VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SYNTAKTICKÁ ANALÝZA ZALOŽENÁ NA REGULOVANÝCH GRAMATIKÁCH

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE LUBOŠ VANÍČEK
AUTHOR

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SYNTAKTICKÁ ANALÝZA ZALOŽENÁ NA REGULOVANÝCH GRAMATIKÁCH

PARSING BASED ON REGULATED GRAMMARS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE LUBOŠ VANÍČEK
AUTHOR

VEDOUCÍ PRÁCE Prof. RNDr. ALEXANDER MEDUNA, CSc.
SUPERVISOR

BRNO 2015

Abstrakt

Tato práce se zabývá regulovanými gramatikami. Popisuje vlastnosti různých druhů regulovaných gramatik a srovnává je. Ke srovnání využívá příklady postavené na kontextových jazycích. Práce se nejvíce zaměřuje na programované gramatiky. Modifikuje metody syntaktické analýzy a využívá je pro programované gramatiky. Práce aplikuje navrženou metodu syntaktické analýzy na kontextové jazyky a zkoumá jejich sílu.

Abstract

This thesis deals with regulated grammars. It describes their behavior and it compares them using different examples of context-sensitive languages. The thesis focuses mostly on programmed grammars. It modifies the methods of parsing and it uses them for programmed grammars. This thesis applies the developed method of parsing on context-sensitive languages and examines their power.

Klíčová slova

regulované gramatiky, programované gramatiky, syntaktická analýza

Keywords

regulated grammars, programmed grammars, parsing

Citace

Luboš Vaníček: Syntaktická analýza založená na regulovaných gramatikách, bakalářská práce, Brno, FIT VUT v Brně, 2015

Syntaktická analýza založená na regulovaných gramatikách

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením profesora Alexandra Meduny. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Luboš Vaníček
10. května 2015

Poděkování

Rád bych poděkoval svému vedoucímu práce, profesoru Alexandru Medunovi, za jeho odborné vedení, doporučení materiálů a pomoc při řešení problémů.

© Luboš Vaníček, 2015.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Základní pojmy **5**
2.1 Abeceda . . . 5
2.2 Řetězec a prázdný řetězec . . . 5
2.2.1 Délka řetězce . . . 5
2.2.2 Konkatenace řetězců . . . 6
2.2.3 Mocnina řetězce . . . 6
2.3 Množina všech řetězců a jazyk . . . 6
2.4 Bezkontextová gramatika . . . 7
2.4.1 Derivace, přímá derivace, větná forma, věta, terminální derivace . . . 7
2.4.2 Generovaný jazyk . . . 7
2.4.3 Bezkontextový jazyk . . . 7
2.4.4 Označení pravidel . . . 8
2.5 Chomského hierarchie gramatik . . . 8
2.5.1 Gramatiky typu 0 . . . 8
2.5.2 Gramatiky typu 1 . . . 9
2.5.3 Gramatiky typu 2 . . . 9
2.5.4 Gramatiky typu 3 . . . 9
2.5.5 Hierarchie jazyků . . . 9

3 Regulované gramatiky **10**
3.1 Regulárně řízená gramatika . . . 10
3.1.1 Generovaný jazyk . . . 10
3.2 Maticová gramatika . . . 11
3.2.1 Generovaný jazyk . . . 12
3.3 Programovaná gramatika . . . 12
3.3.1 Generovaný jazyk . . . 13
3.4 Gramatika s nahodilým kontextem . . . 13
3.4.1 Generovaný jazyk . . . 14
3.5 Gramatika s rozptýleným kontextem . . . 15
3.5.1 Generovaný jazyk . . . 15
3.6 Stavová gramatika . . . 15
3.6.1 Generovaný jazyk . . . 16
3.7 Generativní síla regulovaných gramatik . . . 17

4 Syntaktická analýza **18**
4.1 Teoretické a praktické jazyky 18
4.2 Derivační strom 18
4.3 Modifikace metod syntaktické analýzy 19

5 Vývoj aplikace **21**
5.1 Specifikace a analýza požadavků 22
5.1.1 Formát vstupní gramatiky 22
5.1.2 Parametry aplikace 22
5.2 Objektový návrh a implementace 23
5.2.1 Třída Program 24
5.2.2 Třída Input 24
5.2.3 Třída InputGrammar 24
5.2.4 Třída Rule 24
5.2.5 Třída InputString 26
5.2.6 Třída ConsoleOutput 26
5.2.7 Třída Automaton 26
5.2.8 Třída GeneratedString 28
5.3 Testování 29
5.4 Praktické využití 30

6 Závěr **33**

A Obsah CD **36**

Kapitola 1

Úvod

Formální jazyky hrají významnou roli nejen v informatice, ale i v mnoha dalších vědních oborech. Využívají se například v lingvistice, molekulární genetice a ve většině odvětvích moderní informatiky [10].

V novodobé informatice je nutností pro práci programátora znalost alespoň jednoho programovacího jazyka. Co se ovšem děje za tímto programovacím jazykem, to už se obvykle neřeší. Na překlad mezi programovacím jazykem a strojovým kódem slouží překladač. Aby běžní programátoři mohli programovat tak, jak jsou zvyklí, je potřeba, aby jim někdo takový překladač vytvořil. Ten se neobejde bez teoretického základu – formálních jazyků.

Překladač se skládá minimálně z těchto čtyř částí: lexikálního analyzátoru, syntaktického analyzátoru, generátoru vnitřního kódu a generátoru cílového kódu. Lexikální analyzátor nejprve rozdělí zdrojový text na posloupnost znaků, které jsou následně zpracovány syntaktickým analyzátorem. Syntaktický analyzátor určí, zda zdrojový text reprezentuje syntakticky správně napsaný program. Pro ověření, zda tomu tak je, se využívají gramatická pravidla. V této fázi by bylo možné překládat program rovnou do cílového jazyka (například jazyka symbolických instrukcí), ale bylo by to příliš složité a ne zcela zřetelné. Pro jednoduchost se využívá mezičlánku – vnitřního kódu (většinou se jedná o tří-adresný kód). Ten je vytvořen pomocí generátoru vnitřního kódu. Vnitřní kód bývá často optimalizován, avšak pro funkční překladač není optimalizace nezbytná. Neoptimalizovaný kód je pouze méně efektivní. Poslední částí překladače je generátor cílového kódu, který z vnitřního kódu vytvoří kód v požadovaném jazyce.

Pro konstrukci překladačů nebo jeho částí je nutná znalost gramatik, k čemuž patří i znalost formálních jazyků. Pro užívání formálních jazyků a gramatik je potřebné znát základní pojmy, které musí být precizně definovány. Důležitý je matematický základ z oblasti množin, relací, funkcí a grafů. Z důvodu dodržení maximálního rozsahu práce je u čtenáře předpokládána znalost tohoto matematického pozadí.

Klasická teorie formálních jazyků rozděluje gramatiky obvykle do dvou skupin: kontextové a bezkontextové. Bezkontextové gramatiky jsou schopny nezávisle na okolním kontextu přepisovat symboly za pomoci pravidel. To znamená, že užití posloupnosti pravidel je možné určit zcela deterministicky a celý tento problém je algoritmicky řešitelný. Proto se bezkontextové gramatiky v praxi hojně využívají. Velkou nevýhodou je, že tyto gramatiky nejsou schopny zpracovávat kontextové jazyky. Kontextové jazyky je možné zpracovat za pomoci kontextových gramatik. Ty mají ovšem ještě větší nevýhodu. Přepisování je závislé na okolním kontextu, což je velmi nešikovné a špatně použitelné v praxi.

Nejlepším krokem k vytvoření gramatik, které budou silnější než bezkontextové a zároveň aplikovatelné v praxi, jsou regulované gramatiky. V podstatě se jedná o bezkontextové

gramatiky, které jsou rozšířeny o různé matematické mechanismy, jenž slouží k ovládání toho, jakým způsobem vybrat správná pravidla během generování jazyka.

Cílem této práce je návrh a implementace syntaktického analyzátoru založeném na regulovaných gramatikách. Regulovaných gramatik je několik různých druhů. Pro implementaci byly zvoleny programované gramatiky.

V kapitole 2 jsou vysvětleny základní pojmy formálních jazyků, které jsou pro tuto práci nezbytné. Jsou zde popsány bezkontextové gramatiky a Chomského hierarchie jazyků.

Kapitola 3 definuje šest různých druhů regulovaných gramatik. U každé gramatiky je uveden matematický aparát, který rozšiřuje bezkontextové gramatiky a přidává jim na síle. Jednotlivé regulované gramatiky jsou srovnávány na základě generování stejného kontextového jazyka. V této kapitole je rovněž popsána generativní síla regulovaných gramatik.

Kapitola 4 popisuje obecné principy syntaktické analýzy. Prezentuje modifikaci syntaktické analýzy pro účely programovaných gramatik a uvádí algoritmus, který je použit při tvorbě vlastní aplikace.

Kapitola 5 se věnuje vývoji syntaktického analyzátoru. Je zde popsán koncept celé aplikace a její testování. Na konci této kapitoly je uveden způsob, jakým je možné použít tuto práci v praxi.

Kapitola 2

Základní pojmy

Základem této práce jsou formální jazyky. Ty musí být velmi podrobně matematicky definovány. Jak již bylo uvedeno v úvodní kapitole, matematický základ potřebný k pochopení základních pojmů formálních jazyků je možné najít například v knize [4].

V této kapitole jsou definovány jednotlivé pojmy formálních jazyků, které budou dále v této práci používány. Pro každý pojem je uvedena formální definice a alespoň jeden příklad pro názornost a hlubší pochopení celkové problematiky. Formální definice jsou převzaty z literatury [8].

2.1 Abeceda

Definice 2.1.1. Abeceda Σ je konečná, neprázdná množina elementů, které nazýváme symboly.

Příklad 2.1.1. $\Sigma = \{a, b, c\}$, kde symbol Σ představuje označení abecedy, jež obsahuje tři symboly – a, b, c.

