HABILITATION THESIS

Pushdown Automata: New Modifications and Transformations

Dušan Kolář
Department of Information Systems
Faculty of Information Technology
Technical University of Brno

January '01 – September '04
Version 1.0

Abstract

Pushdown automata play a key role in the efficient syntax analysis of context-free languages (in particular, the languages mentioned should belong either to the set of LL_1 languages or to the set of $(\mathrm{LA})\mathrm{LR}_1$ languages; both of these sets belong to the set of all context-free languages). The great advantage is also that the construction of the pushdown automata for the particular language described by a proper grammar is straightforward. On the other hand, the efficiency of automata constructed for, for instance, LL_2 languages is not as good as for LL_1 languages. Moreover, we cannot use pushdown automata for analysis of context-sensitive languages and thus their power is far below the one of Turing machine.

This thesis demonstrates a transformation of pushdown automata to achieve efficient behaviour even for LL_2 and other more powerful LL_k languages. After these introductory pages, an extension of pushdown automata, which increases their power to the one of Turing machine is presented. Extended automata are further studied to clarify their possibilities and limitations. Especially, we propose an extended pushdown automaton together with an algorithm of its construction, which can be used for the efficient analysis of languages, a power of which is higher than that for context-free languages.

Many thanks to all who encouraged and helped me when working on this thesis.

Contents

Chapter 1

Preface

This is a Habilitation thesis which aspires to summarise certain new extensions and utilizations of pushdown automata. The basic research on this topic was started in 1999 and the work presented shows the latest results from the first half of 2004. Nevertheless, many conclusions come from practical development and research performed at the Technical University of Brno, Faculty of Information Technology (former Department of Computer Science and Engineering, Faculty of Electrical Engineering and Computer Science) over programming languages & their compilers since 1995.

Pushdown as an abstract data structure and pushdown automata have played an important role in computer science for many years already. In particular, we can recognise them in quite a few places in compilers, for instance. Nevertheless, it seems like the usage of these formal elements can be seen only within certain ”schemes”—almost no new modifications and terms of usage can be seen. The reason we could not find any extensions to these formalisms, may be explained by the increasing power of hardware and, moreover, in the development of other techniques, which do not require higher reasoning concerning new features (of these formalisms), and in the exploitation of ”traditional” programming structures. Thus, compilers built these days are built over context-free languages and contextual dependencies are verified using symbol tables and other appropriate techniques. Moreover, the recursive descent approach of syntactic analysis (based on LL_1 grammars) is adopted where possible. This is probably because of the possibility that the work with various kinds of attributed grammars and syntax driven translation is very attractive and it gives high expressive power to programmers. That is also a reason this approach is adopted even for languages a grammar of which cannot be described by LL_1 grammars (for example GNU C++ compiler).

Nevertheless, programming languages become more and more complicated for translation (either by their evolution—C++, or by simply inventing a new language). This situation motivated us to start work on simpler and more powerful description of programming languages, their analysis, and compiler construction. The first necessary step is presented in this thesis—bringing to life new concepts applicable in syntactic analysis. Such concepts enable a programmer and language designer to have higher expressive power during design and implementation.

The thesis is structured into eight chapters, this is Chapter 1. Chapter 2 contains mainly definitions used further in the thesis and thus introduces the broad aspect of the topic. Chapter 3 deals with pushdown automata used for analysis of LL_k analysis, where wider contexts ($k > 1$) are difficult to implement and a transformation to automata with single symbol context is presented. Chapters 4 and 5 introduce regulated pushdown automata and study their possible minimisation. As this is quite a new concept the introduction is provided in a broader way. Deterministic regulated pushdown automata are newly introduced in Chapter 6 together with some reasoning about them. Finally, Chapter 7 presents exploitation of the concept of deterministic regulated pushdown automata in syntactic analysis of languages. In particular, analysis of languages based on scattered context grammars is presented. Last, but not least, is Chapter 8, where the thesis is briefly concluded as a whole to give overall summary as the chapters are concluded separately.

Chapter 2

Preliminaries

This chapter summarises some known terms and techniques from the area of formal languages and automata, below (see, for instance, [53]). If the reader is familiar with the theory of formal languages and automata and their application in compilers he/she can skip to the next chapter.

2.1 General Preliminaries

This section introduces the notation of natural and integral numbers used below in the thesis. Set $\mathcal{N} = \{1, 2, \ldots\}$ and $\mathcal{I} = \{0, 1, 2, \ldots\}$.

Moreover, we define for a set, X, $card(X)$ to denote its cardinality.

2.2 Formal Languages Theory Preliminaries

First of all a set of all strings over an alphabet is defined with respect to the operation of concatenation and a language:

Definition 2.2.1 *A* semigroup $\mathcal{S} = (S, .)$ *is a set* S *(carrier of* $\mathcal{S}$*), with an associative operation '.' (a semigroup multiplication). A* monoid $\mathcal{M} = (S, ., 1)$ *is a semigroup* $\mathcal{S} = (S, .)$*, with a* unit element *1 such that* $a.1 = 1.a = a$*, for each* $a \in S$.

Definition 2.2.2 *Let* V *be an alphabet.* V^* *represents the free monoid generated by* V *under the operation of concatenation. The unit of* V^* *is denoted by* ε*. Set* $V^+ = V^* - \{\varepsilon\}$*; algebraically,* V^+ *is thus the free semigroup generated by* V *under the operation of concatenation.*

Definition of language can be derived straightforwardly as a (proper) subset of a set of all strings over a given alphabet. Indeed, for example, not all sequences of English words compose an English sentence.

Definition 2.2.3 *A language L with respect to the V^* is defined as $L \subseteq V^*$.*

Next, we define a set of operations that can be performed over a string over the given alphabet:

Notation 2.2.1 *For $w \in V^*$, $|w|$ denotes the length of w.*

Notation 2.2.2 *For $w \in V^*$, reversal(w) denotes the reversal of w.*

Notation 2.2.3 *For $w \in V^*$ set prefix$(w) = \{x \mid x \text{ is a prefix of } w\}$.*

Notation 2.2.4 *For $w \in V^*$ set suffix$(w) = \{x \mid x \text{ is a suffix of } w\}$.*

Notation 2.2.5 *For $w \in V^*$ set alph$(w) = \{a \mid a \in V, \text{ and } a \text{ appears in } w\}$.*

Notation 2.2.6 *For $w \in V^+$ and $i \in \{1, \dots, |w|\}$, sym(w, i) denotes the ith symbol of w; for instance, sym$(abcd, 3) = c$.*

Another possibility to define a language is via *grammar*. Of course, a (formal) grammar is quite a well known term in the area of information technology. In general, we can define a grammar the following way:

Definition 2.2.4 *A grammar, G, is a quadruple $G = (N, T, P, S)$, where N is a final set of non-terminals, T is a final set of terminals, $T \cap N = \emptyset$, P is a final set of production rules, it is a subset of $(N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$, an element $(\alpha, \beta) \in P$ will be written $\alpha \to \beta$, and the symbol S is the starting non-terminal, $S \in N$.*

To define language defined by a grammar, we have to define the term *derivation*. A definition of it, together with a definition of language defined by a grammar, follows:

Definition 2.2.5 *If $\alpha \to \beta \in P$ and $u, v, \beta \in (N \cup T)^*$, $\alpha \in (N \cup T)^* N (N \cup T)^*$, then $u\alpha v \Rightarrow u\beta v\ [\alpha \to \beta]$ or, simply, $u\alpha v \Rightarrow u\beta v$ is called a* simple derivation. *In the standard manner, extend $\Rightarrow$ to $\Rightarrow^n$, where $n \geq 0$; then, based on $\Rightarrow^n$, define $\Rightarrow^+$ and $\Rightarrow^*$, a* (general) derivation.

The language *of G, $L(G)$, is defined as $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$.*

By restricting the form of production rules used for a grammar definition, we can recognise several kinds of grammars and languages defined by such grammars. Next are defined some kinds of them (for extension see, for example, [53, 19]).

Definition 2.2.6 *A* context-sensitive grammar*, G, restricts P (a finite set of productions) such a way, so that for every $\alpha \to \beta \in P$: $|\alpha| \leq |\beta|$. If an empty string (ε) is in the language a special rule is allowed in P: $S \to \varepsilon$, where S is the starting non-terminal.*

A language, L, is context-sensitive *if and only if $L = L(G)$, where G is a context-sensitive grammar.*

Definition 2.2.7 *A* context-free grammar*, G, restricts P (a finite set of productions) to the form $A \to x$, where $A \in N$ and $x \in (N \cup T)^*$. If there are several rules of the form $A \to \alpha_1$, $A \to \alpha_2$, …, $A \to \alpha_n$, where $A \in N$, $\alpha_i \in (N \cup T)^*$ for $i \in \{1, \ldots, n\}$, α_i are mutually different, then we can write them in the form $A \to \alpha_1|\alpha_2|\ldots|\alpha_n$.*

A language, L, is context-free *if and only if $L = L(G)$, where G is a context-free grammar.*

An example of a simple grammar defining additive and multiplicative expressions with brackets and traditional precedence is presented next:
Let $G = (N, T, P, S)$ be a grammar defining such expressions, then

$$\begin{aligned} N &= \{S, M, B\} \\ T &= \{+, -, *, /, (,), num\} \\ P &= \{S \to M + S, S \to M - S, S \to M, M \to B * M, S \to B/M, S \to \\ & B, B \to (S), B \to num\} \end{aligned}$$

For better readability, we usually join right-hand-sides of the rules from P together and delimit them by a pipe, |. Thus we could obtain for P from our example such a form:
$P = \{S \to M + S \mid M - S \mid M, M \to B * M \mid B/M \mid B, B \to (S) \mid num\}$
which can also be written in a more readable form, where we devote each line to one nonterminal on the left-hand-side of the production rule:

$$\begin{aligned} P = \{& \\ & S \to M + S \mid M - S \mid M, \\ & M \to B * M \mid B/M \mid B, \\ & B \to (S) \mid num \\ \}& \end{aligned}$$

Definitions of other grammar categories continues next.

Definition 2.2.8 *A* linear grammar*, G, restricts P (a finite set of productions) to the form $A \to x$, where $A \in N$ and $x \in T^*(N \cup \{\varepsilon\})T^*$, where ε stands for empty string.*

A language, L, is linear *if and only if $L = L(G)$, where G is a linear grammar.*

Definition 2.2.9 *A* regular grammar*, G, restricts P (a finite set of productions) to the form $A \to x$, where $A \in N$ and $x \in T(N \cup \{\varepsilon\})$.*

A language, L, is regular *if and only if $L = L(G)$, where G is a regular grammar.*

Every category of the grammar defines a set of languages described by all grammars of a particular type. Next we introduce abbreviations to recognise these particular language sets.

Definition 2.2.10 *A set of all languages described by non-restricted grammar from Definition 2.2.4 is called a family of* recursively enumerable *languages and it will be referenced by abbreviation RE.*

A set of all languages described by context-sensitive grammar from Definition 2.2.6 is called a family of context-sensitive *languages and it will be referenced by abbreviation CS.*

A set of all languages described by context-free grammar from Definition 2.2.7 is called a family of context-free *languages and it will be referenced by abbreviation CF.*

A set of all languages described by linear grammar from Definition 2.2.8 is called a family of linear *languages and it will be referenced by abbreviation LIN.*

A set of all languages described by regular grammar from Definition 2.2.9 is called a family of regular *languages and it will be referenced by abbreviation REG.*

Besides the previously presented definition of a grammar, there are even some other possibilities of a grammar definition. Next, we present another grammar definition and a language defined by it, which is further used in this thesis:

Definition 2.2.11 *A* queue grammar *(see [44]) is a six-tuple, $Q = (V, T, W, F, S, P)$, where V (terminals together with nonterminals) and W (state-like representation symbols) are alphabets satisfying $V \cap W = \emptyset$, T stands for terminals, $T \subseteq V$, F stands for final states, $F \subseteq W$, S is a starting pair nonterminal&state, $S \in (V-T)(W-F)$, and $P \subseteq (V \times (W-F)) \times (V^* \times W)$ is a finite relation such that for every $a \in V$, there exists an element $(a, b, x, c) \in P$. If $u, v \in V^*W$ such that $u = arb$, $v = rzc$, $a \in V$, $r, z \in V^*$, $b, c \in W$ and $(a, b, z, c) \in P$, then $u \Rightarrow v\ [(a, b, z, c)]$ in G or, simply, $u \Rightarrow v$. In the standard manner, extend $\Rightarrow$ to $\Rightarrow^n$, where $n \geq 0$. Based on $\Rightarrow^n$, define $\Rightarrow^+$ and $\Rightarrow^*$.*

The language of Q, $L(Q)$, is defined as $L(Q) = \{w \in T^* \mid S \Rightarrow^* wf \text{ where } f \in F\}$.

Next is presented a slight modification of the notion of a queue grammar and of a language defined by such grammar.

Definition 2.2.12 *A* left-extended queue grammar *is a six-tuple, $Q = (V, T, W, F, S, P)$, where V, T, W, F, S, P have the same meaning as in a queue grammar; in addition, assume that $\# \notin V \cup W$. If $u, v \in V^*\{\#\}V^*W$ so $u = w\#arb$, $v = wa\#rzc$, $a \in V$, $r, z, w \in V^*$, $b, c \in W$, and $(a, b, z, c) \in P$, then $u \Rightarrow v\ [(a, b, z, c)]$ in G or, simply, $u \Rightarrow v$. In the standard manner, extend $\Rightarrow$ to $\Rightarrow^n$, where $n \geq 0$. Based on $\Rightarrow^n$, define $\Rightarrow^+$ and $\Rightarrow^*$.*

The language of Q*,* $L(Q)$*, is defined as* $L(Q) = \{v \in T^* \mid \#S \Rightarrow^* w\#vf$ for some $w \in V^*$ and $f \in F\}$*.*

The modification provides, in fact, as a part of the derivation string, complete information about the states the derivation goes through (placed left of the symbol #). Thus, the features of the left-extended queue grammar are the same as those of the queue grammar.

2.3 Formal Automata Theory Preliminaries

This section formally defines automata used in this thesis. We start with a definition of pushdown automata:

Definition 2.3.1 *A* pushdown automaton *(PA) is a 7-tuple,* $M = (Q, \Sigma, \Omega, R, s, S, F)$*, where* Q *is a* finite set of states*,* Σ *is an* input alphabet*,* Ω *is a* pushdown alphabet*,* R *is a* finite set of rules *of the form* $Apa \rightarrow wqb$*, where* $A \in \Omega$*,* $p, q \in Q$*,* $a \in \Sigma \cup \{\varepsilon\}$*,* $w \in \Omega^*$ *and* $b \in \{a, \varepsilon\}$ *(if* $b \neq \varepsilon$ *then the rule "tests" the value under the reading head, the head is not shifted, the symbol is not read),* $s \in Q$ *is the* start state*,* $S \in \Omega$ *is the* start symbol*,* $F \subseteq Q$ *is a* set of final states*.*

Next, an atomic pushdown automaton is defined:

Definition 2.3.2 *An* atomic pushdown automaton *is a 7-tuple,* $M = (Q, \Sigma, \Omega, R, s, \$, F)$*, where* Q *is a finite set of states,* Σ *is an input alphabet,* Ω *is a pushdown alphabet (*Q*,* Σ*, and* Ω *are pairwise disjoint),* $s \in Q$ *is the start state,* $\$$ *is the pushdown-bottom marker,* $\$ \notin Q \cup \Sigma \cup \Omega$*,* $F \subseteq Q$ *is a set of final states,* R *is a finite set of rules of the form* $Apa \rightarrow wq$*, where* $p, q \in Q$*,* $A, w \in \Omega \cup \{\varepsilon\}$*,* $a \in \Sigma \cup \{\varepsilon\}$*, such that* $|Aaw| = 1$*. That is,* R *is a finite set of rules such that each of them has one of these forms*

(1) $Ap \rightarrow q$ *(popping rule)*

(2) $p \rightarrow wq$ *(pushing rule)*

(3) $pa \rightarrow q$ *(reading rule)*

The role of determinism increases if we want to use pushdown automata in a computer application. Thus we define determinism of (atomic) PA here:

Definition 2.3.3 *An (atomic) pushdown automaton* $M = (Q, \Sigma, \Omega, R, s, \$, F)$ *is deterministic, if from* $(Apa \rightarrow wqb) \in R$ *and* $(Apa \rightarrow w'q'b') \in R$ *it follows that* $q = q' \wedge w = w' \wedge b = b'$ *(for atomic PA symbols* b *and* b' *do not appear at all).*

Informally, for a given state, the top of the pushdown, and the symbol under the reading head, there is at most one rule in R*.*

If we want to denote the sequence of operations of a pushdown automata we use sequences of configurations. Two possible definitions (differing in notation only) are presented below:

Definition 2.3.4 *A* configuration *of M is a triple $(q, w, \alpha) \in Q \times \Sigma^* \times \Omega^*$, where*

1. *q represents the current state of the finite control,*
2. *w represents the unused portion of the input; the first symbol of w is under the input head; if $w = \varepsilon$ then it is assumed that all of the input tape has been read,*
3. *α represents the contents of the pushdown list; the leftmost symbol of α is the topmost pushdown symbol; if $\alpha = \varepsilon$, then the pushdown list is assumed to be empty.*

A move *performed by M will be represented by the binary relation $\vdash_M$ (or $\vdash$ whenever identification of M is clear) on configuration. We write*

$$(q, aw, Z\alpha) \vdash (q', w, \gamma\alpha)$$

if $Zqa \to \gamma q' \in R$ for any $q \in Q$, $a \in (\Sigma \cup \{\varepsilon\})$, $w \in \Sigma^*$, $Z \in \Omega$, and $\alpha \in \Omega^*$. And, we write

$$(q, aw, Z\alpha) \vdash (q', aw, \gamma\alpha)$$

if $Zqa \to \gamma q' a \in R$.

The relation $\vdash^i$, for $i \geq 0$ can be defined in a standard customary fashion. Define $\vdash^*$ and $\vdash^+$ in the standard manner, when $\vdash^*$ stands for reflexive-transitive closure of $\vdash$ and $\vdash^+$ stands for transitive closure of $\vdash$.

Definition 2.3.5 *A* configuration *of M, χ, is alternatively any word from $\Omega^* Q \Sigma^*$. For every $x \in \Omega^*$, $y \in \Sigma^*$, and $r = Apa \to wq$, $r \in R$, M makes a move from configuration $xApay$ to configuration $xwqy$ according to $r = Apa \to wq$, $r \in R$, written as $xApay \vdash xwqy\ [r]$. If $r = Apa \to wqa$, $r \in R$ then we write $xApay \vdash xwqay\ [r]$*

Let χ be any configuration of M. M makes zero moves from χ to χ according to ε, symbolically written as $\chi \vdash^0 \chi\ [\varepsilon]$. Let there exist a sequence of configurations $\chi_0, \chi_1, \ldots, \chi_n$ for some $n \geq 1$ such that $\chi_{i-1} \vdash \chi_i\ [r_i]$, where $r_i \in R$, for $i = 1, \ldots, n$, then M makes n moves from χ_0 to χ_n, symbolically written as $\chi_0 \vdash^n \chi_n\ [r_1 \ldots r_n]$ or, more simply, $\chi_0 \vdash^n \chi_n$. Define $\vdash^*$ and $\vdash^+$ in the standard manner.

2.4 Usage of Pushdown Automata in Compiler Construction

Pushdown automata play a significant role in compiler construction. Their key task is to perform syntactic analysis based on a particular context-free grammar. Basically, we recognise two main proper subsets of context-free languages used for syntactic analysis—so called LL and LR languages. In this thesis, we mainly focus on the LL languages and thus additional definitions of automata and other necessary ones are targeted at them. To extend to both categories see, for instance, [3, 4, 5]. The abbreviation LL stands for *l*eft-to-right reading of the input and *l*eft parse.

Before we get to the definition of automata and their construction we need a helping definition:

Definition 2.4.1 *Let $L_1 +_k L_2$, where L_1 and L_2 are arbitrary languages, is defined as:*

$$L_1 +_k L_2 = \{uv \mid u \in L_1, v \in L_2, |uv| = k\}$$

A pushdown automata used for parsing of the languages, which can be described by LL grammars, can be defined this way:

Definition 2.4.2 *9-tuple $M = (Q, \Sigma, \Omega, \delta, q_0, z_0, \$, \#, Q_F)$ is a k-context grammar-based parsing pushdown automaton, if Ω stands for pushdown alphabet, Σ for tape alphabet (terminal symbols), where $\Sigma \subseteq \Omega$, Q is a set of states of automaton (Ω and Q are disjoint), q_0, $q_0 \in Q$, is a starting state, Q_F, $Q_F \subseteq Q$, is a set of final states of automaton, z_0, $z_0 \in \Omega$, is initial symbol on the top of the pushdown, # represents bottom marker of pushdown, \$ stands for end marker of input tape ($\#, \$ \notin (Q \cup \Omega)$), and δ is a mapping such that:*

$$\delta : Q \times (\Omega \cup \{\#, \varepsilon\}) \times ((\Sigma^+ +_k \{\$\}^*) \cup \{\$\}) \rightarrow Q \times \Omega^* \times \{S, \varepsilon\}$$

Every step of automata is driven by the mapping δ in such a way that according to the actual state, symbol on the top of the pushdown, and string of the k/1 symbol(s) on the input tape (starting under the reading head) the state is changed to the new one (possibly the same one), the symbol on the top of the pushdown is replaced by a string of new symbols and the reading head is (optionally) shifted (S) one symbol to the right.

For LL languages, it is not necessary to use all the possibilities of mapping δ. In practice, we use just four operations, they are:

Definition 2.4.3

1. **expand***: δ is of the form $Q \times \Omega \times (\Sigma^+ +_k \{\$\}^*) \rightarrow Q \times \Omega^* \times \{\varepsilon\}$*
   This operation replaces the top of the pushdown with a string of symbols while not moving the reading head.

2. **pop**: *δ is of the form $Q \times \Omega \times (\Sigma^+ +_k \{\$\}^*) \to Q \times \{\varepsilon\} \times \{S\}$*
   This operation removes one symbol from the top of the pushdown and moves the reading head one symbol to the right.

3. **accept**: *δ is of the form $Q \times \{\#\} \times \{\$\} \to Q_F \times \{\varepsilon\} \times \{\varepsilon\}$, where $Q_F \subseteq Q$ (see Definition 2.4.2)*
   This operation verifies that the pushdown is empty and the reading head is at the end of the tape and stops the operation of the automata.

4. **error**: *δ is of the form $Q \times (\Omega \cup \{\#\}) \times ((\Sigma^+ +_k \{\$\}^*) \cup \{\$\}) \to$* **error**
   This is a special operation and is invoked to report a syntactic error. It can be invoked in any state (possibly not the final one) and has no special relation to the content of the pushdown or tape under the reading head. Thus, reading the symbol under the reading head and watching the top of the pushdown we can recognise the faulty symbol.

Grammars that can be used to build the mapping δ and, thus, to define a deterministic parsing pushdown automata are known as LL$_k$ grammars, where k represents a width of the context. That means, how many symbols are read by a reading head of the automata at a time. Before we get to the definition of the LL$_k$ grammar, we have to introduce one definition, which will be useful even later, for construction of the automata:

Definition 2.4.4 *Let $G = (N, T, P, S)$ is a context-free grammar. $FIRST_k(\alpha)$=$\{\ a_1a_2\dots a_k \mid a_i \in T, i \in \{1,\dots,k\}, \alpha \Rightarrow^*_{|P|} a_1a_2\dots a_k\beta\ \} \cup \{\ a_1a_2\dots a_{k-1}\varepsilon \mid a_i \in T, i \in \{1,\dots,k-1\}, \alpha \Rightarrow^*_{|P|} a_1a_2\dots a_{k-1}\ \} \cup \dots \cup \{\ \varepsilon_1\varepsilon_2\dots\varepsilon_k \mid \alpha \Rightarrow^*_{|P|} \varepsilon\ \}$, where $\alpha, \beta \in (N \cup T)^*$.*

Definition of the LL$_k$ grammar follows:

Definition 2.4.5 *Let $G = (N, T, P, S)$ is a context-free grammar. It is an* LL$_k$ *grammar, where $k > 0$, if the following holds for any two derivations: if*

$$S \Rightarrow^* wA\beta \Rightarrow w\alpha_1\beta \Rightarrow^* wx \text{ and}$$
$$S \Rightarrow^* wA\beta \Rightarrow w\alpha_2\beta \Rightarrow^* wy$$

then it must also hold that from

$$FIRST_k(x) = FIRST_k(y)$$

it follows that

$$\alpha_1 = \alpha_2.$$

Informally, for a given string $w\alpha\beta$ and a sequence of tokens $a_1a_2\dots a_k$, which start the α, there is exactly one rule $A \to \alpha$ such that we could derive the complete string and, moreover, there is no other way to do that.

A definition of mapping δ is not an easy task, in general. Moreover, a suitable representation has to be chosen as well. Thus, for certain kinds of languages, we can use a simple table. Such a table describes this mapping a very useful way, which is easy for implementation. The key features are determinism of operation, creation directly from an appropriate language grammar (it is the only input and one-step operation), etc. The definition of such a parsing table, which stands for the complete definition of pushdown automata for parsing of LL$_k$ languages, can be found, for instance, in [4]. It is also presented directly below:

Definition 2.4.6 *A* parsing table *for LL$_k$ language, M', is defined on* $(\Omega \cup \{\#\}) \times ((\Sigma^+ +_k \{\$\}^*) \cup \{\$\})$. *Unlike the [4], the pushdown bottom marker is the symbol #, we are not using ε to denote the <end_of_file> symbol as usual, but we use the symbol* \$.

