VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

GRAMATICKÉ SYSTÉMY A JEJICH APLIKACE

GRAMMAR SYSTEMS AND THEIR APPLICATIONS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **DALIBOR KŘÍČKA**
AUTHOR

VEDOUCÍ PRÁCE **prof. RNDr. ALEXANDR MEDUNA, CSc.**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

153858

Ústav: Ústav informačních systémů (UIFS)
Student: **Křička Dalibor**
Program: Informační technologie
Název: **Gramatické systémy a jejich aplikace**
Kategorie: Teoretická informatika
Akademický rok: 2023/24

Zadání:

1. Seznamte se podrobně s gramatickými systémy a jejich vlastnostmi.
2. Dle pokynů vedoucího zaveďte alternativní typy gramatických systémů.
3. Studujte vlastnosti gramatických systémů zavedených v předchozím bodě a porovnejte je s již zavedenými gramatickými systémy. Zaměření tohoto studia konzultujte s vedoucím.
4. Demonstrujte aplikovatelnost zavedených systémů v oblasti syntaktické analýzy. Zaměřte se na kombinovanou syntaktickou analýzu založenou na systémech z bodu 2.
5. Aplikujte a implementujte syntaktickou analýzu navrženou v předchozím bodě. Testujte ji na sadě minimálně 10 příkladů.
6. Zhodnoťte dosažené výsledky. Diskutujte další vývoj projektu.

Literatura:

1. Meduna, A.: Automata and Languages, Springer, 2000, ISBN 978-1-4471-0501-5
2. Rozenberg, G. and Salomaa, A. (eds.): Handbook of Formal Languages, Volume 1 through 3, Springer, 1997, ISBN 3-540-60649-1
3. Aho, A. V., Sethi, R., Ullman, J. D.: Compilers : principles, techniques, and tools, Addison-Wesley, 2nd ed., 2007, ISBN 0-321-48681-1

Při obhajobě semestrální části projektu je požadováno:
Splnění prvních 2 bodů zadání a část bodu 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Meduna Alexandr, prof. RNDr., CSc.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1.11.2023
Termín pro odevzdání: 9.5.2024
Datum schválení: 30.10.2023

Abstrakt

Cílem této práce je zavést nový typ kooperačně distribuovaného (CD) gramatického systému na základě typů již existujících, následně konkrétní gramatický systém tohoto typu definovat a aplikovat ho v rámci syntaktického analyzátoru. Nově zavedený typ kombinuje vlastnosti hybridních CD gramatických systémů a CD gramatických systémů s vnitřním řízením a klade důraz na determinismus komunikačního protokolu. Konkrétně definovaný gramatický systém aplikuje tři metody syntaktické analýzy (prediktivní LL, precedenční a SLR) a přijímá podmnožinu jazyka C++. Praktický aspekt práce demonstruje aplikovatelnost zmíněného gramatického systému formou konzolové aplikace implementující přední část překladače, do které je gramatický systém zakomponován.

Abstract

This thesis aims to introduce a new type of cooperating distributed (CD) grammar system based on already existing types, then define a specific grammar system of the new type and apply it within a parser. The newly introduced type combines features of the hybrid CD grammar system and CD grammar system with internal control and emphasizes the determinism of cooperation protocol. The explicitly defined grammar system applies three methods of syntactic analysis (LL predictive, precedence and SLR) and accepts a subset of C++ language. The practical aspect of this thesis demonstrates the applicability of the mentioned grammar system by console application, which implements the forepart of a compiler based on this system.

Klíčová slova

gramatický systém, kooperačně distribuovaný gramatický systém, CD gramatický systém, překladač, syntaktický analyzátor, bezkontextová gramatika, formální jazyk, prediktivní syntaktická analýza, precedenční syntaktická analýza, SLR syntaktická analýza, tabulka syntaktické analýzy

Keywords

grammar system, cooperating distributed grammar system, CD grammar system, compiler, parser, context-free grammar, formal language, predictive parsing, precedence parsing, SLR parsing, parsing table

Citace

KŘÍČKA, Dalibor. *Gramatické systémy a jejich aplikace*. Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. RNDr. Alexandr Meduna, CSc.

Gramatické systémy a jejich aplikace

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana prof. RNDr. Alexandra Meduny CSc. Další informace mi poskytl Ing. Zbyněk Křivka Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Dalibor Kříčka
2. května 2024

Poděkování

Mé poděkování patří prof. RNDr. Alexandrovi Medunovi CSc. za odborné vedení, vstřícný přístup a cenné rady, které vedly k vytvoření této práce. Rád bych také poděkoval své rodině a blízkým za podporu a trpělivost.

Obsah

Seznam obrázků

Kapitola 1

Úvod

K oblasti informačních technologií neodmyslitelně patří proces psaní počítačového programu (neboli programování). Počítačový program je posloupnost instrukcí sloužící jako prostředek pro komunikaci mezi člověkem a počítačem. Dnes populární programovací jazyky jsou většinou koncipovány tak, aby alespoň v malé míře připomínaly jazyk přirozený a psaní v nich bylo pro uživatele srozumitelnější a intuitivnější. To však s sebou přináší nutnost program v takovém jazyce přeložit, aby mohl být počítačem zpracován. V počáteční fázi zmíněného překladu je provedena kontrola, zdali je vůbec překládaný program v souladu s gramatickými pravidly předpokládaného jazyka. Tato kontrola se nazývá syntaktická analýza a je předmětem této práce.

Syntaktická analýza je nezbytnou součástí překladače každého vyššího kompilovaného jazyka a téma formálních jazyků a konstrukce překladačů je tak v oblasti teoretické informatiky aktuální. Ačkoliv se teoretická informatika na první pohled může jevit jako ta nudná oblast oboru informačních technologií, je obdivuhodné, jak zajímavou se může stát v momentě, kdy si člověk uvědomí, že za pomoci základních znalostí diskrétní matematiky a logiky je možné pochopit, nebo naopak popsat některé problémy, se kterými se v informatice setkáváme dnes a denně. Řeč je například o teorii výpočetní složitosti, teorii grafů nebo právě o teorii automatů, které velmi úzce souvisí právě s teorií formálních jazyků a syntaktickou analýzou.

Syntaktický analyzátor, coby prostředek pro provedení syntaktické analýzy (také počítačový program), může být založen na bezkontextové gramatice, jež specifikuje konkrétní gramatická pravidla přijímaného jazyka. Je-li založen na více gramatikách, musí mezi sebou jednotlivé gramatiky komunikovat na základě stanoveného komunikačního protokolu. Tento útvar je nazývaný jako gramatický systém a přesně těmi se tato práce zabývá.

Původní zmínky o gramatických systémech vedou až do minulého století a za jejich formální definicí stojí snaha dohnat to, co již v praxi bylo běžně využíváno. Motivem jejich vzniku tedy bylo formálně, pomocí matematických prostředků, vyjádřit situaci, kdy více různých gramatik spolupracuje, komunikuje mezi sebou a dohromady pracují jako jeden celek. Využití více spolupracujících gramatik v rámci jednoho syntaktického analyzátoru přirozeně vyplývá ze základního principu dělení problému na podproblémy. To umožňuje jednotlivým částem syntaktického analyzátoru, založeným na konkrétních gramatikách, které se nazývají komponenty, zaměřit se na určitý aspekt jazyka, který navíc může každá komponenta analyzovat svým specifickým způsobem.

V této práci budou představeny kooperačně distribuované (CD) gramatické systémy, přesněji řečeno jejich základní vlastnosti, definice a způsob komunikace mezi gramatikami. V rámci toho budou prezentovány specifické typy CD gramatických systémů, ze kterých

bude práce dále vycházet. Konkrétně se jedná o hybridní CD gramatický systém a o gramatický systém s vnitřním řízením. Cílem je tedy zavést nový unikátní typ gramatických systémů, jehož přední vlastností bude přívětivost aplikovatelnosti systému v praxi (v rámci syntaktického analyzátoru). Důraz bude kladen na specifikaci komunikačního protokolu tak, aby bylo zřejmé, kdy mají jednotlivé gramatiky začít pracovat a kdy naopak předat řízení dál. Dalším krokem bude definovat konkrétní gramatický systém nově zavedeného typu, jež bude přijímat podmnožinu jazyka C++.

Praktickou částí bude implementace přední části překladače, tedy naprogramování lexikálního a syntaktického analyzátoru, který bude rozhodovat o správnosti jazyka na vstupu. Syntaktický analyzátor bude založený na představeném gramatickém systému, který přijímá podmnožinu jazyka C++. Pro realizaci jednotlivých komponent bude využito různých typů syntaktické analýzy. Jedná se o LL, precedenční a SLR syntaktickou analýzu. Kromě verdiktu, zdali je program na vstupu po lexikální a syntaktické stránce validní či nikoliv, je po implementovaném syntaktickém analyzátoru požadováno, aby byl schopen zotavit se po chybě a poskytnout zpětnou vazbu uživateli v podobě stručného popisu chyby a lokalizování chyby v analyzovaném souboru.

První část práce má za cíl seznámit čtenáře s teoretickými i praktickými znalostmi z oblasti formálních jazyků a překladačů. Druhá část prezentuje gramatické systémy a implementaci praktické části. Kapitola 2 je věnována teoretickým základům jako je definice řetězce, formálního jazyka nebo bezkontextové gramatiky včetně zásadního pojmu derivace. Kapitola 3 představuje úvod do oblasti kompilátorů, později se zaměří na syntaktickou analýzu a to především na jednotlivé metody, kterými lze syntaktickou analýzu prakticky realizovat. Uvedena bude LL prediktivní, precedenční a SLR syntaktická analýza a postupy konstrukce jejich tabulek analýzy. V kapitole 4 je objasněn pojem kooperačně distribuovaný (CD) gramatický systém. Také zde jsou popsány již existující typy gramatických systémů, na kterých je vybudován nově zavedený systém. Ten je zavedený a formálně popsaný v kapitole 5, kde je následně konkrétní navržený gramatický systém tohoto typu definovaný i včetně jazyka, který přijímá. V poslední kapitole 6 je popsaný způsob realizace praktické části práce, tedy aplikace navrženého gramatického systému v rámci syntaktického analyzátoru.

Kapitola 2

Základy teorie formálních jazyků

Tato kapitola je věnována představení základní teorie formálních jazyků, na níž je vystavěna celá tato práce. V první řadě budou definovány elementární pojmy jako *abeceda*, *řetězec* a *jazyk*. Následovat bude představení *bezkontextových gramatik* a k nim vztahujících se pojmů. Znalost bezkontextových gramatik a jejich *derivací* je totiž nezbytná pro pochopení následující kapitoly, která pojednává o syntaktické analýze. Pro jednodušší porozumění jednotlivým oblastem následuje téměř za všemi definicemi praktický příklad. Pojmy a terminologie z této kapitoly jsou převzaty z [2] (kapitola Abecedy, řetězce a jazyky), [7], [8], [9] a [13].

2.1 Abeceda, řetězec a jazyk

Před objasněním samotného termínu *formální jazyk* je nezbytné se seznámit s pojmy *abeceda* a *řetězec*.

Abeceda

Definice 2.1 (viz [9])**.** *Abeceda* Σ je konečná, neprázdná množina prvků, které jsou nazývany *symboly abecedy* Σ.

Příklad 2.1. Přirozeným příkladem je anglická abeceda obsahující 52 symbolů, známých jako malá a velká písmena a až z. Dalším příkladem může být abeceda Σ_{bin}, pomocí které lze reprezentovat binární čísla nebo analogicky abeceda Σ_{hex} pro reprezentaci hexadecimálních čísel, kde

- $\Sigma_{bin} = \{0, 1\}$,
- $\Sigma_{hex} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f\}$.

Symboly abecedy nemusí být pouze jednotlivé znaky nebo číslice, jako je uvedeno v příkladech výše, ale mohou jimi být třeba *neterminály bezkontextové gramatiky*, jak bude vysvětleno v podkapitole 2.2.

Řetězec

Definice 2.2 (viz [9] a [2] kapitola Abecedy, řetězce a jazyky)**.** *Řetězec* nad abecedou Σ je konečná sekvence symbolů z Σ. Řetězec, který obsahuje právě 0 symbolů se nazývá *prázdný řetězec* a je označován řeckým písmenem ϵ. Je-li x řetězcem nad Σ a $a \in \Sigma$, potom xa je

řetězcem nad Σ. Výsledkem konkatenace libovolného řetězce x s prázdným řetězcem ϵ je opět x, tedy $\epsilon x = x$.

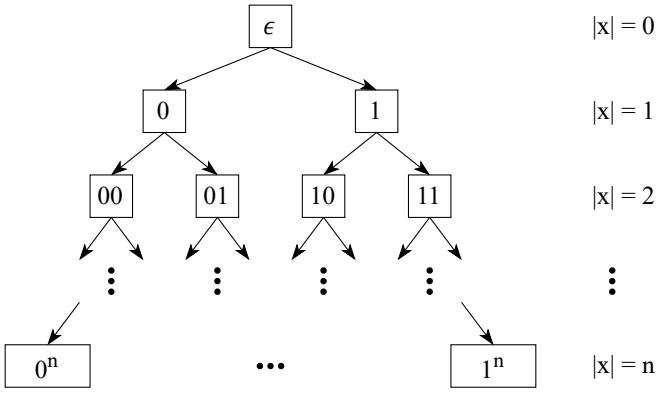
Množina všech řetězců nad abecedou Σ, včetně prázdného řetězce, se značí jako Σ^*. Jako Σ^+ je značena množina všech řetězců nad abecedou Σ s výjimkou prázdného řetězce. Platí tedy vztah $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$ (viz [13]).

Definice 2.3. Uvažujme řetězec x nad abecedou Σ. *Délku* x, neboli počet symbolů, ze kterých je řetězec složen, značíme jako $|x|$. V případě, že $x = \epsilon$, pak $|x| = 0$.

Příklad 2.2. Mějme abecedu $\Sigma_{bin} = \{0, 1\}$ z příkladu 2.1, kde

$$\epsilon, 0, 1, 00, 01, 10, 11, 000, 001$$

je několik vybraných řetězců nad Σ_{bin}. Na obrázku 2.1 je možné vidět stromové schéma demostrující Σ^*_{bin}



Obrázek 2.1: Stromové schéma demonstrující Σ^*_{bin}, kde uzly reprezentují jednotlivé řetězce x nad abecedou Σ_{bin} a kde hloubka, na které se řetězec nachází, demonstruje jeho délku $|x|$.

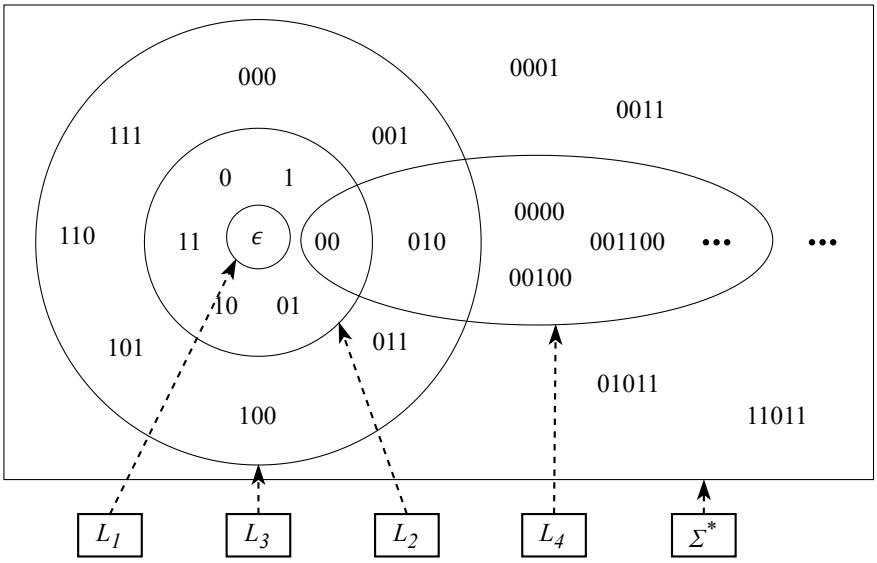
Jazyk

Definice 2.4 (viz [13])**.** Množina řetězců L, pro kterou platí $L \subseteq \Sigma^*$ je nazývána jako *jazyk* L nad abecedou Σ. Jazykem tedy může být zcela libovolná podmnožina všech řetězců nad danou abecedou. Jako *věta jazyka* L je označován řetězec x pro který platí $x \in L$.

Příklad 2.3. Mějme jazyky L_1, L_2, L_3 a L_4 nad abecedou $\Sigma_{bin} = \{0, 1\}$ z příkladu 2.1, kde

- $L_1 = \{\epsilon\}$
- $L_2 = \{x : |x| \leq 2\} = \{\epsilon, 0, 1, 00, 01, 10, 11\}$
- $L_3 = \{x : |x| \leq 3\} = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111\}$
- $L_4 = \{0^n 1^m 0^n : n \geq 1, m \geq 0\} = \{00, 010, 0000, 00100, 001100, \dots\}$

Při bližším pohledu na jednotlivé jazyky si je možné povšimnout vztahu $L_1 \subseteq L_2 \subseteq L_3$. Dále, že řetězec $00 \in L_2, L_3, L_4$ a řetězec $010 \in L_3, L_4$. Tyto vztahy je možné podrobněji sledovat ve Vennově diagramu na obrázku 2.2.



Obrázek 2.2: Vennův diagram vyobrazující vztahy mezi jazyky L_1, L_2, L_3 a L_4 nad abecedou Σ_{bin}.

2.2 Bezkontextové gramatiky

Bezkontextová gramatika (zkráceně gramatika) má schopnost generovat řetězce jazyka specifikovaného konečným počtem gramatických pravidel. Abeceda gramatiky obsahuje dva typy symbolů. *Terminální symboly*, jež představují typy jednotlivých tokenů (lexémy) a *neterminální symboly*, které zastupují celé syntaktické konstrukce (například větvení, cykly či výrazy) a které mohou být dále rozvedeny jedním z množiny pravidel (dále v textu zkracováno na *neterminály* a *terminály*)(viz [7]).

Definice 2.5 (viz [7])**.** Bezkontextová gramatika je čtveřice

$$G = (N, T, P, S),$$

kde

- N a T představují disjunktní abecedy *neterminálů* a *terminálů*,
- P je konečná množina pravidel ve tvaru $A \to x$ ($A \in N$, $x \in (N \cup T)^*$),
- S je počáteční neterminál ($S \in N$).

Příklad 2.4. Mějme gramatiku $G_{eg} = (\{S, B\}, \{a, b, c\}, P, S)$, kde

$$\begin{aligned} P \quad = \{ \quad & 1\colon S \to aSc, \\ & 2\colon S \to B, \\ & 3\colon B \to BB, \\ & 4\colon B \to b \quad \}, \end{aligned}$$

která generuje bezkontextový jazyk $L(G_{eg}) = \{a^n b^m c^n \colon n \geq 0, m \geq 1\}$. Za povšimnutí stojí fakt, že právě kvůli stejnému počtu n výskytů terminálů a a c, není možné jazyk $L(G_{eg})$ generovat pomocí *regulárního výrazu* (viz [13]).

Konvence 2.1 (viz [7])**.** Cílem zavedení následující konvence je předejití opakovanému upřesňování, jaký symbol či řetězec symbolů znak v textu reprezentuje. Pro bezkontextovou gramatiku $G = (N, T, P, S)$ tedy mějme znaky

- A, B, C, D, E, S reprezentují *neterminál* z množiny N, S je *počáteční symbol*,
- a, b, c, d, e reprezentují *terminál* z množiny T,
- U, V, W, X, Y, Z reprezentují *neterminál* nebo *terminál* z množiny $(N \cup T)$,
- u, v, w, x, y, z reprezentují řetězec *neterminálů* a *terminálů* z $(N \cup T)^*$,
- $\alpha, \beta, \gamma, \delta$ reprezentují řetězec *terminálů* z T^*.

Definice 2.6 (viz [8])**.** *Větná forma* je libovolný řetězec w vyprodukovaný gramatikou G z počátečního symbolu S konečnou sekvencí derivačních kroků.

$$S \Rightarrow^* w$$

Definice 2.7 (viz [7])**.** *Větou* gramatiky G je větná forma w taková, že $w \in T^*$. Množina všech *vět* G je *jazyk* $L(G)$ generovaný gramatikou G.

$$L(G) = \{w \colon w \in T^*, S \Rightarrow^* w \text{ v } G\}$$

Derivační krok

Upravení větné formy (viz definice 2.6) za použití gramatického pravidla je nazýváno *derivačním krokem*. Zpravidla se jedná o nahrazení levé strany pravidla za stranu pravou. Provedení derivačního kroku v gramatice G nad větnou formou uAv aplikací pravidla $p \colon A \to x \in P$ značíme (viz [2] kapitola Modely bezkontextových jazyků)

$$uAv \Rightarrow uxv \ [p]$$

Sekvenci $n \in (\mathbb{N} \cup 0)$ za sebou následujících derivačních kroků, kde $u_0, u_1, \ldots, u_{n-1}, u_n \in (N \cup T)^*$, značenou jako

$$u_0 \Rightarrow u_1 \Rightarrow \cdots \Rightarrow u_{n-1} \Rightarrow u_n,$$

lze zkráceně zapsat jako

$$u_0 \Rightarrow^n u_n.$$

Pro vyjádření neurčitého počtu derivačních kroků je možné konkrétní hodnotu n zaměnit za symboly $*$ a $+$ s následujícím významem

$$u_0 \Rightarrow^* u_n \text{ pro } n \geq 0 \quad \text{a} \quad u_0 \Rightarrow^+ u_n \text{ pro } n \geq 1.$$

Konvence 2.2 (viz [8])**.** Pro vyznačení, že v derivačním kroku $uAv \Rightarrow uxv$ bude přepsán právě neterminál A, je možné využít notaci $u\underline{A}v \Rightarrow uxv$.

