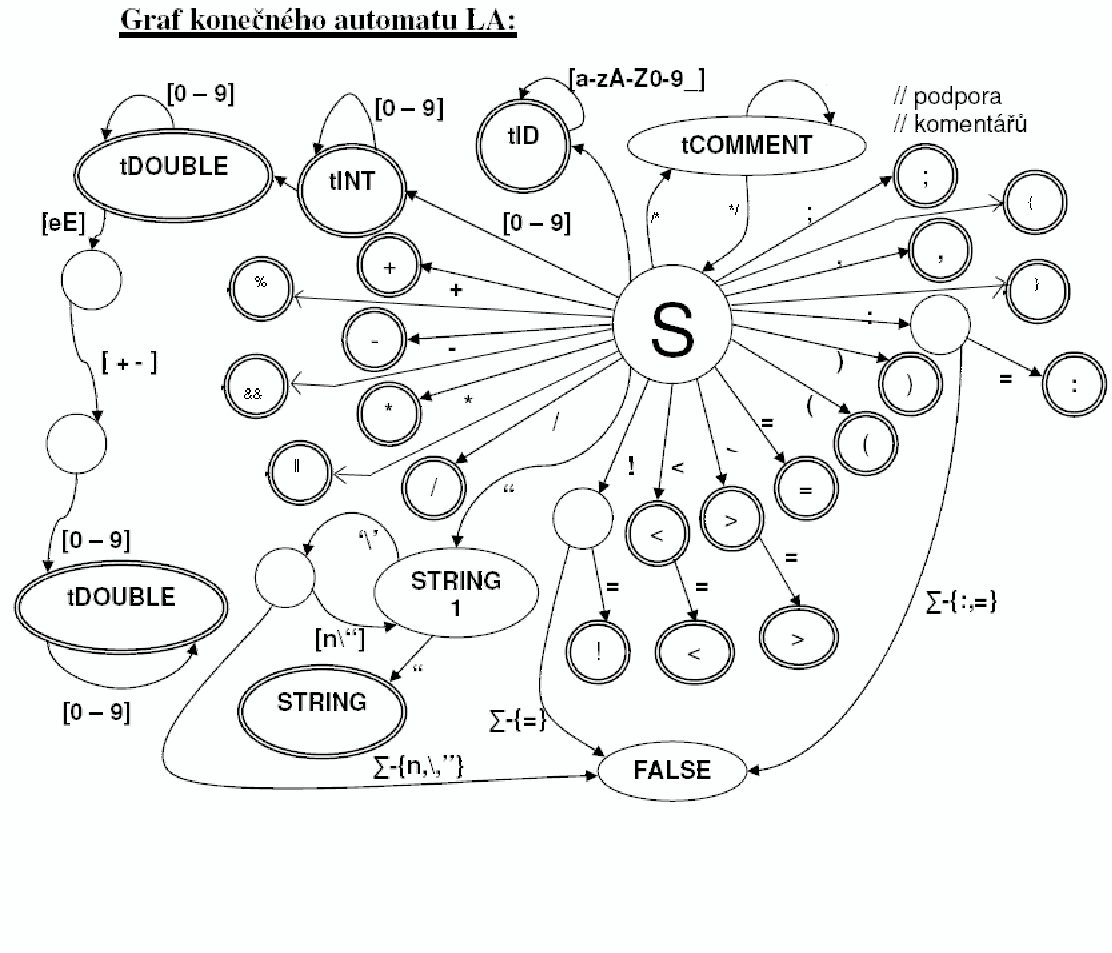
Na začátku své práce jsem sice uvedl, že lexer bude vygenerován pomocí nástroje GNU Flex, ovšem nakonec se tak nestalo. Po problémech s instalací nástroje a jeho následnou funkčností na 64 bitové platformě byl celý lexikální analyzátor napsán ručně. V architektuře překladače tvoří lexer první linii. Jeho vstupem je zdrojový kód a výstupem jsou pak lexémy. Lexémy jsou reprezentovány strukturou token. V dnešní době, kdy se uplatňuje takzvaný syntaxí řízený překlad, je lexikální analyzátor volán syntaktickým v případě, že je potřeba další token, abychom věděli, jak máme dál pokračovat v překladu. Cílem lexikálního analyzátoru je tedy určit typ lexému. Lexer dále vynechává prázdná místa a přeskakuje komentáře, aby se tyto případy odfiltrovaly co nejdříve a parser se jimi již nemusel zabývat. Samotný lexikální analyzátor není nic jiného než konečný automat jehož schéma je uvedeno níže. Vstupem syntaktického analyzátoru jsou pak tokeny, na jejichž strukturu se nyní podíváme.