Every place of the parsing table contains an operation *from Definition 2.4.3 (expand, pop, accept, error).*

The automaton described by such a parsing table has then only two states—one of them is the starting state the other one is the final state. The starting one is also the only one used during parsing (we usually call it q in this thesis). On the contrary, the other one (the final one) is used only for the successful stopping of the automata (we usually call it q_F in this thesis). Thus, from the table and additional features, we can derive a complete definition of the parsing pushdown automata.

Even if a table describes the automata completely, it still remains to define an algorithm that enables the creation of such a table. The way to do that is quite easy, even if it is not straightforward. First of all, we have to start with two helping definitions. The first of them introduces a limitation to the length of the sentences for the given language and the second one introduces a new function (defined and extended in [4]):

Definition 2.4.7 *Let Σ is an alphabet then let the notation, Σ^{*k}, represents such a set of sentences over Σ, so that $w \in \Sigma^{*k} : w \in \Sigma^* \wedge |w| \leq k$.*

Definition 2.4.8 *Let $G = (N, \Sigma, P, S)$ be a context-free grammar (2.2.7). For each $A \in N$ and $L \subseteq \Sigma^{*k}$ we define $B_{A,L}$, the* LL$_k$ table associated with A and L *to be a function which given a lookahead string $u \in \Sigma^{*k}$ returns either the symbol* **error** *or an A-production and a finite list of subsets of Σ^{*k}.*

Specifically,

1. $B_{A,L}$ = **error** *if there is no production $A \rightarrow \alpha$ in P such that $FIRST_k(\alpha) +_k L$ contains u.*
2. $B_{A,L} = (A \rightarrow \alpha, < Y_1, Y_2, \ldots, Y_m >)$ *if $A \rightarrow \alpha$ is the unique production in P such that $FIRST_k(\alpha) +_k L$ contains u. If* $\alpha =$

$x_0X_1x_1X_2x_2 \dots X_mx_m$, $m \geq 0$, where each $X_i \in N$ and $x_i \in \Sigma^*$, then $Y_i = FIRST_k(x_iX_{i+1}x_{i+1} \dots X_mx_m) +_k L$. We shall call Y_i a local follow set *for X_i.*

3. *$B_{A,L}$ is undefined if there are two or more productions $A \rightarrow \alpha_1|\alpha_2|\dots|\alpha_n$ such that $FIRST_k(\alpha_i) +_k L$ contains u, for $1 \leq i \leq n$, $n \geq 2$. This situation will not occur if G is an LL_k grammar, though.*

These two definitions help us in the construction of LL$_k$ tables, which is a necessary input to the construction of a parsing table, besides an appropriate grammar, of course. An algorithm of LL$_k$ tables construction can also be found in [4]:

Algorithm 2.4.1 *The algorithm of LL_k tables construction is defined in such a way:*
Input*: An LL_k context-free grammar, $G = (N, \Sigma, P, S)$*
Output*: $\Im$, the set of LL_k tables needed to construct a parsing table for G.*
Method*:*

1. *Construct B_0, the LL_k table associated with S and \$.*
2. *Initially set $\Im = \{B_0\}$.*
3. *For each LL_k table $B \in \Im$ with entry*

$$B(u) = (A \rightarrow x_0X_1x_1X_2x_2 \dots X_mx_m, < Y_1, Y_2, \dots, Y_m >)$$

add to $\Im$ the LL_k table B_{X_i,Y_i}, for $1 \leq i \leq m$, if B_{X_i,Y_i} is not already in $\Im$.

4. *Repeat step (3) until no new LL_k tables can be added to $\Im$.*

Finally, we have all the necessary inputs to construct an LL$_k$ parsing table for a given grammar. The algorithm comes from [4] again:

Algorithm 2.4.2
Input*: An LL_k context-free grammar $G = (N, \Sigma, P, S)$ and $\Im$, the set of LL_k tables for G (the algorithm for their construction can be found in Algorithm 2.4.1, or in [4]).*
Output*: M', a valid parsing table for G.*
Method*: From Definition 2.4.6, set $\Omega = \Im \cup \Sigma$. The content of M' is defined as follows:*

1. *If $A \rightarrow x_0X_1x_1X_2x_2 \dots X_mx_m$ is the* i*th production in P and $B_{A,L}$ is in $\Im$, then for all u such that*

$$B_{A,L}(u) = (A \rightarrow x_0X_1x_1X_2x_2 \dots X_mx_m, < Y_1, Y_2, \dots, Y_m >)$$

we have $M'(B_{A,L}, u) = (x_0B_{X_1,Y_1}x_1B_{X_2,Y_2}x_2 \dots B_{X_m,Y_m}x_m, i)$—operation **expand**.

2. *$M'(a, av) =$ **pop** for all $v \in (\Sigma^{*(k-1)} +_{(k-1)} \{\$\}^{*(k-1)})$.*
3. *$M'(\#, \$) =$ **accept**.*
4. *Otherwise, $M'(X, u) =$ **error**.*
5. *$B_{S,\{\$\}}$ is the initial table.*

We will show the complete table creation in the following example. It uses quite a simple grammar defining a finite language, nevertheless, itself having an LL$_2$ features:
Let $G = (N, \Sigma, P, S)$ be a grammar having such features:

$$N = \{S, A\}$$

$$\Sigma = \{a, b\}$$

Note: *We used T to denote this set, which is traditional for language theory, Σ is used in connection with automata.*

$$P = \{$$

$$S \rightarrow aAaa \mid bAba$$

$$A \rightarrow b \mid \varepsilon$$

$$\}$$

When building the parsing table, we have to start with the LL$_k$ tables, namely with table $B_0 = B_{S,\{\$\}}$:

Table B_0

u	Production	Sets
aa	$S \rightarrow aAaa$	$< \{aa\} >$
ab	$S \rightarrow aAaa$	$< \{aa\} >$
bb	$S \rightarrow bAba$	$< \{ba\} >$

From this table, another two tables can be derived:

Table $B_{A,\{aa\}}$

u	Production	Sets
ba	$A \rightarrow b$	—
aa	$A \rightarrow \varepsilon$	—

Table $B_{A,\{ba\}}$

u	Production	Sets
ba	$A \rightarrow \varepsilon$	—
bb	$A \rightarrow b$	—

Construction of the LL$_2$ parsing table is started with enumeration of the grammar rules:

1. $S \rightarrow aAaa$
2. $S \rightarrow bAba$
3. $A \rightarrow b$
4. $A \rightarrow \varepsilon$

	aa	ab	$a\$$	ba	bb	$b\$$	$\$$
B_0	aB_1aa, 1	aB_1aa, 1			bB_2ba, 2		
B_1	ε, 4			b, 3			
B_2				ε, 4	b, 3		
a	**pop**	**pop**	**pop**				
b				**pop**	**pop**	**pop**	
$\#$							**accept**

Figure 2.1: Parsing table for LL_2 language

Finally, the parsing table (blank entries indicate **error**) is presented in figure 2.1. For the sake of better readability, we rename the parsing tables the following way: $B_0 = B_{S,\{\$\}}$, $B_1 = B_{A,\{aa\}}$, and $B_2 = B_{A,\{ba\}}$.

Now, if we have an input string bba the pushdown automaton defined by the table would make the following sequence of moves:

$$\begin{array}{rll} (bba\$, B_0\#, \varepsilon) & \vdash & (bba\$, bB_2ba\#, 2) \\ & \vdash & (ba\$, B_2ba\#, 2) \\ & \vdash & (ba\$, ba\#, 24) \\ & \vdash & (a\$, a\#, 24) \\ & \vdash & (\$, \#, 24). \end{array}$$

Chapter 3

Analysis of LL$_k$ Languages

The LL grammars play an important role in programming languages description. The construction of their efficient and simple analysers (pushdown automata) is limited to the LL$_1$ grammars, however. The descriptive power of these grammars is quite low and, in addition, there are problems with analysis of the LL$_{k+1}$, $k \geq 1$, grammars. This section presents an algorithm that allows transformation from pushdown automaton with (k+1)-symbol reading head used for LL$_{k+1}$ language analysis to the one-symbol reading head pushdown automaton. Thus, we can simulate a function of the former by using the much simpler constructs of the latter.

3.1 Introduction

The context-free grammars contain some proper subsets for which an efficient analyser (parser) can be built. LL grammars belong to these subsets. Languages such as Pascal, Modula, and Oberon were described by using context-free grammars satisfying conditions of being LL. In particular, LL$_1$ grammars were used. It was proven (see [5]) that LL$_k$ grammar has a lower descriptive power than LL$_{k+1}$ grammars for any $k > 0$. Moreover, a description of systems using LL grammars with higher k can lead to much readable and understandable notation. Unfortunately, the analysis of languages described by such grammars is not that easy task, because using sequences of symbols for indexing or matching operations requires a special approach and makes an implementation more difficult.

Pushdown automata used for the analysis of LL$_{k+1}$ languages are constructed from the context-free grammars describing the particular language using parsing tables. The algorithm for the parsing table construction can be found in [4] and is briefly presented in Chapter 2. The main problem of analysis of LL$_{k+1}$ languages lies in the comparison of several symbols under the pushdown automaton reading head (in file) with the same number of symbols in the parsing table and/or on the top of the pushdown.

The algorithm presented below enables the simulation of the automata with several symbols under the reading head by automata using just one symbol under the reading head. Moreover, as we will see, the transition from one automaton/parsing table to another can be done very easily when the process of transformation is fully understood.

The background of parsing table construction is presented in [4] or in Chapter 2. The algorithm of new automata construction is presented in several steps below. Finally, the two parsing tables are compared so that the easy transition from one to another can be seen and the section itself is concluded afterwards.

3.2 One-Symbol Automata Construction

We have already presented the main reasons as to why multi-symbol parsers are not suitable—especially because searching for actions in a table based on more than 1 symbol is not easily implementable.

The idea of simulating automata with several symbols by automata with just one symbol lies in the storing of a sufficiently recent history of actually processed symbols on the tape into states of the pushdown automata. Thus, we will need just one symbol to decide on the next step, while fully simulating all the features of the original automata.

3.2.1 Parsing Automata and Table Modification

An algorithm for the creation of a new pushdown automata, generally multi-state ones, starts with the formal modification of the original parsing table and automaton.

Definition 3.2.1 *A* modified parsing pushdown automaton *is the one defined in Definition 2.4.2 with the exception that mapping δ is defined the following way:*

$$\delta : Q \times (\Omega \cup \{\#, \varepsilon\}) \times (\Sigma^* +_k \{\$\}^*) \to Q \times \Omega^* \times \{S, \varepsilon\}$$

Definition 3.2.2 *Definition of LL_k parsing table from 2.4.6 is modified in accordance with the automata definition. The* modified parsing table *is defined over*

$$(\Omega \cup \{\#\}) \times (\Sigma^* +_k \{\$\}^*)$$

The rest remains untouched.

Table 2.1, from example at the end of the Chapter 2, has to be modified in such a way then (see figure 3.1). We can see that, the former column names were formally changed, so that they could always have the same number, k, of

	aa	ab	$a\$$	ba	bb	$b\$$	$\$\$$
B_0	aB_1aa, 1	aB_1aa, 1			bB_2ba, 2		
B_1	ε, 4			b, 3			
B_2				ε, 4	b, 3		
a	**pop**	**pop**	**pop**				
b				**pop**	**pop**	**pop**	
$\#$							**accept**

Figure 3.1: Modified parsing table for LL_2 language

symbols, in our example $k = 2$. It is done by the appending of the appropriate number of symbols representing the end-of-file symbol (\$).

3.2.2 Empty Automaton Construction

The automaton reading a single symbol under the reading head while analysing LL_k language for $k > 1$ is just a straightforward modification of the automaton with k-context reading head:

Definition 3.2.3 *A single symbol k-context grammar-based parsing pushdown automaton (SSkPDA), M, follows Definition 2.4.2 except formal definition of mapping δ, which is:*

$$\delta : Q \times (\Omega \cup \{\#, \varepsilon\}) \times (\Sigma \cup \{\$\}) \rightarrow Q \times \Omega^* \times \{S, \varepsilon\}$$

Pushdown and tape alphabet remains the same for the new automata, of course. The states will be defined by their names. They are all column names from the modified parsing table, plus all the prefixes of those not starting with symbol \$. In the case of our example, they are the following states: aa, ab, $a\$$, ba, bb, $b\$$, $\$\$$, a, b. For the set to be complete, starting (0) and final (X) states have to be added. Names of the states are telling us what the recent history is, what symbols were already read and skipped by the reading head. Formally:

Definition 3.2.4 *Set of states, Q, of SSkPDA is derived in such a way:*

$$\begin{array}{rcl} Q & = & \{\ \bar{x} \mid x \in (\Sigma^* +_k \{\$\}^*)\ \} \cup \\ & & \{\ \bar{y} \mid x \in (\Sigma^+ +_k \{\$\}^*), y \in \mathit{prefix}(x)\ \} \cup \\ & & \{0, X\},\ \ 0, X \notin \Sigma \end{array}$$

Note: if x is a string $a_1a_2 \ldots a_n$ then $\bar{x}$ stands for $\bar{a}_1\bar{a}_2 \ldots \bar{a}_n$.

Moreover, the actions of new automata have to be added/modified to be able to handle a new feature. The actions *expand*, *accept*, and *error* remain informally the same, with the small exceptions described below. The action *pop* is modified though—it removes the correct symbol from the pushdown not taking into account a symbol under the reading head. Moreover, as another parameter it has a name of a new state of automata which will be active after this action is performed. A completely new action is the action *read*, which has as a parameter symbol, which should be on the tape under the reading head and it is moved to the top of the pushdown, while the reading head is moved one symbol to the right. Formally:

Definition 3.2.5 *The operations of SSkPDA are the following:* **expand***,* **read***,* **pop***,* **accept***,* **error**. *They are formally defined the following way:*

- **expand***:* δ *is of the form* $Q \times \Omega \times \{\varepsilon\} \rightarrow Q \times \Omega^* \times \{\varepsilon\}$
  We can see that the history stored in the states influences this operation in such a way so that it need not test the symbol under the reading head. It is denoted by the state itself.

- **read***:* δ *is of the form* $Q \times \{\varepsilon\} \times \Sigma \rightarrow Q \times \{\varepsilon\} \times \{S\}$
  According to the state and symbol under the reading head the new state is denoted and the reading head shifted one symbol to the right.

- **pop***:* δ *is of the form* $Q \times \Omega \times \{\varepsilon\} \rightarrow Q \times \{\varepsilon\} \times \{\varepsilon\}$
  The correctness of the pushdown top content is "checked" by the state.

- **accept***:* δ *is of the form* $Q \times \{\#\} \times \{\$\} \rightarrow Q_F \times \{\varepsilon\} \times \{\varepsilon\}$

- **error***:* δ *is of the form* $Q \times (\Omega \cup \{\#\}) \times (\Sigma \cup \{\$, \varepsilon\}) \rightarrow$ **error**

Thus, we have all the basic elements for the new automata creation ready. So far, actions connecting the states already defined are the only missing thing, but the most important one.

3.2.3 Automaton Completion

The connection of states with actions starts with the read actions. From the modified parsing table, we can see that symbols a and b can be read at the beginning, but not symbol \$. Thus, we can perform actions reading symbols a and b and changing the state from the starting one to the appropriate one (state a or b). Moreover, for any combination of 2 symbols, there is a possible action. Thus the appropriate read actions should continue even to relevant states (see solid arrows in figure 3.2). For instance, when in state a a symbol b is read from the tape and the reading head is moved one symbol to the right then the state is changed from state a, which says that recently symbol a was read, to state ab, which says that symbols ab were recently read, symbol a directly before symbol b. Formally:

Definition 3.2.6 *The SSkPDA mapping δ must, besides others (other definitions completing the mapping δ will follow), contain the following maps (***read** *operations):*

1. $(0, \varepsilon, a) \rightarrow (p, \varepsilon, \{S\})$ *if in the modified parsing table, there is for table B_0 an expand action in a column, which is marked with a string starting with the symbol a, $a \in \Sigma$, and the name of state p is composed from the single symbol string $\bar{a}$,*
2. $(q, \varepsilon, a) \rightarrow (p, \varepsilon, \{S\})$ *if a name of state q is composed from string α and a name of state p is composed from string $\alpha\bar{a}$, $a \in \Sigma$, moreover, if $|\alpha\bar{a}| < k$ then for symbol x, $\bar{x} = sym(\alpha, 1)$, $x \in \Sigma$, an action must be defined in point 1,*
3. $(0, \varepsilon, \$) \rightarrow (p, \varepsilon, \varepsilon)$ *if in the modified parsing table, there is for table B_0 an expand action in a column, which is marked with a string $\$^k$, and the name of state p is composed from string $\bar{\$}^k$—this happens if an empty string belongs to the language.*

The symbol on the top of the pushdown can be removed whenever the history stored in the state and symbol on the top of the pushdown matches (see dashed lines with boxed labels in figure 3.2). For instance, being in state *aa* and having symbol *a* on the top of the pushdown, we can remove the symbol from the pushdown top, but we have to change the state from the one called *aa* to the one called *a*. Thus, we keep coherence of preservation of history of processed symbols with the state activity. Formally:

Definition 3.2.7 *The SSkPDA mapping δ must, besides others, contain the following maps:*

1. $(q, a, \varepsilon) \rightarrow (p, \varepsilon, \varepsilon)$ *if a name of state q is composed from string $x\alpha$, $x = \bar{a}$, $a \in (\Sigma \cap \Omega)$, and a name of state p is composed from string α, $\alpha \neq \bar{\$}^{(k-1)}$, if $\bar{y} \in alph(\alpha)$ then $y \in (\Sigma \cup \{\$\})$.*
2. $(q, a, \varepsilon) \rightarrow (p, \varepsilon, \varepsilon)$ *if a name of state q is composed from string $x\bar{\$}^{k-1}$, $x = \bar{a}$, $a \in (\Sigma \cap \Omega)$, and a name of state p is composed from string $\bar{\$}^k$.*

These can be seen as operations **pop** *defined in such a way, so that symbol a on the top of the pushdown and actual state q denotes performance of the operation itself, which has as a parameter state p that is a new state after completion of the operation.*

Table symbols (replacing nonterminals) stored on the top of the pushdown must be expanded at the correct time. The information for when to do that is stored in the modified parsing table, of course. Now, instead of tape content, we just work with automata states. Thus, whenever a state for name of which a

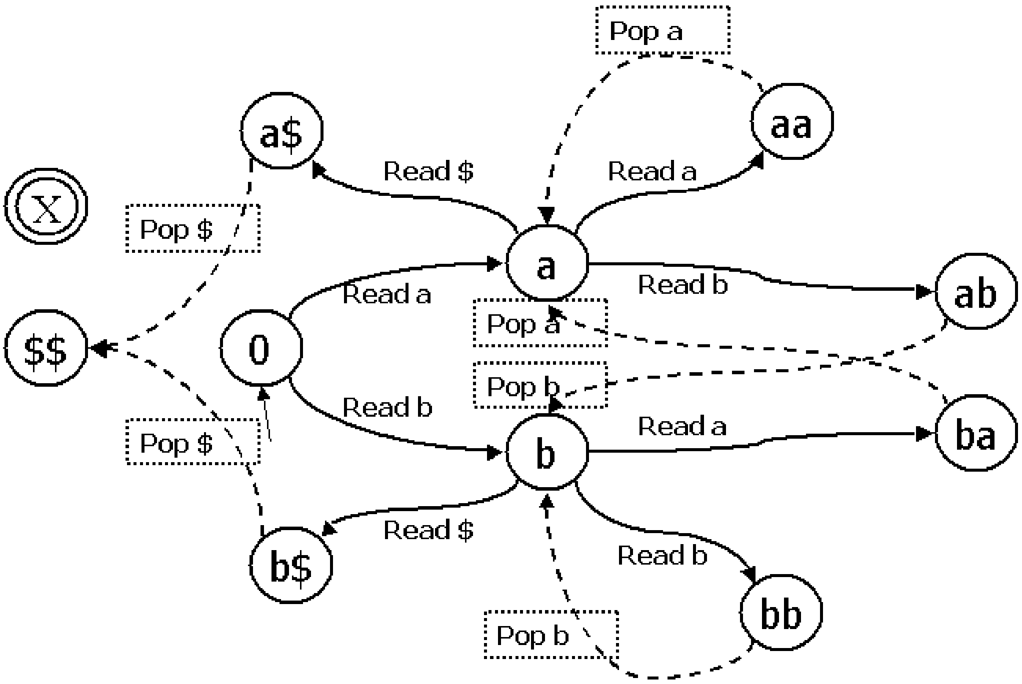

Figure 3.2: Read and Pop actions are easy to define

column of the modified parsing table contains expand action appears, the same action is added to the new automaton. The tape contents is represented by state, the top of the pushdown must match the symbol being expanded by the action (see solid arrows with labels boxed in solid boxes in figure 3.3—the figure uses numbers to denote the proper actions, they are: 1 for rule ($B_0 \to aB_1aa$, 1), 2 for rule ($B_0 \to bB_2ba$, 2), 3 for rule ($B_1 \to b$, 3), 4 for rule ($B_1 \to \varepsilon$, 4), 5 for rule ($B_2 \to b$, 3), 6 for rule ($B_2 \to \varepsilon$, 4)). Formally:

Definition 3.2.8 *The SSkPDA mapping δ must, besides others, contain the following maps (***expand*** operations): $(q, B_i, \varepsilon) \to (q, \gamma, \varepsilon)$ if a name of state q is composed from string $\bar{\alpha}$ and the modified parsing table contains for table B_i and string under the reading head, α, operation expand, which replaces the top of the pushdown with string γ.*

Note: besides expansion, the number of used grammar rule is sent to the output.

Finally, the accept action must be added, so that the automata can stop its operation. This can be done only in one case when there is no symbol on the pushdown and the tape is read till the end (see dashed arrow with label boxed in dashed box in figure 3.3). Formally:

Definition 3.2.9 *The SSkPDA mapping δ must, besides others, contain the following map (***accept*** operation): $(q, \#, \$) \to (X, \varepsilon, \varepsilon)$ if a name of state q is*

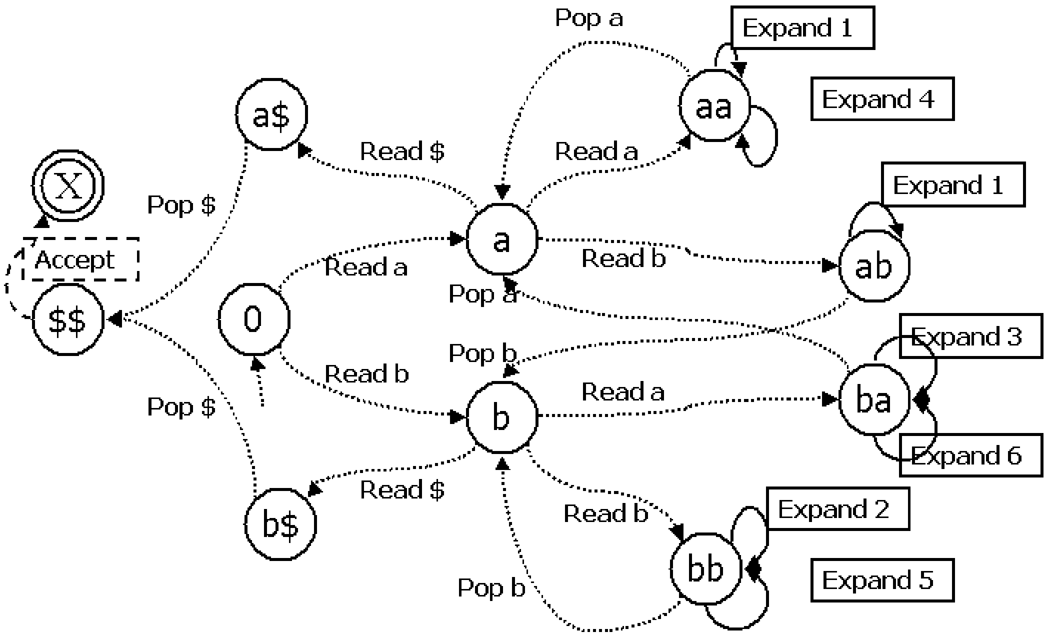

Figure 3.3: Expand and Pop actions added

composed from string $\bar{\$}^k$.

If the automaton is in a state, for which there is no action defined for the actual content of the pushdown top and/or symbol under the reading head the error occurs. Formally:

Definition 3.2.10 *If Definitions 3.2.6, 3.2.7, 3.2.8, 3.2.9 do not define the mapping* δ *completely the remaining entries are filled with action* error*.*

Note: Not for all combinations is it necessary, see next subsection.

3.2.4 Parsing Table Notation

The pushdown automata creation was described above (without the formal approach presented in [49]). The representation by a labelled state transition diagram is not useful, though. Moreover, even if the transformation algorithm is not complicated it is not straightforward.

Nevertheless, a table representation for the new automata can be found. Moreover, the representation can even have such features so that the new automata can be derived very easily and straightforwardly.

The columns of the new table will be denoted by automata states. The rows will be divided into two groups. In the first one, lines will be denoted by pushdown alphabet (the same way as in the case of the regular parsing table). The second part will use the tape alphabet. Moreover, the second lower part will not need columns, where state names are based on k-symbol strings.

Even the upper part of the table, where lines are denoted by pushdown alphabet, can be logically separated into two parts. Moreover, the same way as the lower part—one part is denoted by k-symbol names, while the other not. And what is even more, one part, in this case, the one denoted by columns where state names do not contain k symbols, is also always empty.

