VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

SYNTAKTICKÁ ANALÝZA ZALOŽENÁ NA NĚKOLIKA GRAMATIKÁCH

PARSING BASED ON SEVERAL GRAMMARS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **ONDŘEJ KOUMAR**
AUTHOR

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

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

153846

Ústav: Ústav informačních systémů (UIFS)
Student: **Koumar Ondřej**
Program: Informační technologie
Název: **Syntaktická analýza založená na několika gramatikách**
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

Tato práce se zabývá výzkumem CD gramatických systémů a jejich využití v syntaktické analýze. Cílem je navrhnout syntaktický analyzátor kombinující techniku shora dolů i zdola nahoru pro CD gramatický systém popisující jazyk generovaný LL gramatikou za použití deterministického zásobníkového automatu. Cílů je dosaženo zavedením pozměněné LL tabulky s uspořádanými dvojicemi, které odkazují na pravidla v jednotlivých komponentách. Tento koncept je demonstrován na novém programovacím jazyce Koubp, pro který je implementován syntaktický analyzátor v jazyce C++.

Abstract

This thesis focuses on research of CD grammar systems and their utilization in syntax analysis. The objective is to design a parser that combines both top-down and bottom-up techniques for a CD grammar system that describes a language generated by an LL grammar, using a deterministic pushdown automaton. The goals are achieved by introducing a modified LL table with ordered pairs that reference rules within individual components. This concept is demonstrated on a new programming language called Koubp, for which a parser is implemented in C++.

Klíčová slova

formální jazyk, programovací jazyk, syntaktická analýza, gramatický systém, CD gramatický systém, bezkontextová gramatika, LL gramatika, LL tabulka, abstraktní syntaktický strom

Keywords

formal language, programming language, syntax analysis, grammar system, CD grammar system, context-free grammar, LL grammar, LL table, abstract syntax tree

Citace

KOUMAR, Ondřej. *Syntaktická analýza založená na několika gramatikách.* Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. RNDr. Alexandr Meduna, CSc.

Syntaktická analýza založená na několika gramatikách

Prohlášení

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

.....
Ondřej Koumar
3. května 2024

Poděkování

Chtěl bych poděkovat panu profesoru Medunovi za jeho cenné rady, trpělivost, odborné konzultace a nadšení, které se mnou pro tuto práci sdílel. Také děkuji mým rodičům a přítelkyni za jejich neustálou podporu a všem mým přátelům, kteří se mnou v nejhorších chvílích zašli na pivo nebo zajeli na kolo. V neposlední řadě děkuji spolužákům, kteří poskytli svoje počítače s různými operačními systémy pro zkušební běhy programové části této práce, a tím přispěli k ověření správnosti implementace.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Jednou z oblastí, kterou se zabývá teoretická informatika, je teorie formálních jazyků. Tato oblast využívá jednoduchých konceptů, jako jsou množiny, relace, symboly a řetězce, k popisu konceptů složitějších – například programovacích jazyků. Programovací jazyky jsou kompromisem mezi přirozenými a strojovými jazyky, nicméně od posloupnosti nul a jedniček, kterým počítač rozumí, mají daleko. Jsou ale navrhnuty tak, aby do nul a jedniček byly algoritmicky přeložitelné a zároveň poskytovaly takovou abstrakci, která je člověku, respektive programátorovi, srozumitelná.

Mezi kroky překladu se mimo jiné řadí lexikální, syntaktická a sémantická analýza, generování mezikódu a generování cílového kódu. Syntaktická analýza hraje v procesu překladu klíčovou roli, ne-li nejdůležitější – moderní překladače dokonce často používají paradigma *syntaxí řízeného překladu*. Je to proces, který ověřuje správnost posloupnosti jednotlivých symbolů řetězce, který má syntaktický analyzátor na vstupu. Přesněji řečeno zkontroluje, zda vstupní řetězec mohl být vygenerován modelem, který daný programovací jazyk popisuje.

Nejčastěji používaný model pro popis jazyka je gramatika. Gramatiky jsou čtyř typů a vytváří hierarchii generativní síly podle typu pravidel, které jsou v nich použity. Tuto skutečnost popsal americký filozof a lingvista Noam Chomsky v roce 1956. Tato práce se zabývá pouze jedním typem gramatik, a to jsou gramatiky bezkontextové, také nazývané gramatiky typu 2. Přesněji LL gramatikami, které mají menší generativní sílu než bezkontextové, nicméně jazyky generované těmito modely mohou být přijímány relativně jednoduchými a hlavně deterministickými konstrukcemi, jako je například deterministický zásobníkový automat. Díky tomu je tento typ gramatik v praxi velmi rozšířený pro budování syntaktických analyzátorů. Dokonce existuje několik nástrojů, které syntaktické analyzátory automaticky generují z LL gramatik napsaných ve speciální syntaxi.

Velký trend teoretické informatiky na přelomu 20. a 21. století byl výzkum gramatických systémů. Bylo v zájmu přijít na způsob popsání jazyka několika gramatikami, jejichž spolupráce vyústila ve větší generativní sílu systému oproti samostatným gramatikám stejného typu. Nakonec se uchytily především kooperující distribuované (CD) a paralelní komunikující (PC) gramatické systémy.

Tato práce se zabývá upravením principu syntaktické analýzy jazyka popsaného LL gramatikou pro CD gramatický systém. Cílem je navrhnout CD gramatický systém popisující ekvivalentní jazyk jako LL gramatika a následně implementovat syntaktický analyzátor pomocí deterministického zásobníkového automatu, který tento jazyk přijímá. Pro návrh analyzátoru budou použity množiny $Empty$, $First$, $Follow$ a $Predict$, které se standardně

využívají k návrhu LL tabulky, která je nedílnou součástí syntaktických analyzátorů. Pro výrazy bude využita precedenční analýza.

V Kapitole 2 jsou popsány základní pojmy a principy teorie formálních jazyků, které jsou prerekvizitou k pochopení této práce. Kapitola 3 zavádí nové teoretické koncepty v podobě gramatických systémů jako rozšíření klasických gramatik. Principy syntaktické analýzy jsou popsány v Kapitole 4, která zároveň uvádí algoritmy pro sestavení potřebných množin a tabulek, se kterými syntaktický analyzátor pracuje. Vlastní principy implementace systému na abstraktní úrovni jsou popsány v Kapitole 5. Zároveň se zmiňuje o implementačním jazyce, adresářové struktuře, sestavovacím systému a podobně. Poslední kapitola pak pojednává o testování implementovaného systému.

Kapitola 2

Základy teorie formálních jazyků

Základní přehled matematických znalostí potřebných k pochopení této kapitoly (a tím i celé této práce) je popsán například v [13].

2.1 Abeceda, řetězec a jazyk

Definice v této kapitole založeny na definicích z [2, 15, 19].

Abecedy, řetězce a operace s nimi

Definice 2.1.1. *Abeceda* je neprázdná množina prvků, které se nazývají *symboly* nebo *písmena*.

Konvence 2.1.1. Abeceda se označuje velkými písmeny řecké abecedy; v této práci bude použito výhradně písmeno Σ (*sigma*).

Definice 2.1.2. Nechť Σ je abeceda.
Slovo nebo *řetězec* nad abecedou Σ je konečná posloupnost symbolů z Σ, symbol se může v řetězci několikrát opakovat. Formálně se dá zapsat jako posloupnost

$$x = (a_1, a_2, \ldots, a_n),\ n \geq 0,\ a_i \in \Sigma \text{ pro všechna } i \in \{1, \ldots, n\}\ [1].$$

Speciálním případem je *prázdný řetězec* pro $n = 0$, označuje se písmenem ε.

Konvence 2.1.2. Při zápisu řetězců je jednodušší vynechat závorky a oddělovače symbolů. Proto zápis $(a_1, a_2, \ldots, a_n)$ je ekvivalentní se zápisem $a_1 a_2 \ldots a_n$ [7].

Řetězce v této práci budou označovány malými písmeny abecedy stejně jako symboly. Pro řetězce budou používána písmena z konce abecedy (w, x, y, z a podobně), pro symboly písmena ze začátku abecedy (a, b, c a podobně).

Definice 2.1.3. Nechť Σ je abeceda $x = a_1 a_2 \ldots a_n$ je řetězec nad Σ.
Délka řetězce x je počet pozic se symboly z abecedy Σ v řetězci, značí se $|x|$. Dále $alph(x)$ označuje množinu symbolů, které se v řetězci x vyskytují.

Definice 2.1.4. Nechť Σ je abeceda $x = a_1 a_2 \ldots a_n$ je řetězec nad Σ.
Reverzace řetězce x, $reversal(x)$ nebo také x^R je

$$x^R = x_n x_{n-1} \ldots x_1 x_0.$$

Prázdný řetězec zřejmě bude stejný, jako jeho reverzace.

$$\varepsilon^R = \varepsilon$$

Definice 2.1.5. Nechť Σ je abeceda a w je řetězec nad Σ.
Řetězec z je *podřetězec* řetězce w, jestliže existují řetězce x a y takové, že

$$w = xzy.$$

Řetězec x_1 je prefixem (předponou) řetězce w, jestliže existuje řetězec y_1 takový, že

$$w = x_1 y_1.$$

Řetězec x_2 je sufixem (příponou) řetězce w, jestliže existuje řetězec y_2 takový, že

$$w = y_2 x_2.$$

Je-li $y_1 \neq \varepsilon$, pak x_1 je *vlastní prefix* řetězce w; je-li $y_2 \neq \varepsilon$, pak x_2 je *vlastní sufix* řetězce w.

Definice 2.1.6. Nechť Σ je abeceda.
Množina všech řetězců abecedy Σ určité délky k je Σ^k. $\Sigma^0 = \{\varepsilon\}$ pro libovolnou abecedu.

Příklad 2.1.1. Nechť $\Sigma = \{0, 1\}$ je abeceda.
Všechny řetězce délky 2 jsou 00, 01, 10, 11. Zřejmě tedy $\Sigma^2 = \{00,\ 01,\ 10,\ 11\}$.

Definice 2.1.7. Nechť Σ je abeceda.
Množina všech řetězců nad abecedou Σ je

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$$

a množina všech *neprázdných* řetězců nad abecedou Σ je

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots \text{ nebo } \Sigma^+ = \Sigma^* \setminus \{\varepsilon\}.$$

Definice 2.1.8. Nechť Σ je abeceda a $x = a_1 a_2 \ldots a_i$ a $y = b_1 b_2 \ldots b_j$ jsou řetězce nad Σ.
Konkatenace řetězců, symbolicky zapsána xy, je

$$xy = a_1 a_2 \ldots a_i b_1 b_2 \ldots b_j.$$

Jinými slovy, vznikne nový řetězec spojením x a y hned za sebe. Konkatenace řetězce s prázdným řetězcem v libovolném pořadí nemění původní řetězec.

$$\varepsilon x = x\varepsilon = x$$

Příklad 2.1.2. Nechť $\Sigma = \{0, 1\}$ je abeceda, $x = 011$, $y = 1101$ řetězce nad Σ.
Konkatenace řetězců $xy = 0111101$ a $yx = 1101011$.

Jazyky a operace s nimi

Definice 2.1.9. Nechť Σ je abeceda.
Jazyk L nad abecedou Σ je konečná či nekonečná množina řetězců nad Σ, nebo také $L \subseteq \Sigma^*$.
Řetězce jazyka se nazývají *věty* nebo *slova*.

Množiny $\emptyset$ a $\{\varepsilon\}$ jsou jazyky nad libovolnou abecedou Σ, nicméně $\emptyset \neq \{\varepsilon\}$. $\emptyset$ je jazyk, který neobsahuje žádné řetězce, zatímco $\{\varepsilon\}$ obsahuje právě jeden řetězec, a to prázdný.

Definice 2.1.10. Nechť L_1 a L_2 jsou jazyky nad Σ.
Typické množinové operace mohou jsou použitelné i pro jazyky:

- sjednocení jazyků je definováno jako

$$L_1 \cup L_2 = \{x : x \in L_1 \vee x \in L_2\},$$

- průnik jazyků je definován obdobně jako

$$L_1 \cap L_2 = \{x : x \in L_1 \wedge x \in L_2\},$$

- rozdíl jazyků je

$$L_1 \setminus L_2 = \{x : x \in L_1 \wedge x \notin L_2\}.$$

Pro jazyky existuje speciální operace *konkatenace*, která vychází z konkatenace řetězců:

$$L_1 L_2 = \{xy : x \in L_1 \wedge y \in L_2\}.$$

Definice 2.1.11. Nechť Σ je abeceda a L jazyk nad Σ.
Reverzace jazyka, $reversal(L)$ nebo L^R je

$$L^R = \{x^R : x \in L\},$$

mocnina jazyka L^i je definována jako

$$\begin{aligned} &1) \quad L^0 = \{\varepsilon\}, \\ &2) \quad L^i = LL^{i-1},\ i \geq 0, \end{aligned}$$

iterace jazyka je sjednocení všech mocnin jazyka:

$$L^* = \bigcup_{n \geq 0} L^n,$$

pozitivní iterací rozumíme obyčejnou iteraci bez nulté mocniny:

$$L^+ = \bigcup_{n \geq 1} L^n.$$

Z těchto definic dále vyplývá

$$\begin{aligned} L^* &= L^+ \cup \{\varepsilon\}, \\ L^+ &= L^* L = LL^*, \end{aligned}$$

důkaz lze nalézt například v [19].

2.2 Gramatika

Informace pro tuto kapitolu byly převzaty z [19].

V Kapitole 2.1 byly ukázány obecné a pouze základní principy jazyků. S triviálními způsoby reprezentace jazyka, tedy například výčtem všech vět, si nevystačíme už jen díky tomu, že jazyky často bývají nekonečné. Gramatiky jsou jedním ze způsobů, a to velmi elegantním, jak jazyky reprezentovat.

Používají dva typy symbolů – *nonterminální* symboly a *terminální* symboly. Nonterminální symboly slouží k popisu syntaktických celků jazyka a terminální symboly se shodují se symboly ze vstupní abecedy.

Konvence 2.2.1. Pro nonterminální symboly bude dále v práci používáno zkrácené označení *neterminály* a pro terminální symboly pojem *terminály*.

Konvence 2.2.2. V kontextu práce s gramatikami bude nadále používáno následující označení pro různé typy řetězců:

- malá písmena ze začátku abecedy $(a, b, c, \ldots)$ budou reprezentovat terminály,
- velká písmena ze začátku abecedy $(A, B, C, \ldots)$ budou vyhrazena pro neterminály,
  - výjimkou je symbol S, který je často používán pro označení *startovacího symbolu*,
- malá písmena ze začátku řecké abecedy $(\alpha, \beta, \gamma, \ldots)$ budou označovat řetězce terminálů a neterminálů,
- malá písmena z konce abecedy $(u, v, w, \ldots)$ budou pro řetězce terminálů.

Gramatika představuje generativní systém, ve kterém lze z jistého vyznačeného neterminálu generovat, aplikací *přepisovacích pravidel*, řetězce tvořené neterminálními a terminálními symboly, které nazýváme *větnými formami*. Větné formy, které jsou tvořeny pouze terminálními symboly, reprezentují věty gramatikou definovaného jazyka. Pokud gramatika negeneruje žádnou větu, pak reprezentuje prázdný jazyk.

Přepisovací pravidla jsou jádrem gramatik. Jejich využití spočívá v neustálém přepisování neterminálních symbolů, dokud není vygenerován řetězec terminálů (podrobněji v definicích 2.2.2, 2.2.3 a 2.2.4). Každé pravidlo je ve tvaru uspořádané dvojice (α, β) řetězců, kdy se při aplikaci tohoto pravidla podřetězec α nahradí řetězcem β. Podmínkou je, aby α obsahoval alespoň jeden neterminál.

Definice 2.2.1. Gramatika G je čtveřice $G = (N, T, P, S)$, kde

- N je konečná množina neterminálů,
- S je konečná množina terminálů,
- P je konečná relace $R \subseteq (N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$,
- S je počáteční (výchozí, startovací) symbol gramatiky.

Prvek $(\alpha, \beta) \in P$ (*přepisovací pravidlo*) bývá zjednodušeně zapisováno jako $\alpha \to \beta$.

V následujícím příkladě je ukázka gramatiky v obecném tvaru.

Příklad 2.2.1.

$$G = (\{A, S\},\ \{0, 1\},\ P,\ S)$$
$$P = \{S \to 0A1,\ 0A \to 00A1, A \to \varepsilon\}$$

Derivační krok

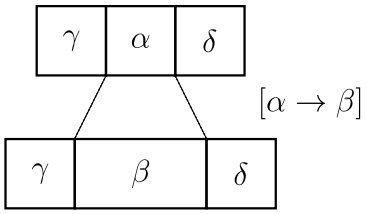
Myšlenkou derivačního kroku je přepsat aktuální řetězec na řetězec nový za použití přepisovacího pravidla $p \in P$.

Definice 2.2.2. Nechť $G = (N, T, P, S)$ je gramatika, $\lambda, \mu \in (N \cup T)^*$ a $p = \alpha \to \beta \in P$. Mezi řetězci λ a μ platí binární relace $\underset{G}{\Rightarrow}$, nazývaná *přímá derivace*, jestliže můžeme řetězce λ a μ vyjádřit ve tvaru

$$\lambda = \gamma\alpha\delta$$
$$\mu = \gamma\beta\delta$$

kde γ a δ jsou libovolné řetězce z $(N \cup T)^*$. Říkáme také, že gramatika G provádí *derivační krok* z $\gamma\alpha\delta$ do $\gamma\beta\delta$, respektive z λ do μ.

To znamená, že pokud lze přepsat řetězec $\lambda = \gamma\alpha\delta$ na řetězec $\mu = \gamma\beta\delta$ za použití pravidla $\alpha \to \beta$, potom mezi řetězci α a β je relace přímé derivace, zapsáno $\alpha \Rightarrow \beta$.



Obrázek 2.2.1: Ilustrace derivačního kroku za použití pravidla $\alpha \to \beta$.

Sekvence derivačních kroků

Definice 2.2.3. Nechť $G = (N, T, P, S)$ je gramatika a $\lambda, \mu \in (N \cup T)^*$.
Mezi řetězci λ a μ platí relace $\Rightarrow^+$ nazývaná *derivace*, jestliže existuje posloupnost přímých derivací $\chi_{i-1} \Rightarrow \chi_i, i \in \{1, \ldots, n\}, n \geq 1$ taková, že platí

$$\lambda = \chi_0 \Rightarrow \chi_1 \Rightarrow \ldots \Rightarrow \chi_{n-1} \Rightarrow \chi_n = \mu.$$

Tato posloupnost se nazývá *derivace délky n*. Platí-li $\lambda \Rightarrow \mu$, pak řetězec μ lze *generovat* z řetězce λ, nebo také μ je *derivovatelný* z λ v gramatice G. Relace $\Rightarrow^+$ je tranzitivním uzávěrem relace přímé derivace $\Rightarrow$. Symbolem $\Rightarrow^n$ se značí n-tá mocnina přímé derivace $\Rightarrow$.

Jinými slovy, pokud řetězec λ derivuje řetězec χ v nenulovém počtu kroků a zároveň χ derivuje řetězec μ v nenulovém počtu kroků, pak je zřejmé, že λ derivuje μ v nenulovém počtu kroků, zapsáno $\lambda \Rightarrow^+ \mu$.

Definice 2.2.4. Nechť $G = (N, T, P, S)$ je gramatika a $\lambda, \mu \in (N \cup T)^*$.
Jestliže v G platí pro řetězce λ a μ relace $x \Rightarrow^+ y$ nebo identita $\lambda = \mu$, pak $\lambda \Rightarrow^* \mu$. Relace $\Rightarrow^*$ je reflexivním a tranzitivním uzávěrem relace přímé derivace $\Rightarrow$.

Reflexivní uzávěr relace přímé derivace $\Rightarrow$ znamená, že řetězec přímo derivuje sám sebe v nula krocích. Také je možné říci, že se nepoužije žádné pravidlo k přepsání řetězce na sebe sama, zapsáno $\lambda \Rightarrow^0 \lambda$.

2.3 Chomského hierarchie gramatik

Informace k následující kapitole převzaty z [19].

Gramatiky se dělí do čtyř skupin podle tvaru přepisovacích pravidel. Označují se jako typ 0, typ 1, typ 2, typ 3, respektive neomezené gramatiky, kontextové gramatiky, bezkontextové gramatiky a pravé lineární gramatiky. Tyto gramatiky generují příslušné jazyky L_i, $i \in \{0, 1, 2, 3\}$, i je příslušné s typem gramatiky. Platí $L_3 \subseteq L_2 \subseteq L_1 \subseteq L_0$.

Neomezená gramatika

Gramatika typu 0 obsahuje pravidla v nejobecnějším tvaru, shodným z Definice 2.2.1.

$$\alpha \to \beta,\ \alpha \in (N \cup T)^* N (N \cup T)^*,\ \beta \in (N \cup T)^*$$

Kontextová gramatika

Gramatika typu 1 obsahuje pravidla ve tvaru:

$$\alpha A \beta \to \alpha \gamma \beta,\ A \in N,\ \alpha, \beta \in (N \cup T)^*,\ \gamma \in (N \cup T)^+$$

nebo $S \to \varepsilon$, pokud se S nevyskytuje na pravé straně žádného pravidla.

Příklad 2.3.1. Příklad kontextové gramatiky:

$$\begin{aligned} G &= (\{A, S\}, \{0, 1\}, P, S) \text{ s pravidly} \\ S &\to 0A \mid \varepsilon \\ 0A &\to 00A1 \\ A &\to 1 \end{aligned}$$

Těmto gramatikám se říká *kontextové*, protože neterminál A může být přepsán řetězcem γ pouze tehdy, když jeho levým kontextem je řetězec α a jeho pravým kontextem řetězec β. Tyto gramatiky nepřipouštějí, aby neterminál byl nahrazen prázdným řetězcem, tedy zakazuje pravidla ve tvaru

$$\alpha A \beta \to \alpha \beta,$$

kromě výše zmiňované výjimky se startovacím symbolem S.

