Regulated Rewriting in Formal Language Theory

by: Mohamed A.M.S Taha

Thesis presented in partial fulfilment of the requirements
for the degree of Master of Science in Computer Science
at the University of Stellenbosch

Supervisor: Prof. A.B. van der Merwe

March, 2008

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.

Signature: ..... Date: .....

Abstract

Context-free grammars are well-studied and well-behaved in terms of decidability, but many real-world problems cannot be described with context-free grammars. Grammars with regulated rewriting are grammars with mechanisms to regulate the applications of rules, so that certain derivations are avoided. Thus, with context-free rules and regulated rewriting mechanisms, one can often generate languages that are not context-free.

In this thesis we study grammars with regulated rewriting mechanisms. We consider problems in which context-free grammars are insufficient and in which more descriptive grammars are required. We compare bag context grammars with other well-known classes of grammars with regulated rewriting mechanisms. We also discuss the relation between bag context grammars and recognizing devices such as counter automata and Petri net automata. We show that regular bag context grammars can generate any recursively enumerable language. We reformulate the pumping lemma for random permitting context languages with context-free rules, as introduced by Ewert and Van der Walt, by using the concept of a string homomorphism. We conclude the thesis with decidability and complexity properties of grammars with regulated rewriting.

Opsomming

Konteksvrye grammatikas is al deeglik bestudeer. Konteksvrye grammatikas het ook goeie beslisbaarheid eienskappe, maar baie werklike probleme kan dikwels nie met konteksvrye grammatikas beskryf word nie. Grammatikas met reguleerbare herskrywing is grammatikas met meganismes om die toepassing van reëls te beheer, sodat sommige afleidings vermy word. Dit is dus dikwels moontlik om met reguleerbare herskrywing en konteksvrye reëls tale te genereer wat nie konteksvry is nie.

In hierdie tesis bestudeer ons reguleerbare herskrywingsmeganismes. Ons beskou probleme waar konteksvrye grammatikas onvoldoende is en waar meer beskrywende grammatikas dus benodig word. Ons vergelyk multi-versameling grammatikas met ander bekende grammatikas met reguleerbare herskrywings meganismes. Ons wys dat regulêre multi-versameling grammatikas enige rekursiewe enumereerbare taal kan genereer. Ons herformuleer die pomp lemma vir willekeurige toelaatbare konteks tale met konteksvrye reëls, soos beskryf in [19], deur van die konsep van 'n string homomorfisme gebruik te maak. Ons sluit die tesis af met die beslisbaarheids en komplesiteitseienskappe van grammatikas met reguleerbare herskrywing.

Dedication

To my precious mother (Alawyia)
To the memory of my father
To my lovely family
I dedicate my work

Acknowledgements

First of all, I would like to express my sincere gratitude and appreciation to ALLAH, the one who kept me with his care and support all the time and facilitated the difficulties along my way.

I also would like to thank my supervisor Prof. Brink van der Merwe for introducing me to this field and teaching me how to conduct quality research.

A special thanks to the African Institute for Mathematical Sciences (AIMS), and Faculty of Science at the University of Stellenbosch for sponsoring my study.

All love and care to my mother, my father, and all my family for their love and support.

Contents

List of figures

Chapter 1

Introduction

The study of formal language theory started in the middle of the 20^{th} century, motivated in part by natural languages. The development of programming languages also contributed strongly to the development of this discipline. Many other branches of science, such as developmental biology and logic, also contributed to the development of formal language theory.

In the Chomsky hierarchy (see Section 3.2), the context-free grammars (including regular grammars) (see Definitions 3.4 and 3.7) are the best developed and well-studied class of grammars. This is due to their applications in programming languages. Context-free languages have many good properties in terms of decidability. The membership, emptiness, and finiteness problems (see Definition 6.2) are all decidable for the class of context-free languages. Unfortunately, many real-world problems can only be described by grammars that are not context-free.

Grammars with regulated rewriting (see Section 3.3) were introduced in 1965. These grammars use rules similar to the grammars in the Chomsky hierarchy, but with additional restrictions on when rules can be applied.

This thesis is a survey of regulated rewriting. We compare bag context grammars (see Definition 3.20), which are relatively new, with counter automata (blind and partially blind) (see Definitions 4.1 and 4.2) and Petri net automata (see Definition 4.3). In each case we show how bag context grammars can simulate these recognizing devices such that they describe the same class of languages. We show that regular bag context grammars generate the recursively enumerable languages (see Definition 3.2). A well-known way to show that a language is not context-free is by applying the pumping lemma for context-free languages (see Theorem

3.1). A pumping lemma for random permitting context languages with context-free rules (see Definition 3.17) was introduced in [19]. We state and prove this lemma in terms of string homomorphisms (see Definition 3.1). Decidability and complexity properties of grammars with regulated rewriting are also considered. We consider various decidability properties and explain how Petri nets, vector addition systems (see Definition 6.4) and Groebner bases can help in determining decidability. Finally we compare the complexity of regulated rewriting mechanisms in terms of nonterminal complexity.

We present in Chapter 2 circumstances in which context-free languages are not sufficient. We show how grammars with regulated rewriting and ET0L systems (see Definition 3.22) can generate some of these languages. Then we briefly consider a generalization of string languages, namely tree languages.

In Chapter 3 we introduce various generating mechanisms. We consider the Chomsky hierarchy, grammars with regulated rewriting, and branching synchronization grammars.

Chapter 4 considers the relationship between recognizing devices and generating mechanisms. We show how to simulate counter automata (blind and partially blind) and Petri net automata with bag context grammars. We also show that bag context grammars can generate any recursively enumerable language.

We state and prove in Chapter 5 the pumping lemma for random permitting context languages with context-free rules in terms of string homomorphisms. We also state without proof the shrinking lemma for random forbidding context languages with context-free rules (see Definition 3.17).

We discuss in Chapter 6 decidability and complexity properties of grammars with regulated rewriting mechanisms. We consider the membership, emptiness, finiteness and equivalence problems. We show how one can use Petri nets, vector addition systems and Groebner bases to answer certain decidability problems for grammars with regulated rewriting. We use the decidability of reachability in Petri nets (see Definition 6.3) to show the decidability of emptiness and membership in random permitting context grammars with context-free rules. We conclude the chapter with complexity of regulated rewriting. We compare various classes of grammars with regulated rewriting by considering nonterminals complexity.

Finally, we give a conclusion and present avenues for future consideration.

Chapter 2

Beyond context-free grammars

In investigating aspects of formal language theory in Computer Science, the motivation normally starts by referring to the Chomsky hierarchy of languages. We will not deviate from this practice. In this hierarchy, the regular and context-free languages are well-studied and well understood. This is mainly due to the use of regular expressions and the extended Backus-Naur form (EBNF) in programming languages. The problems, of course, with the type-0 (recursively enumerable) and type-1 (context sensitive) languages are that the membership, emptiness, and finiteness problems are undecidable for type-0 languages, and that the emptiness and finiteness problems are undecidable for type-1 languages. In spite of the fact that the class of context-free languages is nicely behaved in terms of decidability, many real-world problems cannot accurately be described by context-free languages. This thesis deals mainly with classes of languages that properly contain the context-free languages, and that are contained in the recursively enumerable languages. Next we list problems that can be described by languages, but not context-free languages. This list of problems is mainly from the introduction in [10]. We illustrate with our own examples how grammars with regulated rewriting and ET0L systems can generate some of these languages.

The reader that is not familiar with the concepts of regulated rewriting is advised to start by Chapter 3.

2.1 Natural languages

In our first example we show that the English language is not context-free. The language consisting of grammatically correct English sentences is denoted by L.

First we consider sentences of the following form in L.

`John, Mary, David, ... are a widower, a widow, a widower, ..., respectively.`

Let h be the homomorphism (see Definition 3.1) defined on L that maps each occurrence of `widower` or any masculine name to a, and similarly each occurrence of `widow` or any feminine name to b. All other words or punctuation marks are deleted by h. We assume that we do not have any names that are both masculine and feminine. Let $N = \{$`John, Mary, David, ...`$\}$ be the set of names and R the language described by the following regular expression (see Definition 3.6): $($$N$ `comma`$)^*$ N `are(a widower comma | a widow comma)`* `respectively fullstop`. In the regular expression we use `comma` and `fullstop` instead of "," and "." , in order to avoid confusion. Note that $h(L \cap R) = \{xx \mid x \in \{a,b\}^*, |x| \geq 2\}$, where $|x|$ is the length of the string x. From the pumping lemma for context-free languages (see Theorem 3.1) it follows that $h(L \cap R)$ is not context-free. Thus, since the context-free languages are closed under homomorphisms and intersection with regular languages [30], L is not a context-free language.

The following matrix grammar G (see Definition 3.8) with context-free rules generates the language $\{xx \mid x \in \{a,b\}^*, |x| \geq 2\}$. Let $G = (\{S, A, B\}, \{a, b\}, R, S)$, where R consists of the following matrices: $(S \rightarrow aAaB)$, $(S \rightarrow bAbB)$, $(A \rightarrow aA, B \rightarrow aB)$, $(A \rightarrow bA, B \rightarrow bB)$, $(A \rightarrow a, B \rightarrow a)$, $(A \rightarrow b, B \rightarrow b)$.

Note that the matrix grammar G ensures that $|x| \geq 2$ by applying the matrix $(S \rightarrow aAaB)$ or $(S \rightarrow bAbB)$ initially. The two matrices $(A \rightarrow aA, B \rightarrow aB)$ and $(A \rightarrow bA, B \rightarrow bB)$ keep the two parts of the string similar during the derivation, and the last two matrices for termination.

2.2 Graphs of discrete functions

Let $f : \mathbb{N}_0^k \rightarrow \mathbb{N}_0$ be a function ($\mathbb{N}_0 = \mathbb{N} \cup \{0\}$) where $\mathbb{N}$ is the set of natural numbers. We represent the graph of the function f by the language L_f, which is defined as follows. Let $a_1, a_2, ..., a_{k+1}$ be distinct terminal symbols, then $L_f = \{a_1^{n_1} a_2^{n_2} \ldots a_k^{n_k} a_{k+1}^{f(n_1,n_2,...,n_k)} \mid (n_1, n_2, ..., n_k) \in dom(f)\}$.

Before we continue, we first state Parikh's theorem [35]. Parikh's theorem states that the Parikh image of every context-free language is semi-linear. Let $\Sigma = \{a_1, a_2, \ldots, a_n\}$, where the order of $a_1, a_2, \ldots, a_n$ is arbitrary but fixed. For $w = a_1^{m_1} \ldots a_n^{m_n} \in \Sigma^*$, the Parikh image $\psi(w)$ is defined as $\psi(w) = (m_1, m_2, \ldots, m_n)$. Next we give the definition of a semi-

linear set as given in [35]. Let $\mathbb{N}_0^n$ denote the Cartesian product of n copies of $\mathbb{N}_0$. A subset Q of $\mathbb{N}_0^n$ is linear if there exist elements $\alpha, \beta_1, \ldots, \beta_m$ of $\mathbb{N}_0^n$ such that Q is equal to $\{x \mid x = \alpha + n_1\beta_1 + \cdots + n_m\beta_m, \quad n_i \in \mathbb{N}_0 \text{ for } 1 \leq i \leq m\}$. A set Q is semi-linear if Q is the union of a finite number of linear sets.

From Parikh's theorem we have that a necessary condition for the graph of a function to be represented by a context-free language as above, is that the graph L_f is semi-linear.

As an example we consider the function $f : \mathbb{N}_0 \to \mathbb{N}_0$, defined by $f(x) = x^k$, $k > 1$. Thus, $L_f = \{a_1^n a_2^{n^k} \mid n \geq 0\}$. From Parikh's theorem it follows that L_f is not context-free.

As another example we consider the function $f : \mathbb{N}_0^2 \to \mathbb{N}_0$, defined as follows.

$$f(x,y) = \begin{cases} x & \text{if } x = y \\ 0 & \text{otherwise.} \end{cases}$$

Thus $L_f = \{a_1^n a_2^n a_3^n \mid n \geq 0\} \cup \{a_1^n a_2^m \mid n, m \in \mathbb{N}_0, \quad n \neq m\}$. From the pumping lemma we have that L_f is not context-free.

The language L_f can be generated by the bag context grammar (see Definition 3.20) $G = (\{S, S_1, S_2, S_3, A_1, A_2\}, \{a_1, a_2, a_3\}, R, S, 0)$, where R contains the rules below.

$$\begin{array}{ll}
S \to S_1 S_2 S_3 & (0,0;1) \\
S \to \varepsilon & (0,0;1) \\
S \to A_1 A_2 & (0,0;0) \\
S \to A_1 & (0,0;1) \\
S \to A_2 & (0,0;-1) \\
 & \\
S_1 \to a_1 S_1 & (1,1;1) \\
S_1 \to a_1 & (1,1;-1) \\
S_2 \to a_2 S_2 & (2,2;1) \\
S_2 \to a_2 & (0,0;0) \\
S_3 \to a_3 S_3 & (3,3;-2) \\
S_3 \to a_3 & (0,0;0), \\
 & \\
A_1 \to a_1 A_1 & (-\infty,\infty;1) \\
A_2 \to a_2 A_2 & (-\infty,\infty;-1) \\
A_1 \to a_1 & (\neq 0;0) \\
A_2 \to a_2 & (\neq 0;0)
\end{array}$$

Note that we use ε (or sometimes λ) to indicate the empty string.

This grammar generates $\{a_1^n a_2^n a_3^n \mid n \geq 0\}$ by applying the rule $S \rightarrow S_1 S_2 S_3 \mid \varepsilon\,(0,0;1)$ initially, where it ensures that all a_1's come before all a_2's and a_3's, and all a_2's come before all a_3's. The bag position in this case ensures that the rules $S_1 \rightarrow a_1 S_1$, $S_2 \rightarrow a_2 S_2$ and $S_3 \rightarrow a_3 S_3$ are applied consecutively. Then it does the same with the termination rules $S_1 \rightarrow a_1$, $S_2 \rightarrow a_2$ and $S_3 \rightarrow a_3$. Thus, we get an equal numbers of a_1's, a_2's and a_3's.

When G generates a string in $\{a_1^n a_2^m \mid n, m \in \mathbb{N}_0,\ \ n \neq m\}$ it starts by replacing S with either $A_1 A_2$, A_1 or A_2. The bag position here represents the difference $(\#a_1 + \#A_1) - (\#a_2 + \#A_2)$, where $\#a_1$, for example, indicates the number of a_1's. The derivation can terminate only if the difference is not zero.

2.3 The courier problem

Let $\Gamma(X, U)$ be a directed graph, with nodes X and directed edges U. A courier is moving from node to node picking up and delivering messages. The only assumption is that all messages will be delivered. In our example we consider the graph $(\{x_1, x_2\}, \{(x_1, x_2), (x_2, x_1)\})$ with nodes x_1 and x_2, and directed edges (x_1, x_2) and (x_2, x_1) see Figure 2.1. We encode

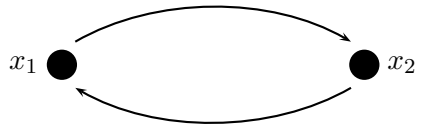


Figure 2.1: The courier problem.

picking up a message from x_i by 'a_i', and delivering a message to x_j by 'b_j'. Let L be all strings that describe the correct courier activity. Let R be the regular language described by the regular expression $a_1^+ a_2^+ b_2^+ b_1^+$. Since $L \cap R$ equals $\{a_1^i a_2^j b_2^i b_1^j \mid i, j \geq 1\}$, which is not context-free, L is also not context-free. Note that $L = \{x \in \{a_1, a_2, b_1, b_2\}^* \mid |x| \geq 1, |x|_{a_1} = |x|_{b_2}, |x|_{a_2} = |x|_{b_1}, |s|_{a_1} \geq |s|_{b_2}$ and $|s|_{a_2} \geq |s|_{b_1}$ for each prefix s of $x\}$, where $|x|_{a_1}$, for example, indicates the number of occurrences of a_1 in the string x. The language L is generated by the 2-bag context grammar $G = (\{S\}, \{a_1, a_2, b_1, b_2\}, R, S, (0,0))$, (see Definition

3.20) where R contains the following rules:

$$\begin{aligned}
S &\to a_1 S \qquad ((0,0),(\infty,\infty);(1,0)),\\
S &\to a_2 S \qquad ((0,0),(\infty,\infty);(0,1)),\\
S &\to b_1 S \qquad ((0,1),(\infty,\infty);(0,-1)),\\
S &\to b_2 S \qquad ((1,0),(\infty,\infty);(-1,0)),\\
S &\to b_1 \qquad ((0,1),(0,1);(0,-1)),\\
S &\to b_2 \qquad ((1,0),(1,0);(-1,0)).
\end{aligned}$$

In this grammar, the first bag position counts the number of messages from x_1 that still need to be delivered to x_2. The second bag position does the same but for the messages from x_2 that still need to be delivered to x_1. A derivation in the grammar can only terminate if the courier delivers the last message to its destination.

2.4 Developmental biology

This example is taken from [28], where the growth of a red algae is described in terms of formal language theory. The first few stages are depicted in Figure 2.2 using Treebag [12]. There are nine cell types which are simply denoted by 1 to 9, respectively.

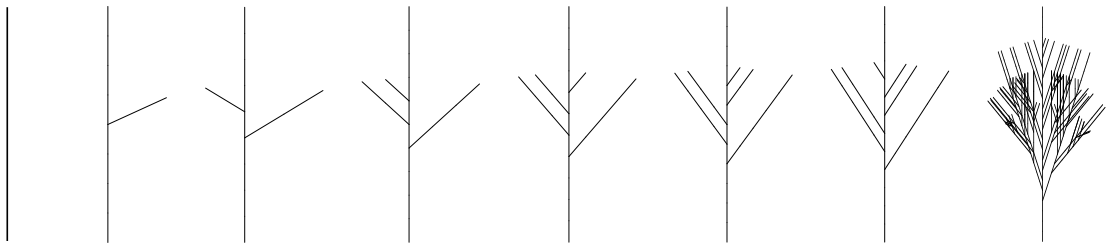


Figure 2.2: Steps 1, 8, 9, 10, 11, 12, 13, 23 in the growth of red algae using Treebag

The first nine stages can be denoted by [1], [23], [224], [2225], [22265], [222765], [2228765], [2229[3]8765] and [2229[24]9[3]8765]. The growth of a red algae can be described by a D0L grammar (see Definition 3.22), given by $S = (\{1,2,\ldots,9,[,]\}, P, [1])$, where P consists of the production rules below.

$$1 \to 23 \quad 2 \to 2 \quad 3 \to 24 \quad 4 \to 25 \quad 5 \to 65 \quad 6 \to 7 \quad 7 \to 8 \quad 8 \to 9[3] \quad 9 \to 9$$

Recall that D0L systems, and ET0L systems in general, apply their rules in parallel during derivations in contrast to context-free grammars where rules are applied sequentially.

Using the rules in the D0L grammar, we can get the following derivation:

[1] ⇒ [23] ⇒ [224] ⇒ [2225] ⇒ [22265] ⇒ [222765] ⇒ [2228765] ⇒ [2229[3]8765] ⇒ [2229[24]9[3]8765].

In Figure 2.2 the nine types of cell 1, . . . , 9 are interpreted as lines with unit length, initially vertical. The pair of brackets [and] indicates the branching, where all cells within a pair of brackets form a branch from the current branch. Thus, steps 1, . . . , 7, in terms of pictures, are vertical lines with unit length difference between any two consecutive stages.

We denote the consecutive stages of a red algae by $s_1, s_2, s_3, \ldots$. By $s'_2, s'_3, \ldots$, we denote the stages from stage two onwards, but with the first symbol, that is 2, removed from each s_i. Thus, for example $s'_2 = [3]$ and $s'_3 = [24]$.

Then we can derive the following inductive definition for s_n, $n \geq 8$.

$$s_n = [2229[s'_{n-6}] \;\; 9[s'_{n-7}] \;\; \ldots \;\; 9[s'_2] \;\; 8765]\,.$$

Since $s_{n+1} = [2229[s'_{n-5}] \;\; 9[s'_{n-6}] \;\; \ldots \;\; 9[s'_2] \;\; 8765]$, we conclude that $|s_{n+1}| = |s_n| + |s_{n-5}| \geq 2|s_{n-5}|$. From the pumping lemma for context-free languages, it follows that the set of lengths of words in an infinite context-free language must contain an infinite arithmetic progression. Thus, since $|s_{n+1}| \geq 2|s_{n-5}|$, the language describing the growth of a red algae is not context-free.

In the previous sections we used string languages to describe real-world problems. In the next section we discuss circumstances in which string grammars and string languages are not appropriate.

2.5 Beyond string languages

A good motivation to study not only string languages, but also the more general class of tree languages, is provided by XML (eXtensible Markup Language). XML plays an essential role in data exchange due to its flexibility, since users are allowed to define their own custom markup languages.

Our contribution in this section is that we show how to express XML documents in terms of regular tree grammars.

Consider the following XML document:

```
<fruitshop>
```

```
   <item>
       <name> apple </name>
       <price> 5 </price>
       <type> green </type>
   </item>
   <item>
       <name> orange </name>
       <price> 2 </price>
   </item>
</fruitshop>.
```

The tree representation of the given XML document is given by the tree shown in Figure 2.3.

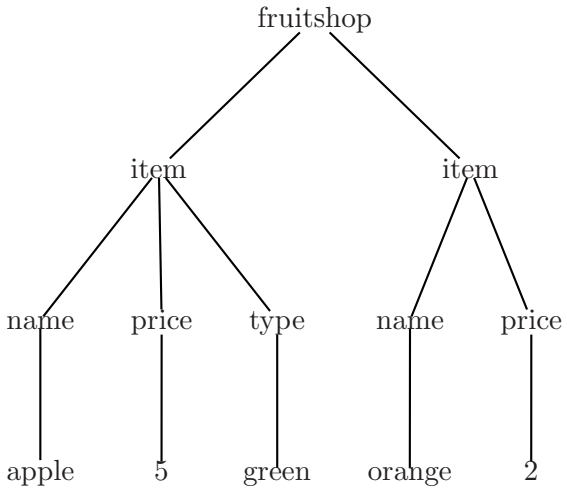


Figure 2.3: Tree representation for the XML document

The leaf nodes in the tree are parsable character data, which we will collectively denote by PCDATA.

More formally, we define the associated tree t of an XML document recursively as follows [2]:

- The leaves of t have labels PCDATA.
- All other nodes have labels from the set of element names of the document.
- A document $< a > w < /a >$, where w contains only PCDATA, has an associated tree with one node labeled a with one child labeled w.
- A document of the form $< a > x_1 \dots x_k < /a >$, where the documents $x_1, \dots, x_k$ have associated trees $t_1, \dots, t_k$, has an associated tree with root a at which the trees $t_1, \dots, t_k$ are attached, from left to right.

The DTD (Document Type Definition) associated with the XML document above, is given by

⟨! ELEMENT fruitshop (item+) ⟩
⟨! ELEMENT item (name, price, type?)⟩
⟨! ELEMENT name (#PCDATA) ⟩
⟨! ELEMENT price (#PCDATA) ⟩
⟨! ELEMENT type (#PCDATA) ⟩.

We can think of the given DTD file as representing the following rewriting rules:

fruitshop → item$^+$
item → name price (type | ε)
name → PCDATA
price → PCDATA
type → PCDATA

where ε indicates the empty string.

In these rules "fruitshop" is considered as the start symbol. The rule "fruitshop → item$^+$", for example, indicates that we may replace "fruitshop" with one or more instances of "item". Similarly, "item → name price (type | ε)" indicates that "item" may be replaced with "name" and "price" or with "name", "price" and "type".

Formally, DTD's are defined as follows.

Definition 2.1. *[2] A DTD is a triple (Σ, d, s_d), where Σ is a finite alphabet (the element names), d is a function that maps Σ symbols to PCDATA or to regular expressions over Σ, and $s_d \in \Sigma$ is the start symbol. We write d instead of (Σ, d, s_d) if there is no confusion by doing so.*

A tree t is valid with respect to d, or satisfies d if its root is labeled by s_d, and for every node with label a, the sequence $a_1, \ldots, a_n$ of labels of its children is in the language defined by $d(a)$. By $L(d)$ we denote the set of all trees that satisfies d.

