VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

A NEW APPROACH TO LL AND LR PARSING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE ŠTEFAN MARTIČEK
AUTHOR

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NOVÝ PŘÍSTUP K LL A LR SYNTAKTICKÉ ANALÝZE
A NEW APPROACH TO LL AND LR PARSING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE ŠTEFAN MARTIČEK
AUTHOR

VEDOUCÍ PRÁCE Prof. RNDr. ALEXANDER MEDUNA, CSc.
SUPERVISOR

BRNO 2015

Abstrakt

Cílem této práce je vytvořit nový efektivní způsob syntaktické analýzy propojením LL a LR přístupů. Pro demonstrační účely je zhotoven nový programovací jazyk podle vzoru programovacího jazyka PHP. Tento jazyk je rozdělen na části, kde pro každou část je použita ta nejvhodnejší ze zmíněných metod. Jednotlivé metody jsou zde podrobněji popsané v kontextu dvou typů přístupů. Jedním z nich je syntaktická analýza shora dolů a tím druhým opačná verze, syntaktická analýza zdola nahoru. Pro každou separovanou část je vytvořen samostatný syntaktický analyzátor. Táto práce poskytuje kompletní teoretický základ k sestrojení všech zde použitých syntaktických analyzátorů a rozkladových tabulek. Nakonec jsou sestrojené analyzátory společne propojeny, což je úspěšné zakončení praktické demonstrace naší metody. V závěru jsou diskutovány dosažené výsledky práce jako efektivnejší druh syntaktické analýzy, modularita přístupu a podobně. Je zde také diskutovaná použitelnost navržené metody za účelem zefektivnení vývoje a rychlosti překladu. Jako poslední jsou uvedeny náměty pro další výzkum v této oblasti.

Abstract

The aim of this thesis is to create a new effective parsing method via connection of LL and LR approaches. For demonstration purpose is made a new programming language according to the pattern of PHP. The language is separated into the sections and for constituent sections is chosen the most appropriate from the mentioned methods. For every section is created its own syntax analyser. The thesis provides a complete theoretical basis to construct every syntax analyser that has been used here. Finally, the syntax analysers are connected together and new method is practically presented. In conclusion, contributions of this work are discussed, such as the faster parser or the improved development. It also discusses usability of the designed method and suggestions for the next possible research in this area.

Klíčová slova

LL-parser, LR-parser, bezkontextová gramatika, Panic-mode zotavení z chyb, Phrase-level zotavení z chyb, spojení syntaktických analyzátorů

Keywords

LL-parser, LR-parser, context-free grammar, Panic-mode error recovery, Phrase-level error recovery, connection of syntax analysers

Citace

Štefan Martiček: A New Approach to LL and LR Parsing, bakalářská práce, Brno, FIT VUT v Brně, 2015

A New Approach to LL and LR Parsing

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana prof. RNDr. Alexandra Medunu, CSc. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Štefan Martiček
July 30, 2015

Poděkování

Tímto bych rád poděkoval panu Prof. Alexandru Medunovi za jeho kvalitní odbornou pomoc a ochotu spolupracovat i na dálku. Dále bych rád poděkoval Prof. Wolfgang Auer za jeho cenné rady a konstruktivní kritiku. Nakonec bych rád poděkoval Prof. Regine Bolter za její vynikající kurs, který poskytl cenné informace a rady k tvorbě této bakalářské práce.

Contents

Chapter 1

Introduction

1.1 Motivation

The world is pretty fast these days and who is effective and able to keep the pace with modern trends is successful. We apply the principles of effectiveness that also bring innovations to what we do in every sphere of our activity. Companies pay for specialists to invent new technologies, sports teams have specialists to provide them with new strategies, and so on. The IT realm is spreading with relentless speed all over the world. With technical innovations of hardware in computing we can see also a huge improvement in software. Computer programmes are more complex, longer and thus the compiling process is often very slow. The most effective way and evolving method today to solve the problem of speed is parallel computing. Let us get off the mainstream subject of parallel computing and try to look at another area.
The structure of a basketball team is the perfect example of what is being discussed in this thesis. Usually the tallest players play the position near the basket where they are the most effective. They can use their long hands to get rebounds and score with ease from a close distance. Contrariwise, small players are suitable to play further from the basket. This is because they are quicker than someone who is tall with better motor skills that allows them to have better ball handling and better shooting ability, even from a further distance.
Let us try to apply this principle into the syntax analysis. So far, there has been many parsing methods invented and usually all of them have some advantages and disadvantages just as our basketball players. The basic idea is to try to separate programming language into the sections and use different parsing methods to make a syntax analysis of the given sections. For every section we are going to choose the method that is more proper considering its advantages and disadvantages. The goal of the thesis is not to build the new faster parsing method that can face competition from other parsing methods. It is just to demonstrate our idea on the simple example and to discuss new findings as well as results and the next development of this concept. For a demonstration purpose, we will use an LL and LR parsers as the most commonly used parsing methods. We suppose that using an LL and LR syntax analysis, the concept of our idea can be clearly presented, which is our expected achievement.

1.2 Structure of the document

The content of the thesis is separated into chapters that cover the basic concepts that we divided to logical units and particularly described in sections and subsections. In chapter 2 we define basic terms from the scope of formal languages that are essential to understand the presented concept. chapter 3 deals with a basic description of syntax analysis. It has been separated into two main sections, where first one describes principles of top-down parsing and tutorial how to construct one specific type of top-down parser which is an LL(1) parser. The second section describes the method of bottom-up parsing called an LR analysis. It as well provides the user with tutorial how to construct an SLR(1) parser. In both sections we describe parsing tables in detail, because they are the core of the parsers and the error recovery methods that have been used for our two parsers. The main part of the thesis is described in chapter 4. We introduce here a new programming language using 2 grammars. Every grammar specifies the part of the language it has been chosen for. After specifying the language we define the connection of parsing methods to create one parser for the language as a puzzle. In the following sections are specified details in the form of parsing tables and error recovery routines for every method as well as their connection according to the pattern of the connection of the parsing methods. The penultimate chapter 5 is dedicated to implementation. The main focus is to describe the structure of application, the data representation and we will also shortly deal with some specifics of the code. The last but not least is chapter 6 which is a conclusion. Here we discuss the achievement of the thesis and the next possible development, i.e., improvements of the method and next possible applications of the method.

Chapter 2

Basic terms and definitions

Following definitions and theorems are taken from [3].

Definition 2.0.1 (Alphabet)**.**
An alphabet is a finite nonempty set of elements, which are called symbols.

Definition 2.0.2 (String)**.**
Let Σ be an alphabet. ϵ is a string over Σ. If x is a string over Σ and $a \in \Sigma$ then xa is a string over Σ.

Definition 2.0.3 (Concatenation of Strings)**.**
Let x and y be two strings over Σ. The concatenation of x and y is xy.

Definition 2.0.4 (Reversal of String)**.**
Let x be a string over Σ. The *reversal* of x, $reversal(x)$, is defined as:

1. if $x = \epsilon$ then $reversal(\epsilon) = \epsilon$.
2. if $x = a_1...a_n$ then $reversal(a_1...a_n) = a_n...a_1$ for some $n \geq 1$, and $a_i \in \Sigma$ for all $i = 1, ..., n$.

Definition 2.0.5 (Length of String)**.**
Let x be a string over Σ. The *length* of x, $|x|$, is defined as follows:

1. if $x = \epsilon$ then $|x| = 0$.
2. if $x = a_1...a_n$ then $|x| = n$ for some $n \geq 1$, and $a_i \in \Sigma$ for all $i = 1, ..., n$.

Definition 2.0.6 (Language)**.**
Let Σ^* denote the set of all strings over Σ. Every subset $L \subseteq \Sigma^*$ is a language over Σ.

2.1 Context-free Grammar

Definition 2.1.1 (Context-free grammar)**.**
A context-free grammar (CFG) is a quadruple $G = (N, T, P, S)$, where

- N is an alphabet of nonterminals
- T is an alphabet of terminals, $N \cap T = \varnothing$
- P is a finite set of rules of the form $A \to x$, where $A \in N$, $x \in (N \cup T)^*$

- $S \in N$ is the start nonterminal

Definition 2.1.2 (Derivation Step in CFG).
Let $G = (N, T, P, S)$ be a CFG. Let $u, v \in (N \cup T)^*$ and $p = A \to x \in P$. Then, uAv directly derives uxv according to p in G, written as $uAv \Rightarrow uxv\,[p]$ or, simply, $uAv \Rightarrow uxv$.

Definition 2.1.3 (Leftmost Derivation in CFG).
Let $G = (N, T, P, S)$ be a CFG, let $u \in T^*$, $v \in (N \cup T)^*$. Let $p = A \to x \in P$ be a rule. Then, uAv directly derives uxv in the *leftmost way* according to p in G, written as $uAv \Rightarrow_{lm} uxv\,[p]$ or, simply, $uAv \Rightarrow_{lm} uxv$.

Definition 2.1.4 (Rightmost Derivation in CFG).
Let $G = (N, T, P, S)$ be a CFG, let $u \in (N \cup T)^*$, $v \in T^*$. Let $p = A \to x \in P$ be a rule. Then, uAv directly derives uxv in the *rightmost way* according to p in G, written as $uAv \Rightarrow_{rm} uxv\,[p]$ or, simply, $uAv \Rightarrow_{rm} uxv$.

Definition 2.1.5 (Sequence of Derivation Steps in CFG).
Let $G = (N, T, P, S)$ be a CFG.

- Let $u \in (N \cup T)^*$. G makes a zero-step derivation from u to u; in symbols, $u \Rightarrow^0 u\,[e]$ or, simply, $u \Rightarrow^0 u$.
- Let $u_0, ..., u_n \in (N \cup T)^*, n \geq 1$, and $u_{i-1} \Rightarrow u_i\,[p_i], p_i \in P$, for all $i = 1, ..., n$; that is $u_0 \Rightarrow u_1\,[p_1] \Rightarrow u_2\,[p_2] ... \Rightarrow u_n\,[p_n]$ Then, G makes n derivation steps from u_0 to u_n, $u_0 \Rightarrow^n u_n\,[p_1...p_n]$ or, simply, $u_0 \Rightarrow^n u_n$.
  If $u_0 \Rightarrow^n u_n\,[\pi]$ for some $n \geq 1$, then u_0 *properly derives* u_n in G, written as $u_0 \Rightarrow^+ u_n\,[\pi]$ or, simply, $u_0 \Rightarrow^+ u_n$.
  If $u_0 \Rightarrow^n u_n\,[\pi]$ for some $n \geq 0$, then u_0 *derives* u_n in G, written as $u_0 \Rightarrow^* u_n\,[\pi]$ or, simply, $u_0 \Rightarrow^* u_n$.

