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.

Contents

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 3.3, 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.

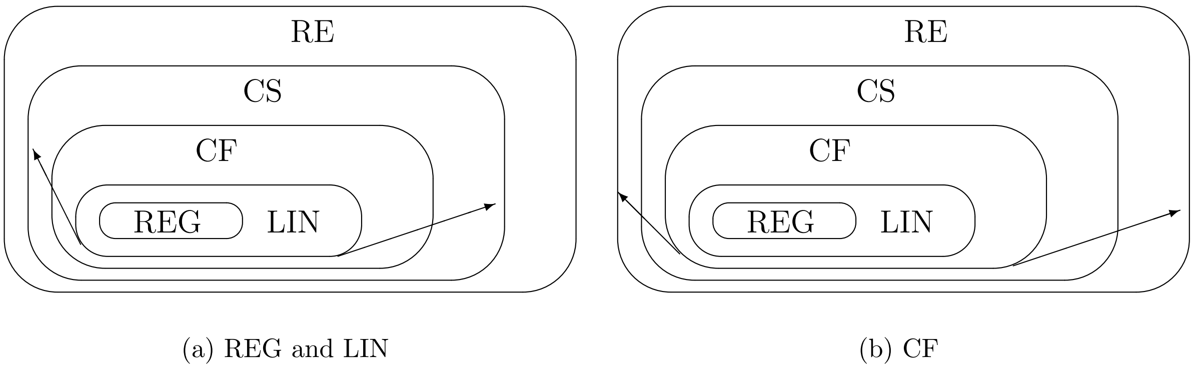


Figure 1: Regulation of languages

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 Definitions

In this section we define some notations and models in formal language theory. We omit basic notations such as rewriting system, automata and grammars.

2.1 Regulated Rewriting

Definition 2.1. 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 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

$$m = [P_1 \to Q_1, \ldots, P_i \to Q_i],\ i \geq 1. \tag{2.1}$$

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 (2.1) 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 $i = 1, \ldots, r$.
$Mat = \{L \mid L = L(G)$, where $G = (N, T, R, S)$ is a matrix grammar$\}$.

Definition 2.2. 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) $P = P_1 P' P_2$ and $Q = P_1 Q' P_2$, (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 2.3. 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 of a grammar, but P is a finite set of random-context rules, that is, triplets in the form of $(C \rightarrow \alpha, Q, R)$, $C \rightarrow \alpha$ 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 $x = x_1 C x_2$, $y = x_1 \alpha x_2$ for some $x_1, x_2 \in (N \cup T)^*$, $(C \rightarrow \alpha, Q, R)$ is a triplet in P, all symbols of Q appear and no symbol of R appears in $x_1 x_2$ (Q is called the permitting context, and R is called the forbidding context of the rule $C \rightarrow \alpha$. If Q and/or R are empty, then no check is necessary.)

Definition 2.4. (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. $S \rightarrow X_1 \ldots X_n, \quad X_i \in N_i, 1 \leq i \leq n,$
  2. $X \rightarrow aY, \quad X, Y \in N_i$, for some $1 \leq i \leq n, a \in T^*$, and
  3. $X \rightarrow a, \quad X \in N_i$, 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 $S \rightarrow y \in P$,
- or $x = y_1 X_1 \ldots y_n X_n, y = y_1 x_1 \ldots y_n x_n$, where $y_i \in T^*, x_i \in T^* N \cup T^*, X_i \in N_i$, and $X_i \rightarrow x_i \in P, 1 \leq i \leq n$.

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

Theorem 2.1 (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 2.5. *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 of a grammar, but P is a finite set of production rules of the form

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

Let $(A_1, A_2, \ldots, A_n) \rightarrow (w_1, w_2, \ldots, w_n) \in P$ 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 2.1.

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

where

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

The language generated by grammar G is

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

2.2 Control Languages

Definition 2.6. 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, $u \rightarrow v \in P$, to ρ, where $\rho \in \Psi$, we write $\rho.u \rightarrow v \in P$; in other words, $\rho.u \rightarrow v$ means $\psi(u \rightarrow v) = \rho$.

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

Definition 2.7. Let $G = (N, T, P, S)$ be a grammar. 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

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

Definition 2.8. 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

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

Theorem 2.2.

