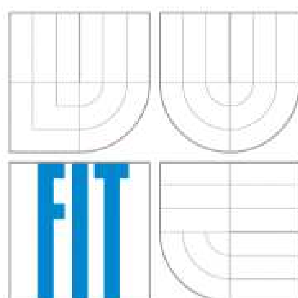
Brno University of Technology

Faculty of Information Technology



Lukáš Rychnovský

GRAMMATICAL MODELS OF COMPUTATIONAL DISTRIBUTION AND CONCURRENCY
Theory and Application

Ph.D. Thesis

Study programme: Information Technology
Advisor: prof. RNDr. Alexander Meduna, CSc.

Abstract This work first defines regulated formal systems such as regulated rewriting system, regulated grammar and regulated automaton. Also some basic theorems are presented. The fourth chapter defines right-linear grammars with a start string of length n regulated by regular language and postulates its equality with n-parallel right-linear languages formed in Wood hierarchy. Then we restrict the number of changing derivation positions and the main result follows. The fifth chapter describes parsing techniques for context-sensitive languages and their implementation.

Keywords regulated rewriting, Wood hierarchy, regulated automata, regulated grammars, right-linear grammar with start string of length n regulated by regular language, parsing of context-sensitive languages, parser implementation.

Abstrakt V této práci nejdříve definujeme několik řízených formálních systémů, jako například obecný řízený přepisovací systém, řízenou gramatiku a řízený automat. Také prezentujeme některé základní poznatky z teorie řízených přepisovacích systémů. Ve čtvrté kapitole definujeme pravě-lineární gramatiky se startovacím řetězcem délky n, řízené regulárními jazyky a postulujeme jejich ekvivalenci s n-paralelními pravě-lineárními gramatikami, které definují Woodovu hierarchii. Následně omezíme počet změn derivační pozice a formulujeme hlavní výsledky práce. Pátá kapitola pak popisuje techniky parsingu pro kontextové jazyky a jejich implementaci.

Klíčová slova řízené přepisování, Woodova hierarchie, řízený automat, řízená gramatika, pravě-lineární gramatika se startovacím řetězcem délky n, řízená regulárním jazykem, parsing kontextových jazyků, implementace parseru.

Rychnovský, L.: *Grammatical Models of Computational Distribution and Concurrency : Theory and Application*, Ph.D. Thesis, FIT VUT, Brno, 2009.

I'd like to thank prof. RNDr. Alexander Meduna, CSc., the advisor of my work, for valuable advices, pedagogic and scientific leading, inspiration and references to literature.

I hereby declare that this work is my genuine work, created solely by me under the direction of my advisor prof. RNDr. Alexander Meduna, CSc. All information sources used are properly cited including complete reference to the original work.

Brno, December 2009 Lukáš Rychnovský

....

Contents

I Introduction 7

1 Motivation 7

2 Preliminaries 11
2.1 Basic notations . . . 11
2.2 Semigroups and Monoids . . . 11

3 Definitions 13
3.1 Automata . . . 13
3.2 Grammars . . . 16
3.3 Regulated Rewriting . . . 19
3.4 Control Languages . . . 21
3.5 Grammar Systems . . . 27

II New Grammatical Models of Distribution and Concurrency 31

4 Theoretical Results 31
4.1 Definitions . . . 32
4.2 New Results . . . 34

5 Applications in Parsing 42
5.1 Classic Parsing Technique . . . 44
5.2 Alternative Approach to Parsing . . . 47
5.3 The Power of Modified Multistack Machine . . . 50
5.4 Type Checking . . . 53

5.5 Other Applications of CS Languages 54
5.6 Implementation 55

III Conclusion 62

6 Summary 62

7 Historical and Bibliographical Remarks 63

8 Future research 64

IV Appendixes 68

Appendix A: Program Documentation 68
A.1 Installation 79

Appendix B: Authors Curriculum vitæ 80

List of Figures

1 Regulation of languages 9
2 The Chomsky hierarchy 18
3 Regulation of REG and LIN languages 42
4 Regulation of CF languages 42
5 Another look at language hierarchy 43
6 Correct and incorrect programs 44
7 Parser in compiler model 45
8 False ambiguity 45
9 True ambiguity 46
10 Injecting type system into grammar 54
11 Implemented parser 55
12 Main window of parser application 56
13 Successfully parsed program 57
14 String assigned to integer variable 58
15 Unused variable b 59
16 Read but not set variable c 59
17 Limiting grammar with 5 statements 60
18 5 statements 61
19 6 statements 61
20 Regulation of languages 63

Part I
Introduction

1 Motivation

In the late fifties the linguist Noam Chomsky defined his famous formal language hierarchy based on the restriction of the form of productions. Very soon many mathematicians and computer scientists began to extend this simple hierarchy by adding new forms of production rules.

In the seventies a new approach of extending the Chomsky hierarchy was developed. The new approach was not only to restrict the form of the production rules but also the way in which grammar is allowed to generate words. This approach opened a brand new part of formal language theory called regulated rewriting or grammar with controlled derivations. Mathematicians such as Salomaa, Dassow and Păun started their research and very soon a complex theory with many results was born.

Nowadays, many books cover this part of formal language theory. But still there are many grey areas worthy of interest.

This work tries to map regulated rewriting models and their properties. It starts with necessary mathematical background for formal language theory in the second chapter. The third chapter starts with a definition of a rewriting system as a basic concept of all formal language theory. The definition of this rewriting system is then extended and divided into two well-known approaches: grammar and automaton. The first approach,

automaton-based, is divided into three parts according to the Chomsky hierarchy. Finite automaton and language accepted by finite automaton are defined in the first part and a relation between the set of regular languages and languages accepted by finite automata is postulated. In the second part, we move toward the set of context-free languages. We define pushdown automata and language accepted by such automata and define relation between these languages. In the third part we define the Turing machine and postulate the relation between it and context-sensitive and recursive enumerable languages. The other approach, grammar-based, is defined separately and the Chomsky hierarchy is formed.

Later in the third chapter regulated rewriting systems are mapped. A regulated rewriting system is any formal system where every application of any production rule can be prohibited. We can achieve this behavior by several main concepts.

The first is to tie some productions together, as we can see in the case of matrix grammar or programmed grammar. Instead of single productions, matrix grammar uses a finite set of finite sequences of productions. Productions cannot be applied separately, but a whole sequence has to be applied. In applying such a sequence, one first rewrites according to the first production, then according to the second production, and so on, until one has rewritten according to the last production. The sequences are referred to as *matrices*. Programmed grammars are based on a similar method of regulating as matrix grammars. In the case of programmed grammar G, one is given two sets, $\sigma(f)$ and $\varphi(f)$, together with each production f of the entire production set P of G, referring to the *success* and *failure* field of f, respectively. If we have applied f, then the next production to be applied must belong to $\sigma(f)$. If we have applied f in the appearance checking sense, that is, noticing that the left side of f is not a subword of the word under scan, then the next production to be applied must belong to $\varphi(f)$. The sets $\sigma(f)$ and $\varphi(f)$ are also noted as the go-to fields of f.

The second concept is to permit some productions only in some cases or deny usage of some productions in some cases based on actual sentence form as we can see in the case of permitting and forbidding grammars or random-context grammars. Unlike previous cases, the behavior of permitting and forbidding grammars depends on actual sentential form. In permitting grammar G, one is given a set of terminals and nonterminals $P(f)$ together with each production set P of G, referred to *permitting set*. In a forbidding grammar this set $F(f)$ is called *forbidding set*. If we should apply rule f, we first look into permitting set $P(f)$ and rewrite only if all symbols of $P(f)$ are subwords of word under scan. In forbidding case we check whether none of the symbols of $F(f)$ are subword of word under scan. Random-context grammars are combination of permitting and forbidding grammars. Every production rule is a triple (f, Q, R) where Q, R are sets. If we should apply production f, we must first check whether all symbols of Q appear and no symbol in R appears in word under scan. Only in such a case is production allowed.

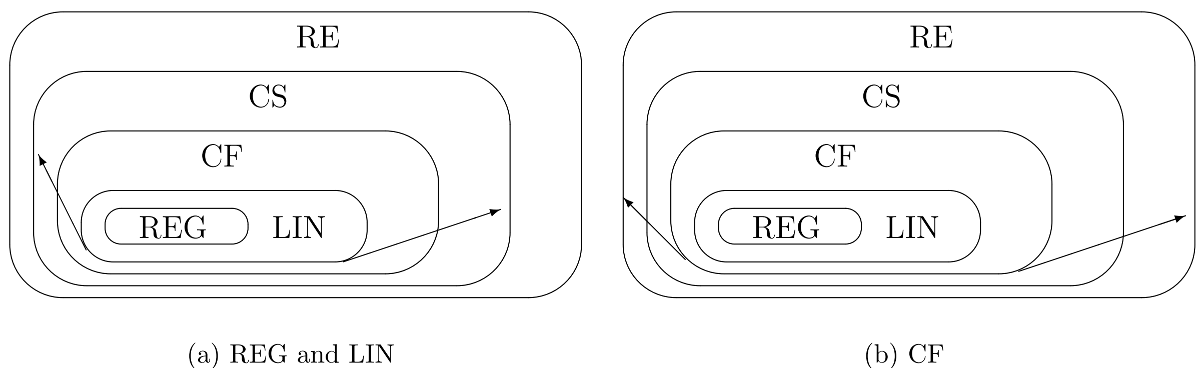
The third concept is to define control language. In this case we label production rules and demand successful derivation to form a word from control language over the alphabet of production labels. This construction allows us to obtain a large family of new languages based on the combination of regulated and regulating languages. We can also regulate automata in the same way as grammars. Every automaton transition is described by

a specific symbol and only certain words over such an alphabet are accepted according to control language. The question is whether equivalent models (regular languages and finite automata, context-free grammars and pushdown automata, etc.), when regulated by the same language, have the same generative power. As shown in the fourth section, the answer is no. Hence, it is necessary to study even equivalent formal models separately when they are regulated. In general, regulation greatly increases the generative power of a formal model. For example pushdown automaton regulated by linear language is as powerful as the Turing machine.

The fourth and fifth chapters are dedicated to the author's own results. The fourth chapter starts with the definition of a formal model with start string of length n. It is easy to prove that any formal model from classic Chomsky hierarchy (even if regulated) doesn't extend its generative power if we start from start string rather than start nonterminal. This is not true if we enrich right-linear grammar regulated by regular language by start string. In this case we obtain stronger formalism than with start symbol. Moreover, the longer start string we allow, the more powerful model we obtain. In theorem 4.3, equivalence with the Wood language hierarchy based on n-parallel right-linear languages is proved.

In the next part of the fourth chapter there is another result presented. We start again with right-linear grammars with start string of length n and we define a new way of limitation: we restrict the number of times that derivation position switches from one position in start string to another. This kind of limitation does not restrict the number of derivations in general, but the impact on generative power is significant. The whole former language hierarchy collapses again to only the set of regular languages.

We can study regulated formal models in two main ways, as described in Figure 1. The first one is based on regular or linear languages and regulation extends their power towards context-sensitive languages. The second way is based on context-free languages and it is possible to reach recursive enumerable languages.



(a) REG and LIN

(b) CF

Figure 1: Regulation of languages

Both concepts greatly increase the power of underlying grammars. But each of these concepts focuses on different parts – theory and application. The first approach describes the regulation of REG and LIN languages, which is attractive from the theoretical point of

view. It is easier to understand the mechanisms of regulation on REG and LIN languages than on CF. On the other hand, the second approach is the practical one. The theory gained during the studying of previous parts can be applied to CF languages to move towards CS and RE languages. This approach is very promising in compiler theory because, by the regulation, we get a more powerful formal model on which we can use slightly modified classic parsing techniques. These non context-free parsers and compilers are able to recognize semantic errors that standard context-free parsers can not. For example, we can restrict the number of code lines directly in grammar. Or we can limit the number of variables or we can require that any variable is defined before used. This approach is described in the fifth chapter.

This work tries to connect two main approaches to formal languages – theory and application. From this connection, both sides profit. The theory driven by application brings a new look at proofs of known theorems. They need to be lead in a constructive way because application needs implementation and implementing algorithms according to non-constructive proof is impossible. On the other hand, parsing algorithms based on more powerful theory can bring us many advantages.

2 Preliminaries

In this section we define some basic notations and necessary mathematical background to formal language theory.

2.1 Basic notations

$\mathbb{N} = \{0, 1, 2, \ldots\}$ is the set of nonnegative integers. $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ is the set of integers. Let E be a set. Then card(E) or $|E|$ is the number of its elements. The empty set is denoted by $\emptyset$. If A, B are subsets of E, then we write $A \subseteq B$ if and only if $x \in A \Rightarrow x \in B$, and $A \subset B$ if and only if $A \subseteq B$ and $A \neq B$. The set of all subsets of E, i.e. the powerset of E, is denoted by 2^E. If X and Y are sets, then *Cartesian product*, denoted $X \times Y$, is the set of all possible ordered pairs:

$$X \times Y = \{(x, y) \mid x \in X, y \in Y\}.$$

2.2 Semigroups and Monoids

A *semigroup* consists of a set M and a binary operation on M, denoted by multiplication $(\cdot)$, and is postulated to be *associative*:

$$\forall m_1, m_2, m_3 \in M \ : \ m_1(m_2m_3) = (m_1m_2)m_3.$$

A *neutral element* or a *unit* is an element $1_M \in M$ (also denoted by 1 for short if no confusion can arise) such that

$$\forall m \in M \ : \ 1_M m = m 1_M = m.$$

A semigroup which has a neutral element is a *monoid*. The neutral element of a monoid is unique.

Given two subsets A, B of a monoid M, the product AB is defined by

$$AB = \{c \in M \mid \exists a \in A, \exists b \in B \ : \ c = ab\}.$$

This definition converts 2^M into monoid with unit $\{1_M\}$. A subset A of M is a *subsemigroup* (*submonoid*) of M if $A^2 \subseteq A$ ($1 \in M$ and $A^2 \subseteq A$). Given any subset A of M, the sets

$$A^+ = \bigcup_{n \geq 1} A^n, \qquad A^* = \bigcup_{n \geq 0} A^n,$$

where $A^0 = \{1\}$ and $A^{n+1} = A^n A$ are subsemigroups resp. submonoids of M. A^+ (resp. A^*) is called the subsemigroup (submonoid) *generated* by A. If $M = A^*$ for some $A \subset M$, then A is a *system of generators* of M. A monoid is *finitely generated* if it has a finite system of generators. The unary operations $A \to A^+$ and $A \to A^*$ are called Kleene *plus* and *star* operations. It is clear that

$$A^+ = AA^* = A^*A \quad \text{and} \quad A^* = 1 \cup A^+.$$

If M, M' are monoids a *homomorphism* $\alpha : M \to M'$ is a function satisfying

$$\begin{aligned} \alpha(m_1 m_2) &= \alpha(m_1)\alpha(m_2) \quad \forall m_1, m_2 \in M \\ \alpha(1_M) &= 1_{M'}. \end{aligned}$$

A homomorphism $\alpha : X^* \to M'$, where X is an alphabet, is completely defined by the values $\alpha(x)$ of the letters $x \in X$.

For any set X, the *free monoid* X^* *generated by* X is defined so that there exists exactly one homomorphism f so that for every monoid M following diagram commutes : $\kappa = !f \circ \iota$.

$$\begin{array}{ccc} X & \xrightarrow{\ \iota\ } & X^* \\ & \searrow^{\kappa} \quad \swarrow_{!f} & \\ & M & \end{array}$$

The elements of X^* are n-tuples

$$u = (x_1, x_2, \ldots, x_n) \qquad n \geq 0$$

of elements of X. If $v = (y_1, y_2, \ldots, y_m)$ is another element of X^*, the product uv is defined by concatenation

$$uv = (x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m).$$

Sometimes, if no confusion can arise, we will not distinguish elements of X from singletons. Thus elements of X^* may be written as

$$u = x_1 x_2 \ldots x_n.$$

u is called a *word*, $x \in X$ is called a *letter* and X itself is called an *alphabet*, set *alph(u)* is defined as $\{x \mid x \in X, x \text{ appears in } u\}$. The length $|u|$ of a word $u \in X^*$ is the number of letters composing it. The neutral element of X^* is called *empty word* and is noted 1 or ε. The *reversal* of word $u = x_1 x_2 \ldots x_n$ ($n \geq 0, x_i \in X$) is denoted by $\overline{u}$ or u^R and is defined by $\overline{u} = x_n x_{n-1} \ldots x_1$. A *formal language* over X is any subset of X^*.

For any set X, *n-ary relation* on set X is any subset of X^n. A *binary relation* R is a special case of n-ary relation for $n = 2$, for any $(x, y) \in R$ we write $R(x, y)$ or xRy. Relation R is said to be *reflexive* if for every $x \in X$ holds xRx. Relation R is said to be *symmetric* if for every $x, y \in X$ holds that if xRy then yRx. Relation R is said to be *transitive* if for every $x, y, z \in X$ holds that if xRy and yRz then xRz. *Reflexive closure* of relation R is the smallest reflexive relation over X containing R. *Transitive closure* of relation R, noted as R^+, is the smallest transitive relation over X containing R. *Reflexive and transitive closure* of relation R, noted as R^*, is the smallest transitive and relative relation over X containing R.

3 Definitions

In this section we define some basic notations and models in formal language theory. We start with rewriting system and extend definition to automata and grammars. The Chomsky hierarchy is introduced in the end of this chapter.

Definition 3.1. A *rewriting system* is ordered pair (V, P), where V is a finite alphabet and $P \subseteq V^* \times V^*$ is a finite set of rewriting rules. Every rewriting rule is a pair (u, v), where $u, v \in V^*$ and will be written as $u \rightarrow v$.

Let us define the relation $\Rightarrow$ over V^*. If $u \rightarrow v \in P$ and $x, y \in V^*$, then $xuy \Rightarrow xvy$ $[u \rightarrow v]$ or simply $xuy \Rightarrow xvy$ is called a *direct derivation*. Let $\Rightarrow^n$, where $n \geq 0$, denotes the n-th power of relation $\Rightarrow$. Furthermore let $\Rightarrow^+$ and $\Rightarrow^*$ denote the transitive closure and reflexive and transitive closure of relation $\Rightarrow$ respectively. We call $\Rightarrow^*$ a (*general*) *derivation*.

3.1 Automata

Definition 3.2. A *deterministic finite automaton* (dFA or FA, for short) is a rewriting system, usually noted as a 5-tuple $T = (Q, \Sigma, \delta, s, F)$, where

1. Q is a finite set of states.
2. Σ is a finite set of the input alphabet.
3. δ is a finite transition relation $Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow Q$.
4. s $\in$ Q is the start state.
5. F $\subseteq$ Q is a set of final states.

A *configuration* of finite automaton is an ordered pair (q, w), where $q \in Q$ is current state and $w \in \Sigma^*$ are non read input characters.
A *computational step* of finite automaton is a binary relation $\vdash_T$ (or simply $\vdash$ if no confusion can arise) defined as

$$(q_1, aw) \vdash_T (q_2, w) \Leftrightarrow \delta(q_1, a) = q_2.$$

Let $\vdash^n$, where $n \geq 0$, denotes the n-th power of relation $\vdash$. Furthermore let $\vdash^+$ and $\vdash^*$ denote the transitive closure and reflexive and transitive closure of relation $\vdash$ respectively.
The language accepted by finite automaton $T = (Q, \Sigma, \delta, s, F)$ is

$$L(T) = \{w \mid w \in \Sigma^*, (s, w) \vdash_T^* (q, \varepsilon), q \in F\}$$

Alternatively we can extend definition of transition relation to *extended transition relation* $\delta^* : Q \times \Sigma^* \rightarrow Q$ this way:

1. $\delta^*(q, \varepsilon) = q$ for every $q \in Q$,

2. $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$ for every $q \in Q, w \in \Sigma^*$ and $a \in \Sigma$.

The language accepted by finite automaton $T = (Q, \Sigma, \delta, s, F)$ is

$$L(T) = \{w \mid w \in \Sigma^*, \delta^*(s, w) \in F\}$$

Definition 3.3. A *nondeterministic finite automaton* (non-dFA) is a rewriting system, usually noted as a 5-tuple $T = (Q, \Sigma, \delta, s, F)$, where Q, Σ, s and F has the same meaning as in previous definition and δ is a finite transition relation $Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$.

Definition 3.4. A *deterministic pushdown automaton* (dPDA or PDA, for short) is a rewriting system, usually noted as a 7-tuple $T = (Q, \Sigma, \Omega, \delta, s, \nabla, F)$, where

1. Q is a finite set of states.
2. Σ is a finite set of the input alphabet.
3. Ω is a finite set of the stack alphabet.
4. δ is a finite transition relation $(Q \times (\Sigma \cup \{\varepsilon\}) \times \Omega) \rightarrow Q \times \Omega^*$.
5. s $\in$ Q is the start state.
6. $\nabla \in \Omega$ is the initial stack symbol
7. F $\subseteq$ Q is a set of final states.

A *configuration* of the pushdown automaton is a triple (q, w, γ), where $q \in Q$ is current state, $w \in \Sigma^*$ are non read characters and $\gamma \in \Omega^*$ are symbols on stack.
A *computational step* of pushdown automaton is a binary relation $\vdash_T$ (or simply $\vdash$ if no confusion can arise) defined as

$$(q_1, aw, Z\gamma) \vdash_T (q_2, w, Y\gamma) \Leftrightarrow \delta(q_1, a, Z) = (q_2, Y).$$

Let $\vdash^n$, where $n \geq 0$, denotes the n-th power of relation $\vdash$. Furthermore let $\vdash^+$ and $\vdash^*$ denote the transitive closure and reflexive and transitive closure of relation $\vdash$ respectively.
The language accepted by a pushdown automaton $T = (Q, \Sigma, \delta, s, F)$ *by final state* is

$$L(T) = \{w \mid w \in \Sigma^*, (s, w, \nabla) \vdash_T^* (q_F, \varepsilon, \gamma), q_F \in F, \gamma \in \Omega^*\}$$

The language accepted by a pushdown automaton $T = (Q, \Sigma, \delta, s, F)$ *by empty pushdown* is

$$L(T) = \{w \mid w \in \Sigma^*, (s, w, \nabla) \vdash_T^* (q, \varepsilon, \varepsilon), q \in Q\}$$

Definition 3.5. Let $M = (Q, \Sigma, \Omega, \delta, s, \nabla, F)$ be a PDA and let $x, x', x'' \in \Omega^*, y, y', y'' \in \Sigma^*$, q, q', $q'' \in Q$, and $\nabla xqy \vdash \nabla x'q'y' \vdash \nabla x''q''y''$. If $|x| \leq |x'|$ and $|x'| > |x''|$, then $\nabla x'q'y' \vdash \nabla x''q''y''$ 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* (OTSA).

Definition 3.6. A *nondeterministic pushdown automaton* (non-dPDA) is a rewriting system, usually noted as a 7-tuple $T = (Q, \Sigma, \Omega, \delta, s, \nabla, F)$, where $Q, \Sigma, \Omega, s, \nabla$ and F has the same meaning as in previous definition and δ is a finite transition relation $(Q \times (\Sigma \cup \{\varepsilon\}) \times \Omega) \rightarrow 2^{Q \times \Omega^*}$.

Definition 3.7. *A deterministic Turing machine* (DTM or TM, for short) is a 5-tuple $T = (Q, \Sigma, \Gamma, q_0, \delta)$, where

1. Q is a finite set of states, assumed not to contain the halt state (q_F).
2. Σ, the input alphabet, is a set of symbols, Σ is assumed not to contain Δ, the blank symbol
3. Γ, the tape alphabet, is a finite set with $\Sigma \subseteq \Gamma$.
4. $q_0 \in Q$ is the initial state,
5. δ is a partial function from $Q \times \Gamma \rightarrow (Q \cup \{q_F\}) \times \Gamma \times \{R, L, S\}$.

