Zbyněk Křivka

Rewriting Systems with Restricted Configurations

Dissertation thesis

Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic

Published by Faculty of Information Technology,
Brno University of Technology, Brno, Czech Republic

Printed by MJ servis, spol. s r.o.

ISBN 978-80-214-3722-7

Preface

This theoretically oriented dissertation discusses rewriting systems, including various automata and grammars. It concentrates its attention upon their combination. More specifically, the central role of the present dissertation plays the general notion of a configuration as an instantaneous description of a rewriting system. Based upon various restrictions placed upon configurations and rewriting modes, the systems are classified and studied. Apart from this major topic, the dissertation also discusses dynamic complexity, which is based upon metrics placed upon the process of yielding strings.

As its fundamental topic, the dissertation discusses #-rewriting system, reducing deep pushdown automaton, and pushdown automata with restricted pushdowns. In addition, it studies some variants of #-rewriting systems, including n-right linear and generalized #-rewriting system. In general, the dissertation demonstrates how the generative power of the systems under discussion depends upon the restrictions placed upon them. In terms of dynamic complexity, it discusses a close relationship between various rewriting systems, including newly introduced systems. As its main result, the dissertation demonstrates several infinite hierarchies of language families defined by rewriting systems in dependency on their restrictions.

The conclusion demonstrates the application-important properties of the systems discussed in the dissertation. It sketches two new types of determinism and canonical rewriting; then, it demonstrates their potential practical usage.

Acknowledgment

This work was supported by GRAFO 2008 grant, the Czech Ministry of Education under the Research Plan No. MSM 0021630528 (Security-Oriented Research in Information Technology), and the Czech Ministry of Education grant No. 2C06008 (Virtual Laboratory of Microprocessor Technology Application).

I thank to Alexander Meduna, my teacher and friend. He supported me and inspired me during the whole time when I wrote this book. I am looking forward to our next collaboration. My thanks also go to Pavel Bělohlávek and Petr Sosík for their criticism of this book.

Contents

Part IV Conclusion

Part I

Present State

1

Introduction

The theory of formal languages is a mathematically oriented area of computer science. The importance of this part follows from the fact that every problem or task that can be mathematically described is possible to specify by a language. The fundamental principle of the description utilizes the fact that every solution to a problem can be described by a sentence of a language. Because of this, there is the way of unifying the question about the solution to each given problem: Does the sentence that represents the sought solution belong to the language that describes all possible solutions to the problem? From this point of view, we can look at searching the solutions in two ways:

a) we verify the sentence that represents the problem and we try to say if the sentence belongs to the language describing all possible solutions to the problem (procedural approach);
b) we describe/search all valid solutions to the given problem i.e. we search the solution language description (declarative approach).

The theory of formal languages explores the way of solving the problem that is described by formally specified language. Therefore, it makes sense of defining problems like these in terms of formal languages because we can use the properties and the automatic processing that is available for the particular language family to which the language describing this problem belongs. There is a proportion between the problem complexity and solution complexity. The same proportion holds for the corresponding language representation of the problem and the complexity of the language processing. For example, the natural language description of the problem with its complexity and ambiguity is more complicated to process by machine than, for instance, by simple regular expression describing variable identifiers.

A language, which carries information and/or possible solutions, is based on the fundamental mathematical notion of a set. Since a non-trivial language is often an infinite set, for its finite description we need formal models.

Formal models serve for the exact mathematical description of languages. We can classify them into four basic categories:

- set enumeration that rigorously defines all items of the set; it is useful only for finite sets;
- set construct that is built from the list of mathematical or natural language requirements and conditions that make some demands on the set items i.e. sentences of the defined language; for instance, $\{w|\ w$ has length divisible by 3, where w is a sentence over given alphabet of symbols$\}$;
- usage of algebraic operations over already defined languages for the language specification; for instance, $L = L_1 \cup L_2$, where L_1, L_2 are languages;
- rewriting systems that define algorithmic mechanism specifying how to process language sentences. This approach is very important in practice, so the theory of formal languages deals with rewriting systems a lot.

A rewriting system changes its inner state by rewriting. The rewriting is an operation of the replacement of a substring of the system inner state by another string of symbols.

The rewriting system is defined by two basic components—total alphabet and finite set of rules. The total alphabet contains all symbols that can appear in the input, output, or inner state of the rewriting system. The finite set of rules dictates possible rewritings (changes) of the system inner state.

The most common rewriting systems are grammars and automata.

For instance, let us have some well-known type of grammars, context-free grammars. A context-free grammar consists of a total alphabet, terminal alphabet that is a subset of the total alphabet, starting symbol (axiom), and finite set of rewriting rules. The grammar starts from its starting symbol in its inner state (called sentential form) and rewrites nonterminals (symbols of the total alphabet that are not terminals) according to its rewriting rules. By this rewriting mechanism, the grammar generates a string of terminal symbols. The set of terminal strings generated in this way defines the language generated by this context-free grammar.

An automaton works in a different and more practical way. That is, the automaton verifies the language sentence by a step-by-step reading and processing according to its transitions in its state control. We say that the grammar generates the language and that the automaton accepts the language.

To illustrate the notion of an automaton as a rewriting system, consider a finite automaton that consists of: a finite set of states, Q, one of which is defined as a starting state and some states are specified as final states; an alphabet of input symbols, T, where union of Q and T creates the total alphabet of the system; and a set of rewriting rules allowing to make transitions.

During a move (or transition), the automaton changes its current state and reads an input symbol from the input tape. The input tape contains the input string that represents the processed sentence. If with an input string, the automaton makes the sequence of moves according to its rules, so it starts from the starting state, reads the entire input string from the input tape, and reaches a final state, then the automaton accepts the input string. The

set of all strings accepted in this way represents the language that the finite automaton defines.

Considering the finite automaton as an instance of a rewriting system, the inner state is composed of the input string and the current state. This pair is called a configuration. In a suitable configuration representation, every move does the rewriting of the configuration substring that consists of the current state and the current input symbol to a string that contains only target state of the used rewriting rule. An input symbol is read by such an action.

As demonstrated above, the combination of grammars and automata that are both rewriting systems leads again to a rewriting system, and this work focuses on this combination of rewriting systems. The formal language theory has paid little attention to this area. The combinations and modifications of grammars and automata were mostly studied separately and independently.

While studying the combination of automata and grammars, we find useful to define all notions and properties in the context of rewriting systems. The basic idea is to work with notions related to various types of rewriting systems in the uniform way. We are heading to the unification of both essential approaches. For example, the notion of configuration is used in the general way to denote the description of the inner state of a rewriting system, so even the notion of sentential form will be, from the grammar approach, generally called a configuration.

The combination of automata and grammars, including the unification of notations, is demonstrated through the whole book. It is particularly emphasized in the definitions of two new rewriting systems (see Chapter 4): #-rewriting systems and reducing deep pushdown automata.

More specifically, the majority of languages and language families can be described or characterized by more than one formal model (or more specifically, rewriting system). In the appropriate formal model selection, it is very important to consider the property of the complexity of the model.

The complexity theory is very important part of the theory of formal languages. We distinguish two branches in the complexity theory: the descriptional complexity (i.e. static) and dynamic time and space complexity:

- The descriptional complexity deals with the space complexity of the description of the language itself, such as the necessary number of rules or nonterminals for the description of a given language or a language family;
- The space and time complexity is focused on the efficiency of the language sentence processing, thus this type of complexity is more important in practice than the previous one.

The disadvantage of the space and time complexity is that it deals with the formal model characterization in a rather rough way. It only finds out the complexity class (e.g. asymptotic) of the given model. For instance, in the worst case, a sentence of a context-free language can be processed in cubic time complexity depending on the length of the sentence. Moreover, there is

a trend to improve studied metrics of investigated formal models or their way of processing in the descriptional and space and time complexity. In addition, there is a requirement not influence the generative or acceptance power of these models.

This book uses fine-grained and slightly different approach to the complexity problem, more specifically, space complexity. Some specific metrics related to the space complexity are restricted. Then, the resulting language families defined by these restricted rewriting systems are investigated. To achieve this goal, Chapter 3 introduces the classification of rewriting systems and the notion of dynamic complexity.

As already mentioned, we study the combinations of grammars and automata. These combinations are inspired by regulated grammars and automata. The first types of the regulated grammars were investigated in the seventies of the twentieth century (for instance, matrix or programmed grammars). The main motivation for the introduction of regulating mechanisms to grammars is to increase their generative power by some small modification of their computational mechanism. All these regulated grammars are based on a quite simple context-free grammar, or more precisely context-free rule cores. When we generalize the notion of the regulation, we get the mechanism of general formal model restrictions, because the regulation really restricts the formal model. For instance, the mechanism of the selection of rules that can be applied in the next derivation step in a programmed grammar is a regulation and restriction.

We distinguish the following restrictions of formal models:

- restriction of the rewriting systems (the topic of this book);
- restriction of the derivation domain (for example, instead of alphabet symbols, the words of a finite language can be used to construct strings, or we can define a language based on free groups instead of free monoids);
- others (for example, the requirements of the satisfaction of several algebraic properties, such as the closure with respect to the operation substitution).

The fundamental part of the study is usually concerned about the first item of the previous classification. It contains for instance (1) restriction of the changes between configurations or directly (2) the restriction of configurations themselves. The second case in the context of the existing rewriting systems establishes the target research area for this monograph.

1.1 Chapters Survey

This book consists of seven chapters divided into four parts:

1. Present State (Chapters 1, 2, and 3)
2. New Formal Models and Their Restrictions (Chapters 4 and 5)

3. Relationship with Practice (Chapter 6)
4. Conclusion (Chapter 7)

The first chapter introduces the work as a whole, including the motivation.

The second chapter revises all notions and definitions for the further use in the text (from fundamental to advanced). Besides the definitions, we generalize some other notions from formal languages in the form of concepts, such as formal model, instance, and configuration. Formally, we discuss the unification definition of a formal model—rewriting system—and its variants: grammars and automata. Let us emphasize that the following chapters primarily contain the basic research of the author (if not mentioned otherwise).

The third chapter classifies formal models and rewriting systems, respectively. It focuses on their restricting. The notion restriction is understood in more general context and it is closely associated with the complexity notion. From the complexity point of view, the formal model restricting is divided into static, dynamic, and hybrid restrictions.

The text concentrates on the results concerning hybrid and dynamic restrictions of rewriting systems. More specifically, we deal with the restriction of the sequence of applied rewriting rules and configuration restriction. At the end of the chapter, the relation of the configuration restriction to the time and space complexity and to practical usage of rewriting systems is stressed. For more concrete approach to the space complexity, we introduce the notion of dynamic complexity that studies the space complexity with emphasis on the particular way of the configuration restriction, such as the limitation or finite index.

Naturally, since the dynamic complexity depends on the static complexity, the border between dynamic, static, and hybrid restricting is not always so sharp.

Chapter 4 defines new types of regulated rewriting systems that are, from the point of view of the dynamic complexity, investigated more deeply in the following chapters. Firstly, we introduce #-rewriting systems that are the regulated rewriting systems combining the properties of grammars and automata. Like grammars, #-rewriting systems are generative devices, and like automata, they implement finite-state control. In addition, they contain the static restriction to only one type of symbol that can be rewritten by the rewriting rule that determines which symbol from the beginning of the configuration (from the left) will be replaced. Apart from the basic #-rewriting system based on the context-free form of rules, we introduce right-linear and context-sensitive variants as well. Next studied formal model is based on the reducing modification of deep pushdown automata. This is a classical example of the dynamic restriction. The last explored formal model is a pushdown automaton with a restricted pushdown content that is regulated by the language that says which strings in the pushdown are allowed during the sentence processing.

The majority of formal language properties are related to entire language families. The language family is usually specified by the formal model that is able to describe exactly the languages that belong to this family. The power of formal models (and the language families described by them) is a crucial property for investigating other information and properties of the family or a particular language of the family. That is the reason why it is important to study generative power for new rewriting systems too. This topic is discussed in the fifth chapter that summarizes the results and properties of new and existing investigated formal models with the emphasis on their restricting. The construction proofs are included. In some cases the restriction of the dynamic complexity (for example n-limitation or finite index) leads to the infinite hierarchies that are very important properties to study in formal languages to investigate expressing abilities of the given formal model. The proofs of the infinite hierarchies of the models are based on the demonstration of the equivalence with another formal model that creates the same infinite hierarchy, so we obtain a new property for our model as well.

Although the text is strongly theoretically oriented, the sixth chapter studies the fundamental properties of introduced rewriting systems in a practical usage. We investigate the possibilities of a deterministic and canonical rewriting that are essential, for instance, for the context-free language processing.

The last, seventh, chapter drafts the future development of the research in the studied area. Some expected results and hypothesis are presented. They only wait to be proved. Apart from hypothesis, some essential and related open problems are discussed there as well.

1.2 Conventions

To improve the readability, orientation, and uniformity of the text, we distinguish two types of definitions: Definitions and Concepts. The less formal definitions are denoted as Concepts. In addition, the notes about the form of the text are denoted as Conventions.

Convention 1.1. The form of some notations is inspired by books [Med00, MŠ05] and [Sal69, Sal73, Sal85].

2

Preliminaries

In this chapter, we define notions from the theory of formal languages. The basic notions are defined in a very brief way. For certain introduction to the topic of formal languages and for fully understanding of defined terms see [Lin90, Med00]. Some compiler issues are covered in [ALSU06, AU72, AU73, AU77].

The advanced formal models, that are in most cases studied at PhD level, are described in more detail (see [DP89, Med04, RS97]), including some examples.

2.1 Fundamental Mathematical Definitions

To unify all notions, let us revise some basic mathematical terms (see [Šla01]) that concern sets and relations. Apart from this, we define some denotations to be used throughout.

We use a notion of set in an intuitive meaning, such as in naive approach. Informally, a *set* Σ is a collection of elements taken from a pre-specified universe. If element a belongs to Σ, we write $a \in \Sigma$ and refer to a as a member of Σ. Otherwise, if a is not in Σ, we write $a \notin \Sigma$.

The *cardinality* of set Σ, $\mathrm{card}(\Sigma)$, is a number of members[1] in Σ. The set that has no member is called *empty set*, denoted by $\emptyset$. Note that $\mathrm{card}(\emptyset) = 0$. If the number of members in Σ is finite, then Σ is a *finite set*. Otherwise, Σ is an *infinite set*.

We can specify a finite set Σ by listing its members:

$$\Sigma = \{a_1, a_2, \ldots, a_n\},$$

[1] In the literature, there is an alternative denotation of the cardinality of a set, $|\Sigma|$, that is used exclusively in this work for the denotation of the length of a sequence.

where a_1 through a_n are all members of Σ. An infinite set Ω is usually specified by a property π, so Ω contains all elements from the universe that satisfy the property π (a *set constructor*). The symbolic denotation respects the following format:

$$\Omega = \{a|\ \pi(a)\}\,.$$

Sets whose members are other sets are usually called *families of sets* rather than sets of sets. Let Σ and Ω are two sets. Σ is *subset* of Ω, symbolically written as $\Sigma \subseteq \Omega$, if each member of Σ belongs to Ω too. Σ is *proper subset* of Ω, written as $\Sigma \subset \Omega$, if $\Sigma \subseteq \Omega$ and Ω contains the member that is not a member of Σ. Σ equals to Ω, denoted by $\Sigma = \Omega$, if $\Sigma \subseteq \Omega$ and $\Omega \subseteq \Sigma$. The *power set* of Σ, denoted as 2^{Σ}, is the set of all subsets of Σ.

The union, intersection, and difference of Σ and Ω are denoted by $\Sigma \cup \Omega$, $\Sigma \cap \Omega$, and $\Sigma - \Omega$, respectively. These binary operations over sets are defined as

$$\Sigma \cup \Omega = \{a|\ a \in \Sigma \text{ or } a \in \Omega\}\,,$$
$$\Sigma \cap \Omega = \{a|\ a \in \Sigma \text{ and } a \in \Omega\}\,, \text{ and}$$
$$\Sigma - \Omega = \{a|\ a \in \Sigma \text{ and } a \notin \Omega\}\,.$$

Definition 2.1. Let $\mathbb{N}_0$ denotes a set of all non-negative integers, then define a set of all positive integers $\mathbb{N} = \mathbb{N}_0 - \{0\}$. Further, let $K \subseteq \mathbb{N}_0$ is a finite set. We define $\max(K)$ as the smallest number k such that for every $h \in K$, $k \geq h$, and $\min(K)$ as the greatest number l such that every for $h \in K$, $l \leq h$.

Now, we revise notions concerning relations, functions, and their properties.

Definition 2.2. Let a and b be two objects, then the pair (a, b) denotes the *ordered pair* that consists of the object a and b just in this order. Let A and B be two sets. We define *Cartesian product* of A and B, denoted as $A \times B$, as

$$A \times B = \{(a, b)|\ a \in A, b \in B\}\,.$$

A binary relation or, shortly, a *relation*, ρ from A to B is any subset of Cartesian product $A \times B$. Symbolically, that is

$$\rho \subseteq A \times B.$$

The *domain* of relation ρ, denoted by $\mathrm{Domain}(\rho)$, and the *range* of ρ, denoted by $\mathrm{Range}(\rho)$ are defined as

$$\mathrm{Domain}(\rho) = \{a|\ (a, b) \in \rho \text{ for some } b \in B\}$$

and

$$\mathrm{Range}(\rho) = \{b|\ (a, b) \in \rho \text{ for some } a \in A\}\,.$$

If $A = B$, ρ is said to be a *relation on* A.

We say that a relation ρ on A is (see [Šla01]):

a) *reflexive* if and only if $a\rho a$ for every $a \in A$;
b) *ireflexive* if and only if $(a, a) \notin \rho$ for every $a \in A$;
c) *symmetric* if and only if $a\rho b$ implies $b\rho a$ for every $a, b \in A$;
d) *antisymmetric* if and only if $a\rho b$ and $b\rho a$ implies $a = b$;
e) *transitive* if and only if $a\rho b$ and $b\rho c$ implies $a\rho c$.

A reflexive, antisymmetric, and transitive relation ρ on a set A is called a *partial order*. The pair (A, ρ) is called the *partially ordered set*.

Let A be a set and $\leq$ be a partial order on A. If for all $a, b \in A$ holds that $a \leq b$ or $b \leq a$, then $\leq$ is called a *linear order* and $(A, \leq)$ is the *linearly ordered set*.

An ireflexive and transitive relation $<$ on A is called a *strict order*, the pair $(A, <)$ is the *strictly partially ordered set*.

Definition 2.3. Let $(X, \leq)$ be a partially ordered set, $A \subseteq X$. We say that $x \in X$ is an *upper bound* (*lower bound*, respectively) of A if for all $a \in A$ holds $a \leq x$ ($x \leq a$, respectively).

Next, $x \in X$ is called a *maximum* (*minimum*, respectively) of $(X, \leq)$, if for every $y \in X$ holds $y \leq x$ ($x \leq y$, respectively). If there is the least upper bound of A, we denote it as $\sup(A)$ and call it a *supremum* of A. By duality principle of the order theory, if there is the greatest lower bound of A, we denote it $\inf(A)$ and call it an *infimum* of A.

Definition 2.4. A *function* (or *mapping*) from A to B is a relation ϕ from A to B such that

$$\text{for every } a \in A,\ \text{card}(\{b|\ b \in B,\ (a, b) \in \phi\}) \leq 1.$$

Let ϕ be a function from A to B. If $\text{Domain}(\phi) = A$, ϕ is a *total function*; otherwise, ϕ is a *partial function*. If for every $b \in B$, $\text{card}(\{a|\ a \in A,\ (a, b) \in \phi\}) \leq 1$, ϕ is an *injection*. If for every $b \in B$, $\text{card}(\{a|\ a \in A,\ (a, b) \in \phi\}) \geq 1$, ϕ is a *surjection*. If ϕ is both injection and surjection, we say ϕ is a *bijection*.

Instead of $(a, b) \in \rho$, we often write $b \in \rho(a)$ or $a\rho b$. In other words, $(a, b) \in \rho$, $b \in \rho(a)$, and $a\rho b$ are used interchangeably. We prefer the latter. Moreover, if ϕ is a function, we usually write $\rho(a) = b$.

Let ρ be a relation on a set A. For $k \geq 0$, the *k-fold product* of ρ, ρ^k, is recursively defined as

(1) $a\rho^0 b$ if and only if $a = b$,
(2) $a\rho^1 b$ if and only if $a\rho b$, and
(3) $a\rho^k b$ if and only if $a\rho c$ and $c\rho^{k-1} b$, for some c and $k \geq 2$.

The *transitive closure* of ρ, ρ^+, is defined as $a\rho^+ b$ if and only if $a\rho^k b$ for some $k \geq 1$. The *reflexive-transitive closure* of ρ, ρ^*, is defined as $a\rho^k b$ for some $k \geq 0$.

2.2 Formal Languages

The notions concerning languages, families of languages, and language operations follow.

Definition 2.5. A *sequence* is an intuitive well-known notion describing a list of items from some universe. A sequence is *finite* if it represents a finite list of elements; otherwise, it is *infinite*. The *length* of a finite sequence x, denoted by $|x|$, is the number of elements in x. The empty sequence, denoted by ε, is the sequence consisting of no element. That is, $|\varepsilon| = 0$. A finite sequence is usually specified by listing its elements. For instance, consider a finite sequence x specified as $x = 0, 1, 1, 0, 1$. That is, $|x| = 5$.

An *alphabet* T is a finite, non-empty set, whose members are called *symbols*. A finite sequence of symbols from T is a *string* (or *word*) over T. Specifically, ε is referred to as the *empty string*. By T^*, we denote the set of all strings over T. $T^+ = T^* - \{\varepsilon\}$.

Any subset $L \subseteq T^*$ is a *language* over T. If L is a finite set of strings, L is a *finite language*; otherwise, L is an *infinite language*. For instance, T^*, called the universal language over T, is an infinite language while $\emptyset$ and $\{\varepsilon\}$ are finite languages. Specifically, $\emptyset \neq \{\varepsilon\}$ because $\mathrm{card}(\emptyset) = 0 \neq \mathrm{card}(\{\varepsilon\}) = 1$. For a finite language L, $\max(L)$ denotes the length of the longest word in L. By analogy with the set theory, sets whose members are languages are called *families of languages*.

Convention 2.1. We omit all separating commas in strings. That is, we write $a_1 a_2 \ldots a_n$ instead of $a_1, a_2, \ldots, a_n$.

Definition 2.6. Let T be an alphabet, let $x, y \in T^*$ be two strings over T, and let $L, K \subseteq T^*$ be two languages over T. As languages are sets, all set operations apply to them, too. Specifically, $L \cup K$, $L \cap K$, and $L - K$ denote the union, intersection, and difference of languages L and K, respectively.

The most important operation is the *concatenation* of x with y, denoted by xy. xy is the string obtained by appending y to x. Notice that from an algebraic point of view, T^* and T^+ are *free monoid* and *free semigroup*, respectively, generated under the operation of concatenation. Observe that for every $w \in T^*$, $w\varepsilon = \varepsilon w = w$. The concatenation of L and K, denoted by LK, is defined as

$$LK = \{xy |\ x \in L,\ y \in K\}\,.$$

Apart from binary operations taken from the set theory, we also define some unary and binary operations over strings and languages.

Definition 2.7. Let T be an alphabet and let $x \in T^*$ and $L \subseteq T^*$. The *complement* of L, denoted by $\overline{L}$, is defined as $\overline{L} = T^* - L$.

The *reversal* of x, denoted by $\mathrm{rev}(x)$, is x written in the reverse order. That is, if $x = a_1 a_2 \dots a_n$, where $a_1, a_2, \dots, a_n \in T$, then $\mathrm{rev}(x) = a_n \dots a_2 a_1$.

For all $i \geq 0$, the ith *power of* x, denoted by x^i, is recursively defined as:

(1) $x^0 = \varepsilon$, and
(2) $x^i = xx^{i-1}$ for $i \geq 1$.

To demonstrate the recursive aspect, consider, for instance the ith power of x, x^i, with $i = 3$. By the second part of the definition, $x^3 = xx^2$. By repeated applying of the second part to x^2 and x^1, respectively, we obtain $x^2 = xx^1$ and $x^1 = xx^0$, respectively. By the first part of this definition, $x^0 = \varepsilon$. Thus, $x^1 = xx^0 = x\varepsilon = x$. Hence, $x^2 = xx^1 = xx$. Finally, $x^3 = xx^2 = xxx$.

By this convenient recursive method, we frequently introduce new notions, including the ith power of L, L^i which is defined as

(1) $L^0 = \{\varepsilon\}$, and
(2) $L^i = LL^{i-1}$, for $i \geq 1$.

The *closure* of L, L^*, is defined as

$$L^* = \bigcup_{i \geq 0} L^i,$$

and *positive closure* of L, L^+, is defined as

$$L^+ = \bigcup_{i \geq 1} L^i.$$

Notice that

$$L^+ = LL^* = L^*L$$

and

$$L^* = L^+ \cup \{\varepsilon\}.$$

Sometimes, the literature uses notions of reflexive-transitive and transitive closure of a language instead of closure and positive closure, respectively.

If there is $z \in T^*$ such that $xz = y$, x is a *prefix* of y; in addition, if $x \notin \{\varepsilon, y\}$, x is a *proper prefix* of y. By $\mathrm{prefixes}(y)$, we denote the set of all prefixes of y.

Analogically, if there is $z \in T^*$ such that $zx = y$, x is a *suffix* of y; in addition, if $x \notin \{\varepsilon, y\}$, x is a *proper suffix* of y. By $\mathrm{suffixes}(y)$, we denote the set of all suffixes of y.

For $n \geq 0$ and $x \in T^*$, $\mathrm{prefix}(x, n)$ and $\mathrm{suffix}(x, n)$, respectively, denote the prefix and suffix of x of length n, respectively. If $|x| < n$, $\mathrm{prefix}(x, n) = \mathrm{suffix}(x, n) = x$.

Let $w \in T^*$ and K be a string and a set such that $\{\varepsilon\} \subseteq K \subseteq T$. $\mathrm{maxprefix}(w, K)$ and $\mathrm{maxsuffix}(w, K)$, respectively, denote the longest prefix

and suffix of w such that every symbol of maxprefix(w, K) and maxsuffix(w, K) is a member of K, respectively.

If there are $u, v \in T^*$ such that $uxv = y$, x is a *substring* of y; in addition, if $x \notin \{\varepsilon, y\}$, x is a *proper substring* of y. The set of all substrings of y is denoted by sub(y).

Observe that for every string w,

$$\text{prefixes}(w) \subseteq \text{sub}(w),$$

$$\text{suffixes}(w) \subseteq \text{sub}(w),$$

and

$$\{\varepsilon, w\} \subseteq \text{prefixes}(w) \cap \text{suffixes}(w) \cap \text{sub}(w).$$

Let w be a non-empty string; then, first(w) and last(w), respectively, denote the leftmost symbol of w and rightmost symbol of w, respectively. For $1 \leq i \leq |w|$, sym(w, i) denotes the ith leftmost symbol of w. Given a string w, alph(w) is a set of all symbols occurring in w. For instance, alph$(aAbAaBCc) = \{a, b, c, A, B, C\}$. In terms of a language over T, L, we extend the definition as

$$\text{alph}(L) = \bigcup_{y \in L} \text{alph}(y).$$

For two strings x and y, where $|y| \geq 1$, occur(y, x) denotes the number of occurrences of y in x. A generalized form, occur(W, x), where W is a finite language and $\varepsilon \notin W$, denotes the number of all occurrences of substrings of x that belong to W. For instance, occur$(\{a, C\}, aAbAaBCc) = 3$.

Let T and U be two alphabets. A total function τ from T^* to 2^{U^*} such that $\tau(uv) = \tau(u)\tau(v)$ for every $u, v \in T^*$ is a *substitution* from T^* to U^*. By this definition, $\tau(\varepsilon) = \varepsilon$ and $\tau(a_1 a_2 \dots a_n) = \tau(a_1)\tau(a_2) \dots \tau(a_n)$, where $a_i \in T$, $1 \leq i \leq n$, for any $n \geq 1$. It means that τ is totally specified by the definition of $\tau(a)$ for every symbol $a \in T$. A total function χ from T^* to U^* such that $\chi(uv) = \chi(u)\chi(v)$ for every $u, v \in T^*$ is a *homomorphism* or, synonymously and briefly, a *morphism* from T^* to U^*. As any homomorphism is by the definition a special case of a substitution, we specify χ by analogy to the specification of τ.

Convention 2.2. Except when explicitly stated otherwise, we use lower-case and capital letters from the beginning of the Latin alphabet for denotation of symbols (for instance, a, b, c, A, B, C). On the other hand, we use lower-case letters from the end of the English alphabet or occasionally lower-case letters from the beginning of the Greek alphabet to denote strings (for instance, $u, v, w, x, y, z, \alpha, \beta, \gamma, \delta$). Numerical constants and variables are mostly denoted by lower-case Latin letters, such as i, j, k, m, n.

The last operation over strings is quite complicated, so we start with an informal description of the multisubstring operation. We take a source string, and we delete a finite number of its substrings. The result is called multisubstring. Other advanced operations over strings are investigated, for example, in [Kar91, MV04].

Definition 2.8. Let Σ is an alphabet and $x, y_i, z_j \in \Sigma^*$, where $1 \leq i \leq n$, $0 \leq j \leq n$, for some $n \in \{0, 1, \ldots, |x|\}$, and $x = z_0 y_1 z_1 \ldots y_n z_n$. Then, $y_1 y_2 \ldots y_n$ is a *multisubstring* of x. A function multisub(x) denotes the set of all multisubstrings of x.

Before we close this section, let us define n-tuples and other related notions.

Definition 2.9. Let $A_1, A_2, \ldots, A_n$ be sets and $a_1, a_2, \ldots, a_n$ be some objects, where $a_i \in A_i$ for every $1 \leq i \leq n$, $n \geq 1$. A non-empty sequence, $(a_1, a_2, \ldots, a_n)$, constructed from these objects is called an *n-tuple* and its members, $a_1, a_2, \ldots, a_n$, are called *components*.

Let $S = (a_1, a_2, \ldots, a_n)$ be an n-tuple. For $1 \leq i \leq n$, a_i denotes the ith component of S. A function component(S, i) denotes the ith leftmost component from S; that is, component(S, i) $= a_i$.

2.3 Formal Model Concept

This section presents the set of new concepts and definitions to unify automata-based and grammar-based approaches.

The literature is very often focused on particular part of the theory of formal languages and the unification with other parts is not necessary. On the other hand, this monograph concentrates on the possibilities of grammars and automata combinations. That is the reason why we need to specify common concepts and definitions for both automata and grammars approaches in the theory of formal languages.

There were several similar attempts of a unification in the theory of formal languages, such as [Fer97], [MŠ05], and [Woo76].

The informal specification of the concept for the notion of formal model follows. The general characteristics of the formal model even outside the theory of formal languages are (quoted from www.google.com):

Formal model is the mathematical representation of a concept of interest. The formal models are the basis of any language design and of the rigorous scientific investigation in general.

This book understands the notion of formal model as the mathematical mechanism that rigorously describes the way of defining sentences and/or languages (the sets of sentences). We can use a formal model as a mechanism that defines the language by algebraic operations and properties (declarative

approach), or procedural approach that is used by rewriting systems (e.g. automata and grammars).

Concept 2.1. A *formal model* is the mechanism that defines a language family by the model itself and its semantics.

For instance, context-free grammars, pushdown automata, Turing machines ([Med00]), and L-systems ([RS97]) are formal models.

Concept 2.2. An *instance of formal model* is the formal model concretization that defines a particular language. This language is called a *language defined by a formal model instance*.

For instance, a particular grammar or automaton that defines a specific language (for example, the context-free grammar that generates language $\{a^{2^n} \mid n \geq 1\}$ or the Turing machine that accepts language $\{b(a^i b)^{2n} \mid i \geq 1, n \geq 1\}$) are formal model instances.

2.3.1 Rewriting Systems

In the theory of formal languages, the most important type of formal models is the rewriting system that is based on finitely many rules that represent the algorithm processing a sentence of the defined language. The rewriting system rewrites inner state by the rules, which are of the form that corresponds to the particular formal model. The generality of rewriting systems makes them ideal for the unification of the formalization by two fundamental approaches, generative grammars and accepting automata ([Sal85]).

Definition 2.10. A *rewriting system* is a pair, $H = (\Sigma, R)$, where Σ is an alphabet and R is a finite relation on Σ^*. Σ is called the *total alphabet* of H. The members of R are called *rules* of H; in other words, R denotes the set of rules of H.

Convention 2.3. Rather than $(x, y) \in R$, we write $x \to y \in R$. For brevity, we denote $x \to y$ by a label r; that is, $r\colon x \to y$, and instead of $r\colon x \to y \in R$, we shortly write $r \in R$. Thus, the rule, $r\colon x \to y$, and the rule label, r, are fully interchangeable. For $r\colon x \to y \in R$, x and y represent the left-hand side and right-hand side of the rule r, respectively, and we denote them by $\mathrm{lhs}(r)$ and $\mathrm{rhs}(r)$. R^* denotes the set of all sequences of rules from R. By $\rho \in R^*$, we briefly express that ρ is a sequence consisting of $|\rho|$ rules from R. By analogy with the definition of a string (see Definition 2.5), we omit separating commas between individual rules (rule labels) from ρ. That is, instead of $\rho = r_1, r_2, \ldots, r_n$, we write $\rho = r_1 r_2 \ldots r_n$. To explicitly express that Σ and R are components of H, we denote the components by Σ_H and R_H, respectively.

In several rewriting systems, we define some notions in an analogical way. To keep this chapter as readable as possible, we present some implicit definitions of common notions that are valid for most rewriting systems and their variants.

Definition 2.11. Let $H = (\Sigma, R)$ is a rewriting system. *Rewriting relation* on Σ^* is denoted by $\Rightarrow$ and defined such that for every $u, v \in \Sigma^*$, $u \Rightarrow v$ in H, if and only if there is a rule, $x \to y \in R$, and $w, z \in \Sigma^*$ such that $u = wxz$ and $v = wyz$. Let $u, v \in \Sigma^*$. If $u \Rightarrow v$ in H, we say that H *directly rewrites* (*computes*) u to v by a *rewriting step* (*computational step*). As usual, for every $n \geq 0$, n-fold product of relation $\Rightarrow$ is denoted by $\Rightarrow^n$. If $u \Rightarrow^n v$, H rewrites u to v in n steps, called *n-step rewrite* or *n-step computation*. Further, the reflexive-transitive closure and transitive closure of $\Rightarrow$ is standardly denoted by $\Rightarrow^*$ and $\Rightarrow^+$, respectively. If $u \Rightarrow^* v$, we simply say that H *rewrites* (*computes*) u to v, and if $u \Rightarrow^+ v$, H rewrites u to v in a non-trivial way.

Sometimes, we need to explicitly specify the rules applied during the rewriting. Consider that H makes $u \Rightarrow v$ such that $u = wxz$, $v = wyz$, and H replaces x with the string y by application of the rule $r\colon x \to y \in R$. To express this rewriting step symbolically, we write $u \Rightarrow v\ [r]$ or, in detail, $wxz \Rightarrow wyz\ [r]$ in H, and we say that H directly rewrites wxy to wyz by r. More generally, let n be a non-negative integer, $w_0, w_1, \ldots, w_n$ be sequences, where $w_i \in \Sigma^*$, $0 \leq i \leq n$, and $r_j \in R$ for $1 \leq j \leq n$. If $w_{j-1} \Rightarrow w_j\ [r_j]$ in H for $1 \leq j \leq n$, H rewrites w_0 to w_n in n steps according to the sequence of rules, $r_1 r_2 \ldots r_n$, symbolically written as $w_0 \Rightarrow^n w_n\ [r_1 r_2 \ldots r_n]$ in H ($n = 0$ in case that $w_0 \Rightarrow^0 w_0\ [\varepsilon]$). The denotation $u \Rightarrow^* v\ [\rho]$, where $\rho \in R^*$, expresses that H makes $u \Rightarrow^* v$ by application of ρ; $u \Rightarrow^+ v\ [\rho]$ has analogical meaning ($\rho \neq \varepsilon$).