The trees which represent the XML documents associated with a given DTD (the tree language defined by the DTD) are unranked trees, since an element may have an arbitrary number of children. For instance, in our example the root element "fruitshop" can contain an arbitrary number of the children "item". Also, these tree languages are homogeneous (a tree language is homogeneous if all trees have the same root [2]).

Next we show how we can express the DTD in our example in terms of a regular tree grammar (see Definition 3.25). Since regular tree grammars are defined over ranked alphabets (see the notation in Section 3.6), we need a strategy to encode an unranked tree as a ranked tree. We use here the binary tree encoding for unranked trees which is described as follows [32]:

- The first child of a node remains the first child of that node in the encoding, but it is explicitly encoded as a left child.
- The remaining children are right descendants of the first child.
- Whenever there is a left child but no right child in the encoded tree, a # is inserted for the right child.
- When there is only a right child but no left child in the encoded tree, a # is inserted for the left child.

According to this encoding the binary tree that represents our XML document is shown in Figure 2.4.

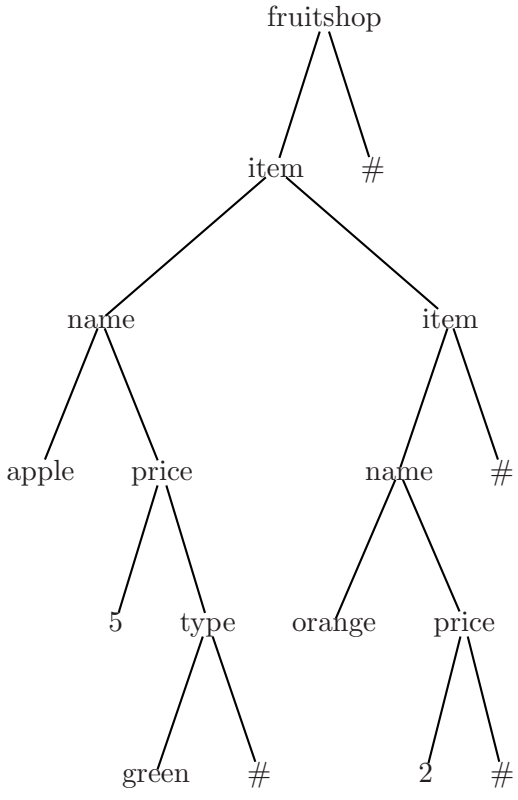


Figure 2.4: Binary tree representation for the XML document

We can describe the DTD in terms of the regular tree grammar $G = (N, \Sigma, R, S)$, where:

$N = \{Fruitshop, Item, Name, Price, Type\}$,
$\Sigma = \{fruitshop^{(2)}, item^{(2)}, name^{(2)}, price^{(2)}, type^{(2)}, PCDATA^{(0)}, \#^{(0)}\}$,
$R = \{ \quad Fruitshop \rightarrow fruitshop[Item, \#]$,
$\quad\quad Item \rightarrow item[Name, Item] \mid item[Name, \#]$,
$\quad\quad Name \rightarrow name[PCDATA, Price]$,
$\quad\quad Price \rightarrow price[PCDATA, Type] \mid price[PCDATA, \#]$,
$\quad\quad Type \rightarrow type[PCDATA, \#] \quad \}$,
and $S = Fruitshop$.

A symbol $x^{(2)}$, with $x \in \Sigma$, indicates that the node with a label x must have exactly two children, and $x^{(0)}$ is a leaf node.

Using the technique explained in this example, one can express any DTD in terms of a regular tree grammar. This shows that the binary encoding for the tree language defined by a DTD is a regular tree language.

In this chapter we gave examples of string languages that model real-world problems, but which are not context-free. We considered grammars with regulated rewriting and ET0L systems as an alternative generating mechanisms to describe these problems. Then we presented circumstances in which string languages are not appropriate. In the next chapter we present various generating mechanisms and give examples to illustrate them.

Chapter 3

Generating mechanisms

In this chapter we introduce various generating mechanisms. After giving elementary definitions, we define the grammars in the Chomsky hierarchy. In decreasing descriptive power they are: the unrestricted, context sensitive, context-free, and regular grammars. After that we introduce various rewriting mechanisms. Using these mechanisms we obtain matrix, programmed, random context, valence, and bag context grammars. Bag context grammars were introduced by Drewes et al. in 2006 (see [14]). We consider, for example, the non-context-free language $L = \{a^n b^m a^n b^m \mid n, m \geq 1\}$ and show how each of these grammars can generate L. Then we introduce branching synchronization grammars. These grammars use parallel, instead of sequential, derivation steps. The ET0L grammars are a well-known subclass of this class of grammars. We conclude with a definition of regular tree grammars. We give various examples of our own to illustrate the definitions.

3.1 Elementary definitions

Homomorphisms

Definition 3.1. *[30] Let Σ and Δ be two alphabets. A homomorphism f is a function from Σ^* to Δ^* such that $f(xy) = f(x)f(y)$ for all $x, y \in \Sigma^*$.*

Example 3.1. As an example of a homomorphism, consider $h : \{a, b, \ldots, z\}^* \rightarrow \{a, b, \ldots, z\}^*$, where h encrypts each word in $\{a, b, \ldots, z\}^*$ by shifting each letter 3 positions as follows: $h(a) = d$, $h(b) = e$, $\ldots$, $h(w) = z$, and the last three letters to the first three letters, $h(x) = a$, $h(y) = b$, and $h(z) = c$. In other words, if we associate an index with each letter defined by a function i as follows: $i(a) = 0$, $i(b) = 1$, $\ldots$, $i(z) = 25$, then $i(h(\alpha)) = (i(\alpha) + 3) \, mod \, 26$,

for all $\alpha \in \{a, b, \ldots, z\}$. We note that i also can be extended to a homomorphism from $\{a, b, \ldots, z\}^* \to \{0, 1, \ldots, 9\}^*$. □

3.2 The Chomsky hierarchy

Unrestricted grammars

Definition 3.2. *[10] An unrestricted grammar (type-0 grammar) G is a 4-tuple (N, Σ, R, S), where*

- *N is a non-empty set of symbols, called nonterminals;*
- *Σ is a set of symbols, called terminals, with $N \cap \Sigma = \emptyset$;*
- *if $V_G = N \cup \Sigma$, then R is a set of production rules of the form $v \to w$, where $v \in V_G^* N V_G^*$ and $w \in V_G^*$;*
- *$S \in N$ is the start symbol.*

A string $x \in V_G^+$ directly derives $y \in V_G^*$ if $x = w_1 v w_2$, $y = w_1 w w_2$, with $w_1, w_2 \in V_G^*$, and there is a rule $v \to w \in R$. If x directly derives y, we denote this by $x \Rightarrow_G y$, or simply $x \Rightarrow y$. The reflexive and transitive closure of the relation $\Rightarrow$ is denoted by $\Rightarrow^*$. If $x \Rightarrow^* y$, we say that x derives y.

The language generated by G is defined as $\{\, x \mid x \in \Sigma^* \text{ and } S \Rightarrow^* x \,\}$ and denoted by $L(G)$. The class of languages generated by unrestricted grammars is called the family of recursively enumerable languages and denoted by $\mathcal{L}(RE)$.

Example 3.2. As an example of an unrestricted grammar, consider $G = (\{S, A, B, C, X\}, \{a, b\}, R, S)$, where R contains the following rules: $S \to aAaX$, $A \to aAa \mid bBC$, $B \to bBC$, $Ca \to aC$, $CX \to bX$, $Cb \to bb$, $X \to \varepsilon$, $B \to \varepsilon$.
It is easy to see that $L(G) = \{a^n b^m a^n b^m \mid n, m \geq 1\}$.

The grammar G generates the string $a^2b^3a^2b^3$ as follows:

$S \Rightarrow aAaX \Rightarrow a^2Aa^2X \Rightarrow a^2bBCa^2X \Rightarrow^* a^2b^3BC^3a^2X \Rightarrow a^2b^3C^3a^2X \Rightarrow^* a^2b^3a^2C^3X \Rightarrow a^2b^3a^2C^2bX \Rightarrow a^2b^3a^2C^2b \Rightarrow^* a^2b^3a^2b^3$. □

Context sensitive grammars

Definition 3.3. *[20, 27] A context sensitive grammar (type-1 grammar) is a type-0 grammar with production rules of the form $x_1 A x_2 \to x_1 w x_2$, where $x_1, x_2 \in V_G^*$, $A \in N$ and $w \in V_G^+$.*

The exception to this is that we allow the rule $S \to \varepsilon$, when S does not appear in the right-hand side of any rule.

Equivalently, the context sensitive grammars can be defined as length increasing type-0 grammars, that is, for each production rule $x \to y$ it holds that $|x| \leq |y|$. Again we obviously allow $S \to \varepsilon$, when S does not appear in the right-hand side of any rule.

Again the language generated by G is defined as $\{ x \mid x \in \Sigma^* \text{ and } S \Rightarrow^* x \}$ and denoted by $L(G)$. The class of all languages generated by the context sensitive grammars is called the family of context sensitive languages and denoted by $\mathcal{L}(CS)$.

Example 3.3. We show in this example that $L = \{a^n b^m a^n b^m \mid n, m \geq 1\} \in \mathcal{L}(CS)$. One can see this by considering the context sensitive grammar G, where $G = (\{S, A, B, C, X\}, \{a, b\}, R, S)$, and R contains the following rules: $S \to AX$, $A \to aAa \mid aBa$, $B \to bBC$, $Ca \to aC$, $Cb \to bb$, $X \to b$, $B \to b$.

The grammar G generates the string $a^2b^3a^2b^3$ as follows

$S \Rightarrow AX \Rightarrow aAaX \Rightarrow a^2Ba^2X \Rightarrow^* a^2b^2BC^2a^2X \Rightarrow a^2b^3C^2a^2X \Rightarrow^* a^2b^3a^2C^2X \Rightarrow a^2b^3a^2C^2b \Rightarrow^* a^2b^3a^2b^3$. □

Context-free grammars

Definition 3.4. *[30] A context-free grammar is a type-0 grammar where the rules are of the form $A \to x$, with $A \in N$ and $x \in V_G^*$.*

The language generated by G is defined as $\{ x \mid x \in \Sigma^* \text{ and } S \Rightarrow^* x \}$ and denoted by $L(G)$. The class of all languages generated by context-free grammars is called the class of context-free languages and denoted by $\mathcal{L}(CF)$.

Example 3.4. As an example of a context-free grammar, consider $G = (\{S\}, \{\wedge, \vee, \sim, (,), x\}, R, S)$, and R contains the rules: $S \to S \wedge S \mid S \vee S \mid \sim S \mid (S) \mid x$. Then G is a context-free grammar that generates all the well-formed logic expressions over the variable x with the operations $\wedge$ "and", $\vee$ "or", and $\sim$ "not". □

Pumping lemma for context-free languages

Theorem 3.1. *[36] Let A be a context-free language. Then there is a number p (the pumping length), such that if $s \in A$ and $|s| \geq p$, then s can be written as $uvxyz$ such that:*

- *for each $i \geq 0$, $uv^ixy^iz \in A$;*

- $|vy| > 0$, *and*
- $|vxy| \leq p$. ■

One can use for example the pumping lemma to show that $L = \{a^n b^m a^n b^m \mid n, m \geq 1\}$ is not context-free.

It should be noted that the family of context-free languages is closed under union, homomorphism, concatenation, and intersection with regular languages [30].

Regular grammars

Next we define the regular operations on languages.

Definition 3.5. *[36] Let A and B be any two languages, we define the regular operations, union, concatenation, and star as follows:*

- *Union:* $A \cup B$ *or* $A \mid B = \{x \mid x \in A \text{ or } x \in B\}$.
- *Concatenation:* $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$, *and we simply write* AB.
- *Star:* $A^* = \{x_1 x_2 \dots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$. *The star operation is also called repetition or Kleene Closure.*

Next we define regular expressions.

Definition 3.6. *[36] A regular expression R over a finite alphabet Σ is defined recursively as follows:*

- $\emptyset$ *is a regular expression;*
- ε *is a regular expression;*
- a, *for* $a \in \Sigma$ *is a regular expression;*
- *if* R_1 *and* R_2 *are regular expressions, then* $(R_1 \mid R_2)$ *is a regular expression;*
- *if* R_1 *and* R_2 *are regular expressions, then* $(R_1 \circ R_2)$ *is a regular expression;*
- *if* R *is a regular expression, then* R^* *is a regular expression.*

Definition 3.7. *[30] Let G be a type-0 grammar. We say that G is linear if all the rules of G are of the form $A \to \alpha B \alpha'$ or $A \to \alpha$, with $\alpha, \alpha' \in \Sigma^*$ and $A, B \in N$. G is right-linear if the rules are of the form $A \to \alpha B$ or $A \to \alpha$, and left-linear if the rules are of the form $A \to B\alpha$ or $A \to \alpha$, where $A, B \in N$ and $\alpha \in \Sigma^*$.*

A regular grammar is a type-0 grammar that has production rules of the form $A \to aB$, or $A \to a$, where $A, B \in N$ and $a \in \Sigma \cup \{\varepsilon\}$.

As before, the language generated by G is defined as $\{ x \mid x \in \Sigma^* \text{ and } S \Rightarrow^* x \}$ and denoted by $L(G)$. The class of languages generated by regular grammars is called the class of regular languages and denoted by $\mathcal{L}(REG)$.

It can been shown that the classes of regular, right-linear, and left-linear grammars are equivalent and that they characterize the family of regular languages [30]. In other words the family of languages that is generated by these classes of grammars is the family of regular languages.

Next we give an example of a regular language.

Example 3.5. Let $\Sigma = \{0, \ldots, 9\}$, $\Sigma' = \{1, \ldots, 9\}$. Note that Σ, for example, can be written as a regular expression $(0|1|\ldots|9)$ which will also be denoted by Σ. Consider the set of rational numbers, which can be described by the regular expression $0 \mid (\varepsilon \mid + \mid -)(\Sigma'\Sigma^*(\varepsilon \mid /\Sigma'\Sigma^*))$. This language can be generated using the regular grammar $G = (\{S, E, A, B, C, D\}, \Sigma \cup \{+, -, /\}, R, S)$, where R contains rules: $S \to 0 \mid E \mid +E \mid -E$, $E \to d'A \mid d'$, $A \to dA \mid dB \mid d$, $B \to /C$, $C \to d'D \mid d'$, $D \to dD \mid d$, where $d \in \Sigma$ and $d' \in \Sigma'$. □

3.3 Grammars with regulated rewriting

Grammars with regulated rewriting are type-0 grammars with an additional mechanism to restrict the application of the rules in order to avoid certain derivations. There are many types of restrictions that can be considered, but we will focus on the grammars defined below.

Matrix grammars

Definition 3.8. *[10] A matrix grammar is a quadruple $G = (N, \Sigma, M, S)$, where N, Σ and S are defined exactly as in any of the grammars in the Chomsky hierarchy. The rules in M are sequences $m = (r_1, r_2, ..., r_n)$, where each r_i is of the same form as the rules in a type-0 grammar in the Chomsky hierarchy. If the rules in each sequence of rules are of type i, where $i \in \{0, 1, 2, 3\}$, then G is a type-i matrix grammar.*

Definition 3.9. *[10] Let $G = (N, \Sigma, M, S)$ be a matrix grammar. For $x, y \in V_G^*$, we write $x \Rightarrow y$ if there are $x_0, ..., x_n$, $x'_0, ..., x'_{n-1}$ and $x''_0, ..., x''_{n-1}$ in V_G^* with $x_0 = x$ and $x_n = y$, and a matrix rule $(\alpha_1 \to \beta_1, \alpha_2 \to \beta_2, ..., \alpha_n \to \beta_n) \in M$, such that $x_{i-1} = x'_{i-1}\alpha_i x''_{i-1}$ and $x_i = x'_{i-1}\beta_i x''_{i-1}$ for $1 \leq i \leq n$.*

Definition 3.10. *[10] A matrix grammar with appearance checking is a grammar $G = (N, \Sigma, F, M, S)$, with $F \subseteq M$ and with (N, Σ, M, S) a matrix grammar as defined above.*

Definition 3.11. *[10] Let $G = (N, \Sigma, F, M, S)$ be a matrix grammar with appearance checking. For $x, y \in V_G^*$, we write $x \Rightarrow_{ac} y$ if there are $x_0, ..., x_n$, $x'_0, ..., x'_{n-1}$ and $x''_0, ..., x''_{n-1}$ in V_G^**

with $x_0 = x$ and $x_n = y$, and a matrix rule $(\alpha_1 \to \beta_1, \alpha_2 \to \beta_2, ..., \alpha_n \to \beta_n) \in M$, such that for $1 \leq i \leq n$, $x_{i-1} = x'_{i-1}\alpha_i x''_{i-1}$ and $x_i = x'_{i-1}\beta_i x''_{i-1}$, or $\alpha_i \to \beta_i \in F$, α_i is not a substring of x_{i-1}, and $x_{i-1} = x_i$.

As usual, the language generated by G is defined as $L(G) = \{ x \mid x \in \Sigma^* \text{ and } S \Rightarrow^* x \}$ and denoted by $L(G)$.

The class of matrix grammars with context-free rules without λ productions (productions of the form $A \to \lambda$) and no appearance checking, are for example denoted by $\mathcal{L}(M, CF - \lambda)$, and the same class with appearance checking by $\mathcal{L}(M, CF - \lambda, ac)$. Also, the class of matrix grammars of type-0, type-1, type-2, and type-3 without appearance checking are denoted by $\mathcal{L}(M, RE)$, $\mathcal{L}(M, CS)$, $\mathcal{L}(M, CF)$, and $\mathcal{L}(M, REG)$ respectively, and with appearance checking by $\mathcal{L}(M, RE, ac)$, $\mathcal{L}(M, CS, ac)$, $\mathcal{L}(M, CF, ac)$, and $\mathcal{L}(M, REG, ac)$.

Example 3.6. As an example of a matrix grammar we consider the following grammar (exercise in [10]). Let $G = (\{S, A, B, C, D\}, \{a, b\}, M, S)$, be the matrix grammar with the matrix rules given below.
$m_1 : (S \to ABCD)$, $m_2 : (A \to aA, C \to aC)$, $m_3 : (B \to bB, D \to bD)$, $m_4 : (A \to a, C \to a)$, $m_5 : (B \to b, D \to b)$.
Then $L(G) = \{a^n b^m a^n b^m \mid n, m \geq 1\}$.

The grammar G generates the string ab^2ab^2 as follows:

$$S \Rightarrow_{m_1} ABCD \Rightarrow_{m_4} aBaD \Rightarrow_{m_3} abBabD \Rightarrow_{m_5} ab^2ab^2.$$

It should be noted that each word in $L(G)$ consists of four parts A, B, C and D this is given by m_1. The two parts A and C contain only a's with the same number. Also B and D contain only b's with the same numbers. This is given by the two matrices m_2 and m_3, respectively. Finally m_4 and m_5 for termination. □

Programmed grammars

Definition 3.12. *[10] A programmed grammar is a quadruple $G = (N, \Sigma, P, S)$, where P contains finitely many production rules of the form $(\, r : \alpha \to \beta,\, \sigma(r),\, \phi(r)\,)$, and N, Σ, S, and $\alpha \to \beta$ are as in the Chomsky grammars. In the rule $(\, r : \alpha \to \beta,\, \sigma(r),\, \phi(r)\,)$, r is a label and each rule in P has a unique label. The set of all labels is denoted by $Lab(P)$ thus, $Lab(P) = \{r : (\, r : \alpha \to \beta,\, \sigma(r),\, \phi(r)\,) \in P\}$. In the production rule $(\, r : \alpha \to \beta, \sigma(r), \phi(r)\,)$, we have that $\sigma(r) \subseteq Lab(P)$ and $\phi(r) \subseteq Lab(P)$. The sets $\sigma(r)$ and $\phi(r)$ are referred to as the success and failure fields associated with r, respectively.*

Definition 3.13. *[10] If $(x, r_1), (y, r_2) \in V_G^* \times Lab(P)$, we write $(x, r_1) \Rightarrow (y, r_2)$ if either:*

- $x = x_1\alpha x_2$, $y = x_1\beta x_2$, $x_1, x_2 \in V_G^*$, $(\, r_1 : \alpha \to \beta,\ \sigma(r_1),\ \varphi(r_1)\,) \in P$ *and* $r_2 \in \sigma(r_1)$, *or*
- $x = y$, $(\, r_1 : \alpha \to \beta,\ \sigma(r_1),\ \varphi(r_1)\,) \in P$ *with* α *not a substring of* x, *and* $r_2 \in \varphi(r_1)$.

Definition 3.14. *[10] If all the failure fields of rules in P are empty, then the programmed grammar G is without appearance checking.*

For a programmed grammar G, the language generated by G is given by $\{\, x \in \Sigma^* \mid (S, r_1) \Rightarrow^* (x, r_2)$ for some $r_1, r_2 \in Lab(P)\,\}$.

We denote the classes of programmed grammars with and without appearance checking by $\mathcal{L}(P, X, ac)$ and $\mathcal{L}(P, X)$, respectively. In $\mathcal{L}(P, X, ac)$ and $\mathcal{L}(P, X)$ the symbol X could for example be equal to RE, CS, CF, $CF{-}\lambda$, or REG. Thus, by $\mathcal{L}(P, RE)$, $\mathcal{L}(P, CS)$, $\mathcal{L}(P, CF)$, $\mathcal{L}(P, CF - \lambda)$ and $\mathcal{L}(P, RE)$ for example, we denote the classes of programmed grammars without appearance checking and with production rules of the same form as in the recursively enumerable, context sensitive, context-free, context-free without λ productions, and regular grammars, respectively.

Example 3.7. As an example of a programmed grammar, consider the language $L = \{a^n b^m a^n b^m \mid n, m \geq 1\}$. The language L can be generated using programmed grammar $G = (\{S, A, B, C, D\}, \{a, b\}, P, S)$, where R contains the production rules:

$$\begin{array}{lll}
(r_1 : S \to ABCD, & \{r_2, r_4, r_6, r_8\}, & \emptyset), \\
(r_2 : A \to aA, & \{r_3\}, & \emptyset), \\
(r_3 : C \to aC, & \{r_2, r_4, r_6, r_8\}, & \emptyset), \\
(r_4 : B \to bB, & \{r_5\}, & \emptyset), \\
(r_5 : D \to bD, & \{r_2, r_4, r_6, r_8\}, & \emptyset), \\
(r_6 : A \to a, & \{r_7\}, & \emptyset), \\
(r_7 : C \to a, & \{r_4, r_8\}, & \emptyset), \\
(r_8 : B \to b, & \{r_9\}, & \emptyset), \\
(r_9 : D \to b, & \{r_2, r_6\}, & \emptyset).
\end{array}$$

The grammar G generates the string ab^2ab^2 as follows:

$$(S, r_1) \Rightarrow (ABCD, r_6) \Rightarrow^* (aBaD, r_4) \Rightarrow^* (abBabD, r_8) \Rightarrow^* (ab^2ab^2, r_2).$$

In this derivation we start from (S, r_1), by applying r_1 the start symbol is replaced with $ABCD$ and we choose the next rule from r_1 success field, that is $\sigma(r_1) = \{r_2, r_4, r_6, r_8\}$, thus we have $(ABCD, r_6)$. In general, when we apply the rule r_2 we must apply r_3 next, since $\sigma(r_2) = \{r_3\}$ also note that we cannot apply r_3 again unless we apply r_2. Thus we get equal

number of a's generated from A and C. Similarly for B and D but we get equal number of b's. □

Random context grammars

Definition 3.15. *[10] A random context grammar is a quadruple $G = (N, \Sigma, R, S)$, where R contains finitely many production rules of the form $(\alpha \to \beta, P, F)$, with $P, F \subseteq N$, and $N, \Sigma, S, \alpha \to \beta$ are as in the Chomsky grammars. In the rule $(\alpha \to \beta, P, F)$, the sets P and F are called the permitting and forbidding context respectively.*

Definition 3.16. *[10] For two strings $x, y \in V_G^*$, we write $x \Rightarrow y$ if $x = x'\alpha x''$, $y = x'\beta x''$ for some $x', x'' \in V_G^*$, and $(\alpha \to \beta, P, F)$ is a rule in R such that all nonterminals in P appear in x' and x'', and none of the nonterminals in F appear in x' or x''.*

Definition 3.17. *[10] A random context grammar is without appearance checking if all forbidding contexts of rules are empty. A random context grammar without appearance checking is also called a random permitting context grammar. We denote the class of random permitting context grammars where all rules are of type i, by $rPcg$-i.*
A random forbidding context grammar is a random context grammar in which the permitting contexts of all rules are empty. The class of random forbidding context grammars with type-i rules are denoted by $rFcg$-i. The families of languages corresponding to $rPcg$-i and $rFcg$-i are denoted by $rPcl$-i and $rFcl$-i, respectively.