A configuration of the dTM is a pair $(q, x\underline{a}y)$ where q is a state, $x, y \in \Gamma^*, a \in \Gamma$, and the underlined symbol represents current position of the head, which allows to read from and write 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 makes a sequence of moves 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 makes a sequence of moves from the first configuration in zero or more moves.

Definition 3.8. An input string $x \in \Sigma^*$ is *accepted by Turing machine* T if starting T with an input x leads eventually to halting configuration. In other words, x is accepted if for some strings $y, z \in \Gamma^*$ and some $a \in \Gamma$

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

In this situation we say T halts on the input x. *The language accepted by* T is the set of input strings that are accepted by T.

Definition 3.9. *A nondeterministic Turing machine* (non-dTM, for short) is a 5-tuple $T = (Q, \Sigma, \Gamma, q_0, \delta)$, where Q, Σ, Γ and q_0 has the same meaning as in previous definition and δ is a finite transition relation $Q \times \Gamma \rightarrow 2^{(Q \cup \{q_F\}) \times \Gamma \times \{R, L, S\}}$.

Definition 3.10. An input string $x \in \Sigma^*$ is *accepted by Linear Bounded Turing machine (LBTM)* T if starting T with an input x leads eventually to halting configuration but only a finite contiguous portion whose length is a linear function of the length of the initial input can be accessed by the read/write head. *The language accepted by linear bounded Turing machine* T is the set of input strings that are accepted by LBTM T.

3.2 Grammars

Definition 3.11. A *grammar* G is a quadruple $G = (N, T, P, S)$ where N is a finite set of *nonterminals*, T is a finite set of *terminals*, $T \cap N = \emptyset$, P is a finite set of *production rules* $P \subseteq \{(N \cup T)^* N (N \cup T)^* \times (N \cup T)^*\}$. An element $(\alpha, \beta) \in P$ will be written as $\alpha \rightarrow \beta$. Symbol $S \in N$ is the starting nonterminal.
A grammar is called *propagating* if and only if $(\alpha, \varepsilon) \notin P$.

Definition 3.12. Let $G = (N, T, P, S)$ be a grammar. If $\alpha \rightarrow \beta \in P$ and $u, v \in (N \cup T)^*$, then $u\alpha v \Rightarrow u\beta v$ $[\alpha \rightarrow \beta]$ or simply $u\alpha v \Rightarrow u\beta v$ is called a *simple derivation*. In the previously defined manner, we extend $\Rightarrow$ to $\Rightarrow^n$, where $n \geq 0$ and $\Rightarrow^+$ and $\Rightarrow^*$ and we call it a (*general*) *derivation*.

Definition 3.13. The *language generated by grammar* $G = (N, T, P, S)$, $L(G)$, is defined as $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$.

Definition 3.14. A grammar $G = (N, T, P, S)$ is called *regular grammar* if every rule from P is in the form $\alpha \rightarrow \beta$, where

$$\alpha \in N,$$
$$\beta \in TN \cup T.$$

A language, L, is regular if and only if $L = L(G)$, where G is a regular grammar. The set of languages generated by regular grammars is denoted by *REG*.

Definition 3.15. A grammar $G = (N, T, P, S)$ is called *linear grammar* if every rule from P is in the form $\alpha \rightarrow \beta$, where

$$\alpha \in N,$$
$$\beta \in T^* N T^* \cup T^*.$$

The grammar G is called *left-linear* (resp. *right-linear*) if $\beta \in NT^* \cup T^*$ (resp. $\beta \in T^* N \cup T^*$).

Definition 3.16. A grammar $G = (N, T, P, S)$ is called *context-free grammar* if every rule from P is in the form $\alpha \rightarrow \beta$, where

$$\alpha \in N,$$
$$\beta \in (N \cup T)^*.$$

A language, L, is context-free if and only if $L = L(G)$, where G is a context-free grammar. The family of languages generated by context-free grammars is denoted by *CF*.

Definition 3.17. A grammar $G = (N, T, P, S)$ is called *propagating context-free grammar* or *context-free grammar without epsilon rules* if for every rule from P is in the form $\alpha \rightarrow \beta$, where

$$\alpha \in N,$$
$$\beta \in (N \cup T)^+.$$

Definition 3.18. A grammar $G = (N, T, P, S)$ is called *context-sensitive grammar* if every rule from P is in the form $\alpha \rightarrow \beta$ or $S \rightarrow \varepsilon$ where,

$$|\alpha| \leq |\beta|.$$

A language, L, is context-sensitive if and only if $L = L(G)$, where G is a context-sensitive grammar. The family of languages generated by context-sensitive grammars is denoted by *CS*.

Definition 3.19. A grammar in general form as defined in Definition 3.11 is called *unrestricted grammar*. The family of languages generated by unrestricted grammars is known as recursive enumerable and is denoted by *RE*.

Definition 3.20. The Chomsky hierarchy contains four families of languages defined by the following four types of grammar:

1. Type-0 languages correspond to unrestricted grammars (this level includes all formal grammars). They generate exactly all languages that are recognized by a Turing machine. These languages are also known as the recursively enumerable languages. Note that this is different from the recursive languages which are recognized by an always halting Turing machine.

2. Type-1 languages correspond to context-sensitive grammars. These grammars have rules of the form $\alpha \rightarrow \beta$, where $|\alpha| \leq |\beta|$. The rule $S \rightarrow \epsilon$ is allowed if S does not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognized by a non-deterministic Turing machine whose tape is bounded by a constant times the length of the input.

3. Type-2 languages correspond to context-free grammars. These are defined by rules of the form $A \rightarrow \gamma$ with A a nonterminal and γ a string of terminals and nonterminals. These languages are exactly all languages that can be recognized by a non-deterministic pushdown automaton. Context free languages are the theoretical basis for the syntax of most programming languages.

4. Type-3 languages correspond to regular grammars. Such grammars restrict their rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed by a single nonterminal. The rule $S \rightarrow \epsilon$ is also here allowed if S does not appear on the right side of any rule. These languages are exactly all languages that can be decided by a finite state automaton. Additionally, this family of formal languages is obtained as the family of languages accepted by regular expressions. Consequently these languages are commonly used to define search patterns and the lexical structure of programming languages.

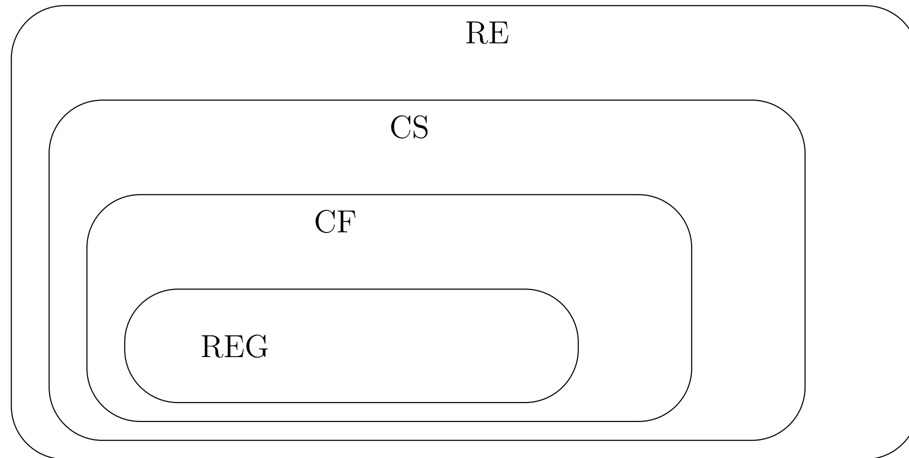


Figure 2: The Chomsky hierarchy

Definition 3.21. A type-2 grammar $G = (N, T, P, S)$ is in *Chomsky normal form* if every production $p \in P$ has one of these forms

1. $A \to BC$
2. $A \to a$
3. $S \to \varepsilon$

where $A, B, C \in N, a \in T$.

Definition 3.22. A type-2 grammar $G = (N, T, P, S)$ is in *Greibach normal form* if every production $p \in P$ has one of these forms

1. $A \to aX$
2. $S \to \varepsilon$

where $A \in N, a \in T$ and $X \in (N - \{S\})^*$.

Definition 3.23. A type-1 grammar $G = (N, T, P, S)$ is in *Kuroda normal form* if every production $p \in P$ has one of these forms

1. $AB \to CD$
2. $A \to BC$
3. $A \to B$
4. $A \to a$

where $A, B, C, D \in N, a \in T$.

Definition 3.24. A type-0 grammar $G = (N, T, P, S)$ is in *Penttonen normal form* if every production $p \in P$ has one of these forms

1. $CB \to CD$
2. $D \to BC$
3. $C \to c$
4. $C \to \varepsilon$

where $B, C, D \in N, c \in T$.

The Chomsky hierarchy of languages (see definition 3.20) was obtained by restricting the form of productions. It is natural to ask what happens if we restrict also the manner in which a grammar is allowed to generate words. Not every derivation of a terminal word is acceptable.

We can make such restrictions by three main principles. First, to join some productions together in some circumstances (a matrix grammar, a programmed grammar, a scattered context grammar, etc.). Second, to enable or disable applying a rule according to the presence of some terminals or nonterminals in the processed word (a permitting or forbidding grammar, a random-context grammar, etc.). Finally, to modify the generation process of the word by control language. We then accept only such words in which only certain productions in a certain order are applied.

Another approach to regulate derivations of a grammar is grammar systems. They are cooperating sets of grammars that either work together with common sentence string (CD grammars systems) or exchange information between them (PC grammar systems).

3.3 Regulated Rewriting

Definition 3.25. A *matrix grammar* is a special case of rewriting system, usually noted as $M = (N, T, R, S)$, where N, T and S are exactly the same as in the definition 3.11 of a grammar, but R is a finite set of finite nonempty sequences of productions

$$P \to Q, \text{ where } P \in N, \ Q \in (N \cup T)^*.$$

The sequences are referred to as matrices and written

$$m = [P_1 \to Q_1, \ldots, P_i \to Q_i], \ i \geq 1. \tag{3.1}$$

Let F be the collection of all productions appearing in the matrices m of a matrix grammar M. Then matrix grammar M is of type linear, context-free, context-sensitive, etc. if and only if the grammar $G = (N, T, F, S)$ has the corresponding property.

For a matrix grammar M, we define yield relation $\Rightarrow_M$ or, in short, $\Rightarrow$ as follows. For any $P, Q \in (N \cup T)^*$, $P \Rightarrow Q$ holds if there exist an integer $r \geq 1$ and words

$$\alpha_1, \ldots, \alpha_{r+1}, \ P_1, \ldots, P_r, \ Q_1, \ldots, Q_r, \ R_1, \ldots, R_r, \ R^1, \ldots, R^r$$

over $(N \cup T)^*$ such that (i) $\alpha_1 = P$ and $\alpha_{r+1} = Q$, (ii) the matrix (3.1) is one of the matrices of M, and (iii) $\alpha_i = R_i P_i R^i$ and $\alpha_{i+1} = R_i Q_i R^i$ for every $i = 1, \ldots, r$.
$Mat = \{L \mid L = L(G), \text{ where } G = (N, T, R, S) \text{ is a matrix grammar}\}$.

Definition 3.26. A *programmed grammar* is a special case of rewriting system, usually noted as an ordered triple (G, σ, φ) where $G = (N, T, R, S)$ is a context-free grammar, and σ and φ are sets of production labels.

For a programmed grammar PG, we define yield relation $\Rightarrow$ and $\Rightarrow_{ac}$ on the set of all pairs (P, f), where $P \in (N \cup T)^*$ and f is the set of production labels of P as follows:

$(P, f_1) \Rightarrow (Q, f_2)$ holds if there are words P_1, P_2, P' and Q' such that (i) $P = P_1P'P_2$ and $Q = P_1Q'P_2$, (ii) the production in R labeled as f_1 is $P' \to Q'$, and (iii) f_2 belongs to the set $\sigma(f_1)$; $(P, f_1) \Rightarrow_{ac} (Q, f_2)$ holds if $(P, f_1) \Rightarrow (Q, f_2)$ holds, or else each of the following conditions is satisfied for some words P' and Q': $P = Q$, (i) the production in R labeled as f_1 is $P' \to Q'$, (ii) P' is not a subword of P, and (iii) f_2 belongs to the set $\varphi(f_1)$ (Thus, only the relation $\Rightarrow_{ac}$ depends on φ).

The language generated by the programmed grammar PG is defined by $L(PG, \sigma) = \{w \in T^* \mid (S, f) \Rightarrow^* (w, f')\}$. $L_{ac}(PG, \sigma, \varphi) = \{w \in T^* \mid (S, f) \Rightarrow^*_{ac} (w, f')\}$.

Definition 3.27. A *random-context grammar* (RCG, for short) is a special case of rewriting system, usually noted as $G = (N, T, P, S)$, where N, T and S are exactly the same as in the definition 3.11 of a grammar, but P is a finite set of random-context rules, that is, triplets in the form of $(C \to \alpha, Q, R)$, $C \to \alpha$ is a CF rule over $N \cup T$, where $C \in N$, and Q and R are subsets of N. For $x, y \in (N \cup T)^*$, we write $x \Rightarrow_{rc} y$, or $x \Rightarrow y$ for short, if $x = x_1Cx_2$, $y = x_1\alpha x_2$ for some $x_1, x_2 \in (N \cup T)^*$, $(C \to \alpha, Q, R)$ is a triplet in P, all symbols of Q appear and no symbol of R appears in x_1x_2 (Q is called the permitting context, and R is called the forbidding context of the rule $C \to \alpha$. If Q and/or R are empty, then no check is necessary.)

Definition 3.28. (See [Wood–73]) For $n \geq 1$, an *n-parallel right-linear grammar*, n-PRLG for short, is an (n+3)-tuple $G = (N_1, \ldots N_n, T, S, P)$ where

- $N_i, 1 \leq i \leq n$ are pairwise disjoint nonterminal alphabets,
- T is a terminal alphabet, $N \cap T = \emptyset$,
- $S \notin N_1 \cup \ldots \cup N_n$ is the start symbol,
- P is a finite set of rules. P contains three kinds of rules
  1. $S \to X_1 \ldots X_n, \quad X_i \in N_i, 1 \leq i \leq n,$
  2. $X \to aY, \quad X, Y \in N_i$, for some $1 \leq i \leq n, a \in T^*$, and
  3. $X \to a, \quad X \in N_i$, for some $1 \leq i \leq n, a \in T^*$.

For $x, y \in (N \cup T \cup \{S\})^*, x \Rightarrow y$ if and only if

- either $x = S$ and $S \to y \in P$,
- or $x = y_1X_1 \ldots y_nX_n, y = y_1x_1 \ldots y_nx_n$, where $y_i \in T^*, x_i \in T^*N \cup T^*, X_i \in N_i$, and $X_i \to x_i \in P, 1 \leq i \leq n$.

${}_{par}\mathfrak{R}(i) = \{L \mid L = L(G)$, where $G = (N_1, N_2, \ldots, N_n, T, R, S)$ is a i-PRLG$\}$.

Theorem 3.1 (Wood hierarchy). *For all $i \geq 1$, ${}_{par}\mathfrak{R}(i) \subset {}_{par}\mathfrak{R}(i+1)$.*

Proof. See [Wood–73]. □

For more information about n-parallel right-linear grammars, see [Wood–73].

Definition 3.29. *A scattered context grammar* (SCG, for short) is a special case of rewriting system, usually noted as $G = (N, T, S, P)$, where N, T and S are exactly the same as in the definition 3.11 of a grammar, but P is a finite set of production rules of the form

$$(A_1, A_2, \ldots, A_n) \to (w_1, w_2, \ldots, w_n), \quad n \geq 1, A_i \in N, w_i \in (N \cup T)^*, 1 \geq i \geq n.$$

Let $(A_1, A_2, \ldots, A_n) \to (w_1, w_2, \ldots, w_n) \in P$ and $x_i \in (N \cup T)^*, 1 \geq i \geq n + 1$. We write

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

Example 3.1.

$$G = (\{S, A, B, C\}, \{a, b, c\}, S, P)$$

where

$$P = \{S \to ABC, (A \to aA, B \to bB, C \to cC), (A \to a, B \to b, C \to c)\}$$

The language generated by grammar G is

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

3.4 Control Languages

Definition 3.30. Let $G = (V, P)$ be a rewriting system. Let Ψ be an alphabet of *rule labels* such that $\text{card}(\Psi) = \text{card}(P)$, and ψ be a bijection from P to Ψ. For simplicity, to express that ψ maps a rule, $u \to v \in P$, to ρ, where $\rho \in \Psi$, we write $\rho.u \to v \in P$; in other words, $\rho.u \to v$ means $\psi(u \to v) = \rho$.

If $u \to v \in P$ and $x, y \in V^*$, then $xuy \Rightarrow xvy$ $[u \to v]$ or simply $xuy \Rightarrow xvy$ $[\rho]$. Let there exists a sequence $x_0, x_1, \ldots, x_n \in V^*$ for some $n \geq 1$ such that $x_{i-1} \Rightarrow x_i$ $[\rho_i]$, where $\rho_i \in \Psi$, for $i = 1, \ldots, n$. Then G rewrites x_0 to x_n in n steps according to $\rho_1, \ldots, \rho_n$, symbolically written as $x_0 \Rightarrow^n x_n$ $[\rho_1 \ldots \rho_n]$.
Let Ξ be a *control language* over Ψ; that is $\Xi \subseteq \Psi^*$.

Definition 3.31. Let $G = (N, T, P, S)$ be a grammar defined in Definition 3.11. Let Ψ be an alphabet of rule labels and let Ξ be a control language. A *language generated by regulated grammar G by control language* Ξ is the set

$$L(G, \Xi) = \{w \mid w \in T^*, S \Rightarrow^n w\ [\rho_1, \ldots, \rho_n], \rho_1 \ldots \rho_n \in \Xi\}$$

Definition 3.32. Let $T = (Q, \Sigma, \delta, s, F)$ be a finite automaton. Let Ψ be an alphabet of rule labels and let Ξ be a control language. A *language generated by finite automaton T regulated by control language* Ξ is the set

$$L(T, \Xi) = \{w \mid w \in \Sigma^*, (s, w) \vdash_T^n (q, \varepsilon)\ [\rho_1, \ldots, \rho_n], \rho_1 \ldots \rho_n \in \Xi \text{ and } q \in F\}$$

Theorem 3.2.

$$REG = L(FA, REG) = L(REG, REG).$$

Proof. The proof can be found in [Sal–73] on page 184. □

Definition 3.33. Let $T = (Q, \Sigma, \Omega, \delta, s, \nabla, F)$ be a pushdown automaton. Let Ψ be an alphabet of rule labels and let Ξ be a control language. A *language generated by pushdown automaton T regulated by control language Ξ* is the set

$$L(T, \Xi) = \{w \mid w \in \Sigma^*, (s, w, \nabla) \vdash_T^n (q_F, \varepsilon, \gamma)\ [\rho_1, \dots, \rho_n], \rho_1 \dots \rho_n \in \Xi, q_F \in F, \gamma \in \Omega^*\}.$$

If it is useful to distinguish, T defines the following types of accepted languages:

1. $L(T, \Xi, 1)$ – the language accepted by the final state.
2. $L(T, \Xi, 2)$ – the language accepted by an empty pushdown.
3. $L(T, \Xi, 3)$ – the language accepted by the final state and an empty pushdown.

Theorem 3.3. *For any pushdown automaton T and context-free grammar G so that*

$$L(T) = L(G)$$

and for any regular language Ξ

$$CF = L(T, \Xi) \subset L(G, \Xi).$$

Proof. The proof of the first equality can be found in [Kol–04] on page 31 as Lemma 4.4.1. To prove the second relation we need to find regulated context-free grammar by regular language that generates language, that is not context-free.
Consider following context-free grammar $G = (N, T, P, S)$ where

- $N = \{S, A, B\}$,
- $T = \{a, b, c\}$,
- $P = \{1.S \to AB, 2.A \to aA, 3.B \to bBc, 4.A \to a, 5.b \to bc\}$

and the regular control language $\Xi = 1(23)^*45$. It is easy to verify, that $L(G, \Xi) = \{a^n b^n c^n \mid n \geq 1\}$. This language is well known not context-free language. □

The following theorem shows that regulation by control language can greatly increase the power of underlying model.

Theorem 3.4. *For every recursive enumerable language L there exists pushdown automaton T and linear control language Ξ so that $L = L(T, \Xi)$. Hence*

$$RE = L(CF, LIN).$$

Proof. Proof of this theorem can be found in [Med–00]. □

The following theorem postulates that any recursive enumerable language can be generated as an one-turn stack automaton regulated by a linear language. This theorem was first proved in [Med–00] and the proof is based on equivalence of regulated pushdown automata and queue grammars. But for implementation purposes we need the constructive alternative introduced here. This proof was published in [Rych–09].

Theorem 3.5. *Any recursive enumerable language L can be generated as $L = L(M, L_1, 3)$ where M is an OTSA and L_1 is a linear language.*

Proof. Let L be any recursive enumerable language and $L = L(G)$ where $G = (N, T, P, S)$ is type-0 grammar in Penttonen normal form (see def. 3.24). Let $M = (Q, \Sigma, \Omega, \delta, s, \nabla, F)$ be an OTSA, where

1. $Q = \{q, q_{in}, q_{out}\}$,
2. $\Sigma = T$,
3. $\Omega = T \cup N \cup \{\#\} \cup \{\nabla\}$, where $\# \notin \{N \cup T\}$,
4. $s = q$,
5. $\nabla \in \Omega$ is the initial stack symbol
6. $F = \{q_{out}\}$.
7. $\delta = \delta' \cup \delta_{in} \cup \delta_{out}$, where
   $\delta' = \{\langle a\rangle.aq \to qa \mid \text{for every } a \in T\} \cup \{\langle\#\rangle.q \to q_{in}\#\}$,
   $\delta_{in} = \{\langle A\rangle.q_{in} \to q_{in}A \mid \text{for every } A \in T \cup N \cup \{\#\}\} \cup \{\langle 2\rangle.q_{in} \to q_{out}\}$,
   $\delta_{out} = \{\langle\overline{A}\rangle.q_{out}A \to q_{out} \mid \text{for every } A \in T \cup N \cup \{\#\}\}$.

A control language L_1, which is linear, is defined by the following grammar $G_1 = (N_1, T_1, P_1, S_1)$:

1. $N_1 = \{S_1, K, M, M', O\}$,
2. $T_1 = \{\langle A\rangle, \langle\overline{A}\rangle \mid A \in T \cup N \cup \{\#\} \text{ and } \langle A\rangle \text{ is a label from } \Psi\} \cup \{\langle 2\rangle\}$,
3. $P_1 = P_a \cup P_{\langle\#\rangle} \cup P_b \cup P_c \cup P_d \cup P_{\overline{c}} \cup P_{\overline{d}} \cup P_e \cup P_f \cup P_g \cup P_h \cup P_{\langle 2\rangle}$, where
   $P_a = \{S_1 \to \langle a\rangle S_1 \mid \text{for every } a \in T\}$,
   $P_{\langle\#\rangle} = \{S_1 \to \langle\#\rangle K\}$,
   $P_b = \{K \to \langle A\rangle K\langle\overline{A}\rangle \mid \text{for every } A \in T \cup N\}$,
   $P_c = \{K \to \langle C\rangle M\langle\overline{C}\rangle \mid \text{for every rule in the form } CB \to CD \in P\}$,
   $P_d = \{M \to \langle B\rangle O\langle\overline{D}\rangle \mid \text{for every rule in the form } CB \to CD \in P\}$,
   $P_{\overline{c}} = \{K \to \langle D\rangle M'\langle\overline{C}\rangle \mid \text{for every rule in the form } D \to BC \in P\}$,
   $P_{\overline{d}} = \{M' \to O\langle\overline{B}\rangle \mid \text{for every rule in the form } D \to BC \in P\}$,

$P_e = \{K \to \langle C\rangle O\langle\bar{c}\rangle \mid$ for every rule in the form $C \to c \in P\}$,
$P_f = \{K \to \langle C\rangle O \mid$ for every rule in the form $C \to \varepsilon \in P\}$,
$P_g = \{O \to \langle A\rangle O\langle\overline{A}\rangle \mid$ for every $A \in T \cup N\}$,
$P_h = \{O \to \langle\#\rangle K\langle\overline{\#}\rangle\}$,
$P_{\langle 2\rangle} = \{K \to \langle 2\rangle\langle\overline{\#}\rangle\langle\overline{S}\rangle\}$.

