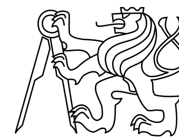
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁ SKÉ PRÁCE

Název:	Generátor syntaktického analyzátoru s transformacemi gramatik
Student:	Mat j Uzel
Vedoucí:	Ing. Jan Trávní ek
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Nastudujte postupy transformací bezkontextových gramatik na LL(1) gramatiky.
Navrhn te zápis atributovaných bezkontextových gramatik vhodný pro strojové zpracování i uživatelský zápis.
Pokuste se navrhnout transformace S-atributovaných bezkontextových gramatik na L-atributované LL(1) gramatiky.
Implementujte nástroj pro automatické generování syntaktického analyzátoru rozší eného o sémantické operace podle L-atributované gramatiky ve vašem zápisu.
Pokuste se detekovat zacyklení p i provád ní transformací.
Otestujte asovou a pam ovou náro nost syntaktického analyzátoru vytvo eného Vaším nástrojem se syntaktickým analyzátorem vytvo eným nástrojem ANTLR na Vámi navržených gramatikách a v tách jimi generovaných.

Seznam odborné literatury

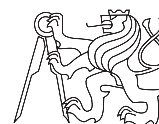
Dodá vedoucí práce.

L.S.

doc. Ing. Jan Janoušek, Ph.D. vedoucí katedry	prof. Ing. Pavel Tvrdík, CSc. d kan

V Praze dne 9. února 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky



Bakalářská práce

Generátor syntaktického analyzátoru s transformacemi gramatik

Matěj Uzel

Vedoucí práce: Ing. Jan Trávníček

16. května 2016

Poděkování

Touto cestou bych rád poděkoval vedoucímu mé práce Ing. Janu Trávníčkovi za cenné konzultace a rady. Dále bych chtěl poděkovat své rodině za podporu při mém studiu na vysoké škole.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 16. května 2016 ....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2016 Matěj Uzel. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Uzel, Matěj. *Generátor syntaktického analyzátoru s transformacemi gramatik.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Práce rozebírá principy provádění deterministické jednoprůchodové syntaktické analýzy. Především je kladen důraz na přístup shora dolů metodu rekurzivního sestupu. Dále se práce zabývá transformacemi gramatik na LL(1) gramatiky, které jsou očekávaným vstupem zmíněné metody, dále také překladem popsaným S–atributovanými a L–atributovanými překladovými gramatikami. Součástí práce je dokumentace implementovaného generátoru parseru, který implementuje zmíněnou teorii.

Klíčová slova formální jazyk, bezkontextová gramatika, syntaktická analýza, transformace gramatik

Abstract

The thesis analyzes principals of performing deterministic single-pass parsing. In particular it is focused on the top-down parsing method called recursive descent parser. The thesis also describes grammar transformations in order to get LL(1) grammar. A part of the thesis is a documentation of implemented parser generator. The generator implements semantic processing by using S–attributed and L–attributed grammars. The implemented parser is a part of attachment.

Keywords formal language, context–free grammar, syntactic analysis, grammar transformations

Obsah

Úvod **1**

1 Cíl práce **3**

2 Analýza a návrh **5**
2.1 Formální jazyk 5
2.2 Atributovaná bezkontextová gramatika 6
2.3 Lexikální analýza 8
2.4 Syntaktická analýza 8
2.5 Sémantická analýza 10
2.6 Metoda rekurzivního sestupu 11
2.7 Přehled existujících řešení generátorů syntaktické analýzy . . . 13
2.8 Transformace bezkontextových gramatik na LL(1) gramatiky . 15

3 Implementace **23**
3.1 Programové celky 23
3.2 Datové struktury 23
3.3 Načítání gramatiky 26
3.4 Generování kódu a jeho struktura 29
3.5 Implementace transformací 33

4 Uživatelská část a testování **37**
4.1 Kompilace a spuštění 37
4.2 Napojení lexikálního analyzátoru 38
4.3 Testování implementovaného nástroje 39

Závěr **41**

Literatura **43**

A Seznam použitých zkratek **45**

B Adresářová struktura přiloženého CD nosiče **47**

C Gramatika pro načítání uživatelem zadaných gramatik **49**

D Gramatika pro kontrolu a překlad sémantických operací **51**

E Příklad transformace levé rekurze se sémantickými operacemi **53**

Seznam obrázků

2.1 Derivační strom 10
2.2 Zpracování sémantiky po převedení na pravou rekurzi 21

Seznam tabulek

2.1 Konflikt first–first 13
2.2 Konflikt first–follow 13
2.3 Transformace sémantiky – levá faktorizace 16
2.4 Transformace sémantiky – rohová substituce 18
2.5 Transformace sémantiky – pohlcení terminálního symbolu 19
2.6 Transformace sémantiky – levá rekurze 22

3.1 Seznam nejdůležitějších zdrojových souborů s popisem 24
3.2 Ukázka reprezentace sémantického pravidla 26

4.1 Srovnání parseru aritmetických výrazů s ANTLR 40

Úvod

Teorie formálních jazyků je velmi významnou oblastí počítačové vědy. Přestože se jedná o teoretickou oblast, její výsledy mají široké uplatnění a to především od druhé poloviny 20. století, kdy začala být rozvíjena pro účely využití počítačem. Bez formálních jazyků by jen stěží mohly existovat překladače počítačových programů v takové podobě jako jsou známé dnes. Jejich významná část je na těchto prostředcích postavená. Využití samozřejmě nekončí pouze u překladačů. Obecně lze říci, že prostředek je možné využít při tvorbě jakékoli aplikace, která na základě vstupujícího textu v přesně specifikované podobě získává informace obsažené v textu pro následné zpracování. Formální jazyky jsou poměrně rozsáhlou oblastí, která poskytuje mnoho teoretických modelů pro popis určitých tříd jazyků. Mezi tyto modely patří například konečné automaty, gramatiky, regulární výrazy, turingovy stroje a podobně. Cílem této práce není přehled všech těchto modelů, ale analýza a implementace jedné konkrétní metody, která je pro praktické účely přínosná, a to zvláště pro počítačové zpracování strukturovaného textu. Předmětem práce jsou tedy zejména metoda provádějící syntaktickou analýzu určité podmnožiny formálních jazyků a algoritmy s tímto související.

Aplikace formálních jazyků umožňuje zpracovávat počítačem informace obsažené v textu, které jsou strukturovány člověku srozumitelnou formou. Tyto informace poté uloží do vnitřní reprezentace, s níž je možné dále pracovat. Například je možné přeložit je do podoby, která obvykle již pro člověka není snadno interpretovatelná, zato je přizpůsobená pro strojové zpracování.

Teorie formálních jazyků poskytuje různé prostředky pro popis jazyků a vyjádření překladů. Jsou také známy efektivní algoritmy pro provádění analýzy jazyků. Ve chvíli, kdy programátor implementuje analyzátor textu, může prostředku využít a navrhnout jeho pomocí strukturu jazyka. Následně analyzátor naimplementovat podle těchto algoritmů. Významným přínosem takového přístupu je, že jej lze poměrně snadno automatizovat. To má za důsledek značný vzrůst efektivity vytváření aplikací, eliminaci programátorských chyb a v neposlední řadě snadné provádění změn.

Kapitola **1**

Cíl práce

Cílem práce je provést analýzu problematiky zpracování textu počítačem při použití metody rekurzivního sestupu a naimplementovat nástroj na generování syntaktického analyzátoru rozšířeného o zpracování sémantiky pro LL(1) jazyky na základě vstupní gramatiky. Nástroj by měl také automaticky provádět transformace vstupních bezkontextových gramatik na tvar LL(1). V práci bude navrhnuta a implementována struktura zápisu vstupujících atributovaných gramatik a to takovým způsobem, aby jeho použití bylo pro uživatele snadné a přehledné. Načítání gramatik podle navrženého schématu bude implementováno do programu. Dále bude následovat testování implementovaného nástroje a srovnání s již existujícím nástrojem ANTLR z hlediska rychlosti a paměťové složitosti generovaných parserů na testovacích vstupech.

Kapitola **2**

Analýza a návrh

2.1 Formální jazyk

Před zavedením formálního jazyka je potřeba definovat abecedu. Pojmem abeceda se rozumí „*končená množina symbolů*“ [1]. Abeceda bude značena symbolem T. V některých textech je možné se též setkat s označením Σ. Jako příklad takové abecedy může být množina lexikálních elementů programovacího jazyka, nebo textového komunikačního protokolu.

$$\{begin, end, if, then, else, class, template, \ldots\}$$

Řetězec – nebo také věta – nad abecedou je libovolná konečná množina tvořená symboly dané abecedy. Dále se zavádí speciální řetězec, který reprezentuje *prázdný řetězec*. Prázdný řetězec se obvykle značí symbolem ε. Tohoto značení se bude držet i následující text.

Formální jazyk L nad abecedou T je množina řetězců nad abecedou T. Množina nemusí být pouze konečná a v praxi tomu tak ani nebývá. Formální jazyk je tedy podmnožinou všech řetězců sestávajících ze znaků dané abecedy. Formálně zapsáno $L \subseteq T^*$ [1]. Příkladem formálního jazyka mohou být všechny potencionální zápisy programů konkrétního programovacího jazyka, zatímco větou formálního jazyka by byl jeden zápis zdrojového kódu počítačového programu.

```
bool Grammar::createTable () {
    table.clear();
    for (int k=0; k<P.getCount(); k++) {
        const Rule & rule = P[k];
        for (int i=0; i<firsts[k].getSize(); i++) {
            ...
```

Chceme-li s jazykem smysluplně pracovat, je potřeba zavést prostředky, které jsou schopny jazyky popisovat. Mezi takové prostředky patří např. *konečné automaty* a *gramatiky.*[1]. Automat popisuje jazyk schopností rozhodnout, zda zadaná věta patří do jazyka, zatímco gramatika popisuje jazyk generativním způsobem – tedy pomocí pravidel postupně generuje věty. Následující text se bude zabývat především určitou skupinou gramatik.

2.2 Atributovaná bezkontextová gramatika

Teorie formálních jazyků zavádí pojem *formální gramatika.* Gramatika je formalismus, který slouží pro popis obsahu a struktury konkrétního jazyka. Podle definice je bezkontextová gramatika uspořádaná čtveřice (N, T, P, S), kde

N je množina *neterminálních symbolů* – zkráceně *neterminálů*

T je množina *terminálních symbolů* – zkráceně *terminálů*, $N \cap T = \emptyset$

P je množina *přepisovacích pravidel* – pravidlo je zobrazení $N \rightarrow (N \cup T)^*$

S je *počáteční symbol* $S \in N$

Atributovaná bezkontextová gramatika je bezkontextová gramatika rozšířená o syntetizované a dědičné atributy a dále o sémantické operace. Syntetizované atributy mohou být přiřazeny libovolnému terminálnímu nebo neterminálnímu symbolu. Dědičné atributy mohou být přiřazovány pouze neterminálním symbolům. Sémantické operace se přiřazují přepisovacím pravidlům gramatiky. Každé pravidlo může mít libovolný počet sémantických operací. Sémantické operace slouží k přiřazení hodnot atributům. Obecně lze sémantickou operaci chápat jako zobrazení syntetizovanému nebo dědičnému atributu, kde vstupem jsou atributy, které již byly v průběhu průchodu derivačním stromem spočteny [2].

Syntetizované atributy expandovaného neterminálu se vyhodnocují až na závěr provádění expanze – po vyhodnocení dědičných atributů expandovaného neterminálu a všech atributů patřících symbolům pravé strany expandovaného pravidla. Získávají tedy informace z nižších vrstev derivačního stromu a zpřístupňují je vyšší vrstvě. Dědičné atributy se vyhodnocují během expanze syntaktického pravidla a to v pořadí v jakém jdou po sobě symboly

[1] Výčet není zdaleka úplný. Mezi další mechanismy patří např. *turingovy stroje.* Jednotlivé prstředky se od sebe liší tím, jaký volí přístup k popisu jazyka, jak komplikované jazykové struktury jsou schopny obsáhnout a také s tím související rychlost algoritmů pro práci s nimi.

[2] *Derivační strom* je prostředek pomocí kterého lze názorně vyobrazit strukturu věty generované gramatikou. Derivační strom je strom z teorie grafů tedy souvislý acyklický neorientovaný graf, kde kořen derivačního stromu je startovací symbol gramatiky, vnitřní uzly stromu jsou neterminální symboly a listy stromu jsou terminální symboly. Hrany mezi symboly přesně kopírují pořadí aplikací pravidel gramatiky.

pravé strany syntaktického pravidla. Dědičné atributy tedy předávají informace z nižší vrstvy derivačního stromu vyšším vrstvám.