When G is a $rPcg$ or $rFcg$ we write the rules in the form $(\alpha \to \beta, C)$, where C is the permitting context, or the forbidding context, respectively.

As before we use the notation $\mathcal{L}(RC, X, ac)$ and $\mathcal{L}(RC, X)$ with $X \in \{RE, CS, CF, CF - \lambda, REG\}$.

Example 3.8. This example illustrates random context grammars. We consider the language $L = \{a^n b^m a^n b^m \mid n, m \geq 1\}$. This language can be generated using the random context grammar $G = (\{S, A, B, C, D, A', B', C', D'\}, \{a, b\}, R, S)$, where R contains the following rules:

$$\begin{aligned}
r_1 &: (S \to ABCD, \quad \emptyset, \quad \emptyset),\\
r_2 &: (A \to aA', \quad \{C\}, \quad \emptyset),\\
r_3 &: (C \to aC', \quad \{A'\}, \quad \emptyset),\\
r_4 &: (A' \to A, \quad \{C'\}, \quad \emptyset),\\
r_5 &: (C' \to C, \quad \{A\}, \quad \emptyset),
\end{aligned}$$

$$
\begin{aligned}
&r_6 : (B \to bB', \quad \{D\}, \quad \emptyset),\\
&r_7 : (D \to bD', \quad \{B'\}, \quad \emptyset),\\
&r_8 : (B' \to B, \quad \{D'\}, \quad \emptyset),\\
&r_9 : (D' \to D, \quad \{B\}, \quad \emptyset),
\end{aligned}
$$

$$
\begin{aligned}
&r_{10} : (A \to a, \quad \{C\}, \quad \emptyset),\\
&r_{11} : (C \to a, \quad \emptyset, \quad \{A, A'\}),\\
&r_{12} : (B \to b, \quad \{D\}, \quad \emptyset),\\
&r_{13} : (D \to b, \quad \emptyset, \quad \{B, B'\}).
\end{aligned}
$$

The grammar G derives the string ab^2ab^2 as follows:

$$S \Rightarrow_{r_1} ABCD \Rightarrow^*_{r_{10},r_{11}} aBaD \Rightarrow^*_{r_6,r_7} abB'abD' \Rightarrow^*_{r_8,r_9} abBabD \Rightarrow^*_{r_{12},r_{13}} ab^2ab^2.$$

The permitting context in the rules $r_2, \ldots, r_5$ ensures that once we apply r_2 then r_2 cannot be applied again unless r_3, r_4 and r_5 are applied. Also if we apply r_3 then r_3 cannot be applied again unless r_4, r_5 and r_2 are applied. Thus, ensures that A and C generate equal number of a's. Note that A' is used to check if r_2 has been applied, similarly C' is used to check if r_3 has been applied. In a similar way the rules $r_6, \ldots, r_9$ are applied. The forbidding context in r_{11} shows that C cannot terminate till A terminates and similarly in r_{13} D cannot terminate till B terminates. □

Valence grammars

Definition 3.18. *[25] A valence grammar G over $\mathbb{Z}^k$ (k copies of the integers) is a quadruple $G = (N, \Sigma, R, S)$, where R contains rules of the form $(v \to w, r)$, $r \in \mathbb{Z}^k$ is the valence of the rule, and N, Σ, S and $v \to w$ are as in Chomsky grammars.*

Definition 3.19. *[25] For $x, y \in V_G^*$ and $a, b \in \mathbb{Z}^k$ we write $(x, a) \Rightarrow (y, b)$ or $x_a \Rightarrow y_b$ if there is a rule $(v \to w, r)$ such that $x = x_1vx_2$ and $y = x_1wx_2$ and $b = a + r$, with $x_1, x_2 \in V_G^*$.*

The language generated by G is defined by $L(G) = \{x \mid x \in \Sigma^*, (S, \vec{0}) \Rightarrow^* (x, \vec{0})\}$.

We will also use the notation $\mathcal{L}(V, X)$, with $X \in \{RE, CS, CF, CF - \lambda, REG\}$.

Example 3.9. To illustrate valence grammars, consider the language $L = \{a^nb^ma^nb^m \mid n, m \geq 1\}$. This language can be generated by using the valence grammar G over $\mathbb{Z}^2$, where

$G = (\{S, A, B, C, D\}, \{a, b\}, R, S)$, and R contains the following rules:

$$\begin{array}{llll} r_1 : (S \to ABCD, & (0,0)), & & \\ r_2 : (A \to aA, & (1,0)), & r_3 : (A \to a, & (1,0)), \\ r_4 : (C \to aC, & (-1,0)), & r_5 : (C \to a, & (-1,0)), \\ r_6 : (B \to bB, & (0,1)), & r_7 : (B \to b, & (0,1)), \\ r_8 : (D \to bD, & (0,-1)), & r_9 : (D \to b, & (0,-1)). \end{array}$$

The string ab^2ab^2 is generated by G as follows:

$$S_{(0,0)} \Rightarrow_{r_1} ABCD_{(0,0)} \Rightarrow^*_{r_3,r_6} abBCD_{(1,1)} \Rightarrow_{r_7} ab^2CD_{(1,2)} \Rightarrow^*_{r_5,r_8,r_9} ab^2ab^2_{(0,0)}.$$

In this grammar the derivation starts by replacing S with $ABCD$ without changing the valence. When the first component of the valence is zero that means the number of a's generated from A is equal to the number of a's generated from C. Similarly for the second component of the valence with B and D. □

Bag context grammars (BCGs)

Definition 3.20. *[14] A k-bag context grammar (k-BCG) is a 5-tuple $G = (N, \Sigma, R, S, \beta_0)$, where N, Σ, S are as in Chomsky grammars, $\beta_0 \in \mathbb{Z}^k$ (k copies of the integers) is the initial bag, and R contains rules of the form: $v \to w\,(\lambda, \mu; \alpha)$, where $v \to w$ are as in Chomsky grammars and $\lambda, \mu \in \mathbb{Z}^k_\infty$ ($\mathbb{Z}_\infty = \mathbb{Z} \cup \{+\infty, -\infty\}$) are the lower and upper limits respectively, and $\alpha \in \mathbb{Z}^k$ is the bag adjustment.*

Definition 3.21. *[14] We say (x, β) derives (y, β') in one step, and we write $(x, \beta) \Rightarrow (y, \beta')$ or $x_\beta \Rightarrow y_{\beta'}$, if $x = x_1 v x_2$ and $y = x_1 w x_2$, and there is a rule $v \to w\,(\lambda, \mu; \alpha)$ in R with $\lambda \leq \beta \leq \mu$ and $\beta' = \beta + \alpha$.*

The language generated by G is defined by $L(G) = \{x \mid x \in \Sigma^*, (S, \beta_0) \Rightarrow^* (x, \beta), \beta \in \mathbb{Z}^k\}$.

Again we use the notation $\mathcal{L}(BCG, X)$, with $X \in \{RE, CS, CF, CF - \lambda, REG\}$.

For a vector in $\mathbb{Z}^k_\infty$ with all its components equal to x, we write x_k when there is no confusion. The rule $v \to w\,(\neq a; \alpha)$ for $a, \alpha \in \mathbb{Z}^k$ means this rule is applicable for all $\mathbb{Z}^k \setminus \{a\}$. For a k-BCG when k is clear we omit k from notation.

Example 3.10. A 2-BCG that generates the language $L = \{a^n b^m a^n b^m \mid n, m \geq 1\}$ is given by the grammar $G = (\{S, A, B, C, D\}, \{a, b\}, R, S, (0,0))$, where R contains the following rules:

$$
\begin{array}{ll}
r_1 : S \to ABCD & ((0,0),(0,0);(1,1)),\\
r_2 : A \to aA & ((1,0),(1,\infty);(1,0)),\\
r_3 : C \to aC & ((2,0),(2,\infty);(-1,0)),\\
r_4 : B \to bB & ((0,1),(\infty,1);(0,1)),\\
r_5 : D \to bD & ((0,2),(\infty,2);(0,-1)),\\
r_6 : A \to a & ((1,0),(1,\infty);(-1,0)),\\
r_7 : C \to a & ((0,0),(0,\infty);(0,0)),\\
r_8 : B \to b & ((0,1),(\infty,1);(0,-1)),\\
r_9 : D \to b & ((0,0),(\infty,0);(0,0)).
\end{array}
$$

The grammar G generates the string ab^2ab^2 as follows:

$$S_{(0,0)} \Rightarrow_{r_1} ABCD_{(1,1)} \Rightarrow^*_{r_6,r_7} aBaD_{(0,1)} \Rightarrow^*_{r_4,r_5} abBabD_{(0,1)} \Rightarrow^*_{r_8,r_9} ab^2ab^2_{(0,0)}.$$

In this bag context grammar the first bag position ensures that if we apply r_2 we cannot apply it again unless we apply r_3. It also ensures that when we apply r_3 it cannot be applied again till we apply r_2. Thus, shows r_2 and r_3 are applied the same number of times and hence A and C generate equal number of a's. The same holds for r_4 and r_5 and they generate equal number of b's. The rules r_6 and r_7 show if A terminates then C must terminate. □

3.4 ET0L systems

Definition 3.22. *[10] (E stands for extended, T for tables, 0 (zero) for no context, and L for Aristid Lindenmeyer, the Hungarian biologist that was the originator of this kind of generating mechanism).*

An ET0L system is a quadruple $G = (N, \Sigma, R, w)$, where Σ is a nonempty subset of the alphabet N, R is a finite set of tables, $R = \{R_1, \ldots, R_k\}$, where each R_i, for $1 \leq i \leq k$, is a finite subset of $N \times N^*$ which satisfies the condition that for each $a \in N$ there is a word $w_a \in N^*$ such that $(a, w_a) \in R_i$. We write $a \to w_a$ if $(a, w_a) \in R_i$. When there is no rule for $a \in N$ in R_i, then the rule $a \to a$ is included. The string $w \in N^+$ is the starting string.

For $x, y \in N^*$ we say x derives y in G and we write $x \Rightarrow y$ if the following holds:

- *$x = x_1x_2 \ldots x_m$, with $x_i \in N$ for $1 \leq i \leq m$;*
- *$y = y_1y_2 \ldots y_m$, with $y_i \in N^*$ for $1 \leq i \leq m$;*
- *there is a table $R_k \in R$ such that $x_i \to y_i \in R_k$ for all i with $1 \leq i \leq m$.*

The language generated by G is defined to be $L(G) = \{x \in \Sigma^* \mid w \Rightarrow^* x\}$,

where $\Rightarrow^*$ *is the transitive and reflexive closure of* $\Rightarrow$.

An ET0L system is deterministic if for each table $R_i \in R$ we have precisely one rule $a \to w \in R_i$ for each $a \in N$. We denote the class of such generating mechanisms by EDT0L, where the D is short for deterministic. The class of ET0L systems for which $N = \Sigma$, is denoted by T0L, and the class of deterministic T0L systems by DT0L. Note that a T0L system is defined by a triple (N, R, w). An ET0L system is λ-free if there are no rules of the form $a \to \lambda$ in any table. The class of Lindenmayer systems with a single table is denoted by E0L, ED0L, 0L, and D0L respectively.

The classes of languages corresponding to Lindenmayer systems are denoted by $\mathcal{L}$(ET0L), $\mathcal{L}$(EDT0L), $\mathcal{L}$(T0L), $\mathcal{L}$(DT0L), etc.

The main difference between these systems and phrase structure grammars is the use of parallelism in the derivations in Lindenmayer systems. In Lindenmayer systems we replace all the symbols of sentential forms in parallel during a derivation. There is also no distinction between terminals and nonterminals in Lindenmayer systems, but a subset Σ of the alphabet N is identified as the set of output symbols. Derivations in a Lindenmayer system also start with a word in N^+, instead of a single nonterminal symbol, but this difference has no real effect on the generating power of Lindenmayer systems.

Example 3.11. Our first example of an ET0L system is from [11]. We consider an EDT0L grammar $G = (N, \Sigma, R, S)$, with $N = \{A, B, C, D, E, F, r, l, u, d\}$, $\Sigma = \{r, l, u, d\}$, A the initial non-terminal and $R = \{R_1, R_2\}$. The table R_1 contains rules: $A \to BAF$, $B \to ABD$, $C \to DCE$, $D \to CDB$, $E \to FEC$, $F \to EFA$. The table R_2 contains rules: $A \to rurd$, $B \to ldlu$, $C \to urul$, $D \to dldr$, $E \to rulu$, $F \to ldrd$.

We interpret r, l, u, and d as lines of unit length in the directions, right, left, up, and down respectively and ru, for example, is the concatenation of two line segments where the line that is going up starts at the end point of the line going to the right. After replacing A, B, C, D, E and F by the right-hand sides of the rules in R_2, and interpreting the strings as line drawings, we obtain the drawings in Figure 3.1.

Figure 3.1: After replacing A, B, C, D, E and F by making use of the rules in R_2 and interpreting the strings as line drawings

The first few steps of a derivation are given next:

$A \Rightarrow BAF \Rightarrow ABDBAFEFA \Rightarrow BAFABDCDBABDBAFEFAFECEFABAF.$
Using the package Treebag [12] we obtain the line drawings shown in Figure 3.2 that correspond to this derivation.

Figure 3.2: Steps 1, 2, 3, 4 in terms of pictures

After a few steps, using Treebag, we obtain the line drawings shown in Figure 3.3. The drawings are approximations of the Sierpiński gasket.

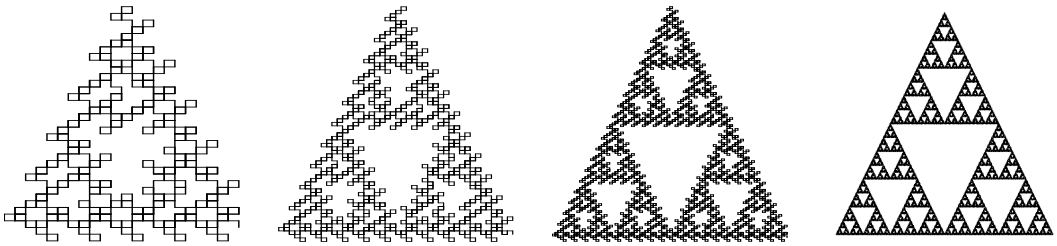


Figure 3.3: Steps 8, 9, 10, 11 in Sierpiński gasket using Treebag

We note that the Sierpiński gasket consists of three copies of itself. The rules in R_1 are related to this fact. Let X, Y and Z be the three corners of the triangle read counterclockwise, where X is the bottom left corner. Let A represent the edge XY, B represent XZ, C represent YX, D represent YZ, E represent ZX, and F represent ZY. Consider, for example, the rule $A \rightarrow BAF$. This rule shows that the edge XY in the triangle is replaced with $XZ\,XY\,ZY$ which related to a triangle has initial and end points which coincide with X and Y, respectively. Applying this to all nonterminals and with suitable substitution with R_2 we get an approximation for the Sierpiński gasket. □

Example 3.12. As a second example we consider the D0L system $G = (\{F, +, -\}, \{F \rightarrow F + F - F - FF + F + F - F\}, F + F + F + F)$ [34]. In order to express this D0L system in terms of pictures we use the concept of turtle graphics as in [11]. Informally, turtle graphics with angles (α_0, α) gives a special interpretation for symbols $F, +, -$ and $\circ$. The symbol 'F' is interpreted as a line segment of unit length from $(0, 0)$ to $(sin(\alpha_0), cos(\alpha_0))$, where α_0 is initially 90°. The symbol '+' is a rotation with angle α where, for example, $+F$ is a unit line segment rotated with α, and '$-$' is similar to '+' except the rotation is with angle $-\alpha$. Lastly, '$\circ$' stands for concatenation where, for example, $F \circ F$ concatenates the two lines where the second line starts at the end point of the first line. Usually we drop '$\circ$' and write FF.

In our example, $\alpha_0 = 90°$ and $\alpha = 90°$. Then using Treebag we find that this D0L system gives approximations to the quadratic Koch island as shown in Figure 3.4.

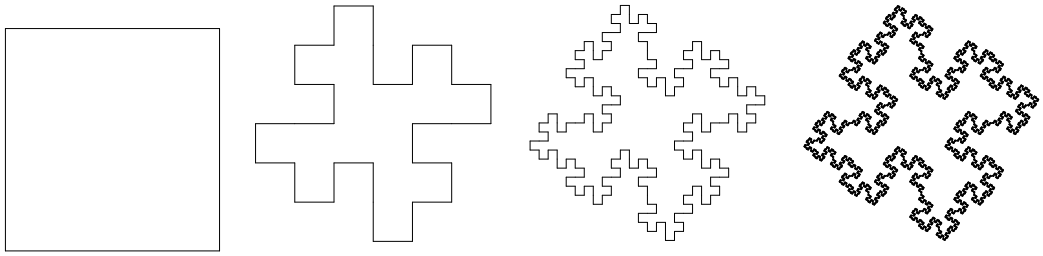


Figure 3.4: Steps 1, 2, 3, 5 in quadratic Koch island using Treebag

□

Example 3.13. As a third example we consider the D0L system $G = (\{F, +, -, [,]\}, \{F \rightarrow F[-F]F[+F][F]\}, F)$ [34]. We use here turtle graphics with angles $(0°, 25°)$. The brackets [and] are used in turtle graphics to reset the position. In other words, after drawing what is quoted with the brackets [and], the origin is reset to the end point before the open bracket [. Again the line drawings (or turtle graphics) in Figure 3.5 were obtained by interpreting the strings appropriately and using Treebag.

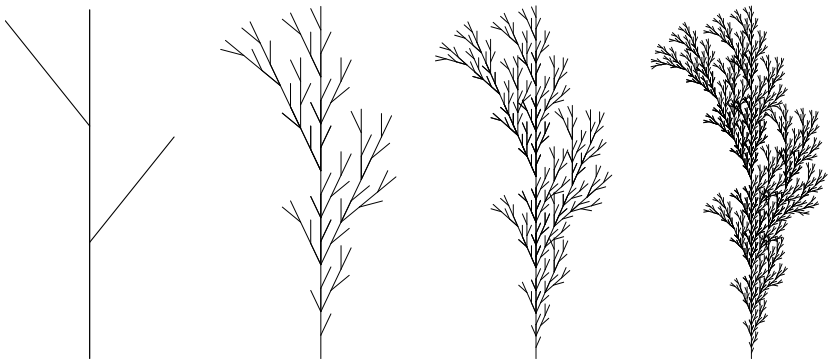


Figure 3.5: Steps 2, 4, 5, 6 in the plant generated by G using Treebag

□

3.5 Branching synchronization grammars

Before we define branching synchronization grammars in their general form, we first define a special case, namely branching ET0L grammars.

Branching ET0L grammars

Definition 3.23. *[13] A branching ET0L grammar is a tuple $G = (N, \Sigma, I, J, R, S)$, where N and Σ are disjoint alphabets of nonterminals and terminals respectively, I and J are nonempty sets of alphabets of synchronization symbols and table symbols respectively, R is the table specification where it assigns to every $\tau \in J$ a finite set $R(\tau)$ of rules $A \to \zeta$ such that $A \in N$ and $\zeta \in ((N \times I) \cup \Sigma)^*$, and $S \in N$ is the start symbol or the initial nonterminal.*

Related definitions [13]

A synchronization string is any element of I^*. A synchronized nonterminal (A, φ) is an element of $N \times I^*$, which consists of a nonterminal A and a synchronization string φ. SN_G denotes the set of all synchronized nonterminals of G. The initial synchronized nonterminal is (S, λ).

Derivation steps [13]

For every $(A, \varphi) \in SN_G$ and every rule $r = A \to v_0(B_1, \alpha_1)v_1 \dots (B_l, \alpha_l)v_l$, where $l \in \mathbb{N}$, $A, B_1, \dots, B_l \in N$, $v_0, \dots, v_l \in \Sigma^*$, and $\alpha_1, \dots \alpha_l \in I$, we define a derivation step $(A, \varphi) \Rightarrow_r v_0(B_1, \varphi\alpha_1)v_1 \dots (B_l, \varphi\alpha_l)v_l$. Note that the nonterminals in the right hand side concatenate their synchronization symbols to the synchronization string of A.

To define a derivation step in general, consider $\xi_1, \xi_2 \in (SN_G \cup \Sigma)^*$, where $\xi_1 = w_0(A_1, \varphi_1)w_1 \dots (A_h, \varphi_h)w_h$, for some $(A_i, \varphi_i) \in SN_G$ and $w_i \in \Sigma^*$. Then there is a derivation step $\xi_1 \Rightarrow \xi_2$ if there are $\tau_1, \dots, \tau_h \in J$ and rules $r_1 \in R(\tau_1), \dots, r_h \in R(\tau_h)$ such that:

- $\xi_2 = w_0\zeta_1w_1 \dots \zeta_hw_h$, where $(A_i, \varphi_i) \Rightarrow_{r_i} \zeta_i$ for $1 \le i \le h$, and
- for $1 \le i, j \le h$, $\varphi_i = \varphi_j$ implies $\tau_i = \tau_j$.

The language generated by G is defined to be $L(G) = \{w \in \Sigma^* \mid (S, \lambda) \Rightarrow^* w\}$.

We may redefine the ET0L systems as follows (see [13]): an ET0L system is a tuple $G = (N, \Sigma, J, R, S)$, where N and Σ are disjoint alphabets of nonterminals and terminals respectively, J is a nonempty alphabet of table symbols, R is the table specification where it assigns to every $\tau \in J$ a finite set $R(\tau)$ of rules $A \to \zeta$ such that $A \in N$ and $\zeta \in (N \cup \Sigma)^*$, and $S \in N$ is the start symbol.

Let $\xi_1, \xi_2 \in (N \cup \Sigma)^*$ where $\xi_1 = w_0A_1w_1 \dots A_hw_h$, $w_0, \dots, w_h \in \Sigma^*$, and $A_1, \dots, A_h \in N$.

Then there is a derivation step $\xi_1 \Rightarrow \xi_2$ if there is some $\tau \in J$ such that $R(\tau)$ contains rules $A_1 \to \zeta_1, \ldots, A_h \to \zeta_h$ and $\xi_2 = w_0\zeta_1 w_1 \ldots \zeta_h w_h$.

From the above discussion we notice that ET0L systems are a special case of branching ET0L systems with $|I| = 1$.

Branching synchronization grammars

Definition 3.24. *[13] Let $n \in \mathbb{N}$ (n is called the nesting depth). A grammar with branching synchronization and nested tables is a tuple $G = (N, \Sigma, I, J, R, S)$, where N, Σ, I, J, S are as in branching ET0L systems, and R is the table specification that assigns to every $\tau \in J^n$ a finite set $R(\tau)$ of rules $A \to \zeta$, such that $A \in N$ and $\zeta \in ((N \times I^n) \cup \Sigma)^*$.*

Related definitions [13]

Let $x = (x_1, \ldots, x_n) \in X^n$, where X is any set and $n \in \mathbb{N}$. We define $first_k(x)$ to be equal to $(x_1, \ldots, x_k)$, where $0 \leq k \leq n$. We extend the definition of $first_k$ from X^n to $(X^n)^*$ as follows: for $s = \gamma_1 \ldots \gamma_m$ with $\gamma_i \in X^n$ we define $first_k(s)$ to be $first_k(\gamma_1) \ldots first_k(\gamma_m)$. If $s, s' \in (X^n)^*$ with $|s| = |s'|$, then we define $level(s, s')$ to be equal to $max\{k \in \{0, \ldots, n\} \mid first_k(s) = first_k(s')\}$.