Convention 2.4. To specify which rewriting system was used to make another step, we write the denotation of the system into lower-left index of $\Rightarrow$. For instance, $x \Rightarrow_H y$ denotes the rewriting step made by H, $x \Rightarrow y$ in H.

Convention 2.5. If no confusion may exist, we omit the definition of reflexive-transitive and transitive closure, respectively.

We introduce the notion for the denotation and description of global inner state of an instance of rewriting system during its processing of a language sentence.

A finite sequence of the components that represents inner state of a formal model is called configuration. The notion of configuration is used in the theory of automata (see [Med00]), grammar systems (see [CVDKP94], [Kři04a], and the second volume of [RS97]), and often in combined systems, such as [Hoo87]

and [Kas70]. This monograph understands configuration as the notion that denotes inner state of any rewriting system (including classical grammars).

Concept 2.3. A *configuration* of an instance of a rewriting system (shortly rewriting system configuration or just configuration) is a well-enough specified inner state of the rewriting system instance that consists of a sub-state of all its computational parts, such as processed input (if there is some), usage and state of intern memory (if the model provides it), and the record of already written output of the system (if any). Mathematically speaking, the configuration of a rewriting system instance is an n-tuple of components with specified length[2], where $n \geq 1$.

Let $c = (\Gamma_1, \Gamma_2, \ldots, \Gamma_n)$ be a configuration. The length of the configuration c is defined as

$$|c| = |\Gamma_1| + |\Gamma_2| + \cdots + |\Gamma_n|.$$

Since the configuration, c, is a sequence, the items of c are called components and ith component is denoted by a function component(c, i).

A configuration component of a rewriting system can be finite or infinite. The finite component often represents finite-state control of the rewriting system (for example, in finite automata or state grammars).

A special case is a configuration in classical grammars, where $n = 1$. Such configuration is called *sentential form* and we do not use brackets around the string that represents it.

If a configuration component represents a potentially infinite sentential form or pushdown content, we call the component as *sentential configuration component* or *pushdown configuration component*, respectively.

The part of a rewriting rule that rewrites only symbols in the sentential configuration component or pushdown configuration component is said to be *rule core*.

Definition 2.12. Let H be a rewriting system and u, v be two configurations such that H rewrites u to v. Then, in the computation from u to v, u and v are *initial configuration* and *target configuration*, respectively. The first configuration of H in every computation is called *starting configuration*. The form of the starting configuration is specified by the particular formal model of H.

During the incremental fine down of the way of a family, language, or sentence definition, we get the notions that are suitable for the unification of properties common for the world of grammars and automata, which are mostly investigated separately.

[2] Some components can have infinitely many variants.

Convention 2.6. Every rewriting system investigated in this monograph is based on rules of the context-free form; that is, one symbol without any context-dependency is rewritten by a string of symbols, unless we explicitly state otherwise. Subsequently, these systems can be extended by a regulation or by another means of restrictions (see Chapter 3).

In most investigated rewriting systems, there are symbols that are required not to occur in the language sentences that are defined by these systems. In generative systems, they are often called *nonterminals*. If accepting systems are considered, we talk about non-input symbols (for instance, pushdown symbols in pushdown automata). To unify these notions, in general rewriting systems, we denote these symbols as variables (inspired from [Sud96]). The rest of the symbols of total alphabet are called passive symbols.

Concept 2.4. A symbol of total alphabet is a:

a) *variable* if the symbol in the configuration of the rewriting system can be replaced by a string of symbols. For instance, a nonterminal symbol or non-input pushdown symbol is a variable.
   1) *active symbol* if the variable can be rewritten in current configuration of the system. In most cases, we are interested in a particular occurrence of an active symbol.
   2) *potentially active symbol* if the variable is not an active symbol, but sometime in the future it can become the active symbol in a configuration that will come after the current one.

b) *passive symbol* if it is not a variable. Mostly, language sentences processed by a rewriting system consist only of passive symbols (for instance, terminal symbols or input symbols).

Notice that variables of a formal model may not be variables in another formal model (for example, pure grammar or L-system versus context-free grammar). Generally speaking, some symbols can even change their role during the processing of the configuration. For example, a nonterminal symbol of n-limited grammar can be active in one configuration, whereas in another configuration it can be only potentially active because we cannot rewrite it by the definition of limited grammar.

Convention 2.7. Let $H = (\Sigma, R)$ be a rewriting system and $r \in R$ its rule. To keep this monograph as readable as possible, we denote rules by a unique *label*. Further, we bound the rule by brackets, $\lfloor$ and $\rfloor$, if necessary. That is, the full form of a rewriting rule is written as

$$\lfloor r\colon \mathrm{lhs}(r) \to \mathrm{rhs}(r) \rfloor.$$

Sometimes, we reduce the denotation to $r\colon \mathrm{lhs}(r) \to \mathrm{rhs}(r)$ or just $\mathrm{lhs}(r) \to \mathrm{rhs}(r)$, respectively. If we use labels for rules, we define a function $\mathrm{lab}(\lfloor r\colon \mathrm{lhs}(r) \to \mathrm{rhs}(r) \rfloor) = r$ to get the label of the given rule. A function $\mathrm{Lab}(P) =$

$\{\text{lab}(r) \mid r \in P\}$ serves to obtain the set of labels from P, where $P \subseteq R$. In case of unambiguity in the construction proofs, we can use the rule itself and its label substitutionally.

The investigation of various rewriting systems and their combinations leads to many new formal models that can specify the same language family or in instances the same language, respectively. The instances of formal models that define the same language are called *equivalent instances* of formal model(s). More generally, the models defining the same language family are denoted as equivalent models or models having the same power.

2.4 Definitions of Basic Rewriting Systems

In this section, we focus on generally known and important formal models. First, we define basic versions of grammars and automata. At the end, we recall the notion of Chomsky hierarchy of formal language families based on different types of grammars or automata.

2.4.1 Grammars

This section revises basic types of grammars that are needed in the following chapters of this book.

More specifically, we discuss context-free, context-sensitive, and unrestricted grammars and the relations between them. In addition, the relations are summarized in the Chomsky hierarchy of formal language families. The definitions of grammars are constructed from the most general one. Then, by consecutive restrictions of rules, we define more specific grammars without any fundamental change of the universal definition of an unrestricted grammar or of a computation mechanism, such as a rewriting step called *derivation step*.

Notice that in grammars we prefer the notion of sentential form instead of the notion of configuration. We use the notion of nonterminal and terminal symbol instead of an active and passive symbol (see Concept 2.4), respectively.

Definition 2.13. An *unrestricted grammar* or *phrase-structure grammar* is a quadruple

$$G = (\Sigma, T, P, S),$$

where

Σ is the *total alphabet*,
T is the set of *terminals* ($T \subset \Sigma$),
$P \subseteq \Sigma^*(\Sigma - T)\Sigma^* \times \Sigma^*$ is a finite relation,
$S \in \Sigma - T$ is the *axiom* or *starting nonterminal* or *starting symbol* of G.

The symbols in $\Sigma - T$ are referred to as *nonterminals*. Furthermore, every $(x, y) \in P$ is called a *rule* or a *production* and written as

$$x \to y \in P.$$

Accordingly, the set P is according to its content called the *set of rules* in G. Let $p\colon x \to y \in P$ be a rule, then we set $\mathrm{lhs}(p) = x$ and $\mathrm{rhs}(p) = y$. The rewriting relation in G is called *direct derivation*. It is a binary relation on Σ^* denoted by $\Rightarrow_G$ and defined in the following way. Let $p\colon x \to y \in P$, $u, v, z_1, z_2 \in \Sigma^*$, and $u = z_1 x z_2$, $v = z_1 y z_2$; then,

$$u \Rightarrow_G v\ [p].$$

When there is no danger of confusion, we simplify $u \Rightarrow_G v\ [p]$ to $u \Rightarrow_G v$. We denote the k-fold product of $\Rightarrow_G$ by $\Rightarrow_G^k$. By $\Rightarrow_G^+$ and $\Rightarrow_G^*$, we denote the transitive closure of $\Rightarrow_G$ and the reflexive-transitive closure of $\Rightarrow_G$, respectively. If $S \Rightarrow_G^* x$ for some $x \in \Sigma^*$, x is called a *sentential form*.

If there exists a derivation $S \Rightarrow_G^* w$, where $w \in T^*$, $S \Rightarrow_G^* w$ is said to be a *successful derivation* in G. The *language of* G, denoted by $L(G)$, is defined as

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

In the literature, the unrestricted grammars are also often defined by listing its rules of the form

$$xAy \to xuy,$$

where $u, x, y \in \Sigma^*$, $A \in \Sigma - T$ (see [HU79]). Both definitions are interchangeable, which means that the grammars defined in these two ways generate the same language family—the family of *recursively enumerable languages*, denoted by **RE**.

Definition 2.14. A *context-sensitive grammar* is an unrestricted grammar,

$$G = (\Sigma, T, P, S),$$

such that each rule in P is of the form

$$xAy \to xuy,$$

where $A \in \Sigma - T$, $u \in \Sigma^+$, $x, y \in \Sigma^*$. A *context-sensitive language* is a language generated by a context-sensitive grammar. The family of context-sensitive languages is denoted by **CS**.

Definition 2.15. A *context-free grammar* is an unrestricted grammar,

$$G = (\Sigma, T, P, S),$$

such that each rule $x \to y \in P$ satisfies $x \in \Sigma - T$. Analogically, a *context-free language* is the language generated by a context-free grammar. The family of context-free languages is denoted by **CF**.

Definition 2.16. A *linear grammar* is an unrestricted grammar,

$$G = (\Sigma, T, P, S),$$

such that each rule $x \to y \in P$ satisfies $x \in \Sigma - T$ and $\text{occur}(\Sigma - T, y) \leq 1$. A linear grammar defined in this way generates a *linear language*. The family of linear languages is denoted by **LIN**.

Definition 2.17. A *regular grammar* and a *right-linear grammar* is an unrestricted grammar,

$$G = (\Sigma, T, P, S),$$

such that each rule $x \to y \in P$ satisfies $x \in \Sigma - T$, $y \in T \cup T(\Sigma - T)$ and $x \in \Sigma - T$, $y \in T^* \cup T^*(\Sigma - T)$, respectively. A regular and a right-linear grammar defines *regular* and *right-linear language*, respectively. The family of both of these types of languages is denoted as the family of regular languages by **REG**.

Convention 2.8. To improve readability when we work with several formal model instances, such as $M = (K_1, K_2, \ldots, K_n)$, we change K_i in M to K_{iM}, for every $1 \leq i \leq n$, to identify appropriate owning instance of the component; that is, $M = (K_{1M}, K_{2M}, \ldots, K_{nM})$.

2.4.2 Automata

By analogy with the grammatical approach, there is another approach in the theory of formal languages called the theory of automata. An automaton is a computational mechanism that accepts a sentence of the defined language. As already mentioned, with respect to declarative grammars, the automata represent a procedural approach.

Definition 2.18. A *finite automaton* is a quintuple $M_{FA} = (Q, T, R, s, F)$, where Q is the finite set of *states*, T is the *input alphabet*, $R \subseteq Q \times (T \cup \{\varepsilon\}) \times Q$ is the finite set of *rules* of the form $pa \to q$, where $p, q \in Q$ and $a \in T \cup \{\varepsilon\}$, $s \in Q$ is the *starting state*, and $F \subseteq Q$ is the set of *final states*. If we consider a finite automaton as a rewriting system, $\Sigma = Q \cup T$ is its total alphabet.

A *configuration* of M is a pair of the form (p, w), where $p \in Q$ and $w \in T^*$. Since Q and T are disjunctive, in short we can write the configuration as a string pw. Let u, v be two configurations of M. M makes a rewriting step or *transition* from $u = (p, aw)$ to $v = (q, w)$, where $p, q \in Q$, $a \in T \cup \{\varepsilon\}$, $w \in T^*$, if R contains $pa \to q$. This transition or *accepting step* between u and v is denoted by $u \Rightarrow v$. Like other rewriting systems, we define $u \Rightarrow v$, $u \Rightarrow^n v$ with $n \geq 0$, and $u \Rightarrow^* v$, where $u, v \in QT^*$ (see Definition 2.11). If $sw \Rightarrow^* f$ in M, where $w \in T^*$, $s \in Q$, $f \in F$, M *accepts* w. The set of

all strings accepted by M specifies the language accepted by M, denoted by $L(M)$.

The family of languages accepted by finite automata is denoted by $\mathscr{L}(\mathbf{FA})$.

The expressing power of this formal model is stated in the following theorem (see, for example, [Med00]).

Theorem 2.1. $\mathscr{L}(\mathbf{FA}) = \mathbf{REG}$.

Apart from the inspiration during the study of combined rewriting systems, we use this type of automaton to restrict the content of configurations in Section 5.1.4.

Definition 2.19. A *pushdown automaton* is a septuple $M_{PDA} = (Q, T, \Gamma, R, s, S, F)$, where Q is the finite set of *states*, $T \subseteq \Gamma$ is the *input alphabet*, Γ is the *pushdown alphabet*, $R \subseteq \Gamma^* \times Q \times (T \cup \{\varepsilon\}) \times \Gamma^* \times Q$ is the finite set of *rules*, $s \in Q$ is the *starting state*, $S \in \Gamma$ is the *starting pushdown symbol*, and $F \subseteq Q$ is the set of *final states*. Q, T, Γ are pairwise disjoint. The rules are written in the form $zpa \to yq$ that corresponds to $(z, p, a, y, q) \in R$, where $z \in \Gamma^*$, $p, q \in Q$, $a \in T \cup \{\varepsilon\}$, and $y \in \Gamma^*$.

In the literature, M_{PDA} according to Definition 2.19 is sometimes called an *extended pushdown automaton*. Further, in its classical definition, we do not include a total alphabet, Σ, into the list of its components, because Σ is divided into more than two subsets, such as Q, T, and Γ like in state grammars (see Definition 2.23).

Definition 2.20. A *configuration* of the pushdown automaton is a triple $(\alpha, p, x) \in \Gamma^* \times Q \times T^*$, where α is the content of the pushdown or *pushdown-string* or *pushdown configuration component*, p denotes the current state, and x is a non-processed part of the input string. The *transition* or *accepting step* is a binary relation $\Rightarrow$ on $\Gamma^* \times Q \times T^*$ such that $(z\gamma, p, aw) \Rightarrow (y\gamma, q, w)$ if and only if exists $zpa \to yq$ in R, where $z, y, \gamma \in \Gamma^*$, $p, q \in Q$, $a \in T \cup \{\varepsilon\}$, and $w \in T^*$.

A rule of M_{PDA} is applied to the current configuration, $(z\gamma, p, aw)$, in the following way. If p is the current state, a is the input symbol, z is a substring from the *pushdown top*, and $zpa \to yq \in R$; then, M_{PDA} reads a from the input, changes the pushdown top from z to y, and changes the current state from p to q. Notice that if $a = \varepsilon$ in the rule $zpa \to yq$, then M_{PDA} reads no input symbol. Note that if M_{PDA} uses *reversed pushdown-string*, which means that the pushdown top is on the right end of the pushdown-string, the power of such automaton stays unchanged and the conversion is trivial.

We extend $\Rightarrow$ in the standard way to $\Rightarrow^n$ for $n \geq 0$. Based on $\Rightarrow^n$, we define transitive and reflexive-transitive closure of $\Rightarrow$ as $\Rightarrow^+$ and $\Rightarrow^*$, respectively.

A language accepted by the pushdown automaton M, $L(M)$, is defined as $L(M) = \{w |\ w \in T^*,\ (S, s, w) \Rightarrow (\varepsilon, f, \varepsilon),\ f \in F\}$. Further, the family of languages accepted by the pushdown automata is denoted by $\mathscr{L}(\mathbf{PDA})$.

2.4.3 The Chomsky Hierarchy of Formal Language Families

The Chomsky hierarchy of formal language families (see Theorem 2.2) shows the relation between several language families that are specified in the previous definitions.

Theorem 2.2 (see [Med00])**.** $\mathbf{REG} \subset \mathbf{LIN} \subset \mathbf{CF} \subset \mathbf{CS} \subset \mathbf{RE}$.

To see the relation between basic grammars and defined automata, let us remind that

$$\mathbf{REG} = \mathscr{L}(\mathbf{FA}) \subset \mathbf{CF} = \mathscr{L}(\mathbf{PDA}).$$

2.5 Regulated Rewriting Systems

This section defines some modified rewriting systems. The attention is focused on the regulated grammars, such as programmed and state grammars. Since these formal models require a detailed description, we discuss them more deeply than basic formal models.

2.5.1 The Ways of Regulation

In this book, regulated grammars are based on context-free or right-linear grammars. The regulation dictates some restrictions on the application of rules of classical grammars, such as context-free grammars:

- usage of a relation to describe the previous and subsequent rule application, such as in programmed and matrix grammars;
- include some automata characteristics into grammars, such as a finite set of states that restricts the application of rules to cases when appropriate state is the current state of the configuration, such as in state grammars;
- scattered context checking in the sentential form; that is, unlike context-sensitive grammars we check an entire sentential form, not only immediate neighbor of the rewritten symbol, such as in random-context and scattered context grammars;
- parallel version of the rewriting step in the configuration; thus, we can rewrite all active symbols, or, for instance, all the same active symbols at once, such as in L-systems, pure grammars, and Indian grammars;
- in some cases, even restriction of formal models can be understood as the regulation, which is discussed in Chapter 3.

By using of these restrictions, we often reach a greater power of such rewriting systems.

2.5.2 Regulated Grammars

When we describe some non-context-free languages, the rewriting systems with less generative power than context-sensitive grammars are often preferred. In other words, there exist many regulated grammars less powerful than the context-sensitive grammars that are sufficient to process important non-context-free languages. That was the reason of introducing many new modifications of context-free grammars in the seventies of the twentieth century. The main purpose of these modifications was to increase the power of basic formal model while the context-free form of *rule cores* remains unchanged. We study several types of modified grammars, generally named *regulated grammars*, in this book.

Unless stated otherwise, we consider regulated grammars based on rules of the context-free form (see Definition 2.15); that is, the core of such a rule contains a nonterminal on the left-hand side and an arbitrary string on the right-hand side. The special case of formal model such that the right-hand side of any rule is non-empty is called formal model without *erasing rules*.

Convention 2.9. Apart from the basic language families, such as those of the Chomsky hierarchy, we denote other language families defined by regulated and/or restricted rewriting systems as follows:

$$\mathscr{L}_X(\mathbf{Y}, Z),$$

where X is a way of restricting, $\mathbf{Y}$ denotes a basic formal model, and Z specifies an additional restriction. For example, X can denote a finite index and sometimes can be written on the left side of $\mathscr{L}$, $\mathbf{Y}$ can be **CF** for a context-free grammar, and Z can denote the type of restricting language or mark of appearance checking.

Let Y be a basic formal model, $Y - \varepsilon$ denotes the same formal model but without erasing rules.

Programmed Grammars

The notion of a programmed grammar, introduced in 1969 by [Ros69], has additional conditions put on the successful derivation step. It dictates the set of rules that can be used after an application of a rule. If there is no applicable rule in the set, we use a rule from the special recovery set.

Definition 2.21. A *programmed grammar* (see page 28 in [DP89]) is a quadruple, $G = (\Sigma, T, P, S)$, where all components have the same meaning as in the context-free grammar. Every rule is of the form

$$\lfloor r\colon A \ \rightarrow x, R, F \rfloor,$$

where $A \in \Sigma - T$, $x \in \Sigma^*$, $R, F \subseteq \mathrm{Lab}(P)$, $A \to x$ is a context-free rule core, r is a unique rule label, R is a set of subsequent rules called *success field*, and F is a set of rules called *failure field* of r.

If at least one rule is in the failure field of any rule, we say that the programmed grammar is with an *appearance checking*; otherwise, we can permanently omit empty failure field in the rule notation.

The derivation step during the application of a rule $\lfloor r\colon A \to x, R, F \rfloor$ in G is analogical to the derivation step in a classical context-free grammar (see Definition 2.15). Moreover, it is necessary to check the success and failure fields, R and F.

R denotes the set of rule labels (or rules; see Convention 2.3) from which we can choose the next rule to be applied in G.

When no rule can be applied from R, the sentential form remains unchanged and G continues with the application of a rule from F.

In the standard way, we define $\Rightarrow_G^m$, where $m \geq 0$, $\Rightarrow_G^+$, and $\Rightarrow_G^*$. The language generated by a programmed grammar G, $L(G)$ is defined as $L(G) = \{w \in T^* |\ S \Rightarrow_G^* w\}$.

Example 2.1. Consider the following programmed grammar without appearance checking that generates a context-sensitive language. By this grammar, we demonstrate that this formal model is stronger than a context-free grammar.

Let $G = (\Sigma, T, P, S)$ be a programmed grammar, where the total alphabet contains every useful symbol from the rules, $\Sigma = \{S, A, B, C, a, b, c\}$; according to the convention, terminal symbols are in lower case, $T = \{a, b, c\}$; starting nonterminal, S; the set of rules, P, contains the rules of the prescribed form:

1: $S \to ABC, \{2, 5\}$
2: $A \to aA, \{3\}$
3: $B \to bB, \{4\}$
4: $C \to cC, \{2, 5\}$
5: $A \to a, \{6\}$
6: $B \to b, \{7\}$
7: $C \to c, \{7\}$

Along with each rule, there is a set of the rules that can be applied in the next step. For instance, if the first rule is applied, the next step is performed by the second or fifth rule.

For example, *aabbcc* is generated by the following derivation $S \Rightarrow ABC$ [1] $\Rightarrow aABC$ [2] $\Rightarrow aAbBC$ [3] $\Rightarrow aAbBcC$ [4] $\Rightarrow aabBcC$ [5] $\Rightarrow aabbcC$ [6] $\Rightarrow$ *aabbcc* [7]. When thinking about the relations between rules, we see that rules 2, 3, 4 and 5, 6, 7 are applied one by one. Only after the application of rule 1 or 4, the decision is made about which subsequence of rules (2, 3, 4 or 5, 6, 7) is executed next. The language generated by G, $L(G) = \{a^n b^n c^n |\ n \geq 1\}$.

Example 2.2. Let us give a more complex programmed grammar that contains less rules but uses an appearance checking; that is, some of its rules have non-empty failure fields.

Let $G = (\{S, A, a\}, \{a\}, P, S)$ be a programmed grammar with appearance checking, where the set of the rules contains:

1: $S \to AA, \{1\}, \{2, 3\}$
2: $A \to S, \{2\}, \{1\}$
3: $A \to a, \{3\}, \emptyset$

By the following analysis of the programmed grammar, we demonstrate that G generates the context-sensitive language $L(G) = \left\{a^{2^n} \mid n \geq 1\right\}$. The derivation begins by rewriting the starting nonterminal. Immediately after the application of the first rule, the next application of the first rule again is not possible. Now, we examine the failure field because we reach the end of the derivation only by using the success field. The sentential form remains unchanged, and we select a new rule from the failure field of the first rule. We choose between rules 2 and 3. This decision determines whether we continue to generate more As or we finish the entire computation process by rewriting all nonterminals to terminals. For instance:

$$S \Rightarrow AA\ [1]\ \Rightarrow^2\ SS\ [22]\ \Rightarrow^2\ AAAA\ [11]\ \Rightarrow\ AaAA\ [3] \Rightarrow^3\ aaaa\ [333]$$

A repeated application of the second rule only renames all nonterminals A to S. Considering the definition of programmed grammars, the second rule cannot avoid this task and use its failure field until all As are renamed to S. The last rule simply creates the string of terminals from the generated string of nonterminals. Thus, its failure field is the empty set.

$\mathscr{L}(\mathbf{P})$, $\mathscr{L}(\mathbf{P}, CF{-}\varepsilon)$, $\mathscr{L}(\mathbf{P}, CF, ac)$, and $\mathscr{L}(\mathbf{P}, CF{-}\varepsilon, ac)$ denote the families of the languages generated by programmed grammars, programmed grammars without erasing rules, programmed grammars with appearance checking, and programmed grammars with appearance checking and without erasing rules, respectively.

Random-Context Grammars

As follows from its definition, the context-sensitive grammar checks the immediate neighbor of the rewritten symbol. As its name indicates, a random-context grammar with respect to a context-sensitive grammar does not require the immediateness. The checked context symbols may appear anywhere in the sentential form.

In addition, the opposite condition holds in the case of the forbidding context. It says that the marked symbols may not appear anywhere in the sentential form to apply the corresponding rule.

Furthermore, there are generalized variants that work with the finite sets of strings instead of individual symbols (see [MŠ05]).

Definition 2.22. A *random-context grammar* (see [Wal70] or page 30 in [DP89]) is a quadruple, $G = (\Sigma, T, P, S)$, where all components have the same meaning as in the context-free grammar. Every rule is of the form

$$\lfloor q\colon A \ \rightarrow x, P, F \rfloor,$$

where $A \in \Sigma - T$, $x \in \Sigma^*$, $P, F \subseteq (\Sigma - T)$, $q\colon A \rightarrow x$ is a context-free rule (core), P and F are sets of nonterminals called *permitting set* of q and *forbidding set* of q, respectively. An application of such a rule requires that all nonterminals from P and no from F appear in the rewritten sentential form. Then, G makes a derivation step, denoted by $\Rightarrow$, by analogy with a context-free grammar. If at least one rule has a non-empty forbidding set, we say that the random-context grammar is with an *appearance checking*; otherwise, we can permanently omit empty forbidding sets in the rule notation and call the grammar as random-context.

Example 2.3. Consider the following random-context grammar, $G = (\Sigma, T, P, S)$, that defines language $L(G) = \{a^n b^n c^n \mid n \geq 1\}$. P contains the following rules:

$\lfloor 1\colon S \rightarrow ABC, \emptyset, \emptyset \rfloor$,
$\lfloor 2\colon A \rightarrow aA', \{B\}, \emptyset \rfloor$, $\lfloor 3\colon B \rightarrow bB', \{C\}, \emptyset \rfloor$, $\lfloor 4\colon C \rightarrow cC', \{A'\}, \emptyset \rfloor$,
$\lfloor 5\colon A' \rightarrow A, \{B'\}, \emptyset \rfloor$, $\lfloor 6\colon B' \rightarrow B, \{C'\}, \emptyset \rfloor$, $\lfloor 7\colon C' \rightarrow C, \{A\}, \emptyset \rfloor$,
$\lfloor 8\colon A \rightarrow a, \{B\}, \emptyset \rfloor$, $\lfloor 9\colon B \rightarrow b, \{C\}, \emptyset \rfloor$, $\lfloor 10\colon C \rightarrow c, \emptyset, \emptyset \rfloor$.

$\mathscr{L}(\mathbf{RC})$, $\mathscr{L}(\mathbf{RC}, CF, ac)$, and $\mathscr{L}(\mathbf{RC}, CF-\varepsilon, ac)$ denote the families of the languages generated by random-context grammars, random-context grammars with appearance checking, and random-context grammars with appearance checking and without erasing rules, respectively.

State Grammars

The state grammar belongs to the oldest effort to combine grammars and automata. In fact, it is a context-free grammar enriched by a finite-state control taken from finite automata (see Definition 2.18).

Definition 2.23. A *state grammar* (see [Kas70]) is a quintuple $G = (V, W, T, P, S)$, where V is an alphabet of terminal and nonterminal symbols, T and S has the same meaning as in the context-free grammar (see Definition 2.15), W is a finite set of *states*, and P is a finite subset of a relation $(W \times (V - T)) \times (W \times V^+)$.

Instead of $(q, A, p, v) \in P$, we write the rules in the form $(q, A) \rightarrow (p, v) \in P$. For every string $z \in V^*$, we define a set ${}_G states(z) = \{q \mid (q, B) \rightarrow (p, v) \in P$, where $B \in (V - T) \cap \mathrm{alph}(z)$, $v \in V^+$, $q, p \in W\}$. Let (q, xAy), $(p, xvy) \in W \times V^+$ be two configurations of G. If $r\colon (q, A) \rightarrow (p, v) \in P$, $x, y \in V^*$, ${}_G states(x) = \emptyset$, then G can make a *derivation step* from (q, xAy) to (p, xvy),

symbolically written as $(q, xAy) \Rightarrow (p, xvy)$ $[r]$ in G. If there is a positive integer n such that $\text{occur}(V - T, xA) \leq n$, we say that $(q, xAy) \Rightarrow (p, xvy)$ $[r]$ is an *n-limited derivation* and we write $(q, xAy)\ {}_n\Rightarrow (p, xvy)$ $[r]$.

The *language generated by* G, denoted by $L(G)$, is defined as $L(G) = \{w \in T^* \mid (q, S) \Rightarrow^* (p, w),\ q, p \in W\}$. $\mathscr{L}(\mathbf{ST})$ denotes the family of languages generated by state grammars.

Next, for every $n \geq 1$, we define *n-limited language* generated by G, $L(G, n) = \{w \in T^* \mid (q, S)\ {}_n\Rightarrow^* (p, w),\ q, p \in W\}$. The family of n-limited languages defined by state grammars is denoted by ${}_n\mathscr{L}(\mathbf{ST}) = \{L \mid L = L(G, k),\ 1 \leq k \leq n,\ G \text{ is a state grammar}\}$, where $n \geq 1$. Next, ${}_\infty\mathscr{L}(\mathbf{ST}) = \bigcup_{n\geq 1}^{\infty} {}_n\mathscr{L}(\mathbf{ST})$.

Notice that erasing rules are not allowed in the definition of state grammars. A total alphabet, Σ, is omitted from the list of the components of G. It consists of the disjoint subsets of symbols and states; that is, $\Sigma = V \cup W$, $V \cap W = \emptyset$.

Informally speaking, a state grammar uses one controlling and one restricting mechanism:

1. controlling by finite-state control: Analogically to finite automata, every rule is enriched by an initial and a target state. The *initial state* of the rule dictates in which current state is the rule applicable. The *target state* of the rule determines the state in which the grammar enters by using this rule. That is the reason for extending the configuration of this formal model from a simple sentential form to the pair that contains the current state and the current sentential form (called *sentential configuration component*).
2. restricting by the leftmost possible rewriting: First, the definition of a state grammar (see Definition 2.23) ensures that the leftmost possible rewriting of a nonterminal in a sentential configuration component has to be done in the current state. The rewritten nonterminal may not necessarily be the leftmost but it is sufficient if no other nonterminal on the left from the rewritten one can be rewritten by a rule in the current state. This is necessitated by mathematical condition ${}_G states(x) = \emptyset$ in the current configuration (q, xAy). This condition holds for every rewriting step.

Thus, every configuration contains two active symbols. The first symbol is the current state, whereas the second symbol is the leftmost possibly rewritable nonterminal. The number of passive and potentially active symbols is unlimited.

Example 2.4. First, let us present a classical non-context-free language $\{a^n b^n c^n \mid n \geq 1\}$ generated by a state grammar, $G = (V, W, T, P, S)$, where the components V, W, T can be easily inferred from the set of the rules, P:

1: $(s, S) \rightarrow (s, AC)$

2: $(s, A) \rightarrow (p, aAb)$
3: $(p, C) \rightarrow (s, cC)$
4: $(s, A) \rightarrow (q, ab)$
5: $(q, C) \rightarrow (q, c)$

Next, we illustrate the 2-limited derivation in G, which means that G never works with more than the second nonterminal from the left in the sentential configuration component.

$$\begin{aligned}(s, S) &\Rightarrow (s, AC) && [1]\\ &\Rightarrow (p, aAbC) && [2]\\ &\Rightarrow (s, aAbcC) && [3]\\ &\Rightarrow (q, aabbcC) && [4]\\ &\Rightarrow (q, aabbcc) && [5]\end{aligned}$$

Example 2.5. The next example of a state grammar that generates $\{ww \mid w \in T^*\}$, $T \in \{a, b\}$, that is non-context-free as well. Again, we list only its rules:

1: $(s, S) \rightarrow (s, AB)$
2: $(s, A) \rightarrow (p, aA)$
3: $(p, B) \rightarrow (s, aB)$
4: $(s, A) \rightarrow (q, bA)$
5: $(q, B) \rightarrow (s, bB)$
6: $(s, A) \rightarrow (f, a)$
7: $(f, B) \rightarrow (s, a)$
8: $(s, A) \rightarrow (g, b)$
9: $(g, B) \rightarrow (s, b)$

A computation of G illustrates a generation of string *abbabb*: $(s, S) \Rightarrow (s, AB)$ [1] $\Rightarrow (p, aAB)$ [2] $\Rightarrow (s, aAaB)$ [3] $\Rightarrow (q, abAaB)$ [4] $\Rightarrow (s, abAabB)$ [5] $\Rightarrow (g, abbabB)$ [8] $\Rightarrow (s, abbabb)$ [9]. The principle is similar to the previous example (see Example 2.4). Again, we ensure to make a sequence of rewriting steps by states. Each step is performed in a different place in the sentential configuration component. Moreover, the states serve to remember the information about the symbol that has already been generated in the first substring w. Such information encoded in the state ensures the same substring generation in the second substring w. This mechanism guarantees the equivalence of these two substrings w and w in ww.

For better understanding of some results of this book, we recall that Kasai (see [Kas70]) proved essential theorems concerning state grammars. We give them without proofs.

Theorem 2.3. $\mathscr{L}(\mathbf{ST}) = \mathbf{CS}$.

Corollary 2.4. ${}_{\infty}\mathscr{L}(\mathbf{ST}) \subset \mathscr{L}(\mathbf{ST})$.

Notice that for every $n \geq 1$, ${}_n\mathscr{L}(\mathbf{ST}) \subseteq {}_{n+1}\mathscr{L}(\mathbf{ST})$ as follows from the definition of the state grammar.

Theorem 2.5. *For every* $n \geq 1$, ${}_n\mathscr{L}(\mathbf{ST}) \subset {}_{n+1}\mathscr{L}(\mathbf{ST})$.

m-Parallel n-Right-Linear Simple Matrix Grammars

Since 1970's, the theory of formal languages has studied the nature and properties of parallelism. In 1975, a new rewriting system was introduced. It combines parallelism with regulated rewriting: (1) parallelism like it is known from n-parallel right-linear grammars ([RW73], [RW75]) or from L-systems ([RD71]), and (2) simple matrix grammars ([Iba70]).

This new system rewrites $m \cdot n$ nonterminals in one step so it applies m n-right-linear simple matrix rules, where a matrix is a sequence of the rules applied atomically in the rewriting step.

Definition 2.24. For every $m, n \geq 1$, an *m-parallel n-right-linear simple matrix grammar* (see [Woo75]), m-Pn-G for short, is an $(mn + 3)$-tuple

$$G = (N_{11}, \ldots, N_{1n}, \ldots, N_{m1}, \ldots, N_{mn}, T, S, P),$$

where

N_{ij}, $1 \leq i \leq m$, $1 \leq j \leq n$ are pairwise disjoint *alphabets of nonterminals*,
T is an *alphabet of terminals*,
$S \notin N \cup T$ is a starting symbol (or axiom), where $N = N_{11} \cup \cdots \cup N_{mn}$, and
P is a finite set of *matrix rules*.

A matrix rule can have one of the following three forms:

(i) $[S \to X_{11} \ldots X_{mn}]$, $X_{ij} \in N_{ij}$, $1 \leq i \leq m$, $1 \leq j \leq n$,
(ii) $[X_{i1} \to \alpha_{i1}, \ldots, X_{in} \to \alpha_{in}]$, $X_{ij} \in N_{ij}$, $\alpha_{ij} \in T^*$, $1 \leq j \leq n$, for some i, $1 \leq i \leq m$, and
(iii) $[X_{i1} \to \alpha_{i1}Y_{i1}, \ldots, X_{in} \to \alpha_{in}Y_{in}]$, $X_{ij}, Y_{ij} \in N_{ij}$, $\alpha_{ij} \in T^*$, $1 \leq j \leq n$, for some i, $1 \leq i \leq m$.