Now, we prove two standard inclusions. First, $L \subseteq L(M, L_1)$. For every $w \in L$ there exists some successful derivation $S = w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n = w$ in L. We will construct the control string R as follows (for the sake of simplicity we omit $\langle$ and $\rangle$ if no confusion can arise)

$$R = w\#w_{n-1}\#\ldots\#w_1\#S\#\langle 2\rangle\overline{\#S\#w_1^R\#\ldots\#w_{n-1}^R\#w^R}.$$

It is easy to verify, that OTSA M under regulation of R reaches the final state and empties its pushdown (because $R = R'\langle 2\rangle\overline{rev(R')}$).

We need to prove that $R \in L(G_1)$. For every R_i:

$$R_0 = w\#\ K$$

$$R_1 = w\#w_{n-1}\#\ K\ \overline{\#w^R}.$$

$$\vdots$$

$$R_m = w\#w_{n-1}\#\ldots\#w_{n-m}\ K\ \overline{\#w_{n-(m-1)}^R\#\ldots\#w_{n-1}^R\#w^R}.$$

holds $S_1 \Rightarrow^* R_i$ by induction on i.
$i = 0$: $S_1 \Rightarrow^{|w|} wS_1 \Rightarrow w\#\ K$, hence $w\#K \in L(G_1)$.
$i = k$:

$$R_k = w\#w_{n-1}\#\ldots\#w_{n-k}\ K\ \overline{\#w_{n-(k-1)}^R\#\ldots\#w_{n-1}^R\#w^R}.$$

That is, $K \Rightarrow^* w_{n-(k+1)}\ O\ \overline{w_{n-k}^R} \Rightarrow w_{n-(k+1)}\#\ K\ \overline{\#w_{n-k}^R}$ by using rules from P_b to elements not affected in the rewriting of w_{n-k} to w_{n-k+1}. Then one or two rules from sets $P_c, P_d, P_{\overline{c}}, P_{\overline{d}}, P_e$ and P_f are used according to used rule from P. The rest rules are taken from P_g and finally one rule from P_h rewrites nonterminal O to K.

$$R_k \Rightarrow^* w\#w_{n-1}\#\ldots\#w_{n-k}\#w_{n-(k+1)}\#\ K\ \overline{\#w_{n-k}^R\#w_{n-(k-1)}^R\#\ldots\#w_{n-1}^R\#w^R} = R_{k+1}.$$

Let us see a short example. For the sake of simplicity we again omit $\langle$ and $\rangle$ if no confusion can arise. The derivation $S \Rightarrow AX \Rightarrow ABC \Rightarrow aBC \Rightarrow aDC \Rightarrow aDc \Rightarrow abc$ in grammar $G = (\{S, A, B, C, X\}, \{a, b, c\}, S, \{S \to AX, X \to BC, BC \to DC, A \to a, D \to b, C \to c\})$ results in

$$abc\#aDc\#aDC\#aBC\#ABC\#AX\#S\#\langle 2\rangle\overline{\#S\#XA\#CBA\#CBa\#CDa\#cDa\#cba}$$

as the control string.

The underlying OTSA under such derivation string operates as follows:
$a.aq \to qa : (abc, q, \nabla) \vdash (bc, q, a)$

$b.bq \rightarrow qb : (bc, q, a) \vdash (c, q, ab)$
$c.cq \rightarrow qc : (c, q, ab) \vdash (\varepsilon, q, abc)$
$\#.q \rightarrow q_{in}\# : (\varepsilon, q, abc) \vdash (\varepsilon, q_{in}, abc\#)$
$a.q_{in} \rightarrow q_{in}a : (\varepsilon, q_{in}, abc\#) \vdash (\varepsilon, q_{in}, abc\#a)$
$D.q_{in} \rightarrow q_{in}D : (\varepsilon, q_{in}, abc\#a) \vdash (\varepsilon, q_{in}, abc\#aD)$
$c.q_{in} \rightarrow q_{in}c : (\varepsilon, q_{in}, abc\#aD) \vdash (\varepsilon, q_{in}, abc\#aDc)$
$\#.q_{in} \rightarrow q_{in}\# : (\varepsilon, q_{in}, abc\#aDc) \vdash (\varepsilon, q_{in}, abc\#aDc\#)$
$\vdots$
$S.q_{in} \rightarrow q_{in}S : (\varepsilon, q_{in}, abc\#aDc\#aDC\#aBC\#ABC\#AX\#) \vdash$
$\vdash (\varepsilon, q_{in}, abc\#aDc\#aDC\#aBC\#ABC\#AX\#S)$
$\#.q_{in} \rightarrow q_{in}\# : (\varepsilon, q_{in}, abc\#aDc\#aDC\#aBC\#ABC\#AX\#S) \vdash$
$\vdash (\varepsilon, q_{in}, abc\#aDc\#aDC\#aBC\#ABC\#AX\#S\#)$
$\langle 2\rangle.q_{in} \rightarrow q_{out}\# : (\varepsilon, q_{in}, abc\#aDc\#aDC\#aBC\#ABC\#AX\#S\#) \vdash$
$\vdash (\varepsilon, q_{out}, abc\#aDc\#aDC\#aBC\#ABC\#AX\#S\#)$
$\overline{\#}.q_{out}\# \rightarrow q_{out} : (\varepsilon, q_{out}, abc\#aDc\#aDC\#aBC\#ABC\#AX\#S\#) \vdash$
$\vdash (\varepsilon, q_{out}, abc\#aDc\#aDC\#aBC\#ABC\#AX\#S)$
$\overline{S}.q_{out}S \rightarrow q_{out} : (\varepsilon, q_{out}, abc\#aDc\#aDC\#aBC\#ABC\#AX\#S) \vdash$
$\vdash (\varepsilon, q_{out}, abc\#aDc\#aDC\#aBC\#ABC\#AX\#)$
$\vdots$
$\overline{\#}.q_{out}\# \rightarrow q_{out} : (\varepsilon, q_{out}, abc\#) \vdash (\varepsilon, q_{out}, abc)$
$\overline{c}.q_{out}c \rightarrow q_{out} : (\varepsilon, q_{out}, abc) \vdash (\varepsilon, q_{out}, ab)$
$\overline{b}.q_{out}b \rightarrow q_{out} : (\varepsilon, q_{out}, ab) \vdash (\varepsilon, q_{out}, a)$
$\overline{a}.q_{out}a \rightarrow q_{out} : (\varepsilon, q_{out}, a) \vdash (\varepsilon, q_{out}, \nabla)$
so OTSA is in final state and has empty stack.

The derivation of control string in control language is

$$S_1 \Rightarrow aS_1 \Rightarrow abS_1 \Rightarrow abcS_1 \Rightarrow abc\#K \Rightarrow abc\#aK\overline{a} \stackrel{D\rightarrow b}{\Longrightarrow} abc\#aD\ O\ \overline{ba} \Rightarrow$$

$$\Rightarrow abc\#aDc\ O\ \overline{cba} \Rightarrow abc\#aDc\#\ K\ \overline{\#cba} \Rightarrow \ldots \Rightarrow$$

$$\Rightarrow abc\#aDc\#aDC\#aBC\#ABC\#AX\#S\#\ K\ \overline{\#XA\#CBA\#CBa\#CDa\#cDa\#cba} \Rightarrow$$

$$\Rightarrow abc\#aDc\#aDC\#aBC\#ABC\#AX\#S\#\langle 2\rangle\overline{\#S\#XA\#CBA\#CBa\#CDa\#cDa\#cba}.$$

The second inclusion is $L(M, L_1) \subseteq L$. Let us suppose that the word $w_m = x_1x_2\ldots x_p \in L(M, L_1)$. We will prove the following theorem by induction on n:
For any integer n, the word $w_{m-n}, 1 \leq n \leq m$ in the control string R_n

$$R_0 = w_m\#\ K$$

$$R_1 = w_m\#w_{m-1}\#\ K\ \overline{\#w_m^R}$$

$$\vdots$$

$$R_n = w_m\#w_{m-1}\#\ldots\#w_{m-n}\#\ K\ \overline{\#w_{m-(n-1)}^R\#\ldots\#w_{m-1}^R\#w_m^R}$$

can be derived from w_{m-n} to w_m in n steps in G, hence $w_{m-n} \Rightarrow^n w_m$ in G.
$n = 0$: $w_m \Rightarrow^0 w_m$.
$n = k$:

$$R_k = w_m \ldots \# w_{m-k} \#\ K\ \overline{\# w^R_{m-(k-1)} \# \ldots \# w^R_m}$$

$w_{m-k} = y_1 y_2 \ldots y_q$. As M is OTSA, the sequence of pushed symbols onto the stack will be popped in reverse order. Hence,

$$K \Rightarrow^* z_1 z_2 \ldots z_r \#\ K\ \overline{\# y_q \ldots y_2 y_1}$$

where $z_i \in (N \cup T)$ and there exists index i such as $y_1 = z_1, \ldots, y_i = z_i$ and

$$K \Rightarrow^i y_1 y_2 \ldots y_i\ K\ \overline{y_i \ldots y_2 y_1},$$

according to i applications of rules from P_b. Now there are 4 possible rules to apply $P_c, P_{\overline{c}}, P_e$, and P_f. The next step has to generate $\overline{y_{i+1}}$ on the right side of K.

1. P_c: $K \Rightarrow C\ M\ \overline{C} \Rightarrow CB\ O\ \overline{DC} \Leftrightarrow CB \rightarrow CD \in P$ and $y_{i+2} = D$ and $y_{i+1} = C$.
2. $P_{\overline{c}}$: $K \Rightarrow D\ M'\ \overline{C} \Rightarrow D\ O\ \overline{BC} \Leftrightarrow D \rightarrow BC \in P$ and $y_{i+2} = B$ and $y_{i+1} = C$.
3. P_e: $K \Rightarrow C\ O\ \overline{c} \Leftrightarrow C \rightarrow c \in P$ and $y_{i+1} = c$.
4. P_f: $K \Rightarrow C\ O\ \ \Leftrightarrow C \rightarrow \varepsilon \in P$.

Now there are two possible rules to apply. From P_g and P_h. As there are still some elements of y_k on the right side of O, we have to use rules from P_g until there is complete $\overline{y_q \ldots y_2 y_1}$ generated on the right side of O. Consequently, there exists index j such that $y_j = z_k, \ldots, y_q = z_r$. Then, the last rule from P_h generates $\#$ and $\overline{\#}$ on both sides of O and O rewrites to K. Then,

$$R_k \Rightarrow^* w_m \# \ldots \# w_{m-k} \# w_{m-(k+1)}\ K\ \overline{\# w^R_{m-k} \# w^R_{m-(k-1)} \# \ldots \# w^R_m} = R_{k+1}$$

and $w_{m-k} \Rightarrow w_{m-(k+1)}$ in G.

So, the complete control string will be

$$R = w_n \# w_{n-1} \# \ldots \# w_1 \# S \# \langle 2 \rangle \overline{\# S \# w^R_1 \# \ldots \# w^R_{n-1} \# w^R_n}$$

and there exists the derivation $S \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_{n-1} \Rightarrow w_n$ in G and $w_n \in L = L(G)$. □

Remark 1. However, we can in the same manner define regulated Turing machine, it is of little or no interest because it is as powerful as ordinary Turing machine.

3.5 Grammar Systems

While discussing grammatical models of computational distribution and concurrency we cannot skip grammar systems. The grammar systems are cooperating sets of grammars that either work together with common sentence string (CD grammars systems) or exchange information between them (PC grammar systems). We introduce only basic definitions and main theorems in this section. Much more information can be found in [Roz–73], volume 2.

Definition 3.34. A *cooperating distributed grammar system of degree* n $(n \geq 1)$, *CD grammar system* for short, is a special case of rewriting system, usually noted as

$$\Gamma = (N, T, S, P_1, \ldots, P_n),$$

where N, T and S are defined as usual and P_i is a finite set of context-free productions, called *component* of Γ, for each $i \in \{1, \ldots, n\}$.
For each $i = 1, \ldots, n$, we denote *terminating derivation* by the ith component

$$x \;{}_i\!\Rightarrow^t y$$

where $x \Rightarrow^* y$ in $G_i = (N, T, S, P_i)$, $y \in (N \cup T)^*$ and $y \not\Rightarrow z$ for all $z \in (N \cup T)^*$ in G_i.
For each $i = 1, \ldots, n$, we denote *k-step derivation* by the ith component

$$x \;{}_i\!\Rightarrow^{=k} y$$

where $x \Rightarrow^k y$ in $G_i = (N, T, S, P_i)$ and $y \in (N \cup T)^*$.
For each $i = 1, \ldots, n$, we denote *at most k-step derivation* by the ith component

$$x \;{}_i\!\Rightarrow^{\leq k} y$$

where $x \Rightarrow^j y$ in $G_i = (N, T, S, P_i)$, for some $j \leq k$ and $y \in (N \cup T)^*$.
For each $i = 1, \ldots, n$, we denote *at least k-step derivation* by the ith component

$$x \;{}_i\!\Rightarrow^{\geq k} y$$

where $x \Rightarrow^j y$ in $G_i = (N, T, S, P_i)$, for some $j \geq k$ and $y \in (N \cup T)^*$.
In short, set of derivation modes D

$$D = \{*,\ t\} \cup \{\leq k,\ =k,\ \geq k \mid k = 1, 2, \ldots\}.$$

Set of possible derivations F is defined by

$$F(G_j, u, f) = \{v \mid u \;{}_j\!\Rightarrow^f v\},\ \text{where } j \in \{1, 2, \ldots, n\}, f \in D, u \in (N \cup T)^*$$

and finally language generated by CD-grammar system Γ in derivation mode f is defined by

$L_f(\Gamma) = \{w \in T^* \mid \exists v_0, v_1, \dots v_m : v_i \in F(G_{j_i}, v_{i-1}, f), i \in \{1, \dots, m\}, j_i \in \{1, \dots, n\},$
$v_0 = S, v_m = w, m \geq 1\}$
Denotation of CD language families is

$$CD_x^y(f)$$

where f is derivation mode, $f \in D$,

$$y = \begin{cases} \text{nothing - no } \varepsilon\text{-productions,} \\ \varepsilon \text{ - } \varepsilon\text{-productions allowed.} \end{cases}$$

$$x = \begin{cases} n \text{ - degree at most } n, n \geq 1, \\ \infty \text{ - the number of components is not limited.} \end{cases}$$

Example 3.2.

$$\Gamma = (\{S, A, A', B, B'\}, \{a, b, c\}, S, P_1, P_2)$$

where

$$P_1 = \{S \to S, S \to AB, A' \to A, B' \to B\}$$

$$P_2 = \{A \to aA'b, B \to cB', A \to ab, B \to c\}.$$

The languages generated by different derivation modes are

$$L_f(\Gamma) = \{a^n b^n c^m \mid m, n \geq 1\}, f \in \{=1, \geq 1, *, t\} \cup \{\leq k \mid k \geq 1\}$$

$$L_{=2}(\Gamma) = L_{\geq 2}(\Gamma) = \{a^n b^n c^n \mid n \geq 1\}$$

$$L_{=k}(\Gamma) = L_{\geq k}(\Gamma) = \emptyset, k \geq 3.$$

Theorem 3.6. *Generative power of CD grammar systems*

- $CD_\infty^y(f) = CF$*, for all* $f \in \{=1, \geq 1, *\} \cup \{\leq k \mid k \geq 1\}$,
- $CF = CD_1^y(f) \subset CD_2^y(f) \subseteq CD_r^y(f) \subseteq CD_\infty^y(f) \subseteq Mat$*, for all* $f \in \{=k, \geq k \mid k \geq 2\}, r \geq 3$,
- $CD_r^y(\geq k) \subseteq CD_r^y(\geq k+1)$,
- $CD_\infty^y(\geq) \subseteq CD_\infty^y(=)$,
- $CF = CD_1^y(t) = CD_2^y(t) \subset CD_3^y(t) = CD_\infty^y(t) \subseteq ET0L$

Proof. Proof can be found in [Roz–73], volume 2, chapter 4. □

Definition 3.35. A *parallel communicating grammar system of a degree n* ($n \geq 1$), *PC grammar system* for short, is a special case of rewriting system, usually noted as

$$\Gamma = (N, K, T, (S_1, P_1), \ldots, (S_n, P_n)),$$

where N and T are defined as usual, K is a finite set of *query* symbols, $K = \{Q_1, \ldots Q_n\}$, P_i is a finite set of productions of the form

$$A \rightarrow x$$

where $A \in N$ and $x \in (N \cup T \cup K)^*$, for all $i = 1, \ldots, n$ and S_i is the start symbol of the ith component, $S_i \in N$ for all $i = 1, \ldots, n$. N, T are defined as usual, N, T and K are pairwise disjoint.
Generating step (g-step) is defined as

$$(x_1, \ldots, x_n) \ {}_g\!\Rightarrow (y_1, \ldots, y_n)$$

if

- either $x_i \Rightarrow y_i$ in $G_i = (N \cup K, T, S_i, P_i)$,
- or $x_i = y_i \in T^*$

for all $1 \leq i \leq n$.
Communicating step (c-step) is defined as

$$(x_1, \ldots, x_n) \ {}_c\!\Rightarrow (y_1, \ldots, y_n)$$

if we set $z_i = x_i$ for all $i = 1, \ldots, n$ and if $alph(x_i) \cap K \neq \emptyset$ and for each Q_j in x_i $alph(x_j) \cap K = \emptyset$, then for each Q_j in x_i

1. set $x_j = S_j$,
2. replace Q_j with x_j in x_i,
3. set z_i to the string resulting from (2.).

then $y_i = z_i$, for all $i = 1, \ldots, n$.
Direct derivation

$$(x_1, \ldots, x_n) \ \Rightarrow (y_1, \ldots, y_n)$$

is defined if

$$(x_1, \ldots, x_n) \ {}_g\!\Rightarrow (y_1, \ldots, y_n)$$

or

$$(x_1, \ldots, x_n) \ {}_c\!\Rightarrow (y_1, \ldots, y_n).$$

The language generated by parallel communicating grammar system Γ of degree n is defined by

$$L(\Gamma) = \{x \in T^* \mid (S_1, S_2, \ldots, S_n) \Rightarrow^* (x, \alpha_2, \ldots, \alpha_n), \alpha_i \in (N \cup T \cup K)^*, 2 \leq i \leq n\}.$$

If after communicating, each component that has sent its string to another component returns to its axiom, then we call this grammar *returning* PC Grammar System. Generated language is denoted by $L_r(\Gamma)$.
If after communicating, each component that has sent its string to another component continues to process the current string, then we call this grammar *non-returning* PC Grammar System. Generated language is denoted by $L_{nr}(\Gamma)$.
Denotation of PC language families is

$$XPC_nY$$

where

$$X = \begin{cases} \text{N -- non-returning mode} \\ \varepsilon \text{ -- returning mode.} \end{cases}$$

n– number of components

Y – specification of the type of productions (REG, LIN, CF)

Example 3.3.

$$\Gamma = (\{S_1, S_1', S_2, S_3\}, K, \{a, b\}, (S_1, P_1), (S_2, P_2), (S_3, P_3))$$

where

$$P_1 = \{S_1 \to abc, S_1 \to a^2b^2c^2, S_1 \to aS_1', S_1 \to a^3Q_2,$$
$$S_1' \to aS_1', S_1' \to a^3Q_2, S_2 \to b^2Q_3, S_3 \to c\}$$

$$P_2 = \{S_2 \to bS_2\}$$
$$P_3 = \{S_3 \to bS_3\}$$
$$L_r(\Gamma) = L_{nr}(\Gamma) = \{a^nb^nc^n \mid n \geq 1\}.$$

Theorem 3.7. *Generative power of PC grammar systems*

- $PC_nREG \subset PC_{n+1}REG$, *for* $n \geq 1$,
- $NPC_\infty CF \subseteq PC_\infty CF$,
- $L(Mat) \subset PC_\infty CF$,
- $L(LIN) \subset PC_\infty REG$.

Proof. Proof can be found in [Roz–73], volume 2, chapter 4. □

Part II
New Grammatical Models of Distribution and Concurrency

4 Theoretical Results

In this chapter, we discuss right-linear grammar that starts its derivations from start strings rather than single symbols. Specifically, we study these grammars regulated by regular languages. We demonstrate that the language family generated by these grammars with start strings of length n or shorter is properly included in the language family generated by these grammars with start strings of length $n+1$ or shorter, for all $n \geq 1$. From a broader perspective, by obtaining this infinite hierarchy of language families, we contribute to a classical trend of the formal language theory that demonstrates that some properties of grammars affect the language families that the grammars generate.

Surprisingly, however, if during the derivation of any sentence from the generated language, these grammars change the position of rewriting finitely many times, they just generate the family of regular languages no matter how long their start strings are. In other words, only if the number of these changes is unlimited, the above hierarchy holds true.

The key parts of this chapter were published in [Med–08] and [Rych–08].

4.1 Definitions

Definition 4.1. Let $n \geq 1$. A *linear grammar with a start string of length n*, n-LG for short, is a quadruple $G = (N, T, R, S)$, where N and T are alphabets such that $N \cap T = \emptyset, S \in N^+, |S| \leq n$, and R is a finite set of productions of the form $A \to x$, where $A \in N$ and $x \in T^*(N \cup \{\varepsilon\})T^*$. Set $V = T \cup N$.

Let Ψ be an alphabet of rule labels such that card(Ψ) = card(R), and ψ be a bijection from R to Ψ. For simplicity, to express that ψ maps a rule $A \to x \in R$, to ρ, where $\rho \in \Psi$, we write $\rho.A \to x \in R$; in other words, $\rho.A \to x$ means $\psi(A \to x) = \rho$.

If $\rho.A \to x \in R$ and $u, v \in V^*$, then we write $uAv \Rightarrow uxv\ [\rho]$ in G.

Let $\chi \in V^*$. Then G makes the zero-step derivation from χ to χ according to ε, symbolically written as $\chi \Rightarrow^0 \chi\ [\varepsilon]$. Let there exist a sequence of derivation steps $\chi_0, \chi_1, \ldots, \chi_n$ for some $n \geq 1$ such that $\chi_{i-1} \Rightarrow \chi_i\ [\rho_i]$, where $\rho_i \in \Psi$, for all $i = 1, \ldots, n$, then G makes n derivation steps from χ_0 to χ_n according to $\rho_1 \ldots \rho_n$, symbolically written as $\chi_0 \Rightarrow^n \chi_n\ [\rho_1 \ldots \rho_n]$. If for some $n \geq 0, \chi_0 \Rightarrow^n \chi_n\ [\rho]$, where $\rho \in \Psi^*$ and $|\rho| = n$, we write $\chi_0 \Rightarrow^* \chi_n\ [\rho]$.

We call a derivation $S \Rightarrow^* w$ *successful*, if and only if, $w \in T^*$.

Let Ξ be a control language over Ψ; that is, $\Xi \subseteq \Psi^*$.