Příklad 2.5. Berme v potaz gramatiku G_{eg} z příkladu 2.4. Dále uvažujme řetězec terminálů $\alpha = aaabbccc$, jež je větou jazyka $L(G_{eg})$. Víme-li, že α je větou jazyka $L(G_{eg})$, je poté možné tvrdit, že platí

$$S \Rightarrow^+ \alpha$$

Podrobným rozepsáním jednotlivých přímých derivačních kroků lze poté ověřit, že α je skutečně generována gramatikou G_{eg}.

$$\underline{S} \Rightarrow a\underline{S}c\ [1] \Rightarrow aa\underline{S}cc\ [1] \Rightarrow aaa\underline{S}ccc\ [1] \Rightarrow aaa\underline{B}ccc\ [2] \Rightarrow aaa\underline{B}Bccc\ [3] \Rightarrow$$

$$\Rightarrow aaab\underline{B}ccc\ [4] \Rightarrow aaabbccc\ [4]$$

Nyní lze navíc spočítat, že došlo právě k *sedmi* derivačním krokům a říci, jaká pravidla $p \in P$ byla použita a v jakém pořadí. Původní výraz $S \Rightarrow^{+} aaabbccc$ je možné specifikovat jako

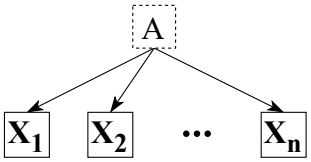
$$S \rightarrow^{7} \alpha\ [1112344]$$

Derivační strom

Derivační strom t graficky reprezentuje strukturu derivace gramatiky G pomocí stromového schématu. Derivační strom je možné použít za předpokladu, že pořadí, ve kterém byla jednotlivá gramatická pravidla aplikována, je nepodstatné pro řešený problém. Při konstrukci t totiž dochází ke ztrátě tohoto pořadí. Derivační strom se skládá z dílčích podstromů reprezentujících jednotlivá aplikovaná pravidla, které se nazývají *pravidlové stromy* (viz [7]).

Definice 2.8 (viz [7])**.** Nechť G je gramatika a $p : A \rightarrow x \in P$. Pro každé pravidlo p existuje *pravidlový strom* pt, pomocí kterého je možné p reprezentovat. Kořenovým uzlem pt je zpravidla *neterminál* A z levé strany pravidla p. Pravou stranu p reprezentuje $|x|$ listových uzlů označených zleva doprava jednotlivými symboly z řetězce x. Tyto listové uzly jsou přímými potomky kořenového uzlu A. Z této definice vyplývá, že pt má právě *dvě* úrovně.

Uvažujme pravidlo $p : A \rightarrow X_1X_2 \ldots X_n$, kde $n \geq 1$. S pravidlem p koresponduje pravidlový strom vyobrazený na obrázku 2.3.



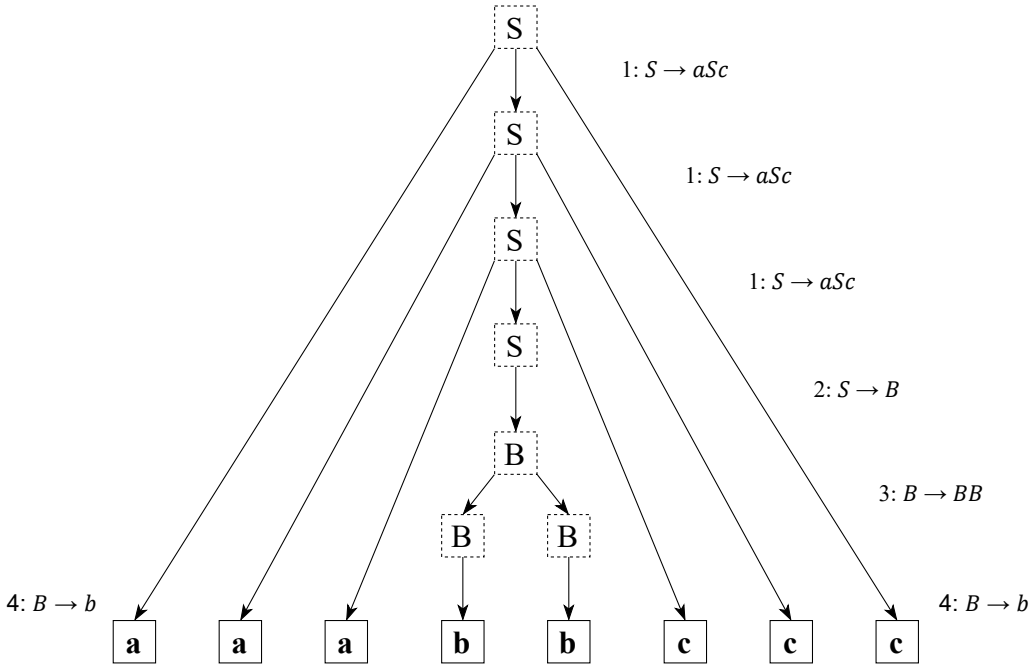
Obrázek 2.3: Pravidlový strom korespondující s pravidlem p: $A \rightarrow X_1X_2 \ldots X_n$, kde $n \geq 1$ (viz [7]).

Definice 2.9 (viz [7])**.** Nechť G je gramatika. Pro *derivační strom* t gramatiky G platí, že kořenovým uzlem t je počáteční neterminál S. Každý elementární podstrom, jenž se v t vyskytuje, je pravidlovým stromem korespondujícím s pravidlem $p \in P$. Sekvence listových úzlů t představuje větnou formu vygenerovanou pomocí G.

Příklad 2.6. Tento příklad navazuje na příklad 2.5, ve kterém je podrobně rozepsána derivace, díky níž je generována věta $aaabbccc \in L(Geg)$. *Derivační strom* této derivace je vyobrazen na obrázku 2.4.

Nejlevější derivace

Derivaci je možné považovat za *nejlevější* pouze v případě, kdy je v každém derivačním kroku přepsán nejlevěji vyskytující se neterminál ve větné formě.



Obrázek 2.4: Derivační strom vyobrazující derivaci, díky níž je generována věta $aaabbccc \in L(Geg)$.

Definice 2.10 (viz [9])**.** Mějme bezkontextovou gramatiku G. Nechť $p\colon A \to x \in P$, $\alpha \in T^*$ a $u \in (N \cup T)^*$, potom G provede nejlevější derivační krok tak, že upraví větnou formu αAu na αxu aplikací pravidla p. Tato situace je formálně značena jako

$$\alpha Au \ {}_{lm}\Rightarrow \alpha xu \ [p].$$

Nejpravější derivace

Derivaci je možné považovat za *nejpravější* pouze v případě, kdy je v každém derivačním kroku přepsán nejpravěji vyskytující se neterminál ve větné formě.

Definice 2.11 (viz [9])**.** Mějme bezkontextovou gramatiku G. Nechť $p\colon A \to x \in P$, $\alpha \in T^*$ a $u \in (N \cup T)^*$, potom G provede nejpravější derivační krok tak, že upraví větnou formu $uA\alpha$ na $ux\alpha$ aplikací pravidla p. Tato situace je formálně značena jako

$$uA\alpha \ {}_{rm}\Rightarrow ux\alpha \ [p].$$

Příklad 2.7. Mějme gramatiku $G_{der} = (\{S, A, B\}, \{a, b, c\}, P, S)$, kde

$$\begin{aligned} P \quad = \{ \quad & 1\colon S \to SbB, \\ & 2\colon S \to B, \\ & 3\colon B \to BcA, \\ & 4\colon B \to A, \\ & 5\colon A \to a \quad \}, \end{aligned}$$

již generuje bezkontextový jazyk $L(G_{der}) = \{a(ba + ca)^n\colon n \geq 0\}$. Nechť $\alpha = abaca$ je věta jazyka $L(G_{der})$.

Nejlevější derivace věty α má tvar

$$\underline{S} \ {}_{lm}\Rightarrow \underline{S}bB \ [1] \ {}_{lm}\Rightarrow \underline{B}bB \ [2] \ {}_{lm}\Rightarrow \underline{A}bB \ [4] \ {}_{lm}\Rightarrow ab\underline{B} \ [5] \ {}_{lm}\Rightarrow ab\underline{B}cA \ [3] \ {}_{lm}\Rightarrow$$

$$ {}_{lm}\Rightarrow ab\underline{A}cA \ [4] \ {}_{lm}\Rightarrow abac\underline{A} \ [5] \ {}_{lm}\Rightarrow abaca \ [5]$$

Zkráceně zapsáno jako

$$S \ {}_{lm}\Rightarrow^{*} abaca \ [12453455]$$

Nejpravější derivace věty α má tvar

$$\underline{S} \ {}_{rm}\Rightarrow Sb\underline{B} \ [1] \ {}_{rm}\Rightarrow SbBc\underline{A} \ [3] \ {}_{rm}\Rightarrow Sb\underline{B}ca \ [5] \ {}_{rm}\Rightarrow Sb\underline{A}ca \ [4] \ {}_{rm}\Rightarrow \underline{S}baca \ [5] \Rightarrow_{rm}$$

$$ {}_{rm}\Rightarrow \underline{B}baca \ [2] \ {}_{rm}\Rightarrow \underline{A}baca \ [4] \ {}_{rm}\Rightarrow abaca \ [5]$$

Zkráceně zapsáno jako

$$S \ {}_{rm}\Rightarrow^{*} abaca \ [13545245]$$

Kapitola 3

Základní přístupy k syntaktické analýze

Tato kapitola pojednává o několika různých metodách *syntaktické analýzy*, pomocí kterých je možné v praxi implementovat *syntaktický analyzátor*. Pro uvedení do kontextu bude zpočátku představen základní princip funkce *kompilátorů*. Následně již budou postupně rozebrány jednotlivé principy *syntaktické analýzy*, podle způsobu konstrukce derivačního stromu. V rámci přístupu *shora dolů*, bude prezentována metoda *prediktivní LL* syntaktické analýzy. V rámci přístupu *zdola nahoru* budou poté představeny *precedenční* a *SLR* syntaktická analýza. Jelikož jsou analyzátory založené na zmíněných metodách řízeny tabulkou, bude vysvětlen také způsob, jak ke každé metodě tabulku sestrojit. Pojmy, terminologie a algoritmy prezentované v této kapitole vychází z [2] kapitola Syntaktická analýza shora dolů, [3] kapitola Bottom-Up Parsing, [4], [7], [8], [9] a [10].

3.1 Úvod do kompilátorů

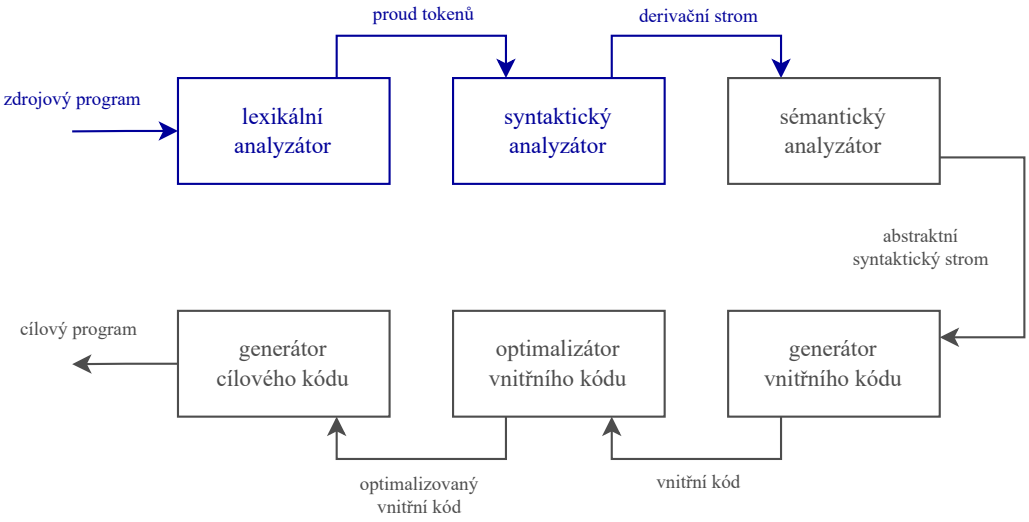
Kompilátor je nástroj sloužící pro překlad *zdrojového programu* napsaného ve *zdrojovém jazyce* do *cílového programu* v *cílovém jazyce* tak, že funkcionalita obou programů je totožná. V praxi se kompilátory běžně využívají pro překlad programu napsaného programátorem v obvykle ve vyšším programovacím jazyce (C++, Java) do strojového kódu (viz [8]).

Kompilace musí z pravidla projít několika fázemi, během kterých je zdrojový kód analyzován (je zkotrolována jeho správnost) a přeložen do cílového jazyka. Jedná se konkrétně o fáze vyobrazené na obrázku 3.1.

Tato práce se věnuje primárně fázi syntaktické analýzy, která bezprostředně navazuje na fázi analýzy lexikální. Proto budou dále více přiblíženy právě tyto dvě části překladače.

Lexikální analýza

Funkcí lexikálního analyzátoru (skeneru) je znak po znaku procházet zdrojový program a dělit jej na takzvané *lexémy*. Lexém lze chápat jako elementární jednotku jazyka programu, například klíčová slova, identifikátory proměnných, konstanty, ale také samostatné závorky či středníky. Lexémy jsou v další fázi překladu reprezentovány pomocí tzv. *tokenů*, které mohou kromě konkrétního typu disponovat také informací o hodnotě (např. hodnotou u konstanty nebo názvem u identifikátoru). Skener poskytuje posloupnost jednotlivých tokenů syntaktickému analyzátoru (více v [4]).



Obrázek 3.1: Obecné schéma kompilátoru vyobrazující jednotlivé části překladu včetně jejich vstupů a výstupů. Modře vyznačenou částí se zabývá tato práce.

Syntaktická analýza

Syntaktická analýza (dále jen *SA*) je proces ověřující správnost syntaktické konstrukce jazyka, ve kterém je zdrojový kód napsán. Tato syntaxe je zpravidla specifikována gramatikou, jež se skládá z konečného počtu pravidel.

Posloupnost tokenů, jež je označována jako *řetězec tokenů*, je vstupem syntaktického analyzátoru. Nad tímto řetězcem jsou poté aplikována gramatická pravidla. Z faktu, že jsou na vstup dodávané tokeny v rozporu s těmito pravidly, vyplývá, že jazyk není danou gramatikou generován a je vyhodnocen jako syntakticky chybný (lze nalézt v [8] a [4]).

Výsledkem *SA* je konstrukce derivačního stromu, která reprezentuje aplikaci jednotlivých gramatických pravidel potřebných pro přijmutí jazyka na vstupu, viz příklad 2.6.

Syntax

Slovo syntax není spjato jen s jazyky v oboru informačních technologií, ale je spojeno také s jazyky přirozenými, které lidstvo používá pro komunikaci jak v psané, tak v mluvené podobě. Přirozený jazyk slouží pro komunikaci člověka s ostatními lidmi. Je velmi komplexní, flexibilní, expresivní a ovlivňují jej emoce, záměr nebo i oblast, kde se autor textu/řeči nachází. Na druhou stranu jazyk programovací slouží pro komunikaci člověka s počítačem a musí být přesný, stručný, jednoznačný a efektivní.

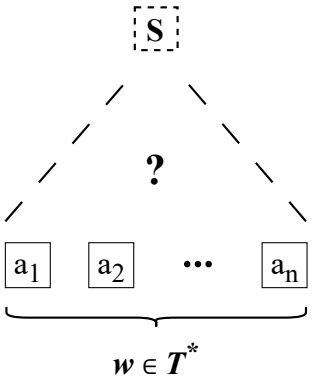
Ačkoliv je rozdíl mezi těmito typy jazyků značný, právě kvůli faktu, že každý slouží pro jiný účel, cíl mají společný. Cílem je předat informaci interpretovanou pomocí daného jazyka a k němu vázajících se pravidel tak, aby jí druhá strana komunikace porozuměla (více na [10]).

3.2 Metody syntaktické analýzy

Úkolem *syntaktického analyzátoru* je ověřit syntaktickou správnost zdrojového programu převedeného lexikálním analyzátorem na řetězec tokenů. Syntaktická správnost je kontro-

lována za pomoci bezkontextové gramatiky $G = (N, T, P, S)$ (bezkontextová gramatika vysvětlena v podkapitole 2.2), která generuje jazyk $L(G)$. Tokeny představují terminály T gramatiky G. Řetězec tokenů na vstupu syntaktického analyzátoru lze označit jako řetězec terminálů $w \in T$. Cílem syntaktické analýzy je tedy nalézt derivaci $S \Rightarrow^* w$ gramatiky G, aby bylo prokázáno že w je větou jazyka generovaného gramatikou G, formálně zapsáno jako $w \in L(G)$. Pokud derivace $S \Rightarrow^* w$ neexistuje, potom $w \notin L(G)$ a syntaktický analyzátor vyhodnotí zdrojový program jako syntakticky chybný. Základní model syntaktické analýzy představují *zásobníkové automaty*, které nejsou předmětem této práce a jejich formální definici lze nálézt v knize [9] na straně 113.

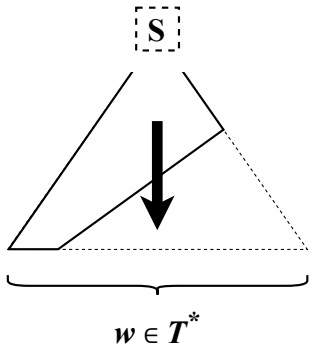
Pro zdrojový program, který je po syntaktické stránce vyhovující, je během syntaktické analýzy sestrojen derivační strom, který je předán následující částí překladače (sématické analýze) k dalšímu zpracování. Derivační strom musí být vybudován na základě znalosti výchozího stavu, mezi které patří kořen stromu (počáteční symbol S) a sekvence listových uzlů (řetězce terminálů w). Výchozí stav konstrukce stromu je ilustrován na obrázku 3.2.



Obrázek 3.2: Počáteční stav konstrukce derivačního stromu při syntaktické analýze (viz [7]).

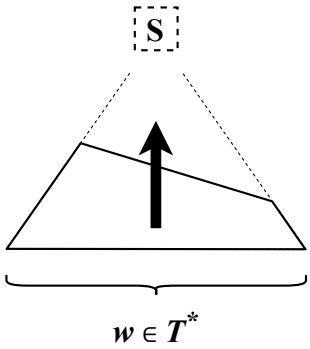
Existují dvě metody, jak na základě znalosti počátečního stavu vybudovat derivační strom:

1. Syntaktická analýza *shora dolů* – buduje derivační strom od kořene S a postupnými derivačními kroky dojde až k řetězci terminálů w, jak je ilustrováno na obrázku 3.3.



Obrázek 3.3: Znázornění postupu budování derivačního stromu syntaktické analýzy *shora dolů* (viz [7]).

2. Syntaktická analýza *zdola nahoru* – buduje derivační strom od řetězce terminálů w a postupnými redukčními kroky dojde až ke kořeni S, jak je ilustrováno na obrázku 3.4.



Obrázek 3.4: Znázornění postupu budování derivačního stromu syntaktické analýzy *zdola nahoru* (viz [7]).

3.3 Syntaktická analýza shora dolů

Při syntaktické analýze *shora dolů* je konstruována *levá derivace* věty jazyka $L(G)$, kde G je bezkontextová gramatika a vstupní řetězec tokenů na vstupu je skenován *zleva doprava* po *jednom symbolu*. Často bývá založená na *LL gramatikách*. Problém u syntaktické analýzy shora dolů nástává při výběru pravidla, které bude použito v následujícím derivačním kroku. Syntaktický analyzátor LL totiž musí být schopen na základě nejlevějšího neterminálu aktuální větné formy a aktuálního tokenu na vstupu jednoznačně určit, jaké pravidlo bude na neterminál aplikováno. Schopnost jednoznačně rozhodnout poskytuje *LL tabulka* (viz [8]).

LL syntaktická analýza

Syntaktické analyzátory založené na LL gramatikách si zakládají právě na vlastnostech metody *shora dolů*, které zmiňuje i samotná zkratka LL. První výskyt zkratky „L“ značí skenování tokenů na vstupu zleva doprava a druhé pak konstrukci nejlevější derivace. Specifikací LL gramatik může být počet n tokenů na vstupu, kde $n \geq 1$, který bere syntaktický analyzátor v potaz v každém jednom kroku procesu analýzy. Tato specifikace bývá značena jako LL(n). Text se zabývá pouze gramatikami typu LL(1), a proto bude dále používáno jen zkráceně jako LL.

Definice 3.1 (viz [8])**.** Gramatika G je LL jen tehdy, kdy pro pravidla $p_1 : A \to u$, $p_2 : A \to v \in P$, platí, že $u \neq v$ a průnikem množin $Predict(p_1)$ a $Predict(p_2)$ je množina prázdná, formálně zapsáno jako $Predict(p_1) \cap Predict(p_2) = \emptyset$.

Parametr nulovatelnosti

Pro každý symbol $X \in (N \cup T)^*$ nechť je stanoven parametr nulovatelnosti $Nullable(X)$, který značí, zdali derivací X je možné získat prázdný řetězec ϵ, tedy $X \Rightarrow^* \epsilon$. Pokud to možné je, tak $Nullable(X) = true$, jinak $Nullable(X) = false$. Způsob, kterým lze

určit parametr nulovatelnosti pro X, je demonstrován algoritmem 3.1 (viz [2] kapitola Syntaktická analýza shora dolů).

Algoritmus 3.1: $Nullable(X)$

Vstup: Gramatika G
Výstup: $Nullable(X)$ pro každé $X \in (N \cup T)$

pro každé $X \in T$**:**
    $Nullable(X) = false$

pro každé $X \in N$**:**
    if $X \to \epsilon$ **:**
        $Nullable(X) = true$
    else :
        $Nullable(X) = false$

while je stále možné měnit nějaký parametr $Nullable(X)$ **:**
    if $X \to U_1 \dots U_n$ **and** $Nullable(U_i) == true$ pro všechna $i = 1, \dots, n$ **:**
        $Nullable(X) = true$

Tento parametr lze mimo samostatné symboly určit také pro celé řetězce symbolů $X_1 \dots X_n$ viz algoritmus 3.2. $Nullable(X_1 \dots X_n)$ bude využito pro určení množin v následujících sekcích.

Algoritmus 3.2: $Nullable(X_1 \dots X_n)$

Vstup: Gramatika G, $Nullable(X)$ pro všechna $X \in (N \cup T)$
Výstup: $Nullable(X_1 \dots X_n)$

if $Nullable(X_i) == true$ pro každé $i = 1, \dots, n$ **:**
    $Nullable(X_1 \dots X_n) = true$
else :
    $Nullable(X_1 \dots X_n) = false$

Množiny First a Follow

Množiny $First()$ a $Follow()$ korespondující s gramatikou G hrají klíčovou roli při výběru pravidla, které bude aplikováno na větnou formu podle aktuálního tokenu na vstupu. Na jejich základě je totiž sestrojena množina $Predict()$.

