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 ŘÍZENÝCH GRAMATIK

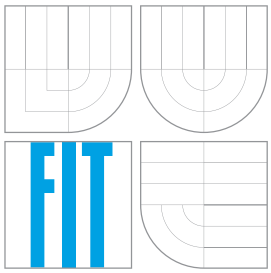
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. ROMAN ŠRAJER

BRNO 2011

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 ŘÍZENÝCH GRAMATIK

PARSING OF REGULATED GRAMMARS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. ROMAN ŠRAJER

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ZBYNĚK KŘIVKA, Ph.D.

BRNO 2011

Zadání diplomové práce/4790/2010/xsrajo00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2010/2011

Zadání diplomové práce

Řešitel: **Šrajer Roman, Bc.**
Obor: Informační systémy
Téma: **Syntaktická analýza řízených gramatik**
Parsing of Regulated Grammars
Kategorie: Překladače

Pokyny:

1. Seznamte se s pokročilými typy tzv. řízených gramatik z teorie formálních jazyků a jeden typ vyberte pro podrobnější zkoumání (např. programované nebo gramatiky s nahodilým kontextem).
2. Navrhněte algoritmus syntaktické analýzy těchto gramatik nebo jejich vybrané podmnožiny.
3. Implementujte syntaktický analyzátor založený na navržených algoritmech.
4. Jeho funkčnost ověřte na souboru jednoduchých příkladů.
5. Zhodnoťte vlastnosti vašeho přístupu (efektivita, složitost implementace, obecnost) a nastiňte možnosti budoucího vývoje.

Literatura:

- Meduna, A.: Automata and Languages, Springer, London, 2000.
- Rozenberg, G., Salomma, A. (Eds.): Handbook of Formal Languages, Volume 2: Linear Modelling: Background and Application, Springer, 1997.
- Aho, A. V., Sethi, R., Ullman, J. D.: Compilers - Principles, Techniques, and Tools, Addison-Wesley, 1986.

Při obhajobě semestrální části diplomového projektu je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Křivka Zbyněk, Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 20. září 2010
Datum odevzdání: 25. května 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Originální licenční smlouva je přiložena ve výtisku diplomové práce v knihovně FIT VUT Brno.

Abstrakt

Diplomová práce se zabývá řízenými gramatikami, jejich principem a vyjadřovacími schopnostmi vzhledem k Chomského klasifikaci jazyků. Více do detailů je probrána programovaná gramatika z hlediska různých typů nejlevějších derivací. V práci je prezentována varianta deterministické syntaktické analýzy programovaných gramatik, která pracuje podobně jako analýza $LL(k)$ gramatik. Dále jsou představeny algoritmy pro převod maticových gramatik na programované bez změny vyjadřovací síly. Rovněž je prezentován mechanismus syntaktické analýzy regulárně řízených gramatik za pomocí programované gramatiky. Nakonec jsou prezentovány kooperující distribuované gramatiky s levě povolujícími gramatikami jako komponenty. Jejich syntaktická analýza je zajištěna buď deterministicky pomocí prediktivní tabulky nebo prohledáváním stavového prostoru.

Abstract

This thesis studies regulated grammars, their fundamentals and expressing power regarding Chomsky hierarchy of languages. Programmed grammars are investigated in more depth considering a few types of leftmost derivations. A variant of deterministic syntax analysis of programmed grammars is introduced. This analysis works similarly as $LL(k)$ parsing. Transformations of matrix grammars into programmed grammars without changing their expressing power are introduced. The syntax analysis by regularly controlled grammars partly using programmed grammars are presented. In the end, cooperating distributed grammars with left permitting grammars as components are mentioned. Their deterministic syntax analysis uses predictive table or exhaustive exploration of the whole state space.

Klíčová slova

Syntaktická analýza, řízená gramatika, regulárně řízená gramatika, maticová gramatika, programovaná gramatika, gramatika s nahodilým kontextem, levě povolující gramatika, kooperující distribuovaná gramatika, jednoduchá programovaná gramatika, rozšířená jednoduchá programovaná gramatika, jednoduchá maticová gramatika, rozšířená jednoduchá maticová gramatika, jednoduchá regulárně řízená gramatika, jednoduchá levě povolující kooperující distribuovaná gramatika, kanonická derivace

Keywords

Syntax analysis, regulated grammar, regularly controlled grammar, matrix grammar, programmed grammar, random context grammar, left permitting grammar, cooperating distributed grammar, simple programmed grammar, extended simple programmed grammar, simple matrix grammar, extended simple matrix grammar, simple regularly controlled grammar, simple left permitting cooperating distributed grammar, canonical derivation

Citace

Roman Šrajer: Syntaktická analýza řízených gramatik, diplomová práce, Brno, FIT VUT v Brně, 2011

Syntaktická analýza řízených gramatik

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Zbyňka Křivky Ph.D a uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Roman Šrajer
14. května 2011

Obsah

1 Úvod **2**

2 Chomského klasifikace **3**

3 Řízené gramatiky **7**
3.1 Regulárně řízená gramatika 8
3.2 Maticová gramatika 10
3.3 Programovaná gramatika 11
3.3.1 Nejlevější derivace typu 1 14
3.3.2 Nejlevější derivace typu 2 15
3.3.3 Nejlevější derivace typu 3 15
3.4 Gramatika s nahodilým kontextem 16

4 Syntaktická analýza **20**
4.1 Programované gramatiky 20
4.1.1 $ESPG(k)$ gramatika 22
4.1.2 Složitost $ESPG(k)$ 28
4.1.3 Vyjadřovací síla $ESPG(k)$ 29
4.1.4 Problém s výpočtem množin $FIRST_k$ a $FOLLOW_k$ 31
4.2 Maticové gramatiky 33
4.2.1 $SMG(k)$ gramatika 35
4.2.2 $ESMG(k)$ gramatika 38
4.2.3 Vlastnosti $SMG(k)$ a $ESMG(k)$ 41
4.3 Regulárně řízené gramatiky 42
4.3.1 $SRCG(k)$ gramatika 43
4.4 Gramatiky s nahodilým kontextem 48
4.4.1 $SLPCD(k)$ gramatika 49
4.4.2 Složitost $SLPCD(k)$ 54
4.4.3 Vyjadřovací síla $SLPCD(k)$ 55

5 Implementace analyzátoru **56**

6 Závěr **59**

A Obsah CD **64**

B Manuál k analyzátoru **65**

Kapitola 1

Úvod

V praxi zdaleka nejpoužívanějšími gramatikami jsou regulární gramatiky a bezkontextové gramatiky [21], jejichž vyjadřovací síla je omezena na regulární resp. bezkontextové jazyky v Chomského klasifikaci jazyků. Regulární gramatiky a jím ekvivalentní konečné automaty nacházejí svá uplatnění zejména v programovacích jazycích pro vyhledávání vzorů (*pattern matching*) a při tvorbě lexikálního analyzátoru při výstavbě překladačů [12]. Bezkontextové gramatiky jsou používány při popisu syntaktické struktury jazyků a to nejen těch programovacích. Důvodů, proč jsou tyto dvě gramatiky nejvíce používány, je několik. Z praktického hlediska jsou snadněji použitelné a implementovatelné a jejich návrh a pochopení je jednodušší než u kontextových a neomezených gramatik. Například existují rozsáhlé možnosti ve formě *LL(k)* a *LR(k)* gramatik [2]. Další výhodou bezkontextových gramatik je možnost určit příslušnost slova (*membership*) do jazyka v polynomiálním čase (viz algoritmus CYK s kubickou složitostí). Vyjadřovací síla bezkontextových gramatik však v mnoha aplikacích není dostačující, proto je snaha používat silnější modely popisu jazyků a zároveň se vyhnout kontextovým a neomezeným gramatikám, se kterými se obtížně pracuje. Cílem této práce je popsat řízené gramatiky, které jako základ používají bezkontextová pravidla.

V kapitole 2 je pro úplnost popsána Chomského klasifikace jazyků a gramatiky, které jej popisují. V kapitole 3 je popsáno několik typů řízených gramatik, přičemž každý typ je doprovázen formální definicí, příkladem a textovým popisem, jak daná gramatika funguje. Speciálně u programovaných gramatik jsou zmíněny různé druhy nejlevějších derivací a jejich vyjadřovací síla. Kapitola 4 pojednává o syntaktické analýze programovaných gramatik, regulárně řízených gramatik, maticových gramatik a kooperujících distribuovaných gramatik s levě povolující gramatikou (varianta gramatiky s nahodilým kontextem). U programovaných gramatik je představena modifikace jejich syntaktické analýzy z [17], přičemž je nakonec dokázáno, že jsou ekvivalentní. Dále jsou prezentovány algoritmy převodů z maticových a regulárně řízených gramatik na programované gramatiky a jejich příklady použití. U kooperujících distribuovaných gramatik je představena kombinovaná syntaktická analýza, tedy jak s pomocí prediktivní tabulky tak s prohledáváním stavového prostoru a s derivací omezenou na nejlevější možný přepis. V kapitole 5 je stručně popsán způsob implementace syntaktického analyzátoru, který realizuje navržené algoritmy z kapitoly 4. V poslední kapitole 6 jsou shrnuty výsledky diplomové práce vzhledem k navrženým algoritmům a je naznačen další možný vývoj syntaktického analyzátoru, který by mohl implementovat další druhy řízených gramatik.

Kapitola 2

Chomského klasifikace

Pro popis jazyků se používá generativní gramatika, se kterou je možné i nekonečné jazyky popsat konečným způsobem.

Poznámka: V dalších částech práce budou používány základní pojmy z teorie formálních jazyků jako řetězec/slovo, prázdný řetězec/slovo ε, iterace, zřetězení, regulární výraz/množina, cyklus v gramatice a z matematiky jako inkluze, vlastní podmnožina, sjednocení, průnik, apod. Pro jejich nastudování je možné použít například první 3 kapitoly z [12]. Dále názvy gramatik budeme označovat normálně a třídy jazyků, které generují, budeme potom zapisovat tučně.

Definice 2.1. Generativní gramatika G je čtveřice $G = (N,\ T,\ P,\ S)$, kde:

- N je konečná množina neterminálních symbolů
- T je konečná množina terminálních symbolů, přičemž $N \cap T = \emptyset$
- S je počáteční neterminální symbol, $S \in N$
- P je konečná podmnožina $P \subseteq (N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$

Prvek (α, β) množiny P se také nazývá přepisovacím pravidlem nebo jen pravidlem a je možné ho zapisovat jako $\alpha \to \beta$. Pravidlo, pro které platí $\beta = \varepsilon$, označujeme jako vymazávací pravidlo nebo ε-pravidlo. Část α resp. β budeme označovat jako levou resp. pravou stranu pravidla. V případě, že existuje více pravidel se shodnou levou stranou, místo zápisu

$$\begin{aligned} \alpha &\to \beta_1 \\ &\vdots \\ \alpha &\to \beta_n \end{aligned}$$

budeme zapisovat zkráceně

$$\alpha \to \beta_1 \mid \ldots \mid \beta_n$$

pro nějaké $n \geq 1$.

Definice 2.2. Nechť máme gramatiku $G = (N,\ T,\ P,\ S)$ a řetězce $\lambda, \mu \in (N \cup T)^*$. Mezi λ a μ platí binární relace $\lambda \Rightarrow_G \mu$ nazývaná přímá derivace, pokud platí:

$$\lambda = x\,\alpha\,y$$
$$\mu = x\,\beta\,y$$

kde $x, y \in (N \cup T)^*$ pro nějaké pravidlo $\alpha \to \beta \in P$.

Pokud je z kontextu zřejmé, že uvažujeme gramatiku G, je možné dolní index v přímé derivaci vypustit, tedy zapsat $\lambda \Rightarrow \mu$.

Definice 2.3. Nechť máme gramatiku $G = (N,\ T,\ P,\ S)$ a řetězce $\lambda, \mu \in (N \cup T)^*$. Mezi řetězcem λ a μ platí relace $\lambda \Rightarrow^n \mu$, pokud existuje sekvence přímých derivací $\nu_{i-1} \Rightarrow \nu_i$ pro $i = 1, \ldots, n$ a $n \geq 1$ tak, že platí:

$$\lambda = \nu_0 \Rightarrow \nu_1 \Rightarrow \cdots \Rightarrow \nu_n = \mu$$

Takovou posloupnost potom nazýváme derivaci délky n. Pokud potřebujeme zapsat derivaci obecně délky $n \geq 1$, zapisujeme $\lambda \Rightarrow^+ \mu$. Pokud navíc v gramatice G platí $\lambda \Rightarrow^+ \mu$ nebo $\lambda = \mu$, potom píšeme $\lambda \Rightarrow^* \mu$ pro počet derivací obecně $n \geq 0$.

Definice 2.4. Nechť máme gramatiku $G = (N,\ T,\ P,\ S)$. Řetězec $\alpha \in (N \cup T)^*$ se nazývá větnou formou gramatiky G, pokud platí $S \Rightarrow^* \alpha$. Pokud platí $\alpha \in T^*$, potom α nazýváme slovem gramatiky.

Definice 2.5. Jazyk $L(G)$ generovaný gramatikou G je definován jako množina slov, pro která platí:

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

Chomského klasifikace definuje 4 třídy jazyků, jejichž generativní gramatiky se liší tvarem pravidel v definici 2.1. Následující poznatky jsou převzaty z [23] a [12].

- Typ 0 - třída rekurzivně vyčíslitelných jazyků (*recursively enumerable*) — zkratka **RE**
- Typ 1 - třída kontextových jazyků (*context-sensitive*) — zkratka **CS**
- Typ 2 - třída bezkontextových jazyků (*context-free*) — zkratka **CF**
- Typ 3 - třída regulárních jazyků (*regular*) — zkratka **REG**

Rekurzivně vyčíslitelné jazyky

Rekurzivně vyčíslitelné jazyky jsou generovány gramatikou se stejnými tvary pravidel jako v definici 2.1, tedy pro každé pravidlo $\alpha \to \beta$ platí:

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

Definice 2.6. Nechť máme gramatiku $G = (N,\ T,\ P,\ S)$ z definice 2.1, potom platí $L(G) \in \mathbf{RE}$. Taková gramatika je také označována jako neomezená, protože tvar jejích pravidel není nijak omezen.

Příklad 2.1. Gramatika $G = (\{A, B\}, \{a, b\}, P, A)$, kde množina P obsahuje pravidla:

$$A \to AbB \mid a$$
$$AbB \to baB \mid BAbB$$
$$B \to b \mid \varepsilon$$

Například derivace: $A \Rightarrow AbB \Rightarrow baB \Rightarrow bab$

Kontextové jazyky

Definice 2.7. Nechť máme gramatiku $G = (N,\ T,\ P,\ S)$. Jazyk generovaný gramatikou G náleží do třídy kontextových jazyků, $L(G) \in \mathbf{CS}$, pokud každé pravidlo má tvar:

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

V případě, že $\varepsilon \in L(G)$, potom pravidlo $S \to \varepsilon \in P$, ale neterminál S se nesmí vyskytovat na pravé straně žádného pravidla.

Gramatika se nazývá kontextová, protože podřetězce α a β tvoří kontext neterminálu A. Neterminál A může být nahrazen řetězcem γ jen tehdy, pokud jeho levý kontext tvoří podřetězec α a pravý kontext podřetězec β. Navíc řetězec γ nesmí být prázdný. Z toho je tedy zřejmé, že kontextové gramatiky neumožňují zkracování větné formy během derivace, tedy platí $|A| \leq |\gamma|$ (s výjimkou pravidla $S \to \varepsilon$).

Příklad 2.2. Gramatika $G = (\{A, B, S\}, \{a, b\}, P, S)$, kde množina P obsahuje pravidla

$$S \to aABS \mid AB$$
$$aA \to aa$$
$$AB \to Aa \mid ABA$$
$$A \to a$$
$$B \to b$$

je kontextová gramatika, protože pro každé pravidlo existuje levý a pravý kontext, který není změněn. Například pro pravidlo $AB \to ABA$ platí $\alpha = A$, $\beta = \varepsilon$, $\gamma = BA$.

Například derivace: $S \Rightarrow AB \Rightarrow ABA \Rightarrow AaA \Rightarrow aaA \Rightarrow aaa$

Bezkontextové jazyky

Definice 2.8. Nechť máme gramatiku $G = (N,\ T,\ P,\ S)$. Jazyk generovaný gramatikou G náleží do třídy bezkontextových jazyků, $L(G) \in \mathbf{CF}$, pokud každé pravidlo má tvar:

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

Bezkontextové gramatiky jsou speciálním případem gramatik kontextových, kde levý i pravý kontext je prázdný, tedy $\alpha = \varepsilon$ a $\beta = \varepsilon$.

Příklad 2.3. Gramatika $G = (\{S\}, \{a, b, c\}, P, S)$, kde množina P obsahuje pravidla

$$S \to aSb \mid c$$

je bezkontextová gramatika, protože levá strana všech pravidel obsahuje právě jeden neterminální symbol.

Například derivace: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aacbb$

Regulární jazyky

Definice 2.9. Nechť máme gramatiku $G = (N,\ T,\ P,\ S)$. Jazyk generovaný gramatikou G náleží do třídy regulárních jazyků, $L(G) \in \mathbf{REG}$, pokud každé pravidlo má tvar:

$A \to aB$ nebo $A \to a$, $A, B \in N$, $a \in T$

Pokud platí $\varepsilon \in L(G)$, potom pravidlo $S \to \varepsilon \in P$ a neterminál S se nesmí nacházet na pravé straně žádného pravidla.

Příklad 2.4. Gramatika $G = (\{S, A, B\}, \{a, b, c\}, P, S)$, kde množina P obsahuje pravidla

$$S \to \varepsilon \mid c \mid aA \mid cB$$
$$A \to aA \mid c \mid cB$$
$$B \to bB \mid b$$

je pravá regulární gramatika, protože pravá strana všech pravidel nejdříve obsahuje právě jeden terminální symbol (s výjimkou $S \to \varepsilon$) a poté maximálně jeden neterminální symbol.

Například derivace: $S \Rightarrow aA \Rightarrow aaA \Rightarrow aacB \Rightarrow aacbB \Rightarrow aacbb$

Vztah tříd jazyků Chomského klasifikace

Pro lepší přehled zavedeme další dvě třídy jazyků:

- **FIN** ... třída všech konečných jazyků (s konečným počtem slov)
- **ALL** ... třída všech jazyků

Mezi zmíněnými třídami Chomského klasifikace platí věta 2.1.

Věta 2.1. *(viz [12])*

(*1*) $\mathbf{FIN} \subset \mathbf{REG} \subset \mathbf{CF} \subset \mathbf{CS} \subset \mathbf{RE} \subset \mathbf{ALL}$

Mezi všemi třídami platí ostrá inkluze, tedy určitá třída je vlastní podmnožinou nějaké třídy napravo od ní (ve výše uvedeném zápisu). Ze zápisu $\mathbf{RE} \subset \mathbf{ALL}$ například plyne, že existují jazyky, které nejsou ani částečně rozhodnutelné (viz důkaz diagonalizací v [23]).

Kapitola 3

Řízené gramatiky

Jak bylo napsáno v úvodu práce, bezkontextové jazyky nejsou v některých případech dostačující pro popis některých konstrukcí jazyků. Mnoho konstrukcí v programovacích jazycích obsahuje kontextové závislosti, které nejsou v bezkontextových gramatikách řešitelné. Příkladem může být nutnost deklarace proměnné před jejím použitím. Tato vlastnost není řešena na gramatické úrovni (pomocí pravidel), ale až na sémantické úrovni pomocí tabulek symbolů.

Smyslem řízených gramatik je zachování bezkontextových pravidel (viz definice 2.8) a zvýšení jejich vyjadřovacích schopností tak, že se do aplikace pravidel na neterminální symboly zavede nějaký způsob řízení. Následující podkapitoly vycházejí z prací [8, 16, 20, 21, 7, 3].

Nedeterminismus v gramatikách

Z matematického pohledu nemusí být nedeterminismus v gramatikách problematický. V případě syntaktické analýzy řízené nedeterministickou gramatikou je situace jiná. Mějme například bezkontextovou gramatiku $G = (\{A, B\}, \{a, b, c\}, P, A)$, kde P obsahuje pravidla:

(*1*) $A \rightarrow AB$
(*2*) $B \rightarrow AAB$
(*3*) $A \rightarrow a$
(*4*) $A \rightarrow b$
(*5*) $B \rightarrow c$

Dále mějme větnou formu $ABAAB$. Lze rozlišit dva druhy nedeterminsmu, se kterými se musí analyzátor vypořádat:

1. analyzátor neví, který neterminál ve větné formě přepsat v dalším kroku
2. pokud analyzátor vybere nějaký neterminál z větné formy k přepsání, neví, které pravidlo použít, protože neterminál A i B se vyskytuje na levé straně více jak jednoho pravidla

Taková gramatika je v syntaktické analýze nevhodná, protože vyžaduje prohledávat stavový prostor. Proto je snaha upravit gramatiku či omezit výběr neterminálů ve větné formě tak, aby gramatika byla vhodnější pro syntaktickou analýzu. Základní možnosti jsou:

1. upravit gramatiku tak, aby v každé derivaci existovalo maximálně jedno pravidlo použitelné pro derivaci
2. zavést predikci pro rozhodnutí, které pravidlo použít (jako u $LL(k)$ analýzy)
3. zavést omezení větné formy (např. maximální či minimální počet neterminálů, apod.), které znemožní použití některých pravidel v případě nedeterminismu
4. *k–kanonická* derivace pro $k \geq 1$ ([9], strana 100)

Bod 1 je těžko splnitelný. Například pro bezkontextové gramatiky, které generují nekonečný jazyk, není možné tuto podmínku splnit [23]. Predikce (bod 2) je možná jen u některých bezkontextových gramatik. U řízených gramatik je predikce rovněž možná jen pro některé gramatiky s dodatečnými podmínkami (viz kapitola 4). Při k-kanonické derivaci je vždy přepsán maximálně k-tý neterminál zleva ve větné formě. Lze rozlišit dvě varianty:

1. *striktní* – je bráno k neterminálů zleva bez ohledu na to, zda-li některý neterminál je vůbec možné přepsat
2. *obecná* – zleva je bráno k neterminálů, které mohou být přepsány

V případě bezkontextových gramatik z hlediska generativní síly na pořadí výběru neterminálů z větné formy nezáleží [13]. Pokud tedy máme bezkontextovou gramatiku G a jazyky $L_{k-can}(G)$, $L(G)$, které po řadě označují jazyky generované bezkontextovou gramatikou G k-kanonickou derivací pro libovolné $k \geq 1$, libovolnou derivací, platí $L_{k-can}(G) = L(G)$.

U řízených gramatik obecně neplatí to, co u gramatik bezkontextových. Může totiž dojít k situaci, že při přepisování nějakého neterminálu A na různých pozicích ve větné formě může dojít k vygenerování různých slov. Mějme například programovanou gramatiku (viz definice 3.4) $G = (\{S, A\}, \{a, b\}, P, S)$, kde P obsahuje pravidla:

(*1*) $S \rightarrow AA$, $\{2\}$, $\emptyset$
(*2*) $A \rightarrow a$, $\{3\}$, $\emptyset$
(*3*) $A \rightarrow b$, $\emptyset$, $\emptyset$

Pokud nebudeme uvažovat kanonickou derivaci, gramatika G generuje jazyk $L(G) = \{ab, ba\}$, protože ve větné formě AA je možné dále přepsat první (ab) nebo druhý (ba) neterminál A. Pokud budeme naopak uvažovat 1-*kanonickou* derivaci, ve větné formě AA bude muset být přepsán nejlevější neterminál A, tedy jazyk s tímto typem derivace bude $L_{1-can}(G) = \{ab\}$. Situace u ostatních řízených gramatik popsaných v této kapitole je podobná.

V následujících částech této kapitoly jsou popsány řízené gramatiky i z hlediska jejich vyjadřovacích schopností s ohledem na výběr libovolného neterminálu při derivování (tedy bez uvažování kanonické derivace). Speciálně u programovaných gramatik jsou popsány tři varianty kanonických derivací pro $k = 1$ a jejich generativní síla.

3.1 Regulárně řízená gramatika

Definice 3.1. Regulárně řízená gramatika (*regularly controlled grammar*) s kontrolou výskytu je šestice $G = (N,\ T,\ P,\ S,\ R,\ F\ )$, kde:

- N, T, P, S jsou definovány jako u bezkontextové gramatiky

- R je regulární množina nad P (pro přehlednost bude R dále v práci popisována jen pomocí regulárního výrazu)
- $F \subseteq P$

Mějme pravidlo $p = A \to w \in P$ a větné formy $x, y \in (N \cup T)^*$. Mezi x a y platí přímá derivace $x \Rightarrow_p^{ac} y$, pokud platí:

$$x = x_1 A x_2 \text{ a } y = x_1 w x_2 \text{ pro nějaké } x_1, x_2 \in (N \cup T)^*$$

nebo

$$x = y, \text{ pokud se neterminál } A \text{ nenachází v } x \text{ a zároveň } p \in F$$

Jazyk $L(G)$ je definován pro každé slovo $w \in T^*$, pro které platí sekvence derivací

$$S \Rightarrow_{p_1}^{ac} w_1 \Rightarrow_{p_2}^{ac} w_2 \Rightarrow_{p_3}^{ac} \cdots \Rightarrow_{p_n}^{ac} w_n = w$$

pro nějaké větné formy $w_1, w_2, \ldots, w_{n-1} \in (N \cup T)^*$ pro $n \geq 1$ a platí $p_1 p_2 \ldots p_n \in R$.

Princip řízení v této gramatice byl představen v [6]. Tato gramatika je řízena tak, že posloupnost použitých pravidel při derivacích musí tvořit řetězec, který se nachází v regulární množině R. Gramatika je bez kontroly výskytu, pokud $F = \emptyset$. Kontrola výskytu znamená, že pokud je nějaké pravidlo v množině F a není ho možné aplikovat na větnou formu, může být vynecháno a může se pokračovat jiným pravidlem podle regulární množiny R. Pokud je $F = \emptyset$, v derivacích je možné vynechat horní index ac. Pokud platí $R = P^*$, posloupnost pravidel je libovolná a gramatika generuje jen bezkontextové jazyky.

Definice 3.2. Mějme regulárně řízenou gramatiku G s resp. bez kontroly výskytu. Gramatika G je bez vymazávacích pravidel, pokud neobsahuje ε-pravidla.

Zkratkami $\mathbf{rC}$, $\mathbf{rC_{ac}}$, $\mathbf{rC}^{\varepsilon}$, $\mathbf{rC}_{\mathbf{ac}}^{\varepsilon}$ označujeme po řadě třídy jazyků generovaných regulárně řízenou gramatikou bez resp. s kontrolou výskytu bez vymazávacích pravidel, regulárně řízenou gramatiku bez kontroly výskytu s vymazávacími pravidly, regulárně řízenou gramatiku s kontrolou výskytu s vymazávacími pravidly. Mezi nimi platí věta 3.1.

