VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

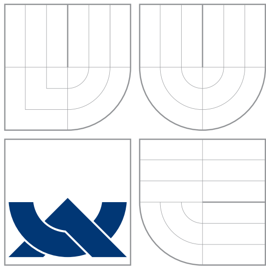
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SYNTAKTICKÁ ANALÝZA NA PROGRAMOVANÝCH GRAMATIKÁCH

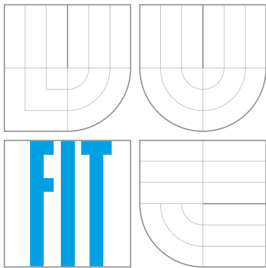
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JAN PAČES
AUTHOR

BRNO 2012



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 NA PROGRAMOVANÝCH GRAMATIKÁCH

PARSING BASED ON PROGRAMMED GRAMMARS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAN PAČES
AUTHOR

VEDOUCÍ PRÁCE — prof. RNDr. ALEXANDER MEDUNA, CSc.
SUPERVISOR

BRNO 2012

Abstrakt
Tato práce se zabývá syntaktickou analýzou na programovaných gramatikách. Modifikuje postupy prediktivní syntaktické analýzy známé pro bezkontextové gramatiky a aplikuje je na gramatiky programované. Studuje síly této metody a zaměřuje se především na některé jazyky, které nejsou bezkontextové.

Abstract
This thesis is researching methods of parsing based on programmed grammars. It modifies known algorithm of table driven non-recursive predictive parser for context-free grammars and applies it on programmed grammars. It studies strength of this method, focusing on some languages which are not context-free.

Klíčová slova
bezkontextová gramatika, programovaná gramatika, syntaktická analýza, LL(k) tabulky

Keywords
context-free grammar, programmed grammar, predictive parsing, LL(k) tables

Citace
Jan Pačes: Syntaktická analýza na programovaných gramatikách, bakalářská práce, Brno, FIT VUT v Brně, 2012

Syntaktická analýza na programovaných gramatikách

Prohlášení

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

....
Jan Pačes
16. května 2012

© Jan Pačes, 2012.
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 4
2.1 Abeceda, slovo a jazyk 4
2.1.1 Abeceda 4
2.1.2 Slovo 4
2.1.3 Délka slova 4
2.1.4 Zřetězení slov 4
2.1.5 Mocnina slova 5
2.1.6 Reverzace řetězce 5
2.1.7 Prefix slova 5
2.1.8 Jazyk 5
2.1.9 Zřetězení jazyků 5
2.2 Gramatika 5
2.3 Chomského hierarchie 6

3 Syntaktická analýza 8
3.1 Postup shora dolů 8
3.2 Postup zdola nahoru 8

4 Řízené gramatiky 9
4.1 Programované gramatiky 9
4.1.1 Programovaná gramatika 10
4.2 Syntaktická analýza na programovaných gramatikách 11
4.2.1 Startovací pravidlo pro programované gramatiky 11

5 Syntaktická analýza hrubou silou 12

6 Prediktivní syntaktická analýza 15
6.1 Model syntaktické analýzy 16
6.2 Množiny $First_k$ a $Follow_k$ 17
6.3 Konstrukce LL(k) tabulek 18
6.4 LL(k) programované gramatiky 19
6.5 Syntaktický analyzátor 20
6.6 Algoritmus syntaktického analyzátoru 22

7 Zkoumání síly algoritmu 25
7.1 Příklady *LL(k)* programovaných gramatik 25

8 Implementace **28**
8.1 Modul pgparser 28
8.2 Modul lexer 28
8.3 Modul parseGrammar 28
8.4 Modul grammar 29
8.5 Modul llparser 29

9 Závěr **30**

A Obsah CD **32**

Kapitola 1

Úvod

V oblasti teorie formálních jazyků, zejména pro překladače, jsou velmi dobře prozkoumány metody pro syntaktickou analýzu bezkontextových jazyků. Většina programovacích jazyků je tak využívá jako prostředek syntaktické analýzy. Celý svět se však zdá být kontextovým [3]. Tento problém se týká především lingvistiky a přirozených lidských jazyků. Ovšem i v oblasti programovacích jazyků by bylo v některých případech vhodné použít silnější nástroj než jsou gramatiky bezkontextové.

Jedním z možných řešení je využít výhod bezkontextových gramatik a jejich definici rozšířit tak, aby jejich generativní síla byla větší. Jedno takové rozšíření navrhl ve článku [5] profesor Rosenkrantz a vytvořil skupinu gramatik, které nazval programované.

Tato práce se zabývá studiem programovaných gramatik a popisuje jejich syntaktickou analýzu. Cílem práce je vytvořit syntaktický analyzátor založený na programovaných gramatikách. Tento analyzátor je navržen tak, aby mohl pracovat i s gramatikami, které generují jazyky, které nejsou bezkontextové. Práce vychází především z článku [7] a snaží se zmíněné postupy přiblížit těm, které se používají u gramatik bezkontextových.

Kapitola 2 se věnuje základním definicím z teorie formálních jazyků týkající se především gramatik a jejich hierarchie.

Kapitola 3 se věnuje úvodu do problematiky syntaktické analýzy.

Kapitola 4 se zaměřuje na princip fungování speciálního typu tzv. řízených gramatik. Do této kategorie spadají i studované programované gramatiky. V této kapitole programované gramatiky definujeme a zkoumáme vlastnosti, na které se musíme zaměřit, pokud na těchto gramatikách chceme zavést postupy syntaktické analýzy.

Kapitola 5 a 6 se zaměřuje na modifikaci postupů syntaktické analýzy právě pro programované gramatiky. Především část 6 je pro celou práci nejdůležitější a popisuje konstrukci prediktivního syntaktického analyzátoru. Síla tohoto algoritmu je diskutována v kapitole 7, kde jsou uvedeny příklady gramatik, se kterými je schopen analyzátor pracovat.

Kapitola 8 popisuje implementaci navrženého syntaktické analyzátoru.

Finální kapitola 9 pak shrnuje získané závěry a navrhuje možnosti dalšího rozšíření.

Kapitola 2

Základní pojmy

Tato kapitola se věnuje základní teorii v oblasti formálních jazyků a pojmů s ní spojených. Shrnuje a vysvětluje definice, které jsou využívány ve zbytku práce. Všechny uvedené poznatky jsou čerpány z [2] a především z [4].

2.1 Abeceda, slovo a jazyk

2.1.1 Abeceda

Definice 2.1.1. *Abeceda* V je konečná neprázdná množina, jejíchž prvky se nazývají *symboly*. Posloupnost symbolů pak tvoří slovo. *Prázdné slovo*, označujeme ε, je slovo, které neobsahuje žádné symboly.

2.1.2 Slovo

Definice 2.1.2. Nechť V je abeceda.

- ε je *slovo* nad V (též označujeme jako řetězec)
- Pokud je x slovo nad V a $a \in V$, pak xa je slovo nad V

2.1.3 Délka slova

Definice 2.1.3. Nechť x je slovo nad V. *Délka slova* x, $|x|$ je definována:

- pokud $x = \varepsilon$ pak $|x| = 0$
- pokud $x = a_1 \dots a_n$ pro $n \geq 1$ a $a_i \in V$ pro všechna $i = 1 \dots n$ pak $|x| = n$

2.1.4 Zřetězení slov

Definice 2.1.4. Zřetězení slov Nechť x a y jsou dvě slova nad abecedou emphV. Pak xy je *zřetězením slov* x a y. Pro každé slovo x platí:

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

2.1.5 Mocnina slova

Definice 2.1.5. Nechť x je slovo nad abecedou V. Pro $i \geq 0$, *i-tá mocnina slova* x,x^i je rekurzivně definována:

1. $x^0 = \varepsilon$
2. pro $i \geq 1$: $x^i = xx^{-1}$

2.1.6 Reverzace řetězce

Definice 2.1.6. Nechť x je slovo nad abecedou V. *Reverzace řetězce x* $reversal(x)$ je definována:

- pokud $x = \varepsilon$, pak $reversal(\varepsilon) = \varepsilon$
- pokud $x = a_1 \ldots a_n$ pro $n \geq 1$ a $a_i \in V$ pro $i = 1, \ldots, n$ pak $reversal(a_1 \ldots a_n) = a_n \ldots a_1$

2.1.7 Prefix slova

Definice 2.1.7. Nechť x a y jsou slova nad abecedou V. Pak x je *prefix* y pokud existuje slovo z nad V, přičemž platí $xz = y$.

2.1.8 Jazyk

Definice 2.1.8. Nechť V je abeceda. V^* značí množinu všech slov na V. Množina $V^+ = V^* - \{\varepsilon\}$. *Jazyk* L na abecedou V je $L \subseteq V^*$

2.1.9 Zřetězení jazyků

Definice 2.1.9. Nechť L_1 a L_2 jsou dva jazyky. *Zřetězení* jazyků L_1L_2 je definováno:

$$L_1L_2 = \{xy|x \in L_1 ay \in L_2\}$$

2.2 Gramatika

V teorii formálních jazyků definujme gramatiku jako strukturu, která umožňuje popis těchto jazyků. Gramatika se skládá z abecedy symbolů a množiny pravidel. Jedná se o generativní prostředek. Na základě pravidel určuje, jaké tvořit řetězce tak, aby jejich syntax odpovídala gramatice. Její název vznikl na základě podobnosti s gramatikou tak jak je známa pro přirozené jazyky.