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

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

Definition 2.9. 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 2.3. *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

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

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 2.4. *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 2.5. *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.*

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.

Part II
New Grammatical Models of Distribution and Concurrency

3 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].

3.1 Definitions

Definition 3.1. 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 $A \rightarrow x$, 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 $\text{card}(\Psi) = \text{card}(R)$, and ψ be a bijection from R to Ψ. For simplicity, to express that ψ maps a rule $A \rightarrow x \in R$, to ρ, where $\rho \in \Psi$, we write $\rho.A \rightarrow x \in R$; in other words, $\rho.A \rightarrow x$ means $\psi(A \rightarrow x) = \rho$.

If $\rho.A \rightarrow x \in R$ 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 3.2. Let $G = (N, T, R, S)$ be an n-LG for some $n \geq 1$ (See Definition 3.1). $G = (N_1, N_2, \ldots, N_n, T, R_1, R_2, \ldots, R_n, S)$ 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 $\Psi_i = \{\rho \mid \rho.A \to aBb \in R_i \text{ or } \rho.A \to a \in R_i, \text{where } A, B \in N_i \text{ and } a, b \in T^*\}$.

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 3.3. (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. $S \to X_1 \dots X_n, \quad X_i \in N_i, 1 \le i \le n,$
3. $(X_1 \to x_1, X_2 \to x_2, \dots, X_n \to x_n),$
   $X_i \in N_i, x_i \in T^*N_iT^* \cup T^*, 1 \le i \le n.$

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 $x = y_1X_1 \dots y_nX_n, y = y_1x_1 \dots y_nx_n,$
  where $y_i \in T^*, x_i \in T^*N_iT^* \cup T^*, X_i \in N_i, 1 \le i \le n$
  and $(X_1 \to x_1, \dots, X_n \to x_n) \in P$.

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

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

Definition 3.4. Let $i \ge 1$ and X be a family of languages. Let $L(G, \Xi)$ be a language generated by G and regulated by Ξ. 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 $G = (N_1, N_2, \dots, N_n, T, R_1, R_2, \dots, R_n, S)$
  is a $i-{}_{dis}$RLG and $\Xi \in X\}$.
- ${}_{dis}\mathfrak{L}(X, i) = \{L \mid L = L(G, \Xi)$,
  where $G = (N_1, N_2, \dots, N_n, T, R_1, R_2, \dots, R_n, S)$
  is a $i-{}_{dis}$LG and $\Xi \in X\}$.
- ${}_{SM}\mathfrak{R}(i) = \{L \mid L = L(G)$,
  where $G = (N_1, N_2, \dots, N_n, T, R, S)$ is a i-RLSM$\}$.
- ${}_{SM}\mathfrak{L}(i) = \{L \mid L = L(G)$,
  where $G = (N_1, N_2, \dots, N_n, T, S, P)$ is a i-LSM$\}$.

3.2 New Results

Lemma 3.1. *For every n-LG $G = (N, T, R, S)$, there exists an equivalent $n-{}_{dis}$LG $G' = (N'_1, N'_2, \dots, N'_n, T', R'_1, R'_2, \dots, R'_n, S')$ 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 $1 \leq j \leq n$;
- $T' = T$;
- $R'_j = \{(A, j) \rightarrow x(B, j)y \mid A \rightarrow xBy \in R,$
  $(A, i), (B, i) \in N'_j,\ x, y \in T^*\}$ where $1 \leq j \leq n$;
- $S' = (A_1, 1)(A_2, 2) \dots (A_n, n)$, where $S = A_1A_2 \dots A_n$.

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

Lemma 3.2. *For every $n{-}_{dis}LG$ $G' = (N'_1, N'_2, \dots, N'_n, T', R'_1, R'_2, \dots, R'_n, S')$, 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

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

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

Theorem 3.1. *For all $n \geq 1$, $\mathcal{L}(n{-}_{dis}LG) = \mathcal{L}(n\text{-}LG)$.*

Proof. This theorem directly follows from Lemma 3.1 and Lemma 3.2. □

Theorem 3.2. *For all $n \geq 1$, $\mathcal{L}(n{-}_{dis}RLG) = \mathcal{L}(n\text{-}RLG)$.*

Proof. This theorem directly follows from Theorem 3.1. □

Lemma 3.3. *Let $i \geq 1$. ${}_{dis}\mathfrak{L}(REG, i) \subseteq {}_{SM}\mathfrak{L}(i)$. That is,*
for every $n{-}_{dis}LG$ $G = (N_1, \dots, N_n, T, R_1, \dots, R_n, S)$ regulated by regular language Ξ there exists equivalent n-LSM $G' = (N'_1, \dots, N'_n, T', S', P')$ such that $L(G) = L(G')$.