Under the regulation by Ξ, the language that G generates is denoted by $L(G, \Xi)$ and defined as

$$L(G, \Xi) = \{w \mid S \Rightarrow^* w\ [\rho], \rho \in \Xi, w \in T^*\}.$$

Let i be a positive integer and X be a family of languages. Set

$$\mathfrak{L}(X, i) = \{L \mid L = L(G, X), \text{where } G \text{ is a } i\text{-LG}\}.$$

In the same manner we define a *right-linear grammar with a start string of length n*, n-RLG for short, where R is a finite set of productions of the form $A \to x$, where $A \in N$ and $x \in T^*(N \cup \{\varepsilon\})$ and define

$$\mathfrak{R}(X, i) = \{L \mid L = L(G, X), \text{where } G \text{ is a } i\text{-RLG}\}.$$

Specifically, $\mathfrak{R}(REG, i)$ and $\mathfrak{L}(REG, i)$ are central to this paper, where REG denotes the family of regular languages.

Definition 4.2. Let $G = (N, T, R, S)$ be an n-LG for some $n \geq 1$ (See Definition 4.1). $G = (N_1, N_2, \ldots, N_n, T, R_1, R_2, \ldots, R_n, S)$ is a *distributed n-LG*, $n-_{dis}$LG for short, if

- $N = N_1 \cup N_2 \cup \ldots \cup N_n$, where $N_i, 1 \leq i \leq n$ are pairwise disjoint nonterminal alphabets,
- $S = X_1 X_2 \ldots X_n,\ X_i \in N_i, 1 \leq i \leq n$,
- $R = R_1 \cup R_2 \cup \ldots \cup R_n$,
  such that for every $A \to xBy \in R_i,\ A, B \in N_i$, for some $1 \leq i \leq n; x, y \in T^*$
  and for every $A \to a \in R,\ A \in N, a \in T^*$.

Set $\Psi_i = \{\rho \mid \rho.A \to aBb \in R_i$ or $\rho.A \to a \in R_i$, where $A, B \in N_i$ and $a, b \in T^*\}$.

In the same manner we define a *distributed n-RLG*, $n{-}_{dis}$RLG for short, if this grammar is $n{-}_{dis}$LG and all rules are right-linear.

Definition 4.3. (See [Das–89]) For $n \geq 1$, a *linear simple matrix grammar of degree* n, n-LSM for short, is an (n+3)-tuple $G = (N_1, \ldots N_n, T, S, P)$ where

- $N_i, 1 \leq i \leq n$ are pairwise disjoint nonterminal alphabets,
- T is a terminal alphabet, $N_i \cap T = \emptyset, 1 \leq i \leq n$,
- $S \notin N_1 \cup \ldots \cup N_n$ is the start symbol,
- P is a finite set of rules. P contains three kinds of rules
  1. $S \to x, \quad x \in T^*$,
  2. $S \to X_1 \ldots X_n, \quad X_i \in N_i, 1 \leq i \leq n$,
  3. $(X_1 \to x_1, X_2 \to x_2, \ldots, X_n \to x_n)$,
     $X_i \in N_i, x_i \in T^* N_i T^* \cup T^*, 1 \leq i \leq n$.

For $x, y \in (N \cup T \cup \{S\})^*, x \Rightarrow y$ if and only if

- either $x = S$ and $S \to y \in P$,
- or $x = y_1 X_1 \ldots y_n X_n, y = y_1 x_1 \ldots y_n x_n$,
  where $y_i \in T^*, x_i \in T^* N_i T^* \cup T^*, X_i \in N_i, 1 \leq i \leq n$
  and $(X_1 \to x_1, \ldots, X_n \to x_n) \in P$.

In the same manner we define a *right-linear simple matrix grammar of degree* n, n-RLSM for short, if in definition of P the last rule is

3. $(X_1 \to x_1, X_2 \to x_2, \ldots, X_n \to x_n)$,
   $X_i \in N_i, x_i \in T^* N_i \cup T^*, 1 \leq i \leq n$.

For more information about simple matrix grammars, see [Das–89].

Definition 4.4. Let $i \geq 1$ and X be a family of languages. Let $L(G, \Xi)$ be a language generated by G and regulated by Ξ (See definition 3.11). Set

- $\mathfrak{R}(X, i) = \{L \mid L = L(G, \Xi)$, where $G = (N, T, R, S)$ is a i-RLG and $\Xi \in X\}$.
- $\mathfrak{L}(X, i) = \{L \mid L = L(G, \Xi)$, where $G = (N, T, R, S)$ is a i-LG and $\Xi \in X\}$.
- ${}_{dis}\mathfrak{R}(X, i) = \{L \mid L = L(G, \Xi)$,
  where $G = (N_1, N_2, \ldots, N_n, T, R_1, R_2, \ldots, R_n, S)$
  is a $i{-}_{dis}$RLG and $\Xi \in X\}$.

- ${}_{dis}\mathfrak{L}(X, i) = \{L \mid L = L(G, \Xi)$,
  where $G = (N_1, N_2, \ldots, N_n, T, R_1, R_2, \ldots, R_n, S)$
  is a $i-_{dis}$LG and $\Xi \in X\}$.
- ${}_{SM}\mathfrak{R}(i) = \{L \mid L = L(G)$,
  where $G = (N_1, N_2, \ldots, N_n, T, R, S)$ is a i-RLSM$\}$.
- ${}_{SM}\mathfrak{L}(i) = \{L \mid L = L(G)$,
  where $G = (N_1, N_2, \ldots, N_n, T, S, P)$ is a i-LSM$\}$.

4.2 New Results

Lemma 4.1. *For every n-LG $G = (N, T, R, S)$, there exists an equivalent $n-_{dis}LG$ $G' = (N'_1, N'_2, \ldots, N'_n, T', R'_1, R'_2, \ldots, R'_n, S')$ such that $L(G) = L(G')$.*

Proof. We will define nonterminals of G' in the form (A, k) so that $(A, k) \in N'_k$. Hence,

- $N'_j = \{(A, j) \mid A \in N\}$, where $1 \leq j \leq n$;
- $T' = T$;
- $R'_j = \{(A, j) \to x(B, j)y \mid A \to xBy \in R$,
  $(A, i), (B, i) \in N'_j,\ x, y \in T^*\}$ where $1 \leq j \leq n$;
- $S' = (A_1, 1)(A_2, 2) \ldots (A_n, n)$, where $S = A_1A_2 \ldots A_n$.

For $G' = (N'_1, N'_2, \ldots, N'_n, T', R', S')$ holds $N'_i \cap N'_j = \emptyset$ for $i \neq j,\ 1 \leq i, j \leq n$. For every derivation $a \Rightarrow b\ [\rho],\ a, b \in \{N \cup T\}^*,\ \rho.A \to xBy \in R,\ x, y \in T^*,\ A, B \in N$ of grammar G there always exists equivalent derivation in G' in form $a' \Rightarrow b'\ [\rho'],\ a', b' \in \{N' \cup T'\}^*,\ \rho'.(A, i) \to x(B, i)y \in R',\ x, y \in T'^*,\ (A, i), (B, i) \in N'_i$. □

Lemma 4.2. *For every $n-_{dis}LG$ $G' = (N'_1, N'_2, \ldots, N'_n, T', R'_1, R'_2, \ldots, R'_n, S')$, there exists an equivalent n-LG $G = (N, T, R, S)$ such that $L(G) = L(G')$.*

Proof. We define grammar $G = (N, T, R, S)$ in the following way

- $N = N'_1 \cup N'_2 \cup \ldots \cup N'_n$,
- $T = T'$,
- $R = R'_1 \cup R'_2 \cup \ldots \cup R'_n$,
- $S = A_1A_2 \ldots A_n$, where $S' = A_1A_2 \ldots A_n \in R'$.

A rigorous proof that $L(G) = L(G')$ is left to the reader. □

Theorem 4.1. *For all $n \geq 1$, $\mathcal{L}(n-_{dis}LG) = \mathcal{L}(n\text{-}LG)$.*

Proof. This theorem directly follows from Lemma 4.1 and Lemma 4.2. □

Theorem 4.2. *For all $n \geq 1$, $\mathcal{L}(n{-}_{dis}RLG) = \mathcal{L}(n\text{-}RLG)$.*

Proof. This theorem directly follows from Theorem 4.1. □

Lemma 4.3. *Let $i \geq 1$. $_{dis}\mathfrak{L}(REG, i) \subseteq {}_{SM}\mathfrak{L}(i)$. That is, for every $n{-}_{dis}LG$ $G = (N_1, \ldots, N_n, T, R_1, \ldots, R_n, S)$ regulated by regular language Ξ there exists equivalent n-LSM $G' = (N'_1, \ldots, N'_n, T', S', P')$ such that $L(G) = L(G')$.*

Proof. Let $\Xi = L(G_\Xi), G_\Xi = (N_\Xi, T_\Xi, R_\Xi, S_\Xi)$. Let $R = R_1 \cup R_2 \cup \ldots \cup R_n$. We will define grammar $G' = (N'_1, \ldots, N'_n, T', S', P')$ this way:

- $N'_1 = \{[A, X] \mid A \in N_1, X \in N_\Xi\}$,
- $N'_i = N_i, 2 \leq i \leq n$,
- $T' = T$,
- $P'_1 = \{([A_1, X], A_2, \ldots, A_n) \rightarrow (u[B_1, Y]v, A_2, \ldots, A_n) \mid$
  $\mid A_i \in N_i, 1 \leq i \leq n,\ X, Y \in N_\Xi$ and
  $f.A_1 \rightarrow uB_1v \in R_1,\ X \rightarrow fY \in R_\Xi,$
  $u, v \in T^*\}$,
- $P'_2 = \{([A_1, X], A_2, \ldots, A_j, \ldots, A_n) \rightarrow$
  $\rightarrow ([A_1, Y], A_2, \ldots, uB_jv, \ldots, A_n) \mid$
  $\mid A_i \in N_i, 1 \leq i \leq n,\ 2 \leq j \leq n,\ X, Y \in N_\Xi$
  and $f.A_j \rightarrow uB_jv \in R_j,\ X \rightarrow fY \in R_\Xi,$
  $u, v \in T^*\}$,
- $P' = P'_1 \cup P'_2 \cup \{S' \rightarrow [X_1, S_\Xi]X_2 \ldots X_n \mid$
  $|S = X_1 \ldots X_n \in G,\ X_i \in N_i,\ 1 \leq i \leq n\}$.

Note that P'_1 is a special case of P'_2 with $j = 1$.

Let $L_n(G) = \{x \mid S \Rightarrow^n x \text{ in } G,\ x \in \{N \cup T\}^*\}$ and $L_n(G') = \{x \mid S' \Rightarrow^{n+1} x \text{ in } G',\ x \in \{N' \cup T'\}^*\}$. We will prove that $L_n(G) = h(L_n(G'))$ for every $n \geq 0$, where h is surjective function $h : \{N'_1 \cup \ldots \cup N'_n \cup T'\} \rightarrow \{N_1 \cup \ldots \cup N_n \cup T\}$ defined as

$$h(w) = \begin{cases} A, & \text{if } w \in N'_1, w = [A, Y], \\ w, & \text{otherwise.} \end{cases}$$

First we will prove that $L_n(G) \subseteq h(L_n(G'))$ by induction on n:
Let $n = 0$.
$L_0(G) = \{X_1X_2 \ldots X_n\}$, $L_0(G') = \{[X_1, Y]X_2 \ldots X_n\}$ because $S' \rightarrow [X_1, Y]X_2 \ldots X_n \in P'$ and, therefore,
$h(L_0(G')) = \{X_1X_2 \ldots X_n\} = L_0(G)$.
Let us suppose that the claim holds for all $n \leq k$, where k is a non-negative integer.
Let $n = k + 1$.

Consider $w \in L_{k+1}(G)$ and a derivation $S \Rightarrow^k v \Rightarrow w$ in G, so that $v \Rightarrow w$ $[p]$, where $v = C_1C_2 \dots C_{i-1}XC_{i+1} \dots C_n$, $w = C_1C_2 \dots C_{i-1}uYvC_{i+1} \dots C_n, C_j \in N_j \cup \{T\}^*, 1 \leq j \leq n, p.X \to uYv \in R, A \to pB \in R_\Xi$. From the induction step, $v \in h(L_k(G'))$. Since $([C_1, A]C_2 \dots C_{i-1}XC_{i+1} \dots C_n) \to ([C_1, B]C_2 \dots C_{i-1}uYvC_{i+1} \dots C_n) \in P'$, we have $w \in h(L_{k+1}(G'))$.

Now we prove that $L_n(G) \supseteq h(L_n(G'))$ by induction on $n \geq 0$:
Let $n = 0$. By analogy with the previous part of this proof.
Let us suppose that our claim holds for all $n \leq k$, where k is a non-negative integer.
Let $n = k + 1$.
Consider $w \in L_{k+1}(G')$ and a derivation $S \Rightarrow^k v \Rightarrow w$ in G', where
$v = [C_1, A]C_2 \dots C_{i-1}XC_{i+1} \dots C_n$, $w = [C_1, B]C_2 \dots C_{i-1}uYvC_{i+1} \dots C_n, C_j \in N_j \cup \{T\}^*$, $1 \leq j \leq n$. From the induction step, $h(v) \in L_k(G)$. Since $p.X \to uYv \in R, A \to pB \in R_\Xi$, we have $h(w) \in L_{k+1}(G)$. □

Lemma 4.4. *Let $i \geq 1$. ${}_{dis}\mathfrak{L}(REG, i) \supseteq {}_{SM}\mathfrak{L}(i)$ That is, for every n-LSM $G' = (N'_1, \dots, N'_n, T', S', P')$ there exists equivalent $n-{}_{dis}LG$ $G = (N_1, \dots, N_n, T, R_1, \dots, R_n, S)$ regulated by regular language Ξ such that $L(G) = L(G')$.*

Proof. G is defined in this way:

- $N_i = N'_i, 1 \leq i \leq n$;
- $T = T'$;
- $S = S'$;
- $R_i = \{r_{ij}.A_i \to u_iB_iv_i \mid$ for the jth rule
  $(A_1, \dots, A_i, \dots, A_n) \to$
  $\to (u_1B_1v_1, \dots, u_iB_iv_i, \dots, u_nB_nv_n) \in P'$,
  $u_i, v_i \in T^*,\ 1 \leq j \leq |P'|\},\ 1 \leq i \leq n.$

where $\Xi = L(G_\Xi), G_\Xi = (N_\Xi, T_\Xi, R_\Xi, S_\Xi)$ is defined as follows:

- $N_\Xi = \{Q\} \cup \{Q_{ij} \mid 1 \leq i \leq n-1,\ 1 \leq j \leq |P'|\}$;
- $T_\Xi = \{r_{ij} \mid 1 \leq i \leq n,\ 1 \leq j \leq |P'|\}$;
- $R_\Xi = \{Q \to r_{1j}Q_{1j} \mid 1 \leq j \leq |P'|\} \cup$
  $\cup \{Q_{ij} \to r_{i+1j}Q_{i+1j} \mid 1 \leq i \leq n-2,$
  $1 \leq j \leq |P'|\} \cup \{Q_{n-1j} \to r_{nj}Q \mid 1 \leq j \leq |P'|\}$;
- $S_\Xi = Q$.

□

Theorem 4.3. *For all $i \geq 1$, ${}_{dis}\mathfrak{L}(REG, i) = {}_{SM}\mathfrak{L}(i)$.*

Proof. This theorem directly follows from Lemma 4.3 and Lemma 4.4 □

Theorem 4.4. *For all* $i \geq 1$, ${}_{dis}\mathfrak{R}(REG, i) = {}_{SM}\mathfrak{R}(i)$.

Proof. This theorem directly follows from Theorem 4.3. □

Theorem 4.5. *For all* $i \geq 1$, ${}_{SM}\mathfrak{L}(i) \subset {}_{SM}\mathfrak{L}(i+1)$.

Proof. See [Das–89]. □

The main result of this paper follows next.

Theorem 4.6. *For all* $i \geq 1$,

$$\mathfrak{L}(REG, i) \subset \mathfrak{L}(REG, i+1).$$

Proof. This theorem follows from Theorems 4.1, 4.3 and 4.5. □

Theorem 4.7. *For all* $i \geq 1$,

$$\mathfrak{R}(REG, i) \subset \mathfrak{R}(REG, i+1).$$

Proof. This theorem follows from Theorem 4.6. □

Let G be an $n-_{dis}$RLG satisfying Definition 4.2. Let $S \Rightarrow^* w\ [\sigma]$, $w \in T^*, \sigma = \rho_1\rho_2\ldots\rho_m$, for some $m \geq 1, 1 \leq i \leq m, \rho_i \in \Psi, \sigma \in \Xi$.
Set

$$d = card(\{\rho_j\rho_{j+1} \mid j = 1, \ldots, m-1,\ \rho_j \in \Psi_k, \rho_{j+1} \in \Psi_h, k \neq h\}).$$

Then, during the generation of $w \in L(G, \Xi)$ by $S \Rightarrow^* w\ [\sigma]$, *$G$ changes the derivation position d times*. If there is a constant $k \geq 0$ such that for every $x \in L(G, \Xi)$ there is a generation of x during which G changes the derivation position k or fewer times, then the *generation of $L(G, \Xi)$ by G requires no more than k changes of derivation positions*. Let k be the minimal possible than we write $d(G) = k$.

Let $i \geq 1, k \geq i-1$ and X be a family of languages. Set

- $\mathfrak{R}(X, i, k) = \{L \mid L = L(G, \Xi)$, where $G = (N, T, R, S)$ is a i-RLG, $\Xi \in X$ and $d(G) = k$, the generation of $L(G, \Xi)$ by G requires no more than k changes of derivation positions$\}$.
- ${}_{dis}\mathfrak{R}(X, i, k) = \{L \mid L = L(G, \Xi)$, where $G = (N_1, N_2, \ldots, N_n, T, R_1, R_2, \ldots, R_n, S)$ is a $i-_{dis}$RLG, $\Xi \in X$ and $d(G) = k$, the generation of $L(G, \Xi)$ by G requires no more than k changes of derivation positions$\}$.

Theorem 4.8. *Let* $i \geq 1, k \geq 0$. *Then,* $\mathfrak{R}(REG, i, k) = {}_{dis}\mathfrak{R}(REG, i, k)$.

Proof. This proof is analogous to the proof of Theorem 4.1. □

Definition 4.5. Let G be an $n-_{dis}$RLG $G = (N_1, N_2, \ldots, N_n, T, R_1, R_2, \ldots, R_n, S)$ regulated by regular language Ξ. Let $\Xi = L(H), H = ({}_HN, {}_HT, {}_HS, {}_HP)$. Let $A, B \in {}_HN$.

We write $A \ {}_i\!\Rightarrow B$ and say *B is achievable from A in i-th component of G in one derivation step* if and only if there exists derivation $A \Rightarrow xB, \ x \in {}_HT$ in H and x is the label of some rule from R_i.

We write $A \ {}_i\!\Rightarrow^* B$ and say *B is achievable from A in i-th component of G* if and only if there exists derivation $A \Rightarrow^* xB, \ x \in {}_HT^*$ in H, and x are the labels of rules from R_i.

We write $i(A) = \{B \mid B \in {}_HN \text{ and } A \ {}_i\!\Rightarrow^* B\}$.

Theorem 4.9. *For any $n, k \geq 1$, $\mathfrak{R}(REG, n, k) \subseteq REG$. That is, let $G = (N_1, N_2, \ldots, N_n, T, R_1, R_2, \ldots, R_n, S)$ be an $n-_{dis}RLG$ regulated by regular language Ξ. Let generation of $L(G, \Xi)$ by G require no more than k changes of derivation positions. Then, there exists an equivalent regular grammar $G' = (N', T', S', P')$ such that $L(G, \Xi) = L(G')$.*

Proof. Let $\Xi = L(H), H = ({}_HN, {}_HT, {}_HS, {}_HP)$, $N = N_1 \cup N_2 \cup \ldots \cup N_n$, and $S = S_1 S_2 \ldots S_n$. We will construct set $\widehat{N}$ in this way:

- if ${}_HS \ {}_i\!\Rightarrow^* A$ in H, $A \in {}_HN$, add $\langle \varepsilon, \varepsilon, \ldots, {}_HSA\#, \ldots, \varepsilon \rangle$ to $\widehat{N}$, where ${}_HSA\#$ is at the ith position.
- if $C \ {}_j\!\Rightarrow^* A$ and $A \ {}_i\!\Rightarrow^* B$ in H, $i < j, \ A, B, C \in {}_HN$ and $\langle y_1, y_2, \ldots, y_i, \ldots, y_j, \ldots, y_n \rangle \in \widehat{N}$, such that $y_j \in \{{}_HN {}_HN\}^* \{C\}\{A\}\{\#\}$, then add $\langle y_1, y_2, \ldots, y_i AB\#, \ldots, y_j, \ldots, y_n \rangle$ to $\widehat{N}$.
- if $C \ {}_j\!\Rightarrow^* A$ and $A \ {}_i\!\Rightarrow^* B$ in H, $i > j, \ A, B, C \in {}_HN$ and $\langle y_1, y_2, \ldots, y_j, \ldots, y_i, \ldots, y_n \rangle \in \widehat{N}$, such that $y_j \in \{{}_HN {}_HN\}^* \{C\}\{A\}\{\#\}$, then add $\langle y_1, y_2, \ldots, y_j, \ldots, y_i AB\#, \ldots, y_n \rangle$ to $\widehat{N}$.
- if $A \ {}_i\!\Rightarrow x$ in H, $A \in {}_HN, x \in {}_HT$ and $\langle y_1, y_2, \ldots, y_i, \ldots, y_n \rangle \in \widehat{N}$, such that $y_i \in \{{}_HN {}_HN\}^* {}_HN\{A\}\{\#\}$, then add $\langle y_1, y_2, \ldots, y_i \bullet, \ldots, y_n \rangle$ to $\widehat{N}$.

The construction of $\widehat{N}$ is completed. $\widehat{M} = \{X \mid X \in \widehat{N}, sub(X) \cap \{\bullet\} \neq \emptyset\}$.

Next, we construct grammar $G' = (N', T', S', P')$ as follows:

1. if $X \in \widehat{M}$, then add $[X, S_1]$ to N' and $S' \rightarrow [X, S_1]$ to P'.
2. if
   $(*)$ $X = \langle y_1, y_2, \ldots, y_i, \ldots, y_n \rangle \in \widehat{M}, y_h = \varepsilon, 0 \leq h \leq i-1, y_i = AB\#\overline{y}_i, \ A, B \in {}_HN$ and $A \ {}_i\!\Rightarrow^* C \ {}_i\!\Rightarrow D \ {}_i\!\Rightarrow^+ B$ and $C \Rightarrow qD$ in H and $Y \Rightarrow aZ \ [q]$ in G,
   then add $[X, Z]$ to N' and rule $[X, Y] \rightarrow a[X, Z]$ to P'.

3. if $(*)$ is untrue and if $X = \langle y_1, y_2, \ldots, y_i, \ldots, y_n\rangle \in \widehat{M}, y_h = \varepsilon, 0 \leq h \leq i-1, y_i = AB\#\overline{y}_i,\ A, B \in {}_HN$ and $A\ {}_i\Rightarrow^*\ C\ {}_i\Rightarrow\ B$ and $C \Rightarrow qB$ in H and $Y \Rightarrow aZ\ [q]$ in G, then add
$[\langle y_1, y_2, \ldots, \overline{y}_i, \ldots, y_n\rangle, Z]$ to N' and rule $[X, Y] \to a[\langle y_1, y_2, \ldots, \overline{y}_i, \ldots, y_n\rangle, Z]$ to P' and replace $\langle y_1, y_2, \ldots, y_i, \ldots, y_n\rangle$ with $\langle y_1, y_2, \ldots, \overline{y}_i, \ldots, y_n\rangle$ in $\widehat{M}$.

