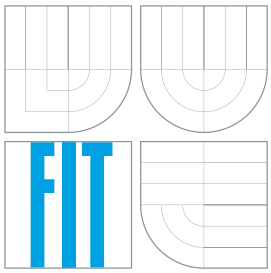
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PATH-CONTROLLED GRAMMARS

CESTAMI ŘÍZENÉ GRAMATIKY

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Bc. ONDŘEJ ADAMEC

SUPERVISOR
VEDOUCÍ PRÁCE

Ing. ZBYNĚK KŘIVKA, Ph.D.

BRNO 2015

Brno University of Technology – Faculty of Information Technology

Department of information systems Academic year 2014/2015

Master thesis assignment

Student: **Adamec Ondřej, Bc.**
Branch: Mathematical Methods in Information Technology
Topic: **Path-controlled Grammars**
Category: Theoretical information technology

Specification:

1. Familiarize yourself with path-controlled grammars and other types of regulated grammars (e.g. state grammars and matrix grammars). Study the state of the art regarding their theoretical properties.
2. Design and formally define a conversion algorithm between path-controlled grammars and other types of regulated grammars according to supervisor's instructions.
3. Implement the algorithm and test it on a set of at least ten grammars. Discuss the complexity.
4. According to supervisor's instructions implement a program for determining whether given grammar generates a given sentence.

Literature:

- Meduna, A.: Automata and Languages: theory and applications, London: Springer, 2000, 916 s. ISBN 1-85233-074-0.
- Koutný, J., Meduna, A.: On Normal Forms and Erasing Rules in Path-Controlled Grammars. Schedae Informaticae. Krakov: 2014, roč. 2013, č. 22, s. 9-18. ISSN 0860-0295.

The Term Project discussion items:

- Items 1 and 2.

Supervisor: **Křivka Zbyněk, Ing., Ph.D., UIFS FIT VUT**
Entry date: Semptember 22, 2014
Submission date: May 27, 2015

Abstract

This thesis deals with path-controlled grammars, which are grammars that place restrictions on the paths in a derivation tree of a context-free grammar. The goal of this thesis is to create an algorithm for conversion between the path-controlled grammars and the state grammars, which is a different type of regulated grammars. Another goal is to study the generative power of path-controlled grammars based on the conversion algorithm. The conversion algorithm is implemented and tested on a number of path-controlled grammars. Also, its complexity is discussed. Finally, a parsing tool for path-controlled grammars is implemented. Complexity of this tool is analyzed as well.

Abstrakt

Tato diplomová práce se zabývá cestami řízenými gramatikami, gramatikami, které kladou omezení na cesty v derivačním stromě bezkontextové gramatiky. Cílem této diplomové práce je tvorba algoritmu pro převod mezi cestami řízenými gramatikami a stavovými gramatikami, což je jiný typ řízené gramatiky. Dalším cílem je na základě tohoto převodního algoritmu studovat vyjadřovací sílu cestami řízených gramatik. Převodní algoritmus je naimplementován v C++ a testován na sadě cestami řízených gramatik. Složitost algoritmu, jak časová, tak prostorová, je diskutována. Také nástroj pro syntaktickou analýzu cestami řízených gramatik je naimplementován. Složitost této analýzy je také diskutována.

Keywords

path-controlled grammars, matrix grammars, state grammars, regulated grammars, formal languages, syntactic analysis.

Klíčová slova

cestami řízené gramatiky, maticové gramatiky, statové gramatiky, řízené gramatiky, formální jazyky, syntaktická analýza.

Citation

Ondřej Adamec: Path-Controlled Grammars, master's thesis, Brno, FIT VUT, 2015

Citace

Ondřej Adamec: Cestami řízené gramatiky, diplomová práce, Brno, FIT VUT, 2015

Path-Controlled Grammars

Declaration

I declare that this thesis is my own work that has been created under the supervision of Zbyněk Křivka. All sources and literature that I have used are duly cited.

.....
Ondřej Adamec
May 27, 2015

Contents

Chapter 1

Introduction

The motivation for regulated rewriting comes from the fact that many languages of interest are not context-free. Rather than using context-sensitive grammars, it may be better, from practical point of view, to place some restrictions on derivation in context-free grammars and thus significantly increasing the generative power.

It is believed that the first mechanism for regulated rewriting was introduced by S. Abraham, in 1965. The *matrix grammars* regulate the derivation in a context-free grammar by defining sequences of rewriting rules, which are applied together in a derivation step.

Many other form of regulated rewriting were introduced and studied since, see [1].

This thesis deals with *path-controlled grammars*, in which a restriction is placed on the paths in a derivation tree of a context-free grammar. This mechanism was introduced by I. Bellert, in 1965, the same year as *matrix grammars*. Surprisingly, this concept was not investigated further until 2001.

In the following chapters we will provide formal definitions of path-controlled grammars and the language they generate. The pumping property and the generative power will be discussed as well.

The question of generative power is still open, since the construction proving the original proposition regarding the generative power using a conversion of path-controlled grammar into a matrix grammar in [2] was proven incorrect in [3]. We will try to address this issue by converting the path-controlled grammar into a different type of grammar — *state grammar*, introduced by T. Kasai in [4].

Chapter 2

Preliminaries

In this chapter, we present all definitions and conventions which are used throughout this thesis. Section 2.1 covers the essential definitions concerning formal languages. Section 2.2 is dedicated to derivation trees. In Section 2.3 we introduce state grammars, one of the mechanisms for regulated rewriting. And finally, in Section 2.4 we mention generalized sequential machine mapping.

We assume that the reader is familiar with the well-known core of set theory, graph theory and other mathematical knowledge on which the following definitions are based on. Definitions are presented formally, without further details or examples.

For most of the definitions we use [5].

2.1 Formal Languages

This section introduces the basic notions of formal languages, grammars and derivations as well as conventions regarding their representation in following chapters.

Definition 1. (Alphabet and symbol). An *alphabet* is a finite, nonempty set of elements called *symbols*.

Definition 2. (Word). Let Σ be an alphabet, then ε is a word over Σ. If x is a word over Σ and $a \in \Sigma$, then xa is a word over Σ. Σ^* denotes the set of all words over Σ and $\Sigma^+ = \Sigma^* - \{\varepsilon\}$.

Definition 3. (Language). Let Σ be an alphabet and let $L \subseteq \Sigma^*$. Then, L is a *language* over Σ.

Definition 4. (Unrestricted grammar, generated language) *Unrestricted grammar* is a quadruple $G = (V, T, P, S)$, where

V is a total alphabet,

$T \subset V$ is an alphabet of terminal symbols,

$P \subseteq V^*(V - T)V^* \times V^*$ is a finite binary relation,

$S \in (V - T)$ is the start symbol.

$(V - T)$ is the set of nonterminal symbols, denoted by N. The nonterminal symbols are represented by upper case letters, the terminal symbols are represented by lower case letters. The elements of P are called *productions* or *rules*, therefore, P is referred to as the *set of productions* or the *set of rules*. A production, $(x, y) \in P$ is symbolically written as $x \to y$. The productions may be labeled for easier reference, then we write $p : x \to y$, where p is the label of the production (x, y). The left-hand side of p, represented by x, is denoted by $lhs(p)$ and the right-hand side of p, represented by y, is denoted by $rhs(p)$. A production with $|rhs(p)| = 0$ is called *ε–production* and is written as $x \to \varepsilon$. For rules with the same left-hand side we often use the notation $x \to y|z \in P$ which means $x \to y \in P$ and $x \to z \in P$.

A *direct derivation* is a binary relation on V^* defined as follows. Given $p : x \to y \in P$ and $u, v \in V^*$, uxv directly derives uyv according to p in G denoted by $uxv \Rightarrow_G uyv\,[p]$ or $uxv \Rightarrow uyv$ for short. Alternatively, we may write $uxv \overset{p}{\Rightarrow} uyv$ for better readability.

A *zero–step derivation* from u to u for any $u \in V^*$ according to ε in G is written as $u \Rightarrow^0_G u\,[\varepsilon]$. An *$n$–step derivation* from u_0 to u_n for some $n \geq 1$ is a sequence of direct derivations $u_{i-1} \Rightarrow_G u_i\,[p_i]$, where $p_i \in P$, for $1 \geq i \geq n$. We write $u_o \Rightarrow^n_G u_n\,[p_1 p_2 \dots p_n]$. Moreover, $\Rightarrow^+_G$ denotes transitive closure of $\Rightarrow_G$ and $\Rightarrow^*_G$ denotes transitive and reflexive closure of $\Rightarrow_G$.

If $S \Rightarrow^*_G w$, where $w \in V^*$, then w is a *sentential form* of G. A sentential form w, such that $w \in T^*$ is a *word* generated by G. The *language generated* by G, denoted by $L(G)$ is defined as $L(G) = \{w \in T^* | S \Rightarrow^*_G w\}$.

A language L is *recursively enumerable* if there is an unrestricted grammar G, such that $L(G) = L$. The *class of recursively enumerable languages* is denoted by **RE**.

Definition 5. (Context-sensitive grammar, context-sensitive language). Let $G = (V, T, P, S)$ be an unrestricted grammar. G is a *context-sensitive grammar* if for all $p \in P : |lhs(p)| \leq |rhs(p)|$. A language L is *context-sensitive language* if there is a context-sensitive grammar G, such that $L(G) = L$. The *class of context-sensitive languages* is denoted by **CS**.

Definition 6. (Context-free grammar, context-free language). Let $G = (V, T, P, S)$ be an unrestricted grammar. G is a *context-free grammar* if for all $p \in P : lhs(p) \in (V - T)$. A language L is *context-free language* if there is a context-free grammar G, such that $L(G) = L$. The *class of context-free languages* is denoted by **CF**.

Lemma 1. *(Pumping lemma for context-free languages). Let L be a context-free language. Then, there is a constant k, such that each word $z \in L$ with $|z| \geq k$ can be written in the form $z = uvwxy$, such that $vx \neq \varepsilon$, $|vwx| \leq k$ and $uv^i wx^i y \in L$, for all $i \geq 0$.*

Definition 7. (Linear grammar, linear language). Let $G = (V, T, P, S)$ be an unrestricted grammar. G is a *linear grammar* if for all $p \in P : lhs(p) \in (V - T)$ and $rhs(p) \in T^*((V - T) \cup \{\varepsilon\})T^*$. A language L is *linear language* if there is a linear grammar G, such that $L(G) = L$. The *class of linear languages* is denoted by **LIN**.

Lemma 2. *(Pumping lemma for linear languages). Let L be a linear language. Then, there is a constant k, such that each word $z \in L$ with $|z| \geq k$ can be rewritten in the form $z = uvwxy$, such that $vx \neq \varepsilon$, $|uvxy| \leq k$ and $uv^i wx^i y \in L$, for all $i \geq 0$.*

2.2 Derivation Trees

The idea of path-controlled grammars revolves around derivation trees, this section provides necessary definitions concerning them.

Definition 8. (Production tree). Let $G = (V, T, P, S)$ be a context-free grammar and $p \in P$. The *production tree* $pt(p)$, corresponding to p is a labelled elementary tree t, such that $lhs(p)$ labels the root node and the leaves are defined as follows:

1. If $|rhs(p)| = 0$, then t has one leaf node labelled by ε.
2. If $|rhs(p)| \geq 1$, then t has $|rhs(p)|$ leaf nodes labelled by the symbols appearing in $rhs(p)$ from left to right.

Definition 9. (Derivation tree). Let $G = (V, T, P, S)$ be a context-free grammar. A *derivation tree* of G is a labelled tree t, satisfying following two conditions:

1. The root node is labelled by a nonterminal symbol $A \in V - T$.
2. Every elementary subtree t' in t represents $pt(p)$ for some $p \in P$.