Let $\tau \in J^k$. A supertable $R(\tau)$ at nesting depth k for $0 \leq k \leq n$ is obtained by taking the union $\bigcup\{R(\tau') \mid \tau' \in J^n \text{ and } first_k(\tau') = \tau\}$. In particular, when $k = n$ then $R(\tau)$ contains the actual tables $R(\tau)$ where $\tau \in J^n$. Also, $R()$ is the unique table of depth 0. It contains all the rules of G and is simply denoted by R.

A synchronization string is an element of $(I^n)^*$. A synchronized nonterminal of G is an element of $SN_G = N \times (I^n)^*$. It consists of a nonterminal and synchronization string. The initial synchronized nonterminal is (S, λ).

Derivation steps [13]

For every $(A, \varphi) \in SN_G$ and every rule $r = A \to v_0(B_1, \alpha_1)v_1 \ldots (B_l, \alpha_l)v_l$ with $(B_i, \alpha_i) \in N \times I^n$ and $v_i \in \Sigma^*$ we define the derivation step $(A, \varphi) \Rightarrow_r v_0(B_1, \varphi\alpha_1)v_1 \ldots (B_l, \varphi\alpha_l)v_l$.

Consider two strings $\xi_1, \xi_2 \in (SN_G \cup \Sigma)^*$, where $\xi_1 = w_0(A_1, \varphi_1)w_1 \ldots (A_h, \varphi_h)w_h$, for some $(A_i, \varphi_i) \in SN_G$ and $w_i \in \Sigma^*$. Then there is a derivation step $\xi_1 \Rightarrow \xi_2$ if there are tables $\tau_1, \ldots, \tau_h \in J^n$ and rules $r_1 \in R(\tau_1), \ldots, r_h \in R(\tau_h)$ such that:

- $\xi_2 = w_0\zeta_1 w_1 \dots \zeta_h w_h$, where $(A_i, \varphi_i) \Rightarrow_{r_i} \zeta_i$ for $1 \leq i \leq h$, and
- $level(\tau_i, \tau_j) \geq level(\varphi_i, \varphi_j)$, for all $1 \leq i, j \leq h$.

The language generated by G is defined to be $L(G) = \{w \in \Sigma^* \mid (S, \lambda) \Rightarrow^* w\}$.

The class of all languages that can be generated by branching synchronization grammars of nesting depth n is denoted by BS_n, and $BS = \bigcup_{n\in\mathbb{N}} BS_n$.

Note that branching ET0L grammars are a special case of the branching synchronization grammars with nesting depth 1.

Example 3.14. Consider the language $L_k = \{x\$x \mid x \in D_{2k}\}$, where D_{2k} is the Dyck language composed of strings of balanced parentheses of k distinct pairs of parentheses. The language D_{2k} is generated by the context-free grammar $G_1 = (\{A\}, \{a_1, \dots, a_k, b_1, \dots, b_k\}, R, A)$ where R contains the rules:

$$A \to a_i A b_i A \mid \varepsilon, \qquad 1 \leq i \leq k.$$

But context-free grammars cannot generate the language L_k (this follows from the pumping lemma), and it seems the ET0L systems also cannot.

In [1], Atcheson, Ewert, and Shell gave a random context grammar, with context-free rules, that generates L_1. The language L_1 was conjectured by Dassow and Păun in [10] to be context sensitive but not random context. The result by Atcheson, Ewert, and Shell thus shows that this is not the case.

Next we show that a branching ET0L system with two synchronization symbols and $k + 2$ table symbols can generate L_k. Let $G = (\{S, A\}, \{a_1, \dots, a_k, b_1, \dots, b_k, \$\}, \{0, 1\}, \{0, 1, \dots, k+1\}, R, S)$, where R contains the following tables:

$$\begin{aligned} R(0) &= \{S \to A(0)\$A(0)\}, \\ R(i) &= \{A \to a_i A(0) b_i A(1)\} \qquad 1 \leq i \leq k, \\ R(k+1) &= \{A \to \varepsilon\}. \end{aligned}$$

To see how the derivation works in G, consider a string $w = a_1 a_2^2 b_2^2 b_1 \$ a_1 a_2^2 b_2^2 b_1$. The string w is generated as follows:

$$
\begin{aligned}
S &\Rightarrow_0 && A(0)\$A(0)\\
&\Rightarrow_1 && a_1A(00)b_1A(01)\$a_1A(00)b_1A(01)\\
&\Rightarrow_{k+1} && a_1A(00)b_1\$a_1A(00)b_1\\
&\Rightarrow_2 && a_1a_2A(000)b_2A(001)b_1\$a_1a_2A(000)b_2A(001)b_1\\
&\Rightarrow_{k+1} && a_1a_2A(000)b_2b_1\$a_1a_2A(000)b_2b_1\\
&\Rightarrow^*_{2,k+1} && a_1a_2^2A(0000)b_2^2b_1\$a_1a_2^2A(0000)b_2^2b_1\\
&\Rightarrow_{k+1} && a_1a_2^2b_2^2b_1\$a_1a_2^2b_2^2b_1
\end{aligned}
$$

where $\Rightarrow_i$ indicates the use of table $R(i)$, $0 \leq i \leq k+1$, in the derivation step.

In general to generate n copies of D_{2k}, we use the table $\{S \to A(0)\$A(0)\$\ldots\$A(0)\}$ as the first table, with n copies of $A(0)$ in the right-hand side of the rule. □

3.6 Regular tree grammars

First we present the notation that is required for tree grammars.

Notation: A set of terminals is a finite set of symbols $\Sigma = \bigcup_{k\in\mathbb{N}} \Sigma^{(k)}$, where the symbols in $\Sigma^{(k)}$ are said to have rank k. We assume that only finitely many of the $\Sigma^{(k)}$'s are non-empty. If a symbol f is in $\Sigma^{(k)}$, we can denote this by writing f as $f^{(k)}$. The set of all trees over Σ is denoted by T_Σ and is defined inductively as follows. It is the smallest set of strings containing Σ and $f[t_1,\ldots,t_k]$ for every $f \in \Sigma^{(k)}$, and all $t_1,\ldots,t_k \in T_\Sigma$. The yield of $t = f[t_1,\ldots,t_k] \in T_\Sigma$ is defined recursively as follows:

$$
yield(t) = \begin{cases} f & \text{if } k = 0 \\ yield(t_1)\ldots yield(t_k) & \text{otherwise.} \end{cases}
$$

Let $X = \{x_1, x_2, \ldots\}$ be a set of distinct symbols, all of rank zero that is disjoint with every other alphabet used here. If $t \in T_\Sigma(X)$ for some arbitrary alphabet Σ, then we denote by $t[\![t_1,\ldots,t_k]\!]$ the tree that results when each occurrence of x_i in t is replaced by t_i for $1 \leq i \leq k$. We denote the subset $\{x_1,\ldots,x_m\}$ of X by X_m.

Regular tree grammars

Definition 3.25. *[15, 16] A regular tree grammar is a quadruple $G = (N, \Sigma, R, S)$, where N is a finite set of nonterminals of rank zero, Σ is a set of ranked terminals with $N \cap \Sigma = \emptyset$, R is a finite set of rules of the form $A \to t$ with $A \in N$ and $t \in T_\Sigma(N)$, and $S \in N$ is the start symbol.*

Let $t, t' \in T_{\Sigma \cup N}$. There is a derivation step from t to t' and we write $t \Rightarrow t'$ if $t = s[\![A]\!]$ and $t' = s[\![r]\!]$, where $s \in T_{\Sigma \cup N}(X_1)$ contains x_1 exactly once and there is a rule $A \to r$ in R.

The tree language generated by G is defined to be the set $L(G) = \{t \in T_\Sigma \mid S \Rightarrow^* t\}$.

Example 3.15. As an example, consider the regular tree language over the ranked alphabet $\Sigma = \{a^{(0)}, b^{(0)}, p^{(1)}, q^{(2)}\}$ that contains all the trees over Σ which have yield of the form a^+b^+ (Exercise 3.20 in [16]). This language is generated by the regular tree grammar $G = (N, \Sigma, R, S)$ where $N = \{S, A, B\}$, and R contains the rules below:

$S \to a \mid b \mid p[S] \mid q[A, S] \mid q[A, B] \mid q[S, B]$,
$A \to a \mid p[A] \mid q[A, A]$,
$B \to b \mid p[B] \mid q[B, B]$. □

In this chapter we introduced various kinds of generating mechanisms. We considered sequential, and parallel generating mechanisms. We defined the Chomsky hierarchy in decreasing generative power. Then we considered grammars with regulated rewriting. These grammars have rules similar to the grammars in the Chomsky hierarchy, but with regulation mechanisms in order to avoid certain derivations. After that we introduced branching synchronization grammars that use parallel instead of sequential derivation steps. Finally we defined regular tree grammars.

In the next chapter we consider various recognizing devices and discuss their relation with grammars with regulated rewriting mechanisms.

Chapter 4

Recognizing devices and bag context grammars

In this chapter we consider the relationship between counter automata, Petri nets and BCGs (bag context grammars). We introduce counter automata in general, and also blind, and partially blind counter automata. We show how to simulate k-counter automata (blind and partially blind) with k-BCGs. Then we show that regular BCGs generate the class of recursively enumerable languages. Next we define k-place Petri net automata and consider the languages associated with them. Then we show how these automata can be simulated with k-BCGs. We illustrate our definitions and results with our own examples.

4.1 Counter automata

Definition 4.1. *[22] A k-counter automaton $M = (Q, \Sigma, \delta, q_0, F)$ consists of a finite set Q of states, a designated initial state q_0, a set F of final or accepting states with $F \subseteq Q$, a finite set Σ of input symbols, and a transition function $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \{0, 1, -1\}^k \to \mathcal{P}(Q \times \{0, 1, -1\}^k)$.*

An instantaneous description (ID) of M is an element of $Q \times \Sigma^* \times \mathbb{Z}^k$. If $(q', v_1, \ldots, v_k)$ is in $\delta(q, a, u_1, \ldots, u_k)$ and $(q, aw, y_1, \ldots, y_k)$ is an ID with $sgn(u_i) = sgn(y_i)$ for $1 \leq i \leq k$, (where for an integer x $sgn(x) = 1, 0,$ or -1 if $x > 0, x = 0,$ or $x < 0$, respectively), then we write $(q, aw, y_1, \ldots, y_k) \vdash (q', w, y_1 + v_1, \ldots, y_k + v_k)$. If $a = \varepsilon$ the transition is called an ε-transition.

For any ID I we write $I \vdash^0 I$. If $ID_1 \vdash ID_2$ and $ID_2 \vdash^n ID_{n+1}$ we write $ID_1 \vdash^{n+1} ID_{n+1}$. If $ID_1 \vdash^n ID_n$ for $n \geq 0$, we call $ID_1 \vdash^n ID_n$ an n-step computation, and also write $ID_1 \vdash^* ID_n$. If $ID_1 = (q_0, w, 0, \ldots, 0)$ and $ID_n = (q, \varepsilon, 0, \ldots, 0)$ for any $q \in F$, then $ID_1 \vdash^* ID_n$ is an accepting computation for w. The language accepted by M is $L(M) = \{w \in \Sigma^* \mid M$ has an accepting computation for $w\}$.

Example 4.1. As an example of a counter automaton, consider Figure 4.1. This figure shows a counter automaton that recognizes the language of well balanced expressions over one pair of parenthesis "[" and "]". This language is denoted by D_2, and is called the one-sided Dyck language on one pair of parenthesis (see [22]). The counter automaton recognizing D_2 is formally defined by $M = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{L, R, R'\}$, $\Sigma = \{[,]\}$, $q_0 = L$, $F = \{R\}$, and δ is defined as follows:

$\delta(L, [, 1) = \delta(L, [, 0) = \{(L, 1)\}$,
$\delta(R, [, 1) = \delta(R, [, 0) = \{(L, 1)\}$,
$\delta(L,], 0) = \{(R', 0)\}$,
$\delta(L,], 1) = \delta(R,], 1) = \{(R, -1)\}$,
$\delta(R,], 0) = \{(R', 0)\}$.

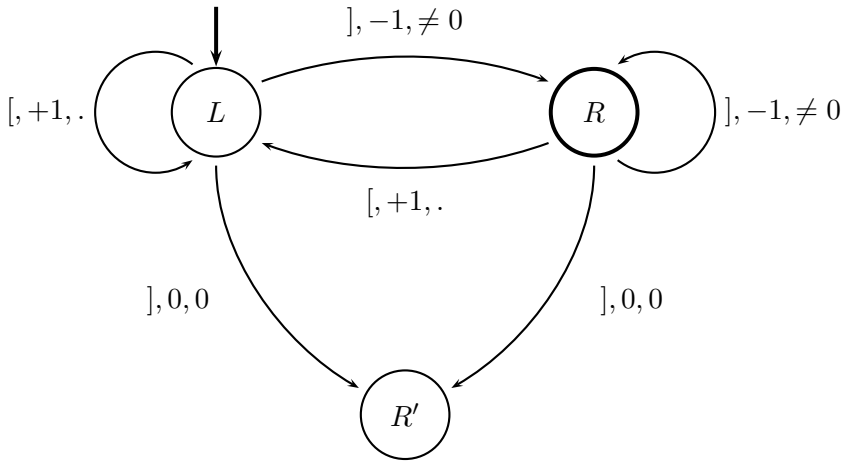


Figure 4.1: A counter automaton that recognizes D_2

An accepting computation for a string $w = [\,[\,]\,]\,[\,]$ in D_2 is given by:

$$(L, [\,[\,]\,]\,[\,], 0) \vdash (L, [\,]\,]\,[\,], 1) \vdash (L,]\,]\,[\,], 2) \vdash (R,]\,[\,], 1) \vdash (R, [\,], 0) \vdash (L,], 1) \vdash (R, \varepsilon, 0).$$

In Figure 4.1, the symbols on each arc represent the input symbol, the action on the counter ('+1' for increase, '−1' decrease, or '0' for no change), and the check for the counter value ($= 0, \neq 0$, or '.' for don't care), respectively. □

4.2 Blind and partially blind counter automata

Definition 4.2. *[22] A k-counter automaton $M = (Q, \Sigma, \delta, q_0, F)$ is called blind if for each $q \in Q$ and $a \in \Sigma \cup \{\varepsilon\}$ we have that $\delta(q, a, u_1, \ldots, u_k) = \delta(q, a, v_1, \ldots, v_k)$ for all $u_i, v_i \in$*

$\{0, 1, -1\}$. *A k-counter automaton is partially blind if for all* $q \in Q$ *and* $a \in \Sigma \cup \{\varepsilon\}$ *we have that* $\delta(q, a, u_1, \ldots, u_k) = \emptyset$ *whenever any* u_i *is* -1, *and* $\delta(q, a, u_1, \ldots, u_k) = \delta(q, a, v_1, \ldots, v_k)$ *for all* $u_i, v_i \in \{0, 1\}$.

Informally, a counter automaton is blind if its counters cannot be checked until the end of the computation. It is called partially blind if it cannot be checked whether counters are 0 or positive until the end of the computation, and on decrementing a zero counter the automaton blocks the remainder of the computation.

In the case of blind and partially blind counter automata, counter values have no effect on transitions. Thus, in this case we consider the transition function as a function from $Q \times (\Sigma \cup \{\varepsilon\})$ to $\mathcal{P}(Q \times \{0, 1, -1\}^k)$.

Example 4.2. A partially blind counter automaton that recognizes D_2 is shown in Figure 4.2, and is formally described by $M = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{L, R\}$, $\Sigma = \{[,]\}$, $q_0 = L$, $F = \{R\}$, and δ is defined as follows:

$\delta(L, [\,) = \delta(R, [\,) = \{(L, 1)\}$,
$\delta(L,]\,) = \delta(R,]\,) = \{(R, -1)\}$.

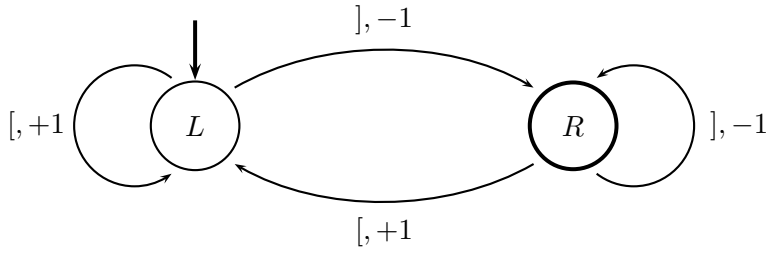
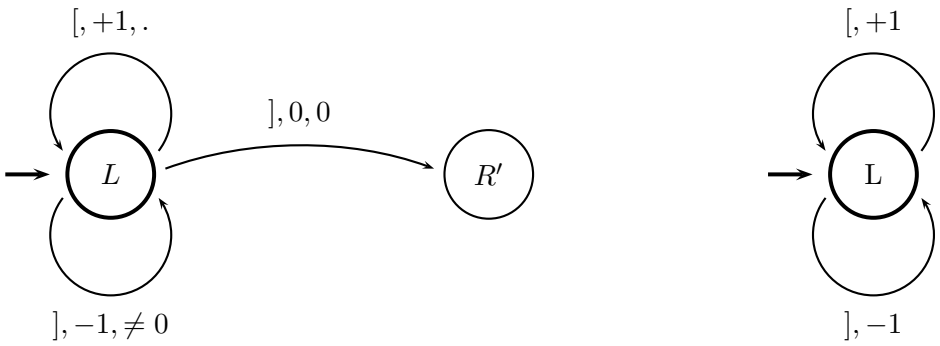


Figure 4.2: A partially blind counter automaton that recognizes D_2

□

We notice that our automata in both cases do not accept the empty string. In order to accept the empty string, we simply add L to the set of the final states. Since it is always interesting to use minimum number of states then Figures 4.1 and 4.2 can be simplified by the following figures:



We will see later in Chapter 4 and 6 that the nonterminals complexity of a BCG that simulates counter automaton depends on the number of states of this automaton.

It has been shown by Greibach in [22] that no blind counter automata (even with an arbitrary number of counters) can recognize the one-sided Dyck language over one pair of parenthesis. Thus, blind counter automata are strictly weaker than the partially blind.

Example 4.3. We consider as an example of blind counter automata, the automaton shown in Figure 4.3, which recognizes the language $L = \{a^i b^j c^{i+j} \mid i, j \geq 0\}$. It is described by $M = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{A, B, C\}$, $\Sigma = \{a, b, c\}$, $q_0 = A$, $F = \{A, C\}$, and δ is defined as follows:

$\delta(A, a) = \{(A, 1)\}$,
$\delta(A, b) = \delta(B, b) = \{(B, 1)\}$,
$\delta(A, c) = \delta(B, c) = \delta(C, c) = \{(C, -1)\}$.

An accepting computation for a string $w = a^2b^2c^4$ is given by:

$$(A, a^2b^2c^4, 0) \vdash^* (A, b^2c^4, 2) \vdash^* (B, c^4, 4) \vdash^* (C, \varepsilon, 0).$$

Note that the counter of M increases with each input a or b and decreases with each input c. Thus, the counter becomes empty if $\#c = \#a + \#b$ where, for example, $\#a$ indicates the number of a's. □

4.3 Counter automata and bag context grammars

In this section we show how to simulate counter automata with BCGs. As a result we show that any recursively enumerable language can be generated by a regular BCG. In a regular

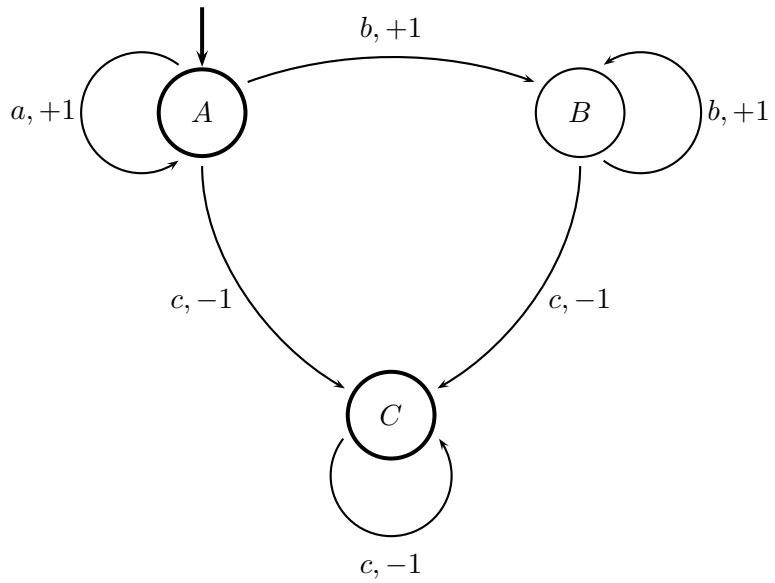


Figure 4.3: A blind counter automaton that recognizes $\{a^i b^j c^{i+j} \mid i, j \geq 0\}$

BCG all rules are, with the exception of the bag conditions, of the same form as in a regular grammar.

Theorem 4.1. *[26] Any recursively enumerable language can be recognized by a 2-counter automaton.* ■

Theorem 4.2. *For any 1-counter automaton M, there exists a regular 1-BCG G such that $L(M) = L(G)$.*

Proof: Let $M = (Q, \Sigma, \delta, q_0, F)$ be a 1-counter automaton. We construct a regular 1-BCG $G = (Q, \Sigma, R, q_0, 0)$, such that $L(G) = L(M)$.

- For all $q_r, q_s \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, and $x, y \in \{0, 1, -1\}$, if $(q_s, y) \in \delta(q_r, a, x)$, add to R:

  $q_r \to aq_s \quad (0, 0; y) \qquad$ if $x = 0$
  $q_r \to aq_s \quad (1, \infty; y) \qquad$ if $x = 1$
  $q_r \to aq_s \quad (-\infty, -1; y) \quad$ if $x = -1$

- For each state $q_s \in F$, add to R: $q_s \to \varepsilon \quad (0, 0; 0)$.

Next we show that $L(M) \subseteq L(G)$. Consider a word $w = a_0 a_1 \dots a_n \in L(M)$, $n \geq 0$, and $a_i \in \Sigma \cup \{\varepsilon\}$, $0 \leq i \leq n$. We show that $w \in L(G)$. Since $w \in L(M)$, there is an accepting

computation for w in M given by:

$$(q_0, a_0a_1 \dots a_n, 0) \vdash (q_1, a_1 \dots a_n, c_1) \vdash \cdots \vdash (q_n, a_n, c_n) \vdash (q_f, \varepsilon, 0),$$

with $q_i \in Q$ and $c_i \in \mathbb{Z}$ for $1 \leq i \leq n$, and $q_f \in F$.

The accepting computation of w implies that: $(q_1, c_1) \in \delta(q_0, a_0, 0), (q_2, c_2 - c_1) \in \delta(q_1, a_1, sgn(c_1)), \dots, (q_n, c_n - c_{n-1}) \in \delta(q_{n-1}, a_{n-1}, sgn(c_{n-1}))$, and $(q_f, -c_n) \in \delta(q_n, a_n, sgn(c_n))$. From the way we constructed G, we have the following rules in R:

$$\begin{array}{ll}
q_0 \to a_0q_1 & (0, 0; c_1), \\
q_1 \to a_1q_2 & (x_1, y_1, c_2 - c_1), \\
\vdots & \vdots \\
q_{n-1} \to a_{n-1}q_n & (x_{n-1}, y_{n-1}, c_n - c_{n-1}), \\
q_n \to a_nq_f & (x_n, y_n, -c_n), \\
q_f \to \varepsilon & (0, 0; 0),
\end{array}$$

where $x_i = y_i = 0$ if $sgn(c_i) = 0$, $x_i = 1$ and $y_i = \infty$ if $sgn(c_i) = 1$, and $x_i = -\infty$ and $y_i = -1$ if $sgn(c_i) = -1$.