4. if $(*)$ is untrue and if $X = \langle y_1, y_2, \ldots, y_i, \ldots, y_n\rangle \in \widehat{M}, y_h = \varepsilon, 0 \leq h \leq i-1, y_i = AB\#,\ A, B \in {}_HN$ and $A\ {}_i\Rightarrow^*\ C\ {}_i\Rightarrow\ B$ and $C \Rightarrow qB$ in H and $Y \Rightarrow a\ [q]$ in G, then add
$[\langle y_1, y_2, \ldots, \varepsilon, \ldots, y_n\rangle, S_{i+1}]$ to N' and rule $[X, Y] \to a[\langle y_1, y_2, \ldots, \varepsilon, \ldots, y_n\rangle, S_{i+1}]$ to P' and replace $\langle y_1, y_2, \ldots, y_i, \ldots, y_n\rangle$ with $\langle y_1, y_2, \ldots, \varepsilon, \ldots, y_n\rangle$ in $\widehat{M}$.
Suppose that $S_{n+1} = \varepsilon$.

5. Add $[\langle\varepsilon, \ldots, \varepsilon\rangle, \varepsilon] \to \varepsilon$ to P'.

6. If $X = [\langle\varepsilon, \ldots, \varepsilon, \bullet, y_i, \ldots, y_n\rangle, Y] \in \widehat{M}$, then replace X with $[\langle\varepsilon, \ldots, \varepsilon, \varepsilon, y_i, \ldots, y_n\rangle, Y]$ in $\widehat{M}$.

7. $T' = T$.

Next we prove that $L(G, \Xi, k) = L(G')$.
$L(G, \Xi, k) \subseteq L(G')$: for every $w \in L(G, \Xi, k)$, there exists a derivation of the form

$$ {}_HS\ {}_{i_1}\Rightarrow^*\ q_1A_1\ {}_{i_2}\Rightarrow^*\ q_1q_2A_2\ {}_{i_3}\Rightarrow^*\ \ldots\ {}_{i_{p-1}}\Rightarrow^*\ q_1\ldots q_{p-1}A_{p-1}\ {}_{i_p}\Rightarrow^*\ q_1\ldots q_p = q \text{ in H} \quad (4.1)$$

and

$$\begin{aligned} S_1\ldots S_{i_1}\ldots S_{i_2}\ldots S_n &\Rightarrow^* S_1\ldots w_1X_1\ldots S_{i_2}\ldots S_n\ [q_1] \Rightarrow^* \\ &\Rightarrow^* S_1\ldots w_1X_1\ldots w_2X_2\ldots S_n\ [q_2] \Rightarrow^* \ldots \Rightarrow^* \\ &\Rightarrow^* w_{i_1}\ldots w_{i_{p-1}}\ldots \overline{w}_{i_p}X_{p-1}\ldots w_{i_n}\ [q_{p-1}] \Rightarrow^* \\ &\Rightarrow^* w_{i_1}\ldots w_{i_n}\ [q_p] = w \end{aligned}$$

in G, where $q_h \in R_h, 1 \leq h \leq n$.
Derivation (4.1) can be rewritten in this form

$$X = \langle y_1, y_2, \ldots, {}_HSA_1\#\overline{y}_{i_1}, \ldots, A_1A_2\#\overline{y}_{i_2}, \ldots, \overline{y}_p\#\bullet, \ldots, y_n\rangle$$

which belongs to $\widehat{M}$. We start derivation in G' from start symbol $[X, S_1]$.

$$\begin{aligned} [X, S_1] &= [\langle y_1, y_2, \ldots, {}_HSA_1\#\overline{y}_{i_1}, \ldots, A_1A_2\#\overline{y}_{i_2}, \ldots, \overline{y}_p\#\bullet, \ldots, y_n\rangle, S_1] \Rightarrow^* \\ &\Rightarrow^* w_{i_1}[\langle\varepsilon, y_2, \ldots, {}_HSA_1\#\overline{y}_{i_1}, \ldots, A_1A_2\#\overline{y}_{i_2}, \ldots, \overline{y}_p\#\bullet, \ldots, y_n\rangle, S_2] \Rightarrow^* \ldots \Rightarrow^* \\ &\Rightarrow^* w_{i_1}\ldots w_{i_{n-1}}[\langle\varepsilon, \ldots, \varepsilon, y_n\rangle, S_n] \Rightarrow^* w_{i_1}\ldots w_{i_n}[\langle\varepsilon, \ldots, \varepsilon\rangle, \varepsilon] \Rightarrow w_{i_1}\ldots w_{i_n} = w. \end{aligned}$$

Hence, a $w \in L(G, \Xi, k)$ implies $w \in L(G')$.
$L(G, \Xi, k) \supseteq L(G')$: for every $w \in L(G')$, there exists a successful derivation from start symbol $[X, S_1]$

$$\begin{aligned}[X, S_1] = [\langle y_1, y_2, \dots, {}_HSA_1\#\overline{y}_{i_1}, \dots, A_1A_2\#\overline{y}_{i_2}, \dots, \overline{y}_p\#\bullet, \dots, y_n\rangle, S_1] \Rightarrow^* \\ \Rightarrow^* w_{i_1}[\langle \varepsilon, y_2, \dots, {}_HSA_1\#\overline{y}_{i_1}, \dots, A_1A_2\#\overline{y}_{i_2}, \dots, \overline{y}_p\#\bullet, \dots, y_n\rangle, S_2] \Rightarrow^* \dots \Rightarrow^* \\ \Rightarrow^* w_{i_1} \dots w_{i_{n-1}}[\langle \varepsilon, \dots, \varepsilon, y_n\rangle, S_n] \Rightarrow^* w_{i_1} \dots w_{i_n}[\langle \varepsilon, \dots, \varepsilon\rangle, \varepsilon] \Rightarrow w_{i_1} \dots w_{i_n} = w.\end{aligned}$$

$X \in \widehat{M}$ is of the form

$$X = \langle y_1, y_2, \dots, {}_HSA_1\#\overline{y}_{i_1}, \dots, A_1A_2\#\overline{y}_{i_2}, \dots, \overline{y}_p\#\bullet, \dots, y_n\rangle$$

X defines the derivation

$${}_HS \ {}_{i_1}\!\Rightarrow^* q_1A_1 \ {}_{i_2}\!\Rightarrow^* q_1q_2A_2 \ {}_{i_3}\!\Rightarrow^* \dots \ {}_{i_{p-1}}\!\Rightarrow^* q_1 \dots q_{p-1}A_{p-1} \ {}_{i_p}\!\Rightarrow^* q_1 \dots q_p = q \text{ in H}$$

which regulates grammar G in this way

$$\begin{aligned}S_1 \dots S_{i_1} \dots S_{i_2} \dots S_n \Rightarrow^* S_1 \dots w_1X_1 \dots S_{i_2} \dots S_n \ [q_1] \Rightarrow^* \\ \Rightarrow^* S_1 \dots w_1X_1 \dots w_2X_2 \dots S_n \ [q_2] \Rightarrow^* \dots \Rightarrow^* \\ \Rightarrow^* w_{i_1} \dots w_{i_{p-1}} \dots \overline{w}_{i_p}X_{p-1} \dots w_{i_n} \ [q_{p-1}] \Rightarrow^* w_{i_1} \dots w_{i_n} \ [q_p] = w\end{aligned}$$

in G, where $q_h \in R_h, 1 \leq h \leq n$.
Thus, $w \in L(G, \Xi, k)$ so $L(G, \Xi, k) = L(G')$.

Because $card(N') \leq card({}_HN)^{2nk+1}$ and all rules are regular, $G' \in REG$. □

Example 4.1. $G = (N, T, P, S)$:

- $N = \{A, B, C, D\}$,
- $T = \{x, y, u, v\}$,
- $S = AC$,
- $P = \{a.A \to xB, b.B \to yA, \overline{a}.A \to \varepsilon, c.C \to uD, d.D \to vC, \overline{c}.C \to \varepsilon\}$.

$\Xi = L(H), H = ({}_HN, {}_HT, {}_HS, {}_HP)$

- ${}_HN = \{X, \overline{X}, Y, U, V\}$,
- ${}_HT = \{a, \overline{a}, b, c, \overline{c}, d\}$,
- ${}_HS = X$,
- ${}_HP = \{X \to aY, Y \to bU, U \to cV, V \to dX, X \to \overline{a}\overline{X}, \overline{X} \to \overline{c}\}$.

$L(G, \Xi) = \{(xy)^n(uv)^n \mid n \geq 0\} \subseteq \mathfrak{R}(REG, 2)$
$\widehat{N} = \{\langle XY\#, \varepsilon\rangle, \langle XU\#, \varepsilon\rangle, \langle X\overline{X}\#, \varepsilon\rangle\ \}$ *due to the first rule, because* $i({}_HS) = \{Y, U\}$ *where* $i = 1$ *and* ${}_HS = X$.
$\widehat{N} = \widehat{N} \cup \{\langle XU\#, UV\#\rangle, \langle XU\#, UX\#\rangle, \langle X\overline{X}\#, \bullet\rangle\ \}$ *due to the second rule.*
$\widehat{N} = \widehat{N} \cup \{\langle XU\#XY\#, UX\#\rangle, \langle XU\#XU\#, UX\#\rangle\}$ *due to the second rule.*
$\widehat{N} = \widehat{N} \cup \{\langle XU\#XY\#, UX\#UV\#\rangle, \langle XU\#XU\#, UX\#UX\#\rangle, \langle XU\#X\overline{X}\#, UX\#\bullet\rangle\}$ *due to the second rule.*
$\widehat{N} = \widehat{N} \cup \{\langle XU\#XY\#X\overline{X}\#, UX\#UX\#\rangle, \langle XU\#XU\#X\overline{X}\#, UX\#UX\#\bullet\rangle\}$ *due to the last rule.*
$\widehat{M} = \{\langle XU\#XU\#X\overline{X}\#, UX\#UX\#\bullet\rangle, \langle XU\#X\overline{X}\#, UX\#\bullet\rangle, \langle X\overline{X}\#, \bullet\rangle\}$.

As opposed to Theorem 4.6, the next theorem demonstrates that if during the derivation of any sentence from the generated language, these grammars change the position of rewriting finitely many times, then they always generate only the family of regular languages independently of the length of their start strings.

Theorem 4.10.

$$\mathfrak{R}(REG, n, k) = REG.$$

Proof. $REG = \mathfrak{R}(REG, 1, 0) \subseteq \mathfrak{R}(REG, n, k) \subseteq REG$ (see Theorem 4.9). □

5 Applications in Parsing

In previous chapters we saw that concurrency and regulation can extend the power of a formal system based on regular or linear language towards context-sensitive language.

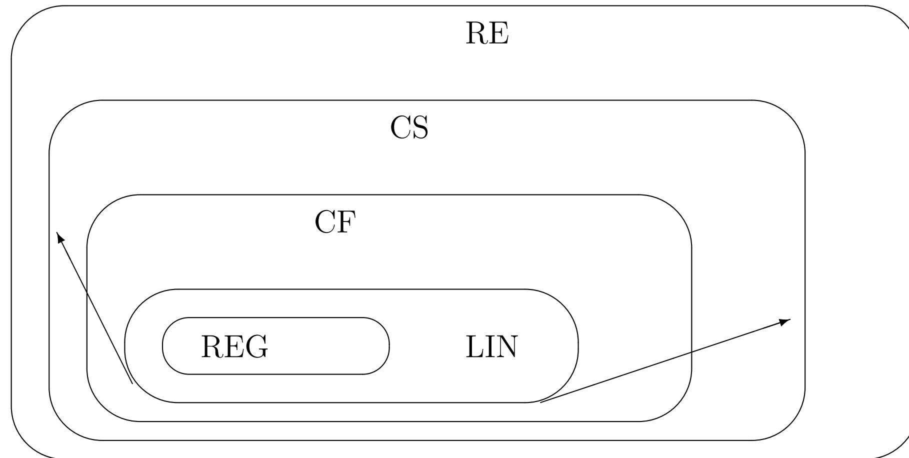


Figure 3: Regulation of REG and LIN languages

These kinds of results are very interesting from the theoretical point of view. But for practical results, it is much more interesting to regulate context-free languages rather than regular or linear ones. There are two reasons for this. Firstly, achieving some context-sensitive programming language by regulation by linear or regular language leads to a very complex and chatty grammars (see [Rych–05]). Secondly, we already have the whole theory for managing context-free languages. So it is natural to use this theory and regulate context-free languages and move towards context-sensitive and recursive enumerable languages.

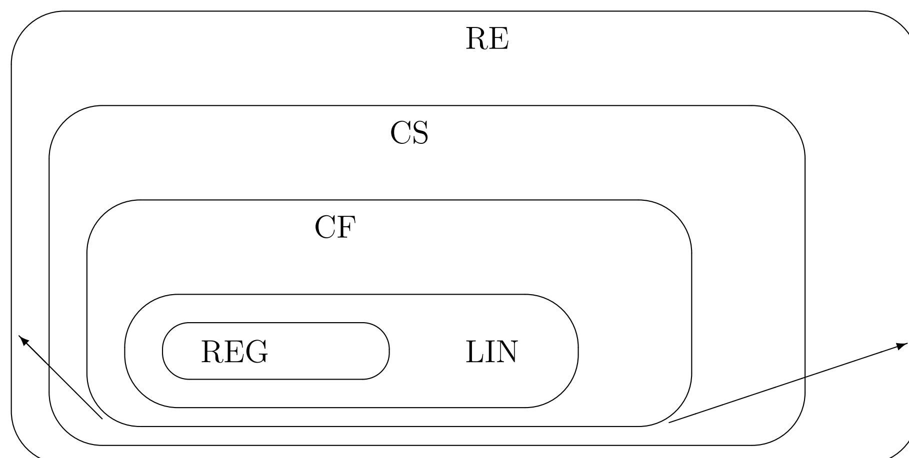


Figure 4: Regulation of CF languages

Now the natural question arises: why would we need to have parsers for more powerful languages than context-free ones? Is not C, C# or Java enough? In one way they are. But the more powerful language family we choose, the more complex requirements we can demand.

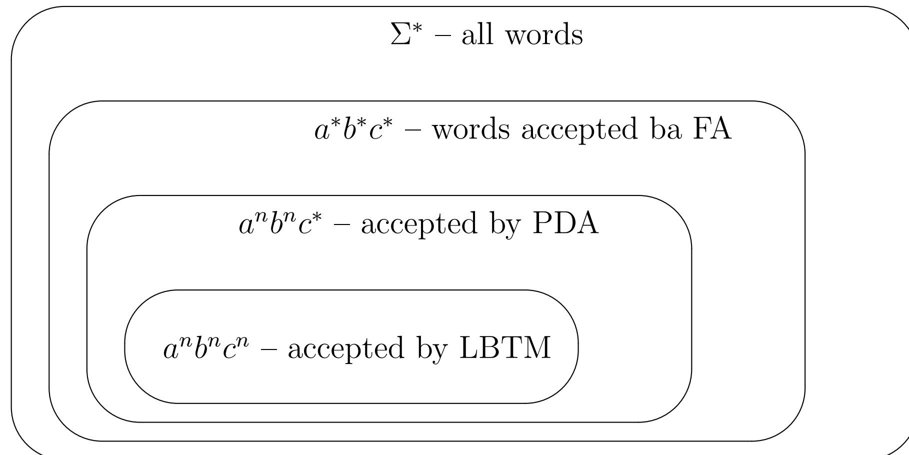


Figure 5: Another look at language hierarchy

In Figure 5 we can see that if we have only finite automata for parsing programming languages we cannot distinguish between program in the form $a^*b^*c^*$ and $a^nb^nc^*$. If we have pushdown automata for parsing we cannot distinguish $a^nb^nc^*$ and $a^nb^nc^n$.

Consider the following two programs:

```
program A;
int :  a, b;
string :  s;
begin;
 a := 1; b := 2;
 a := a + b;
 s := "foo";
end.
```

```
program B;
int :  a, b;
string :  s;
begin;
 a := 1; b := 2;
 a := a + b;
 s := 1;
end.
```

Both programs are described by standard, context-free Pascal-like grammar. It is easy to see that program A is correct and program B is incorrect, because of assigning integer value `1` to string variable `s`. This is the place where the power of classic context-free parsers fails. They are not able to distinguish correct from incorrect programs like program A from program B.

In Figure 6 we can see the reversed Chomsky hierarchy. Σ^* represents all programs (text files) over ASCII character set. No parser is needed for distinguishing whether any text file is or is not a program. If we use finite automaton for parser we can define tokens and key words of our new programming language. Parser based on such finite automaton

can distinguish whether any program consists of allowed tokens and key words without any syntactic analysis. Next, pushdown automaton comes into play. Now we can easily describe our programs by context-free grammars, construct LL table and parse input files and decide whether they are programs in our language or not. This is the usual case of parsing. All present programming languages such as C, C# or Java belong in this category. We can prescribe syntax for such programming language but we are not able to prescribe that program A is the correct program of such language but program B is not correct.

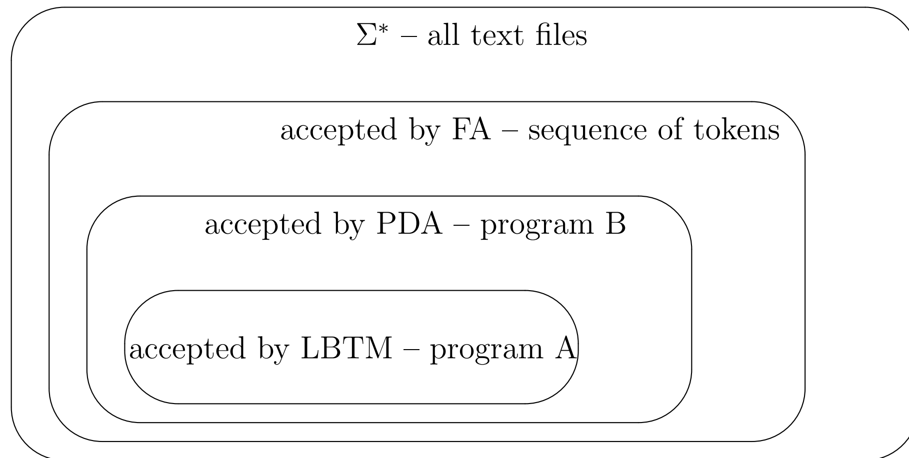


Figure 6: Correct and incorrect programs

Finally, we would like to move further towards parsing by more powerful languages. We would like to define non context-free grammars and corresponding parsing techniques, which can decide whether a certain program file is correct, whether it is a word from some context-sensitive language, or is not correct.

Some kind of described possibilities we can already see in modern development environments. It is achieved by repetitive running of the whole compile process, including data flow analysis. This approach is quite complex and time consuming task and also usually needs to be hard-coded in development environment and compiler. The described approach reveals all mistakes made during the parsing phase and it is possible to make a general parser that accepts any grammar and verifies the program.

This chapter first, in short, describes classic parsing techniques of context-free languages. Next we discuss possibilities of moving beyond the classic methods and present parsing techniques for scattered-context grammar. The implementation is then described generally.

5.1 Classic Parsing Technique

Parsing a string according to a grammar means deciding whether the input word (source program) is generated by the grammar and if so, to construct the parse tree that shows how the given word can be derived from the grammar. This task of parser is the key

part in the compiler operation. As shown in Figure 7 (taken from [Aho–07]), the parser together with the symbol table is responsible for constructing the parse tree. To do so, it has to communicate with a lexical analyzer to obtain the sequence of tokens from the input string.

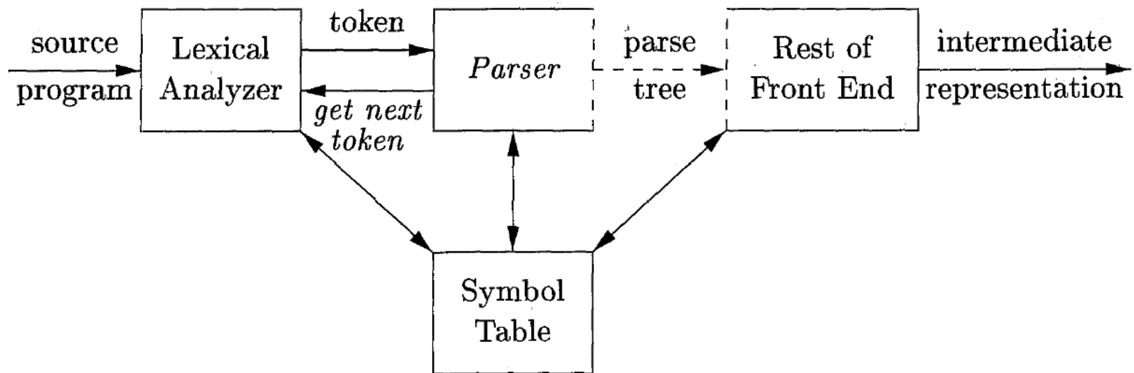


Figure 7: Parser in compiler model

A word can have more than one parse tree. In this case we call it *ambiguous*. The ambiguity can be true or false. The false ambiguity in Figure 8 does not change the semantics. The string `3+5+1` has two parse trees but the semantics is `9` in both cases.

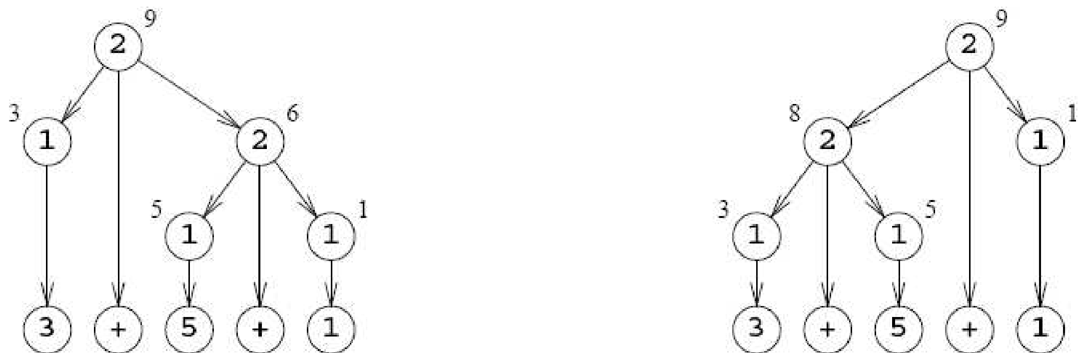


Figure 8: False ambiguity

If we change the `+` into a `-` in the previous example, the ambiguity changes the semantics. In the first case, in Figure 9, the semantics is `-1` but in the other case, it is `-3`.

Figure 9: True ambiguity

To reconstruct the parse tree we need the parse technique. If we look into literature [Aho–72], [Aho–07] or [Med–05] we find many of such techniques. The two main techniques are top-down parsing and bottom-up parsing. Top-down parsing starts from start symbol of a grammar and by simulating application of rules until the final word is reached. Bottom-up parsing works in reversed mode. It takes the input word and replaces the right side of some rule by its left side. This is repeated until the start symbol is reached. In both cases the main question is to decide which rule to apply if there are more of them which can be applied. The answer for grammar $G = (N, T, P, S)$ is to construct an LL table $\alpha(A, a) \in P$, where $A \in N$ and $a \in T$.

Definition 5.1. Let $G = (N, T, P, S)$ is a context-free grammar, $\alpha \in (N \cup T)^*$.

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

Definition 5.2. Let $G = (N, T, P, S)$ is a context-free grammar, $A \in N$.

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

Definition 5.3. Condition FF holds if for every set of production rules:

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

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

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

Definition 5.4. Condition FFL holds if for every set of production rules:

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

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:

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

Definition 5.5. A context-free grammar G is LL_1 *grammar* if conditions FF and FFL are satisfied for the G.