Bezkontextová gramatika

Práce s gramatikami typu 2 je jedním z jader této práce. Definujme tedy tento typ gramatiky podrobněji než ostatní typy.

Definice 2.3.1. *Bezkontextová gramatika* je čtveřice $G = (N, T, P, S)$, kde:

- N, T, S jsou definovány stejně jako v Definici 2.2.1,
- P je množina přepisovacích pravidel ve tvaru $A \to \gamma$, $A \in N$ a $\gamma \in (N \cup T)^*$,
    - je tudíž podmnožinou kartézského součinu $P \subseteq N \times (N \cup T)^*$.

Konvence 2.3.1. Pro označení bezkontextových gramatik bude v textu dále využívána zkratka BKG.

Substituci neterminálu A je možné provést bez závislosti na pravém a levém kontextu, ve kterém je neterminál A uložen. Tyto gramatiky smějí obsahovat pravidla ve tvaru $A \to \varepsilon$.

Příklad 2.3.2. Příklad bezkontextové gramatiky:

$$\begin{aligned} G &= (\{S\}, \{0, 1\}, P, S) \text{ s pravidlem} \\ S &\to 0S1 \mid \varepsilon \end{aligned}$$

Pravá lineární gramatika

Gramatika typu 3 obsahuje pravidla ve tvaru

$$A \to xB \text{ nebo } A \to x,\ A, B \in N,\ x \in T^*.$$

Jediný možný neterminál v pravidlech stojí úplně vpravo, proto *pravá lineární* gramatika. Další možný název je *regulární gramatika*.

Příklad 2.3.3. Příklad pravé lineární gramatiky:

$$\begin{aligned} G &= (\{A, B\}, \{a, b, c\}, P, S) \text{ s pravidly} \\ A &\to aaB \mid ccB \\ B &\to bB \mid \varepsilon \end{aligned}$$

2.4 Konečný automat

Definice související s konečnými automaty převzaty z [5], není-li řečeno jinak.

Definice 2.4.1. *Konečný automat* je pětice

$$M = (Q, \Sigma, R, s, F),$$

kde

- Q je konečná množina stavů,
- Σ je konečná vstupní abeceda, $Q \cap \Sigma = \emptyset$,
- R je konečná relace, $R \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$,
  - je nazývána *množinou pravidel* ve tvaru $qa \to p,\ q, p \in Q,\ a \in \Sigma \cup \{\varepsilon\}$. Jakákoli uspořádaná trojice $(q, a, p) \in R$ je pravidlem, zapsáno $qa \to p$.
- $s \in Q$ je počáteční stav automatu,
- $F \subseteq Q$ je konečná množina koncových stavů.

Konvence 2.4.1. Pro označení konečných automatů bude dále v textu použito zkrácené KA.

U konečných automatů popisujeme jejich *konfiguraci* – ve kterém stavu se nachází a jaký řetězec na vstupní pásce.

Definice 2.4.2. Nechť $M = (Q, \Sigma, R, s, F)$ je KA. Dle autorů v [19] je konfigurace M uspořádaná dvojice

$$(q, w) \in Q \times \Sigma^*.$$

Nechť X_M značí množinu všech konfigurací M.

Pomocí konfigurací můžeme definovat *přechody* KA, které reprezentují jejich výpočetní kroky. Výpočetní krok znamená přechod z jedné konfigurace do druhé za přečtení symbolu ze vstupní pásky.

Definice 2.4.3. Nechť $M = (Q, \Sigma, R, s, F)$ je KA.
Binární relace $\vdash_M$, nazývána přechodem KA M, je

$$\beta \vdash_M \chi,$$

kde $\beta = (q, ax)$, $\chi = (p, x) \in X_M$, a $qa \to p \in R$. Je to binární relace na množině konfigurací – pokud $\beta \vdash_M \chi$, pak $(\beta, \chi) \in X_M \times X_M$. Pokud $a = \varepsilon$, není ze vstupní pásky přečten žádný symbol.

Symboly $\vdash_M^n, \vdash_M^+$ a $\vdash_M^*$ nechť značí příslušně n-tou mocninu pro $n \geq 0$, tranzitivní a reflexivně-tranzitivní uzávěr relace $\vdash_M$ podobně jako u derivačního kroku gramatik v Definici 2.2.2, čímž reprezentují *sekvenci přechodů* KA M. Například pro konfigurace β a χ platí $\beta \vdash_M^+ \chi$ právě tehdy, když

$$\beta = c_0 \vdash_M c_1 \vdash_M \ldots \vdash_M c_{n-1} \vdash_M c_n = \chi,$$

pokud $\beta, \chi, c_1, \ldots, c_{n-1} \in X_M,\ n \geq 1$.

Konvence 2.4.2. Bude-li z kontextu jasné, že se jedná o přechod automatu M, pak bude relace přechodu $\vdash$ psána bez indexu M.

Definice 2.4.4. Nechť $M = (Q, \Sigma, R, s, F)$ je KA.
Jazyk přijímaný M, značen $L(M)$, je

$$L(M) = \{w \in \Sigma^* : sw \vdash^* f,\ f \in F\}.$$

Jsou to řetězce takové, po jejichž zpracování skončí M v koncovém stavu.

Definice 2.4.5. Nechť $M = (Q, \Sigma, R, s, F)$ je KA.
M je *KA bez ε-přechodů*, pokud pro každé pravidlo $qa \to p \in R$ platí, že $a \neq \varepsilon$.

Definice 2.4.6. Nechť $M = (Q, \Sigma, R, s, F)$ je KA bez ε-přechodů.
M je *deterministický KA* právě tehdy, když pro každé $q \in Q$ a každé $a \in \Sigma$ neexistuje více než jedno $p \in Q$ takové, že $qa \to p \in R$. Jinými slovy, z jednoho stavu není možné přejít do několika stavů přečtením stejného symbolu.

Konečný automat může být reprezentován několika způsoby. Zřejmě může být definován výčtem dle Definice 2.4.1, ale existují grafické i výčtové způsoby, ze kterých je funkcionalita KA jednoduše čitelná na první pohled. Nejpopulárnější z nich je tabulka přechodů (*state table*) a diagram přechodů (*state diagram*). Obě dvě reprezentace budou ukázány v následujícím příkladu z [5].

Příklad 2.4.1. Nechť $M = (Q, \Sigma, R, s, F)$ je KA, kde:

$$\begin{aligned} Q &= \{1, 2, 3, 4, 5\}, \\ \Sigma &= \{a, b\}, \\ R &= \{1 \to 2, 1 \to 4, 2a \to 2, 2a \to 3, 3b \to 3, 4b \to 4, 4b \to 5\} \\ s &= 1, \\ F &= \{3, 5\}. \end{aligned}$$

Stav 1 je počáteční a stavy 3 a 5 jsou koncové. Při pohledu na množinu přechodových pravidel může M udělat buď přechod ze stavu 1 do stavu 2 nebo do stavu 3 za přečtení

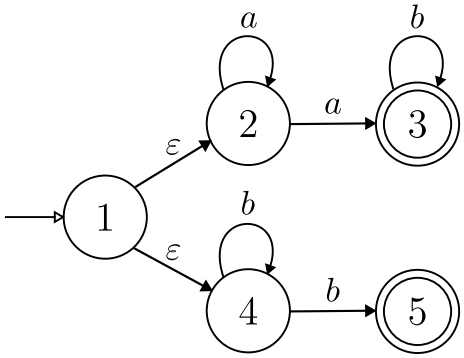
žádného symbolu ze vstupní pásky, respektive za přečtení ε. Ve stavu 2 buď přečte symbol a a přejde do stavu 3, případně za přečtení a může také zůstat ve stavu stejném. Stav 3 je pak koncový a v něm M přijme pouze případné další symboly b. Ze stavu 4 se M dostane do koncového stavu 5 za přečtení b, případně zůstat ve stejném stavu také za přečtení b.

Jak bylo zmíněno, slovní popis a definice se dají přepsat na přehlednější reprezentace, což ilustrují Tabulka 2.4.1 a Obrázek 2.4.1.

Ze všech reprezentací je zřejmé, že KA M určitě nebude deterministicky rozhodovat svoje kroky, což je nepoužitelné pro praxi. Mnohem ideálnější je používat takové automaty, které pro konkrétní vstupní symbol v konkrétním stavu udělají jeden konkrétní krok.

Stav	a	b	ε
1			$2, 4$
2	$2, 3$		
3		3	
4		$4, 5$	
5			

Tabulka 2.4.1: Přechodová tabulka KA M z Příkladu 2.4.1.



Obrázek 2.4.1: Přechodový diagram KA M z Příkladu 2.4.1.

2.5 Zásobníkový automat

Zásobníkový automat je rozšíření konečného automatu, popsaného v Definici 2.4.1, o zásobník. Zásobník slouží jako paměťové médium, na které si automat může ukládat informace v podobě symbolů zásobníkové abecedy. Následující definice převzaty z [5, 19].

Definice 2.5.1. *Zásobníkový automat* (ZA) je sedmice

$$M = (Q, \Sigma, \Gamma, R, s, S, F),$$

kde:

- Q, Σ, s, F jsou definovány stejně jako v Definici 2.4.1,
- Γ je konečná zásobníková abeceda,
- R je konečná relace $R \subseteq \Gamma \times Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^* \times Q$ nazývána množinou pravidel ZA,

- každé pravidlo (A, q, a, w, p) může být zapsáno ve tvaru $Aqa \to wp$, kde $A \in \Gamma$, $q, p \in Q,\ a \in \Sigma \cup \{\varepsilon\},\ w \in \Gamma^*$ [10],

- $S \in \Gamma$ je počáteční symbol na zásobníku.

Konvence 2.5.1. Zásobníkové automaty budou dále značeny zkráceně ZA.

Definice 2.5.2. Nechť $M = (Q, \Sigma, \Gamma, R, s, S, F)$ je ZA.
Konfigurací ZA nazveme trojici

$$(\alpha, q, w) \in \Gamma^* \times Q \times \Sigma^*,$$

kde

- α je obsah zásobníku,
- q je aktuální stav automatu (*řídící jednotky*),
- w je doposud nepřečtená část vstupního řetězce, jehož první symbol je aktuálně pod čtecí hlavou. Pokud $w = \varepsilon$, všechny symboly byly ze vstupní pásky již přečteny.

Nechť X_M reprezentuje množinu všech konfigurací ZA M.

Definice 2.5.3. Nechť $M = (Q, \Sigma, \Gamma, R, s, S, F)$ je ZA.
Přechod ZA je binární relace definována nad množinou konfigurací (podobně jako v Definici 2.4.3) v podobě

$$\beta \vdash \chi,$$

kde $\beta = (uA, q, av),\ \chi = (uw, p, v)$ a $Aqa \to wp \in R$. Symbolem A je reprezentován vrchol zásobníku a symbol a nechť reprezentuje aktuální symbol pod čtecí hlavou. Pokud $w = \varepsilon$, pak se pouze vyjme A ze zásobníku bez náhrady. Relace $\vdash^n$, $\vdash^+$, $\vdash^*$ jsou opět n-tou mocninou relace, tranzitivním a reflexivně-tranzitivním uzávěrem.

Oproti klasickým KA se při přechodu musí pracovat se zásobníkem, ze kterého se vyjme symbol A, namísto něj se vloží symbol w.

Automat M přijímá řetězec právě tehdy, když sekvencí přechodů dosáhne koncového stavu.

Definice 2.5.4. Nechť $M = (Q, \Sigma, \Gamma, R, s, S, F)$ je ZA.
Jazyk přijímaný M je

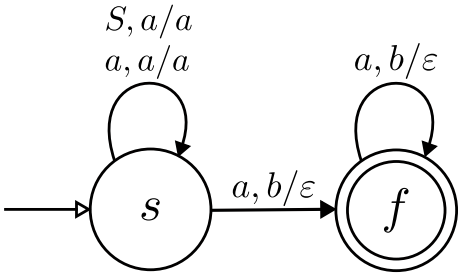
$$L(M) = \{w : w \in \Sigma^*,\ Ssw \vdash^* uf,\ u \in \Gamma^*,\ f \in F\}.$$

Příklad 2.5.1. Nechť $M = (\{s, f\}, \{a, b\}, \{S, a\}, R, s, S, \{f\})$ je ZA a

$$R = \{Ssa \to as,\ asa \to aas,\ asb \to f,\ afb \to f\}.$$

Počáteční konfigurace nechť je $(S, s, aaabbb)$. Přechody M budou vypadat následovně:

$$\begin{aligned}
(S, s, aaabbb) &\vdash (a, s, aabbb) && [Ssa \to as] \\
&\vdash (aa, s, abbb) && [asa \to aas] \\
&\vdash (aaa, s, bbb) && [asa \to aas] \\
&\vdash (aa, f, bb) && [asb \to f] \\
&\vdash (a, f, b) && [afb \to f] \\
&\vdash (\varepsilon, f, \varepsilon) && [afb \to f]
\end{aligned}$$



Obrázek 2.5.1: Diagram přechodů ZA M z příkladu 2.5.1.

Je dobré dodat, že zásobníkové automaty, podobně jako konečné automaty, mohou být také reprezentovány graficky. Díky komplikaci zásobníkem se používá především přechodový diagram. Z výčtu pravidel, přechodového diagramu a kroků přijímání řetězce je vidět, že přijímaný jazyk ZA M je $a^n b^n,\ n \geq 1$.

Rozšířený zásobníkový automat

Rozšířené zásobníkové automaty reprezentují přirozené rozšíření klasických ZA, které na zásobníku mohou pracovat pouze s jeho vrcholem. Rozšířené zásobníkové automaty mohou provádět expanzi symbolů na zásobníku v libovolné hloubce, jinak pracují identicky. Text a definice v této kapitole převzaty z [5, 13], není-li řečeno jinak.

Definice 2.5.5. *Rozšířený zásobníkový automat* (RZA) je sedmice

$$M = (Q, \Sigma, \Gamma, R, s, S, F),$$

kde:

- Q, Σ, s, S, F jsou definovány stejně jako u klasických ZA v Definici 2.5.1,
- Γ je zásobníková abeceda, $\mathbb{N},\ Q$, a Γ jsou navzájem disjunktní, $\Sigma \subseteq \Gamma$ a $\Gamma \setminus \Sigma$ obsahuje speciální symbol $\#$ (*spodní symbol*), který je považován za dno zásobníku,
- R je konečná relace

$$\begin{aligned}&(\mathbb{N} \times Q \times (\Gamma \setminus (\Sigma \cup \{\#\})) \times Q \times (\Gamma \setminus \{\#\})^+) \cup \\ &(\mathbb{N} \times Q \times \{\#\} \times Q \times (\Gamma \setminus \{\#\})^* \{\#\}),\end{aligned}$$

  místo uspořádané pětice $(m, q, A, p, v) \in R$ píšeme $mqA \to pv \in R$ a R nazýváme množinou pravidel.

  - Z definice R lze vidět, že jsou dva typy přechodových pravidel, dají se zapsat ve tvaru

$$\begin{aligned}&mqA \to pv, \\ &mq\# \to pv\# \text{ [8]}.\end{aligned}$$

    O jejich použití a rozdílech je psáno v Definici 2.5.7.

Konvence 2.5.2. Rozšířené zásobníkové automaty budou dále označeny zkratkou RZA.

Přechody RZA pracují, stejně jako přechody ZA, nad konfiguracemi. Konfigurace jsou podobné těm u klasických ZA, nicméně stav zásobníku musí končit symbolem # a zbytek řetězce na zásobníku jej nesmí obsahovat.

Definice 2.5.6. Nechť $M = (Q, \Sigma, \Gamma, R, s, S, F)$ je RZA.
Konfigurací RZA nazveme trojici

$$(q, w, \alpha) \in Q \times \Sigma^* \times (\Gamma \setminus \{\#\})^*\{\#\}.$$

Nechť X_M reprezentuje množinu všech konfigurací RZA M.

Další podobnost je práce s řetězci abecedy Σ a s pravidly při přechodech. Jediná změna od klasických ZA již byla zmíněna na začátku této kapitoly – při přechodech se na vrcholu zásobníku mohou měnit celé řetězce.

Definice 2.5.7. Nechť $M = (Q, \Sigma, \Gamma, R, s, S, F)$ je RZA a $\beta,\ \chi \in X_M$ jeho konfigurace.
M vyjme (anglicky *pops*) ze zásobníku symbol a přejde z konfigurace β do konfigurace χ za přečtení symbolu vstupní pásky, symbolicky zapsáno

$$\beta\ {}_p\vdash \chi,$$

pokud konfigurace jsou ve tvaru $\beta = (q, au, az),\ \chi = (q, u, z),\ a \in \Sigma,\ u \in \Sigma^*,\ z \in \Gamma^*$. M rozšíří (anglicky *expands*) symbol na zásobníku a přejde z konfigurace β do konfigurace χ za přečtení symbolu ze vstupní pásky, symbolicky zapsáno

$$\beta\ {}_e\vdash \chi,$$

pokud konfigurace jsou ve tvaru $\beta = (q, w, uAz),\ \chi = (p, w, uvz)$ a zároveň $mqA \to pv \in R$; $q, p \in Q,\ w \in \Sigma^*,\ A \in \Gamma,\ u, v, z \in \Gamma^*$; m je hloubka symbolu A v zásobníku, respektive u obsahuje $m - 1$ *nevstupních* symbolů (symboly a, pro které platí $\{a : a \in \Gamma \setminus \Sigma\}$). Pro ilustraci, že automat udělá přechod $\beta\ {}_e\vdash \chi$ podle pravidla $mqA \to pv$, píšeme

$$\beta\ {}_e\vdash \chi\ \ [mqA \to pv].$$

M udělá *přechod* z β do χ, psáno

$$\beta \vdash \chi$$

právě tehdy, když M udělá vyjmutí symbolu $\beta\ {}_p\vdash \chi$ nebo expanzi symbolu $\beta\ {}_e\vdash \chi$. Tranzitivní uzávěry ${}_p\vdash^+, {}_e\vdash^+$, reflexivně-tranzitivní uzávěry ${}_p\vdash^*, {}_e\vdash^*$ a n-té mocniny ${}_p\vdash^n, {}_e\vdash^n$ jsou definovány standardně.

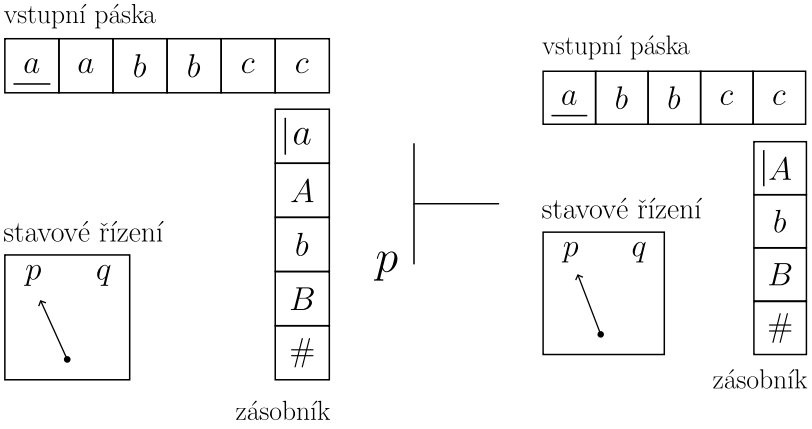
Ilustrace obou typů přechodů jsou zobrazeny na obrázcích 2.5.2 a 2.5.3.

Pokud existuje $n \in \mathbb{N}$ takové, že pro všechna pravidla M platí, že jejich hloubka je maximálně rovna n, pak říkáme, že M je hloubky maximálně n, zapsáno ${}_nM$.

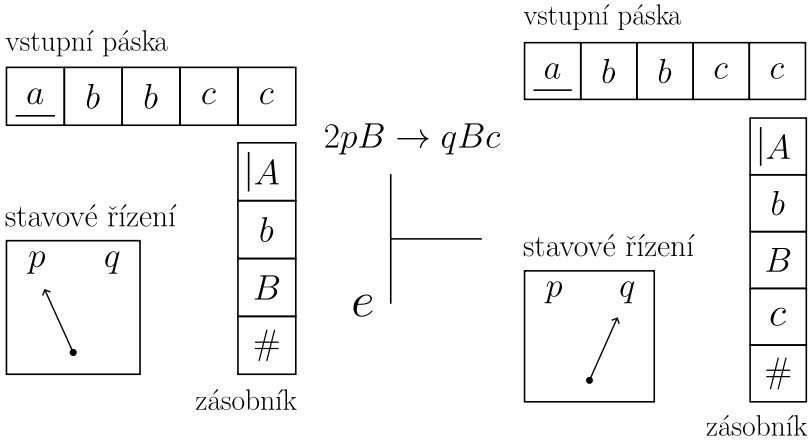
Definice 2.5.8. Nechť ${}_nM = (Q, \Sigma, \Gamma, R, s, S, F)$ je RZA hloubky $n \in \mathbb{N}$.
Jazyk přijímaný automatem M je

$$L({}_nM) = \{w \in \Sigma^* : (s, w, S\#) \vdash^* (f, \varepsilon, \#) \text{ v } {}_nM,\ f \in F\}.$$

Tyto jazyky jsou přijímané vyprázdněním zásobníku a zároveň přechodem do koncového stavu automatu. Dále existují jazyky přijímané RZA pouze vyprázdněním zásobníku, přičemž automat se nemusí dostat do koncového stavu.



Obrázek 2.5.2: Ilustrace p-přechodu RZA (inspirace čerpána v [8]).



Obrázek 2.5.3: Ilustrace e-přechodu RZA (inspirace čerpána v [8]).