Definice 3.2 (viz [4] a [2] kapitola Syntaktická analýza shora dolů)**.** Mějme gramatiku G. Nechť $First(x)$, kde $x \in (N \cup T)^*$, je množina všech terminálů, kterými může začínat řetězec derivovaný z x. Formálně zápsáno jako

$$First(x) = \{a : a \in T, x \Rightarrow ay, \text{ kde } y \in (N \cup T)^*\}.$$

Způsob, jak získat množinu $First(X)$ pro všechny symboly $X \in (N \cup T)$ gramatiky G, je popsán níže algoritmem 3.3 (viz [2] kapitola Syntaktická analýza shora dolů).

Algoritmus 3.3: $First(X)$

Vstup: Gramatika G
Výstup: $First(X)$ pro každé $X \in (N \cup T)$

1: pro každé $X \in T$**:**
2: $First(X) = X$
3:
4: pro každé $X \in N$**:**
5: $First(X) = \emptyset$
6:
7: **while** je stále možné měnit nějakou množinu $First(X)$ **:**
8: **if** $X \to U_1 \dots U_{k-1} U_k \dots U_n \in P$ **:**
9: **if** $Nullable(U_i) == true$ pro každé $i = 1, \dots, k-1,$ kde $k \leq n$ **:**
10: $First(X) = First(X) \cup First(U_k)$
11: **else :**
12: $First(X) = First(X) \cup First(U_1)$

Stejně jako tomu je u parametru nulovatelnosti, je možné určit množinu $First()$ pro celé řetězce symbolů $X_1 \dots X_n$ viz algoritmus 3.4. $First(X_1 \dots X_n)$ bude využito pro určení množin $Follow()$ a $Predict()$.

Algoritmus 3.4: $First(X_1 \dots X_n)$

Vstup: Gramatika G, $Nullable(X)$ a $First(X)$ pro všechna $X \in (N \cup T)$
Výstup: $First(X_1 \dots X_n)$

// $First(X_1 \dots X_{k-1} X_k \dots X_n)$ `je jiný zápis pro` $First(X_1 \dots X_n)$

1: **if** $Nullable(X_i) == true$ pro každé $i = 1, \dots, k-1,$ kde $k \leq n$ **:**
2: $First(X_1 \dots X_n) = First(X_k)$
3: **else :**
4: $First(X_1 \dots X_n) = First(X_1)$

Definice 3.3 (viz [4] a [2] kapitola Syntaktická analýza shora dolů)**.** Mějme gramatiku G. Nechť $Follow(A)$, kde $A \in N$, je množina všech terminálů $a \in T$, které mohou následovat bezprostředně vpravo od A ve větné formě vygenerované pomocí G. Formálně zapsáno jako

$$Follow(A) = \{a \colon a \in T, S \Rightarrow^* xAay, \text{ kde } x, y \in (N \cup T)^*\} \cup \{\$ \colon S \Rightarrow^* xA, \text{ kde } x \in (N \cup T)^*\}.$$

Algoritmus pro získání množiny $Follow(A)$, kde $A \in N$, která poskytne zbývající informace potřebné k vytvoření množin $Predict()$, viz algoritmus 3.5.

Algoritmus 3.5: $Follow(A)$

Vstup: Gramatika $G = (N, T, P, S)$
Výstup: $Follow(A)$ pro každé $A \in N$

1: $Follow(S) = \{\$\}$
2: **while** je stále možné měnit nějakou množinu $Follow(A)$ **:**
3: **if** $A \to xBy \in P$ **:**
4: **if** $y \neq \epsilon$ **:**
5: $Follow(B) = Follow(B) \cup First(y)$
6: **if** $Nullable(y) == true$ **:**
7: $Follow(B) = Follow(B) \cup Follow(A)$

Množina Predict

Definice 3.4. Mějme gramatiku G. Nechť $Predict(p\colon A \to x)$, kde $A \in N$ a $x \in (N \cup T)^*$, je množina všech terminálů, které se mohou vyskytnout na první pozici řetězce vyplývajícího z derivace, jíž prvním krokem byla aplikace pravidla p (viz [8]).

Množina $Predict()$ je zásadní pro určení pravidla použitého v následujícím kroku syntaktické analýzy právě díky tomu, že díky ní lze sestavit *LL tabulku*. Tuto množinu je možné definovat na základě *parametrů nulovatelnosti* a *množin* $First()$ *a* $Follow()$ pro každé pravidlo, viz algoritmus 3.6 (viz [2] kapitola Syntaktická analýza shora dolů).

Algoritmus 3.6: $Predict(p\colon A \to x)$

Vstup: Gramatika G
Výstup: $Predict(p\colon A \to x)$ pro každé $p \in P$

1: **if** $Nullable(x) == true$ **:**
2: $Predict(p\colon A \to x) = First(x) \cup Follow(A)$
3: **else :**
4: $Predict(p\colon A \to x) = First(x)$

Prediktivní syntaktická analýza založená na LL tabulce

Jedním ze způsobů, jak implementovat syntaktický analyzátor vycházející z gramatiky G je *prediktivní* syntaktická analýza založená na *LL tabulce*. *LL tabulka* je sestrojena na základě množiny $Predict()$. Každý neterminál $B \in N$ je reprezentován právě jedním řádkem LL tabulky a každý terminál $a \in T$ právě jedním sloupcem. Navíc je přidán sloupec pro *zakončovač řetězce* \$, který symbolizuje konec větné formy uložené na zásobníku. Hodnotou buněk LL tabulky jsou poté samotná pravidla $p : A \to x \in P$. Pro dvojici B a a existuje buňka s hodnotou p, pokud $B = A$ a $a \in Predict(p)$. V případě, že pro tuto dvojici v tabulce neexistuje žádné pravidlo, vyskytuje se v tabulce prázdná buňka, která značí syntaktickou chybu. K aplikaci pravidla p dochází tehdy, kdy B je neterminál na vrcholu zásobníku a a aktuální symbol na vstupu.

Důležité je nezapomenout, že pokud je G opravdu LL, nesmí být v jedné buňce více než jedno pravidlo (viz definice 3.1). V opačném případě by byl analyzátor nederministický, jelikož by nebylo zřejmé, jaké pravidlo v daném případě aplikovat. Významným pozitivem tohoto způsobu syntaktické analýzy shora dolů je jednoduchost aplikace změn v gramatice

G. Změna v G totiž znamená pouze adekvátní úpravu LL tabulky, zatímco naprogramovaná metoda řídící syntaktickou analýzu je ponechána beze změny (viz [8]).

Příklad 3.1. Mějme gramatiku $G_{def} = (\{S, DEF, ASSIGN, TYPE\}, \{\textbf{id}, \textbf{;}, \textbf{=}, \textbf{int}, \textbf{float}, \textbf{expression}\}, P, S)$, kde

$$\begin{aligned} P \quad = \{ \quad & 1\colon S \to DEF\ \textbf{\$}, \\ & 2\colon DEF \to TYPE\ \textbf{id}\ ASSIGN\ \textbf{;}, \\ & 3\colon ASSIGN \to \textbf{=}\ \textbf{expression}, \\ & 4\colon ASSIGN \to \epsilon, \\ & 5\colon TYPE \to \textbf{int}, \\ & 6\colon TYPE \to \textbf{float} \quad \}, \end{aligned}$$

již reprezentuje zjednodušenou konstrukci definice proměnné výchozího programovacího jazyka. Terminál **expression** zde pro jednoduchost zastupuje neterminál, kterým by mohla být, vytvořením dalších pravidel, specifikována syntaxe matematického výrazu.

Na základě algoritmů 3.1, 3.2, 3.3, 3.4 a 3.5 je sestrojena tabulka 3.1 obsahující *parametr nulovatelnosti* a množiny $First(A)$ a $Follow(A)$ pro všechny neterminály $A \in N$. Zmiňované vlastnosti terminálů jsou více méně jednoznačné, proto nejsou v tabulce uvedeny.

Neterminál $\boldsymbol{A}$	**Nulovatelnost**	$\boldsymbol{First(A)}$	$\boldsymbol{Follow(A)}$
S	×	**int**, **float**	
DEF	×	**int**, **float**	**$**
$ASSIGN$	✓	**=**	**;**
$TYPE$	×	**int**, **float**	**id**

Tabulka 3.1: Tabulka obsahující *parametr nulovatelnosti* a množiny $First(A)$ a $Follow(A)$ pro všechny neterminály $A \in N$. Symbol „✓“ reprezentuje hodnotu *true* a „×“ hodnotu $false$.

Následně z informací z tabulky 3.1 je možné určit množinu $Predict(p)$ pro všechna $p \in P$ pomocí algoritmu 3.6, viz tabulka 3.2.

Pravidlo $\boldsymbol{p}$	$\boldsymbol{Predict(p)}$
$1\colon S \to DEF$ **$**	**int**, **float**
$2\colon DEF \to TYPE$ **id** $ASSIGN$ **;**	**int**, **float**
$3\colon ASSIGN \to$ **=** **expression**	**=**
$4\colon ASSIGN \to \epsilon$	**;**
$5\colon TYPE \to$ **int**	**int**
$6\colon TYPE \to$ **float**	**float**

Tabulka 3.2: Množina $Predict(p)$ pro všechna $p \in P$ gramatiky G_{def}.

V poslední fázi, kdy už jsou známé jednotlivé množiny $Predict()$, bude sestrojena *LL tabulka*. Pravidlo $p\colon A \to x \in P$ bude zaznamenáno v tabulce na řádku A ve sloupcích všech neterminálů a, kde $a \in Predict(p)$, viz tabulka 3.3.

	id	;	=	int	float	expression	$
S				1	1		
DEF				2	2		
$ASSIGN$		4	3				
$TYPE$				5	6		

Tabulka 3.3: LL tabulka gramatiky G_{def}.

3.4 Syntaktická analýza zdola nahoru

Stejně jako u syntaktické analýzy shora dolů, i u té *zdola nahoru* je vstupní řetězec tokenů na vstupu skenován *zleva doprava* po jednom symbolu. Zásadním rozdílem je tedy konstrukce *pravé derivace* věty jazyka $L(G)$, kde G je bezkontextová gramatika. Ke konstrukci derivačního stromu dochází zpětně, tedy od věty na vstupu k počátečnímu symbolu S. Každý krok syntaktické analýzy je reprezentován *posunem* nebo *redukcí*. Posunem je myšleno přesunutí aktuálního terminálu na vstupu na vrchol zásobníku. Redukce potom znamená nahrazení podřetězce x větné formy z vrcholu zásobníku, který odpovídá pravé straně pravidla $p \in P$, za neterminál z levé strany pravidla p. Problém u tohoto typu syntaktické analýzy činí nalezení vhodného podřetězce větné formy, nad kterým bude redukce provedena.

V této podkapitole budou přiblíženy dvě metody syntaktické analýzy zdola nahoru. První je *precedenční syntaktická analýza*, jež je vhodná pro zpracování matematických výrazů, kde operátory a jejich priority prakticky řídí syntaktickou analýzu. Druhou metodou je *LR syntaktický analyzátor* založený na LR gramatikách, z nichž lze sestrojit *LR tabulku*, pomocí které je syntaktická analýza řízena. LR představují nejsilnější deterministicky pracující syntaktické analyzátory, proto lze často nalézt jejich využití v praxi (viz [8]).

Precedenční syntaktická analýza

Precedenční syntaktický analyzátor, v literatuře nazývaný také jako syntaktický analyzátor *priority operátorů*, je vhodný a využívaný především pro zpracování matematických výrazů, jak již bylo zmíněno výše. Předpokladem pro tvorbu takového analyzátoru je stanovená priorita pro každé dva operátory a asociativita stanovená pro každý operátor.

Konvence 3.1. Nechť v podkapitole precedenční syntaktické analýzy figuruje gramatika G, kde $aTop \in T$ je nejvyšší terminál umístěný na zásobníku, $xTop \in (N \cup T)^+$ je podřetězec větné formy gramatiky G na vrcholu zásobníku a $iSymb \in T$ je aktuální symbol na vstupu.

Chod syntaktického analyzátoru je řízen *precedenční tabulkou*. Každý terminál $t \in T$ je reprezentován jedním řádkem a jedním sloupcem v tabulce. Nechť hodnota buňky v tabulce na řádku $aTop$ a ve sloupci $iSymb$ určí následující operaci, rozpozná syntaktickou chybu nebo vyhodnotí syntaktickou analýzu jako úspěšnou (viz [9]).

Operace *redukce* a *posunu*, které definujeme u precedenční syntaktické analýzy, modifikují vrchol zásobníku následujícím způsobem:

- *Redukce* – existuje-li pravidlo $p : A \to x \in P$, kde $x = xTop$, je vrchol zásobníku $xTop$ nahrazen neterminálem A.
- *Posun* – přesunutí vstupního symbolu $iSymb$ na vrchol zásobníku a následné přečtení náledujícího symbolu ze vstupu.

Konstrukce precedenční tabulky

Konstrukce precedenční tabulky je spíše praktická záležitost, ke které je zapotřebí základních matematických pravidel *priority* a *asociativity operátorů* a selského rozumu.

Konvence 3.2. Mějme hodnoty $\{<, >, =, OK\}$, kterých může nabýt buňka precedenční tabulky, kde $<$ a $=$ reprezentují operaci posunu (rozdíl mezi $<$ a $=$ bude specifikován v kapitole 6 konrétně v algoritmu 6.2), $>$ reprezentuje operaci redukce a OK představuje úspěch syntaktické analýzy. Prázdná buňka značí syntaktickou chybu.

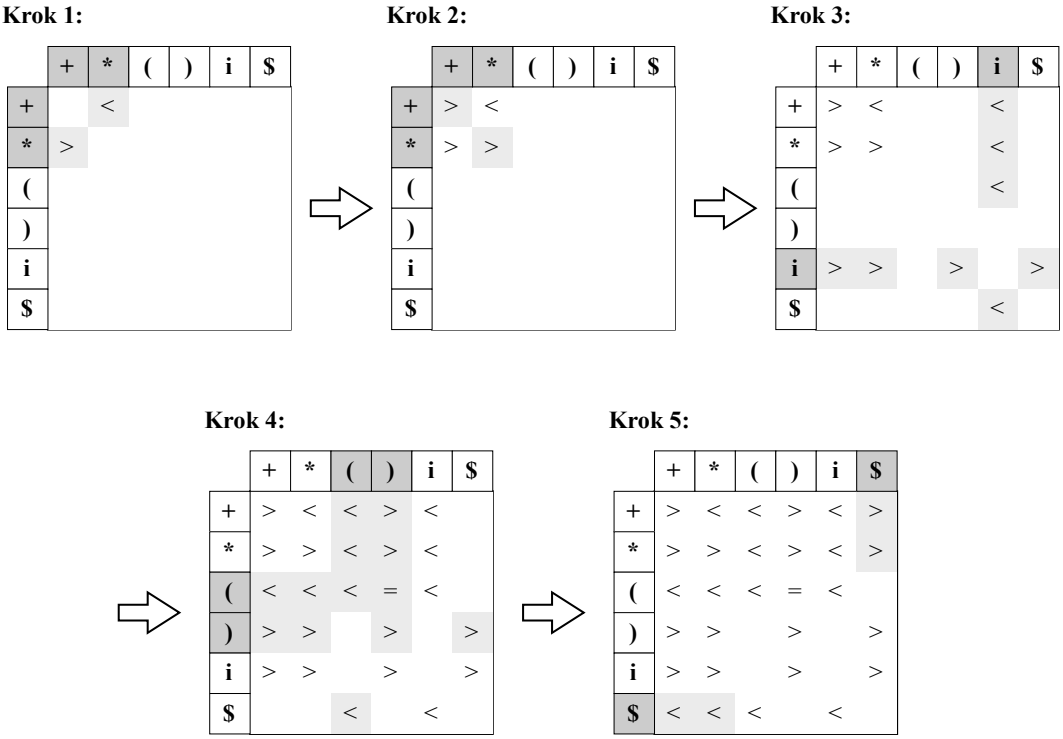
Tabulku lze zkonstruovat provedením následujících kroků 1 až 5 (viz [3] kapitola Bottom-Up Parsing).

1. *Precedence operátorů:*
   - Pokud $op_1, op_2 \in T$ jsou operátory, kde op_1 má větší matematickou prioritu než op_2, tak $op_1 > op_2$ a $op_2 < op_1$.
2. *Asociativita:*
   - Pokud $op_1, op_2 \in T$ jsou *levě asociativní* operátory se stejnou prioritou, tak $op_1 > op_2$ a $op_2 > op_1$.
   - Pokud $op_1, op_2 \in T$ jsou *pravě asociativní* operátory se stejnou prioritou, tak $op_1 < op_2$ a $op_2 < op_1$.
3. *Identifikátory:*
   - Pokud $a \in T$ a id je identifikátor, kde a může legálně přímo předcházet id, tak $a < id$.
   - Pokud $a \in T$ a id je identifikátor, kde a může legálně přímo následovat za id, tak $id > a$.
4. *Závorky:*
   - Pro pár závorek platí $(=)$.
   - Pokud $a \in T - \{), \$\}$, tak $(< a$.
   - Pokud $a \in T - \{(, \$\}$, tak $a >)$.
   - Pokud $a \in T$, kde a může legálně přímo předcházet (, tak $a <$ (.
   - Pokud $a \in T$, kde a může legálně přímo následovat za), tak $) > a$.
5. *Ukončovač řetězce:*
   - Pokud $op \in T$ je operátor, tak $\$ < op$ a $op > \$$.

Příklad 3.2. Mějme gramatiku $G_{exp} = (\{E\}, \{+, *, (,), \mathbf{i}\}, P, E)$, kde

$$\begin{aligned} P \quad = \{ \quad & 1\colon E \to E + E, \\ & 2\colon E \to E * E, \\ & 3\colon E \to (\ E \ ) \ , \\ & 4\colon E \to \ \mathbf{i} \quad \}, \end{aligned}$$

již představuje konstrukci výrazu podporující operátory „+“ a „−“. Obrázek 3.5 demonstruje postup sestavení precedenční tabulky G_{exp} dle jednotlivých kroků představených na straně 21.



Obrázek 3.5: Postup sestavení precedenční tabulky pro gramatiku G_{exp}.

LR syntaktická analýza

Syntaktická analýza LR, podobně jako analýza LL, zmiňuje klíčové vlastnosti analyzátorů založených na principu *zdola nahoru* už ve svém názvu, kde „L“ symbolizuje skenování tokenů na vstupu zleva doprava a „R“ poté konstrukci nejpravější derivace. Jako dříve představené metody syntaktické analýzy prediktivní LL a precedenční, je i LR řízena tabulkou, která je sestrojena pro gramatiku G. Gramatika, pro kterou není možné LR tabulku vytvořit, není LR gramatikou (viz [8]).

Pro každou LR gramatiku je sestavena množina stavů Q definujících v jaké fázi analýzy se analyzátor nachází. Pomocí aktuálního stavu a tokenu na vstupu je poté možné jednoznačně určit operaci, jež bude provedena, stav, do kterého analyzátor následně přejde či vyhodnotit úspěch i neúspěch syntaktické analýzy. Operace typické pro skupinu analyzátorů zdola nahoru upravující vrchol zásobníku jsou *redukce* a *posun*.

Třída gramatik, které je možné pomocí LR metody analyzovat je vlastní nadmnožinou tříd gramatik, které je možné analyzovat pomocí metody precedenční a LL. Nevýhodou je potom náročnost ručního sestrojení LR tabulky pro větší gramatiky. Existují ovšem programy, jež jsou schopné tabulku vygenerovat na základě znalosti gramatiky (viz [4]).

Stejně jako u LL gramatik může být pro LR specifikován počet n tokenů na vstupu, kde $n \geq 1$, který bere syntaktický analyzátor v potaz v každém jednom kroku procesu analýzy, značeno jako LR(n). Tato práce se zabývá pouze gramatikami typu LR(1), dále tedy pouze zkráceně jako LR.

LR tabulka

Jak již bylo zmíněno dříve, tabulka založená na LR gramatice G je pro LR syntaktickou analýzu klíčová. Je tedy třeba představit, jak vlastně vypadá a z jakých prvků se skládá. LR tabulka se skládá ze *dvou částí* a to z *akční* (*Action*) a *přechodové* (*Goto*) části. Obě části tabulky mají řádek reprezentován stavem analyzátoru.

Akční část má poté sloupce reprezentované terminály a zakončovačem řetězce \$. Sloupec tedy existuje pro každý symbol $a \in (T \cup \$)$. Hodnoty, kterých mohou buňky akční části nabývat jsou:

- Hodnota sq, kde q je číslo nasledujícího stavu a s symbolizuje operaci *posunu*, tedy umístění aktuálního symbolu na vstupu na zásobník.
- Hodnota rp, kde $p : A \to x \in P$ a r symbolizuje operaci *redukce*, tedy nahrazení řetězce x na vrcholu zásobníku za neterminál A. Po redukci vždy dojde k určení následujícího stavu pomocí přechodové části LR tabulky.
- Hodnota OK představuje přijmutí řetězce na vstupu a ukončení syntaktické analýzy.
- Prázdné pole představuje syntaktickou chybu.

Sloupce *přechodové části* jsou reprezentovány neterminály tak, že pro každý symbol $A \in N$ existuje sloupec. Buňka přechodové části může nabýt pouze hodnoty q, kde q je číslo následujícího stavu analýzy (viz [4]). Prázdná pole v této části nemají žádný význam, jelikož chyba je vždy odhalena už v akční části a zde k ní tedy nikdy nedojde.

Konvence 3.3. Mějme LR gramatiku G. Nechť zápis $Table[q, X]$, kde $Table$ je LR tabulka nebo její část, q je stav LR analyzátoru a $X \in (N \cup T)$, značí položku tabulky $Table$ na řádku q a ve sloupci X.

Příklad 3.3. Mějme LR gramatiku $G_{tab} = (\{A_1, \ldots, A_i, \ldots, A_l\}, \{a_1, \ldots, a_j, \ldots, A_m\}, P, S')$ a množinu stavů $Q = \{q_1, \ldots, q_k, \ldots, q_n\}$, kde $l, m, n \geq 1$ a $1 \leq i \leq l, 1 \leq j \leq m, 1 \leq k \leq n$. Tabulka 3.4 znázorňuje, jak by vypadala LR tabulka pro gramatiku G_{tab} a stavy Q.

	Action	**Goto**
	$a_1 \ \ldots \ a_j \ \ldots \ a_m$	$A_1 \ \ldots \ A_i \ \ldots \ A_l$
q_1 $\vdots$ q_k $\vdots$ q_n	$Action[q_k, a_j]$	$Goto[q_k, A_i]$

Tabulka 3.4: Znázornění LR tabulky pro gramatiku G_{tab} z příkladu 3.3.