Definition 10. (Derivation tree corresponding to derivation). Let $G = (V, T, P, S)$ be a context-free grammar. A derivation tree of G corresponds to a derivation in G in a way recursively defined as follows:

1. Let t be a derivation tree consisting of a single node, such that the root node is labelled by $A \in V - T$. Then, t corresponds to $A \Rightarrow A\,[\varepsilon]$ in G.
2. Let t be a derivation tree corresponding to $A \Rightarrow^* u\,lhs(p)\,v\,[\pi]$ in G for some $p \in P$. Then, the derivation tree corresponding to $A \Rightarrow^* u\,lhs(p)\,v\,[\pi] \Rightarrow u\,rhs(p)\,v\,[p]$ is constructed by appending $pt(p)$ to the $|u| + 1$st leaf node from left in t.

2.3 State Grammars

State Grammars were introduced by T. Kasai in 1970 in [4]. This formal model is a context-free grammar extended by the notion of states. The states restrict which rules are applicable in a derivation step, thus regulating the derivation in order to increase the generative power of context-free grammars.

Definition 11. (State grammar). *A state grammar* is a 6-tuple $G = (Q, V, T, P, q_0, S)$, where

Q is an alphabet of *states*,

V is a total alphabet,

$T \subset V$ is an alphabet of terminal symbols,

$P \subseteq (Q \times N) \times (Q \times (N \cup T)^+)$ is a finite set of productions,

$q_0 \in Q$ is the initial state,

$S \in V - T$ is the start symbol.

Definition 12. (Direct derivation, derivation). Given a state grammar $G = (Q, N, T, P, q_0, S)$, let $\Rightarrow_G$ be a relation on $Q \times (N \cup T)^*$ defined as follows: Let $p, q \in Q$, $w = uAv$, $u, v \in (N \cup T)^*$ and $A \in N$. If A is the *leftmost* occurrence of a rewritable nonterminal in w and $(p, A) \to (q, x) \in P$, then we write $(p, uAv) \Rightarrow_G (q, uxv)$ and say that (p, uAv) *directly derives* (q, uxv) in G.

Analogically to the corresponding definitions for the unrestricted grammar (see Definition 4), we define for all $n \geq 0$: $\Rightarrow_G^n$, $\Rightarrow_G^*$ and $\Rightarrow_G^+$.

Definition 13. (State language). Given a state grammar G, the *state language* is defined as $L(G) = \{w \in V^+ \mid (q_o, S) \Rightarrow_G^* (q, w)$, for some $q \in Q\}$.

Furthermore, Kasai defines an *n–limited derivation*, a *degree of state grammar*, a *degree of state language* and consequentially the hierarchy of state languages. For more details on this hierarchy see [4]. For our purposes it is enough to know that $\mathscr{L}_\omega$ denotes the family of all state languages.

Theorem 1. (Theorem 2 in [4]). *$\mathscr{L}_\omega$ is identical to the family of context-sensitive languages.*

2.4 Generalized Sequential Machine Mapping

We will need a way to translate one language into another. *Generalized sequential machine mapping* is one of the models used for translating.

Definition 14. (Generalized sequential machine). A *generalized sequential machine* is a 6-tuple $M = (Q, \Sigma, \Omega, \tau, q_0, F)$, where

Q is a finite set of *states*,

Σ is an alphabet of input symbols,

Ω is an alphabet of output symbols,

$\tau \subseteq (Q \times \Sigma) \times (Q \times \Omega^*)$ is a finite *transition function*,

$q_0 \in Q$ is the initial state,

$F \subseteq Q$ is a finite set of final states.

Definition 15. (Generalized sequential machine mapping). Let $M = (Q, \Sigma, \Omega, \tau, q_0, F)$ be a generalized sequential machine. Given an input word $u \in \Sigma^*$, *generalized sequential machine mapping* of u, *gsm mapping* for short, is defined as $GSM_M(u) = \{v \in \Omega^* \mid (q, v) \in \tau(q_0, u)$, for some $q \in F\}$.

Given a language L, a gsm mapping of L is defined as $GSM_M(L) = \bigcup_{u \in L} GSM_M(u)$.

Chapter 3

Path-Controlled Grammars

In this chapter we will introduce the *path-controlled grammars*. First, we will mention a little bit of historical context for these grammars in Section 3.1, then, in Section 3.2, we formally define them, and the language they generate. A simple example of a path-controlled grammar will be presented in Section 3.3. We will continue with showing a significant property these grammars have, that is the *pumping property* in Section 3.4. And finally, in Section 3.5, we discuss the generative power.

3.1 History

In the same year when S. Abraham introduced the *matrix grammars* in [6], I. Bellert published a paper introducing several ideas of regulated rewriting. Some of them, *tuple grammars*, *matrix grammars*, were investigated later by several authors.

However, one of the idea, to impose restriction on the paths in a derivation tree of a context-free grammar, was overlooked until 2001, when S. Marcus, C. Martín-Vide, V. Mitrana and Gh. Păun revisited the idea in [2].

3.2 Definition

The formal definition of path-controlled grammars and the language they generate was given in [2] as follows.

Definition 16. Given two context-free grammars, G_1 and G_2, where G_2 generates a language over the total alphabet of G_1. A string w generated by G_1 is accepted only if there is a derivation tree τ of w with respect to G_1 such that there is a path in τ, from the root to a leaf node, which is described by a string which is in $L(G_2)$. We call G_1 the *generating grammar* and G_2 the *controlling grammar*. We denote a pair $\gamma = (G_1, G_2)$ as a *path-controlled grammar*.

$path(\tau)$ denotes the language of all strings describing paths in a derivation tree τ, by $path(x)$ the union of all languages $path(\tau)$ where τ is a derivation tree for x in some grammar G, finally, we denote by $path(G)$ the union of all these languages.

We define the language generated by a path-controlled grammar.

Definition 17. Let $\gamma = (G_1, G_2)$ be a path-controlled grammar. The language generated by γ is defined $L(\gamma) = \{w \in L(G_1) \mid path(w) \cap L(G_2) \neq \emptyset\}$.

$PC(F_1, F_2)$ denotes the family of languages generated by path-controlled grammars $\gamma = (G_1, G_2)$, where G_1 is of type F_1 and G_2 is of type F_2. In this study we consider $F_i \in \{REG, LIN, CF\}$ for $i \in \{1, 2\}$.

3.3 Example

Consider the linear grammar $G = (V, T, P, S)$ with

$$\begin{aligned} V &= \{S, B, D, a, b, c, d\}, \\ T &= \{a, b, c, d\}, \\ P &= \{S \to aSd, S \to aBd, B \to bBc, B \to D, D \to bc\}, \end{aligned}$$

generating $L(G) = \{a^n b^m c^m d^n \mid n, m \geq 1\}$ and the linear grammar $G' = (V', V, P', S')$ with

$$\begin{aligned} V' &= \{S', A', S, B, D, a, b, c, d\}, \\ P' &= \{S' \to SA'BDb, S' \to SBDb, A' \to SA'B, A' \to SB\}, \end{aligned}$$

generating $L(G') = \{S^n B^n D b \mid n \geq 1\}$. We construct the path-controlled grammar $\gamma = (G, G')$ where G is the generating grammar and G' is the controlling grammar.

Figure 3.1 shows the derivation tree of the word $aabbccdd$ in γ. It is easy to see that γ generates $L(\gamma) = \{a^n b^n c^n d^n \mid n \geq 1\}$, which is not linear nor context-free.

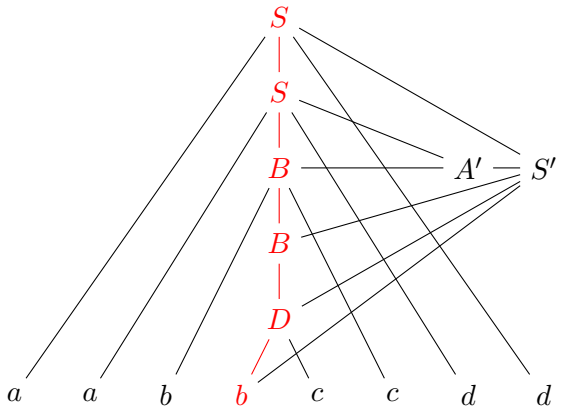


Figure 3.1: Derivation tree of the word $aabbccdd$ in γ.

Using two, fairly simple, linear grammars, we achieved a significant increase in generative power. The generative power of path-controlled grammars is discussed in detail in Section 3.5.

3.4 Pumping Lemma

A pumping property can be directly obtained for languages in the family $PC(CF, CF)$.

Theorem 2. (Proposition 7 in [2]). *If $L \subseteq V^*$, $L \in PC(CF, CF)$, then there are two constants p and q such that each string $z \in L$ with $|z| > p$ can be written in the form $z = u_1 v_1 u_2 v_2 u_3 v_3 u_4 v_4 u_5$, such that $0 < |v_1 v_2 v_3 v_4| \leq q$ and $u_1 v_1^i u_2 v_2^i u_3 v_3^i u_4 v_4^i u_5 \in L$ for all $i \geq 1$.*

For a proof (partially formal) see Proposition 7 in [2].

Analogically to context-free grammars having special form of the pumping lemma for linear languages, the path-controlled grammars have a special form of pumping lemma for languages in $PC(LIN, LIN)$.

Theorem 3. (Proposition 8 in [2]). *If $L \subseteq V^*$, $L \in PC(LIN, LIN)$, then there are two constants p and q such that each string $z \in L$ with $|z| > p$ can be written in the form $z = u_1v_1u_2v_2u_3v_3u_4v_4u_5$, such that $0 < |v_1v_2v_3v_4| \leq q$, $|u_1v_1v_4u_5| \leq q$ and $u_1v_1^iu_2v_2^iu_3v_3^iu_4v_4^iu_5 \in L$ for all $i \geq 1$.*

3.5 Generative Power

When the controlling grammar is regular, we do not increase the generative power of the generating grammar, whether regular, linear, or context-free.

Theorem 4. (Proposition 2 in [2]). *$F = PC(F, REG)$, for all $F \in \{REG, LIN, CF\}$.*

When the generating grammar is regular, the resulting path-controlled grammar does not exceed the power of the controlling grammar, whether linear or context-free.

Theorem 5. (Proposition 3 in [2]). *$PC(REG, F) \subseteq F$, for all $F \in \{LIN, CF\}$.*

Languages, linear or context-free, with words of length one, cannot be generated by a path-controlled grammar with the generating grammar being regular. However, the following theorem holds.

Theorem 6. (Proposition 4 in [2]). *If L is a language in $F \in \{LIN, CF\}$ without words of length one, then $L \in PC(REG, F)$.*

Consider the language $L = \{a^nb^nc^nd^n \mid n \geq 1\}$ from Section 3.3. L belongs to the family $PC(LIN, LIN)$, however the concatenation LL does not, because it does not satisfy the pumping property for this family of languages (see Theorem 3).

Theorem 7. (Consequence in [2]). *The family of languages $PC(LIN, LIN)$ is not closed under concatenation.*

There are context-free languages which are not in $PC(LIN, LIN)$. For a proof of this claim see Proposition 9 in [2]. Here, let us state the consequences of this.

Theorem 8. (Proposition 10 in [2]). (i) $CF - PC(LIN, LIN) \neq \emptyset$. (ii) *The inclusion $PC(LIN, LIN) \subset PC(CF, CF)$ is proper.*

Theorem 9. (Consequence in [2]). *$PC(LIN, LIN)$ is incomparable with* **CF**.

A relation between path-controlled grammars and matrix grammars was proposed, making MAT an upper boundary for $PC(CF, CF)$.

Theorem 10. (Proposition 6 in [2]). *$PC(CF, CF) \subseteq MAT$.*

As a consequence of the pumping lemma (see Theorem 2), the inclusion is proper.

Theorem 11. (Consequence in [2]). *The inclusion $PC(CF, CF) \subset MAT$ is proper.*

However, the construction used in the proof of Theorem 10 in [2] was proven incorrect in [3], making theorems 10 and 11 debatable.