Definice 2.5.9. Nechť ${}_nM = (Q, \Sigma, \Gamma, R, s, S, F)$ je RZA hloubky $n \in \mathbb{N}$.
Jazyk přijímaný automatem M vyprázdněním zásobníku je

$$E({}_nM) = \{w \in \Sigma^* : (s, w, S\#) \vdash^* (q, \varepsilon, \#) \text{ v } {}_nM,\ q \in Q\}.$$

Příklad 2.5.2. Nechť ${}_2M = (\{s, q, p, f\}, \{a, b, c\}, \Sigma \cup \{A, S, \#\}, R, s, S, \{f\})$, kde R je

$$\begin{array}{lll} 1sS \to qAA, & 1qA \to fab, & 1fA \to fc, \\ 1qA \to paAb, & 2pA \to qAc. & \end{array}$$

Mějme počáteční konfiguraci $(s, aabbcc, S\#)$. M udělá nejdříve dvakrát expanzi – jednou expanduje startovací symbol na AA, přičemž A není vstupní symbol, takže expanze proběhne podruhé.

$$\begin{array}{rl} (s, aabbcc, S\#) \ {}_e\vdash\ (q, aabbcc, AA\#) & [1sS \to qAA] \\ {}_e\vdash\ (p, aabbcc, aAbA\#) & [1qA \to paAb] \end{array}$$

Na aktuální konfiguraci je vidět, že na vrcholu zásobníku i aktuálně čtený symbol je a. M tento symbol vyjme.

$$(p, aabbcc, aAbA\#) \ {}_p\vdash\ (p, abbcc, AbA\#)$$

Následující krok bude znovu expanze, tentokrát hlouběji v zásobníku.

$$(p, abbcc, AbA\#) \ {}_e\vdash \ (q, abbcc, AbAc\#) \quad [2pA \to qAc]$$

Dále přijímání řetězce probíhá stejným způsobem jako v předešlých krocích.

$$\begin{aligned}
(q, abbcc, AbAc\#) \ {}_e\vdash \ & (q, abbcc, abbAc\#) && [1qA \to fab] \\
{}_p\vdash \ & (f, bbcc, bbAc\#) \\
{}_p\vdash \ & (f, bcc, bAc\#) \\
{}_p\vdash \ & (f, cc, Ac\#) \\
{}_e\vdash \ & (f, cc, cc\#) && [1fA \to fc] \\
{}_p\vdash \ & (f, c, c\#) \\
{}_p\vdash \ & (f, \varepsilon, \#)
\end{aligned}$$

Kapitola 3

Gramatické systémy

Veškeré definice a znalosti použité v této kapitole převzaty z [16, 17, 14], není-li řečeno jinak.

3.1 Kooperující distribuované gramatické systémy

Kooperující distribuovaný (*cooperating distributed*) gramatický systém stupně n je systém gramatik, které mezi sebou sdílejí množinu neterminálů i terminálů a startovací symbol. Spolupracují mezi sebou předáváním řízení derivace aktuálně zpracovávaného řetězce dle pravidel nastaveného *derivačního režimu*.

Definice 3.1.1. Kooperující distribuovaný gramatický systém je $(n+3)$-tice

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

kde:

- N, T, a S jsou definovány stejně jako v Definici 2.3.1,
- P_i je konečná množina pravidel ve tvaru $A \to \alpha$, kde pravidla jsou definována stejně jako v Definici 2.3.1, nazývaná *komponentou* systému, $i \in \{1, \ldots, n\}$,
- i-tá gramatika systému se zapisuje jako $G_i = (N, T, S, P_i)$

Konvence 3.1.1. Pro označení kooperujících distribuovaných gramatických systémů bude dále využito zkrácené *CDGS*, případně *CD gramatické systémy*.

Derivační krok v CDGS

Notace derivačního kroku v CDGS je

$$\alpha \,{}_i\Rightarrow^f \beta.$$

Tento zápis znamená, že řetězec $\alpha \in (N \cup T)^*$ derivuje řetězec $\beta \in (N \cup T)^*$ v i-té komponentě za použití *derivačního režimu* f.

Derivační režimy

Prvním příkladem je režim $*$, který byl v Definici 2.2.4 uveden pro bezkontextovou gramatiku a princip v CD gramatických systémech je podobný. Při použití tohoto derivačního režimu řetězec $\alpha \in (N \cup T)^*$ derivuje řetězec $\beta \in (N \cup T)^*$ v libovolném počtu kroků v i-té komponentě, zapsáno $\alpha \,{}_i\Rightarrow^* \beta$. Je možné předat řízení derivace jiné komponentě i v případě, že ve stejné komponentě lze s derivací stejného řetězce pokračovat.

Podobným příkladem je režim *ukončovací*, který spočívá v nutné derivaci řetězce v dané komponentě, dokud je to možné. Značí se písmenem t. Jsou dvě nutné podmínky, aby β bylo derivovatelné z α v komponentě G_i režimem t.

1. $\alpha \,{}_i\Rightarrow^* \beta$ – v dané komponentě lze posloupností derivačních kroků získat řetězec β z řetězce α,
2. $\beta \,{}_i\not\Rightarrow \gamma$ pro všechna $\gamma \in (N \cup T)^*$ – ve stejné komponentě nelze nalézt další pravidlo, které by derivovalo β.

Jinými slovy, pokud můžeme z aktuálního řetězce v aktuální komponentě odvodit řetězec nový, *musí* se s derivací v této komponentě pokračovat.

Další derivační režimy jsou například

- alespoň k derivací, tedy $\alpha \,{}_i\Rightarrow^{\geq k} \beta$,
- nejvíce k derivací, tedy $\alpha \,{}_i\Rightarrow^{\leq k} \beta$,
- právě k derivací, tedy $\alpha \,{}_i\Rightarrow^{=k} \beta$,

kde $k \in \mathbb{N} \cup \{0\}$ a i symbolizuje i-tou komponentu gramatického systému. Tyto režimy mohou být reprezentovány jako množina, což pomůže definovat další pojmy v následující podkapitole o generovaném jazyce.

Definice 3.1.2. Nechť $k \in \mathbb{N}$ a $*$, t představují derivační režimy.
Potom množina

$$D = \{*, t\} \cup \{\leq k, \geq k, = k\}$$

reprezentuje derivační režimy použitelné v CD gramatických systémech.

Jazyk generovaný CD gramatickým systémem

Než bude definován samotný jazyk, je vhodné definovat pomocnou množinu, která reprezentuje *možné derivace* z řetězců.

Definice 3.1.3. Nechť $\Gamma = (N, T, S, P_1, \ldots, P_n)$.
Potom

$$F(G_j, \beta, f) = \{\beta : \alpha \,{}_j\Rightarrow^f \beta\},\ j \in \{1, \ldots, n\},\ f \in D,\ \alpha \in (N \cup T)^*$$

je množina všech řetězců v derivovatelných z u v j-té komponentě za použití derivačního režimu f.

Definice 3.1.4. Nechť $\Gamma = (N, T, S, P_1, \ldots, P_n)$.
Jazyk generovaný systémem Γ za derivačního režimu f je

$$\begin{aligned} L_f(\Gamma) = \{w \in T^* :\ &\text{existují } \alpha_0, \alpha_1, \ldots, \alpha_m \text{ takové, že } \alpha_k \in F(G_{j_k}, \alpha_{k-1}, f), \\ &k \in \{1, \ldots, m\},\ j_k \in \{1, \ldots, n\},\ \alpha_0 = S,\ \alpha_m = w \text{ pro } m \geq 1\}. \end{aligned}$$

Výsledný řetězec w, který vznikl postupnou derivací startovacího symbolu α_0. Měl několik mezikroků, které jsou reprezentovány řetězci $\alpha_1, \ldots, \alpha_{m-1}$. Každý řetězec α_k, kde $k \in \{1, \ldots, m\}$ byl zderivován z řetězce α_{k-1} v komponentě G_{j_k}, kde $j_k \in \{1, \ldots, n\}$, za derivačního režimu f.

Příklad 3.1.1. Nechť $\Gamma = (\{S, A\}, \{a\}, S, P_1, P_2, P_3)$, kde $P_1 = \{S \to AA\}$, $P_2 = \{A \to S\}$, $P_3 = \{A \to a\}$. Nechť $f = t$, Γ pracuje na ukončovacím derivačním režimu.

Počáteční řetězec je počáteční symbol, tedy S. Je pouze jedna možnost, jak S přepsat, a to použitím pravidla z P_1, $S \to AA$.

$$S \to AA \quad [S \to AA]$$

Díky tomu, že v komponentě P_1 už neexistuje pravidlo, kterým by mohla derivace dále pokračovat, řízení derivace je předáno komponentě jiné. Aktuálně zpracovávaný řetězec je AA. Komponenty P_1 a P_2 mají pravidla pro přepsání neterminálu A. Při použití derivačního režimu t je zřejmé, že celý řetězec bude přepisovat pouze jedna komponenta. Při předání komponentě P_3 je řetězec přepsán na aa,

$$\begin{aligned} AA &\to aA \quad [A \to a] \\ aA &\to aa \quad [A \to a] \end{aligned}$$

komponenta P_2 stejný řetězec přepíše na SS.

$$\begin{aligned} AA &\to SA \quad [A \to S] \\ SA &\to SS \quad [A \to S] \end{aligned}$$

$$\begin{aligned} SS &\to AAS \quad [S \to AA] \\ AAS &\to AAAA \quad [S \to AA] \end{aligned}$$

Ze startovacího symbolu dostaneme buď terminál a nebo dva startovací symboly. To znamená, že pokaždé, kdy se AA přepíše na SS, počet terminálů a se zdvojnásobí. Jazyk generovaný systémem Γ za derivačního režimu t, $L(\Gamma)_t = \{a^{2^n}, n \geq 1\}$.

Klasifikace skupin CD jazyků

CD jazyky jsou jazyky, které jsou generovány CD gramatickými systémy. Tyto jazykové rodiny se rozdělují podle různých typů použitých CDGS. Jejich označení je

$$CD_x^y(f),$$

kde:

- y určuje použití ε-pravidel:
  - $y = \varepsilon$ – ε-pravidla v komponentách jsou povolena,
  - y chybí – ε-pravidla se v komponentách nemohou vyskytovat,
- x je stupeň gramatického systému:
  - $n, n \geq 1$ – gramatický systém může obsahovat maximálně n komponent,
  - ∞ – gramatický systém nemá omezen počet komponent,
- f je derivační režim, $f \in D$.

Příklad 3.1.2. $CD_6^\varepsilon(t)$ je rodina jazyků generována CD gramatickými systémy s maximálně šesti komponentami, povolenými ε-pravidly a pracujícími nad ukončovacím režimem.

Hybridní CD gramatické systémy

Hybridní CD gramatické systémy nabízí možnost definovat derivační režim samostatně pro jednotlivé komponenty systému.

Definice 3.1.5. *Hybridní CDGS* je n-tice $\Gamma = (N, T, S, (P_1, f_1), \ldots, (P_n, f_n))$, kde:

- N, T, P_i, S jsou definovány stejně jako v klasických CDGS v Definici 3.1.1,
- f_i je derivační režim i-té komponenty, $f_i \in D$ pro všechna $i \in \{1, \ldots, n\}$.

Jazyk generovaný těmito gramatickými systémy je velmi podobný jazykům generovaným klasickými CDGS. Jediný rozdíl je v použití derivačního režimu příslušné komponenty, který je v definici označen jako f_{j_k}, místo společného derivačního režimu pro celý gramatický systém.

Definice 3.1.6. Nechť $\Gamma = (N, T, S, (P_1, f_1), \ldots, (P_n, f_n))$.
Jazyk generovaný systémem Γ,

$$L(\Gamma) = \{w \in T^* : \text{existují } \alpha_0, \alpha_1, \ldots, \alpha_n \text{ takové, že } \alpha_k \in F(G_{j_k}, \alpha_{k-1}, f_{j_k}),$$
$$k \in \{1, \ldots, m\}, j_k \in \{1, \ldots, n\}, \alpha_0 = S, \alpha_m = w \text{ pro } m \geq 1\}.$$

Zápis jazykových skupin generovaných hybridními CDGS je

$$X\,CD^{y}_{x,v}(f),$$

kde:

- x, y, f jsou definovány stejně jako u nehybridních CDGS,
- v je nepovinné omezení počtu pravidel v komponentách:
  - m – každá komponenta $P_i, i \in \{1, \ldots, n\}$ obsahuje nejvíce m pravidel, $m \geq 1$,
  - ∞, chybí – maximální počet pravidel pro komponenty není omezen,
- X určuje determinismus gramatického systému:
  - D – gramatický systém je deterministický, tedy pro každé $A \to u, A \to w \in P_i$, $i \in \{1, \ldots, n\}$ platí, že $u = w$. Jinými slovy, pro všechny neterminály platí, že pro jejich přepis neexistují pravidla (ve *všech* komponentách), která by měla různé pravé strany.
  - *nic* – gramatický systém je nedeterministický. To znamená, že v *každé* komponentě existuje pravidlo pro nějaký neterminál, které má různé pravé strany.
  - H – gramatický systém je hybridní a obsahuje alespoň jednu nedeterministickou komponentu a jednu deterministickou komponentu. Nezapisuje se derivační režim, který je určen samostatně pro každou komponentu.

3.2 Paralelní komunikující gramatické systémy

Paralelní komunikující (*parallel communicating*) – PC gramatický systém stupně n je systém gramatik, v němž každá začíná vlastním startovacím symbolem (*axiomem*), které sdílí množinu neterminálů i terminálů a nově množinu komunikačních symbolů. Komunikační

symboly slouží ke komunikaci na vyžádání. Kdykoliv se vyskytnou ve větné formě libovolné komponenty, proběhne *komunikační krok* (také nazýván *c-derivační krok*).

Komunikačních symbolů je stejné množství jako komponent v PC gramatickém systému, $\{Q_1, \ldots, Q_n\}$, kde index symbolu Q_i odkazuje na komponentu P_i [4].

Definice 3.2.1. Paralelní komunikující gramatický systém stupně $n, n \geq 1$ je $(n+3)$-tice

$$\Gamma = (N, K, T, (S_1, P_1), \ldots, (S_n, P_n)),$$

kde:

- N, T jsou definovány stejně jako u bezkontextových gramatik v Definici 2.3.1,
- $K = \{Q_1, \ldots, Q_n\}$ je množina komunikačních symbolů (N, K, T jsou navzájem disjunktní); index i komunikačního symbolu Q_i koresponduje s indexem i-té komponenty,
- $P_i,\ i \in \{1, \ldots, n\}$ je množina pravidel nazývané komponenty, stejně jako u CD gramatických systémů v Definici 3.1.1,
- i-tá gramatika systému je konstrukt $G_i = (N \cup K, T, S_i, P_i),\ i \in \{1, \ldots, n\}$.

Konvence 3.2.1. Pro označení paralelních komunikujících gramatických systémů bude dále využito zkrácené *PCGS* nebo *PC gramatické systémy*.

Derivační kroky v PCGS

V PC gramatických systémech existují dva druhy derivačního kroku, a to g-derivační krok a c-derivační krok. První zmíněný slouží k přímé derivaci řetězce v rámci jedné komponenty bez zásahu komponent jiných a druhý slouží pro vzájemnou pomoc při derivaci řetězce. PC gramatický systém *vždy* preferuje c-derivační krok nad g-derivačním krokem. Ten se provede pokaždé, obsahuje-li alespoň jeden z řetězců $\alpha_i,\ i \in \{1, \ldots, n\}$ alespoň jeden komunikační symbol.

Derivační kroky pracují nad *konfigurací* PCGS. Následující definice převzata z [18].

Definice 3.2.2. Nechť $\Gamma = (N, K, T, (S_1, P_1), \ldots, (S_n, P_n))$ je PC gramatický systém. Potom n-tice

$$(\alpha_1, \ldots, \alpha_n),\ \alpha_i \in (N \cup K \cup T)^*,\ i \in \{1 \ldots, n\}$$

se nazývá *konfigurací* Γ. $(S_1, \ldots, S_n)$ je *počáteční konfigurací* Γ.

Nutná podmínka k proběhnutí libovolného derivačního kroku je $\alpha_1 \notin T^*$. Pokud tato situace nastane, už je vygenerována věta jazyka definovaného PCGS a dále se s kroky nepokračuje. Více v Definici 3.2.

g-derivační krok

Nechť $\Gamma = (N, K, T, (P_1, S_1), \ldots, (P_n, S_n))$ je PCGS a $(\alpha_1, \ldots \alpha_n),\ (\beta_1, \ldots, \beta_n)$ jsou konfigurace Γ. Γ udělá g-derivační krok, formálně zapsáno

$$(\alpha_1, \ldots, \alpha_n)\ {}_g\Rightarrow (\beta_1, \ldots, \beta_n)$$

pokud $alph(\alpha_i) \cap K = \emptyset$ a zároveň:

- $\alpha_i \Rightarrow \beta_i$ v $G_i = (N, K, T, (S_i, P_i))$ – β_i je přímo derivován z α_i v gramatice G_i (komponentě P_i), nebo
- $\alpha_i = \beta_i \in T^*$ – α_i již je řetězcem terminálů

pro všechna $i \in \{1, \ldots, n\}$.

c-derivační krok

Někdy taky nazýván *komunikační krok*. Tento koncept umožňuje gramatikám spolupracovat a vzájemně si mezi sebou měnit vygenerované řetězce pomocí komunikačních symbolů. Algoritmus ukazující princip komunikačního kroku je následující:

Algoritmus 3.2.1 c-derivační krok v PCGS

Vstup: konfigurace $(\alpha_1, \ldots, \alpha_n)$
Výstup: konfigurace $(\beta_1, \ldots, \beta_n)$

1: **for all** $i \in \{1, \ldots, n\}$ **do**
2: $\quad \gamma_i \leftarrow \alpha_i$
3: **end for**
4: **for all** $i \in \{1, \ldots, n\}$ **do**
5: $\quad$ **if** $alph(\alpha_i) \cap K \neq \emptyset$ **and foreach** Q_j **in** α_i: $alph(\alpha_j) \cap K = \emptyset$ **then**
6: $\quad\quad$ **for all** Q_j **in** α_i **do**
7: $\quad\quad\quad \gamma_j \leftarrow S_j$ $\quad \triangleright$ vynecháno, pokud PCGS pracuje na nevracejícím se režimu
8: $\quad\quad\quad$ zaměň Q_j za α_j v α_i
9: $\quad\quad\quad \gamma_i \leftarrow \alpha_i$ $\quad \triangleright$ α_i = řetězec, který vznikl o krok zpět
10: $\quad\quad$ **end for**
11: $\quad$ **end if**
12: **end for**
13: proveď $(\alpha_1, \ldots, \alpha_n) \,{}_c\!\Rightarrow (\beta_1, \ldots, \beta_n)$ s $\beta_i = \gamma_i,\ i \in \{1, \ldots, n\}$

Přímá derivace v PCGS

Definice 3.2.3. Konfigurace $(\alpha_1, \ldots, \alpha_n)$ přímo derivuje konfiguraci $(\beta_1, \ldots, \beta_n)$, zapsáno

$$(\alpha_1, \ldots, \alpha_n) \Rightarrow (\beta_1, \ldots, \beta_n)$$

právě tehdy, když

$$(\alpha_1, \ldots, \alpha_n) \,{}_g\!\Rightarrow (\beta_1, \ldots, \beta_n)$$

nebo

$$(\alpha_1, \ldots, \alpha_n) \,{}_c\!\Rightarrow (\beta_1, \ldots, \beta_n).$$

Jazyk generovaný PCGS

Generování větné formy končí v momentě, kdy první komponenta dosáhne řetězce terminálů a na řetězcích ostatních komponent nezáleží.

Definice 3.2.4. Nechť $\Gamma = (N, K, T, (P_1, S_1), \ldots, (P_n, S_n))$ je PCGS. Jazyk generovaný Γ je stejný jako jazyk generovaný jeho první komponentou:

$$\begin{aligned} L_f(\Gamma) = \{w \in T^* : (S_1, \ldots, S_n) \Rightarrow_f^* (w, \alpha_2, \ldots, \alpha_n), \\ for\ \alpha_i \in \{N \cup T \cup K\},\ i \in \{2, \ldots, n\},\ f \in \{r, nr\}\}, \end{aligned}$$

kde r a nr specifikuje *vracející* nebo *nevracející* PCGS.

Vracející a nevracející se režim

Pokud PC gramatický systém pracuje na vracejícím se režimu, potom komponenty, které v rámci komunikačního kroku poslaly svůj řetězec jiným komponentám, generují řetězec od svého axiomu. Při nevracejícím se režimu komponenty pokračují ve zpracovávání aktuálního řetězce.

Tato skutečnost se projeví na samotném komunikačním kroku, u kterého se vynechá přiřazení axiomu do řetězce γ_j – neprovede se krok na řádku 10 Algoritmu 3.2.1.

Centralizované PCGS

Centralizované PC gramatické systémy mají tu vlastnost, že pouze první komponenta (nazývaná *master*) systému může generovat komunikační symboly a tím žádat ostatní komponenty o řetězce. Řeší jeden z možných případů uváznutí, kdy komponenty v cyklu zavádějí komunikační symboly a donekonečna se provádí stejná sekvence komunikačních kroků.

Definice 3.2.5. Nechť $\Gamma = (N, K, T, (S_1, P_1), \ldots, (S_n, P_n))$ je PCGS. Pokud pouze P_1 může uvést komunikační symboly, formálně

$$P_i \subseteq (N \cup T)^* \times (N \cup T)^* \text{ pro } i \in \{2, \ldots, n\},$$

potom Γ je centralizovaný PC gramatický systém. Jinak je necentralizovaný.

Příklady

V prvním příkladu bude pouze demonstrován princip obou derivačních kroků na systému, který generuje jen velmi omezený jazyk.