Struktura tokenu

Token je tvořen strukturou, ve které je jednoznačně určen její typ. Tzn. jedná-li se o proměnnou nebo operátor atd. Dále je v některých případech nutné uchovávat hodnotu, která může být číslelná jedná-li se o numerál, nebo může být v podobě řetězce. Dá se říci, že lexikální analyzátor rozeznává tři hlavní typy lexému a komentáře:

- ♢ proměnné,
- ♢ operátory,
- ♢ literály,
- ♢ komentáře,

Při načtení identifikátoru je také okamžitě testováno, zda nebylo načteno klíčové slovo. Pokud se tak stalo, je reprezentováno odpovídajícím typem. Další věcí, o kterou se stará lexikální analyzátor, je kurzor. Jedná se v podstatě o jednoduché počítadlo řádků, abychom věděli, kde ve zdrojovém kódu se právě nacházíme. Toto počítadlo je velmi užitečné například při výpisu chyb.



Obrázek 4.1: Schéma lexikálního analyzátoru

Pozn. Graf není kompletní. Chybí například stavy před logickými spojkami && a ||. Graf slouží pouze pro představu jak lexer funguje.
struktura tokenu:
typedef struct tTOKEN
{
tTOKENT type;
*void *value;*
} **tTOKENptr;*
kde tTOKENT je výčtový typ určující jakého typu daný token je.

4.2 Syntaktická reprezentace

Druhou linii ve struktuře překladače již tvoří syntaktický analyzátor.

Reprezentace stavů

Vstupem syntaktického analyzátoru jsou tedy tokeny vyžádané od lexikálního analyzátoru. Ve většině případů bývá výstupem AST, nad kterým se pak provádí další operace, především pak sémantická analýza a optimalizace. Výstupem mého programu je naproti tomu pouze posloupnost pravidel v pořadí, v jakém byla aplikována na zdrojový text. Byla tedy vytvořena struktura, do které je možno uložit, jaká operace byla právě provedena. Každá operace je reprezentována příslušným stavem a syntaktické zpracování celého zdrojového kódu pak vytvoří jednosměrně svázaný seznam těchto struktur s ukazatelem na právě aktivní prvek.

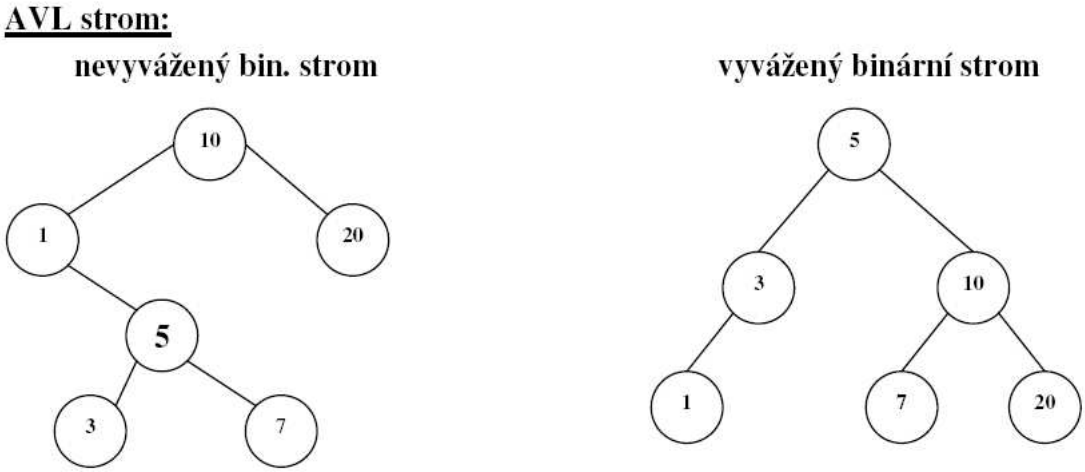
Struktura jednoho prvku seznamu pak vypadá následovně:
typedef struct t3ad
{
*struct t3ad *next3ad; // pointer na dalsi instrukci/*
tTOKENTYPE oper; // operace
tTOKENptr op1; // operand1
tTOKENptr op2; // operand2
pNode result; // vysledek
int line;
} *t3adptr;*
Pozn. Vzhledem k tomu, že máme ukazatele na oba operandy, nebylo by složité provádět do jisté míry sémantické kontroly. Můžeme například ošetřit, zda nesčítáme string s číslem a podobně. Na některé sémantické prohřešky je však tato metoda krátká.