Věta 3.1. *([16], strana 106)*

(1) $\mathbf{CF} \subset \mathbf{rC} \subset \mathbf{rC_{ac}} \subset \mathbf{CS}$

(2) $\mathbf{CF} \subset \mathbf{rC} \subseteq \mathbf{rC}^{\varepsilon} \subset \mathbf{rC}_{\mathbf{ac}}^{\varepsilon} = \mathbf{RE}$

Poznámka: Přestože [16] uvádí $\mathbf{rC} \subset \mathbf{rC}^{\varepsilon}$, podle [10] není tento vztah dokázán.

Příklad 3.1. Mějme regulárně řízenou gramatiku $G_R = (N, T, P, S, R, F)$, kde jednotlivé množiny jsou definovány v tabulce 3.1.

Gramatika G_R generuje jazyk $L(G_R) = \{a^{2^n} \mid n \geq 0\}$. Neterminál S může vygenerovat buď terminál a nebo AA. Důležité je, že pravidlo 2 musí být aplikováno tolikrát, aby byly

množiny	i	p_i
$N = \{S, A, X\}$	(1)	$S \to a$
$T = \{a\}$	(2)	$S \to AA$
$P = \{1, 2, 3, 4, 5\}$	(3)	$A \to S$
$R = (2^*43^*5)^*1^*$	(4)	$S \to X$
$F = \{4, 5\}$	(5)	$A \to X$

Tabulka 3.1: Jednotlivé množiny a pravidla pro příklad 3.1

nahrazeny všechny výskyty S ve větné formě. Po pravidle 2 totiž následuje pravidlo 4, které testuje existenci neterminálu S. Pokud se ve větné formě vyskytuje, S je nahrazeno neterminálem X, který blokuje derivaci, neboť neexistuje žádné pravidlo, které by mělo na levé straně X. Totéž platí pro nahrazení všech výskytů A neterminálem S (pravidla 3 a 5). Aby tedy bylo vygenerováno slovo, pravidly 2 resp. 3 musí být vždy nahrazeny všechny výskyty S resp. A. Pravidla 4 resp. 5 vždy selžou a budou přeskočeny, neboť jsou v módu kontroly výskytu. Příklad vygenerování slova $a^{2^2} = aaaa$ tak vypadá takto:

$S \Rightarrow_2 AA \Rightarrow_4 AA \Rightarrow_{3,3} SS \Rightarrow_5 SS \Rightarrow_{2,2} AAAA \Rightarrow_4 AAAA \Rightarrow_{3,3,3,3} SSSS \Rightarrow_5 SSSS \Rightarrow_{1,1,1,1} aaaa$

Je vidět, že sekvence použitých pravidel odpovídá regulárnímu výrazu $(2^*43^*5)^*1^*$.

3.2 Maticová gramatika

Definice 3.3. Maticová gramatika (*matrix grammar*) s kontrolou výskytu je šestice $G = (N, T, M, S, F)$, kde:

- N, T, S jsou definovány jako u bezkontextových gramatik
- M je konečná množina sekvencí $\{m_1, \ldots, m_n\}$ pro $n \geq 1$, kde $m_i = (p_{i1}, \ldots, p_{ik(i)})$ pro $k(i) \geq 1$, $i \in \{1, \ldots, n\}$ a $j \in \{1, \ldots, k(i)\}$, přičemž p_{ij} je bezkontextové pravidlo.
- $F \subseteq \{p_{ij} \mid i \in \{1, \ldots, n\} \wedge j \in \{1, \ldots, k(i)\} \wedge m_i = (p_{i1}, \ldots, p_{ik(i)}) \wedge m_i \in M\}$

Mějme sekvenci m_i a větné formy $x, y \in (N \cup T)^*$. Platí derivace $x \Rightarrow_{m_i} y$, pokud existuje posloupnost derivací:

$$x = x_0 \Rightarrow^{ac}_{p_{i1}} x_1 \Rightarrow^{ac}_{p_{i2}} \cdots \Rightarrow^{ac}_{p_{ik(i)}} x_{k(i)} = y$$

Jazyk $L(G)$ je potom definován jako množina slov $w \in T^*$, pro která platí

$$S \Rightarrow_{m_{j_1}} y1 \Rightarrow_{m_{j_2}} \cdots \Rightarrow_{m_{j_s}} w$$

pro nějaké $s \geq 1$, $j_i \in \{1, \ldots, n\}$, $i \in \{1, \ldots, s\}$.

Tento typ gramatik byl představen v [1]. Generování slova probíhá tak, že je vybrána nějaká sekvence m_i (resp. matice m_i, která má jen jeden řádek) a poté musí být využita všechna pravidla v m_i v daném pořadí. V případě, že pro určité pravidlo platí $p_{ij} \in F$ a ve větné formě se nenachází levá strana pravidla, může být vynecháno a pokračuje se dalším

pravidlem v sekvenci. Pokud každá sekvence m_i obsahuje právě jedno pravidlo, potom gramatika generuje jen bezkontextové jazyky. Maticová gramatika může být s libovolnými pravidly i bez vymazávacích pravidel, podobně jako v definici 3.2. Pokud platí $F = \emptyset$, gramatika je bez kontroly výskytu a horní index ac je vynecháván.

Zkratkami $\mathbf{M}$, $\mathbf{M_{ac}}$, $\mathbf{M}^{\varepsilon}$ a $\mathbf{M}^{\varepsilon}_{\mathbf{ac}}$ označujeme po řadě třídy jazyků generovaných maticovou gramatikou bez resp. s kontrolou výskytu bez vymazávacích pravidel, maticovou gramatiku bez kontroly výskytu s vymazávacími pravidly, maticovou gramatiku s kontrolou výskytu s vymazávacími pravidly.

Jednotlivé typy maticových gramatik jsou ekvivalentní s jednotlivými typy regulárně řízených gramatik, tedy platí věta 3.2.

Věta 3.2. *([16], strana 110)*

(*1*) $\mathbf{M} = \mathbf{rC}$

(*2*) $\mathbf{M}^{\varepsilon} = \mathbf{rC}^{\varepsilon}$

(*3*) $\mathbf{M_{ac}} = \mathbf{rC_{ac}}$

(*4*) $\mathbf{M}^{\varepsilon}_{\mathbf{ac}} = \mathbf{rC}^{\varepsilon}_{\mathbf{ac}}$

Příklad 3.2. Mějme maticovou gramatiku $G_M = (N, T, M, S, F)$, kde jednotlivé množiny jsou definovány:

$N = \{S, A, B, C\}$
$T = \{a, b\}$
$F = \emptyset$
$M = \{m_1, m_2, m_3, m_4\}$

$$\begin{array}{lcllll} m_1 & = & (& S \to ABC &) \\ m_2 & = & (& A \to a, & C \to a, & B \to BB &) \\ m_3 & = & (& A \to aA, & C \to aC, & B \to BBB &) \\ m_4 & = & (& B \to b &) \end{array}$$

Gramatika G_M z příkladu 3.2 generuje jazyk $L(G_M) = \{a^n b^m a^n \mid n \geq 1 \wedge m = 2n\}$. Sekvence m_3 vždy přidá jeden symbol a na obou stranách větné formy a přidá dva neterminály B uprostřed. Sekvence m_2 ukončuje generování, přičemž přidá jeden neterminál B, aby celkově vycházelo $m = 2n$. Slovo $aabbbbaa$ je vygenerováno následujícím způsobem:

$S \Rightarrow_{p_{11}} ABC \Rightarrow_{p_{31}} aABC \Rightarrow_{p_{32}} aABaC \Rightarrow_{p_{33}} aABBBaC \Rightarrow_{p_{21}} aaBBBaC \Rightarrow_{p_{22}} aaBBBaa$
$\Rightarrow_{p_{23}} aaBBBBaa \Rightarrow_{p_{41}} aabBBBaa \Rightarrow_{p_{41}} aabbBBaa \Rightarrow_{p_{41}} aabbbBaa \Rightarrow_{p_{41}} aabbbbaa$

3.3 Programovaná gramatika

Definice 3.4. Programovaná gramatika (*programmed grammar*) s kontrolou výskytu je čtveřice $G = (N,\ T,\ P,\ S\ )$, kde:

- N, T, S jsou definovány jako u bezkontextové gramatiky
- P je konečná množina trojic $r = (p, \sigma, \varphi)$, kde p je bezkontextové pravidlo, σ a φ jsou konečné podmnožiny množiny P

Jazyk $L(G)$ generovaný touto gramatikou obsahuje všechna slova $w \in T^*$, pro která existuje derivace

$$S = w_0 \Rightarrow^{ac}_{r_1} w_1 \Rightarrow^{ac}_{r_2} w_2 \Rightarrow^{ac}_{r_3} \cdots \Rightarrow^{ac}_{r_k} w_k = w$$

pro $k \geq 1$, a zároveň pro každé pravidlo $r_i = (A_i \to v_i, \sigma_i, \varphi_i)$, $i \in \{1, \ldots, k-1\}$, existuje pravidlo $r_{i+1} \in P$ tak, že platí jedna z následujících podmínek:

(*1*) $w_{i-1} = x_{i-1}A_iy_{i-1}$, $w_i = x_{i-1}v_iy_{i-1}$ pro nějaké $x_{i-1}, y_{i-1} \in (N \cup T)^*$ a $r_{i+1} \in \sigma_i$
(*2*) A_i se nenachází v w_{i-1}, potom $w_{i-1} = w_i$ pokud $r_{i+1} \in \varphi_i$

Pokud pro každé pravidlo $r \in P$ platí $\varphi = \emptyset$, potom se jedná o gramatiku bez kontroly výskytu a horní index *ac* u derivací vynecháváme. Dále, pokud pro každé pravidlo platí $\sigma = P \wedge \varphi = \emptyset$, potom gramatika generuje jen bezkontextové jazyky. Gramatiku je možné použít jak s vymazávacími pravidly tak bez nich.

Programované gramatiky byly poprvé představeny v [15] a jejich funkčnost je následující. Pokud je v nějaké větné formě možné aplikovat nějaké pravidlo, toto pravidlo se použije a v další derivaci je možné použít nějaké pravidlo z množiny σ, nazvaná *úspěch* (*1*). Pokud není možné pravidlo aplikovat, je ignorováno a v další derivaci je možné použít nějaké pravidlo z množiny φ nazvané *neúspěch* (*2*).

Zkratkami $\mathbf{P}$, $\mathbf{P_{ac}}$, $\mathbf{P}^{\varepsilon}$, $\mathbf{P}^{\varepsilon}_{\mathbf{ac}}$ označujeme po řadě třídy jazyků generovaných programovanou gramatikou bez resp. s kontrolou výskytu bez vymazávacích pravidel, programovanou gramatiku bez kontroly výskytu s vymazávacími pravidly, programovanou gramatiku s kontrolou výskytu s vymazávacími pravidly.

Jednotlivé typy programovaných gramatik jsou ekvivalentní s jednotlivými typy maticových gramatik, tedy platí věta 3.3.

Věta 3.3. *([21], věta 3.3.1)*

(*1*) $\mathbf{P} = \mathbf{M}$

(*2*) $\mathbf{P}^{\varepsilon} = \mathbf{M}^{\varepsilon}$

(*3*) $\mathbf{P_{ac}} = \mathbf{M_{ac}}$

(*4*) $\mathbf{P}^{\varepsilon}_{\mathbf{ac}} = \mathbf{M}^{\varepsilon}_{\mathbf{ac}}$

V následujícím případě si představíme komplexní příklad, na kterém se dá dobře demonstrovat to, jakou nespornou výhodu mají programované gramatiky. Především se jedná o to, že v programovaných gramatikách je možné snadno použít dekompozici problému na podproblémy. Další dobrou vlastností je, že pravidla můžeme navazovat za sebou prostřednictvím neprázdné množiny *úspěch* a prázdné množiny *neúspěch* podobně jako sekvenční

příkazy v nějakém imperativním jazyce. Zároveň je snadné implementovat podmínky a cykly v případě neprázdnosti těchto množin.

Příklad 3.3. Mějme programovanou gramatiku s vymazávacími pravidly a s kontrolou výskytu $G_{n^k} = (\{K,N,N',N'',S,X,Y,Z\}, \{a\}, P, S)$, kde množina P obsahuje pravidla z tabulky 3.2.

i	r_i	σ_i	φ_i	popis
(1)	$S \to a$	$\emptyset$	$\emptyset$	vytvoření jednoho a
(2)	$S \to NX$	$\{3,4\}$	$\emptyset$	vygenerování větné formy $N^n K^{k-1}$
(3)	$X \to NX$	$\{3,4\}$	$\emptyset$	
(4)	$X \to NK^{k-1}$	$\{5\}$	$\emptyset$	
(5)	$N \to YN'$	$\{5\}$	$\{6\}$	vygenerování pomocných Y
(6)	$N' \to N$	$\{6\}$	$\{7\}$	
(7)	$K \to \varepsilon$	$\{8\}$	$\{17\}$	další součin dokud je K
(8)	$N \to N$	$\{9\}$	$\{16\}$	dokud existuje neterminál N
(9)	$Y \to Z$	$\{10\}$	$\emptyset$	jedno N nahraď tolika N'', kolik je neterminálů Y
(10)	$Y \to Z$	$\{11\}$	$\emptyset$	
(11)	$N \to N'N'$	$\{12\}$	$\emptyset$	
(12)	$Y \to Z$	$\{13\}$	$\{14\}$	
(13)	$N' \to N'N'$	$\{12\}$	$\emptyset$	
(14)	$N' \to N''$	$\{14\}$	$\{15\}$	
(15)	$Z \to Y$	$\{15\}$	$\{8\}$	
(16)	$N'' \to N$	$\{16\}$	$\{7\}$	všechny N'' na N
(17)	$N \to a$	$\{17\}$	$\{18\}$	konec algoritmu
(18)	$Y \to \varepsilon$	$\{18\}$	$\emptyset$	

Tabulka 3.2: Pravidla pro příklad 3.3 a jejich popis

Gramatika G_{n^k} generuje jazyk $L(G_{n^k}) = \{a^{n^k} \mid n \geq 1\}$ pro nějaké $k \geq 1$. Gramatika pracuje tak, že na začátku vygeneruje větnou formu $N^n K^{k-1}$ (k je vždy pro každou gramatiku konstantní). S každým odstraněním neterminálu K je proveden jeden součin, tedy každé N je nahrazeno tolika neterminály N, kolik jich bylo na začátku (parametr n). Přičemž kolik bylo neterminálů N na začátku, je uchováno počtem pomocných neterminálů Y.

Předpokládejme $k = 3$, G_{n^3}, tedy jazyk $L(G_{n^3}) = \{a^{n^3} \mid n \geq 1\}$. Gramatika G_{n^3} pro $n = 2$ vygeneruje slovo a^8 (pro přehlednost vynecháme horní index ac):

$S \Rightarrow_2 NX \Rightarrow_4 NNKK \Rightarrow_{5,5,5,6,6,6} YNYNKK \Rightarrow_7 YNYNK \Rightarrow_{8,9,10} ZNZNK \Rightarrow_{11} ZN'N'ZNK \Rightarrow_{12,14,14,14} ZN''N''ZNK \Rightarrow_{15,15,15} YN''N''YNK \Rightarrow_8 \cdots \Rightarrow_{15,15,15} YN''N''YN''N''K \Rightarrow_{8,16} YNN''YN''N''K \Rightarrow \cdots \Rightarrow_{16} YNNYNNK \Rightarrow \ldots$

Dále pokračujeme opět pravidlem 7, kde je každé N ve větné formě nahrazeno N^2:

$\cdots \Rightarrow YNNNNYNNNN \Rightarrow_{7,17} YaNNNYNNNN \Rightarrow \ldots \Rightarrow YaaaaYaaaa \Rightarrow_{17,18} aaaaYaaaa \Rightarrow_{18} aaaaaaaa$

Příklad 3.4. Mějme programovanou gramatiku bez kontroly výskytu $G = (\{S, X, A, B\}, \{a, b, c, x, y\}, P, S)$, kde množina P obsahuje pravidla z tabulky 3.3.

i	r_i	σ_i	φ_i
(1)	$S \to XBAA$	$\{2,3\}$	$\emptyset$
(2)	$B \to b$	$\{3\}$	$\emptyset$
(3)	$A \to aA$	$\{3,4\}$	$\emptyset$
(4)	$A \to c$	$\{5\}$	$\emptyset$
(5)	$A \to bA$	$\{5,6\}$	$\emptyset$
(6)	$A \to c$	$\{7\}$	$\emptyset$
(7)	$X \to x$	$\{8\}$	$\emptyset$
(8)	$B \to y$	$\emptyset$	$\emptyset$

Tabulka 3.3: Pravidla pro příklad 3.4

Pokud nebude existovat žádné omezení ohledně toho, jaký neterminál musí být přepsán v jedné derivaci, programovaná gramatika G generuje (regulární) jazyk $L(G)$ popsaný regulárním výrazem $x(b+y)(a^+(b^+ca^* + ca^*b^+) + b^+ca^+ + ca^+b^+)c$.

Například derivace: $S \Rightarrow_1 XBAA \Rightarrow_3 XBaAA \Rightarrow_4 XBaAc \Rightarrow_5 XBabAc \Rightarrow_6 XBabcc \Rightarrow_7 xBabcc \Rightarrow_8 xyabcc$

V následujících třech částech budou popsány typy nejlevějších derivací, které je možné u programovaných gramatik použít ([21], 4.1, 4.2, 4.3).

3.3.1 Nejlevější derivace typu 1

Nejlevější derivace *typu* 1 (striktní 1–kanonická derivace) je totožná s nejlevější derivací v bezkontextových gramatikách. Vždy tedy musí být přepsán nejlevější neterminál. Pokud tuto podmínku není v programované gramatice možné splnit, derivace je zablokována.

Definice 3.5. Mějme programovanou gramatiku G. Derivaci v gramatice G nazveme *nejlevější typu* 1, pokud každé pravidlo v derivaci přepíše vždy nejlevější neterminál v aktuální větné formě, nebo (v případě kontroly výskytu) nemůže být použito na nejlevější neterminál (pak ho označujeme za neaplikovatelné ve větné formě).

Zkratkou $L_{lm1}(G)$ budeme označovat jazyk generovaný gramatikou s nejlevější derivací *typu* 1. Zkratkami $\mathbf{P_{lm1}}$, $\mathbf{P_{lm1,ac}}$, $\mathbf{P^{\varepsilon}_{lm1}}$, $\mathbf{P^{\varepsilon}_{lm1,ac}}$ budeme označovat třídy jazyků generovaných programovanou gramatikou s nejlevější derivací *typu* 1. Pro uvedené typy gramatik platí věta 3.4 (všechny jsou ekvivalentní bezkontextovým jazykům).

Příklad 3.5. Mějme programovanou gramatiku G z příkladu 3.4. Pokud budeme uvažovat nejlevější derivaci *typu* 1, pro gramatikou G generovaný jazyk bude platit $L_{lm1}(G) = \emptyset$. Důvod je ten, že po aplikaci pravidla 1 vznikne větná forma $XBAA$, přičemž dále je možné použít pravidlo 2 nebo 3. Ovšem levé strany pravidel 2 a 3 neobsahují neterminál X, který musí být přepsán jako první. Derivace je tak zablokována.

Věta 3.4. *([21], věta 4.1.3)*

(*1*) $\mathbf{P_{lm1}} = \mathbf{P_{lm1,ac}} = \mathbf{P^{\varepsilon}_{lm1}} = \mathbf{P^{\varepsilon}_{lm1,ac}} = \mathbf{CF}$

3.3.2 Nejlevější derivace typu 2

V nejlevější derivaci *typu* 2 (obecná 1–kanonická derivace) musí být přepsán ten nejlevější neterminál, který přepsán být může.

Definice 3.6. Mějme programovanou gramatiku $G = (N,\ T,\ P,\ S)$. Mezi větnými formami $y, z \in (N \cup T)^*$ platí přímá derivace *typu* 2, pokud platí jedna z následujících podmínek:

- $y = S$ a existuje pravidlo $p_s = (S \to t, \sigma_s, \varphi_s) \in P$, potom $z = t$.
- Větná forma y byla získána z větné formy x za pomocí pravidla $p = (X \to a, \sigma_p, \varphi_p)$. Dále je možné použít pravidlo $p' = (A \to v, \sigma_{p'}, \varphi_{p'}) \in \sigma_p$, $y = y_1 A y_2$, a zároveň neexistuje pravidlo $p'' = (B \to w, \sigma_{p''}, \varphi_{p''}) \in \sigma_p$, $y = y_3 B y_4$ a $|y_3| < |y_1|$. Platí derivace $y \Rightarrow_{p'} z$, kde $z = y_1 v y_2$.
- Větná forma y byla získána z větné formy x za pomocí pravidla $p = (X \to a, \sigma_p, \varphi_p)$. Pokud neexistuje pravidlo $p' \in \sigma_p$, které by šlo aplikovat na větnou formu y, vybereme libovolné pravidlo $p' \in \sigma_p$ a pokračujeme libovolným pravidlem z množiny $\varphi_{p'}$.

Zkratkou $L_{lm2}(G)$ budeme označovat jazyk generovaný gramatikou s nejlevější derivací *typu 2*. Zkratkami $\mathbf{P_{lm2}}$, $\mathbf{P_{lm2,ac}}$, $\mathbf{P^{\varepsilon}_{lm2}}$, $\mathbf{P^{\varepsilon}_{lm2,ac}}$ budeme označovat třídy jazyků generovaných programovanou gramatikou s nejlevější derivací *typu* 2. Pro uvedené typy gramatik platí věta 3.5.

Příklad 3.6. Mějme programovanou gramatiku G z příkladu 3.4. Pokud budeme uvažovat nejlevější derivaci *typu* 2, jazyk $L_{lm2}(G)$ bude obsahovat všechny řetězce popsané regulárním výrazem xba^+cb^+c.

Například: $S \Rightarrow_1 XBAA \Rightarrow_2 XbAA \Rightarrow_3 XbaAA \Rightarrow_4 XbacA \Rightarrow_5 XbacbA \Rightarrow_5 XbacbbA \Rightarrow_6 Xbacbbc \Rightarrow_7 xbacbbc$

Věta 3.5. *([21], věta 4.2.3)*

(*1*) $\mathbf{P_{lm2}} = \mathbf{P_{lm2,ac}} = \mathbf{CS}$

(*2*) $\mathbf{P^{\varepsilon}_{lm2}} = \mathbf{P^{\varepsilon}_{lm2,ac}} = \mathbf{RE}$

3.3.3 Nejlevější derivace typu 3

V nejlevější derivaci *typu* 3 není výběr pravidla nijak omezen. Pokud ovšem vybereme nějaké pravidlo pro derivaci, je vždy nutné přepsat nejlevější výskyt levé strany vybraného pravidla.

Definice 3.7. Mějme programovanou gramatiku G. Derivace v gramatice G je *typu* 3, jestliže každé pravidlo, které při derivaci použijeme, vždy přepíše nejlevější výskyt své levé strany ve větné formě. Pokud se levá strana pravidla nenachází ve větné formě, přirozeně pokračujeme pravidly z množiny φ.

Zkratkou $L_{lm3}(G)$ budeme označovat jazyk generovaný gramatikou s nejlevější derivací *typu* 3. Zkratkami $\mathbf{P_{lm3}}$, $\mathbf{P_{lm3,ac}}$, $\mathbf{P^{\varepsilon}_{lm3}}$, $\mathbf{P^{\varepsilon}_{lm3,ac}}$ budeme označovat třídy jazyků generovaných programovanou gramatikou s nejlevější derivací *typu* 3. Pro uvedené typy gramatik platí věta 3.6.

Příklad 3.7. Mějme programovanou gramatiku G z příkladu 3.4. Při derivaci *typu* 3 bude jazyk $L_{lm3}(G)$ obsahovat všechny řetězce popsané regulárním výrazem $x(b+y)a^{+}cb^{+}c$.

Například: $S \Rightarrow_1 XBAA \Rightarrow_3 XBaAA \Rightarrow_4 XBacA \Rightarrow_5 XBacbA \Rightarrow_5 XBacbbA \Rightarrow_6 XBacbbc \Rightarrow_7 xBacbbc \Rightarrow_8 xyacbbc$

Zatímco v příkladu 3.6 bylo nutné po aplikaci pravidla 1 použít vždy pravidlo 2, v tomto typu nejlevější derivace je možné opravdu si zvolit buď pravidlo 2 nebo 3.

Věta 3.6. *([21], věta 4.3.3)*

(1) $\mathbf{P} \subset \mathbf{P_{lm3}} \subset \mathbf{P_{lm3,ac}} \subset \mathbf{CS}$

(2) $\mathbf{P}^{\varepsilon} \subset \mathbf{P^{\varepsilon}_{lm3}} \subset \mathbf{P^{\varepsilon}_{lm3,ac}} = \mathbf{RE}$

3.4 Gramatika s nahodilým kontextem

Definice 3.8. Gramatika s nahodilým kontextem (*random context grammar*) je čtveřice $G = (N, T, P, S)$, kde:

- N, T, S jsou definovány stejně jako u bezkontextových gramatik
- P je konečná množina trojic (p, R, Q), kde p je bezkontextové pravidlo a R, Q jsou disjunktní podmnožiny množiny N (po řadě povolující a zakazující kontext, anglicky *permitting* resp. *forbidding context*)

Mezi větnými formami $x, y \in (N \cup T)^*$ platí přímá derivace $x \Rightarrow y$, pokud $x = \alpha A \beta$ a $y = \alpha w \beta$ pro nějaké větné formy $\alpha, \beta \in (N \cup T)^*$, existuje pravidlo $(A \rightarrow w, R, Q) \in P$, každý neterminál z množiny R se **nachází** ve větné formě x a každý neterminál z množiny Q se **nenachází** ve větné formě x. Jazyk generovaný touto gramatikou je definován:

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

Tento typ gramatik byl poprvé zaveden van der Waltem v [19]. Od předchozích gramatik se liší tím, že zde nejsou předem dané sekvence pravidel, která mají být použity. Místo toho je používání pravidel řízeno tvarem větné formy. Každé pravidlo této gramatiky je vybaveno tzv. povolujícím a zakazujícím kontextem, což jsou disjunktní podmnožiny množiny

N. Platí, že pravidlo může být použito, pokud se všechny neterminály z povolujícího kontextu nacházejí ve větné formě a zároveň se všechny neterminály ze zakazujícího kontextu nenacházejí ve větné formě.