Definition 2.1.6 (Language generated by CFG).
Let $G = (N, T, P, S)$ be a CFG. The *language generated by G, L(G)*, is defined as $L(G) = \{w : w \in T^*, S \Rightarrow^* w\}$.

Definition 2.1.7 (Context-free Language).
Let L be a language. L is a *context-free language* (CFL) if there exists a context-free grammar that generates L.

Definition 2.1.8 (Left Recursion).
Let $G = (N, T, P, S)$ be a CFG. A rule of the form $A \to Ax$, where $A \in N, x \in (N \cup T)^*$ is called a *left recursive rule.*

Definition 2.1.9 (Grammatical Ambiguity in CFG).
Let $G = (N, T, P, S)$ be a CFG. If there exists $x \in L(G)$ with more than one derivation tree, then G is *ambiguous*; otherwise, G is *unambiguous.* A CFL, L, is *inherently ambiguous* if L is generated by no unambiguous grammar.

2.2 Pushdown Automata

Definition 2.2.1 (Pushdown Automaton).
A pushdown automaton (PDA) is a 7-tuple $M = (Q, \Sigma, \Gamma, R, s, S, F)$, where

- Q is a finite set of states
- Σ is an input alphabet
- Γ is a pushdown alphabet
- R is a finite set of rules of the form: $Apa \to wq$ where $A \in \Gamma, p, q \in Q, a \in \Sigma \cup \{\epsilon\}, w \in \Gamma^*$
- $s \in Q$ is the start state where $S \in \Gamma$ is the start pushdown symbol
- $F \subseteq Q$ is a set of final states

Definition 2.2.2 (PDA Configuration).
Let $M = (Q, \Sigma, \Gamma, R, s, S, F)$ be a PDA. A *configuration* of M is a string $\chi \in \Gamma^* Q \Sigma^*$.

Definition 2.2.3 (PDA Move).
Let $xApay$ and $xwqy$ be two configurations of a PDA, M, where $x, w \in \Gamma^*, A \in \Gamma, p, q \in Q, a \in \Sigma \cup \{\epsilon\}$, and $y \in \Sigma^*$. Let $r = Apa \to wq \in R$ be a rule. Then, M makes a *move* from $xApay$ to $xwqy$ according to r, written as $xApay \vdash xwqy\,[r]$ or, simply, $xApay \vdash xwqy$.

Definition 2.2.4 (Sequence of Moves in PDA).
Let $M = (Q, \Sigma, \Gamma, R, s, S, F)$ be a PDA.

- Let χ be a configuration. M makes a zero moves from χ to χ; in symbols, $\chi \vdash^0 \chi\,[\epsilon]$ or, simply, $\chi \vdash^0 \chi$.
- Let $\chi_0, \chi_1, ..., \chi_n$ be a sequence of configurations, $n \geq 1$, and $\chi_{i-1} \vdash \chi_i\,[r_i], r_i \in R$, for all $i = 1, ..., n$; that is $\chi_0 \vdash \chi_1\,[r_1] \vdash \chi_2\,[r_2] ... \vdash \chi_n\,[r_n]$. Then M makes n *moves* from χ_0 to $\chi_n, \chi_0 \vdash^n \chi_n\,[r_1...rn]$ or, simply, $\chi_0 \vdash^n \chi_n$.
  If $\chi_0 \vdash^n \chi_n\,[\rho]$ for some $n \geq 1$, then $\chi_0 \vdash^+ \chi_n\,[\rho]$ or, simply, $\chi_0 \vdash^+ \chi_n$.
  If $\chi_0 \vdash^n \chi_n\,[\rho]$ for some $n \geq 0$, then $\chi_0 \vdash^* \chi_n\,[\rho]$ or, simply, $\chi_0 \vdash^* \chi_n$.

Definition 2.2.5 (Language accepted by PDA).
Let $M = (Q, \Sigma, \Gamma, R, s, S, F)$ be a PDA.

1. The language that M accets by final state, denoted by $L(M)_f$, is defined as $L(M)_f = \{w : w \in \Sigma^*, Ssw \vdash^* zf, z \in \Gamma^*, f \in F\}$.
2. The language that M accets by empty pushdown, denoted by $L(M)_\epsilon$, is defined as $L(M)_\epsilon = \{w : w \in \Sigma^*, Ssw \vdash^* zf, z = \epsilon, f \in Q\}$.
3. The language that M accets by final state and empty pushdown, denoted by $L(M)_{f\epsilon}$, is defined as $L(M)_{f\epsilon} = \{w : w \in \Sigma^*, Ssw \vdash^* zf, z = \epsilon, f \in F\}$.

Theorem 2.2.1 (Equivalence of Three Types of Acceptance).
$L = L(M_f)_f$ for a PDA $M_f \Leftrightarrow L = L(M_{f\epsilon})_{f\epsilon}$ for a PDA $M_{f\epsilon}$.
$L = L(M_\epsilon)_\epsilon$ for a PDA $M_\epsilon \Leftrightarrow L = L(M_{f\epsilon})_{f\epsilon}$ for a PDA $M_{f\epsilon}$.
$L = L(M_f)_f$ for a PDA $M_f \Leftrightarrow L = L(M_\epsilon)_\epsilon$ for a PDA M_ϵ.

Definition 2.2.6 (Deterministic PDA(DPDA)).
Let $M = (Q, \Sigma, \Gamma, R, s, S, F)$ be a PDA. M is a deterministic PDA if for each rule $Apa \to wq \in R$, it holds that $R - \{Apa \to wq\}$ contains no rule with the left-hand side equal to Apa or Ap.

Theorem 2.2.2 (PDAs are Stronger than DPDAs).
There exists no DPDA $M_{f\epsilon}$ *that accepts* $L = xy : x, y \in \Sigma^*, y = reversal(x)$.

Definition 2.2.7 (Extended PDA(EPDA)).
An Extended pushdown automaton(EPDA) is a 7-tuple $M = (Q, \Sigma, \Gamma, R, s, S, F)$, where $Q, \Sigma, \Gamma, R, s, S, F$ are defined as in a PDA and R is a finite set of rules of the form $vpa \rightarrow wq$, where $v, w \in \Gamma^*, p, q \in Q, a \in \Sigma \cup \{\epsilon\}$.

Definition 2.2.8 (EPDA Move).
Let $xvpay$ and $xwqy$ be two configurations of an EPDA, M, where $x, v, w \in \Gamma^*, p, q \in Q, a \in \Sigma \cup \{\epsilon\}$, and $y \in \Sigma^*$. Let $r = vpa \rightarrow wq \in R$ be a rule. Then, M makes a *move* from $xvpay$ to $xwqy$ according to r, written as $xvpay \vdash xwqy\,[r]$ or, simply, $xvpay \vdash xwqy$.

Definition 2.2.9 (Sequence of Moves in EPDA).
Let $M = (Q, \Sigma, \Gamma, R, s, S, F)$ be a EPDA.

- Let χ be a configuration. M makes a zero moves from χ to χ; in symbols, $\chi \vdash^0 \chi\,[\epsilon]$ or, simply, $\chi \vdash^0 \chi$.

- Let $\chi_0, \chi_1, ..., \chi_n$ be a sequence of configurations, $n \geq 1$, and $\chi_{i-1} \vdash \chi_i\,[r_i], r_i \in R$, for all $i = 1, ..., n$; that is $\chi_0 \vdash \chi_1\,[r_1] \vdash \chi_2\,[r_2] ... \vdash \chi_n\,[r_n]$. Then M makes n *moves* from χ_0 to $\chi_n, \chi_0 \vdash^n \chi_n\,[r_1...rn]$ or, simply, $\chi_0 \vdash^n \chi_n$.
  If $\chi_0 \vdash^n \chi_n\,[\rho]$ for some $n \geq 1$, then $\chi_0 \vdash^+ \chi_n\,[\rho]$ or, simply, $\chi_0 \vdash^+ \chi_n$.
  If $\chi_0 \vdash^n \chi_n\,[\rho]$ for some $n \geq 0$, then $\chi_0 \vdash^* \chi_n\,[\rho]$ or, simply, $\chi_0 \vdash^* \chi_n$.

Chapter 3

Principles of Syntax Analysis

In this chapter we talk about concepts and theory basics that were mentioned [2, 1]. The algorithms and definitions were borrowed from [1, 3] and adjusted according to our notation. Some sources reference to elements of third set in grammar quadruple as *rules* and other use the more specific name *productions*. In this document we will use both terms as synonyms. In the course of *syntax analysis* or *parsing*, a compiler is trying to check if a source chain of input symbols is a string of the given language. These input symbols are called tokens and they are the result of a lexical analyser. Token is a complex structure that carries information not just about its type, but also about its data. The type of a token is important for a parser. Nevertheless, tokens are sent to semantic analyser through our parser, which uses the data for semantic control. That is the reason why the input for syntax analyser are tokens and not just the types of the input symbols.
The result of the syntax analysis is a tree-like intermediate representation that depicts the grammar structure of the token stream. Two basic principles to create a parser are *top-down* and *bottom-up* parsing. The top-down parsing is trying to create the input string by an expansion of nonterminals beginning with the starting nonterminal. Contrariwise, bottom-up parsing is trying to create the starting nonterminal from the input string. To these two approaches correspond two types of a grammar, namely, LL and LR grammars. Top-down as well as bottom-up parsing will be described in this chapter as the main methods to create the parser used in this thesis.