Konstrukce SLR tabulky

Tato sekce představí konstrukci SLR tabulky. SLR je oproti LR o něco slabší, ale výhodou je jednodušší tvorba tabulky a pro potřeby této práce postačí. První pojem, který v rámci konstrukce SLR tabulky bude zaveden, je *rozšířená gramatika*. Existuje-li gramatika G, pro kterou má být sestrojena SLR tabulka, její rozšířená verze G' obsahuje nový počáteční

symbol S' a nové tzv. *nulté pravidlo* $0 : S' \to S$, kde S je původní počáteční symbol G. Rozšíření gramatiky slouží k rozpoznání úspěšné syntaktické analýzy. Je-li provedena redukce nultým pravidlem, znamená to, že řetězec terminálů na vstupu byl přijat a analýza končí (viz [4]).

Položka

Na základě *položek* jsou vytvořeny stavy SLR syntaktického analyzátoru, jež jsou alfou a omegou procesu SLR analýzy. V každém kroku analýzy obsahuje zásobník prefix podřetězce větné formy, který bude zredukován na neterminál při následující operaci redukce pravidlem $p\colon A \to x \in P$. Nechť je tento prefix označen jako y. Snaha analyzátoru je rozšířit y na zásobníku o řetězec z tak, že $yz = x$. Pro vyjádření, že se y již na zásobníku vyskytuje, je zaveden právě pojem *položka*, zapisováno jako $A \to y \bullet z$. Položky jsou zavedeny pro všechna pravila $p \in P$. Později bude ukázáno, že stavy analyzátoru reprezentují množiny položek (viz [9]).

Příklad 3.4. Mějme rozšířenou gramatiku $G_{slr} = (\{S', S, A\}, \{\mathbf{i}, +, (,)\}, P, S')$, kde

$$\begin{aligned} P \quad = \{ \quad & 0\colon S' \to S, \\ & 1\colon S \to S \ + \ A, \\ & 2\colon S \to A, \\ & 3\colon A \to \ (\ S\ ), \\ & 4\colon A \to \ \mathbf{i} \quad \}, \end{aligned}$$

již reprezentuje jednoduchou konstrukci matematického výrazu s operátorem sčítání. G_{slr} poslouží i pro následující příklady demostrující konstrukci SLR tabulky.

Pro pravidlo $1\colon S \to S + A$ budou zavedeny čtyři položky

$$\begin{aligned} S &\to \bullet\ S + A \\ S &\to S \bullet + A \\ S &\to S + \bullet\ A \\ S &\to S + A \bullet \end{aligned}$$

Stejným způsobem jsou položky vytvořeny i pro pravidla $0, 2, 3$ i $4 \in P$. Pro představu, položka $S \to S \bullet + A$ tedy značí, že řetězec, který je možné derivovat ze symbolu S, již byl ze vstupu načten, zatímco řetězec derivovaný z $+A$ je na vstupu teprve očekáván.

Uzávěr položek

Nechť I je množina položek pro gramatiku G', kde $Closure(I)$ neboli *uzávěr položek* je množina položek sestavená pomocí algoritmu 3.7 (viz [4]).

Algoritmus 3.7: $Closure(I)$

Vstup: Gramatika G, množina položek I
Výstup: Množina $Closure(I)$

1: $Closure(I) = I$
2: **while** je stále možné přidávat nové položky do $Closure(I)$ **:**
3: **foreach** $A \to x \bullet By \in Closure(I)$ **:**
4: **foreach** $B \to z \in P$ **:**
5: **if** $B \to \bullet z \notin Closure(I)$ **:**
6: $Closure(I) = Closure(I) \cup B \to \bullet z$

Příklad 3.5. Uvažujme rozšířenou gramatiku G_{slr} z příkladu 3.4. Uzávěr položek $Closure(I)$, kde $I = \{A \to (\bullet S)\}$, sestrojený podle algoritmu 3.7, bude vypadat následovně.

$$\begin{aligned} Closure(I) \quad = \quad & \{\ A \to (\bullet S), \\ & S \to \bullet S{+}A, \\ & S \to \bullet A, \\ & A \to \bullet(S), \\ & A \to \bullet\mathbf{i}\ \} \end{aligned}$$

Funkce *Goto*()

Funkce $Goto(I, X)$, kde I je množina položek pro gramatiku G' a $X \in (N \cup T)$, je definována jako sjednocení uzávěrů všech položek $Closure(A \to xB \bullet y)$, přičemž položka $A \to x \bullet By \in I$. Prakticky $Goto(I, X)$ představuje přechod ze stavu reprezentovaného množinou položek I se symbolem X na vstupu (viz [4]).

Příklad 3.6. Předpokládejme množinu položek $I = \{A \to (\bullet S), S \to \bullet S{+}A, S \to \bullet A, A \to \bullet(S), A \to \bullet\mathbf{i}\ \}$ zastupující stav rozšířené gramatiky G_{slr} z příkladu 3.4.

$$\begin{aligned} Goto(I, S) \quad &= \quad Closure(A \to (S\bullet)) \cup Closure(S \to S\bullet{+}A) = \\ &= \quad \{\ A \to (S\bullet), \\ &\quad\ \ S \to S\bullet{+}A\ \} \end{aligned}$$

$$\begin{aligned} Goto(I, (\ ) \quad &= \quad Closure(A \to (\bullet S)) = \\ &= \quad \{\ A \to (\bullet S), \\ &\quad\ \ S \to \bullet S{+}A, \\ &\quad\ \ S \to \bullet A, \\ &\quad\ \ A \to \bullet(S), \\ &\quad\ \ A \to \bullet\mathbf{i}\ \} \end{aligned}$$

Množina Θ_G

Množina Θ_G pro gramatiku G je soubor množin položek definovaný algoritmem 3.8. Právě prvky množiny Θ_G představují jednotlivé stavy SLR syntaktického analyzátoru (viz [4]).

Algoritmus 3.8: Množina Θ_G

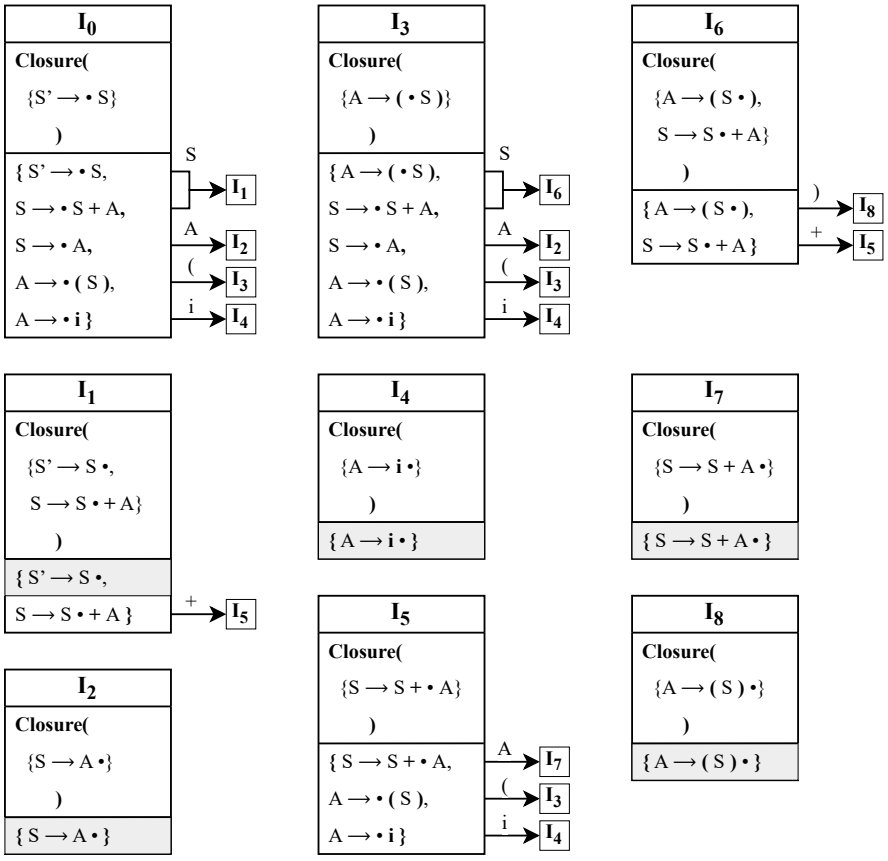
Vstup: Rozšířená gramatika G
Výstup: Množina Θ_G

1: $\Theta_G = \{Closure(\{S' \to \bullet S\})\}$
2: **while** je stále možné přidávat nové množiny položek do Θ_G **:**
3: **foreach** $I \in \Theta_g$ **:**
4: **foreach** $X \in (N \cup T)$ **:**
5: **if** $Goto(I, X) \neq \emptyset$ **and** $Goto(I, X) \notin \Theta_G$ **:**
6: $\Theta_G = \Theta_G \cup Goto(I, X)$

Příklad 3.7. Využijme pro demonstraci gramatiku G_{slr} definovanou v příkladě 3.4. Množina $\Theta_{G_{slr}}$ vytvořená pomocí algoritmu 3.8 má následující podobu.

$$
\begin{aligned}
\Theta_{G_{slr}} = \{ \quad & I_0 : \{S' \to \bullet S, S \to \bullet S{+}A, S \to \bullet A, A \to \bullet(S), A \to \bullet\mathbf{i}\}, \\
& I_1 : \{S' \to S\bullet, S \to S\bullet{+}A\}, \\
& I_2 : \{S \to A\bullet\}, \\
& I_3 : \{A \to (\bullet S), S \to \bullet S{+}A, S \to \bullet A, A \to \bullet(S), A \to \bullet\mathbf{i}\}, \\
& I_4 : \{A \to \mathbf{i}\bullet\}, \\
& I_5 : \{S \to S{+}\bullet A, A \to \bullet(S), A \to \bullet\mathbf{i}\}, \\
& I_6 : \{A \to (S\bullet), S \to S\bullet{+}A\}, \\
& I_7 : \{S \to S{+}A\bullet\}, \\
& I_8 : \{A \to (S)\bullet\} \quad \}
\end{aligned}
$$

Podrobněji je postup vytvoření $\Theta_{G_{slr}}$ znázorněn na obrázku 3.6. Zde je možné vidět přechody mezi jednotlivými stavy I_0 až I_8 a konkrétní symboly, se kterými k přechodům dochází. U každého stavu je také označeno, jakých položek je stav uzávěrem. Položky ve tvaru $A \to x\bullet$ (na schématu podbarveny šedou barvou) nevytváří přechod do jiného stavu a představují operaci redukce. Stav, který obsahuje položku $S' \to \bullet S$ nultého pravidla, je značen číslem 0 a je nazýván jako stav *počáteční* (viz [3] kapitola Bottom-Up Parsing).



Obrázek 3.6: Schéma přechodů mezi jednotlivými stavy gramatiky G_{slr}.

Konstrukce SLR tabulky

V poslední fázi přichází řada na samotný proces konstrukce SLR tabulky. Klíčovou znalostí pro konstrukci tabulky je množina Θ_G, jejíž prvky jsou postupně procházeny položku po položce a na níž je celý proces konstrukce založen. Za využití funkcí $Goto()$ a $Folllow()$ (množina $Follow()$ vysvělena na straně 17) je poté možné pro SLR gramatiku jednoznačně určit hodnoty buňek jak v akční, tak v přechodové části SLR tabulky. Konkrétní způsob, jakým tabulku vytvořit, prezentuje algoritmus 3.9 (viz [3] kapitola Bottom-Up Parsing).

Algoritmus 3.9: KONSTRUKCE SLR TABULKY

Vstup: Rozšířená gramatika G, Θ_G, $Follow(A)$ pro každé $A \in N$
Výstup: SLR tabulka ($Action$ – akční část, $Goto$ – přechodová část)

1: **foreach** $I_i \in \Theta_g$ **:**
2: **foreach** $polozka \in I_i$ **:**
3: **if** $polozka == A \to xa\bullet y$ **and** $a \in T$ **:**
4: $j = Goto(I_i, a)$
5: $Action[I_i, a] = sj$
6: **else if** $polozka == A \to xB\bullet y$ **and** $B \in N$ **:**
7: $Goto[I_i, B] = Goto(I_i, B)$
8: **else if** $polozka == S' \to S\bullet$ **:**
9: $Action[I_i, \$] = \text{OK}$
10: **else if** $polozka == A \to x\bullet$ **and** $A \neq S'$ **:**
11: **foreach** $a \in Folllow(A)$ **:**
12: $Action[I_i, a] = rp$, kde p je číslo pravidla $A \to x$

Příklad 3.8. Mějme rozšířenou gramatiku G_{slr} definovanou v příkladě 3.4 a množinu $\Theta_{G_{slr}}$ z příkladu 3.7, na jejichž základě je, pomocí algoritmu 3.9, tabulka SLR sestrojena, viz tabulka 3.5. Dosavadní značení stavů jako I_i, kde i se rovná číslu stavu, je v tabulce zjednodušeno a zapisováno pouze číslem i (například stav I_0 zapsán jako 0).

	Action					**Goto**		
	+	(	)	**i**	**$**	S'	S	A
0		s3		s4			1	2
1	s5				OK			
2	r2		r2		r2			
3		s3		s4			6	2
4	r4		r4		r4			
5		s3		s4				7
6	s5		s8					
7	r1		r1		r1			
8	r3		r3		r3			

Tabulka 3.5: SLR tabulka pro gramatiku G_{slr} z příkladu 3.4.

Kapitola 4

Kooperačně distribuované gramatické systémy

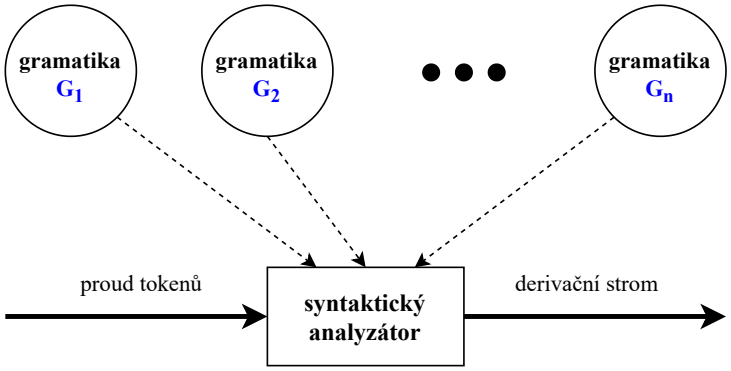
Tato kapitola představí pojem *gramatický systém*, respektive jeho základní definici, motiv vzniku a klíčové vlastnosti. V druhé části budou definovány *kooperačně distribuované* (dále značeny jako *CD*) gramatické systémy a *derivační režimy*, na základě kterých CD systémy pracují. V poslední řadě budou zavedeny pojmy *hybridní* CD gramatické systémy a gramatické systémy *s vnitřním řízením*, ze kterých vychází i samotný návrh gramatického systému pro tuto práci (viz 5). Pojmy a terminologie z této kapitoly jsou převzaty z [1] a [11].

4.1 Gramatické systémy

Tato práce doposud představila formální jazyky generované pouze jednou gramatikou. *Gramatický systém* je poté množina spolupracujících gramatik, které dohromady generují právě *jeden jazyk*. Gramatika, jež je součástí gramatického systému, je nazývána také jako *komponenta*. Důvod zavedení pojmu „gramatický systém“ v teoretické informatice může být překvapivý. V praxi při vývoji překladačů se běžně pro syntaktickou analýzu zdrojového programu využívalo více gramatik, pro dosažení pozitivních vlastností, které právě gramatické systémy popisují. Motivem tedy byl mimo jiné fakt, že praxe v této oblasti předběhla teorii. Obecné schéma myšlenky syntaktického analyzátoru založeného na gramatickém systému je znázorněno na obrázku 4.1.

Stěžejním prvkem, bez kterého by se gramatické systémy neobešly, je komunikace mezi jednotlivými gramatikami. Je proto absolutní nutností definovat *komunikační protokol*, podle kterého si jednotlivé komponenty mezi sebou předávají informace potřebné pro fungování systému.

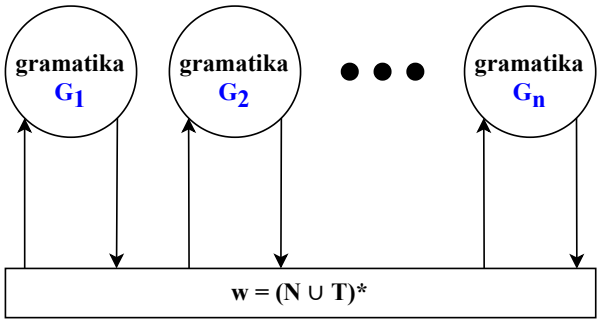
Výhody, jež se ke gramatickým systémům vážou jsou především *modularita* a *paralelismus*. Modularita s sebou nese pozitivní vlastnosti jako je srozumitelnost, udržitelnost a rozšiřitelnost, díky rozdělení většího celku na menší části (komponenty). Každá komponenta se navíc může zaměřit na určitý aspekt jazyka. Paralelismus poté umožnuje distribuovat výpočet mezi více současně pracujících zařízení. Gramatické systémy jsou děleny na dvě základní třídy podle toho, zdali pracují *sekvenčně* nebo *paralelně*. Třída *CD* (cooperating distributed) gramatických systémů pracuje sekvenčně, zatímco třída *PC* (parallel communicating), jak již sám název napovídá, pracuje paralelně (viz [11] a [1]). Tato práce se věnuje výhradně CD gramatickým systémům.



Obrázek 4.1: Obecné schéma myšlenky syntaktického analyzátoru založeného na gramatickém systému vyobrazující jeho vstupy a výstup.

4.2 CD gramatické systémy

Třída *CD* gramatických systémů pracuje sekvenčně. To znamená, že v jeden moment je vždy *aktivní* právě jedna gramatika. Aktivní gramatika (neboli komponenta) je ta, která v daný okamžik rozvíjí větnou formu derivačními kroky za použití pravidel této komponenty. Významným rysem CD gramatických systémů je větná forma. Ta je společná pro všechny komponenty (znázorněno obrázkem 4.2) a ty si tak musí navzájem status aktivní gramatiky předávat. Dobu, po kterou je komponenta aktivní, určuje *komunikační protokol*. V případě CD systémů jsou zavedeny *derivační režimy*, kdy každý definuje ukončovací podmínku. Mezi známé ukončovací podmínky patří například, že aktivní komponenta musí pracovat *právě* k derivačních kroků, *alespoň* k kroků, *nejvíce* k kroků nebo *maximální* počet derivačních kroků, který je komponenta schopná nad větnou formou provést.



Obrázek 4.2: Schéma gramatik G_1 až G_n CD gramatického systému demonstrující vztahy mezi nimi a větnou formou w.

Definice 4.1 (viz [1] přednáška CD Grammar Systems)**.** *CD* gramatický systém stupně n, $n \geq 1$, je $(n+3)$-tice

$$\Gamma = (N, T, S, P_1, \ldots, P_n),$$

kde

- N a T představují disjunktní abecedy *neterminálů* a *terminálů*,
- S je počáteční neterminál ($S \in N$),

- P_i je konečná množina pravidel nazývaná jako *komponenta* gramatického systému Γ pro $i = \{1, \ldots, n\}$.

Specifikovat konkrétní gramatiku jako komponentu je možné zápisem $G_i = (N, T, S, P_i)$ pro gramatický systém značený jako $\Gamma = (N, T, S, G_1, \ldots, G_n)$, kde $1 \leq i \leq n$. Gramatika G_i je tedy i-tou komponentou gramatického sytému Γ.

Režimy derivací

CD gramatické systémy v rámci komunikace mezi jednotlivými komponentami zavádí různé derivační režimy, které nyní budou představeny a formálně definovány. Nejdříve je ale třeba obohatit základní notaci derivačního kroku v rámci jedné gramatiky, jež byla představena na straně 8, o symbol specifikující, kterou komponentou systému byl onen krok proveden. Derivační krok vykonaný komponentou P_i je značen jako $uAv \Rightarrow_{P_i} uxv\ [p]$, kde p je pravidlo $p \in P_i$. Dále význam zápisu $x \not\Rightarrow y$, znamená, že z řetězce x není možné přímým derivačním krokem získat řetězec y. Režim $*$ značený jako $x \Rightarrow^*_{P_i} y$ říká, že pro komponentu P_i není stanovena doba, po kterou bude aktivní.

Definice 4.2 (viz [11])**.** Nechť $\Gamma = (N, T, S, P_1, \ldots, P_n)$ je gramatický systém.

1. Pro *ukončovací derivaci* i-tou komponentou pro všechna $i \in \{1, \ldots, n\}$ značenou jako $x \Rightarrow^t_{P_i} y$ platí

   $$x \Rightarrow^*_{P_i} y \text{ a zároveň } y \not\Rightarrow_{P_i} z \text{ pro všechna } z \in (N \cup T)^*$$

   Komponenta P_i je aktivní dokud má možnost jakýmkoliv svým pravidlem $p \in P_i$ upravit větnou formu.

2. Pro *derivaci k-kroků* i-tou komponentou pro všechna $i \in \{1, \ldots, n\}$ značenou jako $x \Rightarrow^{=k}_{P_i} y$ platí

   $$x \Rightarrow^k_{P_i} y$$

   Komponenta P_i, během doby co je aktivní, provede přesně k derivačních kroků nad větnou formou pomocí pravidel $p \in P_i$.

3. Pro *derivaci nejvíce k-kroků* i-tou komponentou pro všechna $i \in \{1, \ldots, n\}$ značenou jako $x \Rightarrow^{\leq k}_{P_i} y$ platí

   $$x \Rightarrow^{=k'}_{P_i} y \text{ pro některé } k' \leq k$$

   Komponenta P_i, během doby co je aktivní, provede buď přesně k nebo nebo méně derivačních kroků nad větnou formou pomocí pravidel $p \in P_i$.

4. Pro *derivaci alespoň k-kroků* i-tou komponentou pro všechna $i \in \{1, \ldots, n\}$ značenou jako $x \Rightarrow^{\geq k}_{P_i} y$ platí

   $$x \Rightarrow^{=k'}_{P_i} y \text{ pro některé } k' \geq k$$

   Komponenta P_i, během doby co je aktivní, provede buď přesně k nebo více derivačních kroků nad větnou formou pomocí pravidel $p \in P_i$.

Konvence 4.1. Nechť $D = \{*, t\} \cup \{= k, \leq k, \geq k \colon k \geq 1\}$ je množina doposud představených derivačních režimů.

