

Konverze modelů bezkontextových jazyků

Bc. David Navrkal

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Bc. David Navrkal**

Osobní číslo: **A14475**

Studijní program: **N3902 Inženýrská informatika**

Studijní obor: **Informační technologie**

Forma studia: **kombinovaná**

Téma práce: **Aplikace pro konverzi modelů regulárních a bezkontextových jazyků**

Téma anglicky: **Applications for the Conversion of Models of Regular and Context-free Languages**

Zásady pro vypracování:

1. Vypracujte literární rešerši na dané téma.
2. Implementujte ve zvoleném programovém prostředí převodní algoritmy mezi bezkontextovou gramatikou a zásobníkovým automatem.
3. Vytvořte uživatelské rozhraní pro převodní algoritmy.
4. Implementujte převodní algoritmy do dříve vytvořené programové aplikace v rámci Bc. práce.
5. Vytvořte databázi příkladů a otestujte v rámci vytvořeného prostředí.

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

SEZNAM DOPORUČENÉ LITERATURY:

1. VANÍČEK, Jiří. Teoretické základy informatiky. 1. vyd. Praha: Kernberg, 2007, 431 s. ISBN 978-80-903962-4-1.
2. MOLNÁR, Ludovít, Milan ČEŠKA a Bořivoj MELICHAR. Gramatiky a jazyky. 1. vyd. Bratislava: Alfa, 1987. 188 s.
3. HOPCROFT, John E. a Jeffrey D. ULLMAN. Introduction to automata theory, languages, and computation. Reading: Addison-Wesley Publishing Company, 1979. 418 s., ob. ISBN 0-201-02988-X.
4. KOZEN, Dexter C. Automata and computability. New York: Springer, 1997. xiii, 400. ISBN 0-387-94907-0.
5. SIPSER, Michael. Introduction to the theory of computation. Boston: PWS Publishing Company, 1997. xv, 396 s. ISBN 0-534-94728-X.
6. ČERNÁ, Ivana, Mojmír KŘETÍNSKÝ a Antonín KUČERA. Formální jazyky a automaty I. Elportál, Brno: Masarykova univerzita, 2006. ISSN 1802-128X..
7. XAVIER, S. Theory of automata, formal languages and computation. New Delhi: New Age International (P) Ltd., Publishers, c2005, xiii, 345 p. ISBN 978-81-224-2334-1.
8. CHYTIL, Michal. Sbírka řešených příkladů z teorie automatů a formálních jazyků. 1. vyd. Praha: Univerzita Karlova, 1987. 111 s.

Vedoucí diplomové práce:

doc. Ing. Roman Šenkeřík, Ph.D.

Ústav informatiky a umělé inteligence

Datum zadání diplomové práce:

3. února 2017

Termín odevzdání diplomové práce:

16. května 2017

Ve Zlíně dne 3. února 2017



doc. Mgr. Milan Adámek, Ph.D.
děkan



prof. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky. Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně

.....

podpis autora

ABSTRAKT

V této práci se budeme zabývat tím, jak didakticky prezentovat studentům formálních jazyků převody modelů bezkontextových jazyků. Dočtete se zde o teorii bezkontextových jazyků s množstvím příkladů. Budeme se zde zabývat modely bezkontextových jazyků a to zásobníkovým automatem a bezkontextovou gramatikou. U bezkontextové gramatiky bude představena i její nejpoužívanější forma a o je Backusova-Naurova forma. Dále v teorii bude prezentován algoritmus pro převod bezkontextové gramatiky na zásobníkový automat. Praktická část se bude zabývat popisem uživatelského rozhraní pro převodní algoritmy a také zajímavými implementačními detaily.

Klíčová slova: Bezkontextové jazyky, zásobníkový automat, bezkontextová gramatika, Backusova-Naurova forma, převod bezkontextové gramatiky na zásobníkový automat, Qt.

ABSTRACT

In this thesis we will deal how to didactically present conversion conversions of models of context-free languages to the students. You can read here about the theory of context-free languages with plenty of examples. We will discuss models of context-free languages such as pushdown automaton and context-free grammar. For context-free grammars will be presented as well its most widely used form the Backus-Naur form. Furthermore, in the theory will be presented algorithm for conversion of context-free grammar to pushdown automata. Practical part will deal with description of user interface for the conversion algorithms and interesting implementation details.

Keywords: Context-free languages, pushdown automata, context-free grammar, Backus-Naur form, conversion of context-free grammar to pushdown automata, Qt.

Zde bych chtěl poděkovat za vedení a konzultace mé diplomové práce panu doc. Ing. Romanu Šenkeříkovi, Ph.D.

Další poděkování patří autorovi L^AT_EX šablony diplomové práce, panu Ing. Jozefu Říhovi, která je nyní mnohem přehlednější, dá snadněji zkompileovat pomocí programu pdf_latex a také, že jsem díky tomu mohl používat obrázky ve formátech jpg a png.

OBSAH

| | |
|--|-----------|
| ÚVOD | 9 |
| I TEORETICKÁ ČÁST | 10 |
| 1 CHOMSKÉHO HIERARCHIE | 12 |
| 2 BEZKONTEXTOVÝ JAZYK | 14 |
| 3 ZÁSOBNÍKOVÝ AUTOMAT | 15 |
| 3.1 ŠÍPKOVÁ NOTACE PŘECHODOVÝCH PRAVIDEL ZÁSOBNÍKOVÉHO AU- TOMATU | 16 |
| 4 BEZKONTEXTOVÁ GRAMATIKA | 17 |
| 4.1 BACKUSOVA-NAUROVA FORMA | 18 |
| 5 ALGORITMUS PŘEVODU CFG NA PDA | 19 |
| 6 ZÁKLADY QT FRAMEWORKU | 21 |
| 6.1 QWIDGET | 21 |
| 6.2 POPIS UŽIVATELSKÉHO ROZHRAŇÍ POMOCÍ UI SOUBORŮ | 21 |
| 6.3 PROMOTING | 21 |
| 6.4 SIGNÁLY A SLOTS | 22 |
| II PRAKTICKÁ ČÁST | 22 |
| 7 ÚVODNÍ OBRAZOVKA | 25 |
| 8 MOŽNOSTI ZAČÁTKU PRÁCE S PROGRAMEM | 26 |
| 8.1 VÝBĚR Z PŘÍKLADŮ | 26 |
| 8.2 VÝBĚR Z KONVERZÍ | 26 |
| 8.3 UKLÁDÁNÍ A NAČÍTÁNÍ KONVERZÍ | 26 |
| 9 KONVERZNÍ MÓDY | 28 |
| 9.1 MÓD KROKOVÁNÍ ALGORITMU | 28 |
| 9.2 MÓD SAMOSTATNÉ PRÁCE | 28 |
| 9.3 MÓD AUTOMATICKÉ KONTROLY | 29 |
| 10 POPIS KONVERZE BEZKONTEXTOVÉ GRAMATIKY NA ZÁSOB- NÍKOVÝ AUTOMAT | 30 |
| 10.1 VSTUPNÍ WIDGET BEZKONTEXTOVÉ GRAMATIKY | 30 |
| 10.2 ALGORITMICKÝ WIDGET | 31 |
| 10.3 VÝSTUPNÍ WIDGET ZÁSOBNÍKOVÉHO AUTOMATU | 31 |
| 10.4 WIDGET S AKTUÁLNÍMI PROMĚNNÝMI Z ALGORITMICKÉHO WIDGETU | 33 |
| 11 IMPLEMENTACE | 37 |

| | | |
|--------|--|-----------|
| 11.1 | STAŽENÍ ZDROJOVÝCH SOUBORŮ A JEJICH NÁSLEDNÁ KOMPILACE..... | 37 |
| 11.2 | REFAKTOR EXISTUJÍCÍHO KÓDU | 39 |
| 11.2.1 | Konvence pojmenování částí kódu | 39 |
| 11.3 | ZOBRAZOVÁNÍ NÁHLEDŮ PŘÍKLADŮ | 39 |
| 11.4 | DROBNÉ VYLEPŠENÍ EXISTUJÍCÍHO KÓDU | 40 |
| 11.5 | ZAPOUZDŘENÍ KONVERZÍ | 40 |
| 11.6 | TŘÍDA CCFGToPDAWIDGET ZAPOUZDŘUJÍHÍ KONVERZI CFG NA PDA | 41 |
| 11.7 | TŘÍDA CALGORITHMCFGtoPDA OBSAHUJÍCÍ LOGIKU KONVERZE CFG NA PDA..... | 42 |
| 11.8 | PARSOVÁNÍ BEZKONTEXTOVÉ GRAMATIKY V BACKUS NAUR FORMĚ .. | 43 |
| 11.9 | PARSOVÁNÍ PŘECHODOVÝCH PRAVIDEL KONEČNÉHO AUTOMATU..... | 44 |
| | ZÁVĚR..... | 46 |
| | SEZNAM POUŽITÉ LITERATURY | 48 |
| | SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK | 49 |
| | SEZNAM OBRÁZKŮ | 50 |
| | SEZNAM TABULEK | 51 |
| | SEZNAM PŘÍLOH | 52 |
| 1.1 | QT MODELEDITOR..... | 53 |
| 1.2 | DOXYGEN A GRAPHVIZ..... | 53 |
| 1.3 | DIAGRAM TŘÍD VE VISUAL STUDIO EXPRESS 2015 | 54 |
| 1.4 | SIMPLE TOOL TO VISUALIZE CONNECTIONS BETWEEN SIGNALS AND SLOTS IN QT | 54 |

ÚVOD

Tato práce pojednává o vývoji aplikace která má didakticky prezentovat studentům teoretické informatiky oblast bezkontextových jazyků. Aplikace navazuje na bakalářskou práci [1], jejíž praktickou částí bylo implementovat vybrané konverze modelů regulárních jazyků.

Toto téma jsem zvolil z důvodu, že oblast formálních jazyků může být pro studenty hůře pochopitelná a pomocí zde implementované aplikace si tak mohou tuto náročnou teorii procvičit na praktických příkladech.

Aplikace by měla umožňovat jak pochopení teorie, tak si ji i samostatně vyzkoušet, proto obsahuje několik módu, které tuto funkcionalitu umožňují. Dále by měla být lokalizovaná v anglickém jazyce, aby z ní měli užitek i zahraniční studenti.

V teoretické části je definovaná teorie nezbytná pro to pochopení konverzí a modelů bezkontextových jazyků. Dočtete se zde o Chomského hierarchii, bezkontextových jazycích a jejich modelech. Jako první je představen zásobníkový automat spolu s šipkovou notací přechodových pravidel. Druhým modelem je bezkontextová gramatika a její Backusova-Naurova forma. Dále je definován algoritmus převodu bezkontextové gramatiky na zásobníkový automat. Závěrem v teoretické části jsou vysvětleny základy Qt frameworku.

Dále v praktické části je popsán samotný program jak z pohledu vnějšího, ve kterém jsou ukázány všechny jeho funkce a jak se ovládá, tak vnitřního, kde jsou popsány některé zajímavé implementační detaily.

Pro uživatele je zde vysvětleno, jak začít aplikaci používat. Uživatel má na výběr z předdefinovaných příkladů, načtení konverze ze souboru a možnosti vložit vlastní data. Dále následuje popis několika módu určených pro studium konverzí a jejich samostatné procvičení. Následuje popis okna samotné konverze bezkontextové gramatiky na zásobníkový automat, které se skládá z widgetů pro jejich editaci a těch ve kterých probíhá samotné krokování a zobrazování pomocných proměnných algoritmu.

Zdrojové kódy a tento text, jsou dostupné pod open-source licencí na serveru GitHub pod touto url <https://github.com/navrkald/regularConvertor>.

Jak již napovídá název serveru Github, celá práce včetně zdrojových kódů je verována pomocí Gitu, který je navržen decentralizovaně a tak je zaručeno, že ať se stane cokoliv, záloha dat bude vždy z více zdrojů dostupná. Díky commitům je možné monitorovat postup prací, také je možné nový kód ukládat do větví, což umožňuje mít stabilní větev pro koncové uživatele a vývojovou větev ve které se vyvíjí nová funkcionalita. Jelikož jsou zdrojové kódy veřejně přístupné, tak kdokoli může přispět k v vývoji této výukové aplikace pomocí takzvaných *pull requestů*.

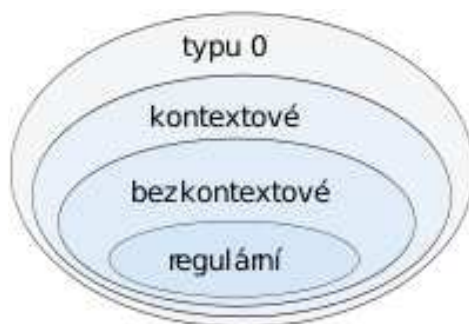
Pro vývoj tohoto programu byl zvolen jazyk C++ ve verzi standardu 11 spolu s multiplatformním frameworkem Qt v nejnovější verzi 5.8, protože požadavky byly, aby aplikace používala nové moderní technologie a také, aby byla přenositelná na všechny hlavní operační systémy.