3.1 Top-down Parsing

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder. Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string. [1]
Top-down parser can be described as a pushdown automaton that consists of an input buffer, a stack containing a sequence of grammar symbols, a parsing table constructed by Algorithm 4 and an output stream. The pushdown automaton reads symbols from input buffer and writes the numbers of applied productions to the output stream. The output sequence of the productions is called a left parse. Left parse is a sequence of rules used in the leftmost derivation of the input string [3]. Initially, configuration of the PDA (see 2.2.2) is the string

$$\epsilon S\#w$$

where # is special symbol representing a bottom of the stack and w is the input string. When the input string w is accepted, the automaton is in a final configuration

$$\pi\$\epsilon$$

where π is the left parse and \$ represents the end of the input string. Two most commonly used methods to implement the top-down parser are recursive descent and a nonrecursive table driven predictive parser.
In this section will be described one of the methods for constructing a parsing table. This method uses the set Empty as well as the sets Fist and Follow to create the set Predict, which is directly used to create the parsing table. Subsequently, the algorithm for a nonrecursive predictive parser will be mentioned. Finally, the basics for panic-mode error recovery used in our top-down parser will be described.

3.1.1 Set Empty

Definition 3.1.1 (Set Empty)**.**
Let $G = (N, T, P, S)$ be a CFG. $Empty(x) = \{\epsilon\}$ if $x \Rightarrow^* \epsilon$; otherwise, $Empty(x) = \varnothing$, where $x \in (N \cup T)^*$.

Algorithm 1 Set Empty

```
Input: $G = (N, T, P, S)$
Output: $Empty(X)$ for every $X \in N \cup T$
Method:
 1: for all $a \in T$ do
 2:     $Empty(a) := \varnothing$
 3: end for
 4: for all $A \in N$ do
 5:     if $A \to \epsilon \in P$ then
 6:         $Empty(a) := \{\epsilon\}$
 7:     else
 8:         $Empty(a) := \varnothing$
 9:     end if
10: end for
11: repeat
12:     if $A \to X_1X_2...X_n \in P$ and $Empty(X_i) = \{\epsilon\}$ for all $i = 1, ..., n$ then
13:         $Empty(a) := \{\epsilon\}$
14:     end if
15: until No Empty set can be changed
```

3.1.2 Set First

Definition 3.1.2 (Set First)**.**
Let $G = (N, T, P, S)$ be a CFG. For every $x \in (N \cup T)^*$, we define the set $First(x)$ as $First(x) = \{a : a \in T, x \Rightarrow^* ay; y \in (N \cup T)^*\}$.

Algorithm 2 Set First

```
Input: $G = (N, T, P, S)$
Output: $First(X)$ for every $X \in N \cup T$
Method:
1: for all $a \in T$ do
2:     $First(a) := \{a\}$
3: end for
4: for all $A \in N$ do
5:     $First(A) := \varnothing$
6: end for
7: repeat
8:     if $A \to X_1X_2...X_{k-1}X_k...X_n \in P$ then
9:         add all symbols from $First(X_1)$ to $First(A)$
10:        if $Empty(X_i) = \{\epsilon\}$ for all $i = 1, ..., k-1$, where $k \geq n$ then
11:            add all symbols from $First(X_k)$ to $First(A)$
12:        end if
13:    end if
14: until No First set can be changed
```

3.1.3 Set Follow

Definition 3.1.3 (Set Follow)**.**
Let $G = (N, T, P, S)$ be a CFG. For every $A \in N$, we define the set $Follow(A)$ as $Follow(A) = \{a : a \in T, S \Rightarrow^* xAay; x, y \in (N \cup T)^*\} \cup \{\$: S \Rightarrow^* xA, x \in (N \cup T)^*\}$.

Algorithm 3 Set Follow

```
Input: $G = (N, T, P, S)$
Output: $Follow(A)$ for every $A \in N$
Method:
1: $Follow(S) := \$$
2: repeat
3:     if $A \to xBy \in P$ then
4:         if $y \neq \epsilon$ then
5:             add all symbols from $First(y)$ to $Follow(B)$
6:         end if
7:         if $Empty(y) = \{\epsilon\}$ then
8:             add all symbols from $Follow(A)$ to $Follow(B)$
9:         end if
10:    end if
11: until No Follow set can be changed
```

3.1.4 Set Predict

Definition 3.1.4 (Set Predict)**.**
Let $G = (N, T, P, S)$ be a CFG. For every $A \to x \in P$, we define $Predict(A \to x)$ so that

- if $Empty(x) = \{\epsilon\}$ then $Predict(A \to x) = First(x) \cup Follow(A)$

- if $Empty(x) = \varnothing$ then $Predict(A \to x) = First(x)$

3.1.5 Parsing Table

In the following definitions and examples we consider the pushdown automaton $C = (Q, \Sigma, \Gamma, R, s, S, F)$, the grammar $G = (N, T, P, S)$, the symbol # that represents the bottom of the stack and the symbol \$ that stands for the end of the input string. Parsing table is a function $M : (\Sigma \cup N \cup \{\#\}) \times (\Sigma \cup \{\$\}) \to \{$**expand** 1,**expand** 2,…,**expand** n, **pop**, **accept**, **error**$\}$. The meaning of the actions in the table is as follows:

- **expand** i Let $p_i : A \to \alpha$ be a rule of the grammar. On top of the stack is nonterminal A and input symbol is a. If $M[A, a] =$ **expand** i automat will do the move
$$\pi A\beta ax \vdash \pi i\alpha\beta ax$$
i.e., top nonterminal A is replaced by α and i is written to the output stream.

- **pop** The same terminal symbol is on top of the stack and in the input. The automaton will do the move
$$\pi a\beta ax \vdash \pi\beta x$$
i.e., the symbol a is removed from top of the stack as well as from the input.

- **accept** The Automaton is in its final configuration, input string has been accepted and complete left parse of the input string is in the output.

- **error** The input string w is not an element of the language L accepted by the grammar G.
$$w \notin L$$

Algorithm 4 Construction of a parsing table

Input: $G = (N, T, P, S)$
Output: Parsing table M
Method:

```
1: for all rules $p_i \in P$ of the form $A \to \alpha$ do
2:     for all $a \in Predict(A \to \alpha)$ do
3:         add expand i to $M[A, a]$
4:     end for
5: end for
6: for all $x \in T$ do
7:     add pop to $M[x, x]$
8: end for
9: $M[\#, \$]$ :=accept
10: for all blank entries in $M$ do
11:     set to an error
12: end for
```

3.1.6 LL Grammar

Algorithm 4 is applicable for every grammar G. However, for some grammars the parsing table M will include more than one action in some entries. If a grammar has a left recursive rule or it is ambiguous, the table M will have at least one multiply defined entry. If every entry uniquely identifies an action, the grammar is called an LL(1) grammar. The first letter „L“ means that input is read from left to right. The second „L“ stands for a left parse that is the result of analysis and „1“ is the number of input symbols that must be known to make decision about the next step.

Definition 3.1.5 (LL(1) grammar).
Let $G = (N, T, P, S)$ be a CFG. G is an *LL grammar* if for every $a \in T$ and every $A \in N$ there is no more than one A-rule $A \to X_1X_2...X_n \in P$ such that $a \in \mathit{Predict}(A \to X_1X_2...X_n)$.

3.1.7 Recursive Descent Parser

One of the well known methods how to implement a top-down parser is the recursive descent. Every nonterminal of the grammar has its own procedure that implements its analyse. When we have the nonterminal A, which has only one production $A \to X_1X_2...X_n$, the body of the procedure for the nonterminal will be a sequence of calls for every X_i with i from 1 to n. If X_i is a nonterminal, corresponding procedure will be called, otherwise algorithm will check if input symbol corresponds to X_i. We initiate the analysis by calling the procedure of the starting nonterminal.

3.1.8 Table-driven Parser

A nonrecursive predictive parser is built with implemented stack, not via recursive calls. The structure of the parser is depict on the figure 3.1. The symbol \$ is an endmarker for the input string and can be used as well to mark the bottom of the stack instead of the symbol #. The parser looks for a production in the table only when a nonterminal is on top of the stack. Solution for terminals is provided in algorithm 5 without using a parsing table. Therefore we can create the parsing table M using the algorithm 4 without code on lines 6, 7, 8 and 9.

3.1.9 Panic-Mode Error Recovery

The main idea of the Panic-mode error recovery is to skip symbols on the input until a token in a selected set of synchronizing tokens appears. Effectiveness of this method depends on the choice of the synchronizing set. The sets are chosen with respect to the grammar, so that in practice, the parser can quickly recover from common errors. Some heuristics are as follows:

1. At first, we place all symbols in *Follow(A)* into the synchronizing set for nonterminal A.

2. Programming languages have often hierarchical structure, e.g., expressions appear within statements, which appear within blocks, and so on. Accordingly we can add the symbols that begin higher-level constructs to the synchronizing set of a lower-level constructs.

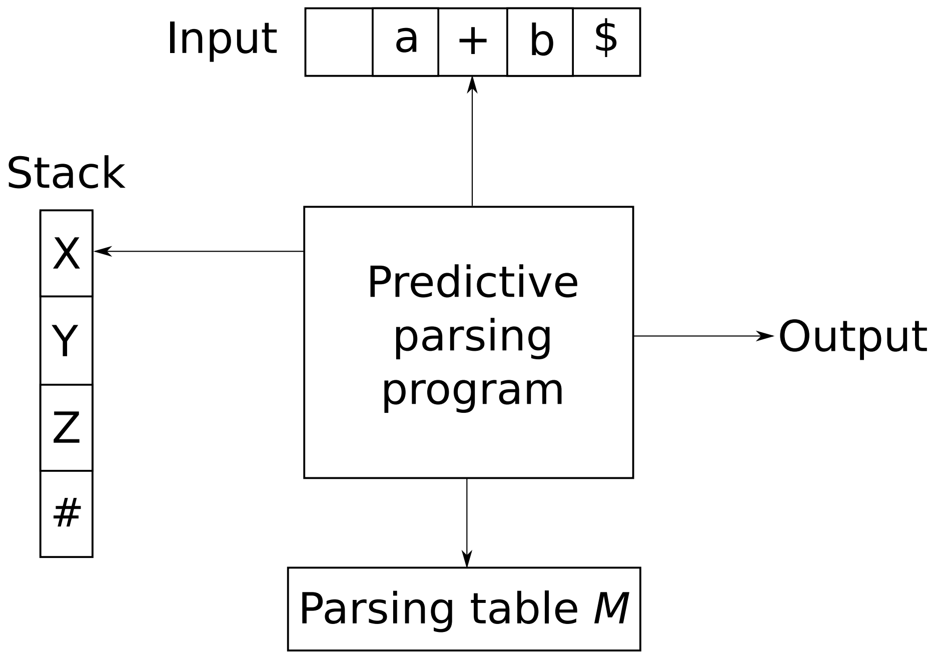


Figure 3.1: Model of table-driven predictive parser [1]

Algorithm 5 Table-driven predictive parser

Input: Parsing table M for grammar $G = (N, T, P, S)$ and string $w\$$, $w \in T^*$
Output: Left parse if $w \in L(G)$; otherwise, an error
Method:

```
push $\$$ on the stack
push $S$ on the stack
set $a$ to the current token
set $X$ to the top stack symbol
while stack is not empty do
    if $X = \$$ then
        if $a = \$$ then success()
        else error()
        end if
    else if $X = a$ then
        pop the stack and $a :=$ next token
    else if $X$ is terminal then error()
    else if $M[X, a]$ is an error entry then error()
    else if $r : X \to x \in M[X, a]$ then
        write production $r$ to the output
        replace $X$ with reversal(x) on the stack
    end if
    set $X$ to the top stack symbol
end while
```

3. If we add symbols from the *First(A)* to the synchronizing set for nonterminal A, we can possibly skip the additional or wrong input symbols and resume according to A if a symbol in *First(A)* appears in the input.