Chapter 4

Conversion into a State Grammar

In [2], matrix grammars were used as an upper boundary of the generative power of path-controlled grammars. However, the construction was proven incorrect in [3]. The problem was that the matrix grammars cannot ensure in any way that the string generated by the controlling grammar, that is the controlling path, is processed from left to right, that is from the root to a leaf node, if there was some symbol occurring more than once.

Using some form of matrix grammar with a leftmost restriction on its derivation ([1], [7]) might be an option. However, we explore a different way, state grammars.

This chapter covers the algorithm for the conversion between path-controlled and state grammars. Section 4.1 introduces the algorithm itself, Section 4.2 explains the ideas behind the algorithm in detail using an example path-controlled grammar. And finally, in Section 4.3 we prove the correctness of the algorithm and discuss the generative power of path-controlled grammars.

4.1 Conversion Algorithm

Proposition 1. Given a path-controlled grammar $\gamma = (G, G')$, we can find a state grammar G'' and a GSM mapping M, such that $GSM_M(L(G'')) = L(\gamma)$.

Proof. Let $\gamma = (G, G')$ be a path-controlled grammar with context-free generating grammar $G = (V, T, S, P)$ and context-free controlling grammar $G' = (V', V, S', P')$. Without loss of generality we may assume that $L(G') \subseteq \{S\}N^*T$. We define a state grammar $G'' = (Q, V'', T \cup \{\diamond\}, P'', q_0, S'')$ with

$$
\begin{aligned}
Q &= \{q_0, q_1, q_F\} \cup \{q_X \mid X \in V\ \}, \\
V'' &= \{X, \hat{X}, \overline{X}, \underline{X} \mid X \in V\} \cup N' \cup \{\diamond\}, \\
P'' = \{&(q_0, S'') \to (q_1, \hat{S}S')\} \cup && \text{(I)} \\
&\{(q_1, \underline{A}) \to (q_1, u\hat{X}v) \mid A\ \to uXv \in P, X \in V, u, v \in V^*\} \cup && \text{(II)} \\
&\{(q_1, A) \to (q_1, h(x)) \mid A\ \to x \in P'\} \cup && \text{(III)} \\
&\{(q_1, \overline{X}) \to (q_X, \diamond) \mid X \in V\ \} \cup && \text{(IV)} \\
&\{(q_A, \hat{A}) \to (q_1, \underline{A} \mid A\ \in N\}) \cup && \text{(V)} \\
&\{(q_a, \hat{a}) \to (q_F, a) \mid a \in T\} \cup && \text{(VI)} \\
&\{(q_F, A) \to (q_F, x) \mid A\ \to x \in P\}, && \text{(VII)}
\end{aligned}
$$

where $N = V - T$, $N' = V' - V$, $\diamond \notin T$ and h is a homomorphism from $(V \cup N')^*$ into $(N' \cup \{\overline{X} \mid X \in V\})^*$ defined by $h(X) = X$, if $X \in N'$, $h(X) = \overline{X}$, if $X \in V$, and $h(\varepsilon) = \varepsilon$.

For easier reference, rules are divided into seven groups labelled by roman numerals. We will refer to them as *type* $\{I$–$VII\}$ *rules.*

Next, we define a gsm M. M reads its input and leaves it unchanged until the symbol $\diamond$ occurs. Then, M begins to remove the rest of its input. If any symbol other than $\diamond$ is read, it is considered invalid, M is blocked and no input is produced.

Then, $GSM_M(L(G'')) = L(\gamma)$

4.2 Illustration

The core idea behind this mechanism is to simulate simultaneous derivation in the generating and the controlling grammar in such way that the derivation tree of the resulting word contains a path described by the controlling grammar.

How is this achieved may not be very clear from the formal definition alone, considering the amount of special symbols and additional rules needed. Hence, let us illustrate the mechanism with an example before providing a full proof.

Consider the path-controlled grammar $\gamma = (G, G')$ where $G = (V, T, S, P)$ is context-free grammar with

$$\begin{aligned} V &= \{S, A, B, C, b, c\}, \\ T &= \{b, c\}, \\ P &= \{S \to AA, A \to B, A \to C, B \to bB, B \to b, C \to cC, C \to c\}, \end{aligned}$$

and $G' = (V', V, S', P')$ is context-free grammar with

$$\begin{aligned} V' &= \{S', A', S, A, B, C, b, c\}, \\ P' &= \{S' \to SAA'b, A' \to BA'B, A' \to BB\}. \end{aligned}$$

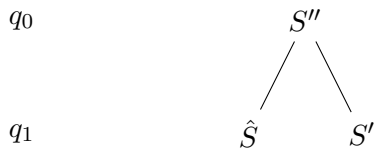
Using the proposed conversion algorithm we construct the state grammar $G'' = (Q, V'', T \cup \{\diamond\}, P'', q_0, S'')$ with

$$\begin{aligned} Q &= \{q_0, q_1, q_S, q_A, q_B, q_C, q_b, q_c, q_F\}, \\ V'' &= \{S, A, B, C, b, c, \hat{S}, \hat{A}, \hat{B}, \hat{C}, \hat{b}, \hat{c}, \overline{S}, \overline{A}, \overline{B}, \overline{C}, \overline{b}, \overline{c}, \underline{S}, \underline{A}, \underline{B}, \underline{C}, \underline{b}, \underline{c}, S', A', \diamond\} \\ P'' &= \{0 : (q_0, S'') \to (q_1, \hat{S}S')\} \cup \\ &\quad \{1 : (q_1, \underline{S}) \to (q_1, \hat{A}A), 2 : (q_1, \underline{S}) \to (q_1, A\hat{A}), 3 : (q_1, \underline{A}) \to (q_1, \hat{B}), \\ &\quad 4 : (q_1, \underline{A}) \to (q_1, \hat{C}), 5 : (q_1, \underline{B}) \to (q_1, \hat{b}B), 6 : (q_1, \underline{B}) \to (q_1, b\hat{B}), \\ &\quad 7 : (q_1, \underline{B}) \to (q_1, \hat{b}), 8 : (q_1, \underline{C}) \to (q_1, \hat{c}C), 9 : (q_1, \underline{C}) \to (q_1, c\hat{C}), \\ &\quad 10 : (q_1, \underline{C}) \to (q_1, \hat{c})\} \cup \\ &\quad \{11 : (q_1, S') \to (q_1, \overline{SA}A'\overline{b}), 12 : (q_1, A') \to (q_1, \overline{B}A'\overline{B}), \\ &\quad 13 : (q_1, A') \to (q_1, \overline{BB})\} \cup \\ &\quad \{14 : (q_1, \overline{S}) \to (q_S, \diamond), 15 : (q_1, \overline{A}) \to (q_A, \diamond), 16 : (q_1, \overline{B}) \to (q_B, \diamond), \\ &\quad 17 : (q_1, \overline{C}) \to (q_C, \diamond), 18 : (q_1, \overline{b}) \to (q_b, \diamond), 19 : (q_1, \overline{c}) \to (q_c, \diamond)\} \cup \\ &\quad \{20 : (q_S, \hat{S}) \to (q_1, \underline{S}), 21 : (q_A, \hat{A}) \to (q_1, \underline{A}), 22 : (q_B, \hat{B}) \to (q_1, \underline{B}), \end{aligned}$$

$$23 : (q_C, \hat{C}) \to (q_1, \underline{C}),\ \cup$$
$$\{24 : (q_b, \hat{b}) \to (q_F, b), 25 : (q_c, \hat{c}) \to (q_F, c)\} \cup$$
$$\{26 : (q_F, S) \to (q_F, AA), 27 : (q_F, A) \to (q_F, B), 28 : (q_F, A) \to (q_F, C),$$
$$29 : (q_F, B) \to (q_F, bB), 30 : (q_F, B) \to (q_F, b), 31 : (q_F, C) \to (q_F, cC),$$
$$32 : (q_F, C) \to (q_F, c)\}.$$

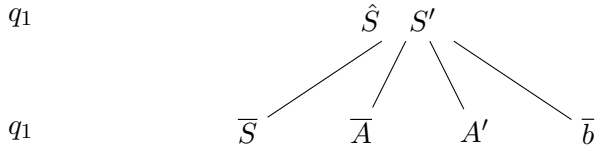
The derivation starts with generating start symbols for both grammars (rule 0), generating grammar start symbol on the left side and controlling grammar start symbol on the right side. The positions are important, since the derivation in state grammars is defined as a leftmost derivation. From this point, we will refer to the substring generated by the generating grammar, G, as the *left side* and to the substring generated by the controlling grammar, G', as the *right side*.

The hat symbol indicates the next symbol on the controlling path. The necessity of this will be explained later. The derivation and its corresponding derivation tree follows.



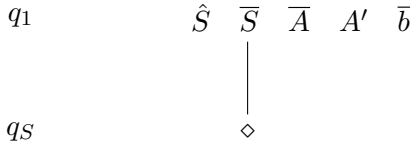
$$(q_0, S'') \Rightarrow (q_1, \hat{S}S')\ [0]$$

Both derivation in G and derivation in G' occurs in q_1, with G having the priority. However, before rewriting the hatted symbol on the left side, we must check whether it is the actual next symbol on the controlling path. The derivation in G' goes on until a symbol $\overline{X}$, for some $X \in V$, is generated (rules 11 – 13). The symbol is overlined in order to differentiate it from its corresponding counterpart on the left side.



$$(q_1, \hat{S}S') \Rightarrow (q_1, \hat{S}\overline{S}\overline{A}A'\overline{b})\ [11]$$

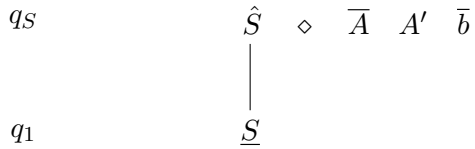
The leftmost overlined symbol $\overline{X}$, for some $X \in V$ is the next symbol on the controlling path. It is rewritten to $\diamond$ to indicate that it is being processed and the grammar enters the corresponding state, q_X (rules 14 – 19).



$$(q_1, \hat{S}\overline{S}\overline{A}A'\overline{b}) \Rightarrow (q_S, \hat{S}\diamond\overline{A}A'\overline{b})\ [14]$$

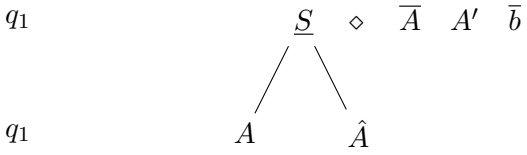
When a symbol $X \in V$ is being processed, that is when the grammar is in q_X, only the corresponding hatted symbol $\hat{X}$ on the left side can be rewritten (rules 20 – 23). It is

rewritten to an underlined symbol $\underline{X}$ which means that the symbol X is the actual next symbol on the controlling path to be rewritten and the grammar returns to q_1.



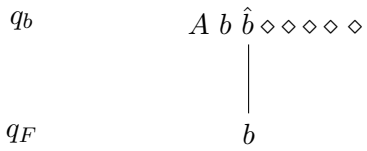
$$(q_S, \hat{S} \diamond \overline{A}A'\overline{b}) \Rightarrow (q_1, \underline{S} \diamond \overline{A}A'\overline{b})\ [20]$$

The underlined symbol $\underline{X}$, for some $X \in V$ is rewritten according to some rule $p \in P$ in G (rules 1 – 10). Note that exactly one symbol in $rhs(p)$ is hatted in the process. Let's repeat that the hat indicates where the controlling path should continue. Necessity of this is clear from the following example. If $rhs(p)$ contains the same symbol X more than once, like in our example, we need a way to select which one will be considered as the next symbol on the controlling path. Without this mechanism only the leftmost occurrence of X would ever be rewritten because of the leftmost restriction on derivation in state grammar.



