VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Program pro analýzu bezkontextových gramatik

A Program for Analysis of Context-free Grammars

Bc. Daniel Merta

Diplomová práce

Vedoucí práce: doc. Ing. Zdeněk Sawa, Ph.D.

Ostrava, 2022

VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Daniel Merta**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Program pro analýzu bezkontextových gramatik
A Program for Analysis of Context-free Grammars

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem diplomové práce je vytvořit program pro práci s bezkontextovými gramatikami. Program by měl umožňovat provádění různých druhů operací s gramatikami a zjišťování jejich různých vlastností. Mimo jiné by měl program umožňovat například počítaní množin FIRST a FOLLOW, výpočet LR položek, detekci typu gramatiky (LR(1), LALR, SLR, LL, apod.), detekci různých druhů konfliktů, konstrukci zásobníkového automatu, apod. Bylo by vhodné, aby program umožňoval načítat gramatiky ve formátu používaném některým generátorem parserů (např. ve formátu používaném programem Menhir).

1. Nastuduje příslušnou problematiku.
2. Navrhněte a implementujte program pro práci s bezkontextovými gramatikami.
3. Demonstrujte činnost vašeho programu na vhodně zvolených příkladech.

Seznam doporučené odborné literatury:

[1] D. Grune, C.J.H. Jacobs. Parsing Techniques: A Practical Guide. Second Edition, Springer-Verlag, 2008.
[2] D.E. Knuth. On the translation of languages from left to right. Information and Control, 8(6), 1965, pp. 607–639.
[3] D.E. Knuth. Top-Down Syntax Analysis. Acta Informatica, vol. 1, Springer, 1971, pp. 79-110.
[4] F.L. DeRemer. Simple LR(k) Grammars. Communications of the ACM, 14(7), 1971, pp. 453-460.
[5] F.L. DeRemer, T. Pennello. Efficient computation of LALR(1) look-ahead sets. ACM Transactions on Programming Languages and Systems, 4(4), 1982, pp. 615–649, .

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Ing. Zdeněk Sawa, Ph.D.**

Datum zadání: 01.09.2021

Datum odevzdání: 30.04.2022

doc. Ing. Petr Gajdoš, Ph.D.	prof. Ing. Jan Platoš, Ph.D.
vedoucí katedry	*děkan fakulty*

Abstrakt

V rámci této diplomové práce vznikl program, který umí analyzovat určité vlastnosti bezkontextových gramatik a také s nimi provádět různé operace. Příslušná problematika a související informace jsou popsány v teoretické části práce. Další samostatné kapitoly se věnují konkrétnímu návrhu tohoto programu a jeho implementaci. V závěru je funkčnost programu demonstrována na příslušných bezkontextových gramatikách. Možných funkcí, vlastností a operací nad gramatikami existuje obrovské množství. Tento program umožňuje výpočet množin $FIRST$ a $FOLLOW$, provádění redukce gramatiky, odstranění epsilon-pravidel a nebo převod na ekvivalentní zásobníkový automat. Dále umí program detekovat $LR(0)$ a $LL(1)$ typy gramatik, počítat LR položky a detekovat konflikty. Od počátku byl program navrhován tak, aby jej bylo možné v budoucnu rozšířit o novou funkcionalitu.

Klíčová slova

bezkontextová gramatika; syntaktická analýza; syntaktický analyzátor; LL(k); LR(k); zásobníkový automat; Python

Abstract

Within this diploma thesis, there was created a program that can analyze certain properties of context-free grammars and also perform various operations on them. The relevant theory and information on this topic are described in the theoretical part of the thesis. Next chapters are devoted to the particular design of this program and its implementation. Finally, the functionality of the program is demonstrated on several examples of context-free grammars. There exist a huge number of possible functions, properties and operations over grammars. This program can compute $FIRST$ and $FOLLOW$ sets, perform grammar reduction, removal of epsilon rules or transform it to an equivalent pushdown automaton. The program can also detect $LR(0)$ and $LL(1)$ types of grammars, compute LR items and detect conflicts. From the beginning, the program was designed so that it could be extended with new functionality in the future.

Keywords

context-free grammar; syntax analysis; parser; LL(k); LR(k); pushdown automaton; Python

Poděkování

Chci tímto poděkovat vedoucímu mé práce, panu doc. Ing. Zdeňku Sawovi, Ph.D., za všechny konzultace, odborné vedení, cenné rady a poskytnuté materiály, díky nimž tato práce mohla vzniknout. Rovněž děkuji manželce a našim dětem za vřelou podporu a trpělivost.

Obsah

Seznam použitých zkratek a symbolů

CFG	–	Bezkontextová gramatika
PA	–	Zásobníkový automat
LR	–	LR analyzátor
SLR	–	Jednoduchý LR analyzátor
LALR	–	LALR analyzátor
CLR	–	Kanonický LR analyzátor

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Bezkontextová gramatika (angl. „*Context-Free Grammar*“, zkráceně CFG) je určitým typem formální gramatiky, která slouží k popisu formálních jazyků. Pojmenování vychází z přirozených jazyků, které pro svůj popis rovněž používají gramatiky. Tyto jazyky spadají do oblasti teoretické informatiky, která poskytuje formální základy a také nástroje pro praktické využití zejména v programování a softwarovém inženýrství. Jedním z hlavních využití bezkontextových gramatik je popis syntaxe programovacích jazyků.

V dnešní době je každý software naprogramován nějakým konkrétním programovacím jazykem. Programovací jazyk má svou specifickou syntaxi, kterou lze definovat právě pomoci CFG, nebo alespoň využít nějaké její vlastnosti či operace s ní spojené. CFG nachází využití v překladačích těchto programovacích jazyků. Program vytvořený v rámci této práce umožňuje provádět různé operace nad gramatikami a zjišťovat jejich vlastnosti. Umožňuje rovněž převedení gramatiky na ekvivalentní zásobníkový automat.

Tato práce je členěna do několika částí. Po úvodní části jsou v kapitole 2 popsány související teoretické informace, jejichž pochopení je nutným předpokladem pro práci s vytvořeným programem. Teoretická část staví na několika publikačních zdrojích a čtenáři dává možnost seznámit se s formálními definicemi okolo CFG, jejími vlastnostmi, či specifickými tvary, ve kterých se může nacházet. Dále je zde popsána ekvivalence se zásobníkovými automaty, využití při výpočtech pomocných množin a také možné rozdělení gramatik do tříd. Poslední zmíněná část je sice nejdelší, avšak ani tak nepokrývá všechny možné typy gramatik, kterých existují spousty. Věnuje se pouze nejčastěji používaným typům.

Další kapitoly již přímo souvisejí s praktickou částí práce. V kapitole 3 je uveden návrh programu, jeho uživatelského rozhraní a funkcí, které umí provádět. Rovněž je zde z programátorského hlediska popsána struktura programu, formát CFG, se kterou program pracuje a také lexikální analyzátor, což je jedna z důležitých částí programu, která je využita při načítání gramatiky ze vstupního souboru. Následuje kapitola 4 popisující konkrétní implementace jednotlivých programových funkcí s úryvky kódu zajímavých částí řešení. Demonstraci použití programu na konkrétních příkladech je

věnována kapitola 5. Jsou zde názorné ukázky všech funkcí programu, které lze nad CFG vykonávat. Poslední kapitola 6 shrnuje dosažené výsledky programu, který v rámci této práce vznikl. Rovněž popisuje možnosti jeho využití či rozšíření do budoucna.

Aplikací, rozšiřujících modulů či různých pluginů podobného účelu, jaký má program popsaný v této práci, již bylo v několika programovacích jazycích vytvořeno vícero. Většinou jsou tyto programy vytvořeny za konkrétním účelem, jako například *pyCFG* [1], který umožňuje různé operace s CFG, nebo knihovna *Pyformlang* [2] pro manipulaci s formálními gramatikami či *firstFollowPredict* online kalkulátor pro výpočty množin $FIRST$, $FOLLOW$ a $PREDICT$ [3]. Cílem těchto nástrojů je téměř vždy buďto demonstrovat nějaké specifické operace nebo ověřovat konkrétní vlastnosti CFG, případně je využívat v kombinaci s dalšími algoritmy. Tento program umí počítat několik základních operací s CFG, lze jej snadno rozšířit o další novou funkcionalitu a tedy by mohl sloužit i jako vhodný pilíř pro vývoj aplikace s konkrétnějším účelem.

Kapitola 2

Bezkontextové gramatiky

Tato kapitola podává základní přehled pojmů bezkontextových gramatik a související teoretické informace v kontextu této práce. V první podkapitole 2.1 je uvedena formální definice CFG. Následuje podkapitola 2.2, jež popisuje některé vlastnosti, které CFG mají. Dále jsou představeny kanonické tvary v 2.3, ekvivalence CFG a zásobníkového automatu v 2.4, výpočty pomocných množin v 2.5 a třídy gramatik v poslední podkapitole 2.6.

Bezkontextová gramatika spadá do oblasti teorie formálních jazyků. Tato oblast vznikla na základě lingvistiky jako způsob, jak porozumět syntaktickým zákonitostem přirozených jazyků. V informatice se formální jazyky používají mimo jiné jako základ pro definování gramatiky programovacích jazyků a formalizovaných verzí podmnožin přirozených jazyků, v nichž slova jazyka představují pojmy, které jsou spojeny určitými významy nebo sémantikou.

Bezkontextové gramatiky mají své upotřebení v překladačích, nebo též kompilátorech programovacích jazyků. Tyto překladače převádějí algoritmy zapsané ve vyšším jazyce do jazyka nižšího. Příkladem vyššího jazyka je jazyk C, C#, Java, Python a mnoho dalších. Příkladem nižšího jazyka je strojový kód a Assembler. Překlad se obecně dělí na několik fází. S bezkontextovými gramatikami se pojí fáze syntaktické analýzy, jejímž cílem je analýza řetězce tokenů a vytvoření odpovídajícího derivačního stromu.

Formální jazyk je v kontextu informatiky tvořen slovy, jejichž písmena jsou převzata z abecedy. *Abeceda formálního jazyka* je konečná množina symbolů, písmen nebo tokenů, které se spojují do řetězců tohoto jazyka. Jinými slovy, každý řetězec nebo slovo je tvořen konečnou posloupností symbolů této abecedy. Vznik slov se řídí pravidly, pomocí kterých lze každé slovo vygenerovat z počátečního symbolu. Formální jazyky mohou být reprezentovány formálními gramatikami, regulárními výrazy nebo konečnými automaty. Tato práce je zaměřena na bezkontextové gramatiky, které jsou specifickým typem formálních gramatik.

Formální gramatika tedy popisuje, jak z abecedy jazyka vytvořit řetězce, které jsou platné podle syntaxe jazyka. Gramatika nepopisuje význam řetězců ani to, co s nimi lze v jakémkoli kontextu dělat, pouze jejich formu.

V tomto textu budou používány následující konvence, díky nimž lze snadněji rozeznat význam dílčích symbolů:

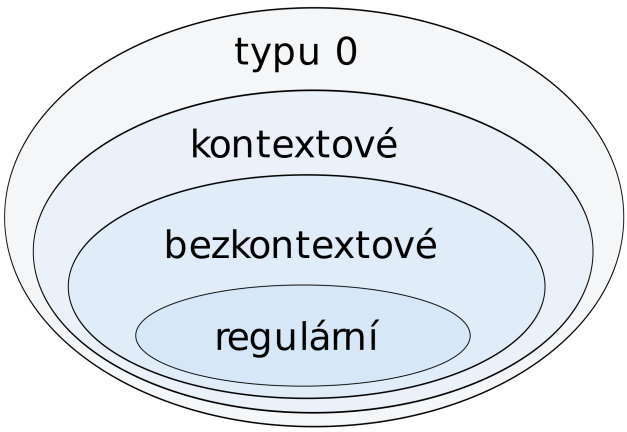
- Terminální symboly se značí malými písmeny, např. *a, b, c,* nebo číslicemi *1, 2,* ...
- Neterminální symboly se značí velkými písmeny, např. *A, B, C,* ...
- Řetězce terminálů se značí *u, v, w,* ...
- Řetězce neterminálů a terminálů se značí malými písmeny řecké abecedy: $\alpha, \beta, \gamma, \ldots$
- Prázdný řetězec se značí ϵ nebo *epsilon.*

V této práci se předpokládá použití uvedených konvencí. To znamená, že nejsou vždy explicitně popsány všechny neterminály, terminály, ale například jen přepisovací pravidla gramatiky pro konkrétnost.

V roce 1956 americký vědec Avram Noam Chomsky formalizoval tyto gramatiky a rozdělil je do čtyř skupin, které jsou známy jako Chomského hierarchie:

- Typ 0 — Všechny formální jazyky bez omezení.
- Typ 1 — Kontextové gramatiky.
- Typ 2 — Bezkontextové gramatiky.
- Typ 3 — Regulární gramatiky. [4]

Rozdíl mezi těmito typy spočívá v tom, že vyšší typ má vždy více omezenější druh pravidel pro tvorbu slov a tedy může vyjadřovat méně formálních jazyků. Tyto vztahy jsou graficky znázorněny na obrázku 2.1. Z praktického hlediska se dnes nejvíce využívají bezkontextové a regulární gramatiky, protože k takovýmto gramatikám lze poměrně snadno zkonstruovat syntaktický analyzátor.



Obrázek 2.1: Chomského hierarchie

2.1 Definice bezkontextové gramatiky

Bezkontextová gramatika G je formálně definována jako uspořádaná čtveřice (Π, Σ, S, P), kde:

1. Π - je konečná množina neterminálních symbolů.
2. Σ - je konečná množina terminálních symbolů, nazývaných taky jako abeceda gramatiky. Průnik mezi Π a Σ je prázdná množina, neboli $\Pi \cap \Sigma = \emptyset$.
3. S - je počáteční neterminální symbol. Neboli $S \in \Pi$.
4. P - je konečná množina přepisovacích pravidel. Neboli $P \subseteq \Pi \times (\Pi \cup \Sigma)^*$. [5]

Pravidla z množiny P můžeme zapisovat ve tvaru $A \Rightarrow \alpha$, kde A je neterminální symbol a α je řetězec, v němž mohou být obsaženy jak neterminální, tak terminální symboly. Z toho vyplývá, že neterminály musí být dále přepsány a terminály už naopak dále přepsat nejdou. [6]

Bezkontextový jazyk je bezkontextovou gramatikou reprezentován jako množina všech řetězců, které bezkontextová gramatika generuje. Řetězec je tvořen posloupností terminálních symbolů, na které se mohou neterminální symboly přepsat. Při aplikaci přepisovacího pravidla se může jedním krokem přepsat sekvence neterminálů na další neterminály nebo terminály.

2.2 Vlastnosti

U bezkontextových gramatik lze zkoumat celou řadu vlastností, které mohou dokazovat různé jevy či být nutnou podmínkou k tomu, aby se dala bezkontextová gramatika použít k určitému účelu. Tato kapitola se věnuje některým z nich, které jsou relevantní v kontextu této práce.

2.2.1 Rekurzivní CFG

Rekurze je důležitý pojem v souvislosti s CFG a značí opakované vnořené volání neterminálních symbolů v přepisovacích pravidlech. Jinými slovy, přepisovací pravidlo je rekurzivní, pokud může odvodit posloupnost obsahující právě tento neterminál. Když není CFG rekurzivní, generuje jen konečný počet řetězců. Následuje příklad nerekurzivní CFG definované těmito pravidly:

- $S \Rightarrow Aa$
- $A \Rightarrow b \mid c$

Z těch lze odvodit, že jazyk generovaný touto CFG je $\{ba, ca\}$ a tudíž je konečný.