Jazyk generovaný CD gramatickými systémy

Přesto, že je gramatický sytém složen z více gramatik, generuje právě jeden jazyk a to pomocí právě jednoho derivačního režimu.

Definice 4.3 (viz [11]). *Jazykem generovaným* pro gramatický systém $\Gamma = (N, T, S, P_1, \ldots, P_n)$ pomocí derivačního režimu $f \in D$, značeným jako $L_f(\Gamma)$, je

$$L_f(\Gamma) = \{w \in T^* : S \Rightarrow^f_{P_{i_1}} w_1 \Rightarrow^f_{P_{i_2}} \cdots \Rightarrow^f_{P_{i_m}} w_m, w_m = w, m \geq 1, 1 \leq i_j \leq n, 1 \leq j \leq m\}.$$

Příklad 4.1 (viz [1] přednáška CD Grammar Systems). Mějme CD gramatický systém druhého stupně $\Gamma_{eg} = (\{S, A, A', B, B'\}, \{a, b, c\}, S, P_1, P_2)$, kde

$$\begin{aligned} P_1 \;=\; & 1: S \to S \\ & 2: S \to AB \\ & 3: A' \to A \\ & 4: B' \to B \end{aligned} \qquad\qquad \begin{aligned} P_2 \;=\; & 1: A \to aA'b \\ & 2: A \to ab \\ & 3: B \to cB' \\ & 4: B \to c \end{aligned}$$

U gramatického systému Γ_{eg} bude možné pozorovat, že jím generovaný jazyk $L(\Gamma_{eg})$ se může lišit v závislosti na použitém derivačním režimu f.

1. Pokud $f \in \{*, t, =1, \geq 1\} \cup \{\leq k : k \geq 1\}$, tak

   $$L_f(\Gamma_{eg}) = \{a^m b^m c^n : m, n \geq 1\}.$$

2. Pokud $f \in \{=2, \geq 2\}$, tak

   $$L_f(\Gamma_{eg}) = \{a^n b^n c^n : n \geq 1\}.$$

   Zde jsou každou komponentou provedeny právě 2 derivační kroky, jelikož při aktivitě obou komponent jich není možné nad větnou formou provést více, viz

   $$S \Rightarrow_{P_1} S \Rightarrow_{P_1} AB \Rightarrow_{P_2} aA'bB \Rightarrow_{P_2} aA'bcB' \Rightarrow_{P_1} aAbcB' \Rightarrow_{P_1} aAbcB \Rightarrow_{P_2} aaA'bbcB \Rightarrow_{P_2}$$

   $$\Rightarrow_{P_2} aaA'bbccB' \Rightarrow_{P_1} aaAbbccB' \Rightarrow_{P_1} aaAbbccB \Rightarrow_{P_2} aaabbbccB \Rightarrow_{P_2} aaabbbccc$$

   Zkráceně zapsáno jako

   $$S \Rightarrow^{=2}_{P_1} AB \Rightarrow^{=2}_{P_2} aA'bcB' \Rightarrow^{=2}_{P_1} aAbcB \Rightarrow^{=2}_{P_2} aaA'bbccB' \Rightarrow^{=2}_{P_1} aaAbbccB \Rightarrow^{=2}_{P_2} aaabbbccc$$

   Výjimku při omezení na 2 derivační kroky může tvořit první aktivita komponenty P_1 při derivačním režimu $f = \geq 2$, kdy je možné opakovaným použitím pravidla $1: S \to S \in P_1$ provést až n derivačních kroků, kde $n \geq 2$.

3. Pokud $f \in \{=k, \geq k\}$, kde $k \geq 3$, tak

   $$L_f(\Gamma_{eg}) = \emptyset.$$

   V tomto případě je jazykem, který Γ_{eg} generuje, prázdná množina, jelikož již při první aktivitě komponenty P_2 není možné provést více než 2 derivační kroky nad větnou formou. Problém, jenž nastane, je možné vidět zde

   $$S \Rightarrow_{P_1} S \Rightarrow_{P_1} S \Rightarrow_{P_1} AB \Rightarrow_{P_2} aA'bB \Rightarrow_{P_2} aA'bcB' \Rightarrow_{P_2} \times$$

Hybridní CD gramatické systémy

Doposud představené gramatické sytémy, kde všechny komponenty pracují se stejným derivačním režimem, se nazývají *homogenní*. Z praktického hlediska se může zdát limitování gramatického systému na jeden derivační režim mírně svazující. To dalo vzniknout gramatickým systémům *hybridním*, kde je pro každou jednu komponentu definován také derivační režim, na jehož základě bude gramatika pracovat.

Definice 4.4 (viz [1] přednáška CD Grammar Systems)**.** Hybridní CD gramatický systém stupně n, $n \geq 1$, je $(n+3)$-tice

$$\Gamma = (N, T, S, (P_1, f_1), \ldots, (P_n, f_n)),$$

kde

- $N, T, S, P_1, \ldots, P_n$ jsou definovány stejně jako u klasického homogenního CD gramatického systému,
- f_i je derivační režim i-té komponenty P_i, kde $f_i \in D$, pro všechna $i = \{1, \ldots, n\}$.

Jazyk generovaný hybridním CD gramatickým systémem se v konečném důsledku liší, od jazyka generovaného homogenním CD systémem z definice 4.3, pouze specifikací derivačního režimu korespondujícího s příslušnou komponentou. Jazykem $L_f(\Gamma)$ tedy je

$$L(\Gamma) = \{w \in T^* : S \Rightarrow_{P_{i_1}}^{f_{i_1}} w_1 \Rightarrow_{P_{i_2}}^{f_{i_2}} \cdots \Rightarrow_{P_{i_m}}^{f_{i_m}} w_m, w_m = w, m \geq 1, 1 \leq i_j \leq n, 1 \leq j \leq m\}.$$

Konkrétní gramatiku, která je i-tou komponentou hybridního gramatického systému Γ, je opět možné specifikovat jako $G_i = (N, T, S, P_i, f_i)$.

CD gramatické systémy s vnitřním řízením

Pro CD gramatické systémy nebylo prozatím třeba *explicitně* definovat *podmínku* pro ukončení aktivity komponenty. Potřeba tomu nebylo, jelikož ukončovací podmínku představoval limit derivačních kroků daný derivačním režimem komponenty. CD gramatické systémy s *vnitřním řízením* přináší možnost definovat u jednotlivých komponent jak startovací, tak ukončovací podmínky na základě větné formy. Jinými slovy může konkrétní komponenta začít/přestat pracovat pouze tehdy, vyhovuje-li složení větné formy stanoveným podmínkám (viz [6]).

Definice 4.5 (viz [6])**.** Dynamicky řízený CD gramatický systém Γ stupně n, $n \geq 1$, je $(n+3)$-tice

$$\Gamma = (N, T, S, (P_1, \pi_1, \rho_1), \ldots, (P_n, \pi_n, \rho_n))$$

kde

- $N, T, S, P_1, \ldots, P_n$ jsou definovány stejně jako u klasického CD gramatického systému,
- π_i je predikát nad $(N \cup T)^*$ definující počáteční podmínku i-té komponenty P_i, pro všechna $i = \{1, \ldots, n\}$
- ρ_i je predikát nad $(N \cup T)^*$ nebo derivační režim definující ukončovací podmínku i-té komponenty P_i, pro všechna $i = \{1, \ldots, n\}$

I zde je možné specifikovat konkrétní gramatiku $G_i = (N, T, S, P_i, \pi_i, \rho_i)$, která je i-tou komponentou CD gramatického systému s vnitřním řízením Γ.

Typy predikátů

Stanovením speciálních typů predikátů, které určují podstatu startovací či ukončovací podmínky, lze usnadnit formální zápis gramatického systému s vnitřním řízením.

Definice 4.6 (viz [6])**.** Mějme gramatický systém s vnitřním řízením Γ. Nechť σ je predikát nad větnou formou w, kde $w \in (N \cup T)^*$.

- Predikát σ je typu (a) v případě, že $\sigma(w) = true$ pro všechna w. Komponenta s podmínkou tohoto typu může začít/přestat pracovat kdykoliv.
- Predikát σ je typu (rc), jestliže existují dvě množiny R a Q, pro které platí $R, Q \subseteq (N \cup T)$ a $\sigma(w) = true$ právě tehdy, kdy w obsahuje všechny symboly z R a zároveň w neobsahuje žádný symbol z Q.
- Predikát σ je typu (K), jestliže existují dva řetězce x a x', pro které platí $x, x' \in (N \cup T)^*$ a $\sigma(w) = true$ právě tehdy, kdy x je podřetězcem w a x' není podřetězcem w.

Konvence 4.2. Nechť Γ je CD gramatický systém stupně n a $X, Y \in \{a, rc, K\}$. Tvrzení, že Γ je typu (X, Y) znamená, že počáteční podmínka π_i je typu X a ukončovací podmínka ρ_i je typu Y, pro všechna $i = \{1, \ldots, n\}$. V případě, že derivační režim f nahrazuje ukončovací podmínku ρ, lze říci, že Γ je typu (X, f).

Kapitola 5

Návrh CD gramatického systému

Cílem této kapitoly je zavést nový modifikovaný typ *CD gramatických systémů* a následně takový systém konkrétně definovat. V části zavedení nového typu gramatického systému bude objasněno z jakých dosavadních typů systémů vychází, dále motiv jeho vzniku a následně i jeho samotná podoba. Podrobněji bude rozebrána struktura jednotlivých komponent a způsob komunikace mezi nimi, jež se výrazně liší od doposud známých CD gramatických systémů. Následovat bude definice konkrétního gramatického systému. Detailně poté budou rozebrány jednotlivé komponenty, tedy jejich gramatická pravidla, počáteční a ukončující podmínky, metody syntaktické analýzy, na kterých jsou založeny a s nimi související tabulky, dle kterých je syntaktická analýza řízena. To vše bude prezentováno s referencí na přílohu A, jež obsahuje rozsáhlé množiny neterminálů, terminálů a gramatických pravidel všech komponent. V poslední části bude dle oficiální literatury (viz [12]) definována podmnožina jazyka *C++*, kterou navržený gramatický systém generuje.

5.1 Zavedení nového typu CD gramatického systému

Nově zavedený gramatický systém je třídy CD a vychází z *hybridních* gramatických systémů (viz definice 4.4) a gramatických systémů *s vnitřním řízením* (viz definice 4.5). Hybridním systémům se přibližuje faktem, že jednotlivé komponenty mají *rozdílné podmínky* pro jejich aktivaci a deaktivaci. Gramatickým systémům s vnitřním řízením se podobá stanovením *počátečních* a *ukončujících* podmínek pro každou komponentu. Tyto podmínky jsou vyjádřeny pomocí *predikátů* nad větnou formou.

Klasické CD gramatické systémy, tak jak jsou zavedeny v definici 4.1, sice určují derivačním režimem, jak dlouho bude každá gramatika aktivní, ale není z nich již zřejmé, která gramatika má být aktivována jako další. Motivem vzniku tohoto druhu gramatických systémů je tedy vytvoření *deterministického komunikačního protokolu* tak, že bude jednoznačné, které komponenta má začít pracovat v momentě, kdy jiná pracovat přestane. Celý koncept tedy zahrnuje vícero prostředků ke specifikaci podmínek pro aktivaci a deaktivaci jednotlivých částí systému.

Definice 5.1. Hybridní CD gramatický systém s vnitřním řízením stupně n, $n \geq 1$, je $(n+3)$-tice

$$\Gamma = (N, T, S, G_1, \ldots, G_n),$$

kde

- N představuje abecedu *neterminálů*, kde $N = N_1 \cup \cdots \cup N_n$,

- T představuje abecedu *terminálů*, kde $T = T_1 \cup \cdots \cup T_n$ a $T \cap N = \emptyset$,
- S je počáteční neterminál, $S \in N$ a zároveň $S = S_i$, kde S_i je počátečním symbolem i-té komponenty gramatického systému Γ,
- G_i je i-tá *komponenta* gramatického systému Γ pro všechna $i = \{1, \ldots, n\}$.

Struktura komponenty

Právě strukturou komponent se navržený gramatický systém liší od těch doposud představených v kapitole 4. Jednotlivé komponenty mají pro komunikaci v systému definovány jak počáteční a ukončovací podmínky, tak derivační režim. Každá gramatika, jež je součástí gramatického systému, má svoji vlastní abecedu terminálů, aby bylo možné určit, který vstupní symbol už daná komponenta není schopna svými pravidly zpracovat a tvoří tak jakousi „zarážku“. Tato zarážka vymezuje pomyslnou hranici na vstupním řetězci, po kterou může komponenta pracovat se vstupními symboly. Stále však může v aktivitě pokračovat aplikací gramatických pravidel nad větnou formou. Stejně jako je tomu u terminálů, i abecedu neterminálů má každá komponenta vlastní. Abecedy neterminálů jednotlivých gramatik musí být navzájem disjunktní napříč celým gramatickým systémem. Disjunktní musí být proto, aby bylo jednoznačné, jakou gramatikou bude zpracován neterminál ve větné formě, který nemůže být zpracován právě aktivní gramatikou.

Definice 5.2. Komponentou gramatického systému $\Gamma = (N, T, S, G_1, \ldots, G_n)$ stupně n je gramatika G_i pro $i = \{1, \ldots, n\}$. Komponenta G_i je sedmice

$$G_i = (N_i, T_i, S_i, P_i, \pi_i, \rho_i, f_i),$$

kde

- N_i je abeceda *neterminálů* i-té komponenty, $N_i \cap N_j = \emptyset$ pro $j = \{1, \ldots, n\} - \{i\}$,
- T_i je abeceda *terminálů* i-té komponenty, $T_i \cap N_i = \emptyset$,
- S_i je *počáteční* neterminál i-té komponenty,
- P_i je konečná množina *pravidel* i-té komponenty ve tvaru $A \to x$, kde $A \in N_i$ a $x \in (N \cup T_i)$,
- π_i je predikát nad $(N \cup T)^*$ definující *počáteční* podmínku i-té komponenty,
- ρ_i je predikát nad $(N \cup T)^*$ definující *ukončovací* podmínku i-té komponenty,
- f_i je *derivační režim* i-té komponenty, $f_i \in D$, kde $D = \{t\} \cup \{= k, \leq k, \geq k : k \geq 1\}$.

Je důležité podotknout, že derivační režim $*$ není možné použít, jelikož nestanovuje jednoznačný počet derivačních kroků, které má daná komponenta provést a do systému by tak byl zaveden nedeterminismus.

Komunikační protokol

Komunikace mezi jednotlivými komponentami je dalším prvkem, který je pro navržený gramatický systém specifický. Jak již bylo zmíněno, tento systém definuje více elementů, které umožní gramatikám díky komunikaci pracovat deterministicky. Těmito elementy jsou

počáteční a ukončující podmínky π a ρ a derivační režim f. Nezbytností pro deterministický gramatický systém je, aby byl počet podmínek pro aktivaci a deaktivaci komponent konečný. Stanoveny jsou tedy dvě situace, podle kterých může výběr následující aktivní komponenty proběhnout. Předávání řízení je založeno na rekurzivním principu, viz definice 5.3 a příklad 5.1.

Konvence 5.1. Nechť G_i je *úvodní komponenta* právě v případě, kdy $S_i = S$. G_i je tak první aktivní gramatikou, kterou gramatický systém Γ započne svoji práci.

Definice 5.3. Mějme hybridní CD gramatický systém s vnitřním řízením stupně n, $n \geq 1$, $\Gamma = (N, T, S, G_1, \ldots, G_n)$. G_i a G_j jsou komponenty Γ, kde $i, j = \{1, \ldots, n\}$ a $i \neq j$ a G_i je právě aktivní. Definujme dvě situace, při kterých dojde předání řízení jiné komponentě.

1. *Předání řízení následující komponentě* nastává, když $\rho_i = true$ a $\pi_j = true$. V této situaci je G_i deaktivována a G_j aktivována. G_i se tak stává komponentou, která bude aktivována při rekurzivním návratu z G_j.

2. *Rekurzivní návrat k předchozí komponentě* nastává v případě, kdy G_i není schopna provést další derivační krok kvůli limitaci derivačním režimem f_i. Tehdy je G_i deaktivována a je aktivována komponenta, jež předala řízení komponentě G_i dle situace 1.

Neexistuje-li komponenta, ke které by byl proveden rekurzivní návrat podle situace 2, znamená to, že doposud aktivní gramatika je *úvodní komponenta* a že byl jazyk přijat.

Konvence 5.2. Derivace i-tou komponentou pro všechna $i \in \{1, \ldots, n\}$

- nad derivačním režimem $f_i \in D$ je značena jako
$$x \Rightarrow_{G_i}^{f_i} y,$$
- omezená ukončující podmínkou ρ_i je značena jako
$$x \Rightarrow_{G_i}^{\rho_i \pi_j} y,$$
kde π_j je vyhovující počáteční podmínka komponenty G_j, $j = \{1, \ldots, n\}$ a $i \neq j$.

Příklad 5.1. Uvažujme hybridní CD gramatický systém s vnitřním řízením $\Gamma = (N, T, S, G_a, G_b, G_c)$. Nechť Γ provede derivaci $x \Rightarrow^* y$, kde $x, y \in (N \cup T)^*$, kterou lze podrobně rozepsat například jako

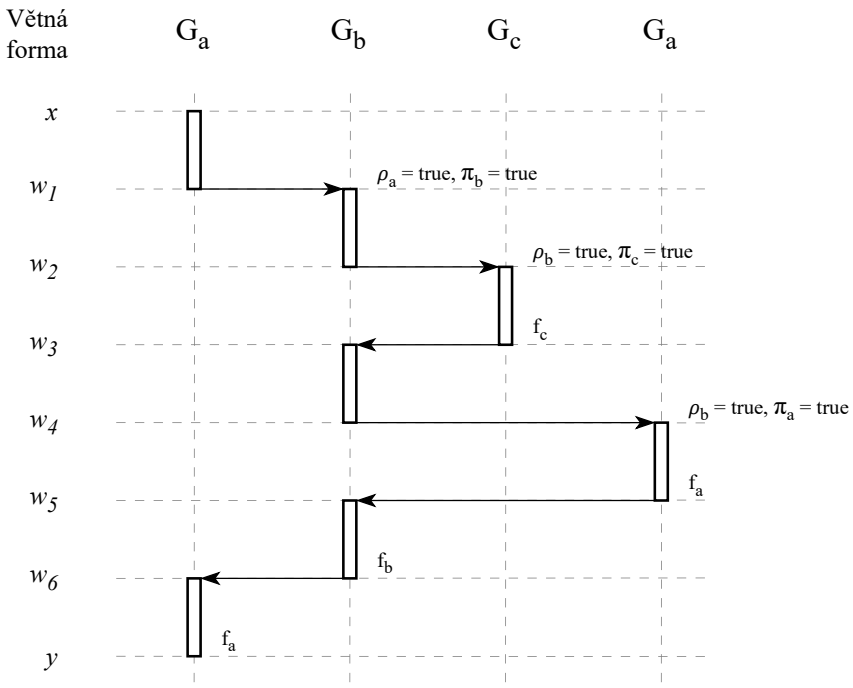
$$x \Rightarrow_{G_a}^{\rho_a \pi_b} w_1 \Rightarrow_{G_b}^{\rho_b \pi_c} w_2 \Rightarrow_{G_c}^{f_c} w_3 \Rightarrow_{G_b}^{\rho_b \pi_a} w_4 \Rightarrow_{G_a}^{f_a} w_5 \Rightarrow_{G_b}^{f_b} w_6 \Rightarrow_{G_a}^{f_a} y$$

Vzájemná komunikace mezi jednotlivými gramatikami je vyobrazena pomocí sekvenčního diagramu na obrázku 5.1, kde je možné pozorovat, jakým způsobem probíhá rekurzivní návrat k předchozím komponentám v čase. Sekvenční diagram přímo odpovídá průběhu derivace uvedené v tomto příkladě.

Predikáty typu (l) a (s)

Definice 5.4. Mějme hybridní CD gramatický systém s vnitřním řízením Γ. Nechť σ je predikát nad větnou formou w, kde $w \in (N \cup T)^*$.

- Predikát σ je typu (l), jestliže existuje množina L, pro kterou platí $L \subseteq (N \cup T)$ a $\sigma(w) = true$ právě tehdy, kdy symbol $X \in L$ je *sufixem* větné formy w.
- Predikát σ je typu (s), jestliže existuje řetězec x, pro který platí $x \in (N \cup T)^*$ a $\sigma(w) = true$ právě tehdy, kdy x je *sufixem* větné formy w.



Obrázek 5.1: Sekvenční diagram vyobrazující komunikaci mezi komponentami hybridního CD gramatického systému s vnitřním řízením při derivaci z příkladu 5.1.

5.2 Definice navrženého gramatického systému

V této podkapitole bude definován konkrétní CD gramatický systém, který je v rámci této práce také prakticky implementován (více viz kapitola 6). Jedná se o typ gramatického systému, jež byl představen v předchozí podkapitole 5.1. Kvůli udržení přehlednosti textu jsou textově rozsáhlejší prvky komponent (konkrétně množiny terminálů, neterminálů a gramatických pravidel) uvedeny v příloze A. V příloze A jsou také k naleznutí odkazy na tabulky syntaktické analýzy jednotlivých komponent.

Mějme tedy hybridní CD gramatický systém s vnitřním řízením stupně 4, který generuje podmnožinu jazyka *C++* (viz podkapitola 5.3), zapsaný jako

$$\Gamma_{cpp} = (N, T, S, G_1, G_2, G_3, G_4),$$

kde

- $N = N_1 \cup N_2 \cup N_3 \cup N_4$,
- $T = T_1 \cup T_2 \cup T_3 \cup T_4$,
- $S =$ <prog_main>, kde <prog_main> $\in G_1$

Každé komponentě G_1 až G_4 gramatického systému Γ_{cpp} bude následně věnována samostatná sekce. Kromě samotné definice dané komponenty bude objasněna také metoda syntaktické analýzy, nad kterou je v praktické části daná komponenta implementována.