Gramatika, ve které všechny sémantické operace splňují podmínku, že dědičný atribut na levé straně sémantického pravidla je funkcí dědičných atributů expandovaného neterminálu, nebo dědičných a syntetizovaných atributů symbolu pravé strany pravidla s výskytem před měněným atributem, a podmínku, že syntetizovaný atribut na levé straně sémantického pravidla je funkcí libovolných atributů daného pravidla s výjimkou syntetizovaných atributů expandovaného neterminálu, taková gramatika se nazývá *L–atributovaná gramatika.* Formálně zapsáno: [3]

$$A_d^i \coloneqq f(A_d^0, A_{s/d}^1, \ldots, A_{s/d}^{i-1})$$

$$A_s^0 \coloneqq f(A_d^0, A_{s/d}^1, \ldots, A_{s/d}^n)$$

Má-li gramatika všechna pravidla ve tvaru, že modifikovaný syntetizovaný atribut je funkcí libovolných syntetizovaných atributů pravé strany syntaktického pravidla, pak se taková gramatika označuje jako *S–atributovaná gramatika.*

$$A_s^0 \coloneqq f(A_s^1, \ldots, A_s^n)$$

Je tedy zřejmé, že každá S–atributovaná gramatika je také zároveň L–atributovanou gramatikou a je možné u obou vyhodnotit všechny atributy jedním preorderovým průchodem derivačního stromu.

Častou úlohou v souvislosti s gramatikami je rozhodnutí o tom, zda vstupní věta patří do jazyka popsaného gramatikou a případně zjištění struktury věty. Tento problém řeší tzv. syntaktická analýza. Dále lze proces rozšířit o sémantickou analýzu, která se provádí zároveň se syntaktickou analýzou a jejímž výsledkem je překlad vstupní věty do vnitřní reprezentace, která je v obecném případě realizována stromovou strukturou.

V souvislosti s gramatikami je důležité zavést pojem *větná forma.* Pojem větná forma bude použit pro označení řetězce skládajícího se z terminálních a neterminálních symbolů, který lze získat ze startovního symbolu libovolnou aplikací pravidel dané gramatiky.

V textu budou uvedeny ukázkové gramatiky pro ilustraci některých tvrzení nebo vlastností. Aby nebylo nutné u každé takové gramatiky explicitně uvádět veškeré údaje, nebude-li uvedeno jinak, má se za to, že všechny symboly sestávající z jednoho nebo více *malých* písmen jsou *terminálními symboly.* Podobně pak symboly z *velkých* písmen jsou považovány za *neterminální symboly.* Startovací symbol takové gramatiky je symbol levé strany *prvního* z uvedených pravidel.

[3]Symbol A s příslušným horním indexem označuje symbol syntaktického pravidla, kde index určuje jeho pozici. Dolní index s reprezentuje libovolný syntetizovaný atribut, index d dědičný atribut a s/d reprezentuje syntetizovaný nebo dědičný atribut příslušného symbolu.

2.3 Lexikální analýza

První částí zpracování textu počítačem je *lexikální analýza*. Lexikální analyzátor načítá jednotlivé znaky vstupního textu, které uskupuje v lexikální elementy a zjišťuje význam, který tyto elementy nesou. Lexikální element obvykle nese informaci o tom, zda reprezentuje klíčové slovo, případně identifikátor, číslo s jeho hodnotou, ap. Produktem lexikálního analyzátoru je tedy posloupnost lexikálních elementů s případnými atributy, které upřesňují jeho význam. Lexikální analýzou vzniklé lexémy jsou připraveny pro následné zpracování syntaktickým analyzátorem. Lexikální analýza bere ohled pouze na jednotlivé lexémy. Neposkytuje tedy žádnou informaci napříč různými lexémy a nedokáže rozeznat chybu v širším kontextu [4]. Lexikální elementy je obvykle možné popsat regulárními jazyky. Z tohoto důvodu se lexikální analýza zpravidla provádí pomocí překladového konečného automatu.

2.4 Syntaktická analýza

Proces syntkatické analýzy je druhým krokem zracování textu počítačem. Syntaktická analýza již neřeší zpracování jednotlivých znaků, ale zpracovává lexikální elementy rozpoznané lexikálním analyzátorem. Úkolem syntaktické analýzy je určit pro vstupní větu a zadanou gramatiku, zda věta je generovaná toto gramatikou a dále nalézt strukturu této věty. Nalezovnou strukturou může být přímo derivační strom, nebo posloupnost pravidel, která byla pro analýzu postupně expandována, nebo jiný prostředek, ze kterého lze přímo rekonstruovat průběh derivací. Sytaktickou analýzu lze tedy použít pro kontrolu syntaxe vzhledem k jazyku generovaného gramatikou a k lokalizaci případné syntaktické chyby ve vstupní větě.

Existují v zásadě dva přístupy, jak syntaktickou analýzu provádět. Prvním – pro člověka přirozenějším způsobem – je tzv. syntaktická analýza shora dolů. Druhým způsobem je syntaktická analýza zdola nahoru. Jak již názvy napovídají, tyto přístupy se liší v základním principu budování derivačního stromu.

2.4.1 Syntaktická analýza shora dolů

Syntaktická analýza shora dolů staví derivační strom od kořene a postupně na základě přečtených lexikálních elementů přidává příslušné pravé strany syntaktických pravidel jako potomky vnitřním uzlům stromu. Takový přístup vytváří tzv. levý rozklad, nebo-li každá expanze se provádí u nejlevějšího neterminálního symbolu větné formy. Do této kategorie patří metoda rekurzivního sestupu, která je hlavním předmětem této práce.

[4] *Lexikální element* nebo-li *lexém* by se dal přirovnat ke slovu přirozeného jazyka s informací slovního druhu a případnými dalšími informacemi upřesňujícími jeho význam.

2.4.2 Syntaktická analýza zdola nahoru

Narozdíl od analýzy shora dolů analýza zdola nahoru vytváří derivační strom od listů ke kořeni stromu. Při srovnání s předchozím přístupem jsou zde v každém kroku hledány podřetězce větné formy odpovídající pravým stranám pravidel a jsou nahrazovány za levé strany pravidel. Tímto se postupně vytváří pravý rozklad. Mezi metody syntaktické analýzy zdola nahoru patří například algoritmus *CYK*. Tento algoritmus, postavený na myšlence dynamického programování, vyžaduje pro svou práci gramatiku v tzv. *Chomského normální formě* [5] a pracuje v čase $O(n^3)$. Dále do kategorie analýzy zdola nahoru patří veškeré LR parsery, které jsou schopny zpracovávat vstup v čase $O(n)$.

2.4.3 Srovnání přístupů syntaktické analýzy

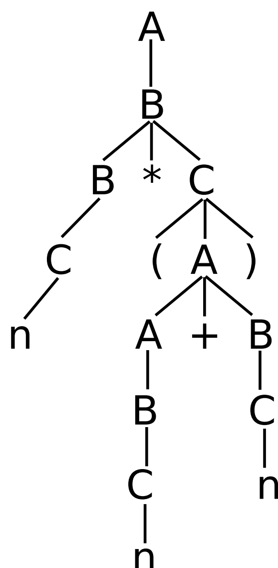
Jako ukázku stavby derivačního stromu pro oba přístupy je uveden názorný příklad na gramatice $G(\{A, B, C\}, \{+, *, n\}, P, A)$. Je to gramatika reprezentující aritmetické výrazy s operátory $+$, $*$ a čísly reprezentovanými symbolem n. Množina P obsahuje pravidla:

$$\begin{aligned} &1.\quad A \to A + B \\ &2.\quad A \to B \\ &3.\quad B \to B * C \\ &4.\quad B \to C \\ &5.\quad C \to n \\ &6.\quad C \to (A) \end{aligned} \tag{2.1}$$

Gramatika je jednoznačná, a tudíž má pro každou větu jediný derivační strom. To je i základním požadavkem pro analýzu textu pomocí deterministických metod, jako jsou LL a LR analýza. Na obrázku 2.1 je ilustrován derivační strom pro větu $n * (n + n)$. Jak již bylo řečeno, přístup anlýzy shora dolů vytváří levý rozklad, zatímco analýza zdola nahoru vytváří pravý rozklad a jistě by existovaly i další způsoby, jak provést rozklad. Jedna věta gramatiky tedy obvykle mívá více různých rozkladů.

Posloupnost čísel $\{2, 3, 4, 5, 6, 1, 2, 4, 5, 4, 5\}$, jejíž prvky odkazují na čísla pravidel, vytváří levý rozklad věty $n * (n + n)$. Pokud jde o pravý rozklad téže věty, ten je určen posloupností těchto pravidel $\{2, 3, 6, 1, 4, 5, 2, 4, 5, 4, 5\}$. Větné formy, které postupně vznikají jejich aplikací, jsou uvedeny níže. Nej-

[5] Gramatika je v Chomského normální formě, jsou-li všechna její pravidla ve tvaru $A \to BC$ nebo $A \to a$. Obsahuje-li gramatika pravidlo $S \to \varepsilon$, pak v žádném z jejích pravidel nesmí mít symbol S na pravé straně. $A, B, C \in N$, $a \in T$. Do tohoto tvaru lze převést každou bezkontextovou gramatiku [2].



Obrázek 2.1: Derivační strom

prve větné formy pro levý rozklad 2.2 a následně pro pravý rozklad 2.3.

$$\begin{split} A \overset{\textcircled{2}}{\Rightarrow} B \overset{\textcircled{3}}{\Rightarrow} B*C \overset{\textcircled{4}}{\Rightarrow} C*C \overset{\textcircled{5}}{\Rightarrow} n*C \overset{\textcircled{6}}{\Rightarrow} n*(A) \overset{\textcircled{1}}{\Rightarrow} n*(A+B) \overset{\textcircled{2}}{\Rightarrow} \\ n*(B+B) \overset{\textcircled{4}}{\Rightarrow} n*(C+B) \overset{\textcircled{5}}{\Rightarrow} n*(n+B) \overset{\textcircled{4}}{\Rightarrow} n*(n+C) \overset{\textcircled{5}}{\Rightarrow} n*(n+n) \end{split} \tag{2.2}$$

$$\begin{split} n*(n+n) \overset{\textcircled{5}}{\Leftarrow} C*(n+n) \overset{\textcircled{4}}{\Leftarrow} B*(n+n) \overset{\textcircled{5}}{\Leftarrow} B*(C+n) \overset{\textcircled{4}}{\Leftarrow} B*(B+n) \overset{\textcircled{2}}{\Leftarrow} \\ B*(A+n) \overset{\textcircled{5}}{\Leftarrow} B*(A+C) \overset{\textcircled{4}}{\Leftarrow} B*(A+B) \overset{\textcircled{1}}{\Leftarrow} B*(A) \overset{\textcircled{6}}{\Leftarrow} B*C \overset{\textcircled{3}}{\Leftarrow} B \overset{\textcircled{2}}{\Leftarrow} A \end{split} \tag{2.3}$$

2.5 Sémantická analýza

Syntaktická analýza často ke zpracování vstupního textu nestačí. Proto se k ní připojuje ještě sémantická analýza, která se obvykle provádí zároveň se syntaktickou analýzou. Úkolem sémantické analýzy je budování datové struktury, která je vnitřní reprezentací informací vstupního textu – uchovává všechny podstatné informace obsažené v textu s ohledem na jejich strukturu. Vstupní text je ze své povahy lineární, přestože uchovává strukturovanou informaci.

S touto lineární podobou informace není příliš praktické pracovat. Vnitřní reprezentace má v obecném případě podobu stromu a obvykle kopíruje strukturu derivačního stromu.

2.6 Metoda rekurzivního sestupu

Metoda rekurzivního sestupu je konkrétní realizací *syntaktické analýzy shora dolů* pracující s LL(1) jazyky. Algoritmus tedy začíná od startovacího symbolu gramatiky a následně expanduje příslušná pravidla jejíž levé strany odpovídají nejlevějšímu neterminálnímu symbolu ve větné formě. Expanze jsou voleny takovým způsobem, aby jednotlivé lexikální elementy v listech stromu odpovídaly vstupní větě.

Vstupem je bezkontextová gramatika, která je typu LL(1). [6] Výsledkem této metody je zdrojový kód, který implementuje syntaktický analyzátor pro zadanou gramatiku.

Při implementaci se používají nepřímo rekurzivní volání funkcí, kde každému neterminálnímu symbolu odpovídá jedna funkce. Zavolání funkce odpovídá provedení expanze pravidla s příslušnou levou stranou. Dále pro každý terminální symbol se definuje funkce, která provádí srovnání symbolu – tedy zjištění, zda se načtený symbol ve výhledu shoduje s očekávaním. V případě, že nedojde ke shodě, znamená to, že vstupní věta obsahuje syntaktickou chybu.

Pro rozhodnutí která expanze se provede, je použita rozkladová tabulka, pomocí které je možné pro podmnožinu bezkontextových jazyků – jazyků typu LL(1) – zaručit, že algoritmus bude pracovat v lineárním čase.