Proof. Rigorous proof can be found in the thesis. □

Lemma 3.4. *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
  $(A_1, \ldots, A_i, \ldots, A_n) \to$
  $\to (u_1B_1v_1, \ldots, u_iB_iv_i, \ldots, u_nB_nv_n) \in P'$,
  $u_i, v_i \in T^*,\ 1 \le j \le |P'|\},\ 1 \le i \le n.$

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

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

□

Theorem 3.3. *For all $i \ge 1$, ${}_{dis}\mathfrak{L}(REG, i) = {}_{SM}\mathfrak{L}(i)$.*

Proof. This theorem directly follows from Lemma 3.3 and Lemma 3.4 □

Theorem 3.4. *For all $i \ge 1$, ${}_{dis}\mathfrak{R}(REG, i) = {}_{SM}\mathfrak{R}(i)$.*

Proof. This theorem directly follows from Theorem 3.3. □

Theorem 3.5. *For all $i \ge 1$, ${}_{SM}\mathfrak{L}(i) \subset {}_{SM}\mathfrak{L}(i+1)$.*

Proof. See [Das–89]. □

The main result of this paper follows next.

Theorem 3.6. *For all $i \ge 1$,*
$$\mathfrak{L}(REG, i) \subset \mathfrak{L}(REG, i+1).$$

Proof. This theorem follows from Theorems 3.1, 3.3 and 3.5. □

Theorem 3.7. *For all $i \ge 1$,*
$$\mathfrak{R}(REG, i) \subset \mathfrak{R}(REG, i+1).$$

Proof. This theorem follows from Theorem 3.6. □

Let G be an $n{-}_{dis}$RLG satisfying Definition 3.2. Let $S \Rightarrow^* w\ [\sigma]$, $w \in T^*, \sigma = \rho_1\rho_2 \dots \rho_m$, for some $m \geq 1, 1 \leq i \leq m, \rho_i \in \Psi, \sigma \in \Xi$.
Set

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

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 $G = (N_1, N_2, \dots, N_n, T, R_1, R_2, \dots, R_n, S)$ 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 3.8. *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 3.1. □

Theorem 3.9. *For any $n, k \geq 1$, $\mathfrak{R}(REG, n, k) \subseteq REG$. That is,
let $G = (N_1, N_2, \dots, N_n, T, R_1, R_2, \dots, R_n, S)$ 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. This proof can be found in [Med–08]. □

As opposed to Theorem 3.6, 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 3.10.

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

Proof. $REG = \mathfrak{R}(REG, 1, 0) \subseteq \mathfrak{R}(REG, n, k) \subseteq REG$ (see Theorem 3.9). □

4 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.

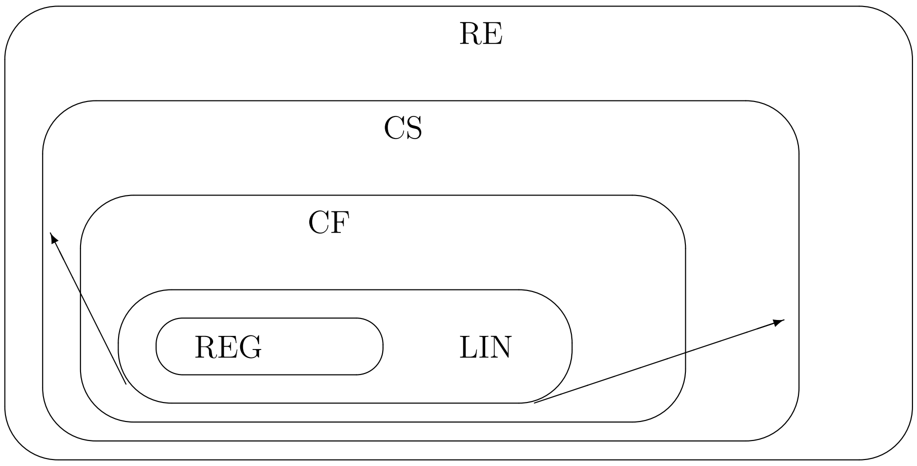


Figure 2: Regulation of REG and LIN languages

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.