O jeden z největších přínosů pro oblast teorie formálních jazyků se zasloužil americký lingvista Noam Chomsky. Ve své práci z roku 1956 formálně definoval gramatiku, a tím popsal mechanismus pro generování formálních jazyků.

Definice 2.2.1. Obecná generativní gramatika je čtveřice

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

kde:

- N je abeceda neterminálů

- T je abeceda terminálů a $N \cup T = \emptyset$
- $P \subset (N \cup T)^* N (N \cup T) \times (N \cup T)^*$
- $S \in N$ se nazývá počáteční symbol

Dvojice $(x, y) \subset P$ nazýváme přepisující pravidla a obvykle zapisujeme ve tvaru $r : x \to y$. Pak r značí navěští pravidla, x, y označujeme jako pravou, respektive levou stranu pravidla. Množina P je pak množina všech pravidel nad G. Přímá derivace nad G je binární relace dvou slov $v \in (N \cup T)^+, (N \cup T)^+$ značena $\Rightarrow$. Řekneme, že v přímo derivuje w podle pravidla r, což zapisujeme

$$v \to w[r]$$

právě tehdy, pokud existují slova $x_1, x_2, y, z \in (N \cup T)^*$ tak, že $u = x_1 y x_2, w = x_1 z x_2$ a $r : y \to z \in P$. Řekneme, že slovo v derivuje w a označíme $w \Rightarrow_* v$, právě když existuje posloupnost $v_1 \dots v_n$, kde $n \geq 1$ slov nad $(N \cup T)$ tak, že $v = v_1 \Rightarrow v_2 \Rightarrow \dots \Rightarrow v_n = w$.

Definice 2.2.2. Jazyk generovaný G označujeme jako $L(G)$ a definujeme:

$$L(G) = \{u : u \in T^*, S \Rightarrow^* u\}$$

Definice 2.2.3. *Větná forma* gramatiky G je řetězec $w \in (N \cup T)^*$

2.3 Chomského hierarchie

Na základě omezení kladených na tvar pravidel gramatik redukujeme jejich generativní sílu. Vzniká tak několik typů gramatik, které pak tvoří Chomského hierarchii jazyků. Pojem derivace a přímá derivace je definován pro všechny gramatiky stejně jako u obecné generativní gramatiky.

Definice 2.3.1. Neomezená gramatika (gramatika typu 0), stejně jako u obecné gramatiky, je čtveřice:

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

kde na tvar pravidel není kladeno žádné omezení.

Množina jazyků generovaná neomezenými gramatikami se nazývá rekurzivně vyčíslitelné jazyky a jsou označovány **jazyky typu 0**.

Definice 2.3.2. Kontextová gramatika (gramatika typu 1), stejně jako u obecné gramatiky, je čtveřice

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

kde pro každé pravidlo $r : u \to v$ musí platit $|u| < |v|$. Tato podmínka nemusí platit u pravidla $r : S \to \varepsilon$, v tom případě se však symbol S nesmí vyskytovat na pravé straně žádného dalšího pravidla.

Množina jazyků generovaná kontextovými gramatikami se nazývá kontextové jazyky a jsou označovány **jazyky typu 1**.

Definice 2.3.3. Bezkontextová gramatika (gramatika typu 3), stejně jako u typu 0, je čtveřice:

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

kde pro každé pravidlo ve tvaru $r : u \to w \in P$ musí platit $u \in N$ a $|u| = 1$

Množina jazyků generována bezkontextovými gramatikami se nazývá bezkontextové jazyky a jsou označovány **jazyky typu 2**.

Definice 2.3.4. Pravá lineární gramatika (gramatika typu 3), stejně jako u typu 0, je čtveřice:

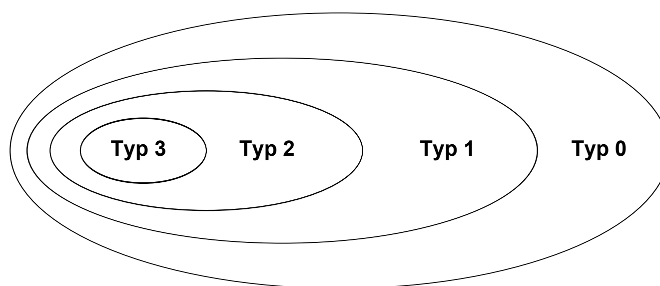
$$G = (N, T, P, S)$$

kde pro každé pravidlo ve tvaru $r : u \to w \in P$ musí platit $u \in N$ a a $w \in T^* \cup T^*N$

Množina jazyků generována pravými lineárními gramatikami se nazývá regulární jazyky a jsou označovány **jazyky typu 3**.

Pro výše uvedené množiny jazyků pak platí věta o Chomského hierarchii [4]:

$$\textbf{Typ 3} \subset \textbf{Typ 2} \subset \textbf{Typ 1} \subset \textbf{Typ 0}$$



Obrázek 2.1: Chomského hierarchie jazyků

Kapitola 3

Syntaktická analýza

Syntaktická analýza je proces, který má za úkol ze vstupní posloupnosti prvků (např. tokenů nebo slov) určit, zda-li posloupnost odpovídá syntaxi jazyka určeného gramatikou. Velmi zjednodušeně můžeme říct, že kontrolujeme předem danou správnost pořadí prvků v posloupnosti.

Prostředek pro syntaktickou analýzu se nazývá syntaktický analyzátor. V oblasti překladačů je ve velkém množství případů analyzátor *srdcem* celého překladače. Úzce spolupracuje s lexikálním analyzátorem a jeho výstupem je datová struktura nazývaná derivační strom.

Tento strom můžeme chápat jako acyklický graf reprezentující syntaktickou strukturu vstupní posloupnosti podle gramatiky. Jednotlivé uzly jsou reprezentovány neterminály gramatiky a listové uzly terminály.

Syntaktickou analýzu rozdělujeme na dva základní typy podle směru tvorby derivačního stromu. Jsou to metody shora dolů a zdola nahoru. Obě tyto metody načítají vstupní posloupnost zleva doprava.

3.1 Postup shora dolů

Při tomto postupu začínáme od kořene derivačního stromu, počátečního neterminálu gramatiky. Postupnou aplikací pravidel z gramatik na neterminály se snažíme dosáhnout posloupnosti terminálních symbolů. V případě úspěchu syntaktické analýzy pak dostáváme posloupnost pravidel nejlevější derivace.

Mezi techniky využívající tento postup se řadí metoda hrubé síly, rekurzivního zanoření a LL analýza. Syntaktické analyzátory využívající metodu shora dolů jsou jednoduše implementovatelné ručně.

3.2 Postup zdola nahoru

Při tomto postupu tvoříme derivační strom v opačném pořadí, začínáme od listů terminálů a aplikací pravidel v obráceném pořadí se snažíme dosáhnout kořenu počátečního neterminálu. V případě úspěchu pak dostáváme posloupnost pravidel nejpravější derivace.

Mezi typické metody využívající postup zdola nahoru patří např. precedenční syntaktický analyzátor nebo LR analyzátor.

Konstrukce takovýchto analyzátorů je poměrně složitá a využívá se k ní automatických prostředků.

Kapitola 4

Řízené gramatiky

Následující myšlenky jsou čerpány především z [6].

Ve většině případů se pro účely syntaktické analýzy využívají bezkontextové gramatiky. Jsou velmi dobře prostudované a je pro ně vytvořeno velké množství algoritmů, na kterých je možné založit syntaktickou analýzu.

Jejich velkou výhodou je především jednoduchá konstrukce těchto gramatik. Mezi jednu z nevýhod patří omezená generativní síla. Z pohledu programovaných jazyků se ve většině případů však jeví jako dostačující.

V některých situacích by ale bylo vhodné používat i jazyky s vyšší vyjadřovací silou než jsou bezkontextové. A to nejen z pohledu pro programovací jazyky, ale i pro ty přirozené. Přirozeně se nabízí používat gramatiky kontextové. Většina lingvistů se však shoduje, že tento nástroj má vyjadřovací sílu až příliš velkou. Kontextové gramatiky navíc obsahují několik problémů. Známé algoritmy řešící problém členství mají exponenciální časovou složitost a problém prázdnosti je nerozhodnutelný. Díky složitosti kontextových pravidel je také velmi těžké určit, jaký je generovaný jazyk.

Proto se zdá vhodné použít bezkontextové gramatiky a upravit jejich pravidla tak, aby obsahovala mechanismy, které by umožňovaly zachování *kontextu* a zvýšily tak jejich generativní sílu na podmnožinu kontextových jazyků.

Jedním z těchto mechanismů je v každém kroku derivace omezit množinu pravidel, podle kterých může být provedena derivace další. Omezování pravidel tak můžeme chápat jako řízení derivace a odtud vzniká pojem *řízené gramatiky*.

