Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Peter Černo

Clearing Restarting Automata

Kabinet software a výuky informatiky

Vedoucí diplomové práce: RNDr. František Mráz, CSc.
Studijní program: Informatika, Teoretická informatika (ITI)

Praha, 2010

Poděkování[1]

Chtěl bych se poděkovat panu RNDr. Františkovi Mrázovi, CSc. za jeho čas a veškeré rady a doporučení, které vedly k napsání téhle práce.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 7. srpna 2010 Peter Černo

---

[1] Tahle práce byla částečně podporována Grantovou agenturou České republiky pod granty s číslem P103/10/0783 a P202/10/1333.

Charles University in Prague
Faculty of Mathematics and Physics

DIPLOMA THESIS

Peter Černo

Clearing Restarting Automata

Department of Software and Computer Science Education

Supervised by RNDr. František Mráz, CSc.
Study program: Informatics, Theoretical Computer Science (ITI)

Prague, 2010

Acknowledgements [2]

I would like to thank RNDr. František Mráz, CSc. for his time and all advices and suggestions that led to writing this thesis.

I declare that I have written all of the thesis on my own with the exceptions explicitly mentioned, and that I cited all used sources of information. I agree with public availability and lending of the thesis.

In Prague August 7, 2010 Peter Černo

---

[2] This work was partially supported by the Grant Agency of the Czech Republic under Grant-No. P103/10/0783 and P202/10/1333.

Contents

Preface **xiii**

1 Introduction to Formal Languages **1**
1.1 Words and Languages 2
1.2 Regular Languages 6
1.2.1 Finite Automata 6
1.2.2 Regular Expressions 8
1.2.3 Properties of Regular Languages 10
1.3 Context-Free Languages 11
1.3.1 Grammars 12
1.3.2 Normal Forms 14
1.3.3 Pushdown Machines 15
1.3.4 Subfamilies 19
1.4 Chomsky Hierarchy 20
1.4.1 Phrase-Structure Grammars 22
1.4.2 Context-Sensitive Grammars 22
1.4.3 Closure properties 24

2 Selected Models of Formal Languages **25**
2.1 Marcus Contextual Grammars 25
2.2 Pure Grammars 28
2.3 Church-Rosser String Rewriting Systems 29
2.4 Associative Language Description 32
2.5 Restarting Automata 35

3 Clearing Restarting Automata **43**
3.1 Context Rewriting Systems 45
3.2 Clearing Restarting Automata 47
3.3 Clearing restarting automata and regular languages 50

3.4 (Non-)closure properties of $\mathcal{L}(\mathsf{cl\text{-}RA})$ 52
3.5 A hierarchy with respect to the length of contexts 53
3.6 4-cl-RA-automata recognizing non-context-free language . . . 55
3.7 1-cl-RA-automata 61
3.8 2-cl-RA-automata recognizing non-context-free language . . . 63
3.9 3-cl-RA-automata recognizing non-context-free language . . . 68

4 Extended Clearing Restarting Automata **75**
4.1 Greibach's hardest context-free language 75
4.2 Δ-Clearing Restarting Automata 77
4.3 Δ^*-Clearing Restarting Automata 82

Conclusion **89**

Bibliography **91**

Název práce: Clearing Restarting Automata
Autor: Peter Černo
Katedra (ústav): Kabinet software a výuky informatiky (KSVI)
Vedoucí diplomové práce: RNDr. František Mráz, CSc.
e-mail vedoucího: Frantisek.Mraz@mff.cuni.cz

Abstrakt: Restartovací automaty byly navrženy jako model pro redukční analýzu, která představuje lingvisticky motivovanou metodu pro kontrolu korektnosti věty. Cílem práce je studovat omezenější modely restartovacích automatů, které smí vymazat podřetězec nebo jej nahradit speciálním pomocným symbolem, jenom na základě omezeného lokálního kontextu tohoto podřetězce. Tyto restartovací automaty se nazývají clearing restarting automata. V práci jsou taktéž zkoumány uzávěrové vlastnosti těchto automatů, jejich vztah k Chomskeho hierarchii a možnosti učení těchto automatů na základě pozitivních a negativních příkladů.
Klíčová slova: redukční analýza, restartovací automaty, formální jazyky, gramatická inference

Title: Clearing Restarting Automata
Author: Peter Černo
Department: Department of Software and Computer Science Education
Supervisor: RNDr. František Mráz, CSc.
Supervisor's e-mail address: Frantisek.Mraz@mff.cuni.cz

Abstract: Restarting automata were introduced as a model for analysis by reduction which is a linguistically motivated method for checking correctness of a sentence. The goal of the thesis is to study more restricted models of restarting automata which based on a limited context can either delete a substring of the current content of its tape or replace a substring by a special symbol, which cannot be overwritten anymore, but it can be deleted later. Such restarting automata are called clearing restarting automata. The thesis investigates the properties of clearing restarting automata, their relation to Chomsky hierarchy and possibilities for machine learning of such automata from positive and negative samples.
Keywords: analysis by reduction, clearing restarting automata, formal languages, grammatical inference

Preface

Analysis by reduction [16, 19] consists of stepwise simplifications (reductions) of a given (lexically disambiguated) extended sentence until a correct simple sentence is obtained, which is then accepted, or until an error is found and the input is rejected. Each simplification replaces a short part of the sentence by an even shorter one. It can be modelled by restarting automata [13, 27]. In this thesis, we propose a new restricted version of restarting automata called *clearing restarting automata*. This model is motivated by the need for simpler definitions of such automata and simultaneously by aiming for efficient machine learning of such automata. Analysis by reduction in its simplest form does not use auxiliary symbols. This is similar to several previous generative and analytical models of languages like contextual grammars by Solomon Marcus [20], pure grammars in Maurer et al. [24], Church-Rosser string rewriting systems [25], associative language description in Cherubini et al. [7], and others.

Marcus' contextual grammars are based on adjoining (inserting) pairs of strings/contexts into a word according to a selection procedure. The selection procedure controls which strings (contexts) can be inserted in which places of a string. As the selection procedure can be of arbitrary complexity [28], many language classes generated by contextual grammars are of high complexity and without practical applications.

Pure grammars [24] are similar to Chomsky grammars, but they do not use auxiliary symbols – nonterminals. Both sides of their productions are terminal strings only. There were considered mainly grammars with length increasing rules which correspond to the length-decreasing reductions used by restarting automata. In the case when the left-hand sides of the rules are single letters only, such grammars generate only a subset of the class of context-free languages (CFL), while in general they can generate a subset of the class of context-sensitive languages (but not all of CFL). Our base model of clearing restarting automata differs from both these types of pure grammars as it can reduce a subword to an empty word only.

A Church-Rosser language is a set of words which can be reduced to a special auxiliary symbol Y (it stands for 'Yes') by applying rules of a Church-Rosser string rewriting system. Such rules are length reducing. The rewriting system is confluent and Noetherian which guarantees that each maximal sequence of reductions according to such rules ends with the same irreducible string. However, Church-Rosser string rewriting systems use auxiliary symbols which we would like to avoid.

Associative language description is based on locally testable properties only. The expressive power of associative language description does not exceed the expressive power of context-free languages. It works on so-called stencil trees which are similar to derivation trees but without nonterminals. The inner nodes of stencil trees are marked by a special auxiliary symbol Δ. The correctness of a stencil tree is checked according to a finite set of rules, which specify what subtrees can appear in a correct input stencil tree. The immediate drawback of this definition is that it requires building such a stencil tree for the analyzed input word.

Our base model of clearing restarting automata also uses only the terminal symbols, which are present in an input word, and its extended version uses only one auxiliary symbol. It uses a set of rules for iterated local simplifications until the input word is reduced into the empty word – in which case the word is accepted – or a nonempty word that cannot be simplified anymore is obtained – in which case the word is rejected.

While still being nondeterministic, clearing restarting automata can recognize some non-context-free languages (even without the use of auxiliary symbols). Additionally, we have an extremely simple scheme for learning such automata from samples of reductions.

Clearing restarting automata do not use states. This is similar to stateless restarting automata introduced by Kutrib et al. in [18]. However, a stateless restarting automaton still has the ability to rewrite a subword on its tape by a shorter subword or in its weakest version to delete a scattered subword of its tape. Moreover, a stateless restarting automaton scans its tape from the left to the right and it can check that all the subwords seen in its scanning window are from a finite set of words. Our models of restarting automata can delete only whole subwords, or in the stronger version they can also rewrite a continuous subword by a special symbol Δ.

In Chapter 1 we give a short introduction to the theory of automata and formal languages and fix the notation for all subsequent chapters. The main source for this chapter is Rozenberg and Salomaa [30], Hopcroft et al. [12], and the free encyclopedia Wikipedia (`http://en.wikipedia.org/`).

The following Chapter 2 gives an overview of several selected models related to our model of clearing restarting automata. These selected models include contextual grammars by Solomon Marcus [20] in Section 2.1, pure grammars by Maurer et al. [24] in Section 2.2, Church-Rosser string rewriting systems [25] in Section 2.3, associative language description by Cherubini et al. [7] in Section 2.4, and restarting automata [13, 27] in Section 2.5. We omit the proofs and mention only the most important results.

The core chapter of the thesis is Chapter 3 where we introduce the model of clearing restarting automata. In this chapter we compare the class of languages recognized by these automata to the Chomsky hierarchy. We show that the class of languages recognized by clearing restarting automata, while being a proper subset of the class of context-sensitive languages, is incomparable to CFL with respect to inclusion. We study what is the minimal size of alphabet and context used by clearing restarting automata in order to be able to recognize a non-context-free language.

As clearing restarting automata cannot recognize all context-free languages, in Chapter 4 we introduce two extended versions – the so-called Δ-clearing and Δ^*-clearing restarting automata. Both of them can use a single auxiliary symbol Δ only. Δ-clearing restarting automata can leave a mark – a symbol Δ – at the place of deleting besides rewriting into the empty word λ. This model can recognize Greibach's hardest context-free language H ([10]), from which all context-free languages can be obtained by using a proper inverse homomorphism. Δ^*-clearing restarting automata can rewrite a subword w into Δ^k where k is bounded from above by the length of w. This is enough to enable this model to recognize all context-free languages.

The last concluding chapter contains some remarks and problems for further research.

Chapter 1

Introduction to Formal Languages

In this chapter we give a short survey of the theory of automata and formal languages and introduce the notation for all subsequent chapters. We restrict ourselves only to the most elementary definitions and theorems relevant to the subject of this thesis.

In Section 1.1 (according to Mateescu and Salomaa [23]) we want to give the reader some preliminary views about the very basics of the formal language theory, about words and languages. In Section 1.2 (using Sheng Yu [31]) we focus on regular languages. We describe several types of finite automata, introduce regular expression, and mention some standard properties of regular languages. In Section 1.3 we investigate context-free languages, grammars, and their normal forms (we follow Autebert et al. [1]). We also define pushdown automata as a device for accepting context-free languages, and mention some of the most important subfamilies of this family. The last Section 1.4 (according to Mateescu and Salomaa [22]) is devoted to Chomsky hierarchy of grammars and languages. We cover some properties of type 0 (phrase-structure) and type 1 (context-sensitive) grammars and languages.

Most of the material used in this chapter comes from Rozenberg and Salomaa [30], Hopcroft et al. [12], and the free encyclopedia Wikipedia (`http://en.wikipedia.org/`).

1.1 Words and Languages

An *alphabet* is a finite nonempty set. The elements of an alphabet Σ are called *letters* or *symbols*. A *word* or *string* over an alphabet Σ is a finite sequence consisting of zero or more letters of Σ, whereby the same letter may occur several times. The sequence of zero letters is called the *empty word*, written λ. The set of all words (all nonempty words, respectively) over an alphabet Σ is denoted by Σ^* (Σ^+, respectively). If x and y are words over Σ, then so is their *catenation* (or *concatenation*) xy (or $x \cdot y$), obtained by juxtaposition, that is, writing x and y one after another. Catenation is an associative operation and the empty word λ acts as an identity: $w\lambda = \lambda w = w$ holds for all words w. Because of the associativity, we may use the notation w^i in the usual way. By definition, $w^0 = \lambda$.

Let u be a word in Σ^*, say $u = a_1 \dots a_n$ with $a_i \in \Sigma$. We use $u[i]$ to denote the ith letter of u, i.e. $u[i] = a_i$. We say that n is the *length* of u and we write $|u| = n$. The sets of all words over Σ of length k, or at most k, are denoted by Σ^k and $\Sigma^{\leq k}$, respectively. By $|u|_a$, for $a \in \Sigma$, we denote the total number of occurrences of the letter a in u. The *reversal* (*mirror image*) of u, denoted u^R, is the word $u_n \dots u_1$. Finally a *factorization* of u is any sequence u_1, ..., u_t of words such that $u = u_1 \cdots u_t$.

For a pair u, v of words we define four relations:

1. u is a *prefix* of v, if there exists a word z such that $v = uz$,
2. u is a *suffix* of v, if there exists a word z such that $v = zu$, and
3. u is a *factor* (or *subword*) of v, if there exist words z and z' such that $v = zuz'$.
4. u is a *scattered subword* of v, if v as a sequence of letters contains u as a subsequence, i.e. there exist words $z_1, \dots, z_t$ and $y_0, \dots, y_t$ such that $u = z_1 \dots z_t$ and $v = y_0 z_1 y_1 \dots z_t y_t$.

Observe that u itself and λ are subwords, prefixes and suffixes of u. Other subwords, prefixes and suffixes are called *nontrivial*.

$Pref_k(u)$ denotes either the (nontrivial) prefix of length k of the word u in case $|u| > k$, or the whole of u in case $|u| \leq k$. Similarly, $Suff_k(u)$ denotes either the (nontrivial) suffix of length k of the word u in case $|u| > k$, or the whole of u in case $|u| \leq k$. The set of all subwords of length k of u that occur in u in a position other than the prefix or suffix is denoted $Int_k(u)$ (interior words).

The *shuffle* operation between words, denoted $⧢$, is defined recursively by

$$(au ⧢ bv) = a(u ⧢ bv) \cup b(au ⧢ v),$$

and

$$(u ⧢ \lambda) = (\lambda ⧢ u) = \{u\},$$

where $u, v \in \Sigma^*$ and $a, b \in \Sigma$. Obviously,

$$(u ⧢ v) = \{x_1 y_1 \ldots x_n y_n \mid u = x_1 \ldots x_n, v = y_1 \ldots y_n, n \geq 1\}.$$

We now proceed from words to languages. Subsets, finite or infinite, of Σ^* are referred to as (*formal*) *languages* over Σ. Thus $L_1 = \{\lambda, 0, 111, 1001\}$ and $L_2 = \{0^p \mid p \text{ prime}\}$ are languages over the *binary alphabet* $\{0, 1\}$. A finite language can always, at least in principle, be defined as L_1 above: by listing all of its words. Such a procedure is not possible for infinite languages. Some finitary specification other than simple listing is needed to define an infinite language. Much of language theory deals with such finitary specifications of infinite languages: grammars, automata etc.

Regarding languages as sets, we may define the *Boolean* operations of *union*, *intersection* and *complementation* in the usual fashion. The operation of *catenation* (or *concatenation*) is extended to concern languages in the natural way:

$$L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}.$$

The notation L^i is extended to concern languages, now $L^0 = \{\lambda\}$. The *catenation closure* or *Kleene star* (*Kleene plus*, respectively) of a language L, in symbols L^* (L^+, respectively) is defined to be the union of all nonnegative powers of L (of all positive powers of L, respectively). Observe that this definition is in accordance with our earlier notations Σ^* and Σ^+ if we understand Σ as the finite language whose words are the singleton letters. The *reversal* (*mirror image*) of a language L, denoted L^R, is the set

$$L^R = \{w^R \mid w \in L\}.$$

For each $x \in \Sigma^*$, the *left-derivate* of L with respect to $x \in \Sigma^*$ is the set

$$\delta_x(L) = x \backslash L = \{y \in \Sigma^* \mid xy \in L\},$$

and for a language $L_0 \subseteq \Sigma^*$, the *left-quotient* of L by L_0 is the set

$$L_0 \backslash L = \bigcup_{x \in L_0} x \backslash L = \{y \in \Sigma^* \mid xy \in L, x \in L_0\}.$$

Similarly, the *right-derivate* of L with respect to $x \in \Sigma^*$ is the set

$$\delta^x(L) = L/x = \{y \in \Sigma^* \mid yx \in L\},$$

and the *right-quotient* of L by a language $L_0 \subseteq \Sigma^*$ is

$$L/L_0 = \bigcup_{x \in L_0} L/x = \{y \in \Sigma^* \mid yx \in L, x \in L_0\}.$$

It is clear by the above definition that

$$(L_1 \backslash L) \cup (L_2 \backslash L) = (L_1 \cup L_2) \backslash L$$

for any $L_1, L_2 \subseteq \Sigma^*$. Similar equality holds for right-quotient of languages.

Also the shuffle operation is extended in the natural to languages:

$$(L_1 \sqcup\!\sqcup L_2) = \bigcup_{u \in L_1, v \in L_2} (u \sqcup\!\sqcup v).$$

The operations of union, catenation, and Kleene star are referred as *regular* operations. A language L over Σ is termed *regular* if L can be obtained from the "atomic" languages $\emptyset$ (the empty language) and $\{a\}$, where a is a letter of Σ, by applying regular operations finitely many times. By applying union and catenation, we obviously get all finite languages (Fin). The star operation is needed to produce infinite languages. In fact, for any k, there are regular languages that cannot be represented with fewer than k nested star operations. When we speak of the *family* of regular languages, we mean all languages that are regular over some Σ, that is, the alphabet may vary from language to language. This family is very central in formal language theory. It corresponds to strictly finite computing devices, finite automata. The family has many desirable formal properties. It is closed under most of the usual operations for languages, in particular, under all Boolean operations.

A language L over Σ is *star-free* if L can be obtained from the atomic languages $\{a\}$, where $a \in \Sigma$, by applying Boolean operations (complement is taken with respect to Σ^*) and catenation finitely many times. Apparently every star-free language is regular.

We say that a language L is *noncounting* if there is an integer n such that, for all words x, y, z,

$$xy^n z \in L \Leftrightarrow xy^{n+1} z \in L.$$

It can be proved that every star-free language is noncounting. The converse holds in the form: every regular noncounting language is star-free.

A special subclass of star-free languages which has attracted much attention is the *locally testable languages.* Informally, for a locally testable language L, one can decide whether a word w is in L by looking at all subwords of w of a previously given length k. Formally, a language $L \subseteq \Sigma^*$ is said to be k-*testable* if, for any words $x, y \in \Sigma^*$, $|x|, |y| \geq k$, the conditions $\mathit{Pref}_k(x) = \mathit{Pref}_k(y)$, $\mathit{Suff}_k(x) = \mathit{Suff}_k(y)$, and $\mathit{Int}_k(x) = \mathit{Int}_k(y)$ imply that $x \in L \Leftrightarrow y \in L$. A language is said to be *locally testable* if it is k-testable for some integer $k \geq 1$.

Let Σ and Γ be two finite alphabets. A mapping $h : \Sigma^* \to \Gamma^*$ is called a *morphism* if $h(xy) = h(x)h(y)$ for all $x, y \in \Sigma^*$. In particular, $h(\lambda) = \lambda$.

For a set S, let 2^S (or $\mathcal{P}(S)$) denote the *power set* of S, i.e. the collection of all subsets of S. A mapping $\phi : \Sigma^* \to 2^{\Gamma^*}$ is called a *substitution* if $\phi(\lambda) = \{\lambda\}$ and $\phi(xy) = \phi(x)\phi(y)$ for all $x, y \in \Sigma^*$.

Clearly, a morphism is a special kind of substitution where each word is associated with a singleton set. Morphisms and substitutions are usually defined by specifying only the image of each letter in Σ under the mapping. We extend the definitions of h and ϕ, respectively, to define

$$h(L) = \{h(w) \mid w \in L\}$$

and

$$\phi(L) = \bigcup_{w \in L} \phi(w)$$

for $L \subseteq \Sigma^*$.

A morphism $h : \Sigma^* \to \Gamma^*$ is said to be λ-free if $h(a) \neq \lambda$ for all $a \in \Sigma$. A substitution $\phi : \Sigma^* \to 2^{\Gamma^*}$ is said to be λ-free if $\lambda \notin \phi(a)$ for all $a \in \Sigma$. And ϕ is called a *finite substitution* if, for each $a \in \Sigma$, $\phi(a)$ is a finite subset of Γ^*. Similarly, ϕ is called a *regular substitution* if, for each $a \in \Sigma$, $\phi(a)$ is a regular language over Γ.

Let $h : \Sigma^* \to \Gamma^*$ be a morphism. The *inverse* of the morphism h is a mapping $h^{-1} : \Gamma^* \to 2^{\Sigma^*}$ defined by, for each $y \in \Gamma^*$,

$$h^{-1}(y) = \{x \in \Sigma^* \mid h(x) = y\}.$$

Similarly, for a substitution $\phi : \Sigma^* \to 2^{\Gamma^*}$, the *inverse* of the substitution ϕ is a mapping $\phi^{-1} : \Gamma^* \to 2^{\Sigma^*}$ defined by, for each $y \in \Gamma^*$,

$$\phi^{-1}(y) = \{x \in \Sigma^* \mid y \in \phi(x)\}.$$

1.2 Regular Languages

In formal language theory in general, there are two major types of mechanisms for defining languages: acceptors and generators. For regular languages in particular, the acceptors are finite automata and the generators are regular expressions and right (left) linear grammars (see Section 1.3.4).

In Section 1.2.1 we describe several types of finite automata, all of which accept exactly the family of regular languages. In Section 1.2.2 we introduce regular expressions and their relationship to finite automata. The last Section 1.2.3 gives some properties of regular languages.

Most of the material used in this Section is taken from Sheng Yu [31].

1.2.1 Finite Automata

Regular languages and finite automata have had a wide range of applications. Their most celebrated application has been lexical analysis in programming language compilation and user-interface translations. Other notable applications include circuit design, text editing, and pattern matching.

A finite automaton (FA) consists of a finite set of internal states and a set of rules that govern the change of the current state when reading a given input symbol. If the next state is always uniquely determined by the current state and the current input symbol, we say that the automaton is deterministic. Formally, we define a deterministic finite automaton as follows:

A *deterministic finite automaton* (DFA) A is a quintuple $(Q, \Sigma, \delta, s, F)$, where:

Q is the finite set of *states*,

Σ is the *input alphabet*,

$\delta : Q \times \Sigma \to Q$ is the *state transition function*,

$s \in Q$ is the *starting state*, and

$F \subseteq Q$ is the set of *final states.*

Note that, in general, we do not require the transition function δ to be total, i.e. to be defined for every pair in $Q \times \Sigma$. If δ is total, then we call A a *complete* DFA.

A DFA such that every state is reachable from the starting state and reaches a final state is called a *reduced* DFA. A reduced DFA may not be a complete DFA.

A *configuration* of $A = (Q, \Sigma, \delta, s, F)$ is a word in $Q\Sigma^*$, i.e. a state $q \in Q$ followed by a word $x \in \Sigma^*$ where q is the current state of A and x is the remaining part of the input. The *starting configuration* of A for an input word $x \in \Sigma^*$ is sx. *Accepting configuration* are defined to be elements of F (followed by the empty word λ).

A computation step of A is a transition from a configuration α to a configuration β, denoted by $\alpha \vdash_A \beta$, where $\vdash_A$ is a binary relation on the set of configurations of A. The relation $\vdash_A$ is defined by: for $px, qy \in Q\Sigma^*$, $px \vdash_A qy$ if $x = ay$ for some $a \in \Sigma$ and $\delta(p, a) = q$. We use $\vdash$ instead of $\vdash_A$ if there is no confusion. The transitive closure and the reflexive and transitive closure of $\vdash$ are denoted $\vdash^+$ and $\vdash^*$, respectively.

The language accepted by a DFA $A = (Q, \Sigma, \delta, s, F)$, denoted $L(A)$, is defined as follows:

$$L(A) = \{w \mid sw \vdash^* f \text{ for some } f \in F\}.$$

For convenience, we define the extension of δ, $\delta^* : Q \times \Sigma^* \to Q$, inductively as follows. We set $\delta^*(q, \lambda) = q$ and $\delta^*(q, xa) = \delta(\delta^*(q, x), a)$, for $q \in Q$, $a \in \Sigma$. Then we can also write:

$$L(A) = \{w \mid \delta^*(s, w) = f \text{ for some } f \in F\}.$$

Nondeterministic finite automata (NFA) are a generalization of DFA where, for a given state and an input symbol, the number of possible transitions can be greater than one. Formally, a *nondeterministic finite automaton* A is a quintuple $(Q, \Sigma, \delta, s, F)$ where Q, Σ, s, and F are defined exactly the same way as for a DFA, and $\delta : Q \times \Sigma \to 2^Q$ is the transition function.

The computation relation $\vdash_A$: $Q\Sigma^* \times Q\Sigma^*$ of an NFA A is defined by setting $px \vdash_A qy$ if $x = ay$ and $q \in \delta(p, a)$, for $p, q \in Q$, $x, y \in \Sigma^*$, and $a \in \Sigma$. The language accepted by A is defined in the same way as for DFA.

NFA can be further generalized to have state transitions without reading any input symbol. Such transitions are called λ-transitions in the following definition.

A *nondeterministic finite automaton with λ-transitions* (λ-NFA) A is a quintuple $(Q, \Sigma, \delta, s, F)$ where Q, Σ, s, and F are the same as for an NFA; and $\delta : Q \times (\Sigma \cup \{\lambda\}) \to 2^Q$ is the transition function.

For a λ-NFA $A = (Q, \Sigma, \delta, s, F)$, the binary relation $\vdash_A$: $Q\Sigma^* \times Q\Sigma^*$ is defined by that $px \vdash_A qy$, for $p, q \in Q$ and $x, y \in \Sigma^*$, if $x = ay$ and $q \in \delta(p, a)$ or if $x = y$ and $q \in \delta(p, \lambda)$. The language accepted by A is again defined in the same way as for DFA.

Let $A = (Q, \Sigma, \delta, s, F)$ be a λ-NFA. The *λ-closure* of a state $q \in Q$, denoted by λ-closure(q), is the set of all states that are reachable from q by zero or more λ-transitions, i.e.

$$\lambda\text{-closure}(q) = \{q \in Q \mid q \vdash_A^* p\}.$$

For a λ-NFA $A = (Q, \Sigma, \delta, s, F)$ let $A' = (Q, \Sigma, \delta', s, F')$ be an NFA, such that for each $q \in Q$ and $a \in \Sigma$,

$$\delta'(p, a) = \delta(p, a) \cup \bigcup_{q \in \lambda\text{-closure}(p)} \delta(q, a)$$

and

$$F' = \{q \in Q \mid \lambda\text{-closure}(q) \cap F \neq \emptyset\}.$$

It can be easily verified that $L(A) = L(A')$.

Another form of generalization of NFA is defined in the following.

A NFA *with nondeterministic starting state* (NNFA) $A = (Q, \Sigma, \delta, S, F)$ is an NFA except that there is a set of starting states S rather than exactly one starting state. Thus, for an input word, the computation of A starts from a nondeterministically chosen starting state.

It can be shown that for each NNFA A of n states, we can construct an equivalent DFA A' of at most 2^n states.

There are also several models of finite automata with output, which include Moore machines, Mealy machine, and finite transducers.

1.2.2 Regular Expressions

In the previous section we have described various forms of finite automata. These automata can be easily implemented by computer programs. However, finite automata in any of the above mentioned forms are not convenient to be specified manually by users. A succinct and comprehensible expression in sequential form would be better suited than a finite automaton definition. Such expressions are regular expressions which are often used as user interfaces for specifying regular languages.

We define, inductively, a *regular expression* e over an alphabet Σ and the language $L(e)$ it denotes as follows:

1. $e = \emptyset$ is a regular expression denoting the language $L(e) = \emptyset$.
2. $e = \lambda$ is a regular expression denoting the language $L(e) = \{\lambda\}$.
3. $e = a$, $a \in \Sigma$, is a regular expression denoting the language $L(e) = \{a\}$.

   Let e_1 and e_2 be regular expressions and $L(e_1)$ and $L(e_2)$ be the languages they denote, respectively. Then
4. $e = (e_1 + e_2)$ is a regular expression denoting the language $L(e) = L(e_1) \cup L(e_2)$.
5. $e = (e_1 \cdot e_2)$ is a regular expression denoting the language $L(e) = L(e_1)L(e_2)$.
6. $e = e_1^*$ is a regular expression denoting the language $(L(e_1))^*$.

We assume that $*$ has higher precedence than $\cdot$ and $+$, and $\cdot$ has higher precedence than $+$. A pair of parentheses may be omitted whenever the omission would not cause any confusion. Also, we usually omit the symbol $\cdot$ in regular expressions.

There are three major approaches for transforming regular expressions into finite automata. The first approach is to transform a regular expression into a λ-NFA. This approach is simple and intuitive, but may generate many λ-transitions. The second approach transforms a regular expression into an NFA without λ-transitions. The third and most involved approach is to transform a regular expression directly into an equivalent DFA.

The converse is also possible, i.e. for a given finite automaton A, we can construct a regular expression e such that e denotes the language accepted by A. The construction uses *extended finite automata* where a transition between a pair of states is labeled by a regular expression. For a given finite automaton, a so-called *state elimination technique* deletes a state at each step and changes the transitions accordingly. This process continues until the FA contains only the starting state, a final state, and the transition between them. The regular expression labeling the transition specifies exactly the language accepted by A.

1.2.3 Properties of Regular Languages

There are many ways to show that a language is regular; for example, this can be done by demonstrating that the language is accepted by a finite automaton, specified by a regular expression, or generated by a right-linear grammar. To prove that a language is not regular, the most commonly used tools are the *pumping properties* of regular languages, which are usually stated as "pumping lemmas". The term "pumping" intuitively describes the property that any sufficiently long word of the language has a nonempty subword that can be "pumped", i.e. if the subword is replaced by an arbitrary number of copies of the same subword, the resulting word is still in the language.

There are many versions of pumping lemmas for regular languages. The "standard" version is a necessary but not sufficient condition for regularity.

Lemma 1.2.1. *Let L be a regular language over Σ. Then there is a constant n, depending on L, such that for each $w \in L$ with $|w| \geq n$ there exist $x, y, z \in \Sigma^*$ such that $w = xyz$ and*

1. $|xy| \leq n$,
2. $|y| \geq 1$,
3. $xy^t z \in L$ *for all* $t \geq 0$.

The proof is based on the observation that if an n-state DFA reads a word w of length $|w| \geq n$, then it passes through the sequence of $|w|+1 > n$ states, thus (by the pigeonhole principle) at least two states in this sequence must be equal.

In the following theorem we summarize some selected closure properties of regular languages.

Theorem 1.2.1. *The family of regular languages is closed under the following operations (we assume that $L \subseteq \Sigma^*$):*

1. *Boolean operations: union, intersection, complementation,*
2. *catenation, Kleene star, reversal,*
3. *left and right-quotient by an arbitrary language,*
4. *prefix*$(L) = \{x \in \Sigma^* \mid xy \in L, y \in \Sigma^*\}$,

5. *suffix*$(L) = \{y \in \Sigma^* \mid xy \in L, x \in \Sigma^*\}$,

6. *infix*$(L) = \{y \in \Sigma^* \mid xyz \in L, x, z \in \Sigma^*\}$,

7. *morphism, inverse morphism,*

8. *finite substitution, inverse finite substitution,*

9. *regular substitution, inverse regular substitution.*

Finally, we characterize the family of regular languages by a so-called Myhill-Nerode Theorem. Let $L \subseteq \Sigma^*$ and $x \in \Sigma^*$. The *derivate* of L with respect to x, denoted $D_x L$, is

$$D_x L = x \backslash L = \{y \in \Sigma^* \mid xy \in L\}.$$

For $L \subseteq \Sigma^*$, we define a relation $\equiv_L \subseteq \Sigma^* \times \Sigma^*$ by

$$x \equiv_L y \Leftrightarrow D_x L = D_y L$$

for each pair $x, y \in \Sigma^*$. Clearly, $\equiv_L$ is an equivalence relation. It partitions Σ^* into equivalence classes. The number of equivalence classes of $\equiv_L$ is called the *index* of $\equiv_L$. The following Myhill-Nerode Theorem characterizes the family of regular languages.

Theorem 1.2.2. *A language* $L \subseteq \Sigma^*$ *is regular if and only if* $\equiv_L$ *has a finite index.*

If L is a regular language, then it can be shown that the minimal number of states of a complete DFA that accepts L is equal to the index of $\equiv_L$.

1.3 Context-Free Languages

This section is devoted to context-free languages (CFL). Context-free languages and grammars were designed initially to formalize grammatical properties of natural languages. They subsequently appeared to be well adapted to the formal description of the syntax of programming languages. This led to a considerable development of the theory.

We focus on two basic tools: context-free grammars and pushdown automata. These are indeed the standard tools to generate and to recognize context-free languages. In Section 1.3.1 we give a short introduction to context-free grammars, followed by the survey of their normal forms in Section 1.3.2. In Section 1.3.3 we introduce pushdown automata as a basic accepting device for context-free languages. In the last Section 1.3.4 we present some subfamilies of context-free languages.