As there is not any action outgoing from the final state, it is not necessary to explicitly mention this state in the table.

Thus, we can see that the table is divided into four parts, while two of them are always empty:

	states with fewer than k-symbol names	*states with exactly k-symbol names*
pushdown alphabet	always empty	actions of original table, uses new actions semantics
tape alphabet	always action read, new states stored	always empty

Empty parts of the table represent cases that are not taken into account as there should be action in another part of the table. If there is no action in the non-empty part, an error should be reported.

3.3 Comparison of Parsing Tables

The new parsing table for the example presented would look like this:

	0	$\bar{a}$	$\bar{b}$	$\bar{a}\bar{a}$	$\bar{a}\bar{b}$	$\bar{a}\bar{\$}$	$\bar{b}\bar{a}$	$\bar{b}\bar{b}$	$\bar{b}\bar{\$}$	$\bar{\$}\bar{\$}$
B_0				**aB_1aa, 1**	**aB_1aa, 1**			**bB_2ba, 2**		
B_1				**ε, 4**			**b, 3**			
B_2							**ε, 4**	**b, 3**		
a				**pop** $\bar{a}$	**pop** $\bar{b}$	**pop** $\bar{\$}$				
b							**pop** $\bar{a}$	**pop** $\bar{b}$	**pop** $\bar{\$}$	
#										**acc**
a	$\bar{a}$	$\bar{a}\bar{a}$	$\bar{b}\bar{a}$							
b	$\bar{b}$	$\bar{a}\bar{b}$	$\bar{b}\bar{b}$							
$\$$		$\bar{a}\bar{\$}$	$\bar{b}\bar{\$}$							

The part, which recollects the modified original parsing table is presented in bold. If we take into account that columns are not indexed by the contents of the tape, but by states and that the action pop has a modified behaviour, we can see that the only difference is the left lower corner. Filling it is quite straightforward, though.

3.4 Intermediate Summary

To summarise, this chapter presents, so far, an algorithm that enables simulating the behaviour of pushdown automata used for the analysis of LL languages with wider context (2 and more symbols on the tape are read at a single time). Moreover, the representation of such a new automata is possible by table too. What is even more important, the new representation is very similar to the old one and a formal change of semantics of various parts of the table has to be done. The only extension that is required is quite straightforward.

Storage of such a table in the program memory and implementation of an analyser based on this table seems to be quite easy as well. This is because we talk about two two-dimensional arrays only. Moreover, it is sufficient to read only one symbol from the tape at a time.

The algorithm enables using, in a much broader way, the LL_k grammars and languages based on these grammars in various parts of computer science. As an open item, we can see a construction of automatic parser generator similar to the particularly known one—y.a.c.c. Another open item is the proof of equivalence of both automata. It is presented next.

3.5 Proof of Automata Equivalence

The proof shows the step by step equality of the LL_k parsing table driven pushdown automata and SSkPDA pushdown automata. The proof uses mathematical induction with extra small proofs to demonstrate all possibilities.

0) *Empty string analysis*

First of all, let us assume, that the empty string, ε, belongs to the language.

We start with the LL_k parsing automata. If Algorithm 2.4.2 is followed the automata would go through the following sequence of configurations: $(q, \$, B_0\#)$ $\vdash (q, \$, \#) \vdash (q_F, \$, \#)$. The first configuration, $(q, \$, B_0\#)$, is the starting one and follows from the definition. Transition to the next configuration follows the operation **expand** coming from the Algorithm 2.4.2—it replaces the symbol B_0 with the empty string. The next transition is performance of the operation **accept**, it also comes from Algorithm 2.4.2. The last configuration contains the final state, q_F, and represents successful acceptance of the input.

The SSkPDA automata would behave the following way in the same situation. The starting configuration, coming from the definition is the following one: $(0, \$, B_0\#)$. From the starting configuration, there is, for this case, defined an operation **read** coming from Definition 3.2.6. Applying the operation the configuration changes in such a way: $(0, \$, B_0\#) \vdash (\bar{\$}^k, \$, B_0\#)$. For a given configuration, there is defined operation **expand** coming from Definition 3.2.8.

Applying this operation the configuration changes in such a way: $(\bar{\$}^k, \$, B_0\#) \vdash (\bar{\$}^k, \$, \#)$. This operation performs replacement of the symbol B_0 with the empty string. Finally, for the last configuration, there is defined operation **accept** from Definition 3.2.9. The last transition $(\bar{\$}^k, \$, \#) \vdash (X, \varepsilon, \varepsilon)$ reaches the configuration containing the final state, which represents successful acceptance of the input.

Now, let us assume, that the empty string, ε, does not belong to the language.

We start with the LL_k parsing automata again. If Algorithm 2.4.2 is followed the automata would behave the following way: $(q, \$, B_0\#) \vdash$ **error**. This is because there is no operation **expand** defined for such a configuration and any other operation (except the error one) is not even used in such configuration at all.

The SSkPDA automata would behave the following way in the same situation. The starting configuration is obvious: $(0, \$, B_0\#)$. As the empty string is not in the language assumed in this case, there is no transition defined by operations defined in 3.2.6 and all the other kinds of operations (except the error one) are not used in the situation. Thus, the transition performs an error detecting/reporting operation: $(0, \$, B_0\#) \vdash$ **error**.

We can see that both automata behave the same way in the presented situations.

1) *One-symbol string analysis* (step 1 of the induction part of the proof)

First of all, let us assume, that the one-symbol string, a, belongs to the language.

We start with the LL_k parsing automata. If Algorithm 2.4.2 is followed the automata would go through the following sequence of configurations: $(q, a\$, B_0\#) \vdash (q, a\$, a\#) \vdash (q, \$, \#) \vdash (q_F, \$, \#)$. The first configuration, $(q, a\$, B_0\#)$, is the starting one and follows from the definition. Transition to the next configuration follows operation **expand** coming from Algorithm 2.4.2—it replaces the symbol B_0 with the one-symbol string a. The next transition stands for operation **pop**, which is also defined by 2.4.2. The last transition is the performance of the operation **accept**, it also comes from Algorithm 2.4.2. The last configuration contains the final state, q_F, and represents successful acceptance of the input.

The SSkPDA automata would behave the following way in the same situation. The starting configuration, coming from the definition is the following one: $(0, a\$, B_0\#)$. From the starting configuration, there is, for this case, defined an operation **read** coming from Definition 3.2.6. Applying the operation the configuration changes in such a way: $(0, a\$, B_0\#) \vdash (\bar{a}, \$, B_0\#)$. Now, the operation **read**, defined in 3.2.6, continues reading till k symbols have been read in total (repeatedly reads end_of_file marker—\$, in total $k - 1$ times). Thus, the next transition looks like $(\bar{a}, \$, B_0\#) \vdash (\bar{a}\bar{\$}, \$, B_0\#)$. Reading the \$ symbol repeats till the following configuration is reached: $(\bar{a}\bar{\$}^{(k-1)}, \$, B_0\#)$. For a given configuration, there is defined operation **expand** coming from Definition 3.2.8. Applying this operation the configuration changes in such a way: $(\bar{a}\bar{\$}^{(k-1)}, \$, B_0\#) \vdash (\bar{a}\bar{\$}^{(k-1)}, \$, a\#)$. This operation performs replacement of the symbol B_0 with the one-symbol string a. Now, the only operation that can be performed is the **pop** operation defined in 3.2.7. It changes the configuration in the following way: $(\bar{a}\bar{\$}^{(k-1)}, \$, a\#) \vdash (\bar{\$}^k, \$, \#)$ Finally, for the last configuration, there is defined operation **accept** from Definition 3.2.9. The last transition $(\bar{\$}^k, \$, \#) \vdash (X, \varepsilon, \varepsilon)$ reaches the configuration containing the final state, which represents successful acceptance of the input.

Now, let us assume, that the one-symbol string, a, does not belong to the language.

We start with the LL$_k$ parsing automata again. If Algorithm 2.4.2 is followed the automata would behave the following way: $(q, a\$, B_0\#) \vdash$ **error**. This is because there is no operation **expand** defined for such a configuration and any other operation (except the error one) is not even used in such a configuration at all.

The SSkPDA automata can behave in two ways with the same result depending on the language features. If the language does not allow symbol a at the beginning of any string at all, the SSkPDA would behave this way. The starting configuration is obvious: $(0, a\$, B_0\#)$. As the empty string is not assumed in the language at the beginning of any string in this case, there is no transition defined by the operations defined in 3.2.6 and all the other kinds of operations (except the error one) are not used in the situation. Thus, the transition performs an error detecting/reporting operation: $(0, \$, B_0\#) \vdash$ **error**.

If symbol a is allowed at the beginning of certain strings, but not alone, the automaton behaviour would start the same way as if the one-symbol string is part of the language: $(0, a\$, B_0\#) \vdash (\bar{a}, \$, B_0\#) \vdash (\bar{a}\bar{\$}, \$, B_0\#) \vdash \ldots \vdash (\bar{a}\bar{\$}^{(k-1)}, \$, B_0\#)$. It simply reads the first k symbols and reaches the appropriate state. Nevertheless, for a given state there is no operation **expand** defined and no **pop** operation can be performed. The other operations, except the error one, are not defined in such situations at all. Thus the **error** operation has to be performed: $(\bar{a}\bar{\$}^{(k-1)}, \$, B_0\#) \vdash$ **error**.

We can see that both automata behave the same way in the presented situations.

2) $(n+1)$-*symbol string analysis* (step 2 of the induction part of the proof)

First of all, let us assume, that the $(n+1)$-symbol string, $a\alpha$, belongs to the language. From induction hypothesis, we assume that a string of the length n, α, is processed correctly.

We start with the LL_k parsing automata. If Algorithm 2.4.2 is followed the automata would start with the following sequence of configurations: $(q, a\alpha\$, B_0\#) \vdash (q, a\alpha\$, \beta\#)$. The first configuration, $(q, a\alpha\$, B_0\#)$, is the starting one and follows from the definition. Transition to the next configuration follows operation **expand** coming from Algorithm 2.4.2—it replaces the symbol B_0 with a string of symbols, β, according to the k symbols, starting with the symbol a, under the reading head of the automata. For the next transition, there are two possibilities, in general.

1. String $\beta = Z\gamma$, where $Z \notin (\Sigma \cap \Omega)$, i.e. it is a *table* symbol. In such a case, the **expand** operation is performed till the configuration $(q, a\alpha\$, b\omega\#)$, where $b \in (\Sigma \cap \Omega)$, is reached. The situation following is described under point 2.
2. String $\beta = b\gamma$, where $b \in (\Sigma \cap \Omega), b = a$. In such a case, the operation **pop** is performed: $(q, a\alpha\$, b\gamma\#) \vdash (q, \alpha\$, \gamma\#)$. From now on, the situation described by either point 1 or point 2 can occur till the configuration gets to the following status: $(q, \$, \#)$—we rely on induction hypothesis here, as by performing the operation pop, string α is to be accepted, which is the base of the induction hypothesis.

If the automata reach the status $(q, \$, \#)$ then the last transition is performed—the operation **accept**, it also comes from Algorithm 2.4.2. The last configuration contains the final state, q_F, and it represents successful acceptance of the input: $(q, \$, \#) \vdash (q_F, \$, \#)$.

The SSkPDA automata would behave the following way in the same situation. The starting configuration, coming from the definition is the following one: $(0, a\alpha\$, B_0\#)$. From the starting configuration, there is, for this case, defined an operation **read** coming from Definition 3.2.6. Applying the operation, the configuration changes in such a way: $(0, a\alpha\$, B_0\#) \vdash (\bar{a}, \alpha\$, B_0\#)$. Now, the operation **read** defined in 3.2.6 continues reading till k symbols have been read in total. Thus, the next $(k-1)$ transitions look like $(\bar{a}, \alpha\$, B_0\#) \vdash \ldots \vdash (\bar{a}\bar{\sigma}, \delta\$, B_0\#)$, where $\sigma\delta = \alpha$ if $|\alpha| \geq (k-1)$, or $\sigma\delta = \alpha\$^i$, where $i = k - |\alpha| - 1$ if $|\alpha| < (k-1)$. For a given configuration, there is defined operation **expand** coming from Definition 3.2.8. Applying this operation, the configuration changes in such a way: $(\bar{a}\bar{\sigma}, \delta\$, B_0\#) \vdash (\bar{a}\bar{\sigma}, \delta\$, \beta\#)$. This operation performs the replacement of the symbol B_0 with the string β. For the next transition, there are two possibilities, in general.

1. String $\beta = Z\gamma$, where $Z \notin (\Sigma \cap \Omega)$, i.e. it is a *table* symbol. In such a case, the **expand** operation (def. 3.2.8) is performed till the configuration $(\bar{a}\bar{\sigma}, \delta\$, b\omega\#)$, where $b \in (\Sigma \cap \Omega)$, is reached. The situation following is described under point 2.
2. String $\beta = b\gamma$, where $b \in (\Sigma \cap \Omega), b = a$. In such a case, the operation **pop** (def. 3.2.7) is performed: $(\bar{a}\bar{\sigma}, \delta\$, b\gamma\#) \vdash (\bar{\sigma}, \delta\$, \gamma\#)$. From now on,

the situation described by either point 1 or point 2 can occur till the configuration gets to the following status: $(\bar{\$}^k, \$, \#)$—induction hypothesis.

If the automata reach the status $(\bar{\$}^k, \$, \#)$ then the last transition is performed—the operation **accept** from Definition 3.2.9. The last transition $(\bar{\$}^k, \$, \#) \vdash (X, \varepsilon, \varepsilon)$ reaches the configuration containing the final state, which represents successful acceptance of the input.

Now, let us assume, that the $(n+1)$-symbol string, $a\alpha$, does not belong to the language.

The string may be rejected under two circumstances only:

1. the symbol on the top of the pushdown is a *table* symbol and it cannot be *expanded*, or
2. the symbol on the top of the pushdown is a *terminal/input alphabet* symbol and it cannot be *popped*.

The former one has already been shown for the one-symbol input string. The behaviour could be the same even for longer strings with the exception that it may happen even later. Thus, the detailed proof is left to the reader.

The latter one will be partially demonstrated below. The detailed part of the proof is also left up to the reader.

We start with the LL_k parsing automata, as usual. The error may happen if we get into the following configuration: $(q, a_1 \ldots a_k \alpha b \beta \$, B_i \gamma \#)$. If the expansion is performed in such a way, so that we get configuration $(q, a_1 \ldots a_k \alpha b \beta \$, \delta \gamma \#)$, then for $\delta = a_1 \ldots a_k \alpha c$ we can perform a sequence of **pop** operations (it is performed $k + |\alpha|$ operations): $(q, a_1 \ldots a_k \alpha b \beta \$, a_1 \ldots a_k \alpha c \gamma \#) \vdash \ldots \vdash (q, b\beta\$, c\gamma\#)$. Now, as we assume an error, $b \neq c$, $b, c \in (\Sigma \cap \Omega)$ and, thus, an error operation is performed: $(q, b\beta\$, c\gamma\#) \vdash$ **error**.

The SSkPDA automata would behave the following way in the same situation. It is in the same situation as the LL_k automata above if it is in the configuration: $(\bar{a}_1 \bar{a}_2 \ldots \bar{a}_k, \alpha b \beta \$, B_i \gamma \#)$. If the expansion is performed in such a way, so that we get configuration $(\bar{a}_1 \bar{a}_2 \ldots \bar{a}_k, \alpha b \beta \$, \delta \gamma \#)$, then for $\delta = a_1 \ldots a_k \alpha c$ we can perform a sequence of **pop** and **read** operations (it is performed $k + |\alpha|$ pairs of operations): $(\bar{a}_1 \bar{a}_2 \ldots \bar{a}_k, \alpha b \beta \$, a_1 \ldots a_k \alpha c \gamma \#) \vdash (\bar{a}_2 \ldots \bar{a}_k, \alpha b \beta \$, a_2 \ldots a_k \alpha c \gamma \#) \vdash (\bar{a}_2 \ldots \bar{a}_k \bar{x}, \zeta b \beta \$, a_2 \ldots a_k \alpha c \gamma \#)_{|x\zeta = \alpha|} \vdash \ldots \vdash (\bar{\chi}, \beta' \$, c\gamma\#)$. Now, as we assume an error, $\chi = b\chi'$ and $b \neq c$, $b, c \in (\Sigma \cap \Omega)$, from which it follows that the error operation is performed as there is no expansion possible and no **pop** operation can be done for different symbols, thus: $(\bar{b}\bar{\chi}', \beta'\$, c\gamma\#) \vdash$ **error**.

We can see that both automata behave the same way in the presented situations.

It has been shown that both kinds of automata behave the same way for the same input and, thus, they are equal in the sense of acceptance/rejection of the input strings. ☐

3.6 Summary

It has been shown that a transformation from automata with several symbols under the reading head to those, which use just a one-symbol reading head, is possible. Moreover, the transformation is informally quite easy. Nevertheless, the formal proof of equivalence of behaviour of both kinds of automata was presented too.

At the end of the chapter we ask more or less a rhetorical question: Is it possible to modify SSkPDA automata in such a way so that they remain deterministic, equal to the LL_k ones and they are *atomic* ones?

Chapter 4

Regulated Pushdown Automata

The present chapter demonstrates a recent investigation area of the formal language theory—*regulated automata* (see [54]). Specifically, it investigates pushdown automata that regulate the use of their rules by control languages. It proves that this regulation has no effect on the power of pushdown automata if the control languages are regular. However, the pushdown automata regulated by linear control languages characterise the family of recursively enumerable languages. All these results are established in terms of:

(A) acceptance by final state,

(B) acceptance by empty pushdown, and

(C) acceptance by final state and empty pushdown.

In its conclusion, this chapter formulates several open problems.

4.1 Introduction

Over the past three or four decades, grammars that regulate the use of their rules by various control mechanisms have played an important role in language theory. Indeed, literally hundreds of studies have been written about these grammars (see [21], Chapter 5 in the second volume of [62], and Chapter V in [63] for an overview of these studies). Besides grammars, however, language theory uses automata as fundamental language models, and this very elementary fact gives rise to the idea of regulated automata, which are introduced and discussed in the present paper.

More specifically, this chapter presents pushdown automata that regulate the use of their rules by control languages. First, it demonstrates that this regulation has no effect on the power of pushdown automata if the control languages are regular. Based on this result, it points out that pushdown automata regulated by analogy with the control mechanisms used in most common regulated grammars, such as matrix grammars, are of little interest because their resulting power coincides with the power of ordinary pushdown automata. Then, however, the present chapter proves that the pushdown automata increase their power remarkably if they are regulated by linear languages; indeed, they characterise the family of recursively enumerable languages.

All results given in this paper are established in terms of (A) acceptance by final state, (B) acceptance by empty pushdown, and (C) acceptance by final state and empty pushdown. In its conclusion, this chapter discusses some open problem areas concerning regulated automata.

4.2 Preliminaries

We assume that the reader is familiar with language theory (see [53]). The notation can be found in Chapter 2 as 2.2.1, 2.2.2, 2.2.3, 2.2.4, 2.2.5, 2.2.6. The definitions can be found in Chapter 2 as 2.2.2, 2.2.4, 2.2.5, 2.2.8, 2.2.9, 2.2.10, 2.2.11, and 2.2.12.

4.3 Definitions

Consider a pushdown automaton, M, and a control language, Ξ, over M's rules. Informally, with Ξ, M accepts a word, x, if and only if Ξ contains a control word according to which M makes a sequence of moves so it reaches a final configuration after reading x.

Formally, a *pushdown automaton* is a 7-tuple, $M = (Q, \Sigma, \Omega, R, s, S, F)$. In addition to 2.3.1, this chapter requires that Q, Σ, Ω are pairwise disjoint.

Let Ψ be an alphabet of *rule labels* such that $card(\Psi) = card(R)$, and ψ be a bijection from R to Ψ. For simplicity, to express that ψ maps a rule, $Apa \rightarrow wq \in R$, to ρ, where $\rho \in \Psi$, this paper writes $\rho.Apa \rightarrow wq \in R$; in other words, $\rho.Apa \rightarrow wq$ means $\psi(Apa \rightarrow wq) = \rho$. A *configuration* of M, χ, is any word from $\Omega^*Q\Sigma^*$. For every $x \in \Omega^*$, $y \in \Sigma^*$, and $\rho.Apa \rightarrow wq \in R$, M makes a move from configuration $xApay$ to configuration $xwqy$ according to ρ, written as $xApay \vdash xwqy\ [\rho]$. Let χ be any configuration of M. M makes *zero moves from χ to χ according to ε*, symbolically written as $\chi \vdash^0 \chi\ [\varepsilon]$. Let there exist a sequence of configurations $\chi_0, \chi_1, \ldots, \chi_n$ for some $n \geq 1$ such that $\chi_{i-1} \vdash \chi_i\ [\rho_i]$, where $\rho_i \in \Psi$, for $i = 1, \ldots, n$, then M makes *n moves from χ_0 to χ_n according to $\rho_1 \ldots \rho_n$*, symbolically written as $\chi_0 \vdash^n \chi_n\ [\rho_1 \ldots \rho_n]$.

Let Ξ be a *control language* over Ψ; that is, $\Xi \subseteq \Psi^*$. With Ξ, M defines the following three types of accepted languages:

$L(M, \Xi, 1)$—*the language accepted by final state*

$L(M, \Xi, 2)$—*the language accepted by empty pushdown*

$L(M, \Xi, 3)$—*the language accepted by final state and empty pushdown*

defined as follows. Let $\chi \in \Omega^*Q\Sigma^*$. If $\chi \in \Omega^*F$, $\chi \in Q$, $\chi \in F$, then χ is a *1-final configuration*, *2-final configuration*, *3-final configuration*, respectively. For $i = 1, 2, 3$, define $L(M, \Xi, i)$ as $L(M, \Xi, i) = \{w \mid w \in \Sigma^*,\ and\ Ssw \Rightarrow^* \chi\ [\sigma]\ in\ M\ for\ an\ i-$final configuration, χ, $and\ \sigma \in \Xi\}$.

For any family of languages, X, set $RPD(X, i) = \{L \mid L = L(M, \Xi, i),\ where\ M\ is\ a\ pushdown\ automaton\ and\ \Xi \in X\}$, where $i = 1, 2, 3$. Specifically, $RPD(REG, i)$ and $RPD(LIN, i)$ are central to this chapter.

4.4 Results

This section demonstrates that $CF = RPD(REG, 1) = RPD(REG, 2) = RPD(REG, 3)$ and $RE = RPD(LIN, 1) = RPD(LIN, 2) = RPD(LIN, 3)$.

Some of the following proofs involve several grammars and automata. To avoid any confusion, these proofs sometimes specify a regular grammar, G, as $G = (V[G], P[G], S[G], T[G])$ because this specification clearly expresses that $V[G]$, $P[G]$, $S[G]$, and $T[G]$ represent G's components. Other grammars and automata are specified analogously whenever any confusion may exist.

4.4.1 Regular Control Languages

Next, this section proves that if the control languages are regular, then the regulation of pushdown automata has no effect on their power. The proof of Lemma 4.4.1 presents a transformation that converts any regular grammar, G, and any pushdown automaton, K, to an ordinary pushdown automaton, M, such that $L(M) = L(K, L(G), 1)$.

Lemma 4.4.1 *For every regular grammar, G, and every pushdown automaton, K, there exists a pushdown automaton, M, such that $L(M) = L(K, L(G), 1)$.*

Proof: Let $G = (N[G], T[G], P[G], S[G])$ be any regular grammar, and let $K = (Q[K], \Sigma[K], \Omega[K], R[K], s[K], S[K], F[K])$ be any pushdown automaton. Next, we construct a pushdown automaton, M, that simultaneously simulates G and K so that $L(M) = L(K, L(G), 1)$.

Let f be a new symbol. Define the pushdown automaton $M = (Q[M], \Sigma[M], \Omega[M], R[M], s[M], S[M], F[M])$ as $Q[M] = \{\langle qB\rangle \mid q \in Q[K], B \in N[G] \cup \{f\}\}$, $\Sigma[M] = \Sigma[K]$, $\Omega[M] = \Omega[K]$, $s[M] = \langle s[K]S[G]\rangle$, $S[M] = S[K]$, $F[M] = \{\langle qf\rangle \mid q \in F[K]\}$, and $R[M] = \{C\langle qA\rangle b \rightarrow x\langle pB\rangle \mid a.Cqb \rightarrow xp \in R[K], A \rightarrow aB \in P[G]\} \cup \{C\langle qA\rangle b \rightarrow x\langle pf\rangle \mid a.Cqb \rightarrow xp \in R[K], A \rightarrow a \in P[G]\}$.

Observe that a move in M according to $C\langle qA\rangle b \rightarrow x\langle pB\rangle \in R[M]$ simulates a move in K according to $a.Cqb \rightarrow xp \in R[K]$, where a is generated in G by using $A \rightarrow aB \in P[G]$. Based on this observation, it is rather easy to see that M

accepts an input word, w, if and only if K reads w and enters a final state after using a complete word of $L(G)$; therefore, $L(M) = L(K, L(G), 1)$. A rigorous proof that $L(M) = L(K, L(G), 1)$ is left to the reader. □

Theorem 4.4.1 *For* $i \in \{1, 2, 3\}$, $CF = RPD(REG, i)$.

Proof: To prove $CF = RPD(REG, 1)$, notice that $RPD(REG, 1) \subseteq CF$ follows from Lemma 4.4.1. Clearly, $CF \subseteq RPD(REG, 1)$, so $RPD(REG, 1) = CF$.