Naopak rekurzivní je CFG, když může generovat nekonečný počet řetězců. Příkladem rekurzivní CFG může být gramatika s těmito pravidly:

- $S \Rightarrow SaS$

- $S \Rightarrow b$

Jazyk generovaný CFG s těmito pravidly je nekonečný: $\{b, bab, babab, bababab, \dots\}$. Na základě povahy rekurze v rekurzivní gramatice lze takové CFG dále rozdělit na:

- Levě rekurzivní - obsahuje levou rekurzi, tzn. že se neterminál může objevit na začátku (levém okraji) odvozeného řetězce.

- Pravě rekurzivní - obsahuje pravou rekurzi, tzn. že se neterminál může objevit na konci (pravém okraji) odvozeného řetězce.

- Obecně rekurzivní - obsahuje rekurzi, avšak nemusí obsahovat levou ani pravou rekurzi, nebo naopak může obsahovat oba předchozí typy rekurzí [7][8].

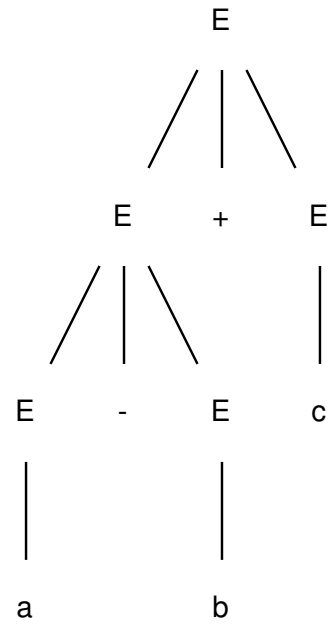
2.2.2 Nejednoznačná CFG

Pro snazší pochopení problematiky nejednoznačnosti u CFG je potřeba vysvětlit pojem derivační strom. V kontextu formálních jazyků je to syntaktická struktura slovního řetězce podle formální gramatiky. Vnitřní uzly stromu jsou označovány jako neterminály a koncové uzly jako terminály. Kořen stromu je počáteční symbol. Na obrázku 2.2 je derivační strom vstupního řetězce $a - b + c$ vygenerovaného CFG $G = (\Pi, \Sigma, S, P)$, kde:
$\Pi = \{E\}$
$\Sigma = \{a, b, c\}$
$S = \{E\}$
$P = E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow a \mid b \mid c$

Gramatika je nejednoznačná, jestliže v jazyce $L(G)$ generovaném touto gramatikou existuje alespoň jeden řetězec u, pro který existují alespoň dva různé derivační stromy, každý s kořenem S a generujícím u. Každému derivačnímu stromu odpovídá levá nebo pravá derivace. Počet různých derivačních stromů řetězce u se nazývá stupeň nejednoznačnosti daného řetězce.

Pokud žádný řetězec vytvořený gramatikou G nemá stupeň nejednoznačnosti větší než k, pak celkový stupeň nejednoznačnosti G je k. V praxi bývají gramatiky klasifikovány na základě jejich stupně nejednoznačnosti. Pokud se počet derivačních stromů zvětšuje s rostoucí délkou řetězců generovaných danou gramatikou, je možné, že stupeň nejednoznačnosti této gramatiky je nekonečný.

Jednoznačná gramatika je taková, pro jejíž jakýkoli vygenerovaný řetězec u existuje pouze jediný derivační strom [9].



Obrázek 2.2: Derivační strom

2.3 Kanonické tvary

Bezkontextová gramatika může být v jistém tvaru, pokud splňuje dílčí pravidla a podmínky daného tvaru. Aby bylo o CFG možné říct, že je v určitém tvaru, je často nutné konkrétním algoritmem ověřit, že podmínky splňuje, případně ji upravit tak, aby po úpravě pravidla daného tvaru splňovala. V této kapitole jsou popsány nejčastější možné tvary gramatik.

2.3.1 Redukovaná CFG

Redukovanou gramatikou se rozumí taková gramatika, která neobsahuje následující dva typy nepoužitelných symbolů:

1. Nenormované symboly, tzn. takové neterminály, které nelze přepsat jakýmkoli počtem kroků na řetězec terminálů.

2. Nedosažitelné symboly, tzn. takové neterminály a terminály, ke kterým se odvozováním z počátečního symbolu není možné dostat [6].

Formální zápis podmínek nutných k tomu, aby byla CFG $G = (\Pi, \Sigma, S, P)$ redukovaná, lze vyjádřit takto. CFG G je redukovaná, pokud pro každé $A \in \Pi$ platí následující tvrzení:

- Existují nějaké řetězce $u, v \in \Sigma^*$ takové, že $S \Rightarrow^* uAv$
- Existuje nějaký řetězec $w \in \Sigma^*$ takové, že $A \Rightarrow^* w$ [10]

Algoritmus redukce bezkontextové gramatiky probíhá tak, že se nejdříve hledá množina $\mathcal{T}$, což je množina neterminálů, které lze přepsat na terminály. Formálně ji lze vyjádřit:

$$\mathcal{T} = \{A \in \Pi \mid (\exists w \in \Sigma^*)(A \Rightarrow^* w)\}[5]$$

Prochází se opakovaně všechna pravidla gramatiky, dokud jsou do množiny přidávány nové neterminály. Následně dochází ke kontrole, že množina $\mathcal{T}$ obsahuje počáteční symbol. Pokud by množina symbol neobsahovala, znamenalo by to, že gramatika není schopna generovat žádné slovo a výpočet by skončil.

V následujícím kroku je zapotřebí odstranit ještě nedosažitelné neterminální symboly. To se dělá tak, že se do množiny $\mathcal{D}$ přidá počáteční symbol a následně se projdou pravidla tohoto neterminálu za účelem zjistit, na jaké jiné neterminály jej lze přepsat. Takto nově nalezené neterminály jsou přidány do množiny $\mathcal{D}$ a hledání pokračuje v pravidlech těchto neterminálů. Hledání pokračuje, dokud jsou do množiny $\mathcal{D}$ přidávány nové dosažitelné neterminální symboly. Množina $\mathcal{D}$ je formálně definována:

$$\mathcal{D} = \{A \in \Pi^{'} \mid (\exists \alpha, \beta \in (\Pi^{'} \cup \Sigma)^*)(S \Rightarrow^* \alpha A \beta)\}[5]$$

Redukovaná gramatika je taková, která obsahuje na levé i pravé straně pouze neterminály z výše nalezené množiny $\mathcal{D}$.

2.3.2 CFG bez ϵ pravidel

Ke každé bezkontextové gramatice G lze vytvořit bezkontextovou gramatiku G' bez ϵ-pravidel takovou, že jazyk generovaný G' je stejný jako jazyk generovaný G, až na nemožnost vygenerovat prázdný řetězec ϵ. Toto tvrzení lze formálně zapsat:

$$L(G') = L(G) - \{\epsilon\}$$

Že CFG $G = (\Pi, \Sigma, S, P)$ je bez ϵ-pravidel lze formálně vyjádřit následujícím způsobem:

- P neobsahuje žádné ϵ-pravidlo, to jest pravidlo ve tvaru $A \Rightarrow \epsilon$
- Nebo v P existuje právě jedno ϵ-pravidlo $S \Rightarrow \epsilon$ a S se nevyskytuje na pravé straně žádného pravidla z P

Algoritmus odstranění ϵ-pravidel funguje tak, že se nejprve zjistí, které neterminály je možné přepsat na ϵ. K tomuto účelu se používá množina $\mathcal{E}$, do níž jsou algoritmem vkládány neterminály přepsatelné na ϵ. $\mathcal{E}$ lze formálně vyjádřit tímto způsobem:

$$\mathcal{E} = \{A \in \Pi \mid A \Rightarrow^* \epsilon\}[5]$$

Algoritmus funguje tak, že opakovaně prochází pravidla gramatiky a hledá taková pravidla, která na pravé straně obsahují pouze neterminální symboly z množiny $\mathcal{E}$. Levé strany takto nalezených pravidel jsou poté vloženy do množiny $\mathcal{E}$. V dalším kroku jsou do gramatiky přidávána nové pravidla tak, že pro každé pravidlo s takovými neterminály na pravé straně přidáváme i všechny variace pravidla s vypuštěním jednoho či více zjištěných neterminálů. V případě, když by tím mělo vzniknout nové ϵ-pravidlo, tak nové pravidlo nepřidáváme. Pokud je nutné upravovat i pravidla pro počáteční symbol, zavádí se nový symbol S' se dvěma pravidly $S' \Rightarrow S \mid \epsilon$. Takto nově vygenerovaná pravidla bezkontextové gramatiky již neobsahují žádná ϵ-pravidla.

2.3.3 Vlastní CFG

CFG G se nazývá vlastní, když splňuje současně následující tři podmínky:

1. Je redukovaná, to znamená, že neobsahuje nepoužitelné symboly. Redukcí se zabývá kapitola 2.3.1.
2. Neobsahuje žádné cykly (2.2.1).
3. Neobsahuje žádná ϵ-pravidla (2.3.2).

2.3.4 Chomského normální forma

CFG $G = (\Pi, \Sigma, S, P)$ je v Chomského normální formě, pokud jsou všechna její pravidla v nějakém z následujících tvarů:

- $A \Rightarrow BC$
- $A \Rightarrow a$
- $S \Rightarrow \epsilon$

kde $A, B, C \in \Pi$, $a \in \Sigma$ a S je počáteční symbol. Třetí pravidlo se může objevit pouze tehdy, je-li ϵ v jazyce, který gramatika generuje, tedy že $\epsilon \in L(G)$. Jestliže třetí pravidlo $\in P$, pak se S nesmí vyskytovat na žádné pravé straně jiných pravidel.

Platí také tvrzení, že každá gramatika v Chomského normální formě je bezkontextová a naopak, každou bezkontextovou gramatiku lze transformovat na ekvivalentní gramatiku, která je v Chomského normální formě a jejíž velikost není větší než druhá mocnina velikosti původní gramatiky [11].

2.3.5 Greibachové normální forma

CFG $G = (\Pi, \Sigma, S, P)$ je v Greibachové normální formě, pokud pravé strany všech přepisovacích pravidel začínají terminálním symbolem, za kterým případně následuje sekvence neterminálních

symbolů. Existuje i méně striktní varianta, která připouští jednu výjimku, a to existenci prázdného řetězce — ϵ v jazyku, který tato gramatika generuje.

Všechna pravidla gramatiky G v Greibachové normální formě mají tvar

$$A \Rightarrow aA_1A_2 \dots A_n$$

kde $A \in \Pi$, $a \in \Sigma$, $A_1A_2 \dots A_n$ je prázdný řetězec, případně posloupnost neterminálních symbolů bez počátečního symbolu a S je počáteční symbol [12].

2.4 Ekvivalence CFG a zásobníkového automatu

Zásobníkový automat je šestice $M = (Q, \Sigma, \Gamma, \delta, q_0, S)$, kde:

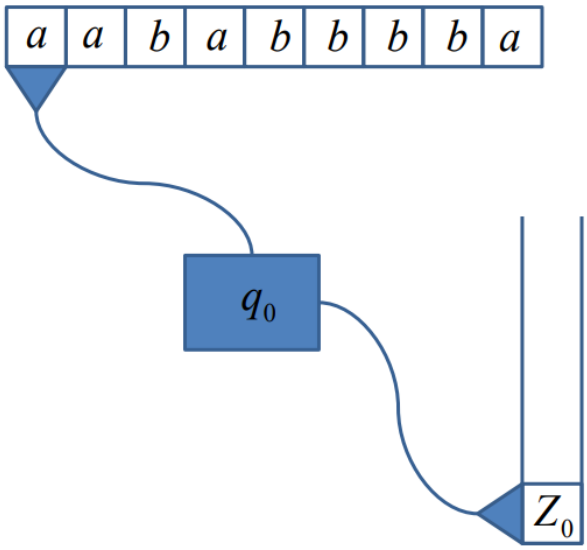
- Q je konečná množina stavů
- Σ je vstupní abeceda
- Γ je zásobníková abeceda
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to (Q \times \Gamma^*)$ je přechodová relace
- $q_0 \in Q$ je počáteční stav
- $Z_0 \in \Gamma$ je počáteční zásobníkový symbol

Zásobníkový automat, neboli PA, se na začátku nachází v počátečním stavu a na zásobníku je počáteční zásobníkový symbol. V každém kroku se podle aktuálního stavu, symbolů na vrcholu zásobníku a symbolu na vstupu provede přechod, při kterém je z vrcholu zásobníku odebrán symbol, ten je nahrazen novými symboly a ze vstupu se přečte další symbol. Tento postup se opakuje až do přečtení celého vstupního řetězce a vyprázdnění zásobníku automatu. Takový stav znamená, že automat řetězec přijal. Pokud automat skončí v nějaké fázi chybou, znamená to, že daný řetězec nepřijímá.

Na obrázku 2.3 je znázorněn zásobníkový automat v počátečním stavu. Automat začíná číst vstupní řetězec *aababbbba* od symbolu a, zásobník obsahuje pouze počáteční symbol Z_0 a řídící jednotka je v počátečním stavu q_0.

Mezi CFG a zásobníkovým automatem platí ekvivalence. Platí tedy, že ke každé bezkontextové gramatice lze sestrojit zásobníkový automat, který rozpoznává stejný jazyk, a naopak. V kontextu této práce je uvažován směr převodu z bezkontextové gramatiky na zásobníkový automat. Důkazem nechť je, že pro CFG $G = (\Pi, \Sigma, S, P)$ vytvoříme zásobníkový automat $M = (Q, \Sigma, \Gamma, \delta, q_0, S)$, kde:

- $\Gamma = \Pi \cup \Sigma$.



Obrázek 2.3: Zásobníkový automat v počátečním stavu [13]

- Pro každé pravidlo $(X \to \alpha) \in P$ z CFG G (kde $X \in \Pi$ a $\alpha \in (\Pi \cup \Sigma)^*$) přidáme do přechodové funkce δ zásobníkového automatu M odpovídající pravidlo:

$$q_0 X \xrightarrow{\epsilon} q_0 \alpha$$

- Pro každý symbol $a \in \Sigma$ přidáme do přechodové funkce δ zásobníkového automatu M pravidlo:

$$q_0 a \xrightarrow{\mathrm{a}} q_0 [13]$$

2.5 Pomocné množiny

S bezkontextovými gramatikami úzce souvisí syntaktické analyzátory. Takových analyzátorů existuje několik druhů a rozlišujeme je podle toho, jakým směrem načítají vstupní řetězec (zleva doprava, zprava doleva), jak postupuje syntaktická analýza (shora dolů, zdola nahoru) nebo jakou konstruují derivaci (levá, pravá).

Dle toho, jakým druhem syntaktického analyzátoru je CFG analyzovatelná, lze určit její typ. Aby mohl být syntaktickým analyzátorem sestaven korespondující derivační strom, vypočítávají se pomocné množiny, jejichž vlastnosti a funkce jsou popsány v této kapitole.

2.5.1 Množina FIRST

Množina $FIRST(\alpha)$ obsahuje terminální symboly, které lze získat derivací z řetězce α. Jedná se tedy o množinu terminálních symbolů, kterými řetězec α může začínat. Množina $FIRST$ je potřebná

pro konstrukci LL parserů.

Nechť G je CFG a α je řetězec tvořený terminálními a neterminálními symboly z G, pak $FIRST(\alpha)$ je funkce dána tímto předpisem:

$$FIRST(\alpha) = \{a \in \Sigma \mid \alpha \Rightarrow^* a\beta, \beta \in (\Pi \cup \Sigma)^*\} \cup \{\epsilon \mid \alpha \Rightarrow^* \epsilon\}$$

Množinu $FIRST$ lze určit pro neterminály, terminály i pro řetězec. Algoritmus výpočtu množiny $FIRST$ je následující:

1. Jestliže je α terminální symbol, výsledná množina obsahuje daný terminální symbol a výpočet končí.

2. Jestli α je neterminální symbol, pak:

   (a) Jestliže α je pravidlo ve tvaru $\alpha \Rightarrow \epsilon$, pak je do výsledné množiny přidán prázdný řetězec ϵ.