The main source for this Section is Autebert et al. [1].

1.3.1 Grammars

A *context-free grammar* G is a quadruple $G = (V_N, V_T, S, P)$, where V_N is disjoint from V_T and:

V_N is the finite nonempty set of *nonterminals* or *variables*,

V_T is the finite nonempty set of *terminals* or *terminal letters*,

$S \in V_N$ is the distinguished nonterminal called the *axiom*,

$P \subseteq V_N \times (V_N \cup V_T)^*$ is the finite set of *productions* or *derivation rules.*

Given words $u, v \in (V_N \cup V_T)^*$, we write $u \Rightarrow_G v$ whenever there exist factorizations $u = xXy$, $v = x\alpha y$, with production $(X, \alpha) \in P$. A *derivation* of length $k \geq 0$ from u to v is a sequence $(u_0, u_1, \ldots, u_k)$ of words in $(V_N \cup V_T)^*$ such that $u_{i-1} \Rightarrow_G u_i$ for $i = 1, 2, \ldots, k$, and $u = u_0, v = u_k$. If this holds, we write $u \Rightarrow_G^k v$. The existence of some derivation from u to v is denoted by $u \Rightarrow_G^* v$. If there is a proper derivation (i.e. of length ≥ 1), we use the notation $u \Rightarrow_G^+ v$. The *language generated by a nonterminal* X in the grammar G is the set

$$L_G(X) = \{w \in V_T^* \mid X \Rightarrow_G^* w\}.$$

The *language generated by the grammar* G is $L_G(S)$ and is denoted, for short, L_G or $L(G)$.

A language L is called *context-free* if it is the language generated by some context-free grammar. Two grammars G and G' are *equivalent* if they generate the same language, i.e. if $L(G) = L(G')$.

More generally, if $x \in (V_N \cup V_T)^*$, we set

$$L_G(x) = \{w \in V_T^* \mid x \Rightarrow_G^* w\}.$$

Context-freeness easily implies that $L_G(xy) = L_G(x)L_G(y)$.

Consider a derivation $u = u_0 \Rightarrow_G u_1 \Rightarrow_G \ldots \Rightarrow_G u_k = v$, with $u, v \in (V_N \cup V_T)^*$. Then there exist productions $p_i = (X_i, \alpha_i)$ and words x_i, y_i such that $u_i = x_i X_i y_i$ and $u_{i+1} = x_i \alpha_i y_i$ for $i = 0, 1, \ldots, k-1$. The derivation is *leftmost* if $|x_i| \leq |x_{i+1}|$ for $i = 0, 1, \ldots k-2$, and *rightmost* if, symmetrically, $|y_i| \leq |y_{i+1}|$ for $i = 0, 1, \ldots k-2$. It is an interesting fact that any word in a context-free language $L_G(X)$ has the same number of leftmost and of rightmost derivations. A context-free grammar G is *unambiguous* for a nonterminal X if every word in $L_G(X)$ has exactly one leftmost (rightmost) derivation. A language is *unambiguous* if there is an unambiguous grammar to generate it, otherwise it is called *inherently ambiguous*.

We define the *parse tree* or the *derivation tree* as a rooted tree which satisfies the following conditions:

1. each node is labeled by some element from $V_N \cup V_T \cup \{\lambda\}$,
2. internal nodes are labeled by nonterminals,
3. if an internal node is labeled A and its children are labeled $B_1, \ldots, B_n$ respectively, from the left, then $(A, B_1 \ldots B_n) \in P$,
4. if some node is labeled by λ, then it is the leaf and the only child of its parent.

If we look at the leaves of any parse tree and concatenate them from the left, we get a string, called the *yield* of the tree, which is always a string that is derived from the root nonterminal. It can be shown that $A \Rightarrow_G^* w$ if and only if there is a parse tree with root A and yield w.

A context-free grammar G is *trim* in the nonterminal S if the following two conditions are fulfilled:

1. for every nonterminal X, the language $L_G(X)$ is nonempty,
2. for every nonterminal X, there exist $u, v \in V_T^*$ such that $S \Rightarrow_G^* uXv$.

It is not difficult to see that a context-free grammar can always be trimmed effectively.

A production is *terminal* if its right side contains no nonterminal. A production is called a *λ-production* if its right side is the empty word. At least one λ-production is necessary if the language generated by the grammar contains the empty word. It is not too difficult to construct, for every context-free grammar G, an equivalent context-free grammar with no λ-production excepted the production (S, λ) if $\lambda \in L(G)$.

A grammar G is *proper* if it has neither λ-productions nor any production of the form (X, Y) with Y a nonterminal. Again, an equivalent proper grammar can effectively be constructed for any grammar G if $\lambda \notin L(G)$.

There are several convenient shorthands to describe context-free grammars. Usually, a production (X, α) is written $X \to \alpha$, and productions with same left side are grouped together, the corresponding right sides being separated by a $|$. Usually, the nonterminals and terminal letters are clear from the context.

Subsequently, we make use of the following notation. Let A be an alphabet. A *copy* of A is an alphabet that is disjoint from A and in bijection with A. A copy is often denoted $\bar{A}$ or A'. This implicitly means that the bijection is denoted similarly, namely as the mapping $a \mapsto \bar{a}$ or $a \mapsto a'$. The inverse bijection is denoted the same, that is $\bar{\bar{a}} = a$ ($(a')' = a$, respectively), and it is extended to a bijection from $(A \cup \bar{A})^*$ into itself by $\overline{xy} = \bar{x}\bar{y}$ (similarly for $(A \cup A')^*$).

Let A be an alphabet and let $\bar{A}$ be a copy. The *Dyck language* over $A \cup \bar{A}$ is the language D_A^* generated by S in the grammar:

$$S \to TS \mid \lambda; \quad T \to aS\bar{a} \text{ for all } a \in A.$$

If A has n letters, then the notation D_n^* is frequently used instead of D_A^*.

1.3.2 Normal Forms

In this section, we present three normal forms of context-free grammars. The two first ones are the Chomsky normal form and the Greibach normal form. They are often used to get easier proofs of results about context-free languages. The third normal form is the operator normal form. It is an example of a normal form that has been used in the syntactical analysis.

A context-free grammar $G = (V_N, V_T, S, P)$ is in *weak Chomsky normal form* if each nonterminal production has a right member in V_N^* and each terminal production has a right member in $V_T \cup \{\lambda\}$. It is in *Chomsky normal form* if it is in weak Chomsky normal form and each right member of a nonterminal production has length 2.

It can be shown, that given a context-free grammar, an equivalent context-free grammar in Chomsky normal form can effectively be constructed.

One of the consequences of Chomsky normal form is the following pumping lemma for context-free languages.

Lemma 1.3.1. *Let L be a context-free language over Σ. Then there exist natural numbers p and q, depending on L, such that for each $z \in L$ with $|z| > p$ there exist a factorization $uvwxy$ of z such that*

1. $|vwx| \leq q$,
2. $v \neq \lambda$ *or* $x \neq \lambda$,
3. $uv^iwx^iy \in L$ *for all* $i \geq 0$.

A context-free grammar $G = (V_N, V_T, S, P)$ is in *Greibach normal form* if each production of the grammar rewrites a nonterminal into a word in $V_TV_N^*$. In particular, the grammar is proper and each terminal production rewrites a nonterminal in a terminal letter.

It is in *quadratic Greibach normal form* if it is in Greibach normal form and each right member of a production of G contains at most 2 nonterminals.

It is in *double Greibach normal form* if each right member of the productions of G are in $V_TV_N^*V_T \cup V_T$. In particular, a terminal production rewrites a nonterminal in a letter or in a word of length 2.

It is in *cubic double Greibach normal form* (in *quadratic double Greibach normal form*, respectively) if it is in double Greibach normal form and each right member of a production contains at most 3 nonterminals (at most 2 nonterminals, respectively).

A proper context-free grammar G can effectively be transformed into an equivalent context-free grammar G' in any of the aforementioned Greibach normal forms.

A context-free grammar $G = (V_N, V_T, S, P)$ is in *operator normal form* if no right member of a production contains two consecutive nonterminals. This normal form has been introduced for purposes from syntactical analysis. For these grammars, an operator precedence can be defined which is inspired by the classical precedence relations of usual arithmetic operators.

Again, given a context-free grammar G, an equivalent context-free grammar G' in operator normal form can effectively be constructed.

1.3.3 Pushdown Machines

In this section, we focus on the accepting device for context-free languages, namely pushdown automaton with the important subclass induced by determinism.

A *pushdown machine* over Σ (a pdm for short) is a quadruple $M = (Q, \Sigma, Z, T)$, where

Q is the finite set of *states*,

Σ is the *input alphabet*,

Z is the *stack alphabet*,

T is a finite subset of $(\Sigma \cup \{\lambda\}) \times Q \times Z \times Z^* \times Q$, called the set of *transition rules.*

An element (y, q, z, h, q') of T is a *rule*, and if $y = \lambda$, it is called a λ-*rule*. The first three components are viewed as preconditions in the behavior of a pdm (and therefore the last two components are viewed as postconditions). T is often seen as a function from $(\Sigma \cup \{\lambda\}) \times Q \times Z$ into the subsets of $Z^* \times Q$, and we use $(h, q') \in T(y, q, z)$ as the equivalent for $(y, q, z, h, q') \in T$.

A pushdown machine is *realtime* if T is a finite subset of $\Sigma \times Q \times Z \times Z^* \times Q$, i.e. if there is no λ-rule. A realtime pdm is *simple*, if it has only one state. In this case, the state giving no information, it is omitted, and T is a subset of $A \times Z \times Z^*$.

An *internal configuration* of a pdm M is a couple $(q, h) \in Q \times Z^*$, where q is the current state, and h is the string over Z^* composed of the symbols in the stack, the first letter of h being the bottom-most symbol of the stack. A *configuration* is a triple $(x, q, h) \in \Sigma^* \times Q \times Z^*$, where x is the input word to be read, and (q, h) is an internal configuration.

The *transition relation* is a relation over configurations defined in the following way: let $c = (yg, q, wz)$ and $c' = (g, q', wh)$ be two configurations, where $y \in (\Sigma \cup \{\lambda\})$, $g \in \Sigma^*$, $q, q' \in Q$, $z \in Z$, and $w, h \in Z^*$. There is a *transition* between c and c', and we note $c \mapsto c'$, if $(y, q, z, h, q') \in T$. If $y = \lambda$, the transition is called a λ-transition, and if $y \in \Sigma$, the transition is said to involve the reading of a letter. A *valid computation* is an element of the reflexive and transitive closure of the transition relation, and we note $c \mapsto^* c'$ a valid computation from c and leading to c'. A convenient notation is to introduce, for any word $x \in \Sigma^*$, the relation on internal configurations, denoted $\xrightarrow{x}$, and defined by:

$$(q, w) \xrightarrow{x} (q', w') \Leftrightarrow (x, q, w) \mapsto^* (\lambda, q', w').$$

We clearly have: $\xrightarrow{x} \circ \xrightarrow{y} = \xrightarrow{xy}$.

An internal configuration (q', w') is *accessible* from an internal configuration (q, w), or equivalently, (q, w) is *co-accessible* from (q', w') if there is some $x \in \Sigma^*$ such that $(q, w) \xrightarrow{x} (q', w')$.

A rule $(y, q, z, h, q') \in T$ is an increasing rule (respectively a stationary, respectively a decreasing rule) if $|h| > 1$ (respectively $|h| = 1$, respectively $|h| < 1$). The use of an increasing rule (respectively a stationary, respectively a decreasing rule) in a computation increases (respectively leaves unchanged, respectively decreases) the number of symbols in the stack. A pdm is in *quadratic form* if for all rules $(y, q, z, h, q') \in T$, we have: $|h| \leq 2$.

A pdm is used as a device for recognizing words by specifying starting configurations and accepting configurations. The convention is that there is only one starting internal configuration $i = (q, z)$, where the state q is the initial state, and the letter z is the initial stack symbol. The sets of internal accepting configurations usually considered are:

1. the set $F \times Z^*$ where F is a subset of Q, called the set of *accepting states*,
2. the set $Q \times \{\lambda\}$,
3. the set $F \times \{\lambda\}$ where F is a subset of Q,
4. the set $Q \times Z^*Z'$ where Z' is a subset of Z.

We call each of these cases a *mode of acceptance*.

A *pushdown automaton* over Σ (a pda for short) is composed of a pushdown machine (Q, Σ, Z, T), together with an *initial internal configuration* i, and a set K of *internal accepting configurations*. It is so a 6-tuple $A = (Q, \Sigma, Z, i, K, T)$, and (Q, Σ, Z, T) is called the pdm *associated* to A.

A word $x \in \Sigma^*$ is *recognized* by a pda $A = (Q, \Sigma, Z, i, K, T)$ over Σ with a specified mode of acceptance if there is $k \in K$ such that $i \xrightarrow{x} k$. Considering the modes of acceptance defined above, in the first case the word is said to be recognized by *accepting states* F, in the second case the word is said to be recognized by *empty storage*, in the third case the word is said to be recognized by *empty storage and accepting states* F, and in the last case the word is said to be recognized by *topmost stack symbols* Z'. The *language accepted* by a pda with a given mode of acceptance is the set of all words recognized by the pda with this mode. For any pda $A = (Q, \Sigma, Z, i, K, T)$, we note $L(A)$ the language recognized by A, and for any set of internal accepting configurations K', we note $L(A, K')$ the language recognized by the pda $A' = (Q, \Sigma, Z, i, K', T)$.

In general, for a given pda, changing the mode of acceptance changes the languages recognized. Nevertheless, the family of languages that are recognized by pda's, using any of these modes remains the same. Moreover, the family of languages recognized by pda's using any of the aforementioned modes of acceptance is exactly the family of context-free languages.

The characterization of context-free languages in terms of languages recognized by pda's allows much simpler proofs of certain properties of context-free languages.

A pushdown automaton is *realtime* (*simple*, respectively) if the associated pdm is realtime (simple, respectively).

The fact that any proper context-free language can be generated by a context-free grammar in Greibach normal form implies that realtime pda's, and even simple pda's, recognize exactly proper context-free languages.

A pdm $M = (Q, \Sigma, Z, T)$ is *deterministic* if the set T of transitions satisfies the following conditions for all $(y, q, z) \in (\Sigma \cup \{\lambda\}) \times Q \times Z$:

1. $|T(y, q, z)| \leq 1$
2. $T(\lambda, q, z) \neq \emptyset \Rightarrow \forall a \in \Sigma : T(a, q, z) = \emptyset$

A *deterministic* pda (dpda for short) is a pda with a deterministic associated pdm.

It is possible to prove that the family of languages recognized by dpda's by empty storage is the same as the family of languages recognized by dpda's by empty storage and accepting states, and that this family is included in the family of languages recognized by dpda's by accepting states. On the other hand, it is easy to verify that a language recognized by empty storage by dpda is prefix, i.e. no proper prefix of a word of this language belongs to this language. So, we are left with two families of languages: the family of languages recognized by accepting states, called the family of *deterministic languages* (DCFL), and the family of languages recognized by empty storage, called the family of *deterministic-prefix languages*. It is easy to check that the family of deterministic-prefix languages is exactly the family of deterministic languages that are prefix languages.

The two families are distinct. As an example, the language $L_1 = \{a^n b^p \mid p > n > 0\}$ is deterministic but not prefix. To avoid these problems, a usual trick is to consider languages with an end marker: indeed, $L\sharp$ is a prefix language which is deterministic if and only if L is deterministic.

For any dpda it is possible to construct a dpda recognizing the same language such that an accepting state cannot be on the left side of an λ-rule. Consequently, in such a dpda, for any recognized word, there is only one successful computation. This proves that deterministic languages are unambiguous. Moreover, the family of deterministic languages is closed under complementation. This property does not hold for the family of context-free languages.

1.3.4 Subfamilies

We present here some subfamilies of the family of context-free languages. We begin with the probably most classical one, namely the family of linear languages. The simplest way to define the family of linear languages is by grammars. A context-free grammar $G = (V_N, V_T, S, P)$ is:

linear, if each production $X \to v \in P$ has $v \in V_T^* \cup V_T^* V_N V_T^*$,

right-linear, if each production $X \to v \in P$ has $v \in V_T^* \cup V_T^* V_N$,

left-linear, if each production $X \to v \in P$ has $v \in V_T^* \cup V_N V_T^*$,

regular, if each production $X \to v \in P$ has $v \in V_T \cup V_T V_N \cup \{\lambda\}$.

The family of languages generated by right- or by left-linear grammars are equal and equal to the family of languages generated by regular grammars, which is exactly the family of regular languages.

However, the family of regular languages is strictly included in the family of linear languages, denoted by Lin.

Let $\Delta = \{a^n b^n \mid n \geq 0\}$. The language Δ is linear, but not regular. Moreover, the language $\Delta\Delta$ is context-free but not linear. Thus, the family of linear languages is a proper subfamily of the family of context-free languages.

The linear languages can be characterized also by pushdown automata. Given a computation of pda A, a *turn* in the computation is a move that decreases the height of the pushdown store and is preceded by a move that has not decreased it. A pda A is said to be *one-turn* if in any computation, there is at most one turn. It can be shown that a language is linear if and only if it is recognized by a one-turn pda. This characterization can further be generalized to finite-turn pda's and languages.

Next we present the family of one-counter languages. It is defined through pda's. A pda is *one-counter* if the stack alphabet contains only one letter. A context-free language is a *one-counter language* if it is recognized by a one-counter pda by empty storage and accepting states.

We denote by Ocl this family of languages. The terminology used here comes from the fact that, as soon as the stack alphabet is reduced to a single letter, the stack can be viewed a counter.

However, it is worth noticing that, contrarily to the case of linear languages, one-counter languages do not enjoy other characterizations through grammars as linear languages did.

We now turn to another subfamily of the family of context-free languages. Consider an alphabet Σ containing two particular letters a and $\bar{a}$. A context-free grammar over the terminal alphabet Σ is *parenthetic* if each rule of the grammar has the following form $X \to a\alpha\bar{a}$ with α containing neither the letter a nor the letter $\bar{a}$. As usual, a language is said to be *parenthetic* if it is generated by some parenthetic grammar. In the particular case where the alphabet Σ does not contain any other letters than the two special ones a and $\bar{a}$, we speak of *pure parenthetic* grammar or language.

Clearly, any pure parenthetic language over $\Sigma = \{a, \bar{a}\}$ is included in the Dyck language D_1^*. However, it should be noted that D_1^* is not (purely) parenthetic.

A context-free grammar $G = (V_N, V_T, S, P)$ is *simple* if it is in Greibach normal form and if, for each pair $(X, a) \in V_N \times V_T$, there is at most one rule of the form $X \to am$. As usual, a language is *simple* if it can be generated by a simple grammar. It is easy to check that any simple language is deterministic, and that there do exist deterministic languages which are not simple. The simple languages are exactly the languages recognized by simple deterministic pda's as defined in Section 1.3.3. Moreover, this family of languages enjoys nice properties. For instance, any simple language is prefix, and the family of simple languages generates a free monoid.

A context-free grammar is *very simple* if it is simple and for any terminal letter a there is at most one rule of the form $X \to am$. Clearly, any very simple language is simple. The converse is not true.

1.4 Chomsky Hierarchy

In this Section we present classical *Chomsky hierarchy* of grammars and languages:

1. *phrase-structure* or *type* 0,
2. *context-sensitive* or *type* 1,
3. *context-free* or *type* 2,
4. *regular* or *type* 3.

The phrase-structure, context-sensitive, context-free, and regular grammars are also called type 0, type 1, type 2, and type 3 grammars, respectively. The families of languages they generate are denoted by $\mathcal{L}_0$ (or RE), $\mathcal{L}_1$ (or CSL), $\mathcal{L}_2$ (or CFL), and $\mathcal{L}_4$ (or Reg), respectively. The following strict inclusions hold:

$$\mathsf{Reg} \subset \mathsf{Lin} \subset \mathsf{CFL} \subset \mathsf{CSL} \subset \mathsf{RE}.$$

Type 0 grammars and languages are equivalent to *computability*: what is in principle computable. Thus, their importance is beyond any question. The same or almost the same can be said about regular grammars and languages. They correspond to strictly finitary computing devices. The remaining two classes lie in-between.

The class of context-sensitive languages has turned out to be of smaller importance than the other classes. The particular type of context-sensitivity combined with linear work-space is perhaps not the essential type, it has been replaced by various complexity hierarchies.

The Chomsky hierarchy still constitutes a testing ground often used: new classes are compared with those in the Chomsky hierarchy. However, it is not any more the only testing ground in the language theory.

In Section 1.4.1 we introduce phrase-structure grammars and formulate the relation between languages generated by these grammars and recursively enumerable languages. In Section 1.4.2 we concentrate on the family of context-sensitive languages. Regular languages and context-free languages have already been covered in Section 1.2 and Section 1.3, respectively. Finally, in Section 1.4.3 we summarize closure properties of families in Chomsky hierarchy.

The main source for this Section is Mateescu and Salomaa [23, 22]. Paragraph concerning *growing context-sensitive languages* is taken from [4]. For brevity, we omit definitions of Turing machines and complexity classes. The interested reader is referred to Mateescu and Salomaa [22], or to a nice survey of structural complexity theory in Balcázar et al. [2, 3].

1.4.1 Phrase-Structure Grammars

A *phrase-structure* grammar or a *type* 0 Chomsky grammar is a construct $G = (V_N, V_T, S, P)$, V_N and V_T are disjoint alphabets, $S \in V_N$ and P is a finite set of ordered pairs (u, v), where $u, v \in (V_N \cup V_T)^*$ and u contains at least one element from V_N.

As in Section 1.3.1, elements in V_N are referred to as *nonterminals*, V_T is the *terminal* alphabet, S is the *start symbol* (or *axiom*) and P is the set of *productions* or *rewriting rules*. Productions (u, v) are written $u \to v$. The *direct derivation* relation induced by G is a binary relation between words over $V_N \cup V_T$, denoted $\Rightarrow_G$, and defined as:

$$\alpha \Rightarrow_G \beta \Leftrightarrow \alpha = xuy, \beta = xvy, \text{ and } (u \to v) \in P,$$

where $\alpha, \beta, x, y \in (V_N \cup V_T)^*$.

The *derivation relation* induced by G, denoted $\Rightarrow_G^*$, is the reflexive and transitive closure of the relation $\Rightarrow_G$.

The *language* generated by G, denoted $L(G)$, is:

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

A language L is of type 0 if there exists a grammar G of type 0 such that $L = L(G)$. The family of all languages of type 0 is denoted by $\mathcal{L}_0$.

The fundamental result of Formal Language Theory is that the family $\mathcal{L}_0$ is equal to the family RE of all recursively enumerable languages. The family RE can be introduced by using Turing machines, although there are many other formalisms for defining this family.

1.4.2 Context-Sensitive Grammars

A *context-sensitive* (*type* 1) grammar is a phrase-structure grammar $G = (V_N, V_T, S, P)$ such that each production in P is of the form $\alpha X \beta \to \alpha u \beta$, where $X \in V_N$, $\alpha, \beta, u \in (V_N \cup V_T)^*$, $u \neq \lambda$. In addition, P may contain the production $S \to \lambda$ and in this case S does not occur on the right side of any production of P.

A language L is context-sensitive if there exists a context-sensitive grammar G such that $L = L(G)$. The family of all context-sensitive languages is denoted by CSL or $\mathcal{L}_1$. Note that $\mathcal{L}_1 = \mathsf{CSL} \subseteq \mathcal{L}_0 = \mathsf{RE}$.

A *length-increasing* (*monotonous*) grammar is a type 0 grammar $G = (V_N, V_T, S, P)$ such that for each production $u \to v$ in P, $|u| \leq |v|$. In addition, P may contain the production $S \to \lambda$ and in this case S does not occur on the right side of any production from P.

It can be shown, that L is a context-sensitive language if and only if there exists a length-increasing grammar G such that $L = L(G)$.

Let $L = \{a^n b^n c^n \mid n \geq 1\}$. It can be shown, that L is a context-sensitive language. This immediately implies, that the family of context-free languages is strictly contained in the family of context-sensitive languages, since $L \notin \mathsf{CFL}$.

Now we characterize the family of context-sensitive languages by a so-called *Workspace Theorem*. Let $G = (V_N, V_T, S, P)$ be a type 0 grammar and consider a derivation D according to G,

$$D : S = w_0 \Rightarrow_G w_1 \Rightarrow_G \ldots \Rightarrow_G w_n = w.$$

The *workspace* of w by the derivation D is:

$$WS_G(w, D) = \max\{|w_i| \mid 0 \leq i \leq n\}.$$

The *workspace* of w is:

$$WS_G(w) = \min\{WS_G(w, D) \mid D \text{ is a derivation of } w\}.$$

Observe that $WS_G(w) \geq |w|$ for all G and w.

The following *Workspace Theorem* is a powerful tool in showing languages to be context-sensitive.

Theorem 1.4.1 (Workspace theorem)**.** *If G is a type 0 grammar and if there is a nonnegative integer k such that*

$$WS_G(w) \leq k|w| \textit{ for all nonempty words } w \in L(G),$$

then $L(G)$ is a context-sensitive language.

The family of context-sensitive languages can also be characterized in terms of so-called *linear bounded automata*. Linear bounded automata are a special type of Turing machines, which are closely related to a certain class of Turing space complexity, $NSPACE(n)$. More precisely, a linear bounded automaton (LBA) is a nondeterministic Turing machine of space complexity $S(n) = n$.

Theorem 1.4.2. *The family $\mathcal{L}_1 = \mathsf{CSL}$ is equal to the family of languages accepted by linear bounded automata, i.e.*

$$\mathsf{CSL} = NSPACE(n).$$

It is not known whether the inclusion $DSPACE(n) \subseteq NSPACE(n)$ is strict or not.

It can be shown, that there exist context-sensitive languages for which the membership problem is PSPACE-complete. Thus, in their full generality these languages are too powerful for practical applications. On the other hand, context-free languages are not powerful enough to completely describe all the syntactical aspects of a programming language like Pascal, since some of them are inherently context dependent.

Dahlhaus and Warmuth [8] consider *growing context-sensitive grammars* (GCSL), that is, context-sensitive grammars for which each production rule is strictly length-increasing. Obviously, for such a grammar the length of a derivation is bounded from above by the length of the sentence derived. Dahlhaus and Warmuth prove that all *growing context-sensitive languages*, that is, the languages that are generated by *growing context-sensitive grammars*, have membership problems that are solvable in polynomial time.

1.4.3 Closure properties

Closure properties of families in Chomsky hierarchy (Y stands for *yes* and N stands for *no*) taken from Mateescu and Salomaa [23]:

	RE	CSL	CFL	Lin	Reg
Union	Y	Y	Y	Y	Y
Intersection	Y	Y	N	N	Y
Complementation	N	Y	N	N	Y
Concatenation	Y	Y	Y	N	Y
Kleene star	Y	Y	Y	N	Y
Intersection with regular languages	Y	Y	Y	Y	Y
Substitution	Y	N	Y	N	Y
λ-free substitution	Y	Y	Y	N	Y
Morphism	Y	N	Y	Y	Y
λ-free morphism	Y	Y	Y	Y	Y
Inverse morphism	Y	Y	Y	Y	Y
Left/right quotient	Y	N	N	N	Y
Left/right quotient with regular language	Y	N	Y	Y	Y
Left/right derivate	Y	Y	Y	Y	Y
Shuffle	Y	Y	N	N	Y
Mirror image	Y	Y	Y	Y	Y

Chapter 2

Selected Models of Formal Languages

In this chapter we give an overview of several selected models related to our model of clearing restarting automata.

In Section 2.1 we introduce contextual grammars by Solomon Marcus [20]. In Section 2.2 we briefly describe pure grammars by Maurer et al. [24]. In Section 2.3 we introduce Church-Rosser string rewriting systems [25]. Associative language description by Cherubini et al. [7] is defined in Section 2.4. Finally, restarting automata [13, 27] are covered in Section 2.5.

2.1 Marcus Contextual Grammars

The main source for this section is Ehrenfeucht et al. [9]. Contextual grammars were introduced by Solomon Marcus in 1969 [20], in an attempt to build a bridge between analytical and generative models of natural languages. In particular, contextual grammars were "translating" the central notion of context from the analytical models into the framework of generative grammars. A lucid account of the motivation behind contextual grammars from the natural point of view can be found in Solomon Marcus [21].

Contextual grammars provide an important tool in the study of *recursion* in formal language theory. In a sense, the research on contextual grammars is a study of recursion in its pure form.

Also, contextual grammars offer a very convenient framework for the systematic study of the basic language theoretic operations, such as insertion and deletion, and the operation of splicing, which provide a nice bridge between formal language theory and DNA computing.

Finally it should be stressed that contextual grammars form a major contribution to our understanding of *pure grammars* (see Section 2.2).

In this section we consider contextual grammars from the formal language theory point of view.

A *total contextual grammar* is a system

$$G = (\Sigma, B, C, \phi),$$

where Σ is an alphabet, B is a finite language over Σ, C is a finite subset of $\Sigma^*\$\Sigma^*$, where \$ is a special symbol not in Σ, and $\phi : \Sigma^* \times \Sigma^* \times \Sigma^* \to 2^C$.

The strings in B are called *axioms*, the elements $u\$v$ in C are called *contexts*, and ϕ is the *choice mapping*. Here we do not impose restrictions on ϕ – we even do not assume that ϕ is computable.

For a total contextual grammar $G = (\Sigma, B, C, \phi)$ we define the relation $\Rightarrow_G$ on Σ^* as follows: $x \Rightarrow_G y$ if and only if $x = x_1x_2x_3$, $y = x_1ux_2vx_3$, for some $x_1, x_2, x_3 \in \Sigma^*$, $u\$v \in C$, such that $u\$v \in \phi(x_1, x_2, x_3)$.

The subscript G may be omitted from $\Rightarrow_G$ whenever G is understood. Denoting by $\Rightarrow_G^*$ the reflexive and transitive closure of $\Rightarrow_G$, the language generated by G is

$$L(G) = \{x \in \Sigma^* \mid w \Rightarrow_G^* x \text{ for some } w \in B\}.$$

Note that, by definition, $B \subseteq L(G)$.

We say that x *directly derives* y in G or that x *derives* y in G whenever $x \Rightarrow_G y$ or $x \Rightarrow_G^* y$ holds, respectively.

Two special cases of total contextual grammars are very natural and have been extensively investigated:

1. A total contextual grammar $G = (\Sigma, B, C, \phi)$ is called *external* if $\phi(x_1, x_2, x_3) = \emptyset$ for all $x_1, x_2, x_3 \in \Sigma^*$ such that $x_1x_3 \neq \lambda$.
2. A total contextual grammar $G = (\Sigma, B, C, \phi)$ is called *internal* if $\phi(x_1, x_2, x_3) = \phi(x_1', x_2, x_3')$ for all $x_1, x_1', x_2, x_3, x_3' \in \Sigma^*$.

Note that in both cases, the adjoining of a context $u\$v \in \phi(x_1, x_2, x_3)$ to the string $x_1x_2x_3$ depends on x_2 only. Therefore, in these cases we can simplify the choice mapping ϕ by having $\phi : \Sigma^* \to 2^C$. The so modified total contextual grammars are called *contextual grammars*.

Since in this way there is no difference between external and internal grammars, we shall distinguish between derivation relations, defining:

1. $x \Rightarrow_{ex} y$ if and only if $y = uxv$ for $u\$v \in \phi(x)$,
2. $x \Rightarrow_{in} y$ if and only if $x = x_1x_2x_3$, $y = x_1ux_2vx_3$, for some $x_1, x_2, x_3 \in \Sigma^*$, $u\$v \in \phi(x_2)$.

We call $\Rightarrow_{ex}$ an *external* (direct) derivation and $\Rightarrow_{in}$ an *internal* (direct) derivation. Accordingly we associate with a contextual grammar G two languages:

$$L_\alpha(G) = \{x \in \Sigma^* \mid w \Rightarrow_\alpha x \text{ for some } w \in B\}$$

for $\alpha \in \{ex, in\}$.

For a contextual grammar $G = (\Sigma, B, C, \phi)$, the relation $u\$v \in \phi(x)$ can be interpreted as a rewriting rule $x \to uxv$. Therefore, a contextual grammar with internal derivation can be considered to be a pure grammar with arbitrarily many (length-increasing) productions of the form $x \to uxv$, where there exists a bound k such that, for each x and each production $x \to uxv$, $|uv| \leq k$ holds. (Pure grammars are grammars that do not use nonterminals. See Section 2.2)

A contextual grammar $G = (\Sigma, B, C, \phi)$ is said to be *without choice* if $\phi(x) = C$ for all $x \in \Sigma^*$. In such a case, the mapping ϕ can be ignored, and we write the grammar in the form $G = (\Sigma, B, C)$.

Five basic families of languages are obtained in this way:

1. TC is the family of languages generated by total contextual grammars,
2. ECC is the family of languages externally generated by contextual grammars,
3. ICC is the family of languages internally generated by contextual grammars,
4. EC is the family of languages externally generated by contextual grammars without choice,
5. IC is the family of languages internally generated by contextual grammars without choice.