By analogy with the demonstration of $RPD(REG, 1) = CF$, prove that $CF = RPD(REG, 2)$ and $CF = RPD(REG, 3)$. □

Let us point out that most fundamental regulated grammars use control mechanisms that can be expressed in terms of regular control languages (c.f. Theorem V.6.1 on page 175 in [63]). However, pushdown automata introduced by analogy with these grammars are of little or no interest because they are as powerful as ordinary pushdown automata (see Theorem 4.4.1 above).

4.4.2 Linear Control Languages

The rest of this section demonstrates that the pushdown automata regulated by linear control languages are more powerful than ordinary pushdown automata. In fact, it proves that $RE = RPD(LIN, 1) = RPD(LIN, 2) = RPD(LIN, 3)$.

Lemma 4.4.2 *For every left-extended queue grammar, K, there exists a left-extended queue grammar $Q = (V, T, W, F, s, P)$ satisfying $L(K) = L(Q)$, $!$ is a distinguished member of $(W - F)$, $V = U \cup Z \cup T$ such that U, Z, T are pairwise disjoint, and Q derives every $z \in L(Q)$ in this way*

$$\begin{array}{lll}
\#S & \Rightarrow^{+} & x\#b_1b_2\ldots b_n! \\
 & \Rightarrow & xb_1\#b_2\ldots b_ny_1p_2 \\
 & \Rightarrow & xb_1b_2\#b_3\ldots b_ny_1y_2p_3 \\
 & \vdots & \\
 & \Rightarrow & xb_1b_2\ldots b_{n-1}\#b_ny_1y_2\ldots y_{n-1}p_n \\
 & \Rightarrow & xb_1b_2\ldots b_{n-1}b_n\#y_1y_2\ldots y_np_{n+1}
\end{array}$$

where $n \in N$, $x \in U^*$, $b_i \in Z$ for $i = 1, \ldots, n$, $y_i \in T^*$ for $i = 1, \ldots, n$, $z = y_1y_2\ldots y_n$, $p_i \in W - \{!\}$ for $i = 1, \ldots, n-1$, $p_n \in F$, and in this derivation $x\#b_1b_2\ldots b_n!$ is the only word containing $!$.

Proof: Let K be any left-extended queue grammar. Convert K to a left-extended queue grammar, $H = (V[H], T[H], W[H], F[H], S[H], P[H])$, such that $L(K) = L(H)$ and H generates every $x \in L(H)$ by making two or more derivation steps (this conversion is trivial and left to the reader).

Define the bijection α from W to W', where $W' = \{q' \mid q \in W\}$, as $\alpha(q) = \{q'\}$ for every $q \in W$. Analogously, define the bijection β from W

to W'', where $W'' = \{q'' \mid q \in W\}$, as $\beta(q) = \{q''\}$ for every $q \in W$. Without any loss of generality, assume that $\{1,2\} \cap (V \cup W) = \emptyset$. Set $\Xi = \{\langle a, q, u1v, p\rangle \mid (a, q, uv, p) \in P[H] \text{ for some } a \in V, q \in W - F, v \in T^*, u \in V^*, \text{ and } p \in W\}$ and $\Gamma = \{\langle a, q, z2w, p\rangle \mid (a, q, zw, p) \in P[H] \text{ for some } a \in V, q \in W - F, w \in T^*, z \in V^*, \text{ and } p \in W\}$. Define the relation χ from $V[H]$ to $\Xi\Gamma$ so for every $a \in V$, $\chi(a) = \{\langle a, q, y1x, p\rangle\langle a, q, y2x, p\rangle \mid \langle a, q, y1x, p\rangle \in \Xi, \langle a, q, y2x, p\rangle \in \Gamma, q \in W - F, x \in T^*, y \in V^*, p \in W\}$. Define the bijection δ from $V[H]$ to V', where $V' = \{a' \mid a \in V\}$, as $\delta(a) = \{a'\}$. In the standard manner, extend δ so it is defined from $(V[H])^*$ to $(V')^*$. Finally, define the bijection ϕ from $V[H]$ to V'', where $V'' = \{a'' \mid a \in V\}$, as $\phi(a) = \{a''\}$. In the standard manner, extend ϕ so it is defined from $(V[H])^*$ to $(V'')^*$.

Define the left-extended queue grammar

$$Q = (V[Q], T[Q], W[Q], F[Q], S[Q], P[Q])$$

so that $V[Q] = V[H] \cup \delta(V[H]) \cup \phi(V[H]) \cup \Xi \cup \Gamma$, $T[Q] = T[H]$, $W[Q] = W[H] \cup \alpha(W[H]) \cup \beta(W[H]) \cup \{!\}$, $F[Q] = \beta(F[H])$, $S[Q] = \delta(S[H])$, and $P[V]$ is constructed in this way

1. if $(a, q, x, p) \in P[H]$ where $a \in V$, $q \in W - F$, $x \in V^*$, and $p \in W$, then add $(\delta(a), q, \delta(x), p)$ and $(\delta(a), \alpha(q), \delta(x), \alpha(p))$ to $P[Q]$;
2. if $(a, q, xAy, p) \in P[H]$, where $a \in V$, $q \in W - F$, $x, y \in V^*$, $A \in V$, and $p \in W$, then add $(\delta(a), q, \delta(x)\chi(A)\phi(y), \alpha(p))$ to $P[Q]$;
3. if $(a, q, yx, p) \in P[H]$, where $a \in V$, $q \in W - F$, $y \in V^*$, $x \in T^*$, and $p \in W$, then add $(\langle a, q, y1x, p\rangle, \alpha(q), \phi(y), !)$ and $(\langle a, q, y2x, p\rangle, !, x, \beta(p))$ to $P[Q]$;
4. if $(a, q, y, p) \in P[H]$, where $a \in V$, $q \in W - F$, $y \in T^*$, and $p \in W$, then add $(\phi(a), \beta(q), y, \beta(p))$ to $P[Q]$.

Set $U = \delta(V[H]) \cup \Xi$ and $Z = \phi(V[H]) \cup \Gamma$. Notice that Q satisfies properties 2 and 3 of Lemma 4.4.2. To demonstrate that the other two properties hold as well, observe that H generates every $z \in L(H)$ in this way

$$\begin{array}{lll}
\#S[H] & \Rightarrow^+ & x\#b_1b_2\ldots b_ip_1 \\
& \Rightarrow & xb_1\#b_2\ldots b_ib_{i+1}\ldots b_ny_1p_2 \\
& \Rightarrow & xb_1b_2\#b_3\ldots b_ib_{i+1}\ldots b_ny_1y_2p_3 \\
& \vdots & \\
& \Rightarrow & xb_1b_2\ldots b_{i-1}\#b_ib_{i+1}\ldots b_ny_1y_2\ldots y_{i-1}p_i \\
& \Rightarrow & xb_1b_2\ldots b_i\#b_{i+1}\ldots b_ny_1y_2\ldots y_{i-1}y_ip_{i+1} \\
& \vdots & \\
& \Rightarrow & xb_1b_2\ldots b_{n-1}\#b_ny_1y_2\ldots y_{n-1}p_n \\
& \Rightarrow & xb_1b_2\ldots b_{n-1}b_n\#y_1y_2\ldots y_np_{n+1}
\end{array}$$

where $n \in \mathcal{N}$, $x \in V^+$, $b_i \in V$ for $i = 1, \ldots, n$, $y_i \in T^*$ for $i = 1, \ldots, n$, $z = y_1 y_2 \ldots y_n$, $p_i \in W$ for $i = 1, \in, n$, $p_{n+1} \in F$. Q simulates this generation of z as follows

$$\begin{array}{lll}
\#S[Q] & \Rightarrow^+ & \delta(x)\#\chi(b_1)\phi(b_2 \ldots b_i)\alpha(p_1) \\
& \Rightarrow & \delta(x)\langle b_1, p_1, b_{i+1} \ldots b_n 1 y_1, p_2\rangle\#\langle b_1, p_1, b_{i+1} \ldots b_n 2 y_1, p2\rangle \\
& & \phi(b_2 \ldots b_i b_{i+1} \ldots b_n)! \\
& \Rightarrow & \delta(x)\chi(b_1)\#\phi(b_2 \ldots b_n) y_1 p_2 \\
& \Rightarrow & \delta(x)\chi(b_1)\phi(b_2)\#\phi(b_3 \ldots b_n) y_1 y_2 p_3 \\
& \vdots & \\
& \Rightarrow & \delta(x)\chi(b_1)\phi(b_2 \ldots b_{n-1})\#\phi(b_n) y_1 y_2 \ldots y_{n-1} p_n \\
& \Rightarrow & \delta(x)\chi(b_1)\phi(b_2 \ldots b_n)\# y_1 y_2 \ldots y_n p_{n+1}
\end{array}$$

Q makes the first $|x| - 1$ steps of $\#S[Q] \Rightarrow^+ \delta(x)\#\chi(b_1)\phi(b_2 \ldots b_i)\alpha(p_1)$ according to productions introduced in 1; in addition, during this derivation, Q makes one step by using a production introduced in 2. By using productions introduced in 3, Q makes the two steps

$$\begin{array}{lr}
\delta(x)\#\chi(b_1)\phi(b_2 \ldots b_i)\alpha(p_0) & \Rightarrow \\
\delta(x)\langle b_1, p_1, b_{i+1} \ldots b_n 1 y_1, p_2\rangle\#\langle b_1, p_1, b_{i+1} \ldots b_n 2 y_1, p_2\rangle\phi(b_2 \ldots b_i b_{i+1} \ldots b_n)! & \Rightarrow \\
\delta(x)\chi(b_1)\#\phi(b_2 \ldots b_n) y_1 p_2 &
\end{array}$$

with

$$\chi(b_1) = \langle b_1, p_0, b_{i+1} \ldots b_n 1 y_1, p_1\rangle\langle b_1, p_0, b_{i+1} \ldots b_n 2 y_1, p2\rangle.$$

Q makes the rest of the derivation by using productions introduced in 4.

Based on the previous observation, it is easy to see that Q satisfies all the four properties stated in Lemma 4.4.2, whose rigorous proof is left to the reader. □

Lemma 4.4.3 *Let Q be a left-extended queue grammar that satisfies the properties of Lemma 4.4.2. Then, there exists a linear grammar, G, and a pushdown automaton, M, such that $L(Q) = L(M, L(G), 3)$.*

Proof: Let $Q = (V[Q], T[Q], W[Q], F[Q], s[Q], P[Q])$ be a left-extended queue grammar satisfying the properties of Lemma 4.4.2. Without any loss of generality, assume that $\{@, \pounds, \P\} \cap (V \cup W) = \emptyset$. Define the coding, ζ, from $(V[Q])^*$ to $\{\langle \pounds a s\rangle \mid a \in V[Q]\}^*$ as $\zeta(a) = \{\langle \pounds a s\rangle\}$ (s is used as the start state of the pushdown automaton, M, defined later in this proof).

Construct the linear grammar $G = (N[G], T[G], P[G], S[G])$ in the following way. Initially, set

$$N[G] = \{S[G], \langle !\rangle, \langle !, 1\rangle\} \cup \{\langle f\rangle \mid f \in F[Q]\}$$

$$T[G] = \zeta(V[Q]) \cup \{\langle \pounds \S s\rangle, \langle \pounds @\rangle\} \cup \{\langle \pounds \S f\rangle \mid f \in F[Q]\}$$

$$P[G] = \{S[G] \rightarrow \langle \pounds \S s\rangle\langle f\rangle \mid f \in F[Q]\} \cup \{\langle !\rangle \rightarrow \langle !, 1\rangle\langle \pounds @\rangle\}$$

Increase $N[G]$, $T[G]$, and $P[G]$ by performing 1 through 3, following next.

1. for every $(a, p, x, q) \in P[Q]$ where $p, q \in W[Q]$, $a \in Z$, $x \in T^*$,

$$\begin{aligned} N[G] &= N[G] \cup \{\langle apxqk\rangle \mid k = 0, \dots, |x|\} \cup \{\langle p\rangle, \langle q\rangle\} \\ T[G] &= T[G] \ \cup \{\langle \pounds sym(y,k)\rangle \mid k = 1, \dots, |y|\} \cup \{\langle \pounds apxq\rangle\} \\ P[G] &= P[G] \ \cup \{\langle q\rangle \rightarrow \langle apxq|x|\rangle\langle \pounds apxq\rangle, \langle apxq0\rangle \rightarrow \langle p\rangle\} \\ &\qquad\qquad \cup \{\langle apxqk\rangle \rightarrow \langle apxq(k-1)\rangle\langle \pounds sym(x,k)\rangle \mid k = 1, \dots, |x|\}; \end{aligned}$$

2. for every $(a, p, x, q) \in P[Q]$ with $p, q \in W[Q]$, $a \in U$, $x \in (V[Q])^*$,

$$\begin{aligned} N[G] &= N[G] \cup \{\langle p, 1\rangle, \langle q, 1\rangle\} \\ P[G] &= P[G] \ \cup \{\langle q, 1\rangle \rightarrow reversal(\zeta(x))\langle p, 1\rangle\zeta(a)\}; \end{aligned}$$

3. for every $(a, p, x, q) \in P[Q]$ with $ap = S[Q]$, $p, q \in W[Q]$, $x \in (V[Q])^*$,

$$\begin{aligned} N[G] &= N[G] \cup \{\langle q, 1\rangle\} \\ P[G] &= P[G] \ \cup \{\langle q, 1\rangle \rightarrow reversal(x)\langle \pounds \$ s\rangle\}. \end{aligned}$$

The construction of G is completed. Set $\Psi = T[G]$. Ψ represents the alphabet of rule labels corresponding to the rules of the pushdown automaton $M = (Q[M], \Sigma[M], \Omega[M], R[M], s[M], S[M], \{\rceil\})$, which is constructed next.

Initially, set $Q[M] = \{s[M], \langle \P !\rangle, \lfloor, \rceil\}$ (throughout the rest of this proof, $s[M]$ is abbreviated to s), $\Sigma[M] = T[Q]$, $\Omega[M] = \{S[M], \S\} \cup V[Q]$, $R[M] = \{\langle \pounds \S s\rangle.S[M]s \rightarrow \S s\} \cup \{\langle \pounds \S f\rangle.\S\langle \P f\rangle \rightarrow\rceil \mid f \in F[M]\}$. Increase $Q[M]$ and $R[M]$ by performing A through D, following next.

A. $R[M] = R[M] \cup \{\langle \pounds bs\rangle.as \rightarrow abs \mid a \in \Omega[M] - \{S[M]\}, b \in \Omega[M] - \{\$\}\}$;

B. $R[M] = R[M] \cup \{\langle \pounds \$ s\rangle.as \rightarrow a\lfloor \mid a \in V[Q]\} \cup \{\langle \pounds a\rangle.a\lfloor\rightarrow \lfloor \mid a \in V[Q]\}$;

C. $R[M] = R[M] \cup \{\langle \pounds @\rangle.a\lfloor\rightarrow a\langle \P !\rangle \mid a \in Z\}$;

D. for every $(a, p, x, q) \in P[Q]$, where $p, q \in W[Q]$, $a \in Z$, $x \in (T[Q])^*$,

$$\begin{aligned} Q[M] &= Q[M] \cup \{\langle \P p\rangle\} \cup \{\langle \P qu\rangle \mid u \in prefix(x)\} \\ R[M] &= R[M] \cup \{\langle \pounds b\rangle.a\langle \P qy\rangle b \rightarrow a\langle \P qyb\rangle \mid b \in T[Q], y \in (T[Q])^*, \\ &\qquad yb \in prefix(x)\} \cup \{\langle \pounds apxq\rangle.a\langle \P qx\rangle \rightarrow \langle \P p\rangle\}. \end{aligned}$$

The construction of M is completed.

Notice that several components of G and M have this form: $\langle x\rangle$. Intuitively, if x begins with £, then $\langle x\rangle \in T[G]$. If x begins with ¶, then $\langle x\rangle \in Q[M]$. Finally, if x begins with a symbol different from £ or ¶, then $\langle x\rangle \in N[G]$.

First, we only sketch the reason $L(Q)$ contains $L(M, L(G), 3)$. According to a word from $L(G)$, M accepts every word w as

$$
\begin{array}{lll}
\S w_1 \dots w_{m-1} w_m & \vdash^{+} & \S b_m \dots b_1 a_n \dots a_1 s w_1 \dots w_{m-1} w_m \\
 & \vdash & \S b_m \dots b_1 a_n \dots a_1 \lfloor w_1 \dots w_{m-1} w_m \\
 & \vdash^{n} & \S b_m \dots b_1 \lfloor w_1 \dots w_{m-1} w_m \\
 & \vdash & \S b_m \dots b_1 \langle \P q_1 \rangle w_1 \dots w_{m-1} w_m \\
 & \vdash^{|w_1|} & \S b_m \dots b_1 \langle \P q_1 w_1 \rangle w_2 \dots w_{m-1} w_m \\
 & \vdash & \S b_m \dots b_2 \langle \P q_2 \rangle w_2 \dots w_{m-1} w_m \\
 & \vdash^{|w_2|} & \S b_m \dots b_2 \langle \P q_2 w_2 \rangle w_3 \dots w_{m-1} w_m \\
 & \vdash & \S b_m \dots b_3 \langle \P q_3 \rangle w_3 \dots w_{m-1} w_m \\
 & \vdots & \\
 & \vdash & \S b_m \langle \P q_m \rangle w_m \\
 & \vdash^{|w_m|} & \S b_m \langle \P q_m w_m \rangle \\
 & \vdash & \S \langle \P q_{m+1} \rangle \\
 & \vdash & \rceil
\end{array}
$$

where $w = w_1 \dots w_{m-1} w_m$, $a_1 \dots a_n b_1 \dots b_m = x_1 \dots x_{n+1}$, and $R[Q]$ contains $(a_0, p_0, x_1, p_1), (a_1, p_1, x_2, p_2), \dots, (a_n, p_n, x_{n+1}, q_1), (b_1, q_1, w_1, q_2), (b_2, q_2, w_2, q_3), \dots, (b_m, q_m, w_m, q_{m+1})$. According to these members of $R[Q]$, Q makes

$$
\begin{array}{llll}
\# a_0 p_0 & \Rightarrow & a_0 \# y_0 x_1 p_1 & [(a_0, p_0, x_1, p_1)] \\
 & \Rightarrow & a_0 a_1 \# y_1 x_2 p_2 & [(a_1, p_1, x_2, p_2)] \\
 & \Rightarrow & a_0 a_1 a_2 \# y_2 x_3 p_3 & [(a_2, p_2, x_3, p_3)] \\
 & \vdots & & \\
 & \Rightarrow & a_0 a_1 a_2 \dots a_{n-1} \# y_{n-1} x_n p_n & [(a_{n-1}, p_{n-1}, x_n, p_n)] \\
 & \Rightarrow & a_0 a_1 a_2 \dots a_n \# y_n x_{n+1} q_1 & [(a_n, p_n, x_{n+1}, q_1)] \\
 & \Rightarrow & a_0 \dots a_n b_1 \# b_2 \dots b_m w_1 q_2 & [(b_1, q_1, w_1, q_2)] \\
 & \Rightarrow & a_0 \dots a_n b_1 b_2 \# b_3 \dots b_m w_1 w_2 q_3 & [(b_2, q_2, w_2, q_3)] \\
 & \vdots & & \\
 & \Rightarrow & a_0 \dots a_n b_1 \dots b_{m-1} \# b_m w_1 w_2 \dots w_{m-1} q_m & [(b_{m-1}, q_{m-1}, w_{m-1}, q_m)] \\
 & \Rightarrow & a_0 \dots a_n b_1 \dots b_m \# w_1 w_2 \dots w_m q_{m+1} & [(b_m, q_m, w_m, q_{m+1})]
\end{array}
$$

Therefore, $L(M, L(G), 3) \subseteq L(Q)$.

More formally, to demonstrate that $L(Q)$ contains $L(M, L(G), 3)$, consider any $h \in L(G)$. G generates h as

$$
\begin{array}{lll}
S[G] & \Rightarrow & \langle \pounds \S s \rangle \langle q_{m+1} \rangle \\
 & \Rightarrow^{|w_m|+1} & \langle \pounds \S s \rangle \langle q_m \rangle t_m \langle \pounds b_m q_m w_m q_{m+1} \rangle \\
 & \Rightarrow^{|w_{m-1}|+1} & \langle \pounds \S s \rangle \langle q_{m-1} \rangle t_{m-1} \langle \pounds b_{m-1} q_{m-1} w_{m-1} q_m \rangle t_m \langle \pounds b_m q_m w_m q_{m+1} \rangle \\
 & \vdots & \\
 & \Rightarrow^{|w_1|+1} & \langle \pounds \S s \rangle \langle q_1 \rangle o \\
 & \Rightarrow^{|w_1|+1} & \langle \pounds \S s \rangle \langle q_1, 1 \rangle \langle \pounds @ \rangle o \\
 & & [\langle q_1 \rangle \rightarrow \langle q_1, 1 \rangle \langle \pounds @ \rangle]
\end{array}
$$

$$\begin{array}{ll}
\Rightarrow & \langle \pounds\S s\rangle\zeta(\mathit{reversal}(x_{n+1}))\langle p_n,1\rangle\langle \pounds a_n\rangle\langle \pounds @\rangle o \\
 & [\langle q_1,1\rangle \rightarrow \mathit{reversal}(\zeta(x_{n+1}))\langle p_n,1\rangle\langle \pounds a_n\rangle\langle \pounds @\rangle] \\
 & \\
\Rightarrow & \langle \pounds\S s\rangle\zeta(\mathit{reversal}(x_n x_{n+1}))\langle p_{n-1},1\rangle\langle \pounds a_{n-1}\rangle\langle \pounds a_n\rangle\langle \pounds @\rangle o \\
 & [\langle p_n,1\rangle \rightarrow \mathit{reversal}(\zeta(x_n))\langle p_{n-1},1\rangle\langle \pounds a_{n-1}\rangle] \\
 & \\
\vdots & \\
 & \\
\Rightarrow & \langle \pounds\S s\rangle\zeta(\mathit{reversal}(x_2\ldots x_n x_{n+1}))\langle p1,1\rangle\langle \pounds a_1\rangle\langle \pounds a_2\rangle\ldots\langle \pounds a_n\rangle\langle \pounds @\rangle o \\
 & [\langle p_2,1\rangle \rightarrow \mathit{reversal}(\zeta(x_2))\langle p_1,1\rangle\langle \pounds a_1\rangle] \\
 & \\
\Rightarrow & \langle \pounds\S s\rangle\zeta(\mathit{reversal}(x_1\ldots x_n x_{n+1}))\langle \pounds\$ s\rangle\langle \pounds a_1\rangle\langle \pounds a_2\rangle\ldots\langle \pounds a_n\rangle\langle \pounds @\rangle o \\
 & [\langle p_1,1\rangle \rightarrow \mathit{reversal}(\zeta(x_1))\langle \pounds\$ s\rangle]
\end{array}$$

where $n, m \in \mathcal{N}$; $a_i \in U$ for $i = 1,\ldots,n$; $b_k \in Z$ for $k = 1,\ldots,m$; $x_l \in V^*$ for $l = 1,\ldots,n+1$; $p_i \in W$ for $i = 1,\ldots,n$; $q_l \in W$ for $l = 1,\ldots,m+1$ with $q_1 = !$ and $q_{m+1} \in F$; $t_k = \langle \pounds sym(w_k,1)\rangle\ldots\langle \pounds sym(w_k,|w_k|-1)\rangle\langle \pounds sym(w_k,|w_k|)\rangle$ for $k = 1,\ldots,m$; $o = t_1\langle \pounds b_1 q_1 w_1 q_2\rangle\ldots\langle \pounds\S s\rangle\langle q_{m-1}\rangle t_{m-1}\langle \pounds b_{m-1} q_{m-1} w_{m-1} q_m\rangle t_m\langle \pounds b_m q_m w_m q_{m+1}\rangle$; $h = \langle \pounds\S s\rangle\zeta(\mathit{reversal}(x_1\ldots x_n x_{n+1}))\langle \pounds\$\rangle\langle \pounds a_1\rangle\langle \pounds a_2\rangle\ldots\langle \pounds a_n\rangle\langle \pounds @\rangle o$.