A derivation step in m-Pn-G is defined as:
For $x, y \in (N \cup T \cup \{S\})^*$ and m-Pn-G G, $x \Rightarrow y$, if and only if

(A) either $x = S$ and $[S \to y] \in P$,
(B) or $x = y_{11}X_{11} \ldots y_{mn}X_{mn}$, $y = y_{11}x_{11} \ldots y_{mn}x_{mn}$, where $y_{ij} \in T^*$, $X_{ij} \in N_{ij}$, $1 \leq i \leq m$, $1 \leq j \leq n$, and $[X_{i1} \to x_{i1}, \ldots, X_{in} \to x_{in}] \in P$, $1 \leq i \leq m$.

If $x, y \in (N \cup T \cup \{S\})^*$ and $m \geq 0$, then $x \Rightarrow^m y$, if and only if there exists a sequence $x_0 \Rightarrow x_1 \Rightarrow \ldots \Rightarrow x_m$, $x_0 = x$, $x_m = y$. Further, we write $x \Rightarrow^+ y$, if and only if there exists $m > 0$ such that $x \Rightarrow^m y$. Next, we write $x \Rightarrow^* y$, if and only if either $x = y$, or $x \Rightarrow^+ y$.

In the standard way, we define transitive, $\Rightarrow^+$, and reflexive-transitive, $\Rightarrow^*$, closure of the relation of the direct derivation, $\Rightarrow$.

The language generated by m-Pn-G G is denoted by $L(G)$ and defined as $L(G) = \{x \mid S \Rightarrow^* x, x \in T^*\}$. A language $L \subseteq T^*$ is said to be an *m-parallel n-right-linear simple matrix language* (m-Pn-L) if and only if there exists m-Pn-G G such that $L = L(G)$. The family of languages defined by m-Pn-G is denoted by $\mathscr{L}(m\text{-}\mathbf{P}n\text{-}\mathbf{G})$.

By setting $n = 1$, we get an *m-parallel right-linear grammar*. Analogically, by setting $m = 1$, we get an *n-right-linear simple matrix grammar* that is studied in Chapter 5 in more detail.

3

Configuration Restrictions

First, this chapter summarizes different kinds of formal model restrictions. The main aim is placed on the restriction of configurations of formal models.

The regulated grammars and automata represent an important trend in restricting formal models. In 1970's were introduced the first regulated grammars (see [Kas70, Ros69, Wal70, Woo75]). The main idea of these grammars is based on the context-free rules (rule cores) that are supplemented with some additional mechanism that controls the way of applying the rules during every computation. More precisely, we can exchange the notion of a regulation with the notion of restriction because the particular controlling mechanism restricts, in fact, the original behavior of the formal model.

The regulated automata were studied later than the regulated grammars. The exploration of the regulated automata is more intensive on the present (see [KS06a, KM00, Med06, MK02]). Therefore, we pay a special attention to this topic as well.

3.1 Ways of Rewriting Systems Restriction

There is no generally accepted type of a classification of rewriting system restrictions. The basic inspiration of the presented classification comes from [DP89] and [MŠ05].

Consider the following basic classification of *formal model restrictions* with emphasis on rewriting systems:

1. *static* (a restriction of a rewriting system component with a relationship to the descriptional complexity)
2. *dynamic* (a restriction of a computation or a sentence processing with a relationship to the time and space complexity)
   a) configuration restrictions
   b) restrictions of a sequence of applied rules

3. *hybrid* (handles an unclear boundary between the static and dynamic restrictions)

As a conclusion, we can do a transformation between the static and dynamic restrictions because of the strong dependency of the model description and its computation between each other. Since such transformations lead to model descriptions that are difficult to understand, we ignore these transformations in the rest of the book. If it is unclear what type of restriction we talk about, we say that the restriction is hybrid.

For instance, consider the erasing rules as a kind of the static restriction because we restrict the set of rules. However, in terms of the dynamic restrictions, we can talk about the restriction of the length of every configuration that is said to be monotonous during the computation.

To conclude, the classification depends on the ways of the definition of the rewriting system restriction under investigation. In this book, we study only well-known and well-defined restriction types.

Notice that every type of restriction can be transformed to a special type—restricting language that controls the sequence of applied rules, forms of configurations, or other parameters of a rewriting system (see Observation 3.1).

The unification of restricting is complex, so we introduce a more fine-grained classification of configuration restrictions and restrictions of the sequence of rules in the following two sections.

3.2 Ways of Restrictions of Applied Rules Sequences

In this type of restriction, we require additional conditions (usually attached to every single rule) on the applicability of particular rules. Because of a huge number of existing regulated rewriting systems, we cannot state complete classification. The most usual ways of restrictions are:

- a regulation by a regulating language, such as in regulated grammars (see [Med04] and [KM00]);
- a dependency of previously applied rules and the next ones, such as in matrix grammars or programmed grammars;
- finite-state control, such as in state grammars or #-rewriting systems;
- an appearance or forbidding of substring occurrences in rewriting configurations, such as in random-context grammars or in (globally) conditional grammars;
- a limitation of a part of current or arbitrary configuration where the rewriting is permitted or forbidden, such as in case of n-limitation (see Section 3.2.1);
- and so on.

3.2.1 *n*-limitation

n-limitation is a type of the computation restriction that regulates rewriting of configurations. It requires to work only with active symbols that occur in the list of the first n variables in the string that represents the jth component of formal model configuration ξ. We denote component(ξ, j) as x_j, where $j \in \{1, 2, \dots m\}$ and m is the total number of components in ξ.

Therefore, n-limitation restricts the way of choosing of the applied rule such that the rule rewrites only one of the first n leftmost occurrences of arbitrary symbols that are, in general, rewritable in the current configuration. In most cases, such restriction is applied to a potentially infinite configuration component, such as a sentential configuration component or pushdown configuration component.

Definition 3.1. Let $H = (\Sigma, R)$ be a rewriting system. Denote a finite set of variables as $N \subseteq \Sigma$ and component$(\xi, j) \in \Sigma^*$, where j is the constant that expresses the order of the particular configuration component of H on which the n-limitation is applied.

Let ξ_1 and ξ_2 be two configurations, where component$(\xi_1,\ j) = uAz \Rightarrow uvz =$ component(ξ_2, j). The rewriting step between ξ_1 and ξ_2 is called *n-limited*, if occur$(N, u) \leq n - 1$ and A is an active symbol. Consecutively, the sequence of rewriting steps or computation is n-limited if every rewriting step is n-limited. n-limited language contains only sentences that can be processed by n-limited computation.

Let X be a formal model. For every $n \geq 1$, ${}_n\mathscr{L}(\mathbf{X})$ is a family of n-limited languages defined by X.

Observation 3.1. Let us illustrate how easily we can express the n-limitation in notions of the restriction of a sequence of rules and configurations:

a) n-limitation means that we allow the rewriting of a limited number of variable occurrences from the left in a configuration or in its part;
b) n-limitation means that we restrict the number of active symbol occurrences in the configuration by a constant and we add a condition that the occurrence has to be the leftmost in the configuration (or its part) without respect to the passive symbols. A potentially active symbol is not allowed to occur on the left from the last active symbol.

One of the most practical usages of n-limitation in the area of grammars concerning the equivalence between the family of context-free languages and that of context-free languages generated by a context-free grammars that work in the leftmost way (left, canonical). It means that every context-free grammar can work only in the leftmost way, which is 1-limited way, during the sentence derivation without any influence on the generative power.

Moreover, n-limitation can be thought of as the generalization of the leftmost derivation and it can be even applied to automata (see Definition 4.8). In Chapter 6, we study the canonical rewriting of some systems in more detail.

Convention 3.1. Since in some notions, such as the ith symbol from the left in a string, there is no dependency on whether we define such notions from the left or from the right side of a string, we focus only on the left variant if there is no difference between these two approaches. This convention considers notions, such as n-limitation, left derivation, ith symbol in a string, etc.

3.3 Kinds of Configuration Restrictions

In the case of configuration restrictions, the use of a control language is more natural than its use in restrictions of rule sequences.

- The *control language* can be:
  - *finite* (When we apply the restriction to the whole configuration, not only to a part of it, we decrease the power of the system rapidly to the family of finite languages.);
  - *infinite* (To preserve the elegance and effectiveness of the restriction, the restricting language should be as simple as possible, such as right-linear, linear, or most context-free languages.).

In this book, we classify *configuration restrictions* as follows:

- a restriction of the *number of occurrences* of particular symbols; for instance:
  - a set of passive symbols (not very useful variant)
  - a set of variables (specifically, the union of a set of active symbols and a set of potentially active symbols; this is the most important variant that leads to the restriction called finite index; see Definition 3.2);
  - a set of active symbols;
  - a set of potentially active symbols;
  - or a combination of previous sets (see Concept 2.4).
- another classification of the restriction according to the number of symbol occurrences:
  - finite (restricted by a constant);
  - infinite (restricted by a function that mostly depends on the length of the configuration, such as the restriction of workspace in Definition 3.3).

Since the configuration restrictions are dependent on each other and on the set of rewriting rules of the system we can apply two kinds how to ensure a particular configuration restriction (orthogonal to the previous classification):

- *implicit configuration restriction*—the definition of such restricted formal model or its set of rules implies that during a computation[1] there is no configuration that violates the restriction; for instance, #-rewriting systems of index k with a suitable set of rules (see Definition 4.1);
- *explicit configuration restriction*—the rewriting system allows to violate the restriction, so we have to do some checks during every computational step. If there is a violation of the restriction, the computational step cannot be permitted because the target configuration would be damaged; for instance, context-free grammar of index k.

In two following sections, we define two fundamental configuration restrictions: finite index and limited workspace.

3.3.1 Finite Index

A finite index was a restriction mechanism introduced in 1970's. As applied to various formal models, such as context-free grammars and regulated rewriting systems, the finite index was studied in more detail.

Informally speaking, the finite index restricts the number of occurrences of variables in a configuration by a constant k. For instance, in the case of a grammar it means that the sentential configuration component does not contain more than k variables. Analogically, the finite index can be applied to the pushdown configuration component of a pushdown automaton but this approach was not extensively studied.

Definition 3.2. Let $H = (\Sigma, R)$ be a rewriting system, $N \subseteq \Sigma$ be an alphabet of variables, $T \subseteq \Sigma$ be an alphabet of passive symbols, and σ be a starting configuration of H. Let D denote a computation occurring between two configurations, w_1 and w_r, such that $D\colon \sigma = w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_r = w$, $r \geq 1$ in H, where jth component of w, component(w, j), that denotes the sentence of the language defined by H contains only passive symbols. That is, component$(w, j) \in T^*$, where j is a constant that expresses the order of the particular component of w on which the finite index restriction is applied. Let us define $\mathrm{Ind}(D, H) = \max(\{\mathrm{occur}(N, \mathrm{component}(w_i, j)) \mid 1 \leq i \leq r\})$. For component$(w, j) \in T^*$, we define $\mathrm{Ind}(w, H) = \min(\{\mathrm{Ind}(D, H) \mid D$ is a sequence that defines string w in $H\})$. An *index of rewriting system* H is denoted as $\mathrm{Ind}(H) = \sup(\{\mathrm{Ind}(w, H) \mid \mathrm{component}(w, j) \in L(H)\})$. For language L from family $\mathscr{L}(\mathbf{X})$ generated by rewriting system of type $\mathbf{X}$, we define $\mathrm{Ind}_X(L) = \inf(\{\mathrm{Ind}(H) \mid L(H) = L, H$ is of type $X\})$. Let $\mathscr{L}(\mathbf{X})$ be a family of languages, then $\mathscr{L}_k(\mathbf{X}) = \{L \mid L \in \mathscr{L}(\mathbf{X}), \mathrm{Ind}_X(L) \leq k\}$ for any $k \geq 1$, and $\mathscr{L}_{fin}(\mathbf{X}) = \bigcup_{n \geq 1} \mathscr{L}_k(\mathbf{X})$.

[1] We assume only the computation from the starting configuration such that the computation consists of valid applications of rules.

Since the restriction limits only the most economical computation of a sentence, Definition 3.2 is called *weak*. When we extend the restriction to get a *strict* variant of finite index, the restriction is applied to every possible configuration that is reachable by a computation from the starting configuration.

Convention 3.2. When a particular finite index, k, is considered, we omit the word "finite". For instance, rewriting system of index 3.

Two following equations, (3.1) and (3.2) (demonstrated in [DP89]), compare the relationship between regulated grammars without and with finite index. (3.1) and (3.2) emphasize the fact that the finite index rapidly decreases the generative power of a restricted rewriting system.

$$\mathbf{CF} \subset \mathscr{L}(\mathbf{RC}) \subseteq \mathscr{L}(\mathbf{P}) \subset \mathscr{L}(\mathbf{P}, CF - \varepsilon, ac) \subseteq \mathbf{CS} \subset \mathscr{L}(\mathbf{P}, CF, ac) = \mathbf{RE} \tag{3.1}$$

$$\mathscr{L}_{fin}(\mathbf{RC}) = \mathscr{L}_{fin}(\mathbf{P}) = \mathscr{L}_{fin}(\mathbf{P}, CF - \varepsilon, ac) = \mathscr{L}_{fin}(\mathbf{P}, CF, ac) \tag{3.2}$$

The essential theorems and properties of rewriting systems (focused on grammars) with finite index are studied in the third chapter of [DP89] that contains even an alternative definition of a finite index for grammars.

In addition, when we consider the Chomsky hierarchy, all families from (3.2) are incomparable with the family of context-free languages without finite index restriction because it holds that $\{a^n b^n c^n \mid n \geq 1\} \in \mathscr{L}_{fin}(\mathbf{P})$ and $\mathscr{L}(\mathbf{CF}) - \mathscr{L}_{fin}(\mathbf{P}) \neq \emptyset$.

Finally, we briefly compare these two basic types of restrictions—n-limitation and finite index k.

Theorem 3.1. *Let* $\mathbf{X}$ *be a rewriting system that defines the families of languages,* $\mathscr{L}(\mathbf{X})$*,* $\mathscr{L}_k(\mathbf{X})$*, and* ${}_k\mathscr{L}(\mathbf{X})$*. Then,* $\mathscr{L}_k(\mathbf{X}) \subseteq {}_k\mathscr{L}(\mathbf{X})$*.*

Proof of Theorem 3.1. The theorem is implied by the definition of both restrictions (index n and n-limitation) since the finite index is obviously more restrictive than the n-limitation.

In the conclusion, observe that the restrictions can be applied to grammars as well as to automata. First of all, in case of the restriction of pushdown automata, some interesting results appear. We assume similar results as in the area of grammars.

3.3.2 Workspace

The definition of a *workspace* deals with the grammatical notion of a derivation that is replaced by more general notion of a *computation* (a sequence of configurations) of rewriting system in this monograph. The following definition is inspired by page 15 from [DP89].

Definition 3.3. Let us have rewriting system $H = (\Sigma, R)$ of arbitrary type with a total alphabet, Σ, an alphabet of passive symbols, $T \subseteq \Sigma$, and a starting configuration σ. Let ξ be a configuration of H. Assume that workspace restriction is applied to jth component of ξ, denoted by component(ξ, j) (if ξ is m-tuple, then $j \in \{1, 2, \ldots, m\}$).

Further, let $D\colon \sigma = w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n$, $n \geq 1$ be a sequence of configurations in H. We define $WS(w_n, D) = \max(\{|\text{component}(w_i, j)| \mid 0 \leq i \leq n\})$ and $WS(x, H) = \min(\{WS(x, D) \mid D$ is a computation that leads from σ to ξ in H such that component$(\xi, j) = x\})$. $WS(x, H)$ denotes the *workspace* that is needed to process x in H. If there is a constant, $k \geq 0$, such that for every sentence $x \in L(H) - \{\varepsilon\}$ holds that $WS(x, H) \leq k \cdot |x|$, we say that H has *limited workspace*.

In [DP89], you can find two essential theorems related to the restriction of workspace.

Theorem 3.2. *If a grammar, G, of type 0 has a limited workspace, $L(G)$ is a context-sensitive language.*

Theorem 3.3. *Every context-free grammar has a limited workspace implicitly by its definition.*

Proof of Theorem 3.3. It is well-known that context-free grammars are equivalent to context-free grammars without erasing rules, so we can consider only context-free grammars without erasing rules. Let G be a context-free grammar without erasing rules and w be a sentence generated by G. Then, there exists a minimal $k \geq \max(\{|\text{rhs}(p)| \mid p \in P\})$ and every nonterminal generates a substring of the length at least one (after an application of a finite number of rules), so w can be divided at maximum into $|w|$ substrings that are generated by some nonterminal. Since the string cannot be shortened, $WS(w, G) \leq k \cdot |w|$. The rigorous proof can be found in [DP89].

3.4 Complexity of Formal Models

The complexity deals with a quantification and classification of space (memory) and time, demands caused by processing languages and their sentences,

and by storing of finite descriptions of formal models that define languages. A survey of the topic can be found in [Pap94] and [Bro89].

From the point of view of dynamic and static restrictions of rewriting systems, we can classify the complexity in the theory of formal languages into two branches:

1. space complexity and time complexity (dynamic character; more practical approach)
2. descriptional complexity (static character; theoretically oriented approach)

Concept 3.1. A *space complexity* and *time complexity* study how effectively an instance of a formal model is able to process a sentence of a language or a whole language, respectively. A numerical measurement is usually based on a unified general formal model, such as Turing machine (see [Med00]). The time complexity states the particular number of computational steps for a sentence processing. Mostly, it is dependent on the length of the sentence. On the other hand, the maximal number of needed memory cells (for instance, in case of Turing machine) is the key information for the space complexity.

For entire language families or formal models, we talk about the complexity classes that are mostly expressed asymptotically[2].

Besides the practical notions of the time and space complexity, which are often applied to computationally complete models, such as Turing machines, RAM[3] models, or modern programming languages, take a look on the existing notion of the descriptional complexity.

Concept 3.2. A *descriptional complexity* (or syntactic complexity) combines different metrics that characterize an efficiency of the storing of the finite description of a formal model. The most common metric is a cardinality of an individual component of a rewriting system. Another metrics can measure an important component subset that contains items with some special property; for instance, the number of rules with context-sensitive condition when the basic rules or rule cores are context-free.

In the theory of formal languages, the descriptional complexity is a very popular topic (see [BB05, MM07, Med96, Med97a, Med97b, Med98]; [DP89] and [MŠ05] contain the whole collection of related results). The descriptional complexity measures an efficiency of the description by a formal model or rewriting system, respectively. In most cases, the authors focus on the reduction of the descriptional complexity of only one component (one dimension). More exceptional and valuable approach is to try to reduce more components simultaneously, such as in [MF03].

[2] Asymptotically means that the complexity is limited from above, from below, or even from both sides, by some mathematical function that depends on the length of a sentence.

[3] Random Access Memory

3.4.1 Classification of Formal Model Complexity

Now, we examine an alternative classification of complexity of formal models. We introduce a dynamic (run-time) variant of the descriptional complexity called *dynamic complexity* that is closely related to the practical space complexity, which is more general. Measures, such as finite index and n-limitation, can be studied from the point of view of the dynamic complexity that has more concrete relationship to rewriting systems and to the theory of formal languages than the universal time and space complexity.

Two approaches of classification that are important for this book follow:

1. *practical* – uses metrics, such as the number of processor instructions or the number of memory cells needed for a successful computation of an algorithm, dependent on the input or the length of the input, respectively. This approach is dependent on the chosen formal model and on its implementation in the first place (used hardware platform, hardware/software partitioning, etc.).
2. *theoretical* – is more robust and does not require the selection of an implementation technique. We only have to choose a formal model and its particular instances. For example, an instance of a context-free grammar represents an instance of a formal model like this. We observe the properties related to the recording of an instance of the formal model itself and its processing parameters, such as determinism, the maximum length of a configuration, the number of nonterminals, the number of rules, etc. The disadvantage of such approach is the problematic comparison of the complexity of incompatible formal models and non-existence of some metrics in some models.

Another, orthogonal, view of the complexity divides the approaches according to the focus either on the efficiency of the description of a formal model instance or the efficiency of the processing of a language sentence.

1. *descriptional complexity* – static character; for instance, the length of a component in the formal model description and its complexity, such as the number of rules, number of nonterminals, length of the longest rule, size of LR table, or the complexity of the application of a rule that is the most complex one in the model;
2. *dynamic complexity* – dynamic (run-time) character that observes the properties during the processing of a particular formal model instance; for instance, the finite index (see Definition 3.2) during the processing of a particular sentence, minimal/maximal length of a workspace (see Definition 3.3), complexity of the selection of a subsequent rule to be applied, or the growth speed of the state space during the simulation of a non-deterministic behavior of a formal model by a backtracking algorithm.

In this book, we study only the theoretical aspects of the complexity. Nevertheless, the relationship between the practical and theoretical complexity is often obvious.

The main part of this monograph focuses on the systematization of approaches to the formal model configuration restrictions according to the presented criteria and their combinations. Although the majority of the mentioned metrics of the dynamic complexity has already been studied to some extend, we try to put them in a broader context.

Part II

New Formal Models and Their Restrictions

4

Definitions

New formal models discussed in this monograph have several common features. In two cases, we deal with the rewriting systems that cannot be strictly classified as grammars or automata—(1) #-rewriting systems (see Section 4.1) and (2) reducing deep pushdown automata (see Section 4.2 and Section 4.2.2, respectively). The third model, restricted pushdown automata, inspired by the regulated grammars, applies their regulating mechanism to the pushdown automata.

The results concerning these new systems are reported in Chapter 5 with their rigorous proofs. In most cases, we discuss the power of these formal models (see Chapter 5 and partly Chapter 6). From the mathematical point of view, the most interesting corollaries are the resulting infinite hierarchies that depend on the restriction of these rewriting systems, such as index k and n-limitation.

4.1 #-Rewriting Systems

The following section discusses the key formal model of this book, including some its modifications. The results concerning this rewriting system produce the essential part of the text.

We introduce and study #-rewriting systems that represent generative regulated rewriting systems. #-rewriting systems combine properties of automata and grammars. From grammars, they use the method of processing of the language sentence—the generation of a sentence. The #-rewriting systems are inspired by finite-state control used in automata. Next, they omit nonterminals from their definition.

The fundamental part of the study of these new systems concerns the restriction of finite index. Further, few natural modifications, such as n-right-linear #-rewriting systems are explored as well (see Section 4.1.3 and Section 4.1.4).

The main concept of #-rewriting systems were introduced in [Kři05b] and [Kři05c]. The fundamental results are established in [KS06b] and [KMS06a].

4.1.1 Motivation

In the formal language theory, the overwhelming majority of language-defining formal models is based on rewriting systems that represent either grammars or automata.

Recall that grammars generate the languages and automata accepts them, which is the basic difference between these two language-defining approaches. Consider a context-free grammar, G (see Definition 2.15). G contains an alphabet of terminals and an alphabet of nonterminals, from which one is selected as the starting nonterminal. From the starting nonterminal, a consecutive rewriting of each nonterminal generates a sentential form (configuration) that is a string over the total alphabet. The sequence of rewriting begins by the starting nonterminal (axiom) and it finishes in a sentential form containing only terminals. We say that G generates a sentence of a language. The set of generated strings over the terminal alphabet determines the language generated by G.

To illustrate automata, consider a finite-state automaton M (see Definition 2.18). M has a finite set of states, one of which is defined as the starting state. In addition, some states are specified as final states. M works by making moves. During a move, it changes its current state (finite inner memory of the model) and reads an input symbol. If with an input string, M makes a sequence of moves according to its rules so it starts from the starting state, reads the input string, and reaches a final state, then M accepts the input string. The set of all strings accepted in this way represents the language that M defines.

Although it is obviously quite natural to design language-defining formal models based on a combination of grammars and automata and, thereby, make their scale much broader, only a tiny minority of these devices is designed in this combined way (see [BF95], [Kas70], and [MHHO05]). To support this combined design, the present chapter introduces new rewriting systems, called #-rewriting systems (first of all, see [KMS06a]), having features of both grammars and automata. Indeed, like grammars, they are generative formal models. However, like automata, they use finitely many states without any nonterminals.

The inspiration of introducing #-rewriting systems comes from these areas of the formal language theory:

- string splitting operations known from biology (see [PL90] and [Sol]) that can be simulated by a #-rewriting system; from the splitting point of view, # called *bounder* divides a string into two parts. Thus, bounders split a sentential configuration component into finitely many parts consisting only passive symbols;

- pure grammars that do not use nonterminal symbols. As the difference from the pure grammars, #-rewriting systems introduce only variable symbol, called bounder, thus not each symbol in the configuration can be rewritten. From the descriptional complexity point of view, #-rewriting systems restrict the cardinality of the specific subset of the total alphabet to one. More specifically, there is a symbol in the total alphabet that can occur in no sentence of the defined language;
- finite automata for its finite-state control;
- context-free grammars with their simple form of rules. By the context-free form of rule cores in #-rewriting systems, every bounder is rewritten by a string over the total alphabet, including the empty string.
- n-limitation applied to each rule separately so only a few leftmost bounders are rewritable. In case of #-rewriting systems, the previous condition is integrated directly into the definition of the form of rules. Every rule determines the order of the bounder in a sentential configuration component to rewrite.

Before we define #-rewriting systems, notice that classical #-rewriting systems contain rule cores of the context-free form. The rule cores are extended by a finite-state control. By the form of the rules, the rewriting mechanism is restricted by n-limitation that allow to rewrite only several leftmost bounders in the sentential configuration component. It is not a classical n-limitation applied to a whole configuration component as in Definition 3.1. In fact, this n-limitation implicitly restricts each rule of the system as determines the number specifying the order of the active bounder.

Except when explicitly stated otherwise, #-rewriting systems implicitly denote context-free #-rewriting systems.

4.1.2 Definition

Definition 4.1. A *context-free #-rewriting system* is a quadruple $H = (Q, \Sigma, s, R)$, where Q is a finite set of states, Σ is an alphabet containing $\#$ called a *bounder*, $Q \cap \Sigma = \emptyset$, $s \in Q$ is a starting state and $R \subseteq Q \times \mathbb{N} \times \{\#\} \times Q \times \Sigma^*$ is a finite relation whose members are called *rules*. A rule $(p, n, \#, q, x) \in R$, where $n \in \mathbb{N}$, $q, p \in Q$ and $x \in \Sigma^*$, is usually written as $r\colon p\ {}_n\# \to q\ x$ hereafter, where r is its unique label.

A *configuration* of H is a pair from $Q \times \Sigma^*$. Let χ denote the set of all configurations of H. Let $pu\#v, quxv \in \chi$ be two configurations, $p, q \in Q$, $u, v \in \Sigma^*$, $n \in \mathbb{N}$ and $\mathrm{occur}(u, \#) = n - 1$. Then, H makes a *derivation step* or a *computational step* from $pu\#v$ to $quxv$ by using $r\colon p\ {}_n\# \to q\ x$, symbolically written $pu\#v \Rightarrow quxv\ [r]$ in H or simply $pu\#v \Rightarrow quxv$.

In the standard manner, we extend $\Rightarrow$ to $\Rightarrow^m$, for $m \geq 0$; then, based on $\Rightarrow^m$, we define $\Rightarrow^+$ and $\Rightarrow^*$ in the standard way.

The *language generated* by H, $L(H)$, is defined as

$$L(H) = \{w \mid s\# \Rightarrow^* qw,\ q \in Q, w \in (\Sigma - \{\#\})^*\}.$$

Note 4.1. Observe that the form of the rule in CF#RS, $r\colon p\,{}_n\# \to q\ x$, contains unnecessary # on the left-hand side. The reason for this is the unification of the notation of rules in various versions of #-rewriting systems. For instance, in generalized #-rewriting systems, the left-hand side contains a string, not only a symbol.

Definition 4.2. The definition of #-rewriting system of finite index is based on Definition 3.2 for the general rewriting system, where $N = \{\#\}$, $T = \Sigma - \{\#\}$ and the type of rewriting system $X =$ CF#RS. To be precise, $\Sigma \cup Q$ is the total alphabet.

More specifically, let k be a positive integer. A #-rewriting system, H, is of *index* k if for every configuration $x \in \chi$, $s\# \Rightarrow^* qy = x$ implies $\mathrm{occur}(\#, y) \leq k$. Notice that H of index k cannot derive a string containing more than k #s; in this sense, this notion differs from the corresponding notion in terms of programmed grammars, which can without the finite index restriction derive strings containing more than k nonterminals.

Let $\mathscr{L}_k(\mathbf{CF\#RS})$ and $\mathscr{L}_k(\mathbf{P})$ denote the language families defined by context-free #-rewriting systems of index k and by programmed grammars of index k (according to Definition 3.2), respectively.

Definition 4.3. A rule of #-rewriting system is said to be *erasing rule* if its right-hand side contains only target state and the empty string. For instance, $p\,{}_2\# \to q\ \varepsilon$.

Example 4.1. Consider a context-free #-rewriting system, $H_1 = (\{s, p, q, f\}, \{a, b, c, \#\}, s, R_1)$, where R_1 contains:

1: $s\,{}_1\# \to p\ \#\#$
2: $p\,{}_1\# \to q\ a\#b$
3: $q\,{}_2\# \to p\ \#c$
4: $p\,{}_1\# \to f\ ab$
5: $f\,{}_1\# \to f\ c$

$L(H_1) = \{a^n b^n c^n \mid n \geq 1\}$, where $Ind(H_1) = 2$; that is, H_1 is of index 2. For instance, H_1 generates $aaabbbccc$ as $s\# \Rightarrow p\#\#$ [1] $\Rightarrow qa\#b\#$ [2] $\Rightarrow$ $pa\#b\#c$ [3] $\Rightarrow$ $qaa\#bb\#c$ [2] $\Rightarrow$ $paa\#bb\#cc$ [3] $\Rightarrow$ $faaabbb\#cc$ [4] $\Rightarrow$ $faaabbbccc$ [5].

Example 4.2. Let $G = (\{S\}, \{a, b, a', b'\}, S, P)$ be a context-free grammar, where P contains:

1. $S \to SS$
2. $S \to \varepsilon$
3. $S \to aSa'$

4. $S \to bSb'$

$L(G) = D_2$ is Dyck language with two types of brackets. It holds that $L(G) \in \mathbf{CF} - \mathscr{L}_{fin}(\mathbf{CF}\#\mathbf{RS})$ (see [DP89]).

Notice that the language family generated by #-rewriting systems of finite index is incomparable with the family of context-free languages. More specifically, a context-free Dyck language is not generated by any #-rewriting systems of finite index and vice versa. Example 4.1 shows that the family of languages generated by #-rewriting systems of finite index contains at least one non-context-free language.

The most important results concerning #-rewriting systems (see Theorem 5.3 and Theorem 5.19) consider the restriction of finite index. To give an insight into these systems, #-rewriting systems of finite index create the infinite language family hierarchy, which is very crucial mathematical property. The closer view of #-rewriting systems with respect to the dynamic complexity is studied as well.

4.1.3 Based on Right-Linear Rules

Analogically to the Chomsky hierarchy, one of the most natural modifications of every rewriting system is the change of the form of the rule cores.

As a special case of #-rewriting systems, we introduce and study n-right-linear #-rewriting systems as the central topic of this section. As their name indicates, these systems are underlain by rules that are similar to the right-linear grammatical rules (see Definition 2.17).

The right-linear rules never increase the number of variables—in our case, bounders—in a configuration. That is, for every rule r, $\mathrm{occur}(\#, \mathrm{rhs}(r)) \leq 1$. Therefore, the only split of the string by bounder has to be done by the definition of the starting configuration of the form $s\#^n$, where s is the starting state of the system and n is a positive integer. During the computation, the number of bounders may only decrease, which leads to the resulting language sentence, by the application of the rules of the form $p\,\# \to q\ \alpha$, where $\alpha \in (\Sigma - \{\#\})^*$.

These systems characterize the infinite hierarchy of language families defined by m-parallel n-right-linear simple matrix grammars (see Section 5.1.1); however, under some trivial restrictions, they generate only the family of right-linear languages (see Theorem 5.8).

Definition 4.4. Let $H = (Q, \Sigma, s, R)$ be a context-free #-rewriting system, $n \in \mathbb{N}$, and, in addition, R satisfies

$$R \subseteq Q \times \mathbb{N} \times \{\#\} \times Q \times ((\Sigma - \{\#\})^*\{\#\} \cup (\Sigma - \{\#\})^*),$$

then H is said to be an *n-right-linear #-rewriting system*, n-RLIN#RS for short.

A rule $(p, i, \#, q, x) \in R$, where $i \in \mathbb{N}$, $i \leq n$, $p, q \in Q$, and $x \in \{\alpha\#\} \cup \alpha$, $\alpha \in (\Sigma - \{\#\})^*$. The rule is usually written as $r\colon p\ {}_i\# \to q\ x$ hereafter, where r is its unique label, which can be omitted.

The notions of a configuration, computational step, m-step computation ($m \geq 0$), non-trivial computation, and computation are defined by analogy with a context-free #-rewriting system (see Definition 4.1). The only exception is the definition of an initial configuration that is defined as $\sigma = s\#^n$.

The *language generated* by the n-RLIN#RS H, $L(H)$, is defined as

$$L(H) = \{w \mid s\#^n \Rightarrow^* qw,\ q \in Q, w \in (\Sigma - \{\#\})^*\}.$$

Let n be a positive integer and σ be an starting configuration of an n-right-linear #-rewriting system, H. H is of *index* n if for every configuration $x = qy$, $\sigma \Rightarrow^* qy$ implies $\mathrm{occur}(\#, y) \leq n$. Notice that H of index n cannot derive a string containing more than k #s. Furthermore, notice that a n-RLIN#RS H is always of index n.

Let $n \in \mathbb{N}$. $\mathscr{L}(n\text{-}\mathbf{RLIN\#RS})$ denotes the family of languages generated by n-right-linear #-rewriting systems.

Definition 4.5. A computational step is *#-erasing* if # is rewritten with a string of passive symbols or the empty string during this step.

Let d be a k-step computation in H, for some $k \geq 0$. For every $1 \leq i \leq k$, by d_i and ${}_td_i$, we denote the ith computational step in d and the ith computational step rewriting the tth #, respectively. t is called the *degree of step* d_i. The computation d is *successful* if d describes a computation from the starting configuration to a final configuration (q, w) with $w \in (\Sigma - \{\#\})^*$.

Example 4.3. 3-RLIN#RS $H_2 = (\{s, p, q, r, t\}, \{a, b, c, \#\}, s, R_2)$, where R_2 contains

1: $s\ {}_1\# \to p\ a\#$
2: $p\ {}_2\# \to q\ b\#$
3: $q\ {}_3\# \to s\ c\#$
4: $s\ {}_1\# \to r\ a$
5: $r\ {}_1\# \to t\ b$
6: $t\ {}_1\# \to t\ c$

For instance, H_2 computes *aabbcc* by 6-step computation d: $s### \Rightarrow pa### [1] \Rightarrow qa\#b\#\# [2] \Rightarrow sa\#b\#c\# [3] \Rightarrow raab\#c\# [4] \Rightarrow taabbc\# [5] \Rightarrow taabbcc$ [6], where $d = {}_1d_1\ {}_2d_2\ {}_3d_3\ {}_1d_4\ {}_1d_5\ {}_1d_6$.

Obviously, H_1 from Example 4.1 is of index 2. On the other hand, H_2 from the previous example is of index 3. Both systems define the same language, $L(H_1) = L(H_2) = \{a^nb^nc^n \mid n \geq 1\}$.