Jednoduchým zřetězením takovýchto struktur pak získáme zmiňovaný seznam:
typedef struct { *t3adptr first; // pointer na prvni instrukci*
t3adptr active; // pointer na provadenou instrukci
t3adptr last; // pointer na posledni instrukci
} *t3adlist;*

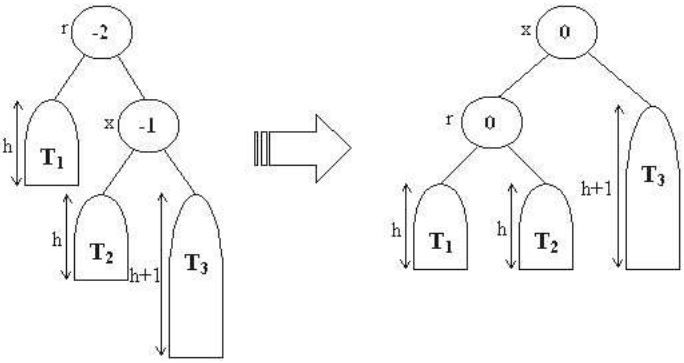
Tabulka symbolů

Tabulka symbolů stojí sice poněkud stranou, nicméně žádný syntaktický analyzátor se bez jejích služeb neobejde. Tabulka symbolů je struktura, která uchovává informaci o všech proměnných deklarovaných v daném programu. Má tabulka symbolů je provedena v podobě AVL stromu[6]. AVL strom je pojmenován po svých vynálezcích G.M. Adelson-Velsky a E.M. Landis, kteří jej poprvé publikovali roku 1962. Ve své podstatě jde o samovyrovnávací binární vyhledávácí strom. V AVL stromu se výška synovských podstromů jakéhokoli uzlu může lišit maximálně o 1, tuďíž jej můžeme nazvat výškově vyrovnaným. Vyhledávání, vkládání a mazání záznamu z takovéto struktury má logaritmickou složitost.

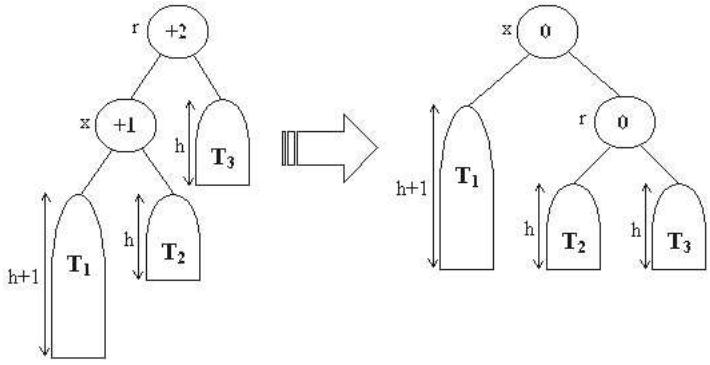
Obrázky ukazují jak vypadá AVL strom, dále jednoduchou levou a pravou rotaci. Existují i složitější operace, které ale nebudu uváďet. O perace nad tabulkou symbolů jsou implementovány rekurzivně. Jedná se o operace vkládání, vyhledávání a uvolnění celé tabulky symbolů po skončení programu.



Obrázek 4.2: Příklad AVL stromu



Obrázek 4.3: Jednoduchá pravá rotace [1]



Obrázek 4.4: Jednoduchá levá rotace [1]

Z výše uvedeného popisu AVL stromu a znalostí struktury, kterou máme uchovávat, můžeme velmi snadno vyvodit, jak bude vypadat jeden uzel tabulky symbolů. Protože se jedná o binární vyhledávací strom, musíme uchovávat ukazatel na levý a pravý podstrom, dále hodnotu, podle které vyhledáváme, kvůli vyvažování potřebujeme výšku, pak také samotná data a určitě se bude hodit i hodnota, zda je proměnná již definována či nikoli.

Přesná definice struktury je zde:
typedef struct tNode {
*struct tNode *left; // ukazatel na levy list nebo podstrom*
*struct tNode *right; // ukazatel na pravy list nebo podstrom*
int height; // vyska stromu tohoto uzlu, strom s jedinym uzlem ma vysku 0
tKey key; // klic uzlu
tTOKENptr token; // data uzlu
bool defined;
bool declared;
*} tNode, *pNode;*

Návrh řešení lexikálních bloků

S tabulkou symbolů velmi úzce souvisí lexikální bloky, které jazyk Simpl rozeznává. V zásadě existují dvě možnosti, jak tento problém řešit. Buď mít pouze jednu tabulku symbolů pro celý program a v ní vytvářet sekce s každým novým vytvořeným blokem. Pokud se pak budeme ptát na nějakou proměnnou, budeme prohledávat nejprve aktuální sekce a nenajdeme-li, tak i sekce ostatní. Tato koncepce je všeobecně uznávaná a zřejmě i nejpoužívanější. Další možností je vytvořit si tabulku pro každý nový blok a takto vzniklé bloky pak uchovávat v jiné struktuře, například je pushovat na zásobník. Pak tedy na úplném vrcholu zásobníku je blok, ve kterém se nacházíme a na úplném dně pak tabulka na globální úrovni. Tato koncepce je poněkud složitější. Jelikož však jsou lexikální bloky vlastně jednou z úrovní sémantické analýzy, není v rámci bakalářské práce řešena. Jedná se o pouhý nástin možného řešení této problematiky.