In greater detail, G makes $S[G] \Rightarrow \langle \pounds\S s\rangle\langle q_{m+1}\rangle$ according to $S[G] \rightarrow \langle \pounds\S s\rangle\langle q_{m+1}\rangle$. Furthermore, G makes

$$\begin{array}{ll}
 & \langle \pounds\S s\rangle\langle q_{m+1}\rangle \\
\Rightarrow^{|w_m|+1} & \langle \pounds\S s\rangle\langle q_m\rangle t_m\langle \pounds b_m q_m w_m q_{m+1}\rangle \\
\Rightarrow^{|w_{m-1}|+1} & \langle \pounds\S s\rangle\langle q_{m-1}\rangle t_{m-1}\langle \pounds b_{m-1} q_{m-1} w_{m-1} q_m\rangle t_m\langle \pounds b_m q_m w_m q_{m+1}\rangle \\
\vdots & \\
\Rightarrow^{|w_1|+1} & \langle \pounds\S s\rangle\langle q_1\rangle o
\end{array}$$

according to productions introduced in step 1. Then, G makes

$$\langle \pounds\S s\rangle\langle q_1\rangle o \Rightarrow \langle \pounds\S s\rangle\langle q_1,1\rangle\langle \pounds @\rangle o$$

according to $\langle !\rangle \rightarrow \langle !,1\rangle\langle \pounds @\rangle$ (recall that $q_1 = !$). After this step, G makes

$$\begin{array}{ll}
 & \langle \pounds\S s\rangle\langle q_1,1\rangle\langle \pounds @\rangle o \\
\Rightarrow & \langle \pounds\S s\rangle\zeta(\mathit{reversal}(x_{n+1}))\langle p_n,1\rangle\langle \pounds a_n\rangle\langle \pounds @\rangle o \\
\Rightarrow & \langle \pounds\S s\rangle\zeta(\mathit{reversal}(x_n x_{n+1}))\langle p_{n-1},1\rangle\langle \pounds a_{n-1}\rangle\langle \pounds a_n\rangle\langle \pounds @\rangle o \\
\vdots & \\
\Rightarrow & \langle \pounds\S s\rangle\zeta(\mathit{reversal}(x_2\ldots x_n x_{n+1}))\langle p_1,1\rangle\langle \pounds a_1\rangle\langle \pounds a_2\rangle\ldots\langle \pounds a_n\rangle\langle \pounds @\rangle o
\end{array}$$

according to productions introduced in step 2. Finally, according to $\langle p_1,1\rangle \rightarrow \mathit{reversal}(\zeta(x_1))\langle \pounds\$\rangle$, which is introduced in step 3, G makes

$$\begin{array}{ll} & \langle \pounds\S s\rangle \zeta(\mathit{reversal}(x_2 \ldots x_n x_{n+1}))\langle p_1, 1\rangle\langle \pounds a_1\rangle\langle \pounds a_2\rangle \ldots \langle \pounds a_n\rangle\langle \pounds @\rangle o \\ \Rightarrow & \langle \pounds\S s\rangle \zeta(\mathit{reversal}(x_1 \ldots x_n x_{n+1}))\langle \pounds \$\rangle\langle \pounds a_1\rangle\langle \pounds a_2\rangle \ldots \langle \pounds a_n\rangle\langle \pounds @\rangle o \end{array}$$

If $a_1 \ldots a_n b_1 \ldots b_m$ differs from $x_1 \ldots x_{n+1}$, then M does not accept according to h. Assume that $a_1 \ldots a_n b_1 \ldots b_m = x_1 \ldots x_{n+1}$. At this point, according to h, M makes this sequence of moves

$$\begin{array}{lll}
\S w_1 \ldots w_{m-1} w_m & \vdash^{+} & \S b_m \ldots b_1 a_n \ldots a_1 s w_1 \ldots w_{m-1} w_m \\
 & \vdash & \S b_m \ldots b_1 a_n \ldots a_1 \lfloor w_1 \ldots w_{m-1} w_m \\
 & \vdash^{n} & \S b_m \ldots b_1 \lfloor w_1 \ldots w_{m-1} w_m \\
 & \vdash & \S b_m \ldots b_1 \langle \P q_1\rangle w_1 \ldots w_{m-1} w_m \\
 & \vdash^{|w_1|} & \S b_m \ldots b_1 \langle \P q_1 w_1\rangle w_2 \ldots w_{m-1} w_m \\
 & \vdash & \S b_m \ldots b_2 \langle \P q_2\rangle w_2 \ldots w_{m-1} w_m \\
 & \vdash^{|w_2|} & \S b_m \ldots b_2 \langle \P q_2 w_2\rangle w_3 \ldots w_{m-1} w_m \\
 & \vdash & \S b_m \ldots b_3 \langle \P q_3\rangle w_3 \ldots w_{m-1} w_m \\
 & \vdots & \\
 & \vdash & \S b_m \langle \P q_m\rangle w_m \\
 & \vdash^{|w_m|} & \S b_m \langle \P q_m w_m\rangle \\
 & \vdash & \S \langle \P q_{m+1}\rangle \\
 & \vdash & \rceil
\end{array}$$

In other words, according to h, M accepts $w_1 \ldots w_{m-1} w_m$. Return to the generation of h in G. By the construction of $P[G]$, this generation implies that $R[Q]$ contains $(a_0, p_0, x_1, p_1), (a_1, p_1, x_2, p_2), \ldots, (a_{j-1}, p_{j-1}, x_j, p_j), \ldots, (a_n, p_n, x_{n+1}, q_1), (b_1, q_1, w_1, q_2), (b_2, q_2, w_2, q_3), \ldots, (b_m, q_m, w_m, q_{m+1})$.

Thus, in Q,

$$\begin{array}{clll}
\# a_0 p_0 & \Rightarrow & a_0 \# y_0 x_1 p_1 & [(a_0, p_0, x_1, p_1)] \\
 & \Rightarrow & a_0 a_1 \# y_1 x_2 p_2 & [(a_1, p_1, x_2, p_2)] \\
 & \Rightarrow & a_0 a_1 a_2 \# y_2 x_3 p_3 & [(a_2, p_2, x_3, p_3)] \\
 & \vdots & & \\
 & \Rightarrow & a_0 a_1 a_2 \ldots a_{n-1} \# y_{n-1} x_n p_n & [(a_{n-1}, p_{n-1}, x_n, p_n)] \\
 & \Rightarrow & a_0 a_1 a_2 \ldots a_n \# y_n x_{n+1} q_1 & [(a_n, p_n, x_{n+1}, q_1)] \\
 & \Rightarrow & a_0 \ldots a_n b_1 \# b_2 \ldots b_m w_1 q_2 & [(b_1, q_1, w_1, q_2)] \\
 & \Rightarrow & a_0 \ldots a_n b_1 b_2 \# b_3 \ldots b_m w_1 w_2 q_3 & [(b_2, q_2, w_2, q_3)] \\
 & \vdots & & \\
 & \Rightarrow & a_0 \ldots a_n b_1 \ldots b_{m-1} \# b_m w_1 w_2 \ldots w_{m-1} q_m & [(b_{m-1}, q_{m-1}, w_{m-1}, q_m)] \\
 & \Rightarrow & a_0 \ldots a_n b_1 \ldots b_m \# w_1 w_2 \ldots w_m q_{m+1} & [(b_m, q_m, w_m, q_{m+1})]
\end{array}$$

Therefore, $w_1 w_2 \ldots w_m \in L(Q)$. Consequently, $L(M, L(G), 3) \subseteq L(Q)$.

A proof that $L(Q) \subseteq L(M, L(G), 3)$ is left to the reader. As $L(Q) \subseteq L(M, L(G), 3)$ and $L(M, L(G), 3) \subseteq L(Q)$, $L(Q) = L(M, L(G), 3)$. Therefore, Lemma 4.4.3 holds. $\square$

Theorem 4.4.2 *For* $i \in \{1, 2, 3\}$, $RE = RPD(LIN, i)$.

Proof: Obviously, $RPD(LIN, 3) \subseteq RE$. To prove $RE \subseteq RPD(LIN, 3)$, consider any recursively enumerable language, $L \in RE$. By Theorem 2.1 in [44], $L(Q) = L$, for a queue grammar. Clearly, there exists a left-extended queue grammar, Q', so $L(Q) = L(Q')$. Furthermore, by Lemmas 4.4.2 and 4.4.3, $L(Q') = L(M, L(G), 3)$, for a linear grammar, G, and a pushdown automaton, M. Thus, $L = L(M, L(G), 3)$. Hence, $RE \subseteq RPD(LIN, 3)$. As $RPD(LIN, 3) \subseteq RE$ and $RE \subseteq RPD(LIN, 3)$, $RE = RPD(LIN, 3)$.

By analogy with the demonstration of $RE = RPD(LIN, 3)$, prove $RE = RPD(LIN, i)$ for $i = 1, 2$. □

4.5 Chapter Summary and Open Problems

As already pointed out, this chapter has presented regulated automata as a recent investigation field of the formal language theory. Therefore, it has defined all notions and established all results in terms of this field. However, this approach does not rule out a relation of the achieved results to the classical formal language theory. Specifically, Theorem 4.4.2 can be viewed as a new characterisation of RE and compared with other well-known characterisations of this family (see pages 180 through 184 in the first volume of [62] for an overview of these characterisations).

Several research topics can be seen as open with respect to the term of regulated pushdown automata:

A. For $i = 1, \dots, 3$, consider $RPD(X, i)$, where X is a language family satisfying $REG \subset X \subset LIN$; for instance, set X equal to the family of minimal linear languages. Compare RE with $RPD(X, i)$.

B. By analogy with regulated pushdown automata, introduce and study some other types of regulated automata.

On the other hand, several topics originally proposed by [54] are further presented in this thesis. They are:

1. Descriptional complexity of regulated pushdown automata.
2. Special cases of regulated pushdown automata, such as their deterministic versions,

Chapter 5

Minimisation of RPA

This chapter presents some simple and natural restrictions of regulated pushdown automata, whose moves are regulated by some control languages. Most importantly, it studies one-turn regulated pushdown automata and proves that they characterise the family of recursively enumerable languages. In fact, this characterisation holds even for atomic one-turn regulated pushdown automata of a reduced size. This result is established in terms of

(A) acceptance by final state,

(B) acceptance by empty pushdown, and

(C) acceptance by final state and empty pushdown.

5.1 Introduction

Chapter 4 has presented pushdown automata, whose moves are regulated by linear languages or, more simply, regulated pushdown automata, which characterise the family of recursively enumerable languages (see [54]). This chapter continues with the discussion of these automata. More specifically, it studies *one-turn regulated pushdown automata*.

To recall the concept of one-turn pushdown automata (see [30]), consider two consecutive moves made by a pushdown automaton, M. If during the first move M does not shorten its pushdown and during the second move it does, then M makes a turn during the second move. A pushdown automaton is *one-turn* if it makes no more than one turn with either of its pushdowns during any computation starting from an initial configuration. Recall that the one-turn pushdown automata characterise the family of linear languages (see [30]) while their unrestricted versions characterise the family of context-free languages. As a result, the one-turn pushdown automata are less powerful than the pushdown automata.

This chapter demonstrates that one-turn regulated pushdown automata characterise the family of recursively enumerable languages. Thus, as opposed to the ordinary one-turn pushdown automata, the one-turn regulated pushdown automata are as powerful as the regulated pushdown automata that can make any number of turns. In fact, this equivalence holds even for some restricted versions of one-turn regulated pushdown automata, including their atomic versions, which are sketched next.

During a move, an *atomic* one-turn regulated pushdown automaton changes a state and, in addition, performs exactly one of the following actions:

1. it pushes a symbol onto the pushdown
2. it pops a symbol from the pushdown
3. it reads an input symbol

This chapter proves that every recursively enumerable language is accepted by an atomic one-turn regulated pushdown automaton of a reduced size in terms of (A) acceptance by final state, (B) acceptance by empty pushdown, and (C) acceptance by final state and empty pushdown. Notice that this characterisation of the family of recursively enumerable languages can be seen as an automata-based counterpart to several grammatically based economical characterisations of this family (see, for instance, [60], [61], and [50]).

5.2 Preliminaries

We assume that the reader is familiar with language theory (see [53]). The notation can be found in Chapter 2 as 2.2.1, 2.2.2, 2.2.3, 2.2.4, 2.2.5, 2.2.6. The definitions can be found in Chapter 2 as 2.2.2, 2.2.4, 2.2.5, 2.2.8, 2.2.10, 2.2.11, and 2.2.12.

5.3 Definitions

This section defines the notion of a one-turn atomic pushdown automaton regulated by a linear language.

An *atomic pushdown automaton* is a 7-tuple, $M = (Q, \Sigma, \Omega, R, s, \$, F)$ (see 2.3.2).

Let Ψ be an alphabet of *rule labels* such that $card(\Psi) = card(R)$, and ψ be a bijection from R to Ψ. For simplicity, to express that ψ maps a rule, $Apa \rightarrow wq \in R$, to ρ, where $\rho \in \Psi$, this paper writes $\rho.Apa \rightarrow wq \in R$; in other words, $\rho.Apa \rightarrow wq$ means $\psi(Apa \rightarrow wq) = \rho$. A *configuration* of M, χ, is any word from $\{\$\}\Omega^*Q\Sigma^*$; χ is an initial configuration if $\chi = \$sw$, where $w \in \Sigma^*$. For every $x \in \Omega^*$, $y \in \Sigma^*$, and $\rho.Apa \rightarrow wq \in R$, M makes a move from configuration $\$xApay$ to configuration $\$xwqy$ according to ρ, written as $\$xApay \vdash \$xwqy$ $[\rho]$ or, more simply, $\$xApay \vdash \$xwqy$. Let χ be any configuration of M. M makes *zero moves from* χ *to* χ *according to* ε, symbolically written as $\chi \vdash^0 \chi$ $[\varepsilon]$. Let there exist a sequence of configurations $\chi_0, \chi_1, \dots, \chi_n$ for some $n \geq 1$ such that $\chi_{i-1} \vdash \chi_i$ $[\rho_i]$, where $\rho_i \in \Psi$, for $i = 1, \dots, n$, then M makes *n moves from χ_0 to χ_n according to $\rho_1 \dots \rho_n$*, symbolically written as $\chi_0 \vdash^n \chi_n$ $[\rho_1 \dots \rho_n]$ or, more simply, $\chi_0 \vdash^n \chi_n$. Define $\vdash^*$ and $\vdash^+$ in the standard manner.

Let $x, x', x'' \in \Omega^*$, $y, y', y'' \in \Sigma^*$, $q, q', q'' \in Q$, and $\$xqy \vdash \$x'q'y' \vdash \$x''q''y''$. If $|x| \leq |x'|$ and $|x'| > |x''|$, then $\$x'q'y' \vdash \$x''q''y''$ is a *turn*. If M makes no more than one turn during any sequence of moves starting from an initial configuration, then M is said to be *one-turn*.

Let Ξ be a *control language* over Ψ; that is, $\Xi \subseteq \Psi^*$. With Ξ, M defines the following three types of accepted languages:

$L(M, \Xi, 1)$—*the language accepted by final state*

$L(M, \Xi, 2)$—*the language accepted by empty pushdown*

$L(M, \Xi, 3)$—*the language accepted by final state and empty pushdown*

defined as follows. Let $\chi \in \{\$\}\Omega^* Q \Sigma^*$. If $\chi \in \{\$\}\Omega^* F$, $\chi \in \{\$\}Q$, $\chi \in \{\$\}F$, then χ is a *1-final configuration*, *2-final configuration*, *3-final configuration*, respectively. For $i = 1, 2, 3$, define $L(M, \Xi, i)$ as $L(M, \Xi, i) = \{w \mid w \in \Sigma^*$, and $\$sw \vdash^* \chi$ $[\sigma]$ in M for an $i-$final configuration, χ, and $\sigma \in \Xi\}$.

For any family of languages, X, and $i \in \{1, 2, 3\}$, set $\mathcal{L}(X, i) = \{L \mid L = L(M, \Xi, i)$, where M is a pushdown automaton and $\Xi \in X\}$. *RE* and *LIN* denote the families of recursively enumerable and linear languages, respectively.

5.4 Results

This section proves that the one-turn atomic pushdown automata regulated by linear languages characterise *RE*. In fact, these automata need no more than one state and two pushdown symbols to achieve this characterisation.

Lemma 5.4.1 *[54] For every left-extended queue grammar, K, there exists a left-extended queue grammar $Q = (V, \tau, W, F, s, P)$ satisfying $L(K) = L(Q)$, ! is a distinguished member of $(W - F)$, $V = U \cup Z \cup \tau$ such that U, Z, τ are pairwise disjoint, and Q derives every $z \in L(Q)$ in this way*

$$\begin{array}{lll}
\#S & \Rightarrow^+ & x\#b_1b_2\ldots b_n! \\
 & \Rightarrow & xb_1\#b_2\ldots b_ny_1p_2 \\
 & \Rightarrow & xb_1b_2\#b_3\ldots b_ny_1y_2p_3 \\
 & \vdots & \\
 & \Rightarrow & xb_1b_2\ldots b_{n-1}\#b_ny_1y_2\ldots y_{n-1}p_n \\
 & \Rightarrow & xb_1b_2\ldots b_{n-1}b_n\#y_1y_2\ldots y_np_{n+1}
\end{array}$$

where $n \in N$, $x \in U^*$, $b_i \in Z$ *for* $i = 1, \ldots, n$, $y_i \in \tau^*$ *for* $i = 1, \ldots, n$, $z = y_1y_2 \ldots y_n$, $p_i \in W - \{!\}$ *for* $i = 1, \ldots, n-1$, $p_n \in F$, *and in this derivation* $x\#b_1b_2 \ldots b_n!$ *is the only word containing* !.

□

Lemma 5.4.2 *Let* Q *be a left-extended queue grammar satisfying the properties of Lemma 5.4.1. Then, there is a linear grammar,* G*, and a one-turn atomic pushdown automaton* $M = (\{\xi\}, \tau, \{0,1\}, H, \xi, \$, \{\xi\})$ *such that* $card(H) = card(\tau) + 4$ *and* $L(Q) = L(M, L(G), 3)$.

Proof: Let $Q = (V, \tau, W, F, s, R)$ be a queue grammar satisfying the properties of Lemma 5.4.1. For some $n \geq 1$, introduce a homomorphism f from R to X, where $X = (\{1\}^*\{0\}\{1\}^*\{1\}^n \cap \{0,1\}^{2n})$. Extend f so it is defined from R^* to X^*. Define the substitution h from V^* to X^* as $h(a) = \{f(r) \mid r = (a,p,x,q) \in R$ for some $p, q \in W$, $x \in V^*\}$. Define the coding d from $\{0,1\}^*$ to $\{2,3\}^*$ as $d(0) = 2$, $d(1) = 3$. Construct the linear grammar $G = (N, T, P, S)$ as follows. Set

$$T = \{0,1,2,3\} \cup \tau$$

$$N = \{S\} \cup \{\tilde{q} \mid q \in W\} \cup \{\hat{q} \mid q \in W\}$$

$$P = \{S \rightarrow \tilde{f} \mid f \in F\} \cup \{\tilde{!} \rightarrow \hat{!}\}$$

Extend P by performing 1 through 3 given next.

1. for every $r = (a,p,x,q) \in R$, $p, q \in W$, $x \in T^*$: $P = P \cup \{\tilde{q} \rightarrow \tilde{p}d(f(r))x\}$
2. for every $(a,p,x,q) \in R$: $P = P \cup \{\hat{q} \rightarrow y\hat{p}b \mid y \in rev(h(x)), b \in h(a)\}$
3. for every $(a,p,x,q) \in R$, $ap = S$, $p, q \in W$, $x \in V^*$: $P = P \cup \{\hat{q} \rightarrow y \mid y \in rev(h(x))\}$

Define the pushdown automaton $M = (\{\xi\}, \tau, \{0,1\}, H, \xi, \$, \{\xi\})$, where H contains the next transition rules:

0. $\xi \rightarrow 0\xi$
1. $\xi \rightarrow 1\xi$

2. $0\xi \rightarrow \xi$

3. $1\xi \rightarrow \xi$

a. $\xi a \rightarrow \xi$ for every $a \in \tau$

We next demonstrate that $L(M, L(G), 3) = L(Q)$.

To demonstrate $L(M, L(G), 3) = L(Q)$, observe that M accepts every word w as

$$\begin{array}{lll}
\$\xi w_1 \dots w_{m-1} w_m & \vdash^+ & \$\bar{b}_m \dots \bar{b}_1 \bar{a}_n \dots \bar{a}_1 \xi w_1 \dots w_{m-1} w_m \\
 & \vdash^n & \$\bar{b}_m \dots \bar{b}_1 \xi w_1 \dots w_{m-1} w_m \\
 & \vdash^{|w_1|} & \$\bar{b}_m \dots \bar{b}_1 \xi w_2 \dots w_{m-1} w_m \\
 & \vdash & \$\bar{b}_m \dots \bar{b}_2 \xi w_2 \dots w_{m-1} w_m \\
 & \vdash^{|w_2|} & \$\bar{b}_m \dots \bar{b}_2 \xi w_3 \dots w_{m-1} w_m \\
 & \vdash & \$\bar{b}_m \dots \bar{b}_3 \xi w_3 \dots w_{m-1} w_m \\
 & \vdots & \\
 & \vdash & \$\bar{b}_m \xi w_m \\
 & \vdash^{|w_m|} & \$\bar{b}_m \xi \\
 & \vdash & \$\xi
\end{array}$$

according to a word of the form $\beta\alpha\alpha'\beta' \in L(G)$ where

$$\begin{array}{lll}
\beta & = & rev(f(r_m))rev(f(r_{m-1})) \dots rev(f(r_1)), \\
\alpha & = & rev(f(t_n))rev(f(t_{n-1})) \dots rev(f(t_1)), \\
\alpha' & = & f(t_0)f(t_1) \dots f(t_n), \\
\beta' & = & d(f(r_1))w_1 d(f(r_2))w_2 \dots d(f(r_m))w_m,
\end{array}$$

for some $m, n \geq 1$ so that

for $i = 1, \dots, m$,
$t_i \;=\; (b_i, q_i, w_i, q_{i+1}) \in R$, $b_i \in V - \tau$, $q_i, q_{i+1} \in Q$, $\bar{b}_i = f(t_i)$

for $j = 1, \dots, n+1$,
$r_j \;=\; (a_{j-1}, p_{j-1}, x_j, p_j)$, $a_{j-1} \in V - \tau$, $p_{j-1}, p_j \in Q - F$, $x_j \in (V - \tau)^*$,
$\bar{a}_j = f(r_j)$, $q_{m+1} \in F$, $\bar{a}_0 p_0 = s$

Thus, in Q,

$$\begin{array}{lll}
\#a_0p_0 & \Rightarrow a_0\#y_0x_1p_1 & [(a_0, p_0, x_1, p_1)] \\
 & \Rightarrow a_0a_1\#y_1x_2p_2 & [(a_1, p_1, x_2, p_2)] \\
 & \Rightarrow a_0a_1a_2\#y_2x_3p_3 & [(a_2, p_2, x_3, p_3)] \\
 & \vdots &
\end{array}$$

$$
\begin{array}{ll}
\Rightarrow a_0a_1a_2\dots a_{n-1}\#y_{n-1}x_np_n & [(a_{n-1},p_{n-1},x_n,p_n)] \\
\Rightarrow a_0a_1a_2\dots a_n\#y_nx_{n+1}q_1 & [(a_n,p_n,x_{n+1},q_1)] \\
\Rightarrow a_0\dots a_nb_1\#b_2\dots b_mw_1q_2 & [(b_1,q_1,w_1,q_2)] \\
\Rightarrow a_0\dots a_nb_1b_2\#b_3\dots b_mw_1w_2q_3 & [(b_2,q_2,w_2,q_3)] \\
\vdots & \\
\Rightarrow a_0\dots a_nb_1\dots b_{m-1}\#b_mw_1w_2\dots w_{m-1}q_m & [(b_{m-1},q_{m-1},w_{m-1},q_m)] \\
\Rightarrow a_0\dots a_nb_1\dots b_m\#w_1w_2\dots w_mq_{m+1} & [(b_m,q_m,w_m,q_{m+1})]
\end{array}
$$

Therefore, $w_1w_2\dots w_m \in L(Q)$. Consequently, $L(M,L(G),3) \subseteq L(Q)$.

A proof that $L(Q) \subseteq L(M,L(G),3)$ is left to the reader.

As $L(Q) \subseteq L(M,L(G),3)$ and $L(M,L(G),3) \subseteq L(Q)$, $L(Q) = L(M,L(G),3)$. Observe that M is atomic and one-turn. Furthermore, $card(H) = card(\tau) + 4$. Thus, Lemma 5.4.2 holds. ☐

Corollary 5.4.1 *For every $L \in RE$ and every $i = 1,2,3$, there is a linear language Ξ, and a one-turn atomic pushdown automaton, $M = (Q,\Sigma,\Omega,R,s,\$,F)$ such that $card(Q) \leq 1$, $card(\Omega) \leq 2$, $card(R) \leq card(\Sigma)+4$, and $L(M,\Xi,i) = L$.*

Proof: By Theorem 2.1 in [44], for every $L \in RE$, there is a queue grammar Q such that $L = L(Q)$. Clearly, there is a left-extended queue grammar, Q', such that $L(Q) = L(Q')$. Thus, for $i = 3$ this corollary follows from Lemmas 5.4.1 and 5.4.2. Analogically, prove this corollary for $i = 1,2$. ☐

Corollary 5.4.2 *For $i \in \{1,2,3\}$, $RE = \mathcal{L}(LIN,i)$.*

Proof: This theorem follows from Corollary 5.4.1. ☐

Chapter 6

Bounded Deterministic RPA

This chapter presents a possible definition of the bounded deterministic regulated pushdown automata, which are regulated by some control languages. What is more important, it demonstrates the equivalence of this automata with bounded deterministic Turing machine. The determinism and power equal to bounded deterministic Turing machine is demonstrated by use of control languages belonging to a set of all context-free languages (*CF*) though. Thus, the question of powerful deterministic regulated pushdown automata regulated by language coming from *LIN* remains open.

6.1 Introduction

Chapter 4 has presented pushdown automata, whose moves are regulated by linear languages or, more simply, regulated pushdown automata, which characterise the family of recursively enumerable languages. Chapter 5 has presented atomic one-turn regulated pushdown automata, whose moves are also regulated by linear languages, which characterise the family of recursively enumerable languages (see [55]).

This chapter demonstrates the definition of determinism in the area of regulated pushdown automata. Moreover, it demonstrates that linear-bounded deterministic pushdown automata (DRPA) regulated by context-free languages are of the same power as linear-bounded deterministic Turing machine (DTM). It is demonstrated that every linear-bounded DRPA regulated by *CF* languages is equal to linear-bounded DTM where the DRPA successfully finishes its work by reaching the final state leaving on the pushdown content representing the one obtained by simulated DTM on its tape.

6.2 Preliminaries

We assume that the reader is familiar with language theory (see [53]) and theory of automata (see [53]). The necessary definitions can be found in Chapter 2 as 2.2.7, 2.2.8, 2.2.10, 2.3.1, 2.3.2.