Gramatika může být jak s vymazávacími pravidly, tak i bez nich. Stav, kdy aspoň jedno pravidlo má neprázdný zakazující kontext, budeme označovat jako kontrola výskytu ac (přestože tato zkratka má odlišný význam než u předchozích gramatik). Variantu bez kontroly výskytu (zakazující kontext všech pravidel je prázdný) budeme označovat jako *povolující gramatiku*. Podobnou variantu, kdy povolující kontext všech pravidel je prázdný, budeme nazývat jako *zakazující gramatiku*. Při zápisu pravidel těchto dvou gramatik je potom možné vždy prázdnou množinu vynechávat (pravidla budou dvojice místo trojic).

Zkratkami $\mathbf{RC}$, $\mathbf{RC_{ac}}$, $\mathbf{RC}^{\varepsilon}$, $\mathbf{RC}^{\varepsilon}_{\mathbf{ac}}$ označujeme třídy jazyků generovaných gramatikou s nahodilým kontextem bez resp. s kontrolou výskytu bez vymazávacích pravidel, gramatikou s nahodilým kontextem bez kontroly výskytu s vymazávacími pravidly, gramatikou s nahodilým kontextem s kontrolou výskytu s vymazávacími pravidly.

Věta 3.7. *([16], strana 121)*

(*1*) $\mathbf{CF} \subset \mathbf{RC} \subset \mathbf{RC_{ac}} = \mathbf{M_{ac}} \subset \mathbf{RC}^{\varepsilon}_{\mathbf{ac}}$

(*2*) $\mathbf{RC} \subseteq \mathbf{RC}^{\varepsilon} \subset \mathbf{RC}^{\varepsilon}_{\mathbf{ac}} = \mathbf{M}^{\varepsilon}_{\mathbf{ac}}$

(*3*) $\mathbf{RC} \subseteq \mathbf{M}$

Příklad 3.8. Mějme gramatiku s nahodilým kontextem $G_C = (\{S, A_1, A_2, B, C_1, C_2\}, \{a, b, c\}, P, S)$, kde množina pravidel P je v tabulce 3.4. Gramatika G_C generuje jazyk $L(G_C) = \{a^n b^m c^n \mid m \geq n > 0\}$.

i	p	R	Q
(1)	$S \to A_1 C_1$	$\{S\}$	$\emptyset$
(2)	$A_1 \to aA_2$	$\{A_1\}$	$\{C_2\}$
(3)	$C_1 \to bC_2c$	$\{C_1, A_2\}$	$\emptyset$
(4)	$A_2 \to A_1$	$\{A_2, C_2\}$	$\emptyset$
(5)	$C_2 \to C_1$	$\{C_2, A_1\}$	$\emptyset$
(6)	$A_1 \to a$	$\{A_1\}$	$\{C_2\}$
(7)	$C_1 \to Bc$	$\{C_1\}$	$\{A_1, A_2\}$
(8)	$B \to bB$	$\{B\}$	$\emptyset$
(9)	$B \to b$	$\{B\}$	$\emptyset$

Tabulka 3.4: Pravidla pro příklad 3.8

Gramatika G_C na začátku vygeneruje větnou formu A_1C_1. A_1 generuje terminály a, C_1 generuje terminály b a c. Pomocné neterminály A_2 a C_2 zajistí, že po vygenerování a není možné vygenerovat další terminál a, dokud není vygenerováno i příslušné c. Slovo $a^2b^3c^2$ je možné vygenerovat následovně (dolní index relace $\Rightarrow$ obsahuje seznam pravidel, která je možné z dané větné formy použít, přičemž podtržení označuje použití pravidla při derivaci):

$S \Rightarrow_{\underline{1}} A_1C_1 \Rightarrow_{\underline{2},6} aA_2C_1 \Rightarrow_{\underline{3}} aA_2bC_2c \Rightarrow_{\underline{4}} aA_1bC_2c \Rightarrow_{\underline{5}} aA_1bC_1c \Rightarrow_{2,\underline{6}} aabC_1c \Rightarrow_{\underline{7}} aabBcc$
$\Rightarrow_{\underline{8},9} aabbBcc \Rightarrow_{8,\underline{9}} aabbbcc$

Levě/pravě povolující/zakazující gramatika

Povolující a zakazující gramatiku je možné modifikovat tak, že větná forma je kontrolována buď jen nalevo nebo napravo od přepisovaného neterminálu [7, 3]. Vzniknou tak čtyři možné varianty, které se liší jen v definici přímé derivace:

Mějme větné formy $x, y \in (N \cup T)^*$, kde $x = \alpha A \beta$, $y = \alpha w \beta$, a pravidlo $(A \to w, R, Q) \in P$. Platí relace přímé derivace $x \Rightarrow y$, pokud

- pro *levě povolující gramatiku* (značenou lRC_p) platí: $R \subseteq nonterm(\alpha)$
- pro *pravě povolující gramatiku* (značenou rRC_p) platí: $R \subseteq nonterm(\beta)$
- pro *levě zakazující gramatiku* (značenou lRC_f) platí: $nonterm(\alpha) \cap Q = \emptyset$
- pro *pravě zakazující gramatiku* (značenou rRC_f) platí: $nonterm(\beta) \cap Q = \emptyset$

přičemž zápis $nonterm(x)$ pro $x \in (N \cup T)^*$ vrací množinu všech neterminálů, které se vyskytují ve větné formě x. Vyjadřovací síla uvedených gramatik je uvedena ve větě 3.8. Je vhodné si všimnout zajímavého rozdílu, že levě resp. pravě povolující gramatiky jsou silnější než bezkontextové gramatiky. Ovšem levě resp. pravě zakazující gramatiky jsou ekvivalentní bezkontextovým gramatikám. Důvod je ten, že u levě resp. pravě zakazujících gramatik lze uvažovat nejlevější resp. nejpravější derivaci. Při této derivaci je však vždy zajištěno, že mohou být použita všechna pravidla, protože při uvažování nejlevější resp. nejpravější derivace se nalevo resp. napravo od přepisovaného neterminálu již nemůže vyskytovat jiný neterminál. Nejlevější resp. nejpravější derivace tak zajistí vygenerování všech slov jako nejlevější resp. nejpravější derivace v bezkontextové gramatice. A protože víme, že na druhu derivace v bezkontextové gramatice nezáleží, je jasné, že oba druhy zakazujících gramatik jsou ekvivalentní bezkontextovým gramatikám.

Věta 3.8. *([7], věta 1, důsledek 2; [3], úvod)*

(*1*) $\mathbf{CF} \subset \mathbf{lRC_p}, \mathbf{CF} \subset \mathbf{rRC_p}, \mathbf{CF} \subset \mathbf{lRC_p^\varepsilon}, \mathbf{CF} \subset \mathbf{rRC_p^\varepsilon}$

(*2*) $\mathbf{lRC_f} = \mathbf{lRC_f^\varepsilon} = \mathbf{rRC_f} = \mathbf{rRC_f^\varepsilon} = \mathbf{CF}$

Spolupracující distribuovaná gramatika

Definice 3.9. Spolupracující distribuovaná gramatika [7, 3] (*Cooperating Distributed Grammar*), zkráceně CD, je $(n+3)$-tice pro $n \geq 1$ $G_{CD} = (N, T, P_1, \ldots, P_n, S)$, kde N je konečná množina neterminálů, T je konečná množina terminálů, $S \in N$ je počáteční neterminál a $P_1, \ldots, P_n$ jsou množiny pravidel (komponenty) nosné gramatiky (viz dále).

Mějme větné formy $u, v \in (N \cup T)^*$. Nad komponentou P_i existuje derivace $u \Rightarrow_{P_i} v$, pokud platí $u \Rightarrow_{P_i} u_1 \Rightarrow_{P_i} u_2 \Rightarrow_{P_i} \cdots \Rightarrow_{P_i} u_k = v$ pro $k \geq 1$ a neexistuje taková větná forma w, že $v \Rightarrow_{P_i} w$. Jazyk G_{CD} je potom definován:

$$L(G_{CD}) = \{w \in T^* \mid S \Rightarrow_{P_{i_1}} w_1 \Rightarrow_{P_{i_2}} w_2 \ldots \Rightarrow_{P_{i_x}} w_x = w,\ 1 \leq i_j \leq n,\ 1 \leq j \leq x\}.$$

Gramatika funguje tak, že slovo může být po částech generováno v různých komponentách obecně v libovolném pořadí, přičemž platí, že pravidla z komponenty musí být aplikována tak dlouho, dokud to je možné. Pokud to již možné není, teprve potom lze vybrat jinou komponentu. Tento způsob přechodu mezi komponentami se nazývá *ukončující mód* (*terminating mode*). Existují i jiné způsoby přechodu, jako například že musí být provedeno *maximálně* k, *minimálně* k nebo *právě* k derivací v komponentách pro nějaké $k \geq 1$. Tyto další druhy přechodů mezi komponentami však v této práci nebudeme uvažovat.

Pro účely tento práce zde uvedeme jen ty CD gramatiky, které jako základ používají *levě povolující* a *levě zakazující* gramatiky. Tato kombinace má totiž tu vlastnost, že zvyšuje vyjadřovací sílu až na kontextové resp. rekurzivně vyčíslitelné jazyky bez resp. s vymazávacími pravidly, pokud počet komponent je aspoň 2, přičemž s vyšším počtem komponent vyjadřovací síla již neroste ([7, 3], důsledek 4, věta 8 , věta 6). Označme CD gramatiku používající levě povolující gramatiku bez resp. s vymazávacími pravidly jako CD^n_{lp} resp. $CD^n_{lp,\varepsilon}$ a CD gramatiku používající levě zakazující gramatiku bez resp. s vymazávacími pravidly jako CD^n_{lf} resp. $CD^n_{lf,\varepsilon}$ pro $n \geq 1$, kde n je počet komponent.

Věta 3.9. *([7], věta 1, důsledek 2; [3], úvod)*

(*1*) $\mathbf{CD^n_{lp}} = \mathbf{CD^n_{lf}} = \mathbf{CS}$ pro $n \geq 2$

(*2*) $\mathbf{CD^n_{lp,\varepsilon}} = \mathbf{CD^n_{lf,\varepsilon}} = \mathbf{RE}$ pro $n \geq 2$

Kapitola 4

Syntaktická analýza

V této kapitole si představíme metody syntaktické analýzy pro určité druhy gramatik probraných v kapitole 3. Budou zde probrány jak deterministické metody, které nevyžadují žádný způsob prohledávání stavového prostoru, tak i kombinované metody s prohledáváním stavového prostoru. U deterministických metod se výrazně sníží časová složitost, ale zároveň jsou pro tyto gramatiky vyžadovány určité dodatečné podmínky, které sice umožní prediktivitu, ale zároveň sníží jejich vyjadřovací sílu.

4.1 Programované gramatiky

Obsah této sekce vychází ze článku [17]. V něm je představena prediktivní syntaktické analýza těchto gramatik pracující na velmi podobném principu jako analýza $LL(k)$ gramatik. Je zde tabulka, pomocí níž se za určitých podmínek provádí predikce dalších pravidel gramatiky. Tento typ gramatiky je označován jako *jednoduchá programovaná gramatika* (*Simple Programmed Grammar*) $SPG(k)$, kde k má stejný význam jako u $LL(k)$ gramatik, tedy maximální počet symbolů nutných k predikci.

V této sekci bude představena modifikovaná varianta $SPG(k)$ nazvaná $ESPG(k)$ (*Extended Simple Programmed Grammar*). Ta upravuje jednu podmínku, která zajistí benevolentnější tvar určitých pravidel. Na základě této změny je nutné rovněž modifikovat algoritmus analýzy a přidat jednu další prediktivní tabulku. Pro začátek je třeba definovat zřetězení jazyků, které je využito v následujících algoritmech.

Definice 4.1. Mějme jazyky $L_1, \ldots, L_n$ nad abecedami $\Sigma_1, \ldots, \Sigma_n$ pro nějaké $n > 0$. Zřetězení jazyků $L_1, \ldots, L_n$ se zapisuje jako $L_1 \ldots L_n$ a je definováno jako:

$$L_1 \ldots L_n = \{x_1 \ldots x_n \mid \forall i \in \{1, \ldots, n\} : x_i \in L_i\}$$

Příklad 4.1. Mějme jazyky $L_1 = \{ab, a\}$, $L_2 = \{c, cc\}$, $L_3 = \{dd, de\}$ nad abecedami po řadě $\Sigma_1 = \{a, b\}$, $\Sigma_2 = \{c\}$, $\Sigma_3 = \{d, e\}$. Zřetězení jazyků $L_1L_2L_3 = \{abcdd, abcde, abccdd, abccde, acdd, acde, accdd, accde\}$.

K vytvoření prediktivních tabulek jsou využity množiny $FIRST_k$ a $FOLLOW_k$ [18]. Zde si představíme algoritmy výpočtu těchto množin pro libovolné $k > 0$. V nich je využita funkce $truncate_k(L)$, jejímž argumentem je jazyk L nad abecedou Σ a která vrací množinu řetězců oříznutých na délku maximálně k. Formálně zapsáno:

$$truncate_k(L) = \{a_1a_2\dots a_k \mid a_1, a_2, \dots, a_k \in \Sigma \wedge a_1a_2\dots a_kw \in L \wedge w \in \Sigma^*\} \cup \{w \mid w \in L \wedge |w| < k\}$$

Je nutné poznamenat, že výpočty množin $FIRST_k$ a $FOLLOW_k$ probíhají u bezkontextových gramatik i programovaných gramatik stejně. V případě programovaných gramatik jsou brány v úvahu jen tzv. základy pravidel neboli jen první prvek z trojice (bez množin *úspěch* a *neúspěch*). Máme-li pravidlo $r = (A \to w, \sigma, \varphi)$, tak základ pravidla r je $A \to w$ a budeme toto označovat jako $BASE(r)$. Máme-li programovanou gramatiku G, tak jako $BASE(G)$ budeme označovat bezkontextovou gramatiku, která obsahuje jen základy pravidel z G.

Množiny $FIRST_k$ jsou počítány jak pro terminální tak i pro neterminální symboly. Mějme bezkontextovou gramatiku $G = (N, T, P, S)$. Pro nějaké $A \in N \cup T$ množina $FIRST_k(A)$ obsahuje takové řetězce nad T, které je možné vygenerovat ze symbolu A na prvních k pozicích. Formálně je to možné zapsat:

$$FIRST_k(A) = truncate_k(\{w \in T^* \mid A \Rightarrow^* w\})$$

Jestliže máme programovanou gramatiku G, algoritmus 4.1 bude pracovat nad $BASE(G)$ iterativně tak, že neustále počítá množiny $FIRST_k$ tak dlouho, dokud se aktuálně spočítané množiny F liší od množin F'.

Vstup: bezkontextová gramatika $G = (N, T, P, S)$, $k > 0$
Výstup: množina $FIRST_k(A)$ pro každé $A \in N \cup T$

(1) **for each** $a \in T$ **do** $F(a) \leftarrow \{a\}$
(2) $F'(a) \leftarrow F(a)$
(3) **for each** $A \in N$ **do** $F(A) \leftarrow \emptyset$
(4) **if** $A \to \varepsilon \in P$ **do** $F(A) \leftarrow \{\varepsilon\}$
(5) **repeat**
(6) **for each** $A \in N$ **do** $F'(A) \leftarrow F(A)$
(7) **for each** $A \to a_1a_2\dots a_n \in P \wedge a_1, a_2, \dots, a_n \in N \cup T \wedge n > 0$ **do**
(8) $F(A) \leftarrow F(A) \cup truncate_k(F'(a_1)F'(a_2)\dots F'(a_n))$
(9) **until** $F(A) = F'(A)$ **for each** $A \in N$
(10) **for each** $A \in N \cup T$ **do** $FIRST_k(A) \leftarrow F(A)$

Algoritmus 4.1. Výpočet množin $FIRST_k$

Algoritmus 4.1 pracuje jen nad jedním terminálem nebo neterminálem. Nicméně je dále třeba zavést množinu $FIRST_k$ nad řetězcem terminálů a neterminálů

pro $n > 0$:
$w = a_1\dots a_n$, $w \in (N \cup T)^+$: $FIRST_k(w) = truncate_k(FIRST_k(a_1)\dots FIRST_k(a_n))$

pro $n = 0$:
$FIRST_k(\varepsilon) = \{\varepsilon\}$

a nad množinou X řetězců terminálů a neterminálů

$$FIRST_k(X) = \bigcup^{x \in X} FIRST_k(x)$$

Množiny $FOLLOW_k$ jsou počítány jen pro neterminály. Množina $FOLLOW_k(A)$ pro nějaký neterminál $A \in N$ obsahuje takové řetězce nad T s maximální délkou k, které se ve větné formě mohou nacházet přímo za neterminálem A. Formálně lze zapsat jako:

$$FOLLOW_k(A) = \{w \in FIRST_k(y) \mid S \Rightarrow^* xAy\}$$

Mějme opět programovanou gramatiku G. Algoritmus 4.2 bude pracovat nad $BASE(G)$ opět iterativně, dokud se množiny FL liší od množin FL'.

Vstup: bezkontextová gramatika $G = (N, T, P, S)$
množina $FIRST_k(X)$ pro každé $X \in N \cup T$
Výstup: množina $FOLLOW_k(A)$ pro každé $A \in N$

(1) $FL(S) \leftarrow \{\varepsilon\}$
(2) **for each** $A \in N - \{S\}$ **do** $FL(A) \leftarrow \emptyset$
(3) **repeat**
(4) **for each** $A \in N$ **do** $FL'(A) \leftarrow FL(A)$
(5) **for each** $A \rightarrow a_1 a_2 \ldots a_n \in P \wedge a_1, a_2, \ldots, a_n \in N \cup T \wedge n > 0$ **do**
(5) $L \leftarrow FL'(A)$
(6) **if** $a_n \in N$ **do** $FL(a_n) \leftarrow FL(a_n) \cup L$
(7) **for** $i \leftarrow n - 1$ **downto** 1 **do**
(8) $L \leftarrow truncate_k(FIRST_k(a_{i+1})L)$
(9) **if** $a_i \in N$ **do** $FL(a_i) \leftarrow FL(a_i) \cup L$
(10) **until** $FL(A) = FL'(A)$ **for each** $A \in N$
(11) **for each** $A \in N$ **do** $FOLLOW_k(A) \leftarrow FL(A)$

Algoritmus 4.2. Výpočet množin $FOLLOW_k$

V další části budeme používat následující označení. Mějme bezkontextové pravidlo $r = A \rightarrow w$, potom jako $LHS(r)$ budeme označovat levou stranu pravidla r ($LHS(r) = A$) a jako $RHS(r)$ pravou stranu pravidla r ($RHS(r) = w$).

4.1.1 $ESPG(k)$ gramatika

Definice 4.2. Mějme programovanou gramatiku $G = (N, T, P, S)$. G je *rozšířená jednoduchá programovaná gramatika* $ESPG(k)$ (*Extended Simple Programmed Grammar*) pro nějaké $k > 0$, pokud $BASE(G)$ je bez cyklů, pro G platí

$$r_1 = (S \rightarrow w, \sigma_1, \varphi_1) \in P \wedge \forall r \in P - \{r_1\} : LHS(BASE(r)) \neq S$$

a pro každé pravidlo $r = (X \rightarrow z, \sigma, \varphi)$, pro které $|\sigma| > 1$ resp. $|\varphi| > 1$, platí:

1. každé pravidlo v množině σ resp. φ má stejnou levou stranu
2. sjednocení množin *neúspěchu* všech pravidel v σ resp. φ má *maximálně* 1 prvek
3. pro každá dvě pravidla $r_x = (A \rightarrow \alpha, \sigma_x, \varphi_x)$ a $r_y = (A \rightarrow \beta, \sigma_y, \varphi_y)$ z množiny σ resp. φ platí:
   (a) A je nejlevější symbol ve větné formě
   (b) $FIRST_k(\{\alpha\}FOLLOW_k(A)) \cap FIRST_k(\{\beta\}FOLLOW_k(A)) = \emptyset$

Jako $\mathbf{ESPG(k)}$, $\mathbf{ESPG_{ac}(k)}$, $\mathbf{ESPG^{\varepsilon}(k)}$ a $\mathbf{ESPG^{\varepsilon}_{ac}(k)}$ budeme označovat třídy jazyků generovaných $ESPG(k)$ gramatikou po řadě bez vymazávacích pravidel bez kontroly výskytu, bez vymazávacích pravidel s kontrolou výskytu, s vymazávacími pravidly bez kontroly výskytu a s vymazávacími pravidly s kontrolou výskytu.

Bezkontextová gramatika $BASE(G)$ je bez cyklů, pokud v ní neexistuje derivace $A \Rightarrow^{+} A$ pro každé $A \in N$ [23]. Pokud by nějaký cyklus existoval, gramatika by byla víceznačná a nebylo by možné provést deterministickou analýzu. Podmínka 1 musí platit, protože se vždy musíme rozhodovat nad právě jedním neterminálem, který navíc musí být podle $3.a$ vždy nejlevější. Pokud bychom měli například množinu $\sigma = \{2, 3\}$ a obě pravidla v ní by měla různé levé strany a tyto strany by se ve větné formě nacházely, neexistoval by způsob, jak se mezi těmito dvěma pravidly rozhodnout. Podmínka $3.b$ je stejná podmínka, kterou se vyznačují $LL(k)$ gramatiky. Podmínka 2 je již zmíněná modifikace $SPG(k)$ gramatik. Původně se v $SPG(k)$ nacházela podmínka:

$$\forall r_i \in \sigma : \varphi_i = \emptyset \text{ resp. } \forall r_i \in \varphi : \varphi_i = \emptyset$$

Tato podmínka vždy vyžadovala, aby se v případě predikce vždy provedla derivace.

Vstup: $ESPG(k)$ gramatika $G = (N, T, P, S)$, $k > 0$
Výstup: tabulky **TAB1**, **TAB2**

```
(1)  for i ← 1 upto |P| do
(2)      Go ← (σ_i, φ_i)
(3)      Part ← (Success, Failure)
(4)      for j ← 1 upto 2 do
(5)          ROW ← 1
(6)          if |Go[j]| = 1 do
(7)              TAB1[i](1, NEXT(Part[j])) ← Go[j][1]
(8)              TAB1[i](1, LOOK(Part[j])) ←„any“
(9)          elseif |Go[j]| > 1 do
(10)             INDEX ← ∅
(11)             for each p_k ∈ Go[j] do
(12)                     NONT ← LHS(p_k)
(13)                     if |φ_k| > 0 do INDEX ← INDEX ∪ {k}
(14)                     LH ← FIRST_k({RHS(p_k)}FOLLOW_k(LHS(p_k)))
(15)                     for each L ∈ LH do
(16)                             TAB1[i](ROW, NEXT(Part[j])) ← k
(17)                             TAB1[i](ROW, LOOK(Part[j])) ← L
(18)                             ROW ← ROW + 1
(19)             TAB2[i](Part[j], Lhs) ← NONT
(20)             TAB2[i](Part[j], Src) ← INDEX
```

Algoritmus 4.3. Výpočet prediktivních tabulek **TAB1**, **TAB2**

Podmínka 2 v $ESPG(k)$ umožňuje, že v případě, že příslušný neterminál není nalezen ve větné formě, pokračuje se pravidlem z množiny, která vznikne sjednocením množin *neúspěch*

pravidel v σ resp. φ (rozhodnutí je možné, protože toto sjednocení musí mít maximálně jeden prvek).

Algoritmus 4.3 vytváří tabulky **TAB1** a **TAB2** zároveň. Tabulka **TAB1** má stejný význam jako prediktivní tabulka u $LL(k)$ gramatik. První sloupec (viz například tabulka 4.3) označuje číslo subtabulky. Prakticky se jedná o číslo pravidla, pro které je tato subtabulka určena. Druhý sloupec označuje jen číslo řádku v subtabulce. Sloupec **LOOK** obsahuje predikce o délce maximálně k symbolů. Speciální záznam „any“ říká, že predikce je libovolná. Sloupec **NEXT** obsahuje číslo pravidla, které bude dále použito v analýze. Vyhledání je provedeno ve sloupci *Success* nebo *Failure* podle toho, zda-li bylo poslední pravidlo aplikováno či ne (kontrola výskytu). Předpokládejme zápis $TAB1[i](j, NEXT(Success))$, tak i označuje číslo subtabulky, j řádek v subtabulce a $NEXT(Success)$ sloupec *Success* ve sloupci **NEXT**.

Pro potřeby modifikace $ESPG(k)$ gramatiky bylo rovněž nutné přidat další prediktivní tabulku **TAB2**. Ta je při analýze použita jen v případě, že je třeba provést predikci podle tabulky **TAB1** a zároveň se příslušný neterminál nenachází ve větné formě. První sloupec (viz například tabulka 4.2) je číslo pravidla. Tabulka obsahuje jen ta pravidla, která obsahují v množině *úspěch* nebo *neúspěch* více jak jedno pravidlo. V případě, že při predikci není daný neterminál (sloupec *Lhs*) ve větné formě, analýza pokračuje libovolným pravidlem z množiny ve sloupci *Src*. Pokud je množina prázdná, analýza končí chybou. Sloupce *Success* a *Failure* rozlišují predikce pro množiny *úspěch* a *neúspěch* daného pravidla. Zápis $TAB2[i](Failure, Src)$ adresuje řádek i, sloupec *Src* ve sloupci *Failure*.

název funkce	**popis**	**vrací**
$LEFT(f)$	vrátí nejlevější symbol ve větné formě f	terminál nebo neterminál
$REMOVELEFT(f)$	odstraní nejlevější symbol ve větné formě f	nic
$LEFTFIND(f, i, t)$	vyhledá nejlevější výskyt neterminálu A ve větné formě f, kde $A = TAB2[i](t, Lhs)$	pokud je A neplatné nebo A není ve větné formě, funkce vrací $false$, jinak vrací $true$
$SEARCHTABLE(w, i, t)$	vyhledá řetězec w v tabulce $TAB1$ v podtabulce číslo i ve sloupci $LOOK(t)$	vrátí číslo nalezeného řádku (> 0); v případě nenalezení vrátí 0
$REPLACE(f, i)$	nahradí nejlevější výskyt neterminálu $LHS(BASE(r_i))$ ($r_i \in P$) ve větné formě f větnou formou $RHS(BASE(r_i))$	v případě úspěchu vrátí $true$, jinak vrátí $false$

Tabulka 4.1: Pomocné funkce pro algoritmus 4.4

Algoritmus 4.3 pracuje pro každé pravidlo (řádek 1). Predikce může probíhat pro množinu *úspěch* a *neúspěch* zvlášť. Výpočet tabulek pro obě množiny je stejný a liší se jen ve sloupcích *Success* a *Failure*. Proto je zaveden cyklus na řádku 4, který provádí výpočet pro obě množiny. Pokud daná množina obsahuje právě jeden prvek (řádek 6), k predikci prakticky nedochází a je tedy do tabulky **TAB1** vložen speciální záznam „any“ s pokračováním jediným pravidlem z množiny. Pro množiny s více jak jedním prvkem (řádek 9) je již třeba provádět predikci. Pro každé pravidlo z množiny (řádek 11) je nutné spočítat predikci podle definice 4.2, 3.b. Obecně může existovat více predikcí, které „směřují“ ke stejnému pravidlu. Neboli množina LH na řádku 14 může obsahovat 0, 1 či více řetězců. Pro každý z nich je

potřeba vložit záznam do tabulky **TAB1**.