Definition 5.6. Let $G = (N, T, P, S)$ is a context-free grammar, $\alpha \in (N \cup T)^*$.

$$Empty(\alpha) ::= \{\varepsilon \mid \text{if } \alpha \Rightarrow^* \varepsilon\}$$

$$Empty(\alpha) ::= \{\emptyset \mid \text{otherwise}\}$$

Definition 5.7. Let $G = (N, T, P, S)$ be a context-free grammar. For every $A \to x \in P$, we define set $PREDICT(A \to x)$ so that

- if $Empty(x) = \{\varepsilon\}$ then
  $PREDICT(A \to x) = FIRST(x) \cup FOLLOW(A)$.
- if $Empty(x) = \emptyset$ then
  $PREDICT(A \to x) = FIRST(x)$.

Definition 5.8. Let $G = (N, T, P, S)$ be a context-free grammar. $A \to x \in P$ and $a \in T$.

- if $a \in PREDICT(A \to x)$,
  then add $A \to x$ to $\alpha(A, a)$.

Now if the parser reaches the state where nonterminal A is on the top of the stack and terminal a is the first terminal from the rest of input string, it looks into the LL table for the rule at position $\alpha(A, a)$.

5.2 Alternative Approach to Parsing

By introducing context-sensitive syntax analysis into the source code parsing process a whole class of new problems may be solved at this stage of a compiler. Namely issues with correct variable definitions, type checking etc. The main goal of regulated formal systems is to extend abilities from standard CF LL-parsing to CS or RE families with preservation of ease of parsing.

In [Kol–04] and [Rych–05] we can find some basic facts from theory of regulated pushdown automata (RPDA). We figured that regulated pushdown automata can in some cases simulate Turing machines so we could use this theory for constructing parsers for context-sensitive languages or even type-0 languages. We have also demonstrated the basic problem of this concept: complexity. Almost trivial Turing machine was transformed to regulated pushdown automata with almost 6 000 rules.

Converting deterministic (linear bounded) Turing machine or scattered-context grammar to deterministic RPDA is very complex task. For the most simple context-sensitive languages corresponding deterministic RPDA has thousands of rules. If we want to use these algorithms for creating some practical parser for real context-sensitive programming language it may result in millions of rules. Therefore, we are looking for another way to parse context-sensitive languages. We would like to extend some context-free grammar

of any common programming language (such as Pascal, C/C# or Java). After extending context-free grammar to corresponding context-sensitive grammar, parsing should be straightforward.

As an example of a context-free language we use a language called ZAP03 [ZAP–03] which has very similar syntax to Pascal. We will use following program as an example program in ZAP03 language.

```
int :  a, b, c, d;
string :  s;

begin
  a = 1;
  b = 2;
  c = 10;
  d = 15;
  s = "foo";
  a = c;
end
```

Now we will define KontextZAP03, the context-sensitive extension of ZAP03. KontextZAP-03 will be described by scattered-context grammar. In the first phase we will enrich KontextZAP03 by variable checking. If the variable is undefined or assigned before initialized, the parser of KontextZAP03 will finish in error state. We will need to analyze three fragments of ZAP03 code where variables are used (variable `c` for example).

Variable definition

`int :  a, b, c, d;`

Assignment statement

`c = 10;`

And using variables in commands

`a = c;`

Corresponding grammar fragments from ZAP03 are following.

Variable definition

DCL → TYPE [:] [id] ID_LIST

ID_LIST → [,] [id] ID_LIST

ID_LIST → ε

Fragment of assignment statement

COMMAND → [id] CMD COMMAND

CMD → [=] STMT [;]

And usage variable in command

STMT → [id] OPER

OPER → ε.

Symbols in brackets [,] are terminals. Complete ZAP03 grammar has about 70 context-free grammar rules.

We define grammar of language KontextZAP03 in the following way. Substitute previous rules with these scattered-context ones:
`(DCL, S')` $\rightarrow$ `(TYPE [:]  [id] ID_LIST, D)`
`(ID_LIST, S')` $\rightarrow$ `([,] [id] ID_LIST, D)`
`(ID_LIST)` $\rightarrow$ (ε)
assignment statement
`(COMMAND, D)` $\rightarrow$ `([id] CMD COMMAND, DL)`
`(CMD)` $\rightarrow$ `([=] STMT [;])`
and usage variable in command
`(STMT, D)` $\rightarrow$ `([id] OPER, DR)`
`(OPER)` $\rightarrow$ (ε).

Parsing now proceeds in the following way: starting symbol is `S S'` and derivation will go as usual until there is `DCL S'` processed and `(DCL, S')` $\rightarrow$ `(TYPE [:]  [id] ID_LIST, D)` rule is to applied. At this moment `S'` is rewritten to `D` indicating that variable [id] is defined. When variable [id] is used on left resp. right side of assignment `D` is rewritten to `DL` resp. `DR` according to second resp. third previously shown fragment. If variable [id] is used without being defined beforehand, a parse error occurs because `S'` is not rewritten to `D` and `S'` cannot be rewritten to `DL` or `DR` directly. When the input is parsed `S` is rewritten to program code and during LL parsing is popped out of the stack. `S'` is rewritten onto `D`$\{$`LR`$\}^*$ and this is only string that remains.
`S S'` $\Rightarrow^*$ `DCL S'` $\Rightarrow$ `TYPE [:]  [id] ID_LIST D` $\Rightarrow^*$ `COMMAND D` $\Rightarrow$
    $\Rightarrow$ `[id] CMD COMMAND DL` $\Rightarrow^*$ `DL`
If the only remaining symbol is `D`, it means that variable [id] was defined but never used. If `DL`$^+$ is the only remaining symbol, we know that variable [id] was defined and used only on the left sides of assignments. Finally if there is the only remaining `DR(LR)`*, we know that the first occurrence of variable [id] is on the right side of an assignment statement and therefore it is being read without being set. In all these cases the compiler should generate a warning. These and similar problems are usually addressed by a data-flow analysis phase carried out during semantic analysis.

Using this algorithm we can only process one variable at a time. But the proposed mechanism can be easily extended to a finite number of variables by adding new `S'`$\cdots$`'` every time we discover a variable definition. Parsing of described scattered-context grammar can be implemented by pushdown automaton with finite number of pushdowns. The first pushdown is classic LL pushdown. The second one is variable specific and every [id] holds its own.

Because original ZAP03 grammar is LL_1 and using described algorithm was not any rule added, KontextZAP03 grammar has unambiguous derivations.

A few examples can clear the idea. This program is well-formed according to ZAP03 grammar, but it's semantics is not correct and parsing it as KontextZAP03 program should reveal this error.

```
int :  a, b, c, d;
string :  s;

begin
  a = 1;
  b = 2;
  d = 15;
  s = "foo";
  a = c;
end
```

Corresponding stack to variable `c` will be `DR` what lead to warning:
`Variable c read but not set.`
Second example shows another variation

```
int :  a, b, c, d;
string :  s;

begin
  a = 1;
  d = 15;
  s = "foo";
  a = c;
end
```

Corresponding stack to variable `b` will be `D` what lead to warnings (together with previous one):
`Variable c read but not set.`
`Variable b is defined but never used.`

5.3 The Power of Modified Multistack Machine

As demonstrated in previous chapter, we will study scattered-context grammars where every component represents its own stack in multistack machine. Therefor we need special case of scattered-context grammars where every component works with different and mutually disjoint alphabets. Our question is whether these grammars are capable to parse context-sensitive languages.

Definition 5.9. Let $G = (N, T, P, S)$ be a SCG. $G' = (N_1, N_2, \ldots, N_n, T, P', S')$ is a *n-distributed SCG*, $n{-}_{dis}SCG$ for short, if

- $N = N_1 \cup N_2 \cup \ldots \cup N_n$, where $N_i, 1 \leq i \leq n$ are pairwise disjoint nonterminal alphabets,

- P' is a finite set of rules. P' contains two kinds of rules
  1. $S' \rightarrow X_1 \dots X_n, \quad X_i \in N_i, 1 \leq i \leq n,$
  2. $(A_1, A_2, \dots, A_m) \rightarrow (w_1, w_2, \dots w_m), \quad m \leq n, A_i \in N_i, w_i \in (N_i \cup T)^*$, for some $1 \leq i \leq m$.

Example 5.1. *$G = (\{A\}, \{B\}, \{C\}, \{a, b, c\}, P, S)$ where P contains three rules*

1. *$S \rightarrow ABC$;*
2. *$(A, B, C) \rightarrow (aA, bB, cC)$;*
3. *$(A, B, C) \rightarrow (a, b, c)$*

is 3-distributed SCG.

Theorem 5.1. *Let $G = (N_1, \dots, N_n, T, P, S)$ be a $n{-}_{dis}SCG$. Then there exists a matrix grammar $M = (N', T', P', S')$ such that $L(G) \subseteq L(M)$.*

Proof. We will construct grammar M in the form

- $N' = N_1 \cup N_2 \cup \dots \cup N_n$;
- $T' = T$;
- $S' = S$;
- $P' = \{[A_1 \rightarrow w_1, A_2 \rightarrow w_2, \dots, A_m \rightarrow w_m] \mid$
  $(A_1, A_2, \dots, A_m) \rightarrow (w_1, w_2, \dots w_m) \in P, \quad m \leq n\} \cup$
  $\cup \{S \rightarrow X_1 X_2 \dots X_n \mid S \rightarrow X_1 X_2 \dots X_n \in P\}$.

For any applied rule $(A_1, A_2, \dots, A_m) \rightarrow (w_1, w_2, \dots w_m)$ in grammar G, there is the equivalent matrix rule $[A_1 \rightarrow w_1, A_2 \rightarrow w_2, \dots, A_m \rightarrow w_m]$ in grammar G applied. The order of nonterminals A_i is trivially preserved. □

Example 5.2. *Let G be an $n{-}_{dis}SCG$ grammar from example 5.1. Corresponding matrix grammar is $M = (\{A, B, C, S'\}, \{a, b, c\}, P', S')$, where P' contains these rules*

1. *$p_1 = [S' \rightarrow ABC]$;*
2. *$p_2 = [A \rightarrow aA, B \rightarrow bB, C \rightarrow cC]$;*
3. *$p_3 = [A \rightarrow a, B \rightarrow b, C \rightarrow c]$.*

Theorem 5.2. *Let $M = (N, T, P, S)$ be a matrix grammar. Then there exists a $2{-}_{dis}SCG$ $G = (N_1, N_2, T', P', S')$ such that $L(M) \subseteq L(G)$.*

Proof. We will construct grammar G in the form

- $N_1 = N$;
- $N_2 = \{p\} \cup \{p_i^j\}$, where p_i^j are labels of rules of matrix grammar M;
- $T' = T$;
- $P' = \{(P_i^1, p) \to (Q_i^1, p_i^1) \mid p_i^1 = P_i^1 \to Q_i^1, 1 \le i \le |P|\} \cup$
  $\cup\ \{(P_i^j, p_i^{j-1}) \to (Q_i^j, p_i^j) \mid p_i^j = P_i^j \to Q_i^j, 2 \le j \le m_i - 1, 1 \le i \le |P|\} \cup$
  $\cup\ \{(P_i^{m_i}, p_i^{m_i-1}) \to (Q_i^{m_i}, p) \mid p_i^{m_i} = P_i^{m_i} \to Q_i^{m_i}, 1 \le i \le |P|\} \cup$
  $\cup\ \{S' \to Sp\}\ \cup\ \{p \to \varepsilon\}$.

For any applied rule $p_i = [P_i^1 \to Q_i^1, P_i^2 \to Q_i^2, \ldots, P_i^{m_i} \to Q_i^{m_i}]$ in grammar M, there is equivalent chain of applications $(P_i^1, p) \to (Q_i^1, p_i^1), (P_i^2, p_i^1) \to (Q_i^2, p_i^2), \ldots, (P_i^{m_i}, p_i^{m_i-1}) \to (Q_i^{m_i}, p)$ in grammar G. □

Example 5.3. *Let M be a matrix grammar from example 5.2. Corresponding $2{-}_{dis}SCG$ is $G = (\{A, B, C, S, S'\}, \{p_2^1, p_2^2, p_3^1, p_3^2, p\}, \{a, b, c\}, P', S')$, where P' contains these rules*

1. $(S' \to Sp)$;
2. $(S \to ABC)$;
3. $(A, p) \to (aA, p_2^1)$;
4. $(B, p_2^1) \to (bB, p_2^2)$;
5. $(C, p_2^2) \to (cC, p)$;
6. $(A, p) \to (a, p_3^1)$;
7. $(B, p_3^1) \to (b, p_3^2)$;
8. $(C, p_3^2) \to (c, p)$;
9. $(p \to \varepsilon)$;

Theorem 5.3.

$$\mathcal{L}(n{-}_{dis}SCG) = \mathcal{L}(2{-}_{dis}SCG) = \mathcal{L}(Mat).$$

Proof. This theorem follows from theorems 5.1 and 5.2. □

Remark 2.

$$CF \subseteq \mathcal{L}(Mat) = \mathcal{L}(n{-}_{dis}SCG) = \mathcal{L}(2{-}_{dis}SCG)) \subset CS.$$

This theorem shows that $n{-}_{dis}SC$ grammars, if successfully parsed, are as powerful as matrix grammars and therefore can help us to move towards parsing of context-sensitive languages (even if not all of them).

5.4 Type Checking

By using scattered-context grammars we can describe type set of language (INT, STR) and type check rules directly in grammar. Almost trivial language with type checking using 5 stacks can look like this:

$(s) \rightarrow (program)$
$(program) \rightarrow (dcl)$
$(program) \rightarrow ([begin] command [end])$
$(dcl) \rightarrow (type [:] dcl2)$
$(dcl2, S, INT, ,) \rightarrow$
$([id] id_list [;] next, D, INT, INT)$
$(dcl2, S, STR, ,) \rightarrow$
$([id] id_list [;] next, D, STR, STR)$
$(id_list) \rightarrow ([,] id_l ist2)$
$(id_list2, S) \rightarrow ([id] id_list, D)$
$(id_list) \rightarrow (\varepsilon)$
$(next) \rightarrow (dcl)$
$(next) \rightarrow (program)$
$(type, ,) \rightarrow ([int], , INT)$
$(type, ,) \rightarrow ([string], , STR)$
$(command, D, INT, ,) \rightarrow$
$([id]\ cmd\ command, D\ L, INT, INT)$
$(command, D, STR, ,) \rightarrow$
$([id]\ cmd\ command, D\ L, STR, STR)$
$(command) \rightarrow (\varepsilon)$
$(cmd) \rightarrow ([=] stmt [;])$
$(stmt, D) \rightarrow ([id], D\ R)$
$(stmt, , , , INT) \rightarrow ([digit], , , ,)$
$(stmt, , , , STR) \rightarrow ([strval], , , ,)$

Stacks at even positions are the variable specific stacks as mentioned in previous chapter. The first stack is classic LL stack and the rest of stacks at odd positions are temporary used stacks for additional information. Although the underlying context-free grammar is not LL grammar because there are several identic rules (command → [id] cmd command), this grammar is unambiguous.

Example of error program code can be

```
int :  a, b, c, d;
string :  s;

begin
  a = 1;
  b = 15;
  c = "foo";
  a = c;
end
```

because `c = "foo"` is not type correct (STR is assigned to INT), parsing will fail.

$(S, S, \varepsilon, \varepsilon, \varepsilon) \Rightarrow^* (DCL, S, \varepsilon, \varepsilon, \varepsilon) \Rightarrow (TYPE\ [:]\ DCL2, S, \varepsilon, \varepsilon, \varepsilon) \Rightarrow$
$\Rightarrow ([int]\ [:]\ DCL2, S, INT, \varepsilon, \varepsilon) \Rightarrow^2 (DCL2, S, INT, \varepsilon, \varepsilon) \Rightarrow$
$\Rightarrow ([id]\ ID_LIST\ [;]\ NEXT, D, INT, INT, \varepsilon) \Rightarrow^*$
$\Rightarrow^* (COMMAND\ [end], D, INT, INT, \varepsilon) \Rightarrow$
$\Rightarrow ([id]\ CMD\ COMMAND\ [end], DL, INT, INT, INT) \Rightarrow^*$
$\Rightarrow^* (STMT\ [;]\ COMMAND\ [end], DL, INT, INT, INT) \Rightarrow$
$\Rightarrow ([digit]\ [;]\ COMMAND\ [end], DL, INT, INT, INT)$

Now parsing will fail because the input character is [*strval*] instead of [*digit*].

This is an easy extension of standard context-free grammar that can describe the type system. In our previous example there are just two types (INT and STR) and only two allowed assignments (INT $\rightarrow$ INT and STR $\rightarrow$ STR). It is very easy to generalize this approach to construct general type injector into a context-free grammar.

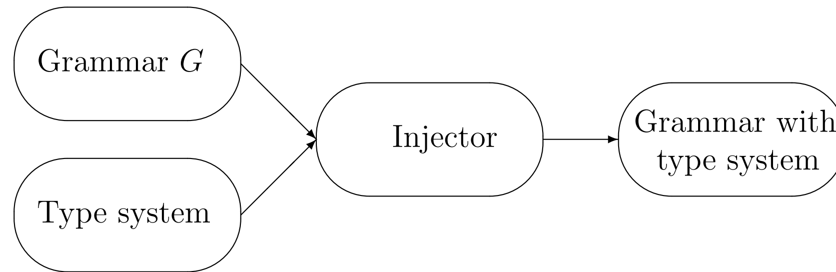


Figure 10: Injecting type system into grammar

5.5 Other Applications of CS Languages

It is obvious, that using a simple scattered-context extension of CF languages, we obtain a grammar with interesting properties with respect to analysis of a programming language source code. We provide some basic motivation examples.

1. Errors related to usage of undefined variables may be discovered and handled at parse time without the need to handle them by static semantic analysis.
2. A CS extension of a grammar of the Java programming language, which copes with problems like mutual exclusion of various keywords, such as `abstract` and `final`, reflecting the fact that abstract methods cannot be declared final and vice versa. This situation can be handled quite easily, by introducing additional symbol S'' and two corresponding rules $S'' \rightarrow A$ (corresponds to `abstract`) and $S'' \rightarrow F$ (corresponds to `final`). Obviously only one of the rules can be used at a time.
3. Introducing an `observer` keyword for methods in Java, which indicates that this method does not modify the state of *this* object (similar to defining method as `const` in C++). Handling of such keyword in the language grammar is similar to approach taken in the previous example.
4. Accounting of statements in a program in a ZAP03 language by introducing new `size` keyword, which defines upper bound on the number of statements in current scope. The parser is then extended such that when the keyword is discovered, the

number of specialized nonterminals (say `X`) is generated on the stack – as specified by the keyword occurrence and the grammar of the language is modified accordingly. The rule:
`COMMAND` $\rightarrow$ `[id] CMD COMMAND`
changes to:
`(COMMAND, X)` $\rightarrow$ `([id] CMD COMMAND,` ε `)`
Then, when a statement rule is used, one `X` nonterminal is eliminated from the stack. If there are no remaining `X` nonterminals, the parsing immediately fails. The context-sensitive language used in this example is:

$$L(G) = \{w.|w|_{10}\},$$

where w is word and $|w|_{10}$ is the length of w written as a decimal number.

5.6 Implementation

The best proof of the concept of this theory is implementation of a context-sensitive parser. This implementation does much more. It is the general parser of scattered-context grammar which takes source program w and scattered-context grammar G and decides whether this program is a word from the language generated by given grammar (see Figure 11).

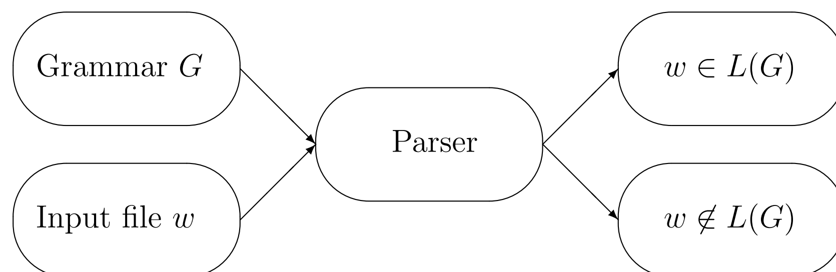


Figure 11: Implemented parser

The complete program documentation is described in Appendix A. The implementation is divided into two parts – the parser and the user interface. The parser is the main part of the program and it is implemented in Java. Graphical user interface is just a thin layer over the parser and it is implemented in C#. The main window of this application is shown in Figure 12.

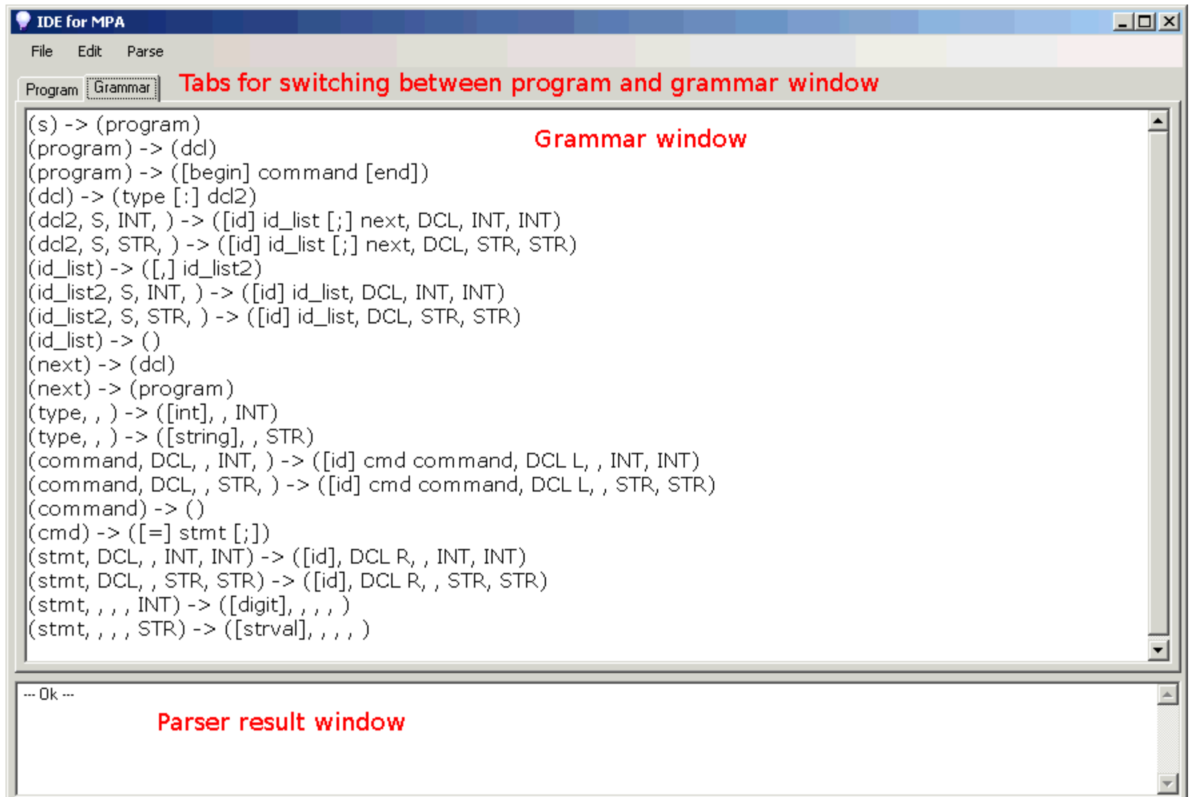


Figure 12: Main window of parser application

The first example of our application can be type-checking grammar, as previously described. Grammar can be loaded by selecting `File` $\rightarrow$ `Open` $\rightarrow$ `Program` or `Grammar` from the drop down menu. The grammar has to be in format

$$(N_1, N_2, \ldots, N_n) \texttt{ -> } (w_1, w_2, \ldots, w_n)$$

where ε is represented as white space. For example

```
(command, DCL, , INT, ) -> ([id] cmd command, DCL L, , INT, INT)
```

Completely loaded context sensitive grammar for type-checking is in Figure 12.