2.2 Řetězec a prázdný řetězec

Definice 2.2.1. Řetězec, někdy též označován slovo, nad abecedou Σ je konečná sekvence symbolů z abecedy Σ. Pokud řetězec není tvořen žádnými symboly, pak jej nazýváme prázdný řetězec a značíme symbolem ε. Prázdný řetězec ε je také řetězec nad abecedou Σ. Jestliže x je řetězec nad Σ a a je symbol patřící do Σ, pak xa je řetězec nad abecedou Σ.

Příklad 2.2.1. Uvažujme abecedu $\Sigma = \{a, b, c\}$. Řetězců nad touto abecedou je nekonečně mnoho. Patří mezi ně například: ε, a, b, c, aa, ab, ac.

2.2.1 Délka řetězce

Definice 2.2.2. Nechť x je řetězec nad abecedou Σ. Nechť tento řetězec obsahuje znaky $a_1a_2 \ldots a_n$, kde $a_i \in \Sigma$, pro všechny $i = 1, \ldots, n$, kde $n \geq 0$ (v případě, že je $n = 0$, pak x je prázdný řetězec ε). Délka řetězce x, označená jako $|x|$, je n.

Příklad 2.2.2. $|abc| = 3$.
Délka řetězce abc, který se skládá ze tří znaků patřících do abecedy $\Sigma = \{a, b, c\}$, je tři.

2.2.2 Konkatenace řetězců

Definice 2.2.3. Nechť x a y jsou dva řetězce nad abecedou Σ. Konkatenací x a y vznikne řetězec xy.

Příklad 2.2.3. Uvažujme abecedu $\Sigma = \{a, b, c\}$:

- Při konkatenaci řetězců abc a abb vznikne řetězec $abcabb$.
- Při konkatenaci řetězců abc a ε vznikne řetězec abc.

2.2.3 Mocnina řetězce

Definice 2.2.4. Nechť n je přirozené číslo a x je řetězec nad abecedou Σ. Pro $n \geq 0$, n-tá mocnina řetězce x, x^n, je definována:

- $x^0 = \varepsilon$
- $x^n = xx^{n-1}$ pro $n \geq 1$

Příklad 2.2.4. Uvažujme řetězec $x = abc$. Třetí mocninu řetězce x, x^3, určíme podle definice takto:

$$\begin{aligned} x^3 &= xx^2 = abcx^2 = abcxx^1 = abcabcx^1 = abcabcxx^0 \\ &= abcabcabcx^0 = abcabcabc\varepsilon = abcabcabc \end{aligned}$$

2.3 Množina všech řetězců a jazyk

Definice 2.3.1. Množina všech řetězců nad abecedou Σ je v teorii formálních jazyků označena jako Σ^*. Jakákoliv podmnožina $L \subseteq \Sigma^*$ je jazyk nad Σ.

Příklad 2.3.1. Uvažujme abecedu $\Sigma = \{a, b, c\}$. Jazyky nad abecedou Σ mohou být například tyto:

- $L_1 = \{\varepsilon\}$ označuje jazyk obsahující prázdný řetězec.
- $L_2 = \emptyset$ označuje jazyk obsahující prázdný řetězec.
- $L_3 = \{a^n b^n c^n \mid n \geq 0\}$ označuje jazyk, který obsahuje prázdný řetězec ε a řetězce se stejným počtem symbolů a, b, c.
- $L_4 = \{a^{2n} \mid n \geq 1\}$ označuje jazyk, který obsahuje řetězce se sudým počtem symbolů a.
- $L_5 = \{ww \mid w \in \Sigma\}$ označuje jazyk, který obsahuje řetězce, jež jsou složeny konkatenací dvou shodných řetězců z abecedy Σ.

2.4 Bezkontextová gramatika

Bezkontextové gramatiky jsou nutným základem, na kterém staví jednotlivé regulované gramatiky. Dobrým úvodem do problematiky bezkontextových gramatik je kniha [5], z jejichž poznatků vychází i tato práce.

Definice 2.4.1. Bezkontextová gramatika je čtveřice $G = \{N, T, P, S\}$, kde

- N je abeceda *neterminálů*;
- T je abeceda *terminálů*, přičemž $N \cap T = \emptyset$;
- P je konečná množina pravidel ve tvaru $A \to x$, kde $A \in N$ a $x \in V^*$ (V je označení pro $N \cup T$);
- $S \in N$ je startovací symbol.

Abeceda *neterminálů* N obsahuje symboly, jenž slouží k přepsání za pomocí některého pravidla z množiny P. Abeceda *terminálů* obsahuje symboly a slova konkrétního jazyka. Symbolem V se označuje úplná abeceda dané gramatiky, tj. sjednocená abeceda terminálů a neterminálů. Konečná množina P je z matematického hlediska relace z N do V^*. Místo relačního zápisu $(A, x) \in P$ zapisujeme pravidla ve tvaru $A \to x$. Pro označení počátečního neterminálu, tj. symbolu, který je určen k přepsání, se obvykle využívá symbol S.

Pro lepší pochopení se při využívání gramatik zavedly konvence. *Neterminály* se píší obvykle velkým písmenem. *Terminály* se obvykle píší malým písmenem, číslicí nebo symbolem. To ovšem platí v případě užívání teoretických jazyků. Pokud je řeč o konkrétních programovacích jazycích, jsou pod pojmem *terminál* chápána klíčová slova daného programovacího jazyka, např. *abstract*, *base*, *decimal* v jazyce *C#* [9].

2.4.1 Derivace, přímá derivace, větná forma, věta, terminální derivace

Definice 2.4.2. Mějme bezkontextovou gramatiku $G = \{N, T, P, S\}$. Přímá derivace nad V^* je označena jako $\Rightarrow$. *Přímá derivace* je definována jako $x \Rightarrow y$, pokud $x = x_1 u x_2$, $y = y_1 v y_2$ a $u \to v \in P$, kde x_1, x_2, y_1, $y_2 \in V^*$.

Definice 2.4.3. Jelikož $\Rightarrow$ je relace, $\Rightarrow^k$ je k-tá mocnina $\Rightarrow$, pro $k \geq 0$. $\Rightarrow^+$ je tranzitivní uzávěr $\Rightarrow$ a $\Rightarrow^*$ je reflexivní-tranzitivní uzávěr $\Rightarrow$. Nechť $D : S \Rightarrow^* x$ je derivace, pro nějaké $x \in V^*$. Potom x označujeme pojmem *větná forma*. Pokud $x \in T^*$, pak x je *věta*. Pokud x je věta, pak D je *terminální derivace*.

2.4.2 Generovaný jazyk

Definice 2.4.4. Generovaný jazyk $L(G)$, je množina všech vět, které jsou definovány následovně:

$$L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$$

2.4.3 Bezkontextový jazyk

Definice 2.4.5. Mějme jazyk L. L je bezkontextový jazyk, pokud existuje bezkontextová gramatika, která tento jazyk generuje.

Příklad 2.4.1. Uvažujme jazyk $L = \{a^n b^n \mid n \geq 0\}$. Tento jazyk obsahuje řetězce: ε, ab, $aabb$, $aaabbb$, atd.
Uvažujme následující bezkontextovou gramatiku:

$G = \{N, T, P, S\}$, kde

$N = \{S\}$,

$T = \{a, b\}$,

$P = \{1 : S \to \varepsilon, 2 : S \to aSb\}$.

Tato gramatika generuje následující řetězce pomocí následujících derivací:

$S \Rightarrow \varepsilon[1]$

$S \Rightarrow aSb[2] \Rightarrow ab[1]$

$S \Rightarrow aSb[2] \Rightarrow aaSbb[2] \Rightarrow aabb[1]$

$S \Rightarrow aSb[2] \Rightarrow aaSbb[2] \Rightarrow aaaSbbb[2] \Rightarrow aaabbb[1]$

...

Tyto generované řetězce jsou shodné s řetězci, které obsahuje jazyk $L = \{a^n b^n | n \geq 0\}$. Z toho vyplývá, že jazyk L je bezkontextový.

2.4.4 Označení pravidel

V gramatikách, které jako základ využívají bezkontextové gramatiky, se velmi často odkazuje na jednotlivá pravidla. Z důvodu větší stručnosti pří psaní pravidel je potřeba jednotlivá pravidla označit.

Definice 2.4.6. Nechť existuje množina označení pravidel Ψ. Mezi množinou pravidel P a množinou označení pravidel Ψ existuje bijekce ψ. Zobrazení ψ mapuje každé pravidlo $u \to v \in P$ na r, kde $r \in \Psi$. Nechť P^* označuje množinu všech sekvencí pravidel z P a Ψ^* nechť označuje množinu všech sekvencí značení pravidel z Ψ.

Jednotlivá označení pravidel r musí být unikátní. V této práci budou očíslovány od 1 do n, kde n je počet pravidel dané gramatiky. Bezkontextová gramatika je pak definována pěticí $G = (N, T, \Psi, P, S)$.

2.5 Chomského hierarchie gramatik

Ne všechny jazyky jsou stejně silné a ne všechny lze napsat pomocí bezkontextových gramatik. Existuje hierarchie jazyků, která dělí jazyky podle gramatik, které je popisují. Tuto hierarchii vytvořil Noam Chomsky v roce 1956 [2].

2.5.1 Gramatiky typu 0

Definice 2.5.1. Gramatiky typu 0 se nazývají *neomezené gramatiky*. Mezi gramatiky tohoto typu patří všechny gramatiky. Jazyky, které jsou generovány neomezenými gramatikami se nazývají *rekurzivně vyčíslitelné* a značí se RE (*recursively enumerable*) nebo L_0. Rekurzivně vyčíslitelné jazyky je možné zpracovávat *Turingovým strojem*. Ne všechny jazyky lze však generovat pomocí nějaké gramatiky. Z tohoto důvodu existují i jazyky, které nejsou ani typu 0.