Pokud bychom neměli žádnou předpočítanou informaci o rozhodování expanzí, museli bychom použít backtracking. Postupně nahrazovat za všechna pravidla s odpovídajícími levými stranami. Takový postup bývá označován jako syntaktická analýza s návratem a má exponenciální časovou složitost. Je tedy pro praktické použití naprosto nepoužitelný, zatímco metoda rekruzivního sestupu zaručuje lineární složitost, za cenu schopnosti zpracovat pouze určitou podmnožinu bezkontextových jazyků. V praxi však toto omezení není příliš limitující.

2.6.1 Množina FIRST

Definice množiny $FIRST$ [3]:

$$FIRST(\alpha) = \{a \mid \alpha \Rightarrow^* a\beta, a \in T, \alpha, \beta \in (N \cup T)^*\} \cup \{\varepsilon \mid alpha \Rightarrow^* \varepsilon\}$$

Množina $FIRST$ se definuje pro všechny řetězce $\alpha \in (N \cup T)^*$. Obsahem množiny jsou takové termminální symboly, kterými může vstupující řetězec

[6] Metodu lze zobecnit, aby přijímala jazyky typu LL(k). Dosáhne se toho tak, že se rozšíří definice množin FIRST a FOLLOW na řetězce o délce k a přizpůsobí se tomu vytváření rozkladové tabulky. Princip fungování metody zůstává nezměněn.

začínat při libovolné kombinaci aplikací pravidel gramatiky. Množina $FIRST$ se zjišťuje pro pravé strany každého syntaktického pravidla dané gramatiky.

2.6.2 Množina FOLLOW

Definice množiny FOLLOW [3]:

$$FOLLOW(A) = \{a \mid S \Rightarrow^* \alpha A \beta, a \in FIRST(\beta)\}$$

Množina $FOLLOW$ udává, které terminální symboly mohou bezprostředně následovat za neterminálem v libovolné větné formě derivovatelné ze startovního symbolu pomocí pravidel příslušné gramatiky. Tato množina se zjišťuje pro každý neterminální symbol gramatiky.

2.6.3 Rozkladová tabulka

Rozkladová tabulka je zobrazení $Z(A, a) \to k, A \in N, a \in T, k \in Z$. Hodnota k indexuje číslo pravidla.

Tabulka slouží k určení expanze, která se má provést v závislosti na nejlevějším právě expandovaném neterminálním symbolu a načteném terminálním symbolu ve výhledu. Aby bylo rozhodnutí deterministické, musí pro každou dvojici *terminál* a *neterminál* existovat maximálně jedno pravidlo pro expanzi. Rozkladová tabulka se sestavuje na základě předem spočtených množin $FIRST$ a $FOLLOW$. Níže je uveden algoritmus pro tvorbu rozkladové tabulky.

Data: LL(1) gramatika
Result: Rozkladová tabulka
foreach *k-té pravidlo* $A \to \alpha$ **do**
    foreach $a \in (FIRST(\alpha) \setminus \varepsilon)$ **do**
        definuj hodnotu zobrazení $Z(A, a) \to k$
    end
    if $\varepsilon \in FIRST(\alpha)$ **then**
        foreach $b \in FOLLOW(A)$ **do**
            definuj hodnotu zobrazení $Z(A, b) \to k$
        end
    end
end

2.6.4 Konflikt first–first

Ke konfliktu first–first dochází v případě, že existují alespoň dvě pravidla ve tvaru $A \to \alpha$, kde $A \in N, \alpha \in (N + T)^+$, která obě mají v množině FIRST alespoň jeden stejný symbol.

V tabulce 2.1 je pro ilustraci uvedena gramatika, ve které dochází ke konfliktu first–first. Pravidla č. 1 a 2 mají obě v množině FIRST terminální symbol a, tudíž na základě načteného terminálního symbolu a ve výhledu a expandovaného neterminálního symbolu A nelze rozhodnout, které z těchto pravidel použít k expanzi. Konfliktům lze v mnoha případech předejít aplikováním transformačních metod na gramatiky.

	Pravidlo			FIRST	FOLLOW
1.	A	$\rightarrow$	aB	$\{a\}$	$\{\varepsilon\}$
2.	A	$\rightarrow$	Bc	$\{a, b\}$	
3.	B	$\rightarrow$	bB	$\{b\}$	$\{\varepsilon, c\}$
4.	B	$\rightarrow$	a	$\{a\}$	

Tabulka 2.1: Konflikt first–first

Konfliktu first–first se zpravidla zbavuje pomocí rohové substituce a následné aplikace levé faktorizace.

2.6.5 Konflikt first–follow

Konflikt first–follow nastává tehdy, existuje-li pravidlo $A \rightarrow \varepsilon$, jež má v množině FOLLOW symbol, který také obsahuje jiné pravidlo $A \rightarrow \alpha$ ve své množině FIRST. Konflikt first–follow je ilustrován v tabulce 2.2. V tomto případě při expanzi neterminálu B při načteném symbolu c ve výhledu nelze rozhodnout, zda použít pravidlo č. 2 nebo pravidlo č. 3. Pro odstranění konfliktu first–follow se používá metoda pohlcení terminálního symbolu.

	Pravidlo			FIRST	FOLLOW
1.	A	$\rightarrow$	aBc	$\{a\}$	$\{\varepsilon\}$
2.	B	$\rightarrow$	cB	$\{c\}$	$\{c\}$
3.	B	$\rightarrow$	ε	$\{\varepsilon\}$	

Tabulka 2.2: Konflikt first–follow

2.7 Přehled existujících řešení generátorů syntaktické analýzy

Počítačové zpracování formálních jazyků je velmi významnou oblastí informatiky. Metody pro analýzu jazyků jsou navrženy takovým způsobem, aby byly snadno automatizovatelné. Z těchto důvodů existuje celá řada programů pro generování lexikálních a syntaktických analyzátorů. Generátory se od sebe liší zejména tím, na jakých metodách jsou postaveny – s tím souvisí, jaké třídy

jazyků jsou schopny zprácovat. Dále také v jakém programovacím jazyce generují cílový kód. Nyní následuje krátký popis některých aplikací, které jsou pro tyto účely k dispozici.

2.7.1 ANTLR

ANother Tool for Language Recognition – zkráceně ANTLR je nástroj pro načítání strukturovaného textu a pro generování parseru. Program má implementované prostředky pro generování lexikální analýzy, syntaktické analýzy i pro stavbu abstraktního syntaktického stromu. Aplikace je schopna zpracovávat bezkontextové jazyky typu LL(k). Původní verze programu byla napsána v jazyce C a generuje cílový kód v jazyce C a C++. Novější verze je napsaná v jazyce Java a generuje kódy v celé řadě cílových jazyků. Mezi tyto cílové jazyky patří: Ada, ActionScript, C, C++, C#, Objective C, Java, JavaScript, Python, Perl, PHP [4]. Vstupní gramatika se zadává v tzv. rozvinuté Backusově–Naurově formě. Za vznikem této aplikace stojí Terence Parr [5].

2.7.2 GNU Bison

Bison je aplikace určená pro generování parseru pro bezkontextové gramatiky. Cílový kód je generován v jazyce C a C++. Je podporován i jazyk Java ten však jen v experimentální verzi. Pro generování používá determinsitickou metodu LALR(1). Aplikace byla vyvinuta pomocí programovacího jazyka C. Veškeré zdrojové kódy jsou veřejně přístupné přes repozitář umístěný na oficiálních stránkách. Aplikace je dostupná zdarma a její použití se řídí licencí GNU General Public License [6].

2.7.3 JavaCC

Java CC™ – *Java Compiler Compiler* je generátor lexikálního a syntaktického analyzátoru. Prostřednictvím nástroje JJTree umožňuje vytvářet AST načítaných gramatik. Aplikace je naprogramována v jazyce Java a generovaný cílový kód je rovněž v Javě. V poslední době je i možnost generování kódu v jazyce C++[7]. Aplikace je k dispozici pod BSD licencí – je tedy volně dostupná. Dále na oficiálních stránkách je k dispozici mnoho ukázkových gramatik [7].

2.7.4 YACC

YACC – *Yet Another Compiler-Compiler* je program pro generování syntaktických analyzátorů, negeneruje však lexikální analyzátor. Pro generování lexikálního analyzátoru je možné použít program *Lex* nebo *Flex*. Cílový kód je v jazyce C. Činnost programu YACC je založena na LALR analýze [8].

[7]Uvedeno: 15.05.2016

2.8 Transformace bezkontextových gramatik na LL(1) gramatiky

Metoda rekurzivního sestupu vyžaduje na vstupu bezkontextovou gramatiku, která je typu LL(1). Není-li gramatika typu LL(1), je potřeba ji na tento tvar transformovat.

Pro transformaci gramatik na tvar LL(1) existuje několik postupů. Často je potřeba postupně a opakovaně aplikovat několik z těchto metod. Existují však bezkontextové jazyky, které není možné generovat bezkontextovou gramatikou typu LL(1). V takovém případě nelze použít metodu rekurzivního sestupu. Ukázkou jednoho z takových jazyků je jazyk $\{a^k b^l a^k | k > 0, l \geq 0\}$. V následující části jsou rozebrány tyto transformace: [8]

- Levá faktorizace
- Rohová substituce
- Pohlcení terminálního symbolu
- Odstranění levé rekurze

2.8.1 Levá faktorizace

Aplikací levé faktorizace lze v některých případech zabránit konfliktu first–first. Pokud máme k dispozici k přednačtených symbolů a dvě nebo více pravidel začínajících stejným podřetězcem terminálů o délce d, tak ve chvíli, kdy je $k \leq d$, nelze rozhodnout, kterým z těchto pravidel provést expanzi. Levá faktorizace řeší zmíněný problém tím, že se zavede nové pravidlo s původní levou stranou, které se expanduje na těch d totožných symbolů a další neterminál, který se následně pro každé z konfliktních pravidel expanduje na odpovídající pravou stranu bez oněch prvních d stejných symbolů. Posune se tedy rozhodnutí o správné expanzi až na moment, kde se pravé strany pravidel začínají lišit. Slovy nastíněný postup transformace lze přesně popsat schématem, ve kterém je nejprve uvedena gramatika před transformací 2.4 a následně po aplikaci levé faktorizace 2.5. symboly $A, N \in N$, $\alpha \in T$, $\beta \in (N \cup T)^*$.

$$A \to \alpha\beta_1|\alpha\beta_2|\dots|\alpha\beta_n \tag{2.4}$$

$$\begin{aligned} A &\to \alpha N \\ N &\to \beta_1|\beta_2|\dots|\beta_n \end{aligned} \tag{2.5}$$

[8] Poznámka k notaci, která bude dále používána. N_{A_d} znamená pomocný atribut A_d symbolu N, který je určen pro předání hodnoty atributu d symbolu A.

Nyní následuje popis transformace sémantických pravidel. Mějme syntaktické pravidlo 2.6, u kterého provedeme levou faktorizaci mezi posledním symbolem α a prvním symbolem β, kde každý symbol $\alpha \in T^+$ a $\beta \in (N \cup T)^*$

$$A \to \alpha\beta \tag{2.6}$$

Dále mějme k tomuto pravidlu přiřazený libovolný počet operací modifikující syntetizované atributy 2.7

$$A_s \coloneqq f(A_d, \alpha_s, \beta_d, \beta_s) \tag{2.7}$$

a libovolný počet sémantických operací 2.8 modifikující dědičné atributy.

$$\begin{aligned} \alpha_d &\coloneqq f(A_d) \\ \beta_d &\coloneqq f(A_d, \alpha_s) \end{aligned} \tag{2.8}$$

Potom po provedení levé faktorizace podle výše uvedeného schématu, bude transformované pravidlo spolu se sémantickými operacemi a nově zavedeným neterminálem N vypadat podle tabulky 2.3.

Princip řešení spočívá v tom, že všechny atributy vytknutých symbolů se zkopírují nově zavedeným dědičným atributům nového neterminálu N. Následně se v pravidlech, kde levá strana je N, provede modifikace atributů příslušnou funkcí a poté – jde-li o syntetizované atributy – se vrátí hodnoty zpět pravidlu s vytknutými symboly.