4. We can postpone error detection, but never miss it, by using production $A \to \epsilon$ for nonterminal A if possible. This approach will reduce the number of nonterminals that have to be considered during error recovery.

5. If terminal on top of the stack does not match the input symbol, we can pop it and issue a message saying that the terminal was inserted.

3.2 Bottom-up Parsing

A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top). [1]
We can imagine bottom-up parsing also as the process of „reducing“ an input string to the start symbol of the grammar. At the reduction step, a specific substring matching the right side of a production is replaced by the nonterminal on the left side of the production. Comparing to top-down parsing, a reduction is the reverse of a step in the derivation. The reverted sequence of productions that is the result of bottom-up parsing is called rightmost derivation (see 2.1.4). In this section we will shortly introduce a general style of bottom-up parsing called shift-reduce parsing. The largest class of grammars for which shift-reduce parsers can be built are LR grammars. The LR grammars and also basic algorithms to build an LR parser will be introduced here as well.

3.2.1 Shift-reduce Parsing

The Shift-reduce means that the two basic operations are used for the parsing method - shift and reduction. In this method a stack is used to hold the grammar symbols. We use \$ symbol to mark the bottom of the stack as well as the end of the input string. As a starting point the stack and the input are in the following configuration

$$Stack : \$ \qquad Input : w\$$$

When the string w is accepted by the parser, the configuration is

$$Stack : \$S \qquad Input : \$$$

where S is the starting symbol of the grammar. In this method that uses a stack, the reduction is made only on top of the stack and it never goes deeper, which is also a practical reason to use the stack. Four possible actions of the parser are as follows:

- **Shift** The next input symbol is pushed on the stack.
- **Reduce** On top of the stack must be a string to reduce. The left symbol of the string is found within the stack and the analyser will try to decide about the nonterminal that will replace it.
- **Accept** The string was accepted by the parser.
- **Error** The analyser has detected a syntax error.

There exist context-free grammars, e.g. ambiguous grammars, for which shift-reduce parsing cannot be used. During the syntax analysis of such grammar $shift/reduce$ or *reduce/reduce* conflicts could appear. In $shift/reduce$ conflict the parser cannot decide whether make $shift$ or *reduction*. When more productions have the same right side, the conflict *reduce/reduce* may appear, because the parser is unable to choose between reductions. These grammars are not LR grammars.

3.2.2 LR Parsing

Following sections will describe the LR(k) syntax analysis, where $k < 1$, that is the most prevalent method of bottom-up parsing. The „L" means that the input string is read from left to right. The „R" represents the rightmost derivation, which is the result of the parsing method and (k) is the number of input symbols used to make a decision about the next step. Sometimes (k) is not used, in that case we assume k equals 1. An equivalent model of LR parser is an extended pushdown automaton (see 2.2.7). A few advantages of LR parsing are as follows:

- We can use LR parsing for almost all language constructions of programming languages defined by context-free grammars.
- The method is nonbacktracking. It means that we can write an effective parser for it.
- The parser detects errors right after they occur.

The drawback is that the manual construction of an LR parser for commonly used programming languages is difficult. Ordinarily we use for that purpose a specialized tool called an LR parser generator.

3.2.3 Construction of LR table

The LR parsing is table-driven method as well as nonrecursive predictive parsing. In this section we describe a construction of Simple LR(SLR) table and the general algorithm for LR parser. An SLR method is the least powerful considering the number of grammars that we can analyse by this method. Nevertheless, for the purpose of this thesis is the method sufficient. The LR parser maintains the states that represent sets of items. Item is a production with the special symbol • that separates part of the production body(right side of a production) that has been analysed and part of the production body which is expected on the input.

Definition 3.2.1 (Item).
Let $G = (N, T, P, S)$ be a CFG, $A \to x \in P, x = yz$. Then, $A \to y \bullet z$ is an item.

For the production $A \to xy$ we have 3 items:

$$A \to \bullet xy$$
$$A \to x \bullet y$$
$$A \to xy \bullet$$

The production $A \to \epsilon$ generates only one item $A \to \bullet$. The items, where no terminal symbol is specified are called LR(0) items. As mentioned before, the states represent sets of LR(0) items. These sets are called *canonical LR(0) collection*. We use canonical LR(0) collection to create deterministic LR(0) automaton that makes parsing decisions. In the automaton every state represents one set of the canonical LR(0) collection. For the purpose of constructing the canonical LR(0) collection we have to define an *extended grammar* as well as the functions *Closure* and *Goto*.

Definition 3.2.2 (Extended grammar)**.**
Let $G = (N, T, P, S)$ be a CFG, $S' \notin N$. Extended grammar for G is grammar $G' = (N \cup \{S'\}, T, P \cup \{S' \rightarrow S\}, S')$.

The dummy production $S' \rightarrow S$ that is part of extended grammar functioning as an indicator to the parser to stop analysis. The input string is accepted only when parser is in the state with item $S' \rightarrow S\bullet$, intending to reduce by this production.

Algorithm 6 Closure

```
Input: $G = (N, T, P, S)$; Set of items $I$
Output: the set $Closure(I)$
Method:
1: $Closure(I) := I$
2: repeat
3:     for all $A \rightarrow y\bullet Bz \in Closure(I)$ and $B \rightarrow x \in P$ do
4:         if $B \rightarrow \bullet x \notin Closure(I)$ then
5:             add $B \rightarrow \bullet x$ to $Closure(I)$
6:         end if
7:     end for
8: until no more items are added to $Closure(I)$ on one round
```

The common trait for the items in one set is the same position in the production body. The main idea of the function *Closure* specified by algorithm 6 is that position marked by $\bullet$ in front of a nonterminal is the same as position on the start of the production body of the nonterminal. The task of the function *Goto* is to simulate the acceptance of an expected input symbol or a nonterminal and create a new set of items according to the new position. *Goto* function also specifies transitions between sets of items. The states and the transitions together create the LR(0) automaton. The start state is represented by the set $Closure(\{S' \rightarrow \bullet S\})$. All states are accepting states.

Definition 3.2.3 (Goto)**.**
Let $G = (N, T, P, S)$ be a CFG, I be a set of items, and $X \in T \cup N$. Then, $Goto(I, X) = Closure(\{p : p = A \rightarrow yX\bullet z, A \rightarrow y\bullet Xz \in I\})$.

The task of the function *Goto*

We specified everything what was needed to introduce an algorithm for creation of SLR table. For naming the states we use the following convention. The state i is created from the set I_i. The state with the item $S' \rightarrow \bullet S$ has the number 0. Other states have numbers from 1 to n, where $n + 1$ is the number of sets in canonical LR(0) collection.

3.2.4 LR Parser

The algorithm 9 specified below can be used also for other LR parsers than just a SLR parser. Only difference is in the parsing tables. The LR parser model is depicted in the figure 3.2.

Algorithm 7 Canonical LR(0) collection

Input: Extended grammar $G' = (N, T, P, S')$
Output: Canonical LR(0) collection C for grammar G'
Method:

```
1: $C := \{Closure(\{S' \to \bullet S\})\}$
2: repeat
3:     for all $I \in C$ and $X \in N \cup T$ do
4:         if $Goto(I, X) \neq \varnothing$ and $Goto(I, X) \notin C$ then
5:             add $Goto(I, X)$ to $C$
6:         end if
7:     end for
8: until no new sets of items are added to $C$ on a round
```

Algorithm 8 SLR(1) table

Input: Extended grammar $G' = (N, T, P, S')$
Output: SLR(1) table for grammar G
Method:

```
1: create canonical LR(0) collection $C$ for $G$
2: for all sets $I$ in $C$ do
3:     for all productions $p$ in $I_i$ do
4:         if $p = A \to \alpha \bullet a \beta$, $a \in T$ and $Goto(I_i, a) = I_j$ then
5:             $action[i, a] := shift\, j$
6:         end if
7:         if $p = A \to \alpha \bullet, A \neq S'$ then
8:             for all $a \in Follow(A)$ do
9:                 $action[i, a] := reduce\, A \to \alpha$
10:            end for
11:        end if
12:        if $p = S' \to S \bullet$ then
13:            $action[i, \$] := accept$
14:        end if
15:        if $p = A \to \alpha \bullet B \beta$, $B \in N$ and $Goto(I_i, B) = I_j$ then
16:            $goto[i, a] := j$
17:        end if
18:    end for
19: end for
20: for all undefined entries in SLR table do
21:     assign an error to the entry
22: end for
```

Algorithm 9 LR parser

Input: An input string $w\$$ and an LR-parsing table with functions *action* and *goto* for a grammar G

Output: If w is in $L(G)$, the reduction steps of a bottom-up parse for w; otherwise, an error indication