Dále je třeba připomenout, že algoritmus 4.3 jen vytváří příslušné tabulky a neprovádí kontrolu správnosti gramatiky podle definice 4.2. Algoritmus 4.4 je algoritmus, který provádí syntaktickou analýzu $ESPG(k)$ gramatik. V něm je použito několik funkcí, jejichž popis je v tabulce 4.1. Aktuální větná forma analyzátoru je uchovávána v proměnné $Form$, která by mohla být typu seznam. V $LL(k)$ analýze je obvykle použit zásobník, protože je přistupováno jen k jeho vrcholu. Nicméně v algoritmu 4.4 je třeba přistupovat k libovolnému prvku, protože derivace, u které není nutná predikce, může být provedena kdekoliv ve větné formě. Algoritmus vyžaduje existenci pravidla r_1 podle definice 4.2. Pravidlo r_1 je totiž aplikováno implicitně vždy a algoritmus začíná s větnou formou rovné pravé straně pravidla r_1 (řádek 1).

Vstup: prediktivní tabulky **TAB1**, **TAB2**
$ESPG(k)$ gramatika $G = (N, T, P, S)$ pro $k > 0$
slovo $w = a_1 a_2 \dots a_n \in T^*$ pro $n \geq 0$
Výstup: ANO, $w \in L(G)$ (seznam pravidel na výstupu)
NE, $w \notin L(G)$

// uchovává větnou formu při derivaci, $r_1 \in P$
(1) $Form \leftarrow RHS(BASE(r_1))$
(2) $i \leftarrow 1$
(3) $Fail \leftarrow false$
(4) $Use \leftarrow Success$
(5) $NextRule \leftarrow 1$
// $|LookAhead| < k$, v případě dosažení konce
(6) $LookAhead \leftarrow a_1 \dots a_k$
(7) **Output** $NextRule$

(8) **while** $Fail = false \wedge |Form| > 0$ **do**
(9) **if** $LEFT(Form) \in T$ **do**
(10) **if** $LEFT(Form) = a_i$ **do**
(11) $i \leftarrow i + 1$
(12) $REMOVELEFT(Form)$
// $|LookAhead| < k$, v případě dosažení konce
(13) $LookAhead \leftarrow a_i \dots a_{i+k-1}$
(14) **else do**
(15) $Fail \leftarrow true$
(16) **else do**
(17) **if exists** $TAB2[NextRule]$ **do**
(18) **if** $LEFTFIND(Form, NextRule, Use) = false$ **do**
(19) $Cont \leftarrow TAB2[NextRule](Use, Src)$
(20) **if** $Cont = \emptyset$ **do**
(21) $Fail \leftarrow true$
(22) **else do**
(23) $Use \leftarrow Failure$
(24) **Output** $Cont$
// zde první prvek, obecně však libovolný prvek z Cont

```
(25)                        NextRule ← Cont[1]
(26)            if  Fail = false do
(27)                Row ← SEARCHTABLE(LookAhead, NextRule, Use)
(28)                if  Row ≠ 0 do
(29)                    NextRule ← TAB1[NextRule](Row, NEXT(Use))
(30)                    if  REPLACE(Form, NextRule) = true do
(31)                        Output NextRule
(32)                        Use ← Success
(33)                    else do
(34)                        Use ← Failure
(35)                else do
(36)                    Fail ← true

(37)  if  Form = ε ∧ LookAhead = ε ∧ Fail = false do
(38)      return ANO
(39)  else do
(40)      return NE
```

Algoritmus 4.4. Algoritmus syntaktické analýzy $ESPG(k)$ gramatiky

Příklad 4.2. V následujících tabulkách 4.2 a 4.3 je popsána $ESPG(2)$ gramatika $G = (\{S', S, A, B, C, X, Y\}, \{a, b, c, 0, 1\}, P, S')$ spolu s prediktivními tabulkami. G je modifikací jazyka $\{a^n b^n c^n \mid n > 0\}$ z [17], kde:

1. v každém podřetězci a^n, b^n, c^n je rozmístěno $n - 1$ nul a jedniček
2. po jedničce nesmí následovat další jednička
3. celá sekvence může být opakována

Oproti původní gramatice v [17] jsou modifikována pravidla 3, 4, 5 (přidán neterminál X), pravidlo 8 (přidán neterminál Y, který umožňuje sekvenci opakovat) a přidána pravidla 9-13. V gramatice je záměrně prezentována nová vlastnost $ESPG(k)$, a to v pravidlech 8, 9, 10, kde množiny *neúspěchu* pravidel 9, 10 obsahují pravidlo 11.

Gramatika funguje tak, že během derivování terminálů a, b, c se generují i neterminály X. Po aplikaci pravidla 8 jsou poté všechna X postupně nahrazována terminály 0 nebo 1. Po nahrazení posledního X je pravidlo 9 nebo 10 neúspěšné a provede se pravidlo 11, kterým se sekvence derivování může opakovat (pravidlo 13) nebo ukončit (pravidlo 12).

Smysl nové tabulky **TAB2** je v tom, aby byla uložena informace o tom, kterými pravidly (0 či více) je možné potenciálně pokračovat množinou *neúspěch* v případě, že neterminál neexistuje ve větné formě. Příkladem může být pravidlo 8. V jeho množině *úspěch* jsou pravidla 9, 10, která mají na levé straně neterminál X (řádek 8, sloupec *Success*, *Lhs*=X). Obě pravidla 9, 10 mají množinu *neúspěch* $\{11\}$. Tedy pokud neterminál X neexistuje ve větné formě, je možné použít pravidlo 9 nebo 10, které není možné aplikovat, a je možné pokračovat pravidlem 11 (řádek 8, sloupec *Success*, *Src*=$\{9, 10\}$).

V tabulce 4.4 jsou uvedeny množiny $FIRST_2$ a $FOLLOW_2$, ze kterých byly vytvořeny tabulky **TAB1** a **TAB2** pro příklad 4.2 algoritmem 4.3.

i	r_i	σ_i	φ_i	**TAB2**	*Success*		*Failure*	
				RULE	*Lhs*	*Src*	*Lhs*	*Src*
(1)	$S' \to S$	$\{2\}$	$\emptyset$	2	A	$\emptyset$		
(2)	$S \to ABC$	$\{3,6\}$	$\emptyset$	5	A	$\emptyset$		
(3)	$A \to aAX$	$\{4\}$	$\emptyset$	8	X	$\{9,10\}$		
(4)	$B \to bXB$	$\{5\}$	$\emptyset$	9	X	$\{9,10\}$		
(5)	$C \to cXC$	$\{3,6\}$	$\emptyset$	11	Y	$\emptyset$		
(6)	$A \to a$	$\{7\}$	$\emptyset$					
(7)	$B \to b$	$\{8\}$	$\emptyset$					
(8)	$C \to cY$	$\{9,10\}$	$\emptyset$					
(9)	$X \to 0$	$\{9,10\}$	$\{11\}$					
(10)	$X \to 1$	$\{9\}$	$\{11\}$					
(11)	$Y \to Y$	$\{12,13\}$	$\emptyset$					
(12)	$Y \to \varepsilon$	$\emptyset$	$\emptyset$					
(13)	$Y \to S$	$\{2\}$	$\emptyset$					

Tabulka 4.2: Pravidla gramatiky z příkladu 4.2 a tabulka **TAB2**

TAB1		**LOOK**		**NEXT**		**TAB1**		**LOOK**		**NEXT**	
Sub	*Row*	*Succ*	*Fail*	*Succ*	*Fail*	*Sub*	*Row*	*Succ*	*Fail*	*Succ*	*Fail*
1	1	„any“		2		8	6	1c		10	
2	1	aa		3			7	10		10	
	2	a0		6			8	11		10	
	3	a1		6		9	1	0b	„any“	9	11
	4	ab		6			2	0c		9	
3	1	„any“		4			3	00		9	
4	1	„any“		5			4	01		9	
5	1	aa		3			5	1b		10	
	2	a0		6			6	1c		10	
	3	a1		6			7	10		10	
	4	ab		6			8	11		10	
6	1	„any“		7		10	1	„any“	„any“	9	11
7	1	„any“		8		11	1	ε		12	
8	1	0b		9			2	aa		13	
	2	0c		9			3	ab		13	
	3	00		9		12	1				
	4	01		9		13	1	„any“		2	
	5	1b		10							

Tabulka 4.3: Tabulka **TAB1** k příkladu 4.2 (*Succ = Success, Fail = Failure*)

$\forall x \in T : FIRST_2(x) = \{x\}$	$FOLLOW_2(S') = \{\varepsilon\}$
$FIRST_2(S') = \{aa, ab\}$	$FOLLOW_2(S) = \{\varepsilon\}$
$FIRST_2(S) = \{aa, ab\}$	$FOLLOW_2(A) = \{b0, b1, bc, 0b, 1b, 00, 01, 10, 11\}$
$FIRST_2(A) = \{a, aa\}$	$FOLLOW_2(B) = \{c0, c1, c, ca\}$
$FIRST_2(B) = \{b, b0, b1\}$	$FOLLOW_2(C) = \{\varepsilon\}$
$FIRST_2(C) = \{c, c0, c1, ca\}$	$FOLLOW_2(X) = \{b0, b1, bc, 0b, 1b, c0, c1, ca, c, 00, 01, 10, 11\}$
$FIRST_2(X) = \{0, 1\}$	$FOLLOW_2(Y) = \{\varepsilon\}$
$FIRST_2(Y) = \{\varepsilon, aa, ab\}$	
pravidlo (1)	$FIRST_2(\{S\}FOLLOW_2(S')) = \{aa, ab\}$
pravidlo (2)	$FIRST_2(\{ABC\}FOLLOW_2(S)) = \{aa, ab\}$
pravidlo (3)	$FIRST_2(\{aAX\}FOLLOW_2(A)) = \{aa\}$
pravidlo (4)	$FIRST_2(\{bXB\}FOLLOW_2(B)) = \{b0, b1\}$
pravidlo (5)	$FIRST_2(\{cXC\}FOLLOW_2(C)) = \{c0, c1\}$
pravidlo (6)	$FIRST_2(\{a\}FOLLOW_2(A)) = \{ab, a0, a1\}$
pravidlo (7)	$FIRST_2(\{b\}FOLLOW_2(B)) = \{bc\}$
pravidlo (8)	$FIRST_2(\{cY\}FOLLOW_2(C)) = \{c, ca\}$
pravidlo (9)	$FIRST_2(\{0\}FOLLOW_2(X)) = \{0b, 00, 01, 0c\}$
pravidlo (10)	$FIRST_2(\{1\}FOLLOW_2(X)) = \{1b, 10, 11, 1c\}$
pravidlo (11)	$FIRST_2(\{Y\}FOLLOW_2(Y)) = \{\varepsilon, aa, ab\}$
pravidlo (12)	$FIRST_2(\{\varepsilon\}FOLLOW_2(Y)) = \{\varepsilon\}$
pravidlo (13)	$FIRST_2(\{S\}FOLLOW_2(Y)) = \{aa, ab\}$

Tabulka 4.4: Množiny $FIRST_2$ a $FOLLOW_2$ pro příklad 4.2

Tabulka 4.5 zobrazuje práci algoritmu pro správný vstup abc pro příklad 4.2. Algoritmus skončí aplikací pravidla 12, kdy je větná forma přepsána na ε a zároveň je vstup celý přečten.

Sloupec $Form$ obsahuje aktuální větnou formu, **LEFT** označuje typ nejlevějšího symbolu ve větné formě, **predikce** obsahuje záznam tabulky 4.3 na dané pozici a **vstup** je aktuálně nepřečtená část vstupu, přičemž je podtržen prefix o délce maximálně 2, který je použit pro vyhledávání.

Další tabulka 4.6 tentokrát zobrazuje činnost algoritmu pro chybný vstup aa. Algoritmus skončí pokusem nalézt predikci a v tabulce 4.3 v subtabulce 5.

4.1.2 Složitost $ESPG(k)$

Složitost algoritmu $SPG(k)$ je kvadratická vzhledem k délce vstupu, přičemž je předpoklad, že $SPG(k)$ neobsahuje vymazávací pravidla. Každá jedna derivace má lineární složitost, neboť je vždy třeba vyhledávat nejlevější výskyt nějakého neterminálu. Počet derivací je ohraničen délkou vstupu, tedy celkově je složitost kvadratická, pokud neuvažujeme režii výhledávání v prediktivní tabulce. V případě existence vymazávacích pravidel může být složitost vyšší.

Algoritmus 4.4 k $ESPG(k)$ gramatice bude mít vyšší složitost, protože je nutné vyhledávat v tabulce **TAB2** a také proto, že nejlevější neterminál může být v některých případech vyhledáván zbytečně dvakrát (viz funkce *LEFTFIND* a *REPLACE*). Nicméně tuto nevýhodu lze na implementační úrovni eliminovat, takže lze říci, že složitost je zvýšena jen o prohledávání tabulky **TAB2**.

Form	**LEFT**	*NextRule*	*Use*	**predikce**	**vstup**
S	Net.	1	*Success*	„any“	$\underline{ab}c$
ABC	Net.	2	*Success*	ab	$\underline{ab}c$
aBC	Term.	6	*Success*	-	$\underline{ab}c$
BC	Net.	6	*Success*	„any“	$\underline{bc}$
bC	Term.	7	*Success*	-	$\underline{bc}$
C	Net.	7	*Success*	„any“	$\underline{c}$
cY	Term.	8	*Success*	-	$\underline{c}$
Y	Net.	8	*Success*	$TAB2[8](Success, Src) = \{9, 10\}$	$\underline{\varepsilon}$
Y	Net.	{9,10}	*Failure*	„any“	$\underline{\varepsilon}$
Y	Net.	11	*Success*	ε	$\underline{\varepsilon}$
ε	Term.	12	*Success*	-	$\underline{\varepsilon}$

Tabulka 4.5: Úspěšná syntaktická analýza pro příklad 4.2

Form	**LEFT**	*NextRule*	*Use*	**predikce**	**vstup**
S	Net.	1	*Success*	„any“	$\underline{aa}$
ABC	Net.	2	*Success*	aa	$\underline{aa}$
$aAXBc$	Term.	3	*Success*	-	$\underline{aa}$
$AXBc$	Net.	3	*Success*	„any“	$\underline{a}$
$AXbXBC$	Net.	4	*Success*	„any“	$\underline{a}$
$AXbXBcXC$	Net.	5	*Success*	-nenalezeno-	$\underline{a}$

Tabulka 4.6: Neúspěšná syntaktická analýza pro příklad 4.2

4.1.3 Vyjadřovací síla $ESPG(k)$

$SPG(k)$ a $ESPG(k)$ gramatiky jsou striktně slabší než programované gramatiky, tedy platí věta 4.1.

Věta 4.1. *(důkaz viz dále)*

(*1*) $\mathbf{ESPG(k) = SPG(k) \subset P}$

(*2*) $\mathbf{ESPG_{ac}(k) = SPG_{ac}(k) \subset P_{ac}}$

(*3*) $\mathbf{ESPG^{\varepsilon}(k) = SPG^{\varepsilon}(k) \subset P^{\varepsilon}}$

(*4*) $\mathbf{ESPG^{\varepsilon}_{ac}(k) = SPG^{\varepsilon}_{ac}(k) \subset P^{\varepsilon}_{ac}}$

Například pro gramatiku G_{n^k} z příkladu 3.3 platí $L(G_{n^k}) \in \mathbf{P^{\varepsilon}_{ac}}$, ale $L(G_{n^k}) \notin \mathbf{ESPG^{\varepsilon}_{ac}(k)}$, protože v této gramatice není zřejmě možné provádět predikci. Další příklady pro ostatní třídy $ESPG(k)$ gramatik by bylo snadné najít.

Přestože se zdálo, že $ESPG(k)$ gramatiky by mohly být silnější než $SPG(k)$ gramatiky, tak ve skutečnosti jsou stejně silné (viz věta 4.1). Provést důkaz $\mathbf{SPG(k) = ESPG(k)}$ je možné přes vztah:

$$\mathbf{SPG(k) \subseteq ESPG(k) \wedge ESPG(k) \subseteq SPG(k) \Rightarrow SPG(k) = ESPG(k)}$$

Inkluze $\mathbf{SPG(k)} \subseteq \mathbf{ESPG(k)}$ zřejmě platí, neboť podmínka 2 z definice 4.2 rovněž platí pro $SPG(k)$ gramatiku, tedy každá $SPG(k)$ gramatika je zároveň $ESPG(k)$ gramatikou.

Druhá inkluze $\mathbf{ESPG(k)} \subseteq \mathbf{SPG(k)}$ rovněž platí, protože existuje algoritmus 4.5, který převede každou $ESPG(k)$ gramatiku na ekvivalentní $SPG(k)$, ať už je libovolného typu z věty 4.1.

Vstup: $ESPG(k)$ gramatika $G_1 = (N, T, P_1, S)$ pro $k \geq 1$
Výstup: ekvivalentní $SPG(k)$ gramatika $G_2 = (N, T, P_2, S)$

(1) $P_2 \leftarrow \emptyset$
(2) $E \leftarrow \emptyset$
(3) **for each** $r_i = (A \rightarrow w,\ \sigma_i,\ \varphi_i) \in P_1$ **do**
(4) **if** $|\sigma_i| > 1$ **do** $r_{(i,1)} = (LHS(p) \rightarrow LHS(p),\ \sigma_i,\ \bigcup_{j \in \sigma_i} \varphi_j),\ p \in \sigma_i$
(5) $P_2 \leftarrow P_2 \cup \{r_{(i,1)}\}$
(6) $E \leftarrow E \cup \sigma_i$
(7) $\sigma_i \leftarrow \{(i,1)\}$
(8) **if** $|\varphi_i| > 1$ **do** $r_{(i,2)} = (LHS(p) \rightarrow LHS(p),\ \varphi_i,\ \bigcup_{j \in \varphi_i} \varphi_j),\ p \in \varphi_i$
(9) $P_2 \leftarrow P_2 \cup \{r_{(i,2)}\}$
(10) $E \leftarrow E \cup \varphi_i$
(11) $\varphi_i \leftarrow \{(i,2)\}$
(12) $P_2 \leftarrow P_2 \cup \{r_i\}$

(13) **for each** $r_i = (A \rightarrow w,\ \sigma_i,\ \varphi_i) \in P_2$ **do**
(14) **if** $\sigma_i = \{x\} \wedge x \in E$ **do** $r_{(i,x)} \leftarrow r_x$
(15) $\sigma_i \leftarrow \{(i,x)\}$
(16) $P_2 \leftarrow P_2 \cup \{r_{(i,x)}\}$
(17) **if** $\varphi_i = \{y\} \wedge y \in E$ **do** $r_{(i,y)} \leftarrow r_y$
(18) $\varphi_i \leftarrow \{(i,y)\}$
(19) $P_2 \leftarrow P_2 \cup \{r_{(i,y)}\}$

(20) **for each** $r_i = (A \rightarrow w,\ \sigma_i,\ \varphi_i) \in P_2$ **do**
(21) **if** $i \in E$ **do** $\varphi_i \leftarrow \emptyset$

Algoritmus 4.5. Převod $ESPG(k)$ gramatiky na ekvivalentní $SPG(k)$ gramatiku

původní pravidla 8, 9, 10				nová pravidla 8, (8,1), 9, (9,1), 10, (10,9)			
r_8	$= (C \rightarrow cY,$	$\{9,10\},$	$\emptyset)$	r_8	$= (C \rightarrow cY,$	$\{(8,1)\},$	$\emptyset)$
				$r_{(8,1)}$	$= (X \rightarrow X,$	$\{9,10\},$	$\{11\})$
r_9	$= (X \rightarrow 0,$	$\{9,10\},$	$\{11\})$	r_9	$= (X \rightarrow 0,$	$\{(9,1)\},$	$\emptyset)$
				$r_{(9,1)}$	$= (X \rightarrow X,$	$\{9,10\},$	$\{11\})$
r_{10}	$= (X \rightarrow 1,$	$\{9\},$	$\{11\})$	r_{10}	$= (X \rightarrow 1,$	$\{(10,9)\},$	$\emptyset)$
				$r_{(10,9)}$	$= (X \rightarrow 0,$	$\{(9,1)\},$	$\{11\})$

Tabulka 4.7: Demonstrace algoritmu 4.5 na pravidlech 8, 9, 10 z příkladu 4.2

Jako demonstraci principu algoritmu 4.5 uvažujme pravidla 8, 9, 10 z příkladu 4.2. Nežádoucí množiny *neúspěchu* v pravidlech 9 a 10 jsou nahrazeny prázdnou množinou a jsou

přidána pravidla $(8,1)$ a $(9,1)$, která nejdříve testují existenci příslušného neterminálu. Množina $\{11\}$ je tak vlastně přesunuta do pravidel $(8,1)$ a $(9,1)$, ve kterých již definici $SPG(k)$ gramatiky neodporují. Výsledek je možné vidět v tabulce 4.7.

4.1.4 Problém s výpočtem množin $FIRST_k$ a $FOLLOW_k$

Pro vygenerování prediktivní tabulky **TAB1** jsou použity algoritmy 4.1 a 4.2, které počítají množiny $FIRST_k$ a $FOLLOW_k$ pro nějaké $k > 0$. Výpočet množin $FIRST_k$ a $FOLLOW_k$ pro nějakou programovanou gramatiku G je proveden nad $BASE(G)$. Jinými slovy, u původní gramatiky G je odstraněno programové řízení (množiny σ a φ) a výpočet $FIRST_k$ a $FOLLOW_k$ probíhá, jako by se jednalo o klasickou bezkontextovou gramatiku. Toto zjednodušení sice umožňuje snadný výpočet těchto množin, ale na druhé straně může způsobit situaci, kdy dojde ke konfliktu v tabulce **TAB1**. Důvod je ten, že před samotným výpočtem $FIRST_k$ a $FOLLOW_k$ množin je odstraněno programové řízení, které zásadním způsobem ovlivňuje to, co může být z nějakého neterminálu v budoucnosti vygenerováno a také to, jaké terminály se mohou nacházet za nějakým neterminálem v průběhu derivování. Odstraněním programového řízení může dojít k tomu, že množiny $FIRST_k$ a $FOLLOW_k$ budou obsahovat slova, která nereflektují programovanou gramatiku G. Na následujícím příkladě si představíme tento problém.

Příklad 4.3. Mějme programovanou gramatiku $G_2 = (\{S', S, A, B, C\}, \{a, b\}, P, S')$ (viz [17], strana 105), kde $L(G_2) = \{a^n b^m a^n b^m \mid n \geq 1 \wedge m \geq 2\}$ a množina P obsahuje pravidla z tabulky 4.8.

i	r_i	σ_i	φ_i
(1)	$S' \to S$	$\{2\}$	$\emptyset$
(2)	$S \to AB$	$\{3\}$	$\emptyset$
(3)	$A \to aAa$	$\{3,4\}$	$\emptyset$
(4)	$A \to bC$	$\{5\}$	$\emptyset$
(5)	$B \to bB$	$\{6,7\}$	$\emptyset$
(6)	$C \to b$	$\{8\}$	$\emptyset$
(7)	$C \to A$	$\{4\}$	$\emptyset$
(8)	$B \to b$	$\emptyset$	$\emptyset$

Tabulka 4.8: Pravidla pro příklad 4.3

Z příkladu 4.3 odstraníme programové řízení a spočítáme množiny $FIRST_2$ a $FOLLOW_2$ algoritmy 4.1 a 4.2.

$\forall x \in T : FIRST_2(x) = \{x\}$	$FOLLOW_2(S') = \{\varepsilon\}$
$FIRST_2(S') = \{aa, ab, ba, bb\}$	$FOLLOW_2(S) = \{\varepsilon\}$
$FIRST_2(S) = \{aa, ab, ba, bb\}$	$FOLLOW_2(A) = \{b, aa, ab, bb\}$
$FIRST_2(A) = \{aa, ab, ba, bb\}$	$FOLLOW_2(B) = \{\varepsilon\}$
$FIRST_2(B) = \{b, bb\}$	$FOLLOW_2(C) = \{b, aa, ab, bb\}$
$FIRST_2(C) = \{b, aa, ab, ba, bb\}$	

Tabulka 4.9: Množiny $FIRST_2$ a $FOLLOW_2$ nad $BASE(G_2)$ z příkladu 4.3

Z pravidel pro gramatiku G_2 je zřejmé, že predikci je nutné provádět v pravidlech 3 a 5. Pro pravidla 3 a 4 z množiny σ_3 mimo jiné platí

(*1*) $FIRST_2(\{aAa\}\ FOLLOW_2(A)) = \{aa, ab\}$
(*2*) $FIRST_2(\{bC\}\ FOLLOW_2(A)) = \{ba, bb\}$

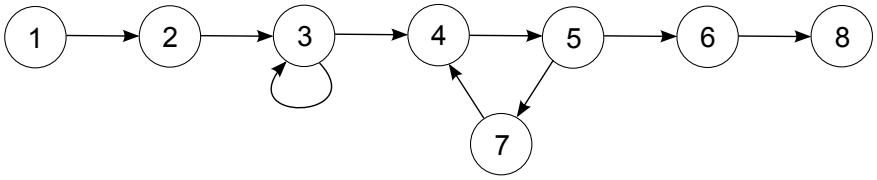
Platí $(1) \cap (2) = \emptyset$, tedy mezi pravidly 3 a 4 ke konfliktu nedochází. Pro pravidla 6 a 7 z množiny σ_5 platí:

(*1*) $FIRST_2(\{b\}\ FOLLOW_2(C)) = \{ba, bb\}$
(*2*) $FIRST_2(\{A\}\ FOLLOW_2(C)) = \{aa, ab, ba, bb\}$

Je zřejmé, že $(1) \cap (2) \neq \emptyset$, tedy gramatika G_2 z příkladu 4.3 není $SPG(2)$ ani $ESPG(2)$ gramatikou.

Návrh modifikace výpočtu množin $FIRST_k$

První problém spočívá ve výpočtu množiny $FIRST_2(A)$ pro pravidlo 7. Algoritmus 4.1 počítá tuto množinu globálně, tedy uvažuje všechna pravidla. Z programového řízení gramatiky G_2 je zřejmě vidět, že po aplikaci pravidla 7 již není nikdy možné aplikovat pravidlo 3, které generuje z neterminálu A na začátku terminál a. Programové řízení gramatiky G_2 je možné zobrazit jako orientovaný graf [11], ve kterém jsou uzly jednotlivá pravidla a ve kterém například hrana $(1, 2)$ představuje situaci, kdy po aplikaci pravidla 1 je dále možné použít pravidlo 2.