After loading both parts, a grammar and a program, we can see that every time the code of the program is modified, the parser runs the verification and displays result in the bottom part of the window. In Figure 13 we can see the previously mentioned source program and its successful parsing result.

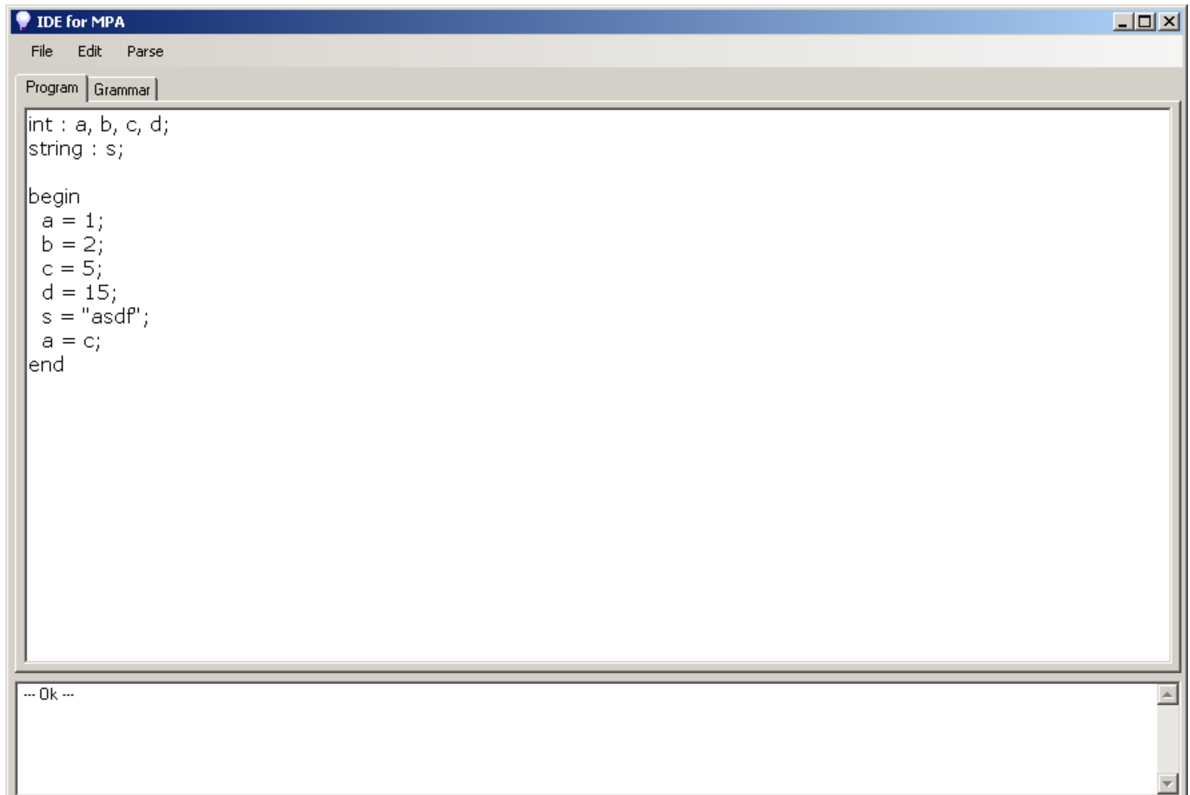


Figure 13: Successfully parsed program

Now we can make this program incorrect by modifying the eighth line and assigning string `"15"` to integer variable `d` (see Figure 14). Immediately after this modification, the parser rescans the source program and, by the previously mentioned algorithm, detects the problem. As shown in our example, the problem is in lexem of length 4 at position 70 in our source program. This place is highlighted by the parser and the error message is:

-70:4- No rule to apply
Lexem:"15" Top: "stmt"

This report means that during the parsing process the parser reaches lexem `"15"` (at position 70) and on the top of the stack it has `"stmt"` and there was no rule to apply according to LL table.

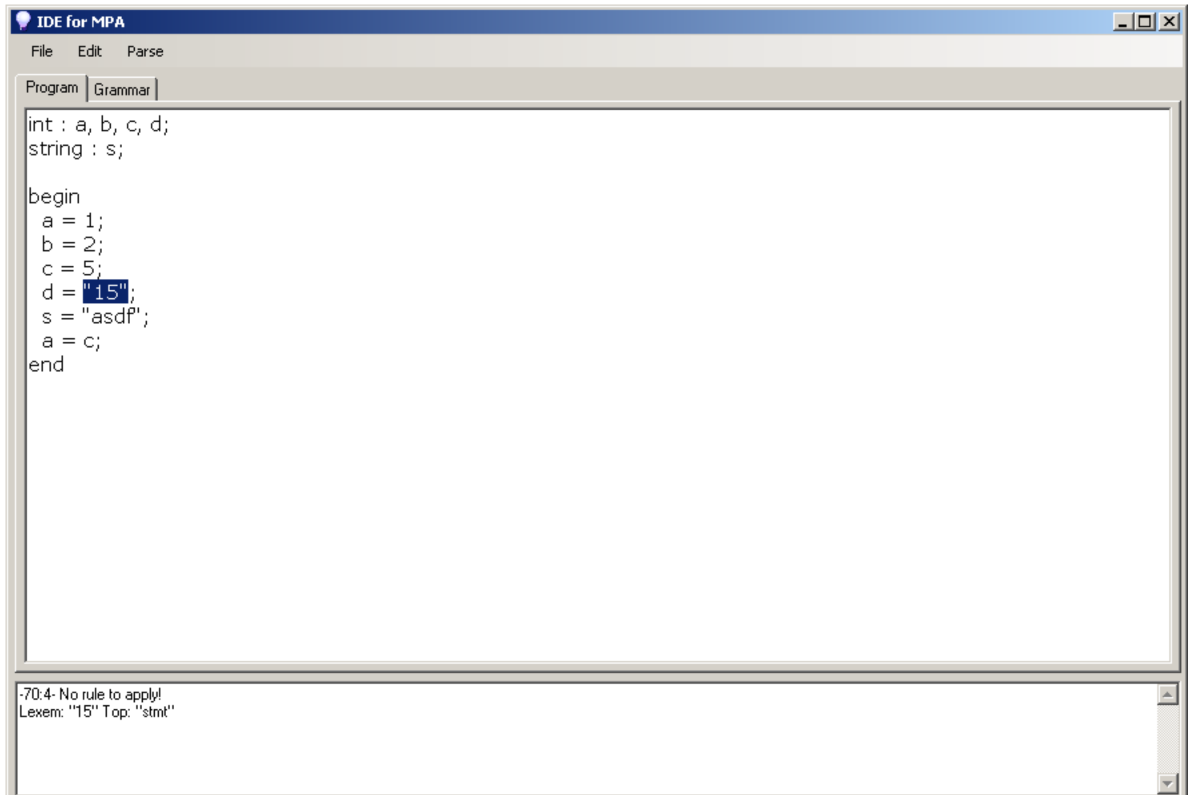


Figure 14: String assigned to integer variable

Our next example shows another mentioned possibility of our parser. The parser verifies whether all variables are first assigned and then used. This ability is already included in the previous example, hence the grammar is the same. In Figure 15 variable `b` is not used, therefore the corresponding variable stack will be `D` and the error message is:

Variable b defined but never used.

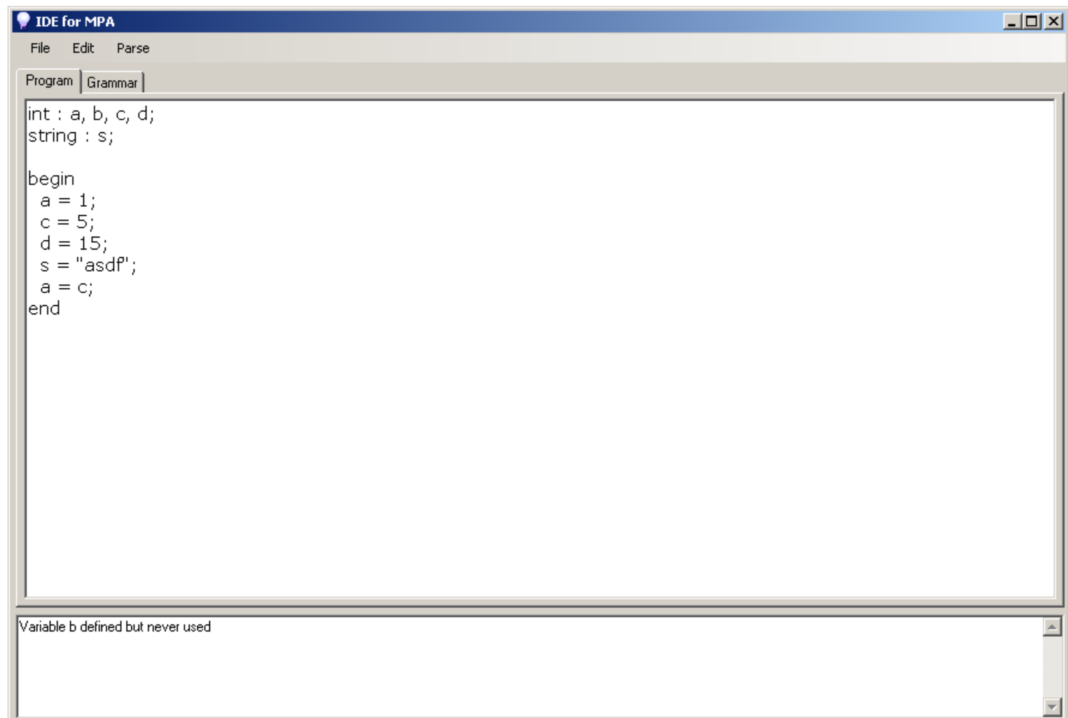


Figure 15: Unused variable b

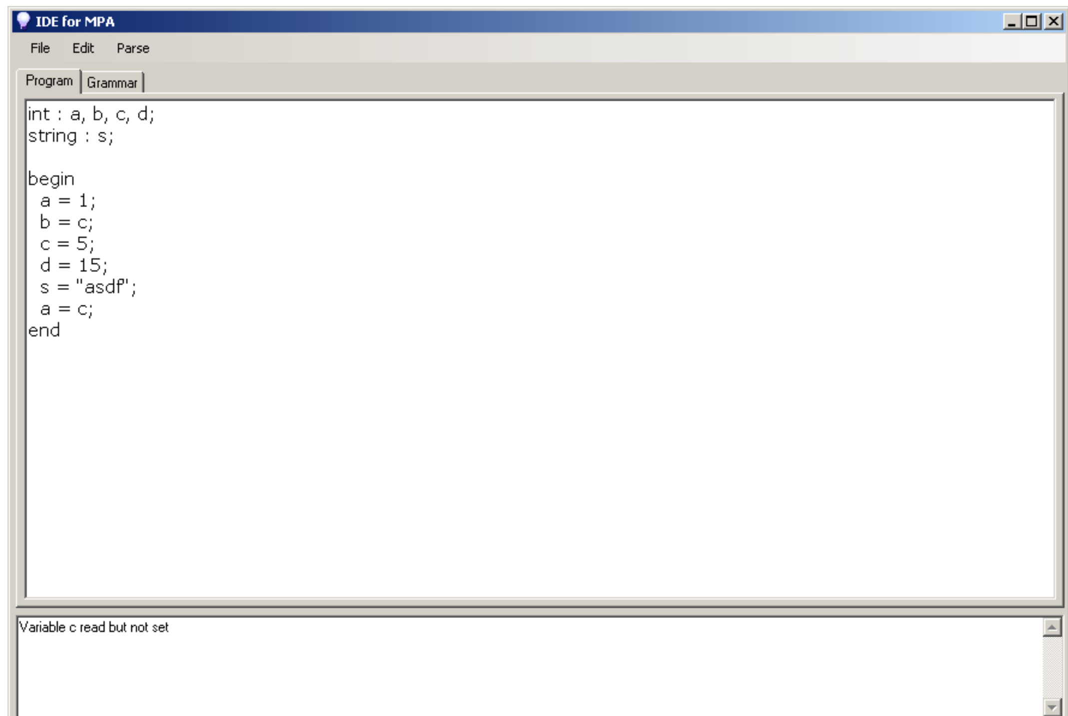


Figure 16: Read but not set variable c

The other alternative of this example is shown in Figure 16. Variable `c` is defined but its first usage is on the right side of the statement `b = c;`. The corresponding variable stack is `D USDR` and therefore the error message is:

Variable c read but not set.

The last example shows the limiting grammar also described in the previous chapter. Loaded grammar can be seen in Figure 17. The limitation is provided by the given number of nonterminals X defined on the third line.

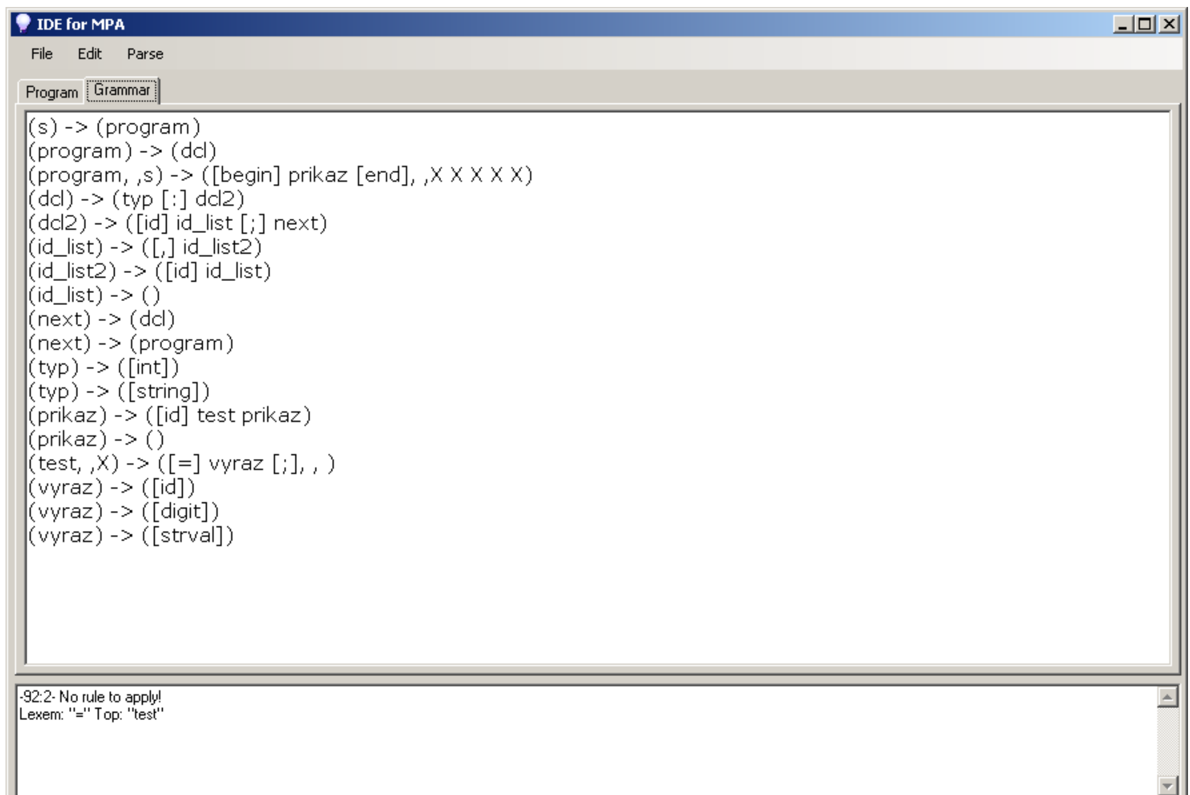


Figure 17: Limiting grammar with 5 statements

Now, if we modify the program to have only 5 statements, the parsing result is:

- - - Ok - - -

which means that the program has 5 or less statements (see Figure 18). If we add one more statement, the parsing will fail (see Figure 19) because the number of statements is greater than the number of nonterminals `X` and parsing will fail during processing the sixth statement. It is also obvious that combining more statements to one line does not affect the results.

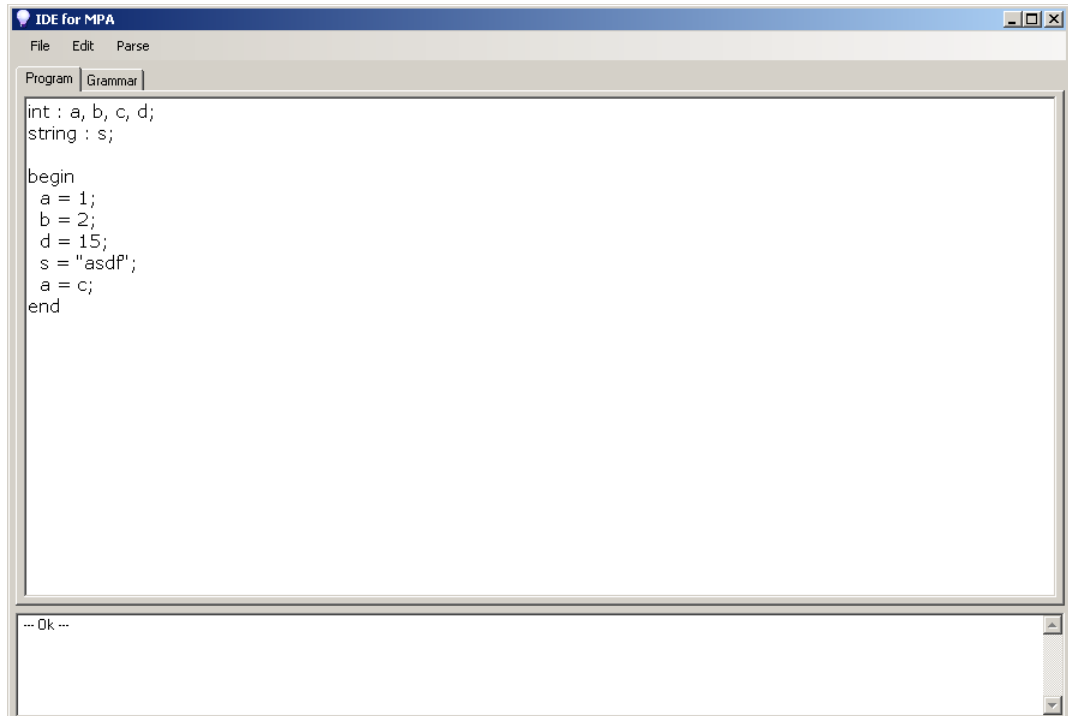


Figure 18: 5 statements

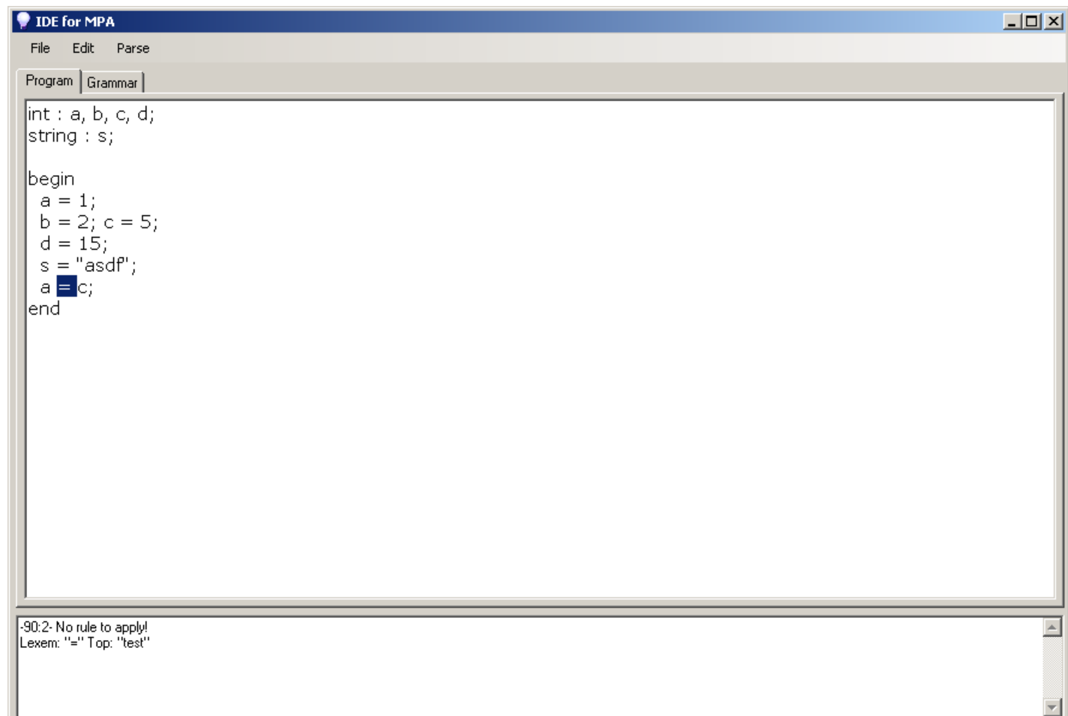


Figure 19: 6 statements

Part III
Conclusion

6 Summary

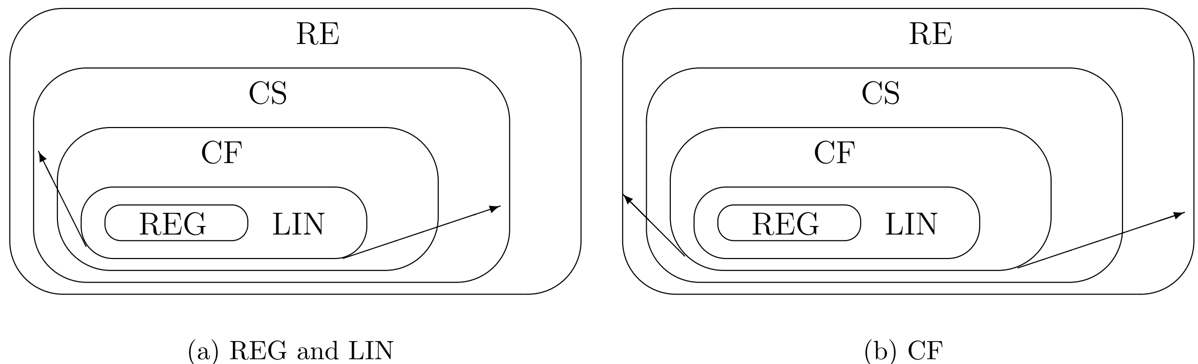
This work attempts to map grammatical models of computational distribution and concurrency. The main tool used to meet this requirement is regulated grammar. We divide regulated formal models into three parts. The first approach is to tie some productions together, as we can see in the case of matrix grammar or programmed grammar. The second approach is to permit or deny applying rules according to presence of some symbols in an actual word, as we can see in permitting and forbidding grammar or random-context grammar. The third (and the most discussed in this work) is the approach based on control language. In this case we label productions and, as a successful derivation, we only define such derivation in which corresponding production labels form a word from control language.

In the fifth chapter new results from regulated rewriting systems are presented. It starts with a definition of the formal model with start string of length n and shows that regular language with start string of length n regulated by regular language forms a language hierarchy according to n. This hierarchy is known as the Wood hierarchy. Later, in the fifth chapter, we limit the derivation position in start string to a finite number. This results in the collapsing the whole hierarchy to the set of regular languages.

The sixth chapter discusses an application of the presented models of distribution and

concurrency. The implementation of a parser based on scattered-context grammar is presented and a couple of examples of non context-free grammar is described.

The whole work is divided into two main parts, which represent two concepts of formal language regulation. The first part is based on regulation of regular and linear languages, which is interesting from the theoretical point of view. This kind of regulation strengthens the power of such formal systems towards context-sensitive languages. The other part discusses regulation of context-free languages and is mainly focused on application. Using standard parsing techniques on concurrent working context-free grammars preserves the ease of classic methods but enables the parsing non context-free programs. The best proof of this concept is the implementation of such a parser and presenting a couple of examples describing the power of a context-sensitive parser.