The main results—Theorem 5.7 and Theorem 5.8—concerning n-right-linear #-rewriting systems are proved in Section 5.1.1.

4.1.4 Based on Generalized Rules

The previous two sections introduce #-rewriting systems based on rule cores of the context-free and right-linear form. During a computational step, these systems rewrite only one occurrence of a bounder. These systems can be naturally extended so they rewrite a substring that contains at least one bounder. Therefore, in Chapter 5, we will discuss the change of the power of such systems.

This section discusses a generalized version of #-rewriting systems with rule cores of the context-sensitive form. It demonstrates that this context-based generalization does not affect the generative power of #-rewriting systems of finite index. A new characterization of the infinite hierarchy of language families generated by programmed grammars of finite index is obtained.

The original version of #-rewriting system is based upon rules of the form $p\,{}_i\# \rightarrow q\ \gamma$, where p, q are states, i is a positive integer, and γ is a string over the alphabet Σ. By using this rule, the system rewrites ith # with γ and, simultaneously, changes the current state p to q. In the present section, we discuss a generalized version of #-rewriting system that uses rules of the form $p\,{}_i\alpha\#\beta \rightarrow q\ \alpha\gamma\beta$, where α and β are strings and the other symbols have the same meaning as above. This generalized rule is applicable to # if this ith # occurs in the α-β context; otherwise, the application is analogical to the original version (see Definition 4.1).

Definition 4.6. A *generalized #-rewriting system*, abbreviated as G#RS, is a quadruple $H = (Q, \Sigma, s, R)$, where the meanings of Q, Σ, and s remain the same as in the definition of the context-free #-rewriting system and $R \subseteq Q\times\mathbb{N}\times\Sigma^*\{\#\}\Sigma^*\times Q\times\Sigma^*$. A rule is usually written as $r\colon p\,{}_i\alpha\#\beta \rightarrow q\ \alpha\gamma\beta \in R$ hereafter, where r is its unique label, $i \in \mathbb{N}$, $q, p \in Q$, and $\alpha, \beta, \gamma \in \Sigma^*$. α and β are the *left* and *right context* of r, respectively.

A *configuration* of H is a pair from $Q \times \Sigma^*$. Let $pu\alpha\#\beta v$, $qu\alpha\gamma\beta v$ be two configurations, $p, q \in Q$, $u, v, \alpha, \beta, \gamma \in \Sigma^*$, $i \in \mathbb{N}$, and $\mathrm{occur}(\#, u\alpha) = i - 1$. Then, H makes a *computational step* from $pu\alpha\#\beta v$ to $qu\alpha\gamma\beta v$ by using $r\colon p\,{}_i\alpha\#\beta \rightarrow q\ \alpha\gamma\beta$, symbolically written $pu\alpha\#\beta v\,{}_i\Rightarrow qu\alpha\gamma\beta v\ [r]$ in H or $pu\alpha\#\beta v \Rightarrow qu\alpha\gamma\beta v\ [r]$ in H when position is not relevant or simply $pu\alpha\#\beta v \Rightarrow qu\alpha\gamma\beta v$ when the applied rule is immaterial.

By analogy with context-free #-rewriting systems (see Definition 4.1), we extend a computational step to an m-step computation ($m \geq 0$), non-trivial computation, and computation, respectively.

The *language generated* by G#RS H, $L(H)$, is defined in the same way as for CF#RS.

As special case of G#RS, if every $r\colon p\,{}_i\alpha\#\beta \rightarrow q\ \alpha\gamma\beta \in R$ satisfies that $\alpha = \beta = \varepsilon$, then H is called a *context-free #-rewriting system* (CF#RS).

Let k be a positive integer. Analogically to Definition 3.2, H is of *index* k if for every configuration $x = qy$ in H, $s\# \Rightarrow^* qy$ implies $\text{occur}(\#, y) \leq k$.

Definition 4.7. For G#RS H, $\max_L(H)$ and $\max_R(H)$ denote the maximum length of left-hand and right-hand side of rules, respectively. Precisely, let $H = (Q, \Sigma, s, R)$ be a G#RS, $\max_L(H) = \max(\{|\alpha| \mid p\,{}_i\alpha \to q\ \beta \in R\})$ and $\max_R(H) = \max(\{|\beta| \mid p\,{}_i\alpha \to q\ \beta \in R\})$.

Let k be a positive integer. $\mathscr{L}_k(\mathbf{G\#RS})$ and $\mathscr{L}(\mathbf{G\#RS})$ denote the families of languages generated by generalized #-rewriting systems of index k and generalized #-rewriting systems without the finite index restriction, respectively.

Example 4.4. G#RS $H_3 = (\{s, p, q\}, \{a, b, c, \#\}, s, R_3)$, where R_3 contains

1: $s\,{}_1\# \to s\ a\#\#$
2: $s\,{}_2a\#\# \to p\ a\#b\#c$
3: $p\,{}_1a\# \to q\ aa\#$
4: $q\,{}_2b\#c \to p\ bb\#cc$
5: $p\,{}_1a\# \to p\ a$
6: $p\,{}_1b\#c \to p\ bc$

For instance, H_3 computes $aabbcc$ as $s\# \Rightarrow sa\#\#$ [1] $\Rightarrow pa\#b\#c$ [2] $\Rightarrow qaa\#b\#c$ [3] $\Rightarrow paa\#bb\#cc$ [4] $\Rightarrow paabb\#cc$ [5] $\Rightarrow paabbcc$ [6].

Let us investigate the application of the second rule in more detail. In case of $s\,{}_2a\#\# \to p\ a\#b\#c$, there are two different ways of choosing the left and right context of the rewritten bounder. The left-hand side of the rule contains two bounders. The selection of the bounder to rewrite depends on the starting configuration since the rule dictates rewriting of the second bounder in the sentential configuration component from the left. In this example, the left context α equals to $a\#$ and the right context β equals to ε.

If we consider a different computational step in another generalized #-rewriting system, the contexts can be different. For instance, $sa\#a\#\# \Rightarrow pa\#a\#b\#c\#$ [$s\,{}_2a\#\# \to p\ a\#b\#c$], where $\alpha = a$ and $\beta = \#$.

Obviously, H_1 from Example 4.1 and H_3 from this example are of index 2. Both systems describe the same language $L(H_1) = L(H_3) = \{a^n b^n c^n \mid n \geq 1\}$ which belongs into $\mathscr{L}(\mathbf{CF\#RS}) - \mathbf{CF}$.

As its main result, Section 5.1.3 demonstrates that the generalization under discussion does not affect the generative power of #-rewriting systems of finite index, so we obtain an alternative characterization of the infinite hierarchy of language families generated by programmed grammars of finite index (see [KMS06a], and Theorems 3.1.2i and 3.1.7 in [DP89]).

This result is of some interest when compared, for instance, to a similar generalization in terms of the classical Chomsky hierarchy (see Theorem 2.2), in which grammars with the generalized rules (context-sensitive grammars) are much stronger than ordinary context-free grammars.

4.1.5 Other Variants of #-Rewriting Systems

In this monograph, we also studied other variants of #-rewriting systems. Furthermore, there are some hypotheses and preliminary results that are mostly discussed in Chapter 6 and Chapter 7.

Let $H = (Q, \Sigma, s, R)$ be a context-free #-rewriting system. Some of its variants are defined as follows:

1. H is a *deterministic* #-rewriting system if for every rule $p \in Q$ and for every positive integer i holds that $p\,{}_i\#$ is the left-hand side of at most one rule in H. The determinism of #-rewriting systems can be reasonably defined even in several other ways that have different properties. For more details, see Section 6.2.1.
2. Let $\alpha, \beta \in \chi$ be two configurations of H. If $\alpha \Rightarrow \beta$ in H, then H makes a *direct reduction* from β to α, symbolically, $\beta \vdash \alpha$. H is said to be a *reducing #-rewriting system*.
3. If H simultaneously rewrites all bounders in the current sentential configuration component in a computational step, H works *parallelly*.

Straightforwardly, we can modify the introduced variants to be based on n-right-linear or general #-rewriting systems. Next, we discuss the above variants in more detail.

Reducing Variant

Instead of the previous top-down generating approach, we find inspiration in syntactical parsers that can work in a bottom-up way. Thus, we change this generative approach to a reducing approach.

Let $H = (Q, \Sigma, s, R)$ be a #-rewriting system. H is a *reducing #-rewriting system* if it reduces a given language by a sequence of reduction steps instead of generating by computational steps. H makes a *reducing step* from $quxv$ to $pu\#v$ according to rule $r\colon p\,{}_n\# \to q\,x$, symbolically written as $quxv \vdash pu\#v\ [r]$ in H. Let $\vdash^*$ denote the reflexive-transitive closure of $\vdash$.

The language defined by H, ${}_rL(H)$, is defined as

$$ {}_rL(H) = \{w \mid qw \vdash^* s\#, q \in Q, w \in (\Sigma - \{\#\})^*\}. $$

Consider $H_1 = (\{s, p, q, f\}, \{a, b, c, \#\}, s, R_1)$ from Example 4.1. Then, the reducing variant of H_1 makes the reduction of string $aaabbbccc$ as follows: $faaabbbccc \vdash faaabbb\#cc\ [5] \vdash paa\#bb\#cc[4] \vdash qaa\#bb\#c\ [3] \vdash pa\#b\#c\ [2] \vdash qa\#b\#\ [3] \vdash p\#\#\ [2] \vdash s\#\ [1]$.

Parallel Variant

A *parallel #-rewriting system* is a quintuple $= (Q, \Sigma, s, P, R)$, where Q, Σ, and s are defined the same way as in CF#RS, $P \subseteq \mathbb{N} \times \Sigma^*$ is a finite relation that

contains items called *parallel rule cores* that are of the form ${}_n\#\,{}_s\!\rightarrow x$, $n \in \mathbb{N}$, $x \in \Sigma^*$; that is, the left-hand and the right-hand side of the rule are separated by the right arrow with a lower left index "s". $R \subseteq Q \times 2^P \times Q$ is a finite relation with the condition that for every rule $(p, F, q) \in R$, $p, q \in Q$, $F \in 2^P$ and for every two parallel rule cores $c, d \in F$, $c\colon\ {}_{i_c}\#\,{}_s\!\rightarrow x_c$, $d\colon\ {}_{i_d}\#\,{}_s\!\rightarrow x_d$ holds $i_c \neq i_d$.

A rule $t = (p_t, \{r_1, \ldots, r_m\}, q_t) \in R$, $m \geq 1$ is applicable in a configuration px, $p \in Q$, $x \in \Sigma^*$ if and only if $p = p_t$, $\mathrm{occur}(\#, x) \geq i_j$, for every $1 \leq j \leq m$, where $r_j\colon\ {}_{i_j}\#\,{}_s\!\rightarrow y_j$.

By a rule $t = (p, \{r_1, \ldots, r_m\}, q)$, if t is applicable to pu, all parallel rule cores $r_1, \ldots, r_m$ are applicable to u and the current state p is changed to a new state q, H makes a parallel computational step from pu to qv, symbolically written as $pu\ {}_p\!\Rightarrow qv\ [t]$ in H.

By analogy with CF#RS of index k, consider a parallel #-rewriting system of index k. The condition of applicability of a rule has to be extended by a subcondition that $\mathrm{occur}(\#, x) - m + \sum_{l=1}^{m} \mathrm{occur}(\#, y_l) \leq k$.

Let ${}_p\!\Rightarrow^*$ denote the reflexive-transitive closure of ${}_p\!\Rightarrow$. The *language generated* by H, ${}_pL(H)$, is defined as

$$ {}_pL(H) = \{w \mid s\#\,{}_p\!\Rightarrow^* qw,\ q \in Q, w \in (\Sigma - \{\#\})^*\}. $$

The parallel and the reducing #-rewriting systems are to be studied in the future in more detail (see Chapter 7).

For an overview of the introduced versions of #-rewriting systems, see Table 4.1.

Type of System	**Abbr.**	**Rule Form**
n-right-linear	n-RL#RS	$Q \times \mathbb{N} \times \{\#\} \times Q \times (\Sigma - \{\#\})^*\{\#, \varepsilon\}$
Context-free (Basic)	CF#RS	$Q \times \mathbb{N} \times \{\#\} \times Q \times \Sigma^*$
Context-sensitive (General)	G#RS	$Q \times \mathbb{N} \times \Sigma^*\{\#\}\Sigma^* \times Q \times \Sigma^*$

Table 4.1. #-Rewriting Systems – Overview

4.2 Deep Pushdown Automata

The study of the dual models to the regulated grammars, *regulated automata*, has started recently (see [Kři04b, KM00, MK02]). For example, Meduna in [Med06] introduced the generalization of classical pushdown automata.

In the following new rewriting system, we see the principle of the combination of automata and grammars. A classical pushdown automaton always works only with the top of its pushdown (see Definition 2.19). On the other

hand, a grammar rewrites any symbol in the sentential form. Now, we are inspired by grammars and we study the modified pushdown automata that allow us to work with the symbols deeper in the pushdown content. Since in practice the access[1] to the top of a pushdown structure is optimized, we restrict the manipulation with symbols on the pushdown such that we can reach the symbol only in some maximum depth. Informally speaking, we can rewrite only first few active symbols from the top toward the bottom and the maximum depth of an active symbol is given by a constant. The rest is like in ordinary pushdown automata.

Definition 4.8. A *deep pushdown automaton*, DTDP for short, is a 7-tuple, $M = (Q, T, \Gamma, R, s, S, F)$, where Q is the finite set of *states*, T is the input alphabet, Γ is the pushdown alphabet, $\mathbb{N}$ denotes the set of the positive integers (see Definition 2.1), $\mathbb{N}$, Q, and Γ are pairwise disjoint, $T \subseteq \Gamma$, $\# \in \Gamma - T$, and the symbol $\#$ is said to be a *pushdown bottom*, $R \subseteq (\mathbb{N} \times Q \times (\Gamma - (T \cup \{\#\})) \times Q \times (\Gamma - \{\#\})^+) \cup (\mathbb{N} \times Q \times \{\#\} \times Q \times (\Gamma - \{\#\})^*\{\#\})$ is a finite relation, $s \in Q$ is the *starting state*, $S \in \Gamma$ is the *starting pushdown symbol*, $F \subseteq Q$ is the set of *final states*.

Instead of $(m, q, A, p, v) \in R$, we write $mqA \to pv \in R$ and call $mqA \to pv$ a *rule*; accordingly, R is referred to as the *set of rules* of M.

Definition 4.9. A *configuration of* M is a triple in $Q \times T^* \times (\Gamma - \{\#\})^*\{\#\}$. Let χ denote the set of all configurations of M. Let $x, y \in \chi$ be two configurations. M *pops* its pushdown from x to y, symbolically written as $x \; {}_p\!\Rightarrow y$, if $x = (q, az, au)$, $y = (q, z, u)$, where $a \in T$, $z \in T^*$, $u \in \Gamma^*$, $q \in Q$. M *expands* its pushdown from x to y by rule $r\colon mqA \to pv \in R$, symbolically written as $x \; {}_e\!\Rightarrow y$, if $x = (q, w, uAz)$, $y = (p, w, uvz)$, where $A \in \Gamma - T$, $u, z \in \Gamma^*$, $v \in \Gamma^+$, $q, p \in Q$, $w \in T^*$, and $occur(\Gamma - T, u) = m - 1$.

To express that M makes $x \; {}_e\!\Rightarrow y$ according to $mqA \to pv$, we write $x \; {}_e\!\Rightarrow y \; [mqA \to pv]$. We say that $mqA \to pv$ is a *rule of depth* m; accordingly, $x \; {}_e\!\Rightarrow y \; [mqA \to pv]$ is an *expansion of depth* m. M makes a *move* or computational step from x to y, symbolically written as $x \Rightarrow y$, if M either $x \; {}_e\!\Rightarrow y$ or $x \; {}_p\!\Rightarrow y$. If $n \in \mathbb{N}$ is the minimal positive integer such that each of rules in M is of depth n or less, we say that M *is of depth* n, symbolically written as ${}_nM$.

In the standard manner, extend ${}_p\!\Rightarrow$, ${}_e\!\Rightarrow$, and $\Rightarrow$ to ${}_p\!\Rightarrow^m$, ${}_e\!\Rightarrow^m$, and $\Rightarrow^m$, respectively, for $m \geq 0$; then, based on ${}_p\!\Rightarrow^m$, ${}_e\!\Rightarrow^m$, and $\Rightarrow^m$, define ${}_p\!\Rightarrow^+$, ${}_p\!\Rightarrow^*$, ${}_e\!\Rightarrow^+$, ${}_e\!\Rightarrow^*$, $\Rightarrow^+$, and $\Rightarrow^*$.

Let M be of depth n, for some $n \in \mathbb{N}$. We define the *language accepted by* ${}_nM$, $L({}_nM)$, as $L({}_nM) = \{w \in T^* | (s, w, S\#) \Rightarrow^* (f, \varepsilon, \#) \text{ in } {}_nM \text{ with } f \in F\}$.

For every $k \geq 1$, set ${}_k\mathscr{L}(\mathbf{DTDP}) = \{L({}_iM) \mid {}_iM \text{ is a deep pushdown automaton, } 1 \leq i \leq k\}$.

[1] The top of the pushdown is accessed by operations *push* and *pop*.

Consider the relationship between the n-limitation in grammars and the depth n in DTDP. When we write the pushdown configuration component as a string from the left to the right where the leftmost symbol is the top of the pushdown and the rightmost symbol is its bottom, we can compare the rewriting in this component with the rewriting in sentential configuration component in grammars. By the n-limitation in grammars, we restrict from the left the number of nonterminals that can be rewritten without giving the concrete order of the nonterminal. In case of DTDP, the restriction of maximum depth n is similar to the n-limitation. In addition, we specify the expansion of the concrete variable given by its order from the top of the pushdown content. Again, the similarity between grammars and automata is illustrated.

Example 4.5. Consider DTDP ${}_2M = (\{s, q, p, f\}, \{a, b, c\}, \{A, B, S, \#, a, b, \mathrm{c}\}, R, s, S, \{f\})$ with

$$\begin{aligned} R = \{ \quad & 1sS \to qAB, \\ & 1qA \to paAb, \\ & 1qA \to pab, \\ & 2pB \to qBc, \\ & 2pB \to fc \qquad \}. \end{aligned}$$

With $aabbcc$, ${}_2M$ makes

$$\begin{aligned} (s, aabbcc, S\#) &\ {}_e\Rightarrow\ (q, aabbcc, AB\#) && [1sS \to qAB] \\ &\ {}_e\Rightarrow\ (p, aabbcc, aAbB\#) && [1qA \to paAb] \\ &\ {}_p\Rightarrow\ (p, abbcc, AbB\#) \\ &\ {}_e\Rightarrow\ (q, abbcc, AbBc\#) && [2pB \to qBc] \\ &\ {}_e\Rightarrow\ (p, abbcc, abbBc\#) && [1qA \to pab] \\ &\ {}_p\Rightarrow\ (p, bbcc, bbBc\#) \\ &\ {}_p\Rightarrow\ (p, bcc, bBc\#) \\ &\ {}_p\Rightarrow\ (p, cc, Bc\#) \\ &\ {}_e\Rightarrow\ (f, cc, cc\#) && [2pB \to fc] \\ &\ {}_p\Rightarrow\ (f, c, c\#) \\ &\ {}_p\Rightarrow\ (f, \varepsilon, \#). \end{aligned}$$

We write $(s, aabbcc, S\#) \Rightarrow^* (f, \varepsilon, \#)$, and we say that the string $aabbcc$ is successfully accepted by DTDP ${}_2M$. Observe that $L({}_2M) = \{a^n b^n c^n | n \geq 1\} \in {}_2\mathscr{L}(\mathbf{DTDP})$, and $L({}_2M) \in \mathbf{CS} - \mathbf{CF}$.

Observation 4.1. Similarly to #-rewriting systems, the deep pushdown automata define the restriction implicitly by the form of the rules and by the requirement of finiteness of the set of the rules. That is, there is always some $k \geq 1$ for a DTDP M, such that $k = \max(\{m |\ mqA \to pv \in R_M\})$.

Before we start, recall Theorem 4.1 from [Med06].

Theorem 4.1 ([Med06]). *For every* $k \geq 1$, ${}_k\mathscr{L}(\mathbf{DTDP}) = {}_k\mathscr{L}(\mathbf{ST})$ *and* ${}_k\mathscr{L}(\mathbf{DTDP}) \subset {}_{k+1}\mathscr{L}(\mathbf{DTDP})$.

Proof of Theorem 4.1. For the construction and rigorous proof, see [Med06]. The proof uses the analogy between the restriction of DTDP to depth n and the restriction of the state grammars to n-limitation. It demonstrates the equivalence between these two rewriting systems. Since the families generated by n-limited state grammars, $n \geq 1$, establish the infinite hierarchy of these families (proved by Kasai in [Kas70]), the deep pushdown automata of depth n create the same infinite hierarchy as well.

A few modifications of deep pushdown automata are introduced in [KM06].

Next section discusses some open problems (see [Med06]) and hypothesis concerning the determinism and erasing rules in deep pushdown automata.

4.2.1 Deterministic Deep Pushdown Automata

The natural and mathematically elegant modification of a deep pushdown automaton is the definition of the determinism with respect to the depth of its expansions. It means that when we choose to expand an active symbol in the particular depth i, then there is at most one possibility which rule to select in the current state. That is, in the definition of deep pushdown automata, we rule out two rules of the forms $r_1 \colon 1pA \to qx$ and $r_2 \colon 2pC \to oy$.

Definition 4.10. M is a *deterministic deep pushdown automaton with respect to the depth of its expansions* if for every $q \in Q$, $q \in Q$, $\mathrm{card}(\{m \mid mqA \to pv \in R,\ A \in \Gamma - T,\ v \in \Gamma^+,\ p \in Q\}) \leq 1$ because at this point from the same state, all expansions that M can make are of the same depth. We say that M is a *strictly deterministic* DTDP if for every $mqA \to pv \in R$, $\mathrm{card}(\{mqA \to ow \in R \mid o \in Q,\ w \in \Gamma^+\} - \{mqA \to pv\}) = 0$.

Notice that ${}_2M$ from Example 4.5 is deterministic with respect to the depth of its expansions but ${}_2M$ is not strictly deterministic.

4.2.2 Reducing Deep Pushdown Automata

This section presents the variant of deep pushdown automata that fundamentally modifies its behavior. Deep pushdown automata (described in Section 4.2 and in [KM06, Med06]) are based on the generalization of the top-down parsers that have an access deeper into their pushdowns. The basic idea of the following modification of these automata is to generalize a bottom-up parser so it works similarly to the bottom-up analysis simulation of context-free grammars in classical pushdown automata except it reads the input from the right to the left. Moreover, instead of expanding operation as in original

deep pushdown automata, reducing pushdown automata uses operation reduction that replaces a string onto the pushdown by a non-input symbol (see [KMS06b, KS06a]). To see this new rewriting system in a wider context, we discuss both directions of reading the input sentence—first, from the right to the left and, then, from the left to the right.

The next chapter presents the equivalence of reducing deep pushdown automata with n-limited state grammars and infinite hierarchy of language families based on the depth of pushdown reductions.

Motivation and Principle

In the theory of formal languages, there exists several cases of a generalization of pushdown automata (see [LM05, Med99, Med06]). Specifically, there is often added an ability of touching into deeper parts of the pushdown. For instance, [Med06] introduces a top-down parser in this way and he points out the raising infinite hierarchy of languages accepted by this automata.

Regarding classical pushdown automata, the top-down and the bottom-up approach are equivalent in general. While restricting these rewriting systems to models based on the deterministic versions of context-free grammars, this equivalence may not hold. For instance, neither LL(k) grammars for the top-down approach nor LR(k) grammars for bottom-up approach are as powerful as the context-free grammars. In the practice, the LR(k) are more suitable than LL(k) since they allow us to describe a wider variety of language constructs.

Consider the frequently used method for the bottom-up parsing—the LR parser (Left-to-right Rightmost parser) and the relation between the language hierarchies of LR(k) and LL(k) for $k \geq 0$ (see [AP02]).

The main motivation to study reducing deep pushdown automata is the question of the relationship of both approaches (top-down and bottom-up) with respect to the depth in which the reductions are allowed.

Consider the standard simulation of a context-free grammar by a classical pushdown automaton acting as a general bottom-up parser (see [Med00]). During every move, the parser either shifts or reduces its pushdown depending on the top pushdown symbol, current input symbol, and state. *Shift* operation takes one input symbol from the input tape and moves it to the top of the pushdown.

If a string on the top of the pushdown equals to any right-hand side of a context-free rule, this string is *reduced* to one non-input symbol. In this rewriting system, the set of variables equals to the pushdown alphabet because every symbol (even input symbol) can be contained in the rewritten substring by reduction.

The automaton accepts an input string, x, if it makes a sequence of moves so it completely reads x, empties its pushdown, and enters a final state; the latter requirement of entering a final state is dropped in some books (see, for instance, Theorem 5.1 in [RS97]).

Being inspired by the previous paragraphs, let us introduce generalized versions of top-down (see Section 4.2) and bottom-up parsers.

Hereafter, the generalized bottom-up parser represented by a pushdown automaton works exactly the same as the basic parser based on Definition 2.19 except that

a) it reads the input tape from the right to the left and
b) it makes reductions of depth m so it replaces the pushdown substring with mth topmost non-input symbol in the pushdown, for some $m \geq 1$.

Such an automaton is called *right-to-left reducing deep pushdown automaton* (abbrev. ^{rl}RDPDA). Reading the input tape from the right to the left is common in some natural languages and leads to the simulation of the rightmost reduction, which is analogical to the rightmost derivation. However, a modification of RDPDA that reads from the left to the right is mentioned as well.

Definition 4.11. A *reducing deep pushdown automaton* (RDPDA) is a septuple, $M = (Q, T, \Gamma, R, s, S, F)$, where Q is a finite set of states, T is an input alphabet, and Γ is a pushdown alphabet, $\mathbb{N}$, Q, and Γ are pairwise disjoint (see Definition 2.1 for $\mathbb{N}$), $T \subseteq \Gamma$, $\Gamma - T$ contains a special *pushdown bottom* symbol denoted by $\#$, $R \subseteq (Q \times \Gamma^+ \times \mathbb{N} \times Q \times (\Gamma - T))$ is a finite relation, $s \in Q$ is the *starting state*, $S \in \Gamma$ is the *starting pushdown symbol*, $F \subseteq Q$ is a set of final states. Instead of $(q, v, m, p, A) \in R$, we write $qv \vdash mpA \in R$ and call $qv \vdash mpA$ a rule; accordingly, R is referred to as the *set of rules* of M.

A *configuration* of M is a triple in $Q \times T^* \times (\Gamma - \{\#\})^*\{\#\}$. The leftmost symbol of the third configuration component is said to be the *top of the pushdown*. Let χ denote the set of all configurations of M. Let $x, y \in \chi$ be two configurations and $p, q \in Q$ two states. M *shifts* its input to pushdown from x to y; that is, M rewrites x to y in one of these two ways:

a) $x \, {}_s\Rightarrow y$, if $x = (q, ua, z)$, $y = (q, u, az)$, where $a \in T$, $u \in T^*$, $z \in \Gamma^*$. Then, M is said to be *right-to-left reducing* (^{rl}RDPDA).
b) $x \, {}_s\Rightarrow y$, if $x = (q, au, z)$, $y = (q, u, az)$, where $a \in T$, $u \in T^*$, $z \in \Gamma^*$. Then, M is said to be *left-to-right reducing* (^{lr}RDPDA).

M *reduces* its pushdown from x to y, symbolically written as $x \, {}_r\Rightarrow y$, if $x = (q, w, uvz)$, $y = (p, w, uAz)$, and $qv \vdash mpA \in R$, where $w \in T^*$, $A \in (\Gamma - T - \{\#\})$, $u, z \in \Gamma^*$, $v \in \Gamma^+$, and $occur(\Gamma - T, u) = m - 1$.

To express that M makes $x \, {}_r\Rightarrow y$ according to $qv \vdash mpA$, we write $x \, {}_r\Rightarrow y$ $[qv \vdash mpA]$. We say that $qv \vdash mpA$ is a *rule of depth* m; accordingly, $x \, {}_r\Rightarrow y$ $[qv \vdash mpA]$ is a *reduction of depth* m. M makes a *move* from x to y, symbolically written as $x \Rightarrow y$, if M makes either $x \, {}_s\Rightarrow y$ or $x \, {}_r\Rightarrow y$.

If $n \in \mathbb{N}$ is the minimal positive integer such that every rule of M is of depth n or less, we say that M is of depth n, symbolically written as ${}_nM$.

In the standard manner, extend ${}_s\Rightarrow$, ${}_r\Rightarrow$, and $\Rightarrow$ to ${}_s\Rightarrow^m$, ${}_r\Rightarrow^m$, and $\Rightarrow^m$ respectively, for $m \geq 0$; then, based on ${}_s\Rightarrow^m$, ${}_r\Rightarrow^m$, and $\Rightarrow^m$, define ${}_s\Rightarrow^+$, ${}_r\Rightarrow^+$, $\Rightarrow^+$, ${}_s\Rightarrow^*$, ${}_r\Rightarrow^*$, $\Rightarrow^*$.

Let M be of depth n, for some $n \in \mathbb{N}$. We define the language accepted by ${}_nM$, $L({}_nM)$, as $L({}_nM) = \{w \in T^* |\ (s, w, \#) \Rightarrow^* (f, \varepsilon, S\#)$ in ${}_nM$ with $f \in F\}$.

For every $k \geq 1$, ${}_k\mathscr{L}(\mathbf{^{rl}RDPDA})$ and ${}_k\mathscr{L}(\mathbf{^{lr}RDPDA})$ denotes the family of languages defined by the right-to-left reducing and the left-to-right reducing deep pushdown automata of depth i, respectively, where $1 \leq i \leq k$.

The following example shows a reduction of a sentence by a ^{rl}RDPDA.

Example 4.6. Consider a ^{rl}RDPDA, ${}_2M = (\{s, p, q, t, f\}, \{a, b, c\}, \{a, b, c, A, C, S\}, R, s, S, \{f\})$ with R containing rules

1. $sab \vdash 1pA$
2. $pc \vdash 2qC$
3. $qaAb \vdash 1tA$
4. $tcC \vdash 2qC$
5. $qAC \vdash 1fS$

With *aabbcc* read from the right to the left from the input tape, ${}_2M$ makes the following sequence of moves

$(s, aabbcc, \#)\ {}_s\Rightarrow (s, aabbc, c\#)\ {}_s\Rightarrow (s, aabb, cc\#)\ {}_s\Rightarrow (s, aab, bcc\#)\ {}_s\Rightarrow (s, aa, bbcc\#)\ {}_s\Rightarrow (s, a, abbcc\#)\ {}_r\Rightarrow (p, a, Abcc\#)$ [1] ${}_r\Rightarrow (q, a, AbcC\#)$ [2] ${}_s\Rightarrow (q, \varepsilon, aAbcC\#)\ {}_r\Rightarrow (t, \varepsilon, AcC\#)$ [3] ${}_r\Rightarrow (q, \varepsilon, AC\#)$ [4] ${}_r\Rightarrow (f, \varepsilon, S\#)$ [5].

Therefore, $aabbcc \in L({}_2M)$. Observe that $L({}_2M) = \{a^n b^n c^n |\ n \leq 1\}$. Notice that $L({}_2M) \in \mathbf{CS} - \mathbf{CF}$.

As demonstrated in Chapter 5, considering the powers of deep pushdown automata and their modifications, they fulfill the space between automata models for context-free and context-sensitive languages in the formal language theory. The power of these new formal models of depth n is proved by the equivalence to the family of n-limited languages generated by state grammars.

4.3 Restricted Pushdown Automata

This section defines a new formal model—restricted pushdown automaton—based on a modification of classical pushdown automaton (see Definition 2.19). The content of the pushdown is restricted by a *restricting language*. This generalized way of configuration restriction relates to the first component of the configuration—pushdown-string (see Definition 2.20). During the application of a rule in this rewriting system, the pushdown-string is checked whether its content creates a sentence of the restricting language; otherwise, the rule is inapplicable. The resulting power of a restricted pushdown automaton depends

on the classification of the restricting language in the Chomsky hierarchy of language families (see Theorem 2.2).

The results show (see Theorem 5.13) that regular restricting language does not increase the accepting power of restricted pushdown automata in comparison with classical pushdown automata; that is, the family of context-free languages. Further, Example 4.7 demonstrates that linear restricting language increases the power of restricted pushdown automata significantly.

Definition 4.12. A *restricted pushdown automaton*, RPDA for short, is a pair $H = (M, \Xi)$, where $M = (Q, T, \Gamma, R, s, S, F)$ is a classical pushdown automaton with reversed pushdown-string (see Definition 2.19) and $\Xi \subseteq \Gamma^*$ is a *restricting language*.

The definition of a configuration of RPDA H is the same as that of a classical pushdown automaton with reversed pushdown-string. Let us define a computational step and the language accepted by this new rewriting system.

Definition 4.13. First, let us define the set of all possible pushdown-strings of $H = (M, \Xi)$, $M = (Q, T, \Gamma, R, s, S, F)$, during the acceptance of w as $K(M, w) = \{\gamma \mid w \in T^*,\ w \in L(M),\ (S, s, w) \Rightarrow_M^* (\gamma, q, u) \Rightarrow_M^* (\gamma_F, q_F, \varepsilon),\ u \in \text{suffixes}(w),\ q \in Q,\ q_F \in F,\ \text{and}\ \gamma, \gamma_F \in \Gamma^*\}$.

$K(M, w)$ contains all strings that can occur in the pushdown during a computation that accepts w by M.

Definition 4.14. The language accepted by a restricted pushdown automaton, $H = (M, \Xi)$,

$$L(H) = \{w \mid w \in L(M),\ K(M, w) \subseteq \Xi\}.$$

The family of languages accepted by RPDA, where the restricting language belongs to the family of languages $\mathscr{L}(\mathbf{X})$, is denoted by $\mathscr{L}(\mathbf{RPDA}, X)$.

Note 4.2. By analogy with the definition of finite index of #-rewriting systems, RPDA allows a weaker restriction in an alternative definition as well. The alternative definition does not restrict every computation of a sentence, but it requires the existence of at least one computation that satisfies the restriction of the pushdown-string of the automaton by restricting language Ξ. The weaker definition of the language accepted by RPDA $H = (M = (Q, T, \Gamma, R, s, S, F), \Xi)$, $L(H) = \{w|$ there exists a computation $(S, s, w) = (\gamma_0, q_0, u_0) \Rightarrow_M (\gamma_1, q_1, u_1) \Rightarrow_M \ldots \Rightarrow_M (\gamma_n, q_n, u_n) = (\varepsilon, q_F, \varepsilon)$ such that $n \geq 1$, $\gamma_i \in \Xi$, $q_i \in Q$, $q_F \in F$, $u_i \in \text{suffixes}(w)$ for all $1 \leq i \leq n\}$.

Example 4.7. Let $M = (Q, T, T \cup \{\#, \Delta\}, R, \#, s, \{f\})$ be a pushdown automaton with reversed pushdown-string, where $Q = \{s, p, f\}$, $T = \{a, b, c\}$, and R contains:

$r_1 : \#s\varepsilon \to \#cs$,
$r_2 : cs\varepsilon \to ccs$,
$r_3 : csa \to cas$,
$r_4 : asa \to aas$,
$r_5 : as\varepsilon \to a\Delta s$,
$r_6 : \Delta s\varepsilon \to \varepsilon q$,
$r_7 : aqb \to \varepsilon q$,
$r_8 : cqc \to \varepsilon q$,
$r_9 : \#q\varepsilon \to \varepsilon f$.