Obrázek 4.1: Grafová reprezentace programového řízení gramatiky G_2

Algoritmus výpočtu $FIRST_2(A)$ pro pravou stranu pravidla 7 je možné upravit tak, že algoritmus 4.1 bude uvažovat jen ta pravidla, která jsou v grafu na obrázku 4.1 dostupná z vrcholu 7. V tomto případě jsou z vrcholu 7 dostupná pravidla 4, 5, 6, 7 a 8 a výpočet množiny $FIRST_2(A)$ pro pravidlo 7 bude:

	inicializace	1. iterace	2. iterace	3. iterace
$FIRST_2(A)$	$\emptyset$	$\emptyset$	$\{bb\}$	$\{bb\}$
$FIRST_2(B)$	$\emptyset$	$\{b\}$	$\{b, bb\}$	$\{b, bb\}$
$FIRST_2(C)$	$\emptyset$	$\{b\}$	$\{b, bb\}$	$\{b, bb\}$

Tabulka 4.10: Iterace algoritmu 4.1 pro pravidla 4, 5, 6, 7 a 8

Množina $FIRST_2(A)$ tak obsahuje jen řetězec bb. Nicméně k odstranění konfliktu toto nestačí.

Návrh modifikace výpočtu množin $FOLLOW_k$

Další problém spočívá ve výpočtu množiny $FOLLOW_2(C)$. Původně obsahuje slova b, aa, ab a bb. Nicméně slova b a bb jsou nesmyslná, neboť spolu s programovým řízením gramatiky G_2 nemůže nikdy nastat situace, že se za neterminálem C bude někdy nacházet symbol b. Po aplikaci pravidla 2 musí být bezprostředně použito pravidlo 3, které zajistí, že po neterminálu A bude vždy následovat minimálně jeden symbol a. Tato vlastnost dále implikuje, že i po neterminálu C bude vždy následovat minimálně 1 symbol a. Správná množina $FOLLOW_2(C)$ by pak měla vypadat takto:

$$FOLLOW_2(C) = \{aa, ab\}$$

Výpočet množiny $FOLLOW_2(C)$ je možné zajistit například tak, že vygenerujeme všechny větné formy, kdy každé pravidlo je použito maximálně k-krát, tedy dvakrát. Mimo jiné bychom dostali například větné formy:

(*1*) $aabC\underline{aa}B$
(*2*) $abC\underline{ab}B$

Z větné formy (*1*) je zřejmé, že $aa \in FOLLOW_2(C)$. Z větné formy (*2*) je zřejmé, že $ab \in FOLLOW_2(C)$.

Po výše uvedených úpravách množin $FIRST_2(A)$ a $FOLLOW_2(C)$ již platí:

(*1*) $FIRST_2(\{b\}\ FOLLOW_2(C)) = \{ba\}$
(*2*) $FIRST_2(\{A\}\ FOLLOW_2(C)) = \{bb\}$

Tedy platí $(1) \cap (2) = \emptyset$ a prediktivní tabulku **TAB1** je možné sestavit bez konfliktu. Pro predikci ba bude použito pravidlo 6, pro predikci bb bude použito pravidlo 7.

Výše uvedené modifikace jsou ovšem určeny pro gramatiku G_2 z příkladu 4.3 a (především úprava výpočtu množin $FOLLOW_k$) nejsou použitelné pro libovolné programované gramatiky.

4.2 Maticové gramatiky

Vzhledem k tomu, že vyjadřovací síla jednotlivých typů maticových gramatik (viz definice 3.3) je stejná jako vyjadřovací síla jednotlivých typů programovaných gramatik (viz věty 3.2 a 3.3), je vhodné syntaktickou analýzu maticových gramatik provádět stejným algoritmem jako v části 4.1. Je však nutné definovat algoritmus, který dokáže libovolnou maticovou gramatiku převést na gramatiku programovanou tak, aby nedošlo ke změně jazyka, který daná maticová gramatika generuje.

Mějme nějakou maticovou gramatiku $G_M = (N, T, M, S, F)$, kde pro množinu sekvencí platí $M = \{m_1, m_2, \ldots, m_n\}$ a jednotlivé sekvence mají tvar:

$$
\begin{array}{lcl}
m_1 & = & (\ p_{11},\ p_{12},\ \ldots,\ p_{1k(1)}\ ) \\
m_2 & = & (\ p_{21},\ p_{22},\ \ldots,\ p_{2k(2)}\ ) \\
\vdots & & \\
m_n & = & (\ p_{n1},\ p_{n2},\ \ldots,\ p_{nk(n)}\ )
\end{array}
$$

Maticovou gramatiku G_M převedeme na programovanou gramatiku G_P tak, že každou sekvenci m_i, kde $i \in \{1, \ldots, n\}$, převedeme na pravidla programované gramatiky zobrazené v tabulce 4.11.

z	r_z	σ_z	φ_z
(x_{i1})	p_{i1}	$\{x_{i2}\}$	$\varphi_{x_{i1}}$
(x_{i2})	p_{i2}	$\{x_{i3}\}$	$\varphi_{x_{i2}}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$(x_{ik(i)})$	$p_{ik(i)}$	$\{x_{11}, x_{21}, \ldots, x_{n1}\}$	$\varphi_{x_{ik(i)}}$

Tabulka 4.11: Převod sekvence m_i na posloupnost pravidel programované gramatiky

Množina *úspěchu* σ každého pravidla v tabulce 4.11 vždy obsahuje jedno pravidlo, které v původní maticové gramatice následuje za daným pravidlem v sekvenci. Výjimkou je poslední pravidlo v sekvenci, které je převedeno na takové pravidlo programované gramatiky, ve kterém množina *úspěchu* σ obsahuje všechna pravidla, která se v maticové gramatice nacházejí na začátku všech sekvencí.

Pokud dále v maticové gramatice G_M pro nějaké pravidlo p_{ij} ze sekvence m_i, kde $i \in \{1, \ldots, n\} \wedge j \in \{1, \ldots, k(i)\}$, zároveň platí $p_{ij} \in F$, potom $\varphi_{x_{ij}} = \sigma_{x_{ij}}$, jinak platí $\varphi_{x_{ij}} = \emptyset$. Tato podmínka umožňuje brát v úvahu mód kontroly výskytu v maticové gramatice G_M.

Vstup: maticová gramatika $G_M = (N, T, M, S, F)$
Výstup: programovaná gramatika $G_P = (N, T, P, S)$

// výpočet množiny P
(1) **for each** $m_i \in M \wedge i \in \{1, \ldots, n\}$ **do**
(2) **for each** p_{ij} **in** $m_i \wedge j \in \{1, \ldots, k(i)\}$ **do**
(3) **if** $j = k(i)$ **do**
(4) $\sigma \leftarrow \{r_{11}, r_{21}, \ldots, r_{n1}\}$
(5) **else do**
(6) $\sigma \leftarrow \{r_{ij+1}\}$
(7) **if** $p_{ij} \in F$ **do**
(8) $\varphi \leftarrow \sigma$
(9) **else do**
(10) $\varphi \leftarrow \emptyset$
(11) $r_{ij} = (p_{ij},\ \sigma,\ \varphi)$
(12) $P \leftarrow P \cup \{r_{ij}\}$

Algoritmus 4.6. Převod maticové gramatiky na ekvivalentní programovanou gramatiku

Výše uvedený postup převodu je zapsán jako algoritmus 4.6. Předpokládejme maticovou gramatiku G_M z příkladu 3.2. Pomocí algoritmu 4.6 je možné tuto gramatiku převést na

sekvence		p_{ij}	σ_{ij}	φ_{ij}
m_1	r_{11}	$S \to ABC$	$\{r_{11}, r_{21}, r_{31}, r_{41}\}$	$\emptyset$
m_2	r_{21}	$A \to a$	$\{r_{22}\}$	$\emptyset$
	r_{22}	$C \to a$	$\{r_{23}\}$	$\emptyset$
	r_{23}	$B \to BB$	$\{r_{11}, r_{21}, r_{31}, r_{41}\}$	$\emptyset$
m_3	r_{31}	$A \to aA$	$\{r_{32}\}$	$\emptyset$
	r_{32}	$C \to aC$	$\{r_{33}\}$	$\emptyset$
	r_{33}	$B \to BBB$	$\{r_{11}, r_{21}, r_{31}, r_{41}\}$	$\emptyset$
m_4	r_{41}	$B \to b$	$\{r_{11}, r_{21}, r_{31}, r_{41}\}$	$\emptyset$

Tabulka 4.12: Algoritmus 4.6 nad příkladem 3.2

ekvivalentní programovanou gramatiku. Tabulka 4.12 zobrazuje tento algoritmus a výslednou programovanou gramatiku. Příklad derivace stejného slova jako pod příkladem 3.2, *aabbbbaa*, by vypadal takto:

$S \Rightarrow_{r_{11}} ABC \Rightarrow_{r_{31}} aABC \Rightarrow_{r_{32}} aABaC \Rightarrow_{r_{33}} aABBBaC \Rightarrow_{r_{21}} aaBBBaC \Rightarrow_{r_{22}}$
$aaBBBaa \Rightarrow_{r_{23}} aaBBBBaa \Rightarrow_{r_{41}} aabBBBaa \Rightarrow_{r_{41}} aabbBBaa \Rightarrow_{r_{41}} aabbbBaa \Rightarrow_{r_{41}}$
$aabbbbaa$

4.2.1 $SMG(k)$ gramatika

Nyní je nutné definovat tvar maticové gramatiky takový, aby výsledná programovaná gramatika byla typu $SPG(k)$ nebo $ESPG(k)$ (viz sekce 4.1). Zde lze postupovat přímočaře. Vždy po použití celé sekvence v nějaké maticové gramatice je vždy nutné se rozhodnout, jakou další sekvenci při následujících derivacích použít. Zde je možné použít podobné podmínky, jaké byly prezentovány v definici 4.2. Tedy každé počáteční pravidlo v každé sekvenci nesmí být v módu kontroly výskytu, musí mít stejnou levou stranu, tato levá strana (neterminál) musí být vždy nejlevější v okamžiku rozhodování a mezi každými dvěma pravidly musí platit stejná podmínka jako podmínka 3.b v definici 4.2. Nicméně tato podmínka je velmi omezující, protože je obtížné vytvořit smysluplnou maticovou gramatiku, která by toto splňovala.

Podmínku je možné zmírnit tak, že bude existovat vždy právě jedna sekvence, ve které bude mít první pravidlo tvar $S \to w$, kde S je počáteční neterminál, přičemž toto pravidlo není v módu kontroly výskytu a neterminál S se nesmí nacházet na pravé straně žádného pravidla. Nazvěme maticovou gramatiku s takovými omezeními jako $SMG(k)$ (*Simple Matrix Grammar* — v [4] je rovněž definována SMG gramatika, ovšem jedná se o jinou definici) pro nějaké $k > 0$. Číslo k, stejně jako u $SPG(k)$ gramatik, říká, kolik terminálů je třeba k predikci.

Mějme nějakou maticovou gramatiku G_M. Jako $BASE(G_M)$ budeme označovat bezkontextovou gramatiku, ve které množina pravidel obsahuje všechna bezkontextová pravidla ze všech sekvencí z množiny sekvencí z G_M.

Definice 4.3. Mějme maticovou gramatiku $G_M = (N, T, M, S, F)$. G_M je *jednoduchá maticová gramatika* $SMG(k)$ (*Simple Matrix Grammar*) pro nějaké $k > 0$, pokud $BASE(G_M)$ je bez cyklů a pokud pro množinu M platí $M = \{m_1, m_2, \ldots, m_n\}$, kde $n \geq 1$, a pokud jednotlivé sekvence mají tvar:

$$
\begin{array}{lcl}
m_1 & = & (\quad p_{11} = S \to w_1, \quad p_{12}, \quad \ldots, \quad p_{1k(1)} \quad) \\
m_2 & = & (\quad p_{21} = A \to w_2, \quad p_{22}, \quad \ldots, \quad p_{2k(2)} \quad) \\
\vdots & & \\
m_n & = & (\quad p_{n1} = A \to w_n, \quad p_{n2}, \quad \ldots, \quad p_{nk(n)} \quad)
\end{array}
$$

1. $\forall i \in \{1, \ldots, n\} \forall j \in \{1, \ldots, k(i)\} : \; RHS(p_{ij}) \in ((N - \{S\}) \cup T)^*$
2. $\forall i \in \{1, \ldots, n\} : \; p_{i1} \notin F$
3. neterminál A je nejlevější symbol ve větné formě
4. $\forall i, j \in \{2, \ldots, n\} \wedge i \neq j \; :$
   $FIRST_k(\{w_i\}\, FOLLOW_k(A)) \; \cap \; FIRST_k(\{w_j\}\, FOLLOW_k(A)) = \emptyset$

Gramatiku $SMG(k)$ je možné převést na ekvivalentní $SPG(k)$ gramatiku pomocí algoritmu 4.7, který je modifikací algoritmu 4.6. V algoritmu 4.7 je nutné uvažovat situaci, kdy sekvence m_1 bude provedena jen jednou, tedy interval rozhodování bude jen mezi sekvencemi m_2 až m_n. Na první pohled se může zdát, že výše uvedené porušuje základní princip maticových gramatik, a sice že pro další derivování může být vybrána libovolná sekvence. Sekvence m_1 již nikdy po prvním použití nebude provedena, protože podle bodu 1 v definici 4.3 se počáteční neterminál nesmí nacházet na pravé straně žádného pravidla, tedy pravidlo $S \to w_1$ již nikdy nebude použito podruhé. A protože toto pravidlo nesmí být v módu kontroly výskytu, sekvence m_1 již nikdy nebude podruhé provedena.

Vstup: $SMG(k)$ gramatika $G_M = (N, T, M, S, F)$
Výstup: $SPG(k)$ gramatika $G_P = (N, T, P, S)$

// výpočet množiny P
(1) **for each** $m_i \in M \wedge i \in \{1, \ldots, n\}$ **do**
(2) **for each** p_{ij} **in** $m_i \wedge j \in \{1, \ldots, k(i)\}$ **do**
(3) **if** $j = k(i)$ **do**
(4) $\sigma \leftarrow \{r_{21}, \ldots, r_{n1}\}$
(5) **else do**
(6) $\sigma \leftarrow \{r_{ij+1}\}$
(7) **if** $p_{ij} \in F$ **do**
(8) $\varphi \leftarrow \sigma$
(9) **else do**
(10) $\varphi \leftarrow \emptyset$
(11) $r_{ij} = (p_{ij}, \sigma, \varphi)$
(12) $P \leftarrow P \cup \{r_{ij}\}$

Algoritmus 4.7. Převod $SMG(k)$ gramatiky na ekvivalentní $SPG(k)$ gramatiku

Příklad 4.4. Mějme maticovou gramatiku $G_{M1} = (N, T, M, S, F)$, kde jednotlivé množiny jsou definovány:

$N = \{S, A, B, C\}$
$T = \{a, b, c\}$
$F = \emptyset$
$M = \{m_1, m_2, m_3\}$

$$\begin{array}{lcl} m_1 & = & (\quad S \to ABC \quad) \\ m_2 & = & (\quad A \to aA, \qquad B \to bB, \quad C \to cC \quad) \\ m_3 & = & (\quad A \to a, \qquad B \to b, \qquad C \to c \quad) \end{array}$$

Maticová gramatika G_{M1} z příkladu 4.4 popisuje jazyk $L(G_{M1}) = \{a^n b^n c^n \mid n \geq 1\}$. G_{M1} je $SMG(2)$ gramatika, protože:

1. počáteční neterminál S se nenachází na pravé straně žádného pravidla
2. pravidla na začátcích všech sekvencí nejsou v módu kontroly výskytu ($F = \emptyset$)
3. neterminál A je během derivování vždy nejlevější ve větné formě
4. algoritmy 4.1 a 4.2 spočítáme množiny $FIRST_2$ a $FOLLOW_2$ (viz tabulka 4.13). Platí:

   (1) $FIRST_2(\{aA\}\ FOLLOW_2(A)) = \{aa\}$
   (2) $FIRST_2(\{a\}\ FOLLOW_2(A)) = \{ab\}$

   Platí $(1) \cap (2) = \emptyset$.

$\forall x \in T : FIRST_2(x) = \{x\}$	$FOLLOW_2(S) = \{\varepsilon\}$
$FIRST_2(S) = \{aa, ab\}$	$FOLLOW_2(A) = \{bb, bc\}$
$FIRST_2(A) = \{a, aa\}$	$FOLLOW_2(B) = \{c, cc\}$
$FIRST_2(B) = \{b, bb\}$	$FOLLOW_2(C) = \{\varepsilon\}$
$FIRST_2(C) = \{c, cc\}$	

Tabulka 4.13: Množiny $FIRST_2$ a $FOLLOW_2$ nad $BASE(G_{M1})$ z příkladu 4.4

Pokud budeme chtít vygenerovat slovo *aabbcc* gramatikou G_{M1}, derivace bude vypadat takto:

$S \Rightarrow_{p_{11}} ABC \Rightarrow_{p_{21}} aABC \Rightarrow_{p_{22}} aAbBC \Rightarrow_{p_{23}} aAbBcC \Rightarrow_{p_{31}} aabBcC \Rightarrow_{p_{32}} aabbcC \Rightarrow_{p_{33}} aabbcc$

Gramatika G_{M1} z příkladu 4.4 je $SMG(2)$, tedy můžeme použít algoritmus 4.7 k převodu na ekvivalentní $SPG(2)$ gramatiku. Výsledek převodu můžeme vidět v tabulce 4.14.

Podle definice $SPG(k)$ gramatiky (viz [17], definice 3.3) je snadné určit, že programovaná gramatika z tabulky 4.14 je $SPG(2)$ gramatikou. Je vhodné si všimnout hlavního rozdílu algoritmů 4.6 a 4.7. Zatímco algoritmus 4.6 by do množin σ_{11}, σ_{23} a σ_{33} vložil pravidlo r_{11}, tak algoritmus 4.7 toto neudělá. Zajistí se tím podmínka, že levá strana všech pravidel, mezi kterými se provádí rozhodování, musí být shodná. Příklad generování slova *aabbcc* je:

sekvence		p_{ij}	σ_{ij}	φ_{ij}
m_1	r_{11}	$S \to ABC$	$\{r_{21}, r_{31}\}$	$\emptyset$
m_2	r_{21}	$A \to aA$	$\{r_{22}\}$	$\emptyset$
	r_{22}	$B \to bB$	$\{r_{23}\}$	$\emptyset$
	r_{23}	$C \to cC$	$\{r_{21}, r_{31}\}$	$\emptyset$
m_3	r_{31}	$A \to a$	$\{r_{32}\}$	$\emptyset$
	r_{32}	$B \to b$	$\{r_{33}\}$	$\emptyset$
	r_{33}	$C \to c$	$\{r_{21}, r_{31}\}$	$\emptyset$

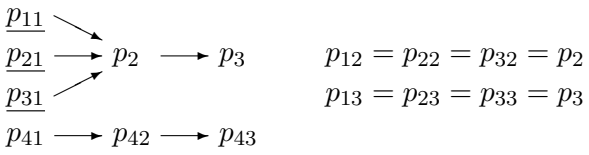
Tabulka 4.14: Ekvivalentní $SPG(2)$ gramatika pro příklad 4.4

$S \Rightarrow_{r_{11}} ABC \Rightarrow_{r_{21}} aABC \Rightarrow_{r_{22}} aAbBC \Rightarrow_{r_{23}} aAbBcC \Rightarrow_{r_{31}} aabBcC \Rightarrow_{r_{32}} aabbcC \Rightarrow_{r_{33}} aabbcc$

4.2.2 $ESMG(k)$ gramatika

Maticové gramatiky je rovněž možné omezit tak, aby byly převoditelné na $ESPG(k)$ gramatiky. Bod 2 v definici 4.3 říká, že pravidla, mezi kterými je nutné provést výběr, nesmí být v módu kontroly výskytu. To zaručuje, že množiny φ těchto pravidel ve výstupní programované gramatice jsou prázdné. Nicméně bod 2 definice 4.2 $ESPG(k)$ gramatiky umožňuje existenci jednoho pravidla, které bude použito, pokud příslušný neterminál ve větné formě neexistuje. Tato vlastnost může být použita tak, že počáteční pravidla sekvencí mohou být v módu kontroly výskytu. Ovšem podmínkou musí být, že taková pravidla nesmí být jediná v sekvencích, protože jejich množina φ by mohla obsahovat více jak jedno pravidlo. Další podmínkou je, aby zbývající pravidla mezi sekvencemi byla shodná. Mějme nějakou maticovou gramatiku se sekvencemi na obrázku 4.2. Pravidla p_{11}, p_{21} a p_{31} jsou v módu kontroly výskytu. Musí platit, že $p_{12} = p_{22} = p_{32} = p_2$ a $p_{13} = p_{23} = p_{33} = p_3$. Tato vlastnost umožňuje sloučit tato stejná pravidla v programované gramatice a zajistit podmínku, že bude existovat jen jedno pravidlo v případě neexistence příslušného neterminálu ve větné formě.

$$
\begin{array}{lcl}
m_1 & = & (\ \underline{p_{11}},\ p_{12},\ p_{13}\ ) \\
m_2 & = & (\ \underline{p_{21}},\ p_{22},\ p_{23}\ ) \\
m_3 & = & (\ \underline{p_{31}},\ p_{32},\ p_{33}\ ) \\
m_4 & = & (\ p_{41},\ p_{42},\ p_{43}\ )
\end{array}
$$



Obrázek 4.2: Vlastnost nutná pro převod maticové gramatiky na $ESPG(k)$

Definice 4.4. Mějme maticovou gramatiku $G_M = (N, T, M, S, F)$. G_M je *rozšířená jednoduchá maticová gramatika* $ESMG(k)$ (*Extended Simple Matrix Grammar*) pro nějaké $k > 0$, pokud $BASE(G_M)$ je bez cyklů a pro množinu M platí $M = \{m_1, m_2, \ldots, m_n\}$,

$n \geq 1$, a jednotlivé sekvence mají tvar:

$$\begin{array}{lcl} m_1 & = & (\quad p_{11} = S \to w_1, \quad p_{12}, \quad \ldots, \quad p_{1k(1)} \quad) \\ m_2 & = & (\quad p_{21} = A \to w_2, \quad p_{22}, \quad \ldots, \quad p_{2k(2)} \quad) \\ \vdots & & \\ m_n & = & (\quad p_{n1} = A \to w_n, \quad p_{n2}, \quad \ldots, \quad p_{nk(n)} \quad) \end{array}$$

1. $\forall i \in \{1, \ldots, n\} \forall j \in \{1, \ldots, k(i)\} : \; RHS(p_{ij}) \in ((N - \{S\}) \cup T)^*$
2. $p_{11} \notin F \wedge (\forall i, j \in \{2, \ldots, n\} : \; \exists p_{i1}, p_{j1} \in F \Rightarrow (k(i) = k(j) \wedge k(i) > 1 \wedge \forall m \in \{2, \ldots, k(i)\} : \; p_{im} = p_{jm}))$
3. neterminál A je nejlevější symbol ve větné formě
4. $\forall i, j \in \{2, \ldots, n\} \wedge i \neq j \; :$
   $FIRST_k(\{w_i\}\, FOLLOW_k(A)) \; \cap \; FIRST_k(\{w_j\}\, FOLLOW_k(A)) = \emptyset$

Vstup: $ESMG(k)$ gramatika $G_M = (N, T, M, S, F)$
Výstup: $ESPG(k)$ gramatika $G_P = (N, T, P, S)$

```
(1)   x ← y ← 0

      // výpočet množiny P
(2)   for each m_i ∈ M ∧ i ∈ {1, ..., n} do
(3)           for each p_ij in m_i ∧ j ∈ {1, ..., k(i)} do
(4)                   if  j = k(i) do
(5)                       σ ← {r_21, ..., r_n1}
(6)                   else do
(7)                       σ ← {r_ij+1}
(8)                   if  p_ij ∈ F do
(9)                       if  j = 1 do
(10)                          if  x > 0 do
(11)                              σ ← {p_xy}
(12)                          else do
(13)                              x ← i
(14)                              y ← j + 1
(15)                      φ ← σ
(16)                  else do
(17)                      φ ← ∅
(18)                  r_ij = (p_ij, σ, φ)
(19)                  P ← P ∪ {r_ij}
```

Algoritmus 4.8. Převod $ESMG(k)$ gramatiky na ekvivalentní $ESPG(k)$ gramatiku

Definice 4.4 $ESMG(k)$ gramatiky se od definice $SMG(k)$ gramatiky liší jen v bodu 2, který povoluje mód kontroly výskytu u počátečních pravidel v sekvencích. Algoritmus 4.8 je oproti algoritmu 4.7 rozšířen o proměnné x a y, ve kterých jsou uloženy indexy pravidla, které bude bezprostředně následovat za všemi pravidly v módu kontroly výskytu, které

jsou na začátku sekvencí. Při inicializaci algoritmu jsou tyto proměnné vynulovány. Pokud je v průběhu algoritmu nalezeno pravidlo p_{i1} pro $i \in \{1, \ldots, n\}$, které je v módu kontroly výskytu, do proměnných x a y jsou uloženy indexy bezprostředně následujícího pravidla, tedy po řadě hodnoty i a $j + 1$. Pokud je po nastavení těchto proměnných nalezeno další pravidlo p_{i1} v módu kontroly výskytu, množina σ a φ bude obsahovat pravidlo p_{xy}.

Je nutné poznamenat, že algoritmus 4.8 neprovádí kontrolu vstupní maticové gramatiky, zda-li je typu $ESMG(k)$. Dále je tento algoritmus zjednodušen tak, že v případě nalezení počátečního pravidla v módu kontroly výskytu dále generuje pravidla v příslušné sekvenci, přestože tato pravidla budou ve výstupní $ESPG(k)$ gramatice nedostupná. Nicméně tuto nepříjemnost je možné eliminovat na implementační úrovni.

