FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Compiler Front-end for the IEC 61131-3 v3 Languages

Bruno Gonçalves Silva

Master in Electrical and Computers Engineering

Supervisor: Mário Jorge Rodrigues de Sousa

February 24, 2017

Resumo

Este projeto surge da necessidade de atualizar o compilador MatIEC tornando-o compatível com a terceira edição da norma IEC 61131-3. Durante o estudo da norma, várias inconsistências entre o texto principal e a especificação formal dos elementos de linguagem foram encontradas o que resultou na publicação de um artigo na conferência internacional ETFA'16. Dois diferentes geradores de analisadores léxicos e sintáticos foram comparados: ANTLR e COCO/R tendo este último sido utilizado para produzir os analisadores para as linguagens textuais definidas na norma IEC 61131-3. O parser utiliza a AST do MatIEC com as adaptações necessárias para incluir as novas características. O resultado foi, por fim, utilizado em conjunto com funções do MatIEC que partem da AST gerada para produzir o código original para validação.

Abstract

The purpose of this project was to update the MatIEC compiler thus making it compliant with the third edition of the IEC 61131-3 standard. During the study of the standard several inconsistencies between the main body and the formal specification of language elements were uncovered which resulted in a publication in ETFA'16 international conference. Two different Lexer and Parser generators were compared: ANTLR and COCO/R and the latter was used to produce the Lexical and Syntactical analysers for the textual languages defined in IEC 61131-3. The parser uses MatIEC's AST with the necessary adaptations to include the new features. Then, the result was linked with MatIEC's functions that produce the original input code from the generated AST for validation purposes.

Acknowledgements

Despite being entirely written in English, all the people I would like to acknowledge are Portuguese and it only seems fair that I thank them in our native language as some meaning might be lost in translation. If the reader is particularly curious about this section of the document, it might be a tremendous excuse to learn the beautiful and complex language born in this little country planted by the sea.

Há alguns que acreditam que a dissertação é para "aprender e explorar uma área específica". Não podiam estar mais enganados. Quero recordar e agradecer às pessoas que me ensinaram nas mais variadas áreas do saber e do saber-pouco durante todo este percurso.

À malta do i105... Manel e Ventura que partilharam comigo discussões interessantíssimas sobre assuntos "importantes" e ainda ao Faria por ter sempre uma palavra engraçada na ponta da língua.

Ao pessoal da i103... Pascoal, Tiago, Russo, Quim, Rajão, Gabriel, Rosa, Filipe, Pedro Silva, Gil, Ciro, Sebe e Cisco que me ensinaram que não é preciso pôr tanto limão nas bifanas. E especialmente ao Ricardo por me ter convencido a ir no InterRail com ele e em que dormimos na rua.

Aos amigos do Pincho.

Ao Pardal: Receba e aceite este forte abraço.

Ao Leo, ao Potter, ao Gabriel António, ao Rui, ao Master e à Ana. São o bando mais estranho, inconstante e do mundo, mas são meus amigos.

Quero também agradecer à minha família por me relembrar que é preciso tomar banho e comer e cortar o cabelo de vez em quando...

Por último e acima de tudo, quero agradecer e dedicar esta dissertação à minha namorada Ana que conseguiu desencantar razões para não me deixar mesmo com a pouquíssima atenção que lhe dei e 90% dessa ter sido para fazer piadas com assuntos sérios!

Bruno Gonçalves Silva

"When your work speaks for itself, don't interrupt."