The linear restricting language, $L(G)$, is defined by linear grammar $G = (\Sigma, T, P, S)$, $\Sigma = N \cup T$, where $N = \{S, X, Y\}$, $T = \{a, c, \#, \Delta\} = \Gamma_M$, and P contains:

$p_1 : S \to \#cX$,
$p_2 : S \to \#$,
$p_3 : S \to \varepsilon$,
$p_4 : S \to \#cYa\Delta$,
$p_5 : X \to cXa$,
$p_6 : X \to cX$,
$p_7 : X \to \varepsilon$,
$p_8 : Y \to cYa$,
$p_9 : Y \to \varepsilon$.

Obviously, since every rule of P contains at most one nonterminal symbol on its right-hand side, G is a linear grammar; that is, $L(G) \in \mathbf{LIN}$ and $L(G) = \{\varepsilon\} \cup \{\#c^n a^m |\ m, n \geq 0,\ m \leq n\} \cup \{\#c^n a^n \Delta |\ n \geq 1\}$.

Let us define a restricted pushdown automaton, $H = (M, L(G))$, and observe that $L(H) = \{a^n b^n c^n |\ n \geq 1\}$.

- Indeed, rules r_1 and r_2 from R non-deterministically generate necessary number of cs.
- Rules r_3 and r_4 read and move as from the input onto the pushdown.
- Rule r_5 non-deterministically decides that all as are already read and inserts Δ onto the pushdown. Now, the regulation restricts following steps by the subset of $L(G)$, $\{\#c^n a^n \Delta |\ n \geq 1\}$. Notice that every string of the subset ends with Δ that ensures the same number of as and cs in this phase of the computation.
- Further, rule r_6 pops Δ.
- Finally, rules r_7 and r_8 read bs and cs from the input and accordingly remove the corresponding as and cs on the pushdown. If the input and pushdown are empty, the last rule r_9 changes the current state to the final state f and pops the bottom of the pushdown.

For instance, H computes $aabbcc$ as follows:

$$\begin{aligned} & (\#, s, aabbcc) \\ \Rightarrow_H\ & (\#c, s, aabbcc)\ [r_1] \end{aligned}$$

$\Rightarrow_H (\#cc, s, aabbcc)\ [r_2]$
$\Rightarrow_H (\#cca, s, abbcc)\ [r_3]$
$\Rightarrow_H (\#ccaa, s, bbcc)\ [r_4]$
$\Rightarrow_H (\#ccaa\Delta, s, bbcc)\ [r_5]$
$\Rightarrow_H (\#ccaa, q, bbcc)\ [r_6]$
$\Rightarrow_H (\#cca, q, bcc)\ [r_7]$
$\Rightarrow_H (\#cc, q, cc)\ [r_7]$
$\Rightarrow_H (\#c, q, c)\ [r_8]$
$\Rightarrow_H (\#, q, \varepsilon)\ [r_8]$
$\Rightarrow_H (\varepsilon, f, \varepsilon)\ [r_9]$.

Moreover, observe that $\{\#, \#c, \#cc, \#cca, \#ccaa, \#ccaa\Delta, \varepsilon\} \subseteq L(G)$. For instance, $S \Rightarrow_G \#cYa\Delta\ [p_4]\ \Rightarrow_G \#ccYaa\Delta\ [p_8]\ \Rightarrow_G \#ccaa\Delta\ [p_9]$.

From the practical point of view, the automaton is restricted by restricting language only in the computational step by the fifth rule when the condition of the same number of as and cs has to be satisfied. In other steps of M, the restriction has no effect on the computation. To inspire the reader, in the practice, only a few steps can be restricted by the additional mechanism during the computation.

4.4 Summary

Let us recall the most fundamental presented rewriting systems (see Table 4.2).

Name	**Abbreviation**	**Language Family**
Right-Linear Grammars	RLIN	**REG**
Context-Free Grammars	CF	**CF**
Context-Sensitive Grammars	CS	**CS**
#-Rewriting Systems	CF#RS	$\mathscr{L}$(**CF#RS**)
Generalized #-Rewriting Systems	G#RS	$\mathscr{L}$(**G#RS**)
n-Right-Linear #-Rewriting Systems	n-RLIN#RS	$\mathscr{L}$(n-**RLIN#RS**)
Restricted Pushdown Automata	RPDA	$\mathscr{L}$(**RPDA**, X)
Deep Pushdown Automata	DTDP	$\mathscr{L}$(**DTDP**)
Reducing Deep Pushdown Automata	RDPDA	$\mathscr{L}$(**RDPDA**)
State Grammars	ST	$\mathscr{L}$(**ST**)
Programmed Grammars	PG	$\mathscr{L}$(**P**)
Random-context Grammars	RC	$\mathscr{L}$(**RC**)
m-Parallel n-Right-Linear Simple Matrix Grammars	m-Pn-G	$\mathscr{L}$(m-**P**n-**G**)

Table 4.2. Survey of the most important rewriting systems used in this book

5

Results

This monograph describes and, more importantly, demonstrates all valuable results concerning various restrictions of rewriting systems with the focus on the new of them. Most of the results can be classified into two sections: (1) the power of rewriting systems and (2) infinite hierarchies of language families based on restrictions of rewriting systems.

Convention 5.1. All results (lemmas and theorems) in this book are established by construction proofs that should be ideally completed by rigorous induction proofs to confirm their correctness. Nevertheless, these induction proofs are often omitted and left to the reader as an exercise.

5.1 Power of Rewriting Systems

In this section, we introduce the generative power of #-rewriting systems with several forms of rules and with the finite index restriction. That is, for arbitrary $k \geq 1$, the configuration contains k or fewer bounders.

5.1.1 Context-Free #-Rewriting Systems

This section establishes an infinite hierarchy of language families resulting from the context-free #-rewriting systems defined in the previous chapter. More concretely, this well-known infinite hierarchy of language families results from programmed grammars of finite index (see Theorem 3.1.2i and Theorem 3.1.7 in [DP89]).

From a broader perspective, this result thus demonstrates that rewriting systems based on a combination of grammars and automata are naturally related to some classical topics and results concerning formal languages, on which they can shed light in an alternative way.

Similarly to the most of the equivalence proofs of two formal models, we demonstrate both directions of inclusions of these resulting language families.

From the computer science point of view, the construction of the transformation between these two formal models with preserving the same defined language is natural.

The basic idea of the construction proof comes from the alternative version of the proof of the equivalence $\mathscr{L}_{fin}(\mathbf{RC}) = \mathscr{L}_{fin}(\mathbf{P})$ (see [KM05b]).

Lemma 5.1. *For every $k \geq 1$, $\mathscr{L}_k(\mathbf{P}) \subseteq \mathscr{L}_k(\mathbf{CF\#RS})$.*

Proof of Lemma 5.1. Let $k \geq 1$ be a positive integer. Let $G = (\Sigma, T, P, S)$ be a programmed grammar of index k, where $N = \Sigma - T$. We construct the #-rewriting system of index k, $H = (Q, T \cup \{\#\}, s, R)$, where $\# \notin T$, $s = \langle\sigma\rangle$, σ is a new symbol, and R and Q are constructed by performing the following steps:

1. For each $\lfloor p\colon S \to \alpha, g(p)\rfloor \in P$, $\alpha \in \Sigma^*$, add $\langle\sigma\rangle_1\# \to \langle[p]\rangle\#$ to R, where $\langle[p]\rangle$ is a new state in Q and $g(p)$ denotes the success field of p.
2. If $A_1A_2 \ldots A_j \ldots A_h \in N^*$, $h \in \{1, 2, \ldots, k\}$, $\lfloor p\colon A_j \to x_0B_1x_1B_2x_2 \ldots x_{n-1}B_nx_n,\ g(p)\rfloor \in P$, $j \in \{1, 2, \ldots, h\}$ for $n \geq 0$, $x_0, x_t \in T^*$, $B_t \in N$, $1 \leq t \leq n$ and $n + h - 1 \leq k$, then
   (a) if $g(p) = \emptyset$, then $\langle A_1A_2 \ldots A_{j-1}[p]A_{j+1} \ldots A_h\rangle$, $\langle A_1A_2 \ldots B_1 \ldots B_n \ldots A_h\rangle$ are new states in Q and the rule $\langle A_1A_2 \ldots A_{j-1}[p]A_{j+1} \ldots A_h\rangle_j\# \to \langle A_1\ A_2 \ldots B_1 \ldots B_n \ldots A_h\rangle\ x_0\#x_1 \ldots x_{n-1}\#x_n$ is added to R;
   (b) for every $q \in g(p)$, $q\colon D_d \to \alpha$, $\alpha \in \Sigma^*$ add new states $\langle A_1A_2 \ldots A_{j-1}[p]\ A_{j+1} \ldots A_h\rangle$ and $\langle D_1D_2 \ldots D_{d-1}[q]D_{d+1} \ldots D_{n+h-1}\rangle$ to Q and add the following rule to R:
   $\langle A_1A_2\ \ldots\ A_{j-1}[p]A_{j+1}\ \ldots\ A_h\rangle_j\# \to \langle D_1D_2\ \ldots\ D_{d-1}[q]D_{d+1}\ \ldots\ D_{n+h-1}\rangle\ x_0\#x_1 \ldots x_n$, where, with an exception of $D_d = [q]$, $A_1 \ldots A_{j-1}B_1\ \ldots\ B_nA_{j+1}\ \ldots\ A_h = D_1\ \ldots\ D_{h+n-1}$, $B_1 \ldots B_n = D_j \ldots D_{j+n-1}$ for some $d \in \{1, 2, \ldots, n + h - 1\}$.

Basic Idea of the Proof of Lemma 5.1. H simulates derivations in G. The information necessary for this simulation is recorded inside of states in new rules of H. Each state in Q carries a string of nonterminals from N^*, where one symbol of this string is replaced with the label of a rule in P.

Let $x_0A_1x_1 \ldots x_{h-1}A_hx_h$ be a sentential form derived by G, where $x_i \in T^*$ for $0 \leq i \leq h$ and $A_l \in \Sigma - T$ for $1 \leq l \leq h$, and let $\lfloor p\colon A_j \to \alpha, g(p)\rfloor$ be a rule in P applicable in the next step to A_j, $1 \leq j \leq h$. Then, the new configuration of H is of the form $\langle A_1A_2 \ldots A_{j-1}[p]A_{j+1} \ldots A_h\rangle x_0\#x_1 \ldots x_{h-1}\#x_h$, which encodes the nonterminals in the sentential form of G and the next applicable rule label. To simulate p, the introduced simulation rule rewrites jth bounder to α and changes the current state so it reflects the new list of nonterminals in a new corresponding configuration of G. The change of the current state also non-deterministically chooses a rule from $g(p)$ to be applied in the subsequent simulation.

Rigorous Proof:

Claim 5.1.1. *If* $S \Rightarrow^m x_0A_1x_1A_2x_2 \dots x_{h-1}A_hx_h$ *in* G, *then* $\langle\sigma\rangle\# \Rightarrow^r \langle A_1A_2 \dots A_h\rangle\ x_0\#x_1 \dots x_h\ [q_1q_2 \dots q_r]$ *in* H, *for* $m \geq 0$. *If* $g(q_r) \neq \emptyset$, *then exists a rule* $\lfloor q_{r+1}\colon A_j \to y_0B_1y_1 \dots y_{h-1}B_ny_n, g(q_{r+1})\rfloor$, *where* $n+h-1 \leq k$, $q_{r+1} \in g(q_r)$, *and* $A_j = [q_{r+1}]$, $q_1, \dots, q_r, q_{r+1} \in \mathrm{Lab}(R)$.

Proof of Claim 5.1.1. This claim is proved by induction on $m \geq 0$.

Induction Basis: Let $m = 0$. For $S \Rightarrow^0 S$ in G exists $\langle\sigma\rangle\# \Rightarrow^1 \langle[p]\rangle\#$ in H, where $\lfloor p\colon S \to \alpha, g(p)\rfloor \in P$ and $\langle\sigma\rangle_1\# \to \langle[p]\rangle\# \in R$.

Induction Hypothesis: Supposing that Claim 5.1.1 holds for all computations of length m or less for some $m \geq 0$.

Induction Step: We consider $S \Rightarrow^m y\ [p_1p_2 \dots p_m]$, where $y = x_0A_1x_1 \dots x_{h-1}A_hx_h$ and $p_1, \dots, p_m, p_{m+1} \in \mathrm{Lab}(P)$ such that $y \Rightarrow x\ [p_{m+1}]$. If $m = 0$, then $p_{m+1} \in \{p \mid lhs(p) = S,\ p \in \mathrm{Lab}(P)\}$; otherwise, $p_{m+1} \in g(p_m)$. For $\lfloor p_{m+1}\colon A_j \to y_0B_1y_1 \dots y_{n-1}B_ny_n,\ g(p_{m+1})\rfloor$, x is of the form $x = x_0A_1x_1 \dots A_{j-1}x_{j-1}y_0B_1y_1 \dots y_{n-1}B_ny_nx_jA_{j+1} \dots x_{h-1}A_hx_h$, where x_0, …, $x_h \in T^*$ and y_0, …, $y_n \in T^*$. By the induction hypothesis, there exists $\langle\sigma\rangle\# \Rightarrow^r \langle A_1A_2 \dots A_{j-1}[p_{m+1}]A_{j+1} \dots A_h\rangle x_0\#x_1 \dots x_{h-1}\#x_h\ [q_1q_2 \dots q_r] \Rightarrow \langle A_1A_2\ \dots\ A_{j-1}B_1\ \dots\ B_nA_{j+1}\ \dots\ A_h\rangle\ x_0\#\ \dots\ \#x_{j-1}y_0\#\ \dots\ \#y_nx_j\#\ \dots\ \#x_h\ [q_{r+1}]$, $r \geq 1$, $q_i \in \mathrm{Lab}(R)$, $1 \leq i \leq r+1$. If $g(p_{m+1}) \neq \emptyset$, then there exists a rule $p_{m+2} \in g(p_{m+1})$ and a sequence $D_1 \dots D_{n+h-1}$ so that $A_1A_2 \dots A_{j-1}B_1 \dots B_nA_{j+1} \dots A_h = D_1D_2 \dots D_{n+h-1}$, where for at most one $d \in \{1, 2, \dots, n+h-1\}$ is $D_d = [q_{r+2}]$, $q_{r+2} \in g(q_{r+1})$.

Claim 5.1.2. *If* $S \Rightarrow^z x$ *in* G, *then* $\langle\sigma\rangle\# \Rightarrow^* \langle\rangle x$ *in* H, *where* $z \geq 0$, $x \in T^*$.

Proof of Claim 5.1.2. Consider Claim 5.1.1 for $h = 0$. At this point, if $S \Rightarrow^z x_0$, then $\langle\sigma\rangle\# \Rightarrow^* \langle\rangle x_0$, so $x_0 = x$.

Thus, Claim 5.1.1 and Claim 5.1.2 formally prove Lemma 5.1. □

Lemma 5.2. *For every* $k \geq 1$, $\mathscr{L}_k(\mathbf{CF\#RS}) \subseteq \mathscr{L}_k(\mathbf{P})$.

Proof of Lemma 5.2. Let $k \geq 1$ be a positive integer. Let $H = (Q, T \cup \{\#\}, s, R)$ be a #-rewriting system of index k, where $\Sigma = T \cup \{\#\}$ and $T \cap \{\#\} = \emptyset$. We construct an equivalent programmed grammar of index k, $G = (\Sigma, T, P, S)$, where the set of nonterminals $N = \Sigma - T$ and the set of rules P are constructed as follows:

1. $S = \langle s, 1, 1\rangle$;

2. $N = \{\langle p, i, h\rangle \mid p \in Q, 1 \leq i \leq h, i \leq h \leq k\} \cup \{\langle q', i, h\rangle \mid q \in Q, 1 \leq i \leq h, i \leq h \leq k\} \cup \{\langle q'', i, h\rangle \mid q \in Q, 1 \leq i \leq h, i \leq h \leq k\} \cup \{\langle q'', 1, 0\rangle \mid q \in Q\}$;
3. For every rule $r\colon p\,{}_i\# \to qy \in R$, where $y = y_0\#y_1 \ldots y_{m-1}\#y_m$, $m \geq 0$, $y_0, y_1, \ldots, y_m \in T^*$, add the following set to P:

   (i) $\{\langle p, j, h\rangle \to \langle q', j, h+m-1\rangle,$
   $\{r' \mid$ if $j+1 = i$, then $r'\colon \langle p, i, h\rangle \to \langle q'', i, h+m-1\rangle$
   else $r'\colon \langle p, j+1, h\rangle \to \langle q', j+1, h+m-1\rangle\ \}$
   $\mid 1 \leq j < i,\ i \leq h \leq h_{max}\}$
   $\cup$

   (ii) $\{\langle p, i, h\rangle \to \langle q'', i, h+m-1\rangle,$
   $\{r' \mid$ if $i = h$, then $r'\colon \langle q'', i, h+m-1\rangle \to y_0\langle q', i, h+m-1\rangle y_1\langle q',$
   $i+1, h+m-1\rangle y_2 \ldots y_{m-1}\langle q', i+m-1, h+m-1\rangle y_m$
   else $r'\colon \langle p, i+1, h\rangle \to \langle q', i+1+m-1, h+m-1\rangle\}$
   $\mid i \leq h \leq h_{max}\}$
   $\cup$

   (iii) $\{\langle p, j, h\rangle \to \langle q', j+m-1, h+m-1\rangle,$
   $\{r' \mid$ if $j = h$, then $r'\colon \langle q'', i, h+m-1\rangle \to y_0\langle q', i, h+m-1\rangle y_1\langle q',$
   $i+1, h+m-1\rangle y_2 \ldots y_{m-1}\langle q', i+m-1, h+m-1\rangle y_m$
   else $r'\colon \langle p, j+1, h\rangle \to \langle q', j+1+m-1, h+m-1\rangle\}$
   $\mid i < j \leq h,\ i \leq h \leq h_{max}\}$
   $\cup$

   (iv) $\{\langle q'', i, h+m-1\rangle \to y_0\langle q', i, h+m-1\rangle y_1\langle q', i+1, h+m-1\rangle y_2 \ldots y_{m-1}$
   $\langle q', i+m-1, h+m-1\rangle y_m,$
   $\{r' \mid r'\colon \langle q', 1, h+m-1\rangle \to \langle q, 1, h+m-1\rangle\}$
   $\mid i \leq h \leq h_{max}\}$
   $\cup$

   (v) $\{\langle q', j, h+m-1\rangle \to \langle q, j, h+m-1\rangle,$
   $\{r' \mid$ if $j < h+m-1$, then $r'\colon \langle q', j+1, h+m-1\rangle \to \langle q, j+1, h+m-1\rangle$
   else $r'\colon \langle \tilde{p}, 1, h+m-1\rangle \to \langle \tilde{q}', 1, h+m-1+\tilde{m}-1\rangle$, where
   $\tilde{p}\,{}_{\tilde{i}}\# \to \tilde{q}\tilde{y}_0\#\tilde{y}_1 \ldots \tilde{y}_{\tilde{m}-1}\#\tilde{y}_{\tilde{m}} \in R$, $\tilde{y}_0, \tilde{y}_1, \ldots, \tilde{y}_{\tilde{m}} \in T^*$,
   if $\tilde{i} = 1$, then $\tilde{q}' := \tilde{q}''\}$
   $\mid 1 \leq j \leq h+m-1,\ i \leq h \leq h_{max}\}$,

where $h_{max} = k$ if $m = 0$; otherwise $h_{max} = k - m + 1$.

Basic Idea of the Proof of Lemma 5.2. By several derivation steps, G simulates a single step in H. Inside of every nonterminal of the form $\langle p, i, h\rangle$ occurring in a sentential form of G, we record

(1) p—the current state of H;
(2) i—the position of the occurrence of # in the current configuration of H;
(3) h—the total number of all #s in the current configuration.

From these three pieces of information and the set $g(p)$ associated with p, we find out whether p is applicable in the next step and if so, we simulate

the step by rules introduced in the third step of the above construction as follows:

(a) inside of all nonterminals in the sentential form, change h to $h+m-1$, where m is the number of nonterminals occurring on the right-hand side of p, so $h+m-1$ is the total number of nonterminals after the application of p (see (i) through (iii));
(b) in the nonterminals that follow the rewritten nonterminal, change their position so it corresponds to the position after the application of p (see (iii));
(c) apply p and select a rule label q from $g(p)$ to be applied in the next step (see (iv));
(d) complete the simulated derivation step in H by rules introduced in (v).

Rigorous Proof:

Claim 5.2.1. *If* $\langle\sigma\rangle\# \Rightarrow^c \langle\vartheta\rangle y_0\#y_1 \dots y_{n-1}\#y_n$ *in* H, *then* $S \Rightarrow^* y_0A_1y_1 \dots y_{n-1}A_ny_n$ *in* G *for some* $c \geq 0$.

Proof of Claim 5.2.1. The claim is proved by induction on c.

Induction Basis: Let $c = 0$. For $\langle\sigma\rangle\# \Rightarrow^0 \langle\sigma\rangle\#$ in H there exists $S \Rightarrow^0 S$ in G.

Induction Hypothesis: Suppose that Claim 5.2.1 holds for all computations of length c or less for some $c \geq 0$.

Induction Step: Consider $\langle\sigma\rangle\# \Rightarrow^c \langle\vartheta\rangle y_0\#y_1 \dots y_h\ [r_1r_2 \dots r_c]$ in H, $r_t \in \mathrm{Lab}(R)$, $1 \leq t \leq c$ and $r_{c+1}\colon \langle\vartheta\rangle_i\# \to \langle\omega\rangle x_0\#x_1 \dots x_{m-1}\#x_m \in R$, $x_0, \dots, x_m \in T^*$ so that $\langle\vartheta\rangle y_0\# \dots \#y_h \Rightarrow \langle\omega\rangle y_0\#y_1\# \dots \#y_{i-1}x_0\#x_1\# \dots \#x_my_i\#y_{i+1}\# \dots \#y_h\ [r_{c+1}]$. Based on Claim 5.2.1 there exists also a derivation $D_{1*} : y_0A_1 \dots A_hy_h \Rightarrow^* y_0A_1y_1 \dots y_{i-1}x_0B_1x_1 \dots B_mx_my_iA_{i+1} \dots A_hy_h$ in G. Based on the construction part of the proof, we show that such a derivation exists.

Let us have a form $y_0A_1y_1 \dots A_hy_h$. Rename nonterminals A_t to $\langle\vartheta, t, h\rangle$ for $1 \leq t \leq h$ and get a base form $y_0\langle\vartheta, 1, h\rangle y_1 \dots y_{h-1}\langle\vartheta, h, h\rangle y_h$ which starts the simulation of the D_{1*} derivation. This simulation must come out of the continuous application of rules from the third construction step.

(3i) $\forall j : 1 \leq j < i$ apply rules of the form $\langle p, j, h\rangle \to \langle q', j, h+m-1\rangle$:
$F_1 = y_0\langle\vartheta, 1, h\rangle y_1 \dots y_{h-1}\langle\vartheta, h, h\rangle y_h \Rightarrow y_0\langle\omega', 1, h+m-1\rangle y_1\langle\vartheta, 2, h\rangle y_2 \dots y_{h-1}\langle\vartheta, h, h\rangle y_h \Rightarrow^{i-2} y_0\langle\omega', 1, h+m-1\rangle y_1 \dots y_{i-2}\langle\omega', i-1, h+m-1\rangle y_{i-1}\langle\vartheta, i, h\rangle y_i \dots y_{h-1}\langle\vartheta, h, h\rangle y_h = F_2$;
(3ii) apply $\langle p, i, h\rangle \to \langle q'', i, h+m-1\rangle$:
$F_2 \Rightarrow y_0\langle\omega', 1, h+m-1\rangle y_1 \dots y_{i-1}\langle\omega'', i, h+m-1\rangle y_i \dots y_{h-1}\langle\vartheta, h, h\rangle = F_3$;
If $i = h$, then set $F_4 = F_3$ and continue with (3iv); otherwise, with (3iii);

(3iii) $\forall j : i < j \leq h$ apply rules of the form $\langle p, j, h\rangle \rightarrow \langle q', j+m-1, h+m-1\rangle$:
$F_3 \Rightarrow y_0\langle\omega', 1, h+m-1\rangle y_1 \dots y_{i-2}\langle\omega', i-1, h+m-1\rangle y_{i-1}\langle\omega'', i, h+m-1\rangle y_i\langle\omega', i+m, h+m-1\rangle\ y_{i+1}\langle\vartheta, i+2, h\rangle y_{i+2} \dots y_{h-1}\langle\vartheta, h, h\rangle y_h \Rightarrow^{h-i-1} y_0\langle\omega', 1, h+m-1\rangle y_1 \dots y_{i-1}\langle\omega'', i, h+m-1\rangle y_{i+1} \dots y_{h-1}\langle\omega', h+m-1, h+m-1\rangle y_h = F_4$;

(3iv) apply $\langle q'', i, h+m-1\rangle \rightarrow y_0\langle q', i, h+m-1\rangle y_1 \dots y_{m-1}\langle q', i+m-1, h+m-1\rangle y_m$:
$F_4 \Rightarrow y_0\langle\omega', 1, h+m-1\rangle y_1 \dots y_{i-1}x_0\langle\omega', i, h+m-1\rangle x_1 \dots x_{m-1}\langle\omega', i+m-1, h+m-1\rangle x_m y_i \dots y_{h-1}\langle\omega', h+m-1, h+m-1\rangle y_h = F_5$;

(3v) $\forall j : 1 \leq j \leq h+m-1$ apply rules of the form $\langle q', j, h+m-1\rangle \rightarrow \langle q, j, h+m-1\rangle$:
$F_5 \Rightarrow^{h+m-1} y_0\langle\omega, 1, h+m-1\rangle y_1 \dots y_{i-1}x_0\langle\omega, i, h+m-1\rangle x_1 \dots x_{m-1}\langle\omega, i+m-1, h+m-1\rangle x_m y_i \dots y_{h-1}\langle\omega, h+m-1, h+m-1\rangle y_h = F_6$ (final form).

Rename all nonterminals of the form $\langle\omega, t, h+m-1\rangle$ in F_6 to A_t for $1 \leq t < i$, $\langle\omega, t, h+m-1\rangle$ to B_{t-i+1} for $i \leq t \leq i+m-1$, and $\langle\omega, t, h+m-1\rangle$ to A_{t-m+1} for $i+m \leq t \leq h+m-1$. We have obtained $y_0A_1y_1 \dots y_{i-1}x_0B_1x_1 \dots B_mx_my_iA_{i+1} \dots A_hy_h$.

Claim 5.2.2. *If $\langle\sigma\rangle\# \Rightarrow^z \langle\rangle x$ in H, then $S \Rightarrow^* x$ for some $z \geq 0$.*

Proof of Claim 5.2.2. This claim follows from Claim 5.2.1 for $n = 0$.

By Claim 5.2.1 and Claim 5.2.2, we formally proved Lemma 5.2. □

Theorem 5.3. *For every $k \geq 1$, $\mathscr{L}_k(\mathbf{CF\#RS}) = \mathscr{L}_k(\mathbf{P})$.*

Proof of Theorem 5.3. The theorem follows from two previous lemmas, Lemma 5.1 and Lemma 5.2. ■

Finally, we proved the equivalence of CF#RS and programmed grammars with the restriction of the same index.

Corollary 5.4. *For every $k \geq 1$ and every language, L, defined by CF#RS of index k, H, exists CF#RS of index k without erasing rules H' (see Definition 4.3) such that $L = L(H) = L(H')$.*

Proof of Corollary 5.4. The corollary follows from Theorem 3.1.2i in [DP89] that claims $\mathscr{L}_k(\mathbf{P}) = \mathscr{L}_k(\mathbf{P}, CF - \varepsilon)$ and from the equivalence in Theorem 5.3.

Alternatively, a construction proof can be derived from [Kři05a] discussing the removal of erasing rules from programmed grammars with appearance checking. However, the necessary information is recorded in the state of the system instead of the label of a rule.

■

5.1.2 n-Right-Linear #-Rewriting Systems

It is impossible to prove that $\mathscr{L}(mn\text{-}\mathbf{RLIN}\#\mathbf{RS}) = \mathscr{L}(m\text{-}\mathbf{P}n\text{-}\mathbf{G})$, which, in other words, means that every #-rewriting system with $m \cdot n$ components working in right-linear way can be simulated by a m-parallel n-right-linear simple matrix grammar (see Definition 2.24). Since m-parallel n-right-linear simple matrix grammars combine the partially parallel application and the regulated application of rules, the partial parallelism is not enough restrictive mechanism to ensure the proper simulation of the state control of a #-rewriting system. On the other hand, the one-way inclusion $\mathscr{L}(m\text{-}\mathbf{P}n\text{-}\mathbf{G}) \subseteq \mathscr{L}(mn\text{-}\mathbf{RLIN}\#\mathbf{RS})$ holds (see Lemma 5.5). In addition, after the relaxation of the parallel applications of simple matrix rules, we can study the power of 1-parallel n-right-linear simple matrix grammars (see Lemma 5.6).

$$\mathscr{L}(1\text{-}\mathbf{P}n\text{-}\mathbf{G}) = \mathscr{L}(n\text{-}\mathbf{RLIN}\#\mathbf{RS}).$$

In the proofs of two following lemmas, we only describe the construction parts, leaving the rigorous verification of these constructions to the reader (see Convention 5.1).

Lemma 5.5. *For every $m, n \geq 1$, $\mathscr{L}(m\text{-}\mathbf{P}n\text{-}\mathbf{G}) \subseteq \mathscr{L}(mn\text{-}\mathbf{RLIN}\#\mathbf{RS})$.*

Proof of Lemma 5.5. Let $G = (N_{11}, \ldots, N_{mn}, T, S, P)$ be an m-parallel n-right-linear simple matrix grammar and let $M_1, \ldots, M_m$ be mutually disjoint *matrix-rule sets*, where for every $1 \leq i \leq m$, $M_i = \{\mu\colon [X_{i1} \to \alpha_{i1}Y_{i1}, \ldots, X_{in} \to \alpha_{in}Y_{in}] \mid \mu \in P,\ X_{ij}, Y_{ij} \in N_{ij},\ \alpha_{ij} \in T^*,\ 1 \leq j \leq n\} \cup \{\mu\colon [X_{i1} \to \alpha_{i1}, \ldots, X_{in} \to \alpha_{in}] \mid \mu \in P,\ X_{ij} \in N_{ij},\ \alpha_{ij} \in T^*,\ 1 \leq j \leq n\}$ such that $P - \{\sigma\colon [S \to X_{11} \ldots X_{mn}] \mid \sigma \in P,\ X_{ij} \in N_{ij},\ 1 \geq i \geq m,\ 1 \geq j \geq n\} = \bigcup_{1 \leq i \leq m} M_i$.

From G, we construct an equivalent mn-right-linear #-rewriting system, $H = (Q, \Sigma, s, R)$, $\Sigma = T \cup \{\#\}$, $T \cap \{\#\} = \emptyset$, by performing of the following steps:

1. $Q = \{s\} \cup \{\langle \eta, \mu, l\rangle \mid \eta \in \text{suffixes}(X_{11} \ldots X_{mn}),\ X_{ij} \in N_{ij}$ for all $1 \leq i \leq m,\ 1 \leq j \leq n,\ \mu \in M_k,\ 1 \leq k \leq m,\ 1 \leq l \leq n\}$, where s is a new symbol for the starting state;
2. $R =$
   (i) $\{s\,1\# \to \langle X_{11} \ldots X_{mn}, \mu_1, 1\rangle\ \#$
   $\quad \mid \mu_1 \in M_1,\ X_{11} \ldots X_{mn} = \text{rhs}(\sigma),\ \sigma\colon [S \to X_{11} \ldots X_{mn}] \in P\}$
   $\cup$
   (ii) $\{\langle Y_{11} \ldots Y_{ij-1} X_{ij} \ldots X_{mn}, \mu_i, j\rangle\ {}_{(i-1)\cdot n+j}\# \to$
   $\quad \langle Y_{11} \ldots Y_{ij} X_{ij+1} \ldots X_{mn}, \mu_i, j+1\rangle \alpha_{ij}\#$
   $\quad \mid \mu_i\colon [X_{i1} \to \alpha_{i1}Y_{i1}, \ldots, X_{in} \to \alpha_{in}Y_{in}] \in M_i,$
   $\quad\ 1 \leq i \leq m,\ 1 \leq j < n\}$
   $\cup$

$$
\begin{aligned}
\text{(iii)}\ & \{\langle Y_{11}\dots Y_{in-1}X_{in}\dots X_{mn},\mu_i,n\rangle\ {}_{i\cdot n}\#\to \\
& \quad \langle Y_{11}\dots Y_{in}X_{(i+1)1}\dots X_{mn},\mu_{i+1},1\rangle\alpha_{in}\# \\
& \quad \mid 1\le i<m,\ \mu_{i+1}\in M_{i+1},\ \mu_i\colon [X_{i1}\to\alpha_{i1}Y_{i1},\dots,X_{in}\to \\
& \quad\ \alpha_{in}Y_{in}]\in M_i\} \\
\cup & \\
\text{(iv)}\ & \{\langle Y_{11}\dots Y_{mn-1}X_{mn},\mu_m,n\rangle\ {}_{m\cdot n}\#\to\langle Y_{11}\ \dots\ Y_{mn},\mu_1,1\rangle\ \alpha_{mn}\# \\
& \quad \mid \mu_1\in M_1,\ \mu_m\colon [X_{m1}\to\alpha_{m1}Y_{m1},\dots,X_{mn}\to\alpha_{mn}Y_{mn}]\in M_m\} \\
\cup & \\
\text{(v)}\ & \{\langle X_{ij}\dots X_{mn},\mu_i,j\rangle\ {}_1\#\to\langle X_{ij+1}\dots X_{mn},\mu_i,j+1\rangle\ \alpha_{ij} \\
& \quad \mid \mu_i\colon [X_{i1}\to\alpha_{i1},\dots,X_{in}\to\alpha_{in}]\in M_i,\ 1\le i\le m,\ 1\le j<n\} \\
\cup & \\
\text{(vi)}\ & \{\langle X_{in}\dots X_{mn},\mu_i,n\rangle\ {}_1\#\to\langle X_{(i+1)1}\dots X_{mn},\mu_{i+1},1\rangle\ \alpha_{in} \\
& \quad \mid 1\le i<m,\ \mu_{i+1}\in M_{i+1}, \\
& \quad\ \mu_i\colon [X_{i1}\to\alpha_{i1},\dots,X_{in}\to\alpha_{in}]\in M_i\} \\
\cup & \\
\text{(vii)}\ & \{\langle X_{mn},\mu_m,n\rangle\ {}_1\#\to\langle\varepsilon,\mu_m,n\rangle\ \alpha_{mn} \\
& \quad \mid \mu_m\colon [X_{m1}\to\alpha_{m1},\dots,X_{mn}\to\alpha_{mn}]\in M_m\}.
\end{aligned}
$$

Basic Idea of the Proof of Lemma 5.5. H simulates each derivation step in G using the states to hold necessary information about each step. Every state is of the form $\langle\eta,\mu_i,l\rangle$. Instead of parallelism, the rules from G are divided into the sets of matrices, M_i. Each state from Q contains a string of nonterminals η, a matrix label μ_i, and a rule-index indicating next rule to be applied. The last rule in M_i changes the state of H so a matrix from M_{i+1} can be used. Finally, the last rule of a matrix from M_m changes the state of H so it can apply the first rule of a matrix from the very first M_1.

Briefly, the finite-state control of H ensures the atomicity of the sequential simulated parallel and matrix rewrites.

The rules from P of the form $X_{ij}\to\alpha_{ij}Y_{ij}$ change nonterminals and the rules of the form $X_{ij}\to\alpha_{ij}$ remove those nonterminals in the string stored in the first component of the state, η. When there are no nonterminals left in η, the system can make no more steps and the computation ends.