Příklad 3.2.1. Nechť $\Gamma = (\{S_1, S_2, S_3\}, \{Q_3\}, \{a, b\}, (S_1, \{S_1 \to Q_3\}), (S_2, \{S_2 \to a\}), (S_3, \{S_3 \to b\}))$ je PCGS. Počáteční konfigurace Γ je zřejmě (S_1, S_2, S_3). Víme, že PC gramatické systémy *vždy* preferují c-krok nad g-krokem, je-li to možné. Nutná podmínka je, aby alespoň jeden z řetězců v aktuální konfiguraci obsahoval alespoň jeden komunikační symbol, což aktuálně není splněno. Γ tedy udělá g-krok.

$$(S_1, S_2, S_3) \,{}_g\!\Rightarrow (Q_3, a, b)$$

Nyní je v konfiguraci komunikační symbol, což indikuje, že bude následovat komunikační krok. Při postupu podle Algoritmu 3.2.1 je postup následující:

1. Zavedení pomocných řetězců $\gamma_1 = Q_3$, $\gamma_2 = a$, $\gamma_3 = b$ podle řádků 4 – 6.
2. Kontrola, zda řetězec α_i, $i \in \{1, \ldots, n\}$ z původní konfigurace obsahuje komunikační symboly (podmínka $alph(\alpha_i) \cap K \neq \emptyset$).
   - V tomto příkladu splňuje podmínku pouze řetězec α_1, který obsahuje Q_3.
3. Pokud α_i obsahuje komunikační symboly Q_j, z každého se přečte index j a proběhne kontrola, zda všechny j-té řetězce konfigurace (α_j) *neobsahují* komunikační symboly. Tato část koresponduje s podmínkou **foreach** Q_j **in** $\alpha_i : alph(\alpha_i) \cap K = \emptyset$ na řádku 8.
   - Řetězec α_1 obsahuje jeden komunikační symbol Q_3, jehož index odkazuje na řetězec α_3 z aktuální konfigurace. Hodnota řetězce α_3 je b, ten žádné další komunikační symboly neobsahuje.

4. Každý Q_j v α_i se nahradí za α_j, pokud prošel podmínkou v předchozím kroku. Zároveň se γ_j nastaví na počáteční symbol j-té komponenty. Tyto kroky jsou v Algoritmu 3.2.1 na řádcích 9 – 12.

   - Komunikační symbol Q_3 bude v α_1 nahrazen řetězcem b z α_3. Dále $\alpha_3 \leftarrow S_3$ a $\gamma_1 \leftarrow \alpha_1$. Aktuálně $\gamma_1 = b$, $\gamma_2 = a$, $\gamma_3 = S_3$.

5. Proveď $(\alpha_1, \ldots, \alpha_n) \,{}_c\!\Rightarrow (\beta_1, \ldots, \beta_n)$, kde $\beta_i = \gamma_i$ pro $i \in \{1, \ldots, n\}$.

   - Nová konfigurace $(\beta_1, \beta_2, \beta_3)$ bude stejná, jako pomocná konfigurace $(\gamma_1, \gamma_2, \gamma_3)$, a to (b, a, S_3).

Γ provede komunikační krok z (Q_3, a, b) do (b, a, S_3).

$$(Q_3, a, b) \,{}_c\!\Rightarrow (b, a, S_3)$$

Další kroky už Γ provádět nebude, protože řetězec generovaný první komponentou je již řetězec terminálních symbolů. V tomto příkladu byla použita sémantika vracejícího se PCGS, nicméně při použití nevracejícího by jazyk vypadal stejně, jen výsledná konfigurace by se lišila v řetězci x_3.

$$L(\Gamma)_r = L(\Gamma)_{nr} = \{b\}$$

Ve druhém příkladu bude demonstrováno několik derivačních kroků a bude se zkoumat výsledný generovaný jazyk.

Příklad 3.2.2. Nechť $\Gamma = (\{S_1, S_1', S_2, S_3\}, K, a, b, c, (S_1, P_1), (S_2, P_2), (S_3, P_3))$, kde:

$$\begin{aligned}
P_1 &= \{S_1 \to abc, S_1 \to a^2b^2c^2, S_1 \to aS_1', S_1 \to a^3Q_2,\\
&\qquad S_1' \to aS_1', S_1' \to a^3Q_2, S_2 \to b^2Q_3, S_3 \to c\},\\
P_2 &= \{S_2 \to bS_2\},\\
P_3 &= \{S_3 \to cS_3\}.
\end{aligned}$$

Komunikační symboly může uvést pouze komponenta P_1, a proto se jedná o *centralizovaný* PC gramatický systém.

Počáteční konfigurace je (S_1, S_2, S_3). Γ může generovat konfiguraci (aS_1', bS_2, cS_3) za použití pravidel $(S_1 \to aS_1', S_2 \to bS_2, S_3 \to cS_3)$.

$$(S_1, S_2, S_3) \Rightarrow (aS_1', bS_2, cS_3)$$

Zřejmě při použití pravidel $(S_1' \to aS_1', S_2 \to bS_2, S_3 \to cS_3)$ n-krát po sobě je možné generovat konfigurace $(a^nS_1', b^nS_2, c^nS_3)$, $n \geq 1$.

$$(aS_1', bS_2, cS_3) \Rightarrow^* (a^nS_1', b^nS_2, c^nS_3)$$

Je vidět, že komponenty P_2 a P_3 neustále používají ta samá pravidla, což je logické vzhledem k jejich kardinalitě. Pro zjednodušení zápisu, při každém g-derivačním kroku bude zmíněno pouze pravidlo komponenty P_1, pravidla komponent P_2 a P_3 budou vždy stejná.

Z konfigurace $(a^nS_1', b^nS_2, c^nS_3)$ se na rozdílný výsledek dá dostat pouze za použití pravidla $S_1' \to a^3Q_2$.

$$(a^nS_1', b^nS_2, c^nS_3) \Rightarrow (a^{n+3}Q_2, b^{n+1}S_2, c^{n+1}S_3)$$

Provedeme c-derivační krok a zaměníme Q_2 za α_2, nový α_2 bude S_2, který je axiomem G_2.

$$(a^{n+3}Q_2, b^{n+1}S_2, c^{n+1}S_3) \Rightarrow (a^{n+3}b^{n+1}S_2, S_2, c^{n+1}S_3)$$

Jediné pravidlo pro symbol S_2 v komponentě P_1 je $S_2 \rightarrow b^2Q_3$.

$$(a^{n+3}b^{n+1}S_2, S_2, c^{n+1}S_3) \Rightarrow (a^{n+3}b^{n+3}Q_3, bS_2, c^{n+2}S_3)$$

Zaměníme Q_3 za α_3, nový α_3 bude S_3, který je axiomem G_3.

$$(a^{n+3}b^{n+3}Q_3, bS_2, c^{n+2}S_3) \Rightarrow (a^{n+3}b^{n+3}c^{n+2}S_3, bS_2, S_3)$$

Pro S_3 je v P_1 také pouze jediné pravidlo, které se může aplikovat, a to $S_3 \rightarrow c$.

$$(a^{n+3}b^{n+3}c^{n+2}S_3, bS_2, S_3) \Rightarrow (a^{n+3}b^{n+3}c^{n+3}, bbS_2, cS_3)$$

α_1 je aktuálně řetězec terminálů, tudíž na něj nelze aplikovat žádné další pravidlo a neobsahuje komunikační symboly. Je tedy generovaným jazykem pro Γ ve vracejícím se režimu. Takto by vypadaly kroky pro nevracející se režim (konfigurace se začnou lišit od výskytu komunikačních symbolů):

$$\begin{aligned}(a^{n+3}Q_2, b^{n+1}S_2, c^{n+1}S_3) &\Rightarrow (a^{n+3}b^{n+1}S_2, b^{n+1}S_2, c^{n+1}S_3)\\ &\Rightarrow (a^{n+3}b^{n+3}Q_3, b^{n+2}S_2, c^{n+2}S_3)\\ &\Rightarrow (a^{n+3}b^{n+3}c^{n+2}S_3, b^{n+2}S_2, c^{n+2}S_3)\\ &\Rightarrow (a^{n+3}b^{n+3}c^{n+3}, b^{n+3}S_2, c^{n+3}S_3)\end{aligned}$$

Je tedy zřejmé, že:

$$L(\Gamma)_r = L(\Gamma)_{nr} = \{a^n b^n c^n,\ n \geq 1\}.$$

Kapitola 4

Syntaktická analýza

Tato kapitola se zabývá syntaktickou analýzou, která provádí kontrolu, zda je vstupní řetězec v jazyce, který je formálně definován. Ostatní části překladačů v této práci nebudou rozebírány, nebude-li to vyžadovat kontext. Celý proces překladu programů je podrobně popsán v [9], odkud bude do této kapitoly převzato největší množství informací.

Syntax jazyka, ve kterém je napsán zdrojový kód, je nejčastěji popsána gramatikou (v praxi nejčastěji bezkontextovou) jazyka s konečnou množinou pravidel. Pomocí těchto pravidel analyzátor zkontroluje, že posloupnost *tokenů*, které jsou na vstupu od lexikálního analyzátoru, je korektní, podle použitých pravidel v gramatice. Při tomto procesu syntaktický analyzátor generuje *derivační strom*, kde každý uzel a jeho potomci reprezentují použité pravidlo. Tento proces může probíhat *shora dolů*, čímž se zabývá Kapitola 4.1, případně *zdola nahoru*, čímž se zabývá Kapitola 4.2. Tyto názvy jsou odvozeny od směru tvorby derivačního stromu, kdy analýza shora dolů postupuje od kořene k listům, analýza zdola nahoru naopak.

Než se začneme zabývat těmito dvěma druhy syntaktické analýzy, pojďme si definovat pojmy *nejlevější a nejpravější derivace* a jejich sekvence.

Definice 4.0.1. Nechť $G = (N, T, P, S)$ je BKG, $r : A \to \alpha \in R$, $u \in T^*$, $\gamma \in (N \cup T)^*$.
G provede nejlevější derivační krok (*leftmost derivation step*) z $uA\gamma$ do $u\alpha\gamma$, podle pravidla r, symbolicky zapsáno

$$uA\gamma \ {}_{lm}\Rightarrow u\alpha\gamma \ [r].$$

G provede nejpravější derivační krok (*rightmost derivation step*) z γAu do $\gamma\alpha u$, podle pravidla r, symbolicky zapsáno

$$\gamma Au \ {}_{rm}\Rightarrow \gamma\alpha u \ [r].$$

Definice 4.0.2. Nechť $G = (N, T, P, S)$ je BKG a $\alpha_0, \alpha_1, \ldots, \alpha_n$ je sekvence řetězců, kde $\alpha_i \in (N \cup T)^*$, $i \in \{1, \ldots, n\}$.
Pokud $\alpha_{j-1} \ {}_{lm}\Rightarrow \alpha_j \ [r_j]$ v G, kde $r_j \in R$, $j \in \{1, \ldots, n\}$, pak G provede sekvenci nejlevějších derivačních kroků (*nejlevější derivaci*) z α_0 do α_n podle $r_1, r_2, \ldots, r_n$, symbolicky zapsáno jako

$$\alpha_0 \ {}_{lm}\Rightarrow \alpha_n \ [r_1 r_2 \ldots r_n].$$

Pokud $\alpha_{j-1} \ {}_{rm}\Rightarrow \alpha_j \ [r_j]$ v G, kde $r_j \in R$, $j \in \{1, \ldots, n\}$, pak G provede sekvenci nejpravějších derivačních kroků (*nejpravější derivaci*) z α_0 do α_n podle $r_1, r_2, \ldots, r_n$, symbolicky zapsáno jako

$$\alpha_0 \ {}_{rm}\Rightarrow \alpha_n \ [r_1 r_2 \ldots r_n].$$

4.1 Syntaktická analýza shora dolů

Uvažujme BKG G. Syntaktický analyzátor pracující shora dolů pro G je reprezentován zásobníkovým automatem M, který je ekvivalentní k G. M na svém zásobníku simuluje nejlevější derivace a tvorbu derivačního stromu pro řetězce generované gramatikou G. Aby tento model fungoval, musí M obsahovat korespondující pravidla pro pravidla G a navíc mít pomocná pravidla pro mazání terminálních symbolů z vrcholu zásobníku, pokud přečte stejný symbol ze vstupní pásky. Princip těchto pomocných pravidel je vlastně stejný, jako *p-přechody* RZA, znázorněn je na Obrázku 2.5.2.

Pokud na vrcholu zásobníku je neterminál A, pak M simuluje nejlevější derivační krok, který by udělala gramatika G pomocí pravidla $r : A \to \alpha \in R$. Provede expanzi pravidla nahrazením neterminálu A z vrcholu zásobníku za $reversal(\alpha)$.

Existují dvě metody syntaktické analýzy shora dolů, a to *prediktivní syntaktická analýza* a *rekurzivní sestup*. Více k těmto metodám je psáno na konci této kapitoly.

Prediktivní množiny

Předpokládejme BKG G a příslušný ZA M. Dále předpokládejme, že existuje neterminál $A \in N$ takový, že pro něj existují dvě pravidla ve tvaru $A \to X\alpha$ a $A \to Y\beta$. A musí být v dalším kroku přepsán za nový řetězec a zároveň musí být deterministicky zvolené pravidlo, které se použije.

K deterministickému výběru slouží množina $Predict(A \to \gamma)$. Pokud pravidla, která na levé straně mají neterminál A, mají množinu $Predict(A \to \gamma)$ navzájem disjunktní, pravidlo je zvoleno deterministicky. Pro konstrukci množin $predict(A \to \gamma)$ je zapotřebí množin $Empty(\gamma)$, $First(\gamma)$ a $Follow(A)$.

Definice a algoritmy v této kapitole převzaty z [11, 6], kde jsou popsány velmi intuitivně, nicméně ne tak podrobně, jako například v [9]. Inspirace pro strukturu této kapitoly a slovní popisy algoritmů čerpána z [6].

Množina $Empty(\alpha)$ je množina, která obsahuje jediný prvek ε, pokud α derivuje ε, jinak je prázdná. Jinými slovy, pokud aktuální řetězec je možné vymazat pomocí ε-pravidel, pak bude množina pro tento řetězec obsahovat jediný prvek ε, jinak je prázdná.

Definice 4.1.1. Nechť $G = (N, T, P, S)$ je BKG.
Množina $Empty(\alpha)$, pro každé $\alpha \in N \cup T$, je definována jako

1) $Empty(\alpha) = \{\varepsilon\}$ pokud $\alpha \Rightarrow^* \varepsilon$,
2) $Empty(\alpha) = \emptyset$ jinak.

Algoritmus 4.1.1 Množina $Empty(\alpha)$

Vstup: BKG $G = (N, T, P, S)$
Výstup: $Empty(\alpha)$ pro každý symbol $\alpha \in N \cup T$

```
1: for all a ∈ T do
2:     Empty(a) ← ∅
3: end for
```

Na začátku výpočtu množin $Empty(\alpha)$ je pro každý terminál nastavena tato množina na prázdnou, protože z terminálů nikdy prázdný řetězec nedostaneme.

Algoritmus 4.1.1 Množina $Empty(\alpha)$ (pokračování)

4: **for all** $A \in N$ **do**
5: **if** $A \to \varepsilon \in P$ **then**
6: $Empty(A) \leftarrow \{\varepsilon\}$
7: **else**
8: $Empty(A) \leftarrow \emptyset$
9: **end if**
10: **end for**
11: **while** je možné měnit nějakou množinu $Empty(A)$ **do**
12: **if** $A \to X_1X_2 \dots X_n \in P$ **and** $Empty(X_i) = \{\varepsilon\}$ **foreach** $i \in \{1, \dots, n\}$ **then**
13: $Empty(A) \leftarrow \{\varepsilon\}$
14: **end if**
15: **end while**

Dále, pokud je možné přímo derivovat prázdný řetězec z aktuálního neterminálu A, pak je zřejmé, že neterminál A bude vymazán, pokud pravidlo pro prázdný řetězec bude použito. Nakonec se algoritmus znovu dívá na všechny řetězce α a jejich množiny $Empty(\alpha)$, které dále mohou být měněny. Jsou to takové neterminály A, které mají množinu $Empty(A)$ neprázdnou, v druhém případě se množina $Empty(A)$ již nezmění. Grafická ilustrace slovního popisu popisu a Algoritmu 4.1.1 je zobrazena na Obrázku 4.1.1.

$$\alpha = \boxed{X_1}\boxed{X_2}\cdots\boxed{X_n}$$
$$\downarrow \quad \downarrow \quad \cdots \quad \downarrow$$
$$\varepsilon \quad \varepsilon \quad \cdots \quad \varepsilon$$
$$\alpha = X_1X_2\cdots X_n \Rightarrow^* \varepsilon$$
$$\Downarrow$$
$$Empty(\alpha) = \{\varepsilon\}$$

Obrázek 4.1.1: Ilustrace tvorby množiny *Empty(α)* dle [11].

V následujícím algoritmu si představíme množinu $Empty(X_1X_2 \dots X_n)$. Tento algoritmus pomůže při definici dalších prediktivních množin. Pracuje především s neterminály, pro terminály nemá smysl jej představovat. Pokud sekvence $X_1X_2 \dots X_n$ může být v konečném počtu kroků přepsána na ε, pak $Empty(X_1X_2 \dots X_n) = \{\varepsilon\}$.

Algoritmus 4.1.2 Množina $Empty(X_1X_2 \dots X_n)$

Vstup: BKG $G = (N, T, P, S)$ a $Empty(X)$ pro všechny symboly $X \in N \cup T$
Výstup: $Empty(X_1X_2 \dots X_n)$, kde $(X_1X_2 \dots X_n) \in (N \cup T)^+$

1: **if** $Empty(X_i) = \varepsilon$ **foreach** $i \in \{1, \dots, n\}$ **then**
2: $Empty(X_1X_2 \dots X_n) \leftarrow \{\varepsilon\}$
3: **else**
4: $Empty(X_1X_2 \dots X_n) \leftarrow \emptyset$
5: **end if**

Množina $First(\alpha)$ je výčet všech terminálů, kterými může začínat řetězec derivovatelný z α.

Definice 4.1.2. Nechť $G = (N, T, P, S)$ je BKG.
Pro každé $\alpha \in (N \cup T)^*$ je definováno $First(\alpha)$ jako

$$First(\alpha) = \{a : a \in T,\ \alpha \Rightarrow^* a\beta,\ \beta \in (N \cup T)^*\}.$$

Ilustrace tvorby množiny *First(α)* je na Obrázku 4.1.2. Tento obrázek pracuje s jednodušší variantou, a to řetězcem terminálů, ve kterém se $X_i,\ i \in \{1, \ldots, n-1\}$ nepřepíše na ε.

Algoritmus 4.1.3 Množina $First(\alpha)$

Vstup: $G = (N, T, P, S)$
Výstup: $First(\alpha)$ pro každé $\alpha \in (N \cup T)^*$

```
1: for all a ∈ T do
2:     First(a) ← {a}
3: end for
4: for all A ∈ N do
5:     First(A) ← ∅
6: end for
7: while je možné měnit nějakou množinu First(A) do
8:     if A → X1X2...Xk−1Xk...Xn ∈ P then
9:         First(A) ← First(A) ∪ First(X1)
10:        if Empty(X1X2...Xk−1) = {ε} then
11:            First(A) ← First(A) ∪ First(Xk)
12:        end if
13:    end if
14: end while
```

Je zřejmé, že řetězec derivovatelný z terminálů a musí určitě začínat terminálem a. Ve druhém kroku algoritmu nastavíme $First(A)$ na prázdnou množinu, bude se měnit až v dalších krocích. Řetězec derivovatelný z neterminálu A pak může začínat:

1. terminály z množiny $First(\alpha_1)$, ať už se jedná o terminál či neterminál,
2. terminály z množiny $First(\alpha_k)$, pokud řetězec neterminálů $X_1X_2 \ldots X_{k-1}$ může být vymazán, respektive $Empty(X_1X_2 \ldots X_{k-1}) = \varepsilon$.

Pro snadnější definici množiny $Follow(A)$ existuje algoritmus tvorby množiny *First(α)* pro libovolné neprázdné řetězce. Je zapsán v Algoritmu 4.1.4. V prvním kroku se nastaví

Algoritmus 4.1.4 Množina $First(\alpha_1\alpha_2 \ldots \alpha_n)$

Vstup: BKG $G = (N, T, P, S)$ a $Empty(\alpha), First(\alpha)$ pro všechny symboly $\alpha \in N \cup T$
Výstup: množiny $First(\alpha_1\alpha_2 \ldots \alpha_n)$, kde $(\alpha_1\alpha_2 \ldots \alpha_n) \in (N \cup T)^+$

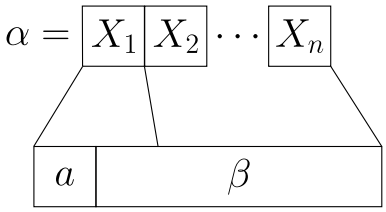
```
1: First(α1α2...αn) ← First(α1)
```

$First(\alpha_1\alpha_2 \ldots \alpha_n)$ na $First(\alpha_1)$, protože řetězec derivovatelný z $\alpha_1\alpha_2, \ldots, \alpha_n$ bude prvním symbolem z řetězce derivovatelného z α_1.

Algoritmus 4.1.4 Množina $First(\alpha_1\alpha_2\dots\alpha_n)$ (pokračování)

2: **while** je možné měnit nějakou množinu $First(\alpha_1\alpha_2\dots\alpha_{k-1}\alpha_k\dots\alpha_n)$ **do**
3: **if** $Empty(\alpha_1\alpha_2\dots\alpha_{k-1}) = \varepsilon$ **then**
4: $First(\alpha_1\alpha_2\dots\alpha_n) \leftarrow First(\alpha_1\alpha_2\dots\alpha_n) \cup First(\alpha_k)$
5: **end if**
6: **end while**

Při dalších krocích se do původní množiny sekvence $\alpha_1\alpha_2\dots\alpha_n$ přidávají symboly z $First(\alpha_k)$, pokud řetězec $\alpha_1\alpha_2\dots\alpha_{k-1} \Rightarrow^* \varepsilon,\ k \in \{1,\dots,n\}$.



$$\alpha = X_1X_2\cdots X_n \Rightarrow^* a\beta$$
$$\Downarrow$$
$$a \in First(\alpha)$$

Obrázek 4.1.2: Ilustrace významu množiny *First(α)* dle [11].