Řízené gramatiky rozdělujeme na několik základních skupin na základě mechanismů, které řídí derivaci. Některé pro úplnost vyjmenujeme. Jsou to gramatiky s předepsanými sekvencemi, gramatiky s kontextovými podmínkami a gramatiky s částečným paralelismem.

Tato práce se zabývá podtypem gramatik s předepsanými sekvencemi a to programovanými gramatikami. V případě zájmu odkazujeme na literaturu [6] a [3], kde jsou řízené gramatiky velmi podrobně rozebrány.

4.1 Programované gramatiky

Programované gramatiky byly zavedeny profesorem D.J. Rosenkrantzem ve článku [5]. Jedná se o gramatiky ze skupiny řízených s předepsanými sekvencemi.

Ve své práci [5] profesor Rosenkrantz navrhl programované gramatiky založeny na různých typech gramatik a studoval jejich generativní sílu. V této práci se zaměříme na programované gramatiky, jejichž jádrem je bezkontextová gramatika. Bezkontextová pravidla

jsou rozšířena o podmnožiny pravidel. Při derivaci je další pravidlo vybíráno právě z těchto podmnožin a ne ze všech pravidel gramatiky, tak jak je tomu u gramatik bezkontextových. Dalo by se říci, že tento způsob vybírání pravidel částečné simuluje udržení kontextu a tak zvyšuje jejich generativní sílu.

Existují dva typy bezkontextových programovaných gramatik. Pokud pravidla obsahují právě jednu množinu, nazýváme je programované gramatiky *bez kontroly na výskyt*. Pokud obsahují pravidla množiny dvě, hovoříme o gramatikách *s kontrolou na výskyt*.

Podmnožiny určující výběr pravidel jsou nazývány *pole úspěchu* a *pole neúspěchu*. Proces derivace funguje následovně. Na základě aktuálního pravidla a jeho množiny úspěchu, resp. neúspěchu vybíráme další pravidlo pro derivaci. Pokud v průběhu derivace lze aktuálně vybrané pravidlo aplikovat na rozvíjenou větnou formu, pak je vybíráno z *pole úspěchu*. V případě, že pravidlo aplikovat nelze, neterminál na pravé straně pravidla se ve větné formě nevyskytuje, tak vybíráme z *pole neúspěchu*. Právě tento mechanismus nazýváme *kontrola na výskyt*. Uvedeme nyní definici bezkontextové programované gramatiky.

4.1.1 Programovaná gramatika

Definice 4.1.1. Programovaná gramatika(viz [6]) je čtveřice:

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

kde:

- N, T a S jsou definovány stejně jak u bezkontextové gramatiky
- P je konečná množina trojic $r = (p, \sigma, \varphi)$, kde p je bezkontextové pravidlo a σ a φ jsou podmnožiny P
- Pokud $r = (p, \sigma, \emptyset)$ platí pro každé $r \in P$, pak gramatiku PG nazýváme programovanou gramatikou bez kontroly výskytu. V opačném případě PG je programovaná gramatika s kontrolou na výskyt.
- Pokud $r = (p, \sigma, \vartheta)$, pak je σ nazýváno pole úspěchu a φ pole neúspěchu.

Definice 4.1.2. Jazyk $L(PG)$,[6], generovaný gramatikou PG je definován jako množina všech slov $w \in T^*$ takových, kde existuje derivace

$$S = w_0 \Longrightarrow_{r_1} w_1 \Longrightarrow_{r_2} w_2 \Longrightarrow_{r_3} w_3 \ldots \Longrightarrow_{r_k} = w,$$

$k \geq 1$ a existuje $r_{k+1} \in P$ takové, kde pro $r_i = (A_i \to v_i, \sigma_i, \varphi_i)$, $1 \leq i \leq k$ a platí jedna z následujících podmínek:

$$w_{i-1} = w'_{i-1} A_i w"_{i-1}, w_i = w'_{i-1} v_i w" i - 1, \text{ pro nějaké } w'_{i-1}, w"_{i-1} \in V^*_{PG} \text{ a } r_{i+1} \in \sigma_i,$$

nebo

$$A_i \text{ se nevyskytuje v } w_{i-1} = w_i \text{ a } r_{r+1} \in \sigma_i$$

Uvedeme příklad programované gramatiky s testováním na výskyt:

Příklad 4.1.1. Nechť $G_1 = (\{S, S'\}, \{a\}, P, S)$,[6] je programovaná gramatika s kontrolou na výskyt, kde:

$$\begin{array}{l}(r_1 : S \rightarrow SS', \{r_1\}, \{r2\}),\\(r_2 : S' \rightarrow S\{r_2\}, \{r_1, r_3\}),\\(r_3 : S \rightarrow a\{r_3\}, \emptyset)\end{array}$$

Tato gramatika generuje jazyk $L(G_1) = \{a^{2^n} | n \geq 1\}$

4.2 Syntaktická analýza na programovaných gramatikách

Původně navržený model programovaných gramatik má velkou vyjadřovací sílu. Podle [5], v případě že povolíme vymazávací pravidla, jsme schopni generovat jazyky patřící do třídy rekurzivně vyčíslitelných. Hlavní požadavek na syntaktický analyzátor pro použití v překladačích je, aby byl deterministický. Sestrojit deterministický analyzátor pro celou třídu jazyků vymezenou programovanými gramatikami je úkol velmi složitý. V této práci tedy vymazávací pravidla nepovolíme.

Následující kapitoly se tedy věnují hledání vhodného způsobu syntaktické analýzy na programovaných gramatikách tak, aby byl deterministický. Zkoumám již zavedené metody nad bezkontextovými gramatikami a zkouším je aplikovat na gramatiky programové. Zaměřuji se na postupy shora - dolů.

V prvé řadě je nutné se zamyslet, v jaké části syntaktické analýzy dochází k rozhodování, jaké další pravidlo bude použito. V případě bezkontextových gramatik je to pokaždé, pokud je aktuálně rozvíjený neterminál obsažen v pravidlech gramatiky na pravé straně více než jednou.

Tato situace je však u programovaných gramatik odlišná. Pravidla nejsou vybírána z celé množiny pravidel, ale jen z její omezené části, jak bylo zmíněno dříve. K procesu rozhodování dochází teprve v případě, pokud vybíráme z pole úspěchu či neúspěchu, které obsahuje více než jedno pravidlo. Na tato pravidla se tedy při zkoumání musíme zaměřit a navrhnout adekvátní rozhodovací mechanismus. Tento druh pravidel tedy ve zbytku práce budeme označovat jako *rozhodovací pravidla*.

Syntaktická analýza není řízena podle právě rozvíjeného neterminálu jako u bezkontextových gramatik, ale stav analýzy je určen aktuálně používaným pravidlem. Z tohoto důvodu není možné, aby se počáteční neterminál vyskytoval na pravé straně více než jednoho pravidla. Pokud tato situaci nastane, lze ji řešit zavedením speciálního startovacího pravidla. Definujeme jej takto:

4.2.1 Startovací pravidlo pro programované gramatiky

Definice 4.2.1. Startovací pravidlo pro PG Nechť $PG = (N, T, P, S)$ je programovaná gramatika a $(r, \sigma, \varphi) \in P$, kde $r : X \rightarrow Y$. Pokud existuje více pravidel, kde $X = S$, pak vytvoříme nové startovací pravidlo ve tvaru $(r : S' \rightarrow S, \varsigma, \emptyset)$. Do množiny úspěchu ς nového startovacího pravidla přiřadíme všechna pravidla, která na pravé straně obsahují neterminál S. Počáteční neterminál gramatiky změníme na S'.

V následujících částech rozebereme dvě metody, které přišly v úvahu - analýzu hrubou silou a nerekurzivní prediktivní analýzu.

Kapitola 5

Syntaktická analýza hrubou silou

První metodou, která přišla v úvahu, je projít určitou podmnožinu slov vygenerovaných programovanou gramatikou a na tomto základě rozhodnout, zda-li vstupní slovo patří do jazyka generovaného gramatikou. I přesto, že se jedná o metodu velmi náročnou z hlediska časové i prostorové složitosti, je její vyjadřovací síla poměrně velká, proto zde algoritmus zmiňujeme.

Problém: Mějme programovanou gramatiku $PG = (N, T, P, S)$ a řetězec $v \in T^+$. Zkusme rozhodnout, zda-li řetězec patři do množiny slov generovaných PG.

Zvolíme metodu prohledávání stavového prostoru do hloubky, protože tato metoda klade menší nároky na paměť. Stavový prostor je určen podmnožinou derivačních stromů, které mohou vzniknout postupnou derivací od počátečního neterminálu. Následující algoritmus tedy generuje tuto podmnožinu derivačních stromů a snaží se najít ten, odpovídající vstupnímu řetězci. Při průchodu stavového prostoru z něj vyřazuje ty stromy, které slovo negenerují, aby se předcházelo prohledávání zbytečných stavů. V případě nalezení stromu je pak na výstupu posloupnost pravidel, podle kterých byl vstupní řetězec vygenerován.