In general case of contextual grammars it is natural to require that ϕ is a computable mapping. The corresponding families are denoted by TC_c, ECC_c, ICC_c, respectively.

We conclude this section with examples that illustrate the above definitions.

Example 2.1.1. *Every finite language, B, belongs to every family of contextual languages: for $G = (\Sigma, B, \emptyset)$ we have $L_{ex}(G) = L_{in}(G) = B$.*

Example 2.1.2. *For each alphabet Σ, the language Σ^* belongs to every family of contextual languages: for $G = (\Sigma, \{\lambda\}, \$\Sigma)$ we have $L_{ex}(G) = L_{in}(G) = \Sigma^*$. If we take $G = (\Sigma, \Sigma, \$\Sigma)$, then $L_{ex}(G) = L_{in}(G) = \Sigma^+$.*

Example 2.1.3. *Consider the grammar $G = (\{a, b\}, \{\lambda\}, \{a\$b\})$. Obviously $L_{ex}(G) = \{a^n b^n \mid n \geq 1\}$, whereas $L_{in}(G)$ is the Dyck language over the alphabet $\{a, b\}$. This second equality can be proved by induction on the length of the strings.*

Note that $L_{ex}(G)$ is not regular and $L_{in}(G)$ is not linear.

Example 2.1.4. *For $G = (\{a, b, c\}, \{\lambda\}, \{\alpha\beta\$\gamma, \alpha\$\beta\gamma \mid \{\alpha, \beta, \gamma\} = \{a, b, c\}\})$, we obtain*

$$L_{in}(G) = \{x \in \{a, b, c\}^* \mid |x|_a = |x|_b = |x|_c\}.$$

The inclusion $\subseteq$ is obvious. The inverse inclusion can be proved by induction on the length of strings.

Note that the above language is not context-free.

Example 2.1.5. *The grammar $G = (\{a\}, \{a^3\}, \{\$a, \$a^2\}, \phi)$, with mapping ϕ defined by*

$$\phi(a^i) = \begin{cases} \$a & \text{if } i + 1 \neq 2^k \text{ for } k \geq 0 \\ \$a^2 & \text{if } i + 1 = 2^k \text{ for } k \geq 0 \end{cases}$$

for all $i \geq 1$, generates

$$L_{ex}(G) = \{a^n \mid n \geq 1, n \neq 2^k, k \geq 0\},$$
$$L_{in}(G) = a^+ - \{a, a^2\}.$$

2.2 Pure Grammars

The main source for this section is Mateescu and Salomaa [22]. In the Chomsky hierarchy and related types of grammars, it has become customary to divide the alphabet into two parts: *nonterminal* letters and *terminal* letters. Only words consisting entirely of terminal letters are considered to be in the language generated. Basically, this distinction stems from linguistic motivation: nonterminals represent syntactic classes of a language. One can view the use of nonterminals also as an auxiliary definitional mechanisms for generative devices. Not everything derived from the axiom or axioms is accepted as belonging to the language. Nonterminals are used to filter out unwanted words.

The study of *pure grammars* continues the line of research concerning rewriting systems. In a pure grammar there is only one undivided alphabet. The starting point, *axiom*, is a word or, more commonly, a finite set of words. Rewriting is defined in the standard way, and a specific pure grammar is obtained by giving a finite set of rewriting rules or productions. All words derivable from some of the axioms belong to the *language* of the pure grammar. A pure grammar is *context-free* (*length increasing*, respectively), if the length of the left side of every production is equal to one (less than or equal to the length of the right side, respectively). Abbreviations PCF and PLI will be used for these special classes of pure grammars and their languages. One can define also pure context-sensitive grammars in the standard way. The resulting class of languages will be strictly smaller than the class of PLI languages.

The language $\{a^n cb^n \mid n \geq 1\}$ is generated by the PCF grammar with the axiom acb and the only production $c \rightarrow acb$. The language $\{a^n b^n \mid n \geq 1\}$ is not PCF but is generated by the PLI grammar with the axiom ab and the only production $ab \rightarrow a^2b^2$. Neither one of the languages

$$\{a^n b^n c^n \mid n \geq 1\} \text{ and } \{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\}$$

is pure, that is, they are not generated by any pure grammar whatsoever.

Every regular language is PLI and, hence, pure. The family of pure languages over a one-letter alphabet $\{a\}$ coincides with the family of regular languages over $\{a\}$, whereas PCF languages constitute a proper subfamily. There are nonrecursive pure languages. Each of the differences $\mathcal{L}_i \backslash \mathcal{L}_{i+1}$, $0 \leq i \leq 2$, where $\mathcal{L}_i$, $0 \leq i \leq 3$, are the language families in the Chomsky hierarchy, contains both pure and nonpure languages.

2.3 Church-Rosser String Rewriting Systems

The main source for this section is McNaughton et al. [25]. A *Thue system* (after the Norwegian mathematician and logician Axel Thue who introduced it in 1914) is a set of unordered pairs of strings. We write $u_1 \leftrightarrow_T u_2$ (sometimes omitting the subscript when T is understood) when for some $(x, y) \in T$ and strings w, z, $u_1 = wxz$ and $u_2 = wyz$. We write $\leftrightarrow_T^*$ for the reflexive and transitive closure of $\leftrightarrow_T$ (which is already symmetrical). The *alphabet* of a Thue system is the set of all characters appearing in the paired strings.

We write $[x]_T$ to mean $\{x' \mid x' \leftrightarrow_T^* x\}$; in words it is the *congruence class* of x *modulo* T. It is well known that $\{x'y' \mid x' \in [x]_T, y' \in [y]_T\} \subseteq [xy]_T$, and that by defining $[x]_T \cdot [y]_T$ to be $[xy]_T$ we get a monoid whose elements are the congruence classes and whose identity is $[\lambda]$. If z is a string such that $(za, z), (az, z) \in T$, for all a in the alphabet of T, then $[z]_T$ acts as a monoid *zero* on the monoid. Thus for all x and y, $[x]_T[z]_T[y]_T = [z]_T$, corresponding to the fact that $xzy \leftrightarrow_T^* z$.

We write $u \rightarrow_T v$ if $u \leftrightarrow_T v$ and $|u| > |v|$. We call $\rightarrow_T^*$ *reduction*, which is the reflexive and transitive closure of $\rightarrow_T$. A Thue system T has the *Church-Rosser property* (after Alonzo Church and J. Barkley Rosser and their work in 1941 and 1942 on the lambda calculus), or T is a CR *system*, if for all u and v, $u \leftrightarrow_T^* v$ implies there is a t such that $u \rightarrow_T^* t$ and $v \rightarrow_T^* t$.

If T is a CR system, $(x, y) \in T$, $|x| = |y|$ and $T' = T \backslash \{(x, y)\}$, then it is a consequence of the definition that T' is equivalent to T. We henceforth assume that no CR system has length-preserving rules.

Generally, when we work with CR systems we are interested in reductions. When we present a CR system in practice we list the pairs not in the form (x, y) but in the form $x \rightarrow y$, where $|x| > |y|$; $x \rightarrow y$ is called a length-reducing rule or, simply, *rule*.

Theorem 2.3.1. *A Thue system consisting of length-reducing rules has the Church-Rosser property if and only if:*

1. *(Overlap condition) For all* $u_1, u_2, u_3 \neq \lambda$ *if* $u_1u_2 \rightarrow y_1$ *and* $u_2u_3 \rightarrow y_3$ *are rules (not necessarily distinct), then there is a* t *such that* $y_1u_3 \rightarrow^* t$ *and* $u_1y_3 \rightarrow^* t$ *(so that* $u_1u_2u_3 \rightarrow y_1u_3 \rightarrow^* t$ *and* $u_1u_2u_3 \rightarrow u_1y_3 \rightarrow^* t$*).*

2. *(Substring condition) For all* u_1, u_2, u_3 *where* $u_1u_3 \neq \lambda$, *if* $u_1u_2u_3 \rightarrow y_{13}$ *and* $u_2 \rightarrow y_2$ *are rules, then there is a* t *such that* $y_{13} \rightarrow^* t$ *and* $u_1y_2u_3 \rightarrow^* t$ *(so that* $u_1u_2u_3 \rightarrow y_{13} \rightarrow^* t$ *and* $u_1u_2u_3 \rightarrow u_1y_2u_3 \rightarrow^* t$*).*

3. *(Identity condition) If* $u \rightarrow y_1$ *and* $u \rightarrow y_2$ *are rules, then there is a* t *such that* $y_1 \rightarrow^* t$ *and* $y_2 \rightarrow^* t$.

An obvious, but useful, consequence of this principle is the following:

Corollary 2.3.1. *If in a Thue system* T *no left side of a rule properly overlaps with the left side of another rule or itself, no left side is a proper substring of another left side, and no two rules have identical left sides but different right sides, then* T *has the Church-Rosser property.*

A CR system offers a convenient computation method for dealing with strings: Given a string, we simply reduce it as far as possible until we obtain an irreducible equivalent, whereupon the original string is identified. The irreducible equivalent of the string is truly a canonical form for its congruence class. The Church-Rosser property ensures that there cannot be two congruent irreducible strings.

It can be proved that the canonical form of a given string can be obtained in linear time by using an interesting automaton to do the computation, which is a kind of generalization of the pushdown automaton. Languages that CR systems are capable of processing are a significant generalization of the deterministic context-free languages.

In the following we mention three ways in which CR systems can be used to define (and hence to test for membership in) a formal language:

1. L is a *Church-Rosser congruential language* if L is the union of finitely many congruence classes of some CR system.
2. $L \subseteq A_0^*$ is a *Church-Rosser language* (CR *language*) if there is a CR system T whose alphabet A_1 has A_0 as a proper subset, strings $t_1, t_2 \in (A_1 \backslash A_0)^*$, and $Y \in A_1 \backslash A_0$ such that, for all $w \in A_0^*$, $t_1 w t_2 \rightarrow_T^* Y$ if and only if $w \in L$. T is a *defining* CR *system* for the CR language L.
3. $L \subseteq A_0^*$ is a *Church-Rosser-decidable* (or CR-*decidable*) *language* if L is a CR language, and, where T, t_1, t_2, and Y are as mentioned in that definition, there is an $N \in A_1 \backslash A_0$ such that, for all $w \in A_0^*$, $t_1 w t_2 \rightarrow_T^* N$ if and only if $w \notin L$. T is then a *defining* CR *system* for the CR-decidable language L.

The characters of $A_1 \backslash A_0$ of a defining CR system of a CR or CR-decidable language L can be thought of as control characters.

Thus, in the defining CR system T of a CR-decidable language, for any $w \in A_0^*$, we can reduce $t_1 w t_2$ to either Y (meaning yes) or N (no). It should be noted that if L is merely a CR language, we have a decision procedure that is almost as good. We reduce $t_1 w t_2$ as far as we can. If we arrive at a string other than Y to which no rule can applied, then we know that $w \notin L$.

We emphasize that languages in all three classes have linear-time membership algorithms. The family of CR languages is the largest family of languages we know that enjoys the computation method of CR systems.

We do not know whether there exist any CR languages that are not CR decidable. The proposition that all CR languages are CR decidable is clearly equivalent to the proposition that every CR language L has a CR system and string t_1 and t_2, as in the definition, such that $t_1 A_0^* t_2$ is contained in the union of finitely many congruence classes, where A_0 is the alphabet of L.

2.4 Associative Language Description

The main source for this section is Cherubini et al. [7]. The Associative Language Description (ALD) model combines locally testable and constituent structure ideas. The family of ALD languages is strictly included in the family of context-free languages. However, it can be shown that the hardest context-free language [10] is in ALD.

Context-free grammars, in spite of their universal adoption in language reference manuals and compilers, have several shortcomings. They are unable to generate various linguistic constructs, or to handle long-distance dependencies. In this section the following shortcomings of context-free languages are considered:

First, the generative capacity of context-free grammars is not only insufficient, but also misdirected, because it affords languages that are never considered for describing programming languages and never appear in computational linguistics. We have in mind counting languages, which violate the noncounting property, since they characterize the legal strings by some numerical congruence.

A second criticism, originally voiced by Marcus' school of contextual grammars, is that context-free grammars require an unbounded number of metasymbols, the nonterminals. A "pure" grammar should not use metavariables, which are 'external' to the language, but rely instead on structural and distributional properties.

The language definition technique to be presented addresses both criticisms, but does not extend the capacity of context-free grammars. The objective of this section is to formalize the definitions and to highlight the explanatory adequacy by representative examples.

Let Σ be a finite alphabet, and let $\Delta \notin \Sigma$ be the *placeholder*. A *stencil tree* is a tree such that: its internal nodes are labeled by Δ; its leaves have labels in $\Sigma \cup \{\lambda\}$. The *constituents* of a stencil tree are its subtrees of height one and leaves with labels in $\Sigma \cup \{\lambda\} \cup \{\Delta\}$. The *frontier* of a stencil tree T or of a constituent K is denoted, respectively, by $\tau(T)$ and $\tau(K)$.

Given a stencil tree T, a *maximal subtree* of T is a subtree of T whose leaves are also leaves of T.

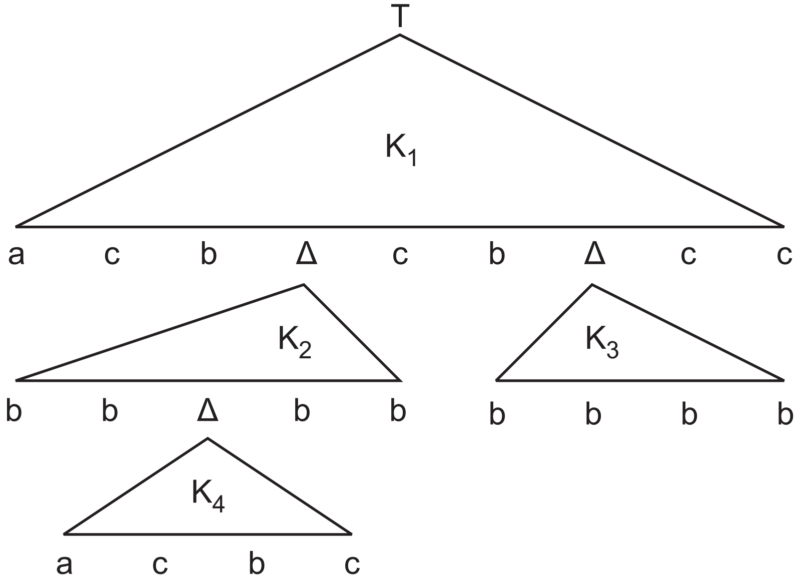

Figure 2.1: A stencil tree T with four constituents K_1, K_2, K_3, K_4 schematized by triangles.

Let T be a stencil tree. For an internal node i of T, let K_i and T_i be, respectively, the constituent and the maximal subtree of T having root i. Consider the tree T' obtained by excising the subtree T_i from T, leaving only the root labeled Δ of T_i behind. Let $s, t \in \Sigma^*$ be two strings such $\tau(T') = s\Delta t$. The *left context* of K_i in T and of T_i in T is $\mathit{Left}(K_i, T) = \mathit{Left}(T_i, T) = s$; the *right context* of K_i in T and of T_i in T is $\mathit{Right}(K_i, T) = \mathit{Right}(T_i, T) = t$.

Let $\perp \notin \Sigma$ be the *left/right terminator*. An associative language description (ALD) A is a finite set of triples (x, z, y), called rules, where $x \in (\lambda \cup \perp)\Sigma^*$, $y \in \Sigma^*(\lambda \cup \perp)$, and $z \in (\Sigma \cup \{\Delta\})^* \backslash \{\Delta\}$. The string z is called the *pattern* of the rule (x, z, y) and the strings x and y are called the *permissible left/right contexts*.

Shorthands: When a left/right context is irrelevant for a pattern, it is represented by the empty string λ or it is omitted. The new symbol Λ may be used to denote the optionality of one occurrence of Δ, that is to merge two rules $(x, z_1\Delta z_2, y)$ and $(x, z_1 z_2, t)$ into the rule $(x, z_1\Lambda z_2, y)$.

Another useful shorthand is the following one: given finite sets X, Z, and Y of words, the notation (X, Z, Y) denotes the set of rules $\{(x, z, y) \mid x \in X, z \in Z, y \in Y\}$. Moreover, instead of $\{w\}$ we can use w.

An ALD defines a set of constraints or test conditions that a stencil tree must satisfy, in the following sense: Let A be an ALD. A constituent K_i of a stencil tree T is matched by a rule (x, z, y) of an ALD A if:

1. $z = \tau(K_i)$,
2. x is a suffix of $\perp \mathit{Left}(K_i, T)$,
3. y is a prefix of $\mathit{Right}(K_i, T)\perp$.

A stencil tree T is *valid* for A if each constituent K_i of T is matched by a rule of A.

Therefore, an ALD is a device for defining a set of stencil trees and a string language, corresponding to their frontiers. This is not achieved by means of a derivation: the validity of a stencil tree is determined by a test. Hence, an ALD is not a generative grammar.

The (*stencil*) *tree language* defined by ALD A, denoted by $T_L(A)$, is the set of all stencil trees valid for A.

The (*string*) *language* defined by ALD A, denoted by $L(A)$, is the set $\{x \in \Sigma^* \mid x = \tau(T) \text{ for some } T \in T_L(A)\}$.

Example 2.4.1. *The language* $\{a^n c b^n \mid n \geq 1\}$ *is defined by the* ALD *rules:*

$$(\perp, a\Delta b, \perp),\ (a, a\Delta b, b),\ (a, c, b).$$

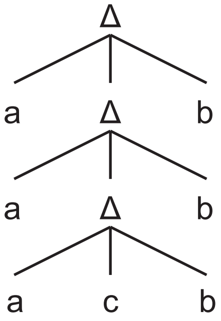

Figure 2.2: A valid stencil tree for Example 2.4.1.

It is also possible to define a simpler, though equivalent, description of the same language with the same patterns but simpler contexts:

$$(\lambda, a\Delta b, \lambda),\ (a, c, b).$$

Similarly, the language $\{a^n b^n \mid n \geq 1\}$ *is defined by the* ALD $\{(\lambda, a\Lambda b, \lambda)\}$.

Example 2.4.2. *Obviously, all* 1*-variable context-free languages are defined by* ALD*s. For instance, the Dyck language* D_1^* *on the opening and closing parentheses* $\{a, \bar{a}\}$ *is defined by the* ALD *rules:*

$$(\lambda, a\Delta\bar{a}, \lambda),\ (\lambda, a\bar{a}, \lambda),\ (\lambda, \lambda, \lambda),$$

where all contexts are empty. The ALD *may be compacted, using the shorthand* Λ*, into* $(\lambda, a\Lambda\bar{a}\Lambda, \lambda)$*.*

Example 2.4.3. *The phenomenon of* ambiguity *can occur in* ALD*s much as in context-free grammars. The following rules ambiguously define Dyck language* D_1^**:*

$$(\lambda, \Lambda\Lambda, \lambda),\ (\lambda, a\Lambda\bar{a}, \lambda),$$

because a sentence like $a\bar{a}a\bar{a}a\bar{a}$ *admits distinct tree structures.*

Example 2.4.4. *It is not known whether all regular languages are* ALD*, but many of them are. For instance, the language* $a^+bc^+ \cup a^+dc^+$ *is defined by the* ALD *rules:* $(\perp, a\Delta c, \perp)$*,* $(a, a\Delta, c)$*,* $(a, \Delta c, c)$*,* (a, b, c)*,* (a, d, c)*.*

As we have already mentioned, the family of ALD languages is strictly included in the family of context-free languages. However, it is possible to define the syntax of Pascal by using ALD. The size of the ALD definition is comparable to a context-free grammar of Pascal when using short notations. Also, the main features of HTML can be described conveniently by ALD.

Several theoretical questions are still open. For instance, it is unknown, at the present, whether the ALD family includes the regular languages (there are many examples of regular languages that are in ALD). There are also other problems such as various decidability properties.

2.5 Restarting Automata

The main source for this section is Friedrich Otto [27]. The restarting automaton was introduced by Jančar et al. in 1995 in order to model the so-called ‘analysis by reduction,’ which is a technique used in linguistics to analyze sentences of natural languages that have a free word order. By now there are many different models of restarting automata, and their investigation has proved very fruitful in that they offer an opportunity to study the influence of various kinds of resources on their expressive power. Here we introduce and discuss the main variants of these automata.

Analysis by reduction is a technique used in linguistics to analyze sentences of natural languages that have a free word order. This analysis consists of a stepwise simplification of a sentence in such a way that the syntactical correctness or incorrectness of the sentence is not affected. After a finite number of steps either a correct simple sentence is obtained, or an error is detected. In the former case the given sentence is accepted as being syntactically correct; if, however, all possible sequences of simplifications yield errors, then the given sentence is not syntactically correct. In this way it is also possible to determine dependencies between various parts of the given sentence, and to disambiguate between certain morphological ambiguities contained in the sentence.

As illustrated by Jančar et al. (e.g. [16]) the restarting automaton was invented to model the analysis by reduction. In fact, many aspects of the work on restarting automata are motivated by the basic tasks of computational linguistics. The notions developed in the study of restarting automata give a rich taxonomy of constraints for various models of analyzers and parsers. Already several programs are being used in Czech and German (corpus) linguistics that are based on the idea of restarting automata.

As defined in [13] the *restarting automaton* is a nondeterministic machine model that processes strings which are stored in a list (or a 'rubber' tape) with end markers. It has a finite control, and it has a read/write window with a finite look-ahead working on the list of symbols. The restarting automaton can only perform two kinds of operations: *move-right transitions*, which shift the read/write window one position to the right, thereby changing the actual state, and combined *delete/restart transitions*, which delete some symbols from the read/write window, place this window over the left end of the list, and put the automaton back into its initial state. Hence, after performing a delete/restart transition a restarting automaton has no way to remember that it has already performed some steps of a computation. Further, by each application of a delete/restart transition the list is shortened. It follows that restarting automata are linearly space-bounded.

Subsequently Jančar et al. extended their model in various ways. Instead of simply deleting some symbols from the actual content of the read/write window during a delete/restart transition, a restarting automaton *with rewriting* has combined *rewrite/restart* transitions that replace the content of the read/write window by a shorter string [14]. Further, the use of auxiliary (that is, non-input) symbols was added to the model in [15], which yields the so-called RWW-*automaton*. Also in [15] the restart transition was separated from the rewrite transition so that, after performing a rewrite step, the automaton can still read the remaining part of the tape before performing a restart transition. This gives the so-called RRWW-*automaton*, which is required to execute exactly one rewrite transition between any two restart transitions. In addition, various notions of *monotonicity* have been discussed for the various types of restarting automata. It turned out that monotone RWW- and RRWW-automata accept exactly the context-free languages, and that all the various types of monotone deterministic RWW- and RRWW-automata accept exactly the deterministic context-free languages. Finally, *move-left transitions* were added to the restarting automaton, which gave the so-called *two-way restarting automaton* (RLWW-automaton) [29]. This automaton can first scan its tape completely, and then move left to apply a rewrite transition to some factor of the tape content.

In defining the restarting automaton and its main variants we will not follow the historical development outlined above, but we will first present the most general model, the RLWW-automaton, and then describe the other variants as restrictions thereof.

As indicated above a restarting automaton is a nondeterministic machine model that has a finite control and a read/write window that works on a list of symbols delimited by end markers (see Figure 2.3).

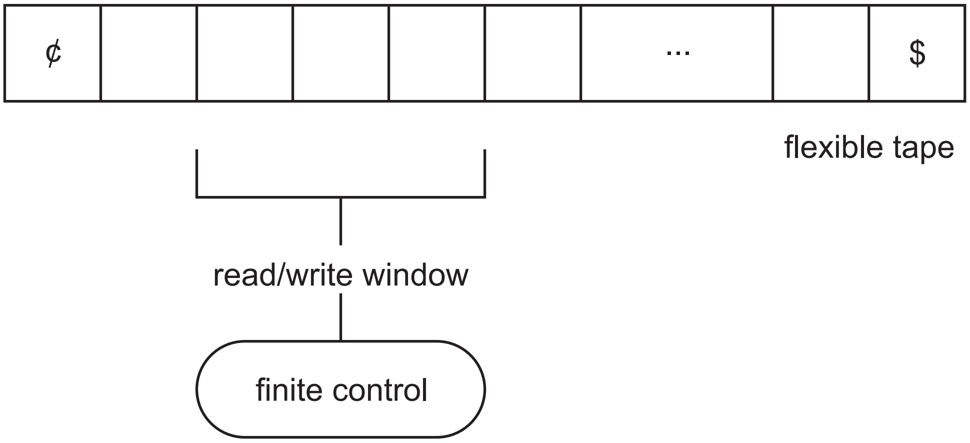

Figure 2.3: Schematic representation of a restarting automaton.

Formally, a *two-way restarting automaton*, RLWW-automaton for short, is a one-tape machine described by an 8-tuple $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$, where Q is a finite set of states, Σ is a finite input alphabet, Γ is a finite tape alphabet containing Σ, $¢, \$ \notin \Gamma$ are symbols that serve as markers for the left and right border of the work space, respectively, $q_0 \in Q$ is the initial state, $k \geq 1$ is the size of the *read/write window*, and

$$\delta : Q \times \mathfrak{PC}^{(k)} \to \mathcal{P}((Q \times (\{\mathsf{MVR}, \mathsf{MVL}\} \cup \mathfrak{PC}^{\leq(k-1)})) \cup \{\mathsf{Restart}, \mathsf{Accept}\})$$

is the *transition relation*. Here $\mathfrak{PC}^{(k)}$ is the set of *possible contents* of the read/write window of M, where

$$\mathfrak{PC}^{(i)} = (¢ \cdot \Gamma^{i-1}) \cup \Gamma^{i} \cup (\Gamma^{\leq i-1} \cdot \$) \cup (¢ \cdot \Gamma^{\leq i-2} \cdot \$) \quad (i \geq 0),$$

and

$$\Gamma^{\leq n} = \bigcup_{i=0}^{n} \Gamma^{i} \text{ and } \mathfrak{PC}^{\leq(k-1)} = \bigcup_{i=0}^{k-1} \mathfrak{PC}^{(i)}.$$

The transition relation describes five different types of transition steps:

1. A *move-right step* is of the form $(q', \mathsf{MVR}) \in \delta(q, u)$, where $q, q' \in Q$ and $u \in \mathfrak{PC}^{(k)}$, $u \neq \$$. If M is in state q and sees the string u in its read/write window, then this move-right step causes M to shift the read/write window one position to the right and to enter state q'. However, if the content u of the read/write window is only the symbol \$, then no shift to the right is possible.

2. A *move-left step* is of the form $(q', \mathsf{MVL}) \in \delta(q, u)$, where $q, q' \in Q$ and $u \in \mathfrak{PC}^{(k)}$ does not begin with the symbol ¢. It causes M to shift the read/write window one position to the left and to enter state q'. This, however, is only possible if the window is not already at the left end of the tape.

3. A *rewrite step* is of the form $(q', v) \in \delta(q, u)$, where $q, q' \in Q$ and $u \in \mathfrak{PC}^{(k)}$, $u \neq \$$, and $v \in \mathfrak{PC}^{\leq(k-1)}$ such that $|v| < |u|$. It causes M to replace the content of u of the read/write window by the string v, thereby shortening the tape, and to enter state q'. Further, the read/write window is placed immediately to the right of the string v. However, some additional restrictions apply in that the border markers ¢ and \$ must not disappear from the tape nor that new occurrences of these markers are created. Further, the read/write window must not move across the right border marker \$, that is, if the string u ends in \$, then so does the string v, and after performing the rewrite operation, the read/write window is placed on the \$-symbol.

4. A *restart step* is of the form Restart $\in \delta(q, u)$, where $q \in Q$ and $u \in \mathfrak{PC}^{(k)}$. It causes M to place its read/write window over the left end of the tape, so that the first symbol it sees is the left border marker ¢, and to reenter the initial state q_0.

5. An *accept step* is of the form Accept $\in \delta(q, u)$, where $q \in Q$ and $u \in \mathfrak{PC}^{(k)}$. It causes M to halt and accept.

If $\delta(q, u) = \emptyset$ for some $q \in Q$ and $u \in \mathfrak{PC}^{(k)}$, then M necessarily halts, and we say that M *rejects* in this situation. Further, the letters in $\Gamma \backslash \Sigma$ are called *auxiliary symbols*.

A *configuration* of M is a string $\alpha q \beta$, where $q \in Q$, and either $\alpha = \lambda$ and $\beta \in \{¢\} \cdot \Gamma^* \cdot \{\$\}$ or $\alpha \in \{¢\} \cdot \Gamma^*$ and $\beta \in \Gamma^* \cdot \{\$\}$; here $q \in Q$ represents the current state, $\alpha\beta$ is the current content of the tape, and it is understood that the read/write window contains the first k symbols of β or all of β when $|\beta| \leq k$. A *restarting configuration* is of the form $q_0¢w\$$, where $w \in \Gamma^*$; if $w \in \Sigma^*$, then $q_0¢w\$$ is an *initial configuration*. Thus, initial configurations are a particular type of restarting configurations. Further, we use Accept to denote the *accepting configurations*, which are those configurations that M reaches by executing an Accept instruction. A configuration of the form $\alpha q \beta$ such that $\delta(q, \beta_1) = \emptyset$, where β_1 is the current content of the read/write window, is a *rejecting configuration*. A *halting configuration* is either an accepting or a rejecting configuration.

In general, the automaton M is *nondeterministic*, that is, there can be two or more instructions with the same left-hand side (q, u), and thus, there can be more than one computation for an input word. If this is not the case, the automaton is *deterministic*. We use the prefix det- to denote deterministic classes of restarting automata.

We observe that any finite computation of a two-way restarting automaton M consists of certain phases. A phase, called a *cycle*, starts in a restarting configuration, the head moves along the tape performing MVR, MVL, and Rewrite operations until a Restart operation is performed and thus a new restarting configuration is reached. If no further Restart operation is performed, any finite computation necessarily finishes in a halting configuration – such a phase is called a *tail*. We require that M performs *exactly one* Rewrite operation during any cycle – thus each new phase starts on a shorter word than the previous one. During a tail at most one Rewrite operation may be executed. By $\vdash^c_M$ we denote the execution of a complete cycle, and $\vdash^{c^*}_M$ is the reflexive and transitive closure of this relation. It can be seen as the *rewrite relation* that is realized by M on the set of restarting configurations.

An input $w \in \Sigma^*$ is accepted by M, if there exists a computation of M which starts with the initial configuration $q_0¢w\$$, and which finally ends with executing an Accept instruction.

In the following we restate some basic facts about computations of restarting automata.

Proposition 2.5.1 (Error Preserving Property)**.** *Let M be an* RLWW-*automaton, and let u, v be words over its* input *alphabet* Σ*. If* $q_0¢u\$ \vdash_M^{c^*} q_0¢v\$$ *holds and* $u \notin L(M)$*, then* $v \notin L(M)$*, either.*

Proposition 2.5.2 (Correctness Preserving Property)**.** *Let M be an* RLWW-*automaton, and let u, v be words over its* input *alphabet* Σ*. If* $q_0¢u\$ \vdash_M^{c^*} q_0¢v\$$ *is an initial segment of an* accepting *computation of* M*, then* $v \in L(M)$*.*

Each cycle of each computation of an RLWW-automaton M consists of three phases: first M scans its tape performing MVR- and MVL-instructions, then it executes a Rewrite step, and finally it scans its tape again performing MVR- and MVL-steps. Hence, in the first and the last phase of each cycle M behaves like a nondeterministic two-way finite-state acceptor (2-NFA). In an analogy to the proof that the language accepted by a 2-NFA is regular (see e.g. [12]), the following result can be established. Here a restarting automaton is called an RRWW-*automaton* if it does not use any MVL-transitions. Thus, in each cycle an RRWW-automaton can scan its tape only once from left to right.

Theorem 2.5.1. *Let* $M_L = (Q_L, \Sigma, \Gamma, ¢, \$, q_0, k, \delta_L)$ *be an* RLWW-*automaton. Then there exists an* RRWW-*automaton* $M_R = (Q_R, \Sigma, \Gamma, ¢, \$, q_0, k, \delta_R)$ *such that, for all* $u, v \in \Gamma^*$,

$$q_0¢u\$ \vdash_{M_L}^{c} q_0¢v\$ \text{ if and only if } q_0¢u\$ \vdash_{M_R}^{c} q_0¢v\$,$$

and the languages $L(M_L)$ *and* $L(M_R)$ *coincide.*

Thus, as far as nondeterministic restarting automata are concerned, the MVL-instruction is not needed. However, this does not hold for deterministic restarting automata. For RRWW-automata we have the following normalization result, which is easily proved by using standard techniques from automata theory.

Lemma 2.5.1. *Each* RRWW-*automaton* M *is equivalent to an* RRWW-*automaton* M' *that satisfies the following additional restriction:*

(*) M' *makes an accept or a restart step only when it sees the right border marker* \$ *in its read/write window.*

This lemma means that in each cycle of each computation and also during the tail of each accepting computation the read/write window moves all the way to the right before a restart is made, respectively, before the machine halts and accepts.