Moreover, we define a deterministic Turing machine this way [52]:

Definition 6.2.1 *A* (deterministic) Turing machine *(DTM) is a 5-tuple* $T = (Q, \Sigma, \Gamma, q_0, \delta)$*, where*

- Q *is a finite set of* states*, the* halt *state (h) is assumed not to be in* Q*,* $h \notin Q$*;*
- Σ*, the* input alphabet*, is a set of symbols;*
- Γ*, the* tape alphabet*, is a finite set with* $\Sigma \subseteq \Gamma$*;* Γ *is assumed not to contain* Δ*, the* blank *symbol;*
- $q_0 \in Q$ *(the initial state); and*
- δ *is a partial function from* $Q \times (\Gamma \cup \{\Delta\})$ *to* $(Q \cup \{h\}) \times (\Gamma \cup \{\Delta\}) \times \{R, L, S\}$*.*

Next we define a configuration of the DTM and language accepted by DTM to make a complete set of definitions binding DTM with language theory:

Definition 6.2.2 *A* configuration *of the DTM is a pair* $(Q, x\underline{a}y)$*, where* q *is a state,* $x, y \in (\Gamma \cup \{\Delta\})^*$*,* $a \in \Gamma \cup \{\Delta\}$*, and the underlined symbol represents the current position of the* head*, which allows reading from and writing to a tape one symbol to the square of current position and which can possibly stay (S) in the same position, move right (R), or move left (L). We say*

$$(q, x\underline{a}y) \vdash_T (r, z\underline{b}w)$$

if T *passes from the configuration on the left to that on the right in one move, and*

$$(q, x\underline{a}y) \vdash_T^* (r, z\underline{b}w)$$

if T *passes from the first configuration in zero or more moves.*

Definition 6.2.3 *An input string* $x \in \Sigma^*$ *is* accepted *by* T *if starting* T *with input* x *leads eventually to halting configuration. In other words,* x *is accepted if for some strings* y *and* z *in* $(\Gamma \cup \{\Delta\})^*$ *and some* $a \in \Gamma \cup \{\Delta\}$*,*

$$(q_0, \underline{\Delta}x) \vdash_T^* (h, y\underline{a}z)$$

In this situation we say T halts on input x*. The* language accepted by T *is the set of input strings that are accepted by* T*. For further details see [52].*

The last, but not least, definition presented in this place presents a definition of linear-bounded deterministic Turing machine.

Definition 6.2.4 *A* linear-bounded deterministic Turing machine *(LBDTM) is a 5-tuple* $T' = (Q, \Sigma, \Gamma, q_0, \delta)$ *that is the same as a (deterministic) Turing machine except in the following respect: there are two extra tape symbols* $\langle$ *and* $\rangle$*, assumed not to be elements of* Γ*;* M *begins in the configuration* $(q_0, \langle x \rangle)$*, where* x *is the input string, and* M *is not permitted to replace the symbols* $\langle$ *or* $\rangle$*, or to move its head left of the square with* $\langle$ *or right of the square with* $\rangle$*.*

See [52] for further details on linear-bounded automaton.

6.3 Definitions

This section defines the notion of a deterministic regulated pushdown automaton regulated by a context-free language. According to Chapter 4 we only give extension to define the deterministic behaviour of the RPA.

Let us extend Definition 2.3.1 and Definition 2.3.2 in such a way, so that we can obtain a deterministic atomic regulated pushdown automata:

Definition 6.3.1 *As a basis, we refer to Definition 2.3.2 and we add regulation and determinism here.*

Regulation *(it is defined the same way as in Chapter 4)*
Let Ψ *be an alphabet of* rule labels *such that* $card(\Psi) = card(R)$*, and* ψ *be a bijection from* R *to* Ψ*. For simplicity, to express that* ψ *maps a rule,* $Apa \rightarrow wq \in R$*, to* ρ*, where* $\rho \in \Psi$*, this paper writes* $\rho.Apa \rightarrow wq \in R$*; in other words,* $\rho.Apa \rightarrow wq$ *means* $\psi(Apa \rightarrow wq) = \rho$*. A* configuration *of* M*,* χ*, is any word from* $\{\$\}\Omega^*Q\Sigma^*$*;* χ *is an initial configuration if* $\chi = \$sw$*, where* $w \in \Sigma^*$*. For every* $x \in \Omega^*$*,* $y \in \Sigma^*$*, and* $\rho.Apa \rightarrow wq \in R$*,* M *makes a move from configuration* $\$xApay$ *to configuration* $\$xwqy$ *according to* ρ*, written as* $\$xApay \vdash \$xwqy\ [\rho]$ *or, more simply,* $\$xApay \vdash \$xwqy$*. Let* χ *be any configuration of* M*.* M *makes* zero moves from χ to χ according to ε*, symbolically written as* $\chi \vdash^0 \chi\ [\varepsilon]$*. Let there exist a sequence of configurations* $\chi_0, \chi_1, \dots, \chi_n$ *for some* $n \geq 1$ *such that* $\chi_{i-1} \vdash \chi_i\ [\rho_i]$*, where* $\rho_i \in \Psi$*, for* $i = 1, \dots, n$*, then* M *makes* n *moves from* χ_0 *to* χ_n *according to* $\rho_1 \dots \rho_n$*, symbolically written as* $\chi_0 \vdash^n \chi_n\ [\rho_1 \dots \rho_n]$ *or, more simply,* $\chi_0 \vdash^n \chi_n$*. Define* $\vdash^*$ *and* $\vdash^+$ *in the standard manner.*

Control Language *(it is defined a similar way as in Chapter 4, context-free languages are used instead of the linear languages to control the operation of automaton)*
Let Ξ *be a* control language *over* Ψ*; that is,* $\Xi \subseteq \Psi^*$*. Let* Ξ *be from the family*

of context-free languages (CF). With Ξ*, M defines the following three types of accepted languages:*

$L(M, \Xi, 1)$—the language accepted by final state

$L(M, \Xi, 2)$—the language accepted by empty pushdown

$L(M, \Xi, 3)$—the language accepted by final state and empty pushdown

defined as follows. Let $\chi \in \{\$\}\Omega^* Q\Sigma^*$*. If* $\chi \in \{\$\}\Omega^* F$*,* $\chi \in \{\$\}Q$*,* $\chi \in \{\$\}F$*, then* χ *is a 1-final configuration, 2-final configuration, 3-final configuration, respectively. For* $i = 1, 2, 3$*, define* $L(M, \Xi, i)$ *as* $L(M, \Xi, i) = \{w \mid w \in \Sigma^*, \text{ and } \$sw \Rightarrow^* \chi\ [\sigma] \text{ in } M \text{ for an } i\text{–final configuration, } \chi, \text{ and } \sigma \in \Xi\}$.

Determinism *(it is a newly introduced notion)*
An RPA is deterministic *(DRPA) if being in a state q,* $q \in Q$*, the appropriate action which should be performed can always be deterministically selected. This can only be due to the following two circumstances:*

1. *For the given state, there is only one rule* $r \in R$ *that is applicable in a given situation (state, symbol on the top of the pushdown or under the reading head) and, moreover, control language admits such a rule.*

2. *If there are more than one rules that are applicable in a given situation then the rule can be deterministically denoted according to the actual context of sentential form of the control language applicable to the current state of operation performed by RPA.*

Informally, the first item describes a situation where DRPA behaves as a common (non-regulated) deterministic PA and the control language just "checks" the correctness of the work. The second case describes a situation where there is a non-determinism on the level of PA and thus the appropriate action must be selected on the level of the control language and the decision can be done due to the actual context of the sentential form/derivation represented by actions performed by DRPA till this point of analysis.

Next we define boundedness of the (deterministic) RPA by limiting a size of the pushdown:

Definition 6.3.2 *A (deterministic) RPA is* linear-bounded *(LB) if the maximal number of elements stored on the pushdown at a time is less then* $kn + c$*, where n is the length of the input string on the tape and* $k, n \in \mathcal{I}$*.*

6.4 Results

Theorem 6.4.1 *For every LBDTM, there exists an atomic DRPA, $L(M, \Xi, i)$, $i \in \{1, 2, 3\}$, which exactly simulates behaviour of LBDTM. (We say they are equivalent—one can be replaced by another.)*
Note: $\Xi = CF$

Proof:

Construction: Let $T' = (Q, \Sigma, \Gamma, q_0, \delta)$ is a LBDTM. The equivalent atomic DRPA $M = (Q', \Sigma', \Omega, R, s, \nabla, F)$ can be constructed in the following way:

1. $\Sigma' = \Sigma \cup \{\langle, \rangle\}$, where it is clear that $\langle, \rangle \notin \Sigma$

2. $\Omega = \Gamma \cup \{\Delta, \langle, \rangle, \nabla\} \cup \{\bar{a} \mid a \in \Sigma \cup \{\Delta, \rangle\}\}$, where $\Delta, \langle, \rangle, \nabla, \bar{a} \notin \Gamma$

3. $$\begin{aligned} Q' = Q &\cup \{h, Fin, S_\nabla, S_0, S_1, S_2, S_3, S_4\} \cup \{q' \mid q \in Q\} \cup \\ &\cup \{\underline{Rq^1} \mid q \in Q\} \cup \{\underline{Rq^2} \mid q \in Q\} \cup \{\underline{Rq^3} \mid q \in Q\} \cup \\ &\cup \{\underline{Aqa} \mid q \in Q,\ a \in \Sigma,\ \delta(q,a) \text{ is defined }\} \cup \\ &\cup \{\underline{Lqa} \mid q, q' \in Q,\ a, a' \in \Sigma,\ \delta(q,a) = (q', a', L)\} \cup \\ &\cup \{\underline{Rqa} \mid q, q' \in Q,\ a, a' \in \Sigma,\ \delta(q,a) = (q', a', R)\} \cup \\ &\cup \{\underline{Sqa} \mid q, q' \in Q,\ a, a' \in \Sigma,\ \delta(q,a) = (q', a', S)\}, \end{aligned}$$
   where all of the $h, Fin, S_\nabla, S_0, S_1, S_2, S_3, S_4$, q', $\underline{Rq^1}$, $\underline{Rq^2}$, $\underline{Rq^3}$, $\underline{Aqa}$, $\underline{Lqa}$, $\underline{Rqa}$, $\underline{Sqa}$ are not in Q

4. $F = \{Fin\}$

5. $s = S_0$

6. $R = R_s \cup (\cup_{q \in Q} R_q) \cup R_f$, where
   $$\begin{aligned} R_s = \{&S_0\langle \rightarrow S_1,\ S_1 \rightarrow \langle S_2,\ S_2\rangle \rightarrow S_4,\ S_4 \rightarrow \bar{\rangle} q_0\} \cup \\ &\cup \{S_2 a \rightarrow S_3 \mid a \in \Sigma\} \cup \\ &\cup \{S_3 \rightarrow \bar{a} S_2 \mid \bar{a} \in \Omega\} \end{aligned}$$
   $$R_f = \{ah \rightarrow h \mid a \in \Gamma\} \cup \{\nabla h \rightarrow Fin\}$$
   $$\begin{aligned} \forall q \in Q : R_q = \{&q \rightarrow \bar{a}q \mid \bar{a} \in \Omega\} \cup \\ &\cup \{\bar{a}q \rightarrow q' \mid \bar{a} \in \Omega,\ \delta(q,a) \text{ is defined for some } a \in \Sigma\} \cup \\ &\cup \{\rangle q \rightarrow q' \mid \delta(q,a) \text{ is defined for some } a \in \Sigma\} \cup \\ &\cup \{\bar{a}q' \rightarrow \underline{Rq^1} \mid \bar{a} \in \Omega,\ \delta(q,a) \text{ is defined for some } a \in \Sigma\} \cup \\ &\cup \{\underline{Rq^1} \rightarrow \bar{a}\underline{Rq^2} \mid \bar{a} \in \Omega,\ \delta(q,a) \text{ is defined for some } a \in \Sigma\} \cup \\ &\cup \{\underline{Rq^2} \rightarrow \bar{a}\underline{Rq^3} \mid \bar{a} \in \Omega,\ \delta(q,a) \text{ is defined for some } a \in \Sigma\} \cup \\ &\cup \{\bar{a}\underline{Rq^3} \rightarrow q \mid \bar{a} \in \Omega,\ \delta(q,a) \text{ is defined for some } a \in \Sigma\} \cup \\ &\cup \{aq' \rightarrow \underline{Aqa} \mid a \in \Sigma,\ \delta(q,a) \text{ is defined }\} \cup \\ &\cup \{\underline{Aqa} \rightarrow \bar{c}\underline{Lqa} \mid a, c \in \Sigma,\ p \in Q,\ \delta(q,a) = (p, c, L)\} \cup \\ &\cup \{\underline{Lqa} \rightarrow \bar{b}p \mid a, c \in \Sigma,\ b \in \Omega,\ p \in Q,\ \delta(q,a) = (p, c, L)\} \cup \\ &\cup \{\underline{Aqa} \rightarrow c\underline{Rqa} \mid a, c \in \Sigma,\ p \in Q,\ \delta(q,a) = (p, c, R)\} \cup \end{aligned}$$

$\cup \; \{\underline{Rqa} \rightarrow bp \mid a, b, c \in \Sigma,\ p \in Q,\ \delta(q, a) = (p, c, R)\} \cup$
$\cup \; \{\underline{Aqa} \rightarrow c\underline{Sqa} \mid a, c \in \Sigma,\ p \in Q,\ \delta(q, a) = (p, c, S)\} \cup$
$\cup \; \{\underline{Sqa} \rightarrow \bar{b}p \mid a, c \in \Sigma,\ b \in \Omega,\ p \in Q,\ \delta(q, a) = (p, c, S)\} \cup$
$\cup \; \{q' \rightarrow \rangle p \mid p \in Q,\ \delta(q, \rangle) = (p, \rangle, S)\} \cup$
$\cup \; \{q' \rightarrow \bar{\rangle} p \mid p \in Q,\ \delta(q, \rangle) = (p, \rangle, L)\}$

Note for rules containing $\rangle$*: Rules in* δ *may not be other than presented, as DLBA is not allowed to move right of the marker* $\rangle$ *position nor change it.*

A control language, Ξ, which is a context-free one, is defined for the equivalent atomic DRPA M by the following grammar $G = (N, T, S, P)$:

1. $N = \{K, L, M, O, P\}$

2. $T = \{< r > \ \mid r \in R\}$

3. $S = K$

4. P contains the following derivation rules:
$K \rightarrow < S_0\langle \rightarrow S_1 >< S_1 \rightarrow \langle S_2 > L$
$L \rightarrow < S_2 a \rightarrow S_3 >< S_3 \rightarrow \bar{a} S_2 > L,\ \forall a \in \Sigma,\ \bar{a}$ derived from a
$L \rightarrow < S_2\rangle \rightarrow S_4 >< S_4 \rightarrow \bar{\rangle} q_0 > M$
$M \rightarrow OM,$ *Note: the only non-linear grammar rule*
$M \rightarrow P$
$P \rightarrow < ah \rightarrow h > P,\ \forall a \in \Omega$
$P \rightarrow < \nabla h \rightarrow Fin >$
If there is defined $\delta(q, a) = (p, c, L), a \neq \rangle$ then $\forall \bar{a}_0, \bar{a}_1 \in \Omega$:
$O \rightarrow < \bar{a}_0 q \rightarrow q' >< \bar{a}_1 q' \rightarrow \underline{Rq^1} >< \underline{Rq^1} \rightarrow \bar{a}_1 \underline{Rq^2} >$
$< \underline{Rq^2} \rightarrow \bar{a}_0 \underline{Rq^3} >< \bar{a}_0 \underline{Rq^3} \rightarrow q > O < p \rightarrow \bar{a}_0 p >$
$O \rightarrow < \bar{a}_0 q \rightarrow q' >< aq' \rightarrow \underline{Aqa} >< \underline{Aqa} \rightarrow \bar{c}\underline{Lqa} >< \underline{Lqa} \rightarrow \bar{a}_0 p >$
If there is defined $\delta(q, a) = (p, c, S), a \neq \rangle$ then $\forall \bar{a}_0, \bar{a}_1 \in \Omega$:
$O \rightarrow < \bar{a}_0 q \rightarrow q' >< \bar{a}_1 q' \rightarrow \underline{Rq^1} >< \underline{Rq^1} \rightarrow \bar{a}_1 \underline{Rq^2} >$
$< \underline{Rq^2} \rightarrow \bar{a}_0 \underline{Rq^3} >< \bar{a}_0 \underline{Rq^3} \rightarrow q > O < p \rightarrow \bar{a}_0 p >$
$O \rightarrow < \bar{a}_0 q \rightarrow q' >< aq' \rightarrow \underline{Aqa} >< \underline{Aqa} \rightarrow c\underline{Sqa} >< \underline{Sqa} \rightarrow \bar{a}_0 p >$
If there is defined $\delta(q, a) = (p, c, R), a \neq \rangle$ then $\forall \bar{a}_0, \bar{a}_1 \in \Omega$:
$O \rightarrow < \bar{a}_0 q \rightarrow q' >< \bar{a}_1 q' \rightarrow \underline{Rq^1} >< \underline{Rq^1} \rightarrow \bar{a}_1 \underline{Rq^2} >$
$< \underline{Rq^2} \rightarrow \bar{a}_0 \underline{Rq^3} >< \bar{a}_0 \underline{Rq^3} \rightarrow q > O < p \rightarrow \bar{a}_0 p >$
$O \rightarrow < \bar{a}_0 q \rightarrow q' >< aq' \rightarrow \underline{Aqa} >< \underline{Aqa} \rightarrow c\underline{Rqa} >< \underline{Rqa} \rightarrow a_0 p >$
If there is defined $\delta(q, \rangle) = (p, \rangle, L)$ ($\rangle$ cannot be modified) then:
$O \rightarrow <\rangle q \rightarrow q' >< q' \rightarrow \bar{\rangle} p >$
If there is defined $\delta(q, \rangle) = (p, \rangle, S)$ ($\rangle$ cannot be modified) then:
$O \rightarrow <\rangle q \rightarrow q' >< q' \rightarrow\rangle p >$

Proof $M \subseteq T'$:
The key idea of the proof is introduced informally. Formal way of proof is left to the reader.

The automaton M starts work by copying input string on the pushdown in such a way so that symbols a lying right of the position of the reading head are encoded as $\bar{a}$. This is done in a deterministic way on the level of PA and the control language just "follows" the work of the automaton.

Every step of the LBDTM is then simulated by popping all the symbols lying right of the simulated position of the reading head, changing the symbol on the top of the pushdown according to the action defined by LBDTM, and pushing the symbols back. While pushing the symbols back de/encoding is performed to simulate reading head position change.

The determinism of DRPA is given by the determinism of LBDTM as the only non-deterministic rules of the automaton (i.e. those that need to be made deterministic by the control language) are those that perform pushing (after popping) the symbols when access to the reading head is simulated. And this issue is sufficiently handled by the control language (exploiting the so called "bracketed structures" of the language).

The automaton finishes its work when it is in state h, which is the final state of the LBDTM. In this state, the pushdown is cleared and while removing the symbol ∇ the atomic DRPA M moves to its only final state *Fin*. As rules of LBDTM do not work with ∇, moreover, LBDTM cannot move out nor change pair $\langle$ and $\rangle$, being in the state *Fin* it is the only state when the pushdown is empty and together it is a final state. Thus such an automaton accepts language by empty pushdown, final state, and final state and empty pushdown.

Proof $T' \subseteq M$ is left to the reader. *Idea*: We can describe the DRPA easily by two-tape TM. It can be re-coded to one-tape TM. The description of such a TM, together with the input, can be used as an input for Universal TM. As the DRPA does not go beyond the bottom of the pushdown and beyond marker on the top of the pushdown (input is copied to the pushdown), the Universal TM is also bounded during its behaviour. □

Theorem 6.4.2 *Every DRPA from Theorem 6.4.1 requires exactly* $n+1$ *cells on the pushdown, where* n *is the length of the input string including symbols* $\langle$ *and* $\rangle$*. Thus, such automata are linear-bounded. That means that LBDTM is equal in power to LB atomic DRPA.*

Proof: This theorem follows directly from Theorem 6.4.1 as the atomic DRPA pushes on the pushdown one extra symbol at the beginning of the work and then it copies input on the pushdown and it does not add any other symbol during work. It always removes and pushes back exactly the same number of symbols. □

Theorem 6.4.3 *Every* $L \in CS$ *is accepted by LB atomic RPA.*

Proof: This theorem follows directly from Theorems 6.4.1, 6.4.2 and Theorem 19.5 from [52], which proves that every $L \in CS$ is accepted by LB TM.

The modification of deterministic LB atomic RPA to a non-deterministic one is straightforward. ☐

6.5 Chapter Summary and Open Problems

This chapter introduced a deterministic version of RPA. Nevertheless, a linear-bounded version of both deterministic and non-deterministic version of modified RPA was studied. The modification used context-free languages to control the RPA. It has been demonstrated that the (deterministic) RPA controlled by a context-free language can simulate any (deterministic) linear-bounded Turing machine. Moreover, the RPA was linear-bounded too.

The open questions remain, whether:

1. It can be defined a linear-bounded (deterministic) RPA controlled by the linear language having the same features as the one controlled by the context-free language presented in this chapter.
2. If omitted linear-boundedness, the power of deterministic RPA increases to the power of deterministic Turing machine (which is the power of Turing machine, in general).

Chapter 7

Usage of DRPA for Syntactic Analysis

This chapter utilises investigation performed in the previous chapters. We will define a new kind of grammars (inspired by LL grammars and *scattered context grammars* [29]) the power of which is higher than that of context-free languages. We use these grammars to describe deterministic RPA controlled by context-free languages. Such automata will be able to parse sentences of language generated by such grammars. The word "parse" is used in the traditional meaning known from compiler theory—it means to decide whether the sentence is or is not part of the language and to detect more or less exactly the place of syntactic error in the input sentence.

7.1 Introduction

Chapters 2 and 3 demonstrated how parsing of LL_k languages (for $k > 1$) can be done (efficiently). Nevertheless, usage of LALR or LL_1 grammars is mostly in the focus of today's parser designers (see [12, 3]). Moreover, the LL_1 grammars are becoming more and more popular thanks to the recursive descent parsing technique. Nevertheless, the trend may be changed if grammar construction remains simple while the power of grammar increases beyond the *CF* languages. A good tip, the scattered context grammars may be so.

Chapters 4, 5, and 6 present the concept of regulated pushdown automata. It has been presented that their power is on the level of Turing machine. A possible exploitation of the RPA for language analysis may be a way of introducing grammars with a higher descriptive power than that of *CF* languages in the area of programming languages.

This chapter presents the utilisation of several concepts:

- scattered context grammars,
- regulated pushdown automata, and

- parsing of LL languages.

These concepts are combined into one form, which enables the efficient deterministic parsing of languages described by grammars with a descriptive power greater than the one of context-free grammars.

7.2 Preliminaries

Even if the concept of LL_k grammars holds generally for any $k > 0$ we present another view on LL_1 grammars, which is much simpler and which will be used below in this chapter. A detailed view on the topic can be found in [3, 4, 5, 12, 52].

First of all, we define a simpler version of set $\mathit{FIRST} = \mathit{FIRST}_1$, just for LL_1 languages:

Definition 7.2.1 *Let* $G = (N, T, P, S)$ *is a context-free grammar,* $\alpha \in (N \cup T)^*$.

$$\mathit{FIRST}(\alpha) ::= \{a \in T \mid \alpha \Rightarrow^* a\beta, \beta \in (N \cup T)^*\} \cup \{\varepsilon \mid \alpha \Rightarrow^* \varepsilon\}$$

Next, we have to define a set FOLLOW, which is new to this thesis, nevertheless a well known term in formal language theory:

Definition 7.2.2 *Let* $G = (N, T, P, S)$ *is a context-free grammar,* $A \in N$.

$$\mathit{FOLLOW}(A) ::= \{a \in T \mid S \Rightarrow^* \alpha A \beta, a \in \mathit{FIRST}(\beta),\ \alpha, \beta \in (N \cup T)^*\}$$

We write FIRST_P or FOLLOW_P to denote a particular set of production rules.

Based on the previous two definitions, we can establish two conditions, which can help us to verify whether a grammar is LL_1 or not.