Pro fungování algoritmu potřebujeme abstraktní datový typ, zásobník, ve kterém budou uchovávány všechny momentálně zkoumané derivační stromy. Jeden prvek v zásobníku bude obsahovat vygenerovanou větnou formu, posloupnost pravidel, podle které byla tato forma vygenerována, a pravidlo, které má být použito. Následuje slovní popis algoritmu.

Mějme stav ve formě trojice $(R, (r, \sigma, \varphi), \alpha)$. Kde $R = r_1 \ldots r_n$ je posloupnost zatím použitých pravidel, r pravidlo, které má být aktuálně použito a α je vygenerovaná větná forma.

- Na začátku na zásobník vložíme stav (r_1, r_1, S), kde r_1 je startovací pravidlo a S počáteční neterminál.

- Vyjmeme ze zásobníku první stav $(R, (r, \sigma, \varphi), \alpha)$.

- Pokud r lze aplikovat na α, pak jej aplikujeme a vznikne nová větná forma α_{new}. Přeřadíme r do R. Když α_{new} odpovídá vstupnímu řetězci, tak analýza končí úspěchem a vrací posloupnost pravidel R. V opačném případě porovnáme prefix α_{new} ohraničený nejlevějším neterminálem a prefix vstupního slova ve stejné délce. Jestliže jsou tyto prefixy shodné, pak vytvoříme nové stavy $(R, r_{sigma}, \alpha_new)$ pro všechna $r_\sigma \in \sigma$ a ty vložíme na zásobník. Pokud prefixy shodné nejsou, pak stav dále nevyšetřujeme.

- Pokud pravidlo aplikovat nelze, vytvoříme nové stavy (R, r_φ, α) pro každé $r_\varphi \in \varphi$ a ty vložíme na zásobník.

- Předchozí dva body zajišťují fungování *kontroly na výskyt*.
- Tyto kroky neustále opakujeme do té doby, dokud nenajdeme stav odpovídající vstupnímu slovu. V případě, že je zásobník prázdný, pak gramatika vstupní slovo negeneruje a algoritmus končí.

Uvažujme následující gramatiku:

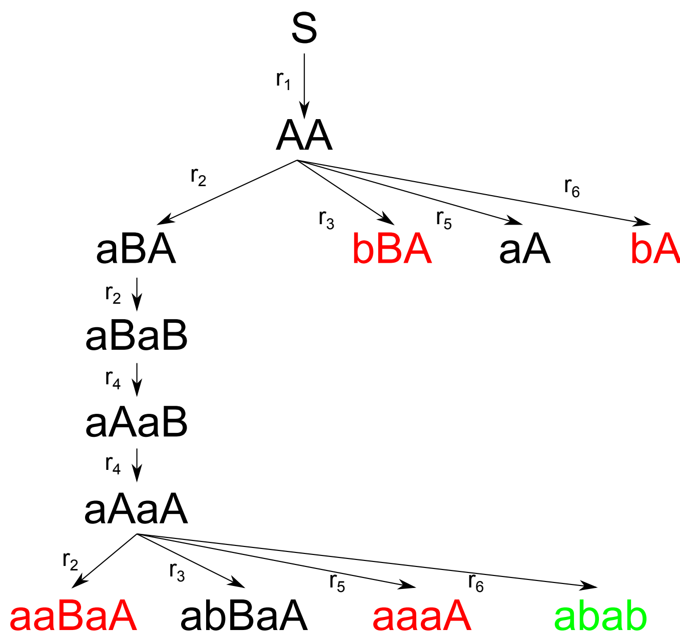
Příklad 5.0.1. Nechť $G = (\{S, A, B\}, \{a, b\}, P, S)$ je programovaná gramatika s kontrolou na výskyt, kde:

$$\begin{array}{l} (r_1 : S \rightarrow AA, \{r_2, r_3, r_5, r_6\}, \emptyset), \\ (r_2 : A \rightarrow aB\{r_2\}, \{r_4\}), \\ (r_3 : A \rightarrow bB\{r_3\}, \{r_4\}), \\ (r_4 : A \rightarrow A\{r_4\}, \{r_2, r_3, r_4, r_5, r_6\}), \\ (r_5 : A \rightarrow a\{r_5\}, \emptyset), \\ (r_6 : A \rightarrow b\{r_6\}, \emptyset), \end{array}$$

Tato gramatika generuje jazyk $L = \{xx|x \in (a, b)^+\}$.

Řetězec *abab* je generován podle následujících pravidel: $S \Rightarrow AA[r_1] \Rightarrow aBA[r_2] \Rightarrow aBaB[r_2] \Rightarrow aAaB[r_4] \Rightarrow aAaA[r_4] \Rightarrow abaA[r_6] \Rightarrow abab[r_6]$.

Popis fungování algoritmu je ilustrován na následujícím obrázku:



Obrázek 5.1: Princip algoritmu obecné analýzy

Hrany jsou označeny pravidlem, které bylo použito. Stavy jsou v grafu zjednodušeně označeny pouze aktuální vygenerovanou větnou formou. Červené stavy by v případě pokračování algoritmu byly z prohledávání vyřazeny. Zelený stav značí hledané slovo.

Navržený algoritmus je poměrně silný a dokáže rozhodovat pro velké množství programovaných gramatik včetně těch, které generují jazyky, které nejsou bezkontextové. Je s ním

však spojeno několik výrazných negativ. Hlavním negativem je jeho náročnost, a to jak časová tak i prostorová. Všeobecně může obsahovat velké množství návratů a také potřebuje udržet informaci, jaká část vstupního řetězce již byla zpracována. Návaznost na lexikální analyzátor by byla obtížná. Také v případě rekurzivních pravidel v gramatice může dojít k zacyklení algoritmu. Pro uvedení do praxe v kompilátorech je důležitým faktorem především časová složitost, a proto tento postup nebude dál rozvíjen a byl uveden pouze pro úplnost.

Kapitola 6

Prediktivní syntaktická analýza

Tato kapitola je pro celou práci nejdůležitější. Popisujeme v ní proces nerekurzivní prediktivní syntaktické analýzy na programovaných gramatikách. V [7] byl zaveden postup pro syntaktickou analýzu nad programovanými gramatikami, který je řízen tabulkami tak jako pro bezkontextové *LL(k)* gramatiky. Modifikujeme tyto postupy tak tak aby v nich byly používány rozhodovací $LL(k)$ tabulky, tak jak je známe pro bezkontextové gramatiky.

Kombinuji tedy metody z [1] a [7] a vytvářím postup prediktivní analýzy pro programované gramatiky. Tato metoda se podobá prediktivní syntaktické analýze pro bezkontextové gramatiky, která zde ovšem nebude uvedena, a rovnou zde představíme její modifikaci pro programované gramatiky.

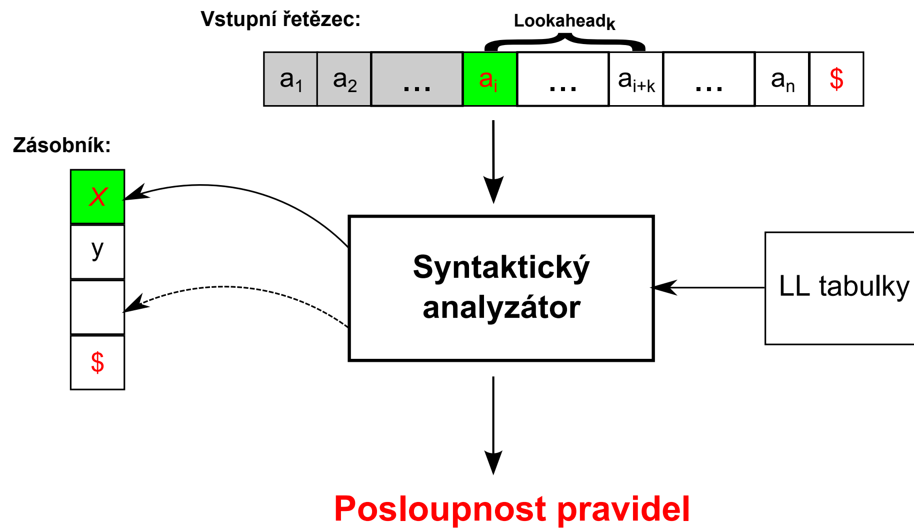
Tato kapitola tedy vysvětluje konstrukci rozhodovacích tabulek a jednotlivých částí nutných pro fungování syntaktického analyzátoru jako celku. Také vymezuje podtřídu programovaných gramatik, na kterou lze postup použít. V závěru je popsán samotný algoritmus řídící syntaktickou analýzu.

Jak již bylo zmíněno, syntaktická analýza má za úkol rozhodnout, zda-li řetězec patří do množiny jazyka generovaného gramatikou. Nejdříve musíme vysvětlit, na základě čeho se prediktivní syntaktická analýza rozhoduje.