Komponenta $\mathbf{G_1}$

První a zároveň *úvodní* komponentou gramatického systému Γ_{cpp} je

$$G_1 = (N_1, T_1, S_1, P_1, \pi_1, \rho_1, f_1) \text{ je typu } (a, l),$$

kde

- N_1, T_1 a P_1 viz příloha A.1,
- $S_1 =$ <prog_main>,
- π_1 není specifikován, jelikož je predikátem typu (a),
- $\rho_1 = (\{AS, S'\})$,
- $f_1 = t$

Komponenta G_1 je zaměřena na analýzu těla programu, konkrétně na definici funkcí a proměnných a základních konstrukcí selekce (*if*, *switch*) a iterace (*while*, *do-while*, *for*). Gramatika G_1 je navržena tak, aby mohla v rámci syntaktického analyzátoru pracovat na základě metody *prediktivní* syntaktické analýzy. Z toho vyplývá, že G_1 je *LL gramatikou*, pro kterou je také sestrojena *LL tabulka*.

Technika nahlédnutí vpřed

Při prozkoumání LL tabulky gramatiky G_1 si je možné povšimnout, že v buňkách na řádku neterminálu <prog_main> a zároveň ve sloupcích terminálů reprezentujících datové typy $(int, float, \dots)$ není jednoznačné, jaké pravidlo má být v daný moment aplikováno. Jedná se o případ, kdy pomocí metody založené na $LL(1)$ prediktivní syntaktické analýze nelze rozhodnout, zda se jedná o začátek definice funkce či proměnné. I když z teoretického hlediska již není G_1 v tomto případě považováno za LL gramatiku, prakticky lze problém vyřešit pomocí *techniky nahlédnutí vpřed*.

Definice 5.5. *Technikou nahlédnutí vpřed* je možné načíst *napřed potřebný počet tokenů*, na základě kterých bude možné určit, jaké pravidlo má být aplikováno při výskytu nejednoznačnosti v LL tabulce.

Komponenta $\mathbf{G_2}$

Druhou komponentou gramatického systému Γ_{cpp} je

$$G_2 = (N_2, T_2, S_2, P_2, \pi_2, \rho_2, f_2) \text{ je typu } (l, s),$$

kde

- N_2, T_2 a P_2 viz příloha A.2,
- $S_2 = AS$,
- $\pi_2 = (\{AS\})$,
- $\rho_2 =$ (**id** ( ),
- $f_2 = t$

Na první pohled si je možné všimnout, že počet prvků v množině neterminálů N_2 gramatiky G_2 je malý. Tento úkaz je velmi příznačný pro gramatiky, které jsou prostředkem pro *precedenční syntaktickou analýzu*. Z toho vyplývá, že komponenta G_2 slouží pro syntaktickou analýzu *matematických výrazů*, a to včetně operátorů přiřazení. Precedenční syntaktickou analýzou je totiž možné zpracovat výhradně matematické výrazy složené z operandů a operátorů s danou precedencí a asociativitou.

Precedenční tabulka

Pro sestrojení precedenční tabulky G_2 je nutné doplnit kroky pro konstrukci tabulky, popsané na straně 21, o následující pravidla.

- *Identifikátory:*
  - Pokud a je operátor přiřazení a id je identifikátor, tak $id = a$.
- *Konstanty:*
  - Pokud $a \in T$ a c je konstanta, kde a může legálně přímo předcházet c, tak $a < c$.
  - Pokud $a \in T$ a c je konstanta, kde a může legálně přímo následovat za c, tak $c > a$.

  Hodnoty buněk konstant jsou stanoveny ve *stejném kroku* jako hodnoty identifikátorů.

Komponenta G_3

Třetí komponentou gramatického systému Γ_{cpp} je

$$G_3 = (N_3, T_3, S_3, P_3, \pi_3, \rho_3, f_3) \text{ je typu } (l, s),$$

kde

- N_3, T_3 a P_3 viz příloha A.3,
- $S_3 = S'$,
- $\pi_3 = (\{S'\})$,
- $\rho_3 = (\ \mathbf{id}\ (\ )$,
- $f_3 = t$

Komponenta G_3 slouží v Γ_{cpp} také pro zpracování *matematických výrazů*, nicméně tentokrát bez operátorů přiřazení. Na této komponentě bude založena část syntaktického analyzátoru, která pracuje na principu *SLR syntaktické analýzy*. Je známo, že metoda SLR syntaktické analýzy je silnější než metody precedenční a LL prediktivní a pravděpodobně by bylo možné generovat jazyk, který generuje gramatický systém Γ_{cpp}, jen s využitím jedné SLR gramatiky. Přesto byla do systému zahrnuta jen pro zpracování jedné menší části kvůli demostraci *modularity* gramatických systémů, tedy že každá jeho komponenta může pracovat na jiném principu, a přesto spolupracovat s ostatními. Díky menšímu množství gramatických pravidel navíc bylo možné ručně sestavit *SLR tabulku* menších rozměrů a systém je tak celkově *didakticky mnohem průhlednější*.

Komponenta G_4

Čtvrtou komponentou gramatického systému Γ_{cpp} je

$$G_4 = (N_4, T_4, S_4, P_4, \pi_4, \rho_4, f_4) \text{ je typu } (s, l),$$

kde

- N_4, T_4 a P_4 viz příloha A.4,
- $S_4 =$ <function_call>,
- $\pi_4 =$ (**id** **(**),
- $\rho_4 = (\{AS\})$,
- $f_4 = t$

Poslední komponenta je stejně jako komponenta G_1 *gramatikou LL*, pracuje na základě metody *prediktivní syntaktické analýzy* a je pro ni sestrojena LL tabulka. Slouží výhradně pro zpracování konstrukce *volání funkce* ve výrazu.

5.3 Jazyk generovaný gramatickým systémem

Jazyk, který generuje gramatický systém Γ_{cpp} definovaný v podkapitole 5.2, je podmnožinou jazyka *C++*. V této podkapitole bude tato podmnožina definována dle knihy *The C++ standard* (viz [12]). Popsány tedy budou jednotlivé konstrukce jazyka, kterých se tato práce dotýká.

Notace 5.1. V rámci syntaktické notace použité v této podkapitole jsou syntaktické kategorie indikovány písmem psaným *kurzívou* a terminální symboly `strojopisným písmem`. Varianty konkrétní syntaktické kategorie jsou vždy odděleny novým řádkem. Volitelný výskyt symbolu je označen zkratkou *„opt“* zapsanou dolním indexem (například $syntax_{opt}$).

Globální rozsah

V globálním rozsahu programu je možné definovat proměnné a funkce dle následujících syntaktických konstrukcí:

- *main-program*
  variable-definition
  function-definition
- *variable-definition:*
  simple-type-specifier `id`
  simple-type-specifier `id =` *assignment-expression*
- *function-definition:*
  simple-type-specifier `id (` *parameter-definition-list*$_{opt}$ `)` *compound-statement*
- *parameter-definition-list:*
  variable-definition
  parameter-definition-list `,` *variable-definition*

- *simple-type-specifier:*
  `int`
  `float`
  `double`
  `char`
  `string`
  `bool`
  `void`

Rozsah těla funkce

V rámci těla funkce je možné využít příkazů selekce (*if*, *switch*), iterace (*while*, *do-while*, *for*) a několika dalších, například skokových, příkazů (*break*, *continue*, *return*) dle následující specifikace syntaxe:

- *compound-statement:*
  `{` *statement-seq*$_{opt}$ `}`

- *statement-seq:*
  statement
  statement-seq statement

- *statement:*
  labeled-statement
  expression-statement
  selection-statement
  iteration-statement
  jump-statement
  variable-definition

- *labeled-statement:*
  `case` *constant-expression* `:` *statement*
  `default :` *statement*

- *expression-statement:*
  expression$_{opt}$ `;`

- *condition:*
  expression
  variable-definition

- *selection-statement:*
  `if (` *condition* `)` *compound-statement*
  `if (` *condition* `)` *compound-statement* `else` *compound-statement*
  `switch (` *condition* `)` *compound-statement*

- *iteration-statement:*
  `while (` *condition* `)` *compound-statement*
  `do` *compound-statement* `while (` *condition* `) ;`
  `for (` *for-init-statement condition*$_{opt}$ `;` *expression*$_{opt}$ `)` *compound-statement*

- *for-init-statement:*
  expression-statement
  variable-definition `;`

- *jump-statement:*
  `break ;`
  `continue ;`
  `return` *expression-statement*

Matematické výrazy

Matematické výrazy mohou obsahovat následující symboly:

- *Aritmetické operátory:* `+`, `-`, `*`, `/`, `%`
- *Relační a logické operátory:* `==`, `!=`, `>`, `<`, `>=`, `>=`, `!`, `||`, `&&`
- *Operátory přiřazení:* `=`, `+=`, `-=`, `*=`, `/=`, `%=`
- *Operátory inkrementace a dekrementace:* `++`, `--` (prefixové i postfixové varianty)
- *Operandy:* literály, identifikátory proměnných
- *Závorky:* `(`, `)`

Syntaktická kategorie *expression* může obsahovat veškeré symboly zmíněné výše, zatímco *assignment-expression* nemůže obsahovat operátory přiřazení. Precedence a asociativita jednotlivých operátorů jsou stanoveny dle [5].

V rámci matematických výrazů se legálně může vyskytovat konstrukce volání funkce, která je definována následovně:

- *function-call:*
  `id (` *argument-expression-list*$_{opt}$ `)`

- *argument-expression-list:*
  expression
  argument-expression-list `,` *expression*

Kapitola 6

Implementace gramatického systému

V této kapitole bude představena praktická stránka práce, tedy samotná implementace přední části překladače, jejímž výsledkem je konzolová aplikace. Přední částí překladače je konkrétně myšlen lexikální a syntaktický analyzátor. V první řadě bude představeno několik základních vlastností výsledného produktu a použité technologie. V další fázi budou popsány metody a struktura *lexikálního analyzátoru*. Hlavní část kapitoly bude věnována *syntaktickému analyzátoru* založenému na gramatickém systému, jež zahrnuje větší množství datových struktur, algoritmů i pomocných metod a je středobodem praktické části. Důraz bude kladen především na funkcionalitu jednotlivých typů komponent, komunikaci mezi komponentami a zotavení analyzátoru po syntaktické chybě. V závěru kapitoly budou prezentovány vstupy a výstupy aplikace a ve zkratce také způsob jejího testování.

6.1 Úvodní specifikace aplikace

Praktická část, tedy přední část překladače, je realizována jako konzolová aplikace, jejíž vstupy a výstupy jsou popsány v podkapitole 6.4. Využito je technologií *C#* a *.NET* verze 8.0 a vývoj probíhal ve vývojovém prostředí *Visual Studio Comunity 2022*. *C#* je vysokoúrovňový objektově orientovaný programovací jazyk, zatímco *.NET* je vývojová platforma poskytující množství knihoven a běhové prostředí pro jazyk *C#*. Přesto, že jsou všechny zmíněné technologie vyvíjeny společností *Microsoft*, je možné výslednou aplikaci, kromě platformy *Windows*, zkompilovat také například pro různé *linuxové distribuce* (Fedora, Ubuntu, ...).

Je důležité podotknout, že ačkoliv jsou implementovány pouze první dvě části překladače, je systém navržený tak, aby jej bylo možné rozšířit o části další. Kompletní struktura *zdrojových souborů* je k vidění v příloze B. Na jednotlivé soubory bude jejich názvem častokrát ve zbytku této kapitoly odkazováno.

6.2 Implementace lexikálního analyzátoru

Veškerá funkcionalita a metody *lexikální analýzy* jsou popsány ve zdrojovém souboru `Lexical/Scanner.cs`. Lexikální analyzátor (také nazývaný jako *skener*) je první fází překladu. Základním modelem lexikálního analyzátoru jsou *deterministické konečné automaty*. Pomocí nich je možné rozpoznávat ve zdrojovém programu jednotlivé *lexémy*, které jsou

poté reprezentovány *tokeny*. Zdrojový soubor je čten znak po znaku. S načtenými znaky jsou realizovány přechody mezi stavy konečného automatu a *stavy koncové* představují přijetí právě jednoho lexému. Situace, kdy je načten znak, se kterým není možné z nekoncového stavu provést přechod, je vyhodnocena jako *lexikální chyba*.

Pro každý stav existuje metoda, která implementuje jeho logiku tak, že podle aktuálně přečteného znaku určí následující stav skeneru. V rámci činnosti lexikálního analyzátoru je udržována informace o stavu, ve kterém se aktuálně nachází, na jejíž základě je v každém kroku volána metoda příslušného stavu. Dále je udržována informace o čísle řádku a o pozici na něm, na kterých se skener ve vstupním zdrojovém souboru nachází. To umožňuje informovat uživatele o přesném místě výskytu lexikální nebo syntaktické chyby.

Token

Struktura *tokenů*, které skener předává na vyžádní syntaktickému analyzáturu, je definována v souboru `Lexical/Token.cs`. Instance třídy *Token* reprezentuje následující data:

- *Typ tokenu*, který je v syntaktické analýze vnímán jako konkrétní *terminál*.
- *Hodnota tokenu*, která je nutná pro případné rošíření aplikace o další části překladače.
- *Řádek* výskytu a *pozice* výskytu na řádku, jež umožní lokalizovat lexém ve zdrojovém souboru.

Metody lexikálního analyzátoru

Lexikální analyzátor poskytuje tři základní metody, které jsou definovány přímo pro potřeby syntaktického analyzátoru. Primárně se jedná o metodu *GetNextToken()*, jejíž návratovou hodnotou je instance třídy `Token`, která reprezentuje následující doposud nepřečtený lexém. Další, již více specifickou metodou, je *ReturnBy(n)*, která poskytuje možnost vrátit skener do stavu, ve kterém se nacházel před načtením posledních n tokenů. Poslední metodou je *Lookahead(n)*, jež umožňuje nahlédnout n tokenů vpřed, aniž by to ovlivnilo aktuální stav skeneru. Přesně to je nutné provést v situaci popsané v podkapitole 5.2 u komponenty G_1. Návratovou hodnotou *Lookahead(n)* je seznam dopředu načtených tokenů.

Prefixová versus postfixová inkrementace/dekrementace

V přijímaném jazyce se může v matematickém výrazu legálně vyskytnout operátor *inkrementace*. Rozlišujeme dva typy inkrementace, *prefixovou* a *postfixovou*, kde je nutné oba typy zastoupit vlastním terminálem, aby mohla být syntaktická analýza provedena korektně. Tím vzniká problém takový, že pro jeden lexém ++ existují dva terminály, které může symbolizovat. Jelikož řešení tohoto problému přesahuje možnosti konečného automatu, je potřeba typ inkrementace určit umělým krokem na základě předcházejícího tokenu následovně. Předchází-li lexému ++ token typu *id*, je považován za postfixovou inkrementaci, v opačném případě za prefixovou. Situace se řeší stejně u dekrementace a lexému −−.

6.3 Implementace syntaktického analyzátoru

Syntaktický analyzátor (dále jen SA) v procesu překladu logicky následuje za analýzou lexikální a z pohledu implementace se jedná celkově o komplexnější problematiku. Komplexita je navíc umocněna tím, že SA pracuje na základě *gramatických systémů* a nikoliv pouze

s jednou gramatikou. SA naprogramovaný pro tuto práci je založen na hybridním CD gramatickém systému s vnitřním řízením Γ_{cpp}, který je definovaný (včetně jeho komponent) v předchozí kapitole v podkapitole 5.2. Cílem bylo realizovat SA tak, aby bylo možné snadno a rychle promítnout změny v definici určité komponenty gramatického systému do implementace. Tohoto cíle je dosaženo tím způsobem, že stačí na konkrétních místech pozměnit data gramatických pravidel a tabulek analýzy na požadované hodnoty a není třeba měnit samotný algoritmus zprostředkovávající funkcionalitu syntaktické analýzy. Následovat bude popis jednotlivých implementovaných součástí SA.

Množiny gramatických pravidel

Gramatická pravidla všech komponent G_1 až G_4 gramatického systému Γ_{cpp}, tak jak jsou definována v příloze A, je možné nalézt v souboru `Grammars/GrammarSetsData.cs`, který slouží jako datová sada pro SA. Zásadní vlastnosti jednoho konkrétního pravidla, kterými jsou číslo pravidla, levá strana pravidla a pravá strana pravidla, jsou definovány třídou *GrammarRule* (viz zdrojový soubor `Grammars/GrammarRule.cs`). Jednotlivé množiny gramatických pravidel P_1 až P_4 jsou reprezentovány seznamem instancí třídy *GrammarRule*.

Tabulky syntaktické analýzy

Průběh syntaktické analýzy všech komponent gramatického systému Γ_{cpp} je založen na tabulkách syntaktické analýzy, jejichž postup konstrukce je popsán v podkapitolách 3.3 a 3.4. Implementace těchto tabulek tedy hraje v SA významnou roli. Všechny soubory související s tabulkami jsou k vidění ve složce `Tables/`. Soubory definující třídy konkrétních typů tabulek se shodují s názvy těchto tříd.

Přirozeně jsou tabulky realizovány pomocí dvojrozměrného pole, které je definováno jako vlastnost abstraktní třídy *ParsingTable*. Tato třída definuje metodu *FillTableFromCSV()*, jež je zásadní pro flexibilitu modifikování tabulek analýzy, dojde-li v komponentě ke změně gramatických pravidel. Zmíněná metoda umožňuje naplnit tabulku daty ze souboru formátu *CSV*. Soubory CSV je totiž možné exportovat z tabulek vytvořených v tabulkových procesorech, kde je správa obsahu tabulek pohodlnější než IDE. Formát CSV se dá také očekávat jako výstup případných externích programů určených pro generování tabulek analýzy na základě znalosti gramatiky.

Pro každý typ syntaktické analýzy, který je v rámci gramatického systému použitý, je zavedena podtřída třídy *ParsingTable*. Konkrétně se jedná o podtřídy *LLParsingTable* pro prediktivní LL SA, *PrecedenceParsingTable* pro precedenční SA a *SLRParsingTable* pro SLR SA. Všechny tyto podtřídy definují, jakým datovým typem mají být buňky tabulky reprezentovány a jak mají být data z CSV souboru zpracována. Data ve formátu CSV však musí naprosto přesně splňovat předem určenou podobu, aby mohla být korektně zpracována.

Slovníky jako mapovací funkce

Řádky a sloupce tabulek analýzy v majoritě případů reprezentují konkrétní terminály nebo neterminály (dále jen symboly). Během syntaktické analýzy je běžně potřeba získat hodnotu určité buňky tabulky na základě znalosti jednoho nebo dvou symbolů. Aby však bylo možné k prvku dvojrozměrného pole přistoupit, je třeba ke každému symbolu, jenž je v tabulce reprezentován sloupcem či řádkem, přiřadit index. Právě pomocí mapovací funkce je možné zjistit, který index je k danému symbolu přidružen. Mapovače jsou realizovány pomocí datové struktury zvané *slovník*. Slovníky umožňují efektivní vyhledávání hodnot podle

klíče. V tomto případě klíč představuje určitý symbol, jehož hodnotou je k němu vázaný index. Množina klíčů v rámci jednoho mapovače může také reprezentovat množinu terminálů T_i nebo neterminálů N_i komponenty G_i, kde $i = \{1, 2, 3, 4\}$. Není-li možné pomocí některého symbolu získat ze slovníku hodnotu, znamená to, že tento symbol není součástí množiny T_i nebo N_i.

Algoritmy jednotlivých komponent

Nyní, když jsou definované všechny datové struktury pro syntaktický analyzátor (množiny gramatických pravidel, tabulky analýzy a mapovací funkce), mohou být popsány algoritmy, na jejichž základě komponenty gramatického systému pracují.

Definice 6.1. *Aktivní doba* je doba, po kterou je komponenta G_i, kde $i = \{1, \ldots, n\}$ a n je stupeň gramatického systému, aktivní. Aktivita je zahájena splněním počáteční podmínky π_i a ukončena dovršením limitu počtu derivačních kroků stanovených derivačním režimem f_i. Aktivita komponenty G_i může být v průběhu činnosti pozastavena splněním ukončující podmínky ρ_i. Po pozastavení je aktivita obnovena rekurzivním návratem z komponenty, která byla aktivována bezprostředně po přerušení.

Jádrem celé implementace SA jsou třídy zavádějící logiku jednotlivých typů syntaktické analýzy, které ze zmiňovaných algoritmů vycházejí. Tyto třídy jsou definovány ve stejnojmenných zdrojových souborech `Parsers/LLParser.cs`, `Parsers/PrecedenceParser.cs` a `Parsers/SLRParser.cs`. Instance jedné z těchto tříd představuje právě jednu *aktivní dobu* konkrétní komponenty G_i. Voláním metody *Parse()* nad konkrétní instancí se stává komponenta vztažená k dané instanci aktivní.

Konkrétní komponenty jsou reprezentovány instancemi tříd definovaných ve zdrojových souborech ve složce `Components/`. Vlastnostmi každé komponenty jsou název komponenty, množina gramatických pravidel, tabulka syntaktické analýzy a mapovací funkce. Komponenty založené na LL prediktivní syntaktické analýze navíc zahrnují množiny *First()* a *Follow()*, jež jsou využity pro zotavení po chybě.

Algoritmus prediktivní LL syntaktické analýzy

Algoritmus 6.1 prediktivní LL syntaktické analýzy vychází z [2] z kapitoly Syntaktická analýza shora dolů. Jelikož se jedná o metodu shora dolů, je v inicializační fázi SA na zásobník umístěn počáteční neterminál dané komponenty. Následně je v každám kroku analýzy na základě typu symbolu na vrcholu zásobníků rozhodnuto, jaká akce bude provedena. Je-li na vrcholu *terminál*, bude porovnán se vstupním symbolem. Po úspěšném porovnání bude odebrán ze zásobníku a načte se následující vstupní symbol. Pokud je nejvyšším symbolem umístěným na zásobníku *neterminál*, bude na základě LL tabulky rozhodnuto, kterou pravou stranou gramatického pravidla komponenty má být neterminál nahrazen. Řetězec je nutné na zásobník vložit v opačném pořadí, jelikož je konstruována nejlevější derivace.