Tato práce na rozdíl od bakalářské práce, která byla psána v programu Open Office, používá sázecí program L^AT_EX [2]. Tento typografický program umožňuje lepší kontrolu nad formátováním a výsledným vzhledem, hodí se pro technické texty a je s ním snadnější dodržování typografických pravidel. Také umožňuje mít práci uloženou v textové podobě, která se snadněji verzuje, nežli pokud by byla v binární formě. Díky tomu je snadnější porovnávání verzí, a menší výsledná velikost Gitového repositáře.

I. TEORETICKÁ ČÁST

1 Chomského hierarchie

Abychom si mohli dát bezkontextové jazyky do souvislosti s ostatními formálními jazyky, je nutné si vysvětlit Chomského hierarchii. Byla vytvořena panem Noamem Chomským v roce 1956. Dává do souvislosti jazyky regulární, bezkontextové, kontextové a jazyky typu 0. Jejich souvislost je naznačena na obrázku (Obr. 1.1) jako vztah množin.



Obr. 1.1 Chomského hierarchie formálních jazyků

Jak je vidět z obrázku regulární jazyky zauímají nejmenší množinu, nebo chcete-li nejmenší počet jazyků. Nenechte se však zmýlit pojmem počet jazyků, protože počet jazyků a množství řetězců generovaných daným jazykem jsou dvě rozdílné věci. Například jednoduchým regulárním výrazem $(a + b)^*$ lze popsat všechny slova nad abecedou $\Sigma = \{a, b\}$. Avšak jazyk, všech slov, kde první půlka slova se skládá ze stejného počtu písmen 'a' jako písmen 'b' v druhé půlce slova regulárním jazykem nepopíšeme. Dalo by se tak říci, že čím výše se jazyk nachází v Chomského hierarchii, tím přesněji dokážeme jazyk popsat, jinými slovy, tím více jazyků jsme schopni generovat.

Nyní si pro lepší představu u každého jazyku uvedeme modely, které daný jazyk popisují s jedním příkladem konkrétního jazyka.

Regulární jazyky můžeme popsat pomocí regulární gramatiky, regulárního výrazu nebo konečného automatu. Příkladem je například, jazyk který obsahuje libovolnou kombinaci písmen 'a' a 'b' končící písmenem 'b'. Tento jazyk by se dal popsat regulárním výrazem $(a + b)^*b$.

Jazyky bezkontextové lze popsat modely jakými jsou bezkontextové gramatiky a konečný zásobníkový automat. Všechny jazyky regulární jsou zároveň bezkontextové, avšak obrácené tvrzení neplatí. Příkladem je jazyk $L = \{a^n b^n : n \geq 1\}$

Jazyky kontextové můžeme popsat lineárně ohraničeným Turingovým strojem. Příkladem nechť je jazyk $L = \{a^n b^n c^n : n \geq 1\}$

Třída jazyků typu nula, v sobě obsahuje všechny doposud zmíněné jazyky plus ještě

jazyky navíc. Popisujeme je úplným Turingovým strojem. [3]

2 Bezkontextový jazyk

V předchozí kapitole byl uveden vzájemný vztah formálních jazyků. Nyní se pojďme podívat podrobněji na bezkontextové jazyky. Tyto jazyky popisujeme pomocí dvou hlavních modelů, kterými jsou bezkontextová gramatika (CFG) a konečný zásobníkový automat (PDA).

Nenechme se však zmýlit představou, že pomocí obou můžeme generovat bezkontextové jazyky. Protože pomocí bezkontextové gramatiky můžeme jazyk generovat, avšak pomocí konečného zásobníkového automatu můžeme rozpoznat, zda konkrétní jazyk je popsán daným automatem.

Důležitá skutečnost je, že oba popisy jsou vzájemně ekvivalentní, tj. že lze převést PDA na CFG a taktéž lze převést CFG na PDA. Formální popis bezkontextové gramatiky a zásobníkového automatu bude uveden dále spolu s konverzí CFG na PDA.

Pro lepší představu, jaké jsou to vlastně ty bezkontextové jazyky, si nyní uvedeme pár příkladů. Krásným příkladem z praxe by mohl být jazyk závorek, tak aby odpovídal počet levých závorek počtu závorek pravých. Složitějším příkladem by mohl být jazyk matematicky správných algebraických výrazů, který se skládá z operátorů plus, mínus, krát, děleno, symbolů 'x', 'y', 'z' a závorek. Dalším příkladem z praxe je, že velká podmnožina programovacích jazyků se dá popsat pomocí bezkontextové gramatiky.

Jak je vidět z těchto příkladů, pomocí bezkontextového jazyka se dají popsat složitější věci a jejich uplatnění můžeme nalézt u popisu programovacích jazyků. Existuje notace bezkontextové gramatiky, zvaná Backusova-Naurova forma, která se právě často používá pro popis syntaxe programovacích jazyků. Znalost bezkontextových jazyků lze využít při konstrukci překladače, nebo interpretu programovacího jazyka a také při počítačovém zpracování algebraických výrazů. [4]

3 Zásobníkový automat

Následuje definice zásobníkového automatu. [5]

Konečný zásobníkový automat M je uspořádaná sedmice $M = (Q, \Sigma, \Gamma, R, q_0, S, F)$ kde:

- Q je konečná množina vnitřních stavů,
- Σ je konečná vstupní abeceda,
- Γ je konečná abeceda zásobníku,
- R je přechodová relace, popisující pravidla činnosti automatu, je definovaná jako konečná množina kartézského součinu $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times Q \times \Gamma^*$,
- q_0 je počáteční stav z množiny Q ,
- S je počáteční symbol na vrcholu zásobníku z množiny Γ ,
- F je množina koncových stavů, $F \subseteq Q$.

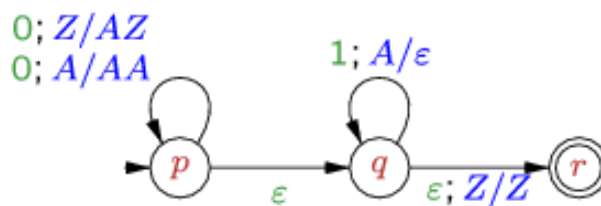
Definice 1 *Definice zásobníkového automatu.*

Pro lepší představu je uveden příklad zásobníkového automatu M , který popisuje jazyk $L = \{0^n 1^n, \text{ kde } n \geq 0\}$.

Nechť M je definován následovně:

- $Q = \{p, q, r\}$,
- $\Sigma = \{0, 1\}$,
- $\Gamma = \{A, Z\}$,
- $R = (p, 0, Z, p, AZ), (p, 0, A, p, AA), (p, \epsilon, Z, q, Z), (p, \epsilon, A, q, A), (q, 1, A, q, \epsilon), (q, \epsilon, Z, r, Z),$
- $q_0 = p$,
- $S = Z$,
- $F = \{r\}$.

Graficky znázorněný stejný zásobníkový automat by vypadal následovně:



Obr. 3.1 Příklad zásobníkového automatu

3.1 Šipková notace přechodových pravidel zásobníkového automatu

Pro větší přehlednost a čitelnost přechodových pravidel zásobníkového automatu je zde uvedena definice šipkové notace, která bude použita dále v této práci.

Přechodová pravidlo $r \in R$ konečného zásobníkového automatu

$$M = (Q, \Sigma, \Gamma, R, q_0, S, F)$$

Definujeme následovně:

$r = A pa \rightarrow wq$, kde p a $q \in Q$, $A \in \Gamma$, $a \in \Sigma$ a w je řetězec symbolů náležících do Γ .

Definice 2 *Definice přechodových pravidel zásobníkového automatu v šipkové notaci.*

4 Bezkontextová gramatika

Definice bezkontextové gramatiky je následující. [6]

Bezkontextová gramatika G je uspořádaná čtveřice $G = (V, \Sigma, R, S)$ kde:

- V je konečná množina non-terminálních symbolů,
- Σ je konečná množina terminálních symbolů,
- R je konečná množina relací z V do $(V \cup \Sigma)^*$, taktéž se nazývá množinou přepisovacích pravidel,
- $S \in V$ se nazývá počáteční non-terminální symbol gramatiky.

Definice 3 *Definice bezkontextové gramatiky.*

Jednotlivé přepisovací pravidla se často zapisují pomocí „šipkové notace“, kde je na levé straně non-terminál, za ním následuje šipka a na pravé straně je řetězec non-terminálních a terminálních symbolů. Přepisovací pravidla se nazývají protože, řetězce daného jazyka, které jsou popsány gramatikou vznikají tak, že na počátku máme řetězec skládající se z počátečního non-terminálního symbolu gramatiky S a na ten aplikuji jedno z přepisovacích pravidel. Pokračuji tím, že postupně nahrazuji non-terminální symboly v řetězci do té doby, dokud mě v něm nezbudou jen terminální symboly a to je pak jedno konkrétní slovo daného bezkontextového jazyka.

Například gramatika $G = (\{S, B\}, \{a, b\}, R = \{\{S \rightarrow aBb\}, \{B \rightarrow aBb\}, \{B \rightarrow \varepsilon\}\}, S)$ popisuje jazyk $L = \{a^n b^n : n \geq 1\}$.

Příklad konkrétního slova, které generuje tato gramatika by mohl vypadat následovně:

1. Na začátku máme slovo skládající se z počátečního symbolu S .
2. Po aplikaci přepisovacího pravidla $S \rightarrow aBb$ dostaneme slovo aBb .
3. Toto pravidlo aplikujeme ještě jednou a dostáváme slovo $aaBbb$.
4. Nakonec aplikujeme pravidlo $B \rightarrow \varepsilon$ a dostáváme slovo $aa\varepsilon bb$.
5. Jelikož ε značí prázdný řetězec, tak výsledné slovo jazyka L je $aabb$. Slovo se nám skládá jen z terminálních symbolů, takže jsme s generováním slova u konce.

4.1 Backusova-Naurova forma

Pro zápis bezkontextové gramatiky se používají převážně dvě notace a to Backusova-Naurova forma (BNF) a jako druhá v pořadí je Van Wijngaardenova gramatika. My si zde představíme pouze první z nich, protože s BNF se můžete setkat častěji. Bezkontextovou gramatikou můžeme popsat různé programovací jazyky, jedním z nich je například Algol. [6]

Syntaktická přepisovací pravidla jsou definována tak, že levou stranu vždy tvoří non-terminál, za kterým následuje oddělovač „::=” (dvě dvojtečky následované rovnítkem) za kterým následuje pravá strana pravidla, která se skládá z řetězců non-terminálů a terminálů. Pravá strana se může skládat z více řetězců oddělených symbolem „|“ značící logický operátor OR. Tento operátor byl zaveden, kvůli optimalizaci zápisu pravidel, tak aby místo zápisu, že jeden non-terminální symbol se může rozepsat na N řetězců, což by muselo být rozepsáno na N pravidel na N řádcích. Místo toho se těchto N řetězců napíše na pravou stranu oddělených symbolem „|“. Terminální symboly jsou uvedeny v uvozovkách. [7]

Gramatika pro popis číselné konstanty by mohla vypadat takto:

$$\begin{aligned} \langle \text{konstanta} \rangle &::= \langle \text{číslice} \rangle \mid \langle \text{konstanta} \rangle \langle \text{číslice} \rangle \\ \langle \text{číslice} \rangle &::= „0“ \mid „1“ \mid „2“ \mid „3“ \mid „4“ \mid „5“ \mid „6“ \mid „7“ \mid „8“ \mid „9“ \end{aligned}$$

Gramatika který by popisovala základní aritmetické operace by měla navíc tato pravidla:

$$\begin{aligned} \langle \text{výraz} \rangle &::= \langle \text{výraz} \rangle „+“ \langle \text{výraz} \rangle \\ \langle \text{výraz} \rangle &::= \langle \text{výraz} \rangle „-“ \langle \text{výraz} \rangle \\ \langle \text{výraz} \rangle &::= \langle \text{výraz} \rangle „\times“ \langle \text{výraz} \rangle \\ \langle \text{výraz} \rangle &::= \langle \text{výraz} \rangle „/“ \langle \text{výraz} \rangle \\ \langle \text{výraz} \rangle &::= „(“ \langle \text{výraz} \rangle „)“ \\ \langle \text{výraz} \rangle &::= \langle \text{konstanta} \rangle \end{aligned}$$

5 Algoritmus převodu CFG na PDA

Nyní si pojdme ukázat algoritmus převodu bezkontextové gramatiky na zásobníkový automat. [8]

Vstup: CFG $G = (V, \Sigma, R, S)$

Výstup: Nedeterministický PDA $M = (Q, \Sigma, \Gamma, R, q_0, S, F)$, který přijímá vstupní řetězec vyprázdněním zásobníku.

1. $Q = q_0$
2. $\Sigma = \Sigma$
3. $\Gamma = V \cup \Sigma$
4. Počáteční stav je q_0
5. Konstrukce přechodové množiny R :
 - (a) Pro každé $a \in \Sigma$: přidej $(q_0, a, a, q_0, \varepsilon)$ do R
 - (b) Pro každé $A \rightarrow x$, kde $A \in V, x \in (V \cup \Sigma)^*$, přidej do $R(q_0, \varepsilon, A, q_0, y)$, kde $y = reversal(x)$
6. $S = S$
7. $G = \emptyset$

Algoritmus 1 Převod CFG na PDA.