Před aplikací levé faktorizace					
A	$\to$	$\alpha\beta$	α_d	$\coloneqq$	$f_1(A_d)$
			β_d	$\coloneqq$	$f_2(A_d, \alpha_s)$
			A_s	$\coloneqq$	$f_3(A_d, \alpha_s, \beta_d, \beta_s)$
Po aplikaci levé faktorizace					
A	$\to$	$\alpha A'$	α_d	$\coloneqq$	$f_1(A_d)$
			A'_{Ad}	$\coloneqq$	A_d
			$A'_{\alpha s}$	$\coloneqq$	α_s
			A_s	$\coloneqq$	$f_3(A_d, \alpha_s, f_2(A_d, \alpha_s), A'_{\beta s})$
A'	$\to$	β	β_d	$\coloneqq$	$f_2(A'_{Ad}, A'_{\alpha s})$
			$A'_{\beta s}$	$\coloneqq$	β_s

Tabulka 2.3: Transformace sémantiky – levá faktorizace

2.8.2 Rohová substituce

Rohovou substituci má smysl provádět za předpokladu, že byl nalezen v gramatice konflikt typu first–first a některé z pravidel podílející se na konfliktu

začíná neterminálním symbolem. Dalším požadavkem je, aby pravidlo neobsahovalo levou rekurzi. Pokud by takové pravidlo bylo levě rekurentní, mělo by to za následek opakování stejného konfliktu a následnou eliminací konfliktu stejnou transformací by způsobilo zacyklení a tím i neustálé rozrůstání gramatiky. Rohová substituce nahradí každé konfliktní pravidlo, které začíná neterminálním symbolem několika novými pravidly. Nová pravidla budou mít namísto prvního neterminálního symbolu vloženy pravé strany těch pravidel s odpovídajícími levými stranami a zbytek zůstane nezměněn. Pro každou možnou kombinaci vznikne jedno takové pravidlo. Opět lze schématicky situaci popsat následovně – před transformací 2.9 a po transformaci 2.10.

$$\begin{aligned} A &\rightarrow B_1\alpha_1|B_2\alpha_2|\dots|B_n\alpha_n \\ B_1 &\rightarrow \beta_{11}|\beta_{12}|\dots|\beta_{1m_1} \\ B_2 &\rightarrow \beta_{21}|\beta_{22}|\dots|\beta_{2m_2} \\ &\dots \\ B_n &\rightarrow \beta_{n1}|\beta_{n2}|\dots|\beta_{nm_n} \end{aligned} \tag{2.9}$$

$$\begin{aligned} A &\rightarrow \beta_{11}\alpha_1|\beta_{12}\alpha_1|\dots|\beta_{1m_1}\alpha_1 \\ A &\rightarrow \beta_{21}\alpha_2|\beta_{22}\alpha_2|\dots|\beta_{2_m2}\alpha_2 \\ &\dots \\ A &\rightarrow \beta_{n1}\alpha_n|\beta_{n2}\alpha_n|\dots|\beta_{nm_n}\alpha_n \end{aligned} \tag{2.10}$$

Konfliktní pravidla, u kterých se provede vložení pravých stran se z gramatiky odstraní, neboť jsou plně nahrazena nově vzniklými pravidly. Pravidla jejichž pravé strany byly použity ke vložení do konfliktních pravidel, musí v gramatice zůstat. Důvodem je, že tato pravidla mohou být používána jinými pravidly gramatiky, která ke konfliktu nepřispívají. Odebráním pravidel by tedy transformovaná gramatika nemusela být ekvivalentní.

Při provádění transformace sémantických operací je potřeba se vypořádat s nahrazením neterminálního symbolu, který může mít své atributy. Lze zvolit dva přístupy jak tuto transformaci provést. Prvním z přístupů je zavedení pomocných atributů, do kterých se uloží produkty sémantických operací, vložených pravých stran. Dalším přístupem je vložit tyto operace přímo do konkrétních míst, kde jsou používány. Převod sémantických operací během transformace rohovou substitucí je ilustrován v tabulce 2.4.

2.8.3 Pohlcení terminálního symbolu

Transformační metoda pohlcení terminálního symbolu je navržena pro odstranění konfliktu follow–follow. Podobně jako u levé faktorizace, ani aplikací této transformace není zaručeno, že konflikt bude z gramatiky odstraněn. V některých případech dokonce může dojít k zacyklení. Odhalení pravidla, které konflikt způsobilo, je v tomto případě poněkud obtížnější, než jak tomu bylo v předešlých případech. Výběr pravidla, u něhož se provede transformace závisí

Před aplikací rohové substituce	
$A \rightarrow B\alpha$	$B_d := f_1(A_d)$ $\alpha_d := f_2(A_d, B_d, B_s)$ $A_s := f_3(A_d, B_d, B_s, \alpha_d, \alpha_s)$
$B \rightarrow \beta$	$\beta_d := f_4(B_d)$ $B_s := f_5(B_d, \beta_d, \beta_s)$

Po aplikaci rohové substituce	
$A \rightarrow \beta\alpha$	$\beta_d := f_4(f_1(A_d))$ $\alpha_d := f_2(A_d, f_1(A_d), f_5(B_d, \beta_d, \beta_s))$ $A_s := f_3(A_d, f_1(A_d), f_5(B_d, \beta_d, \beta_s), \alpha_d, \alpha_s)$

Tabulka 2.4: Transformace sémantiky – rohová substituce

na tom, odkud se konfliktní terminál dostal do množiny FOLLOW konfliktního pravidla. To není triviální zejména v situaci, pokud zmíněný terminál „proputuje“ více pravidly.

Z gramatiky 2.11 je zřejmé, že $a \in FIRST(A)$, pravidlo č. 2 se přepisuje na prázdný řetězec a konfliktním terminálem je symbol a. Pravidlo č. 3 způsobí, že $a \in FOLLOW(C)$. Dále pravidlo č. 4 způsobí, že se symbol zkopíruje do množiny $FOLLOW(D)$ a nakonec kvůli pravidlu č. 5 se symbol dostane do množiny $FOLLOW(A)$. Odhalení pravidla způsobujícího konflikt znamená zjistit, z pravé strany jakého pravidla se terminál a do množiny $FOLLOW(A)$ dostal. Terminál a se do množiny $FOLLOW(A)$ dostal původem z pravidla č. 3 a aplikaci pohlcení terminálního symbolu je tedy nutné provést v tomto pravidle. Pravidlo $B \rightarrow bCab$ se nahradí pravidlem $B \rightarrow bNb$, kde N je nově zavedený neterminál. Dále vznikne nové pravidlo $N \rightarrow bDa$.

$$\begin{aligned}
&1.\quad A \rightarrow aB \\
&2.\quad A \rightarrow \varepsilon \\
&3.\quad B \rightarrow bCab \\
&4.\quad C \rightarrow bD \\
&5.\quad D \rightarrow bA
\end{aligned} \tag{2.11}$$

Stejně jako u levé faktorizace bylo v některých případech potřeba provést nejprve dosazení do prvního neterminálu pravé strany pravidla, může u této transformační metody nastat, že konfliktní terminál se do množiny FOLLOW dostane „nepřímo“ skrze jiný neterminál. Opět je uveden příklad na konkrétní gramatice 2.12. Pravidlo č. 1 zapříčiní rozšíření množiny $FOLLOW(B)$ o symboly množiny $FIRST(C)$ a tím mimo jiné o konfliktní terminální symbol a. Metoda řešící popsanou situaci se nazývá *extrakce pravého kontextu* a je – dá

Před aplikací transformace	
$A \rightarrow \alpha B a \beta$	$\alpha_d := f_1(A_d)$ $B_d := f_2(A_d, \alpha_s, \alpha_d)$ $\beta_d := f_3(A_d, \alpha_s, \alpha_d, B_s, B_d, a_s)$ $A_s := f_4(A_d, \alpha_s, \alpha_d, B_s, B_d, a_s, \beta_s, \beta_d)$

Po aplikaci transformace	
$A \rightarrow \alpha N \beta$	$\alpha_d := f_1(A_d)$ $N_{Ad} := A_d$ $N_{\alpha_s} := \alpha_s$ $N_{\alpha_d} := f_1(A_d)$ $\beta_d := f_3(A_d, \alpha_s, \alpha_d, N_{Bs}, N_{Bd}, N_{as})$ $A_s := f_4(A_d, \alpha_s, \alpha_d, N_{Bs}, N_{Bd}, N_{as}, \beta_s, \beta_d)$
$N \rightarrow Ba$	$B_d := f_2(N_{Ad}, N_{\alpha s}, N_{\alpha d})$ $N_{Bs} := B_s$ $N_{Bd} := f_2(N_{Ad}, N_{\alpha s}, N_{\alpha d})$ $N_{as} := a_s$

Tabulka 2.5: Převod sémantických operací během aplikace pohlcení terminálního symbolu. Na poslední pravidlo je ještě nutné aplikovat rohovou substituci

se říci – zobecněním metody rohové substituce.

$$
\begin{aligned}
&1. \quad A \rightarrow bBC \\
&2. \quad B \rightarrow a \\
&3. \quad B \rightarrow \varepsilon \\
&4. \quad C \rightarrow a
\end{aligned}
\tag{2.12}
$$

2.8.4 Transformace levé rekurze na pravou rekurzi

Obsahuje-li gramatika pravidlo s levou rekurzí, nemůže být typu LL(1). Tento problém je možné vyřešit převedením levé rekurze na pravou rekurzi. Následující schéma ukazuje obecný přepis levé rekurze na pravou rekurzi. Gramatika obsahující pravidla 2.13 je levě rekurentní. Nahrazením těchto pravidel pravidly 2.14 převedeme levou rekurzi na pravou rekurzi při zachování ekvivalence.

$$A \rightarrow A\,\alpha_1|...|A\,\alpha_n|\beta_1|...|\beta_n \tag{2.13}$$

$$
\begin{aligned}
A &\rightarrow \beta_1\,A^{'}|...|\beta_n\,A^{'} \\
A^{'} &\rightarrow \alpha_1\,A^{'}|...|\alpha_n\,A^{'} \\
A^{'} &\rightarrow \varepsilon
\end{aligned}
\tag{2.14}
$$

Data: gramatika obsahující přímou levou rekurzi
Result: transformovaná gramatika zbavená levé rekurze
foreach *symbol* $A \in N$ **do**
  zaveď nový neterminál $A^{'}$
  foreach *pravidlo* $A \rightarrow \alpha$*, kde* $\alpha \in (N \cup T)^{*}$ **do**
    vytvoř nové pravidlo $A^{'} \rightarrow \varepsilon$
    if *pravidlo je ve tvaru* $A \rightarrow A\delta$ **then**
      nahraď pravidlem $A^{'} \rightarrow \delta A^{'}$
    else
      nahraď pravidlem $A \rightarrow \beta A^{'}$
    end
  end
end

Výše uvedený postup transformace bral ohled pouze na syntaktická pravidla. Chceme-li zachovat i zpracování sémantiky, je nutné správně transformovat i sémantické operace. Mějme následující gramatiku 2.15 s levou rekurzí a se sémantickými pravidly:

$$\begin{aligned} A^0 &\rightarrow A^1\alpha & A^0_s &:= f(A^1_s, \alpha_s) \\ A &\rightarrow \beta & A_s &:= g(\beta_s) \end{aligned} \tag{2.15}$$

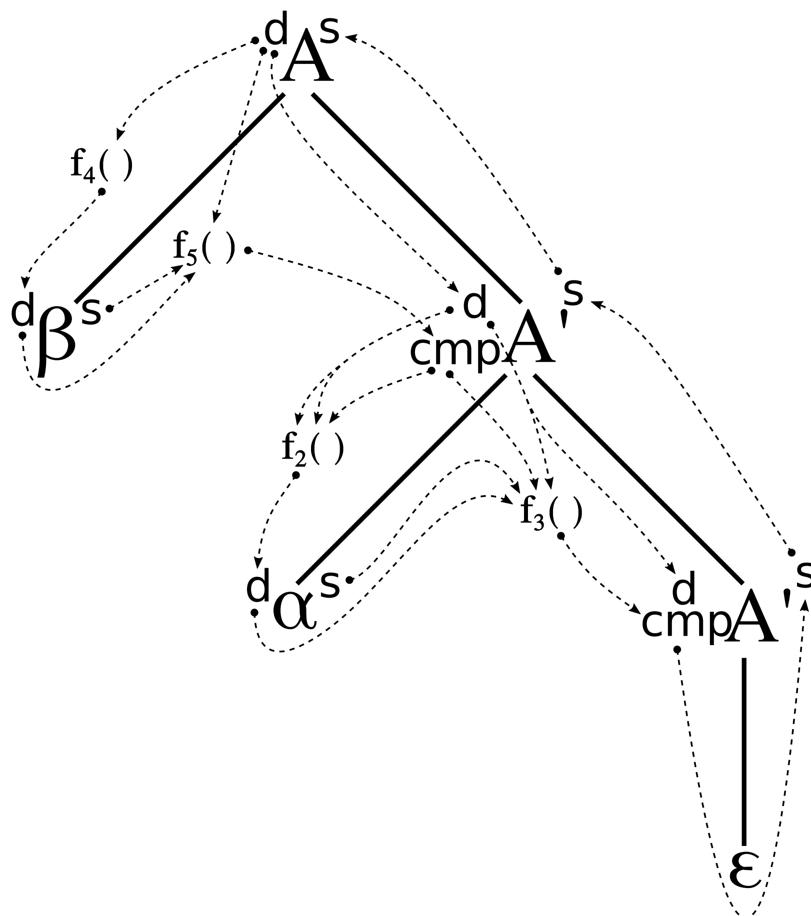
Gramatiku je možné převést na tvar bez levé rekurze 2.16

$$\begin{aligned} A &\rightarrow \beta A^{'} & A^{'}_{cmp} &:= g(\beta_s) & A_s &:= A^{'}_s \\ A^{'0} &\rightarrow \alpha A^{'1} & A^{'1}_{cmp} &:= f(A^{'0}_{cmp}, \alpha_s) & A^{'0}_s &:= A^{'1}_s \\ A^{'} &\rightarrow \varepsilon & A^{'}_s &:= A^{'}_{cmp} \end{aligned} \tag{2.16}$$

Původní gramatika zpracovává mezivýsledky pomocí syntetizovaných atributů. Vypočet tedy probíhá až v konečné fázi expanze nebo-li při návratu k expandujícímu uzlu. Po převedení na pravou rekurzi tento postup uplatnit nelze. Je potřeba mezivýsledky počítat již při provádění expanze. Proto je zaveden pomocný dědičný atribut *cmp*, který mezivýsledky vyhodnocuje při expanzi. Následně se výsledek vrátí v syntetizovaných atributech neterminálů A a $A^{'}$.