□

Lemma 5.6. *For every $n\ge 1$, $\mathscr{L}(n\text{-}\mathbf{RLIN}\#\mathbf{RS})\subseteq\mathscr{L}(1\text{-}\mathbf{P}n\text{-}\mathbf{G})$.*

Proof of Lemma 5.6. Let $n\ge 1$ be a positive integer and $H=(Q,\Sigma,s,R)$ be an n-right-linear #-rewriting system. We construct an equivalent 1-parallel n-right-linear simple matrix grammar $G=(N_{11},\dots,N_{mn},T,S,P)$ by performing the following steps:

1. $T=\Sigma-\{\#\}$.
2. $N_{1i}=\{\langle i,j,q\rangle\mid q\in Q,\ 1\le j\le i\}\cup\{X_i\}$ for every $1\le i\le n$, where X_i is a new nonterminal.

3. Add $S \to \langle 1,1,s\rangle\langle 2,2,s\rangle \dots \langle n,n,s\rangle$ to P.
4. For every rule $r\colon p\,{}_j\# \to q\ \alpha\# \in R$, $\alpha \in T^*$, add $[\eta_1, \dots, \eta_{i-1}, \langle i,j,p\rangle \to \alpha\langle i,j,q\rangle, \eta_{i+1}, \dots, \eta_n]$ to P, where for every $k \in \{1,\dots,n\} - \{i\}$ and $1 \leq k' \leq k$, η_k is of the form $\langle k,k',p\rangle \to \langle k,k',q\rangle$ or $X_k \to X_k$.
5. For every rule $r\colon p\,{}_j\# \to q\ \alpha \in R$, $\alpha \in T^*$, add $[\eta_1, \dots, \eta_{i-1}, \langle i,j,p\rangle \to \alpha X_i, \eta_{i+1}, \dots, \eta_n]$ to P, where
   for every $1 \leq k < i$ and $1 \leq k' \leq k$, η_k is of form $\langle k,k',p\rangle \to \langle k,k',q\rangle$ or $X_k \to X_k$ and
   for every $i < l \leq n$ and $1 \leq l' \leq n$, η_l is of form $\langle l,l',p\rangle \to \langle l,l'-1,q\rangle$ or $X_l \to X_l$.
6. Add $[X_1 \to \varepsilon, X_2 \to \varepsilon, \dots, X_n \to \varepsilon]$ to P.

Basic Idea of the Proof of Lemma 5.6. G simulates each computational step in H as follows. Every nonterminal has three components. To make the nonterminal alphabets $N_{1i}, \dots, N_{1n}$ mutually disjoint, the first component contains the nonterminal-alphabet-index. The second component represents the position of the corresponding bounder in the current configuration of H. The third component consists of information about the state of H.

In addition, the auxiliary nonterminals $X_1, \dots, X_n$ that do not hold any information about states or #s are introduced. They allow us to have all matrices of the same size n as required by the definition of m-Pn-G (see Definition 2.24).

The rules of R of the form $p\,{}_j\# \to q\ \alpha\#$ change the state-related information inside of all nonterminals except for those of form X_i. The rules of R of the form $p\,{}_j\# \to q\ \alpha$ do the same job apart from rewriting a nonterminal $\langle i,j,p\rangle$ into X_i and reindexing nonterminals following the rewritten one. This simulates removing of a #.

When all nonterminals are of the form X_i, the rule $[X_1 \to \varepsilon,\ X_2 \to \varepsilon,\ \dots,\ X_n \to \varepsilon]$ simultaneously removes all nonterminals from the sentential configuration component to get the string of terminals.

□

Theorem 5.7. *For every $m, n \geq 1$, $\mathscr{L}(m\text{-}\mathbf{P}n\text{-}\mathbf{G}) \subset \mathscr{L}(mn\text{-}\mathbf{RLIN\#RS}) = \mathscr{L}(1\text{-}\mathbf{P}mn\text{-}\mathbf{G})$, where $m + n > 2$.*

Proof of Theorem 5.7. Recall that $\mathscr{L}(mn\text{-}\mathbf{P}1\text{-}\mathbf{G}) \subset \mathscr{L}(m\text{-}\mathbf{P}n\text{-}\mathbf{G}) \subset \mathscr{L}(1\text{-}\mathbf{P}mn\text{-}\mathbf{G})$, for every $m + n > 2$ (see Theorem 10 in [Woo75]). Thus, Theorem 5.7 follows from $\mathscr{L}(m\text{-}\mathbf{P}n\text{-}\mathbf{G}) \subset \mathscr{L}(1\text{-}\mathbf{P}mn\text{-}\mathbf{G})$, Lemma 5.5 and Lemma 5.6. ■

Before we present Theorem 5.8, we give an insight into the implication it contains to make it easier to understand.

To illustrate the denotation of a computational step by ${}_ud_i$ (see Definition 4.5), we write $\overset{{}_ud_i}{\Rightarrow}$.

The implication restricts every successful computation in a #-rewriting system.

Let $d\colon s\#^n = p_0w_0 \overset{{}_{u_1}d_1}{\Rightarrow} p_1w_1 \overset{{}_{u_2}d_2}{\Rightarrow} \cdots \overset{{}_{u_i}d_i}{\Rightarrow} p_iw_i \overset{{}_{u_{i+1}}d_{i+1}}{\Rightarrow} \cdots \overset{{}_{u_j}d_j}{\Rightarrow} p_jw_j \overset{{}_{u_{j+1}}d_{j+1}}{\Rightarrow} \cdots \overset{{}_{u_{|d|}}d_{|d|}}{\Rightarrow} p_{|d|}w_{|d|}$ be a successful computation, where $1 \leq i \leq j \leq |d|$, $u = u_i$, $v = u_j$, and $w_d \in (\Sigma - \{\#\})^*$.

If $u = v$ in ${}_ud_i$ and ${}_vd_j$, then there are allowed only two cases:

(a) all computational steps between ${}_ud_i$ and ${}_vd_j$, denoted by ${}_zd_k$ for all $i \leq k \leq j$, rewrite just zth bounder and nothing else, so $z = u = v$;
(b) there can be only one exception in (a), such that ${}_ld_h$, $i < h < j$, is #-erasing computational step (see Definition 4.5).

Theorem 5.8. *Let every successful computation d in an n-right-linear #-rewriting system H, $n \geq 1$, satisfy this implication: if $1 \leq i \leq j \leq |d|$, $u = v$ in ${}_ud_i$, and ${}_vd_j$, then either $z = u$ in all ${}_zd_k$ for all $i \leq k \leq j$ or d_h is #-erasing for some $h \in \{i+1, \ldots, j-1\}$. Then, $L(H) \in \mathbf{REG}$.*

Proof of Theorem 5.8. Let $H = (Q, \Sigma_H, s, R)$ be a n-right-linear #-rewriting system satisfying the preceding implication, for some $n \geq 1$. We transform H to an equivalent right-linear grammar $G = (\Sigma_G, T, P, S)$ by performing the following procedure:

For every $1 \leq i \leq n$, $p, q \in Q$, construct auxiliary sets ${}^p_qR_i = \{r \mid r \in \mathrm{alph}(\rho), \rho \in R^*, p\gamma\, {}_i\Rightarrow^* q\delta\ [\rho], \mathrm{occur}(\#, \gamma) = \mathrm{occur}(\#, \delta)\}$ and ${}^p_q\bar{R}_i = \{p\,\# \to q\ \alpha \in R \mid \alpha \in (\Sigma_H - \{\#\})^*\}$. Then, $\mathcal{Z} = \bigcup_{i \geq 1, p, q \in Q} \{{}^p_qR_i, {}^p_q\bar{R}_i\}$.

1. $T = \Sigma_H - \{\#\}$,
2. $\Sigma_G = N \cup T \cup \{S\}$, where S is a new symbol and N contains nonterminals introduced by the following construction of P,
3. $P = \bigcup_{1 \leq l \leq 5} P_l$, where sets P_1 through P_5 are constructed in the following way:
   (i) initialization: $P_1 = \{S \to \langle \#^n, i, s\rangle \mid 1 \leq i \leq n\}$;
   (ii) preparation: $P_2 = \{\langle \nabla_1\eta_1\nabla\eta_2 \ldots \nabla_i\eta_i\nabla_{i+1}\eta_{i+1} \ldots \nabla_n\eta_n, i, p\rangle \to \langle \nabla_1\eta_1\nabla\eta_2 \ldots \nabla_i\eta_i\, {}^p_qR_{i'}\nabla_{i+1}\eta_{i+1} \ldots \nabla_n\eta_n, j, q\rangle \mid {}^p_qR_{i'} \neq \emptyset,\ 1 \leq j \leq n,\ \eta_t \in \mathcal{Z}^*,\ \nabla_t \in \{\#, \bar{\#}\} \text{ for } 1 \leq t \leq n,\ \nabla_i \neq \bar{\#},\ i' = \mathrm{occur}(\#, \nabla_1\eta_1 \ldots \nabla_i)\}$
   $\cup$
   $\{\langle \nabla_1\eta_1 \ldots \nabla_{i-1}\eta_{i-1}\nabla_i\eta_i\nabla_{i+1} \ldots \nabla_n\eta_n, i, p\rangle \to \langle \nabla_1\eta_1 \ldots \nabla_{i-1}\eta_{i-1}\bar{\#}\eta_i\, {}^p_q\bar{R}_{i'}\nabla_{i+1}\eta_{i+1} \ldots \nabla_n\eta_n, j, q\rangle$

$$\mid {}_q^p\bar{R}_{i'} \neq \emptyset,\ 1 \leq j \leq n,\ \eta_t \in \mathcal{Z}^*,\ \nabla_t \in \{\#, \bar{\#}\} \text{ for } 1 \leq t \leq n,\ \nabla_i \neq \bar{\#},\ i' = \text{occur}(\#, \nabla_1\eta_1 \ldots \nabla_i)\};$$

(iii) latch:
$P_3 = \{\langle\gamma, i, p\rangle \to \langle\gamma, q\rangle \mid p, q \in Q,\ \# \notin \text{alph}(\gamma),\ A \to \langle\gamma, i, p\rangle \in P_2\}$;

(iv) simulation of a derivation step in G ($\eta_t \in \mathcal{Z}^*$ for every $1 \leq t \leq n$):
$P_4 = \{\langle\bar{\#}\,{}_q^pR_{i'}\eta_i \ldots \bar{\#}\eta_n, p'\rangle \to \alpha\langle\bar{\#}\,{}_q^pR_{i'}\eta_i \ldots \bar{\#}\eta_n, q'\rangle$
$\mid p'\ {}_{i'}\# \to q'\ \alpha\# \in {}_q^pR_{i'},\ \alpha \in (\Sigma_H - \{\#\})^*,\ 1 \leq i \leq n,\ {}_q^pR_{i'} \in \mathcal{Z}\}$
$\cup$
$\{\langle\bar{\#}\,{}_q^pR_{i'}\eta_i \ldots \bar{\#}\eta_n, p'\rangle \to \alpha\langle\bar{\#}\eta_i \ldots \bar{\#}\eta_n, q\rangle$
$\mid p'\ {}_{i'}\# \to q\ \alpha\# \in {}_q^pR_{i'},\ \alpha \in (\Sigma_H - \{\#\})^*,\ 1 \leq i \leq n,\ {}_q^pR_{i'} \in \mathcal{Z}\}$
$\cup$
$\{\langle\bar{\#}\,{}_q^p\bar{R}_{i'}\bar{\#}\eta_{i+1} \ldots \bar{\#}\eta_n, p\rangle \to \alpha\langle\bar{\#}\eta_{i+1}\bar{\#}\eta_n, q'\rangle$
$\mid p\ {}_{i'}\# \to q\ \alpha \in {}_q^p\bar{R}_{i'},\ \alpha \in (\Sigma_H - \{\#\})^*,\ 1 \leq i \leq n,\ {}_q^p\bar{R}_{i'} \in \mathcal{Z}\}$;

(v) finalization:
$P_5 = \{\langle\varepsilon, p\rangle \to \varepsilon \mid p \in Q\}$.

The conversion of a right-linear grammar, G, to an n-right-linear #-rewriting system, H, is simple and left to the reader.

Basic Idea of the Proof of Theorem 5.8. Every ${}_q^pR_i$ represents a set of rules which can make a computation of degree i leading from state p to q in H. In every sentential form generated by G, there is only one occurrence of a nonterminal that is composed of three components:

(1) γ—the finite prescription string for the driven simulation, $\gamma \in (\#\mathcal{Z}^*)^+$;
(2) i—the position of the occurrence of active # in the current configuration of H;
(3) p—the currently simulated state of H.

In the preparation phase, there are non-deterministically generated prescription substrings behind every corresponding bounder in a configuration of H. These substrings η_t are of the form $\mathcal{Z}^*$.

In the third step, the second component of nonterminals is removed to ensure that the rules of P_2 cannot be used anymore.

By rules constructed in step (3iv), the generation of terminals is done with correspondence to the γ-prescription string. Each completed ${}_q^pR_i$ is removed from γ until γ is the empty string. Then, the only nonterminal in the sentential form of G is rewritten to the empty string by a rule from P_5 and a string of terminals is reached.

■

Theorem 5.9. **REG** $\subseteq \mathscr{L}$(1-**RLIN#RS**).

Basic Idea of the Proof of Theorem 5.9. The proof that every regular grammar can be transformed into an equivalent 1-right-linear #-rewriting system is quite simple. The only bounder simulates the corresponding nonterminal used in the regular grammar. In more detail, the correspondence between the particular nonterminal and a bounder is recorded inside the state in the first component of the configuration of the 1-right-linear #-rewriting system (by analogy with the proof of Lemma 5.1).

■

Corollary 5.10. $\mathscr{L}(1\text{-}\mathbf{RLIN\#RS}) = \mathbf{REG}$.

Proof of Corollary 5.10. Note that the right-hand side of every rule in 1-RLIN#RS ends with a bounder and no other bounder occurs on the right-hand side of the rule. Since an 1-RLIN#RS contains at most one bounder in each its configuration and Theorem 5.9 holds, Corollary 5.10 follows from more general theorem, Theorem 5.8 for $n = 1$. ■

To conclude this section, recall that, as compared to the other variants of #-rewriting systems, it makes no sense to consider the finite index restriction of n-right-linear #-rewriting systems because the rules never increase the number of bounders set by the starting configuration. As a matter of fact, an n-right-linear #-rewriting system is of finite index as an implicit restriction (see the classification of restrictions in Chapter 3).

5.1.3 Generalized #-Rewriting Systems

Unlike grammars in the Chomsky hierarchy (context-free and context-sensitive grammars), generalized #-rewriting systems of finite index with context-sensitive rule cores do not provide the increase of power in a comparison with context-free #-rewriting systems of finite index.

As proposed in Convention 5.1, we only describe the construction part of the proof, leaving the complete version of this proof to the reader.

Theorem 5.11. *For every* $k \geq 1$, $\mathscr{L}_k(\mathbf{CF\#RS}) = \mathscr{L}_k(\mathbf{G\#RS})$.

Since a context-free #-rewriting system is only special case of a generalized #-rewriting system, we have to prove only that $\mathscr{L}_k(\mathbf{G\#RS}) \subseteq \mathscr{L}_k(\mathbf{CF\#RS})$.

Proof of Theorem 5.11. Let $k \geq 1$ be a positive integer. Let $H = (Q, T \cup \{\#\}, s, R)$ be a generalized #-rewriting system of index k, where $\Sigma = T \cup \{\#\}$, $\# \notin T$. Let $\mu = \max(\{\max_L(H), \max_R(H)\})$ and let \$ be a new symbol, $\$ \notin Q \cup \Sigma$. We construct an equivalent context-free #-rewriting system of index k, $H' = (Q', T \cup \{\#\}, s', R')$, where components Q', s', and R' are constructed as follows:

1. $s' = \langle s, \# \rangle$;
2. $Q' = \{\langle p, y_0 \# y_1 \# \dots y_{h-1} \# y_h \rangle$
   $\mid p \in Q,\ 1 \leq h \leq k,$
   $y_j = y'_j \nabla_j y''_j,\ y'_j, y''_j \in T^*,\ \nabla_j \in \{\varepsilon, \$\},$
   $|y'_j| \leq 2\mu,\ |y''_j| \leq 2\mu,\ 0 \leq j \leq h\}$;
3. $R' = \{\langle p, x_0 \# \dots \# x'_j \nabla_j x''_j \# \dots \# x_h \rangle\ {}_1\# \to \langle p, x_0 \# \dots \# y'_j \$ y''_j \# \dots \# x_h \rangle\ \#$
   $\mid \langle p, x_0 \# \dots \# x'_j \nabla_j x''_j \# \dots \# x_h \rangle,$
   $\langle p, x_0 \# \dots \# y'_j \$ y''_j \# \dots \# x_h \rangle \in Q',$
   $\nabla_j \in \{\varepsilon, \$\},\ x_j = x'_j \nabla_j x''_j,\ y'_j \in \text{prefixes}(x'_j),\ y''_j \in \text{suffixes}(x''_j),$
   $x'_j, x''_j \in T^*,\ 1 \leq h \leq k,\ 0 \leq j \leq h,$
   where $x'_0 = y'_0 = x''_h = y''_h = \varepsilon\}$;
4. For every rule $r\colon p\ {}_i\alpha\#\beta \to q\ \alpha\gamma\beta \in R$, add the following set to R':
   $\{\langle p, \eta'\alpha\#\beta\eta''\rangle\ {}_i\# \to \langle q, \eta'\alpha\gamma\beta\eta''\rangle\ \gamma$
   $\mid \text{occur}(\#, \eta'\alpha) = i - 1,$
   $\langle p, \eta'\alpha\#\beta\eta''\rangle,\ \langle q, \eta'\alpha\gamma\beta\eta''\rangle \in Q'\}$.

Basic Idea of the Proof of Theorem 5.11. By several computational steps, H' simulates a single step in H. Inside of every state of the form $\langle p, \eta \rangle$ occurring in a configuration of H', it is recorded:

(1) p—the current state of H;
(2) η—the context string of current configuration of H that represents a context of each # of length at most μ on both sides of #.

The simulation is done in two steps by rules introduced in the third and forth step of the above construction as follows:

(a) Each substring of terminals between two #s can be non-deterministically shortened by the rules constructed in 3. The position of the shortening is marked by \$ to reflect this in the subsequent context checks. This shortening is necessary only to make enough space in the context string for the rule application in the next step.
(b) With respect to the surrounding—the left and the right context—of a rewritten bounder, every generalized rule of H replaces only one #. In this step, the rules of the context-free form constructed in 4 are applied and, thereby, simulate behavior of H thanks to the context checks, which are managed by the second component of states in H'.

The simulation is completed when the string of terminals is obtained.

Draft of the rigorous proof:

Let π be a homomorphism from a configuration of a context-free #-rewriting system to the corresponding configuration of a generalized #-rewriting system defined as $\pi(\langle p, \eta \rangle z) = pz$.

Claim 5.11.1. *If* $s\# \Rightarrow^m pz_0\#z_1\#\dots\#z_h$ *in* H, *then* $\langle s,\#\rangle\# \Rightarrow^r \langle p, x_0\#x_1\#\dots\#x_h\rangle\ z_0\#z_1\dots\#z_h\ [r'_1r'_2\dots r'_r]$ *in* H', *where* $x_i = x'_i\nabla_i x''_i$, $\nabla_i \in \{\varepsilon,\$\}$, $x'_i \in \text{prefixes}(z_i)$, $x''_i \in \text{suffixes}(z_i)$, $|x_i| \leq 4\mu+1$, x'_i, x''_i, $z_i \in T^*$, $0 \leq i \leq h$, $1 \leq h \leq k$, *for* $m \geq 0$.

Proof of Claim 5.11.1. This claim is established by induction on m.

Induction Basis: Let $m = 0$. For $s\# \Rightarrow^0 s\#$ in H there exists $s'\# \Rightarrow^0 s'\#$ in H', where $s' = \langle s,\#\rangle$.

Induction Hypothesis: Suppose that Claim 5.11.1 holds for all computations of length m or less for some $m \geq 0$.

Induction Step: Consider $s\# \Rightarrow^{m+1} qz'$, where $z' \in \Sigma^*$. Express $s\# \Rightarrow^{m+1} qz'$ as $s\# \Rightarrow^m pz\ [r_1r_2\dots r_m]$, where $z = z_0\#z_1\#\dots\alpha\#\beta\dots\#z_h$, $\#$ between α and β is the ith bounder and $r_1,\dots,r_m,r_{m+1} \in \text{Lab}(R)$, and $pz \Rightarrow qz'\ [r_{m+1}]$. For $r_{m+1}\colon p\ {}_i\alpha\#\beta \to q\ \alpha\gamma\beta$ is z' of the form: $z' = z_0\#z_1\#\dots\alpha\gamma\beta\dots\#z_h$, for $z_0,\dots,z_h \in T^*$.

Based on the induction hypothesis, there exists $\langle s,\#\rangle\# \Rightarrow^r \langle p, x_0\#x_1\#\dots\alpha\#\beta\dots\#x_h\rangle\ z_0\#\dots\alpha\#\beta\dots z_{h-1}\#z_h\ [r'_1r'_2\dots r'_r] \Rightarrow^* \langle p, y_0\#y_1\#\dots\alpha\#\beta\dots\#y_h\rangle\ z_0\#\dots\alpha\#\beta\dots z_{h-1}\#z_h\ [\rho] \Rightarrow \langle q, y_0\#y_1\#\dots\alpha\gamma\beta\dots\#y_h\rangle\ z_0\#\dots\alpha\gamma\beta\dots z_{h-1}\#z_h\ [r'_{r+1}]$, where ρ is a sequence of rules constructed in step 3, $|\rho| \geq 0$, $r \geq 1$, $r'_j \in \text{Lab}(R')$, $1 \leq j \leq r+1$, x_j, $y_j \in T^*\{\varepsilon,\$\}T^*$, $z_j \in T^*$, $0 \leq j \leq h$, $\text{occur}(\#, y_0\#y_1\#\dots\alpha) = \text{occur}(\#, z_0\#z_1\#\dots\alpha) = i-1$, $\alpha = \bar{x}_{i-\bar{\alpha}}\#x_{i-\bar{\alpha}+1}\dots\ x_{i-2}\#x_{i-1}$, $\bar{\alpha} = \text{occur}(\#,\alpha)+1$, $\bar{x}_{i-\bar{\alpha}} \in \text{suffixes}(x_{i-\bar{\alpha}})$, $\beta = x_i\#x_{i+1}\dots x_{i+\bar{\beta}-1}\ \#\bar{x}_{i+\bar{\beta}}$, $\bar{\beta} = \text{occur}(\#,\beta)$, $\bar{x}_{i+\bar{\beta}} \in \text{prefixes}(x_{i+\bar{\beta}})$, and $r'_{r+1}\colon \langle p,\ y_0\#y_1\#\dots\alpha\#\beta\dots\ \#y_h\rangle\ \# \to \langle q, y_0\#y_1\#\dots\alpha\gamma\beta\dots\#x_h\rangle\ \gamma \in R'$ is introduced by step 4.

Therefore, $\pi(\langle p, y_0\#y_1\#\dots\alpha\#\beta\dots\#y_h\rangle\ z_0\#\dots\alpha\#\beta\dots z_{h-1}\#z_h) = pz$ and $\pi(\langle q, y_0\#y_1\#\dots\alpha\gamma\beta\dots\#y_h\rangle\ z_0\#\dots\alpha\gamma\beta\dots z_{h-1}\#z_h) = qz'$, so the claim holds.

Claim 5.11.2. *If* $s\# \Rightarrow^z pw$ *in* H, *then* $\langle s,\#\rangle\# \Rightarrow^* \langle p,\eta\rangle w$ *in* H' *for some* $z \geq 0$, $w \in T^*$.

Proof of Claim 5.11.2. Consider Claim 5.11.1 for $h = 0$. At this point, if $s\# \Rightarrow^z pz_0$ in H, then $\langle s,\#\rangle\# \Rightarrow^* \langle p, x_0\rangle z_0$ in H' and so $z_0 = w$.

Claim 5.11.1 and Claim 5.11.2 sketch the formal proof of Theorem 5.11 including its construction part. ■

Corollary 5.12. *For every* $k \geq 1$ *and for every language* L *defined by* $G\#RS$ *of index* k, H, *there exists* $G\#RS$ *of index* k *without erasing rules (see Definition 4.3),* H', *such that* $L = L(H) = L(H')$.

Proof of Corollary 5.12. This corollary trivially follows from Theorem 5.11 and Corollary 5.4. ■

Unrestricted #-Rewriting Systems

We have discussed the generalized language-defining device that represents a combination of a finite-state control, a context-sensitive core rules, and a few additional conditions how its rules can be applied. Now, let us investigate even more general form of rules inspired by unrestricted grammars (see Definition 2.13), in which there is no difference between nonterminal and terminal symbols as to rewriting. Consider a new form of rules where the left-hand side of a rule is the same as in generalized #-rewriting systems but the right-hand side of a rule rewrites whole left-hand side, not only # in the left and the right context. In other words, we do not replace the bounder with particular left and right context, but in a configuration, we rewrite the entire left-hand side of the rule by the right-hand side, so more than one bounder can be replaced at once.

Definition 5.1. An *unrestricted #-rewriting system* (U#RS), $H = (Q, \Sigma, s, R)$, is a generalized #-rewriting system with rules of the following form:

$$p\ {}_i\alpha \to q\ \beta,$$

where $p, q \in Q$, $i \in \mathbb{N}$, $\alpha, \beta \in \Sigma^*$, $\text{occur}(\#, \alpha) \geq 1$ and a computational step is defined as follows:

Let $pu\alpha'\#\alpha''v$, $qu\beta v$ be two configurations, $p, q \in Q$, $u, v \in \Sigma^*$, $i \in \mathbb{N}$ and $occur(\#, u\alpha') = i - 1$. Then, an unrestricted #-rewriting system makes a *computational step* from $pu\alpha'\#\alpha''v$ to $qu\beta v$ by using the unrestricted rule $r\colon p\ {}_i\alpha'\#\alpha'' \to q\ \beta$.

Let us illustrate the run of an unrestricted #-rewriting system on the following example.

Example 5.1. In [DP89], Theorem 3.1.7 proves that, for every $k \geq 2$, $L_k \in \mathscr{L}_k(\mathbf{CF\#RS})$ and $L_k \notin \mathscr{L}_{k-1}(\mathbf{CF\#RS})$, where $L_k = \{b(a^i b)^{2k} \mid i \geq 1\}$.

Consider $U\#RS$ $H = (\{s, s', p, q, r, r', f\}, \{a, b, \#\}, s, R)$, where R contains:

1: $s\ {}_1\# \to s'\ \#\#a\#\#$
2: $s'\ {}_2\# \to s'\ \#a$
3: $s'\ {}_1\# \to p\ \#$
4: $p\ {}_2\#a \to q\ a\#$
5: $q\ {}_3\# \to p\ \#a$
6: $p\ {}_2\#\# \to r\ \#\#$
7: $r\ {}_1\# \to r'\ b$

8: $r'\ {}_3\# \rightarrow p\ \#\#$
9: $p\ {}_1\#\# \rightarrow f\ b$
10: $f\ {}_1\#\# \rightarrow f\ b$.

For instance, H computes $baabaabaab = b(a^2b)^3$ as follows:

$s\# \Rightarrow s'\#\#a\#\#$ [1] $\Rightarrow s'\#\#aa\#\#$ [2] $\Rightarrow p\#\#aa\#\#$ [3]
$\Rightarrow q\#a\#a\#\#$ [4] $\Rightarrow p\#a\#a\#a\#$ [5]
$\Rightarrow q\#aa\#\#a\#$ [4] $\Rightarrow p\#aa\#\#aa\#$ [5]
$\Rightarrow r\#aa\#\#aa\#$ [6] $\Rightarrow r'baa\#\#aa\#$ [7] $\Rightarrow pbaa\#\#aa\#\#$ [8]
$\Rightarrow qbaa\#a\#a\#\#$ [4] $\Rightarrow pbaa\#a\#a\#a\#$ [5]
$\Rightarrow qbaa\#aa\#\#a\#$ [4] $\Rightarrow pbaa\#aa\#\#aa\#$ [5]
$\Rightarrow rbaa\#aa\#\#aa\#$ [6] $\Rightarrow r'baabaa\#\#aa\#$ [7] $\Rightarrow pbaabaa\#\#aa\#\#$ [8]
$\Rightarrow fbaabaabaa\#\#$ [9] $\Rightarrow fbaabaabaab$ [10].

It is obvious that H is of index 4 and $L(H) = \{b(a^ib)^j \mid i, j \geq 1\} \in \mathscr{L}_4(\mathbf{U}\#\mathbf{RS})$. Thus, this example shows that for every $k \geq 1$, there is a language $L = L_n$, $n > k$, such that $L \notin \mathscr{L}_k(\mathbf{CF}\#\mathbf{RS})$ and L is generated by an unrestricted #-rewriting system of index 4.

Observation 5.1. Observe that for unrestricted #-rewriting systems, there exists no analogical theorem to Theorem 5.21. When we only limit the number of bounders in configurations, the finite index restriction makes no difference to the power of this system. On the other hand, the workspace restriction (see Definition 3.3) is more interesting for these systems to achieve the decrease of their power.

Hypothesis 5.1. $\mathscr{L}(\mathbf{U}\#\mathbf{RS}) = \mathbf{RE}$.

The hypothesis seems to hold already for index 1, because unrestricted rules allow the system to move a bounder through the sentential configuration component; for instance, by rules of the form $p\ {}_1\#a \rightarrow p\ a\#$ and $p\ {}_1a\# \rightarrow p\ \#a$. As the result, we can simulate an unrestricted grammar (see Definition 2.13) and probably, in addition, even by the use of a limited number of states. At the end, we need to solve how to check that the sentence contains no #.

As a result from Observation 5.1, we believe that there is no infinite hierarchy based on finite index in case of unrestricted #-rewriting systems. More specifically, every unrestricted #-rewriting system can be transformed into an equivalent unrestricted #-rewriting system of index 1. However, we have not proved this conjecture so far.

Open problem 5.2. Let us have a #-rewriting system (CF#RS, G#RS, or n-RLIN#RS) without the finite index restriction of their configurations. What families of languages do they define? What is the relationship between them?

We established the generative power of several types of #-rewriting systems of finite index. To encourage the future research, the power of #-rewriting systems without the finite index restriction is still the open question.

5.1.4 Restricted Pushdown Automata

First, we illustrate the difference between controlled grammars regulated by a regular language and restricted pushdown automata. More specifically, we discuss the power of restricted pushdown automata and its dependency on the family of the restricting language (see Theorem 5.13). Second, we demonstrate that a linear restricting language increases the accepting power of RPDAs so they can process even non-context-free languages (see Theorem 5.14).

For every RPDA $H = (M, L)$ with regulating language $L \in \mathbf{REG}$ (see Definition 4.12) we can construct an equivalent pushdown automaton M (see Definition 2.19 and Theorem 2.2).

Theorem 5.13. $\mathscr{L}(\mathbf{RPDA}, \mathbf{REG}) = \mathbf{CF}$.

Proof of Theorem 5.13. Let $K = (Q_K, \Sigma_K, R_K, s_K, F_K)$ be a finite automaton such that $L_K = L(K)$, and let $H = (M, L_K)$ be a restricted pushdown automaton, where $M = (Q_M, \Gamma_M, \Sigma_M, R_M, s_M, S_M, F_M)$ is a pushdown automaton. Now, we construct an equivalent classical pushdown automaton with reversed pushdown-string $N = (Q, \Gamma, \Sigma, R, s, S, F)$ such that $L(H) = L(N)$:

1. $Q = Q_M \cup \{s\}$, where $s \notin Q_M$ is a new state;
2. $\Gamma = \Gamma_M \cup Q_K \cup \{S\}$, where $S \notin \Gamma_M \cup Q_K$ is a new pushdown symbol;
3. $\Sigma = \Sigma_M$;
4. $F = F_M$;
5. $R = \{Ss \to s_K S_M s_M\} \cup$
   $\{p_1 B_1 p_2 B_2 \dots p_m B_m p_{m+1} o_M a \to q_1 A_1 q_2 A_2 \dots q_n A_n q_{n+1} r_M \mid$
   $p_1, p_2, \dots, p_m \in Q_K,\ p_{m+1} \in F_K,\ m \geq 0,$
   $B_1, B_2, \dots, B_m \in \Gamma_M,\ A_1, A_2, \dots, A_m \in \Gamma_M \cap \Sigma_K,$
   $q_1 A_1 \to q_2,\ q_2 A_2 \to q_3,\ \dots,\ q_n A_n \to q_{n+1} \in R_K,\ q_{n+1} \in F_K,$
   $n \geq 0,\ B_1 B_2 \dots B_m o_M a \to A_1 A_2 \dots A_n r_M \in R_M,$
   $o_M, r_M \in Q_M,\ a \in \Sigma_M\}$.

A rigorous proof that demonstrates the equivalence between languages defined by these two systems is left to the reader. A proof will be based on mathematical induction on the sequence of moves from the starting configuration to the final configuration in both rewriting systems.

Basic Idea of the Proof of Theorem 5.13. The transformation of a classical pushdown automaton to an equivalent restricted pushdown automaton follows from the definition. Specifically, every pushdown automaton $M = (Q, \Sigma, \Gamma, R, s, S, F)$ is also a restricted pushdown automaton with restricting language $\Xi = \Gamma^*$ that is regular.

A proof of the inclusion in another direction is made by the previous construction. The first statement stands for the initial phase. Other statements produce rules that are interleaved by auxiliary symbols that represent states from Q_K.

The pushdown-string of M is of the form $(Q_K\Gamma_M)^*$. By interleaving pushdown symbols from Γ_M by symbols from Q_K, we record states, which K has to visit during the processing of the pushdowns-string of M. The interleaving is executed by the transformed rules of the form $p_1B_1p_2B_2 \dots p_mB_mp_{m+1}o_Ma \to q_1A_1q_2A_2 \dots q_nA_nq_{n+1}r_M$. This mechanism guarantees that multisubstrings $B_1B_2 \dots B_m$ and $A_1A_2 \dots A_n$ form suffixes of the restricting language sentence that corresponds to the current pushdown-string of M.

To check the validity of the pushdown automaton configuration restricted by the restricting automaton K, we require that the top symbol of the pushdown of M is some $q_{n+1} \in F_K$.

■

Theorem 5.14. $\mathscr{L}(\mathbf{RPDA}, \mathbf{REG}) \subset \mathscr{L}(\mathbf{RPDA}, \mathbf{LIN})$.

Proof of Theorem 5.14. Example 4.7 describes an instance of a restricted pushdown automaton restricted by a linear restricting language. The automaton accepts language $L = \{a^nb^nc^n \mid n \geq 0\}$. By the pumping lemma for context-free languages (see page 512 in [Med00]), $L \notin \mathbf{CF}$. Since every regular restricting language is linear as well, the proper inclusion in Theorem 5.14 holds. ■

Corollary 5.15. $\mathbf{CF} = \mathscr{L}(\mathbf{RPDA}, \mathbf{REG}) \subset \mathscr{L}(\mathbf{RPDA}, \mathbf{LIN}) \subseteq \mathbf{RE}$

Proof of Corollary 5.15. The corollary follows from Theorem 5.13 and Theorem 5.14. ■

As inspired by restricted pushdown automata, we suggest studying a similar restriction of regulated grammars, which will probably lead to the algebraic restrictions based on the operation intersection and substitution over languages.

5.1.5 Reducing Deep Pushdown Automata

Now, we demonstrate the power and the resulting infinite hierarchy of language families defined by reducing deep pushdown automata of depth n that process input string from the right to the left (see [KMS06b, KS06a]); that is, for every $n \geq 1$, ${}_n\mathscr{L}(\mathbf{^{rl}RDPDA}) \subset {}_{n+1}\mathscr{L}(\mathbf{^{rl}RDPDA}) \subset \mathbf{CS}$. The infinite hierarchy equals to the existing infinite hierarchy of state grammars restricted by n-limitation (proved by Kasai in 1970; see [Kas70] or Theorem 2.5). Accordingly, we prove only the equivalence of ${}_n\mathscr{L}(\mathbf{ST})$ and ${}_n\mathscr{L}(\mathbf{^{rl}RDPDA})$ for every $n \geq 1$.