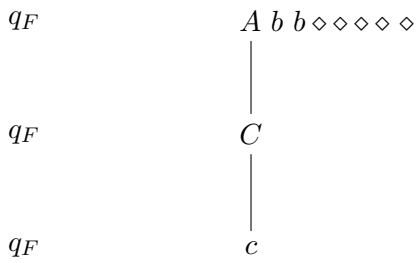
$$(q_1, \underline{S} \diamond \overline{A}A'\overline{b}) \Rightarrow (q_1, A\hat{A} \diamond \overline{A}A'\overline{b})\ [2]$$

When the grammar is in q_a, for some $a \in T$, it means that the end of the controlling path was reached and is being processed. If there is a corresponding hatted symbol $\hat{a}$, the hat is removed and the grammar enters q_F which indicates that the derivation tree of the left side contains a controlling path (rules 24, 25).



$$(q_b, Ab\hat{b} \diamond \diamond \diamond \diamond\diamond) \Rightarrow (q_F, Abb \diamond \diamond \diamond \diamond\diamond)\ [24]$$

In q_F, all remaining derivations in G not controlled by the controlling path are made (rules 26 – 32). Any other derivations are not allowed and the grammar cannot leave q_F.



$(q_F, Abb\diamond\diamond\diamond\diamond\diamond) \Rightarrow (q_F, Cbb\diamond\diamond\diamond\diamond\diamond)$ [28] $\Rightarrow (q_F, cbb\diamond\diamond\diamond\diamond\diamond)$ [32]

Figure 4.1 shows the complete derivation tree of the word $cbb\diamond\diamond\diamond\diamond\diamond$ in G'' and Figure 4.2 shows the corresponding derivations.

Finally, the word $cbb\diamond\diamond\diamond\diamond\diamond$ is given as an input to the gsm M. The prefix cbb is read and left unchanged, then, with the first occurrence of $\diamond$, M begins to remove the rest of the input and since the suffix does not contain any invalid symbols, M successfully finishes, producing cbb.

Clearly, $cbb \in L(G)$ and the derivation tree of cbb contains a path $SABBb \in L(G')$.

4.3 Generative Power

To show that $GSM_M(L(G'')) = L(\gamma)$ we need to prove the following two claims.

Claim 1. A derivation $S \Rightarrow_G^* xay$ such that $a \in T$, $x, y \in T^*$, with a derivation tree containing a path $A_1A_2 \dots A_m a \in L(G')$ exists if and only if $xay\diamond^{m+1} \in L(G'')$.

Only-If Part: That is, if there is a derivation $S \Rightarrow_G^* xay$, such that $a \in T$, $x, y \in T^*$, with a derivation tree containing a path $A_1A_2 \dots A_m a \in L(G')$, then $xay\diamond^{m+1} \in L(G'')$. We will prove this by induction on $m \geq 1$.

Basis: Let $m = 1$, then the derivation tree of $S \Rightarrow_G xay$ contains a path Sa, for some $a \in T$, and there exists a derivation in G'':

$(q_0, S'') \Rightarrow (q_1, \hat{S}S') \Rightarrow^* (q_1, \hat{S}\overline{S}A') \Rightarrow (q_S, \hat{S} \diamond A') \Rightarrow (q_1, \underline{S} \diamond A') \Rightarrow (q_1, A_x\hat{a}A_y \diamond A') \Rightarrow^* (q_1, A_x\hat{a}A_y \diamond \overline{a}) \Rightarrow (q_a, A_x\hat{a}A_y \diamond\diamond) \Rightarrow (q_F, A_xaA_y \diamond\diamond) \Rightarrow^* (q_F, xay \diamond\diamond)$.

Therefore, $xay\diamond^2 \in L(G')$.

Induction Hypothesis: Let us suppose that the only-if part holds for all controlling paths of length m or less, for $m \geq 1$.

Induction Step: Consider a derivation $S \Rightarrow_G^* xay$, such that $a \in T$, $x, y \in T^*$, with a derivation tree containing a path $A_1A_2 \dots A_mA_{m+1}a \in L(G')$. By the induction hypothesis, for a derivation $S \Rightarrow_G^* xay$, such that $a \in T$, $x, y \in T^*$ with a derivation tree containing a path $A_1A_2 \dots A_m a$, there is a word $xay\diamond^{m+1} \in L(G'')$. Consequently, there is a derivation in G'':

$(q_0, S'') \Rightarrow^* (q_1, A_x\underline{A_m}A_y\diamond^m A') \Rightarrow (q_1, A_x\hat{a}A_y\diamond^m A') \Rightarrow^* (q_1, A_x\hat{a}A_y\diamond^m\overline{a}) \Rightarrow^* (q_F, xay\diamond^{m+1})$.

Then, for a derivation $S \Rightarrow_G^* xay$ with a derivation tree containing a path $A_1A_2 \dots A_mA_{m+1}a$, there must exist a derivation in G'':

$(q_0, S'') \Rightarrow^* (q_1, A_x\underline{A_m}A_y\diamond^m A') \Rightarrow (q_1, A_x\hat{A_{m+1}}A_y\diamond^m A') \Rightarrow^* (q_1, A_x\hat{A_{m+1}}A_y\diamond^m\overline{A_{m+1}}A') \Rightarrow^* (q_F, xay\diamond^{m+2})$.

We can see that in this derivation, $\underline{A_m}$ is rewritten to $\hat{A_{m+1}}$ instead of $\hat{a}$. This means that G'' processed one more nonterminal symbol on the controlling path, A_{m+1}, rewriting it to $\diamond$.

Then, $xay\diamond^{m+2} \in L(G'')$, therefore, the only-if part holds.

If Part: That is, if there is a word $xay\diamond^{m+1} \in L(G'')$, then there exists a derivation $S \Rightarrow_G^* xay$ with a derivation tree containing a path $A_1A_2 \dots A_m a \in L(G')$. We will prove this by induction on $m \geq 1$.

Basis: Let $m = 1$, then G'' finished its derivation in final configuration $(q_F, xay \diamond \diamond)$, such that $a \in T$, $x, y \in T^*$, and there must have been a derivation in G'':

$(q_F, xay\diamond\diamond) \,^*\Leftarrow (q_F, A_x a A_y \diamond\diamond) \Leftarrow (q_a, A_x \hat{a} A_y \diamond\diamond) \Leftarrow (q_1, A_x \hat{a} A_y \diamond \overline{a}) \,^*\Leftarrow (q_1, A_x \hat{a} A_y \diamond A') \Leftarrow (q_1, \underline{S} \diamond A') \Leftarrow (q_S, \hat{S} \diamond A') \Leftarrow (q_1, \hat{S}\overline{S}A') \,^*\Leftarrow (q_1, \hat{S}S') \Leftarrow (q_0, S'')$.

Therefore, there must be a derivation $S \Rightarrow_G^* xay$ with a derivation tree containing a path $Sa \in L(G')$.

Induction Hypothesis: Let us suppose that the if-part holds for all suffixes of length $m + 1$ or less, for $m \geq 1$.

Induction Step: Consider $xay\diamond^{m+2} \in L(G'')$, such that $a \in T$, $x, y \in T^*$. By the induction hypothesis, for $xay\diamond^{m+1} \in L(G'')$, such that $a \in T$, $x, y \in T^*$, there exists a derivation $S \Rightarrow_G^* xay$ with derivation tree containing a path $A_1A_2 \dots A_m a$. Consequently, for the final configuration $(q_F, xay\diamond^{m+1})$ there must have been a derivation in G'':

$(q_F, xay\diamond^{m+1}) \,^*\Leftarrow (q_1, A_x \hat{a} A_y \diamond^m \overline{a}) \,^*\Leftarrow (q_1, A_x \hat{a} A_y \diamond^m A') \Leftarrow (q_1, A_x \underline{A_m} A_y \diamond^m A') \,^*\Leftarrow (q_0, S'')$.

Then, for final configuration $(q_F, xay\diamond^{m+2})$ there must have been a derivation in G'':

$(q_F, xay\diamond^{m+2}) \,^*\Leftarrow (q_1, A_x \hat{A_{m+1}} A_y \diamond^m \overline{A_{m+1}} A') \,^*\Leftarrow (q_1, A_x \hat{A_{m+1}} A_y \diamond^m A') \Leftarrow (q_1, A_x \underline{A_m} A_y \diamond^m A') \,^*\Leftarrow (q_0, S'')$.

We can see that in this derivation, $\hat{a}$ was rewritten from $\underline{A_{m+1}}$ instead of $\underline{A_m}$. This means that G'' processed one more nonterminal symbol on the controlling path, A_{m+1}.

Then, there exists a derivation $S \Rightarrow_G^* xay$ with derivation tree containing a path $A_1A_2 \dots A_mA_{m+1}a$, therefore, the if-part holds.

Claim 2. $GSM_M(L(G'')) = L(\gamma)$.

Indeed, by Claim 1 we know that all words in $L(G'')$ are of the form $xay\diamond^n$, where n is the length of the controlling path, for some $n \geq 2$, $a \in T$, $x, y \in T^*$. The left side, generated by G is left unchanged by M. The right side does not contain any invalid characters, meaning that the whole controlling path, described in G', was processed and is present in a derivation tree corresponding to $S \Rightarrow_G^* xay$. M removes the right side and finishes successfully, producing xay.

Therefore, $GSM_M(L(G'')) = L(\gamma)$. □

Now, let us take a moment to discuss the potential consequences of the gsm mapping regarding the generative power. By Theorem 1, $\mathscr{L}_\omega = \mathbf{CS}$. If we could prove that the gsm mapping does not increase the generative power of the state grammar, we could state that $PC(CF, CF) \subseteq \mathscr{L}_\omega$. And, more importantly, we could find a context-sensitive language that does not have the pumping property from Theorem 2, therefore the inclusion would be proper and $PC(CF, CF) \subseteq \mathbf{CS}$.

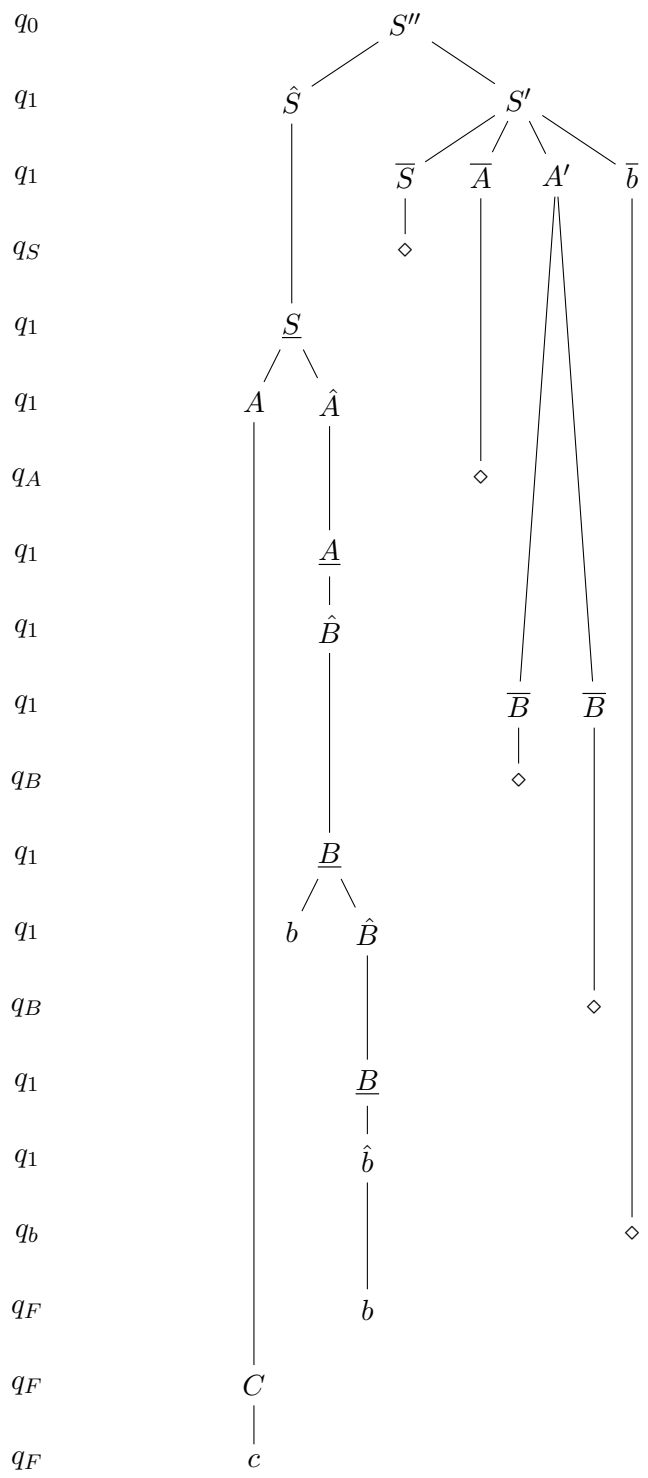


Figure 4.1: Derivation tree of the word $cbb\diamond\diamond\diamond\diamond\diamond$ in G''.