Kapitola 5

Implementace překladače

Po návrhu jednotlivých struktur, které bude překladač využívat, už tedy zbývá pouze samotný překladač naprogramovat. Což by měla být ta nejzajímavější část. Uvidíme, zda-li tomu tak opravdu bude. Implementace překladače je náplní této kapitoly. Nejdříve se podíváme na algoritmy, které se podílejí na rozkladu.

5.1 Algoritmy podílející se na překladu

Na rozkladu zdrojového kódu se vlastně podílejí tři algoritmy. Je to lexikální analyzátor, dále pak parser pracující metodou shora dolů a parser pracující zdola nahoru. Postupně si je všechny projdeme.

Lexikální analýza

Lexer se ke zpracování dostane jako první a jeho úloha již byla popsána výše. Pro zápis lexikální struktury jazyka lze použít regulární výrazy. Jsou to právě regulární výrazy, které předkládáme programům typu Lex či Flex. Ty pak z regulárních výrazů vygenerují konečný automat, neboť právě regulární výrazy jsou ekvivalentní konečnému automatu. V mém projektu byl lexikální ananlyzátor implementován ručně.

Syntaktická analýza shora dolů

Metody rozkladu byly zmíněny již několikrát, nicméně zde se určitě hodí si je znovu připomenout. Při postupu shora dolů můžeme použít buď rekurzivní sestup, nebo rozklad pomocí LL tabulky. Oba dva postupy jsou stejně mocné. Oběma je možné rozkládat LL(k) jazyky, jež tvoří podtřídu jazyků bezkontextových. Nicméně pro rekurzivní sestup hovoří jeho jednoduší implementace. Nemusíme totiž vytvářet tabulky mnohdy vskutku mamutích rozměrů atd.

Syntaktická analýza zdola nahoru

Analýza zdola nahoru nám nabízí velmi mnoho algoritmů. Od nejjednoduššího precedenčního parseru až po SLR či GLR analyzátory. Právě pro tvorbu parserů pracujících zdola nahoru se používají populární nástroje Yacc nebo Bison. Sice nejjednoduší, ale také nejslabší precedenční analyzátor je implementován v mém projektu.

```
"//".*                          # preskočit komentáře
    { }
for                             # klíčové slovo "for"
    { token( KWFOR ) }
[0-9]+                          # celá čísla
    { token( LITINTEGER ) }
[A-Za-z_][A-Za-z0-9_]*          # identifikátory
    { token( IDENTIFIER ) }
[0-9]+[A-Za-z_]+[A-Za-z0-9_]* # neplatné identifiátory
    { error( "Identifier must not begin with a number." ) }
" "+|\t+ {}                     # přeskočit bílé místo
    { }
.                               # neplatné znaky
    { error( "Invalid character on input." ) }
```

Obrázek 5.1: Příklad Lex kódu

5.2 Lexikální analýza

Lexikální analyzátor má dva hlavní úkoly. Jednak posílat tokeny a jednak se stará o posouvání počítadla lineCount, které používám pro hlášení chyb. Vzhledem k tomu, že známe regulární výrazy generující jednotlivé typy lexém, je vytvoření lexikálního analyzátoru hračka. Při hlubším zamyšlení zjistíme, že psaní lexikálního analyzátoru se velmi podobá psaní parseru rekurzivním sestupem. V obou dvou případech existují určitá pravidla, kterým odpovídají jisté akce. Rozdíl je pouze v tom, že lexer se rozhoduje podle toho, co je napsáno ve zdrojovém kódu, a parser pak podle typu lexémy. Přesně tak je naiplementován můj lexikální analyzátor. Podle toho, co načítám na vstupu, se dostávám do určitého stavu, ze kterého buď vrátím token, je-li konstrukce na vstupu správná, nebo se dostávám do jiného stavu. Například jsem-li ve stavu, že jsem načetl znak : , může to být buď děleno, nebo následuje = a pak se jedná o operátor přířazení. Následuje-li jiný znak jedná se o chybu. Ve své podstatě jde akorát o přepsání konečného automatu do daného programovacího jazyka.

Pokud bychom chtěli generovat lexikální analyzátor pomocí Flexu, vypadal by zápis pravidel jako na obrázku 5.1. Jde pouze o demonstrativní zápis. Nejsou zachycena všechna pravidla ani se nejedná o kus skutečného kódu.

Lexikální analyzátor dále vynechává bílá místa ve zdrojovém kódu a ignoruje komentáře uvozené znakem /* a ukončené sekvencí */.