Lemma 5.16. *For every $n \geq 1$, ${}_n\mathscr{L}(\mathbf{ST}) \subseteq {}_n\mathscr{L}(\mathbf{^{rl}RDPDA})$.*

Proof of Lemma 5.16. Let $G = (V, W, T, P, S)$ be an n-limited state grammar and $n \geq 1$. Set $N = V - T$. Define the homomorphism f over $(\{\#\} \cup V)^*$ as $f(A) = A$ for every $A \in \{\#\} \cup N$, and $f(a) = \varepsilon$ for every $a \in T$. Introduce the ^{rl}RDPDA of depth n,

$$ {}_nM = (Q, T, \{\#\} \cup V, R, s, S, \{\$\}), $$

where $Q = \{s, \$\} \cup \{\langle p, u\rangle \mid p \in W,\ u \in \text{prefix}(N^*\{\#\}^n, n),\ |u| \leq n\}$ and R is constructed by performing the following steps:

1. for every $(p, A) \to (q, x) \in P$, $p, q \in W$, $A \in N$, $x \in T^+$,
   add $sx \vdash 1\langle p, A\#^{n-1}\rangle A$ to R;
2. if $(p, A) \to (q, x) \in P$, $\langle q, \text{prefix}(uf(x)v, n)\rangle \in Q$, $p, q \in W$, $A \in N$,
   $x \in V^+$, $u \in N^*$, $v \in N^*\{\#\}^*$, $|uAv| = n$, $p \notin {}_Gstates(u)$, then
   add $\langle q, \text{prefix}(uf(x)v, n)\rangle x \vdash |uA|\langle p, uAv\rangle A$ to R;
3. for every $(p, S) \to (q, x) \in P$, $p, q \in W$, $x \in V^+$,
   add rule $\langle q, \text{prefix}(f(x)\#^n, n)\rangle x \vdash 1\$S$ to R;
4. if $A \in N$, $p \in W$, $u \in N^*$, $v \in \{\#\}^*$, $|uv| \leq n - 1$, $p \notin {}_Gstates(u)$, then
   add rule $\langle p, uv\rangle A \vdash |uA|\langle p, u\#v\rangle A$ and $\langle p, uv\rangle A \vdash |uA|\langle uAv\rangle A$ to R.

Basic Idea of the Proof of Lemma 5.16. Every n-limited derivation step in G is simulated by reversal reduction step in ${}_nM$. So, if some nonterminal (ith from the left) is rewritten by string in G, then exactly the same string on ${}_nM$'s pushdown is replaced by the same non-input symbol in the depth of i, $1 \geq i \geq n$. States of ${}_nM$ are composed of two components:

a) original state of G and
b) string of length n which remembers first n nonterminals in the current sentential form (completed by # symbols from behind if needed).

Unlike state grammars, RDPDAs specify final states, so the first step creates initialization rules. The second step keeps the second component of the current state consistent with the content of the pushdown. Next, the rules from

the third construction step simulate the first derivation step in G (from the starting configuration (p, S)) by emptying pushdown of ${}_nM$. The last, fourth, step fills in the second state component of ${}_nM$, so this component reflects the content of the pushdown. More specifically, the second state component contains n topmost symbols from N in the current pushdown-string.

When G successfully completes the generation of a string of terminals, ${}_nM$ completes by entering the final state \$ and with empty pushdown.

Rigorous Proof: The formal proof that $L(G, n) = L({}_nM)$ follows next. For every $n \geq 1$ and every right-to-left reducing deep pushdown automata of depth n, ${}_nM$, there exists a state grammar, G, such that $L(G, n) = L({}_nM)$.

Claim 5.16.1. *Let* $(p, S)_n \Rightarrow^m (q, dy)$ *in* G, *where* $d \in T^*$, $y \in (NT^*)^*$, $p, g \in W$, $m \geq 0$. *Then* $(\langle q, \text{prefix}(f(y\#^n), n)\rangle, d, y\#) \Rightarrow^* (\langle p, S\#^{n-1}\rangle, \varepsilon, S\#)$ *in* ${}_nM$.

Proof of Claim 5.16.1 (by induction on $m = 0, 1, \ldots$)**.**
Induction Basis: Let $m = 0$, so $(p, S)_n \Rightarrow^0 (p, S)$ in G, $d = \varepsilon$ and $y = S$. Particularly, $(\langle p, S\#^{n-1}\rangle, \varepsilon, S\#) \Rightarrow^* (\langle p, S\#^{n-1}\rangle, \varepsilon, S\#)$ in ${}_nM$, so the induction basis holds.

Induction Hypothesis: Assume that the Claim 5.16.1 holds for all $0 \leq m \leq k$, where $k \in \mathbb{N}_0$ is a non-negative integer.

Induction Step: Let $(p, S)_n \Rightarrow^{k+1} (q, dy)$ in G, where $d \in T^*$, $y \in (NT^*)^*$, $p, q \in W$. Because $k+1 \geq 1$, express $(p, S)_n \Rightarrow^{k+1} (q, dy)$ as $(p, S)_n \Rightarrow^k (h, buAo)_n \Rightarrow (q, buxo)$ $[(h, A) \to (q, x)]$, where $b \in T^*$, $u \in (NT^*)^*$, $A \in N$, $h, q \in W$, $(h, A) \to (q, x) \in P$, $\text{maxsuffix}(buxo, (NT^*)^*) = y$ and $\text{maxprefix}(buxo, T^*) = d$. By the induction hypothesis, $(\langle h, \text{prefix}(f(uAo\#^n), n)\rangle, w, uAo\#) \Rightarrow^* (\langle p, S\#^{n-1}\rangle, \varepsilon, S\#)$ in ${}_nM$, where $w = \text{maxprefix}(buAo, T^*)$. As $(p, A) \to (q, x) \in P$, step 2 of the construction introduces rule $\langle q, \text{prefix}(f(uxo\#^n), n)\rangle x \vdash |uA|\langle h, \text{prefix}(f(uAo\#^n), n)\rangle A$. By using this rule, ${}_nM$ simulates $(h, buAo)_n \Rightarrow (q, buxo)$ by making $(\langle q, \text{prefix}(f(uxo\#^n), n)\rangle, \varepsilon, uxo\#) \Rightarrow (\langle h, \text{prefix}(f(uAo\#^n), n)\rangle, \varepsilon, uAo\#)$ in ${}_nM$. If $uxo \in (NT^*)^*$, $uxo = y$ and the induction step is completed. Assume that $uxo \neq y$, so $uxo = ty$ and $d = wt$ for some $t \in T^+$. Observe $\text{prefix}(f(uxo\#^n), n) = \text{prefix}(f(y\#^n), n)$ at this point. Then, ${}_nM$ adds t by making $|t|$ shifting moves so that $(\langle q, \text{prefix}(f(y\#^n), n)\rangle, t, y\#) \Rightarrow^{|t|} (\langle q, \text{prefix}(f(uxo\#^n), n)\rangle, w, ty\#) \Rightarrow^* (\langle p, S\#^{n-1}\rangle, \varepsilon, S\#)$, which completes the induction step.

By the previous claim for $y = \varepsilon$, if $(p, S)_n \Rightarrow^* (q, d)$ in G, where $d \in T^*$, $p, q \in W$, then $(\langle q, \#^n\rangle, d, \#) \Rightarrow^* (\langle p, S\#^{n-1}\rangle, \varepsilon, S\#)$ in ${}_nM$. As R contains rules introduced in 1 and 3, we also have $(s, d, \#) \Rightarrow^* (\langle p, S\#^{n-1}\rangle, \varepsilon, S\#) \Rightarrow (\$, \varepsilon, S\#)$ in ${}_nM$. Thus, $d \in L(G)$ implies $d \in L({}_nM)$, so $L(G, n) \subseteq L({}_nM)$.

Claim 5.16.2. *Let* $(\langle q, \text{prefix}(f(y\#^n), n)\rangle, c, by\#) \Rightarrow^m (\langle p, S\#^{n-1}\rangle, \varepsilon, S\#)$ *in* ${}_nM$*, where* $c, b \in T^*$*,* $y \in (NT^*)^*$*,* $p, q \in W$*,* $m \geq 0$*. Then,* $(p, S)_n \Rightarrow^* (q, cby)$ *in* G*.*

Proof of Claim 5.16.2 (by induction on $m = 0, 1, \ldots$)**.**
Induction Basis: Let $m = 0$. Then, $c = b = \varepsilon$, $y = S$, and $(\langle q, \text{prefix}(f(S\#^n), n)\rangle, \varepsilon, S\#) \Rightarrow^0 (\langle p, S\#^{n-1}\rangle, \varepsilon, S\#)$ in ${}_nM$. As $(p, S)_n \Rightarrow^0 (p, S)$ in G, the basis holds.

Induction Hypothesis: Assume that Claim 5.16.2 holds for all $0 \leq m \leq k$, where $k \in \mathbb{N}_0$ is a non-negative integer.

Induction Step: Let $(\langle q, \text{prefix}(f(y\#^n), n)\rangle, c, by\#) \Rightarrow^{k+1} (\langle p, S\#^{n-1}\rangle, \varepsilon, S\#)$ in ${}_nM$, where $c, b \in T^*$, $y \in (NT^*)^*$, $p, q \in W$ in ${}_nM$. Because $k + 1 \geq 1$, we can express $(\langle q, \text{prefix}(f(y\#^n), n)\rangle, c, by\#) \Rightarrow^{k+1} (\langle p, S\#^{n-1}\rangle, \varepsilon, S\#)$ as $(\langle q, \text{prefix}(f(y\#^n), n)\rangle, c, by\#) \Rightarrow \alpha \Rightarrow^k (\langle p, S\#^{n-1}\rangle, \varepsilon, S\#)$ in ${}_nM$, where α is a configuration of ${}_nM$ whose form depends on the last move:

(A) Assume that $(\langle q, \text{prefix}(f(y\#^n), n)\rangle, c, by\#)_s \Rightarrow \alpha$ in ${}_nM$. In greater detail, let $\alpha = (\langle q, \text{prefix}(f(y\#^n), n)\rangle, \text{prefix}(c, |c| - 1), aby\#)$ with $a \in T$ such that $c = \text{prefix}(c, |c| - 1)a$. Thus, $(\langle q, \text{prefix}(f(y\#^n), n)\rangle, c, by\#)_s \Rightarrow (\langle q, \text{prefix}(f(y\#^n), n)\rangle, \text{prefix}(c, |c| - 1), aby\#) \Rightarrow^k (\langle p, S\#^{n-1}\rangle, \varepsilon, S\#)$. Since $(\langle q, \text{prefix}(f(y\#^n), n)\rangle, \text{prefix}(c, |c|-1), aby\#) \Rightarrow^k (\langle p, S\#^{n-1}\rangle, \varepsilon, S\#)$, we have $(\langle q, \text{prefix}(f(y\#^n), n)\rangle, \text{prefix}(c, |c|-1), by\#) \Rightarrow^k (\langle p, S\#^{n-1}\rangle, \varepsilon, S\#)$. By the induction, $(p, S)_n \Rightarrow^* (q, \text{prefix}(c, |c| - 1)aby)$ in G. As $c = \text{prefix}(c, |c| - 1)a$, $(p, S)_n \Rightarrow^* (q, cby)$ in G.

(B) On the other hand, assume that $(\langle q, \text{prefix}(f(y\#^n), n)\rangle, \varepsilon, by\#)_r \Rightarrow \alpha$ in ${}_nM$. In greater detail, suppose that $\alpha = (\langle o, \text{prefix}(f(uAv\#^n), n)\rangle, \varepsilon, uAv\#)$ and $(\langle q, \text{prefix}(f(uxv\#^n), n)\rangle, \varepsilon, uxv\#)_r \Rightarrow (\langle o, \text{prefix}(f(uAv\#^n), n)\rangle, \varepsilon, uAv\#)$ by using rule r_1: $\langle q, \text{prefix}(f(uxv\#^n), n)\rangle x \vdash |f(uA)|\langle o, \text{prefix}(f(uAv\#^n), n)\rangle A \in R$ introduced in step 2 of the construction, where $A \in N$, $u \in (NT^*)^*$, $v \in (N \cup T)^*$, $o \in W$, $|f(uA)| \leq n$, and $by\# = uxv\#$. By the introduction hypothesis, $(\langle o, \text{prefix}(f(uAv\#^n), n)\rangle, c, uAv\#) \Rightarrow^k (\langle p, S\#^{n-1}\rangle, \varepsilon, S\#)$ implies $(p, S)_n \Rightarrow^* (o, cuAv)$ in G. As $r_1 \in R$, $(o, A) \to (q, x) \in P$ and $A \notin {}_G states(f(u))$. Thus, $(p, S)_n \Rightarrow^* (o, cuAv)_n \Rightarrow (q, cuxv)$ in G. Thus, $(p, S)_n \Rightarrow^* (q, cby)$ in G because $by\# = uxv\#$.

Consider the previous claim, Claim 5.16.1, for $b = y = \varepsilon$ to see that $(\langle q, \text{prefix}(f(\#^n), n)\rangle, c, \#) \Rightarrow^* (\langle p, S\#^{n-1}\rangle, \varepsilon, S\#)$ in ${}_nM$ implies $(p, S)_n \Rightarrow^* (q, c)$ in G. Let $c \in L({}_nM)$. Then, $(s, c, \#) \Rightarrow^* (\$, \varepsilon, S\#)$ in ${}_nM$. Examine the construction of ${}_nM$ to see that $(s, c, \#) \Rightarrow^* (\$, \varepsilon, S\#)$ starts by using some shifting moves and then a rule introduced in 1, so $(s, c, \#)_s \Rightarrow^* (s, \alpha, \beta\#) \Rightarrow (\langle p, A\#^{n-1}\rangle, \alpha, uAv\#)$. Furthermore, notice that this sequence of moves ends

as $(s, c, \#) \Rightarrow^* (\$, \varepsilon, S\#)$ by using a rule introduced in step 3. Thus, we can express $(s, c, \#) \Rightarrow^* (\$, \varepsilon, S\#)$ as $(s,\ c,\ \#)_s \Rightarrow^* (s,\ \alpha,\ \beta\#) \Rightarrow (\langle p,\ A\#^{n-1}\rangle,\ \alpha,\ uAv\#) \Rightarrow^* (\langle q, \mathrm{prefix}(f(x)\#^n), n)\rangle, \varepsilon, x\#) \Rightarrow (\$, \varepsilon, S\#)$ in ${}_nM$. Thus, $c \in L({}_nM)$ implies $c \in L(G, n)$, so $L({}_nM) \subseteq L(G, n)$.

As $L({}_nM) \subseteq L(G, n)$ and $L(G, n) \subseteq L({}_nM)$, $L(G, n) = L({}_nM)$. Thus, Lemma 5.16 holds.

□

Lemma 5.17. *For every $n \geq 1$, ${}_n\mathscr{L}(\mathbf{^{rl}RDPDA}) \subseteq {}_n\mathscr{L}(\mathbf{ST})$.*

Proof of Lemma 5.17. Let $n \geq 1$ and ${}_nM = (Q, T, V, R, s, S, F)$ be a right-to-left reducing deep pushdown automaton (^{rl}RDPDA) of depth n. Let Z and \$ be two new symbols that occur in no component of ${}_nM$. Set $N = V - T$. Introduce sets $C = \{\langle q, i, \triangleright\rangle |\ q \in Q,\ 1 \leq i \leq n-1\}$, $D = \{\langle q, i, \triangleleft\rangle |\ q \in Q,\ 0 \leq i \leq n-1\}$, an alphabet W such that $\mathrm{card}(V) = \mathrm{card}(W)$, and for all $1 \leq i \leq n$, an alphabet U_i such that $\mathrm{card}(U_i) = \mathrm{card}(N)$. Without any loss of generality, assume that V, Q, and all these newly introduced sets and alphabets are pairwise disjoint. Set $U = \bigcup_{i=1}^{n} U_i$. Introduce a bijection h from V to W. For each $1 \leq i \leq n$, introduce a bijection ${}_ig$ from N to U_i. Define an equivalent state grammar

$$G = (V \cup W \cup U \cup \{Z\}, Q \cup C \cup D \cup \{\$, z\}, T, P, Z),$$

where P is constructed by performing the following steps:

1. add $(z, Z) \to (\langle z, 1, \triangleright\rangle, h(S))$ to P;
2. for every $q \in Q$, $A \in N$, $1 \leq i \leq n-1$,
   add $(\langle q, i, \triangleright\rangle, A) \to (\langle q, i+1, \triangleright\rangle, {}_ig(A))$ and $(\langle q, i, \triangleleft\rangle, {}_ig(A)) \to (\langle q, i-1, \triangleleft\rangle, A)$ to P;
3. if $qxY \vdash ipA \in R$, for some $p, q \in Q$, $A \in N$, $x \in V^*$, $Y \in V$, $i = 1, \ldots, n$, then add $(\langle p, i, \triangleright\rangle, A) \to (\langle q, i-1, \triangleleft\rangle, xY)$ and
   $(\langle p, i, \triangleright\rangle, h(A)) \to (\langle q, i-1, \triangleleft\rangle, xh(Y))$ to P;
4. for every $q \in Q$, $A \in N$, add $(\langle q, 0, \triangleleft\rangle, A) \to (\langle q, 1, \triangleright\rangle, A)$ and
   $(\langle q, 0, \triangleleft\rangle, h(Y)) \to (\langle q, 1, \triangleright\rangle, h(Y))$ to P;
5. for every $q \in F$, $a \in T$, add $(\langle q, 0, \triangleleft\rangle, h(a)) \to (\$, a)$ to P.

Basic Idea of the Proof of Lemma 5.17. G simulates reversal effect of the application of the rule $qx \vdash ipA \in R$. G scans (left-to-right) the sentential configuration component, counts the occurrences of nonterminals until it reaches the ith occurrence of a nonterminal. If it is A, G replaces it with x which corresponds to reducing x to A by ${}_nM$. G completes the simulation of the reduction of a string x by ${}_nM$ so it marks every last symbol by bijection h and in the last step rewrites it to the terminal, to generate x. Bijection h compensates non-existence of the final state in G.

The rigorous proof that $L(G, n) = L({}_nM)$ based on the mathematical induction is left to the reader.

□

Theorem 5.18. *For every* $k \geq 1$, ${}_k\mathscr{L}(\mathbf{RDPDA}) = {}_k\mathscr{L}(\mathbf{ST})$.

Proof of Theorem 5.18. Clearly, Theorem 5.18 is proved by Lemma 5.16, Lemma 5.17, and Theorem 2.5 in [Kas70]. ■

As a conclusion, we introduce open problem areas and variants of *RDPDA*.

Let ${}_n\mathscr{L}({}^{\mathbf{lr}}\mathbf{RDPDA})$ be a language family accepted by reducing deep pushdown automata of depth i, where $1 \leq i \leq n$, that process input string from the left to the right.

Recall the part of Definition 2.7 about the reversal operation over a string and about the reversal language. The reversal of w, denoted by $\mathrm{rev}(w)$, is w written in the reverse order, and the reversal of L, $\mathrm{rev}(L)$, is defined as $\mathrm{rev}(L) = \{\mathrm{rev}(w) |\ w \in L\}$.

Open problem 5.3. Consider the question—Is family ${}_n\mathscr{L}({}^{\mathbf{rl}}\mathbf{RDPDA})$ for $n \geq 1$ closed under this reversal operation? The solution is important for the next open problem—Is it important for generative power of *RDPDA* whether the input tape is read from the right to the left or from the left to the right?

Modifications of *RDPDA*

Every following modification is presented by its difference from *RDPDA* just by a gist, so the same or very analogous definitions of notions are omitted. The short discussion about generative power is included (without proofs). Every introduced version has its corresponding alternative in case of deep pushdown automata in the previous chapter.

RDPDA with erasing rules. Let us note that throughout this section, we have considered only true pushdown reductions in the sense that the pushdown non-empty substring is replaced with a symbol; at this point, no pushdown reduction can result in extending the pushdown string by a non-input symbol from ε. In opposite case, the possibility of a reduction of the empty string rapidly increases non-determinism in moves of *RDPDA*. Only the depth of such reduction states the substring on the pushdown where the reduction occurs. For instance, the reduction of ε in depth i inserts a non-input pushdown symbol somewhere between $(i-1)$th and ith non-input symbol on the pushdown. Nevertheless, the discussion of moves that allow *RDPDA*s to reduce ε to a non-input pushdown symbol and, thereby, extend its pushdown represents a natural generalization of *RDPDA*s discussed in Section 4.2.2. What is the power of *RDPDA*s generalized in this way?

Symbol-dependent RDPDA. In this modification, there is required to count concrete non-input symbols on the pushdown instead of arbitrary non-input symbols.

The definition of symbol-dependent $RDPDA$ is identical with classical version of $RDPDA$ (see Definition 4.11) up to the definition of reduction between two configurations of $RDPDA$:

$$(q, w, uvz) \,{}_r\Rightarrow (p, w, uAz) \ [qv \vdash mpA], \text{ where } \text{occur}(A, u) = m - 1$$
(instead of the original condition $\text{occur}(\Gamma - \Sigma, u) = m - 1$).

Example 5.2. Language $\{a^n b^n c^n \mid n \geq 1\}$ can be described by symbol-dependent $RDPDA$ with rules:

$$sab \vdash 1pA,\ pc \vdash 1qC,\ qaAb \vdash 1tA,\ tcC \vdash 1qC,\ qAC \vdash 1fS.$$

For symbol-dependent $RDPDA$s, we can introduce a generalization of n-limitation denoted as (n, M)-limitation that restricts the rewriting of first n concrete symbol occurrences from set M, where $M \subseteq \Gamma$. Again, the power of symbol-dependent $RDPDA$ under such limitation is an open problem.

In addition, the results concerning the introduced modifications of $RDPDA$ belong to the future investigation.

5.2 Infinite Hierarchies of Language Families

As obvious from the previous section, a few infinite hierarchies of language families can be derived from the results. More generally, most of rewriting systems with some restriction of their dynamic complexity create infinite hierarchies of language families that depend on the parameter of the particular restriction, such as finite index, depth, and limitation.

5.2.1 Based on Finite Index

The finite index is a very strong restriction of rewriting system configurations. Most regulated rewriting systems restricted by the finite index establish the same language family that properly contains **REG** and is contained in **CS** (see Equation 3.1 on page 38).

Theorem 5.19. *For every $k \geq 1$, $\mathscr{L}_k(\mathbf{CF\#RS}) \subset \mathscr{L}_{k+1}(\mathbf{CF\#RS})$.*

Proof of Theorem 5.19. In 1980, Gheorge Păun, using language $\{b(a^i b)^{2k} \mid i \geq 1\}$, $k \geq 1$, proved that programmed grammars of index k define an infinite hierarchy of language families (see [Pău80] or page 160, Theorem 3.1.7 in [DP89]). Theorem 5.19 follows from Theorem 5.3 and from this infinite hierarchy (see [KS06b]). ■

Theorem 5.20. *For every* $n \geq 1$, $\mathscr{L}(n\text{-}\mathbf{RLIN}\#\mathbf{RS}) \subset \mathscr{L}((n{+}1)\text{-}\mathbf{RLIN}\#\mathbf{RS})$.

Proof of Theorem 5.20. Recall that $\mathscr{L}(m\text{-}\mathbf{P}n\text{-}\mathbf{G}) \subset \mathscr{L}(m\text{-}\mathbf{P}(n{+}1)\text{-}\mathbf{G})$, for all $m, n \geq 1$ (see Theorem 8 in [Woo75]), which holds for $m = 1$ as well. Therefore, the theorem follows from Theorem 5.7 proving that $\mathscr{L}(n\text{-}\mathbf{RLIN}\#\mathbf{RS}) = \mathscr{L}(1\text{-}\mathbf{P}n\text{-}\mathbf{G})$. ■

Next, we establish the infinite hierarchy of language families defined by generalized #-rewriting systems based on the finite index restriction.

Theorem 5.21. *For every* $k \geq 1$, $\mathscr{L}_k(\mathbf{G}\#\mathbf{RS}) \subset \mathscr{L}_{k+1}(\mathbf{G}\#\mathbf{RS})$.

Proof of Theorem 5.21. From Theorem 5.11 follows that $\mathscr{L}_k(\mathbf{G}\#\mathbf{RS}) = \mathscr{L}_k(\mathbf{CF}\#\mathbf{RS})$. Recall that $\mathscr{L}_k(\mathbf{P}) = \mathscr{L}_k(\mathbf{CF}\#\mathbf{RS})$ (see [KMS06a]) and $\mathscr{L}_k(\mathbf{P}) \subset \mathscr{L}_{k+1}(\mathbf{P})$ for every $k \geq 1$ (see Theorem 3.1.2i and Theorem 3.1.7 in [DP89]). Then, Theorem 5.21 holds. ■

5.2.2 Based on *n*-limitation

In 1970, Kasai introduced state grammars (see Definition 2.23 in [Kas70]) where he proved that in the case of restriction of the number of nonterminals that can be rewritten during a derivation (n-limitation, see Definition 3.1), an infinite hierarchy of language families arises. By the demonstration of the equivalence of n-limited state grammars and, just introduced, right-to-left reducing deep pushdown automata, we get the same infinite hierarchy as well. As compared to state grammars, the n-limitation is expressed by the maximum depth of reduction operations given by the set of rules of the reducing deep pushdown automata.

Theorem 5.22. *For every* $k \geq 1$, ${}_k\mathscr{L}(\mathbf{^{rl}RDPDA}) \subset {}_{k+1}\mathscr{L}(\mathbf{^{rl}RDPDA})$.

Proof of Theorem 5.22. Theorem 5.18 states that for every $k \geq 1$, ${}_k\mathscr{L}(\mathbf{^{rl}RDPDA}) = {}_k\mathscr{L}(\mathbf{ST})$. Therefore, Theorem 5.22 follows from Theorem 5.18 and Theorem 2.5 in [Kas70]. ■

We have shown several results concerning the changes of the power of various rewriting systems depending on their restrictions. In addition, we established three infinite hierarchies of language families depending on the used rewriting system, on the restriction, and on the particular constraining constant (see Section 5.2).

Part III

Relationship with Practice

6

Application of Configuration Restrictions

This chapter discusses potential applications of the introduced rewriting systems and their restrictions. Of course, the primary attention is dedicated to #-rewriting systems.

First, we discuss the possible tasks that could be solved by these new systems. Next, we study the practical properties of these systems, such as determinism and canonical rewriting. Finally, we sum up all problems when the application area is the syntactical parsing.

6.1 Application Areas

Searching real applications is very difficult and sometimes omitted when a new regulated rewriting system without direct motivation of its definition by some particular problem is concerned. With respect to the restrictions in previous chapters, we can hardly find any applications in very challenging areas, such as natural language description or general description of context-sensitive properties of programming languages.

When we consider the formal language theory, the applications can be classified into the following categories:

- parsers and compilers (the description and processing of programming languages are the foundation stones of the computer science);
- linguistics (the study of the relationship between natural and formal languages; the formal description and processing of natural languages);
- biologically motivated systems (such as microbiology, bioinformatics, genetics; see [Kel97, KKMVM00, Pău02, Coh04]);
- modeling (the usage of a formal language as a formal model for general modeling and verification; see [BHV04]).

The general-purpose problem of rewriting systems is a membership of a given sentence into a given language that is defined by such rewriting system.

This task can be decided especially by the rewriting systems of accepting type, such as deterministic automata of various types.

Some concrete examples of practical use of configuration restrictions of rewriting systems follow next:

- the research of the relationship between k-limitation and syntactical parsers; for instance, there is an analogy between LL(k) top-down parsers and deep pushdown automata of depth k and between LR(k) bottom-up parsers and reducing deep pushdown automata of depth k (see [SSS90]);
- the usage of restriction of the number of active symbols during the parsing; in case of non-deterministic computation, the finite index restriction or the n-limitation restriction reduces the searched state space;
- the introduction of comparable measures for the complexities of formal models; for instance, the relationship of the normalized number of rules to get a final sentence of a language of two different rewriting systems can be studied from the dynamic complexity and descriptional complexity point of view;
- the use of regulation and restriction mechanisms to make the rule application decisions simpler; for instance, deterministic variants of the introduced rewriting systems, such as deterministic #-rewriting systems of type 2 or regulated pushdown automata.

6.2 Suitable Modifications of Formal Models

The deterministic variants of formal models are essential to the majority of applications in the formal language theory. From the beginning, the computers work based upon an algorithm. Hereafter, we consider only deterministic algorithms in order to repeat the computation and to get the same results for the same inputs.

Regarding Turing machines and corresponding formal models, such as unrestricted grammars, the determinism do not decrease the power of the formal model, but it affects the effectivity of its computation. For example, the deterministic simulation of a non-deterministic Turing machine requires greater space and time complexity in comparison with the non-deterministic variant. In the rest of the present chapter, we focus on the weaker formal models, such as #-rewriting systems and deep pushdown automata. Concerning these models, the main advantage is the rapidly lower complexity of the deterministic versions. For instance, this complexity is decreased from the cubic to the linear complexity.

The determinism is the key property of a formal model to work effectively. For instance, a deterministic pushdown automaton accepts or rejects every sentence in the linear time depending on the length of the sentence.

The definition of determinism for a particular regulated rewriting system is rarely mathematically described in a simple way because when the description

is simple, the determinism is mostly inapplicable in practice. In addition, the determinism very often decreases the power of the formal model, and if not, the use of such determinism is quite ineffective. This is illustrated by two types of deterministic #-rewriting systems.

6.2.1 Deterministic #-Rewriting Systems

#-rewriting systems are generative rewriting systems, which implies a difficult definition of determinism. For example, an analogy can be seen in case of context-free grammars and pushdown automata.

More concretely, this section concentrates its attention to the context-free #-rewriting systems. However, all types of determinism can be defined for n-right-linear and generalized #-rewriting systems as well.

Type 1

The first way of defining a determinism of the context-free #-rewriting systems naturally is to restrict the set of rules, so in the current state for every rewritable bounder there is at most one applicable rule, which leads to more straightforward selection of the rule to be applied in comparison with the general, non-deterministic, selection.

Definition 6.1. As a special case of context-free #-rewriting system $H = (Q, \Sigma, R, s)$, when for every $p \in Q$ and every positive integer $i \in \mathbb{N}$, $p_i\#$ is the left-hand side of no more than one rule in H; that is, for every pair $p \in Q$ and $i \in \mathbb{N}$ holds $\mathrm{card}(\{r \mid r\colon p_i\# \to q\ x \in R,\ q \in Q,\ x \in \Sigma^*\}) \leq 1$, then H is called *deterministic context-free #-rewriting system of type* 1 (abbrev. det_1CF#RS).

In general, a deterministic context-free #-rewriting system of type X is denoted by det_XCF#RS and the family of languages defined by these systems is denoted by $\mathscr{L}(\mathbf{det_X CF\#RS})$.

For instance, H_1 from Example 4.1 on page 48 is obviously non-deterministic because its second and fourth rule have the same left-hand side.

On the closer examination, we find out that during a sentence generation there is still the problem with the determination of the bounder that is the most appropriate to be rewritten in the following computational step. This problem of the determinism of type 1 (see Definition 6.1) significantly influences the following results.

Theorem 6.1. $\mathscr{L}(\mathbf{CF\#RS}) = \mathscr{L}(\mathbf{det_1 CF\#RS})$.

Proof of Theorem 6.1. Since $\det_1$CF#RS is a special case of CF#RS, we only need to prove that $\mathscr{L}(\mathbf{CF\#RS}) \subseteq \mathscr{L}(\mathbf{det_1CF\#RS})$.

Let $H = (Q_H, \Sigma, s_H, R_H)$ be a CF#RS, where $\Sigma = T \cup \{\#\}$. We construct the $\det_1$CF#RS, $D = (Q_D, \Sigma, s_H, R_D)$, where R_D and Q_D are constructed by performing the following steps:

1. For every $p \in Q_H$ and $i \in \mathbb{N}$, set ${}_p^i R_H = \{r\colon p\ {}_i\# \to q\ x \mid r \in R_H,\ q \in Q_H,\ x \in \Sigma^*\}$;
2. Set $Q_D = Q_H$ and auxiliary set $R' = \bigcup\{{}_p^i R_H \mid \mathrm{card}({}_p^i R_H) \leq 2\}$;
3. For every ${}_p^i R_H$ with $\mathrm{card}({}_p^i R_H) \geq 3$ compute the following algorithm:
   $o := p$;
   `while` $(\mathrm{card}({}_p^i R_H) \geq 3)$ `do:`
   - exclude r from ${}_p^i R_H$
   - add $\langle {}_p^i R_H \rangle$ into Q_D
   - add $o\ {}_i\# \to \mathrm{rhs}(r)$ and $o\ {}_i\# \to \langle {}_p^i R_H \rangle\ \#$ into R';
   - $o := \langle {}_p^i R_H \rangle$;
4. Set $R_D = \bigcup\{{}_p^i R' \mid \mathrm{card}({}_p^i R') = 1\}$;
5. Let $<$ be a strict order on $\mathrm{Lab}(R')$. For every pair of rules $r_i\colon p\ {}_i\# \to q_1\ x_1$ and $r_j\colon p\ {}_i\# \to q_2\ x_2$ from R' such that $r_i < r_j$, add the following rules into R_D:
   - $p\ {}_i\# \to \langle r_i, r_j \rangle\ \#\#$
   - $\langle r_i, r_j \rangle\ {}_i\# \to \langle r_i \rangle\ \#$
   - $\langle r_i, r_j \rangle\ {}_{i+1}\# \to \langle r_j \rangle\ \#$
   - $\langle r_i \rangle\ {}_i\# \to \langle r_i' \rangle\ x_1$
   - $\langle r_j \rangle\ {}_i\# \to \langle r_j' \rangle\ x_2$
   - $\langle r_i' \rangle\ {}_{i+1}\# \to q_1\ \varepsilon$
   - $\langle r_j' \rangle\ {}_{i+1}\# \to q_2\ \varepsilon$

   and add newly created states $\langle r_i, r_j \rangle, \langle r_i \rangle, \langle r_j \rangle, \langle r_i' \rangle, \langle r_j' \rangle$ into Q_D.

Basic Idea of the Proof of Theorem 6.1.

Step 1. The subsets ${}_p^i R$ denote the sets of rules with the same left-hand side. Therefore, the cardinality of these sets equals to the number of rules with the same left-hand side (the degree of non-determinism).

Steps 2-3. The set of rules, R_H, is transformed into the new set of rules R' that contains at most two rules with the same left-hand side.

Step 4. The new set of rules, R_D, is initialized by all already deterministic rules from R'. Now, we only need to handle the pairs of rules with the same left-hand side.

Step 5. Every pair of rules with the same left-hand side is simulated by the sequence of seven new rules in R_D:
(1) generates the auxiliary $(i+1)$th bounder,
(2) and (3) do the selection from one of these simulated rules,
(4) and (5) rewrite the ith bounder,

(6) and (7) change the state to the target state and erase the auxiliary bounder.

■

Example 6.1. Let us show the construction based on the proof of Theorem 6.1 by transforming CF#RS from Example 4.1 on page 48, H_1, into $\det_1$CF#RS, D. Since the degree of determinism is less than three, we can skip first three steps of the construction. Then, we copy all deterministic rules into R_D by actions in step 4, so $R_D = \{1\colon s\ {}_1\# \to p\ \#\#, 3\colon q\ {}_2\# \to p\ \#c, 5\colon f\ {}_1\# \to f\ c\}$ and $Q_D = \{s, p, q, f\}$. Finally, by step 5, we generate the simulating sequence for the pair of non-deterministic rules, $2\colon p\ {}_1\# \to q\ a\#b$ and $4\colon p\ {}_1\# \to f\ ab$.