   (b) Jestliže α je pravidlo ve tvaru $\alpha \Rightarrow Y_1, Y_2, \ldots, Y_k$, potom se přidá symbol a do množiny FIRST, pokud pro libovolné i platí, že $a \in FIRST(Y_i)$ a ϵ je ve všech množinách $FIRST(Y_1), FIRST(Y_2), \ldots, FIRST(Y_{i-1})$.

   (c) Jestliže $\epsilon \in FIRST(Y_j)$, kde $j = 1, 2, \ldots, k$, potom je do výsledné množiny přidáno i ϵ.

3. Předchozí kroky opakujeme pro všechna pravidla gramatiky tak dlouho, dokud do množiny FIRST přibývají po průchodu nové symboly.

2.5.2 Množina FOLLOW

Množinu $FOLLOW(A)$ je možné počítat pouze pro neterminální symboly, z čehož vyplývá, že A představuje neterminální symbol. Množina $FOLLOW(A)$ je množina obsahující terminální symboly, které se mohou vyskytnout bezprostředně napravo od neterminálu A. Množina $FOLLOW$ se používá jak v LL, tak v LR parserech.

Nechť G je CFG a A je neterminál z G, pak $FOLLOW(A)$ je funkce, pro níž platí následující předpis:

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

Algoritmus výpočtu množiny $FOLLOW$ je následující:

1. Jestliže neterminál A je zároveň počátečním neterminálem S, vloží se do výsledné množiny $FOLLOW$ symbol \$, jenž značí konec vstupního řetězce.

2. Jestliže A je pravidlo ve tvaru $A \Rightarrow \alpha B\beta$, potom se do množiny $FOLLOW(B)$ vloží obsah z $FIRST(\beta)$, kromě ϵ.

3. Jestliže A je pravidlo ve tvaru $A \Rightarrow \alpha B$ nebo $A \Rightarrow \alpha A \beta$, kde $FIRST(\beta)$ obsahuje ϵ, potom jsou přidány všechny prvky z množiny $FOLLOW(A)$ do množiny $FOLLOW(B)$.

4. Kroky opakujeme pro všechna pravidla a pro neterminály pravých stran pravidel, dokud do množiny $FOLLOW$ přibývají nové symboly [14].

2.5.3 Množina PREDICT

Výpočet množiny $PREDICT(A \Rightarrow x)$ udává množinu všech terminálů, jež mohou být v danou chvíli nejlevěji vygenerovány, je-li pro větnou formu použito pravidlo $A \Rightarrow X$.

Nechť G je CFG, pro každé $A \Rightarrow x \in P$ je definována množina $PREDICT(A \Rightarrow x)$ jako:

- Pokud $EMPTY(x) = \{\epsilon\}$, potom $PREDICT(A \Rightarrow x) = FIRST(x) \cup FOLLOW(A)$
- Jinak pokud $EMPTY(x) = \emptyset$, potom $PREDICT(A \Rightarrow x) = FIRST(x)$ [15]

2.6 Třídy gramatik

Úkolem syntaktického analyzátoru je zjistit, je-li možné vstupní text vygenerovat z počátečního symbolu bezkontextové gramatiky. Takovou analýzou se inkrementálně vytváří derivační strom. Podle směru provádění syntaktické analýzy rozlišujeme dva přístupy:

1. Zdola nahoru
2. Shora dolů

Dále se syntaktické analyzátory dělí podle toho, z jaké strany neterminál vždy jako první derivují:

1. Levá derivace
2. Pravá derivace

Levá derivace je strategie, při které se vždy vybere k nahrazení nejprve neterminál nejvíce nalevo. Analogicky se u pravé derivace nahrazuje vždy nejprve neterminál nejvíce napravo. Rozdíl mezi nimi je důležitý, protože každé pravidlo CFG obvykle reprezentuje konkrétní část kódu.

Jestliže je možné z dané CFG odvodit vstupní řetězec nějakým výše uvedeným postupem, pak bezkontextová gramatika patří do dané třídy.

2.6.1 LL(k) gramatiky

Syntaktická analýza shora dolů se snaží sestavit derivační strom od kořene k listům. LL gramatika je bezkontextová gramatika, kterou lze analyzovat LL analyzátorem, který analyzuje vstup zleva doprava a konstruuje levou derivaci. Analyzátor začíná u počátečního symbolu CFG a čte vstupní řetězec zleva doprava a postupuje podle levé derivace.

V názvu $LL(k)$ je první písmeno odvozeno od směru čtení vstupu ((z angl. „***L**eft-to-right*“) a druhé písmeno od provádění levé derivace (z angl. „***L**eftmost derivation*“). k v závorce udává počet následujících symbolů, na které analyzátor nahlédne před tím, než se rozhodne analyzovat předchozí symboly. $LL(k)$ gramatiky generují jazyky typu $LL(k)$.

LL analyzátory jsou založené na tabulkách, podobně jako LR analyzátory. Alternativně lze LL gramatiky charakterizovat jako ty, které lze analyzovat prediktivním analyzátorem. Tedy analyzátorem s rekurzivním sestupem bez zpětného sledování.

Třída jazyků $LL(k)$ tvoří striktně rostoucí posloupnost množin:

$$LL(0) \subset LL(1) \subset LL(2) \subset \ldots [16]$$

Je rozhodnutelné, zda daná CFG G je $LL(k)$, ale není rozhodnutelné, zda libovolná CFG G je $LL(k)$ pro nějaké k. Je také rozhodnutelné, jestli daná $LR(k)$ gramatika je také $LL(l)$ gramatika pro nějaké l [17].

2.6.1.1 LL(0)

$LL(0)$ gramatiky se nehodí pro popis programovacích jazyků, protože neumožňují rekurzi a generují jen konečné jazyky. Pouze čtou vstup a nepotřebují žádné rozhodovací mechanismy na základě dalších symbolů ve vstupním řetězci, jelikož každému neterminálu přísluší právě jedno přepisovací pravidlo.

2.6.1.2 LL(1)

Formálním způsobem lze CFG G označit typem $LL(1)$, pokud každá množina pravidel se stejnou levou stranou $A \Rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$ má tyto vlastnosti:

- $FIRST, FIRST$ — $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \emptyset$ pro všechna $i \neq j$.
- $FIRST, FOLLOW$ — Je-li pro nějaké i $\alpha_i \Rightarrow^* \epsilon$, platí pro všechna $j \neq i$, že $FIRST(\alpha_j) \cap FOLLOW(A) = \emptyset$ [18].

Jedná se o nejpoužívanější typ LL gramatik, kterým i přes určitá omezení stačí k deterministické analýze znát pouze jeden následující symbol, což značně usnadňuje konstrukci analyzátorů. Syntaxe programovacích jazyků jsou často navrženy jako $LL(1)$.

S ohledem na vlastnosti $LL(1)$ gramatik existují dva hlavní typy konfliktů v $LL(1)$ gramatikách:

1. $FIRST/FIRST$ je konflikt, kdy se množiny $FIRST$ dvou různých přepisovacích pravidel pro stejný neterminál vzájemně prolínají.
2. $FIRST/FOLLOW$ je takový konflikt, kdy množina $FIRST$ přepisovacího pravidla obsahuje ϵ a průnik s jeho množinou FOLLOW je neprázdný [19].

Parsovací tabulka $LL(1)$ se konstruuje tak, že každému terminálnímu symbolu odpovídá právě jeden řádek a každému neterminHilfinger2018álnímu symbolu, včetně symbolu označujícího konec vstupu, právě jeden sloupec. Každé buňce takové tabulky může být přiřazeno číslem maximálně jedno přepisovací pravidlo. Pokud by buňka odkazovala na dvě či více přepisovacích pravidel, nejednalo by se o $LL(1)$ gramatiku a bylo by zapotřebí ji upravit. Pokud je buňka prázdná, jedná se o chybu překladu.

LL analyzátor provádí tři typy kroků v závislosti na tom, zda se na vrcholu zásobníku nachází terminál, neterminál nebo symbol konce vstupu $:

1. Pokud je na vrcholu zásobníku terminál, analyzátor jej porovná s aktuálním symbolem ve vstupním řetězci a pokud jsou stejné, odstraní jej. Pokud stejné nejsou, analyzátor ohlásí chybu a skončí.

2. Pokud je na vrcholu zásobníku neterminál, analyzátor vyhledá v parsovací tabulce příslušné přepisovací pravidlo, které použije k nahrazení symbolu na zásobníku. Buňka v parsovací tabulce odpovídá v řádku aktuálnímu symbolu vstupního řetězce a ve sloupci neterminálnímu symbolu na zásobníku. Číslo přepisovacího pravidla se zapíše na výstup. Pokud není žádné pravidlo nalezeno, analyzátor ohlásí chybu a skončí.

3. Pokud je současně na vrcholu zásobníku a ve vstupním řetězci symbol konce vstupu $, analyzátor končí úspěšným zanalyzovaným řetězcem. V opačném případě analyzátor ohlásí neúspěch a skončí.

Tyto kroky analyzátor opakuje do svého zastavení a buďto je výsledkem úspěšně zanalyzovaný vstup zapsaný jako jeho levá derivace, nebo končí neúspěchem.

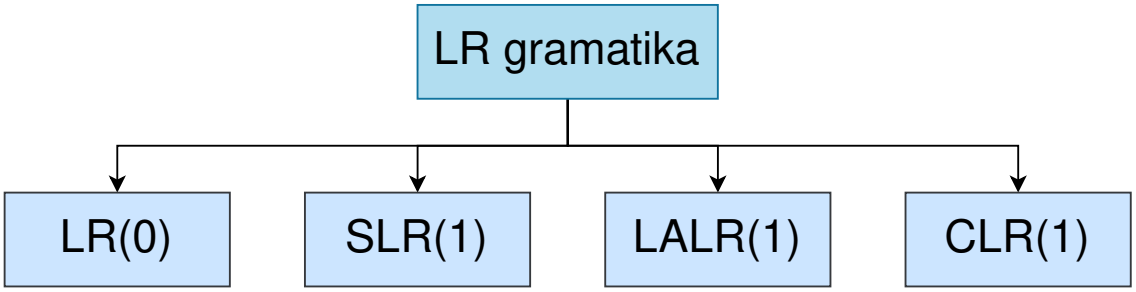
2.6.2 LR(k) gramatiky

Analýza LR gramatik probíhá směrem zdola nahoru od listů derivačního stromu ke kořeni, dokud není k v názvu uvedeno.

Obdobně jako $LL(k)$ gramatiky mají $LR(k)$ gramatiky název odvozen od od směru čtení vstupu (z angl. „***L**eft-to-right*“) a druhé písmeno od provádění pravé derivace (z angl. „***R**ightmost derivation*“). k v závorce také udává počet následujících symbolů, na které analyzátor nahlíží před tím, než se rozhodne analyzovat dřívější symboly. $LR(k)$ gramatiky generují jazyky typu $LR(k)$.

Cílem této analýzy je redukovat vstupní řetězec w na počáteční symbol CFG pomocí reverzní pravé derivace řetězce w. LR analyzátory umí analyzovat pouze jednoznačné bezkontextové gramatiky, případně gramatiky rozšířené o rozhodovací precedenční pravidla. Existuje několik typů LR gramatik. Mezi nejznámější typy patří $LR(0)$, $SLR(1)$ (z angl. „*Simple LR*“), $LALR(1)$ (z angl. „*Look-Ahead LR*“) a $CLR(1)$ (z angl. „*Canonical LR*“), jež jsou hierarchicky znázorněny na obrázku 2.4.

LR syntaktický analyzátor ke svému běhu potřebuje tyto čtyři komponenty:



Obrázek 2.4: Typy LR gramatik

1. Zásobník - používá se k uložení posloupnosti gramatických symbolů. Na dně zásobníku je uložen symbol $.

2. Vstup - obsahuje vstupní řetězec, který má být analyzován, následovaný symbolem $.

3. Výstup - čísla přepisovacích pravidel CFG, která udávají reverzní pravou derivaci vstupního řetězce.

4. Rozkladová, nebo též parsovací tabulka - jedná se o dvourozměrné pole obsahující část přesunu (angl. „*Shift*“) a redukce (angl. „*Reduce*“). Udává, co má analyzátor v kterémkoli stavu dělat.

Při přesunu se načítá symbol ze vstupu, který se stává novým derivačním stromem s jedním uzlem. Při redukci se aplikuje pravidlo gramatiky na několik předchozích derivačních stromů, takže se tyto redukují a vzniká nový derivační strom. Jestliže se nevyskytne syntaktická chyba, analyzátor opakuje tyto kroky, dokud není načten celý vstupní řetězec. Výstupem je jediný derivační strom reprezentující syntaktickou analýzu vstupního řetězce.

Všechny typy LR analyzátorů mají první tři zmíněné komponenty společné. Liší se pouze v tom, jakým způsobem se tvoří parsovací tabulka. V následujících podkapitolách je zkráceně popsáno, jak se konkrétní parsovací tabulky sestavují.

2.6.2.1 LR(0)

Tento typ používá $LR(0)$ položky, což jsou přepisovací pravidla CFG s tečkou v určité pozici na pravé straně pravidla. Jsou užitečná k určení, jak velká část vstupu již byla prohledána až do konkrétního momentu analýzy. Obecně má položka tvar $A \Rightarrow \alpha\beta$, kde α a β jsou jakékoli terminální či neterminální symboly. Z výše zmíněného pravidla lze utvořit tyto $LR(0)$ položky:

- $A \Rightarrow \mathbf{.}\alpha\beta$
- $A \Rightarrow \alpha\mathbf{.}\beta$
- $A \Rightarrow \alpha\beta\mathbf{.}$

Pravidla typu $A \Rightarrow \epsilon$ mají jedinou položku $A \rightarrow \bullet\bullet$. $LR(0)$ parsovací tabulka se řídí těmito třemi pravidly:

1. Pokud stav přechází do jiného stavu na terminálním symbolu, provede se akce posunu v akční části.

2. Pokud stav přechází do jiného stavu na neterminálním symbolu, provede se posun podle přechodové části.

3. Pokud stav obsahuje konečnou položku v daném řádku, zapíše se uzel redukce úplně.

Ukázkovou $LR(0)$ parsovací tabulku lze vidět v tabulce 2.1.

States	**Action**			**Go to**	
	a	**b**	**$**	**A**	**S**
I_0	S3	S4		2	1
I_1			accept		
I_2	S3	S4		5	
I_3	S3	S4		6	
I_4	R3	R3	R3		
I_5	R1	R1	R1		
I_6	R2	R2	R2		

Tabulka 2.1: Ukázka $LR(0)$ parsovací tabulky

2.6.2.2 SLR(1)

$SLR(1)$ parsovací tabulka je velmi podobná $LR(0)$. Pro sestavení $SLR(1)$ parsovací tabulky se používá kanonická kolekce $LR(0)$ položek a redukční tah je umístěn pouze v návaznosti na levou stranu pravidla. $SLR(1)$ parsovací tabulka má tato tři pravidla:

1. Pokud stav přechází do jiného stavu na terminálním symbolu, provede se posun v akční části.

2. Pokud stav přechází do nějakého jiného stavu na neterminálním symbolu, pak tomu odpovídá posun v přechodové části.

3. Pokud stav obsahuje konečnou položku typu $A \Rightarrow ab\bullet$, která nemá žádné přechody do dalšího stavu, pak se pravidlo nazývá redukční pravidlo. Pro všechny terminály X ve $FOLLOW(A)$ se zapíše položka redukce spolu s čísly jejich pravidel.