Po aplikaci tohoto algoritmu nám vznikne nedeterministický PDA s jedním vstupním stavem a žádným ukončovacím stavem. Tento automat přijímá vstupní řetězec vyprázdněním zásobníku. Má stejnou vstupní abecedu, jako byla abeceda terminálů u původní CFG. Zásobníková abeceda automatu vznikla sloučením množin terminálů a non-terminálů vstupní gramatiky. Počátečním symbolem na zásobníku je počáteční non-terminální symbol gramatiky.

První část množiny přechodových pravidel vznikla tak, že pro všechny terminální symboly se přidala přechodová pravidla taková, že počáteční i koncový stav pravidla je vždy počáteční stav automatu. Zároveň aktuální symbol na pásce a stejně tak na vrcholu zásobníku musí být stejný terminální symbol. Znak ε říká, že symbol z vrcholu zásobníku odstraním.

Zbytek přechodových má také počáteční a koncový stav stejný. ε zde říká, že nečtu žádný ze symbolů na pásce, ale přečtu jeden non-terminál z vrcholu zásobníku a nahradím jej řetězcem znaků z pravé části pravidla napsaných pozpátku.

6 Základy Qt frameworku

Qt framework je multiplatformní, to znamená, že programátor napíše jeden kód, který pak zkompile na více cílových systémů. Konkrétně *Regular Convertor* jsem úspěšně zkoušel na platformách Windows a Linux. Tento framework je určen především pro jazyk C++. Obsahuje také IDE pojmenované QtCreator, ve kterém je i integrován nástroj Qt Designer pro tvorbu uživatelského rozhraní vizuální formou a ne jenom pomocí kódu. [9]

6.1 QWidget

QWidget je základní stavební element a zároveň předek v hierarchii tříd všech grafických elementů uživatelského rozhraní. Například widget je předek všech tlačítek, oken, kontrolních prvků, dialogů a dalších. Samotný widget sám o sobě nemá význam, ale definuje a zaštiťuje společnou množinu vlastností pro všechny od něj odvozené třídy.

6.2 Popis uživatelského rozhraní pomocí ui souborů

Ui soubory jsou ve formátu xml, ale needitují se přímo, nýbrž pomocí nástroje zvaného QtDesigner. Dále se při kompilaci zpracovávají a generuje se z nich C++ kód. Slouží k popisu rozmístění a designu jednotlivých buď předdefinovaných nebo uživatelských widgetů. Je vhodné v aplikaci striktně oddělovat samotnou funkčnost od popisu uživatelského rozhraní. Obvykle se používá technika, že funkčnost je definovaná programátorem v přidružené C++ třídě ve které je vložena vnitřní "ui" třída, která je automaticky generovaná a která obsahuje jako členské proměnné jednotlivé vnitřní widgety. V této práci jsem přesunul všechny popisy konverzí do samostatných ui souborů, tak abych zmenšil hlavní třídu okna a více oddělil kód od popisu grafického uživatelského prostředí.

6.3 Promoting

Spolu s tím, že QWidget je původní předek všech grafických elementů (a proto je předek i všech mých uživatelských elementů) jsem mohl v QtDesigneru použít techniku zvanou *promoting*. Volně do češtiny by se to dalo přeložit jako povyšování widgetů. Pro všechny konverze jsem si následně vytvořil widgety, které se skládají z jiných také mnou vytvořených widgetů. Proto jsem si vytvořil rozložení widgetů, které jsem následně povýšil na mnou skutečně předem vytvořené widgety.

Má to tu nespornou výhodu, že jsem se zbavil všude v kódu, programového vytváření obrazovky pro jednotlivé konverze, které jsou nově definovány jako samostatné třídy, které zapouzdřují z grafického klediska všechny konverze. Tímto, že se popis UI přesunul

do samostatných souborů, se značně ulevilo třídě pro popis hlavního okna aplikace. Tato objektově orientované technika rozdělení aplikace do více menších částí a rozdělení pravomocí zvaná zapouzdření.

6.4 Signály a sloty

Signály a sloty se podobá jiné programovací technice zvaná *callbacks*, kde se předává ukazatel na funkci nebo metodu.

Slot je metoda třídy, která se vykoná pokud někdo emitoval signál se kterým je spojená pomocí funkce *connect()*. Stejně jako můžeme propojit emitování signálu s vykonáváním určité metody zvané slot, můžeme později zavolat i opačnou metodu *disconnect()*, která toto spojení zruší.

K jednomu signálu může být propojeno i více slotů. Tak jsem například implementoval, že konečný automat má více grafických reprezentací, tak aby se na základě emitování signálu o změně konečného automatu změnily obě jeho vizuální reprezentace.

Slot musí mít definované svoje tělo, které se vykonává při emitování signálu, pokud jsou propojeni. Naopak tělo signálu nesmí být nikdy definováno.

Signály a sloty si mohou vyměňovat informace pomocí parametrů. V nižších verzích Qt fungovalo propojení na základě maker *SIGNAL* a *SLOT*. V aktuální verzi Qt se může použít i systém ukazatelů na funkce, které jsem použil na propojení konverzních widgetů s hlavním oknem aplikace.

II. PRAKTICKÁ ČÁST

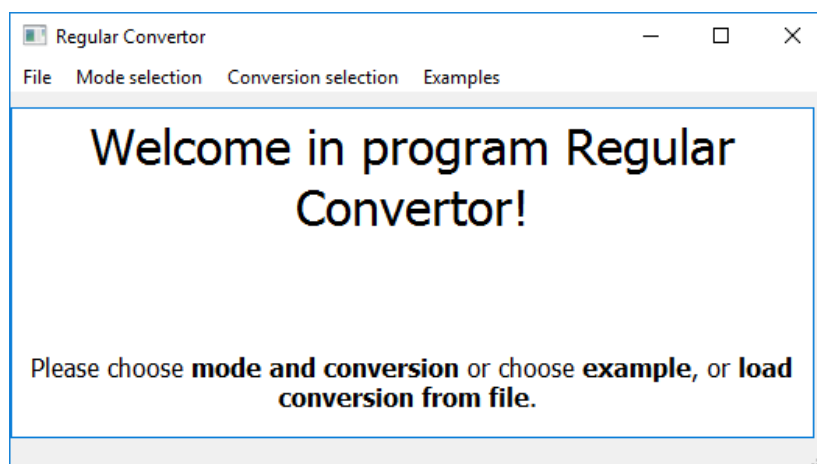
Na začátku této části své diplomové práce vám představím aplikaci *Regular Converter* z uživatelského hlediska, z pohledu chování, ovládání a funkcionalit. Konec věnuji implementaci.

Na úvod jsem chtěl zmínit, že samotnou aplikaci jsem lokalizoval do anglického jazyka, protože jsem se nechtěl omezovat pouze na česky mluvící studenty. Texty z aplikace zde budu citovat anglicky, ale popisovat je budu v češtině.

Aplikace by měla sloužit k účelům prezentace formálních jazyků na přednáškách a zároveň i k samostatnému procvičení studenty. Proto obsahuje část, kde se prezentuje jak vlastní algoritmus funguje, tak i části kde si student nasbíranou teorii prakticky vyzkouší. Důležité je, aby uživatel mohl s danými algoritmy experimentovat, proto program umožňuje vložení libovolného vstupu, s následnou kontrolou jeho validity. Tento vstup si může uživatel vložit ručně a nebo jej načíst ze souboru. Proto lze všechny konverze a jejich módy, včetně vstupních i výstupních dat ukládat a načítat ze souborů. Aplikace obsahuje i řadu zabudovaných příkladů, které jsou členěny od jednodušších ke složitějším.

7 Úvodní obrazovka

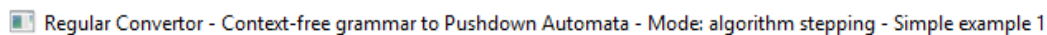
Na obrázku (Obr. 7.1) je pro uživatele připraveno uvítání a zároveň instrukce jak dále postupovat. Uživatel má možnost načíst si již předem připravenou konverzi ze souboru, nebo si otevřít příklad konkrétní konverze a na závěr má možnost si zvolit vlastní konverzi, které předá vstupní data.



Obr. 7.1 Úvodní obrazovka programu Regular Convertor

V záhlaví hlavního okna se zobrazují informace pro snadnější orientaci uživatele, který zde vidí, že spuštěný program se jmenuje "Regular Convertor", dále pak jakou konverzi má vybranou, následuje pak jaký konverzní mód má zvolen a jako poslední je uveden název příkladu, který si uživatel vybral. Pokud si nezvolil žádný z příkladů, tak se jeho název nezobrazuje.

Příklad takového záhlaví hlavního okna, kde je zvolena konverze bezkontextové gramatiky na zásobníkový automat, vybrán je mód krokování algoritmu a první příklad, můžete vidět na obrázku (Obr. 7.2).



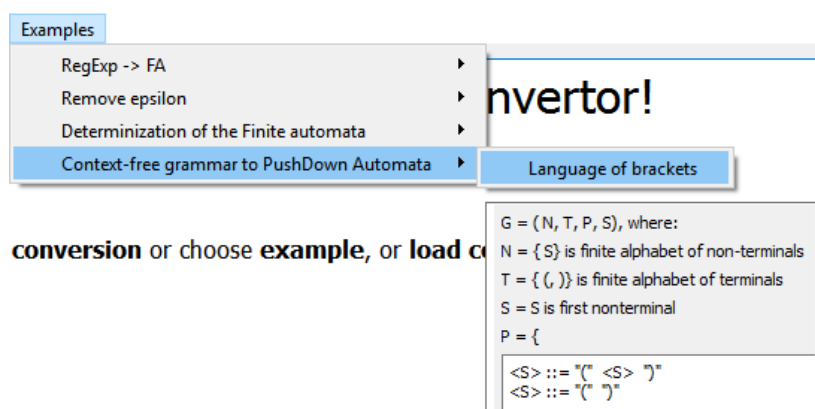
Obr. 7.2 Výřez záhlaví hlavního okna

8 Možnosti začátku práce s programem

V předchozí části bylo popsáno hlavní okno spolu s instrukcemi pro uživatele, jak může začít pracovat s programem. Nyní si je pojďme popsat podrobněji.

8.1 Výběr z příkladů

Příklady pro konverzi bezkontextové gramatiky na zásobníkový automat jsou dostupné v menu pod **Examples -> Context-free grammar to PushDown Automata -> <Název příkladu>**. Po najetí myši na libovolný příklad se zobrazí dole pod položkou menu obrázek vstupních dat pro snadnější orientaci uživatele. Toto ukázání obrázku nad položkou v menu není standardní vlastností Qt frameworku a musel jsem ji ručně naprogramovat. Více se dozvíte dále v části nazvané implementace. Celou situaci můžete vidět na obrázku (Obr. 8.1).



Obr. 8.1 Snímek obrazovky s výběrem příkladu bezkontextové gramatiky na zásobníkový automat

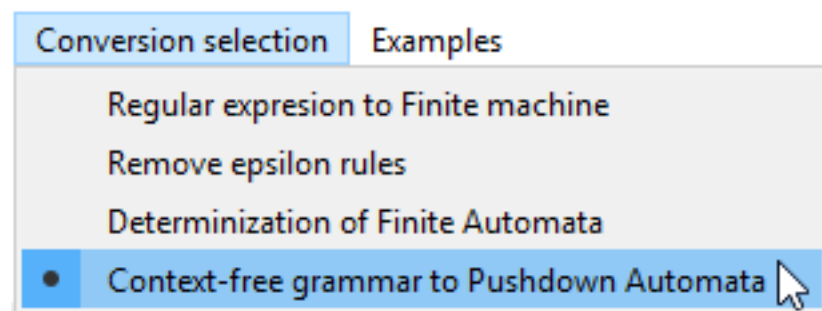
8.2 Výběr z konverzí

Třetí a poslední možností, jak začít používat program je volba konverze. Uživatel má na výběr celkem z 4 konverzí, z čehož poslední z nich a to převod bezkontextové gramatiky na zásobníkový automat byla implementována v rámci této práce. Pokud uživatel využije této volby musí počítat s tím, že vstupní data budou prázdná.

Všechny implementované konverze zobrazuje obrázek (Obr. 8.2).

8.3 Ukládání a načítání konverzí

Tyto volby pro načítání z, respektive ukládání do souboru se nachází v menu pod "File -> Save", respektive pod "File -> Load". Taktéž jsou v aplikaci implementované zkratky **ctrl + o** respektive **ctrl + s** pro otevření, respektive uložení aktuální



Obr. 8.2 Výšeč snímku z obrazovky pro výběr konverzí

konverze do souboru. Tuto funkcionalitu jsem přidal zejména z důvodu toho, že zabudované příklady v aplikaci nemohou obsáhnout vše a proto má vyučující možnost připravit další zajímavé příklady pro studenty.