If we start from $(q_0, 0)$ we have the following derivation:

$$(q_0, 0) \Rightarrow (a_0q_1, c_1) \Rightarrow (a_0a_1q_2, c_2) \Rightarrow \cdots \Rightarrow (a_0a_1 \dots a_n, 0).$$

This shows that $w \in L(G)$ and thus that $L(M) \subseteq L(G)$.

The reverse containment can be obtained in a similar way. We thus conclude that $L(G) = L(M)$. ■

In the previous theorem we showed how to simulate an accepting computation of a 1-counter automaton with a derivation in a regular 1-BCG. One can generalize the argument to show that an accepting computation of a k-counter automaton can be simulated by a derivation in a regular k-BCG.

Corollary 4.3. *For each k-counter automaton M, there exists a regular k-BCG G such that $L(G) = L(M)$.* ■

Corollary 4.4. *Any recursively enumerable language can be generated by a regular 2-BCG.*

Proof: The proof of this corollary follows from Theorem 4.1 and Corollary 4.3. ■

In case of blind and partially blind 1-counter automata we construct a regular 1-BCG in a similar way, except if $(q_s, y) \in \delta(q_r, a)$ we add the rule $q_r \to aq_s\,(-\infty, \infty; y)$ for the blind

automaton, since counter values have no effect on transitions. For the partially blind we add $q_r \rightarrow aq_s\,(0, \infty; y)$, since transitions occur on positive or zero counter values.

Note that in case of a BCG simulating a partially blind counter automata, we have that a derivation is blocked if the bag becomes negative.

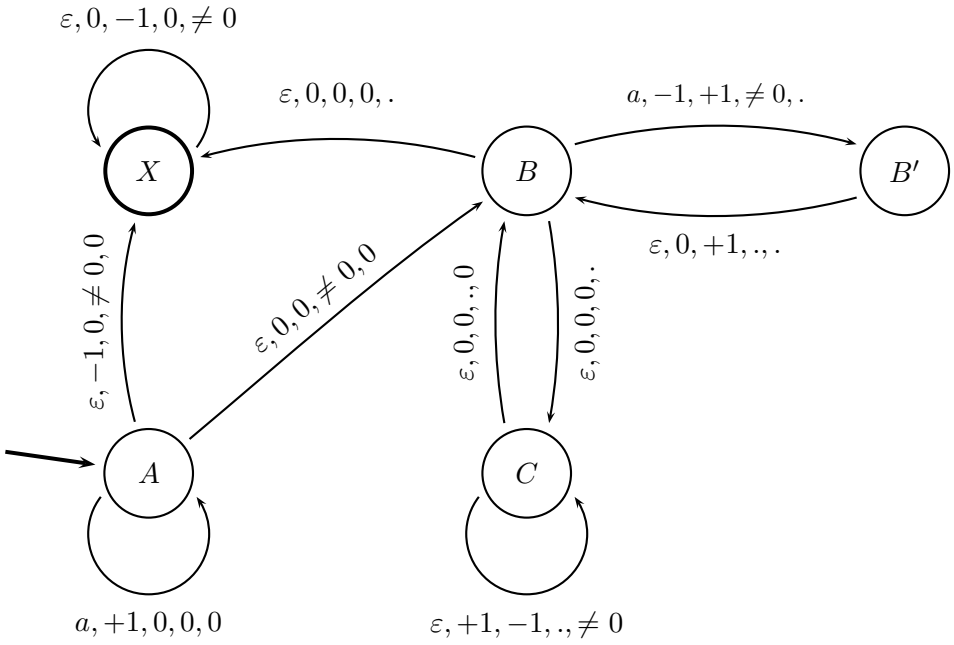


Figure 4.4: A 2-counter automaton that recognizes $\{a^{2^n} \mid n \geq 0\}$

Example 4.4. To illustrate Theorem 4.2, consider the 2-counter automaton shown in Figure 4.4. This automaton recognizes the language $L = \{a^{2^n} \mid n \geq 0\}$ and is described by $M = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{A, B, B', C, X\}$, $\Sigma = \{a\}$, $q_0 = A$ and $F = \{X\}$, and the transition function is defined as follows:

Next we give the transition function. The transition function is given in terms of one-step computations. Each one-step computation is also given a label. Note that we use for example the labels t_{2_1} and t_{2_2} to label $(A, \varepsilon, 1, 0) \vdash (X, -1, 0)$ and $(A, \varepsilon, 1, 0) \vdash (B, 0, 0)$ respectively, since the cardinality of $\delta(A, \varepsilon, 1, 0)$ is two.

$t_1 : (A, a, 0, 0) \vdash (A, 1, 0)$,
$t_{2_1} : (A, \varepsilon, 1, 0) \vdash (X, -1, 0)$,
$t_{2_2} : (A, \varepsilon, 1, 0) \vdash (B, 0, 0)$,
$t_3 : (B, a, 1, 0) \vdash (B', -1, 1)$,
$t_4 : (B, a, 1, 1) \vdash (B', -1, 1)$,
$t_5 : (B', \varepsilon, 0, 1) \vdash (B, 0, 1)$,
$t_6 : (B', \varepsilon, 1, 1) \vdash (B, 0, 1)$,
$t_{7_1} : (B, \varepsilon, 0, 1) \vdash (X, 0, 0)$,
$t_{7_2} : (B, \varepsilon, 0, 1) \vdash (C, 0, 0)$,
$t_8 : (X, \varepsilon, 0, 1) \vdash (X, 0, -1)$,
$t_9 : (C, \varepsilon, 0, 1) \vdash (C, 1, -1)$,
$t_{10} : (C, \varepsilon, 1, 1) \vdash (C, 1, -1)$,
$t_{11} : (C, \varepsilon, 1, 0) \vdash (B, 0, 0)$.

Before we construct the equivalent regular 2-BCG G, we give an intuitive description of the given counter automaton M. Let C_1 and C_2 be the first and second counters respectively. The initial state of M is A. If M receives 'a' as input, it stays in state A and increases C_1 by 1. The automaton M accepts the input 'a' by moving to state X.

If the input string contains more than one 'a', the automaton moves from state A to B during an accepting computation, after reading the first 'a'. Further note that if C_1 is 0 in state B, then a prefix of length 2^m, for $m \geq 1$, has been processed, and the value of C_2 will also be 2^m.

Finally note that state C is used to copy the value of C_2 to C_1 and to set the value of C_2 equal to 0.

Next we apply the procedure given in Theorem 4.2 to construct a regular 2-BCG $G = (N, \Sigma, R, A, (0, 0))$ that generates the language $L = \{a^{2^n} \mid n \geq 0\}$. In this bag context grammar $N = \{A, B, B', C, X\}$, $\Sigma = \{a\}$ and R is given by:

$r_1 : A \to aA$ $((0,0),(0,0);(1,0))$,
$r_2 : A \to X$ $((1,0),(\infty,0);(-1,0))$,
$r_3 : A \to B$ $((1,0),(\infty,0);(0,0))$,
$r_4 : B \to aB'$ $((1,0),(\infty,\infty);(-1,1))$,
$r_5 : B' \to B$ $((0,1),(\infty,\infty);(0,1))$,
$r_6, r_7 : B \to X \mid C$ $((0,1),(0,\infty);(0,0))$,
$r_8 : X \to X$ $((0,1),(0,\infty);(0,-1))$,
$r_9 : C \to C$ $((0,1),(\infty,\infty);(1,-1))$,
$r_{10} : C \to B$ $((1,0),(\infty,0);(0,0))$,
$r_{11} : X \to \varepsilon$ $((0,0),(0,0);(0,0))$.

Consider the string $w = a^4 \in L$. The string w has an accepting computation in M given by:

$(A, a^4, 0, 0) \vdash_{t_1} (A, a^3, 1, 0) \vdash_{t_{2_2}} (B, a^3, 1, 0) \vdash_{t_3} (B', a^2, 0, 1) \vdash_{t_5} (B, a^2, 0, 2) \vdash_{t_{7_2}} (C, a^2, 0, 2)$
$\vdash^*_{t_9,t_{10}} (C, a^2, 2, 0) \vdash_{t_{11}} (B, a^2, 2, 0) \vdash_{t_3} (B', a, 1, 1) \vdash_{t_6} (B, a, 1, 2) \vdash_{t_4} (B', \varepsilon, 0, 3) \vdash_{t_5} (B, \varepsilon, 0, 4)$
$\vdash_{t_{7_1}} (X, \varepsilon, 0, 4) \vdash^*_{t_8} (X, \varepsilon, 0, 0)$.

In this computation, $ID_1 \vdash_{t_k} ID_2$ indicates a transition from instantaneous description ID_1 to ID_2 by applying rule t_k.

The corresponding derivation of w in G is given by:

$A_{(0,0)} \Rightarrow_{r_1} aA_{(1,0)} \Rightarrow_{r_3} aB_{(1,0)} \Rightarrow_{r_4} a^2B'_{(0,1)} \Rightarrow_{r_5} a^2B_{(0,2)} \Rightarrow_{r_7} a^2C_{(0,2)} \Rightarrow^*_{r_9} a^2C_{(2,0)} \Rightarrow_{r_{10}}$
$a^2B_{(2,0)} \Rightarrow_{r_4} a^3B'_{(1,1)} \Rightarrow_{r_5} a^3B_{(1,2)} \Rightarrow_{r_4} a^4B'_{(0,3)} \Rightarrow_{r_5} a^4B_{(0,4)} \Rightarrow_{r_6} a^4X_{(0,4)} \Rightarrow^*_{r_8} a^4X_{(0,0)} \Rightarrow_{r_{11}}$
$a^4_{(0,0)}$. □

4.4 Petri nets

Petri nets constitute an important graphical and mathematical modelling tool that is used to model concurrent systems. In this section we consider the formal language aspects of Petri nets. We define Petri net automata and extend the definition of [22] to allow ε-transitions. We consider the languages associated with Petri net automata and show how to generate these languages with BCGs (bag context grammars).

Definition 4.3. *[22, 10] A k-place Petri net automaton $M = (P, \Sigma, T, F, m_0)$ consists of a finite set of places $P = \{p_1, \ldots, p_k\}$, a finite set of input symbols Σ, a finite set of labelled transitions $T \subseteq \mathbb{N}_0^k \times (\Sigma \cup \{\varepsilon\}) \times \mathbb{N}_0^k$ ($\mathbb{N}_0 = \mathbb{N} \cup \{0\}$), a set F of final places with $F \subseteq P$, and an initial marking $m_0 \in \mathbb{N}_0^k$.*

An instantaneous description (ID) of M is a member of $\Sigma^* \times \mathbb{N}_0^k$. In the instantaneous

description $(w, n_1, \ldots, n_k)$*, we call* w *the input to be processed and* n_i *the number of tokens in place* p_i*. If* $t = (u_1, \ldots, u_k, a, v_1, \ldots, v_k) \in T$*,* $a \in \Sigma \cup \{\varepsilon\}$ *and* $(aw, y_1, \ldots, y_k) \in ID$ *such that* $u_i \leq y_i$ *for all* $i, 1 \leq i \leq k$*, then we say that* t *is enabled. Under these circumstances we have the transition* $(aw, y_1, \ldots, y_k) \vdash (w, (y_1 - u_1) + v_1, \ldots, (y_k - u_k) + v_k)$*. We describe this transition as "firing* t*". As usual we let* $\vdash^*$ *be the transitive and reflexive closure of* $\vdash$*.*

A marking m *for* M *is an element in* $\mathbb{N}_0^k$ *where the* i^{th} *component of* m *represents the number of tokens in place* p_i*. Let* $t = (u_1, \ldots, u_k, a, v_1, \ldots, v_k) \in T$ *be an enabled transition at a marking* $m = (y_1, \ldots, y_k)$*. We define a transition function* δ *that gives the next marking of* M *after firing* t *as follows:* $\delta(m, t) = ((y_1 - u_1) + v_1, \ldots, (y_k - u_k) + v_k)$*. This definition can be extended from* T *to* T^* *as follows:* $\delta(m, \varepsilon) = m$ *for all markings* m*, and* $\delta(m, wt) = \delta(\delta(m, w), t)$ *for all markings* m*,* $t \in T$*, and* $w \in T^*$*.*

We consider here the following languages that are associated with Petri nets as in [10]:

- $L_m(M) = \{\delta(m_0, w) \mid w \in T^*\}$. This language contains the set of all markings of M that can be reached from an initial marking m_0 by firing enabled transitions.
- $L_f(M) = \{w \mid w \in \Sigma^*$ such that $(w, m_0) \vdash^* (\varepsilon, m)$ for some marking m of $M\}$. This language is called the free Petri net language and consists of all strings that can cause a firing sequence in M starting with the initial marking m_0.
- $L_t(M, F_m) = \{w \mid w \in \Sigma^*$ such that $(w, m_0) \vdash^* (\varepsilon, m)$ for some marking $m \in F_m\}$, where F_m is a given set of final markings. This language is called the terminal Petri net language with respect to the set F_m of terminal markings, and consists of all input strings that transform the initial marking m_0 to a final marking m by firing enabled transitions.

We defined the languages $L_f(M)$ and $L_t(M, F_m)$ over the input alphabet Σ, in contrast to [10] where they are defined over symbols in T. Dassow and Păun in [10] considered these three languages and discussed their relation with grammars with regulated rewriting, namely pure matrix grammars (which are, informally, matrix grammars that do not distinguish between terminals and nonterminals). We consider here each of these three languages and show how BCGs can generate them.

We start with a special case of $L_t(M, F_m)$, where we consider the language $css(M)$, which is defined as follows (see [22]):

$$css(M) = \{w \mid w \in \Sigma^* \text{ such that } (w, 1, 0, \ldots, 0) \vdash^* I\},$$

where I is the ID $(\varepsilon, n_1, \ldots, n_k)$ such that $n_t = 1$ for some $p_t \in F$ and $n_i = 0$ for all $i \neq t$.

We call $(w, 1, 0, \ldots, 0) \vdash^* I$ an accepting computation of w in M, and I an accepting ID.

Note that in $L_t(M, F_m)$, when $m_0 = (1, 0, \ldots, 0)$, and $F_m = \{(0, \ldots, 0, 1, 0, \ldots, 0)$ where 1 is in the tth position for some $p_t \in F\}$, then we have $css(M)$.

In the graphical representation of Petri nets, we draw places as circles, transitions as bars, and tokens as small black circles inside places. At each transition we write the input symbol that is consumed when the transition fires. We label each arc from place to transition or transition to place with the number of tokens that are consumed or produced when that transition fires. If there is no label, then one token is consumed or produced.

Example 4.5. As an example of a Petri net automaton, consider the automaton M shown in Figure 4.5 with $css(M) = \{a^i b^j c^{2(i+j)} \mid i, j \geq 0\}$. This automaton is described by: $M = (P, \Sigma, T, F, m_0)$, where $P = \{p_1, p_2, p_3, p_4\}$, $\Sigma = \{a, b, c\}$, $F = \{p_4\}$, $m_0 = (1, 0, 0, 0)$ and T contains the following transitions:

$t_1 = (1, 0, 0, 0, a, 1, 2, 0, 0)$, $t_2 = (1, 0, 0, 0, \varepsilon, 0, 0, 1, 0)$, $t_3 = (0, 0, 1, 0, b, 0, 2, 1, 0)$,
$t_4 = (0, 0, 1, 0, \varepsilon, 0, 0, 0, 1)$, $t_5 = (0, 1, 0, 1, c, 0, 0, 0, 1)$.

Consider the string $w = a^2bc^6$ in $css(M)$. The string w has an accepting computation given by:

$(a^2bc^6, 1, 0, 0, 0) \vdash_{t_1} (abc^6, 1, 2, 0, 0) \vdash_{t_1} (bc^6, 1, 4, 0, 0) \vdash_{t_2} (bc^6, 0, 4, 1, 0) \vdash_{t_3} (c^6, 0, 6, 1, 0) \vdash_{t_4}$
$(c^6, 0, 6, 0, 1) \vdash^*_{t_5} (c, 0, 1, 0, 1) \vdash_{t_5} (\varepsilon, 0, 0, 0, 1)$.

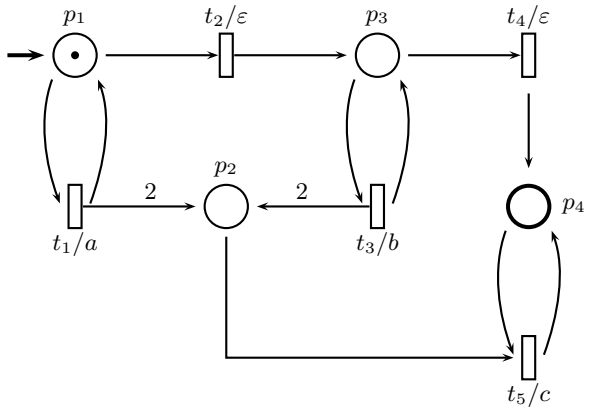


Figure 4.5: A Petri net automaton with $css = \{a^i b^j c^{2(i+j)} \mid i, j \geq 0\}$

The languages $L_m(M)$ and $L_f(M)$ are given by:

$L_m(M) = \{(1, 2n, 0, 0) \mid n \geq 0\} \cup \{(0, 2n, 1, 0) \mid n \geq 0\} \cup \{(0, n, 0, 1) \mid n \geq 0\}$,

and $L_f(M) = \{a^i b^j c^k \mid 0 \leq i, j, k, \text{ and } k \leq 2(i+j)\}$. □

4.5 Petri nets and bag context grammars

In this section we consider the relationship between Petri nets and BCGs. We show how to simulate k-place Petri net automata with k-BCGs.

Theorem 4.5. *For each k-place Petri net automaton M, there exists a regular k-BCG G with one nonterminal such that $css(M) = L(G)$.*

Proof: Let $M = (P, \Sigma, T, F, m_0)$ be a k-place Petri net automaton with $P = \{p_1, \ldots, p_k\}$ and $m_0 = (1, 0, \ldots, 0)$. We construct a regular k-BCG $G = (\{S\}, \Sigma, R, S, (1, 0, \ldots, 0))$ such that $L(G) = css(M)$ as follows:

- For each $t = (u_1, \ldots, u_k, a, v_1, \ldots, v_k) \in T$ add to R:
  $S \to aS \quad ((u_1, \ldots, u_k), \infty_k, (v_1 - u_1, \ldots, v_k - u_k))$.
- For each $p_t \in F$ add to R:
  $S \to \varepsilon \quad ((0, \ldots, 0, 1, 0, \ldots, 0), (0, \ldots, 0, 1, 0, \ldots, 0), 0_k)$, where the 1 is in the tth position.

Next we show $css(M) = L(G)$. We use a similar argument to the argument that we used to simulate a counter automaton with a BCG.

Let $w = a_0 a_1 \ldots a_n \in css(M)$ where $a_i \in \Sigma \cup \{\varepsilon\}$ for $0 \leq i \leq n$. The string w has an accepting computation in M given by:

$$(a_0 a_1 \ldots a_n, 1, 0, \ldots, 0) \vdash (a_1 \ldots a_n, v_{1_1}, \ldots, v_{1_k}) \vdash (a_2 \ldots a_n, v_{2_1}, \ldots, v_{2_k}) \vdash \ldots$$

$$\vdash (a_n, v_{n_1}, \ldots, v_{n_k}) \vdash (\varepsilon, 0, \ldots, 0, 1, 0, \ldots, 0)$$

where 1 is in the tth position.

This accepting computation implies that M has transitions:
$(1, 0, \ldots, 0, a_0, v_{1_1}, \ldots, v_{1_k})$, $(u_{1_1}, \ldots, u_{1_k}, a_1, v_{2_1} - v_{1_1} + u_{1_1}, \ldots, v_{2_k} - v_{1_k} + u_{1_k})$,
$(u_{2_1}, \ldots, u_{2_k}, a_2, v_{3_1} - v_{2_1} + u_{2_1}, \ldots, v_{3_k} - v_{2_k} + u_{2_k})$, $\ldots$,
$(u_{n_1}, \ldots, u_{n_k}, a_n, -v_{n_1} + u_{n_1}, \ldots, 1 - v_{n_t} + u_{n_t}, -v_{n_{t+1}} + u_{n_{t+1}}, \ldots, -v_{n_k} + u_{n_k})$.
In the notation above we have $v_{i_j}, u_{i_j} \in \mathbb{N}_0$ with $u_{i_j} \leq v_{i_j}$ for all i and j.

From the way we constructed G, we have the following rules in R:

$$
\begin{array}{ll}
S \to a_0 S & ((1,0,\ldots,0), \infty_k; (v_{1_1} - 1, v_{1_2}, \ldots, v_{1_k})), \\
S \to a_1 S & ((u_{1_1}, \ldots, u_{1_k}), \infty_k; (v_{2_1} - v_{1_1}, \ldots, v_{2_k} - v_{1_k})), \\
S \to a_2 S & ((u_{2_1}, \ldots, u_{2_k}), \infty_k; (v_{3_1} - v_{2_1}, \ldots, v_{3_k} - v_{2_k})), \\
\vdots & \vdots \\
S \to a_{n-1} S & ((u_{n-1_1}, \ldots, u_{n-1_k}), \infty_k; (v_{n_1} - v_{n-1_1}, \ldots, v_{n_k} - v_{n-1_k})), \\
S \to a_n S & ((u_{n_1}, \ldots, u_{n_k}), \infty_k; (-v_{n_1}, \ldots, 1 - v_{n_t}, -v_{n_{t+1}} \ldots, -v_{n_k})), \\
S \to \varepsilon & ((0,\ldots,0,1,0,\ldots,0), (0,\ldots,0,1,0,\ldots,0); 0_k).
\end{array}
$$

In the last rule we have a 1 in the tth bag position.

Then w has a derivation in G given by:
$S_{(1,0,\ldots,0)} \Rightarrow a_0 S_{(v_{1_1},\ldots,v_{1_k})} \Rightarrow a_0 a_1 S_{(v_{2_1},\ldots,v_{2_k})} \Rightarrow \cdots \Rightarrow a_0 a_1 \ldots a_{n-1} S_{(v_{n_1},\ldots,v_{n_k})} \Rightarrow$
$a_0 a_1 \ldots a_n S_{(0,\ldots,0,1,0,\ldots,0)} \Rightarrow a_0 a_1 \ldots a_{n(0,\ldots,0,1,0,\ldots,0)}$.

This shows that $w \in L(G)$ and $css(M) \subseteq L(G)$.