Zároveň v ní může docházet ke dvěma konfliktům:

1. Posun-redukce (z angl. „*shift-reduce*“) nastává ve stavu, který současně požaduje akci posun, ale také redukci.

2. Redukce-redukce (z angl. „*reduce-reduce*“) nastává ve stavu, který současně požaduje dvě nebo více různých akcí redukce [20].

Na obrázku 2.2 je ukázka $SLR(1)$ parsovací tabulky.

States	**Action**				**Go to**		
	id	**+**	*****	**$**	**E**	**T**	**F**
I_0	S4				1	2	3
I_1		S5		accept			
I_2		R2	S6	R2			
I_3		R4	R4	R4			
I_4		R5	R5	R5			
I_5	S4					7	3
I_6	S4						8
I_7		R1	S6	R1			
I_8		R3	R3	R3			

Tabulka 2.2: Ukázka $SLR(1)$ parsovací tabulky

2.6.2.3 LALR(1)

Ke konstrukci $LALR(1)$ parsovací tabulky se používá kanonická kolekce $LR(1)$ položek. $LR(1)$ položky, které mají stejná přepisovací pravidla, avšak jiný LA, jsou spojeny do jediné množiny položek. Parsovací mechanismus je stejný jako v $CLR(1)$. Rozdíl je pouze v zápisu parsovací tabulky. Na obrázku 2.3 je ukázka $LALR(1)$ parsovací tabulky.

States	**a**	**b**	**$**	**S**	**A**
I_0	S36	S47		12	
I_1			accept		
I_2	S36	S47			5
I_{36}	S36 S47				89
I_{47}	R3 R3	R3			
I_5			R1		
I_{89}	R2	R2	R2		

Tabulka 2.3: Ukázka $LALR(1)$ parsovací tabulky

2.6.2.4 CLR(1)

$CLR(1)$ parsovací tabulka používá kanonickou kolekci $LR(1)$ položek pro své sestavení. Obecně platí, že $CLR(1)$ parsovací tabulky vytváří více stavů v porovnání s $SLR(1)$ parsovacími tabulkami. V této tabulce se umísťuje redukční uzel pouze do LA symbolů. Na obrázku 2.4 je ukázka $CLR(1)$ parsovací tabulky.

States	**a**	**b**	**$**	**S**	**A**
I_0	S3	S4			2
I_1			accept		
I_2	S6	S7		5	
I_3	S3	S4			8
I_4	R3	R3			
I_5			R1		
I_6	S6	S7			9
I_7			R3		
I_8	R2	R2			
I_9			R2		

Tabulka 2.4: Ukázka $CLR(1)$ parsovací tabulky

Kapitola 3

Návrh programu

Hlavním výsledkem této diplomové práce je program pro analýzu bezkontextových gramatik. V této kapitole je popsáno zvolené paradigma programu, jakým způsobem byl program navržen a proč byl zvolen ten či onen přístup v konkrétních částech. V podkapitole 3.1 je popsáno grafické uživatelské rozhraní a funkce v něm, které navržený program umí provádět. Struktura programu je popsána v podkapitole 3.2. Podkapitola 3.3 se věnuje navrženému formátu a struktuře zamýšlené bezkontextové gramatiky. Lexikální analýze, jež by se dala označit jako jádro programu, je věnována podkapitola 3.4.

Bylo velmi důležité důkladně promyslet, jak bude program strukturovaný a robustní, jaký zvolit programovací jazyk a stanovit si, co bude program umět vykonávat. To proto, aby se v průběhu vývoje nemuselo měnit schéma programu a použité datové struktury, protože i menší změna v těchto prvcích by v pozdějších fázích vývoje způsobila podstatnou komplikaci nejen v implementovaných algoritmech. Důraz byl kladen i na to, aby jednotlivé funkce mohly být znovu použity v implementacích případných rozšíření a aby program jako takový bylo možné podle potřeby rozšiřovat.

S ohledem na doporučení vedoucího práce programovat objektově orientovaným přístupem byl zvolen jako nejvhodnější programovací jazyk pro vývoj programu Python verze 3.9. Tento jazyk jednak podporuje objektově orientované paradigma, ale je také uživatelsky přívětivý a obsahuje značné množství datových struktur, které jsou potřeba pro vhodné uložení parametrů bezkontextových gramatik. Téměř všechny algoritmy, které jsou v programu navrženy, potřebují ke svým výpočtům opakovaně procházet jednotlivé konečné množiny, které gramatiku tvoří. Z toho důvodu jsou v programu nejčastěji používány seznamy, skrze které lze poměrně snadno různými způsoby iterovat.

3.1 Uživatelské rozhraní a funkce v něm

Po spuštění programu se zobrazí okno s několika tlačítky a přepínači, které umožňují různé funkce analyzující bezkontextové gramatiky. Obsluha celého programu je řešena pomocí tlačítek, přepínačů

a textových polí. V okně programu jsou tři záložky, přičemž v první z nich je možné provádět základní operace s bezkontextovými gramatikami, ve druhé záložce je možné CFG analyzovat $LR(0)$ parserem a ve třetí záložce $LL(1)$ parserem. Jednotlivé záložky programu jsou detailněji popsány níže.

Každá záložka obsahuje dvě velká textová pole, z nichž pole nalevo lze definovat jako vstupní a pole napravo jako výstupní. Nezávisle na tom, která záložka je zrovna aktivní, je uživateli umožněno načíst CFG z textového souboru, načíst šablonu CFG pro manuální zadání, vymazat obsah polí, zvolit výpis výsledků do souboru a ukončit program.

3.1.1 CFG operations

Na této záložce lze provádět základní operace s bezkontextovými gramatikami. Tyto operace jsou často prerekvizitou k následnému využití CFG. Konkrétně program umí:

- Výpočet množiny $FIRST$
- Výpočet množiny $FOLLOW$
- Redukce gramatiky
- Odstranění epsilon-pravidel
- Konstrukce zásobníkového automatu

3.1.2 LR parser

Ve druhé záložce je možné provést syntaktickou analýzu zdola nahoru pomocí $LR(0)$ parseru. Pro načtenou gramatiku lze provádět následující operace:

- Výpočet LR položek
- Sestavení LR(0) tabulky
- Validace vstupního řetězce LR(0) parserem

3.1.3 LL parser

Ve třetí záložce je umožněno detekovat, jestli je vybraná CFG typu $LL(1)$ nebo jestli rozkladová tabulka obsahuje případné konflikty. Program umí provádět tyto funkce:

- Sestavení $LL(1)$ tabulky
- Validace vstupního řetězce $LL(1)$ parserem
- Detekce konfliktů $FIRST/FIRST$ a $FIRST/FOLLOW$

3.2 Struktura programu

V předchozí podkapitole 3.1 je popsáno, co uživatel vidí a s čím může reálně pracovat, jakmile program spustí. Tato podkapitola se věnuje struktuře navrženého programu

Program sestává ze čtyř zdrojových souborů, kde jeden soubor lze vnímat jako hlavní a zbylé soubory jako vedlejší či pomocné.

- *CFGAnalyzer.py* — hlavní soubor obsahující jádro programu
  - *helper_utils.py* — vedlejší soubor s pomocnými funkcemi
  - *print_utils.py* — vedlejší soubor s funkcemi pro výpis informací
  - *struct_utils.py* — vedlejší soubor s definicemi struktur a tříd

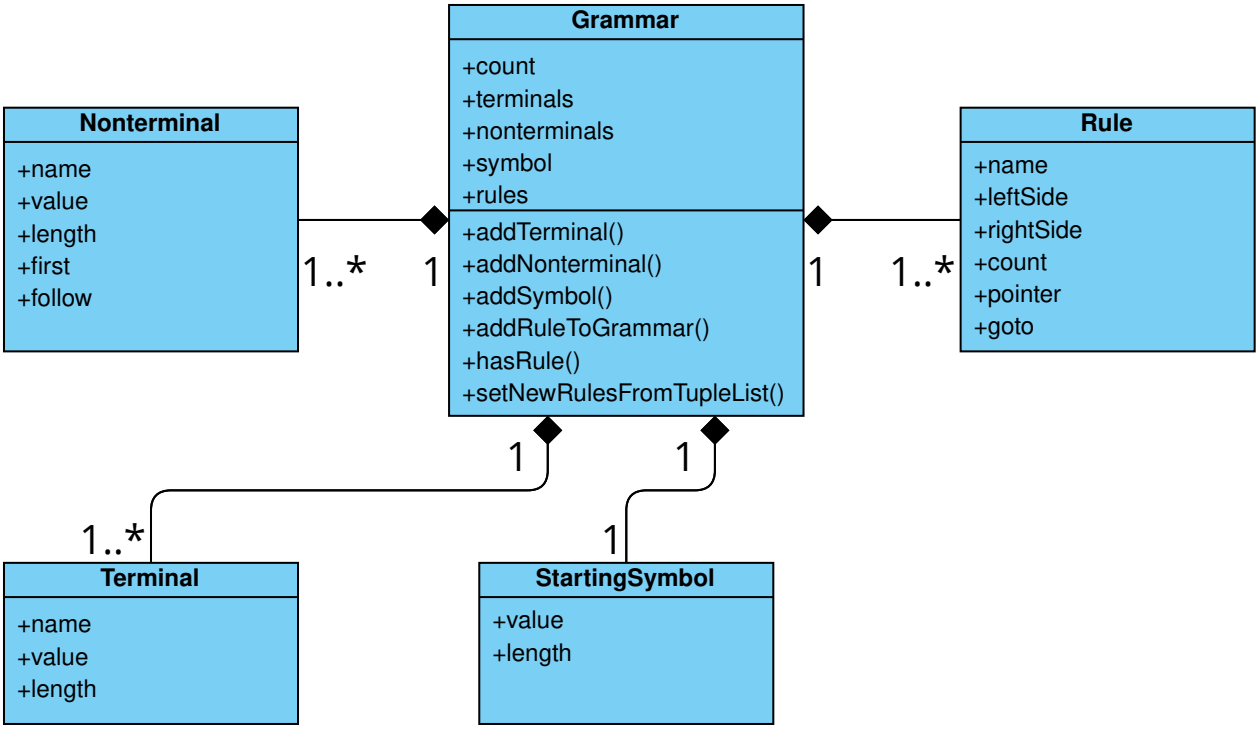
Hlavní soubor obsahuje řídící programovou logiku, definici grafického uživatelského rozhraní, analyzátor vstupního textu, výpočet základních vlastností, množin a další. První vedlejší soubor obsahuje krátké pomocné metody, které jsou volány z několika míst programu. Ve druhém souboru jsou definice metod vypisujících výsledky výpočtů a informace o běhu programu. Třetí soubor obsahuje definice výčtových typů, tříd a příslušných metod pro práci s třídními objekty a rozkladovými tabulkami.

Program v rámci svého běhu využívá další dva textové soubory. Z jednoho souboru dle výběru uživatele načítá bezkontextovou gramatiku a do druhého může vypisovat výsledky, pokud si uživatel výpis do souboru zvolí.

Bezkontextovou gramatiku ze vstupního souboru načítá lexikální analyzátor, který je podrobněji popsán v podkapitole 3.4. Takto načtená data jsou uložena v objektu třídy definující celkovou bezkontextovou gramatiku. Až na počáteční symbol, který je v objektu uložen jako řetězec znaků, jsou zbylé prvky, které gramatiku tvoří, uloženy jako seznamy dílčích objektů. To znamená, že pro neterminály, terminály i přepisovací pravidla jsou definovány specifické třídy a každý zmíněný prvek je objektem dané třídy uloženým v dílčím seznamu příslušné gramatiky. Tyto vztahy jsou znázorněny v diagramu 3.1

Každý takto uložený neterminál a terminál si v rámci objektu uchovává název pro interní použití, svou hodnotu a délku této hodnoty. Pokud by řetězec NNN reprezentoval první neterminální symbol v množině neterminálů CFG, pak by objekt třídy *Nonterminal* měl interní název 0, hodnotu NNN a délku 3, neboť se jedná o tři znaky.

Podobně jsou uložena i přepisovací pravidla s tím rozdílem, že levá strana vždy reprezentuje neterminální symbol a pravá strana je seznam položek, v němž každá položka reprezentuje možnost, jak lze daný neterminál přepsat.



Obrázek 3.1: Diagram tříd načtené CFG

3.3 Formát souboru s bezkontextovou gramatikou

Je žádoucí, aby byl formát zápisu CFG jednoznačný a pro uživatele zřetelný, protože počty prvků množin, ze kterých se CFG skládá, nejsou omezeny. Při použití nevhodného zápisu CFG by se pak mohlo lehce stát, že se uživatel přestane v symbolech gramatiky orientovat, nebo že zápis bude zmatečný. Z toho důvodu je formát bezkontextové gramatiky navržen tak, aby nebyl v rozporu s obecně zažitými konvencemi pro popis CFG a zároveň, aby splňoval výše zmíněné.

V programu je používán následující formát zápisu gramatiky. Tučně zvýrazněné konstrukce jsou závazné pro zápis kterékoli CFG. Kurzívou jsou vyznačeny části, které definují samotnou CFG a které žádná omezení nemají:

cfg=*UkazkovaGramatika*
N={*S,A***}**
T={*a,b***}**
S=*S*
P=*S*→*AA*
A→*aA*|*b*
A→*bb*

Je tedy patrné, že za klíčovým slovem cfg a znakem rovná se = je uveden název gramatiky. Násle-

duje na novém řádku množina neterminálních symbolů ohraničených ve složených závorkách a oddělených čárkou. $N_1, N_2, \ldots, N_i$ neterminálních symbolů je možné zapsat takto: $N = \{N_1, N_2, \ldots, N_i\}$.

Obdobným způsobem se zapisují terminální symboly. $T_1, T_2, \ldots, T_j$ terminálních symbolů je možné zapsat takto: $T = \{T_1, T_2, \ldots, T_j\}$.

Počáteční symbol je vždy jen jeden a na novém řádku se zadává $S = A$, kde A značí počáteční symbol a zároveň A patří do množiny neterminálních symbolů.

Přepisovací pravidla vždy začínají zápisem $P =$ následovaným samotnými pravidly. Levá a pravá strana pravidla je rozdělena znaky ->, avšak pro lepší čitelnost jsou zde tyto znaky nahrazeny šipkou $\rightarrow$. Každé pravidlo je zapsáno samostatně na řádku, případně, pokud je nějaký neterminál možné přepsat více různými způsoby, lze toto zapsat na jeden řádek rozdělením pravých stran znakem |. Jinými slovy, že dvě pravidla: $A \rightarrow a$ a $A \rightarrow b$ lze ekvivalentně zapsat i tímto způsobem: $A \rightarrow a \mid b$.

3.4 Lexikální analýza

CFG je ve vstupním souboru definována jako posloupnost symbolů, které je nutno načíst, aby bylo možné s gramatikou pracovat. Program si musí umět poradit s libovolnou CFG, protože její definice není předem programu známa. Z toho důvodu program obsahuje implementaci lexikálního analyzátoru pro analýzu vstupních symbolů. Ty převádí na tokeny známé též jako lexémy.