2.5.2 Gramatiky typu 1

Definice 2.5.2. *Kontextové gramatiky* neboli gramatiky typu 1 generují *kontextové jazyky.* Pro kontextové gramatiky platí, že každé pravidlo P je ve tvaru $x_1Ax_2 \to x_1yx_2$, kde x_1, x_2 jsou řetězce z V^*, $A \in N$ je neterminální symbol a $y \in V^+$ je neprázdný řetězec. Rodina kontextových jazyků bývá označována CS (*context sensitive*) nebo L_1. Kontextové jazyky je možné zpracovávat *lineárně ohraničenými automaty.*

2.5.3 Gramatiky typu 2

Definice 2.5.3. Gramatiky typu 2 se označují jako *bezkontextové.* Stejný název nesou i jazyky, které jsou bezkontextovými gramatikami generovány. Pravidla bezkontextových gramatik jsou ve tvaru $A \to x$, kde $A \in N$ je neterminální symbol a $x \in V^*$ je řetězec složený z terminálů a neterminálů. Bezkontextové jazyky se nejčastěji označují CF (*context-free*) nebo L_2 a jsou zpracovávány *zásobníkovými automaty.*

2.5.4 Gramatiky typu 3

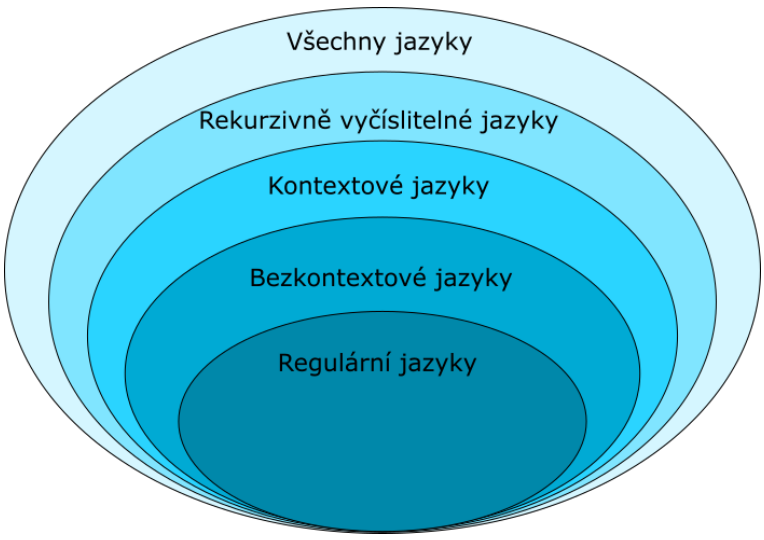
Definice 2.5.4. Gramatiky typu 3 se nazývají *pravolineární* a generují *regulární jazyky.* Pravolineární gramatiky obsahují pouze pravidla ve tvaru $A \to xB$ nebo ve tvaru $A \to x$, kde $A, B \in N$ jsou neterminální symboly a $x \in T^*$ je řetězec terminálů. Regulární jazyky se označují REG (*regular*) nebo L_3 a je možné je zpracovávat *konečnými automaty.*

2.5.5 Hierarchie jazyků

Noam Chomsky ve svém díle [3] v roce 1959 definoval teorém, který nám popisuje hierarchii množin jazyků, které lze zapsat pomocí gramatik:

$$L_0 \supseteq L_1 \supseteq L_2 \supseteq L_3.$$

Na obrázku 2.1 je zobrazena Chomského hierarchie jazyků včetně množiny všech jazyků, do které patří i jazyky, které pouze gramatikami zapsat nelze.



Obrázek 2.1: Hierarchie jazyků podle Chomského

Kapitola 3

Regulované gramatiky

Regulované gramatiky si jako základ berou bezkontextové gramatiky. Tyto gramatiky jsou rozšířeny o matematický aparát, který umožňuje řídit použití pravidel během generování jazyka. V této práci je popsáno šest druhů regulovaných gramatik: regulárně řízené gramatiky, maticové gramatiky, programované gramatiky, gramatiky s nahodilým kontextem, gramatiky s rozptýleným kontextem a stavové gramatiky. Definice jednotlivých regulovaných gramatik jsou čerpány z monografie [7]. Tyto definice jsou v některých případech mírně modifikovány, aby vyhovovaly potřebám této práce. Pro všechny regulované gramatiky je uveden příklad generování kontextového jazyka $L = \{a^n b^n c^n \mid n \geq 0\}$, aby byly lépe vidět rozdíly mezi jednotlivými gramatikami. U jednotlivých definic nejsou uváděny definice přímých derivací, protože jsou obdobné a všechny vychází z bezkontextových gramatik. Tato definice je uvedena pouze u programovaných gramatik, protože na ty se tato práce soustředí nejvíce. Pokud by čtenáře zajímaly přesné definice přímých derivací všech regulovaných gramatik, je možné je najít v knize [7].

3.1 Regulárně řízená gramatika

Regulárně řízená gramatika H je bezkontextová gramatika G, která je rozšířená o regulární řídící jazyk Ξ, jenž je definován nad množinou pravidel. Každé řídící slovo z Ξ reprezentuje sekvenci pravidel, která je možné aplikovat pří generování řetězce.

Definice 3.1.1. Regulárně řízená gramatika je dvojice:

$$H = (G, \Xi),$$

kde

- $G = (N, T, \Psi, P, S)$ je bezkontextová gramatika, nazývaná jádrová gramatika;
- $\Xi \subseteq \Psi^*$ je regulární jazyk, nazývaný řídící jazyk.

3.1.1 Generovaný jazyk

Definice 3.1.2. Jazyk generovaný regulárně řízenou gramatikou je definován následovně:

$$L(H) = \{w \in T^* \mid S \Rightarrow^* w[\alpha], \alpha \in \Xi\}.$$

Symbol α je označení pro sekvenci pravidel P, která mohou být za sebou při generování řetězce w. Jinými slovy by se dalo říci, že jazyk generovaný regulárně řízenými gramatikami se skládá ze všech řetězců $w \in T^*$, pro které existuje derivace v G,

$$S \Rightarrow w_1[r_1] \Rightarrow w_2[r_2] \Rightarrow ... \Rightarrow w_n[r_n],$$

kde

$$w = w_n, r_1 r_2 ... r_n \in \Xi, \text{ pro } n \geq 1.$$

Příklad 3.1.1. Nechť $H = (G, \Xi)$ je regulárně řízená gramatika, kde $G = (\{S, A, B, C\}, \{a, b, c\}, \Psi, P, S)$ je bezkontextová gramatika. Množina P se skládá z následujících pravidel:

$$r_1 : S \to ABC,$$

$$r_2 : A \to aA,$$

$$r_3 : B \to bB,$$

$$r_4 : C \to cC,$$

$$r_5 : A \to \varepsilon,$$

$$r_6 : B \to \varepsilon,$$

$$r_7 : C \to \varepsilon.$$

Řídící jazyk Ξ je definován následovně: $\Xi = \{r_1\}\{r_2 r_3 r_4\}^*\{r_5 r_6 r_7\}$. Nejprve musí být definováno pravidlo r_1. Následně je aplikována sekvence pravidel r_2, r_3 a r_4. Tuto sekvenci lze použít v jakémkoliv množství. Derivace je ukončena pomocí pravidel r_5, r_6 a r_7.

Uvedený příklad generuje jazyk $L(H) = \{a^n b^n c^n | n \geq 0\}$. Jako příklad slova z jazyka $L(H)$ lze uvést *aabbcc*. Ten lze získat pomocí následujících derivací:

$$\begin{aligned}
S &\Rightarrow ABC[r_1] \\
&\Rightarrow aABC[r_2] \\
&\Rightarrow aAbBC[r_3] \\
&\Rightarrow aAbBcC[r_4] \\
&\Rightarrow aaAbBcC[r_2] \\
&\Rightarrow aaAbbBcC[r_3] \\
&\Rightarrow aaAbbBccC[r_4] \\
&\Rightarrow aabbBccC[r_5] \\
&\Rightarrow aabbccC[r_6] \\
&\Rightarrow aabbcc[r_7]
\end{aligned}$$

3.2 Maticová gramatika

Maticová gramatika rozšiřuje bezkontextovou gramatiku G o množinu M, jejíž slova jsou složena z abecedy pravidel P. Řetězce množiny M jsou nazývány matice. Při generování

jazyka je vybrána ta matice, jejíž první pravidlo je možné aplikovat. Následně jsou aplikována všechna pravidla této gramatiky v pořadí, které udává matice. Při úspěšném použití pravidel celé matice je možné derivaci buď ukončit, anebo vybrat novou matici, jejíž pravidla budou aplikována v následujících krocích.

Definice 3.2.1. Maticová gramatika je dvojice:

$$H = (G, M),$$

kde

- $G = (N, T, \Psi, P, S)$ je bezkontextová gramatika, nazývaná jádrová gramatika;
- $M \subseteq \Psi^+$ je konečný jazyk, jehož elementy jsou nazývány *matice*.

3.2.1 Generovaný jazyk

Definice 3.2.2. Jazyk generovaný maticovými gramatikami je definován následovně:

$$L(H) = \{w \in T^* \mid S \Rightarrow^* w\}.$$

Pro generovaný jazyk platí, že terminální řetězec je nutné získat konečným počtem derivací. Posloupnost pravidel, která jsou použita pro generování řetězce w, je získávána z matic množiny M. Při postupné aplikaci těchto matic vznikne celková posloupnost použitých pravidel.

Příklad 3.2.1. Nechť $H = (G, M)$ je maticová gramatika, kde množiny G a P jsou definovány stejně jako v příkladu 3.1.1. Množina M, obsahuje následující matice:

$$M = \{r_1,\ r_2r_3r_4,\ r_5r_6r_7\}.$$

Řetězec jazyka $L(H) = \{a^nb^nc^n \mid n \geq 0\}$ bychom dostali stejnou posloupností derivací jako v příkladu 3.1.1.

3.3 Programovaná gramatika

Programovaná gramatika modifikuje bezkontextovou gramatiku tak, že ke každému pravidlu z množiny P přidá množinu pravidel Q, která určuje pravidla, jež mohou následovat za právě použitým pravidlem. Pokud žádné z pravidel množiny Q nevyhovuje, generovaný řetězec je zamítnut.