Zpracování chyb

Výskyt chyb se nevyhýbá ani lexikální analýze. Nejjednoduším případem je výskyt znaku, který ve vstupním souboru nemá co dělat. V takovém případě je řešení jednoduché, přejdu do chybového stavu. Typ tokenu nastavím na chybovou hodnotu k tomu určenou a vrátím řízení syntaktické analýze. Co ale v případě, že se vyskytnou jiné chyby vzniklé například nedbalým zápisem? Zatím jsou tyto případy řešeny stejně jako výskyt chybného znaku, ale v budoucnu by mohla být implementována nějaká vhodnější metoda.

5.3 Syntaktická analýza

V této části bude nejdříve věnována jedna sekce rozkladu pomocí precedenční analýzy a poté se podíváme na rozklad pomocí rekurzivního sestupu.

Rozklad pomocí precedenční analýzy

Precedenční analyzátor je nejjednodušší z rodiny metod pracujících zdola nahoru. Ten principielně vychází z algoritmu převodu infixové notace na notaci postfixovou též zvanou obrácenou polskou. Je to metoda založená na tabulce precedence operátorů, odtud také pochází její název.

Precedenční analýza pracuje ve dvou režimech. V tom prvním jsou na zásobník postupně pushovány tokeny, které přicházý od lexeru. V tom druhém je na vrcholu zásobníku pravá strana nějakého pravidla a právě podle něj je řetězec na zásobníku minimalizován.

Docela zajímavá situace nastává v okamžicích, kdy se přepínáme mezi stavy. K jejich přepnutí dochází v okamžiku, kdy symbol, který je na zásobníku, musí být minimalizován dříve než symbol, který je aktuálně na vstupu. Minimalizace probíhá tak dlouho, dokud tento stav platí. V momentě, kdy přestane platit, se přepneme zpátky do stavu, kdy pushujeme na zásobník. Takto provádíme redukci, dokud nám nezůstane na zasobníku samotný startovací symbol gramatiky. Pak je analýza hotova.

Z výše uvedeného je tedy jasné, že mezi jednotlivými symboly gramatiky musí být zavedena relace precedence. Tato relace může nabývat jedné ze tří hodnot respektive čtyř hodnot, budeme-li uvažovat prázdné políčko za hodnotu.

- $\diamondsuit$ = značí, že oba dva symboly budou redukovány současně, push na zásobník
- $\diamondsuit$ $<$ značí, že stále načítáme pravou stranu pravidla, tudíž push na zásobník
- $\diamondsuit$ $>$ značí, že jsme na hranici pravé strany pravidla. Obsah zásobníku musí být minimalizován před tím než bude symbol ze vstupu přidán na zásobník.
- $\diamondsuit$ ' ' prázdné místo se dá využít pro chybová hlášení

Tabulka precedence je pak dvourozměrnou strukturou indexovanou v jednom směru terminálními a v druhém neterminálními symboly. Příklad, jak taková precedenční tabulka vypadá, je uveden níže. V každé buňce tabulky je pak jeden ze čtyř symbolů uvedených výše. Jejich použití pak vypadá následovně:

1. Nechť $G = (V, \Sigma, P, S)$ jde gramatikou rozkládaného jazyka a abeceda zásobníku je $\Gamma = N \cup T$
2. Nechť n_1 je symbol na vrcholu zásobníku a n_2 je symbol na vstupu $n_1 \in \Gamma, n_2 \in T$
3. pokud $n_1 = n_2$ nebo $n_1 < n_2$ ulož n_2 na zásobník
4. pokud $n_1 > n_2$ proveď redukci podle následujícího algoritmu

Algoritmus pro minimalizaci je následující:

push $ na zásobník
repeat nechť
a je aktuální token
b je symbol na vrcholu zásobníku
case tabulka[b,a] **of**

♢ $=$ push(a) & read next a from input

♢ $<$ nahraď b za b$<$ na zásobníku &
push(a) & read next a from input

♢ $>$ **if** $<y$ na zásobníku úplně nahoře **and r:** $\mathbf{A} \rightarrow y \in P$
then přepiš $<y$ *naA* & zapiš **r** na výstup **else** ERROR

♢ ' ' ERROR

until $a = \$$ **and** $b = \$$

Je nutné uvést, že symbolem na dně zásobníku je \$, který zároveň také symbolizuje konec vstupu. Relace pro tyto symboly je $\$ < n$ a $\$ > n$ pro všechna $n \in \Gamma$.

Díky této podmínce je zajištěno, že všechny symboly na zásobníku budou minimalizovány, nejsou-li na vstupu žádné další tokeny a že hranice bude nejpozději na dně zásobníku. Bílá místa v tabulce značí syntaktickou chybu, a proto jich lze jednoduše využít pro hlášení chybových stavů. Stejně tak i situace, kdy nemáme žádné pravidlo respektive jich máme více.