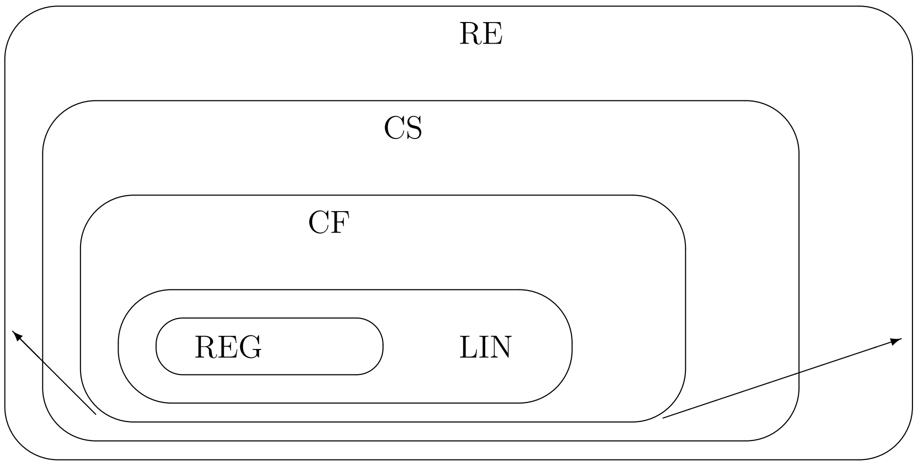


Figure 3: Regulation of CF languages

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.

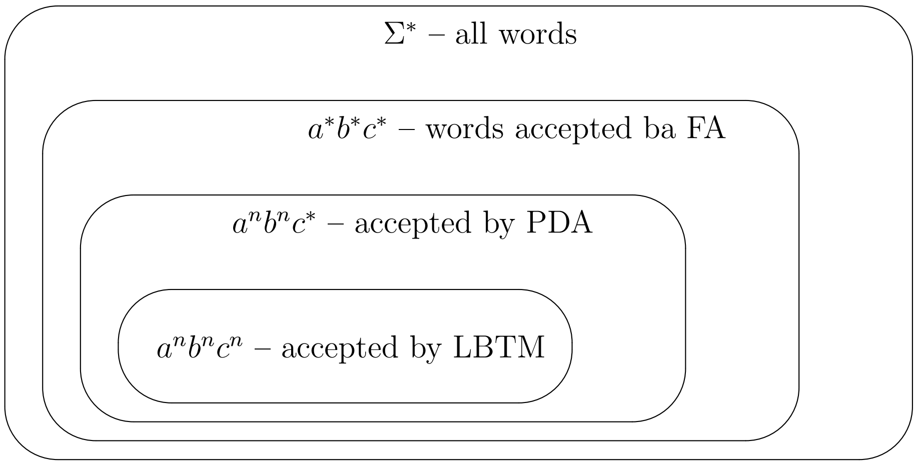

Σ^* – all words
$a^*b^*c^*$ – words accepted ba FA
$a^nb^nc^*$ – accepted by PDA
$a^nb^nc^n$ – accepted by LBTM

Figure 4: Another look at language hierarchy

In Figure 4 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 5 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.