(a) REG and LIN

(b) CF

Figure 20: Regulation of languages

For future research, there is a wide spectrum of still not deeply mapped regulated formal systems and their properties. It is possible to study in depth pure regulated formal systems (as regular grammars with start string regulated by regular grammars) or modified ones (with a given limitation in number of changing derivation positions).

It is also possible to study regulated parallel grammars and other regulated parallel formal systems such as L-Systems. A special case of L-System, T0L system, is a parallel rewriting system which can be studied deeply as a regulated formal system.

7 Historical and Bibliographical Remarks

Grammatical models of computational distribution and concurrency have from theoretical point of view many aspects. Some of them were introduced in [Wood–73] in definition of n-parallel languages. In [Roz–73] grammar systems are described. Later some more general results were presented for example in [Sal–73] and [Das–89]. Practical applications were described in [Aho–72], [Gor–98], [Aho–07], especially in the second edition.

8 Future research

One of the promising applications of the described concept is introducing a notion of *preconditions* and *postconditions* to the ZAP03 programming language. Preconditions resp. postconditions are essentially sets of logical formulae, which are required to hold at entering resp. leaving a program or method. For example, when computing a square root of x, we can require a positivity of x using precondition $\{x >= 0\}$. A computation of sinus function $\{y = sin\ x\}$ a natural postcondition $\{(y <= 1) \wedge (y >= -1)\}$ arises.

Statements of a programming languages then induce transformation rules on precondition and postcondition sets. For example, assignment statement in the form $V := E$ defines a transformation:

$$\{P[E/V]\}V := E\{P\},$$

where V is a variable, E is an expression, P is precondition and P[E/V] denotes a substitution of V for all occurrences of E in P. Other transformation examples may be found in [Gor–98].

The purpose of introducing preconditions and postconditions into a language is to be able to derive postconditions from specified preconditions using transformation rules in a particular program. Such program then carries a formal proof of its correctness with it, which is a desirable property.

In the ZAP03 language we can implement the described concept by introducing two new keywords `pre` and `post` and by extending the rules of the language grammar with above mentioned transformation rules. When the parsing of a program is initiated, the precondition set is constructed using the `pre` declarations and the transformation rules are applied to it as the parsing progresses through the source code. When the parsing terminates, the resulting set of transformed preconditions is compared with the declared postconditions. If these two sets match, the parsed program is correct with respect to the specified preconditions and postconditions.

However, for practical reasons we must define constraints on the possible preconditions and postconditions. Postconditions must be generally derivable from preconditions or (as a corollary of the Gödels incompleteness theorem) the postconditions need not be provable from the preconditions at all, but may still hold. In this particular case we have encountered a program which may be correct, but we cannot verify this fact.

References

[Aho–72] Aho, A., Ullman, J.: *The Theory of Parsing, Translation, and Compiling*, Prentice-Hall INC. 1972.

[Aho–77] Aho, A., Ullman, J.: *Principles of Computer Desing*, Addison-Wesley, Massachusetts, 1977.

[Aho–07] Aho, A., Lam, M., Sethi, R., Ullman, J.: *Compilers: Principles, Techniques, & Tools*, 2nd ed., Addison Wesley, Boston, 2007.

[Ber–79] Berstel, J.: *Transductions and Context-Free Languages*, B. G. Treubner, Stuttgart, 1979.

[Chy–84] Chytil, M.: *Automaty a gramatiky*, Matematický seminář, SNTL, Praha, 1984.

[Das–89] Dassow, J., Păun, Gh.: *Regulated Rewriting in Formal Language Theory*, Springer, 1989.

[Gef–91] Geffert, V., Normal forms for phrase-structure grammars, *Theoretical Informatics and Applications*, Volume 25, Pages 473-496, 1991.

[Gin–66] Ginsburg, S.: *The Mathematical Theory of Context-Free Languages*, McGraw-Hill, New York, 1966.

[Gin–68] Ginsburg, S. and Spanier, E.: Finite-turn pushdown automata. *SIAM J. Control* 4, Pages 429-453, 1968.

[Gor–98] Gordon, J. C. M.: *Programming Language Theory and its Implementation*, Prentice Hall, 1998.

[Gre–69] Greibach, S.: An Infinite Hierarchy of Context-Free Languages, *Journal of the ACM*, Volume 16, Pages 91-106, 1969.

[Gre–69a] Greibach, S., Hopcroft, J.: Scattered Context Grammars, *Journal of Computer and System Sciences*, Volume 3, Pages 233-247, 1969.

[Har–78] Harrison, M. A.: *Introduction to Formal Language Theory*, Addison-Wesley, Reading, Massachusetts, 1978.

[Hop–79] Hopcroft, J., Ullman, J.: *Introduction to Automata Theory, Languages and Computation*, 2nd ed. Addison-Wesley, Massachusettes, 1979.

[Kas–70] Kasai, T.: An Hierarchy Between Context-Free and Context-Sensitive Languages. *Journal of Computer and System Sciences* 4, Pages 492-508, 1970.

[Kol–04] Kolář, D.: *Pushdown Automata: New Modifications and Transformations*, Habilitation Thesis, Brno, 2004.

[Liu–71] Liu, L. Y. and Weiner, P.: An infinite hierarchy of intersections of context-free languages, *Theory of Computing Systems*, Springer, 1971.

[Med–00] Meduna, A., Kolar, D.: Regulated Pushdown Automata, *Acta Cybernetica*, Vol. 14, Pages 653-664, 2000.

[Med–03] Meduna, A., Fernau, H.: On the degree of scattered context-sensitivity, Elsevier, *Theoretical Computer Science* 290, Pages 2121-2124, 2003.

[Med–05] Meduna, A.: *Automata and Languages: Theory and Applications*, Springer, London, 2005.

[Med–08] Meduna, A., Rychnovsky, L.: Infinite Language Hierarchy Based on Regular-Regulated Right-Linear Grammars with Start Strings. *Philippine Computing Journal*, Vol. 2, Pages 1-5, 2008.

[Oka–01] Okawa, S., Hirose, S.: Homomorphic characterizations of recursive enumerable languages with very small language classes, *Theoretical Computer Science* 250, Pages 55-69, 2001.

[Roz–73] Rozenberg, G. and Saloma, A. (eds.): *Handbook of Formal Languages*, Volumes 1 through 3, Springer, 1997.

[Rych–05] Rychnovsky, L.: *Relation between regulated pushdown automaton and Turing machine*, Report from semestral project, 2005.

[Rych–08] Rychnovsky, L.: Start String in Formal Language Theory. *In Proceedings of the 14th Conference STUDENT EEICT 2008*. Pages 422-426,, 2008.

[Rych–09] Rychnovsky, L.: Regulated Pushdown Automata Revisited, *In Proceedings of the 15th Conference STUDENT EEICT 2009*. Pages 440-444, 2009.

[Sal–73] Salomaa, A.: *Formal Languages*, Academic Press, New York, 1973.

[Wood–73] Wood, D.: Properties of n-Parallel Finite State Languages, *Utilitas Mathematica*, Winnipeg, Canada. Vol. 4, Pages 103-113, 1973.

[ZAP–03] ZAP course, Internet pages, online, cited : October 2009, http://www.fit.vutbr.cz/study/courses/ZAP/public/project/

Part IV
Appendixes

Appendix A: Program Documentation

This appendix describes implementation issues in detail.

Package *Source*

Package *Source* contains just one class `Source` which manages all communication with input program source code. It uses Java `PushbackReader` to read characters and push characters back to stream.

method Source(`String f`)

This constructor creates `PushbackReader` connected to program source code `f`. It manages all exceptions during opening the file.

method `char` getChar()

This method reads one character from input stream. It manages position in source file for printing line and position on line if some error arises during parsing. `EOLN` and `EOF`

are standard `char#13` and `char#10`. All exceptions during reading character are managed here.

method `void` ungetChar()

Lexical analyzer sometimes needs to look at the next character in source code. For example if it reads `123` and the next character is `+` then it unreads `+` and returns `123` as read lexem. Method `ungetChar()` returns the last read character back to stream for future processing and adjusts line and position on line.

method `void` close()

This method just closes the input source program.

Package *MyStack*

Package *MyStack* implements methods for working with stacks. It is possible to use standard stack class and its methods, but it is better to implement own class for more complex debug and output methods. Package Stack contains two classes, class `StackItem` and class `MyStack`.

class `StackItem`

First class only defines `StackItem` with no methods. `StackItem` only contains string as data inserted in stack and reference to another `StackItem` class as pointer to previous `StackItem`. `StackTop` is a reference to top of the stack.

class `MyStack`

Class `MyStack` uses `StackItem` to define well-known stack methods.

method `String` top()

If stack is empty returns debug string "`empty`". If stack is not empty it returns data extracted from `StackTop`.

method `int` push(String ch)

Creates a new `StackItem`, inserts data c in it, takes `StackTop` as reference to previous `StackItem` and sets actual `StackItem` as `StackTop`.

method `String` **pop()**

If stack is empty returns error. Else it takes `StackTop` and extracts data and reference to previous `StackItem` from it. Previous `StackItem` then sets as `StackTop` and returns data from popped `StackItem`.

method `void` **printStack()**

This method goes through the stack from `StackTop` until `StackItem` has empty reference to next `StackItem` and prints on standard output data strings from processed `StackItems`.

Package *StackAutomaton*

Package *StackAutomaton* implements stack automaton with its rules and moves. It consists of two classes. First class is `StackRule` and defines rules for stack automaton. Rules are in format `<Stack1, State1, Input1> -> <Stack2, State2, Input2>`. `StackRule` also contains reference to previous `StackRule`.

class `StackAutomaton`

The first class contains stack `st` and implements following methods.

method StackAutomaton(`String` **startState,** `String` **startSymbol)**

This constructor creates new stack automaton in start state `StartState` and with mark # (empty stack) and `StartSymbol` on stack.

method `int` **setRule(**`String` **stack1,** `String` **state1,** `String` **input,** `String` **stack2,** `String` **state2,** `String` **input2)**

This method adds a new rule to stack automaton. It modifies `lastRule` which points to last added rule. Every rule is in the format `<Stack1, State1, Input1> -> <Stack2, State2, Input2>`.

method `void` **printStack()**

Just calls `printStack()` method from containing stack `st`. It prints the whole stack.

method `String` **move(**`String` **input)**

This is main method of `StackAutomaton` class. It makes single move according to input. It goes through rules and if it finds corresponding rule to input and symbol on the top of the stack `st`, it makes this move.

method `String` **getActiveMove(**`String` **input)**

This method goes through all rules of `StackAutomaton` and returns rules which are applicable right now according to input string and symbol on stack.

method `String` **forceMove(**`String` **rule)**

Takes rule and applies it on `StackAutomaton`. This method is used while regulated automaton pops rule and this rule should be followed by this automaton. The string rule is in format `<Stack1, State1, Input1> -> <Stack2, State2, Input2>`.

method `String` **getState()**

This method returns state of `StackAutomaton` for debugging purposes.

method `void` **printRules()**

Prints all previously set `StackAutomaton` rules in the format `<Stack1, State1, Input1> -> <Stack2, State2, Input2>`.

class `StackRule`

This class just holds one stack rule and holds pointer to next stack rule.

Package *RPDA*

Package *RPDA* extends class `StackAutomaton` and implements behavior of regulated pushdown automaton. When `RPDA` is completely loaded (`loadPDA` and `loadGRM`) it can simulate `RPDA` moves.

class `RPDA` **extends** `StackAutomaton`

The first class contains stack `St2` and implements following methods.

method RPDA(`String` **startState1,** `String` **startSymbol1,** `String` **startSymbol2)**

Constructor of `RPDA`. `RPDA` is then prepared for `loadPDA` and `loadGRM` to set up the rules.

method `int` **setGRM(**`String` **NT1,** `String[]` **terminal1,** `String[]` **NT2,** `String[]` **terminal2,** `String[]` **NT3)**

This method is called by `loadGRM`. `loadGRM` loads rule from file and sets corresponding rule.

method String **moveGRM(**String **input)**

This method makes one single move of RPDA according to input. Main routine takes input word and moves RPDA by each input character.

method void **printGRM()**

Prints rules of regulating pushdown automaton. For debugging purposes.

method void **printStack1()**

Prints first of two RPDA stacks. Stack1 is stack of PDA.

method void **printStack2()**

Prints second of two RPDA stacks. Stack2 is stack of regulating pushdown automaton.

class StackRule

This class just holds one stack rule and holds pointer to next stack rule.

class loadPDA

Loads RPDA's PDA from PDA file and returns partially constructed RPDA.

method RPDA **load(**String **PDAfile,** RPDA **newRPDA)**

This method do all the work. It's implemented as finite automaton which goes through input file and scans the rules. The rules are in PDA file in format <Stack1, State1, Input1> -> <Stack2, State2, Input2>.

class loadGRM

Loads RPDA's controlling language from GRM file. Returns complete RPDA.

method RPDA **load(**String **GRMfile,** RPDA **newRPDA)**

This method is again mplemented as finite automaton which goes through input file and scans the rules. The format of rule in GRM file is
L => <A,Ab2B,> -> <,Ab2B,>L<,Ab1rb,> -> <A,Ab1rb,>.

Package *MultiStack*

Package *MultiStack* implements main tool for managing scattered-context grammars – multi stack machine. The only class in this package is MultiStack. MultiStack consists of public property Map<Integer, MyStack> Stacks, that is integer indexed set of MyStacks.

method MultiStack(`String` s)

This constructor of class `MultiStack` initializes multi stack machine and sets start symbol to s.

method `MyStack` get(`int` i)

This method just selects i-th stack from multi stack machine and returns ordinary stack machine `MyStack` where `push(...)` and `pop()` methods can be invoked.

Package *SymbolTable*

Package *SymbolTable* implements only one class `SymbolTable` using `Map<String, Lexem>`.

method `Lexem` add(`String` symbol)

This method takes string symbol from lexical analyzer and checks whether this symbol is already included in symbol table. If so, directly returns corresponding lexem. In other case this symbol is added to symbol table and new lexem is returned.

method `void` print(`int` num)

This method is just for debugging purposes. It prints whole symbol table and also stack from multi stack number num.

method `void` check(`int` num)

`SymbolTable` class is implementing one more feature than standard symbol tables. This method goes through symbol table and checks every identifier's multistack for correct usage of this identifier.

Package *Lex*

Package *Lex* implements Lexical analyzer and its main method `getLexem()`. It consist of two classes, class `Lexem` and class `Lex`.

class `Lexem`

Class `Lexem` defines structure of lexems produced by lexical analyzer. This structure consists of lexem type (e.g. ID), corresponding string representation (e.g. name of the identifier) and multi stack machine for collection information about parsing.

class `Lex`

Class `Lex` implements functionality of lexical analyzer.

method Lex(String file)

This constructor of class Lex just connects itself to Source file to read single characters.

method Lexem getLexem()

Implementation of lexical analyzer is done by finite automaton. This method is one large switch command with 5 cases. The first (default) case 0 changes state to 1 if the first character is letter, to 2 if the first character is digit, to 7 if the first character is = (as part of => sign) and to 6 if the character is ". Finally it directly returns the lexem if the character is one of these: +, -, *, :, ,, ;, ?. The second case 1 means that we read a letter as previous character. If this time read character is not letter, we put back this character and return lexem (in cooperation with SymbolTable class). Case 2 means that we are reading a digit and until we reads digits we stay in this case. In other case we return corresponding lexem of type digit. Case 6 means that we are reading something between " marks and case 7 is only for => and =< signs.

method void close()

Method close() just closes previously opened Source file.

Package *SCG*

Package *SCG* implements methods and data structures for scattered-context grammars. It has basic properties Set<String> Rules, Set<String> terminals and Set<String> nonTerminals. Also derived (and computed) properties Map<String, String> Empty, Map<String, Set<String>> First, Map<String, Set<String>> Follow and of course Map<Integer, Set<String>> Predict. All this methods (properties) are the same as in CF version of grammar. New properties (special for SCG) are Set<String> FirstEmpty and Map<String, List<String>> EmptyList.

method SCG(String start)

Constructor for this method just sets the start symbol of new created grammar.

method int setRule(List<String> L, List<List<String>> R)

This method takes rules in the form (A,B,C) -> (a A,b B,c C) translated to left and right parts List<String> L, List<List<String>> R.

method void printRule(int i)

This method is for debugging purposes, it just prints out i-th rule.

method `void` **printRules()**

This method prints the whole grammar for debugging purposes.

method `List<String>` **getRRule(**`int` **i,** `int` **k)**

This method is getter (extractor) of the k-th part of right side of the i-th rule.

method `String` **getLRule(**`int` **i,** `int` **k)**

This method is getter of the k-th part of left side of the i-th rule.

method `int` **getNRules(**`int` **i)**

This method returns the number of components of the i-th rule. For example for the rule $(A, B, C) \rightarrow (aA, bB, cC)$ it returns 3.

method `int` **computeEmpty()**

This method computes set EMPTY set and stores it in `Map<String, String> Empty`. It stores in the first string nonterminal and in the second string corresponding empty flag.

method `int` **computeFirst()**

This method computes set FIRST set and stores it in `Map<String, Set<String>> First`. It stores in the first string terminal or nonterminal and in the following set corresponding FIRST set.

method `int` **computePredict()**

This method computes PREDICT and stores it in `Map<Integer, Set<String>> Predict`. It stores in the first `Integer` rule number and in the following set corresponding terminals.

method `Set<String>` **getFirstN(**`List<String>` **X)**

This method computes and returns FIRST set of string X. The string can consist of multiple nonterminals and terminals.

method `String` **getEmptyN(**`List<String>` **X)**

This method computes and returns EMPTY set of string X. The string can consist of multiple nonterminals and terminals.

method `int` computeFollow()

This method computes set FOLLOW and stores it in public `Map<String, Set<String>> Follow`. It stores in the first string terminal or nonterminal and in the following set corresponding terminals from FOLLOW set.

method `int` printLLtable()

This method prints the whole LL table. Only for debugging purposes.

method `int` LLtable(`String` Term, `String` NonT, `List<String>` Stacks)

This method returns the rule number from LL table for loaded scattered-context grammar. It takes nonterminal NonT and terminal Term and computes number of rule which is located on corresponding position in LL table. Because we are in scattered-context grammars there can be more than one rule to apply. For example if there are rules $(A, B) \to (aA, bB)$ and $(A, C) \to (aA, cC)$ it is impossible for standard LL table to decide which rule apply. Now comes stacks into play. This method checks all the actual stacks and chooses the one that exactly matches.

Class `loadSCG`

This class is the factory for creating scattered-context grammars from package *SCG*. All it does is open file and translate text rules of the form `(A,B,C) -> (a A,b B,c C)` to full defined scattered-context grammar.

method `SCG` load(`String` SCGfile, `SCG` newSCG)

This class is again implemented as finite automaton. Automaton walks through input stream and expects one rule per line. Every line than automaton scans and changes it's state as it reaches various tokens (such as `(`, `,`, `)`, `->`). Once it reaches final `)`, it converts all left and right parts of the rule to the `SCG` rule (left sides to `List<String>` and right to `List<List<String>>`).

Class `testSCG`

This is the main class for presenting overall result. It implements just one method `main(...)`.

method `void` main (`String` args[])

This function does all the work. First it handles input arguments. The two required arguments are `-grm` which loads corresponding SCG grammar and `-prg` which loads corresponding program.

```
Usage:  testSCG -grm grammar.SCG -prg program.txt
```

Next this method creates new `SCG` grammar and loads it from file

```
SCG myGram = new SCG("s");
loadSCG.load(fGram,myGram);
```

Then all necessary computations are made

```
myGram.computeEmpty();
myGram.computeFirst();
myGram.computeFollow();
myGram.computePredict();
```

New multi stack machine for simulating scattered-context grammar is created.

```
MultiStack mst = new MultiStack("s");
```

Lexical analyzer is initialized and connected to program file.

```
Lex input = new Lex(fPrg);
Lexem lexem = input.getLexem();
```

All following work is done in while loop for every scanned lexem

```
while (!lexem.type.equals("eof")) {
   ...
}
```

Because we are implementing recursive descent parser we erase top of the stack if it matches input lexem

```
top = mst.get(1).pop();
if (lexem.type.equals(top)) {
  lexem = input.getLexem();
} else {
```

Next we create structure `List<String> Lrules` that is the list of top elements from all stacks of `mst`. The construction is a little bit tricky

```
Lrules = new ArrayList<String>();
for (i = 1; i < myGram.Cardinality+2; i++) {
```

```
  Lrules.add("");
}
for (i = 1; i < myGram.Cardinality+2; i++) {
  if (i%2 == 0) {
    if (lexem.type.equals("id")) {
      Lrules.set(i-1,lexem.mstack.get(i-1).top());
    }
  } else {
    if (i==1) {
      Lrules.set(i-1,top);
    } else {
      Lrules.set(i-1,mst.get(i-1).top());
    }
  }
}
```

Next we choose the only one rule to apply according to lexem and top of all stacks (`Lrules`)

```
r = myGram.LLtable(lexem.type, top, Lrules);

if (r == 0) {
  System.err.println("No rule to apply!");
  return;
}
```

The rest of the while loop is to push all right sides of rule number `r` to corresponding stack of `mst`.

A.1 Installation

The whole project is divided into two parts – the parser and the user interface. The parser is implemented in Java and only needs to be unpacked to a directory (eg. c:\Project\Parser\).

To compile the parser it is necessary to install Java version 1.6 or higher. Compilation starts with

```
cd src
javac -d ..\classes testSCG.java
```

To verify the parser run the following command

```
cd ..\classes
java testSCG -prg ..\examples\program.txt -grm ..\examples\limit.SCG
```

To compile the user interface it is necessary to install .NET Framework 2.0 or higher and Visual Studio 2005 or higher. The user interface is implemented in C# and needs to be unpacked to a directory (eg. c:\Project\IDE\). First edit the file `App.config` and modify `WorkingDirectory` to the path to classes of parser (eg. c:\Project\Parser\classes) and `javapath` to the path to your Java interpreter. Then run Visual Studio, open MPAide project from c:\Project\IDE\MPAide.sln. Now you can build and run the user interface working together with the parser.

Appendix B: Authors Curriculum vitæ

Lukáš Rychnovský

Faculty of Information Technology, Brno University of Technology, Czech Republic
e-mail: `rychnov@fit.vutbr.cz`

EDUCATION

2006–today	postgraduate student (PhD thesis topic: "Regulated Rewriting", supervisor: Prof. RNDr. Alexander Meduna, CSc.), Department of Information Systems, Faculty of Information Technology, Brno University of Technology
June 2006	masters degree, Informatics (master thesis topic: "Parsing of Context-Sensitive Languages"), Department of Information Systems, Faculty of Information Technology, Brno University of Technology
1999–2006	undergraduate student, Informatics, Faculty of Information Technology, Brno University of Technology
June 2003	masters degree, Mathematical Analysis (master thesis topic: "Wavelets and Multiresolution Analysis"), Department of Mathematical Analysis, Faculty of Science, Masaryk University Brno
1998–2003	undergraduate student, Mathematic Analysis, Department of Mathematical Analysis, Faculty of Science, Masaryk University Brno

TEACHING AND ACADEMIC ACTIVITIES

2007	FR673/2007/G1 – Innovative Approach to the Compiler Projects
2006	Faculty of Information Technology, Brno University of Technology – IZP: Introduction to Programming Systems (seminar tutor)
2006	Faculty of Science, Masaryk University Brno – PV178: Programming for the CLI Environment (seminar tutor)
2004	Faculty of Science, Masaryk University Brno – M4180: Numerical Methods (seminar tutor)
2003	Faculty of Informatics, Masaryk University Brno – MA012: Statistics II (seminar tutor)
2002	Faculty of Informatics, Masaryk University Brno – MB003: Linear Algebra (seminar tutor)