	ID	**INT**	**DBL**	**STR**	**+**	**-**	*****	**/**	$<$	$>$	$<=$	$>=$	**=**	**!=**	**,**	**(**	**)**	**$**
ID	*X*	*X*	*X*	*X*	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	*E*	$>$	$>$
INT	*X*	*X*	*X*	*X*	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	*X*	$>$	$>$
DBL	*X*	*X*	*X*	*X*	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	*X*	$>$	$>$
STR	*X*	*X*	*X*	*X*	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	*X*	$>$	$>$
+	$<$	$<$	$<$	$<$	$>$	$>$	$<$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$>$
-	$<$	$<$	$<$	$<$	$>$	$>$	$<$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$>$
*****	$<$	$<$	$<$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$>$
/	$<$	$<$	$<$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$>$
$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$>$
$>$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$>$
$<=$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$>$
$>=$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$>$
=	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$>$	$>$	$>$	$<$	$>$	$>$
!=	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$>$	$>$	$>$	$<$	$>$	$>$
,	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$>$	$>$	$>$	$<$	$>$	$>$
(	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$=$	*X*
)	*X*	*X*	*X*	*X*	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	*X*	$>$	$>$
$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	*X*	*X*

Příklad precedenční tabulky z projektu IFJ

Ještě tedy schází vysvětlit jak se k oněm symbolům v tabulce dopracovat. Nejdříve nadefinujme dvě relace, které budou důležité při definování precedenčních relací [5]. Nejdříve nadefinujme relaci head:

head(u)=$\{X | U \Rightarrow^{+} X\beta\}$ *kde*$\beta \in \Gamma^{*}$

Je to vlastně množina prvních symbolů všech řetězců, které lze z U vygenerovat.
Obdobně množina tail:

tail(u)=$\{X|U \Rightarrow^{+} \beta X\}$ *kde*$\beta \in \Gamma^{*}$

Tail je naproti tomu množina posledních symbolů, které lze z U vygenerovat. Známe-li tedy tyto relace, je pak formální zápis precedenčních relaci formalitou:

- ♢ $n_1 = n_2$ pro $n_1, n_2 \in \Gamma$ existuje-li pravidlo tvaru ve tvaru $A \rightarrow \alpha n_1 n_2 \beta$ kde $\alpha\beta \in \Gamma^{*}$
- ♢ $n_1 < n_2$ pro $n_1, n_2 \in \Gamma$ existuje-li pravidlo tvaru ve tvaru $A \rightarrow \alpha n_1 B \beta$ kde $\alpha\beta \in \Gamma^{*}$ a zároveň platí $n_2 \in head(B)$
- ♢ $n_1 > n_2$ pro $n_1, n_2 \in \Gamma$ existuje-li pravidlo tvaru ve tvaru $A \rightarrow \alpha A B \beta$ kde $\alpha\beta \in \Gamma^{*}$ a zároveň platí $n_1 \in tail(A)$ a $n_2 \in head(B)$]

Podle těchto relací byla sestrojena precedenční tabulka obsahující všechny operátory, které jazyk Simpl rozlišuje. Naprogramování precedenční analýzy znamenalo převedení zde prezentovaných poznatků do programovacího jazyka C[2]. Byl tedy vytvořen nutný zásobník a funkce nad ním. Při vykonání libovolného pravidla je generován prvek seznamu, abychom pak mohli nakonec vytisknout posloupnost pravidel aplikovanou na zdrojový kód.

Zotavení z chyb

Chyba se při analýze může vyskytnout například, když narazíme na prázdné místo v tabulce. Zhavaruje-li metoda precedenční analýzy z tohoto nebo jiného důvodu, následují tři události. Je vypsáno chybové hlášení s udáním řádku, kde se chyba nachází, do seznamu instrukcí je vložena chybová instrukce a řízení je navráceno hlavní rozkladové metodě, rekurzivnímu sestupu. Jak ten se s tímto stavem vypořádá bude popsáno níže.

Rozklad pomocí rekurzivního sestupu

Rekurzivní sestup, jako zástupce rodiny metod pracujících shora dolů, jde na parsování poněkud jinou cestou. Každý neterminál je svázán s jednou funkcí. V této funkci je pak rozklad ručně implentován. Každá funkce vrací buď true nebo false v závislosti na tom, zda odpovídá danému pravidlu. Terminály se porovnávají s tokeny od lexéru na neterminály jsou volány jejich funkce. Ve své podstatě lze funkce zaměnit za procedury. [5]. Když se nad touto konstrukcí zamyslíme, tak vlastně ani není potřeba vracet true. Pokud je totiž pravidlo správně stačí zavolat další funkci.