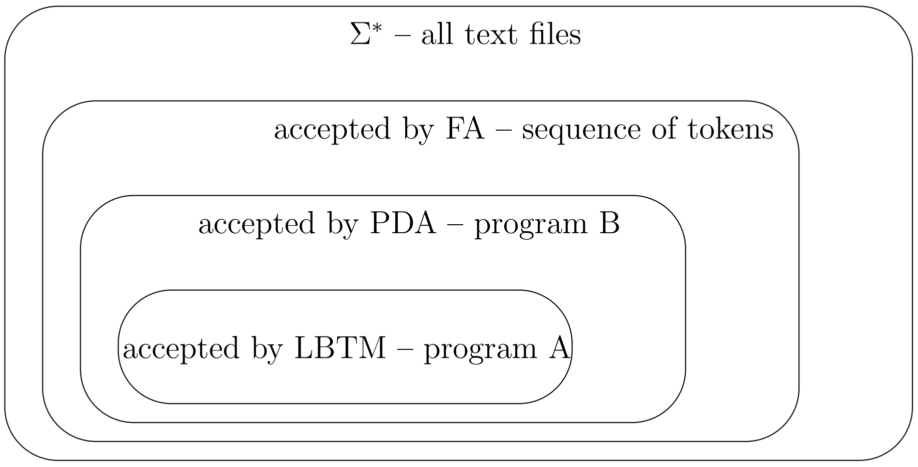

Σ^* – all text files
accepted by FA – sequence of tokens
accepted by PDA – program B
accepted by LBTM – program A

Figure 5: 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.

4.1 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. 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.

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 6 does not change the semantics. The string 3+5+1 has two parse trees but the semantics is 9 in both cases.

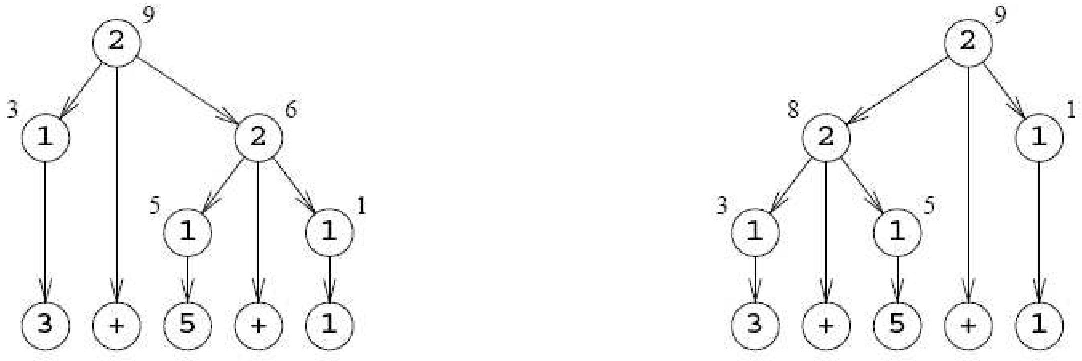


Figure 6: False ambiguity

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

To reconstruct the parse tree we need the parse technique. If we look into literature [Aho–72] or [Aho–07] 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 4.1. 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 4.2. 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 4.3. Condition FF holds if for every set of production rules:

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

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

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

Definition 4.4. Condition FFL holds if for every set of production rules:

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

such that

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

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

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

Definition 4.5. A context-free grammar G is LL_1 *grammar* if conditions FF and FFL are satisfied for the G.

Definition 4.6. 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 4.7. 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 4.8. 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 $\alpha(A, a)$.

4.2 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` $\rightarrow$ `TYPE [:]  [id] ID_LIST`

ID_LIST $\rightarrow$ [,] [id] ID_LIST
ID_LIST $\rightarrow \varepsilon$
Fragment of assignment statement
COMMAND $\rightarrow$ [id] CMD COMMAND
CMD $\rightarrow$ [=] STMT [;]
And usage variable in command
STMT $\rightarrow$ [id] OPER
OPER $\rightarrow \varepsilon$.
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.
```

4.3 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:

$(s) \rightarrow (program)$
$(program) \rightarrow (dcl)$
$(program) \rightarrow ([begin]command[end])$
$(dcl) \rightarrow (type[:]dcl2)$
$(dcl2, S, INT, ,) \rightarrow$
$\quad ([id]id\_list[;]next, D, INT, INT)$
$(dcl2, S, STR, ,) \rightarrow$
$\quad ([id]id\_list[;]next, D, STR, STR)$
$(id\_list) \rightarrow ([,]id_{l}ist2)$
$(id\_list2, S) \rightarrow ([id]id\_list, D)$
$(id\_list) \rightarrow (\varepsilon)$
$(next) \rightarrow (dcl)$
$(next) \rightarrow (program)$
$(type, ,) \rightarrow ([int], , INT)$
$(type, ,) \rightarrow ([string], , STR)$
$(command, D, INT, ,) \rightarrow$
$\quad ([id]\ cmd\ command, D\ L, INT, INT)$
$(command, D, STR, ,) \rightarrow$
$\quad ([id]\ cmd\ command, D\ L, STR, STR)$
$(command) \rightarrow (\varepsilon)$
$(cmd) \rightarrow ([=]stmt[;])$
$(stmt, D) \rightarrow ([id], D\ R)$
$(stmt, , , , INT) \rightarrow ([digit], , , ,)$
$(stmt, , , , STR) \rightarrow ([strval], , , ,)$

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, c;
begin
  a = 1;
  c = "foo";
end
```