Za tímto účelem program obsahuje třídu výčtového typu *TokenKind*, která obsahuje pevně stanovené množiny hodnot. Každému tokenu tedy odpovídá nějaká hodnota z tohoto výčtového typu. Prvky tohoto výčtového typu a jejich význam jsou následující:

- EOF — konec souboru
- IDENT — identifikátor
- NUM — číslo
- EQUALS — znak =
- OpenCurlyBrace — znak počáteční složené závorky
- CloseCurlyBrace — znak ukončující složené závorky
- PIPE — znak |
- COMMA — znak ,
- ARROW — šipka ->
- EOL — znak konce řádku \n
- OtherChar — jiný znak

- Error — chybný token

Program čte vstupní posloupnost znaků postupně po jednom znaku a vždy, když načte token, který odpovídá nějakému druhu z výčtového typu, tak jej zpracuje. Tímto způsobem se přečte obsah celého vstupního souboru a jakmile analyzátor dojde až na konec souboru, lexikální analýza končí a jejím výsledkem je načtená CFG uložená v příslušných objektech.

Kapitola 4

Implementace

Tato kapitola se zabývá programovou implementací. Program má název *CFGAnalyzer*, je napsán v jazyce Python 3.9 a vývoj probíhal ve vývojovém prostředí *PyCharm* verze 2022.1 [21]. V níže uvedených podkapitolách je popsána implementace uživatelského rozhraní (4.1), reprezentace gramatiky (4.2), způsob načítání gramatiky (4.3), základní výpočty a operace (4.4) a také detekce dvou typů gramatik — LR(0) v (4.5) a LL(1) v (4.6).

Program využívá funkce z následujících knihoven:

- *PySimpleGUIQt* — k tvorbě uživatelského rozhraní
- *copy* — pro hluboké kopírování (angl. „*deepcopy*“) proměnných
- *enum* — pro podporu výčtového typu
- *sys* — pro přístup k systémovým proměnným

Pro spuštění programu je nutné mít nainstalovaný Python verze 3.9 nebo vyšší a také zmíněné knihovny. V případě použití příkazové řádky je možné knihovny nainstalovat pomocí příkazu `pip install <název knihovny>`, kde `<název knihovny>` je nutné nahradit názvem konkrétní knihovny. Program se poté spouští příkazem `Python CFGAnalyzer.py`. Při spuštění dochází k volání hlavní funkce `main()`, která řídí chod celého programu.

4.1 Uživatelské rozhraní

Uživatelské rozhraní je vytvořeno použitím knihovny *PySimpleGUIQt*. Jedná se o nástavbu knihovny *PySimpleGUI* využívající *Qt* aplikační rámec (angl. „*framework*“) k tvorbě grafického uživatelského rozhraní [22]. Jako první je v hlavní funkci `main` definováno rozložení (angl. „*layout*“) uživatelského rozhraní pomocí komponent. Chování a vlastnosti komponent jsou nastaveny pomocí jejich atributů a layout tohoto programu tvoří tyto komponenty:

- Text — zobrazení textu v okně
- Input — zobrazení jednořádkového vstupního pole
- Multiline — zobrazení víceřádkového vstupního pole
- Button — zobrazení tlačítka
- FileBrowse — zobrazení tlačítka umožňujícího výběr souboru
- Checkbox — zobrazení přepínače
- Tab — zobrazení záložky
- TabGroup — zobrazení skupiny záložek

Uživatelské rozhraní je zkonstruováno z tohoto layoutu a z dalších parametrů udávajících jeho chování. Programová logika následně běží v nekonečném cyklu, ve kterém naslouchá událostem z uživatelského rozhraní a na jejich základě volá příslušné metody. Tato smyčka je přerušena požadavkem na ukončení programu.

4.2 Reprezentace gramatiky

Celá CFG je reprezentována objektem třídy `Grammar`, v níž jsou neterminály, terminály a přepisovací pravidla uloženy ve formě seznamů a počáteční symbol jako proměnná. Počáteční symbol a jednotlivé položky v těchto seznamech jsou opět samostatné objekty příslušných tříd — `Nonterminal`, `Terminal`, `StartingSymbol` a `Rule`, v nichž jsou uloženy načtené informace. Nad takto zpracovanou a uloženou gramatikou program provádí výpočty a analýzu.

Třídu `Grammar` tvoří tyto atributy: `count, terminals, nonterminals, symbol` a `rules`. Zároveň `Grammar` obsahuje tyto metody: `addTerminal, addNonterminal, addSymbol, addRuleToGrammar, hasRule, setNewRulesFromTupleList`.

Třídu `Terminal` tvoří atributy: `name, value` a `length`. Třída `Nonterminal` obsahuje oproti ní navíc ještě `first` a `follow`. Ve třídě `StartingSymbol` se nacházejí atributy `value` a `length`. Ve třídě `Rule` reprezentující jednotlivé přepisovací pravidla jsou atributy `name, leftSide, rightSide, count, pointer` a `goto`.

4.3 Načtení gramatiky

Aby bylo možné CFG analyzovat, je potřeba ji nejprve načíst. Z toho důvodu je do doby načtení CFG aktivní pouze tlačítko *Load CFG* umožňující načtení gramatiky a tlačítko *Cancel* ukončující program. Při stisku prvně zmíněného tlačítka se stanou aktivní všechna zbylá tlačítka a zavolá se

metoda `loadAndParseData`, v níž probíhá načtení gramatiky z vybraného souboru. Tato metoda tedy reprezentuje lexikální analýzu obsahu vstupního souboru.

V cyklu se zde voláním metody `nextToken` načítají tokeny, jejichž výčet je uveden v 4.1.

```
class TokenKind(Enum):
    EOF = 0 #konec souboru ''
    IDENT = 1 #identifikator
    NUM = 2 #cislo
    EQUALS = 3 #rovna se =
    OpenCurlyBrace = 4 #pocatecni slozena zavorka {
    CloseCurlyBrace = 5 #koncova slozena zavorka }
    PIPE = 6 #svisla cara |
    COMMA = 7 #carka ,
    ARROW = 8 #sipka ->
    EOL = 9 #konec radku \n
    OtherChar = 10 #jiny znak
    ERROR = 11 #chyba parsovani
```

Listing 4.1: Definice druhů tokenů

Token může být tvořen posloupností několika symbolů, takže i samotné načtení jednoho tokenu probíhá opět ve smyčkách a symboly jsou načítány po jednotlivých znacích. Program na vstupu očekává bezkontextovou gramatiku ve formátu popsaném v podkapitole 3.3. Jakýkoli jiný formát vstupní gramatiky by vedl k neúspěšnému načtení a uživatel by byl chybovou zprávou v okně programu vyzván k úpravě formátu dle specifikace.

Program například rozpozná, že zrovna načítá neterminální symboly, pokud přečte sekvenci symbolů `N=` a po ní následují ve složených závorkách identifikátory s velkými písmeny oddělené čárkou. Tyto identifikátory pak reprezentují neterminální symboly. Načtení názvu CFG, neterminálů, terminálů, počátečního symbolu a přepisovacích pravidel probíhá v odpovídajících metodách `loadGrammarName`, `loadNonterminals`, `loadTerminals`, `loadStartingSymbol`, `loadRules`.

4.4 Základní výpočty a operace

Výpočet základních operací nad CFG probíhá na první záložce programu stisknutím tlačítka *Enter*. Jakmile je událost stisknutí vyvolána, volají se konkrétní funkce v návaznosti na tom, které přepínače uživatel vybral. Výsledky takto vypočtených funkcí program vypisuje do výstupního pole pomocí metod definovaných v souboru *print_utils.py*, případně též do souboru `outputFile.txt`, je-li přepínač *Print results to file* vybrán.

4.4.1 Algoritmus výpočtu množin $FIRST$ a $FOLLOW$

Výpočet množin $FIRST$ a $FOLLOW$ probíhá ve funkci `firstAndFollow`. Množinu $FIRST$ program spočítá i v případě, že uživatel zvolí jen výpočet množiny $FOLLOW$. Pro možnost výpočtu množiny $FOLLOW$ je totiž nutné mít vypočtenou množinu $FIRST$. Obě množiny jsou datového typu slovník — `dict`, který udává neuspořádanou množinu dvojic ve formátu klíč-hodnota, v programu označovaném jako `key-value`. Klíč v této dvojici reprezentuje neterminální nebo terminální symbol načtené CFG. Hodnota obsahuje výsledek výpočtu a je datového typu množina — `set`, což je neuspořádaná kolekce unikátních hodnot. Díky tomu je zajištěna unikátnost výsledných hodnot ve výsledcích pro každý neterminál i terminál.

Nejprve se dosadí za klíče v proměnných všechny neterminální symboly pro výpočty $FIRST$, $FOLLOW$ a poté i terminálními symboly pro $FIRST$. Následuje samotný algoritmus výpočtu zobrazený v 4.2, který opakovaně prochází přepisovací pravidla do doby, dokud v množinách přibývají nové symboly. Když se po průchodu všemi pravidly nepřidá žádná nová hodnota, cyklus končí a výsledek je předán hlavní funkci.

```
    while True: #cyklus se opakuje, dokud jsou do mnozin pridavany nove symboly
        updated = False
        for nt, expression in rules:
            if computeFirst or computeFollow: #vypocet mnoziny FIRST
                for symbol in expression:
                    updated |= h_utils.union(first[nt], first[symbol])
                    if symbol not in epsilon:
                        break
                else:
                    updated |= h_utils.union(epsilon, {nt})
        for nt, expression in rules:
            if computeFollow: #vypocet mnoziny FOLLOW
                aux = follow[nt]
                for symbol in reversed(expression):
                    if symbol in follow:
                        updated |= h_utils.union(follow[symbol], aux)
                    if symbol in epsilon:
                        aux = aux.union(first[symbol])
                    else:
                        aux = first[symbol]
        if not updated: #pokud v poslednim cyklu nebyly pridany zadne nove symboly
             do mnozin, funkce konci vypocet a navraci vypoctene mnoziny
            return first, follow, epsilon
```

Listing 4.2: Algoritmus počítající množiny $FIRST$ a $FOLLOW$

4.4.2 Algoritmus redukce gramatiky

Při požadavku na redukci CFG se volá funkce `reduction`. V ní se nejprve do proměnných datového typu `set` vloží zvlášť neterminální a terminální symboly, které se používají v algoritmu hledajícím validní neterminální symboly. Takto nalezené neterminály jsou uloženy v proměnné `T_validNonterminals`. Neterminální symboly, které v této proměnné nejsou, algoritmus odstraní z množiny přepisovacích pravidel.

Druhá množina, kterou je zapotřebí k redukci CFG zjistit, je množina dosažitelných neterminálních symbolů. V kódu je reprezentována proměnnou `D_reachableNonterminals` datového typu `set`. Pravidla, která nedosažitelné neterminály z této množiny obsahují, jsou poté odstraněna a výsledkem je množina přepisovacích pravidel obsahující jen validní a dosažitelné neterminální symboly.

Tento algoritmus proběhne vždy, i když už je mu předána CFG v redukovaném tvaru. Na konci probíhá ověření, jestli se počet přepisovacích pravidel po průchodu algoritmu změnil nebo ne. Jestliže počet pravidel zůstal stejný, znamená to, že CFG již byla redukovaná.

4.4.3 Algoritmus odstranění ϵ-pravidel

Odstranění ϵ-pravidel z CFG probíhá ve funkci `epsRulesRemoval`. Algoritmus nejprve prochází pravé strany všech přepisovacích pravidel za účelem najít neterminální symboly přepsatelné na ϵ. Nalezené neterminály jsou uloženy v proměnné `setE` používající datový typ `set`.

Při průchodu všemi pravidly CFG se kontroluje, zda existují pravidla, jejichž pravé strany obsahují nějaké z neterminálních symbolů uvedených v `setE`. Pokud je takové pravidlo nalezeno, vygenerují se nová pravidla, která odpovídají všem kombinacím vzniklým vypuštěním daného neterminálu z pravé strany pravidla.

4.4.4 Algoritmus konstrukce zásobníkového automatu

Převedení CFG na ekvivalentní zásobníkový automat probíhá v metodě `printPushdownAutomaton`, jejíž implementace je uložena v souboru *print_utils.py*. Tato metoda je na rozdíl od předešlých definována v souboru s pomocnými výpisy, neboť v ní nedochází k nějakým výpočtům, ale pouze se na základě zadané CFG specificky vypisuje uspořádaná šestice odpovídajícího zásobníkového automatu.

4.5 LR(0)

Jestliže chce uživatel na druhé záložce programu detekovat typ LR(0) gramatiky, musí rovněž načíst soubor s bezkontextovou gramatikou, aby se stala tlačítka *Build LR items* a *Validate input* aktivní. Stiskem prvního tlačítka dochází k redukci gramatiky metodou `reduction`, výpočtu LR položek a sestavení LR(0) tabulky. Validaci typu LR(0) gramatiky předchází výpočty v několika krocích. Struktura LR analyzátoru a jednotlivé kroky jsou definovány v souboru *struct_utils.py*.

Nejprve se vytvoří objekt třídy `LRParser` z načtené CFG, který je použit pro výpočet LR(0) položek a uzávěrů množiny položek (angl. „*closures*“). LR(0) položka je pravidlo gramatiky s tečkou přidanou někde v pravé straně pravidla. Tečka udává, v jakém stavu se analyzátor zrovna nachází. Výpočet closures začíná od počátečního symbolu v metodě `buildClosures`, z níž je volána metoda `extendChain`, uvnitř které dochází k vytvoření objektů třídy `Closure`. Algoritmus každou LR(0) položku rozšiřuje rekurzivním přidáváním všech vhodných položek, dokud neprojde všechny neterminály, které se nacházejí za tečkou.

Následuje vytvoření parsovací tabulky v metodě `buildParsingTable`, která prochází dříve spočítané closures. Tento algoritmus je zobrazen v ukázce 4.3. Položky parsovací tabulky jsou uloženy v proměnné `parsingTable` datového typu `dict`. V parsovací tabulce značí symbol r akci redukce, symbol s akci přesunu a *acp* finální stav.

```
    def buildParsingTable(self):
        for key in self.closures:
            self.parsingTable[key] = {}
            for rule in self.closures[key].rules:
                if rule.goto == '' and rule.rightSide[0] == self.grammar.symbol.
                    value:
                    self.parsingTable[key]['$'] = "acp" #koncovy stav
                    continue
                if rule.goto == '': #finalni polozky
                    for terminal in self.grammar.terminals:
                        self.parsingTable[key][terminal.value] = "r%s-%s" % (rule.
                            leftSide, len(rule.rightSide[0]))
                    self.parsingTable[key]['$'] = "r%s-%s" % (rule.leftSide, len(
                        rule.rightSide[0]))
                    continue;
                expandChar = rule.rightSide[0][rule.pointer]
                if expandChar.isupper(): #nasleduje se neterminal, akce je GOTO
                    self.parsingTable[key][expandChar] = rule.goto
                    continue
                else:
```

```
                self.parsingTable[key][expandChar] = "s%s" % rule.goto
        return self.parsingTable
```

Listing 4.3: Algoritmus vytvářející LR(0) parsovací tabulku

Spočítané LR(0) položky a parsovací tabulka jsou poté vypsány v levém poli uživatelského rozhraní.

K validaci řetězce slouží druhé tlačítko dostupné na této záložce. Nejprve se vždy provedou stejné kroky, jako při stisku prvního tlačítka. Poté dochází k validaci zadaného řetězce symbolů v metodě `parseLR0Input`. Řetězec symbolů by měl být zakončen ukončujícím symbolem $, ale pokud jej uživatel nezadá, program si ho doplní sám. Validace začíná v prvním stavu parsovací tabulky a podle příslušných pravidel v ní se algoritmus snaží dojít až do finálního stavu. V případě úspěchu je řetězec potvrzen jako validní (angl. „*valid*“), při neúspěchu jako nevalidní (angl. „*invalid*“), nebo také může být prázdný (angl. „*empty*“). Výsledek validace je vypsán do pravého pole programu.