Definice 3.3.1. programovaná gramatika je pětice:

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

kde

- symboly N, T, Ψ, S jsou definovány jako v bezkontextových gramatikách;
- $P \subseteq \Psi \times N \times (N \cup T)^* \times 2^{\Psi}$ je konečná relace, která je pojmenována množina pravidel. Jestliže (r, A, x, q_r), $(s, A, x, q_s) \in P$, q_r, $q_s \in Q$, pak $(r, A, x, q_r) = (s, A, x, q_s)$.

Místo $(r, A, x, q_r) \in P$, píšeme $r\colon A \to x, q_r \in P$.

Nechť $V = N \cup T$ je úplná abeceda. Relace *přímé derivace* $\Rightarrow$ je definována nad $V^* \times \psi$ následovně: Pro $(x_1, r), (x_2, s) \in V^* \times \psi$,

$$(x_1, r) \Rightarrow (x_2, s) \text{ v } G$$

pouze když:

$$x_1 = yAz,$$

$$x_2 = ywz,$$

$$r\colon A \to x, q_r \in P,$$

$$s \in q_r.$$

3.3.1 Generovaný jazyk

Definice 3.3.2. Jazyk generovaný programovanými gramatikami je definován následovně:

$$L(G) = \{w \in T^* \mid (S, r) \Rightarrow^* (w, s); r, s \in \Psi\}.$$

Při generování řetězce w se pokračuje do té doby, dokud výsledný řetězec není terminální (skládá se pouze z terminálů). Pokud by nešlo aplikovat žádné z předepsaných pravidel, pak je derivace ukončena také, avšak neúspěšně.

Příklad 3.3.1. Nechť $G = (\{S, A, B, C\}, \{a, b, c\}, \{r_1, r_2, r_3, r_4, r_5, r_6, r_7\}, P, S)$ je programovaná gramatika. Množina P se skládá z následujících pravidel:

$$r_1 :\ S \to ABC, \{r_2, r_5\},$$

$$r_2 :\ A \to aA, \{r_3\},$$

$$r_3 :\ B \to bB, \{r_4\},$$

$$r_4 :\ C \to cC, \{r_2, r_5\},$$

$$r_5 :\ A \to \varepsilon, \{r_6\},$$

$$r_6 :\ B \to \varepsilon, \{r_7\},$$

$$r_7 :\ C \to \varepsilon, \{r_7\}.$$

Generovaný jazyk a derivace jeho řetězců jsou shodné jako v příkladu 3.1.1.

3.4 Gramatika s nahodilým kontextem

V gramatikách s nahodilým kontextem G je každému pravidlu přiřazena množina povolených symbolů. Během provádění derivačního kroku je dané pravidlo aplikovatelné, pokud se všechny symboly z kontextu U daného pravidla vyskytují v aktuální větné formě generovaného řetězce.

Definice 3.4.1. Gramatika s nahodilým kontextem je čtveřice:

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

kde

- symboly N, T, S jsou definovány stejně jako v bezkontextových gramatikách;
- $P \subseteq N \times (N \cup T)^* \times 2^N$ je konečná relace. P je nazývána množina pravidel.

Každé pravidlo $(A, x, U) \in P$ se zapisuje jako $A \to x, U$. Symbol U představuje tzv. *povolený kontext*. Pro aplikaci daného pravidla musí být všechny neterminály z množiny U obsaženy v aktuálním stavu generovaného slova.

3.4.1 Generovaný jazyk

Definice 3.4.2. Jazyk generovaný gramatikami s nahodilým kontextem je definován jako:

$$L(G) = \{w \in T^* \mid S \Rightarrow^* w\}.$$

Jednotlivé derivace musí brát v úvahu *povolený kontext*.

Příklad 3.4.1. Nechť $G = (\{S, A, B, C, A', B', C'\}, \{a, b, c\}, P, S)$ je gramatika s nahodilým kontextem. Množina pravidel P je definována následovně:

$$\begin{aligned}
1 &: S \to ABC, \emptyset, \\
2 &: A \to aA', \{B\}, \\
3 &: B \to bB', \{C\}, \\
4 &: C \to cC', \{A'\}, \\
5 &: A' \to A, \{B'\}, \\
6 &: B' \to B, \{C'\}, \\
7 &: C' \to C, \{A\}, \\
8 &: A \to \varepsilon, \{B\}, \\
9 &: B \to \varepsilon, \{C\}, \\
10 &: C \to \varepsilon, \emptyset.
\end{aligned}$$

Tato gramatika generuje již zmíněný jazyk $L(G) = \{a^n b^n c^n \mid n \geq 0\}$. Řetězec $aabbcc$, patřící do tohoto jazyka, lze získat pomocí následujících derivací:

$$\begin{aligned}
S &\Rightarrow ABC[1] \Rightarrow aA'BC[2] \Rightarrow aA'bB'C[3] \Rightarrow aA'bB'cC'[4] \\
&\Rightarrow aAbB'cC'[5] \Rightarrow aAbBcC'[6] \Rightarrow aAbBcC[7] \\
&\Rightarrow aaA'bBcC[2] \Rightarrow aaA'bbB'cC[3] \Rightarrow aaA'bbB'ccC'[4] \\
&\Rightarrow aaAbbB'ccC'[5] \Rightarrow aaAbbBccC'[6] \Rightarrow aaAbbBccC[7] \\
&\Rightarrow aabbBccC[8] \Rightarrow aabbccC[9] \Rightarrow aabbcc[10].
\end{aligned}$$

U pravidel 9 a 10 záleží na pořadí při jejich užití. Kdyby bylo použito nejdříve pravidlo 10, pak by pravidlo 9 nemohlo být aplikováno, protože neterminál C by byl již smazán z generovaného řetězce.

3.5 Gramatika s rozptýleným kontextem

Všechny předchozí regulované gramatiky používaly v každém kroku derivace pouze jediné pravidlo. Gramatiky s rozptýleným kontextem jsou naopak postaveny na sekvencích bezkontextových pravidel. Gramatika s rozptýleným kontextem G může během jednoho derivačního kroku přepsat několik neterminálů zároveň.

Definice 3.5.1. Gramatika s rozptýleným kontextem je čtveřice:

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

kde

- V je *celková abeceda*;
- $T \subset V$ je abeceda *terminálů*;
- P je množina pravidel ve formě ve formě $(A_1, \dots A_n) \to (x_1, \dots x_n)$, kde $n \geq 1$, $A_i \in V - T$, $x_i \in V^*$, pro každé i, $1 \leq i \leq n$;
- $S \in V - T$ je *startovací symbol.*

3.5.1 Generovaný jazyk

Definice 3.5.2. Jazyk generovaný gramatikami s rozptýleným kontextem je definován následovně:

$$L(G) = \{w \in T^* \mid S \Rightarrow^* w\}.$$

Při jednotlivých derivacích se přepisuje několik neterminálů zároveň.

Příklad 3.5.1. Jazyk $L(G) = \{a^n b^n c^n \mid n \geq 0\}$ nechť je generován gramatikou s rozptýleným kontextem $G = (\{S, A, B, C, a, b, c\}, \{a, b, c\}, P, S)$. Množina pravidel P je definována následovně:

$$P = \{(S) \to (AAA),$$

$$(A, A, A) \to (aA, bA, cA),$$

$$(A, A, A) \to (\varepsilon, \varepsilon, \varepsilon)\}.$$

Slovo *aabbcc* jazyka $L(G)$ je generováno gramatikou G takto:

$$S \Rightarrow AAA \Rightarrow aAbAcA \Rightarrow aaAbbAccA \Rightarrow aabbcc.$$

3.6 Stavová gramatika

Stavové gramatiky G rozšiřují bezkontextové gramatiky o další stavový mechanismus. Během derivačního kroku se přepíše nejlevější výskyt daného neterminálu podle používaného pravidla a změní se současný stav z množiny stavů W. Z toho vyplývá, že jednotlivé stavy ovlivňují pravidla, která lze v konkrétní dobu použít.

Definice 3.6.1. Stavové gramatika je pětice:

$$G = (V, W, T, P, S),$$

kde

- V je *celková abeceda*;
- W je konečná množina *stavů*;
- $T \subset V$ je abeceda *terminálů*;
- $S \in V - T$ je *startovací symbol.*
- $P \subseteq (W \times (V - T)) \times (W \times V^+)$ je konečná relace.

Namísto $(a, A, p, v) \in P$ se pravidlo zapíše ve tvaru $(q, A) \to (p, v) \in P$.

3.6.1 Generovaný jazyk

Definice 3.6.2. Jazyk generovaný stavovými gramatikami je definován následovně:

$$L(G) = \{w \in T^* \mid (q, S) \Rightarrow^* (p, w); q, p \in W\}.$$

Při každé derivaci se mění stav. Generování je u konce po získání terminálního řetězce.