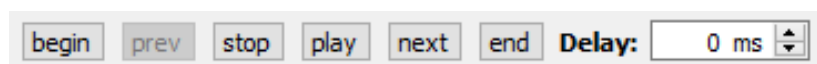
Screenshot není potřeba přikládat, avšak za zmínku stojí fakt, že u módů "Samostatná práce" a "Automatická kontrola" se ukládají data, jak vstupní, tak výstupní, avšak u módu "Krokování algoritmu" se ukládají jenom vstupní data, protože výstup zde netvoří uživatel, nýbrž samotná aplikace.

9 Konverzní módy

Poslední položkou v menu je výběr módu. Jak již bylo řečeno na začátku této kapitoly, aplikace by měla umožňovat, jak samotné studium konverzních algoritmů, tak jejich samostatné praktické procvičení, proto jsem implementoval následující módy.

9.1 Mód krokování algoritmu

První mód určený pro prezentace v hodinách na fakultě a pro samostudium implementovaných konverzí je mód nazvaný *Krokování algoritmu*. V tomto módu si uživatel krokuje jednotlivé kroky algoritmu. Ve widgetu pro danou konverzi jsou v jeho horní části následující kontrolní prvky, které můžete vidět na obrázku (Obr. 9.1). Ještě než se přesuneme k popisu jednotlivých tlačítek stojí za zmínku, že pokud je algoritmus na úplném začátku, tak je zakázáno tlačítko zpět, stejná situace nastává na konci algoritmu pro tlačítko vpřed. První případ je ostatně vidět na obrázku (Obr. 9.1).



Obr. 9.1 Výřez kontrolních prvků z krokovacího módu

Jak je vidět, k dispozici jsou možnosti **next**, respektive **prev**, které nás posunují o jeden krok v algoritmu vpřed, respektive v zad. Dalšími jsou **begin**, respektive **end**, které za nás okamžitě přenesou v algoritmu na začátek, respektive na konec. Poslední dvě možnosti jsou **play** a **stop**. Předposlední tlačítko je nazváno **play** začne přehrávat algoritmus po jednotlivých krocích s časovou prodlevou specifikovanou v pravé části pomocí kontrolky pojmenované **Delay**, ve které se zadává čas v milisekundách. Výchozí volba je 0 ms, která říká programu, že má provádět jednotlivé kroky algoritmu jak nejrychleji dovede. Poslední tlačítko je nazváno **stop**, které nám přestane přehrávat algoritmus, pokud jsme předtím klikli na **play**. Více o algoritmickém widgetu a souvislosti breakpointů a tlačítka **play** si povíme dále.

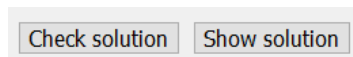
9.2 Mód samostatné práce

Poté co si uživatel prostudoval v praxi, jak funguje algoritmus převodu, si může zkusit konverzi provést sám, tak aby se přesvědčil, že algoritmus plně ovládá.

Poté co ve výstupním widgetu uživatel vytvořil svoje řešení, může si manuálně zkusit, jestli správně převedl daný model formálního jazyka pomocí tlačítka **Check solution**.

Pokud uživatel zjistil, že nevytvořil správné řešení a už neví jak dále, může si nechat správné řešení zobrazit pomocí tlačítka **Show solution**. Po jeho stisku se ve výstupním widgetu zobrazí vzorové řešení. Zároveň s tím se změní text tohoto tlačítka na **Back**, po

jehož stisku se do výstupního widgetu vloží zpět uživatelské řešení, přesně tak jak jej zanechal před stiskem **Check solution**. Zobrazení těchto dvou respektive tří (s právě skrytým tlačítkem zpět) kontrolních prvků můžete vidět na obrázku (Obr. 9.2).



Obr. 9.2 Výřez
kontrolních prvků z módu
samostatné práce

9.3 Mód automatické kontroly

Pokud už si uživatel myslí, že "už už" přece řešení musí být správné, nebo ho jen nebaví manuálně klikat, jestli má už správné řešení, může využít poslední mód pojmenovaný v programu **Instant checking**, taktéž volně přeložený do češtiny *Mód automatické kontroly*. V tomto módu program sám ve velmi krátké periodě kontroluje, jestli je uživatelský vstup ekvivalentní se správným řešením. Více o tom, co je to ekvivalentní řešení, si povíme dále v části popisující převod bezkontextové gramatiky na zásobníkový automat. Stejně jako v předchozím módu se správnost řešení objevuje v levém horním rohu výstupního widgetu.

10 Popis konverze bezkontextové gramatiky na zásobníkový automat

Stejně jako u předchozích konverzí, bylo i u této konverze dodrženo zavedeného schématu z bakalářské práce, že vlevo nahoře je widget, ve kterém je popsán vstupní model formálního jazyka, v pravé horní části je algoritmický widget, v levé dolní části je widget, ve kterém je umístěn výstupní model formálního jazyka a konečně v pravé dolní části je widget, ve kterém jsou zobrazené aktuální hodnoty proměnných použitých v algoritmickém widgetu. Celý pohled je ukázán na obrázku (Obr. 10.1).

Všechny widgety jsou od sebe navzájem odděleny rozdělovači, které umožňují měnit jejich velikost a to až do té míry, že je možné je úplně schovat. To se může hodit v situaci, když si chce student procvičit danou konverzi do té míry, že nechce být rušen samotným algoritmem. Další ze situací je, když studuje, nebo samostatně provádí konverzi a nezajímají jej aktuální hodnoty proměnných, nebo chce jen více místa pro výstupní widget, pokud převádí rozsáhlý model, tak jistě ocení možnost, že si může nechat schovat widget s proměnnými. Oddělovače jsem globálně pro celou aplikaci zvýraznil pomocí kaskádových stylů, aby byly tučné a měly tmavě šedou barvu, protože výchozí styl ve Microsoft Windows byl dosti nevýrazný.

V této konverzi vstupní widget popisuje bezkontextovou gramatiku a ve výstupním se nachází zásobníkový automat.

Při kontrole, jestli je uživatelský zásobníkový automat ekvivalentní se správným řešením, nezáleží na tom jak si uživatel pojmenoval jednotlivé stavy a je jedno v jakém pořadí vloží přechodová pravidla.

10.1 Vstupní widget bezkontextové gramatiky

Do tohoto widgetu se vkládá popis bezkontextové gramatiky pomocí Backus-Naurovi formy popsané v teoretické části této práce. Dále je inteligentní a pozná, jestli je daná gramatika ve správném syntaktickém formátu. Pokud není, tak pozadí editačního kontrolního prvku, kde se daná gramatika vyplňuje zežlutne, čímž uživateli signalizuje, že udělal chybu v syntaxi. V opačném případě se nastaví pozadí na bílou barvu, gramatika automaticky rozparsuje a vyplní se tak počáteční non-terminál, abeceda terminálů a non-terminálů. Uživatel záměrně nemá možnost tyto 3 informace editovat, tak aby se předešlo zbytečným chybám. Za zmínku stojí, že pokud mají jednotlivá gramatická pravidla stejnou levou stranu, lze je zapsat na jeden řádek tak, že pravé strany jsou od sebe odděleny svíslítkem. Pokud uživatel vybere čistou konverzi s prázdnou bezkontextovou gramatikou, widget je předvyplněn textem, který uživateli názorně ukáže příklad správné formy, jak zapsat bezkontextovou gramatiku. Je zde uveden proto, aby bylo pro nové uživatele co nejsnadnější používat tento program. Jeho text je uveden

na obrázku (Obr. 10.2).

Jelikož součástí gramatiky může být i epsilon pravidlo a symbol ϵ není běžně na klávesnici, proto se na samém spodku widgetu nachází tlačítko, které jej vloží na aktuální pozici kurzoru.

10.2 Algoritmický widget

Tento widget na samotném vrcholku obsahuje tlačítka a kontrolní prvky, které se mění v návaznosti na vybraný mód. Jelikož jsou popsány výše, je zde nebudeme blíže rozebírat.

Pod nimi se nachází samotný textový popis algoritmu, kde v jeho vrchní části je zapsáno, co je jeho vstupem a výstupem. Pod tímto popisem jsou už samotné kroky algoritmu.

Výpočetní model každého algoritmu je postaven tak, že kliknutím na tlačítko **next** se následující krok provede, jeho pozadí zežlutne, aktualizuje výstupní widget a nastaví aktuální proměnné do widgetu vpravo dole.

Vlevo od exekutivních kroků je šedé kolečko, které se změní na červené, pokud uživatel kliknul na příslušný řádek. Toto kolečko symbolizuje breakpoint a jeho barva značí, jestli je neaktivní, či aktivní. Breakpointy slouží v režimu krokování algoritmu, kde lze kliknout na tlačítko **play** a tím se spustí automatické krokování algoritmu s danou prodlevou mezi kroky do doby, než algoritmus narazí na aktivní breakpoint, nebo dokud nebylo stisknuto tlačítko **stop**. Avšak ne každý krok umožňuje nastavit breakpoint, například pro krok algoritmu, kde je uvedeno jen informativní návěští pro snadnější orientaci uživatele. Grafickou podobu algoritmu můžete vidět na obrázku (Obr. 10.1).

Nyní bych rád přiložil ještě jeden obrázek (Obr. 10.3) s diagramem, který ukazuje, jak celý algoritmus funguje. Uzly s kolečkem značí jednotlivé kroky algoritmu a šipky s textem napravo znázorňují podmínky, za kterých se přechází z jednoho kroku do druhého. Pokud má šipka u sebe text **always**, tak se následující krok provede vždy. Diagram je dosti názorný, proto jej zde nebudu detailněji popisovat.

10.3 Výstupní widget zásobníkového automatu

Zásobníkový automat, který můžete vidět na obrázku (Obr. 10.1) je zde tvořen žlutými kolečky, uvnitř kterých je pojmenování daného stavu. Pokud je jméno stavu delší, než průměr kolečka, tak se mění tvar na elipsu širokou právě tak, aby obsáhla vnitřní text. Počáteční stav je označen tak, že do něj vede tlustá šipka zleva, která nezačíná v žádném ze stavů zásobníkového automatu. Koncové stavy zásobníkového automatu jsou označeny druhým vnitřním kolečkem, nebo elipsou.

Přechodová pravidla jsou symbolizovaná šipkou s textem, která lze vést z jakéhokoliv stavu do dalšího a to i z, a do toho stejného. Pokud přechodové pravidlo začíná a končí v tom samém stavu, tak je pak označeno částečně zakrytým kruhem nad stavem, nad kterým je text přechodového pravidla.

Na obrázku si taktéž můžete všimnout, je vlevo nahoře tohoto widgetu se nachází zelený text **CORRECT**, ten značí že zásobníkový automat je ekvivalentní se správným řešením. Pokud by nebyl, tak by tam byl červený text **WRONG**.

Pro ilustraci zde uvádím příklad zásobníkového automatu $M = (Q, \Sigma, \Gamma, R, q_0, S, F)$ kde:

- $Q \in \{s, f1, f2\}$,
- $\Sigma \in \{a, b, c\}$ je konečná vstupní abeceda,
- $\Gamma \in \{P, R, S\}$ je konečná abeceda zásobníku,
- $R \in \{\{R, s, c \rightarrow s, R\}, \{P, s, a \rightarrow f1, R, S\}, \{S, s, b \rightarrow f2, R, S\}, \}$ je množina, přechodových pravidel zapsaných v šipkové notaci,
- s je počáteční stav z množiny Q ,
- S je počáteční symbol na vrcholu zásobníku z množiny Γ ,
- $F \in \{f1, f2\}$ je množina koncových stavů, kde $F \subseteq Q$.

viz obrázek (Obr. 10.4).

Pro tvorbu zásobníkového automatu slouží kreslicí plocha, uvnitř které se v pravém horním rohu nacházejí 4 tlačítka. Pokud je vybráno první tlačítko s ikonou kurzoru myši, pak lze jednotlivé stavy zásobníkového automatu přemisťovat v kreslicí ploše. Taktéž lze označovat, jak přechodová pravidla, tak stavy. Pokud máme označen jeden nebo více stavů, nebo přechodových pravidel, lze je smazat jediným klikem na nejpravější tlačítko s ikonou odpadkového koše.

Po kliknutí na tlačítko s ikonou kruhu, uvnitř kterého je obrázek plus, se nastaví mód přidávání stavů, ve kterém každým klikem do kreslicí scény se na daném místě objeví nový uzel s unikátním jménem. Standardně jsou takto nové stavy označeny vzrůstající sekvencí čísel začínající nulou.