E⇒ id | (E)

```
E()
{
return(token(id) or token("(") and E() and token(")") );
}
```

E⇒ E + E

```
E()
{
return(E() and token("+") and E() );
}
```

E⇒ !E

```
E()
```

```
{
return(token(”!”) and E() );
}
```

Pokud známe tato pravidla, lze mechanicky přepsat gramatiku do programovacího jazyka, což je sice správně, nicméně není to nejlepší volba. Vzhledem k tomu, že jsou jednotlivé funkce psány ručně, dává to velmi velkou volnost. Je možné provádět nejrůznější optimalizace či experimentovat s chybovými hláškami. Je pravdou, že přes dobré znalosti navrhovaného jazyka můžeme velmi dobře odhadnout na základě stavu, v němž se překlad nachází, k jaké chybě mohlo dojít. V mém projektu není realizován čistý rekurzivní sestup. Rekurzivní sestup je v mém případě odsimulován pomocí konstrukce switch, která se chová podobně jako rekurzivní sestup.

Zotavení z chyb

Jak jsem naznačil výše, zotavení z chyb je prováděno v této části algoritmu. Vyskytne-li se chyba při precedenční analýze, je vrácen chybový stav a hlavní metoda se s tím musí nějak vyrovnat. V mém případě je implementováno pouze ”hrubé” zotavení, které spočívá v tom, že program se dostane do chybového stavu a žádá si od lexeru další tokeny dokud nenarazí na středník. Jakmile se tak stane, je standardně zpracován zbytek zdrojového kódu. Další z možných metod, kterou lze aplikovat na rekurzivní sestup, je vytvoření množin *starters,followers,stoppers* [5]. Tato metoda pak na základě těchto množin provede zotavení.

Vztah mezi komponentami

Přepínání mezi dvěma metodami se provádí na základě dvojic stopových prvků. Princip je následující. Narazí-li tedy hlavní metoda při překladu na jeden ze stopových prvků, ví, že bude následovat výraz. Zavolá si tedy funkci, v našem případě ji nazvěme parseExpresion(), a ta bude pokračovat tak dlouho, dokud nenačte celý výraz. Je-li celý výraz načten a je-li provedena úspěšná redukce, je řízení vráceno zpět rekurzivnímu sestupu. Dvojice stopových prvků jsou například:

- ♢ := ;
- ♢ if {
- ♢ while {
- ♢ until)

Dvojic stopových prvků je samozřejmě více, ale mám pocit, že není potřeba je uvádět všechny.

Kapitola 6

Uživatelská příručka

Předposlední část práce se zabývá seznámením se s demonstračním programem a s obsahem přiloženého média.

6.1 Přehled distribuce

Příložené médium obsahuje několik archivů, s jejichž obsahem se nyní seznámíme.

- ♢ Simpl.tar.gz obsahuje zdrojové kódy samotného překladače
- ♢ test.tar.gz obsahuje příklady pro testování překladače
- ♢ bp.tar.gz obsahuje zdrojové kódy v latexu pro vybudování bakalářské práce
- ♢ dokumentace.tar.gz obsahuje programovou dokumentaci vygenerovanou nástrojem Doxygen

6.2 Instalace

Program byl vytvořen pod operačním systémem Linux, konkrétně v 64-bitové distribuci Xubuntu. Pro překlad bude nutný překladač gcc a program make. Já jsem používal gcc ve verzi 4.0.3. Archiv se zdrojovým kódem překladače má název **Simpl.tar.gz**. Ten je nutno zkopírovat do adresáře s právem zápisu a rozbalit vhodným nástrojem. To může vypadat následovně:

```
dollson@dollson:~/myfolder>$ cp /media/cdrom0/Simpl.tar.gz Simpl.tar.gz
dollson@dollson:~/myfolder>$ tar -xvzf Simpl.tar.gz
Doxyfile
Makefile
simp_avl.c
simp_avl.h
simp.c
simp.h
simp_expression.c
simp_expression.h
simp_la.c
simp_rules.c
simp_stack.c
simp_stack.h
```

Po rozbalení je nutné zdrojové kódy přeložit. Přiložený makefile by to měl udělat za nás. Výsledek by měl vypadat nějak takto:

```
  dollson@dollson:~/myfolder>$ make
gcc -Wall -std=c99 -pedantic -g   -c -o simp_rules.o simp_rules.c
gcc -Wall -std=c99 -pedantic -g   -c -o simp_avl.o simp_avl.c
gcc -Wall -std=c99 -pedantic -g   -c -o simp.o simp.c
gcc -Wall -std=c99 -pedantic -g   -c -o simp_expression.o simp_expression.c
gcc -Wall -std=c99 -pedantic -g simp_la.o simp_stack.o simp_rules.o simp_avl.o
simp.o simp_expression.o -o simp
```

Pokud se oba kroky povedly, měl by být v cílovém adresáři binární soubor **Simp**. Tento spustitelný soubor má jediný povinný parametr a tím je soubor se zdrojovým kódem jazyka Simpl, který chceme přeložit. Pokud spustíte program bez parametru, vypíše základní informace

```
Nacházíte se v programu simp,
který je součástí bakalářské práce na FIT, VUT
autorem programu je Luděk Dolíhal
použití simp -a
-a soubor se zdrojovým kódem jazyka Simpl
```