Henry J. Kaiser

Contents

List of Figures

List of Tables

List of Code Sections

List of Grammars

Abbreviations and Symbols

ANTLR	ANother Tool for Language Recognition (Lexer and Parser generator)
AST	Abstract Syntax Tree
BNF	Backus–Naur form
COCO/R	COmpiler COmpiler generating Recursive descent parsers (Lexer and Parser generator)
CST	Concrete Syntax Tree
DFA	Deterministic Finite Automata
EBNF	Extended Backus–Naur form
ETFA	Emerging Technologies and Factory Automation
FBD	Function Block Diagram
GNU	GNU's Not Unix
GPL	General Public License
IBM	International Business Machines corporation
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IL	Instructions List
LD	Ladder Diagram
LALR	LookAhead Left to right performing the Rightmost derivation parser with x lookahead tokens
LL(x)	Left to right performing the Leftmost derivation parser with x lookahead tokens
LR(x)	Left to right performing the Rightmost derivation parser with x lookahead tokens
NFA	Nondeterministic Finite Automata
OOP	Object Oriented Programming
POU	Program Organization Unit
SFC	Sequential Function Chart
ST	Structured Text
UML	Unified Modeling Language

Chapter 1

Introduction

1.1 Context and Motivation

The development of Programmable Logic Controllers (PLC) by many different entities originated a large variety of architectures, operating systems and even software development environments to this equipments. This abundance made the code re-utilization very difficult, the learning and complete usage of a new machine very slow and the utilization of controllers from different makers very complex. In an Industrial Environment this can be prejudicial. This problem causes a necessity to abstract the hardware architecture and the operating system to the software developer. The 61131 series of standards from IEC (International Electrotechnical Commission) arises to mitigate this challenge.

The series of standards is divided in eight different parts. The third of these, which is the focus of this dissertation, normalizes the programming languages to use when programming PLCs as well as the files formats where the information is saved[1]. The tree editions of the 61131-3 standard were approved in 1993, 2003 and 2013 and define four programming languages for PLCs: *Structured Text* and *Instruction List* (textual) and *Function Block Diagram* and *Ladder Diagram* (graphical) and yet one language for state machine specification: *Sequential Function Chart*. It is important to notice that the last (third) edition includes, apart from its predecessors, a definition of the programming languages so that they allow the code to be object oriented. This improvement is particularly important because it will permit PLC's code to be better structured as a whole.

The definition of programming languages is of utmost importance for a software engineer to be able to develop his code. However, without a program that translates the high-level code into a format that the computer/PLC may comprehend it, all that work is in vain. That program is called a Compiler. This Dissertation will be about the implementation of a compiler front-end to the third edition of the IEC 61131-3 standard. This topic is really interesting to the PLC market given that the standard has global acceptance proportions.

This project emerges from the need to update the MatIEC compiler so that it accepts the new syntax described in the third edition of the standard. The MatIEC is an open source IEC 61131-3 - second edition compiler developed primly by professor Mário de Sousa. Its relevance to the world

of Industrial Automation is the disconnection to proprietary licensed PLCs and contributing this way to the spread of the standard in its essence thus allowing the users to make portable code and not needing to introduce each manufacturer specificities[2].

1.2 Objectives

The main objective of this Dissertation is to develop a compiler front-end to the textual programming languages (Structured Text and Instruction List) as well as the textual description of the Sequential Function Chart language as they are described in the third edition of the IEC 61131-3 standard.

1.3 Document Organization

Apart from this Chapter, this dissertation includes 6 more chapters.
In Chapter 2 an extensive explanation of a Compiler Structure will take place.
Chapter 3 will include a brief overview of the IEC 61131-3 standard.
Chapter 4 is an analysis of the compiler design tools used in this project.
Chapter 5 shows the inconsistencies that were found in the third edition of the standard.
Chapter 6 consists of an explanation of the designed Lexer and Parser.
Chapter 7 concludes this dissertation and proposes future work.

Chapter 2

Compilers

This chapter first describes the typical Structure of a Compiler, after which a detailed explanation of the several phases involved in the compilation process. Following this discussion some compiler design tools are compared.