4.6 LL(1)

Detekce typu LL(1) gramatiky ve třetí záložce *LL(1) Parser* je naprogramována velmi podobně jako detekce typu LR(0). Rozdíl je pouze v algoritmu konstrukce parsovací tabulky a v přidané detekci konfliktů. Při stisku tlačítka *Build LL(1) table* se nejprve spočítají pro neterminální symboly množiny $FIRST$ a $FOLLOW$. Vytvoří se objekt třídy `LLParser`, který obsahuje informace o načtené gramatice, LL(1) parsovací tabulce a detekovaných konfliktech.

Metoda `buildParsingTable` implementuje algoritmus sestavení parsovací tabulky. Tabulka je validní jen když neobsahuje žádné konflikty typu $FIRST/FIRST$ nebo $FIRST/FOLLOW$. Detekce konfliktů probíhá v metodě `detectConflicts`, která zjistí přítomnost konfliktů pro každý z neterminálů. Tyto jsou uživateli vypsány v případě, že zvolí přepínač *Detect conflicts*.

Tabulku si lze představit jako dvourozměrnou matici, kterou v řádcích tvoří neterminály a ve sloupcích terminály. Jednotlivé položky v této tabulce reprezentují pravidla gramatiky. Ve výsledné tabulce jsou pro přehlednost pravidla nahrazena čísly, přičemž každé číslo odpovídá konkrétnímu pravidlu uvedenému ve výpisu nad tabulkou.

Při stisku tlačítka *Validate input* dochází ke stejným výpočtům, jako při stisknutí *Build LL(1) table* a navíc k validaci zadaného řetězce symbolů. Validace je implementována v metodě `parseInput` a načítá řetězec symbolů ze vstupního pole. Řetězec by měl být zakončen koncovým symbolem $, ale jestliže jej uživatel nezadá, program si jej doplní sám. Výsledkem validace je zpráva v poli na pravé straně uživatelského rozhraní o validním, nevalidním nebo prázdném řetězci.

Kapitola 5

Demonstrace použití

V této kapitole jsou demonstrovány dílčí funkce programu za použití konkrétních bezkontextových gramatik. CFG byly zvoleny tak, aby umožňovaly výpočet dané vlastnosti, nebo provedení operace nad nimi. Jednotlivé podkapitoly obsahují vždy definici vstupní gramatiky, potřebnou konfiguraci v programu a výstup.

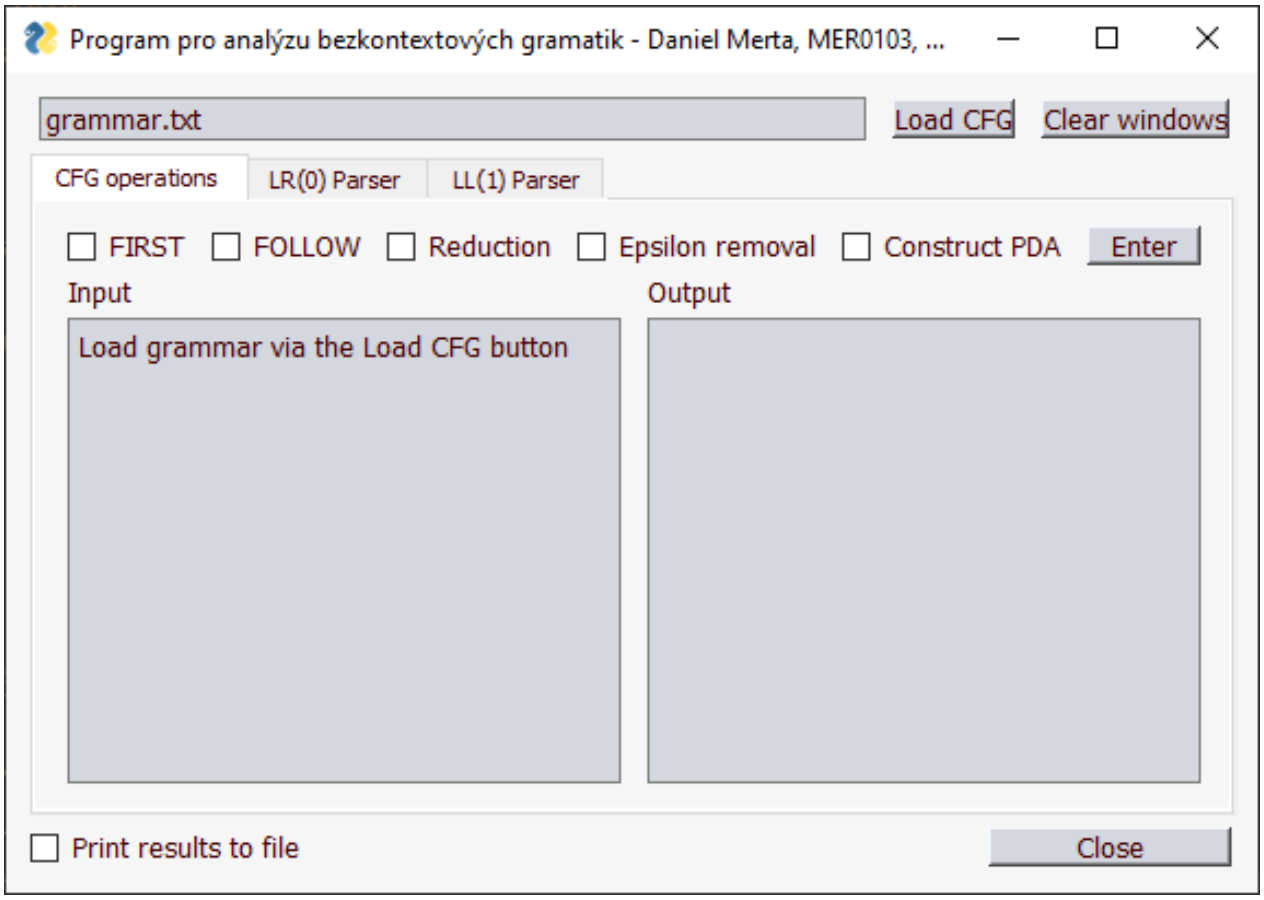
5.1 Spuštění programu

Po spuštění programu je zobrazeno okno aplikace s předvybranou první záložkou *CFG operations*, jak lze vidět na obrázku 5.1. Na této záložce je možné vybírat k výpočtu více přepínačů najednou, přičemž program řadí výsledky jednotlivých funkcí pod sebe ve výstupním poli.

V kterémkoli okamžiku běhu programu má uživatel možnost vymazat obsah obou polí stisknutím tlačítka *Clear windows* v pravém horním rohu. Pro vypsání výsledků do souboru stačí zvolit přepínač *Print results to file* umístěný v levém dolním rohu. V neposlední řadě lze program kdykoli ukončit tlačítkem *Close* nebo standardním křížkem v rohu programu.

Aby bylo uživateli umožněno provést výpočet, je nutné vždy jako první vybrat a načíst textový soubor obsahující definici bezkontextové gramatiky pomocí tlačítka *Load CFG*. Takto načtená gramatika lexikálním analyzátorem je v případě úspěchu okamžitě zobrazena v poli *Input* v levé straně programu. Pole *Output* napravo obsahuje vypočítaný výsledek nebo validaci vstupního řetězce.

Stejným způsobem probíhají výpočty i na dalších záložkách *LR(0) Parser* a *LL(1) Parser*. Aby program umožnil výpočet příslušných položek, tabulek a validaci vstupního řetězce, je nutné mít před tímto načtenou CFG.

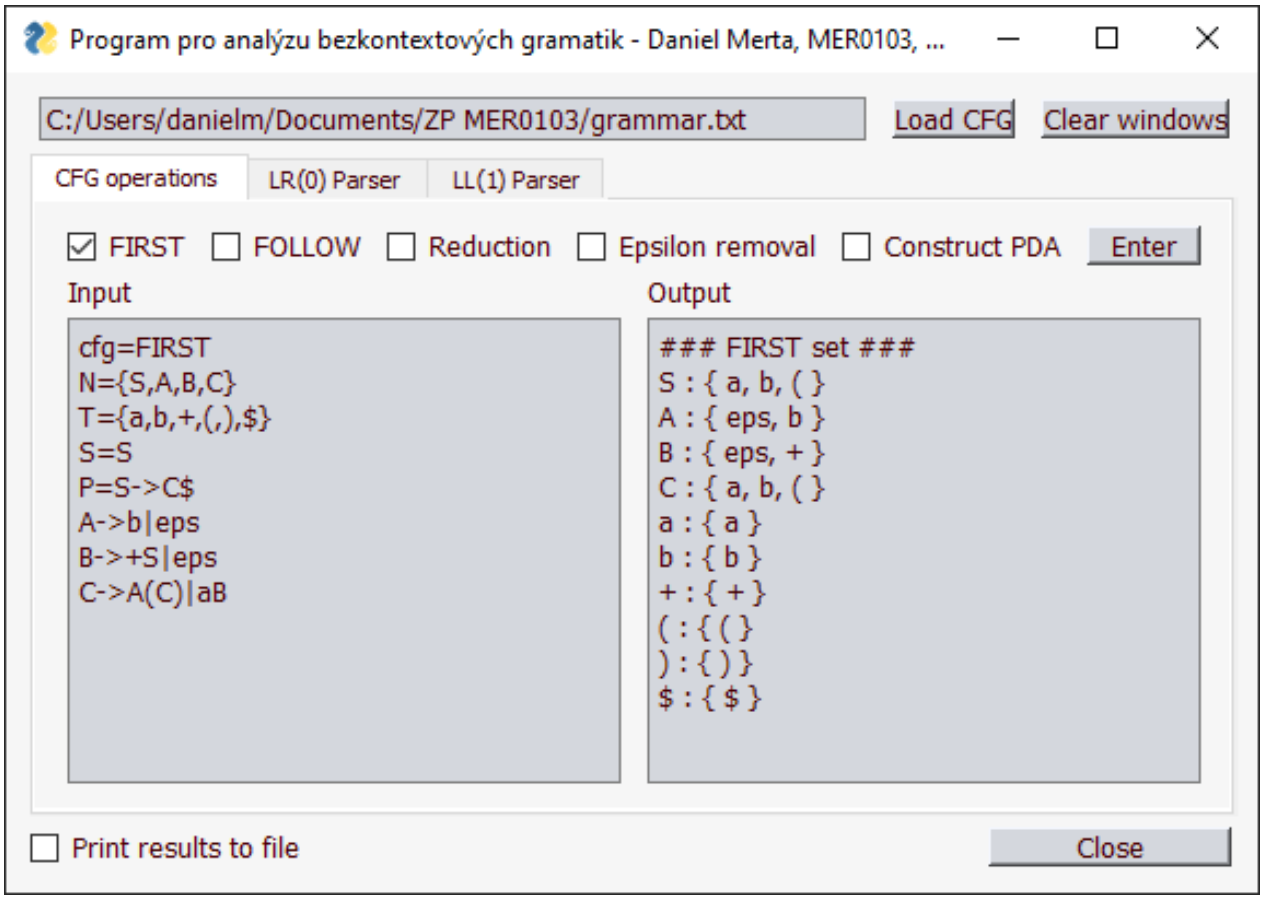


Obrázek 5.1: Výchozí okno programu

5.2 Množina FIRST

Pro výpočet množiny $FIRST$ je potřeba v programu vybrat volbu $FIRST$ a zmáčknout tlačítko *Enter*. Je zadána CFG $FIRST = (N, T, S, P)$, kde:
$N = \{S, A, B, C\}$
$T = \{a, b, +, (,), \$\}$
$S = S$
$P = S \rightarrow C\$$
$A \rightarrow b \mid eps$
$B \rightarrow +S \mid eps$
$C \rightarrow A(C) \mid aB$

Z výsledku výpočtu v okně *Output* na obrázku 5.2 je patrné, že se množina $FIRST$ počítá pro všechny neterminální a terminální symboly.

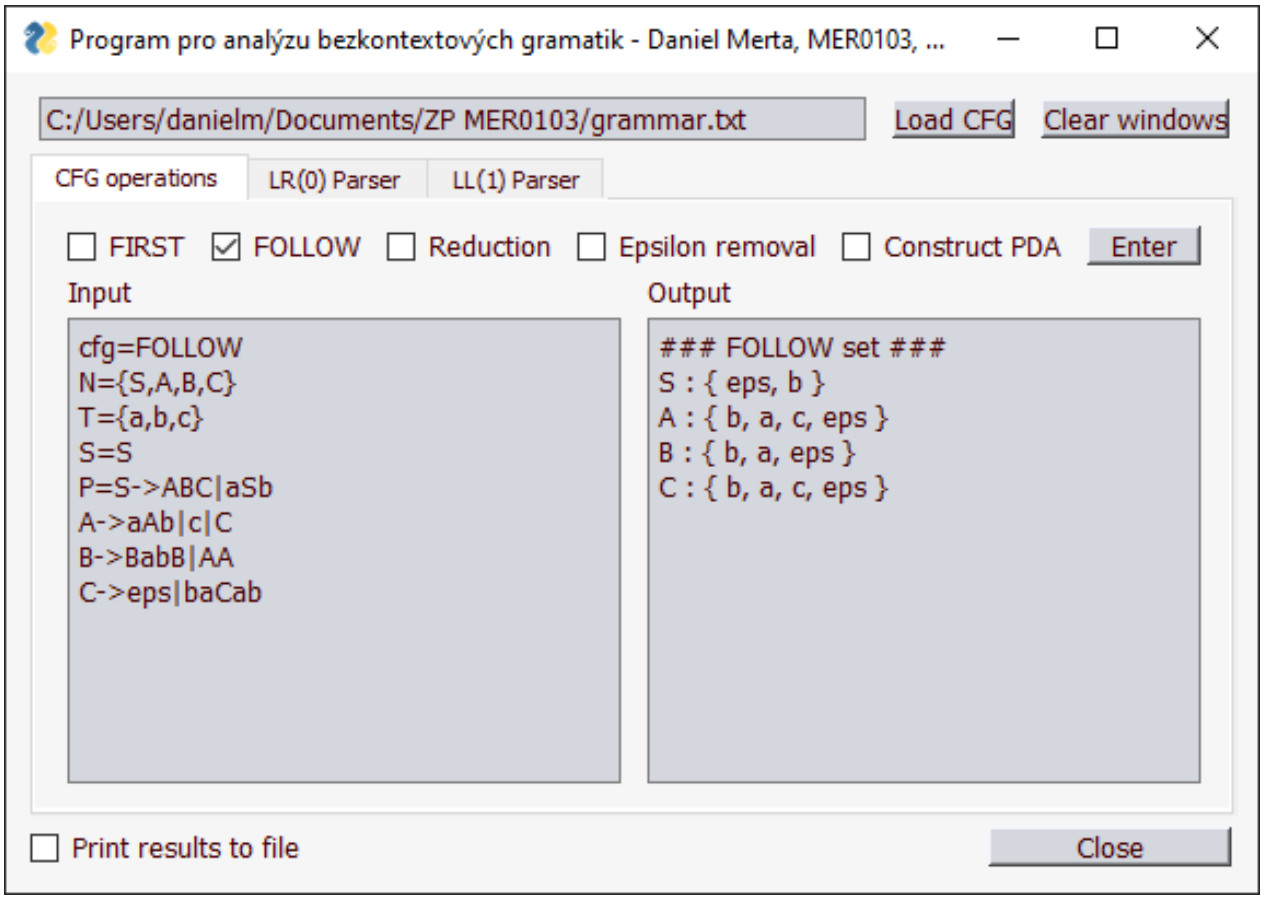


Obrázek 5.2: Výpočet množiny $FIRST$

5.3 Množina FOLLOW

Množinu $FOLLOW$ je možné spočítat vybráním přepínače $FOLLOW$ a stiskem tlačítka *Enter*. Je zadána CFG $FOLLOW = (N, T, S, P)$, kde:
$N = \{S, A, B, C\}$
$T = \{a, b, c\}$
$S = S$
$P = S \rightarrow ABC \mid aSb$
$A \rightarrow aAb \mid c \mid C$
$B \rightarrow BabB \mid AA$
$C \rightarrow eps \mid baCab$

Výpočet probíhá pouze pro neterminální symboly a z obrázku 5.3 lze například vyčíst, že bezprostředně po neterminálu B mohou následovat terminální symboly b, a nebo ϵ.