Based on this fact each cycle (and also the tail) of a computation of an RRWW-automaton consists of three phases. Accordingly, the transition relation of an RRWW-automaton can be described through a sequence of so-called *meta-instructions* of the form $(E_1, u \to v, E_2)$, where E_1 and E_2 are regular languages, called the *regular constraints* of this instruction, and u and v are strings such that $|u| > |v|$. The rule $u \to v$ stands for a Rewrite step of the RRWW-automaton M considered. On trying to execute this meta-instruction M will get stuck (and so reject) starting from the configuration $q_0 ¢ w \$$, if w does not admit a factorization of the form $w = w_1 u w_2$ such that $¢ w_1 \in E_1$ and $w_2 \$ \in E_2$. On the other hand, if w does have a factorization of this form, then one such factorization is chosen nondeterministically, and $q_0 ¢ w \$$ is transformed into $q_0 ¢ w_1 v w_2 \$$. In order to describe the tails of accepting computations we use meta-instructions of the form $(¢ \cdot E \cdot \$, \mathsf{Accept})$, where the strings from the regular language E are accepted by M in tail computations.

Finally, we introduce some restricted types of restarting automata. A restarting automaton is called an RWW-*automaton* if it makes a restart immediately after performing a rewrite operation. In particular, this means that it cannot perform a rewrite step during the tail of a computation.

A cycle of a computation of an RWW-automaton M consists of two phases only. Accordingly, the transition relation of an RWW-automaton can be described by a finite sequence of *meta-instructions* of the form $(E, u \to v)$, where E is a regular language, and u and v are strings such that $|u| > |v|$, and meta-instructions of the form $(¢ \cdot E \cdot \$, \mathsf{Accept})$ for describing tail computations.

An RLWW-automaton is called an RLW-*automaton* if its tape alphabet Γ coincides with its input alphabet Σ, that is, if no auxiliary symbols are available. It is an RL-*automaton* if it is an RLW-automaton for which the right-hand side v of each Rewrite step $(q', v) \in \delta(q, u)$ is a scattered subword of the left-hand side u. Analogously, we obtain RRW- and RR-automata from the RRWW-automaton and RW- and R-automata from the RWW-automaton, respectively.

It can be shown that $\mathcal{L}(\mathsf{R})$ contains languages that are not *growing context-sensitive*. Hence, already the R-automaton has a fairly large expressive power.

We conclude this section with the notion of monotonicity for restarting automata (introduced in [14]). Here we consider a slightly more general notion, which is taken from [17]. Let M be an RLWW-automaton. Each computation of M can be described by a sequence of cycles $C_1, C_2, \ldots, C_n$, where C_n is the last cycle, which is followed by the tail of the computation. Each cycle C_i of this computation contains a unique configuration of the form $\cent xquy\$$ such that q is a state and $(q', v) \in \delta(q, u)$ is the $\mathsf{Rewrite}$ step that is applied during this cycle. By $D_r(C_i)$ we denote the *right distance* $|uy\$|$ of this cycle, and $D_l(C_i)$ is the *left distance* $|\cent x|$ of this cycle. The sequence of cycles $C_1, C_2, \ldots, C_n$ is called *monotone* if $D_r(C_1) \geq D_r(C_2) \geq \ldots \geq D_r(C_n)$ holds. A computation of M is called *monotone* if the corresponding sequence of cycles is monotone. Observe that the tail of the computation is not taken into account here. Finally, the RLWW-automaton M is called *monotone* if each of its computations that starts from an initial configuration is monotone.

Here we want to compare the expressive power of the various types of monotone restarting automata to each other. We use the prefix mon- to denote the classes of monotone restarting automata.

All variants of deterministic monotone restarting automata obtained from the RRWW-automaton coincide in their expressive power:

Theorem 2.5.2. *For all types* $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RWW}, \mathsf{RRWW}\}$,

$$\mathcal{L}(\text{det-mon-X}) = \mathsf{DCFL}.$$

However, it can be shown that $\mathsf{DCFL} \subset \mathcal{L}(\text{det-mon-RL}) \subseteq \mathcal{L}(\text{det-mon-RLWW})$.

For nondeterministic restarting automata it turns out that the use of auxiliary symbols is necessary to obtain a characterization of the class of context-free languages.

Theorem 2.5.3. *For all types* $\mathsf{X} \in \{\mathsf{RLWW}, \mathsf{RRWW}, \mathsf{RWW}\}$,

$$\mathcal{L}(\text{mon-X}) = \mathsf{CFL}.$$

Chapter 3

Clearing Restarting Automata

In this chapter, we study the class of languages recognized by clearing restarting automata (cl-RA-automata). Before we introduce our restricted model of restarting automata in Section 3.2, we define a more general concept called *context rewriting system* in Section 3.1. This concept will serve us as a framework for all restricted models studied in this thesis.

In Section 3.3 we show that all regular languages can be recognized by clearing restarting automata with restricted left (right, respectively) contexts. In case of a one-letter alphabet, clearing restarting automata recognize exactly all regular languages containing the empty word.

In Section 3.4 we present some (non-)closure properties of $\mathcal{L}(\mathsf{cl}\text{-}\mathsf{RA})$. We prove that $\mathcal{L}(\mathsf{cl}\text{-}\mathsf{RA})$ is not closed under several operations (concatenation, intersection, intersection with a regular language, difference) and that there exist context-free languages which cannot be recognized by any cl-RA-automaton.

In Section 3.5 we show that the language classes recognized by k-cl-RA-automata create an infinite proper hierarchy with respect to the length of contexts k.

In the following Sections 3.6, 3.7, 3.8, and 3.9 we study under which restrictions (on the length of contexts and the size of alphabet) k-cl-RA-automata can recognize non-context-free languages.

In Section 3.6 we prove that 4-cl-RA-automata are able to recognize a non-context-free language over a two-letter alphabet. We construct a 4-cl-RA-automaton $M = (\Sigma, I)$ such that $L(M) \cap \{(ab)^n \mid n > 0\} = \{(ab)^{2^l} \mid l = 0, 1, 2, \ldots\}$. Since context-free languages are closed under intersection with regular languages, it follows that $L(M)$ is a non-context-free language.

Rather than describing the language $L(M)$ directly, we use a special inverse morphism of this language called *circle-square representation* of $L(M)$. We use the term circle-square representation since we map the language $L(M)$ into a language on alphabet $\{\bigcirc, \square\}$, i.e. the words of this language are composed of circles and squares.

Similar circle-square representations are used also in other sections. The main motivation behind their use is that they often give us some insight into the structure of the language generated by the clearing restarting automaton. The language itself is often very difficult to describe directly, but usually it contains some regularities, which are easily captured by using the proper circle-square representation.

In Section 3.7 we prove that 1-cl-RA-automata can recognize only context-free languages, i.e. $\mathcal{L}(\text{1-cl-RA}) \subset \text{CFL}$. Thus if we want to recognize a non-context-free language, then we need contexts of length at least two.

In Section 3.8 we prove that there exists a 2-cl-RA-automaton recognizing a non-context-free language over a six-letter alphabet. In the first part we describe a general idea of *sending a signal* through an *ether* and the subsequent *recovery* of the ether by using only instructions of a 2-cl-RA-automaton. You can imagine the ether as a word with some special properties. The signal is usually a single letter, which spreads through this ether. The propagation of the signal disturbs the ether. Therefore, we have special instructions that recover the disturbed ether. In the second part we look at the process of sending a signal through the ether by using a special circle-square representation. This representation substantially simplifies the original language generated by the process of sending a signal and enables us to prove that this language is a non-context-free language. In the last part we show that it is possible to construct a 2-cl-RA-automaton on a six-letter alphabet which simulates the aforementioned process of sending a signal.

In Section 3.9 we construct a 3-cl-RA-automaton $M = (\Sigma, I)$ on a two-letter alphabet $\Sigma = \{a, b\}$ such that $L(M)$ is a non-context-free language. The idea of this automaton is based on the idea of sending a signal through the ether. As you can see this result improves the result from Section 3.6. In spite of this fact, we have decided to include Section 3.6, since we present there some basic ideas, which are then reused in other subsections in a more complex form.

Most of the material used in this chapter is taken from the extended version of [6] accepted for publication in Fundamenta Informaticae.

3.1 Context Rewriting Systems

In this section we introduce a general concept called called *context rewriting system* which will serve us as a framework for all restricted models studied in this thesis.

Definition 3.1.1. *Let k be a positive integer. A k-*context rewriting system *(k-*CRS-system *for short) is a system $R = (\Sigma, \Gamma, I)$, where Σ is an input alphabet, $\Gamma \supseteq \Sigma$ is a working alphabet not containing the special symbols ¢ and \$, called* sentinels*, and I is a finite set of* instructions *of the form:*

$$(x, z \to t, y) \ ,$$

where x is called left context*, $x \in LC_k = \Gamma^k \cup ¢ \cdot \Gamma^{\leq k-1}$, y is called* right context*, $y \in RC_k = \Gamma^k \cup \Gamma^{\leq k-1} \cdot \$$ and $z \to t$ is called* rule*, $z, t \in \Gamma^*$.*

A word $w = uzv$ can be rewritten *into utv (denoted as $uzv \to_R utv$) if and only if there exists an instruction $i = (x, z \to t, y) \in I$ such that x is a suffix of $¢ \cdot u$ and y is a prefix of $v \cdot \$$. We often underline the rewritten part of the word w, and if the instruction i is known we use $\to_R^{(i)}$ instead of $\to_R$, i.e. $u\underline{z}v \to_R^{(i)} utv$. The relation $\to_R \subseteq \Gamma^* \times \Gamma^*$ is called* rewriting relation.

Let $l \in ¢ \cdot \Gamma^* \cup \Gamma^*$, and $r \in \Gamma^* \cup \Gamma^* \cdot \$$. A word $w = uzv$ can be rewritten in the context (l, r) *into utv (denoted as $uzv \to_R utv$* in the context (l, r)*) if and only if there exists an instruction $i = (x, z \to t, y) \in I$, such that x is a suffix of $l \cdot u$ and y is a prefix of $v \cdot r$. Each definition that uses somehow the rewriting relation $\to_R$ can be relativized to any context (l, r). Unless told otherwise, we will use the* standard context *$(l, r) = (¢, \$)$.*

The production language *(*reduction language*, respectively) associated with R is defined as $L^+(R) = \{w \in \Sigma^* \mid \lambda \to_R^* w\}$ ($L^-(R) = \{w \in \Sigma^* \mid w \to_R^* \lambda\}$, respectively), where $\to_R^*$ is the reflexive and transitive closure of $\to_R$. Note that, by definition, $\lambda \in L^+(R)$ ($\lambda \in L^-(R)$, respectively).*

The production characteristic language *(*reduction characteristic language*, respectively) associated with R is defined as $L_C^+(R) = \{w \in \Gamma^* \mid \lambda \to_R^* w\}$ ($L_C^-(R) = \{w \in \Gamma^* \mid w \to_R^* \lambda\}$, respectively).*

Obviously, for each k-CRS*-system R, it holds $L^+(R) = L_C^+(R) \cap \Sigma^*$ ($L^-(R) = L_C^-(R) \cap \Sigma^*$).*

Remark 3.1.1. *We extend Definition 3.1.1 with the following notation: if $X \subseteq LC_k$ and $Y \subseteq RC_k$ are finite nonempty sets, and Z is a finite nonempty set of rules of the form $z \to t$, $z, t \in \Gamma^*$, then we define $(X, Z, Y) = \{(x, z \to t, y) \mid x \in X, (z \to t) \in Z, y \in Y\}$. However, if $X = \{x\}$, then instead of* *writing $(\{x\}, Z, Y)$ we write only (x, Z, Y) for short. The same holds for the sets Z and Y, too.*

By reversing all rewriting rules of a k-CRS-system we obtain a dual system.

Definition 3.1.2. *Let $R = (\Sigma, \Gamma, I)$ be a k-CRS-system. A* dual context rewriting system R^D *is a k-CRS-system $R^D = (\Sigma, \Gamma, I^D)$, where $I^D = \{(x, t \to z, y) \mid (x, z \to t, y) \in I\}$. For an instruction $i = (x, z \to t, y)$, we call $i^D = (x, t \to z, y)$ a* dual instruction *to the instruction i. We also define a* dual rewriting relation *to the relation $\to_R$ as $(\to_R)^D = \to_{R^D}$.*

Theorem 3.1.1 (Duality theorem). *For each k-CRS-system $R = (\Sigma, \Gamma, I)$ and its corresponding dual system R^D the following holds:*

$$\begin{array}{ll} (1) & (\to_R)^D = (\to_R)^{-1}, \\ (2) & (R^D)^D = R, \\ (3) & L^+(R) = L^-(R^D), \\ (4) & L^+_C(R) = L^-_C(R^D). \end{array}$$

Proof. (1) Obviously, for all $w, w' \in \Gamma^* : w(\to_R)^D w' \Leftrightarrow w \to_{R^D} w' \Leftrightarrow w' \to_R w$, thus $(\to_R)^D = (\to_R)^{-1}$.
(2) is trivial, (3) and (4) follow from (1). □

Naturally, if we increase the length of contexts used in instructions of a CRS-system, we can increase their power only.

Theorem 3.1.2 (Context extension theorem). *For each k-CRS-system $R = (\Sigma, \Gamma, I)$ there exists a $(k+1)$-CRS-system $R' = (\Sigma, \Gamma, I')$ such that, for each $w, w' \in \Gamma^*$, it holds $w \to_R w' \Leftrightarrow w \to_{R'} w'$. Moreover, both R and R' use the same rewriting rules:*

$$\{z \to t \mid (x, z \to t, y) \in I\} = \{z' \to t' \mid (x', z' \to t', y') \in I'\} .$$

Proof. For each instruction $i = (x, z \to t, y) \in I$ let us define J_i to be $(X, z \to t, Y)$, where:
(1) If $x \in \Gamma^k$, then $X = (\Gamma \cup \{¢\}) \cdot x$. If $x \in ¢ \cdot \Gamma^{\leq k-1}$, then $X = \{x\}$. Evidently, $X \subseteq LC_{k+1}$.
(2) If $y \in \Gamma^k$, then $Y = y \cdot (\Gamma \cup \{\$\})$. If $y \in \Gamma^{\leq k-1} \cdot \$$, then $Y = \{y\}$. Obviously, $Y \subseteq RC_{k+1}$.

It is easy to see that $u\underline{z}v \rightarrow^{(i)} utv$ if and only if $u\underline{z}v \rightarrow^{(j)} utv$ for some $j \in J_i$. This implies that if we set $I' := \bigcup_{i\in I} J_i$, then we get a $(k+1)$-CRS-system $R' = (\Sigma, \Gamma, I')$ which has the same rewriting relation as the k-CRS-system $R = (\Sigma, \Gamma, I)$ and both R and R' use the same rewriting rules. $\square$

Remark 3.1.2. *Based on the above result, in Definition 3.1.1 we can allow contexts of any length up to k, i.e. we can use:*

$LC_{\leq k} = \Gamma^{\leq k} \cup ¢ \cdot \Gamma^{\leq k-1} = \bigcup_{i\leq k} LC_i$ *instead of* LC_k *and*

$RC_{\leq k} = \Gamma^{\leq k} \cup \Gamma^{\leq k-1} \cdot \$ = \bigcup_{i\leq k} RC_i$ *instead of* RC_k.

The following theorem corresponds to correctness and error preserving properties of restating automata ([16]).

Theorem 3.1.3 (Correctness and error preserving theorem)**.** *Let* $R = (\Sigma, \Gamma, I)$ *be a* k-CRS*-system and* u, v *be two words from* Σ^* *such that* $u \rightarrow_R^* v$. *Then:*

$$\begin{array}{ll} (1) & u \in L^+(R) \Rightarrow v \in L^+(R), \\ (2) & u \notin L^-(R) \Rightarrow v \notin L^-(R). \end{array}$$

Proof. Let us suppose that $u, v \in \Sigma^*$ and $u \rightarrow_R^* v$.
(1) $u \in L^+(R)$ implies $\lambda \rightarrow_R^* u$ and thus $\lambda \rightarrow_R^* u \rightarrow_R^* v$. Hence, v is in $L^+(R)$.
(2) $v \in L^-(R)$ implies $v \rightarrow_R^* \lambda$ and thus $u \rightarrow_R^* v \rightarrow_R^* \lambda$, which implies $u \in L^-(R)$. $\square$

3.2 Clearing Restarting Automata

In this thesis, we will not study k-CRS-systems in their general form, since they are too powerful (they can represent recursively enumerable languages). Instead, we will always put some restrictions on the rules of instructions and then study such restricted models. The first model we study is called *clearing restarting automaton* which is a k-CRS-system such that $\Sigma = \Gamma$ and all rules in its instructions are of the form $z \rightarrow \lambda$, where $z \in \Sigma^+$.

Definition 3.2.1. *Let* k *be a positive integer. A* k-clearing restarting automaton *(*k-cl-RA-automaton *for short) is a system* $M = (\Sigma, I)$*, where* $M = (\Sigma, \Sigma, I)$ *is a* k-CRS*-system such that for each instruction* $i = (x, z \rightarrow t, y) \in I$ *it holds* $z \in \Sigma^+$ *and* $t = \lambda$*. Since* t *is always the empty word, we use the notation* $i = (x, z, y)$*. The* width *of the instruction* $i = (x, z, y)$ *is* $|i| = |xzy|$.

The k-cl-RA-*automaton* M recognizes *the language* $L(M) = \{w \in \Sigma^* \mid w \vdash_M^* \lambda\} = L^-(M)$, *where* $\vdash_M$ *is the* rewriting relation $\to_M$ *of* M*.*

In the following, cl-RA denotes the class of all clearing restarting automata. $\mathcal{L}(k\text{-cl-RA})$ ($\mathcal{L}(\text{cl-RA})$, respectively) denotes the class of all languages accepted by k-cl-RA-automata (cl-RA-automata, respectively), $\mathcal{L}(\text{cl-RA}) = \bigcup_{k=1}^{\infty} \mathcal{L}(k\text{-cl-RA})$. In the rest of the paper we will freely use regular expressions in places where regular languages are required.

The model of clearing restarting automaton was inspired by the *Associative Language Description* (ALD) model from [7]. The simplicity of the cl-RA model implies that the investigation of its properties and the proofs are not so difficult and also the learning of languages is easy, fast and straightforward. Another important advantage of this model is that the instructions are human readable and simpler than the meta-instructions of general restarting automata.

It is easy to see that an instruction (x, z, y) of a k-cl-RA-automaton corresponds to the rewriting meta-instruction $(x', z \to \lambda, y')$ of an RR-automaton, where either $x' = \{x''\}$ in the case when $x = ¢ \cdot x''$, or $x' = \Sigma^* \cdot \{x\}$ in the case when x does not start with ¢, and either $y' = \{y''\}$ in the case when $y = y'' \cdot \$$, or $y' = \{y\} \cdot \Sigma^*$ in the case when y does not end with \$.

Example 3.2.1. *In this example we present a* 1-cl-RA-*automaton* $M = (\Sigma, I)$ *that recognizes the language* $L = \{a^n b^n \mid n \geq 0\}$*. Let* $\Sigma = \{a, b\}$ *and* I *be the following set of instructions:*

$$\begin{array}{ll} (1) & (a, ab, b), \\ (2) & (¢, ab, \$). \end{array}$$

In the following $\alpha \vdash_M^{(i)} \beta$ *means that we get* $\alpha \vdash_M \beta$ *by applying instruction* (i)*. Moreover, we underline the "cleared" part of the word* α*. For example:*

$$aaa\underline{ab}bbb \vdash_M^{(1)} aa\underline{ab}bb \vdash_M^{(1)} a\underline{ab}b \vdash_M^{(1)} \underline{ab} \vdash_M^{(2)} \lambda.$$

It means that $aaaabbbb \vdash_M^* \lambda$*, so the word* $aaaabbbb$ *is accepted by* M*. It is easy to see that* $L(M) = L$*.*

The corresponding restarting automaton $N = (\Sigma, \Gamma, J)$*, recognizing the same language as* M*, has* $\Gamma = \Sigma = \{a, b\}$ *and the following set of meta-instructions* J*:*

$$\begin{array}{ll} (1) & (\Sigma^* \cdot \{a\}, ab \to \lambda, \{b\} \cdot \Sigma^*), \\ (2) & (\{\lambda\}, ab \to \lambda, \{\lambda\}), \\ (3) & (\{\lambda\}, \mathsf{Accept}). \end{array}$$

In the following $\alpha \vdash_N^{(i)c} \beta$ *means that* $\alpha \vdash_N^c \beta$ *by applying the meta-instruction* (i)*. For example, the word* $aaaabbbb$ *is recognized by* N *using the following sequence of reductions:*

$$aaa\underline{ab}bbb \vdash_N^{(1)c} aa\underline{ab}bb \vdash_N^{(1)c} a\underline{ab}b \vdash_N^{(1)c} \underline{ab} \vdash_N^{(2)c} \lambda.$$

The computation ends by the word λ*, which is accepted by the accepting meta-instruction* (3)*.*

The construction used in Example 3.2.1 can be generalized into a construction which, for any given cl-RA-automaton M, constructs an equivalent RR-automaton N recognizing the same language as M. Hence, we have the following theorem.

Theorem 3.2.1. $\mathcal{L}(\textsf{cl-RA}) \subseteq \mathcal{L}(\textsf{RR})$.

As we will see later, the above inclusion is proper. Since cl-RA-automata can be considered as a subclass of RR-automata, they inherit also some of their basic properties. In particular, the automata are *error preserving* which is formally stated in the following theorem.

Theorem 3.2.2 (Error preserving property [16])**.** *Let* $M = (\Sigma, I)$ *be a* cl-RA*-automaton and* u, v *be two words from* Σ^**. If* $u \vdash_M^* v$ *and* $u \notin L(M)$*, then* $v \notin L(M)$*.*

This theorem is also a consequence of Theorem 3.1.3, since $L(M) = L^-(M)$.

Remark 3.2.1. *By definition, each* cl-RA*-automaton accepts* λ*. If we say that a* cl-RA*-automaton* M recognizes *(or* accepts*) a language* L *we always mean that* $L(M) = L \cup \{\lambda\}$*.*

This implicit acceptance of the empty word can be avoided by a slight modification of the definition of clearing restarting automata, or even context rewriting system, but in principle, we would not get a more powerful model.

Remark 3.2.2. *As we have seen, the language recognized by a* k-cl-RA*-automaton* $M = (\Sigma, I)$ *is defined as the reduction language of* M*, i.e.* $L(M) = L^-(M)$*. Also note that* $L^+(M) = \{\lambda\}$*. Now suppose that* $N = M^D$ *is a dual* k-CRS*-system to the* k-CRS*-system* M*.* N *is no longer a clearing restarting automaton, because it contains instructions of the form* $(x, \lambda \to z, y)$*, where* $z \in \Sigma^+$*. But according to the Duality Theorem (Theorem 3.1.1),* $L(M) = L^-(M) = L^+(M^D) = L^+(N)$*. This reasoning suggests that we can look at clearing restarting automata from two points of view:*

1. *We can consider a* cl-RA-*automaton* $M = (\Sigma, I)$ *to be an automaton that recognizes the language* $L(M)$ *by using* reductions*, i.e.* $L(M) = \{w \in \Sigma^* \mid w \vdash_M^* \lambda\}$*, where* $\vdash_M$ *is the rewriting relation of* M*, called* reduction!relation*.*

2. *We can consider a* cl-RA-*automaton* $M = (\Sigma, I)$ *to be a generative device generating the language* $L(M)$ *by using* productions*, i.e.* $L(M) = \{w \in \Sigma^* \mid \lambda \dashv_M^* w\}$*, where* $\dashv_M = (\vdash_M)^{-1}$ *is the rewriting relation of* $N = M^D$*, called* production relation*.*

3.3 Clearing restarting automata and regular languages

In contrast to associative language description ([7]), clearing restarting automata can recognize all regular languages. We show that clearing restarting automata using only instructions with left contexts starting with the left sentinel ¢ recognize exactly the class of regular languages. Further, if we restrict the alphabet used by a cl-RA-automaton to a single letter, the respective automata recognize regular languages over one-letter alphabet only.

Theorem 3.3.1. *All regular languages can be recognized by clearing restarting automata using only instructions with left contexts starting with ¢. Hence, for each regular language* L*, there exists a* cl-RA-*automaton* M *such that* $L(M) = L \cup \{\lambda\}$*.*

Proof. The proof is almost identical to the proof of Proposition 3.4 of [18], which states that each regular language is accepted by a stateless R-automaton. □

Next, we show that the converse also holds if each instruction of the given cl-RA-automaton is either prefix-rewriting or suffix-rewriting. Rewriting rules, which can rewrite only prefix (suffix, respectively) of tape content, are called prefix-rewriting (suffix-rewriting, respectively). A rule (x, z, y) of a k-cl-RA-automaton such that x has prefix ¢ can rewrite only the prefix of a tape content, so it is prefix-rewriting. Similarly, each rule (x, z, y) of a k-cl-RA-automaton such that y has suffix \$ is suffix-rewriting. String rewriting systems having prefix- and suffix-rewriting rules only are called finite combined prefix- and suffix-rewriting systems [11].

Theorem 3.3.2. *Let* $M = (\Sigma, I)$ *be a* k-$\mathsf{cl\text{-}RA}$-*automaton such that for each* $(x, z, y) \in I$*:* $¢$ *is a prefix of* x *or* $\$$ *is a suffix of* y*. Then* $L(M)$ *is a regular language.*

Proof. The rewriting relation of M^D satisfying the conditions of the theorem is a finite combined prefix- and suffix rewriting system. Thus, $L(M) = L^+(M^D) = \{w \in \Sigma^* \mid \lambda \dashv_M^* w\}$ is generated from the regular language $\{\lambda\}$ using a finite combined prefix- and suffix-rewriting system. Hofbauer and Waldman in [11] have shown that finite combined prefix- and suffix-rewriting systems preserve regularity. From this it follows immediately that $L(M)$ is a regular language. $\square$

As we have already seen in Example 3.2.1, if we allow instructions with left context not starting with ¢ and right context not ending with $, then clearing restarting automata can recognize also languages which are not regular.

Also $\mathsf{cl\text{-}RA}$-automata with a one-letter alphabet without any restrictions on contexts cannot recognize more than regular languages over a one-letter alphabet.

Lemma 3.3.1. *For each* $\mathsf{cl\text{-}RA}$-*automaton* $M = (\Sigma, I)$*, where* $\Sigma = \{a\}$*,* $L(M)$ *is a regular language.*

Proof. If we replace every instruction $i = (a^x, a^z, a^y)$ in I by the instruction $i' = (¢ \cdot a^x, a^z, a^y)$ we get an equivalent $\mathsf{cl\text{-}RA}$-automaton $M' = (\Sigma, I')$ such that $L(M') = L(M)$ and $u \vdash_M v$ if and only if $u \vdash_{M'} v$. Since for each instruction $i' = (x', z', y')$ in I' the left context x' starts with $¢$, $L(M) = L(M')$ is a regular language according to Theorem 3.3.2. $\square$

On the other hand by Theorem 3.3.1, for each regular language L there exists a $\mathsf{cl\text{-}RA}$ M such that $L(M) = L \cup \{\lambda\}$. Since over a one-letter alphabet the class of regular languages equals to the class of context-free languages, we get the following corollary:

Corollary 3.3.1. *If we restrict ourselves to a one-letter alphabet, clearing restarting automata recognize exactly all context-free languages containing the empty word.*

3.4 (Non-)closure properties of $\mathcal{L}$(cl-RA)

In this section, we present several results showing that the class of languages recognized by cl-RA-automata does not contain all context-free languages and is not closed under several operations.

Theorem 3.4.1. *The language* $L_1 = \{a^n c b^n \mid n \geq 0\} \cup \{\lambda\}$ *is not recognized by any* cl-RA-*automaton.*

Proof. For a contradiction, let us suppose that there exists a k-cl-RA-automaton $M = (\Sigma, I)$ such that $L(M) = L_1$. Let m be the maximal width of instructions of M. Obviously, $a^m c b^m \in L$ implies $a^m c b^m \vdash_M^* \lambda$ and the word $a^m c b^m$ cannot be reduced to λ in a single step. On the other hand, if we erase any single nonempty continuous proper subword from the word $a^m c b^m$, then we get a word that does not belong to L_1 – a contradiction to $L(M) = L_1$. $\square$

The language L_1 can be recognized by a simple RR-automaton. Consequently, using Theorem 3.2.1 and the fact that L_1 is a context-free language we get the following:

Corollary 3.4.1.

a) $\mathcal{L}$(cl-RA) $\subset \mathcal{L}$(RR).

b) CFL $- \mathcal{L}$(cl-RA) $\neq \emptyset$.

Let $L_2 = \{a^n b^n \mid n \geq 0\}$ and $L_3 = \{a^n b^{2n} \mid n \geq 0\}$ be two sample languages. It is easy to see, that both L_2 and L_3 are recognized by some 1-cl-RA-automata.

Theorem 3.4.2. *The languages* $L_2 \cup L_3$ *and* $L_2 \cdot L_3$ *are not recognized by any* cl-RA-*automaton.*

Proof. For a contradiction, let us suppose that there exists a k-cl-RA-automaton $M = (\Sigma, I)$ such that $L(M) = L_2 \cup L_3$ ($L(M) = L_2 \cdot L_3$, respectively). Let $m > 0$ be the maximal width of instructions of M. Obviously, $a^{4m}b^{4m} \in L(M)$ and $a^{4m}b^{4m} \vdash_M^* \lambda$. Let $a^{4m}b^{4m} \vdash_M^{(i)} a^{4m-s}b^{4m-t}$ be the first step of an accepting computation of M on $a^{4m}b^{4m}$, where $s, t \geq 0, s + t > 0$ and $i = (x, z, y) \in I$. Because $|i| = |xzy| \leq m$, $|x|, |y|, |z| \geq 1$ and $a^{4m-s}b^{4m-t} \in L(M)$, it follows that $s = t$, $0 < 2s < m$ and $z = a^s b^s$. Then $a^{4m+s}b^{8m+s} \vdash_M^{(i)}= a^{4m}b^{8m} \in L(M)$ and we obtain a contradiction to the error preserving property since $a^{4m+s}b^{8m+s}$ is not in $L_2 \cup L_3$ (not in $L_2 \cdot L_3$, respectively). $\square$

Corollary 3.4.2. $\mathcal{L}(\mathsf{cl\text{-}RA})$ *is neither closed under union nor under concatenation.*

Corollary 3.4.3. $\mathcal{L}(\mathsf{cl\text{-}RA})$ *is not closed under morphism.*

Proof. Consider $L = \{a^n b^n c^m d^{2m} \mid n, m \geq 0\}$ recognized by 1-cl-RA-automaton and the morphism $h : a \mapsto a, b \mapsto b, c \mapsto a, d \mapsto b$. Then $h(L) = L_2 \cdot L_3 \notin \mathcal{L}(\mathsf{cl\text{-}RA})$. □

It is easy to see that each of the following languages:

$$\begin{array}{rcl} L_4 & = & \{a^n c b^n \mid n \geq 0\} \cup \{a^n b^n \mid n \geq 0\} \quad \text{and} \\ L_5 & = & \{a^n c b^m \mid n, m \geq 0\} \cup \{\lambda\} \end{array}$$

can be recognized by a 1-cl-RA-automaton. Using these languages we can show several additional non-closure properties of $\mathcal{L}(\mathsf{cl\text{-}RA})$.

Corollary 3.4.4. $\mathcal{L}(\mathsf{cl\text{-}RA})$ *is not closed under*

a) intersection,

b) intersection with a regular language,

c) set difference.

Proof. Part a) follows from the equality $L_1 = L_4 \cap L_5$ and Theorem 3.4.1. For proving b) just notice that L_5 is a regular language. Proof of c) is implied by the equality $L_1 = (L_4 - L_2) \cup \{\lambda\}$. □

Example 3.4.1. *The Dyck language of correct parentheses can be recognized by a* cl-RA-*automaton with a single instruction:* $(\lambda, (\,), \lambda)$. *With respect to the definition of the* 1-cl-RA-*automaton this instruction is in fact a set of instructions:* $(\{¢\} \cup \Sigma, (\,), \Sigma \cup \{\$\})$, *where* $\Sigma = \{(,)\}$.

3.5 A hierarchy with respect to the length of contexts

As we have seen in Theorem 3.1.2, by increasing the length of contexts used in instructions, k-CRS-systems could only increase their power. In this section we show that by increasing the length of contexts in instructions of clearing restarting automata we strictly increase their power. Hence, we obtain the following infinite hierarchy of language classes.

Theorem 3.5.1. $\mathcal{L}(k\text{-}\mathsf{cl\text{-}RA}) \subset \mathcal{L}((k+1)\text{-}\mathsf{cl\text{-}RA})$, *for all* $k \geq 1$.

Proof. First, we show that by increasing the length of contexts for a cl-RA-automaton we do not decrease its power. If $M = (\Sigma, I)$ is a k-cl-RA-automaton, then by Theorem 3.1.2 there exists a $(k+1)$-CRS-system $M' = (\Sigma, \Sigma, I')$ such that for each $w, w' \in \Sigma^*$:

$$w \to_M w' \quad \Leftrightarrow \quad w \to_{M'} w'$$

and both I and I' have the same set of rules. Thus M' is a $(k+1)$-cl-RA-automaton and $L(M) = L^-(M) = L^-(M') = L(M')$.