$(q_0, S'') \overset{0}{\Rightarrow} (q_1, \hat{S}S') \overset{11}{\Longrightarrow} (q_1, \hat{S}\overline{SA}A'\bar{b}) \overset{14}{\Longrightarrow} (q_S, \hat{S}\diamond\overline{A}A'\bar{b}) \overset{20}{\Longrightarrow} (q_1, \underline{S}\diamond\overline{A}A'\bar{b}) \overset{2}{\Rightarrow} (q_1, A\hat{A}\diamond$
$\overline{A}A'\bar{b}) \overset{15}{\Longrightarrow} (q_A, A\hat{A}\diamond\diamond A'\bar{b}) \overset{21}{\Longrightarrow} (q_1, A\underline{A}\diamond\diamond A'\bar{b}) \overset{3}{\Rightarrow} (q_1, A\hat{B}\diamond\diamond A'\bar{b}) \overset{13}{\Longrightarrow} (q_1, A\hat{B}\diamond\diamond\overline{BB\bar{b}}) \overset{16}{\Longrightarrow}$
$(q_B, A\hat{B}\diamond\diamond\diamond\overline{B\bar{b}}) \overset{22}{\Longrightarrow} (q_1, A\underline{B}\diamond\diamond\diamond\overline{B\bar{b}}) \overset{6}{\Rightarrow} (q_1, Ab\hat{B}\diamond\diamond\diamond\overline{B\bar{b}}) \overset{16}{\Longrightarrow} (q_B, Ab\hat{B}\diamond\diamond\diamond\diamond\bar{b}) \overset{22}{\Longrightarrow} (q_1, Ab\underline{B}\diamond$
$\diamond\diamond\diamond\bar{b}) \overset{7}{\Rightarrow} (q_1, Ab\hat{b}\diamond\diamond\diamond\diamond\bar{b}) \overset{18}{\Longrightarrow} (q_b, Ab\hat{b}\diamond\diamond\diamond\diamond\diamond) \overset{24}{\Longrightarrow} (q_F, Abb\diamond\diamond\diamond\diamond\diamond) \overset{28}{\Longrightarrow} (q_F, Cbb\diamond\diamond\diamond\diamond\diamond) \overset{32}{\Longrightarrow}$
$(q_F, cbb\diamond\diamond\diamond\diamond\diamond)$

Figure 4.2: Derivation of the word $cbb\diamond\diamond\diamond\diamond\diamond$ in G''.

Chapter 5

Conversion Tool

In this chapter we will discuss a conversion tool which implements the conversion algorithm introduced in Section 4.1. Section 5.1 deals with the tool design, its interface and input and output format. The core of this chapter, Section 5.2, describes important implementation details and class overview. In Section 5.3 we analyze the complexity of the algorithm, both time and space. And finally, Section 5.4 summarizes experimental results.

5.1 Design

The conversion tool is written in C++. It implements the algorithm presented in Section 4.1 almost to the letter. The only difference between the formal definition and the actual implementation is that the gsm mapping is incorporated directly into the set of rules, namely, the type V rules derive an empty string, rather than deriving the $\diamond$ symbol, which is later removed by M. We can do this since the $\diamond$ symbol does not affect any further derivations in G'', as was shown in Section 4.3.

The tool is very simple, it takes two input files, each containing one context-free grammar, parses them into objects representing the grammars and then converts them into a state grammar. The user is responsible for providing correct input grammars, the tool doesn't perform any checks. Both context-free and state grammars can be printed out, including some additional information about them, such as the number of symbols of different types and the degree of nondeterminism.

The degree of nondeterminism, denoted by $deg_n(G)$ is a simple metric that we will use as a rough estimate of how nondeterministic a grammar is. It is calculated as follows:

$$deg_n(G) = \frac{|P|}{|N|},$$

where G is a grammar, P is the set of rules and N is the set of nonterminal symbols.

5.1.1 Input Format

The format of the input files is following: the four components of context-free grammar, V, T, P and S, are enclosed in parentheses, divided by commas. The sets, V, T and P are represented by its elements, divided by commas, enclosed in braces. Rules are in the form `lhs -> rhs`. All white spaces are ignored, so the input file may be formated for better readability.

For example, the context-free grammar G from Section 3.3 may be represented as Figure 5.1 shows.

5.1.2 Output Format

The output format for the state grammar is similar to the input format for context-free grammars. The set of states is added and is placed first, the rules are labeled and the initial state is placed before the start symbol. Since we can't print special symbols directly, we represent them with two characters. Underlined symbols $\underline{A}$ are printed as `A_`, overlined symbols $\overline{A}$ as `A~`, and hatted symbols $\hat{A}$ as `A^`.

Figure 5.1 shows the output format of the state grammar G'' converted from grammars G and G' from Section 3.3 as well as statistics for G and G''.

```
(
{S, B, D, a, b, c, d},
{a, b, c, d},
{
S -> aSd,
S -> aBd,
B -> bBc,
B -> D,
D -> bc
},
S
)
```

```
(
{q_0, q_1, q_F, q_S, ..., q_d}
{S', A', S, S_, S~, S^, B, ... ,d ^},
{a, b, c, d},
{
0:  (q_0, S'') -> (q_1, S^S'),
1:  (q_1, S_) -> (q_1, a^Sd),
                  ⋮
35:  (q_F, D) -> (q_F, bc)
},
q_0, S''
)
```

```
Number of symbols        :  7
Number of terminals      :  4
Number of nonterminals   :  3
Number of rules          :  5
Degree of nondeterminism :  1.7
```

```
Number of states         :  10
Number of symbols        :  30
Number of terminals      :  4
Number of nonterminals   :  26
Number of rules          :  36
Degree of nondeterminism :  1.4
```

Figure 5.1: Input (upper left), output (upper right) and statistics (bottom) format.

5.2 Implementation

In this section we will look at important implementation details of the conversion tool. Class overview and the conversion algorithm itself can be found in Subsections 5.2.1 and 5.2.2, respectively.

5.2.1 Class Overview

Figure 5.2 shows all the classes and their inheritance used in the conversion tool. Now, let's examine each class individually.

Symbol

This class represents all symbols, namely nonterminals, terminals, start symbols and even states, since most of them correspond to some symbol. The symbol consists of two parts, `base` and `type`. `Base` is the symbol itself, for simplicity, we limit it to one character, therefore it is of type `char`. `Type` determines whether symbol is basic, underlined, overlined, primed or double–primed.

These variables are private, corresponding setters and getters are provided as well as `output` function, which prints the symbol on standard output. Binary operators `==` and `!=` are overloaded for comparison of two symbols.

Rule

Rule has three components, `lhs`, left-hand side is a single Symbol, since we deal with context-free grammars. Whereas the right-hand side of a rule, `rhs`, is a vector of Symbols. And finally, a rule is labeled by a unique number.

Again, these variables are manipulated through corresponding set and get functions. Additionally, symbols can be added to the right-hand side one by one. `Output` function is also provided.

State_Rule

Rules in state grammars are extended with states, therefore State_Rule extends the Rule with `q_lhs`, left-hand side state and `q_rhs`, right-hand side state. Corresponding setters and getters as well as modified `output` function is provided.

CS_Grammar

Class for context-free grammar contains, as expected, the total alphabet `V` and set of terminal symbols `T` as vectors of Symbols, vector of Rules `P` and starting nonterminal symbol `S`. In addition, there is a set of nonterminals `N`, also as a vector of Symbols.

Besides usual setters, getters and `output` functions, there are output functions for each element of the grammar as well as function `outputStats`, which prints some additional information about the grammar, such as the number of different types of symbols and the number of rules. A functions `addRule` for adding rules one by one and `makeN`, which computes the set of nonterminals, are available.

State_Grammar

Analogically to State_Rule extending Rule, State_Grammar extends CF_Grammar by adding a set of states `Q` represented by a vector of Symbols and the initial state `q_0` represented by a Symbol. Since state grammars use different type of rules, State_Grammar has different set of rules, `SP` as a vector of State_Rules.

5.2.2 Conversion Algorithm

The conversion algorithm itself is implemented in function `convertPCG2SG()`. It takes three arguments, references to the objects of the generating context-free grammar G, the controlling context-free grammar G' and the state grammar G''.

Figure 5.3 shows the implemented conversion algorithm in pseudocode form. The rule generation (Algorithm 2) is a significant part of the conversion algorithm, therefore, it is implemented by function `generateP3()`, separated from the rest(Algorithm 1).

5.3 Complexity

In this section we discuss the complexity of the conversion algorithm. The complexity of the whole conversion tool doesn't concern us. First, in Subsection 5.3.1, we analyze the time complexity, and second, in Subsection 5.3.2 we discuss the space complexity.

5.3.1 Time Complexity

Since Algorithm 1 uses Algorithm 2, let's start with the analysis of the latter.

- 1: This can be done in constant time, $O(1)$.
- 2–6: The outer for-each cycle body is executed $|P|$ times. The number of executions of the inner for-each cycle body depends on the length of the right-hand side of the rule. However, there must exist a constant c, such that c is the average length of a right-hand side of a rule, since the right-hand side of a rule is finite. Then the inner for-each cycle body is executed c times and the time complexity of lines $2 - 6$ is in $O(|P|)$.
- 7–9: This can be done in $O(|P'|)$.
- 10–12: This can be done in $O(|V|)$.
- 13–15: This can be done in $O(|N|)$.
- 16–18: This can be done in $O(|T|)$.
- 19–21: This can be done in $O(|P|)$.

Since $V = T \cup N$, the time complexity of Algorithm 2 lies in $O(|V| + |P| + |P'|)$. Now let's analyze Algorithm 1.

- 1: This can be done in $O(1)$.
- 2–4: This can be done in $O(|V|)$.
- 5: This can be done in $O(N')$.
- 6–8: This can be done in $O(|V|)$.
- 9: This can be done in $O(|T|)$.
- 10–16: The outer for-each cycle body is executed $|V''|$ times. The inner for-each cycle body is executed at most $|T''|$ times and since $T \subseteq V$, the inner for-each cycle body is executed at most $|V''|$ times. Then, the lines 10–16 can be done in $O(|V''|^2)$.
- 17: As we determined earlier, this can be done in $O(|V| + |P| + |P'|)$.
- 18–19: This can be done in $O(1)$.

We know that V'' consists of symbols from N' and symbols generated from V, then $O(|V''|^2) = O((|V| + |N'|)^2)$, therefore the time complexity of Algorithm 1 lies in $O((|V| + |N'|)^2 + |P| + |P'|)$. Note, that the quadratic part can be easily lowered by incorporating the building of the set of nonterminals into previous steps or by leaving it to the user altogether. Then the time complexity can be reduced to $O(|V| + |N'| + |P| + |P'|)$.

5.3.2 Space Complexity

In the same fashion we analyzed the time complexity, we now analyze the space complexity.