Další velmi důležitou množinou, kterou syntaktický analyzátor potřebuje, je množina $Follow(A)$. Ta určuje, jaké symboly se mohou vyskytovat za neterminálem A. Nutnost této znalosti vyplývá z možnosti přepsat neterminály, pro které platí $Empty(A) = \{\varepsilon\}$, na prázdný řetězec. Kdyby analyzátor tuto množinu neměl k dispozici, neměl by v těchto případech jak zjistit, zda je aktuální symbol na vstupu korektní nebo ne. K definici množin $Follow(A)$ se používá pomocný symbol \$, který reprezentuje konec vstupního řetězce.

Definice 4.1.3. Nechť $G = (N, T, P, S)$ je BKG.
Pro všechny neterminály $A \in N$ definujeme množinu $Follow(A)$ jako

$$Follow(A) = \{a : a \in T,\ S \Rightarrow^* \alpha Aa\beta,\ \alpha,\beta \in (N \cup T)^*\} \cup$$
$$\{\$: S \Rightarrow^* \alpha A,\ \alpha \in (N \cup T)^*\}.$$

Z uvedené Definice 4.1.3 a Algoritmu 4.1.5 vyplývá, že v množině $Follow(A)$ se mohou vyskytovat:

1. terminální symboly vyskytující se za neterminálem A v libovolném zderivovaném řetězci,

2. symbol ukončující vstupní řetězec \$, pokud za neterminálem A již nebudou žádné další neterminály.

Množina $Follow(A)$ byla poslední množina, která byla třeba definovat, abychom mohli definovat množinu $Predict(A \to \alpha)$, ze které se poté může zkonstruovat *LL tabulka*. Množina $Predict(A \to \alpha)$ už je prakticky dobře použitelná pro syntaktickou analýzu – je to množina všech terminálů, které mohou být aktuálně nejlevěji vygenerovány, pokud pro libovolnou větnou formu použijeme pravidlo $A \to \alpha$.

Algoritmus 4.1.5 Množina $Follow(A)$

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

```
1: Follow(S) ← $
2: while je možné měnit nějakou množinu Follow(A) do
3:     if A → αBβ ∈ R then
4:         if β ≠ ε then
5:             Follow(B) ← Follow(B) ∪ First(β)
6:         end if
7:         if Empty(β) = {ε} then
8:             Follow(B) ← Follow(B) ∪ Follow(A)
9:         end if
10:    end if
11: end while
```

Definice 4.1.4. Nechť $G = (N, T, P, S)$ je BKG.
Pro každé pravidlo $A \to \alpha \in P$ definujeme množinu $Predict(A \to \alpha)$ jako

$$Predict(A \to \alpha) = First(\alpha) \cup Follow(A)$$

pokud $Empty(\alpha) = \{\varepsilon\}$,

$$Predict(A \to \alpha) = First(\alpha)$$

pokud $Empty(\alpha) = \emptyset$.

Prakticky díky této množině je syntaktický analyzátor schopen deterministicky vybrat pravidlo, které se aplikuje. Mějme dvě pravidla, p a q, mezi kterými se analyzátor rozhoduje a nechť na vstupu je terminál a. Pokud $a \in Predict(p)$, pak se vybere pravidlo p, pokud $a \in Predict(q)$, pak se vybere pravidlo q. Jak už bylo zmiňováno na začátku této kapitoly, všechny množiny $Predict(p)$ musí být navzájem disjunktní, jinak by výběr byl opět nedeterministický. Tento koncept je aplikován v *LL(1) gramatikách*.

LL gramatiky

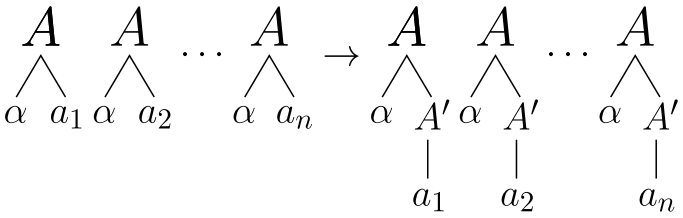
Výše uvedené množiny pracují s gramatikami, které se nazývají *LL(1) gramatiky*. Obecně gramatiky mohou být *LL(k)*. To jsou gramatiky, u kterých stačí k vstupních tokenů k tomu, aby jejich analyzátor mohl deterministicky vybrat pravidlo pro aktuální neterminál. Obecně platí, že LL(1) gramatiky jsou slabší (generují podmnožinu jazyků) než BKG [9]. LL(1) gramatika je definována následovně:

Definice 4.1.5. Nechť $G = (N, T, P, S)$ je BKG.
G je *LL(1) gramatikou*, pokud pro libovolná dvě pravidla $p, q \in P$ platí

$$p \neq q \text{ a zároveň } Predict(p) \cap Predict(q) = \emptyset.$$

Jinými slovy neexistuje žádné $a \in T$ a $A \in N$ takové, že $A \to \alpha, A \to \beta \in P$ a zároveň $a \in First(\alpha)$ a $a \in First(\beta)$.

Konvence 4.1.1. Tato práce se zabývá především LL(1) gramatikami. Pro zjednodušení, LL(1) gramatiky budou implicitně označovány jako *LL gramatiky*. Bude-li se jednat o LL(k), bude to explicitně zapsáno.



Obrázek 4.1.3: Grafické znázornění úpravy pravidel pomocí faktorizace dle [11].

Některé bezkontextové gramatiky se dají převést na ekvivalentní LL gramatiky pomocí technik *faktorizace* (vytýkání) a *odstranění levé rekurze*.

Myšlenka faktorizace je vytknutí stejné sekvence terminálů, která je společným prefixem několika pravých stran pravidel a nahrazení zbývajícího řetězce za nový neterminál. Z nového neterminálu dále vznikají nová pravidla. Například, mějme tato pravidla:

$$\begin{aligned} A &\to \alpha a_1 \\ A &\to \alpha a_2 \\ &\vdots \\ A &\to \alpha a_3 \end{aligned}$$

Sekvence symbolů se může vytknout a místo $a_1, a_2, \ldots, a_n$ se vloží nový neterminál B, ze kterého se dále generují $a_1, a_2, \ldots, a_n$. Graficky je tato úprava ukázána na Obrázku 4.1.3.

$$\begin{aligned} A &\to \alpha A' \\ A' &\to a_1 \\ A' &\to a_2 \\ &\vdots \\ A' &\to a_3 \end{aligned}$$

Levá rekurze je případ, kdy stejný neterminál, jako je na levé straně pravidla, je zároveň nejlevějším symbolem řetězce na pravé straně pravidla, například:

$$\begin{aligned} A &\to A\alpha \\ A &\to a \end{aligned}$$

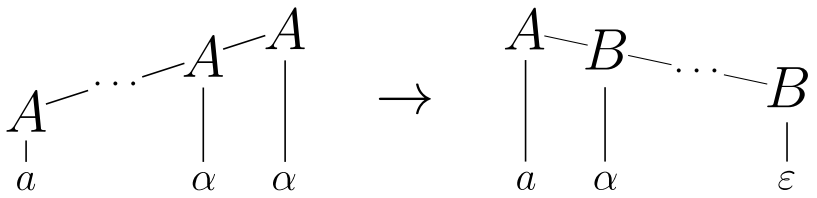
Tato pravidla se dají přepsat do tvaru:

$$\begin{aligned} A &\to aB \\ B &\to \alpha B \\ B &\to \varepsilon \end{aligned}$$

Graficky znázorněný význam odstranění levé rekurze je na Obrázku 4.1.4.

LL tabulka a její konstrukce

LL tabulka je abstrakcí k množinám $Predict(p)$ jednotlivých pravidel LL gramatiky $G = (N, T, P, S)$. Řádky tabulky jsou indexovány neterminály z množiny N, sloupce potom terminály z množiny $T \cup \{\$\}$. Samotné položky tabulky jsou čísla pravidel, která byla přidělena funkcí mapující dvojici (A, a), $a \in T \cup \{\$\}$, $A \in N$ na



Obrázek 4.1.4: Grafické zobrazení významu odstranění levé rekurze dle [11].

- pravidlo, respektive symbol reprezentující konkrétní pravidlo,
- symbol vyjadřující neexistenci pravidla pro danou dvojici; v tomto případě syntaktická analýza hlásí chybu.

Jinak řečeno, nechť $\alpha(A, a)$ reprezentuje políčko v LL tabulce a nechť $p : A \to X_1 X_2 \dots X_n \in P$. Pokud $a \in First(X_1)$, pak $\alpha(A, a) = p$, jinak nastává chyba. Následující příklad převzat z [9].

Příklad 4.1.1. Nechť $G = (N, T, P, S)$ je LL gramatika a nechť má sestavené množiny $Predict(p)$ pro každé $p \in P$.
Pravidla a jejich množiny $Predict$ nechť jsou definovány následovně:

Pravidlo $r \in R$	$Predict(r)$
$E \to TA$	$i, ($
$A \to \vee\, TA$	$\vee$
$A \to \varepsilon$	$), \$$
$T \to FB$	$i, ($
$B \to \wedge\, FB$	$\wedge$
$B \to \varepsilon$	$\vee,)$
$T \to F$	$i, ($
$T \to (E)$	$($
$F \to i$	i

Tabulka 4.1.1: Pravidla G a jejich množiny $Predict$.

Nechť symbol $\times$ reprezentuje prázdné políčko LL tabulky. LL tabulka pro pravidla a jejich $Predict$ množiny z tabulky 4.1.1, zkonstruována podle výše popsané metody, vypadá takto:

i	i	$\vee$	$\wedge$	$($	$)$	\$
E	1	$\times$	$\times$	1	$\times$	$\times$
A	$\times$	2	$\times$	$\times$	3	3
T	4	$\times$	$\times$	4	$\times$	$\times$
B	$\times$	6	5	$\times$	6	6
F	9	$\times$	$\times$	8	$\times$	$\times$

Tabulka 4.1.2: LL tabulka pro pravidla z tabulky 4.1.1.

Prediktivní syntaktická analýza

Prediktivní syntaktická analýza a *rekurzivní sestup* jsou dvě metody, pomocí kterých lze implementovat syntaktický analyzátor. Jak je zmíněno na začátku této kapitoly, je potřeba nějakým způsobem provádět analýzu pomocí zásobníkového automatu. Při použití prediktivní analýzy je nutné tento zásobník implementovat explicitně, rekurzivní sestup si zásobník tvoří v pozadí automaticky díky volání funkcí pro každý neterminál. Součástí této práce je implementace explicitního zásobníkového automatu, proto nebude rekurzivní sestup podrobně popsán; této metodě se detailně věnuje autor v [9].

Výhodou implementace prediktivní analýzy s LL tabulkou je jediná implementace algoritmu. V případě změny pravidel v gramatice je nutné pouze změnit LL tabulku, odkud automat odebírá pravidla, která má použít. Oproti tomu v rekurzivním sestupu je nutné měnit celé funkce. Algoritmus prediktivní syntaktické analýzy je popsán v Algoritmu 4.1.6.

Algoritmus 4.1.6 Prediktivní syntaktická analýza založená na LL tabulce

Vstup: LL tabulka pro $G = (N, T, P, S)$, řetězec $x \in T^*$
Výstup: Levý rozbor pro x, pokud $x \in L(G)$, jinak chyba.

```
1:  stack.push($) and stack.push(S)
2:  repeat
3:      X ← stack.top()
4:      a ← aktuální token ze vstupního řetězce
5:      switch X :
6:          case $ :
7:              if a = $ then
8:                  úspěch
9:              else
10:                 chyba
11:             end if
12:         case X ∈ T :
13:             if X = a then
14:                 stack.pop(X)
15:                 a ← nový token ze vstupního řetězce
16:             else
17:                 chyba
18:             end if
19:         case X ∈ N :
20:             if r : X → α ∈ LL table[X, a] then
21:                 stack.pop(X)
22:                 stack.top() ← reversal(α)
23:             else
24:                 chyba
25:             end if
26: until úspěch or chyba
```

4.2 Syntaktická analýza zdola nahoru

Syntaktická analýza zdola nahoru provádí konstrukci derivačního stromu zleva doprava a zespodu nahoru, tudíž jde od listových uzlů (tokenů) směrem nahoru ke kořeni (výchozí neterminál) a provádí nejpravější derivaci vstupního řetězce. Každý krok této metody je reprezentován operacemi posunu (*shift*) nebo redukce (*reduce*). Operace shift provádí vložení symbolu na zásobník a redukce simuluje nejpravější derivační krok. Během redukce analyzátor musí na zásobníku najít pravou stranu libovolného pravidla (nazývanou *handle*), zredukovat jej a výsledný neterminál vložit zpět na zásobník.

Pro tuto práci je důležitá pouze jedna z metod analýzy zdola nahoru, a to *precedenční analýza výrazů*. Existují další metody, jako je například *LR syntaktická analýza*, ta ale není předmětem této práce, popsána bude pouze precedenční analýza. LR analýza je podrobně popsána v [9].

Precedenční analýza výrazů

Precedenční analyzátor pracuje na základě $G = (N, T, P, S)$ pomocí *precedenční tabulky*. Precedenční tabulka je struktura se sloupci i řádky tvořenými symboly z množiny $T \cup \{\$\}$, kterými se tabulka v praxi indexuje. Každé pole precedenční tabulky má hodnotu jednoho ze symbolů množiny $\{<, >, =, \times, \checkmark\}$.

Analyzátor si na zásobník ukládá terminály i neterminály a navíc speciální znak $<$, který určuje počátek řetězce připravený na redukci. Dno zásobníku pak reprezentuje symbol \$; zásobníková abeceda toho analyzátoru je tedy $T \cup \{<, \$\}$.

Než bude ukázán algoritmus pro precedenční analýzu, je třeba zadefinovat operace *SHIFT* a *REDUCE*, se kterými analyzátor pracuje.

Definice 4.2.1. Nechť $G = (N, T, P, S)$.
Precedenční syntaktický analyzátor pracující na základě G používá tyto operace pro práci se zásobníkem:

- $SHIFT(<)$ přesune řetězec $< a,\ a \in T$ na vrchol zásobníku a přečte další symbol ze vstupu,
- $SHIFT(=)$ přesune vstupní symbol $a \in T$ na vrchol zásobníku a přečte další symbol ze vstupu,
- $REDUCE(A \to \alpha)$, kde $A \in N,\ \alpha = a\beta,\ a \in T,\ \beta \in (N \cup T)^*$ nahradí řetězec α za A na vrcholu zásobníku.

Funkcionalita precedenčního analyzátoru demonstrována na Algoritmu 4.2.1. Informace k tomuto algoritmu čerpány z [9, 12], inspirace zápisu čerpána z [6].

Jednotlivé kroky analýzy jsou určeny precedenční tabulkou, jejíž řádek je indexován nejvrchnějším (*topmost*) terminálem na zásobníku a sloupec je indexován vstupním terminálem, případně symbolem \$. Pokud je v políčku tabulky symbol $=$ nebo $<$, pak analyzátor provede operaci $SHIFT$. Je-li varianta $SHIFT(<)$, pak je nejdříve na zásobník vložen symbol $<$, a to před první terminál, který se na zásobníku vyskytuje – pokud je na vrcholu zásobníku neterminál, přeskočí se. Slouží k rozpoznání řetězce, který má být v rámci redukce na zásobníku nahrazen. Poté se vloží na zásobník i vstupní symbol a přečte se nový ze vstupního řetězce. Pokud je v tabulce na políčku $>$, pak proběhne operace $REDUCE$, při které:

Algoritmus 4.2.1 Precedenční syntaktický analyzátor

Vstup: Precedenční tabulka pro $G = (N, T, P, S)$, $x \in T^*$ ukončený symbolem \$
Výstup: Pravý rozbor x, pokud $x \in L(G)$; jinak chyba

```
1:  stack.push($)
2:  repeat
3:      switch Tabulka[topmost_token, input_token] :
4:          case = :
5:              SHIFT(=)
6:          case < :
7:              SHIFT(<)
8:          case > :
9:              if stack.top() = < α and r : A → α ∈ R then
10:                 REDUCE(A → α)
11:             else
12:                 chyba                    ▷ neexistuje pravidlo pro řetězec na zásobníku
13:             end if
14:         case × :
15:             chyba                                  ▷ tabulka detekovala chybu
16:         case ✓ :
17:             úspěch
18: until chyba or úspěch
```

1. najde na zásobníku řetězec α, který má být zredukován (přepsán) řetězcem novým,
2. určí pravidlo, u kterého se pravá strana rovná řetězci α,
3. nahradí řetězec α levou stranou určeného pravidla.

Když tabulka vrátí symbol $\times$, nastává chyba. Při symbolu ✓analýza končí úspěšně. V Příkladu 4.2.1 čerpaného z [9] a zjednodušeného, je ukázán princip fungování Algoritmu 4.2.1. Je ukázán na konci této kapitoly, nejdříve je nutné popsat konstrukci precedenční tabulky.

Konstrukce precedenční tabulky

Doposud nebylo zmíněno, co znamenají položky precedenční tabulky pro analýzu výrazů. Znaménka $<$ a $>$ se interpretují jako priorita operátorů. Mějme dva operátory, například $\odot$ a $\triangle$. Pokud $\odot < \triangle$, pak $\triangle$ má vyšší prioritu, a v algoritmu to znamená, že pravá strana pravidla obsahující operátor $\triangle$ bude redukována dříve než pravá strana obsahující operátor $\odot$ a naopak. Pravidla pro tvorbu tabulky jsou pak následující [9, 12]:

1. Pokud operátor $\triangle$ má vyšší matematickou prioritu než operátor $\odot$, pak $\triangle > \odot$ a $\odot < \triangle$.
2. Pokud operátory $\triangle$ a $\odot$ jsou stejné matematické priority a levě asociativní, pak $\triangle < \odot$ a $\odot < \triangle$. Pro pravě asociativní operátory platí $\triangle > \odot$ a $\odot > \triangle$.
3. Pokud $a \in T$ může předcházet operandu i, pak $a < i$.
4. Pokud $a \in T$ může následovat za operandem i, pak $i > a$.

5. Závorky:
   (a) Pro jeden pár závorek platí $(=)$.
   (b) Pro $a \in T \setminus \{), \$\}$ platí $(< a$.
   (c) Pro $a \in T \setminus \{(, \$\}$ platí $a >)$.
   (d) Pokud $a \in T$ může předcházet (, pak $a < (.$
   (e) Pokud $a \in T$ může následovat za), pak $) > a$.
6. Nechť $\odot$ je libovolný operátor. Pak platí $\$ < \odot$ a $\odot > \$$. Dále platí $\$ \checkmark \$$.
7. Zbytek tabulky je vyplněn symbolem $\times$.

Všimněme si pomocného symbolu \$, který slouží jako ukončovač řetězce podobně jako u syntaktické analýzy shora dolů, poprvé byl zmíněn v Definici 4.1.3.

Příklad 4.2.1. Nechť $G = (N, T, P, S)$ je BKG a nechť množina R obsahuje tato pravidla:

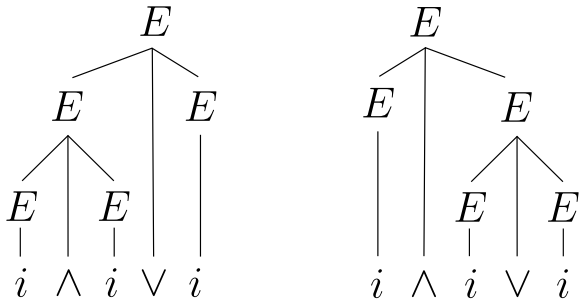
$$1 : C \to C \vee C,$$
$$2 : C \to C \wedge C,$$
$$3 : C \to i.$$

Také předpokládejme precedenční tabulku ukázanou v Tabulce 4.2.1 a nechť vstupní řetězec je $i \wedge i \vee i\$$. Pokud bychom neznali precedenční tabulku, mohly by vzniknout dva derivační

	$\wedge$	$\vee$	i	\$
$\wedge$	$>$	$>$	$<$	$>$
$\vee$	$<$	$>$	$<$	$>$
i	$>$	$>$	$\times$	$>$
\$	$<$	$<$	$<$	$\checkmark$

Tabulka 4.2.1: Precedenční tabulka pro pravidla G.

stromy, které jsou ukázány na Obrázku 4.2.1. Průběh analýzy tohoto řetězce je znázorněn v Tabulce 4.2.2.



Obrázek 4.2.1: Dva možné derivační stromy při neznalosti precedence operátorů pro vstupní řetězec $i \wedge i \vee i\$$.

Precedenční analýza nedokáže pracovat s ε-pravidly a pravidly se stejnou pravou stranou pro jiné neterminály díky hledání pravých stran pravidel na zásobníku. V praxi se

Zásobník	**Precedence**	**Vstup**	**Akce**
$\underline{\$}$	$[\$, i] \to <$	$\underline{i} \wedge i \vee i\$$	$SHIFT(<)$
$\$ < \underline{i}$	$[i, \wedge] \to >$	$\underline{\wedge} i \vee i\$$	$REDUCE(C \to i)$
$\underline{\$}C$	$[\$, \wedge] \to <$	$\underline{\wedge} i \vee i\$$	$SHIFT(<)$
$\$ < C\underline{\wedge}$	$[\wedge, i] \to <$	$\underline{i} \vee i\$$	$SHIFT(<)$
$\$ < C\wedge < \underline{i}$	$[i, \vee] \to >$	$\underline{\vee} i\$$	$REDUCE(C \to i)$
$\$ < C\underline{\wedge}C$	$[\wedge, \vee] \to >$	$\underline{\vee} i$	$REDUCE(C \to C \wedge C)$
$\underline{\$}C$	$[\$, \vee] \to <$	$\underline{\vee} i\$$	$SHIFT(<)$
$\$ < C\underline{\vee}$	$[\vee, i] \to <$	$\underline{i}\$$	$SHIFT(<)$
$\$ < C \vee \underline{i}$	$[i, \$] \to >$	$\underline{\$}$	$REDUCE(C \to i)$
$\$ < C\underline{\vee}C$	$[\vee, \$] \to >$	$\underline{\$}$	$REDUCE(C \to C \vee C)$
$\underline{\$}C$	$[\$, \$] \to \checkmark$	$\underline{\$}$	úspěch

Tabulka 4.2.2: Demonstrace precedenční analýzy popsané v Algoritmu 4.2.1.

tedy většinou používá výhradně pro analýzu výrazů, které jsou reprezentovány například neterminálem `EXPRESSION` a žádný jiný neterminál se jimi nezabývá. Pro efektivní analýzu komplexnějších jazyků se často kombinuje precedenční analýza s analýzou shora dolů. Analýza shora dolů se stará o kontrolu vstupního řetězce mimo výrazy (analyzuje podmíněné příkazy, cykly a podobně) a pro výrazy předává řízení analýze precedenční.