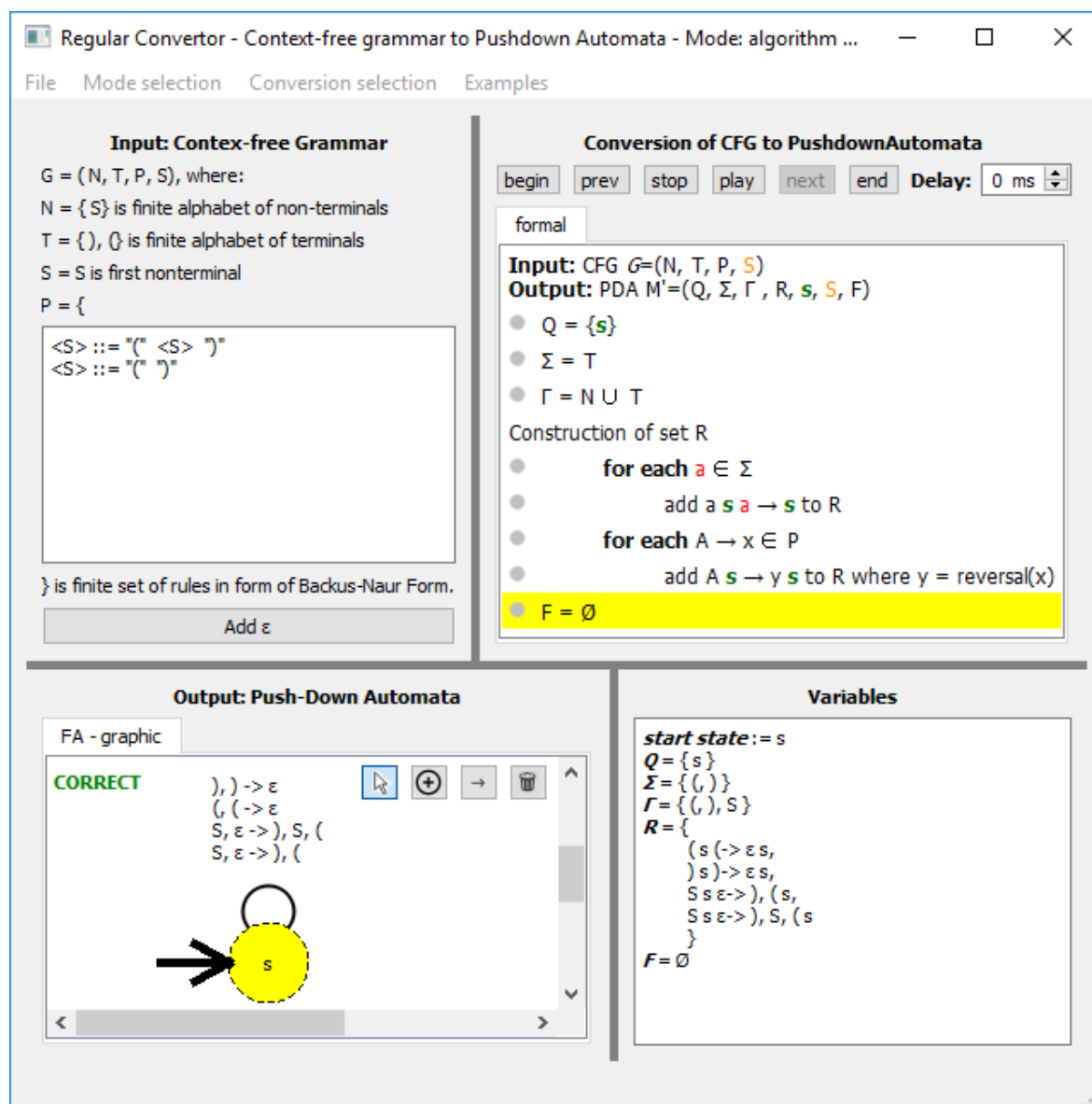
Pokud označíme tlačítko s ikonou šipky, pak můžeme kreslit ve scéně přechodová pravidla. Ty se vkládají tak, že se stiskne levé tlačítko myši uvnitř plochy elipsy označující stav automatu, drží se stisklé a uvolní se v jiném nebo stejném stavu. Po uvolnění levého tlačítka myši vyskočí dialog, ve kterém se zadávají přechodová pravidla. Tento dialog můžeme vidět na obrázku (Obr. 10.5).

Pokud vkládáme první pravidlo mezi stavy automatu, tak je dialog vyplněn šedým textem informujícím v jakém formátu se jednotlivé pravidla zapisují. Formát je takový, že každé pravidlo začíná symbolem, který se vybírá ze zásobníku, následuje vstupní symbol abecedy oddělený čárkou, následuje šipka ("->") a za ní je seznam symbolů, které se vkládají na zásobník. Záměrně se zde nezapisuje z jakého na jaký stav se má zásobníkový automat přesunout po provedení pravidla, tak aby uživatel nemohl udělat chybu. Tuto informaci zvolil už tím, že vytvořil šipku, která vždy musí v nějakých stavech začínat a končit.

Daný uzel lze označit jako koncový, či počáteční kliknutím pravým tlačítkem uvnitř elipsy a zaškrtnutím příslušného checkboxu. Tuto kontextovou nabídku, kde je vybrán daný uzel jako počáteční a zároveň jako koncový můžete vidět na obrázku (Obr. 10.6).

10.4 Widget s aktuálními proměnnými z algoritmického widgetu

V tomto widgetu se zobrazují proměnné použité v algoritmickém widgetu spolu s jejich aktuálními hodnotami. Tyto proměnné se zobrazují pouze tehdy, pokud je vybrán mód krokování algoritmu. Vzhledem k tomu, že jsem tento widget implementoval pomocí ovládacích prvků, které umožňují přijímat vstup ve formátu html, tak jsem mohl použít základní formátování textu, které jsem využil k zvýrazňování a obarvování proměnných tak, aby měly stejnou vizuální podobu s proměnnými použitými v algoritmickém widgetu a to vše proto, abych usnadnil orientaci uživateli v dané konverzi.



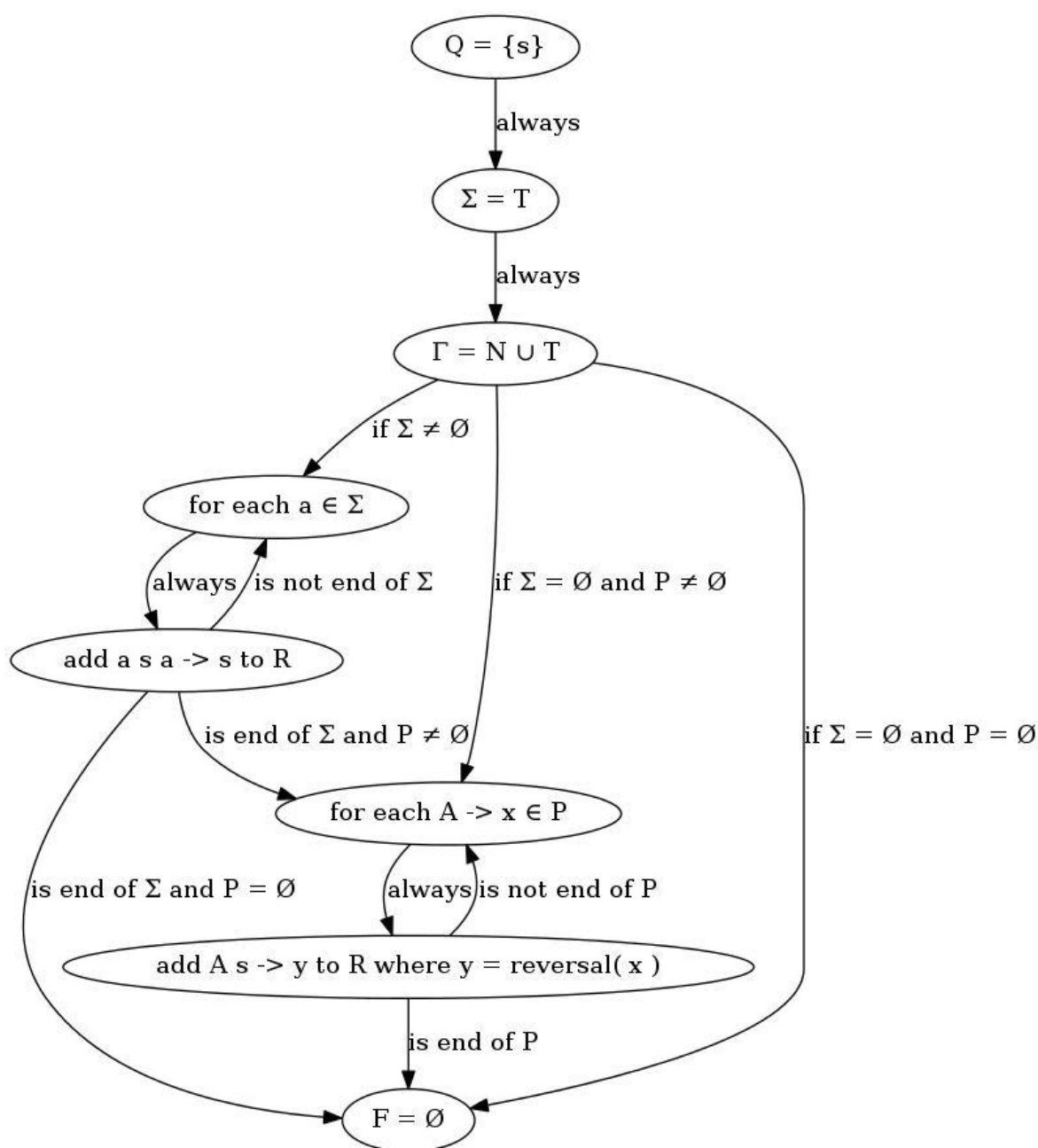
Obr. 10.1 Pohled na hlavní okno, ke je vybrán první příklad převodu bezkontextové gramatiky na zásobníkový automat na konci krokovacího módu

```

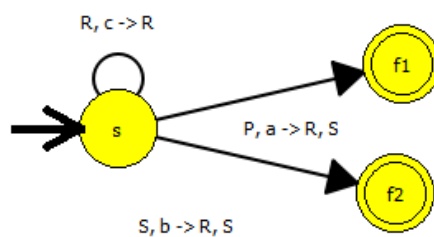
Fill here Context-Free grammar in Backus-Naur Form.
E.g.
<StartNonTerminal> ::= <Nonterminal1> "terminal" | <Nonterminal2>
<Nonterminal1> ::= "terminal2" <Nonterminal2>
<Nonterminal2> ::= <StartNonTerminal>

```

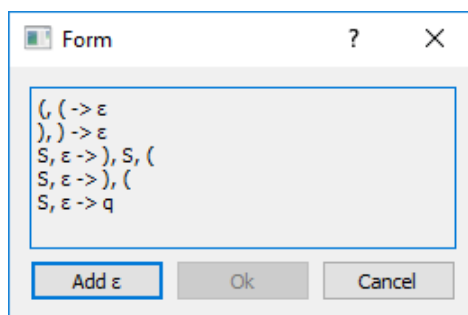
Obr. 10.2 Předvyplněný text bezkontextové gramatiky



Obr. 10.3 Diagram detailně popisující chování algoritmu převodu bezkontextové gramatiky na zásobníkový automat



Obr. 10.4 Příklad zásobníkového automatu



The dialog box 'Form' contains a text area with the following text:

```
(, (-> ε  
) , )-> ε  
S, ε-> ), S, (  
S, ε-> ), (  
S, ε-> q
```

Below the text area are three buttons: 'Add ε', 'Ok', and 'Cancel'.

Obr. 10.5 Okno dialogu pro
vkládání přechodová pravidla
v zásobníkovém automatu



Obr. 10.6 Kontextová
nabídka pro nastavování
počátečního a koncového
stavu automatu

11 Implementace

V této části práce se detailněji podíváme na to, jak samotný program vznikal, z jakých částí se skládá, jak jej sestavit ze zdrojových souborů a na výběr některých zajímavých pasáží z kódu.

Aplikaci jsem programoval v jazyce C++ verze normy 11. A nejen pro tvorbu grafického uživatelského prostředí jsem zvolil multiplatformní framework Qt 5.8.

Prvně si ukážeme, jak získat zdrojové soubory a jak je můžeme zkompileovat. Následně si probereme, refaktoring existujícího kódu, který jsem psal v rámci mé bakalářské práce. A poté si probereme nejdůležitější stavební prvky programu.

V příloze najdete malou analýzu existující řešení dokumentace kódu.

11.1 Stažení zdrojových souborů a jejich následná kompilace

Všechny zdrojové soubory jsou opensource a jsou veřejně k dispozici na serveru Github včetně této práce ve formátech pdf a L^AT_EX. Celou práci jsem verzoval pomocí programu Git a proto si vše můžete naklonovat pomocí příkazu:

```
git clone https://github.com/navrkald/regularConvertor.git.
```

Github umožňuje mimo jiné zobrazovat různé statistiky ohledně množství a rozložení commitů v čase. Github měl pro mě taktéž výhodu, že práce byla uložena na veřejně přístupném serveru a já tak mohl vyvíjet střídavě z různých míst a zařízení. Dále pak jsem mohl mít jistotu, že Github má vše zálohované a i kdyby ne, tak samotný Git má tu úžasnou vlastnost, že každým naklonováním mého repositáře vzniká kompletní off-line záloha a vzhledem k faktu, že jsem pracoval z více počítačů, tak mám hned několik záloh.