Method:

```
 1: set $a$ to the current token
 2: push $ and state 0 on the stack, respectively
 3: repeat
 4:     let $s$ be the the state on top of the stack
 5:     if $action[s, a] = shift\, i$ then
 6:         push at first $a$ and then state $i$ onto the stack
 7:         set $a$ to the next token
 8:     else if $action[s, a] = reduce\, A \to \beta$ then
 9:         pop $2 * |\beta|$ symbols off the stack
10:         let $s'$ be the state on top of the stack
11:         push $a$ and the state $goto[s', A]$ onto the stack, respectively
12:         output the production $A \to \beta$
13:     else if $action[s, a] = accept$ then
14:         return
15:     else
16:         call error recovery routine
17:     end if
18: until 0
```

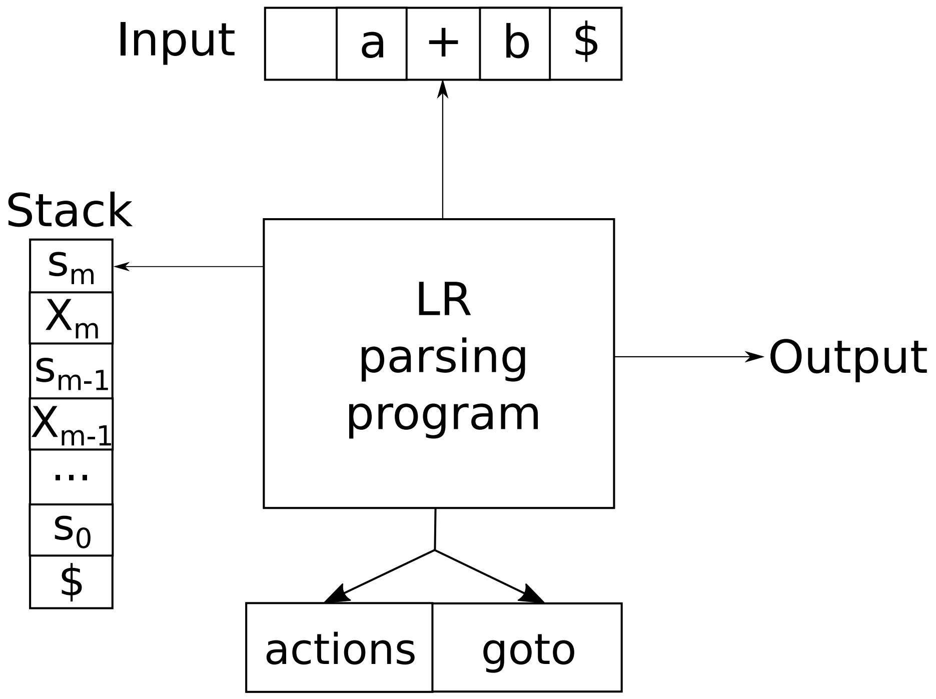


Figure 3.2: Model of an LR parser [1]

3.2.5 Using ambiguous grammars

An ambiguous grammar cannot be the LR grammar, but in some cases it is a faster and easier way how to describe specific constructs of a language. Ambiguity can be resolved by specifying disambiguating rules. A typical example where ambiguous grammar is used are expressions, where we usually use operators with different priority and associativity. Let us have a grammar G with rules

$$E \to E + E \quad E \to E * E \quad E \to (E) \quad E \to i$$

In the grammar specified this way, the position $E \to E + E\bullet$ and the position $E \to E\bullet * E$ are the same. In other words, when on top of the stack is the string $E + E$ and the input symbol is $*$, the grammar does not specify if we should reduce $E + E$ or shift $*$ onto the stack. Therefore, the conflict in corresponding LR parsing table appears. To solve this conflict we can adjust the grammar as follows

$$E \to E + T|T \quad T \to T * F|F \quad F \to (E) \quad F \to i$$

Additional nonterminals will solve the problem of priority, but the grammar has more rules and some additional reductions have to be performed by an LR parser, because of the rules $E \to T$ and $T \to F$. Another more appropriate option would be to resolve conflicts in the LR table by specifying priority and associativity. In our case $*$ has a higher priority, therefore the correct option in the previously mentioned table entry will be to shift $*$ onto the stack. The result of this method is the correct syntax analysis without any changes in the grammar and any additional reductions.

3.2.6 Error Recovery in LR Parsing

For LR Parsing also exists the Panic-mode error recovery, but a much more elaborate and suitable method for our parser will be the Phrase-level recovery. To use this method, we need to examine all error entries in the LR parsing table and to create and choose the appropriate routines to recover from the specific error. It is designed according to the usage of the programming language. Important facts about an LR analysis are that error will never be detected in Goto part of the LR table and an erroneous input symbol is never pushed onto the stack before an error is detected. The created routines can do insertions or deletions in order to recover, but the parser cannot get into an infinite loop. It is not recommended to pop the stack, because the construct, which has been already successfully parsed, is eliminated this way.

Chapter 4

Specification of the new parsing method

For a demonstration purpose has been created the language according to the pattern of PHP. From the set of statements of the PHP programming language have been chosen constructions that will serve as an example to show how our new parsing method works. The language has been separated into two sections.
For the first section we wrote an ambiguous context-free grammar and constructed a top-down LL(1) parser. This section encompasses the syntax of the statements and describes a backbone construction of the language. We chose nonrecursive table-driven version of the parser because of two reasons. If we made changes in the grammar, recursive descent parser would require us to reimplement the procedures for the nonterminals that have been changed. Contrary to recursive descent in table-driven parsing we only need to change the parsing table considering exclusively a syntax analysis. Also we use an error recovery method that would need to implement additional data structures in recursive descent to keep the information about the synchronizing set. In our method this information is already stored on the stack.
For the second section we constructed an ambiguous context-free grammar and a bottom-up LR parser. The second section processes the expressions which are part of the majority of the statements. We decided to use an LR grammar for the expressions, because ambiguity can be easily solved and we will avoid procedures such as the left recursion replacement or additional derivation steps during the syntax analysis.

4.1 Grammars

The top-down section is described by the following grammar $G = (N, T, P, \langle begin \rangle)$, where:

$N = \{\langle begin \rangle, \langle st\_list \rangle, \langle st\_list2 \rangle, \langle st\_list3 \rangle, \langle case\_def \rangle, \langle par\_list \rangle, \langle par\_list2 \rangle, \langle expr1 \rangle, \langle expr2 \rangle, \langle cexpr \rangle\}$.
$T = \{<?php\backslash s, \{, \}, (,), ;, :, ,, var, id, function, while, for, if, elseif, else, return, switch, case, default, break, continue, EOF\}$.

The set of rules P looks as follows:

$$\langle begin \rangle \quad \rightarrow \quad <?php\backslash s\, \langle st\_list \rangle \tag{4.1}$$

$$
\begin{aligned}
\langle st\_list\rangle &\rightarrow \langle st\_list2\rangle\langle st\_list\rangle && (4.2)\\
\langle st\_list\rangle &\rightarrow function\, id\,(\,\langle par\_list\rangle\,)\,\{\,\langle st\_list2\rangle\,\}\,\langle st\_list\rangle && (4.3)\\
\langle st\_list\rangle &\rightarrow EOF && (4.4)\\
\langle st\_list2\rangle &\rightarrow while\,(\,\langle expr1\rangle\,\{\,\langle st\_list2\rangle\,\}\,\langle st\_list2\rangle && (4.5)\\
\langle st\_list2\rangle &\rightarrow for\,(\,var = \langle expr2\rangle\,\langle expr2\rangle\, var = \langle expr1\rangle\,\{\,\langle st\_list2\rangle\,\}\,\langle st\_list2\rangle && (4.6)\\
\langle st\_list2\rangle &\rightarrow if\,(\,\langle expr1\rangle\,\{\,\langle st\_list2\rangle\,\}\,\langle st\_list3\rangle\langle st\_list2\rangle && (4.7)\\
\langle st\_list2\rangle &\rightarrow return\,\langle expr2\rangle\langle st\_list2\rangle && (4.8)\\
\langle st\_list2\rangle &\rightarrow switch\,(\,\langle expr1\rangle\,\{\,\langle case\_def\rangle\,\}\,\langle st\_list2\rangle && (4.9)\\
\langle st\_list2\rangle &\rightarrow var = \langle exp2\rangle\langle st\_list2\rangle && (4.10)\\
\langle st\_list2\rangle &\rightarrow break\,;\,\langle st\_list2\rangle && (4.11)\\
\langle st\_list2\rangle &\rightarrow continue\,;\,\langle st\_list2\rangle && (4.12)\\
\langle st\_list2\rangle &\rightarrow \epsilon && (4.13)\\
\langle st\_list3\rangle &\rightarrow elseif\,(\,\langle expr1\rangle\,\{\,\langle st\_list2\rangle\,\}\,\langle st\_list3\rangle && (4.14)\\
\langle st\_list3\rangle &\rightarrow else\,\{\,\langle st\_list2\rangle\,\} && (4.15)\\
\langle st\_list3\rangle &\rightarrow \epsilon && (4.16)\\
\langle case\_def\rangle &\rightarrow case\,\langle cexpr\rangle\langle st\_list2\rangle\langle case\_def\rangle && (4.17)\\
\langle case\_def\rangle &\rightarrow default\ :\ \langle st\_list2\rangle\langle case\_def\rangle && (4.18)\\
\langle case\_def\rangle &\rightarrow \epsilon && (4.19)\\
\langle par\_list\rangle &\rightarrow var\,\langle par\_list2\rangle && (4.20)\\
\langle par\_list\rangle &\rightarrow \epsilon && (4.21)\\
\langle par\_list2\rangle &\rightarrow ,\ var\,\langle par\_list2\rangle && (4.22)\\
\langle par\_list2\rangle &\rightarrow \epsilon && (4.23)
\end{aligned}
$$

The statements *break* and *continue* can be used only within *for* or *switch* statement. This is not specified in the grammar, but it will be controlled semantically in the code generator part.
The grammar defined like this is not an LL grammar, but we need to create the LL parsing table for the LL parser. Ambiguity is caused by the following two rules:

$$
\begin{aligned}
&(1) & \langle st\_list\rangle &\rightarrow \langle st\_list2\rangle\langle st\_list\rangle\\
&(2) & \langle st\_list2\rangle &\rightarrow \epsilon
\end{aligned}
$$