Kapitola 5

Implementace syntaktického analyzátoru pro jazyk Koubp

5.1 Přijímaný jazyk

Jazyk *Koubp* je založený na jazyce IFJ22, který je podmnožinou jazyka PHP 8, jenž byl specifikován v rámci zadání projektu do předmětu Formální jazyky a překladače v akademickém roce 2022/2023. V této kapitole jsou popsány především syntaktické konstrukty jazyka Koubp a není podrobně popsána sémantika příkazů, protože sémantická analýza není předmětem této práce.

Obecné vlastnosti

Jazyk Koubp je strukturovaný jazyk. Podporuje deklaraci funkcí a proměnných, které při deklaraci musí být explicitně definovány. Hlavní tělo programu se skládá z prolínání sekvence příkazů a definic funkcí, přičemž definice funkcí nemohou být vnořené, ale mohou se vzájemně rekurzívně volat. Vstupním bodem programu není funkce `main()`, jak lze nalézt například u jazyka C [3], analýza probíhá od začátku souboru. Není podporováno slučování souborů se zdrojovým kódem pomocí příkazů `import`, `include` a podobných a tím vytvoření jediného modulu, který by bylo možné zkompilovat. Ve zdrojovém souboru a uživatelem definovaných funkcích může být větvení, iterace a další běžné konstrukce. Veškeré proměnné jsou lokální, i v rámci hlavního těla programu. Proměnné mají rozsah platnosti v *bloku kódu*, což je sekvence příkazů ve složených závorkách `{}`. Tyto bloky kódu se mohou libovolně zanořovat. V jazyce Koubp nejsou důležité *bílé mezery (whitespaces)*. Klíčová slova jazyka jsou

`if, while, for, return, elseif, else, function, int, float, string, bool`.

Identifikátory proměnných jsou posloupnosti alfanumerických znaků a znak `_`; identifikátor nesmí začínat číslicí. Regulárním výrazem zapsáno jako:

`[a-zA-Z_][a-zA-Z0-9_]*`

Jazyk Koubp podporuje čtyři datové typy, a to `int, float, bool, string`. Jejich literály jsou definovány následovně:

- Celočíselné literály jsou neprázdná posloupnost číslic, regulární výraz je

  `[0-9]+`

- Literály pro desetinná čísla, respektive čísla s plovoucí desetinnou čárkou, jsou neprázdná posloupnost číslic následovaná tečkou, za kterou je další neprázdná posloupnost číslic.

```
[0-9]+.[0-9]+
```

- Literály typu `bool` jsou `true` a `false`.
- Řetězcové literály jsou posloupností (i prázdnou) znaků uzavřenou v uvozovkách. Pro zapsání speciálních znaků je možnost použít *escape sekvence*, které začínají zpětným lomítkem.

```
\"(\\.|[^\\"])*\"
```

Jsou také podporovány řádkové a blokové komentáře. Řádkový komentář je sekvence znaků počínající dvojitým lomítkem `//`, končí na konci aktuálního řádku. Blokový komentář může být na více řádků počínaje znaky `/*`, konče znaky `*/`.

Deklarace a definice funkcí

Jak bylo zmíněno v úvodu této kapitoly, deklarované funkce musí být rovnou definovány. Jazyk Koubp nepodporuje *přetěžování (overloading)*[1] funkcí. Definice funkce se skládá z hlavičky za klíčovým slovem `function` a těla funkce. V hlavičce se definuje název funkce, návratový typ a počet parametrů a jejich datové typy. Schematicky vypadá následovně:

```
function <jméno> ( <seznam parametrů> ) : <datový typ> { <tělo funkce> }
```

Dvojtečka před datovým typem musí být uvedena vždy, a to i v případě, že funkce nemá návratovou hodnotu (takzvaně `void` funkce). V tomto případě za dvojtečkou ihned následuje levá složená závorka `{`. Parametry funkce jsou oddělené čárkami a jejich struktura je:

```
<datový typ> <jméno proměnné>
```

Tělo funkce je pak neprázdná sekvence příkazů. Příklad definice funkce je:

```
function factorial(int number): int {
    if (number < 0) {
        return -1;
    }
    elseif (number <= 1) {
        return 1;
    }

    return number * factorial(number - 1);
}
```

[1] Pro C++ například https://learn.microsoft.com/en-us/cpp/cpp/function-overloading?view=msvc-170. V ostatních kompilátorech, jako `gcc`/`g++` nebo `clang`/`clang++` budou principy minimálně velmi podobné.

Příkazy

Jazyk Koubp podoruje obvyklé příkazy a konstrukce jiných programovacích jazyků.

- *Přiřazení* – stejně, jako u definic funkcí, deklarace není možná. Hodnota se může přiřadit do již definované proměnné nebo při vytvoření nové proměnné, která předtím definovaná nebyla.

```
<datový typ> <jméno proměnné> = <hodnota>
             <jméno proměnné> = <hodnota>
```

- *Větvení* – podmíněný příkaz podporuje klíčová slova `if`, `elseif`, `else`. Ačkoliv je možnost používat přímo klíčové slovo `elseif`, jazyk Koubp podporuje i sekvenci `else if`, jejichž sémantický rozdíl je patrně vidět v příloze této práce na obrázcích A.1.1 a A.2.1. Syntaxe je následující:

```
if (vyraz) {
    sekvence_prikazu
}
elseif (vyraz) {
    sekvence_prikazu
}
else {
    sekvence_prikazu
}
```

  Je podporováno psát sekvence příkazů i bez složených závorek `{}`. V tom případě se ke konstrukci vztahuje pouze první příkaz, který se za konstrukcí vyskytuje, podobně jako v jazyce C [3]. Tohle pravidlo platí i pro příkazy nebo sekvence příkazů v cyklech.

```
if (vyraz)
    prikaz;
elseif (vyraz)
    prikaz;
else
    prikaz;
```

- *Cyklus while* – Vykonává se sekvence příkazů (nebo příkaz) tak dlouho, dokud *výraz* není vyhodnocen jako pravda.

```
while (vyraz) {
    sekvence_prikazu
}

while (vyraz)
    prikaz;
```

- *Cyklus for* se skládá z hlavičky a těla. Hlavička se skládá ze tří částí – inicializace a dva výrazy, všechny jsou odděleny středníkem. Inicializace může být výraz, deklarace nebo přiřazení.

```
for (inicializace; vyraz; vyraz) {
    sekvence_prikazu
}

for (inicializace; vyraz; vyraz)
    prikaz;
```

- *Příkaz return* – začíná klíčovým slovem `return` a následuje volitelný výraz. Vrátí hodnotu výrazu z volané funkce nebo spuštěného programu.

```
return vyraz;
return;
```

Výrazy

Výrazy jazyka Koubp jsou tvořeny operátory, operandy a závorkami. Mezi operandy se řadí proměnné, konstanty (literály) a volání funkcí.

- Operátory jsou unární (`! -`) a binární, které se dále dělí na
  - aritmetické `. + - * /`,
  - logické `&& ||`,
  - relační `== != > < >= <=`,
  - přiřazení `=`.

  Unární mínus je v implementaci reprezentováno jiným tokenem než binární mínus. Na přiřazení se v jazyce Koubp hledí jako na operátor s nejmenší prioritou. Proto například výraz

  ```
  variable = 1 + 2 + 3 = retezec1.retezec2;
  ```

  je syntakticky v pořádku. O dodatečnou kontrolu správnosti tohoto výrazu se musí postarat sémantická analýza.

- Tabulka 5.1.1 ilustruje prioritu operátorů sestupně a jejich asociativitu. Asociativita udává, v jakém pořadí se vyhodnotí operátory se stejnou prioritou. Například, mějme výraz

  $$1 + 2 + 3.$$

  Pokud operátor `+` je levě asociativní, pak je výraz vyhodnocen jako

  $$(1 + 2) + 3,$$

  v opačném případě jako

  $$1 + (2 + 3).$$

Pořadí	Seznam operátorů	Asociativita
1)	`! -`	levá
2)	`* /`	levá
3)	`+ - .`	levá
4)	`== != > < >= <=`	levá
5)	`&& ||`	levá

Tabulka 5.1.1: Priorita operátorů a jejich asociativita.

Asociativita je důležitá především u operátorů, které nejsou komutativní, jako například operátor dělení `/`. Nicméně se musí definovat pro všechny operátory, aby precedenční analyzátor věděl, který z výrazů má vyhodnotit dříve. Implementačně to znamená, že výrazy s operátory s vyšší prioritou budou vyhodnoceny dříve než operátory s nižší prioritou.

- Volání funkce je bráno jako operand, který se může ve výrazech běžně používat. Dále má jako argumenty výrazy oddělené čárkami, tudíž se mohou libovolně zanořovat. Následující příklad je syntakticky správně v jazyce Koubp.

```
int p = foo(s1."bar\n", 8*7) + !convert_bool(readline());
```

Vestavěné funkce

Jazyk Koubp má několik vestavěných funkcí, které se ve zdrojovém kódu mohou použít bez jejich předchozí definice. Dělí se na vstupně-výstupní funkce, funkce pro typovou konverzi a funkce pro práci s řetězci.

Vstupně-výstupní funkce pro práci se standardním vstupem/výstupem jsou:

- `function print(string vyraz1, string vyraz2, ...)`,
- `function readline(): string`.

Funkce `print()` na standardní výstup vypíše výrazy postupně zleva doprava bez oddělovačů. Pro zapsání jiných typů než řetězce je možné použít funkce pro typovou konverzi, případně znak `$` uvnitř řetězce pro vepsání hodnoty proměnné do řetězce. Například v následujícím příkladu

```
float cislo = 42.37;
print("$cislo$\n");
print(convert_string(cislo)."\n");
```

se v obou případech vypíše na standardní výstup stejný řetězec. Pro vypsání znaku `$` musí být použita escape sekvence, tudíž `\$`. Pro konstantní hodnoty funguje pouze varianta s funkcí `convert_string()`. Funkce `readline()` přečte sekvenci znaků ukončenou `\n` ze standardního vstupu a vrátí ji jako řetězec.

Pro práci se soubory jsou funkce

- `function file_open(string filename): int`,
- `function file_write(string filename, string buffer)`,
- `function file_readline(string filename): string`,

- `function file_close(string filename): int`.

Sémantika je očekáváná. Funkce `file_open()` otevře soubor specifikovaný v parametru `filename` pro čtení nebo zápis a vrací návratový kód pomocí kterého se může identifikovat chyba. `file_write()` zapíše do souboru `filename` řetězec v parametru `buffer`. Dále funkce `file_readline()` přečte jeden řádek ze souboru, případně vrací prázdný řetězec a funkce `file_close()` zavře soubor `filename` pro čtení i zápis, vrací návratový kód pro detekci chyby.

Typové konverze jsou uskutečněny funkcemi

- `function convert_int(term): int`,
- `function convert_string(term): string`,
- `function convert_float(term): float`,
- `function convert_bool(term): bool`,

a jsou to jediné vestavěné funkce, které pracují s operandem libovolného datového typu. Například:

```
int cislo = convert_int(readline());
bool hodnota = convert_bool(cislo);
print(convert_string(hodnota)); // Vypise "true" nebo "false" na stdin
```

Poslední vestavěné funkce jsou takové, které pracují s řetězci. Jsou to funkce

- `function strlen(string s): int` a
- `function substring(string s, int i, int j): string`.

Funkce `strlen()` vrací délku řetězce, který je dán jako parametr a `substring()` vrátí podřetězec řetězce `s` od indexu `i` po index `j`.

5.2 Gramatický systém definující syntax jazyka Koubp

Gramatický systém popisující syntax jazyka Koubp je CD gramatický systém. Formálně zapsáno, je to devítice $\Gamma = (N, T, S, P_1, P_2, P_3, P_4, P_6, P_6)$, kde:

$N = \{$`PROGRAM`, `STATEMENT_LIST`, `STATEMENT`, `IF2`, `DECLOREXP`, `RETURNEXP`, `FUNCDEF`, `PARAMS`, `PARAMS2`, `EXPRESSION`, `ARGS`, `ARGS2`, `CODEBLOCK`, `STATEMENTS`, `VOLTYPE`, `TYPE`$\}$

$T = \{$`if`, `elseif`, `else`, `while`, `for`, `return`, `function`, `int`, `float`, `string`, `bool`, `variable`, `constant`, `functionName`, `+`, `-`, `*`, `/`, `.`, `&&`, `||`, `==`, `!=`, `>`, `<`, `>=`, `<=`, `:`, `,`, `(`, `)`, `{ }`$\}$

$S =$ `PROGRAM`

Komponenty systému jsou navrhnuty tak, aby každá z nich tvořila ucelenou část jazyka Koubp. První komponenta popisuje strukturu programu – sekvence příkazů a definic funkcí.

$$
\begin{aligned}
P_1 = \{&\texttt{PROGRAM} \rightarrow \texttt{STATEMENT STATEMENT\_LIST},\\
&\texttt{PROGRAM} \rightarrow \texttt{FUNCDEF STATEMENT\_LIST},\\
&\texttt{STATEMENT\_LIST} \rightarrow \texttt{STATEMENT STATEMENT\_LIST},\\
&\texttt{STATEMENT\_LIST} \rightarrow \texttt{FUNCDEF STATEMENT\_LIST},\\
&\texttt{STATEMENT\_LIST} \rightarrow \varepsilon\}
\end{aligned}
$$

Druhá komponenta obsahuje seznam příkazů a běžných konstrukcí doplněné o jejich pomocné neterminály.

$$
\begin{aligned}
P_2 = \{&\texttt{STATEMENT} \rightarrow \texttt{if (EXPRESSION) CODEBLOCK IF2},\\
&\texttt{STATEMENT} \rightarrow \texttt{while (EXPRESSION) CODEBLOCK},\\
&\texttt{STATEMENT} \rightarrow \texttt{for (DECLOREXP; EXPRESSION; EXPRESSION) CODEBLOCK},\\
&\texttt{STATEMENT} \rightarrow \texttt{DECLOREXP;},\\
&\texttt{STATEMENT} \rightarrow \texttt{CODEBLOCK},\\
&\texttt{STATEMENT} \rightarrow \texttt{return RETURNEXP;},\\
&\texttt{STATEMENT} \rightarrow \texttt{;},\\
&\texttt{IF2} \rightarrow \texttt{elseif (EXPRESSION) CODEBLOCK IF2},\\
&\texttt{IF2} \rightarrow \texttt{else CODEBLOCK},\\
&\texttt{IF2} \rightarrow \varepsilon,\\
&\texttt{DECLOREXP} \rightarrow \texttt{TYPE variable = EXPRESSION},\\
&\texttt{DECLOREXP} \rightarrow \texttt{EXPRESSION},\\
&\texttt{RETURNEXP} \rightarrow \texttt{EXPRESSION},\\
&\texttt{RETURNEXP} \rightarrow \varepsilon\}
\end{aligned}
$$

Třetí komponenta zahrnuje definici funkce a pomocné neterminály.

$$
\begin{aligned}
P_3 = \{&\texttt{FUNCDEF} \rightarrow \texttt{function functionName (PARAMS) : VOLTYPE \{ STATEMENTS \}},\\
&\texttt{PARAMS} \rightarrow \texttt{EXPRESSION PARAMS2},\\
&\texttt{PARAMS} \rightarrow \varepsilon,\\
&\texttt{PARAMS2} \rightarrow \texttt{, EXPRESSION PARAMS2},\\
&\texttt{PARAMS2} \rightarrow \varepsilon\}
\end{aligned}
$$

Čtvrtá komponenta popisuje veškerou práci s výrazy a obsahuje pomocné neterminály pro volání funkcí. Nechť symboly $\lfloor$ a $\rceil$ ohraničují možnosti terminálů, které se v daném pravidle mohou vyskytovat.

$$
\begin{aligned}
P_4 = \{&\texttt{EXPRESSION} \rightarrow \texttt{EXPRESSION} \lfloor \texttt{+ - * / . \&\& || == != > < >= <= =} \rceil \texttt{ EXPRESSION},\\
&\texttt{EXPRESSION} \rightarrow \lfloor \texttt{! -} \rceil \texttt{ EXPRESSION},\\
&\texttt{EXPRESSION} \rightarrow \texttt{variable},\\
&\texttt{EXPRESSION} \rightarrow \texttt{constant},\\
&\texttt{EXPRESSION} \rightarrow \texttt{functionName},\\
&\texttt{EXPRESSION} \rightarrow \texttt{(EXPRESSION)}\}
\end{aligned}
$$

Pátá komponenta pojednává o bloku kódu, který byl vysvětlen v úvodu Kapitoly 5.1. Důvod pro použití samostatné komponenty pro tuto konstrukci, a ne znovupoužití neterminálu `STATEMENT_LIST`, je explicitní zakázání zanořování definic funkcí. V této konstrukci se tedy mohou vyskytovat libovolné příkazy a jejich sekvence, nesmí se tam objevit definice funkce.

$$\begin{aligned} P_5 = \{&\texttt{CODEBLOCK} \rightarrow \texttt{\{ STATEMENTS \}},\\ &\texttt{CODEBLOCK} \rightarrow \texttt{STATEMENT},\\ &\texttt{STATEMENTS} \rightarrow \texttt{STATEMENT STATEMENTS},\\ &\texttt{STATEMENTS} \rightarrow \varepsilon\} \end{aligned}$$

Šestá komponenta obsahuje pouze neterminály související s datovými typy.

$$\begin{aligned} P_6 = \{&\texttt{VOLTYPE} \rightarrow \texttt{TYPE},\\ &\texttt{VOLTYPE} \rightarrow \varepsilon,\\ &\texttt{TYPE} \rightarrow \texttt{int},\\ &\texttt{TYPE} \rightarrow \texttt{float},\\ &\texttt{TYPE} \rightarrow \texttt{string},\\ &\texttt{TYPE} \rightarrow \texttt{bool}\} \end{aligned}$$

Všimněme si, že mezi neterminály `STATEMENT` a `CODEBLOCK` teoreticky může nastat situace, kdy se tyto dva neterminály budou vzájemně derivovat donekonečna, přičemž oba budou čekat na jinou derivaci druhého a nastane *deadlock*. Tato situace je vyřešená implementačně – neterminál `STATEMENT` bude derivovat neterminál `CODEBLOCK` jen a pouze tehdy, bude-li na vstupu terminál `{`.

Indexace neterminálů a LL tabulka

Každému neterminálu z množiny N v Γ je přiřazen index, který odkazuje na různé komponenty $P_i \in \Gamma$, $i \in \{1, \ldots, 6\}$. Každý neterminál má pravidla pro expanzi pouze v jedné komponentě, díky čemuž je možné deterministicky zvolit komponentu, které se bude předávat řízení syntaktické analýzy. Pomocí vstupního terminálu se poté v komponentě zvolí pravidlo, které se použije pro expanzi. Tato indexace lze matematicky popsat zobrazením z množiny neterminálů do podmnožiny přirozených čísel, které korespondují s indexy komponent v konkrétním gramatickém systému. V tomto případě se jedná o zobrazení

$$N \rightarrow \{i : P_i \in \Gamma\}$$

které je pro jednotlivé neterminály definováno v Tabulce 5.2.1.

LL tabulka syntaktického analyzátoru obsahuje kromě obvyklých čísel pravidel uspořádané dvojice *(číslo komponenty, číslo pravidla)*. Samotná tabulka je sestavena pomocí množin *Empty*(α), *First*(α), *Follow*(A) a *Predict*($A \rightarrow \alpha$). První tři množiny se vytvoří standardním způsobem pro každý terminál a neterminál. Množina *Predict*($A \rightarrow \alpha$) se sestavuje pro každé pravidlo, stejně jako u LL gramatik, reprezentované uspořádanou dvojicí. Prediktivní množiny a LL tabulka pro GS Γ popisující jazyk Koubp jsou v příloze.

5.3 Návrh řešení syntaktického analyzátoru

Implementace syntaktické analýzy je silně objektově orientována, veškeré datové struktury jsou reprezentovány třídami. Třídy reprezentující neterminály a terminály mají společnou

Neterminál	Index komponenty
PROGRAM	1
STATEMENT_LIST	1
STATEMENT	2
IF2	2
DECLOREXP	2
RETURNEXP	2
FUNCDEF	3
PARAMS	3
PARAMS2	3
EXPRESSION	4
ARGS	4
ARGS2	4
CODEBLOCK	5
STATEMENTS	5
VOLTYPE	6
TYPE	6

Tabulka 5.2.1: Zobrazovací funkce neterminálů na indexy komponent.

nadtřídu, díky čemuž je možné je ukládat do jednoho zásobníku. Jednu nadtřídu mají také gramatiky, jednotlivé instance jsou poté konstruovány pomocí tovární metody. Analyzátory jsou také reprezentovány třídami, společnou nadtřídu ale nemají.

Kromě níže zmíněných principů figurují v programu pomocné třídy pro prediktivní i precedenční syntaktickou analýzu. Nejsou podrobně popsány, protože se zaměřím především na principy, které jsou těmito třídami implementovány. Podrobnosti lze najít přímo v kódu, případně v Příloze B je ukázán třídní diagram, který obsahuje nejdůležitější třídy, se kterými se v implementaci syntaktické analýzy pracuje.