V případě, že se analýza dostane do stavu, kdy je možno použít více pravidel, rozhoduje se na základě podřetězce délky k zatím nezpracované části vstupního řetězce. Tento podřetězec budeme nazývat $lookahead_k$. Podle aktuálně používaného pravidla a $lookahead_k$ nahlédne do rozhodovací *LL(k)* tabulky a určí, které pravidlo má být použito jako další. V ideálních případech je velikost `k` rovna jedné. Pro některé gramatiky, které chceme zkoumat, délka jedna ovšem nestačí. Proto zde uvažujeme i postupy, které umí pracovat se situacemi, kde velikost k je větší než jedna.

V následující části textu celý proces rozebereme podrobněji. Nejdříve se zaměříme na celkový model a pak vysvětlíme funkci jeho jednotlivých částí.

6.1 Model syntaktické analýzy



Obrázek 6.1: Model syntaktické analýzy

Model je velmi obdobný jako ten pro bezkontextové gramatiky. Pro fungování nad programovanými gramatikami jej však bylo nutno pozměnit. Model se skládá ze:

- Vstupního řetězce
- Zásobníku
- Rozhodovacích *LL* Tabulek
- Syntaktického analyzátoru

Popišme nyní funkce jednotlivých částí. Zásobník obsahuje aktuálně rozvíjenou větnou formu. Jeho dno obsahuje speciální ukončovací symbol $. V podstatě se jedná o paměť syntaktického analyzátoru. V případě programovaných gramatik ale musíme uvažovat speciální typ zásobníku. V některých fázích analýzy může docházet i k expanzi neterminálu, který se nenachází na vrcholu zásobníku. Tento zásobník tedy nazveme hluboký zásobník. Jeho principy vysvětlíme později.

Vstupní řetězec je posloupnost tokenů reprezentující aktuálně zpracovávané slovo. Nad touto posloupností je speciální ukazatel `lookahead` délky k, který určuje část vstupního řetězce pro predikci. Vstupní řetězec je též zakončen symbolem $.

Rozhodovací tabulky pak představují reprezentaci gramatiky, podle které syntaktickou analýzu provádíme. V případě programovaných gramatik bez kontroly na výskyt potřebujeme tabulku pouze jednu. Pokud gramatika obsahuje mechanismus kontroly výskytu, jsou zapotřebí tabulky dvě. Jednu pro pole úspěchu a druhou pro pole neúspěchu.

Nejdůležitější část celého modelu je syntaktický analyzátor. Celý proces řídí a zajišťuje komunikaci mezi ostatními komponenty. Abychom mohli detailně popsat činnost syntaktického analyzátoru, nejdříve musíme popsat tvorbu rozhodovacích tabulek.

Pro konstrukci tabulky jsou zapotřebí speciální množiny, jejichž definice a algoritmy pro jejich tvorbu uvedeme v následující části.

6.2 Množiny $First_k$ a $Follow_k$

Definice množin $First_k$ a $Follow_k$ je pro programované gramatiky totožná jako u bezkontextových gramatik. Z trojice $(r, \sigma, \varphi) \in P$ používáme pouze pravidlo r ve tvaru $r : A \rightarrow X$. Tyto množiny jsou potřebné pro konstrukci rozhodovacích tabulek nad programovanými gramatikami. Definice jsou čerpány z [2].

Pro potřeby následujících definic a algoritmů zavedeme operátor $\oplus_k$ pro zřetězení množin slov do délky k.

Definice 6.2.1. Nechť V je abeceda. Pokud L_1 a L_2 jsou podmnožiny V^*, pak:
$L_1 \oplus_k L_2 = \{w|$ pro nějaké $x \in L_1$ a $y \in L_1$, máme $w = xy$ pokud $|xy| \leq k$ a v opačném případě je w prvních k symbolů z $xy\}$

Pro gramatiku $G = (N, T, P, S)$ je $First_k(\alpha)$, kde $\alpha \in N \cup T$, množina všech prefixů složených z terminálů délky k, kterými může začínat řetězec derivovaný z α.

Definice 6.2.2. Množina $First_k$

$$First_k(\alpha) = \{x | \alpha \Rightarrow^* x\beta \text{ a } |x| = k \text{ nebo } \alpha \Rightarrow^* x \text{ a } |x| < k\}$$

Algoritmus pro výpočet $First_k$ pro všechny neterminály a terminály gramatiky je následující:

Algoritmus 1: Výpočet množiny $First_k$

Vstup : Gramatika $G = (N, T, P, S)$ a $k \geq 1$
Výstup: Množina $First_k(X)$ pro každé $X \in N \cup T$

```
foreach a ∈ T do
    F(a) ← a
foreach A ∈ N do
    F(A) ← ∅
    if A ⟶ ε ∈ P then
        F(A) ← ε
repeat
    foreach A ∈ N do
        F'(A) ← F(A)
    foreach a ∈ T do
        F'(a) ← F(a)
    foreach A ⟶ u1u2 ... un ∈ P kde n > 0 do
        F(A) ← F(A) ∪ F'(u1) ⊕k F'(u2) ⊕k ... F'(un)
until F(A) = F'(A) pro každé A ∈ V
for A ∈ N ∪ T do
    Firstk(A) ← F(A)
```

Dále je potřeba zavést algoritmus $First_k$, pro řetězec složený z posloupnosti terminálů a neterminálů. Tedy $First_k(X_1, X_2 \dots X_3)$ kde $X_1, X_2 \dots X_3 \in (N \cup T)^+$. Algoritmus ve zkrácené podobě můžeme zapsat následovně:

$$First_k(X_1, X_2 \dots X_3) = First_k(X_1) \oplus_k First_k(X_2) \oplus_k \dots First_k(X_3)$$

Množina $Follow_k(A)$ (kde $A \in N$) obsahuje taková slova nad terminály T délky maximálně k, které se mohou vygenerovat vpravo od A.

Definice 6.2.3. Množina $Follow_k$

$$Follow_k(A) = \{x|S \Rightarrow^* uA\alpha, x \in First_k(\alpha), x, \alpha \in (N \cup T)^*\}$$

Algoritmus pro konstrukci $Follow_k$ pro všechny neterminály gramatiky je následující:

Algoritmus 2: Výpočet množiny $Follow_k$

Vstup : Gramatika $G = (N, T, P, S)$, množina $FIRST_k$ pro každé $A \in N$ a $k \geq 1$
Výstup: Množina $Follow_k(A)$ pro každé $A \in N$

```
FL(S) ← $
foreach A ∈ N − {S} do
  FL(A) ← ∅
repeat
  foreach A ∈ N do
    FL′(A) ← FL(A)
  foreach A ⟶ u1u2 . . . un ∈ P do
    if un ∈ N then
      FL(un) ← FL(un) ∪ FL′(A)
  for i ← n − 1 downto 1 do
    L ← FIRSTk(ui+1)⊕kL
    if ui ∈ N then
      FL(ui) ← FL(ui) ∪ L
until FL(A) = FL′(A) pro každé A ∈ V
for A ∈ N do
  Followk(A) ← FL(A)
```

Algoritmy 1 a 2 jsou s úpravami převzaty z [8].

Nyní, když ji známe postup pro tvorbu těchto množin, můžeme přistoupit k samotné konstrukci *LL(k)* tabulek.

6.3 Konstrukce LL(k) tabulek

Rozhodovací tabulka by se dala označit jako reprezentace gramatiky a na jejím základě je syntaktický analyzátor schopen rozhodnout, které pravidlo má být použito jako další. V případě bezkontextových gramatik jsou řádky tabulky označeny neterminály a sloupce terminály. Jednotlivé buňky pak obsahují pravidlo, které může být vybráno. Když syntaktický analyzátor narazí na vrcholu zásobníku na neterminál, vybírá pravidlo na průsečíku řádku neterminálu a sloupce aktuálního tokenu. Toto však v případě programovaných gramatik není možné, protože stav syntaktické analýzy mimo jiné závisí na naposledy použitém pravidle. Proto jsou řádky tabulky označeny pravidly a sloupce neterminály. V případě, když pole úspěchu, resp. neúspěchu obsahuje pouze jediné pravidlo, není se třeba rozhodovat. Z toho důvodu rozhodovací tabulky obsahují řádky pouze pro pravidla, jejichž pole úspěchu resp. neúspěchu obsahují pravidel více.

Nyní představíme postup pro konstrukci $LL(k)$ tabulek pro programované gramatiky. Nejdříve však musíme zavést poslední množinu $Predict_k(X \to a)$, která obsahuje řetězec terminálních symbolů délky k, který může být aktuálně nejlevěji vygenerován, pokud na libovolnou větnou formu použijeme pravidlo $X \to a$.