because `c = "foo"` is not type correct (STR is assigned to INT), parsing will fail.
`(S, S,` ε, ε, ε`)` $\Rightarrow^*$ `(DCL, S,` ε, ε, ε`)` $\Rightarrow$ `(TYPE [:] DCL2, S,` ε, ε, ε`)` $\Rightarrow$
$\Rightarrow$`([int] [:] DCL2, S, INT,` ε, ε`)` $\Rightarrow^2$ `(DCL2, S, INT,` ε, ε`)` $\Rightarrow$
$\Rightarrow$`([id] ID_LIST [;] NEXT, D, INT, INT,` ε`)` $\Rightarrow^*$
$\Rightarrow^*$ `(COMMAND [end], D, INT, INT,` ε`)` $\Rightarrow$
$\Rightarrow$ `([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 → INT and STR → STR). It is very easy to generalize this approach to construct general type injector into a context-free grammar.

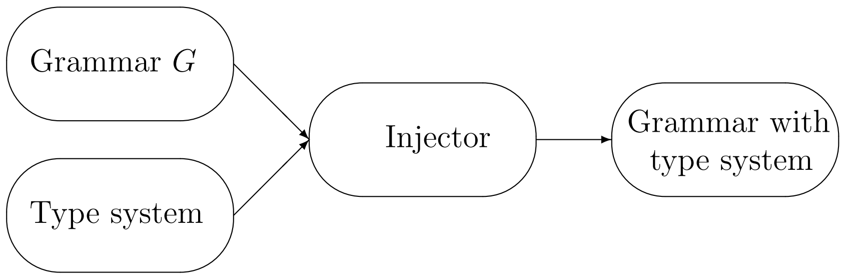


Figure 7: Injecting type system into grammar

4.4 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 $\mathrm{S}'' \rightarrow \mathrm{A}$ (corresponds to `abstract`) and $\mathrm{S}'' \rightarrow \mathrm{F}$ (corresponds to `final`). Obviously only one of the rules can be used at a time.
3. 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.

Part III
Conclusion

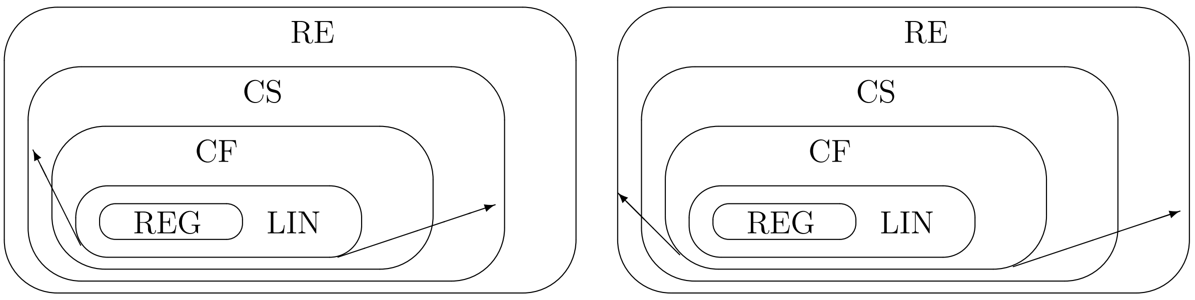
5 Summary

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 8: 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.

6 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], [Aho–07], especially in the second edition.

7 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.

[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.

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

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

[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.

[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/

8 Authors Publications

9 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)