Práce s komponentami

Všechny komponenty jsou využívány pouze prediktivním analyzátorem, mimo čtvrtou. Ačkoliv téma čtvrté komponenty je analýza výrazů, je využívána jak precedenčním, tak prediktivním analyzátorem. Důvod je zvolený postup prediktivní analýzy volání funkcí – o samotné výrazy bez volání funkcí, tedy pouze výrazy s konstantami a proměnnými, se postará analýza precedenční, správnou posloupnost terminálů pro volání funkcí kontroluje analýza prediktivní.

Předávání řízení prediktivní a precedenční analýzy

Prediktivní a precedenční analýza si navzájem předávají řízení. Prediktivní analyzátor předá řízení precedenčnímu v případě, že narazí na terminál, který patří do analýzy výrazů. Předání naopak, tedy precedenční prediktivnímu, se děje v případě, že precedenční analyzátor potřebuje zkontrolovat, že je posloupnost terminálů u volání funkce syntakticky v pořádku. Volání funkce analyzují oba analyzátory. Samotná struktura volání funkce je zkontrolována prediktivním analyzátorem a výrazy, které se vyskytují jako argumenty volání funkce, jsou zkontrolovány precedenčním analyzátorem.

Díky možnosti neustálého zanořování je potřeba, aby analyzátory mezi sebou komunikovaly. To je realizováno pomocným terminálem `tFuncEnd`, který je vložen na zásobník pro syntaktickou analýzu hned za volání funkce. Tento proces nebere v potaz, že mohou být další vnořené funkce či syntaktická chyba. Zkrátka vloží na zásobník `tFuncEnd` tam, kde je volání funkce ukončeno pravou závorkou `)`. Jakmile prediktivní analyzátor přečte tento terminální symbol, vloží na zásobník další pomocný terminál `tFuncConst`. Ten využívá precedenční analýza pro nahrazení volání funkce, o které se stará prediktivní analýza. Pro implementaci precedenční analýzy bylo do čtvrté komponenty přidáno pravidlo

`EXPRESSION` → `functionConstant`.

Aby precedenční analyzátor mohl s terminálem `tFuncConst` pracovat, je nutné, aby se prediktivní analýza pro volání funkce volala až z analýzy precedenční. Mějme například příkaz:

```
float num = 1.0 + foo(var);
```

V tomto případě je vše očekávané. Precedenční analýza zavolá prediktivní analýzu, ta se postará o volání funkce a na zásobník vloží `tFuncConst`, který precedenční analýza použije v rámci pravidla. Pokud bychom ale měli například toto samo o sobě stojící volání funkce:

```
foo(var + convert_int(60.8));
```

pak analýza tohoto příkazu začne prediktivně, nicméně ihned vidí terminál `tFuncName`, který indikuje volání funkce, a proto předá řízení precedenční analýze. Dále už analýza probíhá stejně, jako u předchozího příkladu. Výraz bude mít jediný operand bez operátoru, a to `tFuncConst`.

Zajímavou částí je předávání řízení prediktivní → precedenční analýza. Nutnou podmínkou je, aby na vrcholu zásobníku byl neterminál `nExpression`. Mohou nastat tři možnosti:

1) Aktuálně není na vstupu terminál `tFuncName`, což znamená, že výraz určitě bude analyzovaný precedenčním analyzátorem a může se předat řízení.

2) Pokud na vstupu je `tFuncName`, pak záleží, zda se analyzuje volání funkce, pro které byl prediktivní analyzátor zavolán. To se pozná podle toho, jestli je `tFuncName` první takový terminál přečten.

   - Při prvním výskytu `tFuncName` víme, že aktuální volání funkce má analyzovat prediktivní analyzátor a nebude se volat precedenční.
   - Jakýkoliv další výskyt `tFuncName` znamená, že jsme narazili na vnořené volání funkce a předá se řízení precedenční analýze pro nový výraz.

Obrázek 5.3.1: Ukázka volání funkce a předávání řízení jednotlivých analyzátorů s rozsahem analýzy, kdy na vstupní pásce jsou již vloženy pomocné terminály pro analýzu volání funkce. Plná čára zobrazuje rozsah prediktivní analýzy, přerušovaná čára rozsah precedenční.

Implementace prediktivní analýzy

Prediktivní analyzátor pracuje primárně s metodami `parseToken` a `parseNonterminal`. Obě dvě metody korespondují s Algoritmem 4.1.6. Metoda `parseToken` se podívá, jestli aktuální terminál na zásobníku je stejný jako terminál na vstupu. Pokud ano, pak vyjme symbol ze zásobníku i ze vstupní pásky a pokračuje dál v analýze vstupního řetězce. Speciální případ je pomocný symbol `tEnd`, který má stejný význam jako `$` v Algoritmu 4.1.6. Když je i na vrcholu zásobníku i na vstupní pásce tento terminál, pak analýza končí úspěchem.

Metoda `parseNonterminal`, pokud nepředává řízení precedenční analýze, se podívá do LL tabulky, která se indexuje pomocí vstupního terminálu a neterminálu na vrcholu zásobníku. Při prázdném políčku tabulky analýza končí chybou, pro neprázdnou uspořádanou dvojici udělá následující kroky:

1) Z uspořádané dvojice si nejprve převezme číslo komponenty, ve které se pravidlo nachází.

2) Pomocí tovární metody `GrammarFactory::CreateGrammar(grammarNumber)` si vytvoří instanci dané komponenty.

3) Předá komponentě číslo pravidla a komponenta vrátí reverzovanou pravou stranu pravidla.

4) Pokud pravá strana pravidla není rovna ε, pak se všechny symboly z expandovaného řetězce vloží na zásobník.

Implementace precedenční analýzy

Precedenční analyzátor pracuje se dvěma zásobníky. První zásobník je společný pro oba analyzátory, přičemž precedenční pouze vyjme neterminál `nExpression`, pokud analýza skončí úspěšně. Druhý je exkluzivně pro precedenční analyzátor a simuluje nejpravější derivaci, respektive analýzu zdola nahoru dle Algoritmu 4.2.1. Označení *zásobník* bude v této podkapitole označovat pouze precedenční zásobník, nebude-li explicitně řečeno *společný zásobník*.

Analýza výrazů začíná vložením pomocného terminálu `tExpEnd` na společný zásobník, což je implementováno pomocí procházení společného zásobníku a přeskakování operátorů a operandů. Terminál `tExpEnd` se vloží za první operand, který za sebou již nemá operátor. Dále se nalezne první terminál na zásobníku jednoduchým procházením zásobníku a vrácením prvního výskytu terminálu. Analyzátor se podívá do precedenční tabulky a najde akci, kterou má udělat, na základě vstupního terminálu a prvního terminálu na zásobníku. V precedenční tabulce se na políčkách mohou vyskytovat tyto symboly:

1) '=': vloží vstupní terminál na zásobník,

2) '<': vloží na zásobník symbol $<$ a poté vstupní terminál,

3) '>': najde první pravou stranu pravidla na zásobníku, zkontroluje platnost a provede redukci,

4) '×': ukončí analýzu syntaktickou chybou.

5.4 Lexikální analýza a nástroj Flex

Pro usnadnění práce byl lexikální analyzátor automaticky vygenerován nástrojem *Flex*[2] ze souboru s lexémy popsanými regulárními výrazy. S každým úspěšně analyzovaným lexémem následuje vložení tokenu do vstupní pásky pro syntaktickou analýzu. Znamená to tedy, že se nejdříve v celém zdrojovém souboru ověří lexikální správnost, až poté správnost syntaktická. Při konstrukci tokenu se inicializuje i jeho datová část, do kterého patří datový typ a samotná data.

Lexikální analýza není předmětem této práce, a proto není detailně popsána. Hlavní důvod k vygenerování lexikálního analyzátoru je lepší testovatelnost výsledného programu na reálných zdrojových kódech.

5.5 Abstraktní syntaktický strom

Ačkoliv se na abstraktní syntaktický strom (AST) nejčastěji nahlíží jako na binární strom, je implementován jako obecně n-ární strom. Jsem přesvědčen, že je to mnohem intuitivnější ke čtení výsledného stromu, který může být programem vygenerován. Pro snadnější pochopení implementace AST je vhodné si projít třídní hierarchii uzlů AST v Příloze C.

Tvorba AST v precedenční analýze

Tvorba AST pro precedenční analýzu pracuje se zásobníkem kontextů pro výrazy. Do toho při každé redukci vloží posledně vytvořený výraz, který se následovně může použít (a nemusí) jako operand při dalších redukcích.

Pro precedenční analýzu je tvorba AST jednodušší. Výrazy jsou totiž maximálně binární, navíc simulace tvorby derivačního stromu zdola nahoru je velmi přirozená. Má-li proběhnout redukce proměnných, konstant nebo *konstant funkcí* (vzpomeňme si na pomocný terminál `tFuncConst` z Kapitoly 5.3), vytvoří se nový `Operand` a vloží se na zásobník kontextů výrazů pro AST. Při redukci unárních či binárních výrazů, kde se již pracuje s neterminály, se pokaždé vyjme výraz ze zásobníku kontextů pro výrazy (dva pro binární výrazy) a vytvoří se nový výraz, do kterého jsou všechny vyjmuté výrazy vloženy jako operandy. Operátor se pak přečte z pravidla použitého pro redukci.

Poslední výraz, který na zásobníku zůstane, je převzán prediktivní analýzou a vložen do zbytku AST pro reprezentaci vstupního zdrojového kódu.

Tvorba AST v prediktivní analýze

Prediktivní analýza také pracuje se zásobníkem kontextů, ale takovým, ve kterém se nenacházejí výrazy, pouze příkazy. Spoléhá na dvě polymorfní metody, které jsou přepsány v uzlech, které dědí od třídy `Statement`. Třídy pro reprezentaci výrazů v AST tuto metodu řešit nemusí, protože si veškeré informace o tokenech mohou převzít ze zásobníku, na kterém provádí redukci a případné výrazy ze zásobníku volání výrazů. Tvorba AST pro prediktivní analýzu také pracuje se zásobníkem kontextů, významově je vrchol zásobníku aktuální blok kódu, případně konstrukce, ke které patří aktuálně zpracovávaný příkaz nebo výraz.

Pro každou expanzi neterminálu, není-li neterminál expandován v ε, se zavolá tovární metoda pro instanciaci uzlů AST, `ASTNodeFactory::CreateASTNode()`. Právě tato metoda

[2] Manuál k programu k dispozici na https://westes.github.io/flex/manual/.

se stará o zahození přebytečných informací z derivačního stromu, protože vytvoří instanci pouze pro ty neterminály, pro které to implementačně bylo definováno. Pokud neterminál má korespondující AST uzel, pak je pomocí polymorfní metody `ASTNode::LinkNode()` spojen se zbytkem stromu a vložen na zásobník kontextů. Spojení se stromem proběhne pro aktuální vrchol zásobníku kontextů a nově vytvořený uzel.

Druhá polymorfní metoda `ASTNode::ProcessToken()` je zavolána pokaždé, je-li na vrcholu zásobníku terminál. Tato metoda se stará o zachování dat v AST ze vstupní pásky, kdy všechna relevantní data, která by byla potřeba pro sémantickou analýzu či generování kódu, jsou uložena do uzlů AST. Je volána pro aktuální vrchol zásobníku kontextů.

Pro zachování logiky stromu musí být uzly ze zásobníku odstraňovány, aby se nově vytvořené uzly spojily s relevantním existujícím uzlem ve stromu. Proto je vnesen nový neterminál `STOP`, který pouze značí konec kontextu pro daný neterminál, který je ze zásobníku kontextů vyjmut. Například pravidlo

```
DECLOREXP → TYPE variable = EXPRESSION ;
```

je změněno na

```
DECLOREXP → TYPE variable = EXPRESSION STOP ;,
```

čímž je jasně řečeno, že jakmile se úspěšně přiřadí levá i pravá strana do uzlu AST pro deklaraci, má se vyjmout ze zásobníku a další příkaz nebo výraz bude spojen s rodičovským uzlem.

5.6 Výstupy programu

Implementovaný program analyzuje, zda vstupní řetězec je řetězcem jazyka Koubp. K jednoduchému výstupu `true` nebo `false` je ale vhodné přidat další výstupy, které jednak potvrdí správnost implementace a jednak dělají rozšiřitelnost programu jednodušší. V případě úspěšné analýzy řetězce program produkuje textový soubor, který obsahuje pravidla každého kroku, ať už expanze či vyjmutí symbolů, obou analyzátorů. Volitelný výstup je graf abstraktního syntaktického stromu ve formátu *.dot*, ze kterého program *dot*[3] může vygenerovat vizualizaci orientovaného grafu ve formátu *.pdf*.

Pro výpis syntaktických chyb je implementován systém, který uschovává sedm posledních terminálů. Nastane-li k chybě, všechny tyto uložené terminály jsou vypsány na standardní chybový výstup, za ně vypíše následující tři terminály na vstupní pásce (nebere ohledy na bílé znaky, vypíše je v řadě za sebe), pokud nějaké zbývají. Aktuální vstupní token, u kterého nastala chyba, označí červeným symbolem ∧ na novém řádku.

5.7 Jazyk, sestavovací systém a spuštění programu

Překladač je implementován v jazyce C++ a pro překlad byl použit standard C++20 společně s přepínači `-Wall` a `-Wextra`, `-pedantic` a `-Wno-unused-parameter`. Poslední zmíněný musí být specifikován, protože třída `ASTNode` deklaruje čistě polymorfní metody `ProcessToken` a `LinkNode`, které pracují s parametry. Tyto metody jsou přepsány ve všech třídách, které dědí od `Statement` a také třídou `StatementList`. Pro třídy dědící od `Expression` jsou tyto metody nepodstatné kromě třídy `FunctionCall`, díky čemuž je

[3]Manuál k programu k dispozici na https://linux.die.net/man/1/dot.

potřeba, aby i třída `Expression` tuto metodu dědila. Protože ne všechny třídy tyto metody přepisují, překladače generují varování, které je třeba potlačit.

Syntaktický analyzátor je naprogramován pro UNIXové operační systémy jako konzolová aplikace. Přeložení programu a spuštění testů proběhlo na těchto operačních systémech s těmito překladači:

- Fedora 40 KDE, překladače `gcc` verze 14.0.1 a `clang` verze 18.1.1,
- Apple Sonoma 14.4.1 s překladačem `Apple clang` verze 1500.3.9.4 – za vyzkoušení děkuji Ondřeji Lukáškovi,
- Linuxové distribuce na WSL (*Windows Subsystem for Linux*[4]):
  - Ubuntu 22.04 s `gcc` verze 11.4.0 – za vyzkoušení děkuji Adamu Malysákovi,
  - Debian 11
- Školní server Merlin – CentOS 7 za použití překladače `gcc` verze 10.5; překladač `clang` generuje chybu, protože nedokáže najít standardní knihovny `<cstddef>` a `<string>`.

Pro přeložení a plnou funkcionalitu programu se všemi výstupy je nutné mít na systému nainstalován překladač C++ a nástroje `Make`, `CMake` a `dot`. Dále byl při implementaci použit nástroj `valgrind` pro analýzu správy paměti a nástroj `Flex` pro generování lexikálního analyzátoru.

Sestavovací systém je CMake, což je nástroj pro generování souborů, které poté zařizují překlad programu. Byly vyzkoušeny varianty s překladovými systémy `Ninja` a `Make`. Protože práce v příkazové řádce s tímto nástrojem není nejrychlejší, vytvořil jsem pro jednodušší ovládání v kořenovém adresáři `Makefile`. Tam lze nalézt několik cílů, které se dají spustit:

- `all` – přeloží program a vytvoří spustitelný soubor `Parser` pomocí invokace systému CMake Dále přeloží zdrojové soubory pro jednotkové testy a vytvoří z nich spustitelný soubor `ParserTest`.
- `gcc` – přeloží program pomocí překladače z rodiny `gcc`.
- `clang` – přeloží program pomocí překladače z rodiny `clang`.
- `run` – podívá se, zda je k dispozici spustitelný soubor `Parser`. Pokud ano, spustí jej s parametry, které se přímo v `Makefile` dají upravit. Jinak hlásí chybu.
- `runtest` – spustí program `ParserTest` pro jednotkové testy a následně skript `test.sh` popisující testy chování. Pokud spustitelný soubor `ParserTest` neexistuje, hlásí chybu.
- `tree` – je-li soubor *.dot* vygenerovaný programem `Parser` v kořenovém adresáři projektu, vytvoří z něj soubor *.pdf* s abstraktním syntaktickým stromem.
- `valgrind` – spustí program `valgrind` na přeloženém programu `Parser`. Vygeneruje v kořenovém adresáři soubor *valgrind.log* s informacemi o správě paměti programu.
- `valgrind_test` – spustí program `valgrind` na programu `ParserTest`.
- `thesis` – přeloží LaTeXové zdrojové soubory této práce a vygeneruje ji ve formátu *.pdf*. Pro tuto akci je nutné mít lokálně stáhnuté balíčky pro práci se systémem LaTeX.

[4] Podrobně se o WSL píše například na https://learn.microsoft.com/en-us/windows/wsl/about.

- `clean` – vyčistí všechny nepotřebné soubory po sestavování programu či této práce.

Program `Parser` podporuje tři přepínače, a to `-d`, `-t` a `-f`. Přepínač `-d` byl určen pro vývoj aplikace, zajišťoval výpis různých informací o stavu programu na standardní výstup, nicméně ve finálním programu je stále podporován. Přepínač `-t` povolí vygenerování struktury abstraktního syntaktického stromu do *.dot* souboru. Přepínač `-f` je jediný povinný a má povinný parametr `FILENAME`, který specifikuje název souboru se zdrojovým kódem, který je programem analyzován. Není podporován vstup zdrojového kódu ze standardního vstupu. Výchozí `Makefile` je nastaven na generování stromu a vstup ze souboru `input.koubp`, implicitně je tedy program spouštěn tímto způsobem:

```
./build/src/Parser -f input.koubp -t
```

Přepínače pro spuštění programu se dají v `Makefile` libovolně měnit.

5.8 Testování

Program `Parser` je testován jednotkovými testy a testy chování. Pro jednotkové testování byl použit testovací framework *googletest*[5], který je asi nejznámějším testovacím nástrojem pro C++. Uživatel nemusí řešit stahování tohoto frameworku, jeho instalace je zahrnuta v sestavování programu. Stačí tedy přeložit program příkazem `make` a následně spustit testy příkazem `make runtest` v kořenovém adresáři projektu.

Stejným příkazem se zároveň spouští testy chování, respektive skript `test.sh`. Skript `test.sh` spouští program `Parser` a na vstup přivádí zdrojové soubory v jazyce Koubp. Poté se kontroluje návratový kód programu a porovnává se s očekávaným návratovým kódem, který je definován v příslušném souboru se stejným názvem, avšak s příponou *.koubp.rc* (*return code*) místo *.koubp*.

Jednotkové testy

Jednotkové testy existují pro třídy důležité pro syntaktickou analýzu a generování abstraktního syntaktického stromu. Nejprve testují metody pomocných tříd analyzátorů, například korektní provedení redukcí, expanzí, hledání terminálů a pravidel a podobně. Pro abstraktní syntaktický strom je testována korektní instanciace různých druhů uzlů a jejich vzájemné propojování.

Dále jsou testovány samotné analyzátory s příkazy a výrazy na jejich vstupu. Z příkazů testují pouze velmi jednoduché a základní konstrukce jako sekvence terminálů, pokročilejší testy jsou zdrojové kódy v jazyce Koubp, které jsou součástí testů chování. Výrazy jsou testovány od jednoduchých jednoproměnných až po velmi složité graduálně, také pouze jako sekvence terminálů. V obou případech jsou testovány syntakticky jak korektní, tak nekorektní případy.

Testy chování

Tyto testy zkoumají, jestli program `Parser` vrací očekávanou návratovou hodnotu po spuštění analýzy různých zdrojových kódů. Zdrojové kódy, na kterých je program testován, se nachází ve složce `test/koubp_files/`. Tato složka je rozdělena na další tři podsložky: základní konstrukce jazyka v `basic_constructs`, jednoduché algoritmy v podsložce

[5] Návod například na http://google.github.io/googletest/.

`simple_algorithms/`, kódy se syntaktickými chybami v `syntax_error_files/` a syntakticky korektní, ale trochu obskurní až ezoterické kódy v `extreme_cases/`. Všechny složky obsahují zdrojové kódy s různými návratovými hodnotami.

Kapitola 6

Závěr

Cílem této práce bylo zkoumat různé gramatické systémy a navrhnout nový, pro který se následně implementuje syntaktický analyzátor. Byl navrhnut CD gramatický systém, který má stejnou generativní sílu jako LL gramatika a popisuje jazyk, který byl nazván Koubp. Pro tento gramatický systém byl vytvořen syntaktický analyzátor, který pracuje podobně jako běžný analyzátor LL gramatik. Úprava pro CD gramatický systém spočívala v představení upravené LL tabulky, která místo čísla pravidel obsahuje uspořádané dvojice, které obsahují číslo komponenty systému a číslo pravidla v komponentě. Díky této úpravě bylo třeba indexovat neterminály čísly komponent.

Cíl byl splněn a výsledkem je funkční syntaktický analyzátor, který přijímá zdrojové kódy v jazyce Koubp, dává výstup v podobě úspěch/neúspěch a generuje posloupnost pravidel, která byla pro analýzu použita. Bylo implementováno rozšíření zadání v podobě tvorby abstraktního syntaktického stromu.

Práce byla zahájena úvodem do formálních jazyků, kde byly vysvětleny všechny potřebné teoretické koncepty pro pochopení této práce i s doprovodnými příklady. Jedná se o Kapitolu 2. Poté proběhlo seznámení s teorií gramatických systémů v Kapitole 3. Kapitola 4 pojednává o základních konceptech syntaktické analýzy, jako je prediktivní a precedenční syntaktická analýza a datové struktury s nimi spojeny. Dále byl navržen CD gramatický systém popisující jazyk Koubp, který je popsán v Kapitole 5.2, a pro něj implementovaný syntaktický analyzátor v jazyce C++. Tato implementace byla otestována několika desítkami jednotkových testů a testy chování, které zkoušely rozsáhlejší zdrojové kódy, čímž je splněn bod 5 zadání této práce.