Příklad 3.6.1. Pro srovnání s ostatními gramatikami bude generován opět jazyk $L(G) = \{a^n b^n c^n \mid n \geq 0\}$. Nechť $G = (\{S, A, B, C, a, b, c\}, \{p_0, p_1, p_2, p_3, p_4\}, \{a, b, c\}, P, S)$ je stavová gramatika, která tento jazyk generuje. Množina pravidel P nechť je definována následovně:

$$(p_0, S) \to (p_0, ABC),$$

$$(p_0, A) \to (p_1, aA),$$

$$(p_1, B) \to (p_2, bB),$$

$$(p_2, C) \to (p_0, cC),$$

$$(p_0, A) \to (p_3, \varepsilon),$$

$$(p_3, B) \to (p_4, \varepsilon),$$

$$(p_4, C) \to (p_0, \varepsilon).$$

Slovo *aabbcc* jazyka $L(G)$ je generováno stavovou gramatikou G takto:

$$\begin{aligned}
(p_0, S) &\Rightarrow (p_0, ABC)\ [(p_0, S) \to (p_0, ABC)] \\
&\Rightarrow (p_1, aABC)\ [(p_0, A) \to (p_1, aA)] \\
&\Rightarrow (p_2, aAbBC)\ [(p_1, B) \to (p_2, bB)] \\
&\Rightarrow (p_0, aAbBcC)\ [(p_2, C) \to (p_0, cC)] \\
&\Rightarrow (p_1, aaAbBcC)\ [(p_0, A) \to (p_1, aA)] \\
&\Rightarrow (p_2, aaAbbBcC)\ [(p_1, B) \to (p_2, bB)] \\
&\Rightarrow (p_0, aaAbbBccC)\ [(p_2, C) \to (p_0, cC)] \\
&\Rightarrow (p_3, aabbBccC)\ [(p_0, A) \to (p_3, \varepsilon)] \\
&\Rightarrow (p_4, aabbccC)\ [(p_3, B) \to (p_4, \varepsilon)] \\
&\Rightarrow (p_0, aabbcc)\ [(p_4, C) \to (p_0, \varepsilon)].
\end{aligned}$$

3.7 Generativní síla regulovaných gramatik

Síla zmíněných typů regulovaných gramatik je shodná. Všechny tyto jazyky obsahují podobné principy, které jsou aplikovatelné na stejnou množinu jazyků. Generativní síla těchto jazyků je popsána následujícím teorémem:

$$CF \subset \text{regulované gramatiky} \subset RE \text{ [8]}.$$

Tento teorém říká, že množina jazyků, které je možné generovat pomocí regulovaných gramatik, je větší než množina bezkontextových jazyků, ale menší než množina rekurzivně vyčíslitelných jazyků.

Kapitola 4

Syntaktická analýza

Jednou z nejdůležitějších procedur při překladu jazyků je syntaktická analýza [6]. Při syntaktické analýze se zjistí, zda text v daném jazyce je syntakticky správně zapsán. Syntaktickou analýzu provádí syntaktický analyzátor, někdy též označován *parser*. Vstupem syntaktického analyzátoru je vstupní řetězec a vstupní gramatika. Vstupní řetězec se skládá z logicky oddělených lexikálních jednotek, pro které se využívá zkrácený výraz *lexémy*. Ty jsou reprezentovány tzv. *tokeny*. Tokeny na rozdíl od lexémů mohou obsahovat atributy. Atributem zpravidla bývá typ tokenu, např. číslo, identifikátor, klíčové slovo v programovacích jazycích. Tokeny jsou syntaktickému analyzátoru poskytnuty lexikálním analyzátorem. Ten zároveň kontroluje lexikální správnost vstupního řetězce.

4.1 Teoretické a praktické jazyky

Je rozdíl pokud se mluví o syntaktické analýze z praktického a teoretického hlediska. V praxi nás zajímá mnohem více podrobností, kdežto v teorii se zaměřujeme jenom na jádro věci.

Na teoretické úrovni se pro syntaktickou analýzu využívají teoretické jazyky. Lexémy těchto teoretických jazyků bývají zpravidla malá písmena abecedy. Jako příklad je možné uvést jazyk:

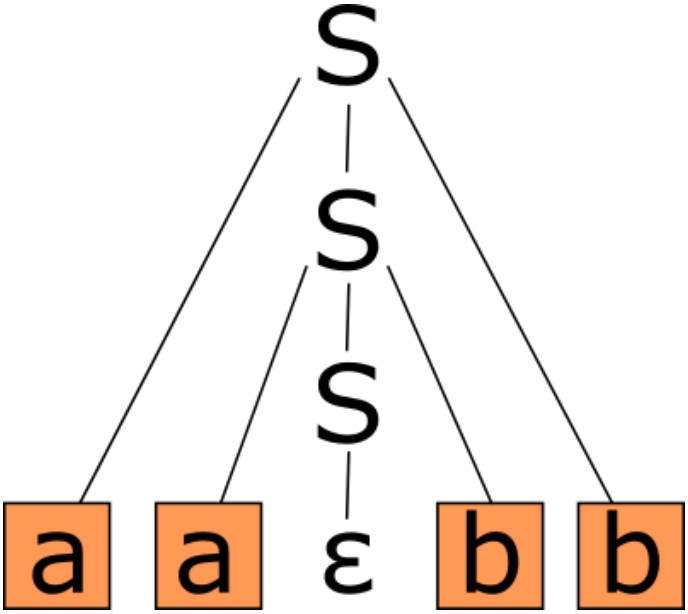
$$L = \{a^n b^n \mid n \geq 0\}.$$

U teoretických jazyků není důležitý typ jednotlivých lexikálních jednotek. Tato informace je zcela vypuštěna. Zjišťuje se pouze to, zda vstupní řetězec patří nebo nepatří do jazyka generovaného vstupní gramatikou.

Při použití jazyků praktických se kromě informace o příslušnosti vstupního řetězce k dané gramatice využívá také typ jednotlivých lexémů. Pokud se mluví o praktických jazycích, nemusí se jednat pouze o jazyky programovací, ale mohou to být i jazyky přirozené, např. čeština.

4.2 Derivační strom

Syntaktický analyzátor čte vstupní řetězec zleva doprava, využívá gramatických pravidel vstupní gramatiky a jejich postupnou aplikací se snaží nalézt derivaci vstupního řetězce. Graficky se tato gramatická derivace zobrazuje jako tzv. derivační strom. Na obrázku 4.1 je zobrazen derivační strom pro bezkontextovou gramatiku z příkladu 2.4.1 při derivování řetězce *aabb*.



Obrázek 4.1: Derivační strom řetězce *aabb*

Konstrukce derivačního stromu je možná dvěma základními způsoby: shora dolů a zespodu nahoru. Syntaktická analýza shora dolů se nejdříve zabývá nejvyšší úrovní derivačního stromu. Postupně prochází dolů k listům s využitím pravidel dané gramatiky. Derivační strom se konstruuje zleva doprava a vstup je čten rovněž zleva doprava. Pokud na nejlevějším místě generovaného řetězce dostaneme terminální symbol, tak se jej pokusíme pokrýt. Pokrytí symbolu spočívá v porovnání terminálních symbolů v generovaném řetězci a ve vstupním řetězci. Tyto terminální symboly se musí nacházet na stejném místě. Toho se docílí tím, že se pokrývá vždy nejlevější možný symbol (na na jehož místě se musí nacházet terminál) a následně se za nejlevější symbol považuje symbol následující.

Pravidla se aplikují vždy na nejlevější neterminál. Pokud žádné pravidlo nelze využít, tak je derivace neúspěšná a vstupní řetězec nepatří do vstupní gramatiky. Syntaktickou analýzu shora dolů používají LL syntaktické analyzátory. Syntaktická analýza zespodu nahoru je analogií k syntaktické analýze shora dolů. Derivační strom se konstruuje od listů ke kořeni.

Pokud se strom podaří úspěšně vytvořit, pak je syntaktická analýza úspěšná a u vstupního řetězce bylo dokázáno, že patří do jazyka generovaného danou gramatikou. Pokud v jakémkoli místě tvoření derivačního stromu dojde k rozporu se vstupním řetězcem, pak je tento řetězec zamítnut.

4.3 Modifikace metod syntaktické analýzy

Pro syntaktickou analýzu řetězců, jejichž derivace se provádí pomocí programovaných gramatik vycházím z klasické syntaktické analýzy shora dolů. Vstupní řetězec je procházen zleva doprava a při pokrývání se pokrývají nejlevější terminály. Na rozdíl od syntaktické analýzy shora dolů se pravidla neaplikují pouze na nejlevější neterminál, ale postupně dochází k procházení všech neterminálů, dokud jedno z předepsaných pravidel nelze využít. V každé části derivace rovněž nelze použít všechna pravidla, ale pouze ta pravidla, která jsou součástí množiny následujících pravidel předchozího použitého pravidla. Na začátku se použije pravidlo číslo jedna, u nějž se očekává, že na levé straně se vyskytuje počáteční

neterminál. Podrobnější informace k metodě syntaktické analýzy, kterou jsem aplikoval na programované gramatiky, lze najít v pseudokódu následujícího algoritmu:

Algorithm 1 Algoritmus syntaktické analýzy pro programované gramatiky

```
procedure AUTOMATON
    Inicializace seznamu následujících pravidel a vložení startovacího pravidla
    while Dají se použít nějaká pravidla do
        Rozdělení následujících pravidel na epsilon a neepsilon pravidla
        Průchod neterminálů řetězce a hledání prvního použitelného pravidla v neepsilon pravidlech
        if Nenalezeno then
            Průchod neterminálů řetězce a hledání prvního použitelného pravidla v epsilon pravidlech
            if Nenalezeno then
                Řetězec zamítnut
        Použití nalezeného pravidla a aktualizování seznamu následujících pravidel
        Pokrytí možných terminálů generovaného a vstupního řetězce
        if Neshoda then
            Řetězec zamítnut
    Řetězec přijat
    Vypsání seznamu použitých pravidel
```

V tomto algoritmu se nevyskytují veškeré detaily. Jedním z důležitých detailů je fakt, že se porovnává počet terminálů vstupního a generovaného řetězce. Jakmile jsou stejné, pak se nesmí používat pravidla, která vytváří další terminály, a jsou preferována ε pravidla.

Kapitola 5

Vývoj aplikace

Při vývoji aplikace je důležité, aby software byl dobře navržen, aby byl pochopitelný a snadno upravitelný. Při konstrukci velkého softwaru je zapotřebí použít nějaké metodiky. Jednou z nejznámějších metodik je tzv. *Unified Process* (zkratka UP). Při popisování této metodiky vycházím z knihy [1].

Metodika UP vychází ze základní myšlenky, která říká, že člověku se lépe řeší menší problémy než větší. Je to z toho důvodu, že při velkém problému je potřeba se zaměřit na mnoho různých detailů, což člověk není schopen vstřebat. Unified Process přichází s tím, že se každý velký softwarový projekt rozdělí na několik menších mini-projektů, které je možno řešit samostatně. Každý takový projekt je řešen iterativně a při skončení jednoho je možné začít další. Každý mini-projekt je samostatnou iterací a skládá se z pěti základních pracovních postupů:

- Požadavky,
- analýza,
- návrh,
- implementace,
- testování.