Algoritmus 6.1: Prediktivní LL syntaktická analýza

Vstup: Komponenta G_i, LL tabulka $Table_{G_i}$ a vstupní řetězec tokenů $x \in T_i^*$
Výstup: *Úspěch* nebo *neúspěch*

a je aktuální token na vstupu
$pTop$ představuje symbol na vrcholu zásobníku

1: $Push(S_i)$ na zásobník
2: **while** nebylo dosaženo *úspěchu* nebo *chyby* **:**
3: **if** $pTop == \$$ **and** $a == \$$ **:**
4: *Úspěch*
5: **else if** $pTop \in T_i$ **and** $a == pTop$ **:**
6: $Pop(pTop)$
7: $a = GetNextToken()$
8: **else if** $pTop \in N$ **and** $p\colon pTop \to X_1 \dots X_n \in Table_{G_i}[pTop, a]$ **:**
9: $Pop(pTop)$
10: $Push(X_n \dots X_1)$
11: **else :**
12: *Chyba*

Algoritmus precedenční syntaktické analýzy

Algoritmus 6.2 precedenční syntaktické analýzy vychází z [3] z kapitoly Bottom-Up Parsing. V každém kroku analýzy je hodnotou buňky v precedenční tabulce komponenty G_i určeno, jestli bude provedena operace *redukce* nebo *posunu*. Operace redukce je provedena nad řetězcem symbolů x na vrcholu zásobníku, kde $x \in (N_i \cup T_i)^*$. Délka x je určena tzv. *zarážkami*, které jsou vkládány mezi symboly na zásobníku. Zarážky jsou v algoritmu 6.2 značeny jako $\lfloor$. Redukovaný řetězec x je tedy tvořen symboly od zarážky nejblíže vrcholu zásobníku až po jeho samotný vrchol. Následně je nalezeno pravidlo $p\colon A \to y$ takové, že $p \in P_i$ a $x = y$. V poslední fázi redukce je x nahrazeno za A. Operace posunu zahrnuje vložení zarážky za doposud nejvýše položený terminál na zásobníku, vložení terminálu na vstupu na zásobník a načtení následujícího vstupního tokenu. Zarážky jsou ve výsledném programu realizovány *pomocným zásobníkem* uržujícím informace o indexech terminálů na zásobníku, za kterými by se zarážka nacházela.

Algoritmus 6.2: Precedenční syntaktická analýza

Vstup: Komponenta G_i, precedenční tabulka $Table_{G_i}$ a vstupní řetězec tokenů $x \in T_i^*$

Výstup: *Úspěch* nebo *neúspěch*

a je aktuální token na vstupu
$tTop$ představuje terminál nejblíže vrcholu zásobníku

```
1:  Push($) na zásobník
2:  while nebylo dosaženo úspěchu nebo chyby :
3:      switch Table_{G_i}[tTop, a] :
4:          case = :
5:              Push(a)
6:              a = GetNextToken()
7:          case < :
8:              zaměň tTop na zásobníku za tTop ⌊
9:              Push(a)
10:             a = GetNextToken()
11:         case > :
12:             if ⌊ u je na vrcholu zásobníku and p: A → u ∈ P_i :
13:                 zaměň ⌊ u na zásobníku za A
14:             else :
15:                 Chyba
16:         case OK :
17:             Úspěch
18:         case prázdná buňka :
19:             Chyba
```

Algoritmus SLR syntaktické analýzy

Algoritmus 6.3 SLR syntaktické analýzy vychází z [3] z kapitoly Bottom-Up Parsing. Výchozí stav SLR SA je počáteční stav q_0. Následné akce provedené v každém jednom kroku analýzy jsou určovány *akční* částí SLR tabulky. Buňka relevantní k danému kroku určuje, podobně jako u precedenční analýzy, zdali bude provedena operace *posunu* či *redukce*. Provést operaci posunu znamená, přesunout aktuální token na vstupu na vrchol zásobníku a načíst token následující. U SLR operace redukce je nespornou výhodou fakt, že tabulka specifikuje gramatické pravidlo, podle kterého má být redukce provedena. Není proto nutné pracovat se zarážkami, ale stačí porovnat pravou stranu pravidla se symboly na vrcholu zásobníku a při shodě je nahradit neterminálem ze strany levé. Na základě zredukovaného neterminálu je poté v *přechodové* části tabulky určen následující stav.

Algoritmus 6.3: SLR SYNTAKTICKÁ ANALÝZA

Vstup: Komponenta G_i, SLR tabulka $Table_{G_i}$, množina všech SLR stavů Q a vstupní řetězec tokenů $x \in T_i^*$

Výstup: *Úspěch* nebo *neúspěch*

a je aktuální token na vstupu
prvkem zásobníku je dvojice $\langle X, q \rangle$, kde $X \in (N \cup T)$ a $q \in Q$

1: $stav = q_0$
2: $Push(\langle \$, stav \rangle)$ na zásobník
3: **while** nebylo dosaženo *úspěchu* nebo *chyby* **:**
4: **switch** $Action_{G_i}[stav, a]$ **:**
5: **case** sq **:**
6: $Push(\langle a, q \rangle)$
7: $stav = q$
8: $a = GetNextToken()$
9: **case** rp **:**
10: **if** $\langle ?, q \rangle \langle X_1, ? \rangle \dots \langle X_n, ? \rangle$ je na vrcholu zásobníku **and** $p\colon A \to X_1 \dots X_n \in P_i$ **:**
11: $stav = Goto_{G_i}[q, A]$
12: zaměň $\langle X_1, ? \rangle \dots \langle X_n, ? \rangle$ na zásobníku za $\langle A, stav \rangle$
13: **else :**
14: *Chyba*
15: **case** OK **:**
16: *Úspěch*
17: **case** prázdná buňka **:**
18: *Chyba*

Komunikace mezi jednotlivými komponentami

Předávání aktivity mezi komponentami probíhá na principu *rekurze*. Přesněji řečeno je realizováno pomocí rekurzivního volání metody mezi instancemi tříd aktivní doby (*LLParser*, *SLRParser*, *PrecedenceParser*), které mají stejnou nadřazenou abstraktní třídu, jež rekurzivně volanou metodu *Parse()* deklaruje. Tato nadřazená třída je definována ve zdrojovém souboru `Parser.cs`. Jak již bylo zmíněno, k rekurzivnímu volání následující komponenty G_j (přerušení aktivity komponenty) dochází tehdy, kdy podoba větné formy vyhovuje ukončujícímu predikátu ρ_i právě aktivní komponenty G_i a zároveň počátečnímu predikátu π_j. Naopak rekurzivní návrat nastává v momentě, kdy aktuálně pracující instance aktivní doby nemůže s danou komponentou v analýze pokračovat. Tímto způsobem je prakticky simulován *derivační režim t*, který je v případě gramatického systému Γ_{cpp} zaveden pro všechna f_i, kde $i = \{1, 2, 3, 4\}$.

Derivažní režim t

Derivační režim t je prakticky realizován následovně. Za dovršení limitu derivačních kroků, které může instance *aktivní doby* provést, se považuje úspěšná dílčí syntaktická analýza danou komponentou. Po úspěšné dílčí syntaktické analýze je tedy příkazem `return` v metodě *Parse()* proveden návrat k předchozí instanci, a to simuluje předání aktivity mezi

komponentami v gramatickém systému. Díky skutečnosti, že každá komponenta má specifikovanou vlastní množinu *terminálů*, je možné u komponent založených na principu *zdola nahoru* jednoduše rozhodnout o délce řetězce načteného ze vstupu. Prakticky je vložen na zásobník symbol $ reprezentující *ukončovač řetězce* okamžitě při načtení terminálu a komponentou G_i, kde $a \neq T_i$. Terminál a se poté pokusí zpracovat následující komponenta, které komponenta G_i předá aktivitu rekurzivním návratem.

Ukončovací podmínka ρ_i a počáteční podmínka π_i

Je-li kdykoliv během činnosti aktivní komponenty G_i gramatického systému Γ_{cpp} splněn predikát specifikovaný *typem* komponenty a ukončující podmínkou ρ_i nad aktuální *větnou formou*, je další postup následující. Na základě počátečního predikátu π_j komponenty G_j systému Γ_{cpp}, kde $i \neq j$ a $\pi_j = true$, je vytvořena instance *aktivní doby* komponenty G_j, nad kterou je následně volána metoda *Parse()*.

Konkrétní popis komunikace gramatického systému Γ_{cpp}

Gramatický systém Γ_{cpp} je koncipován tak, že instance aktivní doby s komponentou G_1 je jediná za celou dobu analýzy. Právě proto, že G_1 zpracovává tělo programu, je touto komponentou syntaktická analýza započata i ukončena. Instancí aktivní doby G_1 je v průběhu její činnosti předávána aktivita komponentám G_2 a G_3, které zpracovávají matematické výrazy. V rámci matematických výrazů se může vyskytovat syntaktická konstrukce volání funkce, pro jejíž zpracování předává komponenta G_2 nebo G_3 aktivitu komponentě G_4. A naopak součástí konstrukce volání funkce jsou argumenty volání zpracovávané jako matematické výrazy a aktivita tedy opět musí být předána komponentě G_2. Konstrukce volání funkce je v komponentách G_2 a G_3 prezentována jako literál představující návratou hodnotu volané funkce, se kterým může být zpracování výrazu korektně dokončeno.

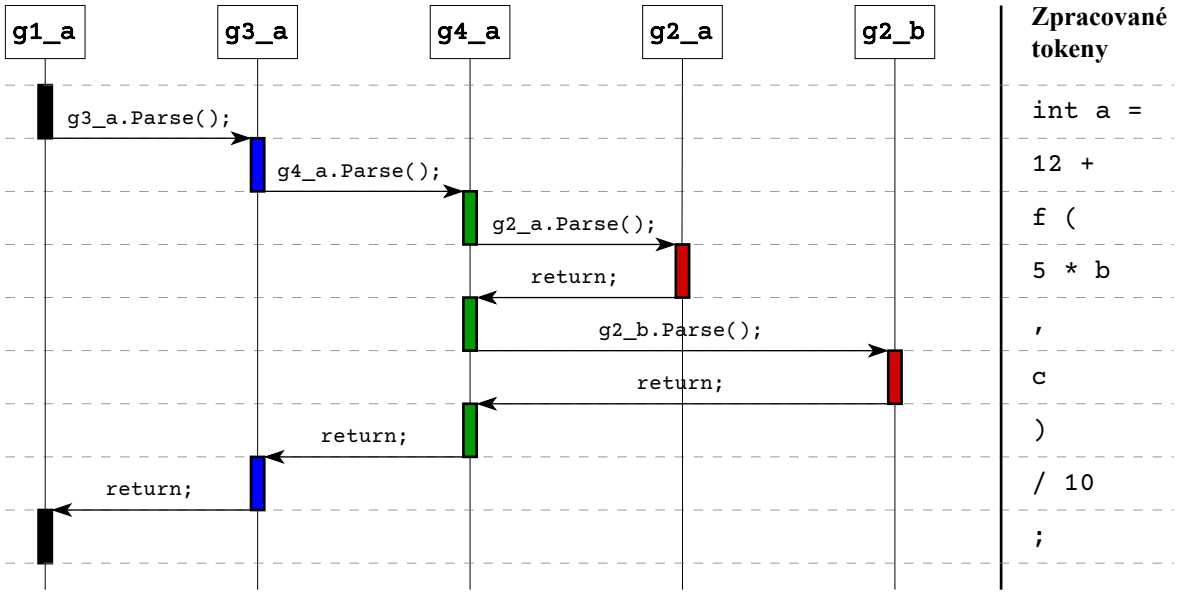
Příklad 6.1. Uvažujme gramatický systém Γ_{cpp}. Mějme řetězec x představující fragment zdrojového programu, kde $x \in L(\Gamma_{cpp})$ a x se rovná

```
int a = 12 + f( 5 * b , c ) / 10 ;
```

Barevné značení v zápisu řetězce x symbolizuje, kterou komponentou bude určitý podřetězec řetězce x zpracován. Přechod mezi barvami v sekvenci terminálů zase značí předání aktivity mezi komponentami. Barvy reprezentují zpracování podřetězců jednotlivými komponentami následovně:

- Komponenta G_1 – **černá** barva,
- Komponenta G_2 – **červená** barva,
- Komponenta G_3 – **modrá** barva,
- Komponenta G_4 – **zelená** barva.

Dále mějme instance aktivní doby `g1_a` pro komponentu G_1, `g2_a` a `g2_b` pro komponentu G_2, `g3_a` pro komponentu G_3 a `g4_a` pro komponentu G_4. Pro upřesnění, jak budou instance aktivní doby využity a jak přesně předávání řízení mezi těmito instancemi probíhá, je zpracování řetězce x vizualizováno pomocí sekvenčního diagramu na obrázku 6.1.



Obrázek 6.1: Sekvenční diagram vyobrazující komunikaci mezi instancemi aktivní doby hybridního CD gramatického systému s vnitřním řízením Γ_{cpp} při zpracování řetězce `int a = 12 + f(5 * b, c) / 10 ;`.

Zotavení po chybě

Při tvorbě syntaktického analyzátoru obecně platí, že by v rámci jednoho spuštění programu analyzátoru mělo být nalezeno co nejvíce syntaktických chyb (v ideálním případě všechny, což není vždy úplně možné). Aby toho mohlo být dosaženo, je třeba analyzovat celý zdrojový soubor. Nicméně algoritmy jednotlivých typů analýzy (viz algoritmy 6.1, 6.2 a 6.3) nedefinují chování, které by umožnilo analyzátoru v činnosti po nálezu první chyby pokračovat. Zpravidla tak dochází k vyvolání výjimky s informací o chybě a následně k ukončení programu.

Aby bylo možné po objevení chyby v činnosti pokračovat, je třeba stanovit postup definující kroky vedoucí k obnovení SA. Takový proces se nazývá *zotavení po chybě* a obvykle bývá realizován *přeskočením* určitého počtu tokenů na vstupu nebo *upravením* zásobníku odebráním či přidáním symbolů. Implementace zotavení po chybě u gramatického systému vyžaduje stanovení různých postupů zotavení minimálně pro každý typ syntaktického analyzátoru (LL, precedenční a SLR), jelikož pracují na jiném principu a zavedení jedné obecné metody tak není reálné.

Zotavení po chybě pro LL syntaktický analyzátor

Pro komponenty G_i, kde $i = \{1, 4\}$, jež jsou založené na principu LL, bylo zvoleno zotavení po chybě v tzv. *panickém režimu*. Panický režim pracuje s množinami $First(X)$ a $Follow(X)$ pro všechna $X \in N_i$, na jejichž základě je možné určit, kdy se může analyzátor po úpravě zásobníku vrátit k běžné činnosti. Syntaktická analýza v panickém režimu probíhá následovně (viz [8]).

- Je-li $a \in T_i$ terminál na vrcholu zásobníku, $b \in T_i$ terminál na vstupu a $a \neq b$, je symbol a z vrcholu zásobníku odebrán a SA pokračuje v běžné činnosti.

- Je-li $A \in N_i$ neterminál na vrcholu zásobníku, $b \in T_i$ je aktuální terminál na vstupu, t je LL tabulka analýzy aktivní komponenty a platí, že $t[A, b] = chyba$, nechť jsou načítány následující vstupní symboly, dokud neplatí, že $b \in First(A)$ nebo $b \in Follow(A)$. V případě, že $b \in First(A)$, pokračuje SA v běžné činnosti. Jestliže $b \in Follow(A)$, je symbol A z vrcholu zásobníku odebrán a SA pokračuje v běžné činnosti.

Je nutné podotknout, že existují i efektivnější metody zotavení po chybě než je panický režim. Jedna z těchto metod například definuje postupy zotavení pro konkrétní chybová políčka LL tabulky analýzy, tedy pro konkrétní chybové situace.

Zotavení po chybě pro syntaktické analyzátory zdola nahoru

Pro komponenty G_i, kde $i = \{2, 3\}$, které vychází z principu syntaktické analýzy zdola nahoru, je zotavení po chybě řešeno simulací dokončení aktivity komponenty pomocí derivačního režimu t. To v praxi znamená, že jsou přeskočeny všechny vstupní terminály a, kde $a \in T_i$. Při prvním případu, kdy $a \notin T_i$, je proveden rekurzivní návrat k předchozí komponentě. Ve zkratce je při objevení první syntaktické chyby přeskočena analýza zbytku chybného matematického výrazu, který měl být původně komponentou G_i zpracován. To však prakticky nevadí, jelikož jednotlivé výrazy obvykle představují krátký úsek zdrojového programu, ve kterém je výskyt většího množství chyb méně pravděpodobný.

6.4 Vstupy/výstupy aplikace a testování

Vstupem do programu, který je produktem této práce, je soubor se zdrojovým textem, nad kterým má být provedena lexikální a syntaktická analýza. Očekává se, že obsah vstupního souboru bude jazykem $L(\Gamma_{cpp})$ neboli podmnožinou jazyka *C++* definovanou v podkapitole 5.3. *Výstupem* je výpis do *konzole* s informací o úspěchu či neúspěchu syntaktické analýzy. Výpis po úspěšné/neúspěšné analýze vypadá následovně.

```
The parsing of the file '[path]' has succeeded / failed.
```

V případě neúspěchu je kromě informace o neúspěchu na výstupu uveden také kompletní seznam všech chyb včetně podrobností o chybách. Pro každou chybu je specifikováno, o jaký typ chyby se jedná (lexikální/syntaktická), kterou komponentou byla chyba odhalena, krátký popis chyby a řádek a pozici na řádku, kde se chyba vyskytuje.

Příklad 6.2. Mějme zdrojový soubor `errorExample.cpp` (viz algoritmus 6.4), který obsahuje právě tři syntaktické chyby. Na řádku 1 se ve výrazu nelegálně vyskytují dva operandy bezprostředně za sebou a příkaz definice proměnné `i` není ukončen středníkem a na řádku 4 se ve výrazu bezprostředně za sebou vyskytují dva operátory.

Algoritmus 6.4: Soubor errorExample.cpp

```
1: int i = 12 10
2:
3: void idFunc(){
4:     i = 10 - + 2;
5: }
```

Po syntaktické analýze by výstup aplikace vypadal následovně:

```
The parsing of the file 'errorExample.cpp' has failed.

Error messages:
  -> G3 - SYNTAX ERROR: Parsing table error [15, CONST_INT]
       Error detected on line '1' and position '12'.
  -> G1 - SYNTAX ERROR: 'SEMICOLON' expected, but got 'VOID [void]'
       Error detected on line '3' and position '1'.
  -> G2 - SYNTAX ERROR: Reduction - No matching rule for sequence: G2_EXP MINUS
       Error detected on line '4' and position '14'.
```

Spuštění aplikace s argumentem `--show`

Program je možné spustit s volitelným argumentem `--show`, který na výstup vypíše sled událostí, které nastávají v průběhu celé syntaktické analýzy. Pro právě aktivní komponentu G_i, kde $i = \{1, 2, 3, 4\}$, se mezi zaznamenávané události řadí následující:

- *Předání aktivity* mezi komponentami, kde je předání řízení následující komponentě G_j, kde $j = \{1, 2, 3, 4\}$ a $i \neq j$, značeno jako $G_i \rightarrow G_j$ a rekurzivní návrat k předchozí komponentě G_h, kde $h = \{1, 2, 3, 4\}$ a $i \neq h$, je značen jako $G_h \leftarrow G_i$.
- *Gramatické pravidlo* použité pro rozvoj neterminálu ve větné formě (v případě metody shora dolů) nebo pro redukci řetězce symbolů ve větné formě na neterminál (v případě metody zdola nahoru) zapisované jako $p : A \rightarrow x$, kde p je číslo gramatického pravidla, $A \in N_i$ a $x \in (N \cup T_i)^*$.
- *Chyba* spolu s podrobnými informacemi, jež se vyskytla během analýzy.

Příklad 6.3. Uvažujme zdrojový soubor `showExample.cpp` (viz algoritmus 6.5). Soubor obsahuje jediný příkaz (definici proměnné), který je po syntaktické stránce validní.

Algoritmus 6.5: Soubor showExample.cpp

```
1: int i = !true * 10;
```

Spuštění syntaktické analýzy s argumentem `--show`, vypíše na výstup sekvenci událostí následujícím způsobem:

```
Active component: G1 - LL body
  0:              G1_START -> G1_PROGRAM_MAIN string_terminator
  3:      G1_PROGRAM_MAIN -> G1_VAR_DEF semicolon G1_PROGRAM_MAIN
  20:           G1_VAR_DEF -> G1_DTYPE id G1_ASSIGN
  45:             G1_DTYPE -> int
  21:            G1_ASSIGN -> assign G3_S_START

--------------------- G1 --> G3 ---------------------
```

```
Active component: G3 - SLR
  32:                G3_I -> const_true
  26:                G3_P -> G3_I
  23:                G3_N -> G3_P
  22:                G3_N -> not G3_N
  19:                G3_T -> G3_N
  27:                G3_I -> const_int
  26:                G3_P -> G3_I
  23:                G3_N -> G3_P
  16:                G3_T -> G3_T multiply G3_N
  15:                G3_E -> G3_T
  12:               G3_C2 -> G3_E
  7:                G3_C1 -> G3_C2
  4:                 G3_A -> G3_C1
  2:                 G3_O -> G3_A

---------------------- G1 <-- G3 ----------------------

Active component: G1 - LL body
  6:      G1_PROGRAM_MAIN -> eof

============================================================

The parsing of the file 'showExample.cpp' has succeeded.
```

Testování aplikace

Funkčnost aplikace (převážně syntaktického analyzátoru) byla testována pomocí .NET nástroje *xUnit.net*. Konkrétní .NET projekt, který obsahuje testy i testovací soubory, je k nalezení v přiloženém paměťovém médiu (viz příloha C ve složce `src/GrammarSystemSA.Tests/`). Aplikace byla testována na přibližně 120 vstupních souborech obsahujících zdrojový text v jazyce, který přijímá implementovaný gramatický systém Γ_{cpp}. Testování probíhalo jak na syntakticky správných souborech, pro které je předpokládán úspěch syntaktické analýzy, tak na souborech syntakticky chybných, u kterých je naopak očekáváno odhalení chyby. Správnost fungování aplikace byla ověřena na platformách *Windows 10*, *Fedora 40* a *Ubuntu 23.10*.

V přiložených souborech jsou také ve složce `app/Windows/ExampleInputFiles/` dostupné ukázkové vstupní soubory pro syntaktickou analýzu (syntakticky správné i chybné). Spuštění aplikace s argumentem `--show` nad těmito programy může posloužit jako zřetelný příklad toho, jak gramatický systém pracuje, popřípadě jak probíhá zotavení po chybě v jednotlivých komponentách.

Kapitola 7

Závěr

Cílem této práce bylo navrhnout gramatický systém a následně jej aplikovat v rámci syntaktického analyzátoru. Tohoto cíle bylo dosaženo a řešení se navíc podařilo obohatit o několik prvků nad rámec zadání. Podrobně byla studována oblast gramatických systémů, především systémů kooperačně distribuovaných (CD) a jejich specifických typů, z odborných literárních pramenů, které obsahovaly klíčové informace pro řešení této práce.