- 1: This generate one rule, $O(1)$.
- 2–6: Same logic as with the time complexity analysis applies here, given the average length of a right-hand side of a rule, c, the number of rules generated is $c \cdot |P|$ and the space complexity of lines 2–6 lies is in $O(|P|)$.
- 7–9: The number of rules generated is in $O(|P'|)$.
- 10–12: The number of rules generated is in $O(|V|)$.
- 13–15: The number of rules generated is in $O(|N|)$.
- 16–18: The number of rules generated is in $O(|T|)$.
- 19–21: The number of rules generated is in $O(|P|)$.

Since $V = T \cup N$, the number of rules generated is $2 \cdot |V| + (c+1) \cdot |P| + |P'| + 1$ and the space complexity of Algorithm 2 lies in $O(|V| + |P| + |P'|)$. For Algorithm 1:

- 1: This generates three states, $O(1)$.
- 2–4: This generates $|V|$ states, $O(|V|)$.
- 5: This generates $|N'|$ symbols, $O(|N'|)$.
- 6–8: This generates $4 \times |V|$ symbols, $O(|V|)$.
- 9: This generates $|T|$ symbols, $O(|T|)$.
- 10–16: In the worst case scenario, when $V'' = T''$, this generates $|V''|$ symbols, $O(|V''|)$.
- 17: We determined that the number of rules generated is in $O(|V| + |P| + |P'|)$.
- 18: This generates one state, $O(1)$.
- 19: This generates one symbol, $O(1)$.

We can see that V'' contains $4 \times |V| + |N'|$ symbols and using the same logic as with the time complexity, the space complexity of Algorithm 1 lies in $O(|V| + |N'| + |P| + |P'|)$.

5.4 Experiments

Using the conversion tool we converted a number of path-controlled grammars into state grammars. All these grammars are listed in Appendix C and some remarks regarding languages they generate are in 6.4. Table 5.1 shows the statistics of path-controlled grammars. The generating grammar is placed first, the controlling grammar second. Table 5.2 shows the statistics of the resulting state grammars. For example, the path-controlled grammar $\gamma_1 = (G_1, G_2)$ is converted into the state grammar γ_{1c}.

Presented numbers confirm our space complexity analysis from Subsection 5.3.2.

PCG	CFG	$\|V\|$	$\|T\|$	$\|N\|$	$\|P\|$	deg_n
γ_1	G_1	6	2	4	7	1.8
	G_2	8	6	2	3	1.5
γ_2	G_3	7	4	3	5	1.7
	G_4	9	7	2	4	2.0
γ_3	G_5	9	3	6	16	2.7
	G_6	11	9	2	4	2.0
γ_4	G_7	7	3	4	7	1.8
	G_8	9	7	2	3	1.5
γ_5	G_9	11	6	5	6	1.2
	G_{10}	13	11	2	3	1.5
γ_6	G_{11}	9	3	6	16	2.7
	G_{12}	12	9	3	5	1.7
γ_7	G_{13}	5	2	3	6	2.0
	G_{14}	6	5	1	1	1.0

Table 5.1: Path-controlled grammars' statistics.

SG	$\|Q\|$	$\|V\|$	$\|T\|$	$\|N\|$	$\|P\|$	deg_n
γ_{1c}	9	26	2	24	33	1.4
γ_{2c}	10	30	4	26	36	1.4
γ_{3c}	12	38	3	35	67	1.9
γ_{4c}	10	30	3	27	37	1.4
γ_{5c}	14	46	6	40	43	1.1
γ_{6c}	12	39	3	36	68	1.9
γ_{7c}	8	21	2	19	33	1.7

Table 5.2: Converted path-controlled grammars' statistics.

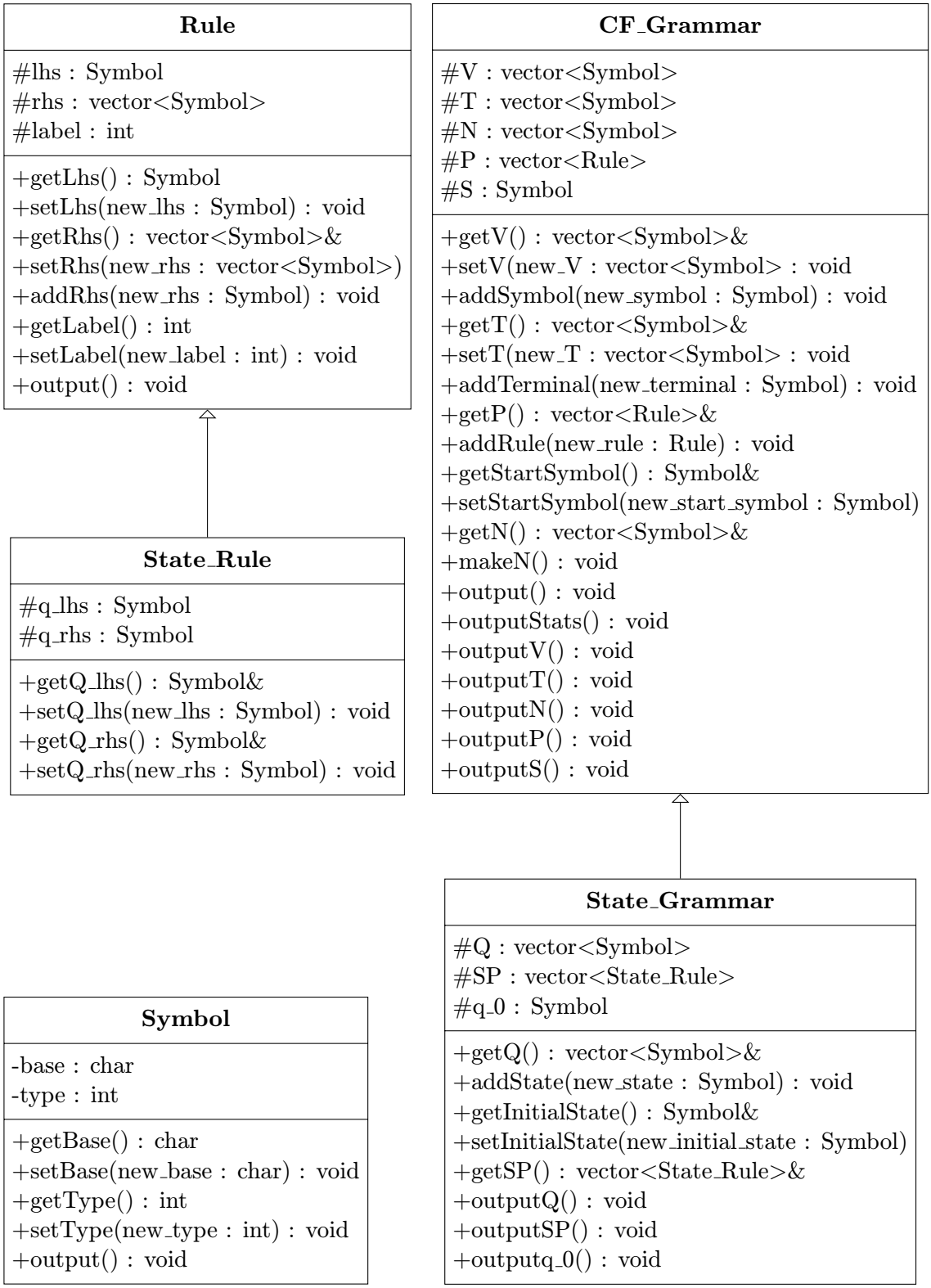


Figure 5.2: Classes used in the conversion tool.

Algorithm 1: convertPCG2SG()

Input : Context-free grammars $G = (V, T, P, S)$ and $G' = (V', V, P', S')$
Output : State grammar $G'' = (Q, V'', T'', P'', q_0, S'')$
Method:

`/* generate the set of states` Q `*/`
$Q \leftarrow q_0, q_1, q_F$
foreach X **in** V **do**
  $Q \leftarrow q_X$
end

`/* generate the total alphabet` V'' `*/`
$V'' \leftarrow N'$
foreach A **in** V **do**
  $V'' \leftarrow A, \underline{A}, \overline{A}, \hat{A}$
end

`/* generate the set of terminals` T'' `*/`
$T'' \leftarrow T$

`/* generate the set of nonterminals` N'' `*/`
foreach X **in** V'' **do**
  foreach T **in** T'' **do**
    if $X! = T$ **then**
      $N'' \leftarrow X$
    end
  end
end

`/* generate the set of rules` P'' `*/`
$P'' \leftarrow generateP''(G, G')$

`/* set the initial state and start symbol */`
$q_0 \leftarrow q_0$
$S'' \leftarrow S''$

return G''

Algorithm 2: generateP3()

Input : Context-free grammars $G = (V, T, P, S)$ and $G' = (V', V, P', S')$
Output : Set of rules P''
Method:

`/* Type I rule */`
$P'' \leftarrow ((q_0, S'') \rightarrow (q_1, \hat{S}\overline{S}))$

`/* Type II rules */`
foreach $(A \rightarrow x)$ **in** P **do**
  foreach uXv **in** x **do**
    $P'' \leftarrow ((q_1, \underline{A}) \rightarrow (q_1, u\hat{X}v))$
  end
end

`/* Type III rules */`
foreach $(A \rightarrow x)$ **in** P' **do**
  $P'' \leftarrow ((q_1, A) \rightarrow (q_1, h(x)))$
end

`/* Type IV rules */`
foreach X **in** V **do**
  $P'' \leftarrow ((q_1, \overline{X}) \rightarrow (q_X, \varepsilon))$
end

`/* Type V rules */`
foreach A **in** N **do**
  $P'' \leftarrow ((q_A, \hat{A}) \rightarrow (q_1, \underline{A}))$
end

`/* Type VI rules */`
foreach a **in** T **do**
  $P'' \leftarrow ((q_a, \hat{a}) \rightarrow (g_F, a))$
end

`/* Type VII rules */`
foreach $(A \rightarrow x)$ **in** P **do**
  $P'' \leftarrow ((q_F, A) \rightarrow (q_F, x))$
end

return P''

Figure 5.3: Path-controlled grammar to state grammar conversion algorithm.

Chapter 6

Parser

This chapter is dedicated to a parsing tool for path-controlled grammars. Section 6.1 covers general design of the tool, in Section 6.2 we examine important implementation details. Complexity analysis of the tool is in Section 6.3 and finally, in Section 6.4 we present experimental results.

6.1 Design

This tool is designed as a general parser for path-controlled grammars. It does not work directly with the path-controlled grammar, it uses an intermediate state grammar generated by using the conversion algorithm presented in Section 4.1. Therefore, it is only natural that the parser is an extension of the conversion tool presented in Chapter 5.

The parser performs a *depth-first search* over a tree of all possible configuration of a state grammar. Figure 6.1 shows part of such tree for a state grammar converted from the path-controlled grammar introduced in Section 4.2. The root node represents the initial configuration (q_0, S''), where q_0 is the initial state and S'' is the start symbol. Two configurations c_1 and c_2 are connected by a forward edge if there is a rule p such that $c_1 \Rightarrow c_2\,[p]$, the edge is directed from c_1 to c_2 and is labelled by p.

A backward edge from c_2 to c_1 indicates that there is a derivation $c_2 \Rightarrow^* (q_F, w)$, where $w \in T^*$ is the sentence we are parsing. In that case the edge is labelled by *True*. We assume that the generating grammar is ε-free and does not contain any chain rules. Then we can state that if the left side of a sentential form in a configuration c, that is the part generated by the generating grammar, is longer than w, there cannot be a derivation $c \Rightarrow^* w$ and the backward edge from c is labelled *False*. Clearly, if there is an edge labelled *True* coming to a root node, $w \in L(\gamma)$.

6.2 Implementation

As was stated, the parses is an extension of the conversion tool. Only one new class is introduced. **Configuration**, as Figure 6.2 shows, contains three private variables. `State` is the actual state of a state grammar, represented by a single Symbol, `s_form` is the sentential form, represented by a vector of Symbols and `l_parse` is the left parse, represented by a vector of rule labels (integers).