Dvojici sémantických pravidel $A^1_d = f(A^0_d)$ a $\beta_d = g(A_d)$ lze transformovat pouze v případě, že funkce $f(x)$ je identita. Důvodem je, ža na dědičný atribut A_d se aplikuje funkce $f(x)$ tolikrát, kolikrát se provede expanze $A \rightarrow A\,\alpha$. Teprve poté se v pravidle $A \rightarrow \beta$ vloží dědičný atribut symbolu A do dědičného atributu symbolu β. Při zpracovávání vstupu transformovanou gramatikou se nejprve provede expanze $A \rightarrow \beta\,A^{'}$. V tuto chvíli je potřeba znát, kolikrát se transformace atributu d provede. To je však možné zjistit až tehdy, když se provedou všechny expanze $A^{'} \rightarrow \beta\,A^{'}$ a na závěr expanze $A^{'} \rightarrow \varepsilon$. Jinými slovy v pravidle $A \rightarrow \beta\,A^{'}$ by se do dědičného atributu symbolu β musela přiřadit hodnota syntetizovaného atributu symbolu $A^{'}$, který bude znám až

po provedení všech expanzí neterminálu A'. To je však možné pouze při víceprůchodovém zpracování. V tabulce 2.6 je uveden obecný předpis pro transformaci sémantiky pro L–atributovanou gramatiku. Výpočet atributů a jejich závislosti po transformaci je znázorněno na obrázku 2.2.



Obrázek 2.2: Zpracování sémantiky po převedení na pravou rekurzi

před transformací							
A^0	$\rightarrow$	$A^1\alpha$	A_d^1	$:=$	$f_1(A_d^0)$	1	
			α_d	$:=$	$f_2(A_d^0, A_s^1, A_d^1)$	2	*
			A_s^0	$:=$	$f_3(A_d^0, A_s^1, A_d^1, \alpha_s, \alpha_d)$	3	
A	$\rightarrow$	β	β_d	$:=$	$f_4(A_d)$	4	*
			A_s	$:=$	$f_5(A_d, \beta_d, \beta_s)$	5	
po transformaci							
A	$\rightarrow$	$\beta A'$	A_d'	$:=$	A_d	1	
			β_d	$:=$	$f_4(A_d)$	4	
			A_{cmp}'	$:=$	$f_5(A_d, \beta_d, \beta_s)$	5	
			A_s	$:=$	A_s'	3	
A'^0	$\rightarrow$	$\alpha A'^1$	$A_d'^1$	$:=$	$f_1(A_d'^0)$	1	
			α_d	$:=$	$f_2(A_d'^0, A_{cmp}'^0, A_d'^0)$	2	
			$A_{cmp}'^1$	$:=$	$f_3(A_d'^0, A_{cmp}'^0, A_d'^0, \alpha_s, \alpha_d)$	3	
			$A_s'^0$	$:=$	$A_s'^1$	3	
A'	$\rightarrow$	ε	A_s'	$:=$	A_{cmp}'	3	

Tabulka 2.6: Schéma transformací sémantických pravidel při aplikaci levé rekurze

KAPITOLA 3

Implementace

Součástí práce je také počítačový program, který realizuje výše popsanou teorii. Konkrétně je implementován automatizovaný generátor překladače pro LL(1) jazyky pomocí syntaktické analýzy shora dolů metodou rekurzivního sestupu. Dále jsou implementovány metody pro převod bezkontextových gramatik na tvar LL(1). Generátor v sobě zahrnuje syntaktickou a sémantickou analýzu. Součástí implementace však není geterátor lexikální analýzy, která je také nezbytná pro chod překladače. Pro tyto účely je implementován jeden obecný lexikální analyzátor, který je připravený pro napojení na generované kódy. Veškeré zdrojové kódy a ukázkové gramatiky jsou součástí přílohy k této práci.

Pro implementaci nástroje byl zvolen programovací jazyk C++. Program byl vyvíjen a testován pod operačním systémem Linux a laděn pomocí překladače GCC. Důvodem pro tuto volbu byla především přenositelnost programu na jiné platformy. Další důvod je kontrola nad prací s pamětí a celkově psaní kódu na nižší úrovni, což při efektivní implementaci má pozitivní efekt na rychlost běhu programu. Posledním méně podstatným důvodem pro tuto volbu byla má zkušenost s jazykem C++.

3.1 Programové celky

Z důvodu snažší orientace je zdrojový kód dělen do logických celků. Tabulka 3.1 ukazuje nejdůležitější z nich a jejich popis.

3.2 Datové struktury

3.2.1 Reprezentace gramatiky

Základní datovou strukturou programu je třída `Grammar`. Instance této třídy drží veškeré informace o načtené gramatice. Atributy třídy kopírují struk-

Hlavičkový soubor	Implementace	Popis
grammar.h	grammar.cpp	Reprezentace načtené gramatiky, výpočet rozkladové tabulky ap.
rule.h	rule.cpp	Reprezentace syntaktického pravidla
semantics.h	semantics.cpp	Reprazentace sémantického pravidla
table.h	table.cpp	Předpočítaná rozkladová tabulka
attr.h	attr.cpp	Reprezentace syntetizovaných a dědičných atributů
codegen.h	codegen.cpp	Knihovna pro generování překladače
seman-replacer.h	seman-replacer.h	Obsahuje nástroje pro generování sémantiky z vnitřní reprezentace
lexan.h	lexan.cpp	Lexikální analyzátor
gram-parser.h	gram-parser.cpp	Parser pro načítání gramatik do programu
seman-parser.h	seman-parser.cpp	Parser pro načítání sémantických pravidel
transformer.h	transformer.cpp	Knihovna pro převody gramatik

Tabulka 3.1: Seznam nejdůležitějších zdrojových souborů s popisem

turu definice formální bezkontextové gramatiky. Třída obsahuje následující položky:

- Množina terminálních symbolů
- Množina neterminálních symbolů
- Množina přepisovacých pravidel
- Startovací symbol
- Množina syntetizovaných a dědičných atributů

Třída obsahuje ještě další informace, které nedefinují gramatiku jako takovou. Jsou to informace spočtené, které jsou použity pro následné zpracování. Jsou jimi:

- Množina FIRST
- Množina FOLLOW
- Rozkladová tabulka
- Názvy symbolických konstant jednotlivých terminálních symbolů pro spojení s lexikálním analyzátorem generovaných kódů

3.2.2 Reprezentace syntaktického pravidla

Jednotlivé symboly v syntaktickém pravidle jsou reprezentovány pomocí textových řetězců. Levá strana pravidla jako jeden řetězec a pravá strana jako pole řetězců pomocí třídy `Container` [9]. Syntaktické pravidlo dále drží seznam sémantických pravidel, která se postupně provádí při expanzi daného syntaktického pravidla. Sémantické pravidlo je reprezentováno třídou `Semantics`. Poslední informace, která se u syntaktických pravidel eviduje, je pole celých čísel s názvem `occurrences`, které pro každý symbol pravidla říká, kolik stejných symbolů se nachází v pravidle před vlastním výskytem. Seznam těchto čísel se využívá pro jednoznačné pojmenování proměnných reprezentující atributy pro provádění sémantické analýzy při generování kódu. Systém pojmenování proměnných je popsán v sekci 3.4.5.

Pro ilustraci syntaktické pravidlo $A \rightarrow aAAbBa$ bude mít hodnoty pole `occurrences` následující: $(0, 0, 1, 2, 0, 0, 1)$.

3.2.3 Reprezentace sémantického pravidla

Vnitřní reprezentace sémantického pravidla se skládá ze dvou základních částí. První částí je textová podoba sémantického pravidla v takové formě, v jaké je následně použito ve vygenerovaném kódu s tím rozdílem, že v této vnitřní podobě ještě neobsahuje názvy jednotlivých atributů. Na příslušná místa těchto názvů je vložen speciální symbol `#` a bezprostředně za ním se nachází index do pole, který drží informace o konkrétním atributu. Zmíněnou druhou částí reprezentace je pole, kde každý prvek obsahuje informace o daném atributu. Jsou to informace, které jednoznačně identifikují atribut: název symbolu, kterému atribut patří, název atributu, typ atributu, o který se jedná, a některé další informace potřebné pro generování kódu. Překlad do této reprezentace realizuje syntaktický analyzátor sémantického pravidla [10]

Opět pro ilustraci syntaktické pravidlo $A \rightarrow aBb$ s přiřazeným sémantickým pravidlem

```
semantics { "A=>s = foo ( 1 + a=>s, ( B=>d - b=>s ) * 3 )" };
```

bude mít vnitřní reprezentaci tvořenu polem znázorněným tabulkou 3.2 a textovým řetězcem:

```
#0=foo(1+#1,(#2-#3)*3)
```

[9] Třída `Container` v programu realizuje dynamické pole pro uložení homogenních dat libovolného datového typu. Při přesažení kapacity se datová oblast realokuje na dvojnásobnou velikost.

[10] Implementace syntaktického analyzátoru sémantických pravidel se nachází v souboru seman-parser.cpp.

Index	0	1	2	3
Symbol	"A"	"a"	"B"	"b"
Atribut	"s"	"s"	"d"	"s"
Typ	syn	syn	inh	syn
...	...	...	...	...

Tabulka 3.2: Ukázka reprezentace sémantického pravidla

3.3 Načítání gramatiky

Nástroj umožňuje uživateli generovat překladače pomocí gramatik zapsaných v textové podobě. Struktura zápisu je pevně daná. Následující schéma ukazuje, příklad akceptovaného vstupu. Jedná se o zápis gramatiky popisující *jazyk aritmetických výrazů* s operátory $+, -, \times, \div$, se závorkami, respektující priority operátorů. Sémantika přidružená ke gramatice popisuje vyhodnocení výrazu.

```
grammar {
  terminals { numb, float, plus, minus, mult, div, lpar, rpar }
  nonterminals { A, A2, B, B2, C }
  syn {
    A      { ("s", "float") }
    A2     { ("s", "float") }
    B      { ("s", "float") }
    B2     { ("s", "float") }
    C      { ("s", "float") }
    numb   { ("s", "numb")  }
    float  { ("s", "float") }
  }
  inh {
    A2     { ("d", "float") }
    B2     { ("d", "float") }
  }
  rules {
    A   ->  B A2        semantics { "A2=>d = B=>s",
                                    "A=>s = A2=>s" };
    A2  ->  plus B A2   semantics { "A2[1]=>d = A2=>d + B=>s",
                                    "A2=>s = A2[1]=>s" };
    A2  ->  minus B A2  semantics { "A2[1]=>d = A2=>d - B=>s",
                                    "A2=>s = A2[1]=>s" };
    A2  ->  eps         semantics { "A2=>s = A2=>d" };
    B   ->  C B2        semantics { "B2=>d = C=>s",
                                    "B=>s = B2=>s" };
```

```
    B2  ->  mult C B2   semantics { "B2[1]=>d = B2=>d * C=>s",
                                    "B2=>s = B2[1]=>s" };
    B2  ->  div C B2    semantics { "B2[1]=>d = B2=>d / C=>s",
                                    "B2=>s = B2[1]=>s" };
    B2  ->  eps         semantics { "B2=>s = B2=>d" };
    C   ->  lpar A rpar semantics { "C=>s = A=>s" };
    C   ->  numb        semantics { "C=>s = numb=>s" };
    C   ->  float       semantics { "C=>s = float=>s" };
  }
  start = A
}
```

3.3.1 Pravidla pro definici gramatik

Bloky a seznamy prvků jsou obklopeny složenými závorkami. Přiřazování hodnot nebo seznamů prvků identifikátorům se provádí pomocí operátoru `=`.