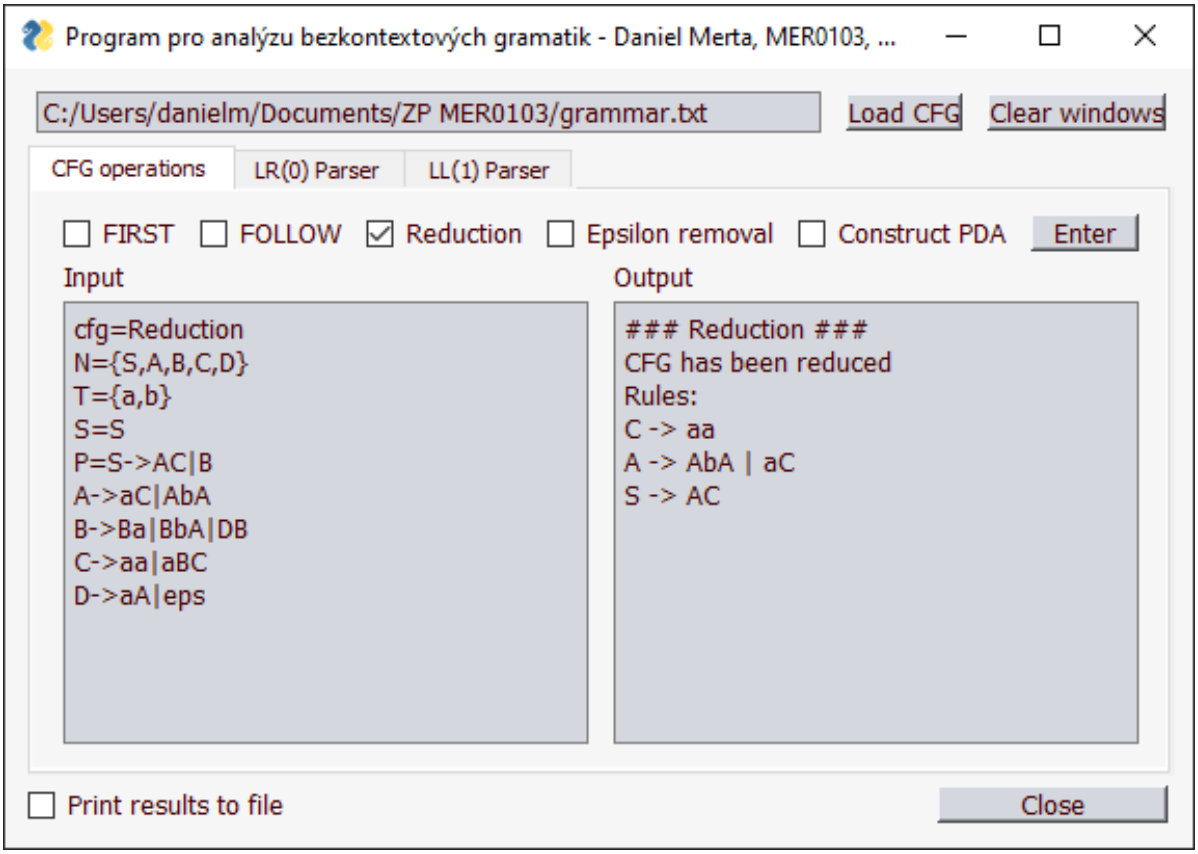


Obrázek 5.3: Výpočet množiny $FOLLOW$

5.4 Redukce gramatiky

Pro redukci CFG slouží přepínač *Reduction*. Při redukci gramatiky jsou odstraněny takové neterminální symboly, ze kterých nelze vygenerovat terminální slovo a rovněž ty, které nejsou dosažitelné z počátečního symbolu. Pokud je CFG již v redukovaném tvaru, program tuto informaci ohlásí. Je zadána CFG $Reduction = (N, T, S, P)$, kde:
$N = \{S, A, B, C, D\}$
$T = \{a, b\}$
$S = S$
$P = S \rightarrow AC \mid B$
$A \rightarrow aC \mid AbA$
$B \rightarrow Ba \mid BbA \mid DB$
$C \rightarrow aa \mid aBC$
$D \rightarrow aA \mid eps$

Z výstupu na obrázku 5.4 je vidět, že gramatice v redukovaném tvaru zbyla pouze pravidla neobsahující neterminály B a D.

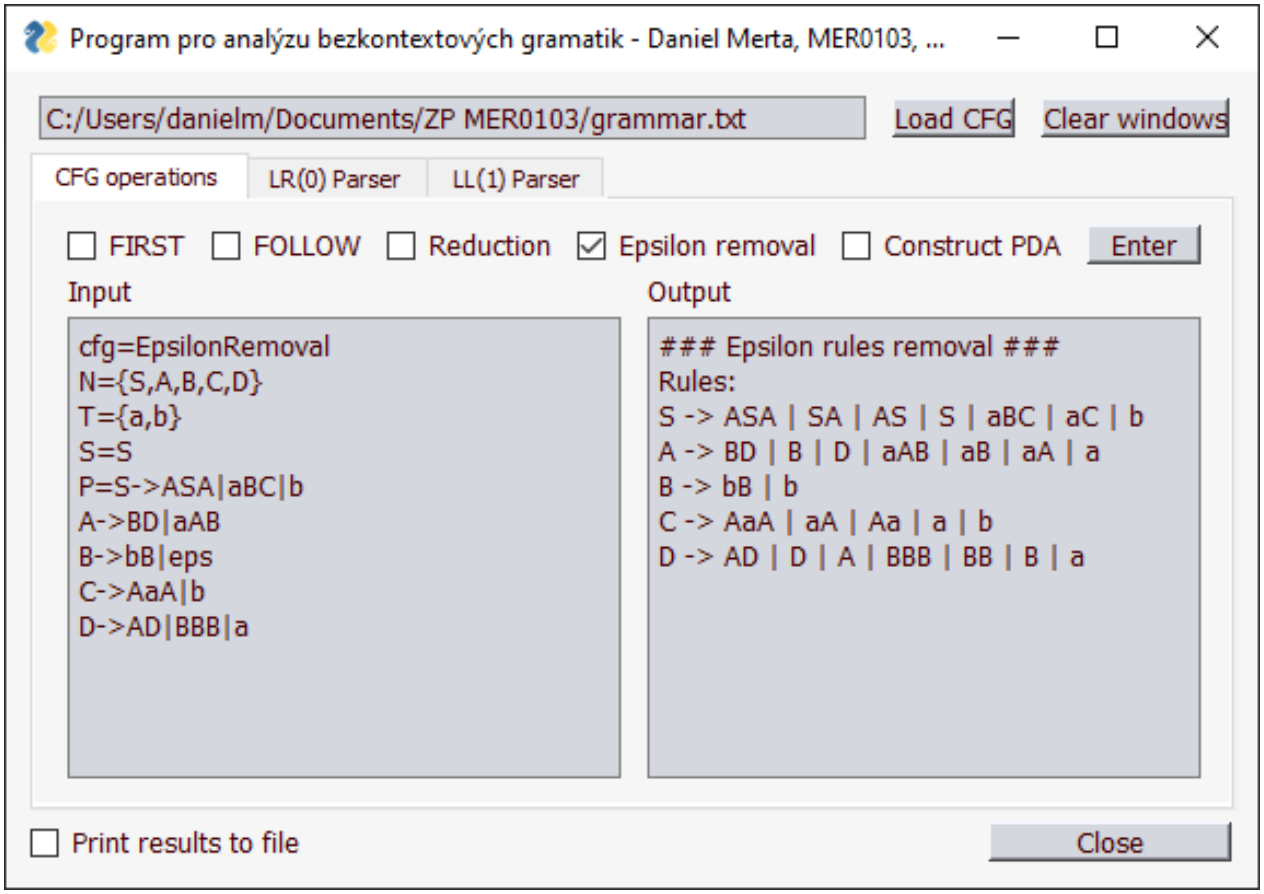


Obrázek 5.4: Redukce gramatiky

5.5 Odstranění epsilon-pravidel

Převést gramatiku do tvaru, aby neobsahovala ϵ-pravidla je možné za použití přepínače *Epsilon removal*. V tomto příkladu je zadána CFG $EpsilonRemoval = (N, T, S, P)$, kde:
$N = \{S, A, B, C, D\}$
$T = \{a, b\}$
$S = S$
$P = S \rightarrow ASA \mid aBC \mid b$
$A \rightarrow BD \mid aAB$
$B \rightarrow bB \mid eps$
$C \rightarrow AaA \mid b$
$D \rightarrow AD \mid BBB \mid a$

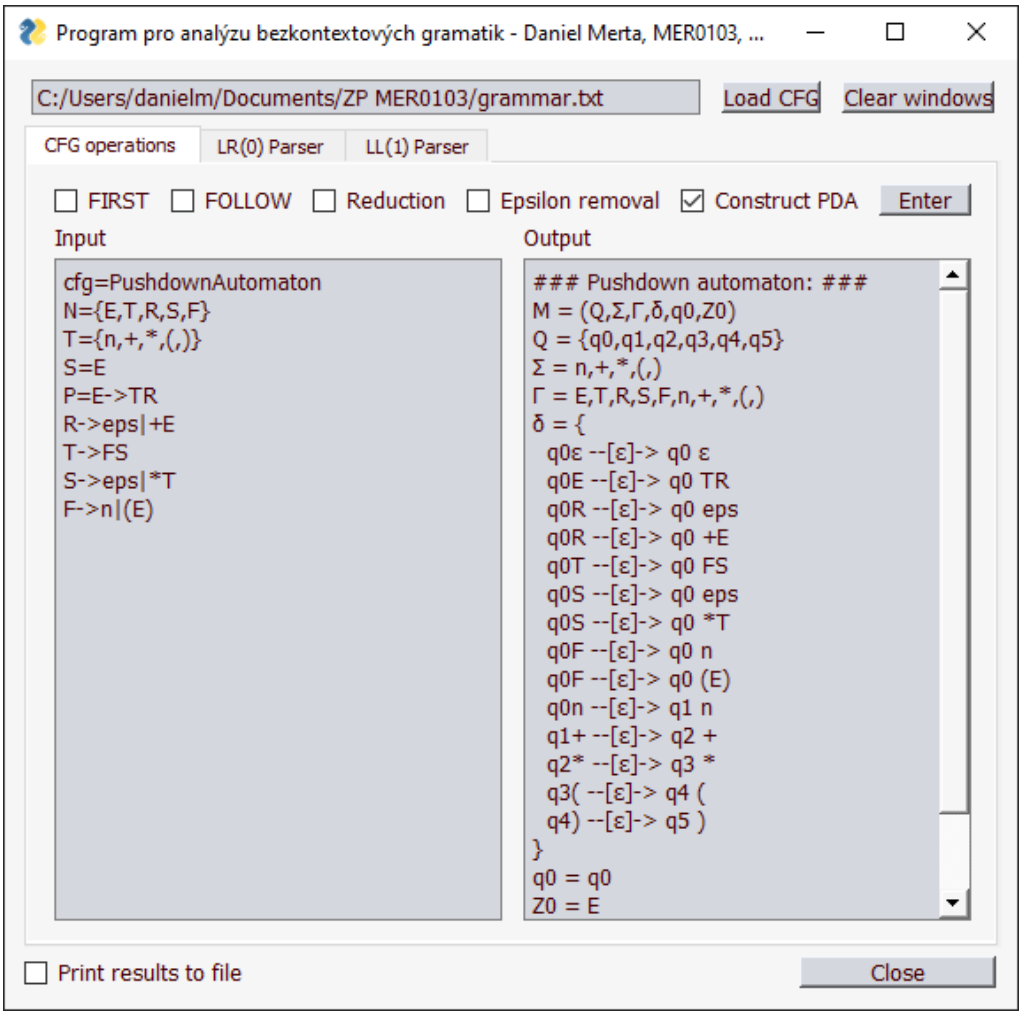
V poli *Output* na obrázku 5.5 jsou uvedena přepisovací pravidla CFG, v nichž nelze žádný neterminální symbol přepsat na ϵ. Takto převedená gramatika generuje stejný jazyk jako zadaná CFG *EpsilonRemoval*, až na již zmíněný ϵ.



Obrázek 5.5: Odstranění ϵ-pravidel

5.6 Konstrukce zásobníkového automatu

Převést CFG na odpovídající zásobníkový automat je možné výběrem volby *Construct PDA* a stisknutím tlačítka *Enter*. Je zadána CFG $PushdownAutomaton = (N, T, S, P)$, kde:
$N = \{E, T, R, S, F\}$
$T = \{n, +, *, (,)\}$
$S = E$
$P = E \rightarrow TR$
$R \rightarrow eps \mid +E$
$T \rightarrow FS$
$S \rightarrow eps \mid *T$
$F \rightarrow n \mid (E)$
Této CFG odpovídá zásobníkový automat, jehož definice je v pravém poli na obrázku 5.6.

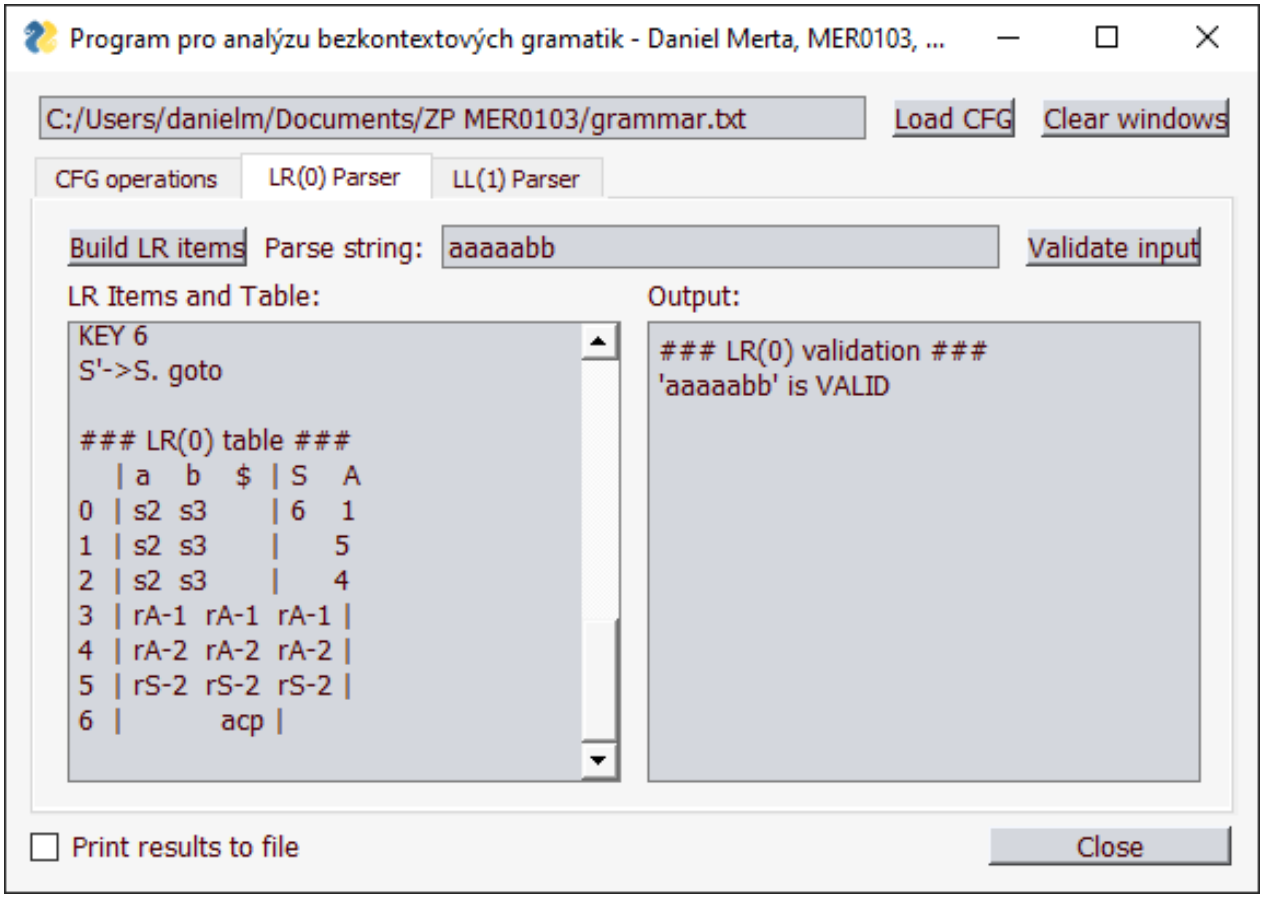


Obrázek 5.6: Konstrukce zásobníkového automatu

5.7 Detekce LR(0) gramatiky

Záložka *LR(0) Parser* umožňuje kliknutím na tlačítko *Build LR items* výpočet LR položek a sestavení LR(0) parsovací tabulky. Validaci vstupního řetězce dle pravidel z parsovací tabulky je možné provést tlačítkem *Validate input*. V poli na levé straně obrázku 5.7 jsou vypsány LR položky a parsovací tabulka, v poli napravo je výsledek validace zadaného řetězce. Je zadána CFG $LR0 = (N, T, S, P)$, kde:
$N = \{S, A\}$
$T = \{a, b\}$
$S = S$
$P = S \rightarrow AA$
$A \rightarrow aA \mid b$

V příkladu uvedeném na obrázku 5.7 je ověřeno, že řetězec *aaaaabb* je možné vygenerovat zadanou CFG typu LR(0).