Příklad 4.5. Mějme maticovou gramatiku $G_{M2} = (N, T, M, S, F)$, kde jednotlivé množiny jsou definovány:

$N = \{S, A, B, C, X\}$
$T = \{a, b, c, x\}$
$F = \{A \to aA, X \to xA\}$
$M = \{m_1, m_2, m_3, m_4\}$

$$
\begin{array}{lcllll}
m_1 & = & (& S \to ABC &) & \\
m_2 & = & (& A \to aA, & X \to xA, & B \to bB, \quad C \to cC \quad) \\
m_3 & = & (& A \to aX, & B \to bB, & C \to cC \quad) \\
m_4 & = & (& A \to a, & B \to b, & C \to c \quad)
\end{array}
$$

Maticová gramatika G_{M2} z příkladu 4.5 popisuje jazyk $L(G_{M2}) = \{mb^n c^n \mid n \geq 1 \wedge |m| = n \ \wedge \ m \in \{a\}\{a, x\}^*\{a\} \ \cup \ \{a\} \ \wedge \ xx \textit{ není podřetězcem } m\}$. G_{M2} je $ESMG(2)$ gramatika, protože jsou splněny všechny podmínky definice 4.4. První pravidlo sekvence m_2 je v módu kontroly výskytu, přičemž sekvence m_2 obsahuje aspoň dvě pravidla. Ke konfliktu v prediktivní tabulce rovněž nedojde, neboť podle tabulky 4.15 vychází:

(*1*) $FIRST_2(\{aA\}\ FOLLOW_2(A)) = \{aa\}$
(*2*) $FIRST_2(\{aX\}\ FOLLOW_2(A)) = \{ax\}$
(*3*) $FIRST_2(\{a\}\ FOLLOW_2(A)) = \{ab\}$

Platí $(1) \cap (2) \cap (3) = \emptyset$.

$\forall x \in T : FIRST_2(x) = \{x\}$	$FOLLOW_2(S) = \{\varepsilon\}$
$FIRST_2(S) = \{aa, ab, ax\}$	$FOLLOW_2(A) = \{bb, bc\}$
$FIRST_2(A) = \{a, aa, ax\}$	$FOLLOW_2(B) = \{c, cc\}$
$FIRST_2(B) = \{b, bb\}$	$FOLLOW_2(C) = \{\varepsilon\}$
$FIRST_2(C) = \{c, cc\}$	$FOLLOW_2(X) = \{bb, bc\}$
$FIRST_2(X) = \{xa\}$	

Tabulka 4.15: Množiny $FIRST_2$ a $FOLLOW_2$ nad $BASE(G_{M2})$ z příkladu 4.5

Derivaci slova *axabbbccc* gramatikou G_{M2} lze provést takto:

$S \Rightarrow_{p_{11}} ABC \Rightarrow_{p_{31}} aXBC \Rightarrow_{p_{32}} aXbBC \Rightarrow_{p_{33}} aXbBcC \Rightarrow_{p_{21}} aXbBcC \Rightarrow_{p_{22}} axAbBcC \Rightarrow_{p_{23}} axAbbBcC \Rightarrow_{p_{24}} axAbbBccC \Rightarrow_{p_{41}} axabbBccC \Rightarrow_{p_{42}} axabbbccC \Rightarrow_{p_{43}} axabbbccc$

sekvence		p_{ij}	σ_{ij}	φ_{ij}
m_1	r_{11}	$S \to ABC$	$\{r_{21}, r_{31}, r_{41}\}$	$\emptyset$
m_2	r_{21}	$A \to aA$	$\{r_{22}\}$	$\{r_{22}\}$
	r_{22}	$X \to xA$	$\{r_{23}\}$	$\{r_{23}\}$
	r_{23}	$B \to bB$	$\{r_{24}\}$	$\emptyset$
	r_{24}	$C \to cC$	$\{r_{21}, r_{31}, r_{41}\}$	$\emptyset$
m_3	r_{31}	$A \to aX$	$\{r_{32}\}$	$\emptyset$
	r_{32}	$B \to bB$	$\{r_{33}\}$	$\emptyset$
	r_{33}	$C \to cC$	$\{r_{21}, r_{31}, r_{41}\}$	$\emptyset$
m_4	r_{41}	$A \to a$	$\{r_{42}\}$	$\emptyset$
	r_{42}	$B \to b$	$\{r_{43}\}$	$\emptyset$
	r_{43}	$C \to c$	$\{r_{21}, r_{31}, r_{41}\}$	$\emptyset$

Tabulka 4.16: Ekvivalentní $ESPG(2)$ gramatika pro příklad 4.5

Gramatika G_{M2} je $ESMG(2)$ gramatikou, takže ji je možné algoritmem 4.8 převést na ekvivalentní $ESPG(2)$ gramatiku. Výsledek převodu je v tabulce 4.16. Z ní je vidět, že platí:

$$|\varphi_{21} \cup \varphi_{31} \cup \varphi_{41}| \leq 1$$

Derivaci slova $axabbbccc$ $ESPG(2)$ gramatikou z tabulky 4.16 je možné provést takto:

$S \Rightarrow_{r_{11}} ABC \Rightarrow_{r_{31}} aXBC \Rightarrow_{r_{32}} aXbBC \Rightarrow_{r_{33}} aXbBcC \Rightarrow_{r_{21}} aXbBcC \Rightarrow_{r_{22}} axAbBcC \Rightarrow_{r_{23}} axAbbBcC \Rightarrow_{r_{24}} axAbbBccC \Rightarrow_{r_{41}} axabbBccC \Rightarrow_{r_{42}} axabbbccC \Rightarrow_{r_{43}} axabbbccc$

4.2.3 Vlastnosti $SMG(k)$ a $ESMG(k)$

Tím, že byly definovány algoritmy 4.7 a 4.8, je zřejmé, že platí věta 4.2.

Věta 4.2.

(*1*) $\mathbf{SMG(k)} \subseteq \mathbf{SPG(k)}$

(*2*) $\mathbf{ESMG(k)} \subseteq \mathbf{ESPG(k)}$

Definice $SMG(k)$ a $ESMG(k)$ gramatiky je velmi přísná na tvar maticové gramatiky. Vlastnost, že levá strana ve všech počátečních pravidlech sekvencí musí být stejná, neumožňuje rozhodování mezi více neterminály jako v $SPG(k)$ a $ESPG(k)$ gramatikách. Vytvořit tak smysluplnou $SMG(k)$ nebo $ESMG(k)$ gramatiku je velmi obtížné. Lze tak intuitivně předpokládat, že jejich vyjadřovací síla bude menší oproti $SPG(k)$ a $ESPG(k)$ gramatikám. Mějme příklad 4.2 $ESPG(2)$ gramatiky. V ní dochází k rozhodování v několika pravidlech mezi neterminály A, X a Y. Otázka, jak převést tuto gramatiku na $ESMG(k)$ tak, aby docházelo k rozhodování jen v jednom neterminálu, je velmi obtížná.

4.3 Regulárně řízené gramatiky

Syntaktickou analýzu regulárně řízených gramatik je možné provádět podobným způsobem jako $SPG(k)$ a $ESPG(k)$ gramatik. Je však nutné vhodně zpracovat regulární výraz nad pravidly v předpisu regulárně řízené gramatiky a poté provádět predikci podle vstupního slova.

V této části budeme používat konečné automaty, jejich varianty a algoritmy pro převod regulárního výrazu na konečný automat a převody na různé typy automatů. Jejich definice a popis algoritmů je možné nastudovat v [22, 23].

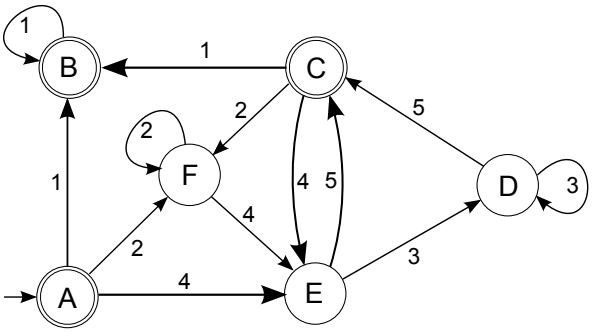
Mějme nějakou regulárně řízenou gramatiku $G_R = (N, T, P, S, R, F)$. Nejprve je nutné převést regulární výraz R na deterministický konečný automat. Postup převodu je obecně zapsán jako algoritmus 4.9.

Vstup: regulární výraz R nad abecedou P
Výstup: deterministický konečný automat M_R

(1) převeď regulární výraz R do postfixové notace
([22], strana 21, algoritmus 4.1)
(2) zpracuj postfixovou notaci a vytvoř rozšířený konečný automat
([23], strana 42, definice 3.14) pomocí definic
3.8, 3.9, 3.10 v [22], strana 10
(3) převeď rozšířený konečný automat na deterministický konečný automat
([23], strana 44, algoritmus 3.6)

Algoritmus 4.9. Převod regulárního výrazu na deterministický konečný automat

Příklad 4.6. Mějme regulárně řízenou gramatiku G_R z příkladu 3.1. Pro zopakování: množina pravidel je $P = \{1, 2, 3, 4, 5\}$ a regulární výraz R nad množinou pravidel P je (2*43*5)*1*. Deterministický konečný automat ekvivalentní regulárnímu výrazu R je na obrázku 4.3.



Obrázek 4.3: Deterministický automat pro regulární výraz (2*43*5)*1*

Celou činnost syntaktické analýzy je možné provádět jako simulaci příslušného konečného automatu. Činnost automatu na obrázku 4.3 začíná ve stavu A. Z něj je možné přejít do stavů B, E, F použitím pravidel po řadě 1, 4, 2 gramatiky G_R. Je tedy možné použít více jak jedno pravidlo, tedy je nutné se deterministicky rozhodnout mezi těmito pravidly.

Pro vlastnosti těchto pravidel musí platit stejné vlastnosti jako v definici 4.2. Všechny levé strany musejí být stejné a v okamžiku rozhodování nejlevější ve větné formě. Aby bylo rozhodnutí možné, pro

(*1*) $FIRST_k(\{RHS(1)\}\ FOLLOW_k(LHS(1)))$
(*2*) $FIRST_k(\{RHS(4)\}\ FOLLOW_k(LHS(4)))$
(*3*) $FIRST_k(\{RHS(2)\}\ FOLLOW_k(LHS(2)))$

musí platit $(1) \cap (2) \cap (3) = \emptyset$ pro nějaké $k > 0$. Pro rozhodování může rovněž existovat prediktivní tabulka podobně jako při analýze $ESPG(k)$ gramatik.

Pokud je z nějakého stavu možné přejít **právě** jedním pravidlem automatu, k predikci pravidla přirozeně nedochází, protože je známo (například stav B). Zde je nutné poznamenat, že automat **nesmí** být úplný. Pokud by byl úplný, v každém stavu by existoval přechod pro každé pravidlo G_R a predikci by bylo nutné provádět v každém stavu a mezi všemi pravidly, což by prakticky znemožnilo jakoukoli rozumnou činnost.

Po vybrání pravidla 1, 4 nebo 2 automat přejde do dalšího stavu, ve kterém opakujeme výše popsanou činnost. Simulace konečného automatu bude probíhat tak dlouho, dokud není vygenerováno slovo v G_R gramatice za pomocí pravidel, které jsme používali pro simulaci automatu. Poté je třeba zjistit, v jakém stavu simulace skončila. Pokud skončila v koncovém stavu, sekvence pravidel G_R gramatiky patří do regulární množiny R, tedy vygenerované slovo patří do $L(G_R)$. V opačném případě vygenerované slovo nepatří do $L(G_R)$.

Z výše uvedené činnosti je zřejmé, že algoritmus je velmi podobný algoritmu 4.4 pro $ESPG(k)$ gramatiky. Navíc je zde jen kontrola, zda-li konečný automat skončil v koncovém stavu. Algoritmus si proto zde nebudeme ukazovat. Místo toho zde bude prezentován mechanismus převodu regulárně řízené gramatiky na **neekvivalentní** $SPG(k)$ resp. $ESPG(k)$ gramatiku. Programovaná gramatika tak bude součástí algoritmu analýzy. Nejprve je však nutné definovat omezení na regulárně řízenou gramatiku tak, aby byla možná predikce.

4.3.1 $SRCG(k)$ gramatika

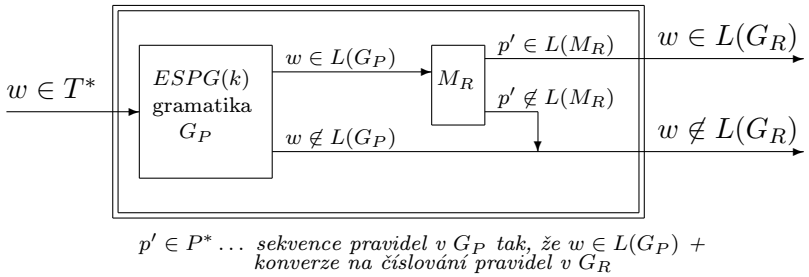
Definice 4.5. Mějme regulárně řízenou gramatiku $G_R = (N, T, P, S, R, F)$. G_R je *jednoduchá regulárně řízená gramatika* $SRCG(k)$ (*Simple Regularly Controlled Grammar*) pro nějaké $k > 0$, pokud $BASE(G_R)$ je bez cyklů a pro deterministický konečný automat $M_R = (Q, P, P_1, s, F_1)$ ekvivalentní regulární množině R platí:

$\forall a, b \in P : (\exists Aa \to B, Ab \to C \in P_1 \wedge\ A, B, C \in Q\ \wedge\ a \neq b)\ \Rightarrow$

1. $a \notin F\ \wedge\ b \notin F \wedge$
2. $LHS(a) = LHS(b) \wedge$
3. $LHS(a)$ *je nejlevější ve větné formě* $\wedge$
4. $(1) \cap (2) = \emptyset$, kde
   - (*1*) $FIRST_k(\{RHS(a)\}\ FOLLOW_k(LHS(a)))$
   - (*2*) $FIRST_k(\{RHS(b)\}\ FOLLOW_k(LHS(b)))$

Výše uvedené číslované podmínky platí jen v případě, že existují přechody z jednoho stavu do druhého pomocí více pravidel G_R gramatiky. Podmínka 1 zajišťuje, že v okamžiku rozhodování nebudou pravidla G_R gramatiky v módu kontroly výskytu.

Jak už bylo zmíněno, syntaktická analýza bude složená. Regulární množinu R je nutné převést na deterministický konečný automat M_R a z něj poté vytvořit $ESPG(k)$ gramatiku G_P, která ovšem nebude ekvivalentní vůči G_R. Vypočítaná G_P gramatika bude prvním článkem algoritmu. Pokud gramatika G_P generuje vstupní slovo $w \in T^*$, potom algoritmus 4.4 vrátí *ANO* a sekvenci pravidel p. Tuto sekvenci pravidel je poté nutné upravit na p' a tuto p' odsimulovat na automatu M_R. Pokud platí $p' \in L(M_R)$, potom vstup w je generován gramatikou G_R. Mechanismus je názorně zobrazen na obrázku 4.4.



Obrázek 4.4: Složená analýza $SRCG(k)$ gramatiky pomocí $ESPG(k)$ gramatiky

---

Vstup: $SRCG(k)$ gramatika $G_R = (N, T, P_1, S, R_1, F_1)$ pro $k > 0$
deterministický konečný automat $M_R = (Q, P_1, P_2, s, F_2)$
Výstup: $ESPG(k)$ gramatika $G_P = (N \cup \{S'\}, T, P, S')$, $S' \notin N$

// výpočet funkce goal: $Q \to 2^{P_1 \times Q}$
(1) **for each** $A \in Q$ **do** $goal(A) \leftarrow \emptyset$
(2) **for each** $Aa \to B \in P_2$ **do** $goal(A) \leftarrow goal(A) \cup \{(a, B)\}$

// výpočet množiny P
(3) **for each** $Aa \to B \in P_2$ **do** $\sigma \leftarrow goal(B)$
(4) $\varphi \leftarrow \emptyset$
(5) **if** $a \in F_1$ **do** $\varphi \leftarrow \sigma$
(6) $r_{(a,B)} = (a,\ \sigma,\ \varphi)$
(7) $P \leftarrow P \cup \{r_{(a,B)}\}$

// přidání dodatečného pravidla podle definice 4.2 $ESPG(k)$ *gramatiky*
(8) $r_1 = (S' \to S,\ goal(s),\ \emptyset)$
(9) $P \leftarrow P \cup \{r_1\}$

Algoritmus 4.10. Vytvoření pomocné $ESPG(k)$ gramatiky G_P z M_R a G_R

Algoritmus 4.10 provádí již zmíněný převod $SRCG(k)$ gramatiky na pomocnou $ESPG(k)$ gramatiku. Vstupem je $SRCG(k)$ gramatika a deterministický konečný automat M_R, který

popisuje regulární množinu R_1. Algoritmus nejdříve spočítá funkci $Q \to 2^{P_1 \times Q}$, která uchovává informaci o tom, která pravidla z množiny P_1 je možné použít z nějakého stavu automatu M_R. Důvod, proč se uchová množina dvojic (*pravidlo z* P_1, *stav z* Q), je ten, že určité pravidlo z P_1 může být použito v různých kontextech a po jeho použití se může automat M_R nacházet v různých stavech. V takovém případě výsledná $ESPG(k)$ gramatika bude obsahovat více takových pravidel ovšem s různými množinami σ (viz příklad 4.7). Další krok převodu spočívá ve výpočtu množiny pravidel P. Pro každý přechod automatu M_R je vytvořeno pravidlo číslované dvojicí z množiny $P_1 \times Q$. Množiny σ a φ jsou vytvořeny s pomocí funkce *goal*. Posledním krokem je přidání pravidla r_1, aby byla splněna podmínka $ESPG(k)$ gramatiky.

Vstup: $SRCG(k)$ gramatika $G = (N, T, P, S, R, F)$ pro $k > 0$
slovo $w = a_1 a_2 \dots a_n \in T^*$ pro $n \geq 0$
Výstup: ANO, $w \in L(G)$ (seznam pravidel na výstupu)
NE, $w \notin L(G)$

(1) Vytvoř deterministický konečný automat M_R z regulární množiny R algoritmem 4.9
(2) Vytvoř pomocnou $ESPG(k)$ gramatiku G_P algoritmem 4.10
(3) Vytvoř prediktivní tabulky **TAB1** a **TAB2** pro G_P algoritmem 4.3
(4) Otestuj členství slova w do $L(G_P)$ algoritmem 4.4
(výsledek a seznam pravidel ulož po řadě do proměnných *ok* a p)

(5) **if** $ok = true$ **do**
// je nutné odstranit první pravidlo, které se v G nenachází
(6) $p.removeFirst()$
// je nutné převést dvojice na původní číslování v G
(7) **for each** (a, B) **in** p **do**
(8) $p'.append(a)$
// algoritmus otestování členství p' do jazyka deterministického konečného
// automatu M_R je triviální a nebudeme ho zde uvádět
(9) **if** $p' \in L(M_R)$ **do**
(10) **Output** p'
(11) **return** *ANO*
(12) **else do**
(13) **return** *NE*
(14) **else do**
(15) **return** *NE*

Algoritmus 4.11. Algoritmus syntaktické analýzy $SRCG(k)$ gramatiky

Algoritmus analýzy 4.11 dělá přesně to, co popisuje obrázek 4.4. Jsou zde použity operace nad seznamem $removeFirst()$ resp. $append(item)$, které smažou první prvek ze seznamu resp. přidají prvek *item* na konec seznamu. Na následujícím příkladu si přesně ukážeme, jak výše uvedené algoritmy fungují.

Příklad 4.7. Mějme regulárně řízenou gramatiku $G_R = (N, T, P, S, R, F)$, kde jednotlivé množiny jsou definovány v tabulce 4.17. Gramatika G_R generuje jazyk $L(G_R) =$

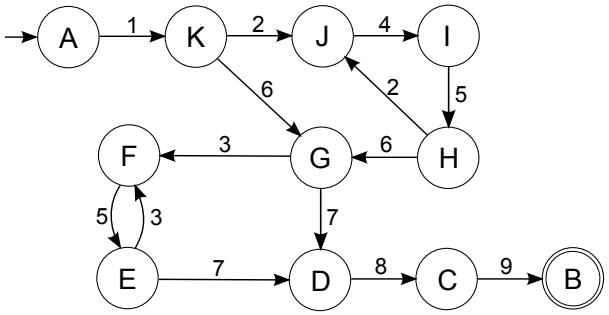
$\{a^n b^m c^{2n} d^{n+m} \mid n \geq 1 \ \wedge \ m \geq 0\}$.

Abychom zjistili, jestli je G_R gramatika $SRCG(2)$ gramatikou, musíme regulární množinu R převést na deterministický konečný automat M_R (viz obrázek 4.5). Z konečného automatu na obrázku 4.5 je možné zjistit, že rozhodování je nutné provádět

- mezi pravidly 2 a 6 ve stavu K
- mezi pravidly 2 a 6 ve stavu H
- mezi pravidly 3 a 7 ve stavu G
- mezi pravidly 3 a 7 ve stavu E

množiny	i	p_i
$N = \{S, A, B, C, D\}$	(1)	$S \rightarrow ABCD$
$T = \{a, b, c, d\}$	(2)	$A \rightarrow aA$
$P = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$	(3)	$B \rightarrow bB$
$R = 1(245)^*6(35)^*789$	(4)	$C \rightarrow ccC$
$F = \emptyset$	(5)	$D \rightarrow dD$
	(6)	$A \rightarrow a$
	(7)	$B \rightarrow \varepsilon$
	(8)	$C \rightarrow cc$
	(9)	$D \rightarrow d$

Tabulka 4.17: Jednotlivé množiny a pravidla pro příklad 4.7



Obrázek 4.5: Deterministický automat pro regulární výraz 1(245)*6(35)*789

Pravidla 2 a 6 mají stejnou levou stranu (neterminál A), která je vždy v okamžiku rozhodování nejlevější. Dále rovněž platí $(1) \cap (2) = \emptyset$ pro:

$(1) \quad FIRST_2(\{aA\}\ FOLLOW_2(A)) = \{aa\}$
$(2) \quad FIRST_2(\{a\}\ FOLLOW_2(A)) = \{ab, ac\}$

Pravidla 3 a 7 mají rovněž stejnou levou stranu (neterminál B), která je vždy nejlevější, protože neterminál A je vždy odstraněn pravidlem 6 před prvním použití pravidla 3 nebo 7. Rovněž platí $(3) \cap (4) = \emptyset$ pro:

$(3) \quad FIRST_2(\{bB\}\ FOLLOW_2(B)) = \{bb, bc\}$

(4) $FIRST_2(\{\varepsilon\}\ FOLLOW_2(B)) = \{cc\}$

Žádná pravidla v gramatice G_R nejsou v módu kontroly výskytu, protože $F = \emptyset$. Jsou tedy splněny všechny podmínky definice 4.5 a gramatika G_R je $SRCG(2)$ gramatikou.

Nejdříve aplikujeme algoritmus 4.10 a vytvoříme pomocnou gramatiku G_P. Prvním krokem je vytvoření funkce *goal*. Tabulka 4.18 zobrazuje tento postup pro každý stav z automatu M_R a pro každé pravidlo z něj vycházející.

Q	P_2	funkce *goal*	Q	P_2	funkce *goal*
A	$A1 \to K$	$goal(A) = \{(1, K)\}$	G	$G3 \to F$	$goal(G) = \{(3, F)\}$
B		$goal(B) = \emptyset$		$G7 \to D$	$goal(G) = \{(3, F), (7, D)\}$
C	$C9 \to B$	$goal(C) = \{(9, B)\}$	H	$H2 \to J$	$goal(H) = \{(2, J)\}$
D	$D8 \to C$	$goal(D) = \{(8, C)\}$		$H6 \to G$	$goal(H) = \{(2, J), (6, G)\}$
E	$E3 \to F$	$goal(E) = \{(3, F)\}$	I	$I5 \to H$	$goal(I) = \{(5, H)\}$
	$E7 \to D$	$goal(E) = \{(3, F), (7, D)\}$	J	$J4 \to I$	$goal(J) = \{(4, I)\}$
F	$F5 \to E$	$goal(F) = \{(5, E)\}$	K	$K2 \to J$	$goal(K) = \{(2, J)\}$
				$K6 \to G$	$goal(K) = \{(2, J), (6, G)\}$

Tabulka 4.18: Postup vytvoření funkce *goal* pro příklad 4.7

Postup vytváření pravidel v pomocné $ESPG(2)$ gramatice G_P je v další tabulce 4.19. Pro každý přechod automatu M_R jsou funkcí *goal* zjištěny obsahy množin σ a φ. V tabulce je vhodné si všimnout, že některá pravidla byla vložena vícekrát ($r_{(2,J)}$, $r_{(6,G)}$, $r_{(3,F)}$, $r_{(7,D)}$), což vůbec ničemu nevadí, protože pravidla vkládáme do množiny, ve které se stejné prvky nemohou opakovat. Rovněž je možné si všimnout dvou pravidel $r_{(5,H)}$ a $r_{(5,E)}$. Obě pravidla mají stejné základní pravidlo $D \to dD$ (pravidlo 5 v G_R). Důvod, proč bylo pravidlo 5 zduplikováno v G_P gramatice, jen ten, že pravidlo 5 je v automatu M_R použito ve více místech a z těchto míst je možné pokračovat různými pravidly. Proto musí být vytvořena dvě pravidla $r_{(5,H)}$ a $r_{(5,E)}$, která mají sice stejné základní pravidlo, ale různé množiny σ a φ. Na konci tabulky je přidáno pravidlo r_1, které zajistí, že bude vždy existovat jen jedno pravidlo, se kterým začne derivace v G_P.