Požadavky specifikují, co by měl daný software dělat. Při analýze se tyto požadavky vybrousí a začnou se strukturovat. V době návrhu je vhodné zapsat funkce softwaru nějakým formálním způsobem. Často se využívají UML diagramy. Následně se vytvoří daný software a je možné ho začít testovat.

Metodika UP je obecná metodika a slouží hlavně pro tvorbu velmi velkých projektů. Program, který je součástí této práce je však malý, a proto je metodiku UP potřeba trochu modifikovat. Celkový projekt by se dal považovat již za mini-projekt. Z tohoto důvodu při tvorbě softwaru proběhne pouze jedna iterace UP. Analýza je vhodná zejména pro projekty, které jsou mnohoznačné a umožňují různé případy užití systému pro různé uživatele. Například v internetovém obchodu bude mít jiná práva běžný uživatel a jiná administrátor systému. Tvořený software ovšem není informační systém a všichni jeho uživatelé budou mít stejná práva. Z tohoto důvodu bude brán větší zřetel na část návrhu aplikace. Pracovní postup analýzy bude redukován.

5.1 Specifikace a analýza požadavků

Cílem této práce je navrhnout a vytvořit syntaktický analyzátor na základě regulovaných gramatik. K implementaci byly zvoleny programované gramatiky. Jako implementační jazyk byl vybrán *C#* za použití vývojového prostředí *Visual Studio 2013*.

Pro aplikaci jako je syntaktický analyzátor není vhodné používat grafické uživatelské rozhraní. Při použití překladačů není důležité, jak vypadá vlastní program, ale jeho funkčnost a použitelnost. Z tohoto důvodu byla zvolena forma konzolové aplikace. Při spouštění programu je nutné vybrat soubor se vstupní gramatikou, jak je specifikováno v sekci 5.1.2. Pro lepší přehlednost a zacházení je vhodné, aby byl soubor s gramatikou v přesně definovaném formátu. Tento formát je uveden v sekci 5.1.1.

Po zadání gramatiky je uživatel vyzván k zadání vstupního řetězce. Vstupní gramatiku i vstupní řetězec dále využívá syntaktický analyzátor ke své hlavní činnosti. Jeho hlavním úkolem je zjistit, zda zadaný vstupní řetězec patří do vstupní gramatiky nebo ne. Pokud ano, program vypíše na výstup, že daný řetězec byl přijat, čísla a pořadí pravidel, v jakém byla použita. Dále jsou vypsány jednotlivé derivace, aby bylo vidět, jakým způsobem se počáteční symbol přepisuje na vstupní řetězec. V případě, že vstupní řetězec neodpovídá vygenerovanému řetězci, syntaktický analyzátor vypíše, že vstupní řetězec byl zamítnut. Hlavní činnost syntaktického analyzátoru se řídí algoritmem 1, který byl popisován v sekci 4.3. Po získání výsledku lze program ukončit stisknutím libovolné klávesy.

5.1.1 Formát vstupní gramatiky

Všechny množiny jsou zapisovány pomocí jednoznakových písmen nebo číslic oddělených čárkami. Na prvním řádku se nachází množina terminálů. Na druhém řádku následuje množina neterminálů. Pravidla je nutné číslovat od jedničky. Levá strana se skládá vždy z jednoho neterminálu. Řetězec na pravé straně může obsahovat terminály i neterminály. Za pravou stranou se nachází ve složených závorkách množina následujících pravidel. Pro zjednodušení vytváření těchto souborů je symbol ε reprezentován znakem e.

```
T: <množina terminálů>

N: <množina neterminálů>

<číslo pravidla>: <levá strana> -> <pravá strana>,
{<množina následujících pravidel>}
```

5.1.2 Parametry aplikace

Příklady spouštění aplikace:

- `ProgrammedGrammars grammar.txt`
- `ProgrammedGrammars grammar.txt -g`
- `ProgrammedGrammars grammar.txt --grammar`

Prvním parametrem je vždy soubor s gramatikou, který je povinný. Druhý parametr je volitelný. Pokud je zadán, tak se na konzoli vypíše obsah souboru s gramatikou, jenž byl zadán v prvním parametru. Druhý parametr existuje v dlouhé i zkrácené verzi.

5.2 Objektový návrh a implementace

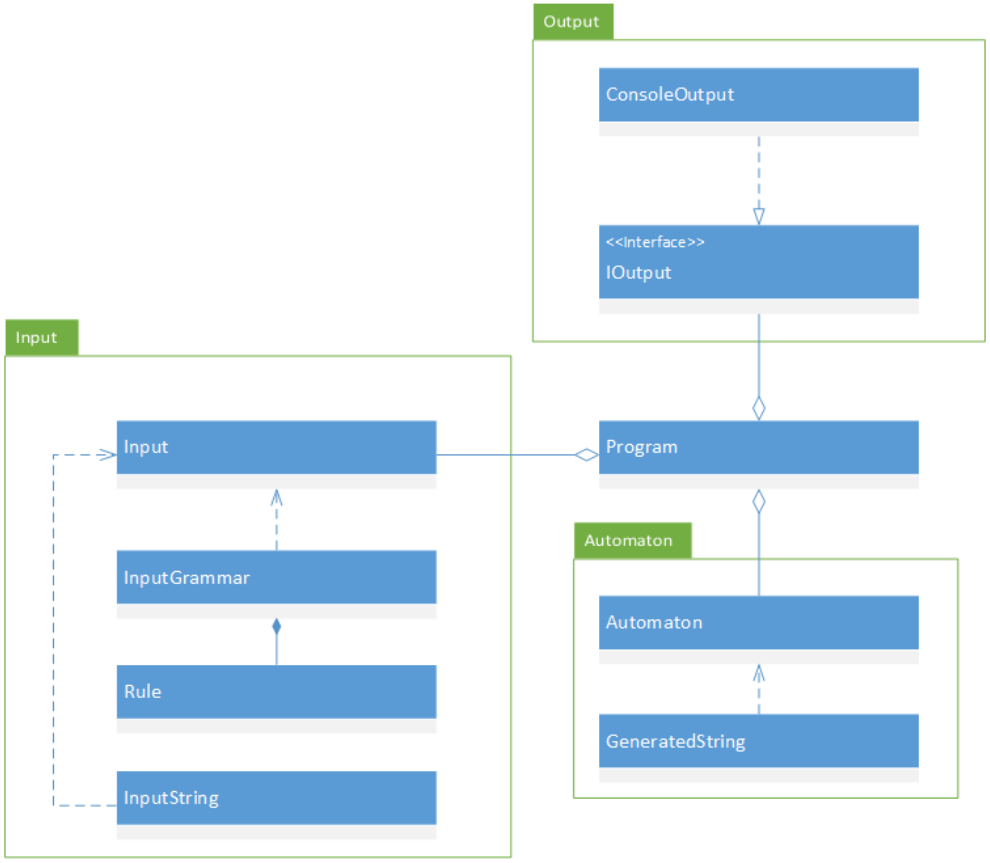
Na objektový návrh aplikace je velmi praktické a názorné použít nějaký diagram jazyka UML. Pro zobrazení návrhu programu byl zvolen diagram tříd, který je zobrazen na obrázku 5.1.

Aplikace se skládá z jedné velké agregace, kde celkovým agregátem je třída *Program*. Tato agregace se skládá ze tří součástí – vstupu *Input*, výstupu *Output* a konečného automatu *Automaton*.

Ze vstupu aplikace se získávají dvě podstatné části – vstupní gramatika *InputGrammar* a vstupní řetězec *InputString*. Vstupní řetězec je tvořen jako samostatný objekt, aby se s ním lépe pracovalo. Vstupní gramatika je kompozicí, která se skládá z jednotlivých pravidel *Rule*.

Výstup aplikace využívá rozhraní *IOutput*, aby bylo jednoduše možné přidat více výstupů. V dalších verzích aplikace by bylo možné přidat výstupy například do textového souboru, PDF, obrázku a podobně. V první verzi aplikace je využit výstup na konzoli, protože se s ním dobře pracuje a je možné ihned pozorovat výsledky.

Konečný automat využívá informace z ostatních tříd, které jsou mu poskytnuty přes třídu *Program*. Na činnosti automatu závisí generovaný řetězec *GeneratedString*, který je ve výsledku porovnán se vstupním řetězcem a výsledek je poslán za pomoci třídy *Program* na výstup.



Obrázek 5.1: Objektový návrh aplikace

5.2.1 Třída Program

Třída *Program* (obr. 5.2) představuje hlavní třídu, pomocí které se volají všechny ostatní části programu. Celá funkčnost aplikace se zpracovává v metodě *Main*. Nejprve se nastaví výstup aplikace na konzoli. Následně se ověří správnost parametrů a otevře se soubor s gramatikou. Vstupní soubor a vstupní řetězec jsou načteny a zpracovány pomocí třídy *Input*. Pokud proběhne zpracování v pořádku, tak se zavolá třída *Automaton*. Tato třída obsahuje konečný automat, jenž implementuje jádro chování programu. Podle výstupu z konečného automatu se na konzolový výstup vypíše, zda byl vstupní řetězec přijat. Pokud byl vstupní řetězec přijat, tak se vypíše seznam pravidel, která byla použita. Následně jsou na konzoli vypsány všechny derivace generovaného řetězce. V této fázi je možné program ukončit stisknutím libovolného tlačítka.