Klíčové slovo `grammar` uvozuje začátek definice gramatiky. Toto klíčové slovo musí být bezprostředně na začátku, a za ním volitelně může být identifikátor reprezentující název gramatiky. V bloku `grammar` se definují pomocí svých klíčových slov:

terminals – terminální symboly

nonterminals – neterminální symboly

syn – syntetizované atributy

inh – dědičné atributy

rules – přepisovací pravidla

semantics – sémantická pravidla

start – startovací symbol

Jednotlivé bloky mohou být deklarovány v libovolném pořadí. Je pouze nutné dodržet, aby každý použitý identifikátor byl předem deklarován, a to nejvýše jednou a zároveň nesmí kolidovat se jménem některého z klíčových slov. Nelze tedy použít například identifikátor při deklaraci pravidla, který nebyl předtím deklarován v bloku `terminals` nebo `nonterminals`. Pokud je uvozen blok, který již byl definován, pak nové deklarace nepřepíší původní deklarace, ale pouze se sjednotí. Výjimkou je deklarace startovacího symbolu, který po znovudeklaraci přepíše původní startovací symbol. Tato operace také upozorní uživatele, protože se tímto předchozí označení startovacího symbolu stává zbytečným. Bílé znaky [11] nemají na význam sémantiky žádný vliv. Veškeré znaky následující za koncem bloku `grammar` jsou ignorovány.

[11] Bílé znaky jsou mezera, tabulátor a odřádkování.

Blok *terminals* slouží k deklaraci terminálních symbolů gramatiky. Dále pro každý terminál se definuje název symbolické konstanty terminálu, který může být vynechán. Symbolické konstanty slouží k propojení generovaného překladače s lexikálním analyzátorem, který bude ke kódu napojen [12]. Jednotlivé záznamy se deklarují jako názvy identifikátorů s volitelným přiřazením textového řetězce určující název symbolické konstanty. Záznamy jsou od sebe odděleny čárkou.

Blok *nonterminals* deklaruje neterminální symboly gramatiky. Způsob deklarace je totožný s deklarací terminálních symbolů s výjimkou, že se nedefinují symbolické konstanty.

Pomocí bloku *syn* se deklarují syntetizované atributy. Blok sestává ze seznamu deklarací atributů oddělených čárkou. Záznam začíná názevem terminálního nebo neterminálního symbolu který uvozuje blok. V bloku je seznam názvů atributů a jejich datových typů vložených do kulatých závorek. Tyto seznamy jsou také odděleny čárkou.

V bloku *inh* se deklarují dědičné atributy. Způsob zápisu se nijak neliší od deklarace syntetizovaných atributů. Je však potřeba respektovat, že dědičné atributy je možné přiřazovat pouze neterminálním symbolům. V opačném případě nebude gramatika přijata.

Konstrukce uvozená klíčovým slovem *start* a následným přiřazením identifikátoru definuje startovací symbol gramatiky. Identifikátor, který je označen jako startovací symbol musí být z množiny neterminálních symbolů.

Načítání gramatik je realizováno pomocí LL(1) syntaktické analýzy. Jedná se z velké části o automaticky generovaný kód [13]. Gramatika, kterou je načítání realizované je uvedena v příloze. [14]

Sémantická pravidla uživatel zadává ve formě řetězce. Tyto řetězce jsou následně s určitými úpravami vloženy do generovaného kódu. Uvedená gramatika pro popis korektních vstupů od uživatele nijak nekotroluje správnost sémantických pravidel a to z důvodu, že tyto řetězce mají spíše povahu lexikálních elementů z pohledu parseru gramatiky, přestože jsou vnitřně strukturované a před použitím je nutné provést kontrolu korektnosti. Tato kontrola se však provádí odděleně. Pro každé načtené sémantické pravidlo se spustí proces, který pravidlo zkontroluje z hlediska správnosti a přeloží do vnitřní reprezentace. Tento proces je realizován vlastním parserem, jehož gramatika je také součástí přílohy.

Sémantická pravidla se zapisují ve formě textového řetězce mezi uvozovky `"..."`. Při přiřazení více sémantických pravidel jednomu symbolu jsou jednotlivá sémantická pravidla oddělena čárkou. V syntaktickém pravidle může

[12] Více podrobností o symbolických konstantách je v sekci 4.2.

[13] Kód realizující syntaktickou analýzu vstupní gramatiky byl generován pomocí nástroje samotného, ve chvíli, kdy již byl zprovozněn generátor. Gramatika popisující jazyk načítaných gramatik byla napevno vložena do zdrojového kódu.

[14] Pro snažší čitelnost není gramatika ve tvaru LL(1). Samotný kód byl vygenerován ekvivalentní gramatikou, která je ve tvaru LL(1) s odpovídající sémantikou.

být více stejných symbolů. Pro odlišení symbolů je použita konstrukce indexace v hranatých závorkách následující bezprostředně za symbolem. Indexace začíná od nuly zleva doprava. Při neuvedení indexu je implicitně index roven nule. Přístup k atributům symbolu je realizován operátorem `=>`. Chceme-li tedy například zpřístupnit atribut *attr* druhého symbolu A v pravidle $A \to abAb$, pak je třeba použít konstrukci `A[1]=>attr`.

Přiřazení hodnot atributům se provádí operátorem `=`. Na pravé straně přiřazení může být konstantní hodnota, hodnota atributu symbolu pravidla, aritmetický výraz kombinovaný s atributy, nebo volání funkce pro jejíž parametry platí stejná pravidla. Při vytváření sémantických operací je nutné dbát na to, aby všechny použité atributy na pravé straně již byly vyčísleny.

3.4 Generování kódu a jeho struktura

Členění generovaného parseru odpovídá běžnému kódu knihovny v jazyce C++. program generuje hlavičkový soubor a implementační soubor s příponami `h` resp. `cpp`.

Na začátku hlavičkového souboru je příkaz `include` s názvem hlavičkového souboru lexikálního analyzátoru. Dále jsou definovány symbolické konstanty odpovídající lexikálním elementům. Poté následuje definice struktury `Lexem` reprezentující lexikální element. Struktura vždy obsahuje proměnnou s názvem `type` a datovým typem `int`. Dále jsou ve struktuře všechny případné atributy lexikálního elementu s jejich odpovídajícími datovými typy. Za touto strukturou je deklarace třídy `Input`, která zprostředkovává informace o lexikálních elementech poskytovaných lexikálním analyzátorem. Zprostředkovávací funkce jsou `read`, která deleguje načtení jednoho lexému na lexikální analyzátor a funkce `show`, jejíž úkolem je zjištění následujícího dosud nepřečteného lexikálního elementu. Propojení těchto informací s vygenerovaným parserem je důkladně rozebráno v sekci 4.2. Na konci hlavičkového souboru je deklarace funkce s názvem `parse_sentence`, jejíž návratová hodnota je `bool`, první prametr je reference na vstupní stream typu `std::istream`. Za prvním parametrem následují vstupní parametry odpovídající dědičným atributům startovacího symbolu resp. výstupní parametry odpovídající syntetizovaným atributům startovacího symbolu s odpovídajícími datovými typy v pořadí, v jakém byly atributy startovacímu symbolu přiděleny.

Implementační soubor je složen těmito položkami:

- Implementace metod `read` a `show` třídy `Input`
- Procedury provádějící sémantické operace
- Funkce pro operaci srovnání
- Hlavičky funkcí pro operaci expanze

- Implementace funkcí pro operaci expanze
- Implementace deklarované funkce `parse_sentence`

3.4.1 Symbolické konstanty lexikálních elementů

Deklarace symbolických konstant slouží pro napojení vygenerovaného kódu na lexikální analyzátor. Jedná se o deklarace celočíselných konstant, které v programu reprezentují terminální symboly [15]. Každému terminálnímu symbolu přísluší jedna symbolická konstanta. Hodnoty těchto konstant určuje lexikální analyzátor a vzájemně se musí lišit.

Obvyklá situace je, že máme k dispozici lexikální analyzátor, který definuje symbolické konstanty pro své lexikální elementy. Syntaktický analyzátor zavede konstanty pro terminální symboly a přiřadí jim hodnoty příslušných konstant lexikálních elementů. Přiřazení těchto hodnot definuje uživatel při psaní gramatiky v sekci terminals a to buďto přiřazením konstant lexikálního analyzátoru, nebo přímo přiřazením číselné hodnoty.

3.4.2 Sémantické operace

Při expanzi pravidel, která mají definovány sémantické operace se v příslušnou chvíli zavolá procedura, která tuto operaci provede. Jsou to často velmi krátké procedury, obvykle jen přiřazení nebo volání funkce. Pro přehlednost a oddělení sémantické analýzy od analýzy syntaktické v rámci zdrojového kódu jsou tyto operace prováděny jako samostatná volání. Deklarace jsou realizovány s direktivou `inline`, tím pádem se při překladu kódy těchto procedur vloží do míst volání. Volání procedury má tvar:

```
inline void semantic_call_xxxx_xx ( typ & atribut1,
                                    typ & atribut2, ... )
```

V názvu procedury jsou dvě čísla. V ukázce je zastupují znaky *x*. První číslo označuje číslo syntaktického pravidla, kterému sémantické volání přísluší. Druhé číslo označuje číslo sémantického pravidla relativně vůči syntaktickému pravidlu. Číslování začáná od nuly. Z toho plyne, že maximální počet potenciálních syntaktických pravidel je 10 000 a maximální počet sémantických pravidel pro každé syntaktické pravidlo je 100.

3.4.3 Operace srovnání

Operace srovnání jsou dvojího druhu. Pokud je voláno srovnání na terminální symbol bez žádných přidělených syntetizovaných atributů, je volána obecná procedura pro provedení srovnání. Má-li terminální symbol přidělen jeden či více syntetizovaných atributů, pak je nutné tyto atributy získat a výstupními

[15] Z pohledu lexikálního analyzátoru se jedná o lexikální elementy.

parametry předat. Pro tento účel je pro každý takový terminální symbol generována procedura, která předání atributů zajistí.

Požadavek na srovnání symbolu obdrží v prvním parametru referenci na objekt `Input` s názvem `input` a dále očekávání terminálního symbolu. V případě obecné procedury je očekávání předáno pomocí druhého vstupního parametru `expection`, u procedur pro srovnání konkrétních symbolů tento parametr není potřeba předávat. Procedura načte lexikální element, zjistí zda došlo ke shodě a vrátí příznak indikující úspěch, případně neúspěch. Dále, pokud se jedná o procedury pro konkrétní elementy, zjistí hodnoty atributů a předá je výstupními parametry.

Názvy procedur srovnání jsou vytvářeny prefixem `match_call_` za nímž následuje název lexikálního elementu, pro který operaci srovnání provádí. Obecné srovnávací procedury mají stejný název jen bez názvu elementu. Srovnání lexikálního elementu `number` s jedním syntetizovaným atributem `value` bude mít srovnávací proceduru:

```
bool match_call_number ( Input & input, int & atnumbervalue )
```

Zatímco při srovnání lexikálního elementu `kw_while` bez atributů bude zavolána procedura:

```
bool match_call_ ( Input & input, int expection )
```

3.4.4 Operace expanze

Expanze mohou být volány nepřímo rekurzivně. Proto jsou všechny procedury expanzí nejprve deklarovány a následně implementovány. Expanze má za úkol provést náhradu levého neterminálu syntaktického pravidla za jeho pravou stranu. Náhrady neprobíhají přímo, ale jsou reprezentovány historií volání rekurzivních procedur na programovém zásobníku. Zásobník, který je nezbytný pro analýzu bezkontextových jazyků je tímto delegován na zásobník běžícího programu.

Pro každý neterminální symbol gramatiky je generována jedna procedura reprezentující expanzi. Procedura implementuje provedení expanze pro pravé strany všech pravidel jejichž levá strana odpovídá expandovanému neterminálu. Která větev expanze se provede, závisí na vstupním terminálním symbolu a předpočítané rozkladové tabulce.

Budeme-li uvažovat gramatiku jejíž pravidla, která mají na levé straně neterminál A jsou uvedena níže 3.1. Dále pak prvnímu pravidlu jsou přiřazeny dvě sémantické operace. První sémantická operace mění hodnotu dědičného atributu symbolu A na pravé straně pravidla. Druhá sémantická operace mění hodnotu syntetizovaného atributu symbolu A na levé straně pravidla.

$$\begin{array}{ll} 1. & A \rightarrow aAb \\ 2. & A \rightarrow b \end{array} \tag{3.1}$$

Neterminálu A bude generátorem vytvořena následující procedura.

```
bool expand_call_A ( Input & input, int atAd00, int & atAs00 ) {
    int show = input.show().type;
    if ( show == LEXEM_a ) {
        int atas01;
        int atAd02;
        int atAs02;
        if (! match_call_a( input, atas01 ) ) {
            return false; }
        semantic_call_0000_00 ( atAd02, atAd00 );
        if (! expand_call_A( input, atAd02, atAs02 ) ) {
            return false; }
        if (! match_call_( input, LEXEM_b ) ) {
            return false; }
        semantic_call_0000_01 ( atAs00, atas01, atas03, atAd02,
                                        atAs02, atAd02, atAs03 );
    }
    else if ( show == LEXEM_b ) {
        if (! match_call_( input, LEXEM_b ) ) { return false; }
    }
    else return false;
    return true;
}
```

3.4.5 Názvy proměnných reprezentující atributy

Proměnné reprezentující atributy symbolů gramatiky se v kódu vyskytují na třech místech. Ve funkcích provádějící srovnání symbolu se syntetizovanými atributy, ve funkcích pro provádění expanzí a v procedurách sémantických operací. Při generování názvů proměnných je nutné volit jejich názvy takovým způsobem, aby nemohlo dojít ke konfliktním názvům. V prvním případě je notace nejjednodušší, protože nemůže dojít ke konfliktu v názvech. Název je tedy tvořen pouze prefixem `at`, poté následuje název terminálního symbolu, kterému atribut patří a nakonec samotný název atributu. V ostatních dvou případech je potřeba odlišit použití stejného atributu na různých pozicích v syntaktickém pravidle. Proto je název těchto proměnných rozšířen o informaci odpovídající pozice. Název je tvořen jako v předchozím případě s přidaným dvoumístným číslem pozice.