2.1 Introduction to Compilers

John Von Neumann introduced two important concepts in 1945: *shared-program technique* means that computer hardware should be simple and reprogrammable by complex instructions and *conditional control transfer* which suggested that a program should be able to be interrupted and reinitialized at will (notion of subroutine). These concepts were the beginning of a faster, more efficient and more adaptable computation era [3]. The first programming language appeared in 1949. It was called *Short Code* and was based in arithmetic expressions. This language, however, had to be converted to binary code by hand but it was an important initial step. Two years later Betty Holberton proposed the *Sort-Merge Generator*. This was the first way to create a program automatically. These two events inspired Grace Murray Hopper to develop the first compiler as she states in her keynote in [4, Grace Hopper].

Shortly after IBM (International Business Machines Corporation) released the FORTRAN I (developed from 1954 to 1957). made its appearance and by 1958 50% of all written code was in FORTRAN[4, John Backus]. FORTRAN introduced the `IF`, `DO` and `GOTO` statements, `boolean`, `integer` and `real` data types[3]. The next big steps in compiler's history were ALGOL (1958) and COBOL (1959) which had a better handle of input/output. From this moment on, the basis for the modern programming languages were set up.

A compiler is a software application that, above everything else and in the most simple way, translates a program written in a programming language to one equivalent program written in a target language [5, Chap. 1]. In order to do this, a compiler must know every single detail of the source programming language therefore it is able to detect the lexical, syntactical and semantic errors in the source program. The difference between these types of errors will be demonstrated later in this Chapter.

2.2 Compiler's Structure

A typical compiler is divided in two main parts:

- **Analysis** — also called *front end*, this part deconstructs the source program into its most basic pieces (*tokens*), checks if they are syntactically and semantically correct (according to the source language specifications) and produces a grammatically structured representation of the tokens that will be used to generate an intermediate representation of the final code;
- **Synthesis** — also called *back end*, this is where the target program will be generated and optionally optimized.

Each of this parts is subdivided in several others that can be seen in Figure 2.1. This phases will be detailedly explained on the following sections of this Chapter.