Obrázek 5.2: Třída Program

5.2.2 Třída Input

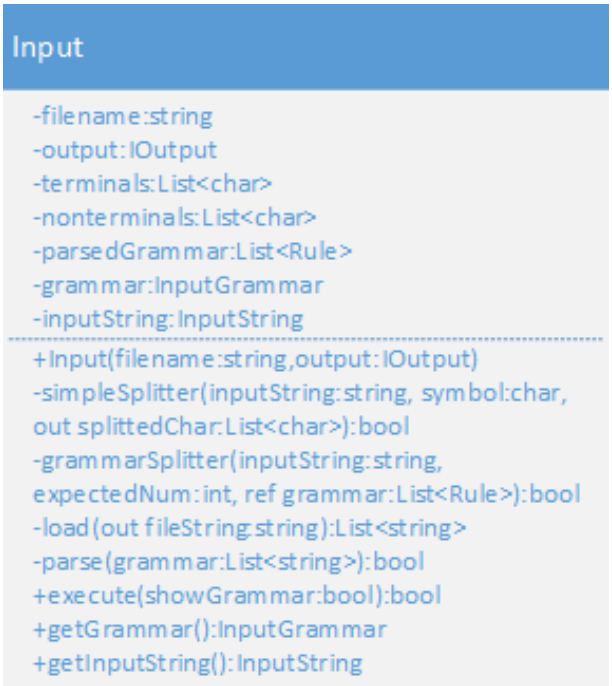
Třída *Input* (obr. 5.3) slouží ke zpracování veškerých vstupů. Metoda *execute* slouží k spuštění činnosti této třídy. Nejprve se program pokusí soubor otevřít pomocí metody *load*. Metoda *parse* slouží k rozdělení vstupního souboru na terminály, neterminály a vstupní gramatiku. K rozpoznání jednotlivých částí slouží pomocné metody *simpleSplitter* a *grammarSplitter*. Nakonec je zavolán konstruktor třídy *InputGrammar*, který vytvoří její instanci pro uložení vstupní gramatiky. Po načtení vstupního souboru je uživatel vyzván k zadání vstupního řetězce. Ten je po zadání uložen do instance třídy *InputString* a následně zkontrolován, zda obsahuje pouze terminály. Pokud by obsahoval nějaké znaky navíc, pak je tento řetězec možno ihned zamítnout. Všechny produkty třídy *Input* jsou k dispozici pomocí public metod ve třídě *Program*.

5.2.3 Třída InputGrammar

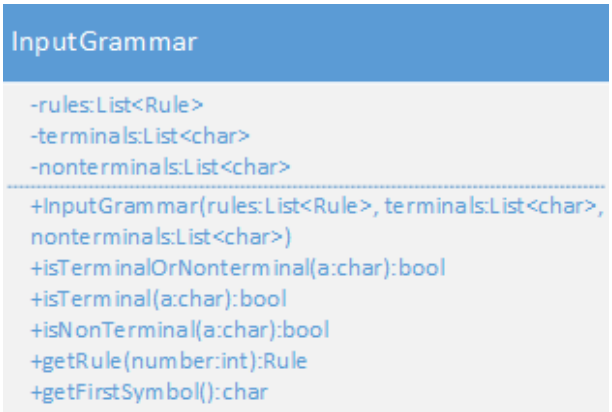
Třída *InputGrammar* (obr. 5.4) slouží k zapouzdření celé vstupní gramatiky. Obsahuje terminály, neterminály a pravidla. Tato třída se zároveň používá na to, aby se zjistilo, zda nějaký znak je terminál, neterminál nebo jedna z těchto možností (zda se nejedná o symbol, který není definován).

5.2.4 Třída Rule

Třída *Rule* (obr. 5.5) zapouzdřuje pravidla a usnadňuje manipulaci s nimi. Je možné jednotlivě přistupovat k levé straně, pravé straně a pravidlům, která mají být využita po použití aktuálního pravidla. Třída *Rule* obsahuje také metodu `isEpsilonRule` pro zjištění, zda se jedná o pravidlo, které vymaže daný neterminál. Pro porovnávání prvního znaku pravidla s aktuálně čteným znakem ze vstupu se v této třídě nachází také metoda `beginsWith`. Ta nám vrátí první znak z levé strany pravidla.



Obrázek 5.3: Třída Input



Obrázek 5.4: Třída InputGrammar

Obrázek 5.5: Třída Rule

5.2.5 Třída InputString

Třída *InputString* (obr. 5.6) slouží k zapouzdření vstupního řetězce. Tato třída umožňuje jednodušší procházení tohoto řetězce a pamatuje si již přečtené symboly, což se hodí například při pokrývání symbolů s generovaným řetězcem.

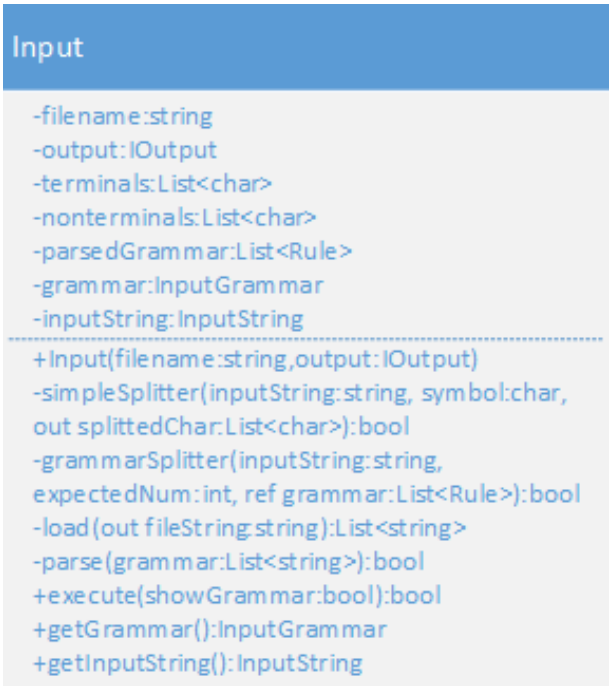
5.2.6 Třída ConsoleOutput

Tato třída (obr. 5.7) je jednou z možných realizací rozhraní *IOutput* (obr. 5.8). Rozhraní je logickou volbou pro jednoduché přidávání dalších výstupních možností. V dalších verzích programu by bylo možné například přidat výstup do textového souboru nebo PDF (portable document format).

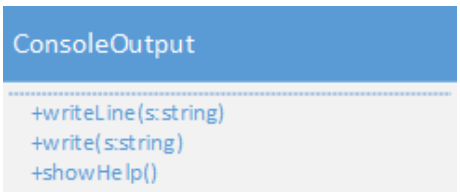
Každé rozhraní obsahuje deklarace funkcí, ale nedefinuje jejich chování. Vlastní chování je definováno ve funkcích, které dědí z tohoto rozhraní. V tomto případě rozhraní *IOutput* deklaruje tří různé metody pro výpis. Jejich vlastní tělo je definováno ve třídě *ConsoleOutput*. Je dobrým zvykem značit rozhraní (anglicky interface) velkým písmenem *I* na začátku názvu.

5.2.7 Třída Automaton

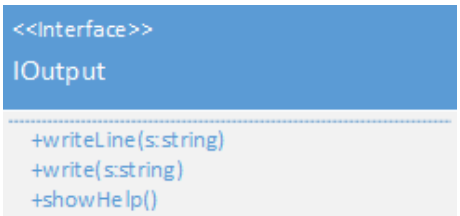
Třída *Automaton* (obr. 5.9) implementuje algoritmus syntaktického analyzátoru popsaný v sekci 1. Tato třída potřebuje jako vstup řetězec zadaný uživatelem a gramatiku. Činnost třídy *Automaton* je spouštěna metodou *execute*. Postupně dochází k odvozování generovaného řetězce (*generatedString*). Množina pravidel, která je možno právě použít, je závislá na množině následujících pravidel *nextRules*. Tato množina je získávána z posledního použitého pravidla. V sekci 5.1.1 je tato množina označena jako `<množina následujících pravidel>`. Jelikož je ve výsledku potřeba vypsat použitá pravidla, tak je k tomuto účelu použit generický seznam *usedRules*.



Obrázek 5.6: Třída InputString



Obrázek 5.7: Třída ConsoleOutput



Obrázek 5.8: Rozhraní IOutput

Obrázek 5.9: Třída Automaton

5.2.8 Třída GeneratedString

Třída *GeneratedString* (obr. 5.10) zapouzdřuje generovaný řetězec. Je vytvářena třídou *Automaton*. Při inicializaci je do této třídy uložen pouze počáteční neterminál a postupnou derivací se převádí na terminální řetězec. V proměnné *content* se nachází vlastní generovaný řetězec. Jako ukazatele na tři různé symboly v řetězci slouží tři číselné proměnné – *readIndex*, *position* a *leftmostNonterminal*. Proměnná *readIndex* je používána funkcí *getReadLetter* pro procházení neterminálů při aplikaci pravidel. Funkce *getLeftmostNonterminal* využívá proměnnou *leftmostNonterminal* pro získání hodnoty nejlevějšího neterminálu, což je potřeba při vícenásobném průchodu neterminálů v generovaném řetězci. Poslední číselná proměnná *position* slouží pro porovnávání vstupního a generovaného řetězce. Tato proměnná si pamatuje poslední porovnávaný terminál v generovaném řetězci.

Obrázek 5.10: Třída GeneratedString

5.3 Testování

Při testování aplikace byl kladen důraz na jazyky, které nejsou bezkontextové. Pro testování byly vytvořeny dvě programované gramatiky. Byly vybírány takové testovací vstupy, které pokryjí předpokládané a potenciální chyby. První gramatika generuje jazyk $L(G_1) = \{a^n b^n c^n \mid n \geq 1\}$. Pro dodržení formátu vstupní gramatiky je obsah souboru s gramatikou následující:

```
T: a, b, c

N: S, A, B, C

1: S -> AC, {2}

2: A -> aAb, {3}

3: C -> Cc, {2, 4}

4: A -> e, {5}

5: C -> e, {}
```

V tabulce 5.1 se nachází ukázka množiny testovacích vstupů a očekávaných výstupů pro první gramatiku. V tabulce se rovněž nachází posloupnost použitých pravidel, pokud definovaný vstup nebyl zamítnut.