Opensource řešení jsem zvolil jednak z důvodů toho, že jsem chtěl, aby moje zdrojové soubory mohl vidět celý svět, aby je kdokoli mohl upravit a pomocí *pull requestu*, které mám ve svém repositáři povolené a dále rozvíjet tuto skvělou aplikaci. Druhým a to právním důvodem bylo, že jelikož nepoužívám multiplatformní Qt framework v komerční licenci, za kterou bych musel platit, jsem povinen všechny své zdrojové soubory zveřejnit.

V předchozích odstavcích jsme se dozvěděli, jak získat zdrojové soubory, nyní si už jen zbývá ukázat, jak zkompileovat zdrojové soubory do spustitelného souboru.

Základní dvě možnosti jsou, použít nativní integrované vývojové prostředí QtCreator a nebo použít Qt plugin do Visual studia. Obě tyto možnosti začínají tím, že si musíme stáhnout samotný Qt framework pro danou platformu, na které zrovna vyvíjíte. Zde je odkaz na všechny možné instalátory Qt opensource frameworku, pro všechny hlavní platformy: <https://www.qt.io/download-open-source/#section-2>. Pro Windows jsou k dispozici mimo jiné verze pro kompilátor Visual studia a MinGW. Já jsem

použil kompilátor Qt 5.8.0 for Windows 64-bit (VS 2015, 1.0 GB). Součástí balíčku je i integrované vývojové prostředí QtCreator spolu se zdrojovými kódy a binární podobou Qt frameworku.

Zdrojové soubory jsou užitečné ve chvíli, pokud chcete udělat staticky zkompilovanou aplikaci používající Qt framework, tak aby v sobě obsahovala všechny Qt závislosti. Výhodou je, že pak můžete distribuovat vaši aplikaci v takzvaném "portable" formátu, kde aplikace neobsahuje instalátor, nevýhodou je za vyšší velikost výsledného binárního souboru. Jelikož je statická kompilace nad rámec této práce, uvedu pouze odkaz na internet: https://wiki.qt.io/Building_a_static_Qt_for_Windows_using_MinGW a zmíním, že mnou upravený skript pro statickou kompilaci Qt frameworku najdete v mém repositáři v adresáři `src/scriptToBuildStaticQt.ps1`.

Po stažení a nainstalování Qt frameworku včetně Qt Creatoru a zdrojových souborů, které se hodí i v případě nejasností v dokumentaci Qt, můžete již začít vyvíjet, tak že si v něm otevřete projektový soubor, který se nachází v cestě <cesta k naklonovanému repositáři>/src/RegularConvertor.pro. Pak již stačí projekt sestavit pomocí `Build -> Build All` a pak spustit `Build -> Run`. Pro debugging programu pouze odkážu na nastavení debuggeru na url: <http://doc.qt.io/qtcreator/creator-debugger-engines.html>.

Mě osobně po čase přestávalo prostředí QtCreatoru stačit a potřeboval jsem více pokročilé IDE, proto jsem si nainstaloval do Visual Studia plugin QtPackage dostupný online: <https://marketplace.visualstudio.com/items?itemName=havendv.QtPackage>. Plugin v sobě neobsahuje celý Qt framework, proto jej je nutné předtím stáhnout. Po nastavení pluginu pak už jen stačí ve Visual Studiu spustit a zbuildit pomocí klávesové zkratky F5.

Pokud nechcete, aby vám editor ve Visual studiu červeně podtrhával neznámé symboly, což vám ale nebrání v kompilaci, je nutné mu nastavit cesty k hlavičkovým souborům Qt. To se provede tak, že ve vlastnostech projektu v `ConfigurationConfiguration Properties -> C/C++ -> General, Additional Include Directories` přidají tyto cesty:

- `.\GeneratedFiles`
- `$(QTDIR)\include`
- `.\GeneratedFiles\$(ConfigurationName)`
- `$(QTDIR)\include\QtCore`
- `$(QTDIR)\include\QtGui`

Pokud by si někdo nechtěl aplikaci kompilovat ze zdrojových souborů může si ji stáhnout na adrese <https://github.com/navrkald/regularConvertor/releases>.

11.2 Refaktor existujícího kódu

Refaktoring je činnost při, které se zlepšuje kvalita kódu bez přímého vlivu na uživatele aplikace.

Nerefaktorovaný kód vede k tomu, že jen obtížně mohou programátoři rozumět kódu a to vede k častým chybám a prodloužení vývoje.

Všechny zdrojové soubory nutné pro úspěšnou kompilaci kódu jsem přesunul do samostatné složky standardně pojmenované *src*, podle anglického slova *sources*.

Kód by měl být pokud možno homogenní a měl by v něm být dodržován systém pojmenovávání.

11.2.1 Konvence pojmenování částí kódu

Pro pojmenovávání nových proměnných, tříd a typů jsem použil systém používaný ve firmě AVG. Vše popisuje následující tabulka.

Tab. 11.1 Konvence pro pojmenovávání tříd a proměnných

| konvence | poznámka |
|----------------------------|--|
| m_<Název členské proměnné> | Písmeno <i>m</i> podle anglického slova member. |
| C_<Název třídy> | Písmeno <i>C</i> Podle anglického slova Class. |
| I_<Název rozhraní> | Písmeno <i>I</i> podle anglického slova Interface. |
| S_<Název struktury> | Písmeno <i>S</i> podle anglického slova Structure. |

Dalšími konvencemi jsou pojmenovávání pomocí *CammelCase*. To znamená, že pokud se nějaký název skládá z více slov, tak jsou od sebe navzájem oddělena velkými písmeny. Názvy metod začínají velkým písmenem a názvy proměnných malým a pro obojí používám *CamelCase*. V bakalářské práci jsem hodně používal pro víceslovné názvy oddělení podtržítka, ale *CamelCase* se ukázal jak efektivnější varianta.

Při pojmenovávání souborů jsem se snažil dodržet konvenci, že jejich název se skládá pouze z malých písmen následovaný příponou.

Jelikož množství a rozsah zdrojových souborů je značný, pojmenovávací konvence popsané výše jsem použil jen u nově psaných nebo starých upravovaných částí.

11.3 Zobrazování náhledů příkladů

Jak jste mohli vidět na obrázku (Obr. 8.1) u každého příkladu se zobrazuje jeho náhled. Toto není standardní chování Qt frameworku a tohoto efektu jsem docílil tím, že jsem

oddědil od základní třídy `QMenu` vlastní třídou `MyTooltipQMenu`, kde přepsal metodu `event(QEvent *)`, ve které reaguji na rozdílně na událost `QEvent::ToolTip`, pro kterou zobrazuji místo textu obrázek. Na závěr jsem položku `Examples -> Context-Free grammar to pushdown automata` povýšil na `MyTooltipQMenu`.

Tento lehce detailní popis zde uvádím hlavně pro případně zájemce, kteří by se chtěli podílet na rozšíření této práce, protože jak jsem docílil tohoto speciálního chování, není v tomto konkrétním příkladě z kódu na první pohled zřejmé.

11.4 Drobné vylepšení existujícího kódu

Pro uložení existujících konverzí do souboru jsem přidal klávesovou zkratku `CTRL+S` podle anglického *Save* a pro uložení `CTRL+L`, podle anglického *Load*.

11.5 Zapouzdření konverzí

Příliš velké metody nebo třídy nejsou tolik přehledné a udržitelné. Proto jsem se rozhodl přepsat moji třídu hlavního okna do více menších celků.

Původně vytváření grafického rozvržení widgetů jsem měl pro všechny konverze implementované v třídě hlavního okna. Naopak nyní jsou všechny konverze popsány graficky v souborech s koncovkou `*.ui` a logika konverzí v asociovaných zdrojových souborech s koncovkami `*.cpp` a `*.h`. Toto jsem dělal ve jménu jedné z hlavních myšlenek OOP a to zapouzdření.

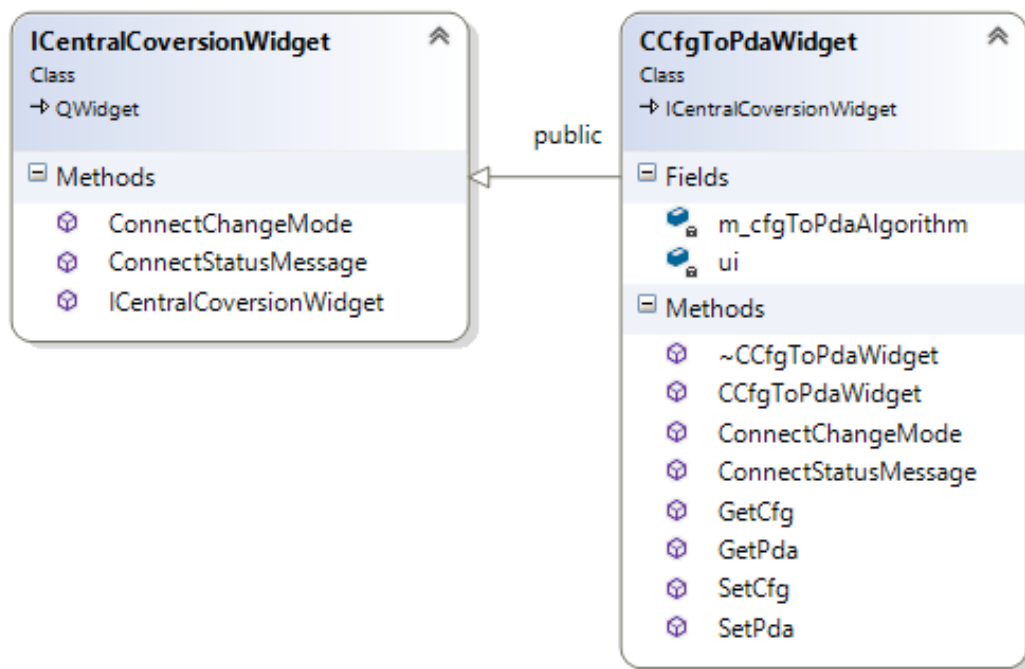
Jelikož všechny, takto přepsané widgety, které popisují jednotlivé konverze, obsahují společnou množinu vlastností, tak všechny dědí od společného rozhraní `ICentralConversionWidget`, které má čistě virtuální metody `ConnectChangeMode()` a `ConnectStatusMessage()`. První metoda má za úkol propojit notifikaci o změně módu z hlavního okna do tříd, které zaštiťují jednotlivé konverze. A druhá naopak propojuje notifikace o různých případných chybových stavech z konverzního widgetu do hlavního okna, kde se pak tyto notifikace zobrazují ve stavové liště. Tento princip, kdy jednotlivé třídy mají společné vlastnosti, které zaštiťuje rozhraní a kde se pak v programu pracuje s ukazatelem na rozhraní se nazývá polymorfismus.

Pro vytváření centrálního widgetu, ve kterém se zobrazují jednotlivé konverze složí ve třídě `MainWindow` metoda `PrepareConversionWidget(Conversions conversion)`, která má jeden vstupní parametr informující o tom, která konverze se má nastavit.

Vzhledem k tomu, že každá z konverzí má jiná vstupní a výstupní data, tak v případech, kdy se ukládají nebo načítají data ze souboru, nebo z příkladu je nutné ukazatel na rozhraní přetypovat na konkrétní instanci třídy.

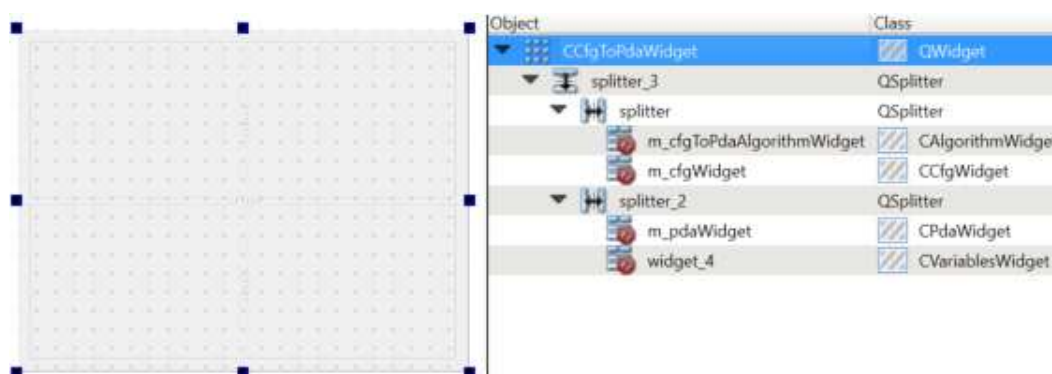
11.6 Třída CCfgToPdaWidget zapouzdřující konverzi CFG na PDA

Nyní si pojdme ukázat na diagramu tříd, z čeho se skládá třída `CCfgToPdaWidget` zaštiťující převod bezkontextové gramatiky na zásobníkový automat. Tento diagram můžeme vidět na obrázku (Obr. 11.1).



Obr. 11.1 Diagram tříd třídy `CCfgToPdaWidget`

Tato třída, chcete-li widget, v sobě obsahuje instanci vnitřní třídy zvanou `ui`, která je generovaná z grafického popisu definovaném v Qt Designeru. Obsahuje jako členské proměnné i subwidgety ze kterých se `CCfgToPdaWidget` skládá.