Na práci je možné pokračovat různými způsoby. CD gramatické systémy mohou mít větší generativní sílu než bezkontextové gramatiky, pokud jsou jejich komponenty bezkontextové a vhodně použité. Případně je možnost změnit typ systému a použít týmové CD gramatické systémy, PC gramatické systémy a jiné. Je taky možné přidat další komponenty do již existujícího systému, případně upravit pravidla existujících komponent, a tím rozšířit jazyk Koubp o různé nové vlastnosti či rysy. Nebo například určit jiný derivační režim tohoto gramatického systému a příslušně upravit syntaktickou analýzu. Také se může doimplementovat tabulka symbolů, sémantická analýza, generování mezikódu a jeho interpret, případně různé jiné části překladače, kterými se tato práce nezabývala.

Na tuto práci bych rád navázal prací diplomovou. V té bych hlouběji zkoumal použitelné gramatické systémy a jejich metody syntaktické analýzy.

Literatura

[1] Harag, M. *Paralelní gramatické systémy: teorie, implementace a aplikace*. Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Meduna, A.

[2] Hopcroft, J. E., Motwani, R. a Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. 3. vyd. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321455363.

[3] International Organization for Standardization (ISO). *ISO/IEC 9899:2018, Programming languages - C*. International Organization for Standardization (ISO), 2018. Dostupné z: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2310.pdf.

[4] Kari, L. a Lutfiyya, H. CHAPTER 2 PARALLEL COMMUNICATING GRAMMAR SYSTEMS Bringing PC Grammar Systems Closer to Hoare ' s CSP ' s 1. In:. 2011. Dostupné z: https://api.semanticscholar.org/CorpusID:2392372.

[5] Kožár, T. a Meduna, A. *Automata: Theory, Trends, And Applications*. 1. vyd. World Scientific Publishing Co Pte Ltd, 2023. 1–418 s. ISBN 978-981-1278-12-9. Dostupné z: https://www.fit.vut.cz/research/publication/13114.

[6] Kunda, M. *Systémy syntaktických analyzátorů*. Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Meduna, A.

[7] Meduna, A. *Formal Languages and Computation: Models and Their Applications*. Taylor & Francis, 2014. ISBN 9781466513457.

[8] Meduna, A. *Deep Pushdown Automata* [online]. 2007. Ústav Informačních Systémů, FIT VUT v Brně. Dostupné z: https://www.fit.vutbr.cz/~meduna/work/lib/exe/fetch.php?media=lectures:phd:rgd:slides:deep_pda.pdf.

[9] Meduna, A. *Elements of Compiler Design*. 1. vyd. Taylor & Francis Informa plc, 2008. 304 s. Taylor and Francis. ISBN 978-1-4200-6323-3. Dostupné z: https://www.fit.vut.cz/research/publication/8538.

[10] Meduna, A. a Lukáš, R. *Modely bezkontextových jazyků* [online]. 2017. Ústav Informačních Systémů, FIT VUT v Brně. Dostupné z: https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj06-cz.pdf.

[11] Meduna, A. a Lukáš, R. *Syntaktická analýza shora dolů* [online]. 2017. Ústav Informačních Systémů, FIT VUT v Brně. Dostupné z: https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj07-cz.pdf.

[12] Meduna, A. a Lukáš, R. *Syntaktická analýza zdola nahoru* [online]. 2017. Ústav Informačních Systémů, FIT VUT v Brně. Dostupné z: https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj08-cz.pdf.

[13] Meduna, A. a Zemek, P. *Regulated Grammars and Automata*. 1. vyd. New York, NY: Springer Science+Business Media, 2014. 694 s. ISBN 978-1-4939-0368-9, 978-1-4939-0369-6, 978-1-4939-4316-6.

[14] Rozenberg, G. a Salomaa, A., ed. *Handbook of Formal Languages: Linear Modeling: Background and Application*. 1. vyd. Berlin Heidelberg: Springer, 1997. ISBN 978-3-642-08230-6.

[15] Rozenberg, G. a Salomaa, A. *Handbook of Formal Languages: Volume 1. Word, Language, Grammar*. Springer, 1997. Handbook of Formal Languages. ISBN 9783540604204.

[16] Techet, J., Masopust, T. a Meduna, A. *Cooperating Distributed Grammar Systems* [online]. 2007. Ústav Informačních Systémů, FIT VUT v Brně. Dostupné z: http://www.fit.vutbr.cz/~meduna/work/lib/exe/fetch.php?media=lectures:phd:tid:frvs:09-cdgspres.pdf.

[17] Techet, J., Masopust, T. a Meduna, A. *Parallel Communicating Grammar Systems* [online]. 2007. Ústav Informačních Systémů, FIT VUT v Brně. Dostupné z: http://www.fit.vutbr.cz/~meduna/work/lib/exe/fetch.php?media=lectures:phd:tid:frvs:10-pcgspres.pdf.

[18] Vaszil, G. Various communications in PC grammar systems. *Acta Cybernetica*. 1. vyd. Szeged, HUN: Institute of Informatics, University of Szeged. květen 1998, sv. 13, č. 2, s. 173–196. ISSN 0324-721X.

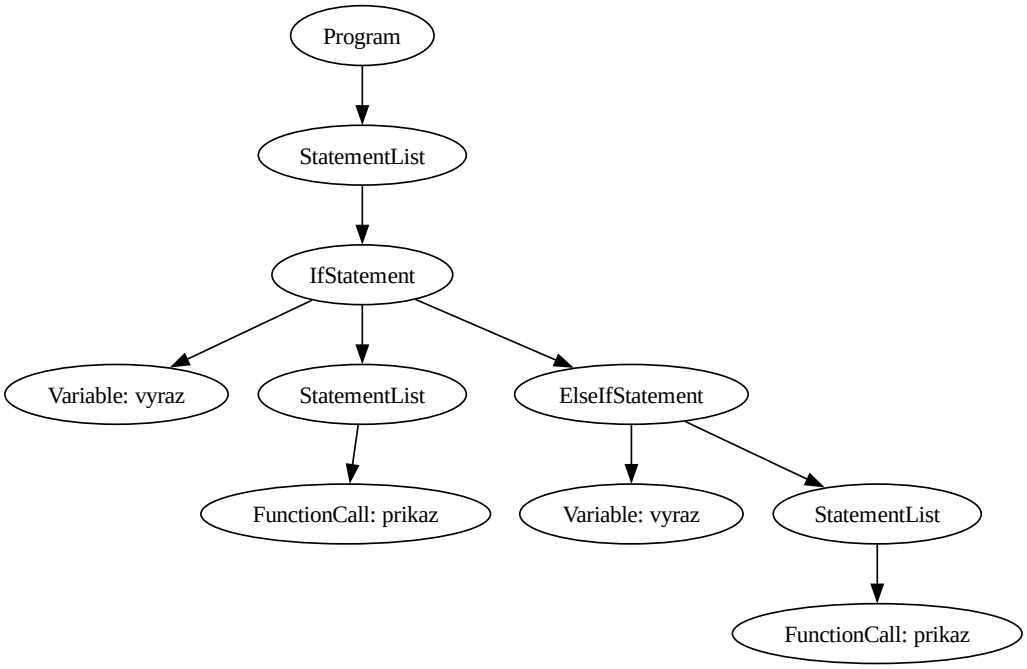
[19] Češka, M., Vojnar, T., Rogalewicz, A. a Smrčka, A. *Teoretická informatika: Studijní opora* [online]. Srpen 2020. Ústav Inteligentních Systémů, FIT VUT v Brně. Dostupné z: https://www.fit.vutbr.cz/study/courses/TIN/public/Texty/TIN-studijni-text.pdf.

Příloha A

Příklady zdrojových kódů v jazyce Koubp a jejich AST reprezentace

A.1 Příkaz if-elseif

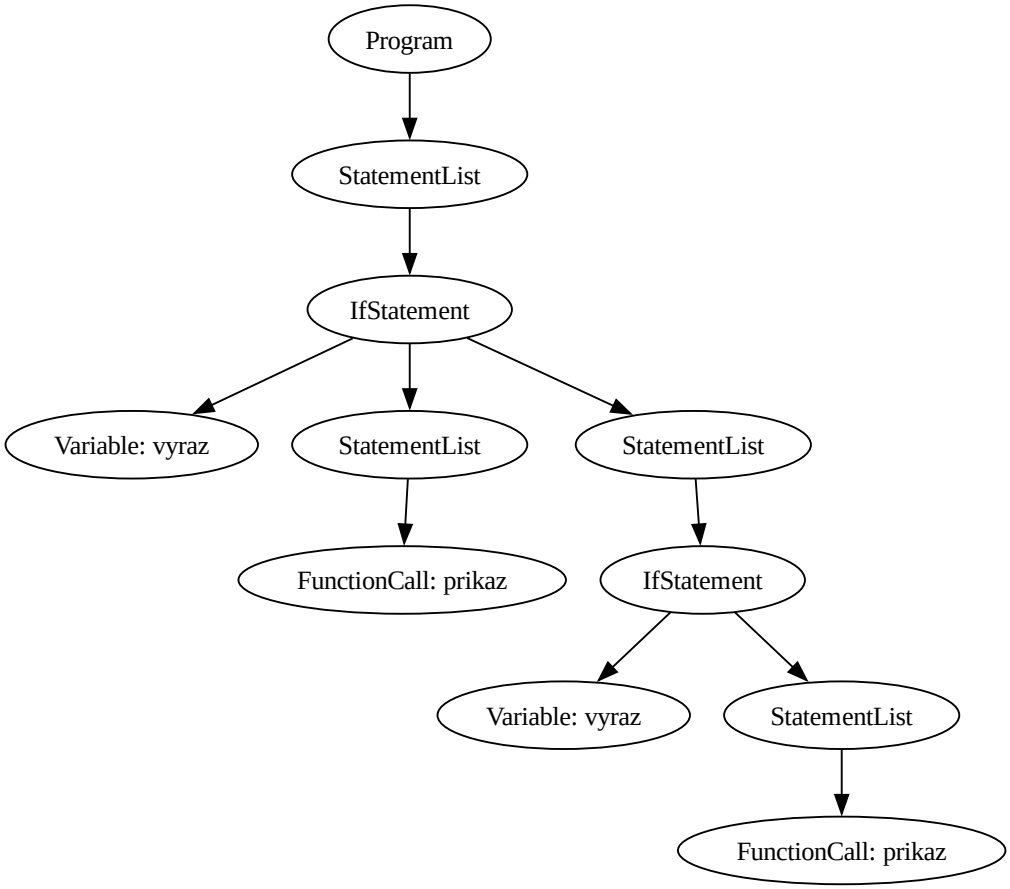
```
if (vyraz)
    prikaz();
elseif(vyraz)
    prikaz();
```



Obrázek A.1.1: Programem vygenerovaný AST pro konstrukci `if-elseif`

A.2 Příkaz if-else if

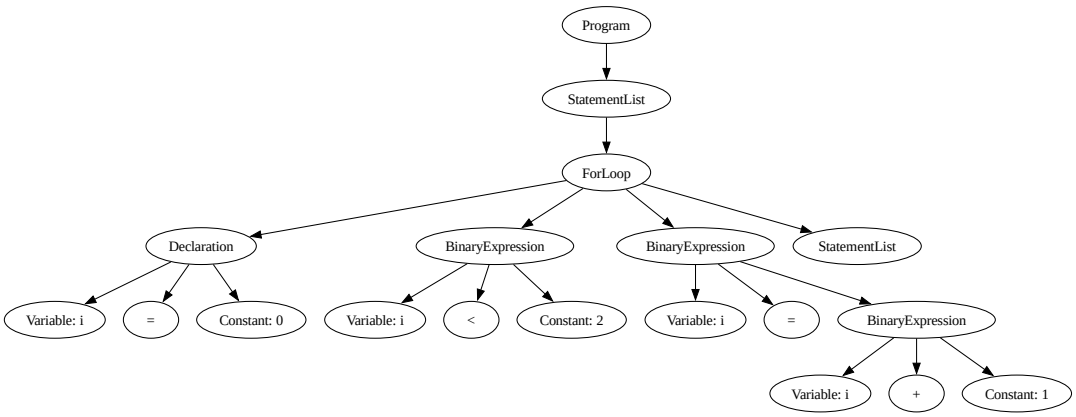
```
if (vyraz)
    prikaz();
else if(vyraz)
    prikaz();
```



Obrázek A.2.1: Programem vygenerovaný AST pro konstrukci `if-else if`

A.3 For cyklus

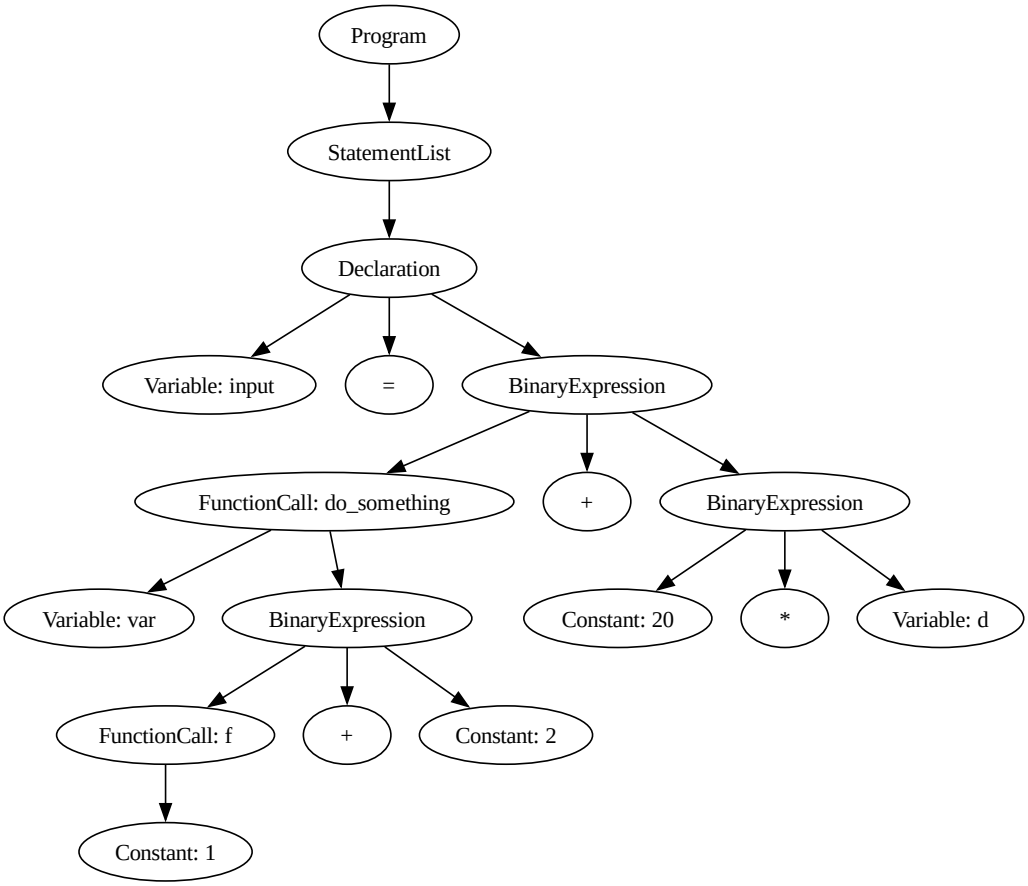
```
for (int i = 0; i < 2; i = i + 1) {}
```



Obrázek A.3.1: Programem vygenerovaný AST pro cyklus `for`.

A.4 Deklarace a přiřazení

```
float input = do_something(var, f(1) + 2) + 20.0*d;
```

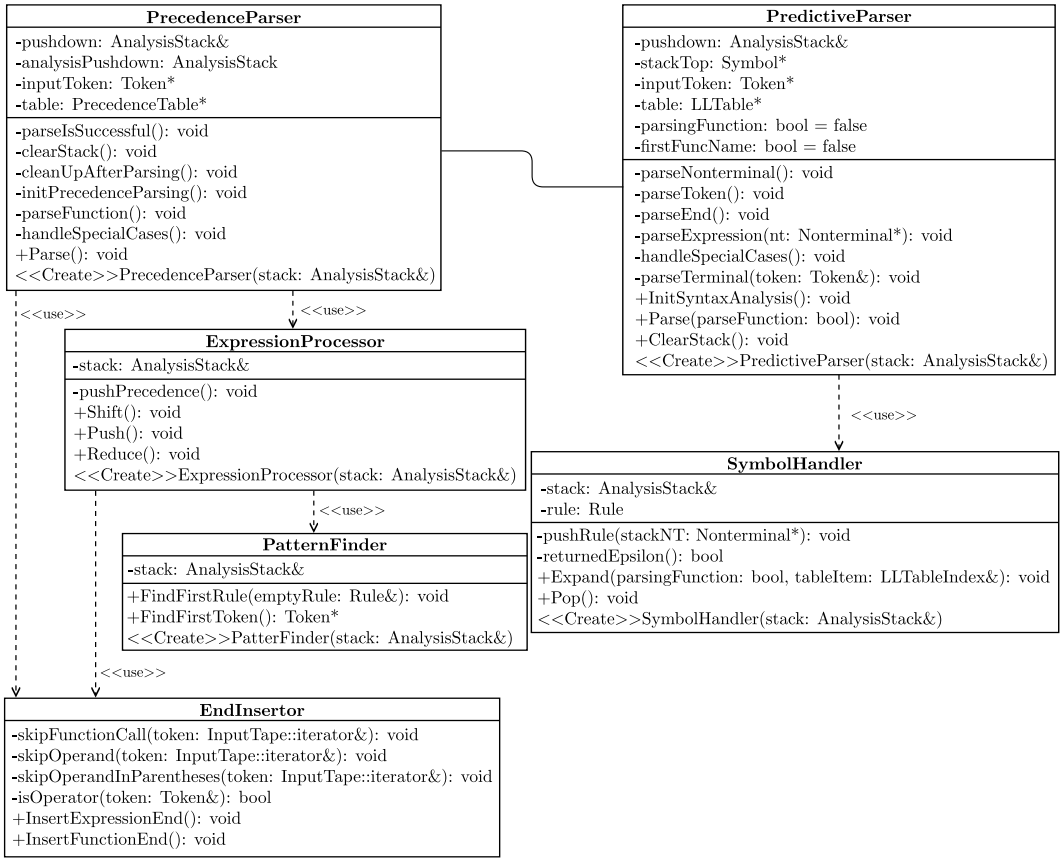


Obrázek A.4.1: Programem vygenerovaný AST pro deklaraci s přiřazením.

Příloha B

Třídní diagram popisující vztahy tříd provádějící syntaktickou analýzu

Diagram obsahuje pouze třídy, které se starají o algoritmus syntaktické analýzy, respektive simulaci zásobníkového automatu. Další pomocné třídy, jako například LL tabulka nebo precedenční tabulka, nejsou do grafu přidány kvůli přehlednosti vzhledem k velikosti diagramu.

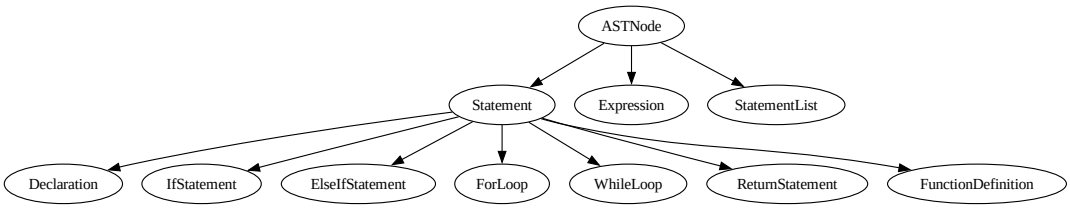


Obrázek B.0.1: Vztahy mezi třídami, které spolupracují na syntaktické analýze.

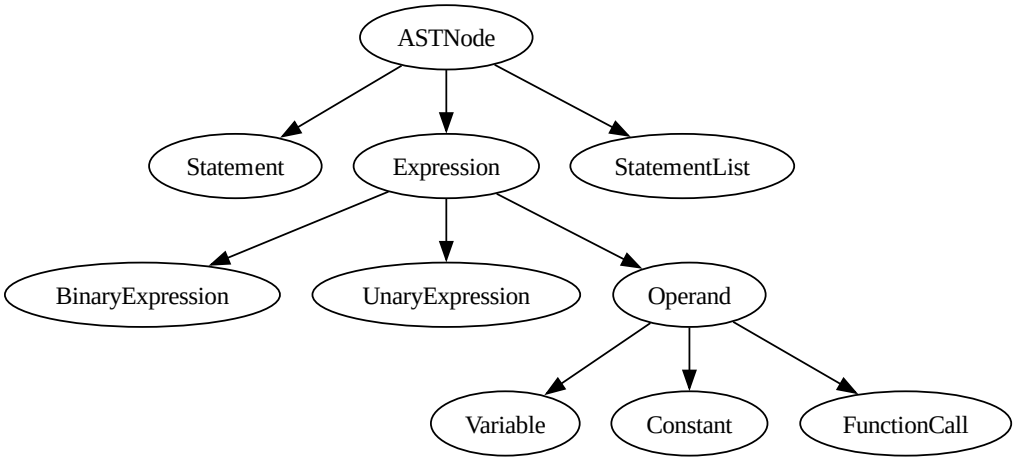
Příloha C

Třídní hierarchie uzlů AST

Pro větší přehlednost je hierarchie rozdělena do dvou obrázků.



Obrázek C.0.1: Třídní hierarchie s třídami reprezentujícími jednotlivé příkazy v AST.



Obrázek C.0.2: Třídní hierarchie s třídami reprezentujícími jednotlivé typy výrazů v AST.