Besides usual get and output functions, there are two functions worth mentioning. The function `getUsableRules` finds the leftmost nonterminal symbol in the current sentential

form that can be rewritten and generates a set of rules, a set of rule labels to be precise, which can rewrite this symbol.

The function `applyRule`, as the name suggests, takes the current configuration, copies it, applies the specified rule and returns the new configuration. The previous configuration is left unchanged for eventual backtracking.

Figure 6.3 shows the parsing algorithm in pseudocode from. It is divided into two parts. Function `derivation()` (Algorithm 3) initializes the process by creating the initial configuration and by calling `derive()` (Algorithm 4) with rule 0 as a parameter. Rule 0 is the only rule that can be applied on initial configuration. `Derive()` performs the actual depth-first search using recursive descent.

6.3 Complexity

We will analyze both time and space complexity of the parsing algorithm, the complexity of the conversion algorithm for the intermediate state grammar was discussed in Section 5.3, we don't need to repeat it. Complexity of auxiliary functions, such as handling input and output, does not concern us. Time complexity is analyzed in Subsection 6.3.1, space complexity in Subsection 6.3.2.

6.3.1 Time Complexity

Algorithm 3 only initializes the search, it can be done in constant time. Let us analyze Algorithm 4.

- 1: A rule p can be applied in $|s\_form| + |rhs(p)|$. Assuming that the controlling grammar does not contain any ε-productions nor unit-productions we can say that the complexity of line 1 lies in $O(|w| + c_2) = O(|w|)$, where c_2 is the average length of a right-hand side of a rule in P''.
- 2–4: This can be done in $O(|w|)$.
- 5–7: This can be done in $O(|w|)$.
- 8: In the worst case scenario, every rule is checked against every symbol in the sentential form, $|P''| \cdot |s\_form|$, therefore the complexity of line 8 lies in $O(|P''| \cdot |w|)$.
- 9–13: In the worst case scenario, the algorithm checks every configuration in the configuration tree. We need to determine how large this tree is. Looking at the Figure 6.1 can give us the idea. The degree of branching corresponds to the degree of nondeterminism of the state grammar and the depth depends on the length of parsed sentence. An estimate of the size of the configuration tree is $deg_n(G'')^{|w|/c_1}$, where c_1 is the average length of a right-hand side of a rule in P.

  The for-each cycle body, not taking into account the complexity of the function `derive()`, can be done in constant time.

We can see that the time complexity of Algorithm 4 lies in $O(|P''|\cdot|w|)\cdot O(deg_n(G'')^{|w|}) = O(deg_n(G'')^{|w|})$.

6.3.2 Space Complexity

The algorithm stores every configuration along the path it currently investigates. If that path doesn't lead to the final configuration, it backtracks while discarding all configurations that don't lead to the final configuration. Therefore, the algorithm never uses more space than the longest path in the configuration tree, that is the longest derivation, and since the length of such derivation depends on the length of the parsed sentence, the space complexity lies in $O(|w|)$.

6.4 Experiments

We performed a series of tests with different path-controlled grammars and length of the input sentence. All grammars are listed in Appendix C, additional information about them can be found in Section 5.4.

For every path-controlled grammar we experimentally found a suitable interval of input sentence lengths. We performed ten measurements, evenly spread on this interval. For each measurement the parser ran three times with input word that did not belong to the language generated by the path-controlled grammar and the results were averaged. This way we measured the worst case scenario when all possible configurations are checked. Three runs per sentence length was enough as we so consistent result throughout the measurements.

Figure 6.4 shows the average time of the worst case parsing for all the grammars and all sentence lengths. Note that the x axis is in logarithmic scale for better visibility. Table 6.2 contains the measured results.

One can easily see that the time needed for traversing through the whole configuration tree grows exponentially with the length of input sentence as was predicted in Subsection 6.3.1, where we stated that the time complexity lies in $O(deg_n(G'')^{|w|})$. The degree of nondeterminism of the intermediate state grammar corresponds to the degree of branching of the configuration tree. However, the calculated values (see Table 5.2) don't reflect this very well. The problem is that the intermediate state grammar can contain useless rules which skews the degree of nondeterminism.

Let us propose a modification of the metric, $deg_{n2}(G'') = deg_n(G) \cdot deg_n(G')$. This way, the metric is not affected by useless rules in P'', but still remains simple. Table 6.1 shows values of this new metric for the path-controlled grammars.

PCG	$deg_n(G)$	$deg_n(G')$	$deg_{n2}(G'')$
γ_1	1.8	1.5	2.7
γ_2	1.7	2.0	3.4
γ_3	2.7	2.0	5.4
γ_4	1.8	1.5	2.7
γ_5	1.2	1.5	1.8
γ_6	2.7	1.7	4.6
γ_7	2.0	1.0	2.0

Table 6.1: Modified degree of nondeterminism of state-grammars.

We can see that this metric reflects the reality much better.

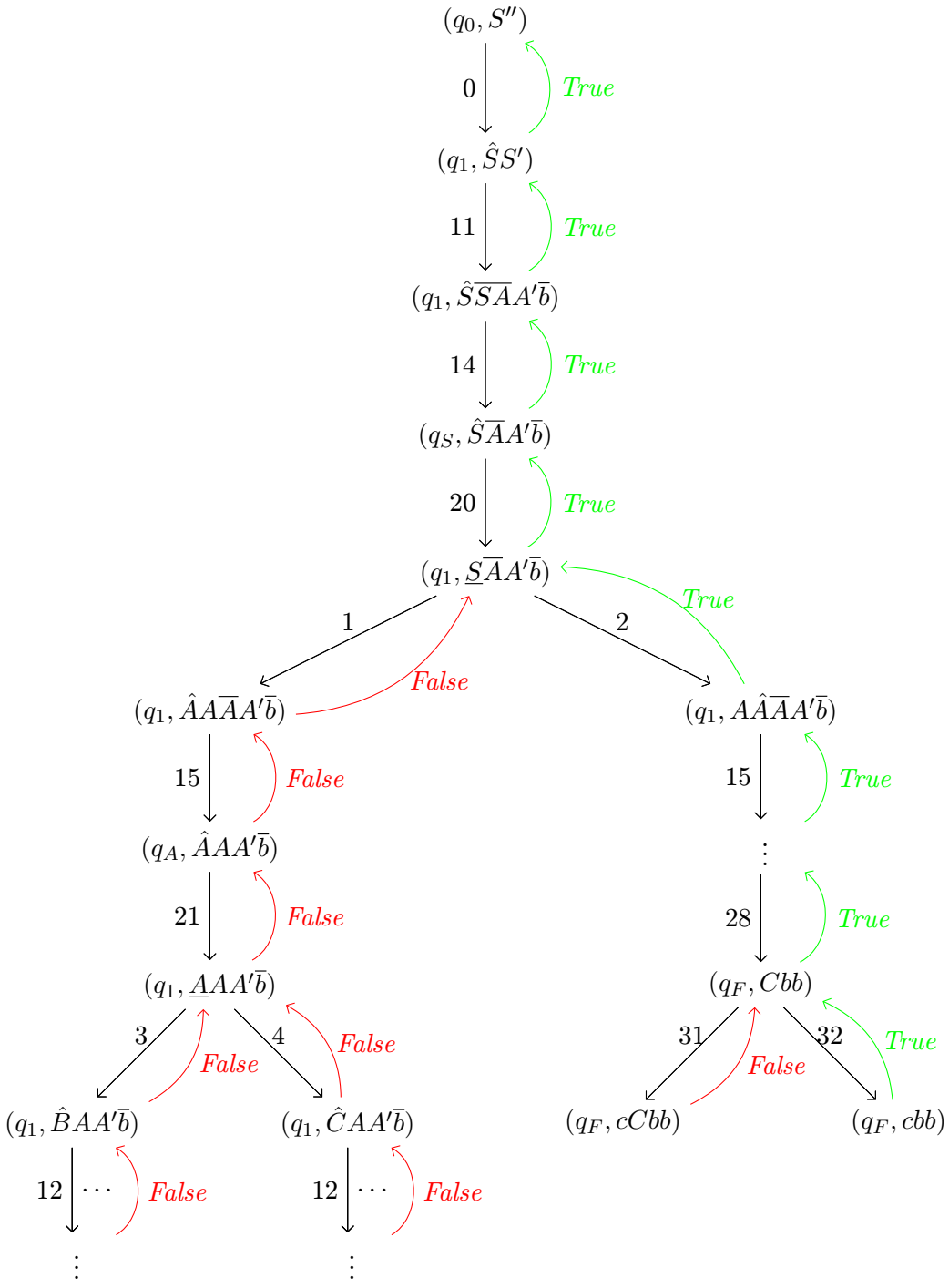


Figure 6.1: Configuration tree for sentence $cbb \in L(\gamma)$

Configuration
-state : Symbol -s_form : vector<Symbol> -l_parse : vector<int>
+getS_form() : vector<Symbol>& +getL_parse() : vector<int>& +applyRule(G : State_Grammar&, rule_label : int) : Configuration +copyConfiguration(conf : Configuration*) : void +getUsableRules(G : State_Grammar&) : vector<int> +output() : void +outputConfiguration(): void +outputL_parse() : void

Figure 6.2: Configuration class.

Algorithm 3: derivation()

Input : State grammar $G'' = (V'', T, P'', S'')$, Context-free grammar $G = (N, T, P, S)$, Sentence w

Output : **True** if $w \in L(G'')$, **False** otherwise

Method:

```
1  conf ← initial configuration
2  if derive(G'', G, w, conf, 0) then
3  |  return True
4  else
5  |  return False
6  end
```

Algorithm 4: derive()

Input : State grammar $G'' = (V'', T, P'', S'')$, Context-free grammar $G = (N, T, P, S)$, Sentence w, Configuration *conf*, Rule *rule*

Output : **True** if w is derived from *conf* by applying *rule*, **False** otherwise

Method:

```
1  new_conf ← applyRule(G'', conf, rule)
2  if sentential form == w then
3  |  return True
4  end
5  if sentential form is too long then
6  |  return False
7  end
8  usableRules ← getUsableRules(G'')
9  foreach rule in usableRules do
10 |  if derive(G'', G, w, new_conf, rule) then
11 |  |  return True
12 |  end
13 end
14 return False
```

Figure 6.3: State grammar parsing algorithm.

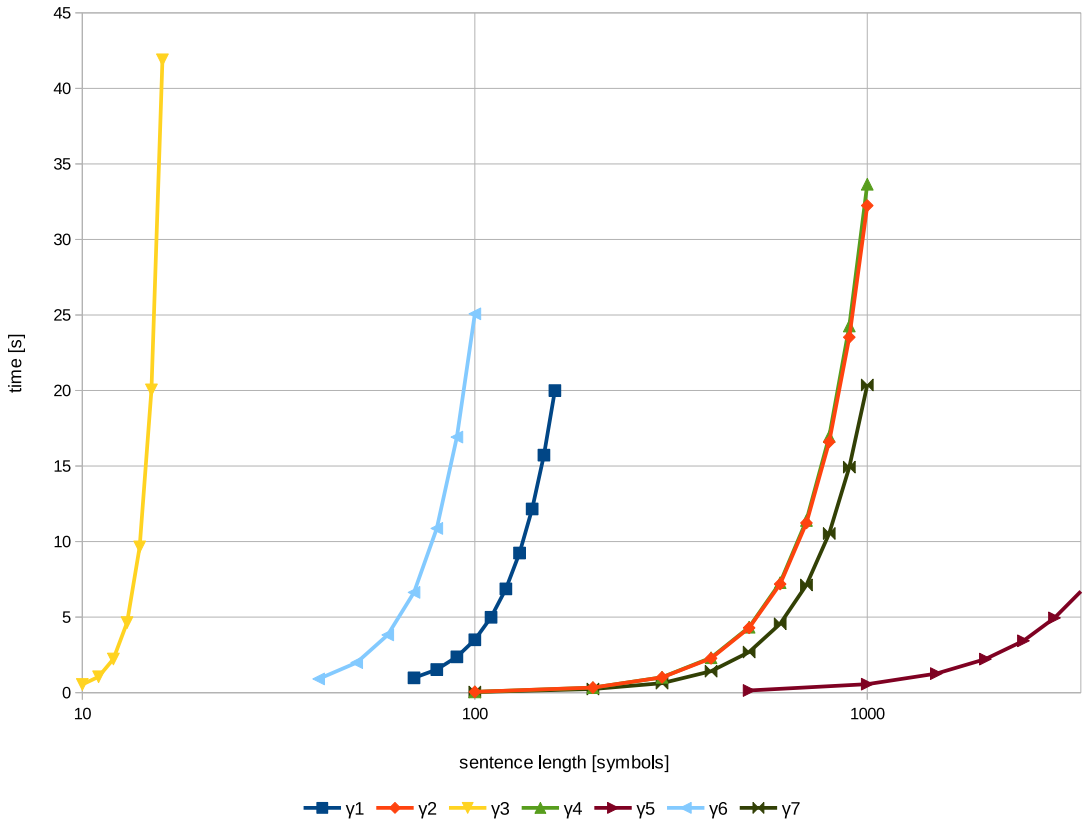


Figure 6.4: Worst case parse time depending on the length of a sentence.