Since $r_i = 2$, $r_j = 4$, $p = p$, $i = 1$, $q_1 = q$, $q_2 = f$, $x_1 = a\#b$, and $x_2 = ab$, add the following rules into R_D:

$p\ {}_1\# \to \langle 2,4\rangle\ \#\#$
$\langle 2,4\rangle\ {}_1\# \to \langle 2\rangle\ \#$
$\langle 2,4\rangle\ {}_2\# \to \langle 4\rangle\ \#$
$\langle 2\rangle\ {}_1\# \to \langle 2'\rangle\ a\#b$
$\langle 4\rangle\ {}_1\# \to \langle 4'\rangle\ ab$
$\langle 2'\rangle\ {}_2\# \to q\ \varepsilon$
$\langle 4'\rangle\ {}_2\# \to f\ \varepsilon$

At the end, $Q_D = \{s, p, q, f, \langle 2,4\rangle, \langle 2\rangle, \langle 4\rangle, \langle 2'\rangle, \langle 4'\rangle\}$.

Corollary 6.2. *For every $k \geq 1$, $\mathscr{L}_k(\mathbf{CF\#RS}) \subseteq \mathscr{L}_{k+1}(\mathbf{det_1CF\#RS})$.*

Proof of Corollary 6.2. Since there is always only one more bounder added in the construction proof of Theorem 6.1, the corollary holds. ■

Theorem 6.3. *For $k = 1$ and language $L \in \mathscr{L}_k(\mathbf{det_1CF\#RS})$, $\mathrm{card}(L) \leq 1$.*

Proof of Theorem 6.3. Let us assume that $k = 1$ and $\mathrm{card}(L) > 1$. The determinism implies that every rule has a unique left-hand side. Thus, the states and their programming by the finite-state control restrict branching and even cycles in the rules, thereby, in the computation. In $\det_1$CF#RS, we cannot generate two different configurations from the initial configuration, $s\#$, by any number of computational steps. So, the assumption that $k = 1$ and $\mathrm{card}(L) > 1$ is wrong. ■

Hypothesis 6.1. There is a construction algorithm that does not increase the index of the $\det_1$CF#RS during the construction from an arbitrary CF#RS of index k. In symbols, for every $k \geq 2$, $\mathscr{L}_k(\mathbf{CF\#RS}) = \mathscr{L}_k(\mathbf{det_1CF\#RS})$.

Let us discuss the solution of the hypothesis in greater detail. Notice that we need not generate auxiliary bounder as the $(i+1)$th bounder. Specifically, we can choose already existing bounder, which is not planned to be rewritten in the following step, by modulo function (remainder after the integer division operation). For instance, for $k = 3$, if we want to rewrite the second bounder, we do not need introduce the fourth bounder and increase index k, but we can use the rewriting of the first or third bounder, instead. Of course, this principle works only for $k \geq 2$. As a result, we can get even better result than in Theorem 6.1; that is, for every $k \geq 2$, $\mathscr{L}_k(\mathbf{CF\#RS}) = \mathscr{L}_k(\mathbf{det_1 CF\#RS})$.

In fact, the way of achieving the determinism (according to Definition 6.1) from non-deterministic CF#RS is the replacement of one type of non-determinism by another. In other words, consider two questions about rule selection:

- "Which bounder will be replaced and by which rule?"
- "Which bounder will be replaced in a fundamental rewriting step of a computation?".

Then, the first question describing a non-deterministic CF#RS is replace by the second question characteristic for the determinism of type 1. Thus, in the determinism of type 1, we do not care about which rule to choose, but only which bounder to rewrite.

In the following paragraphs, we thought of another type of deterministic #-rewriting system defined in a more practical way. For instance, it is useful in the syntax analysis.

Type 2

In general, it is a problem to state reasonable condition for determinism of generative formal models. In most cases, trivial conditions restricting the form of rules are not enough. For example, the definition of deterministic context-free grammars is based on the set of conditions for every rule. Specifically, based on a few input symbols, we unambiguously decide which rule should be used to successfully accept a sentence of a language. In addition, if the sentence belongs to the language there will be no need to reconsider any part of the computation chosen earlier. In other words, no backtracking is needed to throw away some already done part of the computation and to choose another path to the final configuration.

The problem of defining determinism in regulated rewriting systems is even more difficult because additional regulated mechanisms extend the number of variants. For instance, the definition of a determinism can be based on the level of a regulating mechanism, on the level of a rewriting mechanism, or even a combination.

By analogy with the conditions in LL(1) grammars (see [SSS90]), the definition of determinism of type 2 contains less restricting determinism during

the rewriting of the first bounder in a configuration and more restrictions during the rewriting of other bounders. The whole idea is to avoid the necessity to work and read the input in several places in the sentence. This approach can be advantageously used in the case of deep pushdown automata as well.

Concept 6.1. Let $H = (Q, \Sigma, R, s)$ be a *deterministic context-free #-rewriting system of type* 2 ($\det_2$CF#RS) if the following conditions hold:

For every state, there is at most one rule that manipulates any bounder but first. In addition, the existence of a rule that rewrites the first bounder in this state is not allowed. In case of rules working with the first bounder, we apply conditions analogical to LL(1) grammars. That is, only when the first bounder is to be rewritten, the deterministic decision is made based on two pieces of information—(1) the current state and (2) the read input symbol. The rule in other rewriting steps is determined only by the current state. This procedure is illustrated by Example 6.2.

Example 6.2. To illustrate the example, we describe an accepting mode of deterministic CF#RS of type 2, H.

Mostly, H works in the original generative mode. Only when H applies certain rules, it reads an input symbol denoted by in and moves the reading head to the next input symbol. We do not require to read the whole input string, only a prefix necessary to the right decisions during the rewriting of the first bounders. The rules for the rewriting of the first bounder that read an input symbol to do the proper rule selection contain the condition of the form (in = the list of accepted symbols called a *conditional set*). A rule without conditional set does not read an input because the current input symbol does not matter. We require that every pair of rules with the same initial state contains disjunctive conditional sets to avoid indecisiveness.

Let H contain the following rules:

1: $s\,{}_1\# \rightarrow p\ \#\#$
2: $p\,{}_1\# \rightarrow q\ a\#b$ $(in = \{a\})$
3: $q\,{}_2\# \rightarrow p\ c\#$
4: $p\,{}_1\# \rightarrow f\ \varepsilon$ $(in = \{b\})$
5: $f\,{}_1\# \rightarrow f\ \varepsilon$

H processes *aabbcc* as follows: $s\# \Rightarrow p\#\#$ [1] $\Rightarrow qa\#b\#$ [2 $(in = a)$] $\Rightarrow pa\#bc\#$ [3] $\Rightarrow qaa\#bbc\#$ [2 $(in = a)$] $\Rightarrow paa\#bbcc\#$ [3] $\Rightarrow faabbcc\#$ [4 $(in = b)$] $\Rightarrow faabbcc$ [5].

By the determinism definition of type 2, we have solved the problem with the sentence processing. A #-rewriting system works with the configuration on several places (not only at the beginning). However, we hardly determine which part of the configuration corresponds to which part of the input string. Thus, the reading of the input tape can be handled only from the beginning

to preserve the deterministic specification of the reading location in the input tape, even for non-context-free languages, such as in Example 6.2.

To sum up this section, we showed that mathematically oriented determinism of type 1 remains the power of the system unchanged. On the contrary, more practical determinism of type 2 restricts the power significantly. But it is still an open problem what is the exact power of $\det_2$CF#RS?

6.2.2 Canonical #-Rewriting Systems

The canonical derivation plays a crucial role in the syntactical analysis by context-free grammars and in the parsing theory. That is the main motivation to study canonical rewriting in new formal models, especially #-rewriting systems. The canonical step rewrites a bounder in more systematic and deterministic way because it rewrites only the leftmost active symbol. This notion is similar to n-limitation restriction, but more restrictive. In general, the canonical rewriting replaces several leftmost active symbols. In case of the leftmost rewriting (or only left rewriting) we consider a replacement of the leftmost active symbol in a selected configuration component, such as sentential configuration component. Note that the component with the current state is usually not limited in this way (see Note 6.1).

For instance, regulated grammars, which are the inspiration for #-rewriting systems, allow several different types of canonical derivations as investigated in [CMM73, DP89, Fer00, Mau73, Pău85].

Let us define a canonical computational step in #-rewriting systems in two ways as follows:

a) **Strictly canonical** step rewrites one of k leftmost bounders without any reflection to the current state, where $k \geq 1$. This type directly corresponds to the k-limitation restriction of the computation (see Definition 3.1).
b) **State-controlled canonical** rewriting step works with k leftmost bounder as well and, in addition, considers the current state. That is, we select k leftmost occurrences of active symbols rewritable in the current state by existing rules. Then, we choose the symbol from this selection to be rewritten.

In the rest of this section, let $H = (Q, \Sigma, R, s)$ be a context-free #-rewriting system, $n = \max(\{i \mid p\,{}_i\# \to q\ x \in R\})$ be the maximum index of the bounder that can be rewritten by rules from R, and $k \in \mathbb{N}$.

Definition 6.2. *Strictly k-canonical rewriting step* is a computational step of degree j, where $j \leq k$. Strictly k-canonical computation contains only k-canonical rewriting steps. Strictly k-canonical language is defined as $L(H) = \{w \in (\Sigma - \{\#\})^* \mid s\# \Rightarrow^* qw$ by strictly k-canonical computation in $H\}$. $\mathscr{L}(\mathbf{CF\#RS}, k\text{-strictly canonical})$ denotes the family of these languages.

Definition 6.3. For every $p \in Q$, we define $K(H,p) = \{i \mid p\,{}_i\# \rightarrow q\ x \in R\}$ to be the set of indices of all rewritable bounders in the current state p of H. The rewriting step $px \Rightarrow qy$ is said to be *state-controlled k-canonical* if $\max(K(H,p)) - \min(K(H,p)) < k$. State-controlled k-canonical computation and language are defined by analogy with the previous definition. The language family is denoted by $\mathscr{L}$(**CF#RS**, state-controlled k-canonical).

Notice that k can be selected from interval $1 \leq k \leq n$ because every CF#RS is implicitly from its definition strictly n-canonical.

Next, let us illustrate the core of an algorithm for the transformation of each CF#RS into an equivalent state-controlled 1-canonical CF#RS; that is, $\mathscr{L}$(**CF#RS**) $\subseteq$ $\mathscr{L}$(**CF#RS**, state-controlled 1-canonical).

Algorithm 6.1 (sketch)**.** Replace every rule $r\colon p\,{}_i\# \rightarrow q\ x \in R$, where $i > \min(K(H,p)) = m$, by two new rules:

$$p\,{}_m\# \rightarrow \langle r \rangle\ \#$$

$$\langle r \rangle\,{}_i\# \rightarrow q\ x$$

where $\langle r \rangle$ is a new state.

From this algorithm, it follows that the second type of the canonical computation is not very suitable for practical usage. Thus, we focus on the strictly k-canonical rewriting in the rest of this section. The most valuable variant for practice is the case when $k = 1$; that is, we say a *leftmost rewriting step* and *leftmost #-rewriting system*, respectively. The leftmost rewriting steps eliminate the fundamental source of non-determinism in #-rewriting systems. Hence, if we consider strictly 1-canonical computation, only the rules manipulating the first bounder of a configuration are relevant.

In the future, we suggest to study a hypothesis saying that every language defined by a leftmost #-rewriting system belongs to **CF**. The main idea of a potential proof for the previous hypothesis is to demonstrate that every leftmost #-rewriting system can be transformed into an equivalent context-free grammar. The problem is in a precise simulation of a state control of #-rewriting system. Perhaps, we can use the first nonterminal occurrence in the corresponding context-free sentential form to keep the information about the simulated state. The other nonterminal occurrences reflect to the rest of the bounders.

Open problem 6.2. $\mathscr{L}$(**CF#RS**, 1-strictly canonical) $\subset$ **CF**?

As a proof of the previous open problem, we have to prove the previous hypothesis, and even in the other direction, we have to demonstrate that there is a context-free language that can be generated by no leftmost #-rewriting system.

Note 6.1. In more detail, k-canonical rewriting of #-rewriting system configuration does not include the rewriting of the current state into the constant k. For instance, for $k = 1$, we apply the rewriting to the state configuration component and to the leftmost bounder of the sentential configuration component. In other words, we rewrite two leftmost active symbols—(1) the current state in the first component of the configuration and (2) the first bounder in the second component of the configuration.

In conclusion, let us make a note that from a practical point of view, the combination of some type of k-canonical rewriting and some type of determinism should be investigated in further detail.

6.2.3 Comment on Deterministic Deep Pushdown Automata

A similar transformation as for deterministic #-rewriting systems of type 1 can be used for deep pushdown automata that are deterministic with respect to their depth of expansions (see Definition 4.10 and Corollary 1 in [Med06]).

Let us present the main idea of the transformation algorithm that sketches the conversion between a deep pushdown automaton ${}_nM$ and an equivalent deterministic deep pushdown automaton with respect to its depth of expansions.

Algorithm 6.2 (sketch)**.** If for some $p \in Q$ exists two rules $r' \colon ipA \to qu$ and $r'' \colon jpB \to ov$ in R, where $i \neq j$ (that is, r' and r'' do not satisfy the condition about the determinism with respect to depth of expansions), then we replace these two rules by the following three rules:

$$r' \colon ipA \to qu,$$

$$\rho \colon ipA \to \langle r'' \rangle A,$$

$$\rho' \colon j\langle r'' \rangle B \to ov,$$

where ρ, ρ' are new unique rule labels and $\langle r'' \rangle$ is a new state. We repeat this conversion of rules until we get R that contains only deterministic rules with respect to their depth of expansions.

Still, a question whether a deep pushdown automaton can be transformed into an equivalent strictly deterministic deep pushdown automata remains open.

6.3 Syntactical Analysis

The major task of syntactical analysis is to decide whether a given sentence belongs to the language defined by a given rewriting system (see [ALSU06, SSS87]).

In general, the main problem in regulated rewriting systems classified under the family of context-sensitive languages is the possibility to rewrite their configuration in several places. The selection of such place is usually not random or the leftmost, but it is regulated by finite-state control that includes another dimension of non-determinism to these systems. Both generative and accepting systems require a mechanism to predict which active symbol corresponds to which substring segment of the resulting sentence or the input sentence. The straightforward solution to this difficult problem is to introduce a restriction that ensures the disjunctiveness of sets of starting prefixes of these substring segments, so every starting prefix identifies the segment unambiguously.

6.3.1 Programming Languages

The most common usage of syntactical analysis and parsing theory is in the area of programming language processing (see ([IRS76, LMW79, Weg72, Wol96, Woo84]). Mostly, programming languages are specified by context-free grammars with special additional context conditions to their rules. When we build up a parser of a programming language, the essential phase is focused on the integration of these context conditions with rules of the grammar into the parser.

Unfortunately, #-rewriting systems do not suit well to the description of programming languages since these systems cannot even describe every context-free language, such as the language for expressions (see Example 4.2 with unlimited number of bracket types). In fact, #-rewriting systems cannot handle even some context-sensitive problems that are introduced by programming language structures.

For example, consider a programming language that requires the occurrence of a variable declaration before the variable definition and a use of the variable in a program. We call this problem a *declaration-definition problem* of variables. In addition, this problem is complicated by structured programs that can be contained in another program structure. Then, the variables with the same name but different memory location can occur. The location of the variable value depends on its name and program context. The idea of declaration-definition problem is described by language $D = \{(wc)^n |\ n \geq 1, w \in (\Sigma - \{c\})^*\}$ (see [DP89]), where c is a separator, $c \notin \text{alph}(w)$, and w is the name of the variable such that the first occurrence of w corresponds to its declaration and the other occurrences of w correspond to the definition of w or its use. Though #-rewriting systems cannot solve this problem, they are somewhat better than context-free grammars. Indeed, for some positive, constant integer n, there exists a context-free #-rewriting system that generates D.

This chapter studied the properties of context-free #-rewriting systems and their modifications from the practical point of view.

Part IV

Conclusion

7

Conclusion

Traditionally, the last chapter summarizes the results and suggests a future investigation. In addition, we discuss a few hypotheses and open problem areas to be studied consecutively.

7.1 Summary

This monograph generalizes automata and grammars as a common notion of a rewriting system that changes its inner state by steps that rewrite various parts of its configuration according to the concrete rewriting system definition. The notion of configuration represents the inner state and was absorbed from the theory of automata and generalized for rewriting systems. The configuration is a finite sequence of components. A component is a string over the total alphabet of the rewriting system. A rewriting step (computational step) between configurations is made by a rewriting rule. Consecutively, we unify the notions from the area of automata and grammars to be able to study rewriting systems created by their various combinations (see Chapter 1 and Chapter 2).

Next, we introduced an intuitive classification of restrictions and regulation classification of rewriting systems to study easier dynamic complexity, which investigates rewriting systems from another point of view in comparison with the descriptional and time and space complexity. The descriptional complexity describes the effectiveness of language description by an instance of a rewriting system, such as the cardinality of rule set. On the other hand, the dynamic complexity examines the effectiveness of rewriting itself during the processing a language sentence. For instance, the maximum sufficient number of nonterminals in configurations during a computation is the measure of the dynamic complexity (see Chapter 3).

Chapter 4 defines new rewriting systems and Chapter 5 demonstrates results achieved by the restriction of their configuration and the restriction of

sequences of applied rules. More specifically, we consider the following formal models (with the corresponding language families):

- #-rewriting systems:
  - context-free #-rewriting systems ($\mathscr{L}(\mathbf{CF\#RS})$)
  - n-right-linear #-rewriting systems ($\mathscr{L}(n\text{-}\mathbf{RLIN\#RS})$)
  - generalized #-rewriting systems ($\mathscr{L}(\mathbf{G\#RS})$)
- deep pushdown automata ($\mathscr{L}(\mathbf{DTDP})$) and reducing deep pushdown automata ($\mathscr{L}(^{\mathbf{rl}}\mathbf{RDPDA})$)
- restricted pushdown automata ($\mathscr{L}(\mathbf{RPDA}, X)$)

These rewriting systems are restricted by the following types of restrictions that are classified in Chapter 3:

- finite index k ($\mathscr{L}_k(\mathbf{X})$)
- k-limitation (${}_k\mathscr{L}(\mathbf{X})$)
- depth k (${}_k\mathscr{L}(\mathbf{X})$)
- restriction of configurations by a language Y ($\mathscr{L}(\mathbf{X}, Y)$)

The relationship between the dynamic complexity and the configuration restriction mostly leads to the infinite hierarchies of language families, which is illustrated by several examples of various regulated rewriting systems.

Next, recall the most important results of this book (see Chapter 5) that characterize the infinite hierarchies of language families based on the various restrictions of rewriting systems restricted by a parameter $k \geq 1$:

$$ {}_k\mathscr{L}(\mathbf{ST}) = {}_k\mathscr{L}(\mathbf{DTDP}) = {}_k\mathscr{L}(^{\mathbf{rl}}\mathbf{RDPDA}) $$

$$ \mathscr{L}_k(\mathbf{P}) = \mathscr{L}_k(\mathbf{RC}) = \mathscr{L}_k(\mathbf{CF\#RS}) = \mathscr{L}_k(\mathbf{det_1CF\#RS}) = \mathscr{L}_k(\mathbf{G\#RS}) $$

$$ \mathscr{L}(k\text{-}\mathbf{RLIN\#RS}) = \mathscr{L}(1\text{-}\mathbf{P}k\text{-}\mathbf{G}) $$

The investigation of restrictions and regulations of formal models is a part of the formal language theory for several decades. Still, there are some theoretical and practical challenges, such as in [Kol05, KM05a, Kot02, Ryc07]. Therefore, Chapter 6 focuses on properties that are relevant to a practical use of introduced #-rewriting systems that are the core formal model in this monograph. Specifically, we study the following properties and their perspective for applications:

- two types of determinism
- two types of canonical rewriting and their relation to the limitation

As a conclusion of Chapter 6, the potential applications require even more sophisticated algorithms and modifications of regulated rewriting systems that

are not available now. Therefore, the effort of finding a reasonable application of #-rewriting systems continues. On the other hand, more general deep pushdown automata can obviously adopt the introduced concepts of limitation and apply them in practical field, such as syntax analysis of non-context-free languages.

7.2 Contribution

Let us condense the main contributions of this monograph:

1. the study of the combinational rewriting systems and the connected notions that simplify and generalize their investigation (the uniform approach);
2. the classification of formal model restrictions with emphasis on the configuration restrictions of rewriting systems (including a few results concerning infinite hierarchies of language families);
3. the introduction of a new regulated generative rewriting system—#-rewriting system that combines grammars, automata and some additional properties.

7.3 Future Investigation

As usual, some questions remain unanswered. Now, we discuss several questions and areas of the future investigation.

7.3.1 Hypotheses and Open Problems

Apart from the previous discussion, here, we study the modifications of rewriting systems and open problems from the text in more detail.

#-Rewriting Systems

Through the entire book, various restrictions of rewriting systems are studied. The natural question asks what is the influence on the power of such systems if we relax these restrictions from their definitions. For instance, think of the power of unbounded index of #-rewriting systems of all introduced types. In fact, there will be paid some attention to similar open problems, such as Open Problem 5.2 in the future.

Open problem 7.1. ${}_n\mathscr{L}(\mathbf{ST}) \subseteq \mathscr{L}(\mathbf{G}\#\mathbf{RS}) \subseteq \mathbf{CS}$?

Open problem 7.2. $\mathbf{CF} \subseteq \mathscr{L}(\mathbf{CF}\#\mathbf{RS})$?

In brief, we sketch a few natural modifications and extensions of #-rewriting systems:

- reducing and parallel #-rewriting systems as drafted in Section 4.1.5;
- generalized #-rewriting systems with an alphabet of nonterminals that we know from grammars; as a matter of fact, we can define grammars that consider the order of a rewritten nonterminal in their rules;
- #-rewriting systems with limited cardinality of the set of states which probably lead to even more fine-grained infinite hierarchy of language families.

Deep Pushdown Automata

Firstly, notice the similarity between context-free #-rewriting systems (without the finite index restriction) and deep pushdown automata. Both automata and systems have implicitly from-left restricted number of variables that can be rewritten. Both contain a finite-state control. Therefore, what is the relationship between the power of context-free #-rewriting systems and the power of deep pushdown automata?

To gain practical applications, the determinism of the introduced rewriting systems has to be investigated in more detail. The mainstream inspiration comes from LL(k) grammars and deterministic #-rewriting systems of type 2. Next, the combination of such approaches with deep pushdown automata may work as well.

When we consider natural modifications of general n-limitation, such as introduced in deep pushdown automata, we can define more restricted n-limitation dependent on a concrete symbol A; that is, (n, A)-limitation (see symbol-dependent modification of RDPDA in Section 5.1.5).

Restricted Pushdown Automata

As the complementary notion to regulated pushdown automata (see [KM00]), we introduce RPDA. The main difference is that instead of restriction of the sequence of applied rules by some regulating language, we restrict the content of the pushdown of RPDA. If we consider linear restricting languages, there remains an open question whether RPDA has the similar power to regulated pushdown automata.

Open problem 7.3. $\mathscr{L}(\mathbf{RPDA}, \mathbf{LIN}) = \mathbf{RE}$?

There are a few ways of proving this problem. The most natural one is to establish an equivalence with another rewriting system that characterizes **RE**, such as regulated pushdown automata regulated by linear languages, queue grammars (see [KR83]), and unrestricted grammars.

Index

References

[ALSU06] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2006.

[AP02] A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, Cambridge, 2nd edition, 2002.

[AU72] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation and Compiling, Volume I: Parsing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

[AU73] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation and Compiling, Volume II: Compiling*. Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[AU77] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, Massachusetts, 1977.

[BB05] R. Bidlo and P. Blatný. How to generate recursively enumerable languages using only context-free productions and eight nonterminals. In *Proceedings of 11th Conference and Competition Student EEICT 2005, Volume 3*, pages 536–541. Faculty of Electrical Engineering and Communication BUT, 2005.

[BF95] H. Borodihn and H. Fernau. Accepting grammars and systems: an overview. In *Proceedings of Development in Language Theory Conf.*, volume 53, pages 199–208, Magdeburg, 1995.

[BHV04] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *Proc. of 16th International Conference on Computer Aided Verification—CAV'04*, volume 3114 of LNCS, pages 197–202. Springer-Verlag, 2004.

[Bro89] J. G. Brookshear. *Theory of Computation*. Benjamin/Cummings, Redwood City, California, 1989.

[CMM73] A. Cremers, H. A. Maurer, and O. Mayer. A note on leftmost restricted random context grammars. *Information Processing Letters*, 2:31–33, 1973.

[Coh04] J. Cohen. Bioinformatics–an introduction for computer scientists. *ACM Computing Surveys*, 36:122–158, 2004.

[CVDKP94] E. Csuhaj-Varju, J. Dassow, J. Kelemen, and Gh. Păun. *Grammar systems. A grammatical approach to distribution and cooperation. Topics in Computer Mathematics 8.* Gordon and Breach Science Publishers, Yverdon, 1994.

[DP89] J. Dassow and Gh. Păun. *Regulated Rewriting in Formal Language Theory.* Akademie-Verlag, Berlin, 1989.

[Fer97] H. Fernau. Graph-controlled grammars as language acceptors. *Journal Automata, Languages and Combinatorics*, 2(2):79–91, 1997.

[Fer00] H. Fernau. Regulated grammars under leftmost derivation. *Grammars*, pages 37–62, 2000.

[Hoo87] H. J. Hoogeboom. *Coordinated Pair Systems.* Universiteit Leiden, 1987.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[Iba70] O. H. Ibarra. Simple matrix languages. *Information and Control*, 17:359–394, 1970.

[IRS76] P. M. Lewis II, D. J. Rosenkrantz, and R. E. Stearns. *Compiler Design Theory.* Addison-Wesley, Reading, Massachusetts, 1976.

[Kar91] L. Kari. *On insertion and deletion in formal languages.* Turku, Finland, 1991.

[Kas70] T. Kasai. A hierarchy between context-free and context-sensitive languages. *Journal of Computer and System Sciences*, 4:492–508, 1970.

[Kel97] J. Kelemen. Colonies as models of reactive systems. *Lecture Notes in Computer Science: New Trends in Formal Languages*, 1218:220–235, 1997.

[KKMVM00] J. Kelemen, A. Kelemenová, C. Martín-Vide, and V. Mitrana. Colonies with limited activation of components. *Theoretical Computer Science*, 244:289–298, 2000.

[KM00] D. Kolář and A. Meduna. Regulated pushdown automata. *Acta Cybernetica*, 4:653–664, 2000.

[KM05a] D. Kolář and A. Meduna. Regulated automata: From theory towards applications. In *Proceeding of 8th International Conference on Information Systems Implementation and Modelling ISIM'05*, pages 33–48, Ostrava, CZ, 2005. MARQ.

[KM05b] Z. Křivka and A. Meduna. Random context and programmed grammars of finite index have the same generative power. In *Proceedings of 8th International Conference ISIM'05 Information Systems Implementation and Modelling*, 1st edition, pages 67–72, 2005.

[KM06] Z. Křivka and A. Meduna. General top-down parsers based on deep pushdown expansions. In *Proceedings of 1st International Workshop on Formal Models (WFM'06)*, pages 11–18, 2006.

[KMS06a] Z. Křivka, A. Meduna, and R. Schnecker. Generation of languages by rewriting systems that resemble automata. *International Journal of Foundations of Computer Science*, 17(5):1223–1229, 2006.

[KMS06b] Z. Křivka, A. Meduna, and R. Schnecker. Reducing deep pushdown automata and infinite hierarchy. In *MEMICS 2006 Second Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, pages 214–221, 2006.

[Kol05] D. Kolář. *Pushdown Automata: Another Extensions and Transformations*. FIT VUT, Brno, CZ, 2005.

[Kot02] M. Kot. *Řízené gramatiky [Diplomová práce]*. VŠB Technická univerzita Ostrava, Fakulta elektrotechniky a informatiky, Katedra informatiky, 2002.

[KR83] H. C. M. Kleijn and G. Rozenberg. On the generative power of regular pattern grammars. *Acta Informatica*, 20:391–411, 1983.

[Kři04a] Z. Křivka. Dvoucestné k-lineární n-komponentní gramatické systémy. In *Proceedings of the 10th Conference and Competition STUDENT EEICT 2004 Volume 1*, pages 203–205. Faculty of Electrical Engineering and Communication, Brno University of Technology, 2004.

[Kři04b] Z. Křivka. Zásobníkové automaty s omezeným obsahem zásobníku. In *ACM STUDENT CZ*, page 8, 2004.

[Kři05a] Z. Křivka. Recursive erasing in programmed grammars. In *Pre-Proceedings MEMICS 2005*, pages 139–144, 2005.

[Kři05b] Z. Křivka. String-partitioning systems. In *Proceedings of 11th Conference and Competition STUDENT EEICT 2005 Volume 3*, pages 556–560. Faculty of Electrical Engineering and Communication BUT, 2005.

[Kři05c] Z. Křivka. String-partitioning systems. In *Proceedings of International Interdisciplinary HONEYWELL EMI 2005*, pages 217–221. Faculty of Electrical Engineering and Communication BUT, 2005.

[KS06a] Z. Křivka and R. Schnecker. Reducing deep pushdown automata. In *Proceedings of the 12th Conference and Competition STUDENT EEICT 2006 Volume 4*, pages 365–369, 2006.

[KS06b] Z. Křivka and R. Schnecker. String-partitioning systems and an infinite hierarchy. In *Proceedings of 1st International Workshop on Formal Models (WFM'06)*, pages 53–60, 2006.

[Lin90] P. Linz. *An Introduction to Formal Languages and Automata*. D.C. Heath and Co., Mass, Lexigton, 1990.

[LM05] L. Lorenc and A. Meduna. Self-reproducing pushdown transducers. *Kybernetika*, 41(4):531–537, 2005.

[LMW79] R. C. Linger, H. D. Mills, and B. I. Witt. *Structured Programming: Theory and Practice*. Addison-Wesley, Reading, Massachusetts, 1979.

[Mau73] H. A. Maurer. Simple matrix languages with a leftmost restriction. *Information and Control*, 23:128–139, 1973.

[Med96] A. Meduna. Syntactic complexity of context-free grammars over word monoids. *Acta Informatica*, 33:457–462, 1996.

[Med97a] A. Meduna. Four-nonterminal scattered context grammars characterize the family of recursively enumerable languages. *International Journal of Computer Mathematics*, 63:67–83, 1997.

[Med97b] A. Meduna. On the number of nonterminals in matrix grammars with leftmost derivations. *LNCS*, 1217:27–38, 1997.

[Med98] A. Meduna. Descriptional complexity of multi-continuous grammars. *Acta Cybernetica*, 13:375–384, 1998.

[Med99] A. Meduna. Prefix pushdown automata and their simplification. *International Journal of Computer Mathematics*, 71(1):1–20, 1999.

[Med00] A. Meduna. *Automata and Languages: Theory and Applications*. Springer, London, GB, 2000.

[Med04] A. Meduna. *Moderní Teoretická Informatika [materiály k přednáškám]*. FIT VUT, Brno, CZ, 2004.

[Med06] A. Meduna. Deep pushdown automata. *Acta Informatica*, 2006(98):114–124, 2006.

[MF03] A. Meduna and H. Fernau. A simultaneous reduction of several measures of descriptional complexity in scattered context grammars. *Information Processing Letters*, 86:235–240, 2003.

[MHHO05] E. Moriya, D. Hofbauer, M. Huber, and F. Otto. On state-alternating context-free grammars. *Theoretical Computer Science*, 337:183–216, 2005.

[MK02] A. Meduna and D. Kolář. One-turn regulated pushdown automata and their reduction. *Fundamenta Informaticae*, 16:399–405, 2002.

[MM07] T. Masopust and A. Meduna. Descriptional complexity of grammars regulated by context conditions. In *LATA 2007 Preproceedings*, pages 403–411, Tarragona, ES, 2007.

[MŠ05] A. Meduna and M. Švec. *Grammars with Context Conditions and Their Applications*. John Wiley & Sons, Hoboken, New Jersey, USA, 2005.

[MV04] A. Meduna and M. Vítek. New language operations in formal language theory. *Schedae Informaticae*, 2004(13):123–150, 2004.

[Pap94] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Mass, 1994.

[Pău80] Gh. Păun. An infinite hierarchy of matrix languages. *Stud. Cerc. Mat.*, 32:697–707, 1980.

[Pău85] Gh. Păun. On leftmost derivation restriction in regulated rewriting. *Rev. Roumaine Math. Pures Appl.*, 30:751–758, 1985.

[Pău02] Gh. Păun. *Membrane Computing: An Introduction*. Springer, 2002.

[PL90] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York, 1990.

[RD71] G. Rozenberg and P. Doucet. On 0l-languages. *Information and Control*, 19:302–318, 1971.

[Ros69] D. J. Rosenkrantz. Programmed grammars and classes of formal languages. *Journal of the ACM*, 16:107–131, 1969.

[RS97] G. Rozenberg and A. Salomaa. *Handbook of Formal Languages*, volume 1–3. Springer, Berlin, 1997.

[RW73] R. D. Rosebrugh and D. Wood. A characterization theorem for n-parallel right linear languages. *Journal of Computer and System Sciences*, 7:579–582, 1973.

[RW75] R. D. Rosebrugh and D. Wood. Restricted parallelism and right linear grammars. *Utilitas Mathematica*, 7:151–186, 1975.

[Ryc07] L. Rychnovský. Parsing of context-sensitive languages. In *Information Systems and Formal Models (Proceedings of 2nd International Workshop on Formal Models (WFM'07))*. Silesian University, 2007.

[Sal69] A. Salomaa. *Theory of Automata*. Pergamon Press, London, 1969.

[Sal73] A. Salomaa. *Formal Languages*. Academic Press, New York, 1973.

[Sal85] A. Salomaa. *Computation and Automata*. Cambridge University Press, Cambridge, England, 1985.

[Šla01] J. Šlapal. *Metody diskrétní matematiky*. VUT Brno, FSI, Technická 2, Brno, 2001.

[Sol] S. H. von Solms. Modelling the growth of simple biological organisms using formal language theory. *Manuscript*.

[SSS87] S. Sippu and E. Soisalon-Soininen. *Parsing Theory*. Springer-Verlag, New York, 1987.

[SSS90] S. Sippu and E. Soisalon-Soininen. *Parsing Theory II: LR(k) and LL(k) Parsing (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, 1990.

[Sud96] T. A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Addison-Wesley, Reading, Massachusetts, USA, 2nd edition, 1996.

[Wal70] A. P. J. van der Walt. Random context grammars. In *Proceedings of the Symposium on Formal Languages*, Oberwolfach, 1970.

[Weg72] P. Wegner. Programming language semantics. In R. Rustin, editor, *Formal Semantics of Programming Languages*, pages 149–248. Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

[Wol96] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, Inc., 1996.

[Woo75] D. Wood. m-parallel n-right linear simple matrix languages. *Utilitas Mathematica*, 8:3–28, 1975.

[Woo76] D. Wood. Iterated a-ngsm maps and γ systems. *Information and Control*, 32(1), 1976.

[Woo84] D. Wood. *Paradigms and Programming with PASCAL.* Computer Science Press, Rockville, Maryland, 1984.

Název	Rewriting Systems with Restricted Configurations
Autor	Ing. Zbyněk Křivka, Ph.D.

Vydavatel	Vysoké učení technické v Brně Fakulta informačních technologií
Obálka	Mgr. Dagmar Hejduková
Tisk	MJ servis, spol. s r.o.
Vyšlo	Brno, 2008
Vydání	první

Tato publikace neprošla redakční ani jazykovou úpravou.