Definice 6.3.1. Nechť $PG = (N, T, P, S)$ je programovaná gramatika. Pro každé $(r, \sigma, \varphi) \in P$, kde $r : X \to a$ definujeme množinu $Predict_k$ jako:

$$Predict_k(X \to a) = First_k(First_k(a)Follow_k(X))$$

Algoritmus vypadá následovně:

Algoritmus 3: Konstrukce rozhodovacích LL(k) tabulek

Vstup : Programovaná gramatika $PG = (N, T, P, S)$
Výstup: Rozhodovací LL tabulky LL_{uspech} a $LL_{neuspech}$

```
pro všechna pravidla z PG
foreach (r, σ, φ) ∈ P do
    nejdříve pole úspěchu
    if |σ| > 1 then
        Přidáme řádek s pravidlem do LL tabulky úspěchu
        foreach r′ ∈ σ do
            foreach x ∈ Predict_k (r') do
                LL(k)_uspech(r, x) = r′
    stejný proces opakujeme pro pole neúspěchu
    if |φ| > 1 then
        foreach r′ ∈ φ kde r′ = (X → a, σ, φ) do
            foreach x ∈ Predict_k (r') do
                LL(k)_neuspech(r, x) = r′
```

Výstupem algoritmu jsou dvě tabulky pro dané k. Jedna tabulka pro pole úspěchu (označovaná $LL(k)_{uspech}$) a druhá po pole neúspěchu (označovaná $LL(k)_{neuspech}$). V případě, že se jedná o gramatiku bez kontroly na výskyt, je výstupem pouze tabulka $LL(k)_{uspech}$. Tabulky generujeme v minimalizované podobě. Řetězce terminálů, které pro predikci nejsou potřeba, se v tabulce vůbec nevyskytují.

Pokud by se v jedné buňce tabulky vyskytovalo pravidel víc, dochází ke stavu, který nazýváme kolize. Tento stav znamená, že gramatika není *LL(k)* pro dané k.

6.4 LL(k) programované gramatiky

Jako v případě bezkontextových *LL* gramatik, bude potřeba definici programovaných gramatik omezit tak, aby pro ně navrhovaný postup syntaktické analýzy fungoval. Definujeme tedy podtřídu programovaných gramatik, na které lze prediktivní analýzu uplatnit. Omezení budou kladena především na rozhodující pravidla. Omezení přejímáme z [7], kde byla definována obdobná třída *SPG* - *jednoduchých programovaných gramatik* ze které vycházíme.

Definice 6.4.1. Programované LL(k) gramatiky. Čtveřice $PG = (N, T, P, S)$ je *LL(k)* programovaná gramatik pro $k \geq 1$ kde pro každé $A \in N$ neexistuje derivace $A \Rightarrow^* A$, a pro každou trojici $(X \to Y, \sigma, \varphi) \in P$, kde: $|\sigma| > 1$ nebo $|\varphi| > 1$ platí:

1. Všechna pravidla v množině σ, resp. φ mají stejný neterminál na levé straně.

2. Všechna pole neúspěchu pro pravidla v σ, resp. φ jsou prázdné množiny.

3. Pro dvě pravidla $r_i : A \rightarrow \alpha$ a $r_j : A \rightarrow \beta$, ze σ, resp. φ platí:
   - $First_k(\alpha Follow_k(A)) \cap First_k(\beta Follow_k(A)) = \emptyset$
   - Neterminál A je nejlevější v aktuálně rozvíjené větné formě.

Nyní podrobně rozebereme jednotlivá omezení. První omezení znamená, že se musíme rozhodnout nad právě jedním neterminálem. Pokud bychom totiž aplikovali predikci na různé neterminály, byla by nerelevantní. Druhé omezení se týká *polí neúspěchu* a zaručuje, že pokud bude pravidlo vybráno, pak bude i použito. V případě, že bychom se pro pravidlo rozhodli a to selhalo v kontrole na výskyt, pak by rozhodnutí nemuselo být nutně správné. První část třetí podmínky zajišťuje stejně jako u silných LL(k) gramatik, že pravidlo, které můžeme zvolit, je právě jedno. Druhou část třetí podmínky si rozebereme podrobněji, znamená totiž, že v případě rozhodování musíme derivovat nejlevější neterminál. Podle [3] ale programované gramatiky omezené pouze na striktně nejlevější derivace mají stejnou generující sílu jako gramatiky bezkontextové. Mohlo by se tedy zdát, že silnější nástroj pro syntaktickou analýzu nezískáme. Je pravda, že sílu programovaných gramatik na úkor determinismu omezíme. Avšak díky tomu, že v pravidlech, u kterých rozhodnutí provádět nemusíme, přepisujeme nejlevější neterminál, který může být přepsán, je navrhovaný proces syntaktické analýzy schopen zpracovat některé jazyky, které nejsou bezkontextové. Toto bude názorně ukázáno v části 7.1.

6.5 Syntaktický analyzátor

Nyní, když už víme jak sestrojit rozhodovací tabulky, můžeme algoritmus syntaktického analyzátoru pro programované gramatiky představit v kompletní podobě. Pro lepší ilustraci jej předvedeme i na konkrétním případě.

Stav syntaktického analyzátoru je určen symbolem na vrcholu zásobníku a aktuálním tokenem na vstupu. Na začátku analýzy na zásobník vložíme speciální zakončovací symbol $ a počáteční neterminál gramatiky. Pak na zásobník aplikujeme startovací pravidlo a načteme první token ze vstupu. Analyzátor pak opakuje následující kroky, dokud analýza neskončí úspěchem či neúspěchem. V případě, že proces skončí úspěšně, pak vstupní řetězec splňuje syntaxi určenou gramatikou.

- Pokud vrchol zásobníku obsahuje $ a ten je i aktuálním tokenem, končí analýza úspěšně.
- Pokud vrchol zásobníku obsahuje terminál, který se shoduje s aktuálním tokenem, odstraníme vrchol zásobníku, načteme další token ze vstupu a aktualizujeme `lookahead`.
- Pokud vrchol zásobníku obsahuje neterminál, určíme pravidlo a pokusíme se o jeho aplikaci na zásobník.
- V ostatních případech se jedná o chybu syntaktické analýzy.

Analyzátor musí uchovávat informaci o aktuálně používaném pravidle a ukazatel `lookahead` pro predikci délky *k*. Nadále při práci s *LL(k)* programovanými gramatikami s kontrolou na výskyt využívá stavovou proměnnou `use`, která určuje, zda-li byla kontrola na výskyt úspěšná či nikoli.

Proces rozhodování následujícího pravidla probíhá takto:

- Stavová proměnná `use` určuje, zda-li následující pravidlo vybíráme z množiny pole úspěchů, resp. pole neúspěchů a jestli pří výběru používáme tabulku LL_{spch} či LL_{nespch}
- Pokud zkoumaná množina neobsahuje žádné pravidlo, syntaktická analýza skončí chybou.
- Pokud zkoumaná množina obsahuje pouze jedno pravidlo, tak jej označíme za vybrané.
- Pokud zkoumaná množina obsahuje pravidel více, nahlížíme do příslušné rozhodovací tabulky a na základě aktuálního pravidla a ukazatele `lookahead` zkusíme v tabulce vyhledat pravidlo následující. V případě, že tabulka záznam pro `lookahead` neobsahuje, nebo pole neobsahuje žádné pravidlo, končí analýza s chybou. V opačném případě označíme pravidlo za vybrané.

Proces kontroly na výskyt nám situaci při expanzi pravidla komplikuje. Pokud máme vybrané pravidlo, zkusíme neterminál na jeho levé straně nalézt na zásobníku. Pokud se v zásobníku nenachází, kontrola na výskyt selhala a nastavíme `use = fail` a celý proces hledání pravidla se opakuje v množině neúspěchu. V opačném případě, že se terminál v zásobníku nachází, ho vyjmeme ze zásobníku a na jeho místo umístíme pravou stranu pravidla v obráceném pořadí. Je nutno poznamenat, že v některých případech dochází i k expanzi neterminálu, který není přímo na vrcholu zásobníku. Proto musí zásobník umožnit měnit svůj obsah i mimo jeho vrchol. Podle definice k této situaci dochází pouze v případě, pokud množina určující další pravidlo obsahuje pravidlo pouze jedno.

6.6 Algoritmus syntaktického analyzátoru

Celý algoritmus by pak v pseudokódu vypadal následovně:

Algoritmus 4: Prediktivní syntaktický analyzátor pro programované gramatiky

Vstup : $LL(k)_{uspech}$, $LL(k)_{neuspech}$ pro $PG(N, T, P, S)$, $x \in T^+$, $k \geq 1$
Výstup: Posloupnost pravidel pokud $x \in L(PG)$ jinak chyba

```
inicializace zásobníku
stack.push($)
use = true
stack.apply(počáteční pravidlo)
aktuálníPravidlo = počátečníPravidlo
lookahead = a_{i+1} ... a_{i+k}
X je vrchol zásobníku, a aktuální token repeat
    switch vrchol zásobníku X do
        case X = $
            if a = $ then úspěch else
                chyba
        case X ∈ T
            if X = a then stack.pop(X)
            přečti další a_i ze vstupu
            lookahead = a_{i+1} ... a_{i+k}
            else
                chyba
        case X ∈ N
            máme aktuální pravidlo P = (r, σ, φ)
            if use = true then
                LL = LL_uspech
                zkoumanaMnozina = σ
            else
                LL = LL_neuspech
                zkoumanaMnozina = φ
            if |zkoumanaMnozina| = 0 then chyba
            else if |zkoumanaMnozina| = 1 then
                aktualniPravidlo = r ∈ zkoumanaMnozina
            else if |zkoumanaMnozina| > 1 then
                if novePravidlo ∈ LL(lookahead, aktualniPravidlo) then
                    aktualniPravidlo = novePravidlo
                    zapiš aktualniPravidlo na výstup
                chyba
            if stack.apply(aktualniPravidlo) then
                use = true
            else
                selhání kontroly výskytu
                use = fail
until úspěch or chyba
```

Nyní celý proces syntaktické analýzy předvedeme na konkrétní gramatice.