Definition 7.2.3 Condition FF *holds if for every set of production rules:*

$$A \to \alpha_1 \mid \alpha_2 \mid \ \ldots \ \mid \alpha_k \in P$$

from context-free grammar, $G = (N, T, S, P)$*, it is satisfied:*

$$\mathit{FIRST}(\alpha_i) \cap \mathit{FIRST}(\alpha_j) = \emptyset,\ \forall i \neq j, 1 \leq i, j \leq k$$

Definition 7.2.4 Condition FFL *holds if for every set of production rules*

$$A \to \alpha_1 \mid \alpha_2 \mid \ \ldots \ \mid \alpha_k \in P$$

such that

$$\exists i, 1 \leq i \leq k : \alpha_i \Rightarrow^* \varepsilon$$

from context-free grammar, $G = (N, T, S, P)$*, it is satisfied:*

$$\mathit{FIRST}(\alpha_j) \cap \mathit{FOLLOW}(A) = \emptyset, \forall i \neq j, 1 \leq i, j \leq k$$

Finally, we can have an alternative to define an LL_1 grammar:

Definition 7.2.5 *A context-free grammar, G, is LL_1 grammar if conditions FF and FFL are satisfied for the G.*

Scattered context grammars (see [29]) represent an alternative to context-sensitive and non-restricted grammars. Nevertheless, their power can be seen in the usage of production rules known from context-free grammars:

Definition 7.2.6 *A* scattered context grammar*, G, is a quadruple (V, T, P, S), where V is a finite set of symbols, $T \subset V$, $S \in V \backslash T$, and P is a set of production rules of the form $(A_1, \ldots, A_n) \to (w_1, \ldots, w_n), n \geq 1$, $\forall A_i : A_i \in V \setminus T$, $\forall w_i : w_i \in V^+$.*

Definition 7.2.7 *A* reducing *scattered context grammar (SCG), G, is a quadruple (V, T, P, S), where V is a finite set of symbols, $T \subset V$, $S \in V \backslash T$, and P is a set of production rules of the form $(A_1, \ldots, A_n) \to (w_1, \ldots, w_n), n \geq 1$, $\forall A_i : A_i \in V \setminus T$, $\forall w_i : w_i \in V^*$.*

Definition 7.2.8 *Let $G = (V, T, P, S)$ be a SCG. Let $(A_1, \ldots, A_n) \to (w_1, \ldots, w_n)$ be in P and for $1 \leq i \leq n+1$, let $x_i \in V^*$. We write*

$$x_1 A_1 x_2 A_2 \ldots x_n A_n x_{n+1} \Rightarrow x_1 w_1 x_2 w_2 \ldots x_n w_n x_{n+1}$$

Let $\Rightarrow^*$ be a reflexive transitive closure of $\Rightarrow$.

The language generated by G is defined as $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$.
Note: If the SCG is non-reducing, then $w \in T^+$.

Definitions of regulated pushdown automata and their deterministic versions can be found in Chapter 4 and Chapter 6.

7.3 Definitions

The notion of the chapter aims at LL grammars. Therefore, we start with the definition of the "LL notion" for the SCG. First of all, we bring a definition of left derivation to the SCG.

Definition 7.3.1 *Let $G = (V, T, P, S)$ be a SCG. Let G be a* left derivating SCG *then for $(A_1, \ldots, A_n) \to (w_1, \ldots, w_n) \in P$ we write $x_1 A_1 x_2 A_2 \ldots x_n A_n x_{n+1} \Rightarrow x_1 w_1 x_2 w_2 \ldots x_n w_n x_{n+1}, x_i \in V^*$, if $x_1 \neq y_1 A_1 z_1$, $x_2 \neq y_2 A_2 z_2$, $\ldots$, $x_n \neq y_n A_n z_n$, $y_i, z_i \in V^*$.*

Next, we define sets *FIRST′* and *FOLLOW′*. Their definition is the same as for context-free grammar sets *FIRST* or *FOLLOW*. The difference is that the context-free production rules are constructed from the SCG ones by composing appropriate pairs of non-terminals on the left-hand side of the tuple-rule with a corresponding string of symbols on the right-hand side of the tuple-rule from SCG.

Definition 7.3.2 *Let $G = (V, T, P, S)$ is a SCG, $\alpha \in V^*$.*

$$FIRST'(\alpha) ::= FIRST_{P'}(\alpha)$$

where context-free grammar rules P' are constructed from the set P in such a way so that $A \to \beta \in P'$ if $(A_1, \ldots, A, \ldots, A_n) \to (w_1, \ldots, \beta, \ldots w_n) \in P$ for any n and any position of pair A, β in the rule-tuple from P.

Definition 7.3.3 *Let $G = (V, T, P, S)$ is a SCG, $A \in V \setminus T$.*

$$FOLLOW'(A) ::= FOLLOW_{P'}(A)$$

where context-free grammar rules P' are constructed from the set P in such a way so that $A \to \beta \in P'$ if $(A_1, \ldots, A, \ldots, A_n) \to (w_1, \ldots, \beta, \ldots w_n) \in P$ for any n and any position of pair A, β in the rule-tuple from P.

Based on the previous two definitions, we can define conditions FF' and FFL'. Again, they copy the definition of conditions FF and FFL for context-free grammars. Again, we have to build context-free production rules. In the case of conditions FF' and FFL', we take in the account just the leftmost pairs of non-terminal and string.

Definition 7.3.4 Condition FF' *holds if for every set of production rules:*

$$(A, \ldots) \to (\alpha_1, \ldots) \mid (A, \ldots) \to (\alpha_2, \ldots) \mid \ \ldots \ \mid (A, \ldots) \to (\alpha_k, \ldots) \in P$$

from SCG grammar, $G = (V, T, P, S)$, it is satisfied:

$$FIRST'(\alpha_i) \cap FIRST'(\alpha_j) = \emptyset, \ \forall i \neq j, 1 \leq i, j \leq k$$

Definition 7.3.5 Condition FFL' *holds if for every set of production rules*

$$(A, \ldots) \to (\alpha_1, \ldots) \mid (A, \ldots) \to (\alpha_2, \ldots) \mid \ \ldots \ \mid (A, \ldots) \to (\alpha_k, \ldots) \in P$$

such that

$$\exists i, 1 \leq i \leq k : \alpha_i \Rightarrow^* \varepsilon$$

from SCG grammar, $G = (V, T, P, S)$, it is satisfied:

$$FIRST'(\alpha_j) \cap FOLLOW'(A) = \emptyset, \forall i \neq j, 1 \leq i, j \leq k$$

Finally, the notion of LL grammars can be introduced for SCG grammars. It is the same as for context-free grammars with respect to the appropriate definitions.

Definition 7.3.6 *Let $G = (V, T, P, S)$ be a left derivating SCG. G is an* LL SCG *if condition FF' and FFL' are satisfied.*

As an example, we give an LL SCG defining the language $\{a^n b^n c^n | n \geq 0\}$. Let $G = (V, T, P, S)$ where:

1. $V = \{S, A, B, C, a, b, c\}$
2. $T = \{a, b, c\}$
3. S is the starting symbol
4. $P = ($ $(S) \rightarrow (ABC)$
   $(A, B, C) \rightarrow (aA, bB, cC)$
   $(A, B, C) \rightarrow (\varepsilon, \varepsilon, \varepsilon)$ $)$

7.4 Results

The key result of the chapter is twofold:

1. Simple creation of parsing table (in fact "the same" as for LL_1 languages)
2. Formal binding of the table with DRPA (deterministic regulated pushdown automata)

This section starts with a definition of the appropriate parsing table. The algorithm for filling in the table follows.

Definition 7.4.1 *The parsing table for LL SCG, $G = (V, T, P, S)$, is a matrix, columns of which are denoted with set T plus a symbol denoting end-of-tape (we use \$). Rows of the matrix are denoted with set V plus a symbol denoting empty pushdown (we use #). Moreover, for the sake of better readability, the rows are grouped by symbols belonging to set T and symbols belonging to the set $V \setminus T$.*

For a given column and row, there is always exactly one action defined. The actions follow next (informally):

- **expand***: expands pushdown content according to a given rule from P — the rule is usually denoted by its number (the action is written as a single letter e with the appropriate number), but the rule itself may be used if necessary/appropriate.*
  Note: detailed operation is given in the definition of the DRPA defined by the parsing table.
- **pop***: checks that symbols on the top of the pushdown and under the reading head are the same ones and, moreover, they are the requested ones. If the condition is satisfied the symbol from the top of the pushdown is removed and the reading head is moved one symbol to the right.*

- **accept**: *(often written as acc) verifies that the pushdown is empty (just symbol # is present) and that the reading head is at the end of the input tape (symbol \$ under the reading head). If the condition is satisfied the work of the parser is successfully completed—input string accepted.*
- **error**: *(often denoted by blank space) for a given combination of pushdown and tape symbols there is no rule and thus the syntax error may be reported—input string is refused.*

The following scheme sketches the table structure (symbols T and V come from the SCG, $G = (V, T, P, S)$, \$ stands for the end-of-tape, and # stands for empty pushdown):

	T	\$
$V \setminus T$		
T		
#		

Algorithm 7.4.1 *The table defined in Definition 7.4.1 is filled this way:*
Input*: LL SCG* $G = (V, T, P, S)$
Output*: The parsing table*
Algorithm*: Marking of rows and columns of the table is given by Definition 7.4.1. The content is defined in the following way:*

1. *All the cells are filled with the action* error
2. *The cell, which is on the row denoted by symbol a, $a \in T$, and which is at the same time denoted by the column marked by the same symbol a, $a \in T$, should contain the action* pop *a.*
3. *For a rule $(A_1, \ldots, A_n) \to (\alpha_1, \ldots, A_n) \in P$, we fill the row marked with symbol A_1, $A_1 \in V \setminus T$. The columns are denoted by $FIRST'(\alpha_1) \setminus \{\varepsilon\}$ and if $\alpha_1 \Rightarrow^* \varepsilon$ then the columns are also denoted by $FOLLOW'(A_1)$. Such cells are filled with action* expand *for the given rule $(A_1, \ldots, A_n) \to (\alpha_1, \ldots, \alpha_n)$.*
4. *The cell, which is on the row denoted by symbol #, pushdown bottom, and which is at the same time denoted by the column marked by the symbol \$, end of input tape, should contain the action* accept.

As an example, we can use the LL SCG from the end of section 7.3. The table built using Definition 7.4.1 and Algorithm 7.4.1 should look like this one:

	a	b	c	$\$$
S	$e1$			
A	$e2$	$e3$	$e3$	$e3$
B				
C				
a	pop a			
b		pop b		
c			pop c	
$\#$				acc

The rules of the grammar are numbered this way:

$$\begin{array}{rll} P = (& (S) \rightarrow (ABC) & (1) \\ & (A,B,C) \rightarrow (aA, bB, cC) & (2) \\ & (A,B,C) \rightarrow (\varepsilon,\varepsilon,\varepsilon) \quad) & (3) \end{array}$$

The presented notion of the parsing table is very much like the one for LL_1 languages (see [3, 12]), but the semantics of the behaviour is still missing. Next, we give a construction of DRPA and, thus, we formally define the behaviour of the DRPA controlled (and defined) by such a parsing table.

Algorithm 7.4.2 *The deterministic regulated pushdown automata for parsing of LL SCG can be constructed the following way:*
Input*: LL SCG parsing table (Algorithm 7.4.1) and appropriate grammar* G, $G = (V, T, P, S')$
Output*: DRPA (def. 6.3.1)*
Algorithm*: First of all we build the pushdown automaton* $M = (Q, \Sigma, \Omega, R, s, S, F)$*. For the sake of simplicity, we assume that* $\{\$, \#\} \cap V = \emptyset$*, where* $\$$ *stands for end-of-input marker and* $\#$ *stands for pushdown bottom marker.*

Note*: For a SCG rule* $(A_1, \dots, A_n) \rightarrow (\alpha_1, \dots, \alpha_n) \in P$*, we refer further to its components using two indices:* i *and* n*. The index* i *stands for items* $1 \dots (n-1)$ *and the index* n *for items indexed by* n *only. Thus,* A_i *stands for any non-terminal on the left-hand side of the grammar rule except the rightmost one,* A_n*. On the other hand,* α_n *stands for the rightmost string over* V *from the given grammar rule.*

1. $\Sigma = T$
2. $\Omega = V$
3. $S = S'$
4. $Q = \{q, q_F\} \cup \mathcal{A}_2 \cup \mathcal{A}_{2\varepsilon} \cup \mathcal{A}_1 \cup \mathcal{A}_{1\varepsilon}$
   $\mathcal{A}_2 = \{\underline{A_1x2A_2}, \dots, \underline{A_1xnA_n}, \underline{A_1xn_r\alpha_n}, \dots, \underline{A_1x1_r\alpha_1} \mid$
   $\forall A_1, x : (A_1, \dots, A_n) \rightarrow (\alpha_1, \dots, \alpha_n) \in P \wedge x \in \mathit{FIRST}'(\alpha_1) \setminus \{\varepsilon\}\}$

$$\begin{aligned}
\mathcal{A}_{2\varepsilon} &= \{\underline{A_1x2A_2}, \ldots, \underline{A_1xnA_n}, \underline{A_1xn_r\alpha_n}, \ldots, \underline{A_1x1_r\alpha_1} \mid \\
&\quad \forall A_1, x : (A_1, \ldots, A_n) \rightarrow (\alpha_1, \ldots, \alpha_n) \in P \wedge \\
&\qquad\qquad x \in FOLLOW'(A_1) \wedge \varepsilon \in FIRST'(\alpha_1)\ \} \\
\mathcal{A}_1 &= \{\underline{A_1x\alpha_1} \mid \forall A_1, x : (A_1) \rightarrow (\alpha_1) \in P \wedge x \in FIRST'(\alpha_1) \setminus \varepsilon\} \\
\mathcal{A}_{1\varepsilon} &= \{\underline{A_1x\alpha_1} \mid \forall A_1, x : (A_1) \rightarrow (\alpha_1) \in P \wedge \\
&\qquad\qquad\qquad\qquad x \in FOLLOW'(A_1) \wedge \varepsilon \in FIRST'(\alpha_1)\}
\end{aligned}$$

5. $F = \{q_F\}$

6. $s = q$

7. $$\begin{aligned}
R &= \{\#q\$ \rightarrow q_F \mid q, q_F \in Q\} \cup \mathcal{P} \cup \mathcal{E}_2 \cup \mathcal{E}_1 \cup \mathcal{T} \cup \mathcal{U} \\
\mathcal{P} &= \{xqx \rightarrow q \mid q \in Q, \text{pop}x \text{ is defined in the parsing table}, x \in \Sigma\} \\
\mathcal{E}_2 &= \{A_1qx \rightarrow \underline{A_1x2A_2}x \mid q, \underline{A_1x2A_2} \in Q, A_1 \in \Omega \setminus \Sigma, x \in \Sigma\} \cup \\
&\quad \{A_i\underline{A_1xiA_i} \rightarrow \underline{A_1x(i+1)A_{(i+1)}} \mid \\
&\qquad\qquad\qquad \underline{A_1xiA_i}, \underline{A_1x(i+1)A_{(i+1)}} \in Q, A_i \in \Omega \setminus \Sigma\} \cup \\
&\quad \{A_n\underline{A_1xnA_n} \rightarrow \underline{A_1xn_r\alpha_n} \mid \underline{A_1xnA_n}, \underline{A_1xn_r\alpha_n} \in Q, A_n \in \Omega \setminus \Sigma\} \cup \\
&\quad \{\underline{A_1xj_r\alpha_j} \rightarrow \alpha_j\underline{A_1x(j-1)_r\alpha_{(j-1)}} \mid \\
&\qquad\qquad \underline{A_1xj_r\alpha_j}, \underline{A_1x(j-1)_r\alpha_{(j-1)}} \in Q, \alpha_j \in \Omega^*, j \in \{2, \ldots, n\}\} \cup \\
&\quad \{\underline{A_1x1_r\alpha_1} \rightarrow \alpha_1q \mid \underline{A_1x1_r\alpha_1}, q \in Q, \alpha_1 \in \Omega^*\} \\
\mathcal{E}_1 &= \{A_1qx \rightarrow \underline{A_1x\alpha_1}x \mid q, \underline{A_1x\alpha_1} \in Q, A_1 \in \Omega \setminus \Sigma, x \in \Sigma\} \cup \\
&\quad \{\underline{A_1x\alpha_1} \rightarrow \alpha_1q \mid \underline{A_1x\alpha_1}, q \in Q, \alpha_1 \in \Omega^*\} \\
\mathcal{T} &= \{a\underline{A_1xjA_j} \rightarrow \underline{A_1xjA_j} \mid \underline{A_1xjA_j} \in Q, a \in \Omega \setminus \{A_i\}, j \in \{2, \ldots, n\}\} \\
\mathcal{U} &= \{\underline{A_1xi_r\alpha_i} \rightarrow a\underline{A_1xi_r\alpha_i} \mid \underline{A_1xi_r\alpha_i} \in Q, a \in \Omega\}
\end{aligned}$$

This pushdown automaton (non-regulated so far) is non-deterministic, obviously. Informally, we briefly describe the meaning of the construction. Input alphabet equals to terminals of the grammar. That is why the pushdown alphabet equals the set of symbols V. Thus, relation $\Sigma \subset \Omega$ is given. The set of states contains the state q, which is also the starting state and actions pop *are concentrated to this state. Another state, q_F, is the final state, if this is reached the input string was successfully accepted. States described by sets $\mathcal{A}_1$, $\mathcal{A}_{1\varepsilon}$ and $\mathcal{A}_2$, $\mathcal{A}_{2\varepsilon}$ are used to handle actions* expand. *Set $\mathcal{A}_1$ contains states used for grammar rules of a context-free character (grammar rules of the shape $(A) \rightarrow (\alpha)$). Set $\mathcal{A}_2$ contains states useful for expansion of context grammar rules. Sets $\mathcal{A}_{j\varepsilon}$ are counterparts of the non-ε sets handling situation, where ε is a member of set $FIRST'$ and the set $FOLLOW'$ is used to denote the appropriate "expansion" symbols. Operation of the automaton defined in R is non-deterministic due to the set of rules defined in $\mathcal{U}$. Rules defined in $\mathcal{P}$ perform actions* pop, *rules described by set $\mathcal{E}_1$ handle expansion of the context-free rules. Rules described by the sets $\mathcal{E}_2, \mathcal{T}, \mathcal{U}$ perform actions expand. Set $\mathcal{E}_2$ defines operations for removing non-terminals and placing the appropriate strings of symbols from the right-hand side of the grammar rules. To search for appropriate non-terminals, rules defined in $\mathcal{T}$ are used. Symbols skipped by rules in $\mathcal{T}$ are returned back by rules contained in $\mathcal{U}$. This is done a non-deterministic way, so far.*

Thus, we have to define a control language (by a context-free grammar) in such a way, so that the regulated pushdown automaton becomes deterministic. The grammar defining the language follows.
It is a context-free grammar G_c, which describes the control language. Let $G_c = (N_c, T_c, S_c, P_c)$, where:

- $N_c \; = \; \{K, L\}$
- $T_c \; = \; \{< r > \; \mid r \in R\}$
- $S_c \; = \; K$
- $P_c \; = \; \{$
  $K \; \rightarrow < \#q\$ \rightarrow q_F >$ *action* accept
  $K \; \rightarrow < xqx \rightarrow q > K,$ *for all x if* pop x *is defined in parsing table*
  $K \; \rightarrow < A_1qx \rightarrow \underline{A_1x\alpha_1}x >< \underline{A_1x\alpha_1} \rightarrow \alpha_1 q > K,$
  for all suitable rules from R—
  appropriate pairs are taken from set $\mathcal{E}_1$ from Algorithm 7.4.2
  (complete set $\mathcal{E}_1$ is used)
  $K \; \rightarrow < A_1qx \rightarrow \underline{A_1x2A_2}x > L < \underline{A_1x1_r\alpha_1} \rightarrow \alpha_1 q > K,$
  for all suitable rules from R—
  appropriate pairs are taken from set $\mathcal{E}_2$ from Algorithm 7.4.2
  (the first and the last definition subsets are taken into account;
  complete subsets are used)
  $L \; \rightarrow < A_i\underline{A_1xiA_i} \rightarrow \underline{A_1x(i+1)A_{(i+1)}} > L$
  $< \underline{A_1xi_r\alpha_i} \rightarrow \alpha_i\underline{A_1x(i-1)_r\alpha_{(i-1)}} >,$
  for all suitable rules from R—
  appropriate pairs are taken from set $\mathcal{E}_2$ from Algorithm 7.4.2
  (the second and the fourth definition subsets are taken into account;
  complete the second and partial the fourth subsets are used)
  $L \; \rightarrow < A_n\underline{A_1xnA_n} \rightarrow \underline{A_1xn_r\alpha_n} >< \underline{A_1xn_r\alpha_n} \rightarrow \alpha_n\underline{A_1x(n-1)_r\alpha_{(n-1)}} >,$
  for all suitable rules from R—
  appropriate pairs are taken from set $\mathcal{E}_2$ from Algorithm 7.4.2
  (the third and the fourth definition subsets are taken into account;
  complete the third and partial the fourth subsets are used)
  $L \; \rightarrow < a\underline{A_1xjA_j} \rightarrow \underline{A_1xjA_j} > L$
  $< \underline{A_1x(j-1)_r\alpha_{(j-1)}} \rightarrow \underline{aA_1x(j-1)_r\alpha_{(j-1)}} >,$
  for all suitable rules from R, where $j \in \{2, \ldots, n\}$—
  appropriate pairs are taken from sets $\mathcal{T}$ and $\mathcal{U}$ from Algorithm 7.4.2
  (sets $\mathcal{T}$ and $\mathcal{U}$ are used partially)
  $\}$

All the rules bound with the non-terminal L serve for the expansion of the context rules of the SCG. Moreover, these rules make the RPA be a deterministic one.
Note: The grammar G_c is LL_1 context-free grammar.

The definition and respective construction of the DRPA is completed. It also stated a theorem that the resulting RPA is deterministic. The proof is left to the reader, because the idea was only sketched in the construction: The only non-deterministic part of the pushdown automaton is made deterministic by control language as removal of the symbol from the pushdown during expansion of context rules is always paired with storage of the same symbol later, in the correct position.

As an example, we present a definition of such a DRPA for the LL SCG presented above in this section. We recollect the grammar changing the name of the starting symbol to ensure clarity: Let $G = (V, T, P, S')$ where:

1. $V = \{S', A, B, C, a, b, c\}$
2. $T = \{a, b, c\}$
3. S' is the starting symbol
4. $$\begin{array}{lll} P = (& (S') \rightarrow (ABC) & (1) \\ & (A, B, C) \rightarrow (aA, bB, cC) & (2) \\ & (A, B, C) \rightarrow (\varepsilon, \varepsilon, \varepsilon) \quad) & (3) \end{array}$$

The appropriate parsing table directly follows Algorithm 7.4.1. Construction of parsing DRPA will be started with the definition of PA, M, according to the definition. Let $M = (Q,\ \Sigma,\ \Omega,\ R,\ s,\ S,\ F)$, where:

1. $\Sigma \ = \ T \ = \ \{a, b, c\}$
2. $\Omega \ = \ V \ = \ \{a, b, c, S', A, B, C\}$
3. $S \ = \ S'$
4. $$\begin{aligned} Q &= \{q, q_F\} \ \cup \ \mathcal{A}_2 \ \cup \ \mathcal{A}_{2\varepsilon} \ \cup \ \mathcal{A}_1 \ \cup \ \mathcal{A}_{1\varepsilon} \\ \mathcal{A}_2 &= \{\underline{Aa2B}, \underline{Aa3C}, \underline{Aa3_rcC}, \underline{Aa2_rbB}, \underline{Aa1_raA}\} \\ \mathcal{A}_{2\varepsilon} &= \{\underline{Ab2B}, \underline{Ab3C}, \underline{Ab3_r\varepsilon}, \underline{Ab2_r\varepsilon}, \underline{Ab1_r\varepsilon}, \\ &\qquad \underline{Ac2B}, \underline{Ac3C}, \underline{Ac3_r\varepsilon}, \underline{Ac2_r\varepsilon}, \underline{Ac1_r\varepsilon}, \\ &\qquad \underline{A\$2B}, \underline{A\$3C}, \underline{A\$3_r\varepsilon}, \underline{A\$2_r\varepsilon}, \underline{A\$1_r\varepsilon}\} \\ \mathcal{A}_1 &= \{\underline{S'aABC}\} \\ \mathcal{A}_{1\varepsilon} &= \{\} \end{aligned}$$
5. $F \ = \ \{q_F\}$
6. $s \ = \ q$
7. $$\begin{aligned} R &= \{\#q\$ \rightarrow q_F \mid q, q_F \in Q\} \cup \mathcal{P} \cup \mathcal{E}_2 \cup \mathcal{E}_1 \cup \mathcal{T} \cup \mathcal{U} \\ \mathcal{P} &= \{aqa \rightarrow q, bqb \rightarrow q, cqc \rightarrow q\} \\ \mathcal{E}_2 &= \{Aqa \rightarrow \underline{Aa2B}a, B\underline{Aa2B} \rightarrow \underline{Aa3C}, C\underline{Aa3C} \rightarrow \underline{Aa3_rcC}, \\ &\qquad \underline{Aa3_rcC} \rightarrow cC\underline{Aa2_rbB}, \underline{Aa2_rbB} \rightarrow bB\underline{Aa1_raA}, \underline{Aa1_raA} \rightarrow aAq, \\ &\qquad Aqb \rightarrow \underline{Ab2B}b, B\underline{Ab2B} \rightarrow \underline{Ab3C}, C\underline{Ab3C} \rightarrow \underline{Ab3_r\varepsilon}, \end{aligned}$$

$$
\begin{array}{rcl}
 & & \underline{Ab3_r\varepsilon} \rightarrow \underline{Ab2_r\varepsilon}, \underline{Ab2_r\varepsilon} \rightarrow \underline{Ab1_r\varepsilon}, \underline{Ab1_r\varepsilon} \rightarrow q, \\
 & & Aqc \rightarrow \underline{Ac2B}c, B\underline{Ac2B} \rightarrow \underline{Ac3C}, C\underline{Ac3C} \rightarrow \underline{Ac3_r\varepsilon}, \\
 & & \underline{Ac3_r\varepsilon} \rightarrow \underline{Ac2_r\varepsilon}, \underline{Ac2_r\varepsilon} \rightarrow \underline{Ac1_r\varepsilon}, \underline{Ac1_r\varepsilon} \rightarrow q, \\
 & & Aq\$ \rightarrow \underline{A\$2B}\$, B\underline{A\$2B} \rightarrow \underline{A\$3C}, C\underline{A\$3C} \rightarrow \underline{A\$3_r\varepsilon}, \\
 & & \underline{A\$3_r\varepsilon} \rightarrow \underline{A\$2_r\varepsilon}, \underline{A\$2_r\varepsilon} \rightarrow \underline{A\$1_r\varepsilon}, \underline{A\$1_r\varepsilon} \rightarrow q\} \\
\mathcal{E}_1 & = & \{S'qa \rightarrow \underline{S'aABC}a, \underline{S'aABC} \rightarrow ABCq\} \\
\mathcal{T} & = & \{a\underline{Aa2B} \rightarrow \underline{Aa2B}, b\underline{Aa2B} \rightarrow \underline{Aa2B}, c\underline{Aa2B} \rightarrow \underline{Aa2B}, \\
 & & A\underline{Aa2B} \rightarrow \underline{Aa2B}, C\underline{Aa2B} \rightarrow \underline{Aa2B}, S\underline{Aa2B} \rightarrow \underline{Aa2B}, \\
 & & a\underline{Aa3C} \rightarrow \underline{Aa3C}, b\underline{Aa3C} \rightarrow \underline{Aa3C}, c\underline{Aa3C} \rightarrow \underline{Aa3C}, \\
 & & A\underline{Aa3C} \rightarrow \underline{Aa3C}, B\underline{Aa3C} \rightarrow \underline{Aa3C}, S\underline{Aa3C} \rightarrow \underline{Aa3C}, \\
 & & a\underline{Ab2B} \rightarrow \underline{Ab2B}, b\underline{Ab2B} \rightarrow \underline{Ab2B}, \ldots\} \\
\mathcal{U} & = & \{\underline{Aa2_rbB} \rightarrow a\underline{Aa2_rbB}, \underline{Aa2_rbB} \rightarrow b\underline{Aa2_rbB}, \underline{Aa2_rbB} \rightarrow c\underline{Aa2_rbB}, \\
 & & \underline{Aa2_rbB} \rightarrow A\underline{Aa2_rbB}, \underline{Aa2_rbB} \rightarrow B\underline{Aa2_rbB}, \underline{Aa2_rbB} \rightarrow C\underline{Aa2_rbB}, \\
 & & \underline{Aa2_rbB} \rightarrow S\underline{Aa2_rbB}, \\
 & & \underline{Aa1_raA} \rightarrow a\underline{Aa1_raA}, \underline{Aa1_raA} \rightarrow b\underline{Aa1_raA}, \underline{Aa1_raA} \rightarrow c\underline{Aa1_raA}, \\
 & & \underline{Aa1_raA} \rightarrow A\underline{Aa1_raA}, \underline{Aa1_raA} \rightarrow B\underline{Aa1_raA}, \\
 & & \underline{Aa1_raA} \rightarrow C\underline{Aa1_raA}, \underline{Aa1_raA} \rightarrow S\underline{Aa1_raA}, \\
 & & \underline{Ab2_r\varepsilon} \rightarrow a\underline{Ab2_r\varepsilon}, \underline{Ab2_r\varepsilon} \rightarrow b\underline{Ab2_r\varepsilon}, \ldots\}
\end{array}
$$