Na základě nabytých vědomostí z předchozího studia byl zaveden nový unikátní typ CD gramatických systémů, který přebírá některé charakteristické rysy z hybridních CD gramatických systémů a z CD gramatických systémů s vnitřním řízením. Významnou vlastností touto prací zavedeného typu gramatických systémů je možnost stanovit počáteční i ukončovací podmínky pro každou komponentu gramatického systému, které jednoznačně definují stav syntaktického analyzátoru, při němž dojde k předání řízení mezi komponentami. Zmíněná vlastnost jde ruku v ruce s aplikovatelností těchto gramatických systémů právě kvůli determinismu, který je do problematiky vnesen na rozdíl od již existujících řešení. Je totiž jasné, v jaký okamžik mají jednotlivé komponenty začít pracovat a kdy mají řízení předat komponentě jiné. V praxi je poté předání činnosti postaveno na principu rekurzivního volání instancí aktivity dané komponenty. Konkrétní gramatický systém nově představeného typu se skládá ze čtyř komponent a přijímá jazyk, který je podmnožinou jazyka C++. Dvě jsou typu LL pro analýzu těla programu a volání funkce, třetí je založená na precedenční syntaktické analýze a zpracovává matematické výrazy s přiřazením a čtvrtá je typu SLR a slouží pro analýzu matematických výrazů bez přiřazení.

Aplikovatelnost je demonstrována konzolovou .NET aplikací, které dala tato práce vzniknout. Tato aplikace provádí lexikální a syntaktickou analýzu nad vstupním souborem na základě konkrétního gramatického systému definovaného pro tuto práci, který přijímá podmnožinu jazyka C++. Výstupem této aplikace je informace o tom, zda proběhla lexikální i syntaktická analýza bezchybně nebo naopak s chybou. Při neúspěchu je na výstup vypsán seznam všech nalezených chyb s jejich krátkým popisem. Implementace je obohacena o funkcionalitu zotavení po chybě, takže je možné v rámci jednoho běhu syntaktické analýzy detekovat více než jednu chybu. Dalším rozšířením je lokalizace chyb v rámci vstupního souboru informující uživatele o konkrétním místě výskytu jednotlivých chyb. Volitelně navíc může uživatel nechat na výstup vypsat průběh syntaktické analýzy, tedy kdy a jakým způsobem probíhalo předání aktivity mezi komponentami gramatického systému a jaká gramatická pravidla byla aplikována při konkrétní aktivitě určité komponenty.

Jako autorovi mi práce v oboru teoretické informatiky rozvinula schopnost matematicky přemýšlet nad praktickými problémy z oblasti formálních jazyků a překladačů. To se týká také opačné situace, kdy bylo nutné hledat správné datové struktry a algoritmy pro

implementaci pouze matematicky popsaného problému. Celkově mě tedy naučila vnímat rozdíly mezi teoretickou a čistě praktickou informatikou a hledat mezi nimi kompromisy. Stejně tak mě ale naučila rozpoznávat jejich společné rysy. Za cennou zkušenost také považuji příležitost pracovat na dlouhodobém projektu většího rozsahu, jako je právě závěrečná práce.

Vizí do budoucna je zobecnit implementaci aplikace takovým způsobem, že by aplikace mohla provádět syntaktickou analýzu založenou na libovolném gramatickém systému nově zavedeného typu. Představa je taková, že by stačilo aplikaci poskytnout potřebné informace o gramatickém systému a jeho komponentách (typy komponent, množiny gramatických pravidel, tabulky analýzy, ...). Aktuálně jsou některé části implementace přímo vázané na konkrétní komponenty zmíněného gramatického systému a změny ve struktuře gramatického systému by do implementace musely být promítnuty úpravou zdrojových souborů aplikace. Na to by následně mohlo navázat doplnění funkcionalit, které by umožnily na základě znalosti množiny gramatických pravidel a typu jednotlivých komponent vytvářet konkrétní tabulky analýzy, množiny terminálů a neterminálů a mapovací funkce. Promítnout změnu gramatických pravidel do implementace nebo přidávat nové komponenty, by tak bylo ještě více zjednodušeno. Řešení by bylo také možné rozšířit o implementaci dalších částí překladače. To by mělo být proveditelné bez větších zásahů do aktuální implementace, jelikož lexikální i syntaktický analyzátor jsou koncipovány tak, aby na ně bylo možné tabulkou symbolů a sématickou analýzou navázat.

Literatura

[1] *Podklady předmětu Moderní teoretická informatika* [online]. Fakulta informačních technologií VUT v Brně, 2007 [cit. 2024-03-22]. Dostupné z: http://www.fit.vutbr.cz/~meduna/work/doku.php?id=lectures:phd:tid:tid.

[2] *Podklady předmětu Formální jazyky a překladače* [online]. Fakulta informačních technologií VUT v Brně, 2017 [cit. 2024-03-11]. Dostupné z: https://www.fit.vutbr.cz/study/courses/IFJ/public/materials/.

[3] *Podklady předmětu Výstavba překladačů* [online]. Fakulta informačních technologií VUT v Brně, 2022 [cit. 2024-03-18]. Dostupné z: https://www.fit.vutbr.cz/study/courses/VYPa/public/materials/.

[4] Aho, A. V., Lam, M. S., Sethi, R. a Ullman, J. D. *Compilers: Principles, Techniques, & Tools*. 2. vyd. Boston: Pearson Education, Inc, 2007. ISBN 0-321-48681-1.

[5] cppreference.com. C Operator Precedence. *Cppreference.com* [online]. cppreference.com, 16. dubna 2012. Revidováno 31. 7. 2023 [cit. 2024-03-27]. Dostupné z: https://en.cppreference.com/w/c/language/operator_precedence.

[6] Csuhaj Varjú, E., Dassow, J., Kelemen, J. a Paun, G. *Grammar Systems: A Grammatical Approach to Distribution and Cooperation*. 1. vyd. Amsterdam: Gordon and Breach Science Publishers S.A., 1994. ISBN 2-88124-957-4.

[7] Meduna, A. *Automata and languages: Theory and Applications*. 1. vyd. London: Springer, 2000. ISBN 978-1-85233-074-3.

[8] Meduna, A. *Elements of compiler design*. 1. vyd. Boca Raton: Auerbach Publications, 2008. ISBN 978-1-4200-6323-3.

[9] Meduna, A. *Formal Languages and Computation: Models and Their Applications*. 1. vyd. Boca Raton: Auerbach Publications, 2014. ISBN 978-1-4665-1349-5.

[10] Nextbridge. What Is the Difference Between Programming Language and Natural Language? *Nextbridge* [online]. Nextbridge, 2. června 2023 [cit. 2023-12-26]. Dostupné z: https://nextbridge.com/difference-between-programming-language-and-natural-language/.

[11] Rozenberg, G., Salomaa, A. et al. *Handbook of Formal Languages: Volume 2. Linear Modeling: Background and Application* [online]. 1. vyd. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, duben 2013 [cit. 2023-12-21]. ISBN 978-3-662-07675-0. Dostupné z: https://doi.org/10.1007/978-3-662-07675-0.

[12] The British Standards Institution. *The C++ standard: Incorporating Technical Corrigendum 1.* 2. vyd. Chichester: John Wiley and Sons Ltd, 2003. ISBN 0-470-84674-7.

[13] Češka, M., Vojnar, T., Smrčka, A. a Rogalewicz, A. *Teoretická informatika: TIN* [online]. Studijní text, 1. vyd. Brno: Fakulta informačních technologií VUT v Brně, srpen 2020 [cit. 2024-03-09]. Dostupné z: http://www.fit.vutbr.cz/study/courses/TIN/public/Texty/TIN-studijni-text.pdf.

Příloha A

Komponenty navrženého gramatického systému

Tato příloha prezentuje konkrétní hodnoty množin *terminálů*, *neterminálů* a *gramatických pravidel* jednotlivých komponent gramatického systému

$$\Gamma_{cpp} = (N, T, S, G_1, G_2, G_3, G_4)$$

definovaného v podkapitole 5.2. Kromě zmíněných množin se ke komponentám vážou také jejich *tabulky analýzy*, které je možné nalézt v přiloženém paměťovém médiu, viz C (ve složce `doc/others/parsingTables`).

A.1 Komponenta G_1 – Tělo programu

LL tabulka syntaktické analýzy komponenty G_1 je přiložena na paměťovém médiu, viz soubor `doc/others/parsingTables/LLBodyParsingTable.pdf`. Úplná definice komponenty G_1 je následující.

$$G_1 = (N_1, T_1, S_1, P_1, \pi_1, \rho_1, f_1) \text{ je typu } (a, l),$$

kde

- $N_1 = \{$<prog_main>, <construct_block>, <construct_cont>, <construct>, <element>, <var_def>, <assign>, <value>, <define>, <if_cont>, <else_if_cont>, <func_dtype>, <params>, <params_cont>, <dtype> $\}$,
- $T_1 = \{$**eol**, **;**, **(**, **)**, **{**, **}**, **eof**, **if**, **switch**, **while**, **do**, **for**, **return**, **break**, **continue**, **case**, **default**, **:**, **=**, **#define**, **else**, **void**, **,**, **int**, **float**, **double**, **char**, **string**, **bool**, **id** $\}$,
- S_1 = <prog_main>,
- π_1 není specifikován, jelikož je predikátem typu (a),
- $\rho_1 = (\{AS, S'\})$,
- $f_1 = t$,
- $P_1 = \{$

1: <prog_main> ⟶ <define> **eol** <prog_main> ,

2: <prog_main> ⟶ <var_def> **;** <prog_main> ,

3: <prog_main> ⟶ <func_dtype> **id (** <params> **)** <construct_block> <prog_main>,

4: <prog_main> ⟶ **eof** ,

5: <construct_block> ⟶ **{** <construct_cont> **}** ,

6: <construct_cont> ⟶ <construct> <construct_cont> ,

7: <construct_cont> ⟶ ϵ ,

8: <construct> ⟶ **if (** <element> **)** <construct_block> <if_cont> ,

9: <construct> ⟶ **switch (** <element> **)** <construct_block> ,

10: <construct> ⟶ **while (** <element> **)** <construct_block> ,

11: <construct> ⟶ **do** <construct_block> **while (** <value> **) ;** ,

12: <construct> ⟶ **for (** <element> **;** <value> **;** <value> **)** <construct_block> ,

13: <construct> ⟶ **return** <value> **;** ,

14: <construct> ⟶ **break ;** ,

15: <construct> ⟶ **continue ;** ,

16: <construct> ⟶ **case** *S'* **;** ,

17: <construct> ⟶ **default ;** ,

18: <construct> ⟶ <define> **eol** ,

19: <construct> ⟶ <element> **;** ,

20: <element> ⟶ <var_def> ,

21: <element> ⟶ <value> ,

22: <var_def> ⟶ <dtype> **id** <assign> ,

23: <assign> ⟶ **=** *S'* ,

24: <assign> ⟶ ϵ ,

25: <value> ⟶ *AS* ,

26: <value> ⟶ ϵ ,

27: <define> ⟶ **#define id** *S'* ,

28: <if_cont> ⟶ **else** <else_if_cont> ,

29: <if_cont> ⟶ ϵ ,

30: <else_if_cont> ⟶ **if** **(** <element> **)** <construct_block> <if_cont> ,

31: <else_if_cont> ⟶ <construct_block> ,

32: <func_dtype> ⟶ **void** ,

33: <func_dtype> ⟶ <dtype> ,

34: <params> ⟶ <var_def> <params_cont> ,

35: <params> ⟶ ϵ ,

36: <params_cont> ⟶ **,** <var_def> <params_cont> ,

37: <params_cont> ⟶ ϵ ,

38: <dtype> ⟶ **int** ,

39: <dtype> ⟶ **float** ,

40: <dtype> ⟶ **double** ,

41: <dtype> ⟶ **char** ,

42: <dtype> ⟶ **string** ,

43: <dtype> ⟶ **bool** }

A.2 Komponenta G_2 – Výrazy s přiřazením

Precedenční tabulka syntaktické analýzy komponenty G_2 je přiložena na paměťovém médiu, viz soubor `doc/others/parsingTables/PrecedenceParsingTable.pdf`. Úplná definice komponenty G_2 je následující.

$$G_2 = (N_2, T_2, S_2, P_2, \pi_2, \rho_2, f_2) \text{ je typu } (l, s),$$

kde

- $N_2 = \{AS,\ EXP\ \}$,
- $T_2 = \{=,\ +=,\ -=,\ *=,\ /=,\ \%=,\ ||,\ \&\&,\ ==,\ !=,\ >,\ <,\ <=,\ >=,\ +,\ -,$
  $*,\ /,\ \%,\ \mathbf{i}{+}{+},\ \mathbf{i}{-}{-},\ !,\ {+}{+}\mathbf{i},\ {-}{-}\mathbf{i},$ **const_int**, **const_float**, **const_double**,
  const_char, **const_string**, **const_true**, **const_false**,
  id, **(**, **)** },
- $S_2 = AS$,
- $\pi_2 = (\{AS\})$,
- $\rho_2 =$ (**id** **(**),
- $f_2 = t$,
- $P_2 = \{$

1: $AS \longrightarrow \textbf{id} = EXP$,

2: $AS \longrightarrow \textbf{id} \mathrel{+}= EXP$,

3: $AS \longrightarrow \textbf{id} -= EXP$,

4: $AS \longrightarrow \textbf{id} \mathrel{*}= EXP$,

5: $AS \longrightarrow \textbf{id} \mathrel{/}= EXP$,

6: $AS \longrightarrow \textbf{id} \mathrel{\%}= EXP$,

7: $AS \longrightarrow EXP$,

8: $EXP \longrightarrow EXP \mid\mid EXP$,

9: $EXP \longrightarrow EXP \;\&\&\; EXP$,

10: $EXP \longrightarrow EXP == EXP$,

11: $EXP \longrightarrow EXP \mathrel{!}= EXP$,

12: $EXP \longrightarrow EXP > EXP$,

13: $EXP \longrightarrow EXP < EXP$,

14: $EXP \longrightarrow EXP <= EXP$,

15: $EXP \longrightarrow EXP >= EXP$,

16: $EXP \longrightarrow EXP + EXP$,

17: $EXP \longrightarrow EXP - EXP$,

18: $EXP \longrightarrow EXP \mathrel{*} EXP$,

19: $EXP \longrightarrow EXP \mathrel{/} EXP$,

20: $EXP \longrightarrow EXP \mathrel{\%} EXP$,

21: $EXP \longrightarrow EXP \textbf{i}{+}{+}$,

22: $EXP \longrightarrow EXP \textbf{i}{-}{-}$,

23: $EXP \longrightarrow \mathrel{!} EXP$,

24: $EXP \longrightarrow {+}{+}\textbf{i}\ EXP$,

25: $EXP \longrightarrow {-}{-}\textbf{i}\ EXP$,

26: $EXP \longrightarrow \textbf{const\_int}$,

27: $EXP \longrightarrow \textbf{const\_float}$,

28: $EXP \longrightarrow \textbf{const\_double}$,

29: $EXP \longrightarrow \textbf{const\_char}$,

30: $EXP \longrightarrow$ **const_string** ,

31: $EXP \longrightarrow$ **const_true** ,

32: $EXP \longrightarrow$ **const_false** ,

33: $EXP \longrightarrow$ **id** ,

34: $EXP \longrightarrow$ (AS) ,

35: $EXP \longrightarrow$ (EXP) }

A.3 Komponenta G_3 – Výrazy bez přiřazení

SLR tabulka syntaktické analýzy komponenty G_3 je přiložená na paměťovém médiu, viz soubor `doc/others/parsingTables/SLRParsingTable.pdf`. Úplná definice komponenty G_3 je následující.

$$G_3 = (N_3, T_3, S_3, P_3, \pi_3, \rho_3, f_3) \text{ je typu } (l, s),$$

kde

- $N_3 = \{S',\ O,\ A,\ C1,\ C2,\ E,\ T,\ N,\ P,\ I,\ D\ \}$,
- $T_3 = \{||,\ \&\&,\ ==,\ !=,\ >,\ <,\ <=,\ >=,\ +,\ -,\ *,\ /,\ \%,\ \mathbf{i}{+}{+},\ \mathbf{i}{-}{-},$ **!**, $+{+}\mathbf{i}$, $-{-}\mathbf{i}$, **const_int**, **const_float**, **const_double**, **const_char**, **const_string**, **const_true**, **const_false**, **id**, (,) }$,
- $S_3 = S'$,
- $\pi_3 = (\{S'\})$,
- $\rho_3 =$ (**id** ( ),
- $f_3 = t$,
- $P_3 = \{$

0: $S' \longrightarrow O$,

1: $O \longrightarrow O \ ||\ A$,

2: $O \longrightarrow A$,

3: $A \longrightarrow A$ **&&** $C1$,

4: $A \longrightarrow C1$,

5: $C1 \longrightarrow C1 == C2$,

6: $C1 \longrightarrow C1$ **!=** $C2$,

7: $C1 \longrightarrow C2$,

8: $C2 \longrightarrow C2 > E$,

9: $C2 \longrightarrow C2 < E$,

10: $C2 \longrightarrow C2 <= E$,

11: $C2 \longrightarrow C2 >= E$,

12: $C2 \longrightarrow E$,

13: $E \longrightarrow E + T$,

14: $E \longrightarrow E - T$,

15: $E \longrightarrow T$,

16: $T \longrightarrow T$ ***** N ,

17: $T \longrightarrow T$ / N ,

18: $T \longrightarrow T$ **%** N ,

19: $T \longrightarrow N$,

20: $N \longrightarrow$ **++i** D ,

21: $N \longrightarrow$ **−−i** D ,

22: $N \longrightarrow$ **!** N ,

23: $N \longrightarrow P$,

24: $P \longrightarrow D$ **i++** ,

25: $P \longrightarrow D$ **i−−** ,

26: $P \longrightarrow I$,

27: $I \longrightarrow$ **const_int** ,

28: $I \longrightarrow$ **const_float** ,

29: $I \longrightarrow$ **const_double** ,

30: $I \longrightarrow$ **const_char** ,

31: $I \longrightarrow$ **const_string** ,

32: $I \longrightarrow$ **const_true** ,

33: $I \longrightarrow$ **const_false** ,

34: $I \longrightarrow$ (O) ,

35: $I \longrightarrow D$,

36: $D \longrightarrow$ **id** }

A.4 Komponenta G_4 – Volání funkce

LL tabulka syntaktické analýzy komponenty G_4 je přiložená na paměťovém médiu, viz soubor `doc/others/parsingTables/LLFuncParsingTable.pdf`. Úplná definice komponenty G_4 je následující.

$$G_4 = (N_4, T_4, S_4, P_4, \pi_4, \rho_4, f_4) \text{ je typu } (s, l),$$

kde

- $N_4 = \{$<function_call>, <args>, <args_cont> $\}$,
- $T_4 = \{$**id**, **(**, **)**, **,** $\}$,
- $S_4 =$ <function_call>,
- $\pi_4 =$ (**id** **(**),
- $\rho_4 = (\{AS\})$,
- $f_4 = t$,
- $P_4 = \{$

1: <function_call> $\longrightarrow$ **id** **(** <args> **)** ,

2: <args> $\longrightarrow$ AS <args_cont> ,

3: <args> $\longrightarrow$ ϵ ,

4: <args_cont> $\longrightarrow$ **,** AS <args_cont> ,

5: <args_cont> $\longrightarrow$ ϵ $\}$

Příloha B

Struktura zdrojových souborů .NET projektu

Adresářová struktura .NET projektu `GrammarSystemSA`, jenž implementuje veškerou funkcionalitu lexikálního i syntaktického analyzátoru, má následující podobu:

- `Common/`
  - `Constants.cs`
  - `Nonterminals.cs`
  - `ParsingTableAction.cs`
  - `PushDownType.cs`
  - `ScannerState.cs`
  - `Terminal.cs`
  - `TerminalNonterminal.cs`
- `Components/`
  - `Component.cs`
  - `LLComponent.cs`
  - `PrecedenceComponent.cs`
  - `SLRComponent.cs`
- `Grammars/`
  - `GrammarRule.cs`
  - `GrammarSetsData.cs`
- `Lexical/`
  - `Scanner.cs`
  - `Token.cs`
- `Mappers/`
  - `KeywordTokenMapper.cs`

  - LLBodyNontermIndexMapper.cs
  - LLBodyTermIndexMapper.cs
  - LLFunctionNontermIndexMapper.cs
  - LLFunctionTermIndexMapper.cs
  - PrecedenceTermIndexMapper.cs
  - SLRNonTermIndexMapper.cs
- Parsers/
  - BottomUpParser.cs
  - LLParser.cs
  - Parser.cs
  - ParsingStats.cs
  - PrecedenceParser.cs
  - SLRParser.cs
  - SLRPushdownItem.cs
- Tables/
  - LLParsingTable.cs
  - ParsingTable.cs
  - PrecedenceParsingTable.cs
  - SLRParsingTable.cs
  - SLRTableCell.cs
- Program.cs

Příloha C

Obsah přiloženého paměťového média

Obsah přiloženého paměťového média je tvořen následujícími adresáři a soubory:

- `app/` – adresář obsahující spustitelné soubory a soubory nezbytné pro spuštění
  - `Linux/` – adresář se spustitelnou aplikací pro linuxové distribuce
    - `ExampleInputFiles/` – adresář s ukázkovými vstupními soubory aplikace
    - `TablesData/` – adresář s CSV soubory, které obsahují data tabulek analýzy
    - `GrammarSystemSA` – spustitelný soubor aplikace
  - `Windows/` – adresář se spustitelnou aplikací pro platformu Windows
    - `ExampleInputFiles/` – adresář s ukázkovými vstupními soubory aplikace
    - `TablesData/` – adresář s CSV soubory, které obsahují data tabulek analýzy
    - `GrammarSystemSA.exe` – spustitelný soubor aplikace
- `doc/` – adresář technické zprávy
  - `docSrc/` – adresář obsahující zdrojové soubory technické zprávy
  - `others/` – adresář obsahující schémata a tabulky využité pro řešení práce
  - `xkrick01_BP_TechnickaZprava.pdf` – PDF soubor technické zprávy
- `src/` – adresář aplikace
  - `GrammarSystemSA/` – projekt implementující zdrojové soubory aplikace
  - `GrammarSystemSA.Tests/` – projekt s xUnit testy k aplikaci
  - `InputTestingPrograms/` – adresář se vstupními soubory pro testování
  - `TablesData/` – adresář s CSV soubory, které obsahují data tabulek analýzy
  - `GrammarSystemSA.sln` – řešení (solution) .NET aplikace
- `README.md`