Příklad 6.6.1. Nechť $PG_1 = (\{S, S', A, B\}, \{1, 0, h, a\}, P, S)$ [5] je programovaná gramatika s kontrolou na výskyt, kde P:

$$
\begin{array}{ll}
(r_1 : S' \rightarrow S, \{r_2, r_3\}, \emptyset), & (r_2 : S \rightarrow 1SB\{r_4\}, \emptyset), \\
(r_3 : S \rightarrow 0S\{r_4\}, \emptyset) & (r_4 : A \rightarrow BB\{r_4\}, \{r_5\}), \\
(r_5 : B \rightarrow A\{r_5\}, \{r2, r3, r6\}), & (r_6 : S \rightarrow h\{r_7\}, \emptyset), \\
(r_7 : A \rightarrow a\{r_7\}, \emptyset), &
\end{array}
$$

Tato gramatika generuje řetězce nha^n, kde n je celé číslo reprezentované v binární soustavě.

Pokud si důkladně prohlédneme rozhodovací pravidla r_1 a r_5, vidíme, že levé strany pravidel, která po nich mohou následovat, začínají vždy neterminálním symbolem. Pro rozhodování stačí tedy použít pouze množinu $First_1$. Může tedy vypadat, že navržený algoritmus je zbytečně silný. Toto tvrzení však vyvrátíme v další kapitole, kde se budeme snažit analyzovat i jiné gramatiky a to zejména ty generující bezkontextové jazyky. Tuto gramatiku jsme zvolili především, abychom mohli ukázat princip kontroly výskytu.

Vypočteme množinu $Predict_1$ pro pravidla, nad kterými se musíme rozhodovat. Jsou to pravidla r_2, r_3 a r_6:

$$
\begin{array}{l}
Predict_1(r2 : S \rightarrow 1SB) = \{1\} \\
Predict_1(r3 : S \rightarrow 0S) = \{0\} \\
Predict_1(r6 : S \rightarrow h) = \{h\}
\end{array}
$$

Na základě této množiny pak zkonstruujeme tabulky LL_{uspech} a $LL_{neuspech}$:

	1	**0**
$\boldsymbol{r_1}$	r_2	r_3

Tabulka 6.1: Tabulka $LL_{neuspech}$ pro PG_1

	1	**0**	**h**
$\boldsymbol{r_5}$	r_2	r_3	r_6

Tabulka 6.2: Tabulka $LL_{neuspech}$ pro PG_1

Pokud již máme sestrojeny tabulky, tak můžeme předvést jednotlivé kroky analýzy, například slova $1ha$.

Zásobník	**Vstup**	**Use**	**Pravidlo**	**Akce**
$**S'**	**1**ha$	True		$r_1 : S' \to S$
$**S**	**1**ha$	True	r_1	$r_2 : S \to 1SB$
$**BS1**	**1**ha$	True		match(1)
$**BS**	**h**a$	True	r_2	$r_4 : A \to BB$
$**BS**	**h**a$	False	r_4	$r_5 : B \to A$
$**AS**	**h**a$	True	r_5	$r_5 : B \to A$
$**AS**	**h**a$	False	r_5	$r_6 : S \to h$
$**Ah**	**h**a$	True		match(h)
$**A**	**a**$	True	r_6	$r_7 : A \to a$
$**a**	**a**$	True	r_7	match(a)
$	$	True		Úspěšný konec analýzy

Tabulka 6.3: Proces syntaktické analýzy pro PG_1 a slovo *1ha*

Akce `match` znamená, že terminál na vrcholu zásobníku odpovídá tokenu na vstupu a je jak ze vstupu tak ze zásobníku odebrán. Pravidla označená zelenou barvou označují, že kontrola na výskyt proběhla úspěšně a tato pravidla by také byla na výstupu algoritmu jako levá derivace. V tomto případě $r_1r_2r_5r_6r_7$. Pravidla označená červeně se nepodařilo aplikovat, tzn. kontrola na výskyt selhala. Tučně označené tokeny vstupu společně s proměnou `use` a aktuálním pravidlem jsou pak rozhodující pro predikci a případné hledání v tabulce.

Kapitola 7

Zkoumání síly algoritmu

Tato kapitola se zabývá rozhodovací silou algoritmu. Definováním podtřídy 6.4 programovaných gramatik jsme množinu jazyků generovaných programovanými gramatikami výrazně omezili. Při zvolení vhodného k však můžeme analyzovat i některé jazyky, které nejsou bezkontextové.

7.1 Příklady *LL(k)* programovaných gramatik

Nyní prozkoumáme několik gramatik, zaměříme se na ty generující jazyky, které nejsou bezkontextové. Neuvádíme celý proces algoritmu, pouze pro vybrané gramatiky sestrojíme rozhodovací tabulky.

Jako první zvolíme gramatiku, generující jazyk $L = \{a^n b^n c^n \text{ kde } n \geq 1\}$, který je typickým příkladem jazyka, který není bezkontextový.

Příklad 7.1.1. Nechť $PG_2 = (\{S, A, B, C\}, \{a, b, c\}, P, S)$, [5], je programovaná gramatika bez kontroly na výskyt, kde:

$$\begin{array}{ll} (r_1 : S \to ABC, \{r_2, r_5\}), & (r_2 : A \to aA, \{r_3\}), \\ (r_3 : B \to bB, \{r_4\}), & (r_4 : C \to cC, \{r_2, r_5\}), \\ (r_5 : A \to a, \{r_6\}), & (r_6 : B \to b, \{r_7\}), \\ (r_7 : C \to c, \emptyset), & \end{array}$$

Například řetězec $aabbcc$ vygenerujeme gramatikou PG_2 podle následujících derivací: $S \Rightarrow ABC[r_1] \Rightarrow aAbBC[r_2] \Rightarrow aAbBC[r_3] \Rightarrow aAbBcC[r_4] \Rightarrow aabBcC[r_5] \Rightarrow aabbcC[r_6] \Rightarrow aabbcc[r_7]$.

Rozhodovací pravidla v případě této gramatiky jsou r_1 a r_4, obě vybírají zda-li použít pravidlo r_2 nebo r_5. Pravá strana těchto pravidel obsahuje stejný neterminál a tudíž jedna z podmínek pro $LL(k)$ programované gramatiky je splněna. Levá strana pravidel r_2 a r_5 začíná stejným neterminálním symbolem a, gramatika tedy není *LL(1)* programovaná gramatika. Zvolíme tedy $k = 2$ a zkusíme sestavit množinu $predict_2$:

$$\begin{array}{l} Predict_2(r2 : A \to aA) = \{aa\} \\ Predict_2(r5 : A \to a) = \{ab\} \end{array}$$

Sestrojíme $LL(2)_{uspech}$ pro PG_2.

	aa	**ab**
$\boldsymbol{r_1}$	r_2	r_5
$\boldsymbol{r_4}$	r_2	r_5

Tabulka 7.1: $LL(2)_{uspech}$ pro PG_2

Obě množiny se liší, v tabulce $LL(2)_{uspech}$ tedy nedojde ke kolizi a můžeme použít výše zmiňovaný algoritmus prediktivní analýzy.

Jednoduchou modifikací gramatiky PG_2 získáme gramatiku, která generuje jazyk $L = \{a^n db^n c^n d$ kde $n \geq 1\}$, která je také $LL(2)$ programovanou gramatikou:

Příklad 7.1.2. Nechť $PG_3 = (\{S, A, B, C\}, \{a, b, c, d\}, P, S)$, je programovaná gramatika bez kontroly na výskyt, kde:

$$
\begin{array}{ll}
(r_1 : S \to ABC, \{r_2, r_5\}), & (r_2 : A \to aA, \{r_3\}), \\
(r_3 : B \to bB, \{r_4\}), & (r_4 : C \to cC, \{r_2, r_5\}), \\
(r_5 : A \to ad, \{r_6\}), & (r_6 : B \to b, \{r_7\}), \\
(r_7 : C \to cd, \emptyset), &
\end{array}
$$

A její tabulka $LL(2)_{uspech}$ vypadá následovně:

	aa	**ad**
$\boldsymbol{r_1}$	r_2	r_5
$\boldsymbol{r_4}$	r_2	r_5

Tabulka 7.2: $LL(2)_{uspech}$ pro PG_3

Další z programovaných *LL(2)* gramatik je například tato:

Příklad 7.1.3. Nechť $PG_4 = (\{S, A, B, C\}, \{a, b, c, d\}, P, S)$ je programovaná gramatika bez kontroly na výskyt, kde:

$$
\begin{array}{ll}
(r_1 : S \to AD, \{r_2\}), & (r_2 : A \to aAc, \{r_2, r_3\}), \\
(r_3 : A \to bC, \{r_5\}), & (r_4 : D \to dD, \{r_5, r_6\}), \\
(r_5 : C \to b, \{r_7\}, \emptyset), & (r_6 : C \to A, \{r_3\}), \\
(r_7 : D \to d, \emptyset), &
\end{array}
$$

Gramatika PG_4 generuje jazyk $L = \{a^n b^m c^n d^m$ kde $n \geq 1$ a $m \geq 2\}$.