Pro vyzkoušení je možné použít některý z připravených souborů z balíku test.tar.gz. Všechny příklady by měly být korektní a bez chyb. Po zkopírování do adresáře, kde se nachází program Simp,l pak jednoduše spustíme například příkazem:

```
  dollson@dollson:~/myfolder>$ ./Simp pokus4
```

Výsledek by měl vypadat následovně:

```
INT,identifikator,E->i,
E->i,E->E-E,E:=E,for,
E->i,E->i,E->i,E->E<E,
E->i,E->i,E->E+E,{,
INT,identifikator,E->i,E:=E,
FLOAT,identifikator,E->i,E:=E,
If,E->i,E->i,E->E=E,
E->(E),{,},else,
{,INT,identifikator,funkce,
If,E->i,E->i,E->E>E,
E->(E),{,while,E->i,
E->i,E->E<E,E->(E),{,
},},INT,identifikator,
E->i,E:=E,identifikator,E->i,
E->i,E->E+E,E:=E,},
Read,Print,E->i,E->i,
E->E*E,E->(E),},VOID,
identifikator,funkce,{,return,
E->i,E->i,E->E+E,E->(E),
},
```

Jak je patrné, program vypisuje pouze pravidla. Jména proměnných jsou pro program irelevantní.

Přiložené m0dium taktéž obsahuje archiv s dokumentací, kterou lze však kdykoli znovu vygenerovat nástrojem Doxygen. Postup by byl následující

```
dollson@dollson:~/myfolder>$ doxygen
```

6.3 Budování bakalářské práce

Celá bakalářská práce je napsána v LaTeXu. Po rozbalení archivu bp.zip, který obsahuje všechny potřebné soubory pro vybudování bakalářské práce, stačí pomocí programu make vybudovat bakalářskou práci v pdf formátu. Pro úspěšné vybudování práce je zapotřebí disponovat programy cslatex,bibtex,dvips a ps2pdf. pokud máme všechny tyto programy stačí napsat:

```
dollson@dollson:~/myfolder>$  make
```

kde myfolder je adresář s bakalářskou prací. Práce byla překládána na školním serveru merlin.

Kapitola 7

Závěr

V této práci byla navržena metoda kombinující syntaktickou analýzu shora dolů a zdola nahoru. Jako zástupce rodiny shora dolů byl vybrán rekurzivní sestup a byl kombinován s precedenční analýzou. Kombinace těchto dvou metod není ničím převratně novým, protože jej používá například překladač jazyka Perl.

Dále byl navržen poměrně vekloryse jazyk, jehož syntaktický analyzátor využívá právě kombinaci obou zmiňovaných metod. Programu by pro zpřehlednění prospělo přepsání do objektově orientovaného jazyka typu Java nebo C++. Co se týče další práce na projektu, určitě by bylo více než vhodné sestrojit sémantický analyzátor a generování mezikódu. Program by si určitě také zasloužil lepší metodu zotavení z chyb.

Literatura

[1] AVL Tree Algorithm. 2007, [Online; accessed 17-April-2007].
URL `http://sky.fit.qut.edu.au/ maire/avl/System/AVLTree.html`

[2] Herout, P.: *Učebnice jazyka C*. Nakladatelství KOPP, 2004, ISBN 80-7232-220-6, 271 s.

[3] Machata, P.: The design of combined parser. 2005, [Online;accessed 12-April-2007].
URL `http://www.feec.vutbr.cz/EEICT/2005/sbornik/`
`01-Bakalarske_projekty/07-Informacni_systemy/08-xmacha31.pdf`

[4] Majtán, J.: Syntaktická analýza založená na několika gramatikách. 2005, [Online; accessed 7-April-2007].
URL `http://www.feec.vutbr.cz/EEICT/2005/sbornik/`
`02-Magisterske_projekty/07-Informacni_systemy/07-xmajta00.pdf`

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

[6] Wikipedia: AVL tree — Wikipedia, The Free Encyclopedia. 2007, [Online; accessed 28-April-2007].
URL $http://en.wikipedia.org/w/index.php?title=AVL\_tree\&oldid=123477136$

[7] Wikipedia: Canonical LR parser — Wikipedia, The Free Encyclopedia. 2007, [Online; accessed 28-April-2007].
URL $http://en.wikipedia.org/w/index.php?title=LR\_parser\&oldid=126457777$

Seznam zkratek a symbolů

RD - Rekurzivní sestup
AST - Abstraktní syntaktický strom
Simpl - SIMple Programing Language
Lexer - Lexikální analyzátor
Parser - Syntaktický analyzátor
YACC - Yet Another Compiler Compiler
SLR - Simple LR parser
LALR - Look Ahead LR parser
GLR - Generalized LR parser