These rules are created to simplify the grammar and reduce the cardinality of P. A function cannot be defined inside another statement, therefore we must create two nonterminals as $\langle st\_list\rangle$ and $\langle st\_list2\rangle$, where $\langle st\_list2\rangle$ represents the subset of the statements that can be defined inside another statement. Accordingly the first rule represents a substitution for the subset of rules that $\langle st\_list2\rangle$ stands for. We want to make a derivation according to the first rule just when the input symbol is in $First(\langle st\_list2\rangle)$, because that is the purpose of the rule. Due to ϵ-rule for nonterminal $\langle st\_list2\rangle$ to the set $Predict$ of first rule got also the $First(\langle st\_list\rangle)$. The simple solution is that we will consider just the nonterminals in $First(\langle st\_list2\rangle)$. Consequently the nonterminals $function$ and EOF will be excluded from the set $Predict$ of the first rule.
The purpose of the second rule is to finish the block of statements inside another state-

ment or to switch back to $\langle st\_list \rangle$ if the input is $function$ or EOF. According to definition 3.1.4 $Predict(\langle st\_list2 \rangle \rightarrow \epsilon)$ is $Follow(\langle st\_list2 \rangle)$, what is considering the first rule $First(\langle st\_list \rangle)$. For this reason in $Predict(\langle st\_list2 \rangle \rightarrow \epsilon)$ are the symbols from $First(\langle st\_list2 \rangle)$ and they need to be removed. Therefore we will exclude all terminals, which are in the set $First$ of $\langle st\_list2 \rangle$ from $Predict(\langle st\_list2 \rangle \rightarrow \epsilon)$. The adjustment of the mentioned sets will eliminate conflicts in the LL table and it will lead to the correct syntax analysis.

Three of the nonterminals from the previous grammar do not occur at the left side of a rule. These nonterminals $\langle expr1 \rangle$, $\langle expr2 \rangle$ and $\langle cexpr \rangle$ represent the expressions in the statements and are differentiated according to the end-of-expression symbol. Whenever these nonterminals have to be processed, the LL parser is switched to the LR parser for the expressions. The grammar for the expressions looks as follows:

$$\begin{array}{lcl}
E & \rightarrow & (\,E\,) \\
E & \rightarrow & id\,(\,) \\
E & \rightarrow & E\,,\,E \\
E & \rightarrow & E\;op5\;E \\
E & \rightarrow & E\;op3\;E \\
E & \rightarrow & E\;or\;E \\
E & \rightarrow & E\;op2\;E \\
E & \rightarrow & E\,||\,E \\
E & \rightarrow & E\,**\,E \\
E & \rightarrow & E\,\&\&\,E \\
E & \rightarrow & E\;op4\;E \\
E & \rightarrow & id\,(\,E\,) \\
E & \rightarrow & E\;and\;E \\
E & \rightarrow & op1\;E \\
E & \rightarrow & i
\end{array}$$

This grammar is ambiguous, because it does not describe a priority and an associativity of the operators. On the other side it is a fast solution, because we got rid of additional reductions of the nonterminals representing operator precedence. In the corresponding LR table will occur shift/reduction conflicts that will be solved by the specification of priority and associativity of the operators (see subsection 3.2.5).

4.2 Connection of the LL and LR Parser

As mentioned in the previous section the LL parser creates a main part of syntax, therefore the LR parser returns control to the LL parser after finishing its part of the syntax analysis. The language has three different input characters serving as an end-of-expression symbol, accordingly, three nonterminals represent these expressions. The nonterminals $\langle expr2 \rangle$ and $\langle cexpr \rangle$ are terminated by the symbols ; and :, which cannot be part of the expressions. In that case we can simply substitute for the general end-of-input character these two symbols according to the type of the expression. Another situation is when expression represented by the nonterminal $\langle expr1 \rangle$ must be terminated by symbol). We use the method that considers this symbol to be a right parenthesis except the state when the end-of-expression is expected to successfully finish the analysis. This method allows us to use the same LR table for all types of the expressions. Regarding a left parenthesis we slightly adjust the table by replacing the error state with the success when is expected the end-of-input to successfully terminate the analysis. With respect to the error detection, it is possible that error messages may be different, but a given error is still detected in both cases. If the input looks like this:

$$\$foo = 42-;$$
$$if(42-)\{\}$$

Both cases produce the error message „missing operand“. But if the input is this:

$$\$foo = boo;$$
$$if(boo)\{\}$$

The error messages differs, because a right parenthesis has another meaning in this context. For the first line the error message is „A variable must be preceded with \$ and a function must be followed by ()“. For the second line we get two error messages „Missing left parenthesis“ and „Unexpected token in the expression E_LABRACK“. When a right parenthesis follows a function identifier, the algorithm assumes that a left parenthesis is omitted and the error recovery approach is to insert it as a missing part of a function call. As soon as a function call expression is complete, the next token is a left bracket which leads the analysis to an error.

4.3 LR Table

The plan was to use a Lookahead LR and a Simple LR analysis, create the tables accordingly and give the user a chance to choose which one to use. However for the given LR grammar both tables were the same. This text does not describe the method how to construct the LALR table, but the user can use for the grammar generator Yacc to verify our statement. For that reason the following table 4.1 represents the core of both the SLR and the LALR analysis. The table is adjusted to the error recovery that will be described in the following section. The operators that belong to the group with the same priority and associativity have been replaced with the number of the given group. This substitution has been used to create the table with a reduced number of the states. The conflicts in the table have been already eliminated using the method described in section 4.1.

4.4 LL Table

In the undermentioned LL table 4.2 are already applied the heuristics of the error recovery strategy called Panic-Mode Recovery and the ambiguity is solved as mentioned in section 4.1. Two error states in the table are named *err* and *pop*.

4.5 Error Recovery

This section covers the description of the error recovery applied in LR and LL parsing as well as the description of how these two approaches have been connected to produce a

Table 4.1: LR table

	Actions																	Goto
State	1	3	2	5	4	0	7	6	9	8	i	f	,	(	)	;	:	E
0	s1	e2	e2	e2	e2	e2	e2	e2	e2	e2	s2	s27	e4	s24	e3	e2	e2	32
1	s1	e2	e2	e2	e2	e2	e2	e2	e2	e2	s2	s27	e4	s24	e3	e2	e2	3
2	r14	r14	r14	r14	r14	r14	r14	r14	r14	r14	r14	r14	r14	r14	r14	r14	r14	
3	r13	r13	r13	r13	r13	s6	r13	r13	r13	r13	r13	r13	r13	r13	r13	r13	r13	
4	s1	e2	e2	e2	e2	e2	e2	e2	e2	e2	s2	s27	e2	s24	e3	e2	e2	5
5	r9	s8	s10	s12	s14	s6	r9	r9	r9	r9	r9	r9	r9	r9	r9	r9	r9	
6	s1	e2	e2	e2	e2	e2	e2	e2	e2	e2	s2	s27	e2	s24	e3	e2	e2	7
7	r8	r8	r8	r8	r8	s6	r8	r8	r8	r8	r8	r8	r8	r8	r8	r8	r8	
8	s1	e2	e2	e2	e2	e2	e2	e2	e2	e2	s2	s27	e2	s24	e3	e2	e2	9
9	r4	r4	s10	r4	r4	s6	r4	r4	r4	r4	r4	r4	r4	r4	r4	r4	r4	
10	s1	e2	e2	e2	e2	e2	e2	e2	e2	e2	s2	s27	e2	s24	e3	e2	e2	11
11	r6	r6	r6	r6	r6	s6	r6	r6	r6	r6	r6	r6	r6	r6	r6	r6	r6	
12	s1	e2	e2	e2	e2	e2	e2	e2	e2	e2	s2	s27	e2	s24	e3	e2	e2	13
13	r3	s8	s10	r3	s14	s6	r3	r3	r3	r3	r3	r3	r3	r3	r3	r3	r3	
14	s1	e2	e2	e2	e2	e2	e2	e2	e2	e2	s2	s27	e2	s24	e3	e2	e2	15
15	r10	s8	s10	r10	r10	s6	r10	r10	r10	r10	r10	r10	r10	r10	r10	r10	r10	
16	s1	e2	e2	e2	e2	e2	e2	e2	e2	e2	s2	s27	e2	s24	e3	e2	e2	17
17	r7	s8	s10	s12	s14	s6	r7	s4	r7	r7	r7	r7	r7	r7	r7	r7	r7	
18	s1	e2	e2	e2	e2	e2	e2	e2	e2	e2	s2	s27	e2	s24	e3	e2	e2	19
19	r12	s8	s10	s12	s14	s6	s16	s4	r12	r12	r12	r12	r12	r12	r12	r12	r12	
20	s1	e2	e2	e2	e2	e2	e2	e2	e2	e2	s2	s27	e2	s24	e3	e2	e2	21
21	r5	s8	s10	s12	s14	s6	s16	s4	r5	s18	r5	r5	r5	r5	r5	r5	r5	
22	s1	e2	e2	e2	e2	e2	e2	e2	e2	e2	s2	s27	e9	s24	e9	e2	e2	23
23	e5	s8	s10	s12	s14	s6	s16	s4	s20	s18	e5	e5	r2	e5	r2	r2	r2	
24	s1	e2	e2	e2	e2	e2	e2	e2	e2	e2	s2	s27	e4	s24	e3	e2	e2	25
25	e5	s8	s10	s12	s14	s6	s16	s4	s20	s18	e5	e5	s22	e5	s26	e6	e6	
26	r0	r0	r0	r0	r0	r0	r0	r0	r0	r0	r0	r0	r0	r0	r0	r0	r0	
27	e8	e8	e8	e8	e8	e8	e8	e8	e8	e8	e8	e8	e4	s28	e7	e8	e8	
28	s1	e2	e2	e2	e2	e2	e2	e2	e2	e2	s2	s27	e9	s24	s31	e2	e2	29
29	e5	s8	s10	s12	s14	s6	s16	s4	s20	s18	e5	e5	s22	e5	s30	e6	e6	
30	r11	r11	r11	r11	r11	r11	r11	r11	r11	r11	r11	r11	r11	r11	r11	r11	r11	
31	r1	r1	r1	r1	r1	r1	r1	r1	r1	r1	r1	r1	r1	r1	r1	r1	r1	
32	e5	s8	s10	s12	s14	s6	s16	s4	s20	s18	e5	e5	s22	e5	e3	acc	acc	

Table 4.2: LL table

Nonterm	id	,	(	)	;	:	<?php\s	while	if	return	switch	for
⟨begin⟩	err	err	err	err	err	err	1	err	err	err	err	err
⟨st_list⟩	err	err	err	err	err	err	err	2	2	2	2	2
⟨st_list2⟩	err	err	err	err	err	err	err	5	7	8	9	6
⟨st_list3⟩	err	err	err	err	err	err	err	16	16	16	16	16
⟨casedef⟩	err	err	err	err	err	err	1	err	err	err	err	err
⟨par_list⟩	err	err	err	21	err	err	1	err	err	err	err	err
⟨par_list2⟩	err	22	err	23	err	err	1	err	err	err	err	err
Nonterm	break	continue	function	else	elseif	case	default	=	{	}	EOF	var
⟨begin⟩	err	err	err	err	err	err	err	err	err	err	pop	err
⟨st_list⟩	2	2	3	err	err	err	err	err	err	err	4	2
⟨st_list2⟩	11	12	13	err	err	13	13	err	err	13	13	10
⟨st_list3⟩	16	16	16	15	14	16	16	err	err	16	16	16
⟨casedef⟩	err	err	err	err	err	17	18	err	err	19	pop	err
⟨par_list⟩	err	err	err	err	err	err	err	err	err	err	pop	20
⟨par_list2⟩	err	err	err	err	err	err	err	err	err	err	pop	err

complex error recovery for the whole parser.