Vstup	Očekávaný výstup	Výstup	Posloupnost použitých pravidel
(prázdný řetězec)	zamítnut	zamítnut	-
abc	přijat	přijat	1 2 3 4 5
aaabbbcc	zamítnut	zamítnut	-
aabbcc	přijat	přijat	1 2 3 2 3 4 5
aaaabbbbcccc	přijat	přijat	1 2 3 2 3 2 3 2 3 4 5
aaaabbabcccc	zamítnut	zamítnut	-
aaaaaaaaaaaaaaaaa	zamítnut	zamítnut	-
abcabcabcabcabcabcabcabc	zamítnut	zamítnut	-

Tabulka 5.1: Testovací vstupy první gramatiky

Druhá gramatika generuje jazyk $L(G_2) = \{ww \mid w \in \{a, b\}\}$. Druhý soubor s gramatikou vypadá následovně:

```
T: a, b

N: S, A, B

1: S -> AB, {2, 4, 6}

2: A -> aA, {3}

3: B -> aB, {2, 4, 6}

4: A -> bA, {5}

5: B -> bB, {2, 4, 6}

6: A -> e, {7}

7: B -> e, {}
```

Tabulka 5.2 obsahuje očekávané vstupy a výstupy pro druhou gramatiku. Tabulka rovněž obsahuje posloupnost použitých pravidel.

Při testování byly doladěny poslední nedostatky programu. V této fázi je vidět, že aplikace funguje stoprocentně vzhledem k zadaným vstupům. Jelikož vstupy byly vybírány způsobem, aby pokryly co největší množství potenciálních chyb lze předpokládat, že navržený program je zcela funkční. Na obrázku 5.11 je možné vidět implementovanou aplikaci.

5.4 Praktické využití

Na dvou konkrétních příkladech jsem poukázal na to, že aplikace zvládne některé kontextové jazyky. Na základě těchto jazyků je možné vymyslet konstrukce, které nebyly dříve možné. Například na základě již zmiňovaného jazyka $L(G_1) = \{a^n b^n c^n \mid n \geq 1\}$ je možné vytvořit následující konstrukci:

$$x, y, z : int, bool, double : 1, true, 2.3;$$

Přesně tato konstrukce odpovídá jazyku $L(G_x) = \{a^3 b^3 c^3\}$, ale pokud dovolíme, aby počet proměnných v této gramatické konstrukci byl libovolný, pak se dostáváme na kontex-

```
C:\Windows\system32\cmd.exe - ProgrammedGrammars.exe  gramatika2.txt -g

C:\Skola\bakalarka\Program\bin\Release>ProgrammedGrammars.exe gramatika2.txt -g
+-----------------------+
|   POUŽITÁ GRAMATIKA   |
+-----------------------+
T: a, b
N: S, A, B
1: S -> AB, {2, 4, 6}
2: A -> aA, {3}
3: B -> aB, {2, 4, 6}
4: A -> bA, {5}
5: B -> bB, {2, 4, 6}
6: A -> e, {7}
7: B -> e, {}


Zadejte vstupní řetězec:
aaabbaaaabba
Řetězec byl přijat.


Byla použita následující pravidla: 1 2 3 2 3 2 3 4 5 4 5 2 3 6 7.

Generování proběhlo takto:

S
 => AB(1)
 => aAB(2)
 => aAaB(3)
 => aaAaB(2)
 => aaAaaB(3)
 => aaaAaaB(2)
 => aaaAaaaB(3)
 => aaabAaaaB(4)
 => aaabAaaabB(5)
 => aaabbAaaabB(4)
 => aaabbAaaabbB(5)
 => aaabbaAaaabbB(2)
 => aaabbaAaaabbaB(3)
 => aaabbaaaabbaB(6)
 => aaabbaaaabba(7)

Stisknutím libovolného tlačítka ukončíte program.
```

Obrázek 5.11: Výsledná aplikace

Vstup	Očekávaný výstup	Výstup	Posloupnost použitých pravidel
(prázdný řetězec)	přijat	přijat	1 6 7
a	zamítnut	zamítnut	-
aa	přijat	přijat	1 2 3 6 7
ab	zamítnut	zamítnut	-
bab	zamítnut	zamítnut	-
aabba	zamítnut	zamítnut	-
aabbaabb	přijat	přijat	1 2 3 2 3 4 5 4 5 6 7
abaaba	přijat	přijat	1 2 3 4 5 2 3 6 7
babbaababbaa	přijat	přijat	1 4 5 2 3 4 5 4 5 2 3 2 3 6 7
abaaababaaabaa	zamítnut	zamítnut	-
babbbbabbb	přijat	přijat	1 4 5 2 3 4 5 4 5 4 5 6 7
bbabbabb	zamítnut	zamítnut	-
bababbbabaaba	zamítnut	zamítnut	-

Tabulka 5.2: Testovací vstupy druhé gramatiky

tový jazyk $L(G_1)$. Tato jazyková konstrukce je pouhým příkladem. Takovýchto konstrukcí by bylo možné vytvořit mnohem více na základě každého kontextového jazyka, který by algoritmus 1 zvládl.

Kapitola 6

Závěr

Pro překladače běžných programovacích jazyků jsou stěžejní bezkontextové gramatiky. Ty ovšem povolují pouze tvorbu bezkontextových konstrukcí. Bezkontextové gramatiky rozšířené o matematický aparát jsou základem pro regulované jazyky, kterým se věnovala tato práce.

Regulované gramatiky tématicky zapadají do teoretické informatiky a staví na základech formálních jazyků a gramatik. V práci byly uvedeny základní pojmy formálních jazyků, které jsou nutné pro proniknutí do problematiky tohoto tématu. Ovšem i tyto základy se neobejdou bez předchozích znalostí z oblasti matematiky. Matematické znalosti, které jsou pro tuto práci klíčové, se týkají především teorie množin a grafů.

V práci bylo prezentováno šest základních regulovaných gramatik: regulárně řízené gramatiky, maticové gramatiky, programované gramatiky, gramatiky s nahodilým kontextem, gramatiky s rozptýleným kontextem a stavové gramatiky. U každé gramatiky byl uveden základní popis, formální definice a následoval příklad generování řetězce kontextového jazyka $L(G) = \{a^n b^n c^n \mid n \geq 0\}$. Pro každou gramatiku byl uveden matematický aparát, kterým jsou rozšířeny běžné bezkontextové gramatiky. Generativní síla regulovaných gramatik byla popsána hromadně, protože se pro jednotlivé gramatiky neliší.

Tato práce se zabývala tvorbou syntaktického analyzátoru na základě programovaných gramatik. Syntaktická analýza byla popsána z obecného hlediska a následně byla uvedena její modifikace pro programované gramatiky.

Hlavním přínosem této práce je algoritmus, který vychází z modifikace syntaktické analýzy shora dolů. Tento algoritmus zvládne zpracovat nejen bezkontextové jazyky, ale i kontextové. Přesný počet kontextových jazyků, který tento algoritmus zvládne, není znám. Pro tuto záležitost by byla potřeba matematická verifikace.

Zmíněný algoritmus byl implementován do podoby aplikace, která umožní uživatelům zadat vstupní řetězec a gramatiku a na jejich základě zjistit, zda tento vstupní řetězec patří do jazyka generovaného vstupní gramatikou nebo nikoliv. Při vývoji aplikace byly nejprve specifikovány a analyzovány požadavky aplikace. Před vlastní implementací předcházel objektový návrh v jazyce UML. V implementaci bylo rozepsáno rozdělení a účel jednotlivých tříd.

Naprogramovaná aplikace byla testována na množině testovacích vstupů pro dvě kontextové gramatiky. Testováním se dokázalo, že tato aplikace je skutečně schopná zpracovávat gramatiky, které nejsou bezkontextové.

Tento text je možné využít a dále rozšířit pro tvorbu překladačů, které by umožnily vznik nových konstrukcí na základě kontextových jazyků. Tímto směrem se bude ubírat také moje budoucí práce. Nejprve rozšířím vlastní program, aby zvládl libovolné terminály

a neterminály a ne pouze jednopísmenné. Poté se budu snažit vytvořit překladač, který bude schopný překládat jazyky s novými syntaktickými konstrukcemi. Pokud by moje budoucí práce přinesla výsledky, lze předpokládat, že by regulované jazyky nezůstaly pouze v teoretické informatice, ale mohly by se uplatnit i v praxi.

Literatura

[1] Arlow, J.; Neustadt, I.: *UML2 a unifikovaný proces vývoje aplikací*. Computer Press, a. s., 2011, ISBN 978-80-251-1503-9.

[2] Chomsky, N.: Three models for the description of language. In *Information Theory, IRE Transactions on*, ročník 2, IEEE, září 1956, ISSN 0096-1000, s. 113 – 124, doi:10.1109/TIT.1956.1056813.

[3] Chomsky, N.: On certain formal properties of grammars. In *Information and Control*, ročník 2, červen 1959, str. 137–167, doi:10.1016/S0019-9958(59)90362-6.

[4] Gathen, J.; Gerhard, J.: *Modern Computer Algebra*. Cambridge University Press, druhé vydání, 2003.

[5] Meduna, A.: *Automata and Languages: Theory and Applications*. Springer, 2000, ISBN 1-85233-074-0.

[6] Meduna, A.: *Elements of Compiler Design*. Auerbach Publications, 2008, ISBN 978-1-4200-6323-3.

[7] Meduna, A.; Zemek, P.: *Regulated Grammars and Their Transformations*. Brno University of Technology, 2010, ISBN 978-80-214-4203-0.

[8] Meduna, A.; Zemek, P.: *Regulated Grammars and Automata*. Springer, 2014, ISBN 978-1-4939-0368-9.

[9] Microsoft Corporation: C# Keywords. `https://msdn.microsoft.com/en-us/library/x53a06bb.aspx`, [cit. 2015-25-01].

[10] Rozenberg, G.; Salomaa, A.: *Handbook of Formal Languages, vol. 1 through 3*. Springer, 1997.

Příloha A

Obsah CD

Přiložené CD obsahuje:

- zdrojové soubory k písemné zprávě v LaTeXu,
- zdrojové soubory k aplikaci pro VisualStudio 2013,
- text písemné zprávy ve formátu PDF,
- přeloženou aplikaci,
- dva soubory s gramatikami.