The reverse containment can be obtained in a similar way. Thus, we conclude that $L(G) = css(M)$. ■

Example 4.6. To illustrate Theorem 4.5 consider the Petri net automaton M shown in Figure 4.6 with $css(M) = \{a^i b^j c^i d^j \mid i, j \geq 0\}$. This automaton is formally described by: $M = (P, \Sigma, T, F, m_0)$, where $P = \{p_1, \ldots, p_6\}$, $\Sigma = \{a, b, c, d\}$, $F = \{p_6\}$, $m_0 = (1,0,0,0,0,0)$, and T contains the following transitions.

$$
\begin{array}{ll}
t_1 = (1,0,0,0,0,0,a,1,1,0,0,0,0), & t_2 = (1,0,0,0,0,0,\varepsilon,0,0,1,0,0,0), \\
t_3 = (0,0,1,0,0,0,b,0,0,1,1,0,0), & t_4 = (0,0,1,0,0,0,\varepsilon,0,0,0,0,1,0), \\
t_5 = (0,1,0,0,1,0,c,0,0,0,0,1,0), & t_6 = (0,0,0,0,1,0,\varepsilon,0,0,0,0,0,1), \\
t_7 = (0,0,0,1,0,1,d,0,0,0,0,0,1). &
\end{array}
$$

According to the proof of Theorem 4.5, the corresponding regular 6-BCG that generates the language $css(M)$ is $G = (\{S\}, \{a,b,c,d\}, R, S, (1,0,0,0,0,0))$ where R contains the rules:

$$
\begin{array}{ll}
r_1 : S \to aS & ((1,0,0,0,0,0), \infty_6; (0,1,0,0,0,0)), \\
r_2 : S \to S & ((1,0,0,0,0,0), \infty_6; (-1,0,1,0,0,0)), \\
r_3 : S \to bS & ((0,0,1,0,0,0), \infty_6; (0,0,0,1,0,0)), \\
r_4 : S \to S & ((0,0,1,0,0,0), \infty_6; (0,0,-1,0,1,0)), \\
r_5 : S \to cS & ((0,1,0,0,1,0), \infty_6; (0,-1,0,0,0,0)), \\
r_6 : S \to S & ((0,0,0,0,1,0), \infty_6; (0,0,0,0,-1,1)), \\
r_7 : S \to dS & ((0,0,0,1,0,1), \infty_6; (0,0,0,-1,0,0)), \\
r_8 : S \to \varepsilon & ((0,0,0,0,0,1), (0,0,0,0,0,1); 0_6).
\end{array}
$$

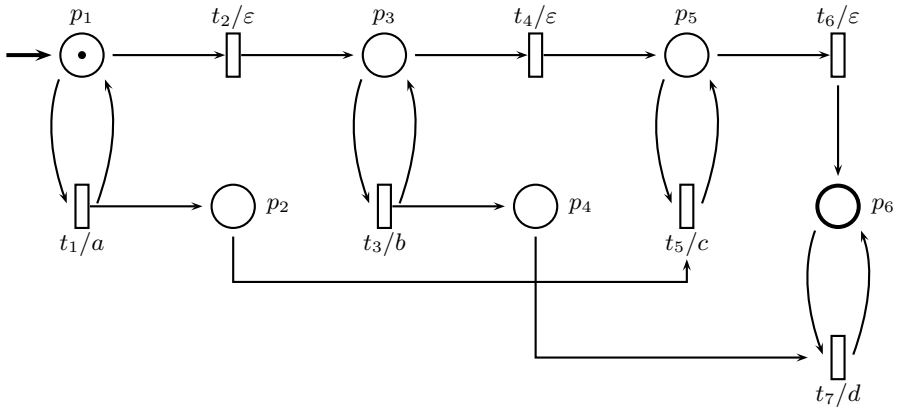


Figure 4.6: A Petri net automaton with $css = \{a^i b^j c^i d^j \mid i, j \geq 0\}$

Consider a string $w = a^2bc^2d \in css(M)$. The string w has an accepting computation given by:

$(a^2bc^2d, 1, 0, 0, 0, 0, 0) \vdash_{t_1}^* (bc^2d, 1, 2, 0, 0, 0, 0) \vdash_{t_2} (bc^2d, 0, 2, 1, 0, 0, 0) \vdash_{t_3} (c^2d, 0, 2, 1, 1, 0, 0)$
$\vdash_{t_4} (c^2d, 0, 2, 0, 1, 1, 0) \vdash_{t_5}^* (d, 0, 0, 0, 1, 1, 0) \vdash_{t_6} (d, 0, 0, 0, 1, 0, 1) \vdash_{t_7} (\varepsilon, 0, 0, 0, 0, 0, 1)$.

The corresponding derivation in G for w is given by:

$S_{(1,0,0,0,0,0)} \Rightarrow_{r_1}^* a^2 S_{(1,2,0,0,0,0)} \Rightarrow_{r_2} a^2 S_{(0,2,1,0,0,0)} \Rightarrow_{r_3} a^2 bS_{(0,2,1,1,0,0)} \Rightarrow_{r_4} a^2 bS_{(0,2,0,1,1,0)} \Rightarrow_{r_5}^*$
$a^2bc^2S_{(0,0,0,1,1,0)} \Rightarrow_{r_6} a^2bc^2S_{(0,0,0,1,0,1)} \Rightarrow_{r_7} a^2bc^2dS_{(0,0,0,0,0,1)} \Rightarrow_{r_8} a^2bc^2d_{(0,0,0,0,0,1)}$.

It can be shown that the languages $L_m(M)$ and $L_f(M)$ are given by:

$L_m(M) = \{(1, n, 0, 0, 0, 0) \mid n \geq 0\} \cup \{(0, n_1, 1, n_2, 0, 0) \mid n_1, n_2 \geq 0\} \cup \{(0, n_1, 0, n_2, 1, 0) \mid n_1, n_2 \geq 0\} \cup \{(0, n_1, 0, n_2, 0, 1) \mid n_1, n_2 \geq 0\}$,

and
$L_f(M) = \{a^i b^j c^k d^l \mid 0 \leq i, j, k, l,\ k \leq i,\ \text{and } l \leq j\}$. □

Example 4.7. For the automaton shown in Figure 4.5 we have from the proof of Theorem 4.5, that the corresponding regular 4-BCG that generates the language $css(M) = \{a^i b^j c^{2(i+j)} \mid i, j \geq 0\}$ is $G = (\{S\}, \{a, b, c\}, R, S, (1, 0, 0, 0))$ and R has the following rules:

$r_1 : S \to aS \quad ((1,0,0,0), \infty_4; (0,2,0,0)),$
$r_2 : S \to S \quad ((1,0,0,0), \infty_4; (-1,0,1,0)),$
$r_3 : S \to bS \quad ((0,0,1,0), \infty_4; (0,2,0,0)),$
$r_4 : S \to S \quad ((0,0,1,0), \infty_4; (0,0,-1,1)),$
$r_5 : S \to cS \quad ((0,1,0,1), \infty_4; (0,-1,0,0)),$
$r_6 : S \to \varepsilon \quad ((0,0,0,1), (0,0,0,1); 0_4).$

The string $w = a^2bc^6$ has a derivation in G given by:

$S_{(1,0,0,0)} \Rightarrow_{r_1} aS_{(1,2,0,0)} \Rightarrow_{r_1} a^2S_{(1,4,0,0)} \Rightarrow_{r_2} a^2S_{(0,4,1,0)} \Rightarrow_{r_3} a^2bS_{(0,6,1,0)} \Rightarrow_{r_4} a^2bS_{(0,6,0,1)} \Rightarrow^*_{r_5}$
$a^2bc^5S_{(0,1,0,1)} \Rightarrow_{r_5} a^2bc^6S_{(0,0,0,1)} \Rightarrow_{r_6} a^2bc^6_{(0,0,0,1)}.$ □

It is not hard to show that Theorem 4.5 holds for an arbitrary initial marking m_0 and arbitrary set of final markings. In the proof of Theorem 4.5 we only need to set the initial bag in G to m_0, and the limits of the termination rules (rules of the form $S \to \varepsilon$) such that they can be applied only to a bag value corresponding to a final marking. In other words, if m_f is a given final marking, then we add the rule $S \to \varepsilon\,(m_f, m_f, (0, \ldots, 0))$ to the BCG. In this case we change our notation of Petri net automata such that we let F refer to the set of final markings instead of final places, and we write F_m, and for the terminal Petri net language with respect to F_m we write $L_t(M)$.

Definition 4.4. *Let $G = (N, \Sigma, R, S, \beta_0)$ be a k-BCG. We define $bags(G)$ to be the set of all bags that can be obtained by applying some applicable rules in R starting from S with initial bag β_0. Thus*

$$bags(G) = \{\beta \in \mathbb{Z}^k \mid (S, \beta_0) \Rightarrow^* (S', \beta) \text{ where } S' \in (N \cup \Sigma)^*\}.$$

Definition 4.5. *For any BCG G, we define an associated BCG that is given by $G_f = (N, \Sigma, R_f, S, \beta_0)$, where R_f is R and the following additional rules: for each nonterminal $A \in N$ we add to R_f a rule $A \to \varepsilon\,(-\infty_k, \infty_k, 0_k)$. Thus, $R_f = R \cup \{A \to \varepsilon\,(-\infty_k, \infty_k, 0_k) \mid A \in N\}$.*

It is easy to see that $\beta \in bags(G)$ if and only if there exists $w \in L(G_f)$ such that $(S, \beta_0) \Rightarrow^*_{G_f} (w, \beta)$. Also, $L(G) \subseteq L(G_f)$.

Let $M = (P, \Sigma, T, F_m, m_0)$ be a k-place Petri net automaton and $G = (\{S\}, \Sigma, R, S, m_0)$ the corresponding regular k-BCG constructed by Theorem 4.5. Note that as observed before, the proof of Theorem 4.5 can be generalized to handle an arbitrary initial marking and set of final markings. Then one can show that $L_t(M) = L(G)$, $L_m(M) = bags(G)$, and $L_f(M) = L(G_f)$. We will not give a proof for this.

Corollary 4.6. *For each k-place Petri net automaton M with a set F_m of final markings, there exists a regular k-BCG G with one nonterminal such that:*

- $L_t(M) = L(G)$;
- $L_m(M) = bags(G)$;
- $L_f(M) = L(G_f)$. ■

Example 4.8. For the regular 4-BCG G corresponding to the automaton in Figure 4.5 one can show that:

$bags(G) = \{(1, 2n, 0, 0) \mid n \geq 0\} \cup \{(0, 2n, 1, 0) \mid n \geq 0\} \cup \{(0, n, 0, 1) \mid n \geq 0\} = L_m(M)$,

and $L(G_f) = \{a^i b^j c^k \mid 0 \leq i, j, k \text{ and } k \leq 2(i + j)\} = L_f(M)$.

For the regular 6-BCG G corresponding to the automaton in Figure 4.6 we have:

$bags(G) = \{(1, n, 0, 0, 0, 0) \mid n \geq 0\} \cup \{(0, n_1, 1, n_2, 0, 0) \mid n_1, n_2 \geq 0\} \cup \{(0, n_1, 0, n_2, 1, 0) \mid n_1, n_2 \geq 0\} \cup \{(0, n_1, 0, n_2, 0, 1) \mid n_1, n_2 \geq 0\} = L_m(M)$

and $L(G_f) = \{a^i b^j c^k d^l \mid 0 \leq i, j, k, l,\ k \leq i, \text{ and } l \leq j\} = L_f(M)$. □

In this chapter we considered the relation between recognizing devices and generating mechanisms. We considered counter automata, blind, partially blind, and Petri net automata as recognizing devices. We showed how to simulate these recognizing devices with BCGs. We also showed that regular 2-BCGs generate the class of recursively enumerable languages. We also introduced various examples of our own in order to illustrate theorems and definitions.

Chapter 5

Pumping and shrinking lemmas

A well-known way to show that a language is not context-free is by using the pumping lemma for context-free languages. A pumping lemma for $rPcl$-2 (random permitting context languages with context-free rules) was introduced in [19]. This pumping lemma generalizes the pumping lemma for context-free languages, since the class of context-free languages is included in $rPcl$-2. We give an alternative description of the pumping lemma for $rPcl$-2 in terms of homomorphisms on strings.

We also state, without proof, the shrinking lemma for $rFcl$-2 (random forbidding context languages with context-free rules) as in [18]. We illustrate the pumping and shrinking lemmas with our own examples.

In this chapter we assume our grammars ($rPcg$-2 and $rFcg$-2) do not allow ε-productions.

5.1 A pumping lemma for $rPcl$-2

First we provide some notation that is required.

Recall that $\mathbb{N} = \{1, 2, 3, \dots\}$, $\mathbb{N}_0 = \{0, 1, 2, 3, \dots\}$ and let $[l] = \{1, 2, \dots, l\}$. All languages will be over the terminal set Σ. Let L be a language, thus $L \subseteq \Sigma^*$. We denote by $L_{\geq m}$ the set $\{w \in L \mid |w| \geq m\}$. The set $X = \{X_1, X_2, \dots\}$ is a new set of symbols with no symbol in common with any terminal or nonterminal set of the grammars and languages that we consider. By $X_{\leq k}$, we denote the set $\{X_1, X_2, \dots, X_k\}$. Let $D_\Sigma : (\Sigma \cup X)^* \to X^*$ be the homomorphism that deletes Σ and is the identity on X. The homomorphism $id_\Sigma : \Sigma^* \to \Sigma^*$, is the identity homomorphism on Σ^*. If $\phi : B \to C$ is a homomorphism and $A \subseteq B$, then we denote the restriction of ϕ to A by $\phi_{|A}$. If $\vec{u} = (u_1, \dots, u_n), \vec{v} = (v_1, \dots, v_n) \in \mathbb{N}_0^n$, then $\vec{u} \leq \vec{v}$

if $u_i \leq v_i$ for $i = 1, \ldots, n$. By $|\vec{c}|$ we denote the sum of the components of $\vec{c}$. For two strings s_1 and s_2, if $\vec{p}(s_1) \leq \vec{p}(s_2)$ we write $s_1 \preceq s_2$, where $\vec{p}(s)$ is the Parikh image for a string s. Recall that the Parikh image of a string w over the ordered alphabet $\Sigma = \{a_1, a_2, \ldots, a_n\}$ is given by $\psi(w) = (|w|_{a_1}, |w|_{a_2}, \ldots, |w|_{a_n})$ where $|w|_{a_i}$ is the number of occurrences of a_i in w, $1 \leq i \leq n$. If s_1 is a substring of s_2, we write $s_1 \subseteq s_2$.

Lemma 5.1. *[19] Let $p_1, p_2, \ldots$ be a sequence of nonnegative integers, and n a positive integer. There exists an integer b that depends only on the sequence $p_1, p_2, \ldots$ with the following property. If $\vec{c}_1, \vec{c}_2, \ldots$ in $\mathbb{N}_0^n$ with $|\vec{c}_i| \leq p_i$ for $i \geq 1$, then there exist indices i and j with $1 \leq i < j \leq b$ and $\vec{c}_i \leq \vec{c}_j$.* ■

Definition 5.1. *Let G be any string grammar. We say that G is sequential if its rules are applied sequentially during the derivation.*

Lemma 5.2. *Let G be a sequential string grammar and $L = L(G)$. There exists a positive integer m such that any $w \in L_{\geq m}$ has a derivation $s_0 \Rightarrow^* s_1 \Rightarrow^* \cdots \Rightarrow^* s_d \Rightarrow^* w$, with $|s_j| < |s_{j+1}|$ for $0 \leq j < d$, and $s_u \preceq s_v$ for some u and v with $0 \leq u < v \leq d$.*

Proof: Let N, Σ, R, S be the nonterminals, terminals, rules, and start symbol of G, respectively. Let w be an arbitrary long string in L, and consider a derivation $S = s_0 \Rightarrow^* s_1 \Rightarrow^* \cdots \Rightarrow^* s_d \Rightarrow^* w$ for w with $s_j \in (N \cup \Sigma)^*, |s_j| < |s_{j+1}|$, $0 \leq j < d$, and $|s_j| \leq j(a-1)+1$, where a is the length of longest right-hand side in R. Let $\vec{p}(s_0), \vec{p}(s_1), \ldots, \vec{p}(s_d)$ be the sequence of Parikh vectors associated with the derivation above. We have that $|\vec{p}(s_j)| = |s_j| \leq j(a-1)+1$ for $0 \leq j \leq d$. Let $p_j = j(a-1)+1$ for $j \geq 0$ and b an integer as in Lemma 5.1. If we set m equal to $b(a-1)+1$, we can find the required u and v with $s_u \preceq s_v$. ■

Next we state the pumping lemma for $rPcl$-2.

5.1.1 The pumping lemma

In this section we state and prove the pumping lemma in terms of string homomorphisms.

Theorem 5.3. *Let $L \in rPcl$-2 and $w \in L_{\geq m}$ with m as in Lemma 5.2. Then we can find:*

- *$l \in [m]$;*
- *a permutation $p : [l] \to [l]$;*
- *$\overline{w} \in (\Sigma \cup X_{\leq l})^*$ with $D_\Sigma(\overline{w}) = X_{p(1)} \ldots X_{p(l)}$;*
- *a homomorphism $\psi : (\Sigma \cup X_{\leq l})^* \to \Sigma^*$ with $\psi_{|\Sigma} = id_\Sigma$, and $\psi(\overline{w}) = w$;*
- *a homomorphism $\phi : (\Sigma \cup X_{\leq l})^* \to (\Sigma \cup X_{\leq l})^*$ with $\phi_{|\Sigma} = id_\Sigma$ and $D_\Sigma\phi(X_1 \ldots X_l) = X_{p(1)} \ldots X_{p(l)}$;*

- *at least one of the strings $\phi(X_i)$ for $i \in [l]$, contains a terminal symbol;*

such that $\psi\phi^i(\overline{w}) \in L$ for all $i \in \mathbb{N}_0$.

Proof: Let $G = (N, \Sigma, R, S) \in rPcg\text{-}2$ and $L = L(G)$. Consider the derivation $S = s_0 \Rightarrow^* s_1 \Rightarrow^* \cdots \Rightarrow^* s_d \Rightarrow^* w$ for $w \in L_{\geq m}$ and assume that $s_u \preceq s_v$ where $u < v$. Consider a derivation tree for w. Let t_u and t_v be the derivation trees for s_u and s_v respectively, that are also subtrees of the derivation tree for w. Let $A = A_1A_2 \dots A_l$ and $B = B_1B_2 \dots B_k$ be the strings s_u and s_v with terminals removed. By deleting some of the nonterminals from B, we obtain a string $B' = B'_1 \dots B'_l$ and a permutation $p : [l] \to [l]$ such that $B'_i = A_{p(i)}$.

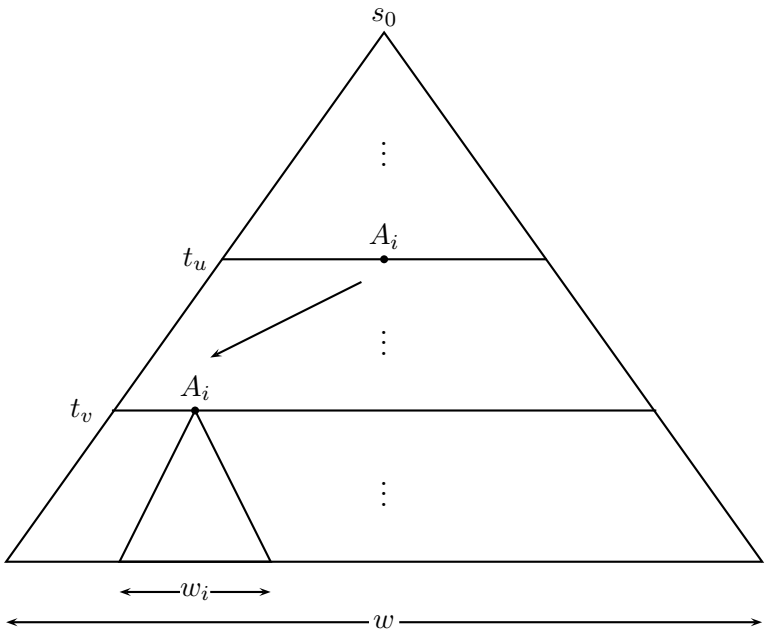


Figure 5.1: Derivation tree for w

Let w_i be the substring of w generated by A_i in B'. Also let $\overline{w} \in (\Sigma \cup X_{\leq l})^*$ be the word obtained by replacing each w_i in w by X_i, as indicated in Figure 5.1. We can thus define a homomorphism $\psi : (\Sigma \cup X_{\leq l})^* \to \Sigma^*$ such that $\psi_{|\Sigma} = id_\Sigma$, and $\psi(X_i) = w_i$ for $i \in [l]$. Clearly $\psi(\overline{w}) = w$ and $D_\Sigma(\overline{w}) = X_{p(1)} \dots X_{p(l)}$.

Starting from t_v, we can apply to each A_i in B' the same sequence of rules that were applied in order to obtain s_v from s_u. Note that the additional context in s_v, which is not present in s_u, will not prohibit this. This can be repeated as many times as we want. In terms of homomorphisms we can thus define a homomorphism $\phi : (\Sigma \cup X_{\leq l})^* \to (\Sigma \cup X_{\leq l})^*$, such that $\phi_{|\Sigma} = id_\Sigma$, $D_\Sigma\phi(X_1 \dots X_l) = X_{p(1)} \dots X_{p(l)}$, and $\phi(X_i)$ is the substring of $\overline{w}$ corresponding to

the substring of w derived from A_i in A. Note that since there are no ε-productions, there exists an X_i such that $\phi(X_i)$ contains terminal symbols. In terms of ϕ and ψ, we thus have that $\psi\phi^i(\overline{w}) \in L$ for all $i \geq 0$. ■

5.1.2 Illustration of the pumping lemma with three nonterminals

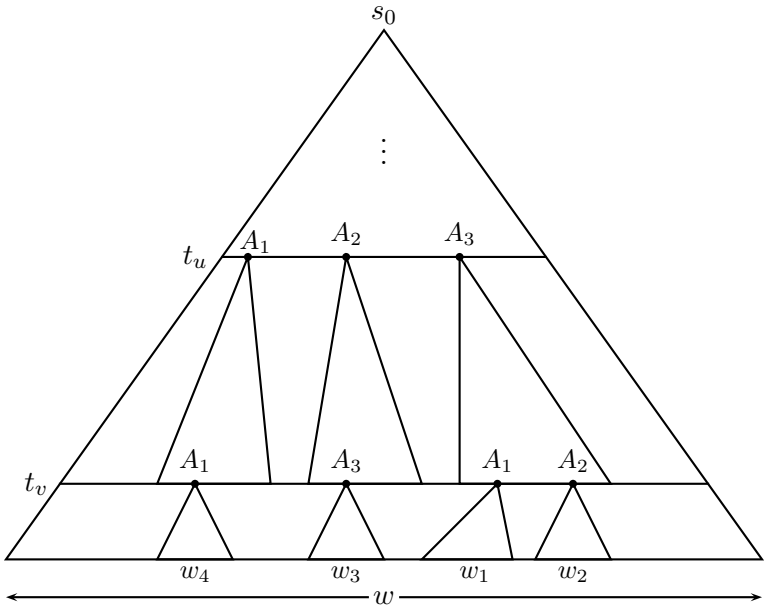


Figure 5.2: Derivation tree for w

In Figures 5.2 and 5.3 we have: $l = 3$, $X_{\leq 3} = \{X_1, X_2, X_3\}$;
$p(1) = 3$, $p(2) = 1$, and $p(3) = 2$;
$\psi(X_i) = w_i$ for $i \in [l]$, $\psi(\overline{w}) = w$, $D_\Sigma(\overline{w}) = X_3X_1X_2$;
$D_\Sigma\phi(X_1) = \varepsilon$, $D_\Sigma\phi(X_2) = X_3$, $D_\Sigma\phi(X_3) = X_1X_2$.

Example 5.1. Consider the language $L = \{(zxy)^{2n-1} \mid n \geq 1\} \cup \{(zxy)^n(xyz)^n \mid n \geq 1\}$. The language L is generated by the grammar $G = (\{S, A, B, C, E, G\}, \{x, y, z\}, R, S) \in rPcg$-2, where R contains the following rules:

$(S \to ABC, \emptyset)$, $(A \to zE, \{B, C\})$, $(B \to x, \{C\})$, $(C \to yG, \{E\})$, $(E \to BC, \{G\})$, $(G \to A, \{B, C\})$, $(A \to z, \{B, C\})$, $(C \to y, \emptyset)$.