Next we show that by increasing the length of contexts of cl-RA-automata we do increase their power. For each $k \geq 1$ and the language $L_1^{(k)} = \{(c^k a c^k)^n (c^k b c^k)^n \mid n \geq 0\}$ the following holds:

$$L_1^{(k)} \in \mathcal{L}((k+1)\text{-}\mathsf{cl\text{-}RA}) - \mathcal{L}(k\text{-}\mathsf{cl\text{-}RA}) \ .$$

It is easy to see that $L_1^{(k)}$ is recognized by the following $(k+1)$-cl-RA-automaton $M_1^{(k+1)} = (\{a, b, c\}, I_1^{(k+1)})$ with instructions:

$$\begin{array}{ll} (1) & (ac^k, c^k a c^k c^k b c^k, c^k b), \\ (2) & (¢, c^k a c^k c^k b c^k, \$). \end{array}$$

Assume to the contrary that there is a k-cl-RA-automaton $M = (\{a, b, c\}, I)$ recognizing the language $L_1^{(k)}$. Let m be the maximal width of an instruction in I. The word $w = (c^k a c^k)^m (c^k b c^k)^m$ is accepted by M. Let us inspect the first reduction of an accepting computation for w: $w \vdash_M^{(i)} w'$. Then in the instruction $i = (x, z, y)$ used in this reduction the deleted part z must be of the form $c^r a c^k (c^k a c^k)^n (c^k b c^k)^n c^k b c^s$ for some $n, 0 \leq n < \frac{m}{6}$, $r, s \geq 0$, $r + s = 2k$. The left context x must contain at least one symbol a, otherwise the instruction i could be applied to the word $w'' = (c^k a c^k)^m (c^k b c^k) (c^k a c^k)^{n+1} (c^k b c^k)^{n+1} (c^k b c^k)^{m-1} \notin L_1^{(k)}$ and we would obtain $w'' \vdash_M^{(i)} w \in L_1^{(k)}$, which contradicts the error preserving property (Theorem 3.2.2). Similarly, the right context y must contain at least one symbol b. Then obviously $|xzy| \geq 2k + 2 + |z|$ and $|x| = |y| \geq k + 1$, which is a contradiction. $\square$

3.6 4-cl-RA-automata recognizing non-context-free language

We have seen that cl-RA-automata can recognize some context-free languages. In the following we show that they can even recognize some non-context-free languages. However, since $\mathcal{L}(\textsf{cl-RA}) \subset \mathcal{L}(\textsf{RR})$ and RR-automata can be simulated by linear bounded automata ([16]), we have the following:

Corollary 3.6.1. $\mathcal{L}(\textsf{cl-RA}) \subset \textsf{CSL}$*, where* CSL *denotes the class of context-sensitive languages.*

In order to construct a cl-RA-automaton recognizing a non-context-free language we first describe a scheme for designing machine learning algorithms for cl-RA-automata. Subsequently we apply this scheme for inferring the desired 4-cl-RA-automaton.

Knowing some reductions that can be made by an unknown clearing restarting automaton M, we can often infer the instructions of the k-cl-RA-automaton M. Let $w_1 \vdash_M w'_1, \ldots, w_n \vdash_M w'_n$, where $w_i, w'_i \in \Sigma^*$ for $i = 1, \ldots, n$, $n > 0$, be a list of known reductions. A meta-algorithm for machine learning of unknown clearing restarting automaton can be outlined as follows:

Meta-Algorithm 3.6.1. *Learning a clearing restarting automaton from a set of sample reductions:*

Step 1 $k := 1$.

Step 2 *For each reduction* $w_i \vdash_M w'_i$ *nondeterministically choose a factorization of* w_i *such that* $w_i = \alpha_i \beta_i \gamma_i$ *and* $w'_i = \alpha_i \gamma_i$.

Step 3 *Construct the clearing restarting automaton* $M = (\Sigma, I)$*, where*

$$I = \bigcup_{i=1}^{n} \{(\mathit{Suff}_k(\text{¢}\alpha_i), \beta_i, \mathit{Pref}_k(\gamma_i \$))\}.$$

Step 4 *Test the automaton* M *using any available information, e.g. some negative samples of words not belonging to* $L(M)$.

Step 5 *If the automaton passed all the tests, return* M *otherwise try another factorization of the known reductions and continue by Step 3. If all possible factorizations have been tried, then increase* k *and continue by Step 2.*

Although Step 2 is nondeterministic, for many sample reductions the factorization in this step is unambiguous. E.g. for the set of sample reductions $\{aabb \vdash_M ab, ab \vdash_M \lambda\}$ we obtain for $k = 1$ only one set of instructions $I = \{(a, ab, b), (¢, ab, \$)\}$, which is exactly the set of instructions of the automaton M from Example 3.2.1 recognizing the language $L = \{a^n b^n \mid n \geq 0\}$.

Even if the algorithm is very simple, it can be used to infer non-trivial clearing restarting automata.

In [14] there was presented a very restricted restarting automaton (deterministic restarting automaton which can delete only) that recognizes a non-context-free language. It is possible to construct a cl-RA-automaton recognizing the same language using Meta-Algorithm 3.6.1 on reductions collected from a sample computation.

Theorem 3.6.1. *There exists a k-cl-RA-automaton M recognizing a language that is not context-free.*

Idea of the proof: We will construct a k-cl-RA-automaton $M = (\Sigma, I)$, where $\Sigma = \{a, b\}$, such that $L(M) \cap \{(ab)^n \mid n > 0\} = \{(ab)^{2^l} \mid l \geq 0\}$. Since context-free languages are closed under intersection with regular languages, it follows that $L(M)$ is a non-context-free language.

The respective automaton from [14] accepts the word $(ab)^8$ by the following sequence of reductions:

$$\begin{array}{l}
ababababababab\underline{a}b \vdash_M ababababab\underline{a}babb \vdash_M ababab\underline{a}babbabb \vdash_M \\
ab\underline{a}babbabbabb \vdash_M abbabbabba\underline{b}b \vdash_M abbabba\underline{b}bab \vdash_M \\
abba\underline{b}babab \vdash_M a\underline{b}bababab \vdash_M ababab\underline{a}b \vdash_M \\
ab\underline{a}babb \vdash_M abba\underline{b}b \vdash_M a\underline{b}bab \vdash_M \\
ab\underline{a}b \vdash_M a\underline{b}b \vdash_M \underline{ab} \vdash_M \lambda \text{ accept.}
\end{array}$$

From this sample computation, we can collect 15 reductions and use them as an input to an algorithm based on Meta-Algorithm 3.6.1. All these reductions have unambiguous factorizations (the deleted symbols are underlined). Hence, the only variable we have to choose is k – the length of the context of the instructions. In the following, we use the abbreviation for sets of instructions introduced in Remark 3.1.1.

1. For $k = 1$ we get the following 3 instructions:
   (b, a, b), (a, b, b), $(¢, ab, \$)$.

Then, however, the automaton would accept the word $ababab$, which does not belong to L: $abab\underline{a}b \vdash ab\underline{a}bb \vdash a\underline{b}bb \vdash a\underline{b}b \vdash \underline{ab} \vdash \lambda$.

2. For $k = 2$ we get the following set of instructions:
   $(ab, \underline{a}, \{b\$, ba\}), \quad (\{¢a, ba\}, \underline{b}, \{b\$, ba\}), \quad (¢, \underline{ab}, \$)$.
   Then, however, the automaton would accept the word $ababab$ that does not belong to L: $abab\underline{a}b \vdash aba\underline{b}b \vdash ab\underline{a}b \vdash a\underline{b}b \vdash \underline{ab} \vdash \lambda$.

3. For $k = 3$ we get the following set of instructions:
   $(\{¢ab, bab\}, \underline{a}, \{b\$, bab\}), \quad (\{¢a, bba\}, \underline{b}, \{b\$, bab\}), \quad (¢, \underline{ab}, \$)$.
   But then again the automaton would accept the word $ababab$, which does not belong to L: $ab\underline{a}bab \vdash a\underline{b}bab \vdash ab\underline{a}b \vdash a\underline{b}b \vdash \underline{ab} \vdash \lambda$.

4. For $k = 4$ we get the following set of instructions:
   $(\{¢ab, abab\}, a, \{b\$, babb\}), \quad (\{¢a, abba\}, b, \{b\$, bab\$, baba\}), \quad (¢, ab, \$)$.
   Let us show that this automaton has the required properties.

Let us denote the instructions I of our 4-cl-RA $M = (\Sigma, I)$ in the following way:

$a_1 = (¢ab, \underline{a}, b\$)$,
$a_2 = (¢ab, \underline{a}, babb)$,
$a_3 = (abab, \underline{a}, b\$)$,
$a_4 = (abab, \underline{a}, babb)$,
$b_1 = (¢a, \underline{b}, b\$)$,
$b_2 = (¢a, \underline{b}, bab\$)$,
$b_3 = (¢a, \underline{b}, baba)$,
$b_4 = (abba, \underline{b}, b\$)$,
$b_5 = (abba, \underline{b}, bab\$)$,
$b_6 = (abba, \underline{b}, baba)$,
$c_0 = (¢, \underline{ab}, \$)$.

In the rest of this section we prove that $L(M)$ is not a context-free language. We use the generative approach described in Remark 3.2.2. The corresponding set of dual instructions I^D can be outlined as (the little vertical arrows in the left-hand sides of these productions mark the positions where some symbols are inserted):

$A_1 : ¢ab^{\downarrow}b\$ \dashv_M ¢ab\underline{a}b\$$,
$A_2 : ¢ab^{\downarrow}babb \dashv_M ¢ab\underline{a}babb$,
$A_3 : abab^{\downarrow}b\$ \dashv_M abab\underline{a}b\$$,
$A_4 : abab^{\downarrow}babb \dashv_M abab\underline{a}babb$,
$B_1 : ¢a^{\downarrow}b\$ \dashv_M ¢a\underline{b}b\$$,
$B_2 : ¢a^{\downarrow}bab\$ \dashv_M ¢a\underline{b}bab\$$,
$B_3 : ¢a^{\downarrow}baba \dashv_M ¢a\underline{b}baba$,
$B_4 : abba^{\downarrow}b\$ \dashv_M abba\underline{b}b\$$,
$B_5 : abba^{\downarrow}bab\$ \dashv_M abba\underline{b}bab\$$,
$B_6 : abba^{\downarrow}baba \dashv_M abba\underline{b}baba$,
$C_0 : ¢^{\downarrow}\$ \dashv_M ¢\underline{ab}\$$.

Now observe that each word generated by M is of the form:

$$w = (ab)^{x_0}(abb)^{y_0} \dots (ab)^{x_n}(abb)^{y_n}$$

where $n \geq 0, x_0 \geq 0, y_0 > 0, \quad x_1 > 0, y_1 > 0, \quad \dots, \quad x_n > 0, y_n \geq 0$. In the case $n = 0$ we mean for x_0 and y_0 only the inequalities $x_0 \geq 0, y_0 \geq 0$. This is because each production preserves this form. This form allows us to define the so-called *circle-square representation* of w as $\bigcirc^{x_0}\Box^{y_0} \dots \bigcirc^{x_n}\Box^{y_n}$ where each circle $\bigcirc$ represents ab and each square $\Box$ represents abb. If we rewrite the productions of M in this circle-square representation, we get:

$$\begin{array}{ll}
A'_1 : \text{¢}\underline{\Box}\$ \dashv_M \text{¢}\bigcirc\bigcirc\$, & B'_3 : \text{¢}\underline{\bigcirc}\bigcirc? \dashv_M \text{¢}\Box\bigcirc?, \\
A'_2 : \text{¢}\underline{\Box}\Box \dashv_M \text{¢}\bigcirc\bigcirc\Box, & B'_4 : \Box\underline{\bigcirc}\$ \dashv_M \Box\Box\$, \\
A'_3 : \bigcirc\underline{\Box}\$ \dashv_M \bigcirc\bigcirc\bigcirc\$, & B'_5 : \Box\underline{\bigcirc}\bigcirc\$ \dashv_M \Box\Box\bigcirc\$, \\
A'_4 : \bigcirc\underline{\Box}\Box \dashv_M \bigcirc\bigcirc\bigcirc\Box, & B'_6 : \Box\underline{\bigcirc}\bigcirc? \dashv_M \Box\Box\bigcirc?, \\
B'_1 : \text{¢}\underline{\bigcirc}\$ \dashv_M \text{¢}\Box\$, & C'_0 : \text{¢}^{\downarrow}\$ \dashv_M \text{¢}\bigcirc\$, \\
B'_2 : \text{¢}\underline{\bigcirc}\bigcirc\$ \dashv_M \text{¢}\Box\bigcirc\$, &
\end{array}$$

where the symbol ? represents either the symbol $\bigcirc$, or the symbol $\Box$.

Lemma 3.6.1. *Consider a* 1-CRS*-system* $R = (\Theta, \Theta, J)$*, where* $\Theta = \{\bigcirc, \Box\}$*, with instructions:*

$$\begin{array}{ll}
(0) & (\text{¢}, \lambda \to \bigcirc, \$), \\
(1) & (\{\text{¢}, \Box\}, \underline{\bigcirc} \to \Box, \{\bigcirc, \$\}), \textit{ and} \\
(2) & (\{\text{¢}, \bigcirc\}, \underline{\Box} \to \bigcirc\bigcirc, \{\Box, \$\}).
\end{array}$$

Then for each instruction $i \in I^D$*:* i *can be applied to the word*

$$w = (ab)^{x_0}(abb)^{y_0} \dots (ab)^{x_n}(abb)^{y_n}$$

if and only if there exists $j \in J$ *which can be applied in the same way to the word*

$$\bigcirc^{x_0}\Box^{y_0} \dots \bigcirc^{x_n}\Box^{y_n}.$$

Thus the circle-square representation of the language $L(M)$ *is equal to* $L^+(R)$*.*

Proof. There is one-to-one correspondence between the productions A_1, A_2, A_3, A_4, B_1, B_4, C_0 from I^D and instructions A'_1, A'_2, A'_3, A'_4, B'_1, B'_4, C'_0 from J. Now observe that the productions B_2 and B_3 correspond to the instruction $(\text{¢}, \underline{\bigcirc} \to \Box, \bigcirc)$ of R, and similarly the productions B_5 and B_6 correspond to the instruction $(\Box, \underline{\bigcirc} \to \Box, \bigcirc)$ of R. $\Box$

The following theorem characterizes the languages recognized by a slightly generalized version of the 1-CRS-system from the previous lemma. Moreover, we will use this theorem also in another section in a different situation.

Theorem 3.6.2. *Consider a* 1-CRS*-system* $R = (\Theta, \Theta, J)$*, where* $\Theta = \{\bigcirc, \Box\}$*, with instructions:*

$$\begin{array}{ll}
(0) & (\text{¢}, \lambda \to \bigcirc^m, \$), \\
(1) & (\{\text{¢}, \Box\}, \underline{\bigcirc} \to \Box^\alpha, \{\bigcirc, \$\}), \\
(2) & (\{\text{¢}, \bigcirc\}, \underline{\Box} \to \bigcirc^\beta, \{\Box, \$\}),
\end{array}$$

where $m, \alpha, \beta \geq 1$ *are integer constants. Then each* $w \in L^+(R)$ *is of the form:*

$$w = \bigcirc^{x_0} \Box^{y_0} \bigcirc^{x_1} \Box^{y_1} \ldots \bigcirc^{x_k} \Box^{y_k},$$

where:

$$\begin{gathered}
k \geq 0, x_0 \geq 0, y_0 > 0, \quad x_1 > 0, y_1 > 0, \quad \ldots, \quad x_k > 0, y_k \geq 0, \\
\alpha^0\beta^0 x_0 + \alpha^0\beta^1 y_0 + \alpha^1\beta^1 x_1 + \alpha^1\beta^2 y_1 + \ldots + \alpha^k\beta^k x_k + \alpha^k\beta^{k+1} y_k = (\alpha\beta)^l m
\end{gathered} \tag{$*$}$$

for some $l \geq 1$*. We call this sum the overall sum of the word* w*. Note that in the case* $k = 0$ *we mean for* x_0 *and* y_0 *only the inequalities* $x_0 \geq 0, y_0 \geq 0$*.*

Proof. (By induction on the number of used instructions)

The word $\bigcirc^m$ evidently satisfies all conditions of the theorem, i.e. $k = 0$, $x_0 = m$, $y_0 = 0$, and $\alpha^0\beta^0 x_0 + \alpha^0\beta^1 y_0 = (\alpha\beta)^0 m$.

Suppose that we have w of the form $w = \bigcirc^{x_0} \Box^{y_0} \bigcirc^{x_1} \Box^{y_1} \ldots \bigcirc^{x_k} \Box^{y_k}$ satisfying all conditions $(*)$ of the theorem. Depending on the last used instruction, there are several cases we have to consider:

1. $(\text{¢}, \underline{\bigcirc} \to \Box^\alpha, \$)$: can be applied only to the word $w = \bigcirc$, i.e. $m = 1$. We get $w' = \Box^\alpha$ satisfying all conditions $(*)$, i.e. $k = 0$, $x_0 = 0$, $y_0 = \alpha$, and $\alpha^0\beta^0 x_0 + \alpha^0\beta^1 y_0 = \alpha\beta = (\alpha\beta)m$.

2. $(\text{¢}, \underline{\bigcirc} \to \Box^\alpha, \bigcirc)$: can be applied to $w = \underline{\bigcirc}\bigcirc^{x_0-1} \Box^{y_0} \bigcirc^{x_1} \Box^{y_1} \ldots \bigcirc^{x_k} \Box^{y_k}$, where $x_0 \geq 2$. We get:

   $$w' = \Box^\alpha \bigcirc^{x_0-1} \Box^{y_0} \bigcirc^{x_1} \Box^{y_1} \ldots \bigcirc^{x_k} \Box^{y_k} = \bigcirc^{x'_0} \Box^{y'_0} \bigcirc^{x'_1} \Box^{y'_1} \ldots \bigcirc^{x'_{k+1}} \Box^{y'_{k+1}},$$

   where $x'_0 = 0$, $y'_0 = \alpha$, $x'_1 = x_0 - 1$, $y'_1 = y_0$, and $x'_{i+1} = x_i$, $y'_{i+1} = y_i$ for all $i \in \{1, 2, \ldots, k\}$. Thus
   $\alpha^0\beta^0 x'_0 + \alpha^0\beta^1 y'_0 + \alpha^1\beta^1 x'_1 + \alpha^1\beta^2 y'_1 + \ldots \alpha^{k+1}\beta^{k+1} x'_{k+1} + \alpha^{k+1}\beta^{k+2} y'_{k+1} = \alpha^0\beta^0 \times 0 + \alpha^0\beta^1 \times \alpha - \alpha^1\beta^1 \times 1 + \alpha^1\beta^1(\alpha^0\beta^0 x_0 + \alpha^0\beta^1 y_0 + \ldots \alpha^k\beta^k x_k + \alpha^k\beta^{k+1} y_k) = (\alpha\beta) \times (\alpha\beta)^l m = (\alpha\beta)^{l+1} m$.

3. $(\square, \underline{\bigcirc} \to \square^{\alpha}, \$)$: can be applied to $w = \bigcirc^{x_0}\square^{y_0}\ldots\bigcirc^{x_{k-1}}\square^{y_{k-1}}\underline{\bigcirc}$, where $k \geq 1$, $y_{k-1} \geq 1$, $x_k = 1$, $y_k = 0$. We get:

$$w' = \bigcirc^{x_0}\square^{y_0}\ldots\bigcirc^{x_{k-1}}\square^{y_{k-1}}\square^{\alpha} = \bigcirc^{x'_0}\square^{y'_0}\ldots\bigcirc^{x'_{k-1}}\square^{y'_{k-1}},$$

where $x'_i = x_i$, $y'_i = y_i$ for all $i \in \{0, 1, \ldots, k-2\}$, and $x'_{k-1} = x_{k-1}$, $y'_{k-1} = y_{k-1} + \alpha$. Thus
$\alpha^0\beta^0x'_0+\alpha^0\beta^1y'_0+\alpha^1\beta^1x'_1+\alpha^1\beta^2y'_1+\ldots+\alpha^{k-1}\beta^{k-1}x'_{k-1}+\alpha^{k-1}\beta^{k}y'_{k-1} = \alpha^0\beta^0x_0+\alpha^0\beta^1y_0+\alpha^1\beta^1x_1+\alpha^1\beta^2y_1+\ldots+\alpha^{k-1}\beta^{k-1}x_{k-1}+\alpha^{k-1}\beta^{k}(y_{k-1}+\alpha) = \alpha^0\beta^0x_0+\alpha^0\beta^1y_0+\ldots+\alpha^{k-1}\beta^{k-1}x_{k-1}+\alpha^{k-1}\beta^{k}y_{k-1}+\alpha^k\beta^kx_k+\alpha^k\beta^{k+1}y_k = (\alpha\beta)^lm$.

4. $(\square, \underline{\bigcirc} \to \square^{\alpha}, \bigcirc)$: can be applied to a subword $u = \square\underline{\bigcirc}\bigcirc$ of the word w. We get a new subword $u' = \square\square^{\alpha}\bigcirc$ and the corresponding w'. The overall sum does not change, since each $\square$ in both subwords contributes to this sum with the weight $\alpha^i\beta^{i+1}$ for some $i \geq 0$ and each $\bigcirc$ contributes to this sum with the weight $\alpha^{i+1}\beta^{i+1}$. Thus w' satisfies all conditions $(*)$ of the theorem.

5. $(\text{¢}, \underline{\square} \to \bigcirc^{\beta}, \$)$: can be applied only to the word $w = \square = \bigcirc^{x_0}\square^{y_0}$, where $k = 0$, $x_0 = 0$, $y_0 = 1$. We get $w' = \bigcirc^{\beta} = \bigcirc^{x'_0}\square^{y'_0}$, where $x'_0 = \beta$, $y'_0 = 0$, and the overall sum $\alpha^0\beta^0x'_0 + \alpha^0\beta^1y'_0 = \beta = \alpha^0\beta^0x_0 + \alpha^0\beta^1y_0$ remains unchanged.

6. $(\text{¢}, \underline{\square} \to \bigcirc^{\beta}, \square)$: can be applied to $w = \underline{\square}\square^{y_0-1}\bigcirc^{x_1}\square^{y_1}\ldots\bigcirc^{x_k}\square^{y_k}$, where $x_0 = 0$ and $y_0 \geq 2$. We get:

$$w' = \bigcirc^{\beta}\square^{y_0-1}\bigcirc^{x_1}\square^{y_1}\ldots\bigcirc^{x_k}\square^{y_k} = \bigcirc^{x'_0}\square^{y'_0}\bigcirc^{x'_1}\square^{y'_1}\ldots\bigcirc^{x'_k}\square^{y'_k},$$

where $x'_0 = \beta$, $y'_0 = y_0 - 1$, and $x'_i = x_i$, $y'_i = y_i$ for all $i \in \{1, 2, \ldots, k\}$. Thus
$\alpha^0\beta^0x'_0+\alpha^0\beta^1y'_0+\alpha^1\beta^1x'_1+\alpha^1\beta^2y'_1+\alpha^k\beta^kx'_k+\alpha^k\beta^{k+1}y'_k = \alpha^0\beta^0\times\beta+\alpha^0\beta^1(y_0-1)+\alpha^1\beta^1x_1+\alpha^1\beta^2y_1+\ldots\alpha^k\beta^kx_k+\alpha^k\beta^{k+1}y_k = (\alpha\beta)^lm$.

7. $(\bigcirc, \underline{\square} \to \bigcirc^{\beta}, \$)$: can be applied to $w = \bigcirc^{x_0}\square^{y_0}\bigcirc^{x_1}\square^{y_1}\ldots\bigcirc^{x_k}\underline{\square}$, where $x_k \geq 1$, $y_k = 1$. We get

$$w' = \bigcirc^{x_0}\square^{y_0}\bigcirc^{x_1}\square^{y_1}\ldots\bigcirc^{x_k}\bigcirc^{\beta} = \bigcirc^{x'_0}\square^{y'_0}\bigcirc^{x_1}\square^{y_1}\ldots\bigcirc^{x'_k}\square^{y'_k},$$

where $x'_i = x_i$, $y'_i = y_i$ for all $i \in \{0, 1, \ldots, k-1\}$, $x'_k = x_k + \beta$, and $y'_k = 0$. Thus

$\alpha^0\beta^0x'_0+\alpha^0\beta^1y'_0+\ldots\alpha^{k-1}\beta^{k-1}x'_{k-1}+\alpha^{k-1}\beta^ky'_{k-1}+\alpha^k\beta^kx'_k+\alpha^k\beta^{k+1}y'_k =$

$\alpha^0\beta^0x_0 + \alpha^0\beta^1y_0 + \ldots\alpha^{k-1}\beta^{k-1}x_{k-1} + \alpha^{k-1}\beta^ky_{k-1} + \alpha^k\beta^k(x_k + \beta) = (\alpha\beta)^lm$.

8. $(\bigcirc, \underline{\square} \to \bigcirc^\beta, \square)$: can be applied to a subword $u = \bigcirc\underline{\square}\square$ of the word w. We get a new subword $u' = \bigcirc\bigcirc^\beta\square$ and the corresponding w'. The overall sum does not change, since each $\bigcirc$ in both subwords contributes to this sum with the weight $\alpha^i\beta^i$ for some $i \geq 0$ and each $\square$ contributes to this sum with the weight $\alpha^i\beta^{i+1}$. Thus w' satisfies all conditions $(*)$ of the theorem.

$\square$

Theorem 3.6.3. *Let R be a* 1-CRS-*system from Theorem 3.6.2. Then*

$$L^+(R) \cap \{\bigcirc\}^+ = \{\bigcirc^x \mid x = (\alpha\beta)^lm,\ l = 0, 1, 2, \ldots\}.$$

Proof. If $\bigcirc^x \in L^+(R)$, then by Theorem 3.6.2 necessarily $x = (\alpha\beta)^lm$ for some $l \geq 0$. On the other hand by using the instructions of R we can easily generate any such word $\bigcirc^x$, where $x = (\alpha\beta)^lm$. For instance, consider the following derivation:
$\bigcirc^m = \underline{\bigcirc}\bigcirc^{m-1} \to_R \square^\alpha\underline{\bigcirc}\bigcirc^{m-2} \to_R \square^\alpha\square^\alpha\underline{\bigcirc}\bigcirc^{m-3} \to_R^* \square^{\alpha(m-1)}\underline{\bigcirc} \to_R \square^{\alpha m} = \underline{\square}\square^{\alpha m-1} \to_R \bigcirc^\beta\underline{\square}\square^{\alpha m-2} \to_R \bigcirc^\beta\bigcirc^\beta\underline{\square}\square^{\alpha m-3} \to_R^* \bigcirc^{\beta(\alpha m-1)}\underline{\square} \to_R \bigcirc^{(\alpha\beta)m}$.
Thus $\bigcirc^m \to_R^* \bigcirc^{(\alpha\beta)m}$, which immediately implies that $\bigcirc^m \to_R^* \bigcirc^{(\alpha\beta)^lm}$ for any $l \geq 0$. $\square$

If we apply Theorem 3.6.3 to the circle-square representation of $L(M)$, i.e. $m = 1$, $\alpha = 1$, and $\beta = 2$, we get the following result:

Theorem 3.6.4. $L(M) \cap \{(ab)^n \mid n > 0\} = \{(ab)^{2^l} \mid l = 0, 1, 2, \ldots\}$.

As context-free languages are closed under intersection with regular languages, $L(M)$ is not a context-free language.

3.7 1-cl-RA-automata

Let $M = (\Sigma, I)$ be a cl-RA-automaton, $l \in ¢ \cdot \Sigma^* \cup \Sigma^*$, and $r \in \Sigma^* \cup \Sigma^* \cdot \$$. Let us denote $L_{(l,r)}(M) = \{w \in \Sigma^* \mid w \vdash_M^* \lambda$ in the context $(l, r)\}$, i.e. $L_{(l,r)}(M)$ is the reduction language $L^-(M)$ in the context (l, r) (Definition 3.1.1). Obviously, $L(M) = L_{(¢,\$)}(M)$. If the cl-RA-automaton M is known from the context, then we use the abbreviation $L_{(l,r)} = L_{(l,r)}(M)$.

Example 3.7.1. *Suppose that we have* 1-cl-RA-*automaton* $M = (\Sigma, I)$, *where* $\Sigma = \{a, b\}$ *and the instructions are* $I = \{(¢, ab, \$), (a, ab, b)\}$. *Of course, we have*

$L_{(¢,\$)}(M) = \{\lambda\} \cup a \cdot L_{(a,b)}(M) \cdot b$ *and*

$L_{(a,b)}(M) = \{\lambda\} \cup a \cdot L_{(a,b)}(M) \cdot b$.

We can rewrite these language equations into the context-free grammar $G = (V_N, V_T, S, P)$ *with* $V_N = \{S, X\}$, $V_T = \Sigma$, *and the following set of production rules:*

$S \to \lambda \mid aXb$,

$X \to \lambda \mid aXb$.

We can generalize this technique to any 1-cl-RA-*automaton.*

Lemma 3.7.1. *Suppose that we have* 1-cl-RA-*automaton* $M = (\Sigma, I)$. *Then for each* $l \in LC_1 = \{¢\} \cup \Sigma$, *and* $r \in RC_1 = \Sigma \cup \{\$\}$*:*

$$L_{(l,r)} = \{\lambda\} \cup \bigcup_{(l,u_1\ldots u_n,r)\in I} L_{(l,u_1)} \cdot u_1 \cdot L_{(u_1,u_2)} \cdot u_2 \ldots u_{n-1} \cdot L_{(u_{n-1},u_n)} \cdot u_n \cdot L_{(u_n,r)} \ .$$

Proof. Let R denote the right-hand side of this equation. Suppose $w \in L_{(l,r)}$. If $w = \lambda$, then $w \in R$. If $w \neq \lambda$, then $w \vdash_M^* w_0 \vdash_M \lambda$ in the context (l, r). Suppose $w_0 = u_1 \ldots u_n$. Then, clearly, $(l, u_1 \ldots u_n, r) \in I$ and $w = z_0 u_1 z_1 u_2 z_2 \ldots u_n z_n$ for some $z_0, \ldots, z_n \in \Sigma^*$ such that $z_0 \vdash_M^* \lambda$ in the context (l, u_1), $z_1 \vdash_M^* \lambda$ in the context (u_1, u_2), etc., $z_n \vdash_M^* \lambda$ in the context (u_n, r). This is equivalent to $z_0 \in L_{(l,u_1)}$, $z_1 \in L_{(u_1,u_2)}$, ..., $z_n \in L_{(u_n,r)}$. Thus w is in R.

Suppose $w \in R$. If $w = \lambda$, then $w \in L_{(l,r)}$. Otherwise suppose $w = z_0 u_1 z_1 u_2 z_2 \ldots u_n z_n$ for some $(l, u_1 \ldots u_n, r) \in I$ and $z_0, \ldots, z_n \in \Sigma^*$ such that $z_0 \in L_{(l,u_1)}$, $z_1 \in L_{(u_1,u_2)}$,..., $z_n \in L_{(u_n,r)}$. Obviously, $w = \underline{z_0} u_1 z_1 u_2 z_2 \ldots u_n z_n \vdash_M^* u_1 \underline{z_1} u_2 z_2 \ldots u_n z_n \vdash_M^* u_1 u_2 \underline{z_2} \ldots u_n z_n \vdash_M^* \ldots \vdash_M^* u_1 u_2 \ldots u_n \underline{z_n} \vdash_M^* u_1 u_2 \ldots u_n \vdash_M \lambda$ in the context (l, r) and hence $w \in L_{(l,r)}$. ☐

Since we can easily rewrite the language equations from Lemma 3.7.1 into production rules of a context-free grammar with the initial nonterminal corresponding to $L_{(¢,\$)}$, we get together with Theorem 3.4.1 the following corollary:

Corollary 3.7.1. $\mathcal{L}(\text{1-cl-RA}) \subset \mathsf{CFL}$.

3.8 2-cl-RA-automata recognizing non-context-free language

In the following, we use the generative approach for clearing restarting automata as it was described in Remark 3.2.2. We restrict ourselves to 2-cl-RA-automata only.

Definition 3.8.1. *Let* Λ *be a finite nonempty set of so-called* basic symbols. *We define a set of so-called* wave symbols *as* $wave(\Lambda) = \{\overset{uv}{\sim} \mid u, v \in \Lambda\}$. *Let* $\Sigma = \Lambda \cup wave(\Lambda)$.