Gramatikou G_P z tabulky 4.19 vygenerujeme slovo *aabccccddd* takto:

$S' \Rightarrow_1 S \Rightarrow_{(1,K)} ABCD \Rightarrow_{(2,J)} aABCD \Rightarrow_{(4,I)} aABccCD \Rightarrow_{(5,H)} aABccCdD \Rightarrow_{(6,G)}$
$aaBccCdD \Rightarrow_{(3,F)} aabBccCdD \Rightarrow_{(5,E)} aabBccCddD \Rightarrow_{(7,D)} aabccCddD \Rightarrow_{(8,C)}$
$aabccccddD \Rightarrow_{(9,B)} aabccccddd$

Seznam pravidel pouužitých při derivaci v G_P je tedy:

$$p = [1, (1, K), (2, J), (4, I), (5, H), (6, G), (3, F), (5, E), (7, D), (8, C), (9, B)]$$

Z tohoto seznamu odstraníme první záznam a všechny ostatní záznamy tvaru (a, B) nahradíme prvním prvkem z této dvojice. Tedy vznikne seznam:

$$p' = [1, 2, 4, 5, 6, 3, 5, 7, 8, 9]$$

P_2	σ	φ	pravidlo v G_P			
$A1 \to K$	$goal(K)$	$\emptyset$	$r_{(1,K)}$	$= (S \to ABCD,$	$\{(2,J),(6,G)\},$	$\emptyset)$
$K2 \to J$	$goal(J)$	$\emptyset$	$r_{(2,J)}$	$= (A \to aA,$	$\{(4,I)\},$	$\emptyset)$
$K6 \to G$	$goal(G)$	$\emptyset$	$r_{(6,G)}$	$= (A \to a,$	$\{(3,F),(7,D)\},$	$\emptyset)$
$J4 \to I$	$goal(I)$	$\emptyset$	$r_{(4,I)}$	$= (C \to ccC,$	$\{(5,H)\},$	$\emptyset)$
$I5 \to H$	$goal(H)$	$\emptyset$	$r_{(5,H)}$	$= (D \to dD,$	$\{(2,J),(6,G)\},$	$\emptyset)$
$H2 \to J$	$goal(J)$	$\emptyset$	$r_{(2,J)}$	$= (A \to aA,$	$\{(4,I)\},$	$\emptyset)$
$H6 \to G$	$goal(G)$	$\emptyset$	$r_{(6,G)}$	$= (A \to a,$	$\{(3,F),(7,D)\},$	$\emptyset)$
$G3 \to F$	$goal(F)$	$\emptyset$	$r_{(3,F)}$	$= (B \to bB,$	$\{(5,E)\},$	$\emptyset)$
$G7 \to D$	$goal(D)$	$\emptyset$	$r_{(7,D)}$	$= (B \to \varepsilon,$	$\{(8,C)\},$	$\emptyset)$
$F5 \to E$	$goal(E)$	$\emptyset$	$r_{(5,E)}$	$= (D \to dD,$	$\{(3,F),(7,D)\},$	$\emptyset)$
$E3 \to F$	$goal(F)$	$\emptyset$	$r_{(3,F)}$	$= (B \to bB,$	$\{(5,E)\},$	$\emptyset)$
$E7 \to D$	$goal(D)$	$\emptyset$	$r_{(7,D)}$	$= (B \to \varepsilon,$	$\{(8,C)\},$	$\emptyset)$
$D8 \to C$	$goal(C)$	$\emptyset$	$r_{(8,C)}$	$= (C \to cc,$	$\{(9,B)\},$	$\emptyset)$
$C9 \to B$	$goal(B)$	$\emptyset$	$r_{(9,B)}$	$= (D \to d,$	$\emptyset,$	$\emptyset)$
	$goal(A)$	$\emptyset$	r_1	$= (S' \to S,$	$\{(1,K)\},$	$\emptyset)$

Tabulka 4.19: Tvorba pravidel pomocné gramatiky G_P pro příklad 4.7

Simulací konečného automatu M_R se seznamem p' zjistíme, že platí $p' \in L(M_R)$. Slovo *aabccccddd* tedy patří do jazyka gramatiky G_R.

Dále by bylo možné uvažovat definování *rozšířené* $SRCG(k)$ gramatiky podobně jako pro $SPG(k)$ a $SMG(k)$ gramatiky. Musely by být definovány omezené vlastnosti na konečný automat M_R pro pravidla v G_R, která by byla v módu kontroly výskytu. Omezení by mělo velmi podobný tvar jako pro $ESMG(k)$ gramatiky (viz část 4.2.2), a proto zde toto rozšíření nebudeme uvádět.

4.4 Gramatiky s nahodilým kontextem

V části 3.4 byly prezentovány některé typy gramatik s nahodilým kontextem. Zásadní nevýhodou těchto gramatik je mechanismus, jakým se zjišťuje seznam pravidel, které je možné použít v další derivaci. Pro každé pravidlo je nutné projít větnou formu a zjistit, jestli obsahuje resp. neobsahuje neterminály z *povolující* resp. *zakazující* množiny. Tato operace má lineární horní časovou složitost vzhledem k délce testovaného slova, protože s délkou vstupu roste i délka větné formy až lineárně (zatímco u předchozích gramatik byla tato složitost konstantní, protože další použitelná pravidla byla předem známa). Složitost tohoto vyhledávání lze trochu snížit, pokud použijeme *levě/pravě povolující/zakazující* gramatiku. V těchto variantách existuje jen jedna množina (první vylepšení) a také při ověřování nemusí být nutné projít vždy celou větnou formu, pokud budeme uvažovat nejlevější resp. nejpravější derivaci (druhé vylepšení). Ovšem jejich vyjadřovací síla není příliš velká, proto bude celkově vhodnější použít *kooperující distribuované gramatiky*, které jako základ budou obsahovat *levě/pravě povolující/zakazující gramatiku*.

Definovat omezení obecně pro gramatiky s nahodilým kontextem je problém. Sice je možné opět definovat omezení, která zajistí predikovatelnost pomocí prediktivní tabulky, ale jen z tvaru nějaké gramatiky a jejích pravidel není možné určit, jestli tato omezení splňuje, protože by bylo nutné otestovat všechny možné větné formy. Nutný tvar pravi-

del CD gramatiky zde tedy nubudeme uvádět, neboť ho není možné určit. Místo toho zde uvedeme způsoby, jakými bude prováděna derivace tak, aby byla trochu efektivnější než úplné prohledávání stavového prostoru. Dále již budeme pracovat jen s $\mathbf{CD^2_{lp}}$ gramatikou (právě dvě komponenty, levě povolující gramatika, která se hodí pro případnou predikci nejlevějšího neterminálu, a žádná vymazávací pravidla).

4.4.1 $SLPCD(k)$ gramatika

Definice 4.6. Mějme $\mathbf{CD^2_{lp}}$ gramatiku $G = (N, T, P_1, P_2, S)$. Gramatiku G nazveme $SLPCD(k)$ (*Simple Left Permitting Cooperating Distributed grammar*) pro $k \geq 1$, pokud pro její tvar a derivace platí:

- $\forall (A \to w, R) \in P_2 : \ A \neq S$
- uvažujeme jazyk $L_{1-can}(G)$ (*obecná* 1*–kanonická derivace*) — při derivaci je tedy vždy přepsán nejlevější neterminál, který přepsán být může (máme-li například tři použitelná pravidla s navzájem různými levými stranami, bude použito to pravidlo, které má svou levou stranu nejlevěji ve větné formě)
- máme-li dvě či více použitelných pravidel, která mají stejnou levou stranu:
  - pokud je přepisovaný neterminál nejlevějším neterminálem ve větné formě, proveď predikci o délce k symbolů pomocí prediktivní tabulky, která je vytvořena nad $BASE(P_1 \cup P_2)$
  - nebo proveď prohledávání stavového prostoru, pokud predikce z předchozího bodu není možná

Prediktivní tabulka musí být spočítána nad oběma množinami P_1 a P_2. Tabulka bude mít stejný tvar jako u $LL(k)$ gramatik, tedy každý její řádek popisuje predikce pro právě jeden neterminál, přičemž ve sloupcích jsou příslušné predikce délky až k symbolů. Způsob výpočtu je zápsán jako algoritmus 4.12. Výpočet prediktivní tabulky je optimalizován tak, že záznamy jsou počítány jen pro taková pravidla, která mají prázdnou povolující množinu (přirozeně, pro pravidlo, které očekává existenci nějakého neterminálu nalevo od přepisovaného neterminálu, není predikce možná).

Vstup: $SLPCD(k)$ gramatika $G = (N, T, P_1, P_2, S)$ pro $k \geq 1$
Výstup: prediktivní tabulka **PREDICT**

(1) **for each** $(A \to v, R) \in P_1 \cup P_2$ **do**
(2) **if** $R = \emptyset$ **do**
(3) **for each** $i \in FIRST_k(\{v\}\ FOLLOW_k(A))$ **do**
(4) $PREDICT[A][i] \leftarrow PREDICT[A][i] \cup \{(A \to v, R)\}$

Algoritmus 4.12. Výpočet prediktivní tabulky pro $SLPCD(k)$ gramatiku

Algoritmus 4.13 provádí vyhledávání použitelných pravidel pro aktuální větnou formu, přičemž jsou dodržovány podmínky z definice $SLPCD(k)$ gramatiky. Vstupem algoritmu je složená struktura nazvaná *item*, která obsahuje:

- f $\sim$ větná forma nad $SLPCD(k)$ gramatikou
- i $\sim$ pozice aktuálního symbolu v testovaném slově w, platí $1 \leq i \leq |w| + 1$
- r $\sim$ seznam pravidel, která byla použita při vygenerování větné formy f
- c $\sim$ číslo komponenty, ve které momentálně jsme, platí $1 \leq c \leq 2$
- h $\sim$ množina, která obsahuje historii větných forem

Algoritmus ve dvou cyklech prochází každé pravidlo z komponenty *item.c* a pro něj prochází větnou formu. Vrací tři hodnoty: množinu použitelných pravidel, pozici nejlevějšího přepsatelného neterminálu ve větné formě a logickou hodnotu, zda má být proveden pokus o predikci podle prediktivní tabulky **PREDICT**. Výraz $POS(y)$ z řádků *9*, *10* a *16* vrací pozici neterminálu y při procházení větné formy cyklem na řádku *7*.

Vstup: $SLPCD(k)$ gramatika $G = (N, T, P_1, P_2, S)$ pro $k \geq 1$
item $\sim$ položka z algoritmu 4.14
Výstup: *set* $\sim$ množina použitelných pravidel
position $\sim$ pozice neterminálu, který může být přepsán
predict $\sim$ logická hodnota *true*, pokud má být provedena predikce

```
(1)  set ← ∅
(2)  position ← ∞
(3)  predict ← false

(4)  for each (A → v, R) ∈ P_item.c do
(5)          tmpR ← R
(6)          net ← false
(7)          for each y in item.f do
(8)                  if y ∈ N do
(9)                      if y = A ∧ tmpR = ∅ ∧ POS(y) ≤ position do
(10)                         if POS(y) < position do
(11)                             set ← {(A → v, R)}
(12)                             predict ← false
(13)                         else do
(14)                             set ← set ∪ {(A → v, R)}
(15)                             predict ← ¬net
(16)                         position ← POS(y)
(17)                         break
(18)                     tmpR ← tmpR − {y}
(19)                     net ← true

(20) return set, position, predict
```

Algoritmus 4.13. Vyhledání množiny použitelných pravidel pro další derivaci

Samotný algoritmus syntaktické analýzy 4.14 je prováděn v cyklu na řádku *7*. Seznam F, který reprezentuje stavový prostor všech vygenerovaných větných forem, je nejprve inicializován jediným prvkem *item* (větná forma obsahuje počáteční neterminál, číslo komponenty

1, pozice na prvním symbolu testovaného slova w). V případě, že gramatika je navržena deterministicky, k prohledávání stavového prostoru nemusí vůbec dojít a seznam F bude obsahovat vždy právě jeden prvek. Nad seznamem (včetně větné formy, kterou lze rovněž chápat jako seznam) jsou definovány tyto operace:

- $getFirst()$ ~ vrátí první prvek seznamu
- $get(k)$ ~ vrátí k-tý prvek seznamu
- $removeFirst()$ ~ smaže první prvek seznamu
- $insertFirst(item)$ ~ vloží prvek $item$ na začátek seznamu
- $append(item)$ ~ vloží prvek $item$ na konec seznamu
- $replace(pos, frm)$ ~ prvek na pozici pos nahradí prvky ze seznamu frm

Na řádcích *8* a *9* je vždy získána a odstraněna první položka z F, je nad ní provedena akce a později *může* být vrácena zpět na začátek seznamu F řádkem *49*. Z řádků *47* a *49* vyplývá, že algoritmus provádí prohledávání stavového prostoru do *hloubky* tím, že právě všechny položky vkládá na začátek seznamu F. Prohledávání do *šířky* lze pak snadno získat tak, že místo vkládání na začátek budeme vždy vkládat položky na konec seznamu F.

Vstup: prediktivní tabulka **PREDICT**
$SLPCD(k)$ gramatika $G = (N, T, P_1, P_2, S)$ pro $k \geq 1$
slovo $w = a_1 a_2 \dots a_n \in T^*$ pro $n \geq 0$
Výstup: ANO, $w \in L(G)$ (seznam pravidel na výstupu)
NE, $w \notin L(G)$

```
(1)  item.f ← S      // uchovává větnou formu
(2)  item.i ← 1      // uchovává pozici ve slově w
(3)  item.r ← [ ]    // uchovává seznam použitých pravidel
(4)  item.c ← 1      // uchovává číslo komponenty {1, 2}
(5)  item.h ← ∅      // uchovává historii větných forem
(6)  F ← [item]

(7)  while |F| > 0 do
(8)      item ← F.getFirst()
(9)      F.removeFirst()

         // otestování mezí větné formy
(10)     if (|item.f| = 0 ∧ item.i ≤ n) ∨ |item.f| > n do
(11)         continue
(12)     else if |item.f| = 0 ∧ item.i > n do
(13)         F.insertFirst(item)
(14)         break

         // nejlevěji ve větné formě je terminál
(15)     if item.f.getFirst() ∈ T do
(16)         if item.f.getFirst() = a_item.i do
(17)             item.f.removeFirst()
```

```
(18)                item.i ← item.i + 1
(19)            else do
(20)                continue

            // nejlevěji ve větné formě je neterminál
(21)        else do
(22)            set, pos, predict ← (volání algoritmu 4.13 se vstupem item)
                // žádné pravidlo → přepnutí do druhé komponenty
(23)            if set = ∅ do
(24)                item.c ← (item.c mod 2) + 1
                // 1 pravidlo → determinismus
(25)            else if set = {(A_1 → v_1, R_1)} do
(26)                item.f.replace(pos, v_1)
(27)                item.r.append( (A_1 → v_1, R_1) )
(28)                if item.f ∈ item.h do continue
(29)                item.h ← item.h ∪ {item.f}
                // více jak 1 pravidlo
(30)            else do
(31)                if predict = true do
(32)                    rules ← set ∩ PREDICT[item.f.get(pos)][a_{item.i} ... a_{item.i+k-1}]
(33)                else do
(34)                    rules ← set
                    // predikce zredukovala počet pravidel na jen jedno → determinismus
(35)                if rules = {(A_2 → v_2, R_2)} do
(36)                    item.f.replace(pos, v_2)
(37)                    item.r.append( (A_2 → v_2, R_2) )
(38)                    if item.f ∈ item.h do continue
(39)                    item.h ← item.h ∪ {item.f}
                    // predikce nepomohla → prohledávání stavového prostoru
(40)                else do
(41)                    for each (A_3 → v_3, R_3) ∈ rules do
(42)                            child ← item
(43)                            child.f.replace(pos, v_3)
(44)                            child.r.append( (A_3 → v_3, R_3) )
(45)                            if child.f ∈ child.h do continue
(46)                            child.h ← child.h ∪ {child.f}
(47)                            F.insertFirst(child)
(48)                    continue
(49)        F.insertFirst(item)

(50)    if |F| > 0 do
(51)        item ← F.getFirst()
(52)        Output item.r
(53)        return true
(54)    else do
(55)        return false
```

Algoritmus 4.14. Algoritmus syntaktické analýzy $SLPCD(k)$ gramatiky

Na řádcích *10–14* je vždy větná forma f z položky otestována na meze. Především se jedná o situaci, kdy nemá význam uvažovat větnou formu, která *je delší* než testované slovo w (díky tomu, že gramatika nesmí obsahovat vymazávací pravidla, je zajištěno, že každý neterminál musí být vždy přepsán aspoň na jeden terminál, tedy z větné formy určité délky nelze nikdy získat slovo kratší).

Na řádku *22* je volán algoritmus 4.13. Pokud neexistuje žádné další pravidlo pro derivaci, dojde k přepnutí do druhé komponenty (řádek *24*). Je-li použitelné *právě* jedno pravidlo (*řádek 25*), k prohledávání stavového prostoru nedochází. Při více použitelných pravidlech může (*řádek 32*) i nemusí (*řádek 34*) být provedena predikce. Pokud je predikce deterministická (*řádek 35*), k prohledávání stavového prostoru opět nedochází. Teprve v cyklu na řádku *41* dochází k prohledávání stavového prostoru, pokud je predikce bez efektu.

Důležitou součástí algoritmu je práce s historií větných forem. Pokaždé, když je provedena derivace, je ověřeno, jestli nová větná forma již nebyla někdy v minulosti vyderivována. Pokud byla, jedná se o zacyklení, které by nikdy neskončilo, proto je taková větná forma zahozena. Toto ošetření je prováděno na dvojicích řádků *28–29*, *38–39* a *45–46*.

Příklad 4.8. Mějme $SLPCD(1)$ gramatiku $G = (\{S, A, A_1, X, X_1, Y\}, \{a, b, c\}, P_1, P_2, S)$, kde množiny P_1 a P_2 jsou popsány v tabulce 4.20.

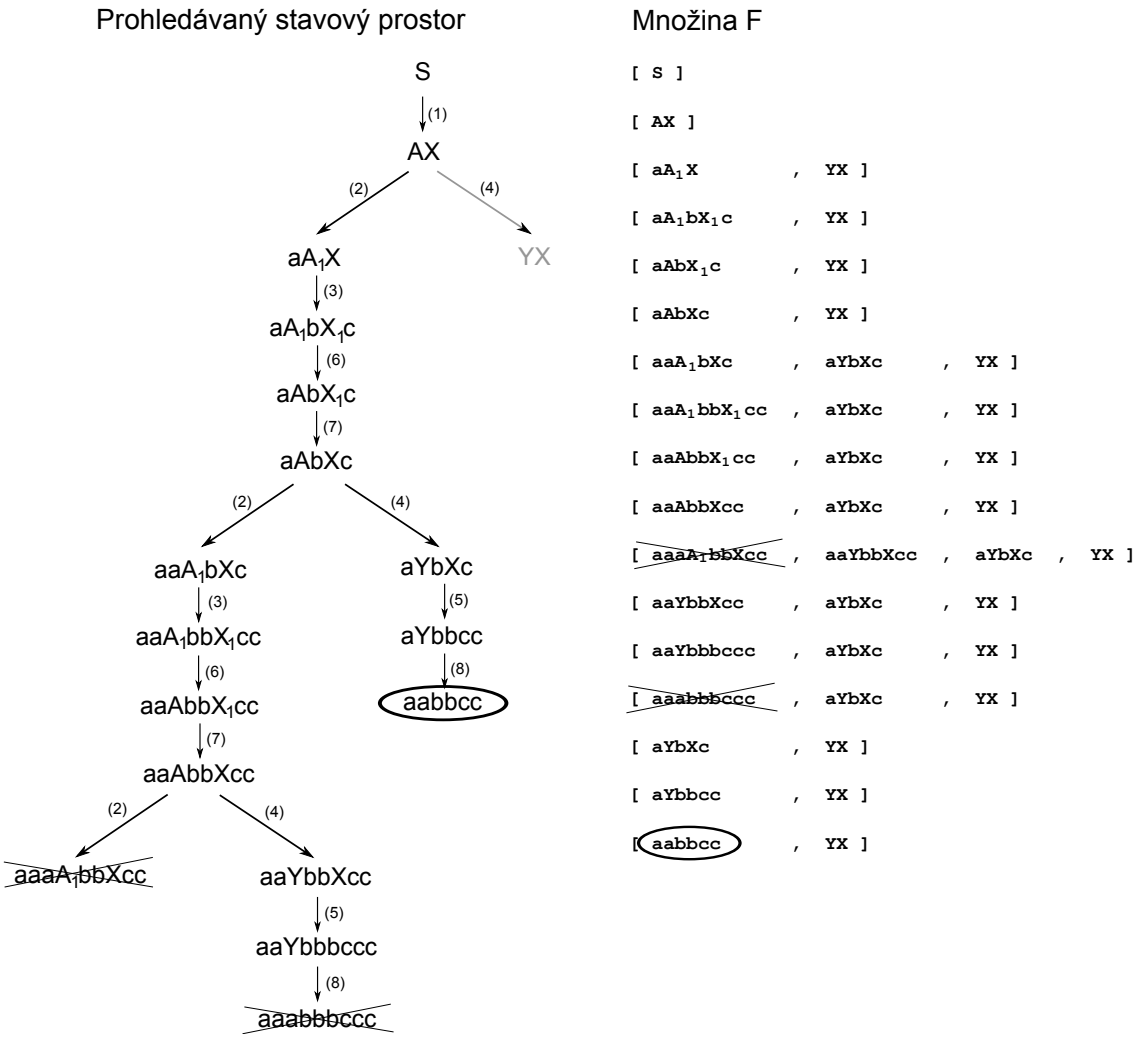
i	P_1		i	P_2	
(1)	$S \to AX$,	$\emptyset$	(6)	$A_1 \to A$,	$\emptyset$
(2)	$A \to aA_1$,	$\emptyset$	(7)	$X_1 \to X$,	$\{A\}$
(3)	$X \to bX_1c$,	$\{A_1\}$	(8)	$Y \to a$,	$\emptyset$
(4)	$A \to Y$,	$\emptyset$			
(5)	$X \to bc$,	$\{Y\}$			

Tabulka 4.20: Komponenty P_1 a P_2 pro příklad 4.8

Gramatika G generuje jazyk $L(G) = \{a^n b^n c^n \mid n > 0\}$ a je navržena tak, že k rozhodování dochází vždy jen mezi pravidly 2 a 4. Pro predikci délky 1 není možné rozhodnutí, proto zde dojde k prohledávání stavového prostoru (pro predikci délky 2 to již možné je). Pro demonstraci činnosti dále předpokládejme prohledávání stavového prostoru do hloubky, přičemž pravidlo 2 je zpracováváno prioritně oproti pravidlu 4. Na obrázku 4.6 je nakreslen prohledávaný stavový prostor pro slovo *aabbcc*, přičemž u každé hrany je v závorce uvedeno číslo použitého pravidla. Dále je zobrazena množina F, která je pro přehlednost tvořena jen samotnými větnými formami místo celé struktury *item*.

Tím, že pravidlo 2 je zpracováno prioritně, jsou „zbytečně“ vygenerovány větné formy $aaaA_1bbXcc$ a *aaabbbccc*, které jsou zahozeny. Slovo *aabbcc* je možné vygenerovat za pomocí pravidel, která tvoří cestu z kořene stromu až k listu *aabbcc*, tedy seznam pravidel je $[1, 2, 3, 6, 7, 4, 5, 8]$.

$S \Rightarrow_1 AX \Rightarrow_2 aA_1X \Rightarrow_3 aA_1bX_1c \Rightarrow_6 aAbX_1c \Rightarrow_7 aAbXc \Rightarrow_4 aYbXc \Rightarrow_5 aYbbcc \Rightarrow_8 aabbcc$



Obrázek 4.6: Stavový prostor a množina F pro příklad 4.8

4.4.2 Složitost $SLPCD(k)$

Je třeba uvažovat dva případy. Pokud bude gramatika navržena tak, že v každém kroku derivace je možné se deterministicky rozhodnout, které pravidlo použít pro další derivaci, horní časová složitost syntaktické analýzy bude kvadratická, konkrétně:

$$O(cnww) = O((cn)w^2)$$

Symboly c, n jsou konstanty a w testované slovo. Symbol c udává počet prvků v povolující množině nějakého pravidla. Konstantou c je myšlena situace, kdy je třeba zjistit, zda se určitý neterminál z větné formy (ne)nachází v této množině (uvažujeme sekvenční vyhledávání). Konstanta n udává počet pravidel v komponentě P_1 nebo P_2. Větnou formu je totiž nutné prohledat n–krát pro povolující množinu každého pravidla v komponentě. První proměnná w udává prohledávání větné formy při filtrování pravidel a druhá proměnná w popisuje počet derivací nutný k vygenerování testovaného slova (počet derivací je zcela určitě závislý na délce testovaného slova).

Pokud bude gramatika navržena nedeterministicky, tedy že bude nutné prohledávat stavový prostor, výše zmíněná složitost bude irelevantní, protože prohledávání stavového prostoru má obecně exponenciální časovou složitost. Nicméně u $SLPCD(k)$ gramatik za použití obecné 1–kanonické derivace je tato složitost trochu snížena oproti CD^2_{lp} gramatikám (řád ale zůstává stejný).

4.4.3 Vyjadřovací síla $SLPCD(k)$

$SLPCD(k)$ gramatiky nedokáží vygenerovat některé jazyky z $\mathbf{CD^2_{lp}}$. Příkladem může být jazyk $\{a^{2^n} \mid n \geq 0\}$, který lze popsat CD^3_{lp} gramatikou:

$$G_{2^n} = (\{S, A\}, \{a\}, \{(S \to AA,\ \emptyset)\}, \{(A \to S,\ \emptyset)\}, \{(S \to a,\ \emptyset)\}, S)$$

Tedy podle věty 3.9 existuje CD^2_{lp} gramatika, která generuje $L(G_{2^n})$. Ovšem predikce není možná, tedy $L(G_{2^n}) \notin \mathbf{SLPCD(k)}$ pro libovolné konečné $k \geq 1$. Naproti tomu nelze vyloučit, že nějaká $SLPCD(k)$ gramatika dokáže vygenerovat nějaký jazyk, který CD^2_{lp} vygenerovat nedokáže, protože omezení derivace může sílu zvýšit (viz porovnání vět 3.3 a 3.5 u programovaných gramatik).

Kapitola 5

Implementace analyzátoru

Syntaktický analyzátor je implementován v jazyce C++ tak, aby bylo možné snadno přidávat další druhy řízených gramatik. Skládá se z řady tříd, které mají mezi sebou vztah dědičnosti. Tyto vztahy jsou graficky znázorněny na obrázku 5.1 (diagram tříd).