4.5.1 Panic-Mode for LL Parser

The method used for the LL section is called Panic-Mode Error Recovery. We use the following heuristics for a nonrecursive predictive parsing:

- A synchronizing set is created from the terminals and the nonterminals that are actually on the stack. When an input for a nonterminal leads to an error marked as *err* in the table, the method starts searching through the stack looking for the matching terminal or the nonterminal that has the input in the set *First*. We start searching from the nonterminal on top of the stack, so when there is some additional input symbol we can skip it and continue with the same nonterminal which led us to an error first. If the algorithm finds no matching terminal or nonterminal, it will continue to read the input symbols until a match in the synchronizing set is found. After the appropriate symbol is found on the stack, algorithm sets top of the stack to the symbol that has been found and syntax analysis can continue.

- The error marked as *pop* represents an opposite relation of the top nonterminal and the input as the error symbol *err*. The top nonterminal in that case is popped from the stack, as it cannot be matched, because the given input symbol is part of the *Follow* set for the nonterminal.
- If the terminal is on top of the stack and it is not matched we pop it and subsequently the error informs user that the symbol has been inserted.

4.5.2 Phrase-level Recovery for LR Parser

For the state that calls for a particular reduction, error entries have been replaced with the reduction to postpone an error that will still be caught before any shift move takes place. For the rest of error entries in the LR table have been created 9 special routines as the most likely assumption for the cause of an error.

e1: This routine does not produce an error, but detects a unary minus operator in an expression where a binary minus will cause an error. Consequently, the binary minus is replaced by the unary minus and the syntax is still correct.

e2: This routine represents an error, where an operand is omitted. Therefore a false operand is inserted.

$$\$foo = \$boo + *42;$$

$$\$foo = \$boo + \$false * 42;$$

e3: Only when the end symbol is not a right parenthesis, this error can appear, otherwise it is replaced by $err2$ or $success$. It represents an unbalanced right parenthesis that is immediately removed from the input.

$$\$foo = \$boo > 42);$$

e4: A comma can appear just as a delimiter for function parameters, every other appearance of that character in the given input is considered to be an error and the comma symbol is skipped.

$$\$foo = (,42);$$

e5: The routine is raised when an operator is omitted. As a solution a false operator is inserted.

$$\$foo = 42\$boo;$$

$$\$foo = 42 + \$boo;$$

e6: A function arguments or an expression in parentheses must be enclosed. This routine inserts a right parenthesis when it is omitted in the states 25 and 29.

$$\$foo = (42;$$

$$\$foo = boo(42;$$

e7: A function identifier must be followed by left parenthesis. This routine is raised only in the state 27. A left parenthesis is pushed on the stack to continue the analysis.

$$\$foo = boo);$$

e8: This error can appear in the state 27 and is similar to $err7$. The difference is that in the input is neither a left nor a right parenthesis. Most likely a user forgot to precede a variable with the symbol \$ or he forgot to write parentheses after a function identifier. The parentheses are pushed on the stack in order to recover and continue.

$$\$foo = boo;$$

$$\$foo = boo();$$

e9: In some cases when a comma and a parenthesis or a comma and a comma symbols are confronted, most likely a user omitted to state a function parameter. Therefore a false operand is inserted to continue.

$$\$foo = boo(, 42);$$

4.5.3 Connection of the Error Recovery methods

The set of input symbols for the LL section and the set of input symbols for the LR section are different. In connection with this fact we must solve the problem when an input symbol from the LL section appears in the LR section and vice versa. In the LL section an invalid input symbol is skipped and the error message „Unexpected token ...“ is produced. In the LR section an invalid input symbol is returned to the LL section, the LR analysis is terminated and the error message „Unexpected token in expression ...“ is produced. This approach expects that the end-of-expression symbol has been omitted. The control is handed over to the superordinate LL section.

Chapter 5

Implementation of the new parsing method

The application has been implemented in C programming language. We use only functions that are a part of standard libraries. The program is written as a console application and it does not provide the user with additional GUI.

5.1 Application structure

The application is separated into four logical units. The first unit encompasses lexical analysis. The implementation is in the module **scanner.c**. The second and the third units contain algorithms for the LL and LR parsers, respectively. Main module that implements the LL parser is **syntax.c**. Syntax analysis for expressions performed by the LR parser is implemented in module **expressions.c**. The last unit **3ac.c** embraces an implementation of the three address code generator. The main module of the application is called **main.c**. The parsing tables, rules and sets are defined in **tables.c**. The other modules are **pushdowns.c**, **expressions.c** and **debug.c**. These modules contain ancillary functions and data such as an implementation of the stacks, debugging functions, tables and other data structures.

5.2 Representation of data

This section describes the most important data structure in the application. To understand how the following concept works it is important to know that the terminals and the nonterminals are represented together in the one type *enum*.

5.2.1 SLR Table

The action part of the SLR table is represented by a two dimensional array. The rows represent the states and the columns stand for the set of input symbols for the LR section. The table entries have four different types. The $shift$ entries are numbers lower than 100 that represent a value of the state in the next step. The *reduction* entries represent numbers of the rules increased by 100. Therefore these entries are numbers higher than 100 and lower than 200. The numbers over 200 represent the error codes. The highest number of a type *unsigned char* stands for the *success* symbol. This concept of a data representation is one of many possible solutions how to differentiate the entries in the table.

The goto part is created by the only one nonterminal E. Therefore the table has been separated and for this part has been created the one dimensional array.

5.2.2 LL table, rules and the set First

The LL table is represented as a two dimensional array. The rows are the nonterminals and the columns represent the set of input symbols for the LL section. The entries are numbers of the rules, where the first rule has a number 1. The number 0 and the highest number of a type *unsigned char* 255 represents the error states *err* and *pop*, respectively.
The rules are represented as a structure of one *integer* and an *array.* The array is a representation of a right side of a rule. The integer represents a number of terminals and nonterminals on the right side.
The set $First$ is two dimensional array where the rows represent the nonterminals. In every row are terminals that belong to the set $First$ of the represented nonterminal. The number of columns is determined by the set $First$ with the highest cardinality.

5.2.3 Three Address Code

The generated code is at first prepared as a structure in memory. At the end of the syntax analysis this structure is used to produce the text output. The created instructions are pushed on the stack, therefore in memory they are stored in order in which they have been created. Additionally every instruction has a reference to the next one. The references are used as a valid chain of instructions that can be written to an output.

5.3 Implementation of the Lexical Analysis

Implementation of lexical analysis has been borrowed from my team's school project for the course IFJ. The language in the mentioned project was as well a subset of PHP. The implementation was adjusted according to the requirements of the language, which is the subject of this thesis. The lexical analyser is programmed as a finite state machine. The token keeps information about its type, current line of code, length and data according to the type of a token. The function to get the token from the scanner is called *scanner_get_token.*

5.4 Implementation of the Syntax Analysis

The algorithm for top-down parser is implemented in the function LL_parser. The bottom-up parser implementation embraces the function LR_parser. To the implementation of the syntax control are added function calls to notify the code generator about the actions. The LR parser uses only one table for three kinds of expressions. When the nonterminal $\langle expr1\rangle$ is going to be analysed we replace the value of one error entry to a success value. Afterwards when the LR analysis is terminated we replace it back. The set of the valid tokens is also different for all three types of the expressions. Therefore we created three different functions to control the validity of tokens and the function pointer that is set to corresponding function at the beginning of LR analysis according to the type of the expression. The implemented algorithms of top-down and bottom-up parsing are described in chapter 3.

5.5 Implementation of the Three Address Code

The task of the code generator is to take data from the syntax analyser and create the chain of instructions accordingly. The code generator uses its own stack to store the context about analysed expressions. This stack is also used to control if *break* or *continue* has a corresponding statement. The labels in the generated code have a name of the corresponding statement and number that is kept and increased for every statement separately.
Three functions are implemented in modules for the syntax analysis. These functions are *Three_AC*, *Expr_3ac* and *Expr_3ac_single_oprnd*. They are called to notify the code generator about a derivation or a reduction that was performed. When the expression is only one token, no operation is detected, hence the function *Expr_3ac_single_oprnd* must create assignment additionally. Otherwise the assignment is a product of an operation, where we create a new local variable that stores the result of an operation.
For small actions of the parser we implemented the code generator routines which create and link instructions. The function that writes the result of the program to the standard output is called *Write3ac*. The three address code statements were primarily designed according to [1] and some additional constructions are taken from [4]. The three address code is just the way how to visualise the results, but the main part is the syntax analysis, therefore we omit to state the number of local variables at the beginning of the function declarations. This feature can be added additionally as a part of a next work.