Consider the string $w = zxyxyz$ with derivation tree as shown in Figure 5.4. According to the pumping lemma we have the following:

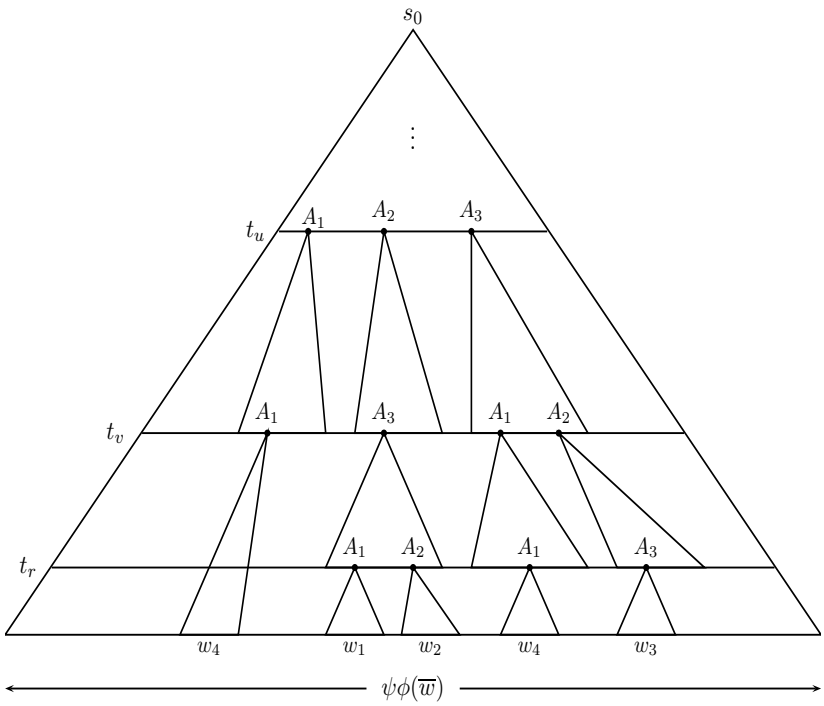


Figure 5.3: Pumping with three nonterminals

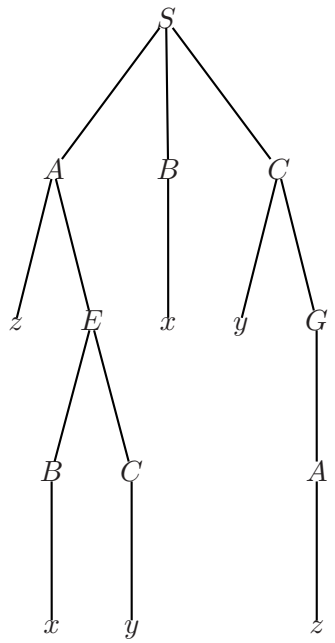


Figure 5.4: Pumping lemma: Derivation tree for w

$l = 3$, $X_{\leq 3} = \{X_1, X_2, X_3\}$, $\psi(X_1) = z$, $\psi(X_2) = x$, $\psi(X_3) = y$;
$p(1) = 2$, $p(2) = 3$, $p(3) = 1$;
$\overline{w} = zX_2X_3xyX_1$, $D_\Sigma(\overline{w}) = X_2X_3X_1$, $\psi(\overline{w}) = zxyxyz = w$;
$\phi(X_1) = zX_2X_3$, $\phi(X_2) = x$, $\phi(X_3) = yX_1$, and $D_\Sigma\phi(X_1X_2X_3) = X_2X_3X_1$.

Now $\psi\phi(\overline{w}) = \psi(zxyX_1xyzX_2X_3) = (zxy)^3$, and
$\psi\phi^2(\overline{w}) = \psi(zxyzX_2X_3xyzxyX_1) = (zxy)^2(xyz)^2$.

We notice that $\psi\phi^{2i-1}(\overline{w}) = (zxy)^{2i+1}$ and $\psi\phi^{2i}(\overline{w}) = (zxy)^{i+1}(xyz)^{i+1}$ are in L for all $i \geq 1$. □

5.2 A shrinking lemma for $rFcl$-2

The shrinking lemma for $rFcl$-2 (random forbidding languages with context-free rules) is in essence the dual to the pumping lemma for $rPcl$-2. For completeness we state this result and illustrate it with an example.

Theorem 5.4. *[18] Let $L \in rFcl$-2. For any integer $t \geq 2$ there exists an integer b, which depends only on L and t, such that for any string $z \in L$ with $|z| \geq b$ there are t strings $z_1, z_2, \ldots, z_t = z$ in L, and $t-1$ numbers $l_2, \ldots, l_t$ with $1 \leq l_2, \ldots, l_t \leq b$, such that for each j with $2 \leq j \leq t$,*

- *z_j contains l_j mutually disjoint nonempty substrings $w_{j1}, \ldots, w_{jl_j}$, and l_j mutually disjoint nonempty substrings $x_{j1}, \ldots, x_{jl_j}$, these being related by a function $\vartheta_j : \{1, \ldots, l_j\} \to \{1, \ldots, l_j\}$ such that for each i with $i \in [l_j]$, $x_{ji} \subseteq w_{j\vartheta_j(i)}$ and for at least one i with $i \in [l_j]$, $x_{ji} \subsetneq w_{j\vartheta_j(i)}$;*
- *the word z_{j-1} is obtained by substituting x_{ji} for w_{ji} for all i with $i \in [l_j]$, in z_j.* ■

Example 5.2. The language $L = \{(zxy)^{2n-1} \mid n \geq 1\} \cup \{(zxy)^n(xyz)^n \mid n \geq 1\}$ can be generated by a grammar $G = (\{S, A, B, C, E, G, X\}, \{x, y, z\}, R, S) \in rFcg$-2, where R contains the following rules:

$(S \to ABC, \emptyset)$, $(A \to zE, \{G, E\})$, $(B \to x, \{A, G\})$, $(C \to yG, \{A, G, X\})$,
$(E \to BC, \{A, B, C\})$, $(G \to A, \{E, X\})$, $(A \to X, \{E, G\})$, $(X \to z, \{A, B, C\})$,
$(C \to y, \{A, G, E\})$.

Consider the string $z_3 = (zxy)^3 \in L$ which has a derivation tree as shown in Figure 5.5. According to the shrinking lemma, starting from z_3 we have the following:

$w_{31} = zxyz$, $w_{32} = x$, $w_{33} = yzxy$;
$x_{31} = zxy$, $x_{32} = x$, $x_{33} = yz$;

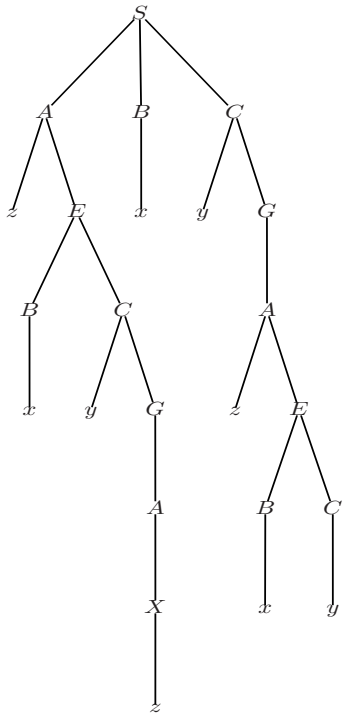


Figure 5.5: Shrinking lemma: Derivation tree for z_3

$\vartheta_3(1) = 3$, $\vartheta_3(2) = 1$, $\vartheta_3(3) = 1$;
$z_2 = zxyxyz$.

Now starting from $z_2 = zxyxyz$ we have the following:

$w_{21} = zxy$, $w_{22} = x$, $w_{23} = yz$;
$x_{21} = z$, $x_{22} = x$, $x_{23} = y$;
$\vartheta_2(1) = 3$, $\vartheta_2(2) = 1$, $\vartheta_2(3) = 1$;
$z_1 = zxy$. □

In this chapter we considered the pumping lemma for $rPcl$-2 as introduced in [19]. We provided a different proof in terms of string homomorphisms. We also stated the shrinking lemma for $rFcl$-2 as in [18].

In the next chapter we consider decidability and complexity properties of grammars with regulated rewriting.

Chapter 6

Decidability and complexity in regulated rewriting

In this chapter we consider the decidability and complexity of regulated rewriting mechanisms. We consider the membership, emptiness, finiteness, and equivalence problems. We list the decidability properties for the grammars with regulated rewriting that were defined in Chapter 3 and show how bag context grammars behave in terms of decidability. Then we discuss the relation between Petri nets, vector addition systems and decidability. We show how we can use Petri nets for decision problems on $rPcg$-2 (random permitting context grammars with context-free rules). After that we discuss the relationship between Groebner bases and the decidability of reachability in Petri nets as in [4, 6]. We show how Groebner bases are used to decide reachability in reversible Petri nets and illustrate this method with an example. We conclude the chapter with a discussion on the complexity of grammars with regulated rewriting. We consider the complexity of different classes of grammars in terms of the number of nonterminals. We state important results about bounds and relationships between the complexity of various grammars with regulated rewriting. Then we give an example that compares the complexity of different classes of grammars.

6.1 Decidability properties of regulated rewriting

First we give a definition of a decision problem.

Definition 6.1. *[10] A decision problem is an expression containing one or more variables, and that becomes "true" or "false" if elements of the basic sets are substituted for the variables. We say a decision problem is decidable, if and only if, there is an algorithm which, for a given tuple, decides whether or not the expression becomes "true".*

In general, a decision problem can be formulated as a question that has the answer "yes" or "no".

There are many decision problems that can be considered. We consider here the following problems.

Definition 6.2. *[10] Let $\mathcal{G}$ be a family of grammars and V any set of alphabets. Then we define:*

- *Membership Problem: Given $w \in V^*$ and $G \in \mathcal{G}$, does $w \in L(G)$ hold?*
- *Emptiness Problem: Is the language generated by a grammar $G \in \mathcal{G}$ empty?*
- *Finiteness Problem: Is the language generated by a grammar $G \in \mathcal{G}$ finite?*
- *Equivalence Problem: Are given grammars $G_1, G_2 \in \mathcal{G}$ equivalent?*

Sometimes we talk about decidability for the families of languages instead of families of grammars. Next we list the decidability properties for the families of grammars in the Chomsky hierarchy and ET0L systems [10].

	membership	*emptiness*	*finiteness*	*equivalence*
$\mathcal{L}(REG)$	+	+	+	+
$\mathcal{L}(CF)$	+	+	+	−
$\mathcal{L}(CS)$	+	−	−	−
$\mathcal{L}(RE)$	−	−	−	−
$\mathcal{L}(ET0L)$	+	+	+	−

Here "+" and "−" indicate the decidability and undecidability respectively and $\mathcal{L}(REG)$, for example, is the class of regular languages.

Let C_1 and C_2 be two classes of languages such that $C_1 \subseteq C_2$, and let d be a decidability property. We note that if d is decidable for C_2 then it must be also decidable for C_1, and if d is undecidable for C_1 then it is also undecidable for C_2. If $C_1 = C_2$ then the two classes have the same decidability properties.

Recall that M, P, RC, V and BCG are the classes of matrix, programmed, random context, valence, and bag context grammars, respectively. Let $X \in \{M, P, RC, V, BCG\}$. From the property mentioned above it is obvious that the membership, emptiness, finiteness and equivalence problems are undecidable for $\mathcal{L}(X, RE)$. Also, the emptiness, finiteness and equivalence problems are undecidable for $\mathcal{L}(X, CS)$, and the equivalence problem is undecidable for $\mathcal{L}(X, CF)$.

Next we state the relationship between various families of languages generated by regulated rewriting mechanisms. This theorem can be used to answer various decidability questions.

Theorem 6.1. *[9, 10]*

1. $\mathcal{L}(RC, CF) \subseteq \mathcal{L}(M, CF) = \mathcal{L}(P, CF)$,
   $\mathcal{L}(RC, CF - \lambda) \subseteq \mathcal{L}(M, CF - \lambda) = \mathcal{L}(P, CF - \lambda)$.
2. $\mathcal{L}(RC, CF, ac) = \mathcal{L}(M, CF, ac) = \mathcal{L}(P, CF, ac) = \mathcal{L}(RE)$,
   $\mathcal{L}(RC, CF - \lambda, ac) = \mathcal{L}(M, CF - \lambda, ac) = \mathcal{L}(P, CF - \lambda, ac)$.
3. $\mathcal{L}(CF) \subset \mathcal{L}(RC, CF - \lambda)$.
4. *For* $X \in \{M, P, RC\}$ *we have:*
   $\mathcal{L}(X, CF - \lambda) \subseteq \mathcal{L}(X, CF) \subseteq \mathcal{L}(X, CF, ac)$.
   $\mathcal{L}(X, CF - \lambda) \subseteq \mathcal{L}(X, CF - \lambda, ac) \subseteq \mathcal{L}(X, CF, ac)$.
5. *For* $X \in \{M, P, RC\}$, *the family* $\mathcal{L}(X, CF - \lambda, ac)$ *is strictly contained in* $\mathcal{L}(CS)$.
6. $\mathcal{L}(CF) \subset \mathcal{L}(V, CF - \lambda) = \mathcal{L}(V, CF) \subset \mathcal{L}(M, CF - \lambda)$. ■

For $X \in \{M, P, RC\}$ we have that $\mathcal{L}(X, CF - \lambda, ac) \subset \mathcal{L}(CS)$. Thus, the membership problem is decidable for the family $\mathcal{L}(X, CF - \lambda, ac)$. Dassow and Păun showed in [10] that the emptiness and finiteness problems are undecidable for the family $\mathcal{L}(M, CF - \lambda, ac)$, but that the emptiness problem is decidable for $\mathcal{L}(M, CF)$. It has also been shown that the membership and finiteness problems are decidable for $\mathcal{L}(M, CF)$ (see [24]). Thus, we have the following table of decidability results:

	membership	*emptiness*	*finiteness*	*equivalence*
$\mathcal{L}(CF)$	+	+	+	−
$\mathcal{L}(V, CF)$	+	+	+	−
$\mathcal{L}(X, CF - \lambda)$	+	+	+	−
$\mathcal{L}(X, CF)$	+	+	+	−
$\mathcal{L}(X, CF - \lambda, ac)$	+	−	−	−
$\mathcal{L}(X, CF, ac)$	−	−	−	−

In the table above, $X \in \{M, P, RC\}$.

We notice that for families of grammars with regulated rewriting and no appearance checking, the decidability properties are equivalent to the decidability properties for the context-free grammars.

Next we consider the families of languages $\mathcal{L}(BCG, Y)$, where $Y \in \{RE, CS, CF, REG\}$. From Corollary 4.4 we have that any recursively enumerable language can be generated by a regular bag context grammar. Thus, $\mathcal{L}(BCG, Y)$ have the following decidability properties:

	membership	*emptiness*	*finiteness*	*equivalence*
$\mathcal{L}(BCG, Y)$	–	–	–	–

6.2 Petri nets, vector addition systems, and decidability

Petri nets were introduced in 1962 by A.C Petri, and vector addition systems were defined independently some years later, in 1969, by Karp and Miller. It was observed that Petri nets and vector addition systems are mathematically equivalent [17]. Most of the interesting decision problems, for example reachability, are decidable for Petri nets (see [4]).

In this section we show how to use the decidability of reachability of Petri nets and vector addition systems to show decidability in regulated rewriting. Our contribution is to prove that the emptiness of $rPcg$-2 is decidable, and give a particular answer for the membership problem using Petri nets reachability.

Next we give a definition of the reachability in Petri nets and vector addition systems. For our purpose here we consider Petri nets in this chapter as a 4-tuple $PN = (P, \{\varepsilon\}, T, m_0)$ where P, T and m_0 are as in Definition 4.3.

Definition 6.3 (Reachability in Petri nets)**.** *[4] A marking m_j is reachable from a marking m_i in a Petri net PN if and only if there exists $w \in T^*$ such that $\delta(m_i, w) = m_j$, where δ is the transition function of PN.*

Definition 6.4 (Vector addition systems)**.** *[10] An n-dimensional vector addition system is a pair (x_0, V) where $x_0 \in \mathbb{N}_0^n$ (the initial point) and V is a finite subset of $\mathbb{Z}^n$. A point $y \in \mathbb{N}_0^n$ is reachable within this vector addition system if and only if there are vectors $v_1, \ldots, v_t$ in V such that:*

- $(x_0 + \sum_{i=1}^{j} v_i) \in \mathbb{N}_0^n$ *for* $1 \leq j \leq t$;
- $(x_0 + \sum_{i=1}^{t} v_i) = y$.

It has been shown by Mayer in 1981 that the reachability for vector addition systems is decidable, and therefore the same holds for Petri nets (see [17]). The reachability problem in Petri nets or vector addition systems gained an increasing importance since many other interesting problems can be reduced to it. For instance, in formal language theory, Dassow

and Păun proved in [10] that the emptiness of $\mathcal{L}(M, CF)$ is decidable by reducing it to the reachability of vector addition systems. Also Hauschildt and Jantzen used the decidability of semilinearity related to a given reachability set of a Petri net, to solve various open problems in [10]. They proved that the finiteness problem is decidable for $\mathcal{L}(M, CF)$, and thus also for $\mathcal{L}(RC, CF)$ [24].

In the rest of this section we discuss decidability problems for $rPcg$-2. We use the reachability of Petri nets to show that the emptiness of $rPcg$-2 is decidable. We also consider the membership problem.

Next we introduce a procedure that transforms any grammar $G \in rPcg$-2 to a Petri net PN.

Let $G = (N, \Sigma, R, S) \in rPcg$-2. We construct a Petri net $PN = (P, \{\varepsilon\}, T, m_0)$ as follows (see [5]):

- $P = \{x \mid x \in (N \cup \Sigma)\}$;
- For each rule $r : (A \to w, Q)$ add to the Petri net a transition $t : (\psi(AQ_c), \varepsilon, \psi(wQ_c))$, where ψ is the Parikh image taken over $N \cup \Sigma$, and Q_c is the string obtained by concatenating the nonterminals in Q. Without loss of generality we assume that $A \notin Q$;
- $m_0 = \psi(S)$.

It is clear that the Petri net is designed in such a way that a transition t in PN is enabled if the corresponding rule r in R is applicable. Moreover $\psi(L'(G)) = L_m(PN)$, where $L'(G) = \{w \in (N \cup \Sigma)^* \mid S \Rightarrow^* w\}$ and $L_m(PN)$ is the language of all reachable markings from the initial marking.

Example 6.1. To illustrate this procedure consider a grammar $G = (N, \Sigma, R, S) \in rPcg$-2, where $N = \{S, A, B, C, D, E, F\}$, $\Sigma = \{a, b, c\}$, and R contains rules:

$r_1 : (S \to ABC, \emptyset)$, $r_2 : (A \to aD, \{B\})$, $r_3 : (B \to bE, \{C\})$, $r_4 : (C \to c^2F, \{D\})$, $r_5 : (D \to A, \{E\})$, $r_6 : (E \to B, \{F\})$, $r_7 : (F \to C, \{A\})$, $r_8 : (A \to a, \{B\})$, $r_9 : (B \to b, \{C\})$, $r_{10} : (C \to c^2, \emptyset)$.

This grammar generates the language $L = \{a^i b^i c^{2i} \mid i \geq 1\}$.

The corresponding Petri net is given by $PN = (N \cup \Sigma, \{\varepsilon\}, T, m_0)$. If we fix the order $S < A < B < C < D < E < F < a < b < c$ on $N \cup \Sigma$, then $m_0 = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0)$, and T contains the following transitions:

$t_1 = (1,0,0,0,0,0,0,0,0,0,\varepsilon,0,1,1,1,0,0,0,0,0,0)$,
$t_2 = (0,1,1,0,0,0,0,0,0,0,\varepsilon,0,0,1,0,1,0,0,1,0,0)$,
$t_3 = (0,0,1,1,0,0,0,0,0,0,\varepsilon,0,0,0,1,0,1,0,0,1,0)$,
$t_4 = (0,0,0,1,1,0,0,0,0,0,\varepsilon,0,0,0,0,1,0,1,0,0,2)$,
$t_5 = (0,0,0,0,1,1,0,0,0,0,\varepsilon,0,1,0,0,0,1,0,0,0,0)$,
$t_6 = (0,0,0,0,0,1,1,0,0,0,\varepsilon,0,0,1,0,0,0,1,0,0,0)$,
$t_7 = (0,1,0,0,0,0,1,0,0,0,\varepsilon,0,1,0,1,0,0,0,0,0,0)$,
$t_8 = (0,1,1,0,0,0,0,0,0,0,\varepsilon,0,0,1,0,0,0,0,1,0,0)$,
$t_9 = (0,0,1,1,0,0,0,0,0,0,\varepsilon,0,0,0,1,0,0,0,0,1,0)$,
$t_{10} = (0,0,0,1,0,0,0,0,0,0,\varepsilon,0,0,0,0,0,0,0,0,0,2)$.

Consider the string $w = a^2b^2c^4 \in L$. Then w has a derivation:

$S \Rightarrow_{r_1} ABC \Rightarrow_{r_2} aDBC \Rightarrow_{r_3} aDbEC \Rightarrow_{r_4} aDbEc^2F \Rightarrow_{r_5} aAbEc^2F \Rightarrow_{r_6} aAbBc^2F \Rightarrow_{r_7} aAbBc^2C \Rightarrow_{r_8} a^2bBc^2C \Rightarrow_{r_9} a^2b^2c^2C \Rightarrow_{r_{10}} a^2b^2c^4$.

Now consider the sequence of markings of PN corresponding to the derivation above:

$(1,0,0,0,0,0,0,0,0,0) \vdash_{t_1} (0,1,1,1,0,0,0,0,0,0) \vdash_{t_2} (0,0,1,1,1,0,0,1,0,0) \vdash_{t_3}$
$(0,0,0,1,1,1,0,1,1,0) \vdash_{t_4} (0,0,0,0,1,1,1,1,1,2) \vdash_{t_5} (0,1,0,0,0,1,1,1,1,2) \vdash_{t_6}$
$(0,1,1,0,0,0,1,1,1,2) \vdash_{t_7} (0,1,1,1,0,0,0,1,1,2) \vdash_{t_8} (0,0,1,1,0,0,0,2,1,2) \vdash_{t_9}$
$(0,0,0,1,0,0,0,2,2,2) \vdash_{t_{10}} (0,0,0,0,0,0,0,2,2,4)$.

Recall that we write $m_i \vdash_t m_j$, if $\delta(m_i, t) = m_j$.

Note that the sequence of markings corresponding to the derivation of w is the Parikh images of sentential forms that appear during the derivation. □

Let $G = (N, \Sigma, R, A_1) \in rPcg\text{-}2$ where $N = \{A_1, \ldots, A_n\}$ and $\Sigma = \{a_1, \ldots, a_m\}$. We use the order $A_1 < A_2 < \cdots < A_n < a_1 < a_2 < \cdots < a_m$ in the Parikh map. Let $w \in L(G)$ and consider the Parikh image of w, thus $\psi(w) = (0, \ldots, 0, |w|_{a_1}, \ldots, |w|_{a_m})$. It is easy to see that when the corresponding Petri net PN completes the computation for w, the places $a_1, \ldots, a_m$ will have $|w|_{a_1}, \ldots, |w|_{a_m}$ tokens respectively, and all places corresponding to the nonterminals $A_1, \ldots, A_n$ will have no tokens.

From the construction of PN we have that the language $L(G)$ is not empty if and only if PN has a reachable marking m with $first_n(m) = (0, \ldots, 0)$. Recall that for a vector $x = (x_1, \ldots, x_n) \in \mathbb{Z}^n$ we define $first_k(x)$ to be equal to $(x_1, \ldots, x_k)$, where $0 \leq k \leq n$. Let $t_1 = (u_1, \varepsilon, v_1), \ldots, t_k = (u_k, \varepsilon, v_k)$ be the transitions of PN where $u_i, v_i \in \mathbb{N}_0^{n+m}$. We construct an n-dimensional vector addition system $X = (x_0, V)$ as follows: $x_0 = first_n(m_0) =$

$(1, 0, \ldots, 0)$ and $V = \{first_n(v_1 - u_1), \ldots, first_n(v_k - u_k)\}$. It is clear that the zero vector is reachable within X if and only if a marking m with $first_n(m) = (0, \ldots, 0)$ is reachable in PN.

Hence the reachability of such a marking is decidable, and thus the emptiness for $rPcl$-2 is decidable.

Definition 6.5. *For a string $w \in \Sigma^*$ we define $per(w)$ to be the set of strings in Σ^* which have the same Parikh image as w. Thus, $per(w) = \{w' \in \Sigma^* \mid \psi(w') = \psi(w)\}$.*

Suppose we are given a string $w \in \Sigma^*$ and we want to decide if $w \in L(G)$. We can decide if $\psi(w)$ is a reachable marking for the corresponding Petri net PN. If it is reachable then one or more elements from $per(w)$ are in $L(G)$, but we cannot decide which specific strings are in the random permitting language.

6.3 Groebner bases and decidability

Groebner bases are an important algebraic tool to solve systems of non-linear algebraic equations. They were introduced by B. Buchberger in 1965 [3]. They have various applications in algebra, where they are used for the description of ideals in polynomial rings, to solve the ideal membership problem for polynomial rings, and to solve systems of polynomial equations in multivariables. They are also applied in integer programming, coding theory, signal processing and differential equations [6].

In this section we use the theory of Groebner bases to show the decidability of reachability in reversible Petri nets as introduced in [4, 6], and give an example to illustrate the method. A good introduction and background on Groebner bases can be found in [8].