Obr. 11.2 Příklad promoting na widgetu pro konverzi bezkontextové gramatiky na zásobníkový automat.

Na obrázku (Obr. 11.2) můžete vidět rozložení 4 widgetů do mřížky, navzájem oddělených rozdělovači pro konverzi bezkontextové gramatiky na zásobníkový automat.

Qt Designer bohužel nezobrazuje konkrétní grafickou podobu pro mnou navržené přízpůsobené widgety popsané ve třídách `CAlgorithmWidget`, `CCfgWidget`, `CPdaWidget` a `CVariablesWidget`. Ke každé v těchto tříd ve kterých je definované chování je přiložen i *ui* soubor popisující grafickou podobu.

11.7 Třída `CAlgorithmCFGtoPDA` obsahující logiku konverze CFG na PDA

Na obrázku (Obr. 11.3) můžete vidět diagram třídy `CAlgorithmCFGtoPDA`, který obsahuje veškerou logiku převodu bezkontextové gramatiky na zásobníkový automat. Má přístup k veškerým widgetům nutných pro úspěšné zvládnutí konverze a obsahuje i samotný algoritmus, který je popsán v metodě `ComputeNextStep()`, která se skládá ze 2 částí. V první části se na základě toho jaký byl předchozí krok algoritmu a jaká jsou vstupní a průběžná data rozhodne, jakým dalším krokem se bude pokračovat. A v druhé části se vyková aktuální krok algoritmu.

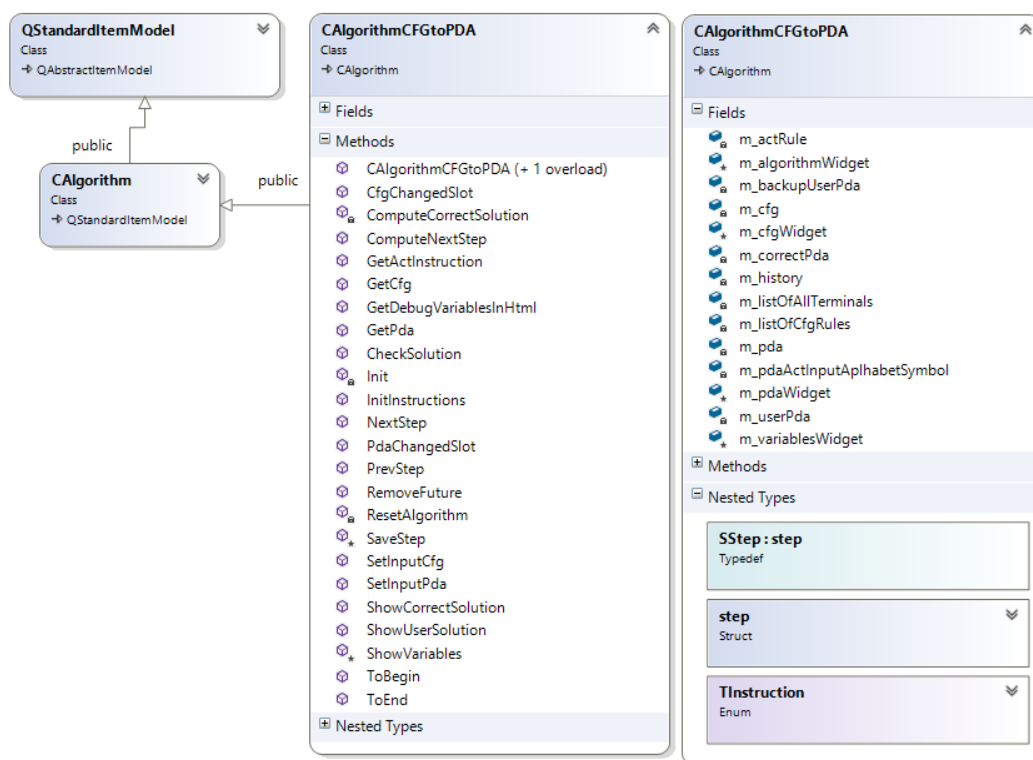
Funkcionalita pod tlačítka `play`, `next` a `end` je implementovaná pomocí volání metody `NextStep()`, která volá metodu `ComputeNextStep()` a dále se stará o to jestli je algoritmus na konci, volá metody pro správné ukládání předchozích kroků, a komunikuje s třídou `CAlgorithmWidget`, která zobrazuje tlačítka a samotný algoritmus převodu. Tuto třídu sdílí všechny implementace konverzí a je proto naprogramovaná obecně.

Aby bylo možné se vracet v krocích algoritmu i nazpět, pomocí tlačítek `begin` a `stop`, je nutné si ukládat informace o předchozích krocích v metodě `NextStep()`. Jeden krok algoritmu je uložen ve struktuře pojmenované `SStep`. A celá historie je uložena v proměnné `QList<SStep> m_history`.

Jelikož všechny algoritmy sdílí některé společné vlastnosti, které jsou definované v třídě `CAlgorithm` a tato třída dědí od vestavěné třídy `QStandardItemModel`, která reprezentuje model pro třídu `AlgorithmView`, která je součástí widgetu `CAlgorithmWidget` a která dědí od třídy `QListView`, což je vestavěný Qt widget. Tímto mechanismem jsem dokonale oddělil samotné uživatelské rozhraní od výpočetní logiky.

Dále již nebudu podrobně vysvětlovat jednotlivé třídy, protože by to bylo nad rámec této práce, jen snad zmíním, že stejný princip oddělení uživatelského rozhraní a logiky programu jsem použil i u widgetů pro zobrazování a editaci zásobníkových automat a bezkontextové gramatiky.

Na konec implementační části vyberu už jenom dva zajímavé momenty a to parsování bezkontextové gramatiky v Backus Naurově formě a parsování přechodových pravidel zásobníkového automatu.



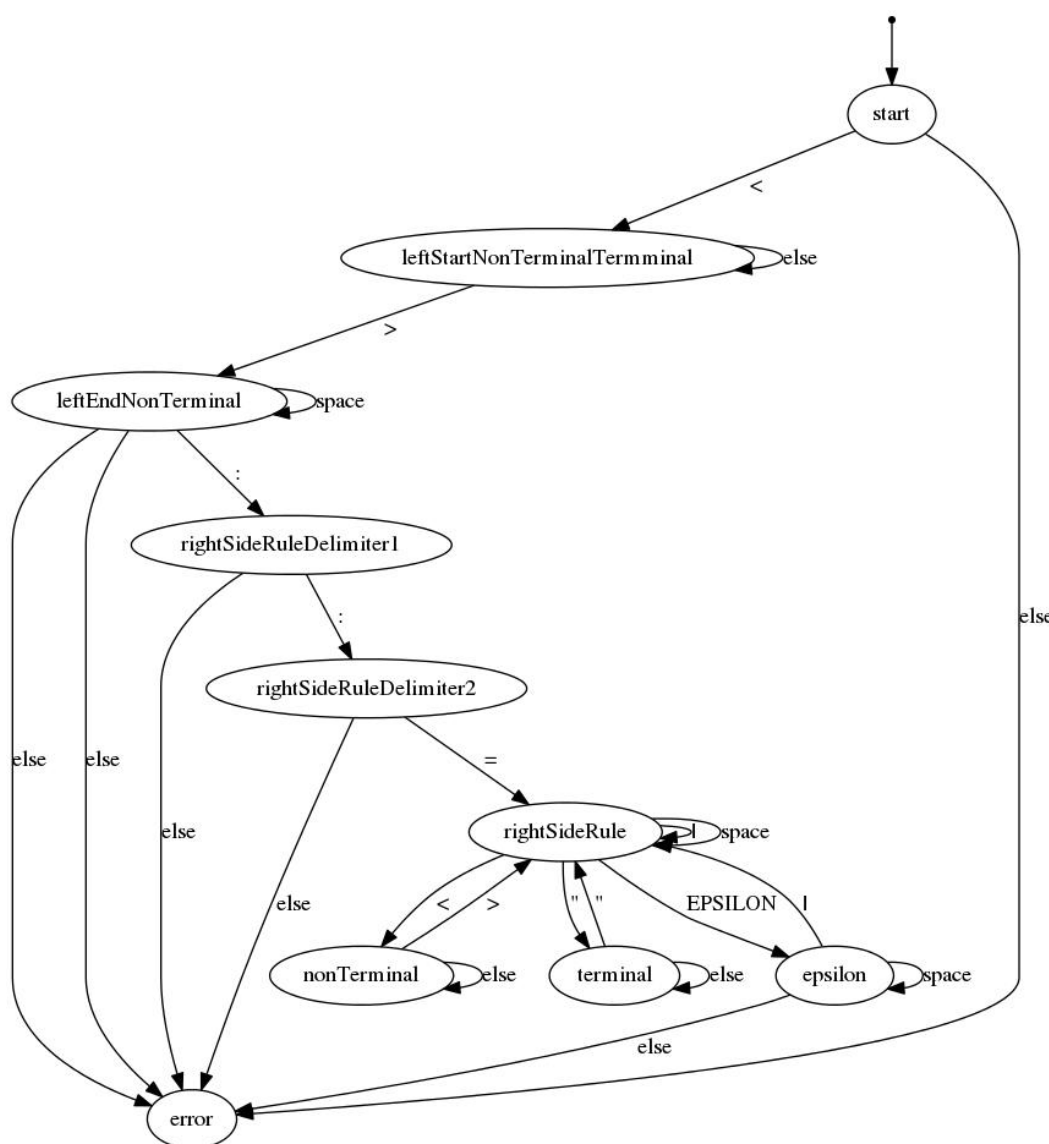
Obr. 11.3 Diagram tříd třídy CAlgorithmCFGToPDA

11.8 Parsování bezkontextové gramatiky v Backus Naur formě

Text gramatiky zpracovávám v metodě `GetFromBackusNaurForm()` třídy `CContextFreeGrammar`, která reprezentuje bezkontextovou gramatiku. Text se rozdělí na jednotlivé řádky, kde se vyfiltrují ty, které jsou prázdné a ty obsahující bílé znaky. Pak se na každý aplikuje statická metoda `GetRulesFromString` z třídy `CCFGRule` reprezentující jedno pravidlo vracející ve výstupním parametru množinu pravidel bezkontextové gramatiky. Následně se zkontroluje, jestli pravá strana pravidla neobsahuje epsilon pravidla, pokud ano epsilon znaky se nepřidávají do množiny terminálů. V opačném případě se všechny znaky řetězce správně rozdělí do množin terminálů a non-terminálů. Sluší se říci, že jedno pravidlo reprezentované třídou `CCFGRule` se skládá z levé strany, což je jeden terminál reprezentovaný třídou `CNonTerminal` a pravá strana je reprezentovaná vektorem terminálů a nonterminálů, které mají oba společného předka `CSymbol`, který má příznak, jestli je terminál nebo nonterminál.

Toto je v kostce popis parsování z výšky, ale ještě jsme si nepověděli jak z jednoho řádku symbolů vznikne množina pravidel. Použil jsem na parsování konstrukci "switch case", která používá stavy mezi kterými se přesouvá a čte řetězec znak po znaku. Diagram tohoto modelu můžete vidět na obrázku (Obr. 11.4). Názvy stavů jsou uvedené v elipsách a přečtený symbol je zobrazený u šipek. Text "else" u šipek znamená, že čtený symbol jakýkoliv jiný, než jaké jsou uvedené u šipek vycházející z daného stavu.

Stav pojmenovaný "error" značí, že gramatika není ve správném syntaktickém formátu.



Obr. 11.4 Diagram parsování pravidel bezkontextové gramatiky v Backus Naurově formě

11.9 Parsování přechodových pravidel konečného automatu

Příklad formy v jaké se zadávají přechodová pravidla jste již mohli vidět na obrázku (Obr. 11.4). Nyní se podíváme na to, jak z textového řetězce dostat strukturovaná data. Jako u předchozího příkladu, zpracovávám stup po řádcích s tím, že odfiltruji prázdné řádky avšak na rozdíl od předchozího jsem zvolil parsování pomocí tohoto rozšířeného regulárního výrazu:

`"^\s*(\S+)\s*,\s*(\S+)\s*->\s*(\S+)\s*((\s*(\S+)\s*))*$"`.

Kde znaky '^' a '\$' značí začátek a konec řetězce, dále pak \s je bílý znak, \S je

nebílý znak a závorky označují tzv. zachytávaný text.

Pojďme si projít jednotlivé části toho rozšířeného regulárního výrazu společně:

- První část `"^\s*(\S+)\s*"` nám zachytí řetězec, který může začínat libovolným množstvím prázdných znaků, následuje samotný neprázdný řetězec, následovaný libovolným množstvím bílých znaků.
- Druhá část `",\s*(\S+)\s*"` začíná čárkou a dále je vzor shodný s první částí.
- Třetí část `"->\s*(\S+)\s*"` začíná šipkou a následuje opět řetězec obklopený libovolným množstvím bílých znaků.
- Poslední a nejzajímavější část `"((,\s*(\S+)\s*))"` nám říká, že se jedná o řetězec začínající čárkou, následuje libovolné množství bílých znaků, pak je neprázdný řetězec nebílých znaků, za kterým je libovolné množství bílých znaků. Třetí část je zajímavá tím, že se může libovolně krát opakovat a to i nulakrát.