Chapter 6

Conclusion

Let us present and discuss the contributions of this thesis. To shortly summarize the aim of our work, it was to demonstrate a simple idea of the new approach to a syntax analysis. According to this idea the language is divided into sections. The most effective and suitable method of parsing is chosen for each section. We construct grammars and the parsers accordingly. The plan is to connect these parsers as a puzzle and finally create one hybrid method that is more effective than if any of these methods were used alone.
The theoretical part of the thesis gives the user complete explanation and a mathematical base to not only understand but also to create the syntax analysers that we use here. We employed definitions and algorithms to avoid fuzzy explanations of terms and notions. After specifying the theory base we created a simple language for a demonstration purpose. We separated the language into the two sections. For one section we created the LL syntax analyser. The LL syntax analyser is simple enough to be created manually without the need to use additional generators. For some constructions of the language it was the best choice. But it also has disadvantages. The LL grammar for expressions is not very effective, because it has additional productions which make the syntax analysis slower. Also the basic expression grammar is left recursive so we need to use the left recursion replacement method to prevent the parser from loops. Using an ambiguous grammar in LL parsing is not a straightforward method. Therefore, we decided to use an LR parser for the expression part.
The grammar created for expressions was ambiguous, but conflicts in the LR table were quickly resolved. The question is now, why did we not use the LR parser for the whole language. The Simple LR table for the whole language would have more than 100 states. In a table of this size, it is not easy to orientate and resolve conflicts. Even if conflicts were resolved by generator software, it would be difficult to implement the Phrase-level recovery method suitable for expressions, by analysing every error entry in the table that is so big.
The last thing that has to be figured out is how to connect those methods to produce only one syntax analyser. This thesis only proved that it is possible, but it was not the main interest. However, this could be an interesting topic for another research.
The result is not only the faster syntax, but also new ideas and suggestions to research in this area of interest. At first, it was a possibility to separate the language into smaller parts and reduce the number of entries in LR table using only LR parsers. The idea also brings a new perspective of a parser modularity, which can improve development of complex languages. The presented method can be used together with parallel parsing methods, in which gives us a new topic to research.
If we consider that LR grammars are stronger than LL grammars (see [1]), we could choose

language which has a few constructions that can be analysed only by an LR parser and try to apply our method there. This approach has been already mentioned and described in the work of Bostjan Slivnik [5]. This principle can be applied also for other kinds of grammars that have different strength.

Bibliography

[1] Alfred V. Aho et al. *Compilers: principles, techniques & tools.* Pearson Education, Inc, 2006.

[2] M. Češka et al. Gramatiky a jazyky. Skriptum VUT Brno. Ediční středisko VUT Brno, 1985.

[3] Alexander Meduna. *Automata and Languages: theory and applications.* Springer, 2000.

[4] Keith Schwarz and Jinchao Ye. Three-address code IR. Lectures for students in Stanford University, California http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/, 2012.

[5] Boštjan Slivnik. LL conflict resolution using the embedded left LR parser. *Computer Science and Information Systems*, 9(3):1105–1124, 2012.

[6] J. P. Tremblay and P. G. Sorenson. *The theory and practice of compiler writing.* McGraw-Hill, c1985.

Appendix A

Content of CD

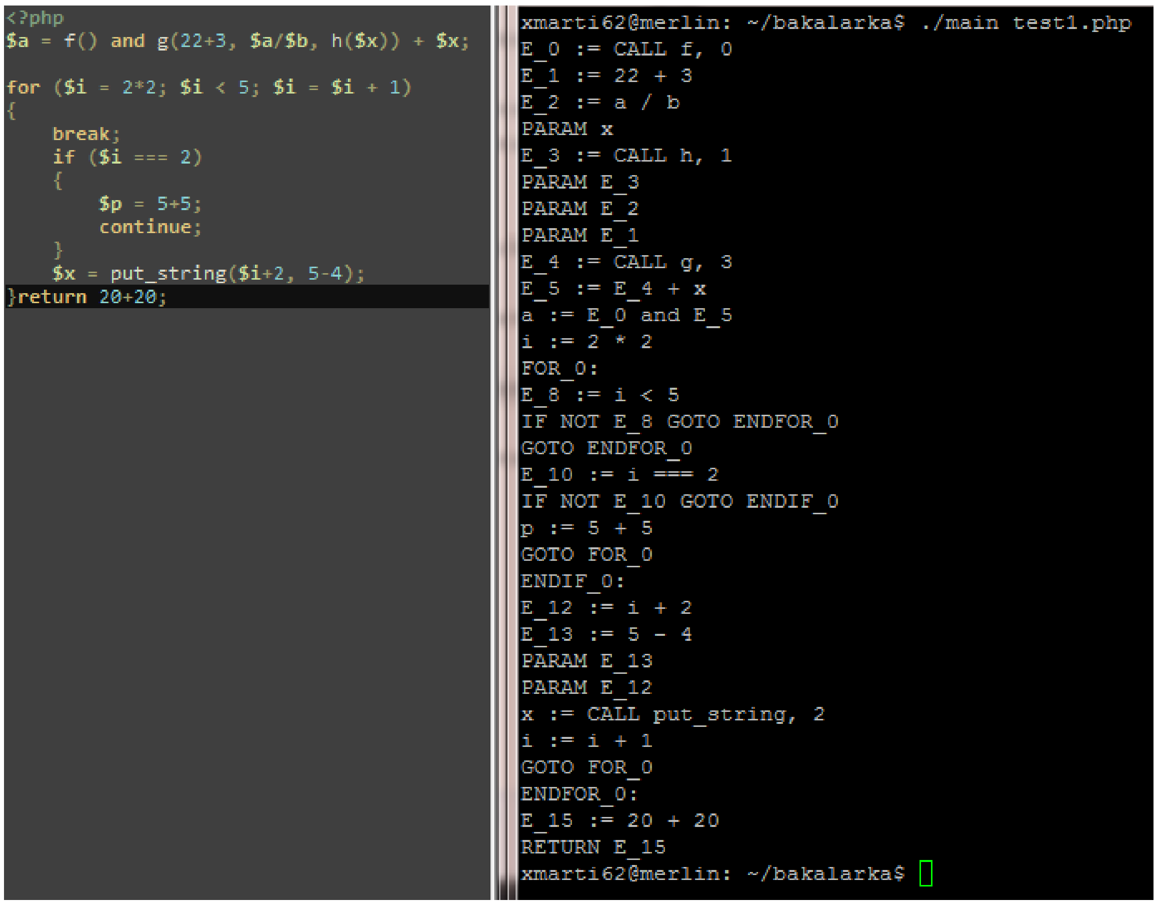
The content of the attached CD is following:

- **src/** folder with application source files
- **input/** folder with examples of input files written in the language, which is introduced by this thesis
- **thesis.pdf** content of the thesis
- **src-latex/** source files for the text of the thesis in latex

Appendix B

Manual

The executable file is called **main**. The program requires only one argument which is the name of an input file. After the program is successfully terminated, the corresponding three address code is written to the standard output. When some errors occur during the syntax analysis, we provide the user with the line number, where it was located and the error message according to the type of an error.

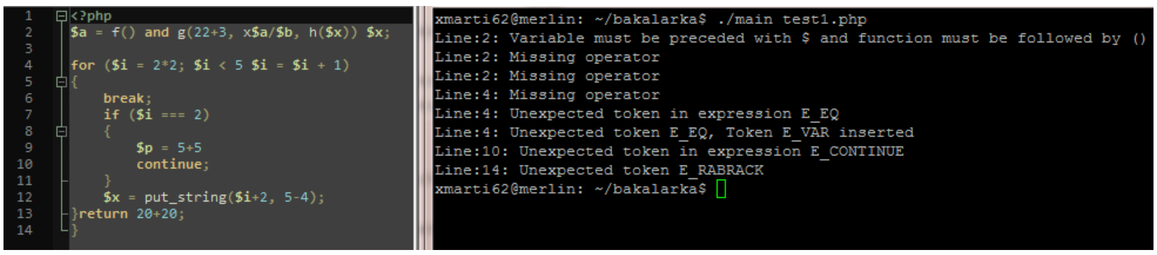


```php
<?php
$a = f() and g(22+3, $a/$b, h($x)) + $x;

for ($i = 2*2; $i < 5; $i = $i + 1)
{
    break;
    if ($i === 2)
    {
        $p = 5+5;
        continue;
    }
    $x = put_string($i+2, 5-4);
}return 20+20;
```

```
xmarti62@merlin: ~/bakalarka$ ./main test1.php
E_0 := CALL f, 0
E_1 := 22 + 3
E_2 := a / b
PARAM x
E_3 := CALL h, 1
PARAM E_3
PARAM E_2
PARAM E_1
E_4 := CALL g, 3
E_5 := E_4 + x
a := E_0 and E_5
i := 2 * 2
FOR_0:
E_8 := i < 5
IF NOT E_8 GOTO ENDFOR_0
GOTO ENDFOR_0
E_10 := i === 2
IF NOT E_10 GOTO ENDIF_0
p := 5 + 5
GOTO FOR_0
ENDIF_0:
E_12 := i + 2
E_13 := 5 - 4
PARAM E_13
PARAM E_12
x := CALL put_string, 2
i := i + 1
GOTO FOR_0
ENDFOR_0:
E_15 := 20 + 20
RETURN E_15
xmarti62@merlin: ~/bakalarka$
```

Figure B.1: Example of the output when the syntax of the input file is correct



```
 1  <?php
 2  $a = f() and g(22+3, x$a/$b, h($x)) $x;
 3
 4  for ($i = 2*2; $i < 5 $i = $i + 1)
 5  {
 6      break;
 7      if ($i === 2)
 8      {
 9          $p = 5+5
10          continue;
11      }
12      $x = put_string($i+2, 5-4);
13  }return 20+20;
14  }
```

```
xmarti62@merlin: ~/bakalarka$ ./main test1.php
Line:2: Variable must be preceded with $ and function must be followed by ()
Line:2: Missing operator
Line:2: Missing operator
Line:4: Missing operator
Line:4: Unexpected token in expression E_EQ
Line:4: Unexpected token E_EQ, Token E_VAR inserted
Line:10: Unexpected token in expression E_CONTINUE
Line:14: Unexpected token E_RABRACK
xmarti62@merlin: ~/bakalarka$
```

Figure B.2: Example of the output when there is a syntax error in the input file