We say that a couple $(x, y) \in \Sigma \times \Sigma$ *satisfies the* ether property *if and only if the couple* (x, y) *is one of the following types:*

$$\begin{array}{lll} (1) & (\overset{uv}{\sim}, \overset{vw}{\sim}) & \textit{for some } u, v, w \in \Lambda, \\ (2) & (\overset{uv}{\sim}, v) & \textit{for some } u, v \in \Lambda, \textit{ or} \\ (3) & (u, \overset{uv}{\sim}) & \textit{for some } u, v \in \Lambda. \end{array}$$

For each wave symbol $x = \overset{uv}{\sim}$ *let us define* $left(x) = u$, $right(x) = v$. *For* $x \in \Lambda$ *let us define* $left(x) = right(x) = x$. *Hence, a couple* $(x, y) \in \Sigma \times \Sigma$ *satisfies the* ether property *if and only if* $right(x) = left(y)$ *and at least one of the symbols* x, y *is a wave symbol.*

We say that a word $w = x_1 x_2 \dots x_n \in \Sigma^*$ *satisfies the* ether property *if and only if all couples* (x_1, x_2), (x_2, x_3), …, (x_{n-1}, x_n) *satisfy the ether property. We also say that the word* w *is an* ether *and for clarity we often mark this property by a line above the ether, e.g.* $\overline{a \overset{ab}{\sim} \overset{bc}{\sim} a \overset{ab}{\sim} b \overset{bc}{\sim} c\ a}$.

First, we will informally describe the motivation behind Definition 3.8.1. Suppose that we have two special symbols $X, Y \in \Lambda$ – a so-called *membranes* – connected by an ether, i.e. a word satisfying the ether property, for instance $\overline{X \overset{Xe}{\sim} \overset{ee}{\sim} \overset{eY}{\sim} Y}$. We want to send some signal from X to Y through the ether. For technical reasons, the signal is represented by two special basic symbols $a, \tilde{a} \in \Lambda$. After sending a signal, we want to recover the ether between X and Y in order to be able to send another signal. Our goal is to simulate this process by using a 2-cl-RA-automaton M. The choice of the term membrane for the symbols X and Y is motivated by the fact that a membrane usually serves as a separator of two spaces. In our case the membranes X and Y are on the border of the space filled by the ether, but in general we can have more membranes such that each two consecutive membranes are connected by different ethers.

We can schematically describe the process of spreading a signal from left to right in the ether as:

$$\sigma\overline{xyz} \dashv_M \sigma x \sigma' \overline{yz} \ ,$$

where $\sigma, \sigma' \in \{a, \tilde{a}\}$. We call this step a *jump* of the signal σ *from left to right.* The jump can occur only if the couple (σ, x) does not satisfy the ether property and both couples (x, y) and (y, z) do satisfy the ether property. We have to choose σ' in such a way that neither (x, σ'), nor (σ', y) satisfy the ether property. It is easy to see that we can always choose such $\sigma' \in \{a, \tilde{a}\}$. If $right(x) = left(y) = a$, then choose $\sigma' = \tilde{a}$. Otherwise, if $right(x) = left(y) \neq a$, then choose $\sigma' = a$.

Now observe that we can simulate the jump of the signal σ from left to right by the instruction:

$$(\sigma x, \sigma', yz) \ ,$$

which is a legal instruction of a 2-cl-RA-automaton.

Example 3.8.1. *In this example we demonstrate the process of sending a signal from X to Y:*

$$\begin{array}{ll}
(1) & \overline{X \overset{Xe}{\sim} \overset{ee}{\sim} \overset{eY}{\sim} Y} \dashv_M \\
(2) & X \ \underline{a} \ \overline{\overset{Xe}{\sim} \overset{ee}{\sim} \overset{eY}{\sim} Y} \dashv_M \\
(3) & X \ a \overset{Xe}{\sim} \underline{a} \ \overline{\overset{ee}{\sim} \overset{eY}{\sim} Y} \dashv_M \\
(4) & X \ a \overset{Xe}{\sim} a \overset{ee}{\sim} \underline{a} \ \overline{\overset{eY}{\sim} Y} \dashv_M \\
(5) & X \ a \overset{Xe}{\sim} a \overset{ee}{\sim} a \overset{eY}{\sim} \underline{a} \ Y \ .
\end{array}$$

Note that in the first step (1) → (2) *the symbol X initiated sending a signal a to Y. On the other hand, in the last step* (4) → (5) *the symbol Y received the signal a from X. Only in the steps* (2) → (3) *and* (3) → (4) *we used the jump operation. Now we have to recover the ether between X and Y to be able to send another signal.*

We can schematically describe the process of recovering an ether from left to right as:

$$\overline{xy}zd \dashv_M \overline{xy \overset{uv}{\sim} z}d,$$

where $u = right(y)$ and $v = left(z)$. We call this step a *recovery* of the ether *from left to right.* The recovery can occur only if the couple (x, y) satisfies the ether property and neither (y, z), nor (z, d) satisfy the ether property.

Once again, observe that we can simulate the recovery of the ether from left to right by the instruction:

$$(xy, \overset{uv}{\sim}, zd),$$

which is a legal instruction of a 2-cl-RA-automaton.

Example 3.8.2. *In this example, we demonstrate the process of recovering the ether from X to Y. We continue from the previous Example 3.8.1.*

$$\begin{array}{ll}
(5) & \underline{X\; a \overset{Xe}{\sim}}\; a \overset{ee}{\sim} a \overset{eY}{\sim} a\; Y \dashv_M \\
(6) & \underline{X \overset{Xa}{\sim} a \;\overset{Xe}{\sim}\; a}\; \overset{ee}{\sim}\; a\; \overset{eY}{\sim}\; a\; Y \dashv_M \\
(7) & \underline{X \overset{Xa}{\sim} a \overset{aX}{\sim} \overset{Xe}{\sim} a\; \overset{ee}{\sim}}\; a\; \overset{eY}{\sim}\; a\; Y \dashv_M \\
(8) & \underline{X \overset{Xa}{\sim} a \overset{aX}{\sim} \overset{Xe}{\sim} \overset{ea}{\sim} a\; \overset{ee}{\sim}\; a}\; \overset{eY}{\sim}\; a\; Y \dashv_M \\
(9) & \underline{X \overset{Xa}{\sim} a \overset{aX}{\sim} \overset{Xe}{\sim} \overset{ea}{\sim} a \overset{ae}{\sim} \overset{ee}{\sim} a}\; \overset{eY}{\sim}\; a\; Y \dashv_M \\
(10) & \underline{X \overset{Xa}{\sim} a \overset{aX}{\sim} \overset{Xe}{\sim} \overset{ea}{\sim} a \overset{ae}{\sim} \overset{ee}{\sim} \overset{ea}{\sim} a\; \overset{eY}{\sim}}\; a\; Y \dashv_M \\
(11) & \underline{X \overset{Xa}{\sim} a \overset{aX}{\sim} \overset{Xe}{\sim} \overset{ea}{\sim} a \overset{ae}{\sim} \overset{ee}{\sim} \overset{ea}{\sim} a \overset{ae}{\sim} \overset{eY}{\sim} a\; Y} \dashv_M \\
(12) & \underline{X \overset{Xa}{\sim} a \overset{aX}{\sim} \overset{Xe}{\sim} \overset{ea}{\sim} a \overset{ae}{\sim} \overset{ee}{\sim} \overset{ea}{\sim} a \overset{ae}{\sim} \overset{eY}{\sim} \overset{Ya}{\sim} a\; Y \dashv_M} \\
(13) & X \overset{Xa}{\sim} a \overset{aX}{\sim} \overset{Xe}{\sim} \overset{ea}{\sim} a \overset{ae}{\sim} \overset{ee}{\sim} \overset{ea}{\sim} a \overset{ae}{\sim} \overset{eY}{\sim} \overset{Ya}{\sim} a \overset{aY}{\sim} Y\; .
\end{array}$$

Note that in the first step (5) → (6) *the symbol X initiated recovering of the ether between X and Y. On the other hand, in the last step* (12) → (13) *the symbol Y finished the recovery of the ether between X and Y. Only in the steps* (6) → (7) *to* (11) → (12) *we used the recovery operation. Now we can send another signal from X to Y.*

It is difficult to describe the language L which is generated from the word $X \overset{Xe}{\sim} \overset{ee}{\sim} \overset{eY}{\sim} Y$ by repeated sending a signal (represented by the symbols a and $\tilde{a}$) from X to Y, as in Example 3.8.1 and Example 3.8.2. But observe that if the ether between X and Y has n symbols, then after sending a signal through this ether we get an unusable transfer medium between X and Y of length $2n + 1$. After subsequent recovering of this medium we get a new ether between X and Y of length $2(2n + 1) + 1$. Thus the transfer medium between X and Y grows exponentially as we send signals from X to Y. Also note that $L_{ether} = \{w \in \Sigma^* \mid |w| \geq 2, w \text{ is an ether } \}$ is a regular language. It can be shown that the language $L \cap L_{ether}$ contains only words of length $4^k + 1$, and for each $k \geq 1$ it contains at least one word. This implies that

L is not a context-free language. In order to prove this formally we need to make some observations.

Suppose that we have $w = a_1 a_2 \dots a_n \in \Sigma^*$. Consider the sequence of couples (a_1, a_2), (a_2, a_3), ..., (a_{n-1}, a_n). Now if we replace each couple in this sequence with the symbol $\bigcirc$ or $\square$ depending on whether the corresponding couple satisfies the ether property or not, we get a so-called *circle-square representation* of the word w. For instance, in the word $a \overline{\overset{ab}{\sim} \overset{bc}{\sim}}\ \overline{a \overset{ab}{\sim} b \overset{bc}{\sim}}\ c\ a$ we have couples:

$(a, \overset{ab}{\sim})$, $(\overset{ab}{\sim}, \overset{bc}{\sim})$, $(\overset{bc}{\sim}, a)$, $(a, \overset{ab}{\sim})$, $(\overset{ab}{\sim}, b)$, $(b, \overset{bc}{\sim})$, $(\overset{bc}{\sim}, c)$, and (c, a).

The corresponding circle-square representation of this word is: $\bigcirc\bigcirc\square\bigcirc\bigcirc\bigcirc\bigcirc\square$.

For $w \in \Sigma^{\geq 2}$, let us define $\psi(w)$ to be the circle-square representation of w. First, note that $|\psi(w)| = |w| - 1$. Second, for each $1 \leq i < j \leq n$: $\psi(w[i \dots j]) = \psi(w)[i \dots j-1]$.

The circle-square representation of the jump and recovery operation is:

(1) circle-square jump: $\square\underline{\bigcirc}\bigcirc \dashv_M \square\square\square\bigcirc$,
(2) circle-square recovery: $\bigcirc\underline{\square}\square \dashv_M \bigcirc\bigcirc\bigcirc\square$.

The corresponding instructions for these operations are:

(1) circle-square jump instruction: $(\square, \underline{\bigcirc} \rightarrow \square\square, \bigcirc)$,
(2) circle-square recovery instruction : $(\bigcirc, \underline{\square} \rightarrow \bigcirc\bigcirc, \square)$.

Lemma 3.8.1. *Suppose $w \in \Sigma^*$.*

(1) If $\sigma \in \{a, \tilde{a}\}$ and $w' = \sigma xyz \in \Sigma^4$ is a subword of w, then there exists a jump instruction $(\sigma x, \lambda \rightarrow \sigma', yz)$ applicable to the subword w' of w if and only if a circle-square jump instruction $(\square, \underline{\bigcirc} \rightarrow \square\square, \bigcirc)$ can be applied to $\psi(w')$, i.e. if $\psi(w') = \square\bigcirc\bigcirc$.

(2) If $w' = xyzd \in \Sigma^4$ is a subword of w, then there exists a recovery instruction $(xy, \lambda \rightarrow \overset{uv}{\sim}, zd)$ applicable to the subword w' of w if and only if a circle-square recover instruction $(\bigcirc, \underline{\square} \rightarrow \bigcirc\bigcirc, \square)$ can be applied to $\psi(w')$, i.e. if $\psi(w') = \bigcirc\square\square$.

Proof. The proof is an immediate consequence of the definition of the jump and recovery operation. $\square$

Thus if the concrete symbols of a word are not important but we study which pairs of consecutive symbols satisfy the ether property, we can restrict ourselves to the circle-square representation of words and the aforementioned circle-square jump and recovery instructions.

Example 3.8.3. *If we rewrite Examples 3.8.1 and 3.8.2 in a circle-square representation, then the signal spreading part from Example 3.8.1 is:*

$$\begin{array}{ll}
(1) & \underline{\bigcirc}\bigcirc\bigcirc\bigcirc \to \\
(2) & \square\square\underline{\bigcirc}\bigcirc\bigcirc \to \\
(3) & \square\square\square\square\underline{\bigcirc}\bigcirc \to \\
(4) & \square\square\square\square\square\square\underline{\bigcirc} \to \\
(5) & \square\square\square\square\square\square\square\square
\end{array}$$

and the recovery part from Example 3.8.2 is:

$$\begin{array}{ll}
(5) & \underline{\square}\square\square\square\square\square\square\square \to \\
(6) & \bigcirc\bigcirc\underline{\square}\square\square\square\square\square\square \to \\
(7) & \bigcirc\bigcirc\bigcirc\bigcirc\underline{\square}\square\square\square\square\square \to \\
(8) & \bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\underline{\square}\square\square\square\square \to \\
(9) & \bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\underline{\square}\square\square\square \to \\
(10) & \bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\underline{\square}\square\square \to \\
(11) & \bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\underline{\square}\square \to \\
(12) & \bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\underline{\square} \to \\
(13) & \bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\ .
\end{array}$$

Consider a 1-CRS-system $R = (\Theta, \Theta, J)$, where $\Theta = \{\bigcirc, \square\}$, with the following instructions:

$$\begin{array}{ll}
(0) & (¢, \lambda \to \bigcirc^m, \$), \\
(1) & (\{¢, \square\}, \underline{\bigcirc} \to \square\square, \{\bigcirc, \$\}), \\
(2) & (\{¢, \bigcirc\}, \underline{\square} \to \bigcirc\bigcirc, \{\square, \$\}).
\end{array}$$

According to Theorem 3.6.3, $L^+(R) \cap \{\bigcirc\}^+ = \{\bigcirc^x \mid x = 4^l m,\ l = 0, 1, 2, \ldots\}$, where $m \geq 1$ and $\alpha = \beta = 2$.

Note that the circle-square representation of the word $w_0 = \overline{X \overset{Xe}{\sim} \overset{ee}{\sim} \overset{eY}{\sim} Y}$ is $\bigcirc^4$ and the circle-square representation of L_{ether} is the language $\{\bigcirc\}^+$. Let L denote the language generated from the word w_0 by continual sending a signal (represented by the symbols a and $\tilde{a}$) from X to Y, as it was described above in Example 3.8.1 and Example 3.8.2. It is easy to see that the circle-square representation of L is equal to $L^+(R)$ for $m = 4$. Thus the circle-square representation of $L \cap L_{ether}$ is $L^+(R) \cap \{\bigcirc\}^+$, which is equal to $\{\bigcirc^x \mid x = 4^k,\ k = 1, 2, 3, \ldots\}$. This implies that the language $L \cap L_{ether}$ contains only words of length $4^k + 1$, and for each $k \geq 1$ at least one word.

The approach described so far can be generalized in many ways. For instance, we can have more membranes, more kinds of signals, and we can also send these signals in both directions. If we send signals in both directions, then we have to ensure that in every transfer medium the communication flows only in one direction at any given time.

On the other hand, this approach allows us to construct a 2-cl-RA-automaton accepting a non-context-free language by using only 6-letter alphabet. First, the signal is represented by symbols a and $\tilde{a}$, and these two symbols are the only basic symbols. Thus $\Lambda = \{a, \tilde{a}\}$ and the corresponding alphabet $\Sigma = \Lambda \cup wave(\Lambda)$ contains exactly $2 + 4 = 6$ letters. Second, we do not use special symbols for membranes, i.e. X and Y. Instead of X we can use a and instead of Y we can use a, as well. The instructions of the resulting automaton simulate continual sending of the signal from the first symbol a to the last symbol a, and the starting word is $w_0 = a \overset{aa}{\sim} \overset{aa}{\sim} \overset{aa}{\sim} a$.

3.9 3-cl-RA-automata recognizing non-context-free language

The question, whether there exists a 2-cl-RA-automaton with alphabet consisting of less than six letters recognizing a non-context-free language, remains open. However, there exists a 3-cl-RA-automaton accepting a non-context-free language on alphabet $\Sigma = \{a, b\}$. Also the idea of this automaton is based on sending a signal through the ether. Let us consider the following productions:

$$aaabbbaaabbb \dashv_M a\underline{b}aabbbaaabbb \dashv_M$$
$$aba\underline{b}abbbaaabbb \dashv_M ababab\underline{a}bbaaabbb \dashv_M$$
$$\ldots$$
$$ababababababababab\underline{a}b \dashv_M$$
$$\underline{aa}abababababababababab \dashv_M$$
$$aaa\underline{bb}bababababababababab \dashv_M$$
$$\ldots$$
$$(aaabbb)^8\underline{aa}abab \dashv_M$$
$$(aaabbb)^8aaa\underline{bb}bab \dashv_M$$
$$(aaabbb)^8aaabbb\underline{aa}ab \dashv_M$$
$$(aaabbb)^8aaabbbaaa\underline{bb}b\ .$$

The corresponding 3-cl-RA-automaton $M = (\Sigma, I)$ has the following instructions:

$a_1 = (bab, \underline{a}, b\$)$,
$a_2 = (bab, \underline{a}, baa)$,
$a_3 = (bab, \underline{a}, bb\$)$,
$a_4 = (bab, \underline{a}, bba)$,
$b_1 = (\text{¢}a, \underline{b}, aab)$,
$b_2 = (aba, \underline{b}, aab)$,
$b_3 = (aba, \underline{b}, abb)$,
$c_1 = (\text{¢}, \underline{aa}, aba)$,
$c_2 = (bbb, \underline{aa}, ab\$)$,
$c_3 = (bbb, \underline{aa}, aba)$,
$d_1 = (aaa, \underline{bb}, b\$)$,
$d_2 = (aaa, \underline{bb}, bab)$,
$e_1 = (\text{¢}, \underline{aaabbbaaabbb}, \$)$.

It is difficult to describe the language $L(M)$ directly. Therefore, we introduce the following circle-square representation: Suppose that we have $w = a_1 a_2 \ldots a_n \in \Sigma^*$. Consider the corresponding sequence of couples (a_1, a_2), (a_2, a_3), ..., (a_{n-1}, a_n). Now if we replace each couple in this sequence with the symbol $\bigcirc$ or $\Box$ depending on whether the corresponding couple (a_i, a_{i+1}) is homogenous (i.e. $a_i = a_{i+1}$) or not, we get a *circle-square representation* of the word w.

For $w \in \Sigma^{\geq 2}$, we define $\chi(w)$ to be the circle-square representation of w. Of course, $|\chi(w)| = |w| - 1$, and for each $1 \leq i < j \leq n : \chi(w[i \ldots j]) = \chi(w)[i \ldots j-1]$.

The circle-square representation of productions defining the automaton M is:

$$\underline{\bigcirc}\bigcirc\Box\bigcirc\bigcirc\Box\bigcirc\bigcirc\Box\bigcirc\bigcirc \to \Box\Box\underline{\bigcirc}\Box\bigcirc\bigcirc\Box\bigcirc\bigcirc\Box\bigcirc\bigcirc \to$$
$$\Box\Box\Box\Box\Box\underline{\bigcirc}\bigcirc\Box\bigcirc\bigcirc\Box\bigcirc\bigcirc \to \Box\Box\Box\Box\Box\Box\Box\underline{\bigcirc}\Box\bigcirc\bigcirc\Box\bigcirc\bigcirc \to$$
$$\ldots$$
$$\underline{\Box}\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box \to$$
$$\bigcirc\bigcirc\underline{\Box}\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box \to$$
$$\bigcirc\bigcirc\Box\bigcirc\bigcirc\underline{\Box}\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box\Box \to$$
$$\ldots$$
$$(\bigcirc\bigcirc\Box)^{16}\bigcirc\bigcirc\underline{\Box}\Box\Box \to$$
$$(\bigcirc\bigcirc\Box)^{16}\bigcirc\bigcirc\Box\bigcirc\bigcirc\underline{\Box}\Box \to$$
$$(\bigcirc\bigcirc\Box)^{16}\bigcirc\bigcirc\Box\bigcirc\bigcirc\Box\bigcirc\bigcirc\underline{\Box} \to$$
$$(\bigcirc\bigcirc\Box)^{16}\bigcirc\bigcirc\Box\bigcirc\bigcirc\Box\bigcirc\bigcirc\Box\bigcirc\bigcirc \ .$$

Consider a 2-CRS-system $R = (\Theta, \Theta, J)$, where $\Theta = \{\bigcirc, \Box\}$, with instructions:

$$\begin{aligned}
X_0 &= (\text{¢}, \bigcirc \to \square\square, \bigcirc\square), & Y_0 &= (\text{¢}, \lambda \to \bigcirc\bigcirc, \square\square),\\
X_1 &= (\square\square, \bigcirc \to \square\square, \bigcirc\square), & Y_1 &= (\bigcirc\bigcirc, \square \to \square\bigcirc\bigcirc, \square\square),\\
X_2 &= (\square\square, \bigcirc \to \square\square, \square\bigcirc), & Y_2 &= (\bigcirc\bigcirc, \square \to \square\bigcirc\bigcirc, \square\$),\\
X_3 &= (\square\square, \bigcirc \to \square\square, \bigcirc\$), & Y_3 &= (\bigcirc\bigcirc, \square \to \square\bigcirc\bigcirc, \$),\\
X_4 &= (\square\square, \bigcirc \to \square\square, \$), & Z_0 &= (\text{¢}, \lambda \to \bigcirc\bigcirc\square\bigcirc\bigcirc\square\bigcirc\bigcirc\square\bigcirc\bigcirc, \$).
\end{aligned}$$

Then, trivially for each instruction $i \in I^D$, if i can be applied to the word $w \in \Sigma^{\geq 2}$, then there exists $j \in J$ which can be applied in the same way to the word $\chi(w)$. Thus the circle-square representation of the language $L(M)$ is included in $L^+(R)$.

To analyze the language $L^+(R)$ we first introduce several auxiliary definitions. An *ether* is a word $w \in \{\bigcirc, \square\}^+$ which starts and ends with $\bigcirc$, and if for some $1 < i < |w|$ we have $w[i] = \square$, then $w[i-1] = w[i+1] = \bigcirc$, i.e. all squares are single. A *factor* is a word $\square^k w \neq \lambda$ such that $k \geq 0$ and w is either λ, or an ether. For a word $w \in \{\bigcirc, \square\}^+$, we define a *factorization* $w = w_0 w_1 \ldots w_n$ recursively as: w_n is the longest suffix of w such that w_n is a factor, and $w_0 w_1 \ldots w_{n-1}$ is the factorization of the rest of the word. Since the word w_n is defined unambiguously, it is easy to see by induction that the factorization of w is unique. For instance, the word $w = \square\square\bigcirc\square\bigcirc\bigcirc\square\square\square\square\square\square\bigcirc\square\bigcirc\bigcirc\square\bigcirc\bigcirc$ has the factors: $w_0 = \square\square\bigcirc\square\bigcirc\bigcirc$ and $w_1 = \square\square\square\square\square\square\bigcirc\square\bigcirc\bigcirc\square\bigcirc\bigcirc$. The weight of a factor u is defined as $\omega(u) = |u|_\square + 2|u|_\bigcirc$.

Lemma 3.9.1. *Suppose that $w \in L^+(R)$ and $w = w_0 w_1 \ldots w_n$ is a factorization of w. Let us denote $a_i = \omega(w_i)$, for all $i = 0, 1, \ldots, n$. Then for some $l \geq 1$:*

$$5^0 a_0 + 5^1 a_1 + \ldots + 5^n a_n = 4 \times 5^{n+l} - 1 \ .$$

We call this sum the overall sum of the word w.

Proof. (By induction on the number of used instructions of R)

The first applied instruction is always $Z_0 = (\text{¢}, \lambda \to \bigcirc\bigcirc\square\bigcirc\bigcirc\square\bigcirc\bigcirc\square\bigcirc\bigcirc, \$)$. This instruction can never be used later. The factorization of the resulting word $w = \bigcirc\bigcirc\square\bigcirc\bigcirc\square\bigcirc\bigcirc\square\bigcirc\bigcirc$ is clearly $w = w_0$, thus $n = 0$, $a_0 = \omega(w_0) = \omega(w) = 19$ and $5^0 a_0 = 19 = 4 \times 5^{0+l} - 1$ for $l = 1$.

Suppose that $w = w_0 w_1 \ldots w_n$ is a factorization of w generated so far by circle-square instructions, $a_i = \omega(w_i)$, for all $i = 0, 1, \ldots, n$, and $5^0 a_0 + 5^1 a_1 + \ldots + 5^n a_n = 4 \times 5^{n+l} - 1$ for some $l \geq 1$. We consider several cases depending on the next applied instruction.

1. $X_0 = (\text{¢}, \bigcirc \rightarrow \square\square, \bigcirc\square)$: can be applied only to a prefix $\underline{\bigcirc}\bigcirc\square$ of w_0, i.e. $w_0 = \underline{\bigcirc}\bigcirc\square u_0$. After application we get $w_0' = \square\square\bigcirc\square u_0$ which is obviously a factor with the same weight as w_0.

2. $X_1 = (\square\square, \bigcirc \rightarrow \square\square, \bigcirc\square)$: can be applied only to a subword $u = \square\square\underline{\bigcirc}\bigcirc\square$ of w. Apparently, u is a subword of some factor w_i, i.e. $w_i = \square^k\square\square\underline{\bigcirc}\bigcirc\square u_i$, $k \geq 0$. After application we get $w_i' = \square^k\square\square\square\square\bigcirc\square u_i$ which is a factor with the same weight as w_i.

3. $X_2 = (\square\square, \bigcirc \rightarrow \square\square, \square\bigcirc)$: can be applied only to a subword $u = \square\square\underline{\bigcirc}\square\bigcirc$ of w. Obviously, u is a subword of some factor w_i, i.e. $w_i = \square^k\square\square\underline{\bigcirc}\square\bigcirc u_i$, $k \geq 0$. After application we obtain $w_i' = \square^k\square\square\square\square\square\bigcirc u_i$, which is a factor with the same weight as w_i.

4. $X_3 = (\square\square, \bigcirc \rightarrow \square\square, \bigcirc\$)$: can be applied only to a suffix $\square\square\underline{\bigcirc}\bigcirc$ of w_n, i.e. $w_n = \square^k\square\square\underline{\bigcirc}\bigcirc$, $k \geq 0$. After application we get $w_n' = \square^k\square\square\square\square\bigcirc$, which is a factor with the same weight as w_n.

5. $X_4 = (\square\square, \bigcirc \rightarrow \square\square, \$)$: can be applied only to a suffix $\square\square\underline{\bigcirc}$ of w_n, i.e. $w_n = \square^k\square\square\underline{\bigcirc}$, $k \geq 0$. After application we obtain $w_n' = \square^k\square\square\square\square$, which is a factor with the same weight as w_n.

6. $Y_0 = (\text{¢}, \lambda \rightarrow \bigcirc\bigcirc, \square\square)$: can be applied only to a prefix of w_0, i.e. $w_0 = \square\square\square^k e_0$, $k \geq 0$, e_0 is either λ, or the ether. After application we get a new factor $w_0' = \bigcirc\bigcirc$ with $a_0' = \omega(w_0') = 4$, i.e. the corresponding $w' = w_0'w_1' \ldots w_{n+1}'$, where $w_{i+1}' = w_i$, $a_{i+1}' = a_i$, for all $i = 0, 1, \ldots, n$. Thus the overall sum is $5^0a_0' + 5^1a_1' + \ldots + 5^{n+1}a_{n+1}' = 4 + 5 \times (4 \times 5^{n+l} - 1) = 4 \times 5^{(n+1)+l} - 1$.

7. $Y_1 = (\bigcirc\bigcirc, \square \rightarrow \square\bigcirc\bigcirc, \square\square)$: can be applied to a subword $\bigcirc\bigcirc\underline{\square}\square\square$ of w, i.e. to a prefix of some factor $w_{i+1} = \underline{\square}\square\square u_{i+1}$, where $w_i = u_i\bigcirc\bigcirc$. After application we get factors: $w_i' = u_i\bigcirc\bigcirc\square\bigcirc\bigcirc$ and $w_{i+1}' = \square\square u_{i+1}$. Obviously, $\omega(w_i) + 5 \times \omega(w_{i+1}) = \omega(w_i') + 5 \times \omega(w_{i+1}')$, because $\omega(w_i') - \omega(w_i) = 5 \times (\omega(w_{i+1}) - \omega(w_{i+1}')) = 5$. Since all other factors remain unchanged, the overall sum does not change.

8. $Y_2 = (\bigcirc\bigcirc, \square \rightarrow \square\bigcirc\bigcirc, \square\$)$: can be applied to a suffix $\bigcirc\bigcirc\underline{\square}\square$ of w, i.e. $n \geq 1$, $w_{n-1} = u_{n-1}\bigcirc\bigcirc$, and $w_n = \underline{\square}\square$. After application we obtain factors: $w_{n-1}' = u_{n-1}\bigcirc\bigcirc\square\bigcirc\bigcirc$ and $w_n' = \square$. Obviously, $\omega(w_{n-1}) + 5 \times \omega(w_n) = \omega(w_{n-1}') + 5 \times \omega(w_n')$, because $\omega(w_{n-1}') - \omega(w_{n-1}) =$

$5 \times (\omega(w_n) - \omega(w'_n)) = 5$. Since all other factors remain unchanged, the overall sum does not change.

9. $Y_3 = (\bigcirc\bigcirc, \square \to \square\bigcirc\bigcirc, \$)$: can be applied to a suffix $\bigcirc\bigcirc\underline{\square}$ of w, i.e. $n \geq 1$, $w_{n-1} = u_{n-1}\bigcirc\bigcirc$, and $w_n = \underline{\square}$. After application we get factor: $w'_{n-1} = u_{n-1}\bigcirc\bigcirc\square\bigcirc\bigcirc$ and the last factor vanishes. Apparently, $\omega(w_{n-1}) + 5 \times \omega(w_n) = \omega(w'_{n-1})$, because $\omega(w'_{n-1}) - \omega(w_{n-1}) = 5 \times \omega(w_n) = 5$. All other factors remain unchanged, thus if we denote $a'_i = \omega(w'_i)$, for all $i = 0, 1, \ldots, n-1$, then $5^0 a'_0 + 5^1 a'_1 + \ldots + 5^{n-1} a'_{n-1} = 5^0 a_0 + 5^1 a_1 + \ldots + 5^{n-1}(a_{n-1} + 5a_n) = 4 \times 5^{n+l} - 1 = 4 \times 5^{(n-1)+(l+1)} - 1$.

$\square$

Note that only the instruction $Y_0 = (¢, \lambda \to \bigcirc\bigcirc, \square\square)$ increases the number of factors and only the instruction $Y_3 = (\bigcirc\bigcirc, \square \to \square\bigcirc\bigcirc, \$)$ decreases the number of factors. Also note that only the instruction Y_0 increases the overall sum, and only the instruction Y_3 increases l. All remaining instructions neither change the number of factors, nor the overall sum.

Lemma 3.9.2. *$L(M) \cap \{(ab)^n \mid n > 0\} = \{(ab)^n \mid n = 2 \times 5^l, l = 1, 2, \ldots\}$.*

Proof. Suppose that $(ab)^m \in L(M)$ for some $m > 0$. The circle-square representation of $(ab)^m$ is $\square^{2m-1}$. Since $\square^{2m-1}$ is a factor, according to Lemma 3.9.1 we get $2m - 1 = 4 \times 5^{n+l} - 1$, where $n = 0$ and $l \geq 1$. Thus $m = 2 \times 5^l$.

On the other hand, we have to show that for each $l \geq 1$, $(ab)^{2\times 5^l} \in L(M)$. First observe:

$aaabbb(aaabbb)^m aaabbb \dashv_M$
$a\underline{b}aabbb(aaabbb)^m aaabbb \dashv_M aba\underline{b}abbb(aaabbb)^m aaabbb \dashv_M$
$ababab\underline{a}bb(aaabbb)^m aaabbb \dashv_M abababab\underline{a}b(aaabbb)^m aaabbb \dashv_M$
$\ldots \dashv_M ababababab(ababababab)^m aaabbb \dashv_M$
$ababababab(ababababab)^m a\underline{b}aabbb \dashv_M ababababab(ababababab)^m aba\underline{b}abbb \dashv_M$
$ababababab(ababababab)^m ababab\underline{a}bb \dashv_M ababababab(ababababab)^m abababab\underline{a}b =$
$(ab)^5(ababababab)^m(ab)^5$.

Similarly:

$abab(ab)^n abab \dashv_M$
$\underline{aa}abab(ab)^n abab \dashv_M aaa\underline{bb}bab(ab)^n abab \dashv_M$
$aaabbb\underline{aa}ab(ab)^n abab \dashv_M aaabbbaaa\underline{bb}b(ab)^n abab \dashv_M$
$\ldots \dashv_M aaabbbaaabbb(aaabbb)^n abab \dashv_M$
$aaabbbaaabbb(aaabbb)^n \underline{aa}abab \dashv_M aaabbbaaabbb(aaabbb)^n aaa\underline{bb}bab \dashv_M$
$aaabbbaaabbb(aaabbb)^n aaabbb\underline{aa}ab \dashv_M aaabbbaaabbb(aaabbb)^n aaabbbaaa\underline{bb}b =$
$(aaabbb)^2(aaabbb)^n(aaabbb)^2$.