The construction of PA is completed (in the example, sets $\mathcal{T}$ and $\mathcal{U}$ are not fully listed because their content can be easily derived and the number of their elements quite large). The last, but not least, remaining task is to build a grammar, G_c, of the control language. Let $G_c = (N_c, T_c, S_c, P_c)$, where:

- $N_c \ = \ \{K, L\}$
- $T_c \ = \ \{< r > \ \mid r \in R\}$
- $S_c \ = \ K$
- $P_c \ = \ \{$

$$
\begin{array}{ll}
K & \rightarrow < \#q\$ \rightarrow q_F > \\
K & \rightarrow < aqa \rightarrow q > K \mid \ < bqb \rightarrow q > K \mid \ < cqc \rightarrow q > K \\
K & \rightarrow < S'qa \rightarrow \underline{S'aABC}a >< \underline{S'aABC} \rightarrow ABCq > K \\
K & \rightarrow < Aqa \rightarrow \underline{Aa2B}a > L < \underline{Aa1_raA} \rightarrow aAq > K \\
 & \mid \ < Aqb \rightarrow \underline{Ab2B}b > L < \underline{Ab1_r\varepsilon} \rightarrow q > K \\
 & \mid \ < Aqc \rightarrow \underline{Ac2B}c > L < \underline{Ac1_r\varepsilon} \rightarrow q > K \\
 & \mid \ < Aq\$ \rightarrow \underline{A\$2B}\$ > L < \underline{A\$1_r\varepsilon} \rightarrow q > K \\
L & \rightarrow < B\underline{Aa2B} \rightarrow \underline{Aa3C} > L < \underline{Aa2_rbB} \rightarrow bB\underline{Aa1_raA} > \\
 & \mid \ < B\underline{Ab2B} \rightarrow \underline{Ab3C} > L < \underline{Ab2_r\varepsilon} \rightarrow \underline{Ab1_r\varepsilon} > \\
 & \mid \ < B\underline{Ac2B} \rightarrow \underline{Ac3C} > L < \underline{Ac2_r\varepsilon} \rightarrow \underline{Ac1_r\varepsilon} > \\
 & \mid \ < B\underline{A\$2B} \rightarrow \underline{A\$3C} > L < \underline{A\$2_r\varepsilon} \rightarrow \underline{A\$1_r\varepsilon} > \\
L & \rightarrow < C\underline{Aa3C} \rightarrow \underline{Aa3_rcC} >< \underline{Aa3_rcC} \rightarrow cC\underline{Aa2_rbB} > \\
 & \mid \ < C\underline{Ab3C} \rightarrow \underline{Ab3_r\varepsilon} >< \underline{Ab3_r\varepsilon} \rightarrow \underline{Ab2_r\varepsilon} > \\
 & \mid \ < C\underline{Ac3C} \rightarrow \underline{Ac3_r\varepsilon} >< \underline{Ac3_r\varepsilon} \rightarrow \underline{Ac2_r\varepsilon} >
\end{array}
$$

$$
\begin{array}{rcl}
 & | & <C\underline{A\$3C} \rightarrow \underline{A\$3_r\varepsilon} >< \underline{A\$3_r\varepsilon} \rightarrow \underline{A\$2_r\varepsilon} > \\
L & \rightarrow & < a\underline{Aa2B} \rightarrow \underline{Aa2B} > L < \underline{Aa1_raA} \rightarrow a\underline{Aa1_raA} > \\
 & | & < b\underline{Aa2B} \rightarrow \underline{Aa2B} > L < \underline{Aa1_raA} \rightarrow b\underline{Aa1_raA} > \\
 & | & < c\underline{Aa2B} \rightarrow \underline{Aa2B} > L < \underline{Aa1_raA} \rightarrow c\underline{Aa1_raA} > \\
 & | & < A\underline{Aa2B} \rightarrow \underline{Aa2B} > L < \underline{Aa1_raA} \rightarrow A\underline{Aa1_raA} > \\
 & | & < C\underline{Aa2B} \rightarrow \underline{Aa2B} > L < \underline{Aa1_raA} \rightarrow C\underline{Aa1_raA} > \\
 & | & < S\underline{Aa2B} \rightarrow \underline{Aa2B} > L < \underline{Aa1_raA} \rightarrow S\underline{Aa1_raA} > \\
 & | & < a\underline{Aa3C} \rightarrow \underline{Aa3C} > L < \underline{Aa2_rbB} \rightarrow a\underline{Aa2_rbB} > \\
 & | & < b\underline{Aa3C} \rightarrow \underline{Aa3C} > L < \underline{Aa2_rbB} \rightarrow b\underline{Aa2_rbB} > \\
 & | & < c\underline{Aa3C} \rightarrow \underline{Aa3C} > L < \underline{Aa2_rbB} \rightarrow c\underline{Aa2_rbB} > \\
 & | & < A\underline{Aa3C} \rightarrow \underline{Aa3C} > L < \underline{Aa2_rbB} \rightarrow A\underline{Aa2_rbB} > \\
 & | & < B\underline{Aa3C} \rightarrow \underline{Aa3C} > L < \underline{Aa2_rbB} \rightarrow B\underline{Aa2_rbB} > \\
 & | & < S\underline{Aa3C} \rightarrow \underline{Aa3C} > L < \underline{Aa2_rbB} \rightarrow S\underline{Aa2_rbB} > \\
 & | & < a\underline{Ab2B} \rightarrow \underline{Ab2B} > L < \underline{Ab1_r\varepsilon} \rightarrow a\underline{Ab1_r\varepsilon} > \\
 & \ldots &
\end{array}
$$

The construction of context-free grammar is completed too (again, not all grammar rules are presented as their construction for rules representing paired retrieval from and storage to pushdown is obvious). Thus, the DRPA is completely defined.

7.5 Chapter Summary and Open Problems

This chapter presented a possibility of how the DRPA can be used for syntactic analysis. In particular, a new kind of grammars (LL SCG) was introduced. Moreover, a way of how the efficient deterministic parser of such grammars can be constructed was also presented. Such a parser is then built over DRPA. A notion known from the LL_1 grammars is utilized in this approach and further extended to cover the possibilities of LL SCG and the power of DRPA.

Even if the presented approach opens big possibilities for powerful syntactic analysis (quite simple construction of grammar, not too "far" from context-free grammars), there are still some open items:

- What is the relation of LL SCG to Chomsky language hierarchy?
- Parsing table construction and DRPA construction contains certain inefficiencies, could they be improved?
- LL SCG class of grammars is, in a fact, LL_1 SCG class. Is it feasible to define LL_k SCG class of grammars for $k > 1$? How complex would a construction of the parser be then? How complex would the parser itself be then?

Besides these open items, we can see as an open item the implementation of a parser constructor similar to particularly known ones such as y.a.c.c. and bison.

Chapter 8

Conclusion

This thesis presented two approaches as to how certain classes of grammars (languages) can be exploited in parser construction. In particular, LL_k, $k > 1$, languages and languages defined by LL scattered context grammars were at the centre of our focus.

Syntactic analysis of LL_k languages is already a known issue and construction of their parsers has already been possible for quite a long time. Nevertheless, the resulting parsers, especially for $k > 1$, are not too efficient. The reason lies in implementation, which does not enable handling multiple symbols under the reading head effectively. The technique presented in this thesis enables transformation of multi-symbol reading head pushdown automata used for syntactic analysis of LL_k languages to "classical" pushdown automata working with a single-symbol under the reading head. This feature offers an efficient way to the implementation of parsers based on such automata. Moreover, designers and programmers of programming language compilers and/or interpreters can feel unbounded by the restricting rules of LL_1 languages during grammar design. Even if the expressive power of (LA)LR languages is always greater than the one of LL_k languages, manipulation with attributes is much easier and wider for LL languages. In particular, inherited attributes are natural to LL languages while their incorporation to (LA)LR languages requires the usage of high skills and tricks during implementation of a parser.

Syntactic analysis of languages derived from scattered context grammars has not been introduced yet. In particular, efficient deterministic syntactic analysis is meant, in this case, that general applicable non-deterministic approaches are always available. The thesis presents a new subclass of scattered context grammars called LL SCG. The rules applied to SCG to make it LL SCG are derived from ones used for definition of the LL context-free grammars. If a grammar satisfies conditions for being LL SCG a deterministic parser can be built for it easily. The parser is based on regulated pushdown automata, though (contrary to LL context-free languages' parsers). This is due to the higher expressive power of LL SCG. Fortunately, the expressive power of regulated pushdown automata is also higher if compared with "classical" pushdown automata. For efficient parsing, a deterministic version of regulated pushdown automata was used. Creation of a deterministic regulated pushdown automata from LL SCG is straightforward and can be done by deterministic algorithm, presented in this thesis too. Implementation of such a parser also seems to be quite easy as the control language satisfies conditions for being LL_1. Talking about control language, a small difference between control languages of deterministic and non-deterministic regulated pushdown automata could be noticed. Linear languages are sufficiently strong to control non-deterministic regulated pushdown automata while context-free languages are used for the deterministic version so far. Nevertheless, this feature does not influence the parser behaviour. Moreover, having the control language from a set of LL_1 languages is advantageous. From our point of view, the existence of efficient parsers for LL SCG languages opens great possibilities in compiler construction for the future.

As already mentioned in this chapter, regulated pushdown automata play a big role in this thesis. That is why this recently introduced concept was presented in detail in this thesis. Its expressive power is equal to the one of Turing machine and, thus, it provides us great possibilities especially in the area of compiler construction because the base of the regulated pushdown automata (the pushdown automata) is well known in the area—compilers of present programming languages use pushdown automata for syntactic analysis. Moreover, it seems that the definition of regulated pushdown automata is more straightforward than the definition of Turing machine of the same function.

8.1 Future Research

Every chapter besides the introducing one presented open items and questions as its conclusion. Our future research would like to provide answers to them. Moreover, practical implementation should be available too. There are three main tasks for the near future:

1. Classification of the expressive power of LL SCG, their position in Chomsky language classification.

2. Further studies over deterministic pushdown automata—expressive power, control language, etc.

3. Attributed parsing based on LL SCG—experiments with the practical implementations of parsers, the definition of attribute classification and usage, etc.

Of course, there are a few more questions to be answered, nevertheless, our closest focus is on providing a practically usable compiler constructor (such as y.a.c.c. or bison) together with pilot implementation of a compiler/interpreter of some known programming language. We think it is possible as LL SCG is very close to context-free grammars, to find a definition of LL SCG for current programming languages in such a way so that context links can be verified on the level of grammar itself.

Bibliography

[1] Abadi, M., Cardelli, L.: *A Theory of Objects*, Springer, New York, 1996, ISBN 0-387-94775-2.

[2] Aho, A.V., Hopcroft, J.E., Ullman, J.D.: *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.

[3] Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading MA, 1986.

[4] Aho, A.V., Ullman, J.D.: *The Theory of Parsing, Translation, and Compiling, Volume I: Parsing*, Prentice-Hall, Inc., 1972, ISBN 0-13-914556-7.

[5] Aho, A.V., Ullman, J.D.: *The Theory of Parsing, Translation, and Compiling, Volume II: Compiling*, Prentice-Hall, Inc., 1972, ISBN 0-13-914564-8.

[6] Appel, A.W.: Garbage Collection Can Be Faster Than Stack Allocation, *Information Processing Letters 25 (1987)*, North Holland, pages 275–279.

[7] Appel, A.W.: *Compiling with Continuations*, Cambridge University Press, 1992.

[8] Augustsson, L., Johnsson, T.: *Parallel Graph Reduction with the* $< \nu, G >$*-Machine*, Functional Programming Languages and Computer Architecture 1989, pages 202–213.

[9] Barendregt, H.P., Kennaway, R., Klop, J.W., Sleep, M.R.: *Needed Reduction and Spine Strategies for the Lambda Calculus*, Information and Computation 75(3): 191-231 (1987).

[10] Beneš, M.: *Object-Oriented Model of a Programming Language*, Proceedings of MOSIS'96 Conference, 1996, Krnov, Czech Republic, pp. 33–38, MARQ Ostrava, VSB - TU Ostrava.

[11] Beneš, M.: *Type Systems in Object-Oriented Model*, Proceedings of MOSIS'97 Conference Volume 1, April 28–30, 1997, Hradec nad Moravicí, Czech Republic, pp. 104–109, ISBN 80-85988-16-X.

[12] Beneš, M., Češka, M., Hruška, T.: *Překladače*, Technical University of Brno, 1992.

[13] Beneš, M., Hruška, T.: *Modelling Objects with Changing Roles*, Proceedings of 23rd Conference of ASU, 1997, Stara Lesna, High Tatras, Slovakia, pp. 188–195, MARQ Ostrava, VSZ Informatika s r.o., Kosice.

[14] Beneš, M., Hruška, T.: *Layout of Object in Object-Oriented Database System*, Proceedings of 17th Conference DATASEM 97, 1997, Brno, Czech Republic, pp. 89–96, CS-COMPEX, a.s., ISBN 80-238-1176-2.

[15] Brodský, J., Staudek, J., Pokorný, J.: *Operační a databázové systémy*, Technical University of Brno, 1992.

[16] Bruce, K.B.: A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics, *J. of Functional Programming*, January 1993, Cambridge University Press.

[17] Cattell, G.G.: *The Object Database Standard ODMG-93*, Release 1.1, Morgan Kaufmann Publishers, 1994.

[18] Češka, M., Hruška, T., Motyčková, L.: *Vyčíslitelnost a složitost*, Technical University of Brno, 1992.

[19] Češka, M., Rábová, Z.: *Gramatiky a jazyky*, Technical University of Brno, 1988.

[20] Damas, L., Milner, R.: *Principal Type Schemes for Functional Programs*, Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982, pages 207–212.

[21] Dassow, J., Paun, G.: *Regulated Rewriting in Formal Language Theory*, Springer, New York, 1989.

[22] Douence, R., Fradet, P.: *A taxonomy of functional language implementations. Part II: Call-by-Name, Call-by-Need and Graph Reduction*, INRIA, technical report No 3050, Nov. 1996.

[23] Ellis, M.A., Stroustrup, B.: *The Annotated C++ Reference Manual*, AT&T Bell Laboratories, 1990, ISBN 0-201-51459-1.

[24] Finne, S., Burn, G.: *Assessing the Evaluation Transformer Model of Reduction on the Spineless G-Machine*, Functional Programming Languages and Computer Architecture 1993, pages 331-339.

[25] Fradet, P.: *Compilation of Head and Strong Reduction*, In Porc. of the 5th European Symposium on Programming, LNCS, vol. 788, pp. 211-224. Springer-Verlag, Edinburg, UK, April 1994.

[26] Georgeff M.: *Transformations and Reduction Strategies for Typed Lambda Expressions*, ACM Transactions on Programming Languages and Systems, Vol. 6, No. 4, October 1984, pages 603–631.

[27] Gordon, M.J.C.: *Programming Language Theory and its Implementation*, Prentice Hall, 1988, ISBN 0-13-730417-X, ISBN 0-13-730409-9 Pbk.

[28] Gray, P.M.D., Kulkarni, K.G., Paton, N.W.: *Object-Oriented Databases*, Prentice Hall, 1992.

[29] Greibach, S., Hopcroft, J.: Scattered Context Grammars, *Journal of Computer and System Sciences*, Vol: 3, pp. 233–247, Academia Press, Inc., 1969.

[30] Harrison, M.: *Introduction to Formal Language Theory*, Addison Wesley, Reading, 1978.

[31] Hruška, T., Beneš, M.: *Jazyk pro popis údajů objektově orientovaného databázového modelu*, In: Sborník konference Některé nové přístupy při tvorbě informačních systémů, ÚIVT FEI VUT Brno 1995, pp. 28–32.

[32] Hruška, T., Beneš, M., Máčel, M.: *Database System G2*, In: Proceeding of COFAX Conference of Database Systems, House of Technology Bratislava 1995, pp. 13–19.

[33] Issarny, V.: *Configuration-Based Programming Systems*, In: Proc. of SOFSEM'97: Theory and Practise of Informatics, Milovy, Czech Republic, Novemeber 22-29, 1997, ISBN 0302-9743, pp. 183-200.

[34] Jensen, K.: *Coloured Petri Nets*, Springer-Verlag Berlin Heidelberg, 1992.

[35] Jeuring, J., Meijer, E.: *Advanced Functional Programming*, Springer-Verlag, 1995.

[36] Jones, M.P.: *A system of constructor classes: overloading and implicit higher-order polymorphism*, In FPCA '93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, June 1993.

[37] Jones, M.P.: *Dictionary-free Overloading by Partial Evaluation*, ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994.

[38] Jones, M.P.: *Functional Programming with Overloading and Higher-Order Polymorphism*, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, Springer-Verlag Lecture Notes in Computer Science 925, May 1995.

[39] Jones, M.P.: *GOFER, Functional programming environment, Version 2.20*, mpj@prg.ox.ac.uk, 1991.

[40] Jones, M.P.: *ML typing, explicit polymorphism and qualified types*, In TACS '94: Conference on theoretical aspects of computer software, Sendai, Japan, Springer-Verlag Lecture Notes in Computer Science, 789, April, 1994.

[41] Jones, M.P.: *A theory of qualified types*, In proc. of ESOP'92, 4th European Symposium on Programming, Rennes, France, February 1992, Springer-Verlag, pp. 287-306.

[42] Jones, S.L.P.: *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.

[43] Jones, S.L.P., Lester, D.: *Implementing Functional Languages.*, Prentice-Hall, 1992.

[44] Kleijn, H.C.M., Rozenberg, G.: On the Generative Power of Regular Pattern Grammars, *Acta Informatica*, Vol. 20, pp. 391–411, 1983.

[45] Khoshafian, S., Abnous, R.: *Object Orientation. Concepts, Languages, Databases, User Interfaces*, John Wiley & Sons, 1990, ISBN 0-471-51802-6.

[46] Kolář, D.: *Compilation of Functional Languages To Efficient Sequential Code*, Diploma Thesis, TU Brno, 1994.

[47] Kolář, D.: *Overloading in Object-Oriented Data Models*, Proceedings of MOSIS'97 Conference Volume 1, April 28–30, 1997, Hradec nad Moravicí, Czech Republic, pp. 86–91, ISBN 80-85988-16-X.

[48] Kolář, D.: *Functional Technology for Object-Oriented Modeling and Databases*, PhD Thesis, TU Brno, 1998.

[49] Kolář, D.: *Simulation of LL_k Parsers with Wide Context by Automaton with One-Symbol Reading Head*, Proceedings of 38th International Conference MOSIS '04—Modelling and Simulation of Systems, April 19–21, 2004, Rožnov pod Radhoštěm, Czech Republic, pp. 347–354, ISBN 80-85988-98-4.

[50] Latteux, M., Leguy, B., Ratoandromanana, B.: The family of one-counter languages is closed under quotient, *Acta Informatica*, 22 (1985), 579–588.

[51] Leroy, X.: *The Objective Caml system, documentation and user's guide*, 1997, Institut National de Recherche en Informatique et Automatique, Francie, Release 1.05, http://pauillac.inria.fr/ocaml/htmlman/.

[52] Martin, J.C.: *Introduction To Languages and The Theory of Computation*, McGraw-Hill, Inc., USA, 1991, ISBN 0-07-040659-6.

[53] Meduna, A.: *Automata and Languages: Theory and Applications*. Springer, London, 2000.

[54] Meduna, A., Kolář, D.: Regulated Pushdown Automata, *Acta Cybernetica*, Vol. 14, pp. 653–664, 2000.

[55] Meduna, A., Kolář, D.: One-Turn Regulated Pushdown Automata and Their Reduction, In: *Fundamenta Informaticae*, 2002, Vol. 16, Amsterdam, NL, pp. 399–405, ISSN 0169-2968.

[56] Mens, T., Mens, K., Steyaert, P.: *OPUS: a Formal Approach to Object-Orientation*, Published in FME '94 Proceedingss, LNCS 873, Springer-Verlag, 1994, pp. 326-345.

[57] Mens, T., Mens, K., Steyaert, P.: *OPUS: a Calculus for Modelling Object-Oriented Concepts*, Published in OOIS '94 Proceedings, Springer-Verlag, 1994, pp. 152-165.

[58] Milner, R.: A Theory of Type Polymorphism In Programming, *Journal of Computer and System Sciences*, 17, 3, 1978.

[59] Mycroft, A.: *Abstract Interpretation and Optimising Transformations for Applicative Programs*, PhD Thesis, Department of computer Science, University of Edinburgh, Scotland, 1981. 180 pages. Also report CST-15-81.

[60] Okawa, S., Hirose, S.: Homomorphic characterizations of recursively enumerable languages with very small language classes, *Theor. Computer Sci.*, 250, 1 (2001), 55–69.

[61] Păun, Gh., Rozenberg, G., Salomaa, A.: *DNA Computing*, Springer-Verlag, Berlin, 1998.

[62] Rozenberg, G., Salomaa, A. — eds.: *Handbook of Formal Languages; Volumes 1 through 3*, Springer, Berlin/Heidelberg, 1997.

[63] Salomaa, A.: *Formal Languages*, Academic Press, New York, 1973.

[64] Schmidt, D.A.: *The Structure of Typed Programming Languages*, MIT Press, 1994.

[65] Odersky, M., Wadler, P.: *Pizza into Java: Translating theory into practice*, Proc. 24th ACM Symposium on Principles of Programming Languages, January 1997.

[66] Odersky, M., Wadler, P., Wehr, M.: *A Second Look at Overloading*, Proc. of FPCA'95 Conf. on Functional Programming Languages and Computer Architecture, 1995.

[67] Reisig, W.: *A Primer in Petri Net Design*, Springer-Verlag Berlin Heidelberg, 1992.

[68] Tofte, M., Talpin, J.-P.: *Implementation of the Typed Call-by-Value* λ-*calculus using a Stack of Regions*, POPL '94: 21st ACM Symposium on Principles of Programming Languages, January 17–21, 1994, Portland, OR USA, pages 188–201.

[69] Traub, K.R.: *Implementation of Non-Strict Functional Programming Languages*, Pitman, 1991.

[70] Volpano, D.M., Smith, G.S.: *On the Complexity of ML Typability with Overloading*, Proc. of FPCA'91 Conf. on Functional Programming Languages and Computer Architecture, 1991.

[71] Wikström, Å.: *Functional Programming Using Standard ML*, Prentice Hall, 1987.

[72] Williams, M.H., Paton, N.W.: *From OO Through Deduction to Active Databases - ROCK, ROLL & RAP*, In: Proc. of SOFSEM'97: Theory and Practise of Informatics, Milovy, Czech Republic, Novemeber 22-29, 1997, ISBN 0302-9743, pp. 313-330.

[73] Lindholm, T., Yellin, F.: *The Java Virtual Machine Specification*, Addison-Wesley, 1996, ISBN 0-201-63452-X.