Definition 6.6. *(Reversibility)[6] A Petri net PN is called reversible if the following condition holds: if a marking m_j is reachable from a marking m_i in PN, then it is also the case that m_i is reachable from m_j.*

We describe a k-place Petri net $PN = (P, \{\varepsilon\}, T, m_0)$ in terms of polynomials as follows [4]:

- for each place $p_i \in P$ we use a variable x_i for $1 \leq i \leq k$;
- for each transition $t_j = (u_1, \ldots, u_k, \varepsilon, v_1, \ldots, v_k)$ we use a polynomial $f_j = x_1^{u_1} \ldots x_k^{u_k} - x_1^{v_1} \ldots x_k^{v_k}$;
- each marking $m = (u_1, \ldots, u_k)$ has an associated polynomial $x_1^{u_1} \ldots x_k^{u_k}$.

Let $t \in T$ and $f = l - r$ the polynomial associated with t, where l and r are the two terms of f. The transition t is enabled in a Petri net PN with a marking m_i, if the polynomial associated with m_i can be written as a product ul, where u is a monomial in $x_1, \ldots, x_k$. After firing t, the marking m_i changes to m_j, where m_j is the marking with associated polynomial $ul - u(l - r) = ur$.

Let G be a Groebner basis of the ideal generated by the polynomials corresponding to transitions of the Petri net PN. A marking m_j is reachable from m_i if and only if $f_j \longrightarrow_G f$ and $f_i \longrightarrow_G f$, where $\longrightarrow_G$ indicates the reduction of polynomials to the normal form with respect to dividing by the elements of G. In other words, m_j is reachable from m_i if and only if the corresponding polynomials have the same normal form when they are reduced with respect to G.

Next we illustrate this procedure with an example of our own.

Example 6.2. To illustrate this procedure, consider the reversible Petri net PN shown in Figure 6.1.

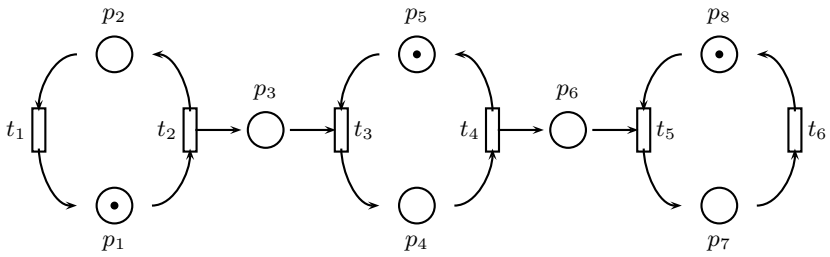


Figure 6.1: Reversible Petri net

This Petri net is formally described by $PN = (\{p_1, \ldots, p_8\}, \{\varepsilon\}, \{t_1, \ldots, t_6\}, m_0)$ where $m_0 = (1, 0, 0, 0, 1, 0, 0, 1)$ and the transitions are defined as follows:

$t_1 = (0, 1, 0, 0, 0, 0, 0, 0, \varepsilon, 1, 0, 0, 0, 0, 0, 0, 0)$, $t_2 = (1, 0, 0, 0, 0, 0, 0, 0, \varepsilon, 0, 1, 1, 0, 0, 0, 0, 0)$,
$t_3 = (0, 0, 1, 0, 1, 0, 0, 0, \varepsilon, 0, 0, 0, 1, 0, 0, 0, 0)$, $t_4 = (0, 0, 0, 1, 0, 0, 0, 0, \varepsilon, 0, 0, 0, 0, 1, 1, 0, 0)$,
$t_5 = (0, 0, 0, 0, 0, 1, 0, 1, \varepsilon, 0, 0, 0, 0, 0, 0, 1, 0)$, $t_6 = (0, 0, 0, 0, 0, 0, 1, 0, \varepsilon, 0, 0, 0, 0, 0, 0, 0, 1)$.

These transitions have associated polynomials:
$f_1 = x_2 - x_1$, $f_2 = x_1 - x_2x_3$, $f_3 = x_3x_5 - x_4$, $f_4 = x_4 - x_5x_6$, $f_5 = x_6x_8 - x_7$ and $f_6 = x_7 - x_8$.
Using Singular [23] and the lexicographic order, we obtain the Groebner basis $G = \{x_7 - x_8, x_6x_8 - x_7, x_4 - x_5x_6, x_3x_5 - x_4, x_2x_5x_6 - x_2x_5, x_2x_3 - x_2, x_1 - x_2\}$.

Suppose the Petri net PN has a marking $m = (1, 0, 20, 0, 1, 0, 0, 1)$, and we want to decide if a marking $m' = (0, 1, 9, 0, 1, 3, 1, 0)$ is reachable from m. Let $f = x_1x_3^{20}x_5x_8$ and $f' = x_2x_3^9x_5x_6^3x_7$ be the polynomials corresponding to m and m' respectively. We have that $f \longrightarrow_G x_2x_5x_8$, and $f' \longrightarrow_G x_2x_5x_8$. Since f and f' have the same normal form with respect to G, we have that m' is reachable from m. Moreover m and m' are reachable from m_0, since the polynomial corresponding to m_0, which is $f_0 = x_1x_5x_8$, has normal form $x_2x_5x_8$.

Now consider the marking $m'' = (0, 5, 2, 1, 0, 3, 5, 0)$. This marking has an associated polynomial $f'' = x_2^5x_3^2x_4x_6^3x_7^5$, which has normal form $x_2^5x_5x_8^5$ with respect to G. Thus, m'' is not reachable from m or m'. □

Note that the reversibility of the Petri net is an essential condition in this procedure.

6.4 Complexity of regulated rewriting

Studying the complexity of grammars is an important issue in theory and practice. In theory it is always interesting to find "simple" grammars to describe languages. The word "simple" can be interpreted in terms of the number of nonterminals, the number of rules, or the length of the description of the grammar in some fixed format (see [9]).

In this section we study the complexity of grammars with regulated rewriting in terms of the number of nonterminals. We define this measure and state important results from [9, 10] that give bounds and describe relations between different classes of grammars with regulated rewriting.

First we need the following definitions.

Definition 6.7. *[10] Let X be a family of grammars and let $G \in X$ and $L \in \mathcal{L}(X)$. By $Var(G)$ we denote the cardinality of the set of nonterminals of G, and we set*

$$Var_X(L) = min\{Var(G) \mid G \in X \text{ and } L = L(G)\}.$$

Theorem 6.2. *[9, 10] Let $\alpha \in \{CF, CF - \lambda\}$ and let β be ac or empty.*

(1) For all $L \in \mathcal{L}(RC, \alpha, \beta)$ we have:

- $Var_{(P,\alpha,\beta)}(L) \leq Var_{(RC,\alpha,\beta)}(L) + 2$,
- $Var_{(M,\alpha,\beta)}(L) \leq Var_{(RC,\alpha,\beta)}(L) + 1$.

(2) For all $L \in \mathcal{L}(M, CF, \beta)$ we have:
$Var_{(P,CF,\beta)}(L) \leq Var_{(M,CF,\beta)}(L) + 2$.

(3) For all recursively enumerable languages L, we have:

- $Var_{(M,CF,ac)}(L) \leq Var_{(P,CF,ac)}(L) + 1$,
- $Var_{(M,CF,ac)}(L) \leq 3$ *and* $Var_{(P,CF,ac)}(L) \leq 3$.

(4) For all context-free languages L we have:
$Var_{CF}(L) = Var_{CF-\lambda}(L)$. ■

Next we consider an example that compares the complexity of different families of grammars in terms of the measure "Var".

Example 6.3. Let $\Sigma_i = \{a_i, b_i, c_i, d_i, e_i\}$, for $1 \leq i \leq n$, be n pairwise disjoint alphabets. Consider the language $L'_i = \{e_i a_i^m b_i^m \mid m \geq 1\} \cup \{e_i c_i^m d_i^m \mid m \geq 1\}$ and let $L_n = \bigcup_{i=1}^n L'_i$ (Exercise 4.1.6 in [10]). Then we show the following:

- $Var_{CF}(L_n) = 2n + 1$,
- $Var_{(RC,CF,ac)}(L_n) \leq n + 3$,
- $Var_{(P,CF,ac)}(L_n) \leq 2$,
- $Var_{(BCG,CF)}(L_n) = 1$.

To show that $Var_{CF}(L_n) = 2n + 1$, consider the context-free grammar
$G = (\{S\} \cup \bigcup_{i=1}^n \{A_i, B_i\}, \bigcup_{i=1}^n \Sigma_i, R, S)$, where R contains the following rules:

$$\begin{aligned} &S \to e_i A_i \mid e_i B_i, && 1 \leq i \leq n, \\ &A_i \to a_i A_i b_i \mid a_i b_i, && 1 \leq i \leq n, \\ &B_i \to c_i B_i d_i \mid c_i d_i, && 1 \leq i \leq n. \end{aligned}$$

This shows that $Var_{CF}(L_n) \leq 2n + 1$. Next we show that no context-free grammar with fewer than $2n + 1$ nonterminals can generate L_n.

Let $\Sigma''_i = \Sigma_i \setminus \{e_i\}$ and $L''_n = \bigcup_{i=1}^n \{a_i^m b_i^m \mid m \geq 1\} \cup \{c_i^m d_i^m \mid m \geq 1\}$. Let $G'' = (V, \bigcup_{i=1}^n \Sigma''_i, R'', S'')$ be a context-free grammar without ε-productions that generates L''_n. Let $w = a_i^l b_i^l$ be a sufficiently long string in L''_n. Consider a derivation for w: $S'' \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w_d \Rightarrow w$. Since G'' is context-free, w_d is of the form $w'_d A_i w''_d$, where $w'_d w''_d$ is one of the words described by $a_i^* b_i^*$, $w'_d w''_d \neq \varepsilon$, $A_i \in V$, and there is a rule $A_i \to u_d$ where u_d is one of the words described by $a_i^* b_i^*$.

Suppose there is a nonterminal $B_j \in V$ such that $B_j = A_i$, and that there is a rule $B_j \to v_d$, where v_d is one of the words described by the regular expression $c_j^* d_j^*$. Then there is a

derivation $S'' \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w'_d B_j w''_d \Rightarrow w'_d v_d w''_d$. But $w'_d v_d w''_d$ is not in L''_n. Similarly, if there is nonterminal A_k with $k \neq i$, but such that $A_k = A_i$, and there is a rule $A_k \rightarrow t_d$, with t_d one of the words described by $a_k^* b_k^*$, then the derivation $S'' \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w'_d A_k w''_d \Rightarrow w'_d t_d w''_d$ generates a word that is not in L''_n. Note that the only possibility for the termination rules is to have right-hand side described by $c_j^* d_j^*$ or $a_k^* b_k^*$.

Suppose $S'' = A_i$. Since there is a derivation $S'' \Rightarrow^* c_j d_j$, the string $w'_d c_j d_j w''_d \in L''_n$, but this is a contradiction.

This argument shows that for each pair a_i, b_i and each pair c_j, d_j, we need at least one nonterminal. The start nonterminal S'' must also be distinct from all other nonterminals. Thus, no context-free grammar with fewer than $2n+1$ nonterminals can generate L''_n.

Next we show that L_n cannot be generated by fewer than $2n+1$ nonterminals.

Since the rule that generates e_i must be applied once during all derivations, we can generate e_i from S'' (without adding additional nonterminals). All $A'_i s$ and $B'_j s$ for $1 \leq i, j \leq n$ are essential in the grammar, since if we remove any, say A_i, then all words of the form $e_i a_i^m b_i^m$ cannot be generated by the grammar. Hence no context-free grammar with fewer than $2n+1$ nonterminals can generate L_n.

To show that $Var_{(RC,CF,ac)}(L_n) \leq n+3$, it suffices to introduce a random context grammar with context-free rules and $n+3$ nonterminals that generates L_n. Consider the random context grammar $G = (\{S, X, Y\} \cup \bigcup_{i=1}^{n} \{E_i\}, \bigcup_{i=1}^{n} \Sigma_i, R, S)$, where R contains the following rules:

$$
\begin{array}{llll}
(S \rightarrow e_i E_i X \mid e_i E_i Y, & \emptyset, & \emptyset), & 1 \leq i \leq n, \\
(E_i \rightarrow a_i E_i b_i, \mid a_i b_i, & \{X\}, & \emptyset), & 1 \leq i \leq n, \\
(E_i \rightarrow c_i E_i d_i, \mid c_i d_i, & \{Y\}, & \emptyset), & 1 \leq i \leq n, \\
(X \rightarrow \varepsilon, & \emptyset, & \bigcup_{i=1}^{n} \{E_i\}), & \\
(Y \rightarrow \varepsilon, & \emptyset, & \bigcup_{i=1}^{n} \{E_i\}). &
\end{array}
$$

This shows that $Var_{(RC,CF,ac)}(L_n) \leq n+3$.

To show that $Var_{(P,CF,ac)}(L_n) \leq 2$, note that the programmed grammar that is defined by $G = (\{S, S'\}, \bigcup_{i=1}^{n} \Sigma_i, R, S)$, where R has the rules below, generates L_n:

$$
\begin{array}{llll}
(i_1 : S \rightarrow e_i S', & \{i_2, i_3\}, & \emptyset), & 1 \leq i \leq n, \\
(i_2 : S' \rightarrow a_i S' b_i \mid a_i b_i, & \{i_2\}, & \emptyset), & 1 \leq i \leq n, \\
(i_3 : S' \rightarrow c_i S' d_i \mid c_i d_i, & \{i_3\}, & \emptyset), & 1 \leq i \leq n.
\end{array}
$$

Thus, $Var_{(P,CF,ac)}(L_n) \leq 2$.

Next we show $Var_{(BCG,CF)}(L_n) = 1$. To show this consider the bag context grammar $G = (\{S\}, \bigcup_{i=1}^{n} \Sigma_i, R, S, 0)$ where R contains:

$$
\begin{array}{lll}
S \rightarrow e_i S & (0,0,i), & 1 \leq i \leq n, \\
S \rightarrow e_i S & (0,0,-i), & 1 \leq i \leq n, \\
S \rightarrow a_i S b_i \mid a_i b_i & (i,i,0), & 1 \leq i \leq n, \\
S \rightarrow c_i S d_i \mid c_i d_i & (-i,-i,0), & 1 \leq i \leq n.
\end{array}
$$

This shows that $Var_{(BCG,CF)}(L_n) \leq 1$. Since $Var_X(L) \geq 1$ for all X and L, we conclude that $Var_{(BCG,CF)}(L_n) = 1$. □

Let $\mathcal{L}_f(PN)$ and $\mathcal{L}_t(PN)$ be the classes of free and terminal Petri net languages, respectively. From Corollary 4.6 it should be noted that for all $L \in \mathcal{L}_f(PN)$ we have $Var_{(BCG,REG)}(L) = 1$ and for all $L \in \mathcal{L}_t(PN)$ we have $Var_{(BCG,REG)}(L) = 1$.

This chapter discussed decidability and complexity in grammars with regulated rewriting mechanisms. Different decidability properties of grammars with regulated rewriting were discussed. We considered membership, emptiness, finiteness and equivalence problems and showed how each class of grammars with regulated rewriting considered in Chapter 3 behaves in terms of these decidability problems. We considered Petri nets and vector addition systems as tools to show decidability. We used reachability in Petri nets to prove decidability of emptiness for $rPcg$-2 (random permitting context grammars with context free rules). The relation between Groebner bases and decidability of reachability in Petri nets was also considered. We showed how Groebner bases can be used to decide the reachability in reversible Petri nets. Complexity of grammars with regulated rewriting was the final topic in this chapter. We considered the nonterminal complexity of various classes of grammars with regulated rewriting.

Chapter 7

Conclusion and future work

This thesis gave a survey of regulated rewriting. We motivated this survey by giving circumstances where context-free grammars are not sufficient and more descriptive grammars are required. Then in the chapter on generating mechanisms, we considered bag context grammars and compared them with other well-known classes of grammars with regulated rewriting mechanisms. We investigated the relationship between counter automata (blind and partially blind) and bag context grammars. It turned out that regular bag context grammars with two bag positions generate the class of recursively enumerable languages. The relationship between Petri net automata and bag context grammars was also investigated. We showed how to simulate k-place Petri net automata with k-BCGs considering the languages css, L_m, L_f, and L_t associated with Petri nets. We modified the pumping lemma for $rPcl$-2 as introduced in [19]. We presented the pumping lemma in terms of string homomorphisms, and gave an illustration of this lemma. Next we considered decidability and complexity properties of grammars with regulated rewriting. We showed how each of the various classes of grammars considered in Chapter 3 behaves in terms of the well-known decidability problems. We considered the relationship between Petri nets, vector addition systems, Groebner bases and various decidability problems. We showed how we can use Petri nets to show that the emptiness problem is decidable for $rPcg$-2 and also considered the membership problem for $rPcg$-2. Finally, we considered the complexity of grammars with regulated rewriting mechanisms, in that we compared different classes of grammars in terms of their nonterminal complexity.

Since regulated rewriting is a well-researched area, there are various topics that are not covered in this thesis. Next we make some suggestions regarding problems to be considered for future research.

We investigated the class of bag context grammars and it turned out that the regular bag

context grammars can generate the class of recursively enumerable languages. Thus, restrictions are needed on the class of bag context grammars in order to find subclasses that are descriptive enough and also have interesting theoretical properties such as the decidability of membership and good complexity properties.

We showed how to simulate counter or Petri net automata with regular bag context grammars. A natural question is to ask what type of restrictions can be placed on regular bag context grammars, so that we can find for each grammar G from this subclass an equivalent counter or Petri net automaton.

Also, the bag context grammars that were considered in the simulation of the recognizing devices allowed ε-productions. What if we consider bag context grammars without ε-productions?

Another topic of interest that concerns bag context grammars is the pumping lemma. The pumping lemma for $rPcl$-2 was introduced in [19]. It should be interesting to investigate the class of bag context grammar for which a similar pumping or shrinking lemma can be established.

Bibliography

[1] B. Atcheson, S. Ewert, and D. Shell. A Note on the Generative Capacity of Random Context. *South African Computer Journal*, 36:95-98, 2006.

[2] G.J. Bex, W. Martens, F. Neven, and T. Schwentick. Expressiveness and Complexity of XML Schema. *ACM Transactions on Database Systems*, 31(3):770-813, 2006.

[3] L. Blair, A. Chandler, A. Heyworth, and D. Seward. Testing Petri Nets for Mobile Robots Using Gröbner Bases. *Proceedings of the Workshop on Software Engineering and Petri Nets, held at the 21st International Conference on Application and Theory of Petri Nets*, Aarhus, Denmark, 21-34, 2000.

[4] O. Caprotti, A. Ferscha, and H. Hong. Reachability Test in Petri Nets by Gröbner Bases. *Parallel Symbolic Computation, No S5302-PHY*, 1995.

[5] M. Češka and V. Marek. Petri Nets and Random-Context Grammars. *Proceedings of the 35th Spring Conference: Modelling and Simulation of Systems - MOSIS'01*, 145-152, 2001.

[6] A. Chandler and A. Heyworth. Gröbner Basis Procedures for Testing Petri Nets. *eprint arXiv:math/0002119, UWB Math Preprint 99.11*, 2007.

[7] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT press, 1999.

[8] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms : An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer, 2007.

[9] J. Dassow. *Grammars with Regulated Rewriting*. [online]. Available from: <http://theo.cs.uni-magdeburg.de/dassow/tarraphd.pdf>. [Accessed 25 January 2007].

[10] J. Dassow and G. Păun. *Regulated Rewriting in Formal Language Theory*. EATCS Monographs in Theoretical Computer Science 18, Springer-Verlag, 1989.

[11] F. Drewes. *Grammatical Picture Generation: A Tree-based Approach*. Texts in Theoretical Computer Science: an EATCS Series, Springer, 2006.

[12] F. Drewes. *The TREEBAG Homepage*. [online]. Available from: <http://www.informatik.uni-bremen.de/theorie/treebag/download.html>. [Accessed 26 February 2007].

[13] F. Drewes and J. Engelfriet. Branching Synchronization Grammars with Nested Tables. *Journal of Computer and System Sciences*, 68(3):611-656, 2004.

[14] F. Drewes, C du Toit, S Ewert, B van der Merwe, and A van der Walt. Bag Context Tree Grammars. *Tenth International Conference on Developments in Language Theory, Lecture Notes in Computer Science*, 4036:226-237, 2006.

[15] F. Drewes, C du Toit, S. Ewert, J. Högberg, B. van der Merwe, and A. van der Walt. Random Context Tree Grammars and Tree Transducers. *South African Computer Journal*, 34:11-25, 2005.

[16] J. Engelfriet. Tree Automata and Tree Grammars. *Lecture Notes in Computer Science*, Institute of Mathematics, University of Aarhus, 1975.

[17] J. Esparza and M. Nielsen. Decidability Issues for Petri Nets - a Survey. *Bulletin of the European Association for Theoretical Computer Science*, 52:245-262, 1994.

[18] S. Ewert and A. van der Walt. A Shrinking Lemma for Random Forbidding Context Languages. *Theoretical Computer Science*, 237(1-2):149-158, 2000.

[19] S. Ewert and A. van der Walt. A Pumping Lemma for Random Permitting Context Languages. *Theoretical Computer Science*, 270:959-967, 2002.

[20] A. Fleck. *Formal Models of Computation*. World Scientific Publishing, 2001.

[21] R. Gilman. *Counter Machines*. Steven Institute of Technology. [online]. Available from: <www.math.stevens.edu/∼rgilman/ccny/counters.pdf>. [Accessed 18 January 2007].

[22] S. Greibach. Remarks on Blind and Partially Blind One-way Multicounter Machines. *Theoretical Computer Science*, 7:311-324, 1978.

[23] G. Greuel, G. Pfister, and H. Schöenemann. *SINGULAR*. [online]. Available from: <http://www.singular.uni-kl.de/download.html>. [Accessed 19 October 2007].

[24] D. Hauschildt and M. Jantzen. Petri Net Algorithms in the Theory of Matrix Grammars. *Acta Informatica*, 31(8):719-728, 1994.

[25] H.J. Hoogeboom. Context-Free Valence Grammars-Revisted. *Developments in Language Theory, Lecture Notes in Computer Science*, 2295:293-303, 2002.

[26] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing, 1979.

[27] A.J. Jones. *Formal Languages and Automata*. School of Computer Science, Cardiff University. [online]. Available from: <http:// users.cs.cf.ac.uk/ Antonia.J.Jones/ Lectures/ AutomataTheory/ AutomataTheory.pdf>. [Accessed 4 October 2006].

[28] L. Kari and P. Prusinkiewicz. Subapical Bracketed L-systems. In J. Cuny, H. Ehrig, G. Engles, and G. Rozenberg, editors, *Grammars and their Application to Computer Science, Lecture Notes in Computer Science*, 1073:550-564, 1996.

[29] C. Martin-Vide, V. Mitrana, eds. and G. Păun. Chapter 6 in: Formal Languages and Applications. *Studies in Fuzziness and Soft Computing*, Springer, Berlin, 148:117-138, 2004.

[30] A. Meduna. *Automata and Languages, Theory and Applications*. Springer, 2000.

[31] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541-580, 1989.

[32] F. Neven. Automata, Logic and XML. University of Limburg. [online]. Available from: <http:// alpha.uhasselt.be/ ∼lucg5503/csl2002.ps>. [Accessed 16 July 2007].

[33] J. Nievergelt. *Finite Automata with External Storage*. Institute of Theoretical Computer Science, Department of Computer Science, ETH Zürich. [online]. Available from: <www.jn.inf.ethz.ch/education/script/chapter4.pdf>. [Accessed 16 January 2007].

[34] G. Ochoa. *An Introduction to Lindenmayer Systems*. School of Cognitive and Computing Sciences, University of Sussex. [online]. Available from: <http://www.biologie.uni-hamburg.de/ b-online/e28_3/lsys.html>. [Accessed 29 March 2007].

[35] R.J. Parikh. On Context-Free Languages. *Journal of the Association for Computing Machinery*, 13(4):570-581, 1966.

[36] M. Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, 2006.