Consequently, for $m = 0$ we get: $\lambda \dashv_M aaabbbaaabbb \dashv_M^* (ab)^{2\times 5}$, and for each $l \geq 1$: $(ab)^{2\times 5^l} \dashv_M^* (aaabbb)^{2\times 5^l} \dashv_M^* (ababababab)^{2\times 5^l} = (ab)^{2\times 5^{l+1}}$. Thus for each $l \geq 1 : (ab)^{2\times 5^l} \in L(M)$. $\square$

The following theorem summarizes the results from the previous Sections 3.3, 3.6, 3.7, 3.8, and 3.9, and compares the class of languages recognized by clearing restarting automata with the class of context-free languages.

Theorem 3.9.1. *Let us consider k-cl-RA-automata over an alphabet Σ.*

a) $\mathcal{L}(\mathsf{cl\text{-}RA}) = \mathsf{CFL}$ *for* $|\Sigma| = 1$.

b) $\mathcal{L}(\mathsf{1\text{-}cl\text{-}RA}) \subset \mathsf{CFL}$ *for an arbitrary alphabet* Σ.

c) $\mathcal{L}(\mathsf{2\text{-}cl\text{-}RA}) - \mathsf{CFL} \neq \emptyset$ *for* $|\Sigma| \geq 6$.

d) $\mathcal{L}(\mathsf{3\text{-}cl\text{-}RA}) - \mathsf{CFL} \neq \emptyset$ *for* $|\Sigma| \geq 2$.

Chapter 4

Extended Clearing Restarting Automata

In this chapter we introduce two extended versions – the so-called Δ-clearing restarting automata in Section 4.2 and Δ^*-clearing restarting automata in Section 4.3. Both of them can use a single auxiliary symbol Δ only. Δ-clearing restarting automata can leave a mark – a symbol Δ – at the place of deleting besides rewriting into the empty word λ. This model can recognize Greibach's hardest context-free language H ([10]). In Section 4.1 we define this language H and prove that there is no cl-RA-automaton recognizing H. Δ^*-clearing restarting automata can rewrite a subword w into Δ^k where k is bounded from above by the length of w. This is enough to enable this model to recognize all context-free languages.

Most of the material used in this chapter is taken from the extended version of [6] accepted for publication in Fundamenta Informaticae.

4.1 Greibach's hardest context-free language

As we have seen in Corollary 3.4.1b) not all context-free languages are recognized by cl-RA-automata. We can still try to characterize CFL by using clearing restarting automata, inverse morphism, and Greibach's hardest context-free language [10].

Greibach constructed a context-free language H such that:

- Any context-free language can be parsed in whatever time or space it takes to recognize H.

- Any context-free language L can be obtained from H by an inverse morphism. That is, for each context-free language $L \subseteq \Sigma^*$, there exists a morphism φ such that $L = \varphi^{-1}(H)$.

The definition of Greibach's language follows. Let $\Sigma = \{a_1, a_2, \overline{a}_1, \overline{a}_2, \#, c\}$ and $\Sigma' = \Sigma \cup \{d\}$, where $d \notin \Sigma$. Let $H \subset (\Sigma')^*$ be the language

$$\begin{aligned} H = \{\lambda\} \cup \{x_1 c y_1 c z_1 d \ldots x_n c y_n c z_n d \mid {} & n \geq 1, y_1 \ldots y_n \in \# D_2, x_i, z_i \in \Sigma^*, \\ & \text{for all } i, 1 \leq i \leq n, \\ & y_1 \in \# \cdot \{a_1, a_2, \overline{a}_1, \overline{a}_2\}^*, \\ & y_i \in \{a_1, a_2, \overline{a}_1, \overline{a}_2\}^*, \text{ for all } i \geq 2\} \end{aligned}$$

(note that x_i's and z_i's can contain c's and $\#$'s), where D_2 is a semi-Dyck language over the alphabet $\{a_1, a_2, \overline{a}_1, \overline{a}_2\}$ generated by the grammar with one nonterminal S and the set of rules:

$$S \to \lambda | SS | a_1 S \overline{a}_1 | a_2 S \overline{a}_2 \ .$$

The symbols d divide naturally w into blocks. Each block is naturally divided by symbols c into parts called segments. It is easy to see that H is context-free. Imagine a pushdown automaton which "guesses" a segment y_1 in the first block and processes it in the natural way a pushdown automaton would work on the Dyck set (i.e. stack everything and pop when the top of the stack is a_i and the input is $\overline{a}_i$). The computation is repeated for each block as delimited by d.

The semi-Dyck language D_2 is recognized by a simple cl-RA-automaton M_D with the following two instructions:

$$(\lambda, a_1 \overline{a}_1, \lambda),$$
$$(\lambda, a_2 \overline{a}_2, \lambda).$$

Hence the main problem of recognizing H by a clearing restarting automaton is the selection of segments $y_1, y_2, \ldots, y_n$ (see the definition of H above). Unfortunately, there is no cl-RA-automaton recognizing H.

Theorem 4.1.1. *H is not accepted by any* cl-RA-*automaton.*

Proof. For a contradiction, let us suppose that there exists a k-cl-RA-automaton $M = (\Sigma', I)$ such that $L(M) = H$. Let m be the maximal width of instructions of M. Obviously, $w = c\#a_1^m cdc\overline{a}_1^m cd \in H$. Let $w \vdash_M^{(i)} w' \vdash_M \cdots \vdash_M \lambda$

be an accepting computation of H on input w. In the first reduction the deleted substring must be of the form $a_1^r cdc\overline{a}_1^s$ for some r, s, $0 \leq r, s < m$, otherwise w' would not be in H. Further, it is easy to see that $r = s$. As $|i| \leq m$, the first applied instruction i is of the form $(a_1^{\alpha}, a_1^r cdc\overline{a}_1^r, \overline{a}_1^{\beta})$, where $\alpha, \beta < m$. Then, however, consider the word $u = c\#a_1^{m+1} a_1^r cdc\overline{a}_1^r \overline{a}_1^m c\#a_1\overline{a}_1 cd$. Obviously, u is not in H, but $u \vdash_M^{(i)} c\#a_1^{m+1}\overline{a}_1^m c\#a_1\overline{a}_1 cd \in H$, which is a contradiction to the error preserving property (Theorem 3.2.2). $\square$

4.2 Δ-Clearing Restarting Automata

In [7] Cherubini et al. defined H using associative language description, which uses one auxiliary symbol Δ. Therefore, we slightly extend the definition of cl-RA-automata in order to be able to recognize more languages including H.

Definition 4.2.1. *Let k be a positive integer. A k-*Δ-clearing restarting automaton *(k-Δcl-RA-*automaton *for short) is a system $M = (\Sigma, I)$, where $M = (\Sigma, \Gamma, I)$ is a k-CRS-system such that $\Delta \notin \Sigma$, $\Gamma = \Sigma \cup \{\Delta\}$, and for each instruction $i = (x, z \to t, y) \in I$: $z \in \Gamma^+$ and either $t = \lambda$, or $t = \Delta$.*

The k-Δcl-RA-automaton M recognizes *the language $L(M) = \{w \in \Sigma^* \mid w \vdash_M^* \lambda\} = L^-(M)$, where $\vdash_M$ is the* rewriting relation *$\to_M$ of M.*

The characteristic language *of M is the language $L_C(M) = L_C^-(M)$.*

Actually, Δcl-RA-automata differ from cl-RA-automata in the ability to leave a mark – a symbol Δ – at the place of deleting.

By Δcl-RA we denote the class of all Δ-clearing restarting automata. $\mathcal{L}(k\text{-}\Delta\text{cl-RA})$ ($\mathcal{L}(\Delta\text{cl-RA})$, respectively) denotes the class of all languages accepted by k-Δcl-RA-automata (Δcl-RA-automata, respectively).

Example 4.2.1. *Let $M = (\Sigma, I)$ be the* 1-Δcl-RA-*automaton with $\Sigma = \{a, b, c\}$ and the set of instructions I consisting of the following instructions:*

$$\begin{array}{ll} (1) & (a, c \to \Delta, b), \\ (2) & (a, a\Delta b \to \Delta, b), \\ (3) & (¢, a\Delta b \to \Delta, \$), \\ (4) & (¢, c \to \Delta, \$), \\ (5) & (¢, \Delta \to \lambda, \$). \end{array}$$

An input word $a^n cb^n$, for arbitrary $n > 1$, is accepted by M in the following way:

$$a^n\underline{c}b^n \vdash_M^{(1)} a^{n-1}\underline{a\Delta b}b^{n-1}n \vdash_M^{(2)} a^{n-1}\Delta b^{n-1} \vdash_M^{(2)} \ldots \vdash_M^{(2)} \underline{a\Delta b} \vdash_M^{(3)} \underline{\Delta} \vdash_M^{(5)} \lambda\ .$$

First, M deletes c while marking its position by Δ. In each of the following steps, M deletes one a and one b around Δ until it obtains single-letter word Δ, which is then reduced into λ.

It is easy to see that M recognizes the language $L = \{a^n cb^n \mid n \geq 0\} \cup \{\lambda\}$.

The characteristic language of M is

$$L_C(M) = \{a^n cb^n, a^n \Delta b^n \mid n \geq 0\} \cup \{\lambda\} \ .$$

In what follows we show that the language H can be recognized by a 1-Δcl-RA-automaton:

Theorem 4.2.1. *H is recognized by a 1-Δcl-RA-automaton.*

Proof. Let us describe a Δcl-RA-automaton M recognizing H informally, first. Let w be a word from H of the following form:

$$(\star) \quad \begin{array}{l} w = x_1cy_1cz_1dx_2cy_2cz_2d \dots x_ncy_ncz_nd, \text{ where} \\ n \geq 1, y_1 \dots y_n \in \#D_2, \\ x_i, z_i \in \Sigma^* = \{a_1, a_2, \overline{a}_1, \overline{a}_2, \#, c\}^*, \\ \text{for all } i, 1 \leq i \leq n, y_1 \in \# \cdot \{a_1, a_2, \overline{a}_1, \overline{a}_2\}^* \\ \text{and } y_i \in \{a_1, a_2, \overline{a}_1, \overline{a}_2\}^*, \text{ for all } i \geq 2. \end{array}$$

In the first phase, we start with deleting letters from the alphabet Σ from the right side of ¢ and from the left and right sides of the letters d. As soon as we think that we have the following word: $cy_1cdcy_2cd \dots cy_ncd$ we introduce symbols Δ: $\Delta y_1 \Delta y_2 \Delta \dots \Delta y_n \Delta$. In the second phase we check whether $y_1y_2 \dots y_n \in \#D_2$.

Let us denote $\Sigma = \{a_1, a_2, \overline{a}_1, \overline{a}_2, \#, c\}$, $\Gamma = \Sigma \cup \{d, \Delta\}$, $d, \Delta \notin \Sigma$. The corresponding set I of instructions of the 1-Δcl-RA-automaton M consists of the following instructions:

Instructions for the first phase	Instructions for the second phase
(1) $(¢, \Sigma \to \lambda, \Sigma)$	(7) $(\Gamma, a_1\overline{a}_1 \to \lambda, \Gamma - \{\#\})$
(2) $(\Sigma, \Sigma \to \lambda, d)$	(8) $(\Gamma, a_2\overline{a}_2 \to \lambda, \Gamma - \{\#\})$
(3) $(d, \Sigma \to \lambda, \Sigma)$	(9) $(\Gamma, a_1\Delta\overline{a}_1 \to \Delta, \Gamma - \{\#\})$
(4) $(¢, c \to \Delta, \Sigma \cup \{\Delta\})$	(10) $(\Gamma, a_2\Delta\overline{a}_2 \to \Delta, \Gamma - \{\#\})$
(5) $(\Sigma \cup \{\Delta\}, cdc \to \Delta, \Sigma \cup \{\Delta\})$	(11) $(\Sigma - \{c\}, \Delta \to \lambda, \Delta)$
(6) $(\Sigma \cup \{\Delta\}, cd \to \Delta, \$)$	(12) $(¢, \Delta\#\Delta \to \lambda, \$)$

Note that we cannot guarantee that M completes the first phase first and then will continue with the second phase. Nevertheless, we will prove that $L(M) = H$ by showing that $H \subseteq L(M)$ and $L(M) \subseteq H$.

Lemma 4.2.1. *$H \subseteq L(M)$.*

Proof. Let $w \in H$ be of the form ($\star$) above. The instructions (1), (2), (3) allow the following computation:

$$\underline{x_1}cy_1c\underline{z_1}d\underline{x_2}cy_2c\underline{z_2}d\dots\underline{x_n}cy_nc\underline{z_n}d \vdash_M^* cy_1cdcy_2cdc\dots cy_ncd.$$

The instructions (4), (5), (6) enable to continue in the following way:

$$\underline{c}y_1cdcy_2cdc\dots cy_ncd \vdash_M \Delta y_1\underline{cdc}y_2cdc\dots cy_ncd \vdash_M \Delta y_1\Delta y_2\underline{cdc}\dots cy_ncd \vdash_M^*$$
$$\dots$$
$$\Delta y_1\Delta y_2\Delta\dots\Delta y_n\underline{cd} \vdash_M \Delta y_1\Delta y_2\Delta\dots\Delta y_n\Delta\ .$$

Note that some of the words y_i can be empty. This explains why we have added Δ symbols to the context of these instructions.

We know that $y_1y_2\dots y_n \in \#D_2$. If we wanted to recognize just the words from $\#D_2$, then the instructions (7), (8) and $(¢, \# \to \lambda, \$)$ would be sufficient. The problem is that we have the word $\Delta y_1\Delta y_2\Delta\dots\Delta y_n\Delta$. We need to add the instructions (9), (10), (11). It is easy to see that these instructions allow the reduction: $\Delta y_1\Delta y_2\Delta\dots\Delta y_n\Delta \vdash_M^* \Delta\#\Delta$. The last word $\Delta\#\Delta$ can be reduced by instruction (12). □

There is an interesting question why we have not used the following instructions:

(9′) $(\Gamma, a_1\Delta\overline{a}_1 \to \lambda, \Gamma - \{\#\})$ and
(10′) $(\Gamma, a_2\Delta\overline{a}_2 \to \lambda, \Gamma - \{\#\})$

instead of instructions (9) and (10)?

The reason why cl-RA-automata are not able to recognize H is caused by the instruction $i = (a_1^u, a_1^\alpha cdc\overline{a}_1^\alpha, \overline{a}_1^v)$, which erases the group of letters cdc. The introduction of these new Δ symbols helped us to save the symbols d, which are now hidden behind these Δ symbols and are important for separating the words y_i. If we use instructions (9′) and (10′) instead of (9) and (10), then take the word $c\#a_1a_1cdc\overline{a}_1c\#a_1\overline{a}_1cd$, which does not belong to H. First, by using the instruction (5) on the group of letters cdc we get: $c\#a_1a_1\Delta\overline{a}_1c\#a_1\overline{a}_1cd$, and next by using the instruction (9′) we get the word: $c\#a_1c\#a_1\overline{a}_1cd$, which evidently belongs to H – a contradiction.

Lemma 4.2.2. *$L(M) \subseteq H$.*

Proof. We prove that each word $w \in L_C(M)$ is of the form: $U_1U_2\dots U_n$ where:

1. $n \geq 0$, $n \neq 1$,
2. U_1 is either Δy_1, or $x_1 c y_1$, for some $y_1 \in \# \cdot \{a_1, a_2, \overline{a}_1, \overline{a}_2\}^*, x_1 \in \Sigma^*$,
3. U_i is either Δy_i, or $c x_i d z_i c y_i$, for $1 < i < n$ and some $x_i, z_i \in \Sigma^*, y_i \in \{a_1, a_2, \overline{a}_1, \overline{a}_2\}^*$,
4. U_n is either Δ, or $c x_n d$, for some $x_n \in \Sigma^*$,
5. $y_1 \ldots y_{n-1} \in \# D_2$.

Let Ω denote the set of all such words. It is easy to see that all words from $\Omega \cap \Sigma^*$ have the following form:

$$x_1 c y_1 c x_2 d\ z_2 c y_2 c x_3 d\ z_3 c y_3 \ldots c x_{n-1} d\ z_{n-1} c y_{n-1} c x_n d\ ,$$

where $n \geq 0$, $n \neq 1$, $x_i, z_i \in \Sigma^*$, $y_1 \ldots y_{n-1} \in \# D_2$. Apparently, this implies that $\Omega \cap \Sigma^* \subseteq H$. To prove that $L(M) \subseteq \Omega \cap \Sigma^* \subseteq H$ it is enough to show that all words $w \in L_C(M)$ are also in the set Ω.

Let us prove this proposition by induction on the number m of applied instructions. For $m = 0$, we have $n = 0$ and $w = \lambda \in \Omega$. For $m = 1$, only the instruction (12) $(¢, \Delta \# \Delta \to \lambda, \$)$ can be used, and the word $w = \Delta \# \Delta \in \Omega$ (take $n = 2$, $U_1 = \Delta y_1, U_2 = \Delta$, where $y_1 = \#$).

Suppose that $U_1 \ldots U_n \dashv_M w$ by using the instruction $i \in I^D$, $n \geq 2$ and $U_1 \ldots U_n$ was obtained from λ by using at most $m - 1$ instructions from I^D, where $m > 1$. In the word $U_1 \ldots U_n$ we place an arrow $^\downarrow$ above the place where the application of the instruction $i \in I^D$ occurred. There are several cases based on the choice of the instruction $i \in I^D$:

1. $(¢, \lambda \to \Sigma, \Sigma)$: this instruction can be applied only within $U_1 = {}^\downarrow x_1 c y_1$. We can easily see that we get a correct form, so the generated word w belongs to Ω.
2. $(\Sigma, \lambda \to \Sigma, d)$: this instruction can be applied either within $U_k = c x_k {}^\downarrow d z_k c y_k$, where $1 < k < n$, or within $U_n = c x_n {}^\downarrow d$. In both cases, we get a word w from Ω.
3. $(d, \lambda \to \Sigma, \Sigma)$: this instruction can be applied only within $U_k = c x_k d {}^\downarrow z_k c y_k$, where $1 < k < n$, and again we get a correct word from Ω.

4. $(\text{¢}, \Delta \to c, \Sigma \cup \{\Delta\})$: this instruction can be applied only within $U_1 = \overset{\downarrow}{\Delta} y_1$. We obtain a new $U_1' = cy_1$, which leads to a correct word from Ω. Note that since y_1 always starts with # (because $y_1 \dots y_n \in \#D_2$), we could replace this instruction with $(\text{¢}, \Delta \to c, \#)$.

5. $(\Sigma \cup \{\Delta\}, \Delta \to cdc, \Sigma \cup \{\Delta\})$: this instruction can be applied only within $U_k = \overset{\downarrow}{\Delta} y_k$, where $1 < k < n$. We get a new $U_k' = cdcy_k$, which leads to a correct word from Ω.

6. $(\Sigma \cup \{\Delta\}, \Delta \to cd, \$)$: this instruction can be applied only within $U_n = \overset{\downarrow}{\Delta}$. We obtain a new $U_n' = cd$, which leads to a correct word from Ω.

7. $(\Gamma, \lambda \to a_1\overline{a}_1, \Gamma - \{\#\})$: see case 8.

8. $(\Gamma, \lambda \to a_2\overline{a}_2, \Gamma - \{\#\})$: we investigate cases 7 and 8 together. These instructions can be applied between any two consecutive characters in the word $U_1 \dots U_n$ except the left end (where the left context equals ¢), the right end (where the right context equals \$) and the case in which the right context equals #. We need to consider these cases:

    (a) $U_1 = {}^{\downarrow}\Delta y_1$ is impossible, since the left context equals ¢.

    (b) $U_1 = \Delta{}^{\downarrow}y_1$ is impossible, as the right context equals #.

    (c) For $U_1 = \Delta \overset{\downarrow}{y_1}$, $U_1 = \Delta y_1{}^{\downarrow}$ we get a correct word.

    (d) $U_1 = {}^{\downarrow}x_1cy_1$is impossible, since the left context equals ¢.

    (e) $U_1 = \overset{\downarrow}{x_1} cy_1$, $x_1{}^{\downarrow}cy_1$, $x_1c{}^{\downarrow}y_1$, $x_1c\overset{\downarrow}{y_1}$, or $x_1cy_1{}^{\downarrow}$: all these cases are possible and all leave a correct word.

    (f) $U_k = {}^{\downarrow}\Delta y_k$, where $1 < k < n$ is possible and we get a new $U_{k-1}' = \dots y_{k-1}a_i\overline{a}_i$, which leads to a correct word from Ω.

    There are many other cases to consider (we have mentioned only few of them for brevity), but we can easily see that all of them preserve the correct form of the word. Also note that the constraint $y_1 \dots y_n \in \#D_2$ is not violated in any of these cases.

9. $(\Gamma, \Delta \to a_1\Delta\overline{a}_1, \Gamma - \{\#\})$: see case 10.

10. $(\Gamma, \Delta \to a_2\Delta\overline{a}_2, \Gamma{-}\{\#\})$: we investigate cases 9 and 10 together. These instructions can be applied only within $U_k = \overset{\downarrow}{\Delta}\, y_k$, where $1 < k < n$. We obtain a new $U'_{k-1} = \ldots y_{k-1}a_i$ and $U'_k = \Delta\overline{a}_i y_k$, which leads to a correct word. Also in this case, the constraint $y_1 \ldots y_n \in \#D_2$ is not violated.

11. $(\Sigma - \{c\}, \lambda \to \Delta, \Delta)$: this instruction can be applied only in the following two cases:

    (a) $U_k = {}^{\downarrow}\Delta y_k$, where $1 < k < n$, $U_{k-1} = \ldots y_{k-1}$ and $y_{k-1} \neq \lambda$. From the word $U_1 \ldots U_{k-1}\Delta y_k \ldots U_n$ we get a new word

    $$U_1 \ldots U_{k-1}\Delta\Delta y_k \ldots U_n,$$

    which is a correct word from Ω.

    (b) $U_n = {}^{\downarrow}\Delta$, where $U_{n-1} = \ldots y_{n-1}$ and $y_{n-1} \neq \lambda$. From the word $U_1 \ldots U_{n-1}\Delta$ we get a new word

    $$U_1 \ldots U_{n-1}\Delta\Delta,$$

    which is a correct word from Ω.

12. $(¢, \lambda \to \Delta\#\Delta, \$)$: this instruction can be applied only in the beginning, when we have the empty word. Evidently, $\Delta\#\Delta$ is in Ω.

We have proven that each word generated by using instructions I^D belongs to Ω. As we have shown before, this implies that $L(M) \subseteq \Omega \cap \Sigma^* \subseteq H$. □

The statement of Theorem 4.2.1 follows from the above two lemmas. □

4.3 Δ^*-Clearing Restarting Automata

Although Δcl-RA-automata can recognize Greibach's hardest context-free language H, we still do not know whether they are able to recognize all context-free languages. Now we introduce a generalization of Δcl-RA-automata, a so-called Δ^*-*clearing restarting automata*, which are able to recognize all context-free languages.

Definition 4.3.1. *Let k be a positive integer. A* k-Δ^*-clearing restarting automaton *(*k-Δ^*cl-RA-automaton *for short) is a system $M = (\Sigma, I)$, where $M = (\Sigma, \Gamma, I)$ is a k-*CRS*-system such that $\Delta \notin \Sigma$, $\Gamma = \Sigma \cup \{\Delta\}$, and for each instruction $i = (x, z \to t, y) \in I$: $z \in \Gamma^+$ and $t = \Delta^i$, where $0 \leq i \leq |z|$.*

The k-Δ^*cl-RA-automaton M recognizes *the language $L(M) = \{w \in \Sigma^* \mid w \vdash_M^* \lambda\} = L^-(M)$, where $\vdash_M$ is the* rewriting relation $\to_M$ *of M.*

The characteristic language *of M is the language $L_C(M) = L_C^-(M)$.*

Δ^*cl-RA-automata differ from Δcl-RA-automata in the ability to leave more than one symbol Δ at the place of deleting. The only constraint is that they can replace a subword z by at most $|z|$ symbols Δ.

By Δ^*cl-RA we denote the class of all Δ^*-clearing restarting automata. $\mathcal{L}(k\text{-}\Delta^*\text{cl-RA})$ ($\mathcal{L}(\Delta^*\text{cl-RA})$, respectively) denotes the class of all languages accepted by k-Δ^*cl-RA-automata (Δ^*cl-RA-automata, respectively).

Next we show that 1-Δ^*cl-RA-automata can recognize any context-free language.

Theorem 4.3.1. *For each context-free language L there exists a* 1-Δ^*cl-RA-*automaton M recognizing L.*

Proof. Let L be a context-free language. Then there exists a context-free grammar $G = (V_N, V_T, S, P)$ in Chomsky normal form generating the language $L(G) = L \setminus \{\lambda\}$. Let $V_N = \{N_1, \ldots, N_m\}$, $S = N_1$ and $\Delta \notin V_N \cup V_T$ and let $G' = (V_N, V_T', S, P')$ be a grammar which we obtain from G by adding a new terminal symbol Δ to V_T and adding new productions $N_i \to \Delta^i a$ to P, for all $1 \leq i \leq m$ and all $a \in V_T$. We will show that we can effectively construct a 1-Δ^*cl-RA-automaton M such that $L(M) = L(G) \cup \{\lambda\}$.

Let us set $\Sigma = V_T$ and $\Gamma = \Sigma \cup \{\Delta\} = V_T'$. We will construct a 1-$\Delta^*$cl-RA-automaton M such that $L_C(M) = L(G') \cup \{\lambda\}$. This implies that $L(M) = L_C(M) \cap \Sigma^* = (L(G') \cup \{\lambda\}) \cap \Sigma^* = L(G) \cup \{\lambda\}$.

For the automaton M all the words $\Delta^i a$ for all $a \in \Sigma$ represent "codes" for the nonterminal N_i. The letter $a \in \Sigma$ serves as a separator for distinguishing several consecutive coded nonterminals.

Lemma 4.3.1. *For the grammar G' there exist natural numbers p, q, $m+1 \leq p \leq q$, such that the following condition holds: if $w \in L(G')$ and $|w| > p$, then there exists a derivation $N_1 \Rightarrow_{G'}^* xN_iy \Rightarrow_{G'}^* xzy$ such that $w = xzy$ and $p < |z| \leq q$.*

Proof. Let us set $p := m + 1$ and $q := 2p$. Suppose that we have $w \in L(G'), |w| > p$. Let us consider the derivation tree T corresponding to some derivation $N_1 \Rightarrow^*_{G'} w$. Now we will inductively define a path $P = A_1 A_2 \dots A_r t$ in the tree T, where $A_1, A_2, \dots, A_r$ are nonterminals and t is a terminal.

First, we set $A_1 := N_1$. Now suppose that A_i is defined and in the derivation tree T a rule r is used to rewrite A_i. There are two possible forms of r:

- Either r is of the form $A_i \to LR$, where $L, R \in V_N$. If the number of terminal leaves under L is not less than the number of terminal leaves under R in T, then we set $A_{i+1} := L$ and $B_{i+1} := R$. Otherwise, we set $A_{i+1} := R$ and $B_{i+1} := L$.

- Or r is of the form $A_i \to u$, where $u \in \Gamma^+$, then we set $r := i$ and t to be any letter from u. Note that $|u| \leq m + 1$.

Let us denote a_i (b_i, respectively) a terminal word generated by the derivation tree T from the occurrence of A_i (B_i, respectively). We can easily see that $a_1 = w$, and for all i, $1 \leq i < r$, we have either $a_i = a_{i+1}b_{i+1}$, or $a_i = b_{i+1}a_{i+1}$, so $|a_i| = |a_{i+1}| + |b_{i+1}|$. Since $|a_1| = |a_2| + |b_2| = |a_3| + |b_3| + |b_2| = \ldots = |a_r| + |b_r| + |b_{r-1}| + \ldots + |b_2|$, we have a descending sequence $|a_1| > |a_2| > \ldots > |a_r| \geq 1$. By the definition of P we have $|a_{i+1}| \geq |b_{i+1}|$, for all i, $1 \leq i < r$, so $|a_{i+1}| = \frac{1}{2}(|a_{i+1}| + |a_{i+1}|) \geq \frac{1}{2}(|a_{i+1}| + |b_{i+1}|) = \frac{1}{2}|a_i|$.

By our assumptions, $|a_1| = |w| > p = m + 1$ and $1 \leq |a_r| \leq m + 1 < q = 2p$. Let i be the smallest integer from $\{1, 2, \dots, r\}$ such that $|a_i| \leq q$. If $i = 1$, then $|a_i| > p$. If $i > 1$, then $|a_i| \geq \frac{1}{2}|a_{i-1}| > \frac{1}{2}q = p$.

Now it is sufficient to set $z := a_i$, because $p < |a_i| \leq q$ and $N_1 \Rightarrow^*_{G'} xA_iy \Rightarrow^*_{G'} xa_iy = w$. $\square$

Now we show how to construct a 1-Δ^*cl-RA M such that $L_C(M) = L(G') \cup \{\lambda\}$:

First, we set $I_1 = \{(¢, w \to \lambda, \$) \mid w \in L(G'), |w| \leq p\}$.

For every $i \in \{1, 2, \dots, m\}$ let $L_i = \{z \in \Gamma^* \mid N_i \Rightarrow^*_{G'} z, p < |z| \leq q\}$. For every such $z \in L_i$, $z = z_1 \dots z_{s-1} z_s$, let us consider the instruction (or to be more precise – the set of instructions):

$$(\Sigma \cup \{¢\}, z_1 \dots z_{s-1} \to \Delta^i, z_s)\,.$$

Let I_2 be the set of all such instructions. Then $M = (\Sigma, I_1 \cup I_2)$ is the required automaton.

Lemma 4.3.2. *$L(G') \subseteq L_C(M)$.*

Proof. (By induction on the length of words from $L(G')$)

Let $w \in L(G')$. If $|w| \leq p$, then $(¢, w \to \lambda, \$) \in I_1$, thus $w \vdash_M \lambda$ which implies $w \in L_C(M)$.

Suppose $|w| > p$. According to Lemma 4.3.1, there are $x, y, z \in \Gamma^*$, $p < |z| \leq q$, and $i \in \{1, 2, \dots, m\}$ such that there exists a derivation $N_1 \Rightarrow^*_{G'} xN_iy \Rightarrow^*_{G'} xzy = w$, where $z = z_1 \dots z_{s-1}z_s$. The definition of I_2 implies that there is an instruction $(\Sigma \cup \{¢\}, z_1 \dots z_{s-1} \to \Delta^i, z_s) \in I_2$. If we use this instruction in the word w, we get the reduction: $w = x\underline{z_1 \dots z_{s-1}}z_sy \vdash_M x\Delta^i z_s y = w'$. Note that x is either λ, or ends with a letter from Σ. If $x \neq \lambda$, then necessarily for some suffix u of x there is $N \in V_N : N \Rightarrow^*_{G'} u$. From the definition of G' we see, that if we take any nonterminal $N \in V_N$ and any derivation $N \Rightarrow^*_{G'} u$, the word u ends either with a nonterminal, or a letter from Σ. Now $s = |z| > p$ implies that $s - 1 \geq p \geq m + 1 > m \geq i$. So $|z_1 \dots z_{s-1}| > |\Delta^i|$ implies $|w| > |w'|$. On the other hand, in G' there is a production rule $N_i \to \Delta^i z_s$, so $N_1 \Rightarrow^*_{G'} xN_iy \Rightarrow_{G'} x\Delta^i z_s y = w'$. Thus $w' \in L(G')$ and by induction hypothesis (since $|w'| < |w|$) we have $w' \in L_C(M)$ and $w' \vdash^*_M \lambda$ which implies $w \vdash_M w' \vdash^*_M \lambda$ and $w \in L_C(M)$. ☐

Lemma 4.3.3. $L_C(M) \subseteq L(G') \cup \{\lambda\}$.