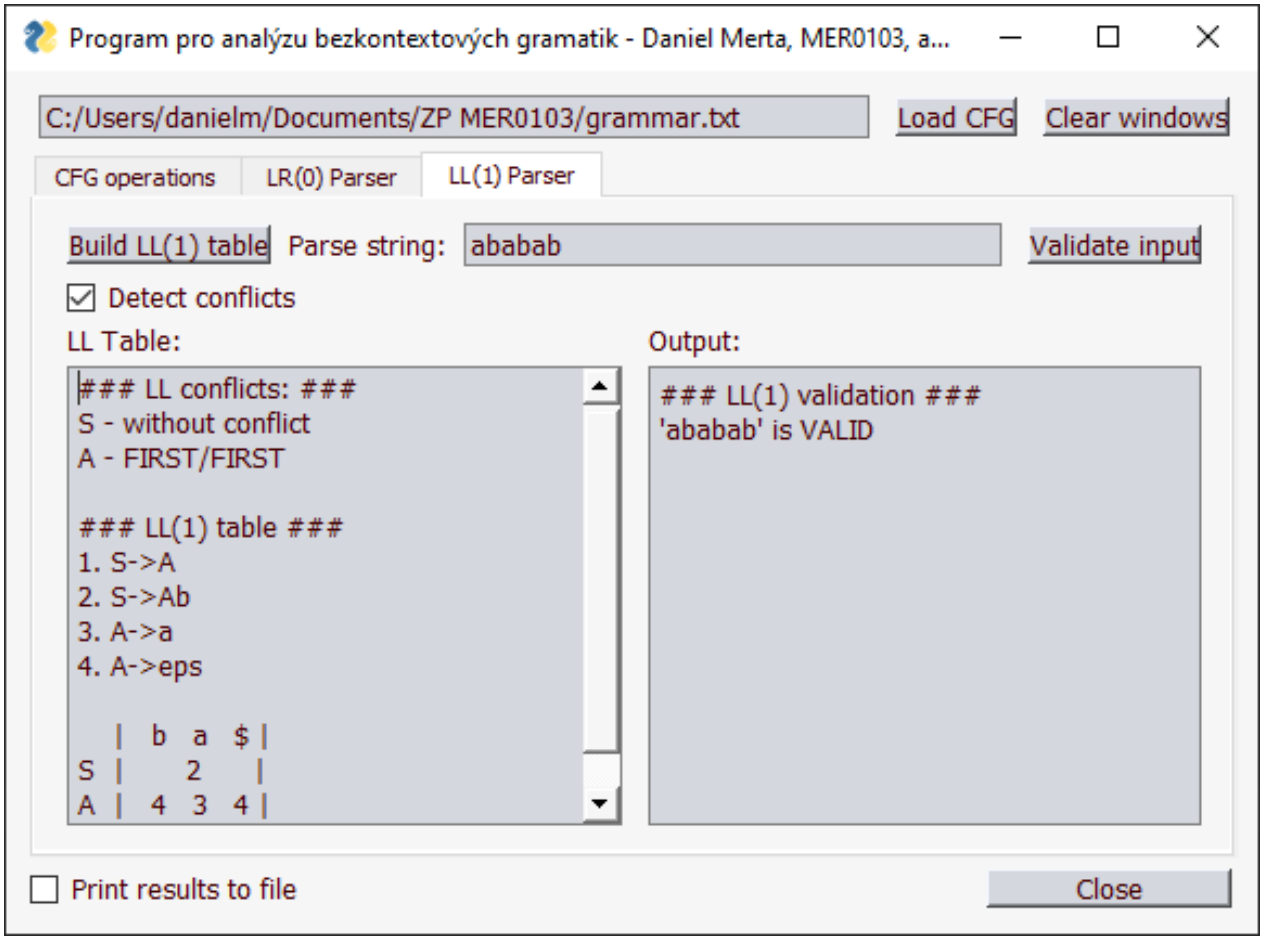


Obrázek 5.7: Detekce typu LR(0) a validace řetězce

5.8 Detekce LL(1) gramatiky a konfliktů

Třetí záložka *LL(1) Parser* umožňuje konstrukci LL(1) tabulky a dva typy konfliktů, které se v ní mohou vyskytnout. Podobně jako v 5.7, levé pole v programu obsahuje výpis případných konfliktů a LL(1) tabulku, v pravém poli je výsledek validace zadaného řetězce. Je zadána CFG $LR0 = (N, T, S, P)$, kde:
$N = \{S, A\}$
$T = \{a, b\}$
$S = S$
$P = S \rightarrow A \mid Ab$
$A \rightarrow a \mid eps$

Dle výsledku na obrázku 5.8 obsahuje neterminál A dané gramatiky konflikt typu FIRST/-FIRST, že řetězec *ababab* je možné vygenerovat zadanou CFG typu LR(0).



Obrázek 5.8: Detekce typu LL(1), validace řetězce a detekce konfliktů

Kapitola 6

Závěr

V rámci této diplomové práce byl vytvořen program v jazyce Python, který dokáže analyzovat bezkontextové gramatiky, určovat jejich typ a provádět s nimi různé operace. Aby mohl být takový program vhodně navržen, bylo zapotřebí nejprve nastudovat příslušnou problematiku, která s bezkontextovými gramatikami souvisí. Obsahem práce je konkrétní návrh tohoto programu, implementační řešení a také ukázky činností s konkrétními gramatikami.

Program by mohl najít uplatnění jako učební pomůcka ve školním prostředí, zejména v oblasti teoretické informatiky nebo matematiky. Postupy výpočtů i základních vlastností a operací jsou pro člověka netriviální a často zdlouhavé. Program umí provést výpočty velmi rychle, takže by si výsledky z něj mohl uživatel buďto porovnat se svými, nebo je přímo použít.

Program byl testován povětšinou s menšími gramatikami, aby bylo možné snadno ověřit a dokázat vypočítané hodnoty. Velikost gramatiky však nemá vliv na implementované algoritmy a z tohoto hlediska program vstupní gramatiku nijak neomezuje.

Jak je již dříve v této práci zmíněno, různých vlastností a typů gramatik existuje celá řada. Program by mohl být rozšířen o výpočty dalších vlastností jako například zjišťování různých typů rekurzí, nebo by mohl umět převádět bezkontextové gramatiky do Chomského či Greibachové normální formy. Rovněž by mohla být doimplementována detekce jiných typů gramatik, například LR(1), LALR(1), nebo schopnost načítat gramatiky ve formátu používaném nějakým generátorem parserů, např. Menhir.

Literatura

1. MAHDAVIPANAH, Hamidreza. *PyCFG* [online]. [B.r.] [cit. 2022-04-26]. Dostupné z: `https://github.com/mahdavipanah/pyCFG`.
2. ROMERO, Julien. *Pyformlang* [online]. 2020 [cit. 2022-04-27]. Dostupné z: `https://github.com/Aunsiels/pyformlang`.
3. BROWN, Chris. *Programming Languages - FirstFollowPredict Calculator* [online]. 2020 [cit. 2022-04-25]. Dostupné z: `https://www.usna.edu/Users/cs/wcbrown/courses/F20SI413/firstFollowPredict/ffp.html`.
4. CHOMSKY, N. Three models for the description of language. *IEEE Transactions on Information Theory* [online]. 1956, roč. 2, č. 3, s. 113–124 [cit. 2022-04-27]. ISSN 0018-9448. Dostupné z DOI: `10.1109/TIT.1956.1056813`.
5. SAWA, Zdeněk. *Úvod do teoretické informatiky - Bezkontextové gramatiky* [online]. Ostrava, 2021 [cit. 2022-04-27]. Dostupné z: `http://www.cs.vsb.cz/sawa/uti/slides/uti-04-cz.pdf`.
6. BÜCHI, J. Richard. *Finite Automata, Their Algebras and Grammars*. 1989. vyd. New York: Springer, New York, NY, 1989. ISBN 978-0-387-96905-3.
7. MOORE, Robert C. Removing Left Recursion from Context-Free Grammars. *Microsoft Research* [online]. [B.r.], s. 1–6 [cit. 2022-04-28]. Dostupné z: `https://www.microsoft.com/en-us/research/wp-content/uploads/2000/04/naacl2k-proc-rev.pdf`.
8. *Classification of Context Free Grammars* [online]. [B.r.] [cit. 2022-04-25]. Dostupné z: `https://www.geeksforgeeks.org/classification-of-context-free-grammars/`.
9. CHEUNG, Bruce S. N.; UZGALIS, Robert C. Ambiguity In Context-free Grammars [online]. [B.r.], s. 1–5 [cit. 2022-04-24]. Dostupné z: `https://dl.acm.org/doi/pdf/10.1145/315891.315991`.
10. BUNT, Harry; MERLO, Paola; NIVRE, Joakim. Dependency Parsing, Domain Adaptation, and Deep Parsing. In: *Dependency Parsing, Domain Adaptation, and Deep Parsing* [online]. New York: Springer Science+Business Media B.V., 2010, s. 202–208 [cit. 2022-04-27]. ISBN 978-90-481-9351-6. Dostupné z: `https://books.google.cz/books?id=IrZc9PMKy8QC&pg=`

PA207&lpg=PA207&dq=cfg+canonical+forms&source=bl&ots=-RxFgNicGb&sig=ACfU3U1i_pph6dDKHMTzBQdfzdzWXeYVTA&hl=en&sa=X&ved=2ahUKEwj684OS6_j2AhVIEncKHU9jCN4Q6AF6BAgfEAM#v=onepage&q=cfg%20canonical%20forms&f=false.

11. HOPCROFT, John E.; ULLMAN, Jeffrey D. *Introduction to automata theory, languages, and computation*. 1979. vyd. Reading: Addison-Wesley, [1979]. ISBN 978-0-201-02988-8.

12. BLUM, Norbert; KOCH, Robert. Greibach Normal Form Transformation Revisited. *Information and Computation* [online]. 1999, roč. 150, č. 1, s. 112–118 [cit. 2022-04-25]. ISSN 08905401. Dostupné z DOI: 10.1006/inco.1998.2772.

13. PRŮŠA, Daniel. *Zásobníkové automaty* [online]. 2010 [cit. 2022-04-28]. Dostupné z: https://cw.fel.cvut.cz/old/_media/courses/a7b33tin/09_zasobnikove_automaty.pdf.

14. GRUNE, Dick; JACOBS, Ceriel J. H. *Parsing techniques: a practical guide*. 2nd ed. New York: Springer, 2008. ISBN 978-0-387-20248-8.

15. ROCHE, Dr. Daniel S. *Computing PREDICT and FOLLOW sets: SI 413 - Programming Languages and Implementation* [online]. 2011 [cit. 2022-04-28]. Dostupné z: https://www.usna.edu/Users/cs/roche/courses/f11si413/c10/ff.pdf.

16. ČEŠKA, Milan; VOJNAR, Tomáš; SMRČKA, Aleš. *Teoretická informatika - TIN, Studijní opora* [online]. Brno, 2011 [cit. 2022-04-26]. Dostupné z: https://is.muni.cz/el/1433/jaro2012/IV121/um/31782618/oporaTIN.pdf.

17. KNUTH, Donald E. Top-down syntax analysis. *Acta Informatica* [online]. 1971, roč. 1, č. 2, s. 79–110 [cit. 2022-04-27]. ISSN 0001-5903. Dostupné z DOI: 10.1007/BF00289517.

18. *Syntaktická analýza: LL překlady* [online]. Opava, 2011 [cit. 2022-04-25]. Dostupné z: http://vavreckova.zam.slu.cz/obsahy/prekl/prezentace/prekl_05_LL.pdf.

19. HILFINGER, Paul N.; CAI, Jonathon; LOHSTROH, Marten. *LL Parsing and Earley's Algorithm: CS 164 Programming Languages and Compilers* [online]. 2018 [cit. 2022-04-25]. Dostupné z: https://inst.eecs.berkeley.edu/~cs164/sp18/discussion/04/04-solutions.pdf.

20. *Handling Parsing Conflicts by Unfactoring* [online]. [B.r.] [cit. 2022-04-25]. Dostupné z: http://www.cs.ecu.edu/karl/5220/spr16/Notes/Bottom-up/conflict.html.

21. *PyCharm* [online]. JetBrains, [b.r.] [cit. 2022-04-27]. Dostupné z: https://www.jetbrains.com/pycharm/.

22. *PySimpleGUIQt: Python Package Index* [online]. 2018 [cit. 2022-04-28]. Dostupné z: https://pypi.org/project/PySimpleGUIQt/.

23. DEREMER, Franklin L.; GRIES, D. Simple LR(k) Grammars: Programming Languages. 1971, roč. 1971, č. 7, s. 1–8.

24. HANDLÍŘ, Jaroslav. *Gramatické systémy a syntaxí řízený překlad založený na nich* [online]. Brno, 2017 [cit. 2022-04-26]. Dostupné z: `https://www.vut.cz/www_base/zav_prace_soubor_verejne.php?file_id=158531`.

25. KNUTH, Donald E. On the Translation of Languages from Left to Right. 1965, roč. 1965, s. 1–33.

Příloha A

Program pro analýzu bezkontextových gramatik

- *2022_MER0103_DP.ZIP* — archiv se zdrojovými kódy
  - *\CFGAnalyzer.py*
  - *\utils\helper_utils.py*
  - *\utils\print_utils.py*
  - *\utils\struct_utils.py*
  - *\grammar.txt*
  - *\sample_grammars.txt*
  - *\readme.txt*