Terminální nebo neterminální symbol `SYMB` na 4. pozici v syntaktickém pravidle a se syntetizovaným nebo dědičným atributem `attr` bude reprezentován proměnnou s názvem:

- `atSYMBattr` v případě krátké notace

- `atSYMBattr04` v případě dlouhé notace

3.5 Implementace transformací

Implementace všech transformací přesně odpovídá teorii probrané výše. Z tohoto důvodu není v této sekci rozebírán teoretický základ, ale především způsob implementace.

Při implementaci všech transformací je třeba dbát na pojmenovávání nových neterminálních symbolů a jejich atributů. Pro tento účel je implementována funkce, která vytvoří název pro nově vznklý neterminální symbol tak, aby byl v každém případě v gramatice jedinečný. Pro nové atributy je takové pojmenování zvlášť důležité ve chvíli, kdy nový neterminál má ve svých pomocných atributech uchovávát hodnoty atributů jiných symbolů. Tato situace je velmi citlivá na potencionální konflikt. Proto se názvy nově vzniklých atributů tvoří podle přesných pravidel.

Implementace rozhodování o prováděných transformací a implementace všech jednotlivých transformací se nacházejí v souboru `transformer.cpp`

3.5.1 Proces rozhodnutí o aplikaci transformací

Rozhodnutí o tom, jaká posloupnost transformací vede k cíli – tedy nalezení ekvivalentní gramatiky typu LL(1) je velmi netriviálním úkolem. Obecně neexistuje zaručený postup, který by pro každou bezkontextovou gramatiku byl schopen nalést ekvivalentní gramatiku LL(1). Pro některé bezkontextové jazyky ani LL gramatika sestrojit nelze. Existují ovšem určité postupy, pomocí kterých lze v mnoha případech tento problém řešit. Nejedná se však o algoritmus, ale spíše o praxí získané postupy, které lze kombinovat.

Samotný proces rozhodování lze provádět pouze souběžně při aplikování transformací. Nelze tedy nejprve spustit proces, který by předem zjistil, které transformace a v jakém pořadí je třeba provést, a následně po zjištění, zda posloupnost transformací vede k cíli, transformace provést.

Implementace rozhodování o provedení transformací je provedeno následujícím způsobem. Nejprve proběhne kontrola, zda některé pravidlo neobsahuje levou rekurzi. Pro každé nalezené levě rekurentní pravidlo se zavolá metoda, která pravidlo převede na pravě rekurentní. Po zbavení se veškeré přímé levé rekurze proběhne zjištění, zda dochází v gramatice ke konfliktu first–first. Pokud ano, zavolá se na konfliktní pravidla, jejichž pravá strana začíná neterminálem metoda realizující rohovou substituci. Po provedení všech rohových substitucí se provede u všech konfliktních pravidel levá faktorizace. Na závěr se spustí kontrola konfliktu first–follow a na všechna konfliktní pravidla se zavolá metoda provádějící pohlcení terminálního symbolu. Celý postup se opakuje tak dlouho, dokud nenastane žádná z následujících situací:

- Gramatika obsahuje levě rekurentní pravidlo nebo pravidla

- Nastal konflikt first–first
- Nastal konflikt first–follow
- Nebyl přesažen maximální povolený počet transformací
- Nebylo detekováno zacyklení

3.5.2 Levá faktorizace

Levá faktorizace je implementována v metodě `leftFactorization`, která na vstupu očekává gramatiku, u které se transformace provádí a seznam čísel indexující konfliktní pravidla. Transformační procedura předpokladá, že levé strany všech pravidel na vstupu jsou totožné. Dále všechna pravidla musí začínat totožným terminálním symbolem a nesmí se přepisovat na prázdný řetězec. Implementace levé faktorizace se skládá z několika nezávislých kroků. Nejprve se na základě konfliktních pravidel zjistí, které terminální symboly jsou souvisle od začátku pravé strany totožné. V bodě, kde se některé z pravidel začne lišit se provede rozdělení pravidel – tedy vytknutí totožných symbolů do jednoho pravidla. Během této činnosti se zavádí jeden nový neterminální symbol, kterým končí pravidlo s vytknutými symboly. Dále pro každé konfliktní pravidlo se vytvoří pravidlo nové bez vytknutých terminálních symbolů, jejichž levá strana je nově zavedený neterminál. Při vytváření transformovaných pravidel vznikají nové sémantické operace a nové dědičné a syntetizované atributy pro pomocné výpočty. Tyto nově vzniklé sémantické operace a atributy je nutné přidat do gramatiky. Posledním krokem je odstranění původních konfliktních pravidel z gramatiky.

3.5.3 Rohová substituce

Metoda provádějící rohovou substituci má název `substitution`. V prvním parametru je očekáván index pravidla, u něhož má dojít k provedení substituce a podmínkou je, že pravá strana začíná neterminálem. Pomocí druhého parametru se předává seznam indexů na pravidla, jejichž pravé strany se mají vložit za odpovídající neterminál předešlého pravidla. Pro každé vložení pravé strany se vytvoří jedno nové pravidlo. Původní pravidlo, do kterého se substituuje se z gramatiky odstraní. Pravidla, jejichž pravé strany jsou vkládány, musí v gramatice zůstat.

3.5.4 Pohlcení terminálního symbolu

Provedení transformace pohlcení terminálního symbolu má na starost metoda `terminalAbsorbtion`. Metodě je předán index pravidla, u kterého má dojít k provedení transformace. Dále je předána informace o pozici neterminálního symbolu pravé strany pravidla. Transformace ovlivní neterminální symbol indexovaný pozicí a terminální symbol, který bezprostředně následuje. Za tyto

dva symboly je vložen nově zavedený neterminální symbol pro který jsou vytvořena nová pravidla podle popisu v sekci 2.8.3

3.5.5 Transformace levé rekurze

Transformační procedura převedení levé rekurze na pravou rekurzi je realizována v metodě `leftRecursionElimination`. Metoda ve svých parametrech obdrží seznam indexů na pravidla obsahující levou rekurzi a seznam indexů na pravidla neobsahující levou rekurzi. Předpokládá se, že všechna předaná pravidla mají na levé straně týž neterminální symbol.

Kapitola **4**

Uživatelská část a testování

4.1 Kompilace a spuštění

Následující postup je určen pro zprovoznění programu pod operačním systémem *Linux* a překladačem *GCC*. Pod tímto systémem byl program vyvíjen a testován. Před spuštěním je nutné provést kompilaci zdrojových kódů. Správný proces kompilace zajišťuje soubor `Makefile`. Zkopírujte adresář `ll-generator` s celým svým obsahem na pevný disk počítače. Poté se v terminále přesuňte do právě vzniklého adresáře a zadejte příkaz `make compile`. Během několika vteřin se zkompilují veškeré zdrojové soubory a následně se provede slinkování. Přeložený binární soubor se uloží do adresáře `ll-generator/bin/ll1-gen`. Spuštění zkompilovaného programu se provede pomocí příkazu `make run`, nebo `./bin/ll1-gen` s příslušnými vstupy a parametry.

Ve výchozím nastavení program při spuštění na standardním vstupu očekává uživatelem definovanou gramatiku. Pokud načtení gramatiky proběhne v pořádku a gramatika je ve tvaru LL(1), nebo program nalezne posloupnost transformací pro převedení gramatiky do očekávaného tvaru, na standardní výstup vypíše kód reprezentující parser jazyka definovaného vstupní gramatikou. Pokud načtení neproběhne v pořádku, program vypíše chybovou hlášku a ukončí se.

Chování programu lze měnit pomocí přepínačů a parametrů zadaných přes příkazovou řádku. Zde je význam jednotlivých přepínačů.

-i filename přepne načítání vstupní gramatiky ze souboru `filename`

-o filename přepne generování parseru do souboru s názvem `filename`

-l filename změní název připojovaného lexikálního analyzátoru na `filename`

-g zapne ignorování nedeterminismu při zpracování gramatik

-e zapne ignorování chyb

-p zapne rozšířené informační výpisy během běhu programu

-d zakáže provádění transformací

--help zobrazí stručnou nápovědu

4.2 Napojení lexikálního analyzátoru

Po vygenerování parseru je nezbytným krokem připojení lexikálního analyzátoru. Bez tohoto kroku nebude možné parser zprovoznit. Generování kódu je navrženo takovým způsobem, aby napojení bylo pokud možno co nejsnažší a bez potřebné znalosti struktury generovaného parseru. Kompatibilní lexikální analyzátor nezbytně musí ve svém rozhraní poskytovat funkce nebo metody pro získávání následujících informací.

- Typ lexikálního elementu reprezentovaný datovým typem `int` nebo jiným kompatibilním
- Hodnoty syntetizovaných atributů lexikálních elementů (pokud jsou atributy použity)
- Načtení jednoho lexikálního elementu ze vstupního streamu

Použití lexikálního analyzátoru, který je součástí programu, je preferováno a jeho napojení je nejsnažší. V takovém případě stačí pouze v implementačním souboru doplnit v metodě `Input::read()` přiřazení syntetizovaných atributů z lexikálního analyzátoru. Pro připojení vlastního lexikálního analyzátoru je určen následující postup. Všechna místa v kódu, kde je potřeba provést změnu, jsou opatřena komentářem s uvozujícím slovem `HERE` spolu se stručnými instrukcemi. Pokud není ve vstupní gramatice k jednotlivým lexikálním elementům přiřazena hodnota symbolické konstanty odkazující na lexikální analyzátor, je potřeba v hlavičkovém souboru tyto konstanty ručně přiřadit. V opačném případě se tato akce provede automaticky. V implementačním souboru je nutné vložit direktivu `#include` s názvem hlavičkového souboru lexikálního analyzátoru. Tento krok se opět provede automaticky v případě, že je program spuštěn s přepínačem `-l` za nímž následuje název hlavičkového souboru. Všechny ostatní změny je potřeba provést v implementačním souboru v metodě `Input::read()`. Nejprve je potřeba změnit datový typ proměnné `lexem_in` na datový typ, kterým lexikální analyzátor reprezentuje lexikální element. Následně nahradit volání metody `read_lexem()` metodou nebo funkcí lexikálního analyzátoru realizující načtení lexikálního elementu ze vstupního streamu `is` a výsledek uložit do výše deklarované proměnné `lexem_in`. Dále je nutné ve struktuře `lexem_out` přiřadit typ lexikálního elementu type a případně dále přiřadit hodnoty atributů z proměnné `lexem_in` lexikálního analyzátoru. Proměnné reprezentující atributy lexikálních elementů mají názvy ve tvaru `at[název lexému][název atributu]`.

Dalším možným přístupem, jak připojit lexikální analyzátor, je vytvořit obalovací třídu se stejným rozhraním, jako dodaný lexikální analyzátor a následně propojit funkcionalitu s tímto rozhraním. Takovým přístupem je možné se téměř vyhnout ručnímu zásahu do generovaného parseru.

4.3 Testování implementovaného nástroje

Pro účely testování aplikace a odhalování případných chyb byly použity testovací gramatiky a vstupní věty pro kontrolu korektního provedení syntaktické a sémantické analýzy. Všechny testovací vstupy jsou k dispozici na přiloženém CD disku v adresáři `/ll-generator/work/`. Dále jsou k dispozici ukázkové vstupy pro program ANTLR ke srovnání produktu této práce s profesionálním softwarem.

Testovací data jsou členěna podle druhu testů. Testováno bylo korektní načítání gramatik, dále aplikace jednotlivých transformací a jejich kombinace. Nakonec bylo provedeno testování nad komplexními gramatikami, které svoji povahou odpovídají reálným aplikacím – například zjednodušený parser pro programovací jazyk *Pascal* nebo parser pro načítání a vyhodnocování aritmetických výrazů. Při testování nebyly zjištěny žádné závažné chyby.

Vybraná gramatika byla dále zapsána pro program ANTLR za účelem otestování funkčnosti a porovnání časové a paměťové složitosti se zmíněným programem. Srovnání proběhlo na gramatice aritmetických výrazů [16]. Vstupní data byla náhodně generována jednoduchým programem [17]. Velikosti vstupních dat jsou řádově od desítek bytů do jednoho MiB. Časově i paměťově vychází lépe implementovaný nástroj oproti programu ANTLR. Tento výsledek je pravděpodobně důsledkem toho, že ANTLR implementuje širší zpracování syntaktických chyb a poskytuje možnosti nejen pro sémantickou analýzu definovanou gramatikou, ale i budování AST, ke kterému poskytuje významnou podporu. Implementovaným nástrojem je také možné vybudovat AST, ale uživatel musí veškeré datové struktury k tomu potřebné sám implementovat a do gramatiky zadat příslušná sémantická pravidla. Výsledky testování jsou k porovnání v tabulce 4.1.