`Token`

Tato třída v sobě zahrnuje informace o tokenu při čtení konfiguračního souboru nebo testovaného slova (viz příloha o struktuře konfiguračních souborů). Výčtový typ `LEXEM_TYPE` obsahuje typ tokenu, který zahrnuje jak například terminály a neterminály (hodnota `STRING`), tak i speciální značky, jako šipka, čárka nebo složená závorka. Pro intuitivní použití jsou definována přetypování pro složitější typy (má smysl, jen když token je typu `STRING`):

`Term`	$\sim$	`set<Token>`	(množina terminálů)
`Nont`	$\sim$	`set<Token>`	(množina neterminálů)
`Word`	$\sim$	`vector<Token>`	(obecně větná forma)
`Language`	$\sim$	`set<Word>`	(jazyk jako množina slov)

`Rule`

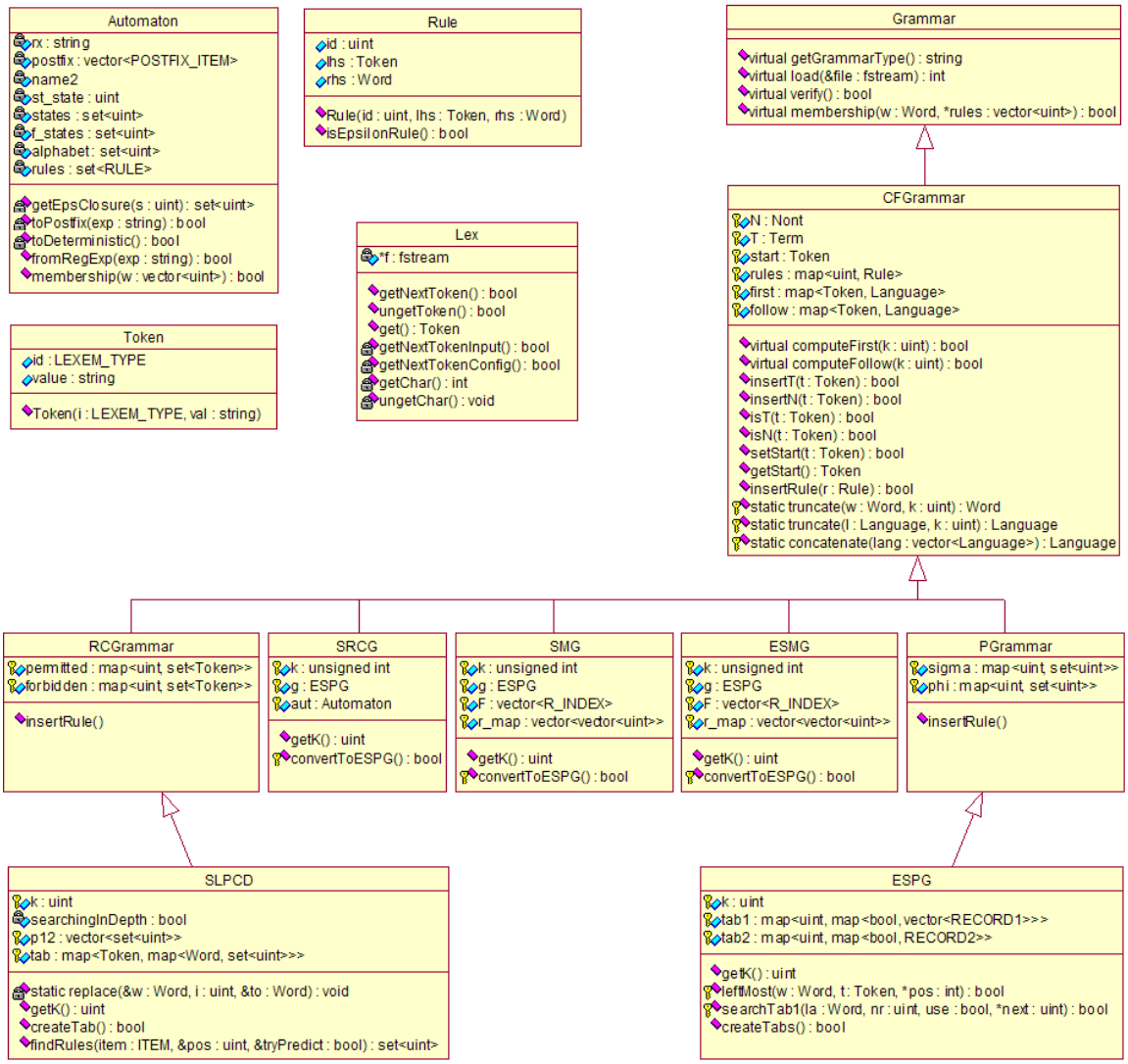
Třída definuje bezkontextové pravidlo. Každé pravidlo musí mít číselnou identifikaci `id` (přirozené číslo), levou stranu typu `Token` a pravou stranu typu `Word`.

`Lex`

Třída `Lex` zapouzdřuje lexikální analyzátor pro čtení konfiguračního souboru nebo testovaného slova. K dispozici jsou standardní metody pro čtení a vracení tokenů.

`Grammar`

Třída `Grammar` jen obsahuje virtuální metody, které jsou poté implementovány v ostatních podtřídách. K dispozici je metoda `load` pro načtení příslušné gramatiky z konfiguračního souboru s použitím třídy `Lex`, metoda `verify` k otestování, zda-li je přečtená gramatika takového typu, který je v konfiguračním souboru zadán, metoda `membership` pro otestování členství testovaného slova do jazyka gramatiky (vrací logickou hodnotu a případně vektor identifikátorů pravidel gramatiky, který je nutné použít pro vygenerování slova).



Obrázek 5.1: Všechny třídy v analyzátoru a jejich závislosti

`CFGrammar`

Tato třída zapouzdřuje bezkontextovou gramatiku. Je definována množina terminálů (typ `Term`), množina neterminálů (typ `Nont`), počáteční neterminál a množina pravidel, která je pro lepší práci implementovaná jako mapovací funkce z identifikátoru pravidla na samotné pravidlo typu `Rule`. Dále jsou k dispozici mapovací funkce pro množiny $FIRST_k$ a $FOLLOW_k$, které je možné spočítat příslušnými metodami se vstupním argumentem k. Dále existují příslušné metody pro zřetězení jazyků, zkracování slova a zkracování slov v jazyce (metoda `truncate`).

`PGrammar`

Třída popisuje programovanou gramatiku a rozšiřuje bezkontextovou gramatiku jen o existenci množin *úspěch* a *neúspěch* pro každé pravidlo a o existenci metody `insertRule` pro vložení pravidla gramatiky.

`ESPG`

Třída implementuje $ESPG(k)$ gramatiku (viz 4.2). Prediktivní tabulky jsou uloženy jako zanořené mapovací funkce. Prediktivní tabulka **TAB1** je uložena jako mapovací funkce s číslem subtabulky jako klíčem a hodnotou jako mapovací funkce, ve které jako klíč vystupuje logická hodnota (*Success*, *Failure*) a hodnota je vektor datových struktur `RECORD1`. Ta obsahuje příslušnou predikci a číslo pravidla, které bude dále použito. Tabulka **TAB2** je vytvořena podobným způsobem s použitím struktury `RECORD2`, která obsahuje *Lhs* a *Src*.

`SMG` **a** `ESMG`

Třída `SMG` resp. `ESMG` popisuje $SMG(k)$ resp. $ESMG(k)$ gramatiku (viz 4.3 resp. 4.4). Obě třídy vnitřně používají třídu `ESPG`. Příslušné převody 4.7 resp. 4.8 jsou implementovány v metodě `convertToESPG`. Pravidla jsou uchovávána jako vektor sekvencí, přičemž sekvence je vektor identifikátorů pravidel. Množina F (pravidla v módu kontroly výskytu) je implementována jako vektor struktur `R_INDEX`, kde struktura `R_INDEX` obsahuje číslo sekvence a zleva pořadové číslo pravidla, které je v módu kontroly výskytu.

`SRCG`

Tato třída implementuje $SRCG(k)$ gramatiku (viz 4.5). Třída vnitřně používá třídu `ESPG` a `Automaton`. Regulární výraz v $SRCG(k)$ gramatice je převeden na deterministický konečný automat, který je reprezentován třídou `Automaton`. Algoritmus 4.10 je implementován v metodě `convertToESPG`. Pokud testované slovo patří do jazyka pomocné $ESPG(k)$ gramatiky, posloupnost pravidel je příslušně upravena (vznikne vektor přirozených čísel) a otestována na členství do jazyka deterministického konečného automatu (metodou `membership`) přesně podle algoritmu 4.11.

`Automaton`

Třída `Automaton` je využita jen třídou `SRCG` a implementuje strukturu a chování deterministického konečného automatu. Stavy a symboly jsou označovány jako přirozená čísla typu `unsigned int`. Třída umožňuje získat deterministický konečný automat z regulárního výrazu metodou `fromRegExp`. V regulárním výrazu musí být použity klasické operátory sjednocení, zřetězení a iterace. Přesný formát zápisu lze nalézt v příloze u popisu konfiguračního souboru pro $SRCG(k)$ gramatiku. Členství slova (vektoru přirozených čísel) do jazyka automatu lze otestovat metodou `membership`.

`RCGrammar`

Tato třída zapouzdřuje gramatiku s nahodilým kontextem a jen rozšiřuje bezkontextovou gramatiku o *povolující* a *zakazující* množinu pro každé pravidlo a o existenci metody `insertRule` pro vložení pravidla gramatiky.

`SLPCD`

Tato třída implementuje $SLPCD(k)$ gramatiku (viz 4.6). Množiny pravidel dvou komponent jsou uchovávány ve dvouprvkovém vektoru. Prediktivní tabulka **PREDICT** je vytvořena opět jako zanořená mapovací funkce. Algoritmy 4.12 resp. 4.13 jsou implementovány v metodách `createTab` resp. `findRules`.

Kapitola 6

Závěr

V této práci byla prezentována Chomského klasifikace gramatik a několik druhů řízených gramatik. Pro $SPG(k)$ gramatiku bylo definováno vlastní rozšíření nazvané jako $ESPG(k)$ gramatika. Toto rozšíření umožňuje benevolentnější tvar pravidel, kdy pravidla, mezi kterými je třeba provádět rozhodování, mohou mít dohromady až jedno pravidlo v množině *neúspěchu*. Tato modifikace byla vytvořena v domnění, že by mohla zvýšit vyjadřovací sílu oproti $SPG(k)$ gramatikám. Nicméně tato domněnka se nepotvrdila tím, že byl nalezen algoritmus, který libovolnou $ESPG(k)$ gramatiku převede na ekvivalentní $SPG(k)$ gramatiku. Tedy $SPG(k)$ a $ESPG(k)$ jsou co do vyjadřovací síly ekvivalentní, nicméně přínos $ESPG(k)$ gramatik aspoň spočívá v jednodušším návrhu některých gramatik, které, kdyby měly být navrhovány jako $SPG(k)$, by byly rozsáhlejší a méně přehledné. Dále byly prezentovány návrhy pro modifikaci výpočtu množin $FIRST_k$ a $FOLLOW_k$ tak, aby lépe odrážely vlastnosti programovaných gramatik. Nicméně tyto úpravy nejsou vhodné pro nasazení pro libovolnou programovanou gramatiku, a proto by bylo vhodné tyto návrhy ještě rozpracovat. $SPG(k)$ a $ESPG(k)$ gramatiky jsou silnější než bezkontextové gramatiky, takže z hlediska syntaktické analýzy má smysl se jimi zabývat. Problém ovšem je v časové složitosti, která je řádově kvadratická. Tato složitost je pro praktické nasazení v syntaktické analýze nevhodná (v praxi používaných LL a LR analyzátorech je složitost řádově lineární, neboť je vždy prováděna nejlevější derivace bezkontextové gramatiky, takže není nutné procházet větnou formu jako v $SPG(k)$ nebo $ESPG(k)$ gramatice, kdy je nutné navíc hledat nejlevější výskyt nějakého neterminálu).

Dále byly zpracovány maticové gramatiky, kdy byl prezentován algoritmus pro převod libovolné maticové gramatiky na ekvivalentní programovanou gramatiku. Syntaktická analýza těchto gramatik byla koncipována tak, že byla definována omezení na maticové gramatiky nazvaná $SMG(k)$ resp. $ESMG(k)$. Dále byly vytvořeny algoritmy pro převod těchto dvou gramatik na ekvivalentní $SPG(k)$ resp. $ESPG(k)$ gramatiku. Syntaktická analýza těchto omezení je tak prováděna za pomocí analýzy $SPG(k)$ resp. $ESPG(k)$. Praktické použití těchto omezených maticových gramatik je ovšem velmi obtížné, protože pro všechny sekvence (kromě první) je vyžadováno, aby všechna počáteční pravidla začínala stejnou levou stranou. Tuto část je tedy nutné brát spíše jen jako demonstrační.

Dalšími gramatikami, které byly zpracovány, byly regulárně řízené gramatiky. U nich byl použit podobný mechanismus jako u maticových gramatik. Tvar regulárně řízených gramatik (vlastně omezení na deterministický konečný automat pro regulární množinu R) byl upraven a nazván jako $SRCG(k)$. Poté byl navržen algoritmus, který převede $SRCG(k)$ gramatiku na **neekvivalentní** $SPG(k)$ gramatiku (implementačně na $ESPG(k)$, protože $SPG(k)$ nebyla implementována). Nad touto gramatikou je vstupní slovo otestováno na

členství do jazyka této gramatiky. Pokud je členem, sekvence použitých pravidel je otestována na členství do jazyka konečného automatu, který popisuje regulární množinu R. Až na základě tohoto testu je rozhodnuto, zda testované slovo patří do jazyka $SRCG(k)$ gramatiky. Byla tak vytvořena složená syntaktická analýza $SRCG(k)$ gramatiky za pomocí $SPG(k)$ gramatiky. Porovnání vyjadřovací síly $SRCG(k)$ a $SPG(k)$ gramatik nebylo dořešeno. Nejedná se o jednoduchou otázku, neboť pomocná $SPG(k)$ gramatika může přijmout určité slovo, které bylo vygenerováno za pomocí sekvence pravidel, která ovšem nepatří do regulární množiny R.

Posledními zpracovanými gramatikami byly gramatiky s nahodilým kontextem. S ohledem na syntaktickou analýzu byla použita levě povolující gramatika. Časová složitost tím byla trochu snížena tak, že není nutné zpracovávat dvě množiny ale jen jednu. Pro jednodušší analýzu byla rovněž zvolena 1-kanonická derivace. Dále, s ohledem na vyjadřovací sílu levě povolujících gramatik byly použity kooperující distribuované gramatiky s použitím právě levě povolujících gramatik jako komponenty. Byla definována varianta kooperujících distribuovaných gramatik nazvaná $SLPCD(k)$, která používá právě dvě komponenty. Právě dvě komponenty byly zvoleny jednak proto, že více komponent již nezvyšuje vyjadřovací sílu, a také proto, že je předem známa sekvence použitých komponent (vždy jen přepínání mezi první a druhou komponentou, přičemž generování začíná v první komponentě). $SLPCD(k)$ gramatika používá kombinovanou analýzu (prediktivní tabulka a prohledávání stavového prostoru). Z tohoto důvodu bylo nutné povolit jen pravidla, která nejsou pravidly vymazávacími, protože jinak by členství do jazyka gramatiky nebylo rozhodnutelné. Vyjadřovací síla $SLPCD(k)$ gramatik nebyla bohužel stanovena. Omezení na 1-kanonickou derivaci může vyjadřovací sílu snížit i zvýšit.

V projektu nebyla zmíněna celá řada řízených gramatik. Další kategorií řízených gramatik jsou tzv. *gramatiky s částečným paralelismem*, rovněž zmíněné v [8, 16]. Ty umožňují provádět více přepisů určitého neterminálu v jedné derivaci. Dále by mohly být uvažovány stavové gramatiky, se kterými je možné provádět syntaktickou analýzu podobně jako u $LL(k)$ gramatik [14]. Velkou výzvou by bylo se pokusit o syntaktickou analýzu *Petriho sítí řízených gramatik* [5].

Literatura

[1] Abraham, S.: *Some questions of language theory.* Morristown, NJ, USA: Preceedings of the 1965 conference on Computational linguistics, page 1–11, 1965.

[2] Aho, A. V.; Sethi, R.; Ullman, J. D.: *Compilers - Principles, Techniques, and Tools.* Addison-Wesley, 1986, ISBN 0-321-48681-1.

[3] Csuhaj-varjú, E.; Masopust, T.; Vaszil, G.: *Cooperating Distributed Grammar Systems with Permitting Grammars as Components.* Romainian Journal of Information Science and Technology, page 175–189, 2009.

[4] Dassow, J.; Păun, G.: *Regulated Rewriting in Formal Language Theory.* Berlin: Springer, 1989.

[5] Dassow, J.; Turaev, S.: *Arbitrary Petri net controlled grammars.* Tarragona, Spain: In ForLing '08: Proceedings of the 2nd International Workshop "Non-Classical Formal Languages in Linguistics", 2008.

[6] Ginsburg, S.; Spanier, E. H.: *Control sets on grammars.* Theory of Computing Systems, 1968.

[7] Goldefus, F.; Masopust, T.; Meduna, A.: *Left-forbidding cooperating distributed grammar systems.* Theoretical Computer Science, 2010.

[8] Kot, M.: *Řízené gramatiky.* Diplomová práce, VŠB, Ostrava, 2002.

[9] Křivka, Z.: *Rewriting Systems with Restricted Configurations*. Faculty of Information Technology BUT, 2008, ISBN 978-80-214-3722-7, 131 s.
URL `http://www.fit.vutbr.cz/research/view_pub.php?id=8768`

[10] Martfn-Vide, C.; Mitrana, V.; Păun, G.: *Formal Languages and Applications.* Springer, 2004, ISBN 3-540-20907-7.

[11] Matoušek, J.; Nešetřil, J.: *Kapitoly z diskrétní matematiky.* nakladatelství Karolinum, 2002, ISBN 80-246-0084-6.

[12] Meduna, A.: *Automata and Languages.* Springer, 2000, ISBN 81-8128-333-3.

[13] Meduna, A.; Lukáš, R.: Formální jazyky a překladače - studijní opora, Brno, FIT VUT. 2006.

[14] Navrátil, P.: Parsing Based on State Grammars. VUT FIT.

[15] Rosenkrantz, D. J.: *Programmed grammars and classes of formal language.* Journal of ACM, 1969.

[16] Rozenberg, G.; Salomaa, A.: *Handbook of Formal Languages, Volume 2: Linear Modelling*. Springer, 1997, ISBN 3-540-60648-3.

[17] Sebesta, R. W.: *On context-free programmed grammars*. University of Colorado, 1989.

[18] VirginiaTech: http://courses.cs.vt.edu/∼cs4114/lectures/16/detparsing.ps.

[19] van der Walt, A. P. J.: *Random context grammars*. In Proceeding of Symposium on Formal Languages, 1970.

[20] Zemek, P.: *Canonical Derivations in Programmed Grammars*. Bakalářská práce, VUT FIT, Brno, 2008.

[21] Zemek, P.: *On Erasing Rules in Regulated Grammars*. Diplomová práce, FIT VUT, Brno, 2010.

[22] Šrajer, R.: *Grafická simulace činnosti konečných automatů*. Bakalářská práce, VUT FIT, Brno, 2009.

[23] Češka, M.; Vojnar, T.; Smrčka, A.: Teoretická informatika - studijní opora, Brno, FIT VUT. 2010.

Seznam příloh

- Příloha A — Obsah CD
- Příloha B — Manuál k analyzátoru

Dodatek A

Obsah CD

Přiložené CD obsahuje:

- *prog* — syntaktický analyzátor v jazyce C++
  - *project* — samotné zdrojové kódy a soubor *Makefile*
  - *examples* — příklady konfiguračních souborů řízených gramatik
  - *test.sh* — *bash* skript pro otestování analyzátoru na unixovém OS
  - *test.bat* — *batch* skript pro otestování analyzátoru na OS Windows
- *tex* — zdrojové kódy textové části diplomové práce v systému LaTeX
- *compiled* — přeložená textová část diplomové práce a PDF k obhajobě semestrálního projektu
- *submission.png* — oskenované zadání diplomové práce
- *README.TXT* — soubor s informacemi o obsahu CD

Dodatek B

Manuál k analyzátoru

Činnost analyzátoru je možné ověřit skriptem *test.sh* ve složce *prog*. Na příslušném počítači musí být nainstalován program *make* a překladač pro jazyk C++ z roku 1998. Po přejití do složky *prog/project* je projekt možné přeložit programem *make* přímo. Binární kód nese název *parser* a nápovědu k němu je možné vyvolat takto:

```
./parser --help
```

Výsledky činnosti analyzátoru jsou prezentovány jak na standardním i chybovém výstupu tak i ve formě těchto návratových kódů:

```
STATUS_YES  ~  0
STATUS_NO   ~  1
STATUS_LEX  ~  2
STATUS_INT  ~  3
```

Mimo nápovědy je možné analyzátor použít dvěma způsoby. První způsob provádí pouhou verifikaci vstupního konfiguračního souboru, přičemž ten obsahuje typ a popis určité gramatiky. Verifikace má následující tvar:

```
./parser --verify grammar_file
```

Toto volání zjistí, zda gramatika z daného souboru je skutečně tou gramatikou, za kterou se považuje, a vrací:

```
STATUS_YES  ~  gramatika je daného typu
STATUS_NO   ~  gramatika není daného typu
STATUS_LEX  ~  lexikální chyba při čtení gramatiky
STATUS_INT  ~  neexistující soubor, neznámý typ gramatiky nebo chybný formát
               konfiguračního souboru pro daný typ gramatiky
```

Druhé možné volání analyzátoru provádí testování členství slova v jazyce gramatiky z konfiguračního souboru. Volání je takovéto:

```
./parser --membership grammar_file input_file
```

Testované slovo je přečteno ze souboru `input_file`. Volání vrací:

`STATUS_YES`	$\sim$	`slovo patří do jazyka gramatiky`
`STATUS_NO`	$\sim$	`slovo nepatří do jazyka gramatiky`
`STATUS_LEX`	$\sim$	`lexikální chyba při čtení gramatiky nebo slova`
`STATUS_INT`	$\sim$	`neexistující soubor(y), neznámý typ gramatiky, gramatika není daného typu nebo chybný formát konfiguračního souboru pro daný typ gramatiky`

Neterminály a terminály mohou být jednopísmenné nebo vícepísmenné. Jednopísmenné mohou být zapisovány ihned za sebou. Například zápis AABC nebo AA B C označuje posloupnost jednopísmenných terminálů a/nebo neterminálů, přičemž velikost písmene nerozhoduje o tom, co je terminál a co neterminál, a bílé znaky nejsou vůbec uvažovány. Vícepísmenné terminály a neterminály v konfiguračním souboru musí být umísťovány do úhlových závorek, vícepísmenné terminály v testovaném slově musí být umísťovány do uvozovek. V konfiguračním souboru můžeme například terminál **switch** zapsat jako <SWITCH> a v testovaném slově se tento terminál může vyskytnout jako "SWITCH". Pokud je třeba použít levou nebo pravou úhlovou závorku nebo uvozovky jako symbol, je možné využít escape sekvencí za pomocí zpětného lomítka. Escape sekvence lze použít i uvnitř úhlových závorek resp. uvnitř uvozovek, tedy je možné například zapsat <A\"B> nebo "\<CCC\>".

Struktura a syntaxe konfiguračního souboru

Konfigurační soubor řízené gramatiky je textový soubor s popisem terminálních a neterminálních symbolů a dalších množin podle druhu gramatiky. Soubor může obsahovat komentáře, které ze začátku řádku začínají mřížkou #, a prázdné řádky, které jsou vždy ignorovány. Obsahy množin jsou zapisovány stylem **klíč=hodnoty**, přičemž část **klíč=** musí být zapsána ihned na začátku řádky bez bílých znaků. Tyto záznamy musí být zapisovány v daném pořadí opět podle druhu gramatiky. Jako první záznam je vždy nutné uvést:

```
type=typ_gramatiky(cislo_k)
```

Jako typy gramatiky je možné aktuálně použít **ESPG**, **SMG**, **ESMG**, **SRCG**, **SLPCD**. V kulatých závorkách je uvedeno číslo k, které udává maximální délku predikce pro danou gramatiku. Další položky souboru jsou již závislé na typu gramatiky. Pro přehlednost definujme zápisy:

$\{t\}$	$\sim$	`právě 1 terminál`
$\{n\}$	$\sim$	`právě 1 neterminál`
$\{p\_t\}$	$\sim$	`posloupnost terminálů`
$\{p\_n\}$	$\sim$	`posloupnost neterminálů`
$\{nt\}$	$\sim$	`větná forma`
$\{c\}$	$\sim$	`přirozené číslo`

$ESPG(k)$ gramatika

```
type= ESPG({c})
T=    {p_t}
N=    {p_n}
S=    {n}
# číslo pravidla   LHS    RHS    sigma         phi
{c}:               {n} -> {nt} , {c} {c} ... , {c} ...
...
...
```

Po definici množin T, N a počátečního neterminálu jsou definována pravidla $ESPG$ gramatiky na jednotlivých řádcích. Každé pravidlo musí mít jedinečné číslo. Po zápisu čísla je definováno bezkontextové pravidlo (levá strana, šipka, pravá strana) a poté množiny σ a φ. Tyto množiny obsahují čísla existujících pravidel, která jsou od sebe oddělena aspoň jednou mezerou nebo tabulátorem.

SMG a $ESMG$ gramatika

```
type= [E]SMG({c})
T=    {p_t}
N=    {p_n}
S=    {n}
# seznam dvojic (číslo_sekvence pořadí_v_sekvenci)
F=    {c} {c}, {c} {c}, ...
# co řádek to jedna sekvence bezkontextových pravidel
{c}: {n} -> {nt}, {n} -> {nt}, ...
...
...
```

Množina F (pravidla v módu kontroly výskytu) je zapisována jako posloupnost dvojic oddělených čárkou, přičemž prvky ve dvojici jsou odděleny aspoň jednou mezerou nebo tabulátorem. Dvojice označuje pozici pravidla, které je v módu kontroly výskytu. Například dvojice čísel 2 a 3 říká, že třetí pravidlo (počítáno zleva) v sekvenci číslo 2 je v módu kontroly výskytu. Sekvence jsou sázeny po řádcích a bezkontextová pravidla jsou zapisována za sebou oddělena čárkou.

$SRCG$ gramatika

```
type= SRCG({c})
T=    {p_t}
N=    {p_n}
S=    {n}
R=    {regulární_výraz_nad_čísly_pravidel}
F=    {c} {c} ...
# bezkontextová pravidla
{c}: {n} -> {nt}
...
...
```

Regulární množina R je popsána regulárním výrazem nad abecedou čísel bezkontextových pravidel. Operátory jsou definovány takto:

- $*$ $\sim$ unární operátor iterace
- $.$ $\sim$ binární operátor zřetězení (povinný)
- $+$ $\sim$ binární operátor sjednocení

Priority operátorů jsou klasicky $* > . > +$, přičemž prioritu je možné upravovat kulatými závorkami. Operátor zřetězení je povinný, protože je potom možné rozlišit čísla pravidel s více číslicemi. Pokud máme například definována pravidla číslo 8, 9, 10 a 11, potom může být zapsán regulární výraz: $9.(11 * +8.10) * .8$

$SLPCD$ gramatika

```
type= SLPCD({c})
way=  DEPTH | WIDTH
T=    {p_t}
N=    {p_n}
S=    {n}
# pravidla levě povolující gramatiky z komponenty P1
P1=
{c}: {n} -> {nt} , {p_n}
...
...
# pravidla levě povolující gramatiky z komponenty P2
P2=
{c}: {n} -> {nt} , {p_n}
...
...
```

Konfigurační soubor opět kromě množiny terminálů, neterminálů a počátečního neterminálu obsahuje i výčet pravidel levě povolující gramatiky pro komponenty P_1 a P_2. Dále je možné specifikovat způsob prohledávání stavového prostoru (je-li použito) do hloubky (`DEPTH`) nebo do šířky (`WIDTH`). Může se stát, že prohledávání do hloubky bude rychlejší v případě, že testované slovo patří do jazyka gramatiky, a v případě, že pravidla jsou v konfiguračním souboru „vhodně“ uspořádána.

Konkrétní příklady zápisů výše popsaných gramatik si je možné prohlédnout v souborech s příponou `conf` ve složce `prog/examples`. Zajímavá je gramatika ze souboru `grammar_10.conf`, jež obsahuje $SLPCD(1)$ gramatiku, která popisuje jazyk $\{a^{2^n} \mid n \geq 0\}$ (spočítána algoritmem z [3] na straně 185). Příklad demonstruje obecně exponenciální časovou složitost při prohledávání stavového prostoru v případě, že predikce pomocí tabulky není možná.