Proof. (By induction on the number of reduction steps)

Suppose $w \in L_C(M)$ and $w = w_n \vdash_M w_{n-1} \vdash_M \dots \vdash_M w_1 \vdash_M \lambda$. For each $i = 1, 2, \dots n$, let us prove that $w_i \in L(G') \cup \{\lambda\}$. $w_1 \vdash_M \lambda$ implies that there is the instruction $(¢, w_1 \to \lambda, \$)$ in I_1, and thus $w_1 \in L(G')$ according to the definition of I_1. Suppose that $w_j \in L(G')$ and in the reduction $w_{j+1} \vdash_M w_j$ we have used the instruction $\phi = (\Sigma \cup \{¢\}, z_1 \dots z_{s-1} \to \Delta^i, z_s) \in I_2$, i.e. $w_{j+1} = x\underline{z_1 \dots z_{s-1}}z_sy \vdash_M x\Delta^i z_s y = w_j$. We have $w_j \in L(G')$. Therefore, $N_1 \Rightarrow^*_{G'} x\Delta^i z_s y$ and (since ϕ is applicable to $x\underline{z_1 \dots z_{s-1}}z_s$) x is either λ, or ends with a letter from Σ. The sequence of letters $\Delta^i z_s$ in our derivation could have been created only by using the production rule $N_i \to \Delta^i z_s$. Thus there exists a derivation $N_1 \Rightarrow^*_{G'} xN_iy \Rightarrow_{G'} x\Delta^i z_s y$ in G'. From the definition of $\phi \in I_2$ we also have $N_i \Rightarrow^*_{G'} z_1 \dots z_{s-1}z_s$, where $p < s \leq q$. Thus in G' there exists a derivation $N_1 \Rightarrow^*_{G'} xN_iy \Rightarrow^*_{G'} xa_1 \dots a_{s-1}a_sy = w_{j+1}$, which immediately implies that $w_{j+1} \in L(G')$. ☐

This completes the proof of Theorem 4.3.1. ☐

We have shown that Δ^*cl-RA-automata are able to recognize all context-free languages (except the empty word – see Remark 3.2.1). This result opens an interesting question whether it is possible to transform each Δ^*cl-RA-automaton into an equivalent Δcl-RA-automaton. If we are interested only in the problem whether Δcl-RA-automata can recognize all context-free languages, then we do not need to do this transformation for all Δ^*cl-RA-automata. We just need to do this transformation to such Δ^*cl-RA-automata which were obtained from a context-free grammar, as was shown above. Moreover, the aforementioned construction can be generalized, i.e. we can put some extra restrictions on the instructions of the resulting Δ^*cl-RA-automaton. We can generalize the construction of the grammar G' and its corresponding Δ^*cl-RA M in the following three ways:

1. We can choose a minimal length $m_0 \geq 1$ of codes for nonterminals, i.e. we can code N_i by using $i + m_0 - 1$ consecutive letters Δ.

2. We can choose a minimal length $m_1 \geq 1$ of shortening for each reduction, i.e. for each instruction $(x, u \to \Delta^r, y)$ such that $r \geq 1$, we can guarantee that $|u| - |\Delta^r| \geq m_1$.

3. We can choose a length $k \geq 1$ of the separator. It means that instead of one letter we use k consecutive arbitrary letters from Σ as a separator.

Again suppose that we have a context-free grammar $G = (V_N, V_T, S, P)$ in Chomsky normal form with $V_N = \{N_1, \ldots, N_m\}$ and $S = N_1$. Let $\Delta \notin V_T$ and $G' = (V_N, V_T', S, P')$ be the grammar which we obtain from G by adding new terminal symbol Δ to V_T and adding new production rules $N_i \to \Delta^{i+m_0-1} z_1 z_2 \ldots z_k$ to P, for all $1 \leq i \leq m$ and all $z_1, z_2, \ldots, z_k \in V_T$.

Lemma 4.3.4. *For the grammar G' there exist natural numbers p, q, $m + m_0 + m_1 - 2 \leq p \leq q$, such that the following condition holds: if $w \in L(G')$ and $|w| > p$, then there exists a derivation $N_1 \Rightarrow^*_{G'} xN_iy \Rightarrow^*_{G'} xzy$ in G', where $w = xzy$ and $p < |z| \leq q$.*

Proof. The proof is the same as in Lemma 4.3.1 except that we set $p := m + m_0 + m_1 + k - 2$ and $q := 2p$. □

Let $\Sigma = V_T$ and $\Gamma = \Sigma \cup \{\Delta\} = V_T'$. The construction of a k-Δ^*cl-RA-automaton M recognizing $L(G') \cup \{\lambda\}$ is similar to the previous one (note that the length of the context is exactly the length of the separator):

First, we set $I_1 = \{(¢, w \to \lambda, \$) \mid w \in L(G'), |w| \leq p\}$.

For every $i \in \{1, 2, \ldots, m\}$, let $L_i = \{z \in \Gamma^* \mid N_i \Rightarrow^*_{G'} z, p < |z| \leq q\}$. For every such $z \in L_i$, $z = z_1 \ldots z_{s-k} z_{s-k+1} \ldots z_s$, let us consider the following instruction (set of instructions):

$$(\Sigma \cup \{¢\}, z_1 \ldots z_{s-k} \to \Delta^{i+m_0-1}, z_{s-k+1} \ldots z_s)\,.$$

Let I_2 be the set of all such instructions. Then $M = (\Sigma, I_1 \cup I_2)$ is the required automaton. Note that $|z_1 \ldots z_{s-k}| - |\Delta^{i+m_0-1}| = (s-k) - (i+m_0-1) \geq (p+1-k) - (m+m_0-1) = m+m_0+m_1+k-2+1-k-m-m_0+1 = m_1$.

The proof of $L(G') \subseteq L_C(M)$ ($L_C(M) \subseteq L(G') \cup \{\lambda\}$, respectively) is analogous to the proof of Lemma 4.3.2 (Lemma 4.3.3, respectively).

Lemma 4.3.5. *For each $t \geq 1$, we can set the parameters m_1 and k so that, for each instruction $\phi = (x, u \to \Delta^r, y) \in I_2$, there exists a subword $v \in \Sigma^*$ (i.e. not containing Δ) of u with length $|v| \geq t$.*

Proof. Set $k := t$ and $m_1 := 2t + m - 1$. Let us consider any $\phi = (x, u \to \Delta^{i+m_0-1}, y) \in I_2$. If $u \in \Sigma^*$, then we set $v := u$ and we obtain $|v| = |u| \geq (i+m_0-1) + m_1 \geq t$. Suppose that there are some letters Δ in u. If in u we can find two consecutive continuous sequences of letters Δ, then at least k letters from Σ separate these two sequences. We can set v to be this separator. Suppose that there is at most one continuous sequence of letters Δ, i.e. $u = w_1 \Delta^{j+m_0-1} w_2$ for some $w_1, w_2 \in \Sigma^*$ and $1 \leq j \leq m$. We know that $|w_1| + (j+m_0-1) + |w_2| - (i+m_0-1) \geq m_1$. On the other hand, $|w_1| + (j+m_0-1) + |w_2| - (i+m_0-1) = |w_1| + |w_2| + j - i \leq |w_1| + |w_2| + m - 1$, since $1 \leq i, j \leq m$. Accordingly, $|w_1| + |w_2| + m - 1 \geq m_1$. The bigger of the words w_1, w_2 has the length at least $\frac{1}{2}(m_1 - m + 1) = t$. □

In the following, we would like to outline what we think is a promising way to transform a k-Δ^*cl-RA-automaton M obtained from the previous construction into an equivalent Δcl-RA-automaton N. First suppose that we do this transformation in a trivial way, i.e. we transform each instruction $\phi = (x, u \to \Delta^i, y)$ of M, where $i > 1$ and $u = z_1 \ldots z_s$, into a set of so-called partial instructions:

$$\begin{aligned}
\phi_1 &= (x, z_1 \to \Delta, z_2 z_3 \ldots z_s y),\\
\phi_2 &= (x\Delta, z_2 \to \Delta, z_3 z_4 \ldots z_s y),\\
&\ldots\\
\phi_{i-1} &= (x\Delta^{i-2}, z_{i-1} \to \Delta, z_i z_{i+1} \ldots z_s y),\\
\phi_i &= (x\Delta^{i-1}, z_i z_{i+1} \ldots z_s \to \Delta, y).
\end{aligned}$$

Apparently, this technique gives us only the inclusion $L(M) \subseteq L(N)$. The other inclusion is not guaranteed. The problem is that we can find two different instructions ϕ and ψ in M such that they have different partial instructions ϕ_i and ψ_j applicable in the same context. One possible way how to avoid such situations is to introduce some new special instructions, which will encode some extra information into u by rewriting some letters with auxiliary Δ-symbols. Using Lemma 4.3.5 we can guarantee a long enough subword $v \in \Sigma^*$ in u, which we can use to encode this information. However, it is an open problem how to encode the extra information into this subword in order to avoid collisions between partial instructions.

Conclusion

We have successfully achieved the main goal of the thesis, which was to study more restricted models of restarting automata – called clearing restarting automata – which based on a limited context can either delete a substring of the current content of its tape or replace a substring by one (or more than one) auxiliary symbol Δ. We have investigated several properties of these automata including their relation to Chomsky hierarchy and possibilities for machine learning of these automata from positive and negative samples. Moreover, the results in this thesis were presented in two workshops: ABCD workshop, held in Prague, Czech Republic, March 27 – 29, 2009 (`http://ksvi.mff.cuni.cz/workshop/abcd09/`) and NCMA workshop, held in Wroclaw, Poland, August 31 – September 1, 2009 (`http://www.informatik.uni-giessen.de/ncma2009/`). An extetended version of the paper from the NCMA workshop was accepted for publication in Fundamenta Informaticae ([6]).

The formal study of clearing restarting automata has just started and several theoretical questions are still open or under investigation. In the following we present some concluding remarks and mention several open problem.

We have seen that knowing some sample computations (or reductions) of a k-cl-RA-automaton (or k-$\Delta\mathsf{cl}$-RA-automaton) it is extremely simple to infer its instructions. This implies an interesting property of being learnable for them. Moreover, the instructions of a $\Delta\mathsf{cl}$-RA-automaton are easily "human readable", which is an advantage for their possible applications, e.g. in linguistics.

$\mathcal{L}(\mathsf{cl}$-$\mathsf{RA})$ is incomparable with the class of string languages defined by ALD – associative language descriptions ([7]). An ALD can define the language L_1 from Section 3.4 ([7]), while L_1 cannot be recognized by any cl-RA-automaton (Theorem 3.4.1). On the other hand, no ALD can define a non-context-free string language, but cl-RA-automata can (Theorem 3.6.1).

Unfortunately, we still do not know whether $\Delta\mathsf{cl\text{-}RA}$-automata can recognize all context-free languages. If we generalize k-$\Delta\mathsf{cl\text{-}RA}$-automata by enabling them to use any number of auxiliary symbols: $\Delta_1, \Delta_2, \ldots, \Delta_n$ instead of single Δ, then we increase their power up-to context-sensitive languages. Such automata can easily accept all languages generated by context-sensitive grammars with productions of the following three forms only: $A \to a, A \to BC, AB \to AC$, where A, B, C are nonterminals and a is a terminal. This normal form of context-sensitive grammars is called *one-sided normal form* [30]. Penttonen showed that for every context-sensitive grammar there exists an equivalent grammar in one-sided normal form.

Many other questions are waiting for further research:

- What is the smallest alphabet Σ such that there exists a 2-$\mathsf{cl\text{-}RA}$-automaton $M = (\Sigma, I)$ recognizing a non-context-free language?
- What is the difference between language classes of $\mathcal{L}(k\text{-}\mathsf{cl\text{-}RA})$ and $\mathcal{L}(k\text{-}\Delta\mathsf{cl\text{-}RA})$ for different values of k?
- What is the difference between language classes $\mathcal{L}(\Delta\mathsf{cl\text{-}RA})$ and $\mathcal{L}(\Delta^*\mathsf{cl\text{-}RA})$?
- Can $\Delta\mathsf{cl\text{-}RA}$-automata recognize all context-free languages?
- Can $\Delta\mathsf{cl\text{-}RA}$-automata recognize all string languages defined by ALD's?
- What is the relation between $\mathcal{L}(\Delta\mathsf{cl\text{-}RA})$ and the class of one counter languages, simple context-sensitive grammars (they have single nonterminal), etc?

Bibliography

[1] Autebert J.-M., Berstel J., Boasson L. (1997): Context-Free Languages and Pushdown Automata, *Handbook of Formal Languages, Volume I. Word, Language, Grammar*, (G. Rozenberg, A. Salomaa, Eds.), Springer-Verlag New York, Inc., 111–174.

[2] Balcázar J. L., Díaz J., Gabarró J. (1988): Structural Complexity I, *EATCS Monographs on Theoretical Computer Science* (W. Brauer, G. Rozenberg, A. Salomaa, Eds.), Volume 11, Springer–Verlag Berlin Heidelberg.

[3] Balcázar J. L., Díaz J., Gabarró J. (1990): Structural Complexity II, *EATCS Monographs on Theoretical Computer Science* (W. Brauer, G. Rozenberg, A. Salomaa, Eds.), Volume 22, Springer–Verlag Berlin Heidelberg.

[4] Buntrock G., Otto F. (1998): Growing Context-Sensitive Languages and Church-Rosser Languages, *Information and Computation* **141**(1), 1–36.

[5] Černo P. (2008): Bachelor Thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2008, Prague.

[6] Černo P., Mráz. F. (2009): Clearing Restarting Automata, *Workshop on Non-Classical Models for Automata and Applications (NCMA)* (H. Bordinh, R. Freund, M. Holzer, M. Kutrib, and F. Otto, Eds.), Volume 256 of *books@ocg.at*, Österreichisches Computer Gesellschaft, 77–90. An extetended version of the paper was accepted for publication in Fundamenta Informaticae.

[7] Cherubini A., Reghizzi S. C., Pietro P. S. (2002): Associative Language Descriptions, *Theoretical Computer Science* **270**(1-2), 463–492.

[8] Dahlhaus E., Warmuth M. K (1986): Membership for Growing Context-sensitive Grammars is Polynomial, *Journal of Computer and System Sciences* **33**(3), 456–472.

[9] Ehrenfeucht A., Păun G., Roszenberg G. (1997): Contextual Grammars and Formal Languages, *Handbook of Formal Languages, Volume II. Linear Modeling: Background and Application*, (G. Rozenberg, A. Salomaa, Eds.), Springer-Verlag New York, Inc., 237–293.

[10] Greibach S. A. (1973): The Hardest Context-Free Language, *SIAM Journal on Computing* **2**(4), 304–310.

[11] Hofbauer D., Waldmann J. (2004): Deleting String Rewriting Systems Preserve Regularity, *Theoretical Computer Science* **327**(3), 301–317.

[12] Hopcroft J. E., Motwani R., Ullman J. D. (2000): Introduction to Automata Theory, Languages, and Computation, *Addison Wesley*, 2nd edition, November 2000.

[13] Jančar P., Mráz F., Plátek M., Vogel J. (1995): Restarting Automata, *Fundamentals of Computation Theory, FCT 1995, Proc.* (H. Reichel, Ed.), LNCS 965, Springer, 283–292.

[14] Jančar P., Mráz F., Plátek M., Vogel J. (1997): On Restarting Automata with Rewriting, *New Trends in Formal Languages* (G. Paun, A. Salomaa, Eds.), LNCS 1218, Springer, 119–136.

[15] Jančar P., Mráz F., Plátek M., Vogel J. (1998): Different Types of Monotonicity for Restarting Automata, *FSTTCS 1998, Proc.* (V. Arvind, R. Ramanujam, Eds.), LNCS 1530, Springer, 343—354.

[16] Jančar P., Mráz F., Plátek M., Vogel J. (1999): On Monotonic Automata with a Restart Operation, *Journal of Automata, Languages and Combinatorics* **4**(4), 287–312.

[17] Jančar P., Mráz F., Plátek M., Vogel J. (2007): Monotonicity of Restarting Automata, *Journal of Automata, Languages and Combinatorics* **12**(3), 355–371.

[18] Kutrib M., Messerschmidt H., Otto F. (2008): On Stateless Two-Pushdown Automata and Restarting Automata, *Automata and Formal*

Languages, AFL, Proc. (E. Csuhaj-Varjú, Z. Ésik, Eds.), Computer and Automation Research Institute, Hungarian Academy of Sciences, Budapest, 257–268.

[19] Lopatková M., Plátek M., Kuboň V. (2005): Modeling Syntax of Free Word-Order Languages: Dependency Analysis by Reduction, *Text, Speech and Dialogue, TSD 2005, Proc.* (V. Matousek, P. Mautner, T. Pavelka, Eds.), LNCS 3658, Springer, 140–147.

[20] Marcus S. (1969): Contextual Grammars, *Revue Roum. Mathy. Pures Appl.* **14**(10), 1525–1534.

[21] Marcus S. (1997): Contextual Grammars and Natural Languages, *Handbook of Formal Languages, Volume II. Linear Modeling: Background and Application*, (G. Rozenberg, A. Salomaa, Eds.), Springer-Verlag New York, Inc., 215–235.

[22] Mateescu A., Salomaa A. (1997): Aspects of Classical Language Theory, *Handbook of Formal Languages, Volume I. Word, Language, Grammar*, (G. Rozenberg, A. Salomaa, Eds.), Springer-Verlag New York, Inc., 175–251.

[23] Mateescu A., Salomaa A. (1997): Formal Languages: an Introduction and a Synopsis, *Handbook of Formal Languages, Volume I. Word, Language, Grammar*, (G. Rozenberg, A. Salomaa, Eds.), Springer-Verlag New York, Inc., 1–39.

[24] Maurer H. A., Salomaa A. K., Wood D. (1980): Pure Grammars, *Information and Control* **44**(1), 47–72.

[25] McNaughton R., Narendran P., Otto F. (1988): Church-Rosser Thue Systems and Formal Languages, *Journal of the ACM (JACM)* **35**(2), 324–344.

[26] Mráz F., Otto F., Plátek M. (2006): Learning Analysis by Reduction from Positive Data, *Grammatical Inference: Algorithms and Applications, ICGI 2006, Proc.* (Y. Sakakibara, S. Kobayashi, K. Sato, T. Nishino, E. Tomita, Eds.), LNCS 4201, Springer, 125–136.

[27] Otto F. (2006): Restarting Automata, *Recent Advances in Formal Languages and Applications*, (Z. Esik, C. Martín-Vide, V. Mitrana, Eds.),

Volume 25 of *Studies in Computational Intelligence*, Springer-Verlag Berlin Heidelberg, 269–303.

[28] Păun G. (1998): Marcus Contextual Grammars, Volume 67 of *Studies in Linguistics and Philosophy*, Springer.

[29] Plátek M. (2001): Two-Way Restarting Automata and J-Monotonicity, *SOFSEM 2001, Proc.* (L. Pacholski, P. Ružička, Eds.), Volume 2235 of *Lecture Notes in Computer Science*, Springer-Verlag Berlin Heidelberg, 316–325.

[30] Rozenberg G., Salomaa A., Eds. (1997): Handbook of Formal Languages, Volume I. Word, Language, Grammar, Springer-Verlag New York, Inc.

[31] Sheng Yu (1997): Regular Languages, *Handbook of Formal Languages, Volume I. Word, Language, Grammar*, (G. Rozenberg, A. Salomaa, Eds.), Springer-Verlag New York, Inc., 41–110.

Index

2^S, 5
LC_k, 45
$LC_{\leq k}$, 47
RC_k, 45
$RC_{\leq k}$, 47
ALD, 32–35, 48, 89, 90
Accept, 39–41
CFL, 11, 21, 24, 42, 62, 73, 75
CR language, 31
CR system, 30, 31
  defining, 31
CR-decidable language, 31
CSL, 21, 22, 24, 55
DCFL, 18, 42
DFA, 6–9
  complete, 7
  reduced, 7
Δ^*cl-RA-automaton, 83, 85, 90
Δcl-RA-automaton, 77, 82, 83, 86, 87, 89, 90
Δ-clearing restarting automaton, *see* $(k-)\Delta$cl-RA-automaton, 75
Δ^*-clearing restarting automaton, *see* $(k-)\Delta^*$cl-RA-automaton, 75, 82
EC, 27
ECC, 27
ECC$_c$, 27
FA, 6, 9
Fin, 4
GCSL, 24
IC, 27
ICC, 27
ICC$_c$, 27
$Int_k(u)$, 3, 5
LBA, 23
Lin, 19, 24
MVL, 38–40
MVR, 38–40
NFA, 7–9
NNFA, 8
Ocl, 20
PCF, 29
PLI, 29
$Pref_k(u)$, 2, 5, 55
R-automaton, 42, 50
RE, 21, 22, 24
RL-automaton, 41, 42
RLW-automaton, 41
RLWW-automaton, 37, 40–42
RR-automaton, 42, 48, 49, 52, 55
RRW-automaton, 42
RRWW-automaton, 37, 40–42
RW-automaton, 42
RWW-automaton, 36, 37, 41, 42
Reg, 21, 24
Restart, 39
Rewrite, 39–42
$Suff_k(u)$, 2, 5, 55
TC, 27

TC_c, 27
cl-RA-automaton, 43, 48–53, 55, 56, 61, 73, 75–77, 79, 89
det-, 39
dpda, 18
k-CRS-system, 45–47, 53, 77, 83
1-CRS-system, 58, 59, 61, 67
2-CRS-system, 69
k-Δ^*cl-RA-automaton, 83, 86, 87
1-Δ^*cl-RA-automaton, 83, 84
k-Δcl-RA-automaton, 77, 89, 90
1-Δcl-RA-automaton, 77, 78
k-cl-RA-automaton, 43, 47, 48, 50, 55, 89, 90
1-cl-RA-automaton, 44, 48, 52, 53, 62, 73
2-cl-RA-automaton, 44, 63–65, 68, 73, 90
3-cl-RA-automaton, 44, 68, 69, 73
4-cl-RA-automaton, 43, 55, 57
λ-NFA, 8, 9
λ-closure, 8
λ-free morphism, *see* morphism
λ-free substitution, *see* substitution
$\mathcal{L}_0$, 21, 22
$\mathcal{L}_1$, 21, 22
$\mathcal{L}_2$, 21
$\mathcal{L}_3$, 21
$\mathcal{P}(S)$, 5
mon-, 42
pda, 17–20
pdm, 16–18
  quadratic form, 17

accepting configuration, *see* configuration
alphabet, 2
  binary, 3
  input, 6, 37, 40, 41, 45
  one-letter, 43, 50, 51
  tape, 38, 41
  working, 45
analysis by reduction, 35, 36
associative language description, *see* ALD, 25, 32, 33, 48, 50, 77, 89
auxiliary symbol, 36, 39, 41, 42, 77, 89, 90
axiom, 12, 22, 26, 29

Boolean operation, *see* operation

canonical form, 31
catenation, *see* concatenation, 2
choice mapping, 26
Chomsky hierarchy, 20, 28, 89
Chomsky normal form, *see* normal form
Church-Rosser language, *see* language, CR-language
Church-Rosser property, 30, 31
Church-Rosser string rewriting system, 25, 29
Church-Rosser system, *see* CR system
Church-Rosser-decidable language, *see* language, CR-decidable language
circle-square jump, 66
circle-square recovery, 66
circle-square representation, 44, 58, 66, 67, 69
clearing restarting automaton, *see* ($k-$)cl-RA-automaton, 25, 43, 47, 53, 75, 89
  meta-algorithm, 55
closure properties, 24, 43, 52
complementation, 3, 24

- computation, 8, 17, 19, 31, 39–42, 49, 56, 76, 89
  - accepting, 41
  - monotone, 42
  - relation, 7
  - step, 7
  - valid, 16
- concatenation, 2, 3, 24, 53
  - closure, 3
- configuration, 7, 16, 39, 41, 42
  - accepting, 7, 17, 39
  - halting, 39
  - initial, 39, 40, 42
  - internal, 16
    - accepting, 17
    - accessible, 17
    - co-accessible, 17
    - initial, 17
  - rejecting, 39
  - restarting, 39
  - starting, 7, 17
- constituent, 32
- constraint, 41
- context, 26, 53
  - left, 33, 45, 50
  - right, 33, 45
- context extension theorem, 46
- context rewriting system, *see* $(k-)\mathsf{CRS}$, 43, 45
  - dual, 46
- context-free grammar, *see* grammar
- context-free language, *see* language, CFL, $\mathcal{L}_2$
- context-sensitive grammar, *see* grammar
- context-sensitive language, *see* language, CSL, $\mathcal{L}_1$
- contextual grammar, 25, 26, 32
  - total, *see* TC, TC_c, 26
    - external, *see* ECC, ECC_c, 26
    - internal, *see* ICC, ICC_c, 26
  - without choice, *see* EC, IC, 27
- copy, 14
- correctness preserving theorem, 40, 47
- cycle, 39–42
  - left distance, 42
  - right distance, 42

- derivation, 12, 24, 26
  - direct, 22, 26
  - external, 27
  - internal, 27
  - leftmost, 13
  - relation, 22
  - rightmost, 13
- derivation rule, *see* production, 12
- derivation tree, *see* tree
- dual system, 46
- duality theorem, 46
- Dyck language, 14, 20, 53
  - semi-Dyck language, 76

- empty word, *see* word
- error preserving theorem, 40, 47, 49
- ether, 44, 63, 64, 68, 70
  - property, 63, 66

- factor, 2, 70
  - weight, 70
- factorization, 2, 70
- finite automaton, *see* FA, 4, 6
  - deterministic, *see* DFA, 6
  - extended, 9
  - nondeterministic, *see* NFA, NNFA, λ-NFA, 7

finite substitution, *see* substitution
finite transducer, 8
frontier, 32

grammar
  context-free, 11, 12, 21, 35, 62, 83, 86
  context-sensitive, 21, 22, 24, 90
    growing, 24
    one-sided normal form, *see* normal form
  left-linear, 6, 19
  length-increasing, 22
  linear, 19
  monotonous, 22
  parenthetic, 20
  phrase-structure, 21, 22
  proper, 14
  pure parenthetic, 20
  regular, 19, 21
  right-linear, 6, 19
  simple, 20
  trim, 13
  type 0, 21, 22
  type 1, 21, 22
  type 2, 21
  type 3, 21
  unambiguous, 13
  very simple, 20
Greibach normal form, *see* normal form
Greibach's hardest context-free language, *see* hardest context-free language
growing context-sensitive grammar, *see* grammar
growing context-sensitive language, *see* language, GCSL

halting configuration, *see* configuration
hardest context-free language, 32, 75, 82

identity condition, 30
initial configuration, *see* configuration
instruction, 45, 89
  dual, 46
  prefix-rewriting, 50
  suffix-rewriting, 50
  width, 47
interior words, *see* $Int_k(u)$, 3
intersection, 3, 24, 53
  with regular language, 24, 53, 56, 61
inverse morphism, *see* morphism
inverse substitution, *see* substitution

jump, 64, 66

Kleene plus, 3
Kleene star, 3, 24

language, 3
  k-testable, 5
  ambiguous, 13
  atomic, 4
  characteristic, 77, 78, 83
  Church-Rosser, *see* CR-language, 31
    congruential, 31
  Church-Rosser-decidable, *see* CR-decidable language, 31
  context-free, *see* CFL, $\mathcal{L}_2$, 11, 12, 18, 20, 21, 24, 35, 37, 42–44, 51, 52, 55, 61, 75, 76, 82, 83, 85, 90

        - deterministic, *see* DCFL, 18, 31, 37
        - deterministic-prefix, 18
    - context-sensitive, *see* CSL, $\mathcal{L}_1$, 21, 22, 24, 55, 90
        - growing, *see* GCSL, 21, 42
    - empty, 4
    - finite, *see* Fin, 3, 4
    - infinite, 3, 4
    - linear, *see* Lin, 19
    - locally testable, 5
    - mirror image, 3
    - non-context-free, 43, 44, 55, 56, 63, 68, 89, 90
    - noncounting, 5
    - one-counter, *see* Ocl, 20, 90
    - parenthetic, 20
    - pure, *see* PCF, PLI, 29
    - pure parenthetic, 20
    - recursively enumerable, *see* RE, $\mathcal{L}_0$, 21, 22, 47
    - regular, *see* Reg, $\mathcal{L}_3$, 4, 6, 10, 19, 21, 29, 35, 41, 43, 50, 51, 65
    - reversal, 3
    - simple, 20
    - star-free, 4
    - stencil tree, 34
    - type 0, *see* $\mathcal{L}_0$, 21
    - type 1, *see* $\mathcal{L}_1$, 21
    - type 2, *see* $\mathcal{L}_2$, 21
    - type 3, *see* $\mathcal{L}_3$, 21
    - unambiguous, 13, 19
- language recognized by
    - accepting states, 17, 18
    - empty storage, 17, 18
- left constraint, *see* constraint
- left context, *see* context
- left terminator, *see* terminator
- left-derivate, 3, 24
- left-linear grammar, *see* grammar
- left-quotient, 4, 24
    - with regular language, 24
- length, 2
- letter, 2
- linear bounded automaton, *see* LBA, 23, 55
- linear grammar, *see* grammar
- linear language, *see* language, Lin

- Marcus contextual grammar, *see* contextual grammar
- marker, 36–39, 41
- maximal subtree, 32
- Mealy machine, 8
- membrane, 63, 68
- meta-instruction, 41, 48
    - rewriting, 48
- metasymbol, 32
- metavariable, 32
- mirror image, 2, 24
- mode of acceptance, 17
- Moore machine, 8
- morphism, 5, 24, 53
    - λ-free, 5, 24
    - inverse, 5, 24, 44, 75, 76
- move-left operation, *see* transition step, MVL
- move-right operation, *see* transition step, MVR
- Myhill-Nerode theorem, 11

- noncounting property, 32
- nonterminal, 12, 22, 28, 32, 83, 86
- normal form, 12, 14
    - Chomsky, 14, 83, 86
    - cubic double Greibach, 15

- double Greibach, 15
- Greibach, 14, 15
- one-sided normal form, 90
- operator, 14, 15
- quadratic double Greibach, 15
- quadratic Greibach, 15
- weak Chomsky, 14

one-sided normal form, *see* normal form

operation
- Boolean, 3, 4
- regular, 4

overlap condition, 30

parse tree, *see* tree

Pascal, 24, 35

pattern, 33

permissible context, 33

phrase-structure grammar, *see* grammar

placeholder, 32

power set, *see* 2^S, $\mathcal{P}(S)$, 5

prefix, *see* $Pref_k(u)$, 2

production, 12, 22, 24, 29, 50, 83
- λ-production, 13
- relation, 50
- terminal, 13

production language, 45
- characteristic, 45

pumping lemma, 10
- context-free language, 15
- regular language, 10

pure grammar, 25–29, 32
- context-free, *see* PCF, 29
- context-sensitive, 29
- length increasing, *see* PLI, 29

pushdown automaton, *see* pda, 11, 15, 17, 31, 76
- deterministic, *see* dpda, 18
- finite-turn, 19
- one-counter, 20
- one-turn, 19
- realtime, 18
- simple, 18, 20

pushdown machine, *see* pdm, 16
- deterministic, 18
- realtime, 16
- simple, 16

read/write window, 36–39, 41
- possible contents, 38

recovery, 44, 63–66

recursively enumerable language, *see* language, RE, $\mathcal{L}_0$

reduction, 30, 50, 86, 89
- relation, 50

reduction language, 45
- characteristic, 45

regular expressions, 6, 9

regular language, *see* language, Reg, $\mathcal{L}_3$

regular operation, *see* operation

regular substitution, *see* substitution

rejecting configuration, *see* configuration

restart operation, *see* transition step, Restart

restarting automaton, *see* RRWW-automaton, 25, 35–37, 40, 41, 43, 48, 89
- deterministic, 39, 40, 42, 56
- monotone, 37, 42
- nondeterministic, 39, 40
- two-way, 37, 39
- with rewriting, 36

restarting configuration, *see* configuration
reversal, *see* mirror image, 2
rewrite operation, *see* transition step, Rewrite
rewriting relation, 40, 45, 48, 77, 83
    dual, 46
    in the context, 45
rewriting rule, 22, 29, 50
    prefix-rewriting, 50
    suffix-rewriting, 50
right constraint, *see* constraint
right context, *see* context
right terminator, *see* terminator
right-derivate, 4, 24
right-linear grammar, *see* grammar
right-quotient, 4, 24
    with regular language, 24
rule, 16, 30, 33, 45, 50
    λ-rule, 16
    decreasing, 17
    increasing, 17
    length-reducing, 30
    prefix-rewriting, 50
    stationary, 17
    suffix-rewriting, 50

scattered subword, *see* subword
sentinel, 45
    left, 50
set difference, 53
shuffle, 3, 4, 24
signal, 44, 63–65, 67, 68
    spreading, 64, 67
start symbol, *see* axiom, 22
state, 6, 37, 42
    accepting, 17
    current, 39
    final, 7
    initial, 36, 38, 39
    starting, 6, 8
stencil tree, *see* tree
step, *see* transition step
string, 2
string language, 34
string rewriting system, 50
substitution, 5, 24
    λ-free, 5, 24
    finite, 5
    inverse, 6
    regular, 5
substring condition, 30
subword, 2
    scattered, 2, 41
suffix, *see* $\mathit{Suff}_k(u)$, 2
symbol, 2
    basic, 63, 68
    wave, 63

tail, 39, 41, 42
tape, 36, 37, 39
terminal, 12, 22, 28, 83
terminal letter, *see* terminal
terminator
    left, 33
    right, 33
Thue system, 29
transition, *see* transition step, 7, 16
    λ-transition, 7, 9, 16
    function, 6, 7
    relation, 16, 38
transition step, 38
    accept, *see* Accept, 39, 41
    delete/restart, 36

  - move-left, *see* MVL, 37, 38
  - move-right, *see* MVR, 36
  - restart, *see* Restart, 36, 37, 39, 41
  - rewrite, *see* Rewrite, 37, 38, 41
  - rewrite/restart, 36
- tree
  - derivation, 13
  - parse, 13
  - stencil, 32
    - valid, 34
- Turing machine, 22

- union, 3, 24, 53

- variable, *see* nonterminal, 12

- word, 2
  - empty, 2, 49, 75
- workspace theorem, 23

- yield, 13