[16] Gramatika pro implementovaný nástroj je umístěna na přiloženém CD v adresáři: `ll-generator/work/grammars/test-arit-expr/gr-aritexpr.txt`.
Gramatika pro nástroj ANTLR je v adresáři:
`ll-generator/work/antlr-grammars/arit-expr/`.

[17] Generovaná vstupní data jsou k dispozici v adresáři:
`ll-generator/work/grammars/test-arit-expr/inputs/`.

velikost vstupu[B]	čas běhu[s]		paměť[B]	
	ll1-gen	ANTLR	ll1-gen	ANTLR
43	0,004	0,004	6 880	7 804
1 342	0,004	0,006	10 372	17 008
25 940	0,017	0,039	21 540	44 844
40 776	0,022	0,048	23 668	50 044
79 384	0,036	0,113	38 756	87 724
102 392	0,041	0,233	50 692	117 488
247 644	0,105	0,612	70 276	166 604
369 531	0,142	1,774	148 420	361 964
441 555	0,165	1,250	92 420	221 964
580 228	0,207	1,117	59 252	139 004
629 263	0,223	0,991	82 644	197 404
804 678	0,282	1,754	87 492	209 644
883 112	0,305	1,325	161 092	221 964

Tabulka 4.1: Srovnání parseru aritmetických výrazů s ANTLR

Závěr

Hlavní cíl práce – implementovat nástroj, který umožňuje generovat parser uživatelem zadanou gramatikou – byl úspěšně dosažen. Generované programy jsou schopny zpracovávat syntaxi i sémantiku zadaných vstupních vět a detekovat chyby. Program byl s úspěchem testován na zkušebních vstupních datech a porovnán s nástrojem ANTLR. Dále jsou v práci teoreticky rozebrány a implementovány základní transformační metody bezkontextových gramatik spolu s transformacemi sémantiky. Vzhledem k rozsahu problematiky je však mnoho dalších věcí, které by stály za dodatečnou implementaci. Pokud jde o transformace sémantiky, program v tuto chvíli není schopen zpracovat veškeré sémantické závislosti. Dále by bylo možné implementovat rozšíření o možnost automatického budování AST. Další věcí je vytvořit k aplikaci generátor lexikálního analyzátoru, díky kterému by se návrh parseru významně zjednodušil.

Celkově hodnotím výsledek práce pozitivně, protože díky této práci jsem získal hlubší porozumění problematiky zpracování textu a podařilo se mi naimplementovat nástroj, který může být v praxi velmi užitečný, přestože podobných aplikací již mnoho existuje.

Literatura

[1] Holub, J. BI-AAG – Přednáška: Základní pojmy, Chomskeho hierarchie. 2014, místnost Ballingův sál.

[2] Melichar, B. *Jazyky a překlady*. České vysoké učení technické v Praze, 2007, ISBN 978-80-01-02776-9.

[3] Janoušek, J. BI-PJP – Přednáška: LL syntaktická analýza – dokončení a implementace. 2014, místnost T2:C3-54.

[4] ANTLR3 Code Generation Targets. Dostupné z: `https://theantlrguy.atlassian.net/wiki/display/ANTLR3/Code+Generation+Targets`, [cit. 10.05.2016].

[5] Antlr. Dostupné z: `www.antlr.org`, [cit. 11.04.2016].

[6] Bison. Dostupné z: `https://www.gnu.org/software/bison/`, [cit. 11.04.2016].

[7] Java Compiler Compiler. Dostupné z: `https://javacc.java.net/`, [cit. 15.05.2016].

[8] The Lex and Yacc Page. Dostupné z: `http://dinosaur.compilertools.net/`, [cit. 15.05.2016].

[9] Chytil, M. *Automaty a gramatiky*. SNTL – Nakladatelství technické literatury, 1984, ISBN 04-012-84.

[10] Černá, I.; Křetínský, M.; Kučera, A. *Automaty a formální jazyky I*. Fakulta informatiky Masarykova univerzita, 2002.

[11] Dickey, T. BYACC. Dostupné z: `http://invisible-island.net/byacc/byacc.html`, [cit. 01.12.2015].

[12] BNF and EBNF: What are they and how do they work? Dostupné z: http://www.garshol.priv.no/download/text/bnf.html#id1, [cit. 10.05.2016].

[13] ANTLR Example in C. Dostupné z: http://contrapunctus.net/blog/2012/antlr-c, [cit. 11.05.2016].

Příloha A

Seznam použitých zkratek

CYK Cocke-Younger-Kasami algoritmus

GCC GNU Compiler Collection

AST Abstract syntax tree – abstraktní syntaktický strom

Příloha **B**

Adresářová struktura přiloženého CD nosiče

```
/
├── ll-generator ....................... parser generátor v jazyce C++
│   ├── bin .................... zde bude po kompilaci přeložený program
│   ├── ofiles .............. adresář pro neslinkované objektové soubory
│   ├── source .....................adresář se zdrojovými kódy programu
│   │   ├── generator
│   │   │   └── ...................zdrojové soubory pro generování kódu
│   │   ├── grammar
│   │   │   └── ..............zdrojové soubory pro reprezentaci gramatiky
│   │   ├── libs
│   │   │   └── .................... zdrojové soubory nezařazených utilit
│   │   ├── loader
│   │   │   └── ...................zdrojové soubory pro načítání gramatik
│   │   ├── transformer
│   │   │   └── ..............zdrojové soubory pro provádění transformací
│   │   ├── config.h
│   │   └── main.cpp..................................... soubor main
│   ├── work ................. adresář s testovacími a ukázkovými vstupy
│   │   ├── antlr-grammars ............... gramatiky pro nástroj ANTLR
│   │   └── grammars......gramatiky a vstupy pro implementovaný nástroj
│   └── Makefile .................................. kompilační soubor
└── thesis
    ├── BP_Uzel_Matej_2016.pdf .................. text bakalářské práce
    └── BP_Uzel_Matej_2016.tex ........... zdrojová podoba dokumentu
```

Příloha C

Gramatika pro načítání uživatelem zadaných gramatik

$G = (N, T, P, S)$
$T = \{$ *kw_grammar*, *kw_terms*, *kw_nonterms*, *kw_syn*, *kw_inh*, *kw_rules*, *kw_semantics*, *kw_start*, *ident*, *comma*, *assign*, *string*, *lbrace*, *rbrace*, *lpar*, *rpar* $\}$
$N = \{$ *ROOT*, *BODY*, *TERMS*, *DEF*, *NONTERMS*, *SYN*, *INH*, *RULES*, *START*, *RULES2*, *ID_LIST*, *SEMAN*, *SEMAN_LIST*, *AT_LIST*, *AT_LIST2* $\}$
$S =$ *ROOT*

Množina P obsahuje následující pravidla:

1. ROOT → kw_grammar lbrace BODY rbrace
2. ROOT → kw_grammar ident lbrace BODY rbrace
3. BODY → kw_terms TERMS BODY
4. BODY → kw_nonterms NONTERMS BODY
5. BODY → kw_syn SYN BODY
6. BODY → kw_inh INH BODY
7. BODY → kw_rules RULES BODY
8. BODY → kw_start START BODY
9. BODY → ε
10. TERMS → ident DEF comma TERMS
11. TERMS → ident DEF

12. DEF → assign string
13. DEF → ε
14. NONTERMS → ident comma NONTERMS
15. NONTERMS → ident
16. SYN → lbrace AT_LIST rbrace
17. INH → lbrace AT_LIST rbrace
18. AT_LIST → ident lbrace AT_LIST2 rbrace AT_LIST
19. AT_LIST → ident lbrace AT_LIST2 rbrace
20. AT_LIST2 → lpar string comma string rpar comma AT_LIST2
21. AT_LIST2 → lpar string comma string rpar
22. RULES → lbrace RULES2 rbrace
23. RULES2 → ident arrow ID_LIST SEMAN semicolon RULES2
24. RULES2 → ε
25. ID_LIST → kw_eps
26. ID_LIST → ident ID_LIST
27. ID_LIST → ident
28. SEMAN → kw_semantics lbrace string SEMAN_LIST rbrace
29. SEMAN → ε
30. SEMAN_LIST → comma string SEMAN_LIST
31. SEMAN_LIST → ε
32. START → assign ident

Příloha **D**

Gramatika pro kontrolu a překlad sémantických operací

$G = (N, T, P, S)$
$T = \{$ *ident*, *arrow*, *assign*, *lsqr*, *rsqr*, *numb*, *lpar*, *rpar*, *comma*, *plus*, *minus*, *mult*, *div* $\}$
$N = \{ S, EXPR, E, ARGS, BINOP \}$
$S = S$
Množina P obsahuje následující pravidla:

1. S → ident arrow ident assign EXPR
2. S → ident lsqr numb rsqr arrow ident assign EXPR
3. EXPR → EXPR BINOP E
4. E → lpar EXPR rpar
5. E → numb
6. E → ident
7. E → ident lpar ARGS rpar
8. E → ident lsqr numb rsqr arrow ident
9. E → ident arrow ident
10. ARGS → EXPR
11. ARGS → EXPR comma ARGS
12. BINOP → plus | minus | mult | div

Příloha **E**

Příklad transformace levé rekurze se sémantickými operacemi

Je dána gramatika $G_1(\{A, B, C\}, \{+, -, *, /, (,), n\}, P_1, A)$, Symboly A, B, C, n mají přidělen syntetizovaný atribut s číselného typu. Množina P_1 obsahuje pravidla:

$$\begin{array}{lll} 1. & A \to A + B & A_s^0 \coloneqq A_s^1 + B_s \\ 2. & A \to A - B & A_s^0 \coloneqq A_s^1 - B_s \\ 3. & A \to B & A_s \coloneqq B_s \\ 4. & B \to B * C & B_s^0 \coloneqq B_s^1 * C_s \\ 5. & B \to B / C & B_s^0 \coloneqq B_s^1 / C_s \\ 6. & B \to C & B_s \coloneqq C_s \\ 7. & C \to n & C_s \coloneqq n_s \\ 8. & C \to (A) & C_s \coloneqq A_s \end{array} \tag{E.1}$$

Gramatiku je nutné převést na tvar LL(1), protože pravidla $\{1, 2, 4, 5\}$ jsou levě rekurentní. Správnou volbou transformace je aplikace levé rekurze ve dvou krocích. Nejprve se aplikuje odstranění levé rekurze na rekurentní pravidla $\{1, 2\}$ a na pravidlo $\{3\}$ ukončující rekurzi. Následná aplikace téže transformace na rekurentní pravidla $\{4, 5\}$ a pravidlo ukončující rekurzi $\{6\}$ odstraní veškerou levou rekurzi z gramatiky a výsledná gramatika je LL(1). Transformace syntaktických pravidel a sémantických operací je provedeno podle 2.8.4. Gramatika $G_2(\{A, B, C, A', B'\}, \{+, -, *, /, (,), n\}, P_2, A)$ generuje stejný jazyk a vyhodnocuje sémantiku stejně jako gramatika G_1. Oproti původní gramatice G_1 po transformaci přibyly dědičné atributy *cmp* a syntetizované atri-

buty s symbolům $A^{'}$ a $B^{'}$. Pravidla množiny P_2 jsou:

$$\begin{array}{rllll} 1. & A \to BA^{'} & A^{'}_{cmp} \coloneqq B_s & A_s \coloneqq A^{'}_s \\ 2. & A^{'} \to +BA^{'} & A^{'}_{cmp} \coloneqq A^{'0} + B_s & A^{'0}_s \coloneqq A^{'1}_s \\ 3. & A^{'} \to -BA^{'} & A^{'}_{cmp} \coloneqq A^{'0} - B_s & A^{'0}_s \coloneqq A^{'1}_s \\ 4. & A^{'} \to \varepsilon & A^{'}_s \coloneqq A^{'}_{cmp} & \\ 5. & B \to CB^{'} & B^{'}_{cmp} \coloneqq C_s & B_s \coloneqq B^{'}_s \\ 6. & B^{'} \to *CB^{'} & B^{'}_{cmp} \coloneqq B^{'0} * C_s & B^{'0}_s \coloneqq B^{'1}_s \\ 7. & B^{'} \to /CB^{'} & B^{'}_{cmp} \coloneqq B^{'0}/C_s & B^{'0}_s \coloneqq B^{'1}_s \\ 8. & B^{'} \to \varepsilon & B^{'}_s \coloneqq B^{'}_{cmp} & \\ 9. & C \to n & C_s \coloneqq n_s & \\ 10. & C \to (A) & C_s \coloneqq A_s & \end{array} \tag{E.2}$$