$\|w\|$	γ_{1c}	γ_{2c}	γ_{3c}	γ_{4c}	γ_{5c}	γ_{6c}	γ_{7c}
10			0.54				
11			1.05				
12			2.227				
13			4.637				
14			9.647				
15			20.04				
16			41.89				
40						0.91	
50						1.99	
60						3.823	
70	0.98					6.637	
80	1.53					10.87	
90	2.37					16.917	
100	3.5	0.053		0.05		25.077	0.042
110	4.99						
120	6.867						
130	9.243						
140	12.16						
150	15.723						
160	19.993						
200		0.337		0.33			0.24
300		1.01		1.017			0.627
400		2.27		2.31			1.42
500		4.29		4.333	0.143		2.69
600		7.187		7.283			4.56
700		11.233		11.4			7.127
800		16.603		16.953			10.537
900		32.53		24.273			14.927
1000		32.243		33.657	0.563		20.367
1500					1.263		
2000					2.22		
2500					3.433		
3000					4.95		
3500					6.7		

Table 6.2: Worst case parse time depending on the length of a sentence.

Chapter 7

Conclusion

We introduced the path-controlled grammars, context-free grammars that place restrictions on the paths in derivation trees in order to increase the generative power. The question of generative power is still open since the original claim that $PC(CF, CF) \subset \mathbf{MAT}$, in [2], was supported by a construction that was later proven incorrect in [3].

In this thesis we explored the relation between path-controlled grammars and state grammars. State Grammars are another type of regulated grammars, they extend context-free grammars with the notion of states, restricting which rules can be applied in a derivation. It is known that $\mathscr{L}_\omega = \mathbf{CS}$.

We proposed a conversion algorithm from path-controlled grammars into state grammars. The proof of correctness was provided. Unfortunately, we were not able to prove that $PC(CF, CF) \subseteq \mathscr{L}_\omega$ since a gsm mapping was used. However, our results confirm conclusions regarding the generative power of path-controlled grammars made in [3].

The conversion algorithm was implemented and tested on a set of path-controlled grammars. The complexity of this algorithm was theoretically analysed and then confirmed by measurements on said set of grammars.

A tool for parsing of path-controlled grammars was implemented. This tool uses the conversion algorithm to create an intermediate state grammar and performs a depth-first search on a configuration tree of the state grammar. Time complexity was carefully analyzed and confirmed by measurements.

The question of the generative power of path-controlled grammar remains open, despite our efforts. Further research is needed.

The parsing tool has exponential time complexity and even though it performs well for some grammars, for others it is virtually unusable. The blind depth-first search could be optimized by using LL parsing methods (see [8]) for the generating grammar, reducing the time complexity.

Bibliography

[1] J. Dassow and G. Păun, *Regulated Rewriting in Formal Language Theory.* Springer-Verlag New York, Inc., 1989.

[2] S. Marcus, C. Martín-Vide, V. Mitrana, and G. Păun, "A new–old class of linguistically motivated regulated grammars," *Language and Computers*, vol. 37, no. 1, pp. 111–125, 2001.

[3] J. Koutný, *Grammars with Restricted Derivation Trees.* PhD thesis, Faculty of Information Technology, Brno University of Technology, Brno, 2012.

[4] T. Kasai, "An hierarchy between context-free and context-sensitive languages," *Journal of Computer and System Sciences*, vol. 4, no. 5, pp. 492–508, 1970.

[5] A. Meduna, *Automata and languages: theory and applications.* London: Springer Science & Business Media, 2000.

[6] S. Abraham, "Some questions of phrase-structure grammars," *Linguistics*, vol. 4, pp. 61–70, 1965.

[7] A. Salomaa, "Matrix grammars with a leftmost restriction," *Information and Control*, vol. 20, no. 2, pp. 143–149, 1972.

[8] D. J. Rosenkrantz and R. E. Stearns, "Properties of deterministic top down grammars," in *Proceedings of the first annual ACM symposium on Theory of computing*, pp. 165–180, ACM, 1969.

Appendix A

Contents of the CD

The CD contains following directories:

`/doc/` – electronic version of this text,

`/tests/` – testing script,

`/tests/grammars/` – set of path-controlled grammars,

`/tests/inputs/` – set of testing inputs,

`/tex/` – source code of this text,

`/tool/` – conversion and parsing tool

`/tool/src/` – source code of the conversion and parsing tool

Appendix B

Tool usage

```
./parser -c|p generating_grammar controlling_grammar [sentence]
    c                   :  convert mode
    p                   :  parse mode
    generating_grammar  :  file name of the generating grammar
    controlling_grammar :  file name of the controlling grammar
    sentence            :  in parse mode (-p), input sentence
                           has to be specified
```

Appendix C

Grammars

$\gamma_1 = (G_1, G_2)$ with $G_1 = (V_1, T_1, P_1, S)$, where

$$V_1 = \{S, A, B, C, b, c\},$$
$$T_1 = \{b, c\},$$
$$P_1 = \{S \to AA, A \to B, A \to C, B \to bB, B \to b, C \to cC, C \to c\},$$

and $G_2 = (V_2, T_2, P_2, S')$, where

$$V_2 = \{S', A', S, A, B, C, b, c\},$$
$$T_2 = \{S, A, B, C, b, c\},$$
$$P_2 = \{S' \to SAA'b, A' \to BA'B, A' \to BB\}.$$

G_1 and G_2 are stored in `G1.txt` and `G2.txt`, respectively.

$\gamma_2 = (G_3, G_4)$ with $G_3 = (V_3, T_3, P_3, S)$, where

$$V_3 = \{S, B, D, a, b, c, d\},$$
$$T_3 = \{a, b, c, d\},$$
$$P_3 = \{S \to aSd, S \to aBd, B \to bBc, B \to D, D \to bc\},$$

and $G_4 = (V_4, T_4, P_4, S')$, where

$$V_4 = \{S', A', S, B, D, a, b, c, d\},$$
$$T_4 = \{S, B, D, a, b, c, d\},$$
$$P_4 = \{S' \to SA'BDb, S' \to SBDb, A' \to SA'B, A' \to SB\}.$$

$L(\gamma_2) = \{a^n b^n c^n d^n \mid n \geq 1\}$

G_3 and G_4 are stored in `G3.txt` and `G4.txt`, respectively.

$\gamma_3 = (G_5, G_6)$ with $G_5 = (V_5, T_5, P_5, S)$, where

$$V_5 = \{S, A, B, C, D, E, a, b, c\},$$
$$T_5 = \{a, b, c\},$$

$$P_5 = \{S \to A, S \to B, A \to aA, A \to aB, A \to aE, B \to bB, B \to bA, B \to bE,$$
$$C \to Ca, C \to Da, C \to ca, D \to Db, D \to Cb, D \to cb, E \to D, E \to C\},$$

and $G_6 = (V_6, T_6, P_6, S')$, where

$$V_6 = \{S', A', S, A, B, C, D, E, a, b, c\},$$
$$T_6 = \{S, A, B, C, D, E, a, b, c\},$$
$$P_6 = \{S' \to SA'c, A' \to AA'C, A' \to BA'D, A' \to E\}.$$

$L(\gamma_3) = \{wcw \mid w \in T^*\}$

G_5 and G_6 are stored in `G5.txt` and `G6.txt`, respectively.

$\gamma_4 = (G_7, G_8)$ with $G_7 = (V_7, T_7, P_7, A$, where

$$V_7 = \{A, B, D, X, a, b, d\},$$
$$T_7 = \{a, b, d\},$$
$$P_7 = \{A \to bB, A \to dD, A \to X, B \to aA, D \to aA, X \to aX, X \to a\},$$

and $G_8 = (V_8, T_8, P_8, S'$, where

$$V_8 = \{S', Y', A, B, D, X, a, b, d\},$$
$$T_8 = \{A, B, D, X, a, b, d\},$$
$$P_8 = \{S' \to ABADAY'a, Y' \to XY', Y' \to X\}.$$

$L(\gamma_4) = \{bada+\}$

G_7 and G_8 are stored in `G7.txt` and `G8.txt`, respectively.

$\gamma_5 = (G_9, G_{10})$ with $G_9 = (V_9, T_9, P_9, S)$, where

$$V_9 = \{S, A, B, C, D, a, b, c, d, e, f\},$$
$$T_9 = \{a, b, c, d, e, f\},$$
$$P_9 = \{S \to aA, A \to bB, B \to cC, C \to dD, D \to eS, D \to f\},$$

and $G_{10} = (V_{10}, T_{10}, P_{10}, S')$, where

$$V_{10} = \{S', A', S, A, B, C, D, a, b, c, d, e, f\},$$
$$T_{10} = \{S, A, B, C, D, a, b, c, d, e, f\},$$
$$P_{10} = \{S' \to SABCDA'f, A' \to SABCDA', A' \to SABCD\}.$$

G_9 and G_{10} are stored in `G9.txt` and `G10.txt`, respectively.

$\gamma_6 = (G_{11}, G_{12})$ with $G_{11} = (V_{11}, T_{11}, P_{11}, S)$, where

$$V_{11} = \{S, A, B, C, D, E, a, b, c\},$$
$$T_{11} = \{a, b, c\}$$
$$P_{11} = \{S \to A, S \to B, A \to aA, A \to aB, A \to aE, B \to bB, B \to bA, B \to bE,$$

$$C \to Ca, C \to Da, C \to ca, D \to Db, D \to Cb, D \to cb, E \to D, E \to C\},$$

and $G_{12} = (V_{12}, T_{12}, P_{12}, S')$, where

$$V_{12} = \{S', A', B', S, A, B, C, D, E, a, b, c, \},$$
$$T_{12} = \{S, A, B, C, D, E, a, b, c\},$$
$$P_{12} = \{S' \to SA'c, A' \to AA'C, A' \to BB'D, B' \to BB'D, B' \to E\}.$$

G_{11} and G_{12} are stored in `G11.txt` and `G12.txt`, respectively.

$\gamma_7 = (G_{13}, G_{14})$ with $G_{13} = (V_{13}, T_{13}, P_{13}, S)$, where

$$V_{13} = \{S, A, B, a, b\},$$
$$T_{13} = \{a, b\}$$
$$P_{13} = \{S \to BAA, S \to ABA, S \to AAB, A \to aAa, A \to aa, B \to b\},$$

and $G_{14} = (V_{14}, T_{14}, P_{14}, S')$, where

$$V_{14} = \{S', S, A, B, a, b\},$$
$$T_{14} = \{S, A, B, a, b\},$$
$$P_{14} = \{S' \to SBb\}.$$

G_{13} and G_{14} are stored in `G13.txt` and `G14.txt`, respectively.