Pro získání všech znaků, které se vkládají na zásobník je ještě nutné zachycenou poslední část regulárního výrazu rozdělit podle čárek a očistit od bílých znaků, ale na to již existují knihovní funkce.

ZÁVĚR

Cílem této práce bylo vytvořit aplikaci pro studenty, která bude rozšířena, o konverze modelů bezkontextových jazyků. Dále pak napsat práci, která se bude skládat z teoretické a praktické části.

V první z nich byla vysvětlena Chomského hierarchie, dále pak bezkontextový jazyk, včetně modelů které ho reprezentují, jakými jsou zásobníkový automat (PDA) a bezkontextová gramatika (CFG). U obou je uvedena formální definice, která je použita i v samotné aplikaci. Pro praktické účely byly uvedeny i definice Backus-Naurovy formy a notace zápisu přechodových pravidel zásobníkového automatu. Následně byl definován algoritmus převodu CFG na PDA. Závěr teoretické části se zabývá základy Qt frameworku, který byl použit pro tvorbu aplikace.

V praktické části práce byla detailně popsána aplikace, jak z pohledu uživatelského, tak z vnitřního. Bylo zde popsáno, jak může začít uživatel pracovat s programem, že má na výběr z předdefinovaných příkladů, vložení vlastní CFG, nebo načtení konverze ze souboru. Dále bylo vysvětleno, že program má několik módu pro studium a samostatné procvičení prezentované látky. Dále následoval detailní popis konverzního okna a jednotlivých částí, ze kterých se skládá a jak se ovládají. Těmito částmi jsou okno pro vložení CFG v Backus-Naurově formě, okno ve které se tvoří PDA, interaktivní okno popisující algoritmus a okno ve kterém jsou vidět průběžné hodnoty všech proměnných použitých v algoritmu.

Poslední část praktické části byla věnovaná samotné implementaci. Bylo možné se v ní dočíst, jak získat a zkompileovat aktuální zdrojové kódy ze serveru Github, na kterém je k dispozici i aplikace v binárním formátu pro okamžité použití. Dále byl zmíněn refactoring a konvence použité v kódu. Bylo zde vysvětleno několik základních tříd použitých v aplikaci a také jakým způsobem byl implementovaný samotný algoritmus převodu, jak probíhalo zpracování bezkontextové gramatiky a přechodových pravidel zásobníkového automatu, obojí z textové formy zadané uživatelem.

Co se týče samotné aplikace, tak její zdrojové soubory jsou opensource s důrazem na striktní rozdělení samotné logiky programu a popisu grafického uživatelského rozhraní. Je zde kladen důraz na členění do více drobných částí a užití pravidel OOP, kterými jsou zapouzdření, polymorfismus, kompozice, abstrakce a dědičnost. V programu se dodržuje striktní a jasné pojmenovávání, tak aby byl kód co nejvíce čitelný. Pro implementaci byl použit jazyk C++ a grafický framework Qt, obojí je multiplatformní a není problém aplikaci sestavit jak na Microsoft Windows, tak Linux a dalších. Počet negenerovaných a neprázdných řádků kódu v souborech s příponami *.h a *.cpp se vyšplhal na 10593.

Cíle této aplikace byly, aby měl student k hůře stravitelné teorii přiložené množství

příkladů, tak aby se pro něj stala problematika více pochopitelnou. Dále pak, aby s teorií měl student možnost experimentovat a následně si ji samostatně procvičit. Aplikace také má pomoci vyučujícímu prezentovat teorii v průběhu přednášky na příkladech. Možnost uložení a načtení konverzí se souboru zároveň umožňuje si tvořit a vyměňovat další zajímavé příklady a tak dále obohatit tuto aplikaci.

Cíle této práce, byly splněny.

SEZNAM POUŽITÉ LITERATURY

- [1] NAVRKAL, David. *Konverze modelů regulárních jazyků*. Brno, 2014. Bakalářská práce. FIT VUT v Brně. Vedoucí práce Ing. Zbyněk Křivka, Ph.D.
- [2] KOPKA, Helmut, Patrick W DALY a Jan GREGOR. *Latex: podrobný průvodce*. Vyd. 1. Brno: Computer Press, 2004, 576 s. ISBN 80-7226-973-9.
- [3] *Chomsky hierarchy* [online]. [cit. 2015-11-16]. Dostupný z WWW: https://en.wikipedia.org/wiki/Chomsky_hierarchy.
- [4] *Context-free language* [online]. [cit. 2015-11-17]. Dostupný z WWW: https://en.m.wikipedia.org/wiki/Context-free_language.
- [5] *Pushdown automaton* [online]. [cit. 2015-11-17]. Dostupný z WWW: https://en.wikipedia.org/wiki/Pushdown_automaton.
- [6] *Context-free grammar* [online]. [cit. 2015-11-17]. Dostupný z WWW: https://en.m.wikipedia.org/wiki/Context-free_grammar.
- [7] *Backus-Naur Form* [online]. [cit. 2015-11-18]. Dostupný z WWW: https://en.wikipedia.org/wiki/Backus-Naur_Form.
- [8] HOPCROFT, John E, Rajeev MOTWANI a Jeffrey D ULLMAN. *Introduction to automata theory, languages, and computation*. 2nd ed. Boston: Addison-Wesley, 2001, xiv, 521 s. ISBN 0201441241.
- [9] CHROBOCZEK, Martin. *Grafická uživatelská rozhraní v Qt a C++: [plně kompatibilní s Qt 5]*. Brno: Computer Press, 2013. ISBN 978-80-251-4124-3.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

| | |
|------|--|
| CFG | Context-free grammar, česky také bezkontextová gramatika |
| PDA | Pushdown automaton, česky též zásobníkový automat |
| BNF | Backusova-Naurova forma. (Notace pro zápis bezkontextové gramatiky.) |
| HTML | HyperText Markup Language (Hypertextový značkový jazyk.) |
| IDE | Integrated Development Environment (Integrované vývojové prostředí) |
| OOP | Objektově orientované programování. |

SEZNAM OBRÁZKŮ

| | | |
|-----------|--|----|
| Obr. 1.1 | Chomského hierarchie formálních jazyků | 12 |
| Obr. 3.1 | Příklad zásobníkového automatu | 16 |
| Obr. 7.1 | Úvodní obrazovka programu Regular Convertor | 25 |
| Obr. 7.2 | Výřez záhlaví hlavního okna | 25 |
| Obr. 8.1 | Snímek obrazovky s výběrem příkladu bezkontextové gramatiky na zásobníkový automat | 26 |
| Obr. 8.2 | Výšeč snímku z obrazovky pro výběr konverzí | 27 |
| Obr. 9.1 | Výřez kontrolních prvků z krokovacího módu | 28 |
| Obr. 9.2 | Výřez kontrolních prvků z módu samostatné práce | 29 |
| Obr. 10.1 | Pohled na hlavní okno, ke je vybrán první příklad převodu bezkontextové gramatiky na zásobníkový automat na konci krokovacího módu | 34 |
| Obr. 10.2 | Předvyplněný text bezkontextové gramatiky | 34 |
| Obr. 10.3 | Diagram detailně popisující chování algoritmu převodu bezkontextové gramatiky na zásobníkový automat | 35 |
| Obr. 10.4 | Příklad zásobníkového automatu | 35 |
| Obr. 10.5 | Okno dialogu pro vkládání přechodová pravidla v zásobníkovém automatu | 36 |
| Obr. 10.6 | Kontextová nabídka pro nastavování počátečního a koncového stavu automatu | 36 |
| Obr. 11.1 | Diagram tříd třídy CCfgToPdaWidget | 41 |
| Obr. 11.2 | Příklad promoting na widgetu pro konverzi bezkontextové gramatiky na zásobníkový automat. | 41 |
| Obr. 11.3 | Diagram tříd třídy CAlgorithmCFGToPDA | 43 |
| Obr. 11.4 | Diagram parsování pravidel bezkontextové gramatiky v Backus Naurově formě | 44 |

SEZNAM TABULEK

| | | |
|-----------|--|----|
| Tab. 11.1 | Konvence pro pojmenovávání tříd a proměnných | 39 |
|-----------|--|----|

SEZNAM PŘÍLOH

P I. Analýza nástrojů vytváření vizualizací z existujícího kódu

PŘÍLOHA P I. ANALÝZA NÁSTROJŮ VYTVÁŘENÍ VIZUALIZACÍ Z EXISTUJÍCÍHO KÓDU

Při hledání nástrojů pro vizualizaci a dokumentaci existujícího kódu jsem měl následující požadavky:

- Všechny nástroje musejí být bezplatné.
- Musejí umět vizualizovat diagram tříd a propojení tříd pomocí Qt mechanismu Signálů a slotů.
- Musí umět ideálně vygenerovat požadované vizualizace automaticky, tak aby se při změně kódu sami aktualizovaly a nemusely se vytvářet pokaždé ručně.

1.1 Qt ModelEditor

Prvním nástrojem je *ModelEditor*. Tento nástroj jsem zkusil jako první, protože je ve formě pluginu do IDE *QtCreator*, který jsem ze začátku používal pro programování této diplomové práce. *ModelEditor* umožňuje dělat sice jednoduché, ale přehledné UML diagramy. Zásadní nedostatek, je že umožňuje například diagram tříd generovat jenom částečně a to tak, že se přetáhne zdrojový soubor do modelovacího editoru, kde se vytvoří element s názvem třídy, ale metody a členské se musí dopisovat ručně. Na domovské stránce (<https://wiki.qt.io/ModelEditor>) a ani na druhé <http://doc.qt.io/qtcreator/creator-modeling.html> tohoto nástroje jsem se nikde nedočel, že by se na této funkcionalitě mělo někdy pracovat, proto jsem tento nástroj prozatím zavrhl.

1.2 Doxygen a Graphviz

Druhým nástrojem je *Doxygen*, který je určen na automatickou tvorbu dokumentace a byl ve svém počátku určen právě pro C++/Qt. *Doxygen* pro tvorbu pokročilých grafů potřebuje doinstalovat *Graphviz* a nastavit systémovou proměnnou PATH na cestu ke spustitelným souborům *Graphviz*. Funguje to tak, že *Doxygen* vytvoří textový popis grafu a *Graphviz* pomocí spustitelného programu *Dot* umí vytvořit grafickou podobu jak ve vektorových, tak v rastrových formátech. Aby bylo generování co nejjednodušší vytvořil jsem si konfigurační soubor pojmenovaný "Doxyfile" v adresáři "src". Následně se celá dokumentace vygeneruje automaticky pomocí zavolání příkazu *doxygen* ve stejném adresáři ve kterém se nachází "Doxyfile". *Doxygen* umí vytvořit UML diagram tříd. U každé třídy ukáže od kterých tříd dědí a také názvy proměnných a metod spolu s jejich modifikátorem viditelnosti (public, protected, private). Zásadním problémem je, že neukazuje názvy typů členských proměnných, ani návratové typy a typy parametrů metod. Pro automatické generování dokumentace z kódu je tento nástroj ideální, ale

já chtěl v této diplomové práci mít kompletní diagram tříd včetně typů a návratových hodnot, proto jsem zkusil ještě třetí nástroj.

1.3 Diagram tříd ve Visual studio Express 2015

Třetím nástrojem, který uspokojil všechny požadavky na diagram tříd se nacházel v IDE *Visual Studio*, ovšem s jednou malou komplikací a to, že si neporadil plně s parsováním tříd, které mají definované makro `Q_OBJECT` a pro správnou funkčnost bylo nutné toto makro dočasně zakomentovat.

Diagram konkrétní třídy se dá vygenerovat tak, že se pravým tlačítkem klikne na hlavičkový soubor třídy a následně vybere možnost "zobrazit diagram třídy". Po kliku kamkoliv do volného místa se dá vyvolat nabídka na export do různých formátů. Dá se i zvolit jestli chceme zobrazit celou signaturu, tj. návratové typy, jména a typy parametrů a typy členských proměnných. Jediná drobná výtkka by mohla být, že modifikátor viditelnosti je a zda se jedná o metodu nebo proměnnou je značen z hlediska UML nestandardní ikonou. Zásadní je, že diagram tříd vygenerovaný tímto nástrojem obsahuje všechnu podstatné informace, proto jej použiji pro generování obrázků tříd do této diplomové práce.

1.4 Simple tool to visualize connections between signals and slots in Qt

Pro vizualizaci propojení pomocí Qt slotů a signálů jsem akorát našel jednoduchý program, jehož zdrojový text je uveden na adrese <http://hackatool.blogspot.cz/2013/05/simple-tool-to-visualize-connections.html>. Tento nástroj pracuje na principu prohledání zdrojových souborů na základě regulárního výrazu a následně vygeneruje textový popis grafu propojení ve formátu, kterému rozumí program Graphviz zmíněný výše.