Řetězec *abbcdd* generuje podle následujících derivací: $S \Rightarrow AD[r_1] \Rightarrow aAcD[r_2] \Rightarrow abCcD[r_4] \Rightarrow abbcD[r_5] \Rightarrow abbcdd[r_7]$.

Opět se jedná o $LL(2)$ programovanou gramatiku, rozhodujícím pravidly jsou r_2 a r_4, musíme tedy sestrojit množinu $Predict_2$ pro pravidla r_2,r_3,r_5 a r_6.

$$
\begin{array}{l}
Predict_2(r_2 : A \to aAc) = \{aa, ab\} \\
Predict_2(r_3 : A \to bC) = \{bb\} \\
Predict_2(r_5 : C \to b) = \{bd, bc\} \\
Predict_2(r_6 : C \to A) = \{bb\}
\end{array}
$$

Sestrojíme $LL(2)_{uspech}$ pro PG_3.

	aa	**ab**	**bb**	**bd**	**bc**
$\boldsymbol{r_2}$	r_2	r_2	r_3		
$\boldsymbol{r_4}$			r_6	r_5	r_5

Tabulka 7.3: $LL(2)_{uspech}$ pro PG_3

Z předchozích příkladů je tedy vidět, že můžeme analyzovat i některé jazyky nad rámec bezkontextových jazyků, čímž jsme splnili jeden z cílů práce.

Kapitola 8

Implementace

V této kapitole se věnujeme popisu implementace výše navrženého syntaktického analyzátoru. Výsledkem implementace je konzolová aplikace, která na vstupu očekává soubor reprezentující gramatiku ve speciálním formátu a soubor s řetězcem, který chceme analyzovat. Na výstupu je pak informace, zda-li tento řetězec patří, nebo nepatří do jazyka specifikovaného vstupní gramatikou. Celá aplikace byla navržena v programovacím jazyku Python ve verzi 3. Tento programovací jazyk byl zvolen z důvodu jednoduché režie nad dynamickými strukturami. Prvním krokem při implementaci bylo navržení tvaru konfiguračního souboru gramatiky, jeho formát je pak k nalezení na přiloženém CD. Na jeho základě pak byly vytvořeny datové typy `Symbol` reprezentující symboly gramatiky. Dále byl navržen datový typ `Rule` reprezentující pravidla gramatiky.

Celá aplikace byla navržena objektově a skládá se z několika modulů. Následuje výčet a popis funkce jednotlivých modulů.

8.1 Modul pgparser

Tento modul je vstupním bodem programu. Zajišťuje interakci s uživatelem a komunikaci všech ostatních částí aplikace. Za pomocí modulu `lexer` vytvoří lexikální analyzátor pro vstupní řetězec. Dále zpracovává všechny chyby, které v běhu programu mohou nastat. O všech chybách informuje uživatele na standardní chybový výstup.

8.2 Modul lexer

Tento modul zajišťuje prostředky pro lexikální analýzu. Jde o generátor tokenů. Na základě výčtu pravidel daného regulárními výrazy, lze nastavit, které typy tokenů bude rozpoznávat. Metoda `getToken` pak navrací speciální datový typ dvojice ve formátu $tokenType, attribute$.

8.3 Modul parseGrammar

Tato část aplikaci zpracovává soubor vstupní gramatiky. Modul nad gramatikou provádí lexikální, syntaktickou i sémantickou analýzu. Většina chyb syntaktických i sémantických chyb ve vstupním souboru gramatiky je tak rozpoznána už při načítání. Syntaktická analýza je implementována pomocí konečného stavového automatu.

8.4 Modul grammar

Obsahuje třídu `grammar`, která zajišťuje vnitřní reprezentaci načtené gramatiky. Udržuje množinu terminálů, neterminálů a pravidel. Metoda `checkGrammar` zajišťuje dodatečné sémantické kontroly, které nebylo možné provést při jejím načítání ze souboru. A to především zda-li množiny úspěchu a neúspěchu obsahují pravidla, která jsou obsažena v gramatice. Metoda `constructSets`, zajišťuje vytvoření slovníku pomocných množin pro tvorbu rozhodovací tabulky.

8.5 Modul llparser

Tento modul je jádrem programu. Obsahuje třídu `Parser`, která na základě vstupní gramatiky, a vstupního řetězce provádí výše navržený algoritmus nerekurzivní prediktivní syntaktické analýzy. Také obsahuje třídu `PGLLTable`, která vytváří rozhodovací *LL(k)* tabulky ve formě dvourozměrného pole. Metoda `search` umožňuje vyhledávání v tabulce. Stará se také o funkci zásobníku, který je reprezentován jednosměrným zřetězeným seznam.

Kapitola 9

Závěr

Tato bakalářské práce se zabývala syntaktickou analýzou nad programovanými gramatikami. Na začátku byla nastudována teorie ze [5]. Byl vysvětlen princip fungování těchto gramatik. Zaměřil jsem se především na skupinu bezkontextový programovaných gramatik. Na tyto gramatiky jsem se pak snažil aplikovat metody syntaktické analýzy, které existují pro bezkontextové gramatiky.

Detailněji byla rozebrána jedna z nejobecnějších metod, metoda hrubé síly. Tato metoda byla zkoumána spíše z důvodu pochopení funkce programovaných gramatik a především, aby bylo jasné, na která pravidla se zaměřit při tvorbě postupů syntaktické analýzy.

V hlavní části práce byl na základě [7] modifikován postup nerekurzivní prediktivní syntaktické analýzy pro programované gramatiky. Byl popsán proces tvorby *LL(k)* rozhodovacích tabulek. Celý model nerekurzivní analýzy byl upraven tak, aby byl schopen pracovat s vlastností *kontroly na výskyt*. Dále bylo nutné upravit typ zásobníku, aby umožňoval expanzi neterminálních symbolů i mimo jeho vrchol.

Následně byl tento postup aplikován na několik gramatik a ukázalo se, že je možné analyzovat takto i gramatiky, které generují jazyky, které nejsou bezkontextové. Tímto byl splněn jeden z hlavních cílů práce.

V poslední fázi byla na navrženém algoritmu implementována aplikace ve formě skriptu v jazyce Python, která jej demonstruje.

Práce nabízí spoustu prostoru pro rozšíření. Algoritmus syntaktické analýzy funguje pouze na omezenou podmnožinu v rámci programovaných gramatik. Bylo by tedy vhodné metody syntaktické analýzy dále modifikovat tak, aby se velikost těchto omezení zmenšila. Dále by bylo možné navrhnout programovací jazyk nad programovanými gramatikami a pro ten sestavit kompletní překladač.

Přínos projektu vidím hlavně osobní. Rozšířil mé znalosti v oblasti teorie formálních jazyků. Implementovaná aplikace by také mohla sloužit jako pomůcka k pochopení principu fungování programovaných gramatik.

Literatura

[1] Aho, A. *Compilers: principles, techniques, & tools*. Berlin: Pearson/Addison Wesley, 2007. Addison-Wesley series in computer science. ISBN 9780321486813.

[2] Aho, A. a Ullman, J. *The Theory of Parsing, Translation, and Compiling: Parsing*. London: Prentice-Hall, 1972. Prentice-Hall Series in Automatic Computation. ISBN 9780139145568.

[3] Dassow, J. a Păun, G. *Regulated rewriting in formal language theory*. Berlin: Springer-Verlag, 1989. ISBN 9783540514145.

[4] Meduna, A. *Automata and Languages: Theory and Applications*. London: Springer, 2000. 920 s. ISBN 9781852330743.

[5] Rosenkrantz, D. J. Programmed Grammars and Classes of Formal Languages. In. New York: ACM, leden 1969. S. 107–131. ISSN 0004-5411.

[6] Rozenberg, G. a Salomaa, A. (ed.). *Handbook of formal languages: linear modeling: background and application*. New York: Springer, 1997. ISBN 3-540-60648-3.

[7] Sebesta, R. W. On context-free programmed grammars. In. New York: Pergamon Press, Inc., červenec 1989. S. 99–108. ISSN 0096-0551.

[8] Virginia Tech - Dr. Lenwood S. Heath. *Deterministic Parsing: LL(k) Grammars* [online]. 2001, Updated 01/12/2001 01:16:57 [cit. !2012-05-1]. Dostupné na: <`http://courses.cs.vt.edu/ cs4114/2000/lectures/16/detparsing.pdf`>.

Příloha A

Obsah CD

Přiložený datový nosič obsahuje:

1. Text bakalářské práce ve formátu PDF
2. Zdrojové kódy navrženého syntaktického analyzátoru
3. Sadu vzorových příkladů pro testování funkce analyzátoru