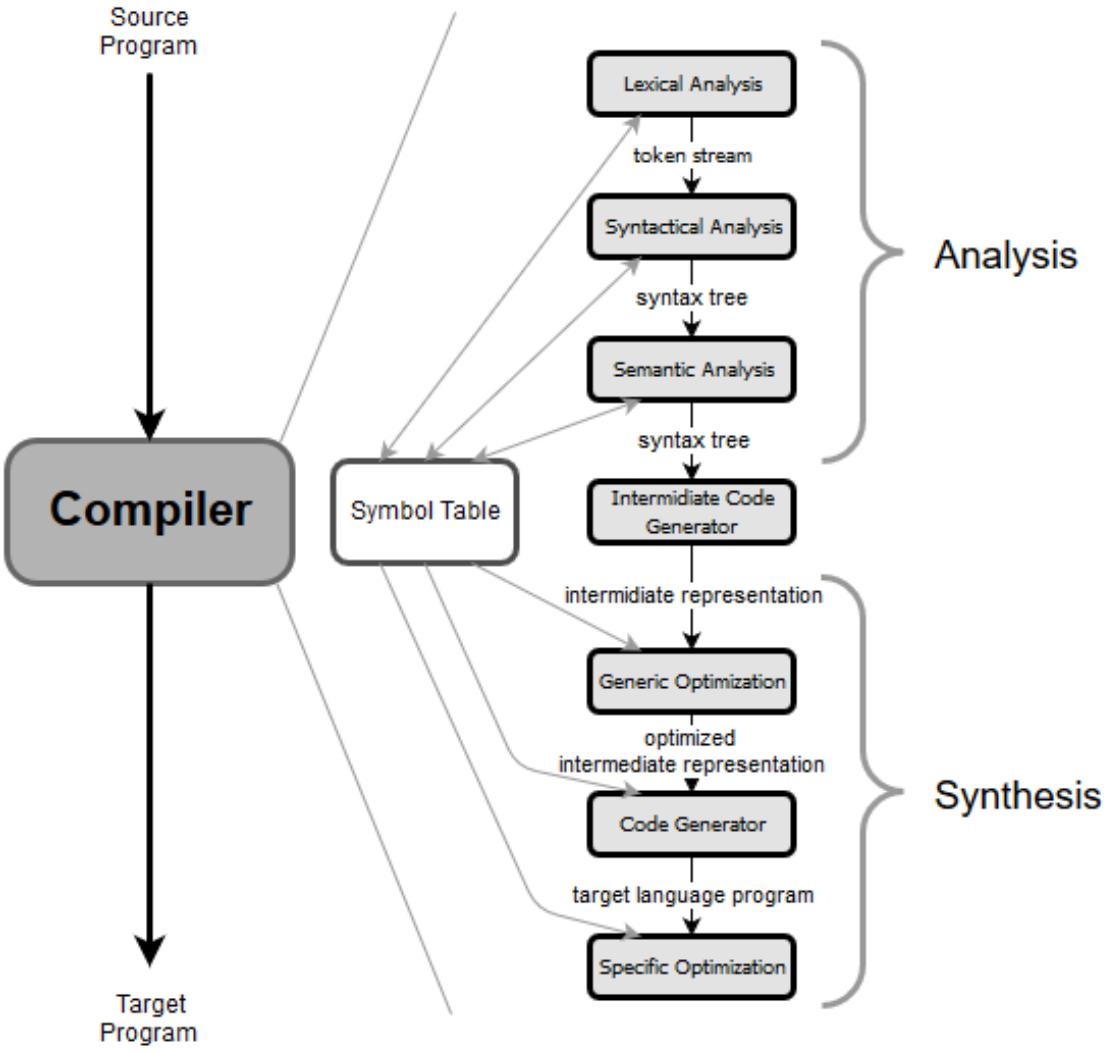

Figure 2.1: Structure of a Compiler (adapted from [5])

2.3 Lexical Analysis

input: stream of characters from the source program
output: sequence of tokens

This is the first phase of the compilation process. In this phase, the characters from the source program are grouped so that each of these groups (*lexemes*) has a meaning in the context of the programming language. For example for the following input (written in Structured Text):

```
variable_a:=variable_b+5;
```

the lexical analyser (also refereed to as lexer) should recognise the following lexemes: <variable_a>, <:=>, <variable_b>, <+> and <5>.

Typically, this phase is divided in two processes: *Scanning* and *Tokenizing*. The Scanner reads the input and finds the lexemes. This process includes the elimination of comments and excessive white space. After this, the Tokenizer determines the class of each lexeme and generates a *token*. A token (or terminal symbol) is a structure that represents a lexeme and it includes the class of the lexeme (token name) and the attribute. Examples of token names are: identifier, keyword ("if", "for", "while", ...), assign operator(":=") comparison operator ("<", ">=", "!="), number... It's important to note that for identifiers (names of variables) there is a need to store its name and value, therefore the attribute is always a pointer to the *symbol table*. The Symbol Table is the data structure that stores information (names, value, type, scope...) on the symbols of the program such as variables, functions, classes, objects. These tables will not be focused in this Section.
For the previous example, the Lexical Analyser should create the following tokens:
<**identifier**; pointer to variable_a in Symbol Table>, <**assign_operator**; :=>, <**identifier**; pointer to variable_b in Symbol Table>, <**operator**; +> and <**number**; integer 5>. For simplicity, tokens will be represented inside <> by its lexeme. For instance <`variable_a`>.

2.3.1 Lexical Errors

A Lexical error is one that can be detected in the Lexical Analysis phase. These errors may be unacceptable characters of the language (example: @) or lexeme that don't belong to any class (for example if an identifier is defined as a sequence of letters or digits beginning with a letter, the following example is a lexical error: 0a).
There are several possible error-recovering actions [5, Chapter 3]:

- Delete one character from the remaining input;
- Insert a missing character into the remaining input;
- Replace a character by another character;
- Transpose two adjacent characters.

(This document won't focus on these actions. The references [5, Section 3.1.4] and [6, Section 3.7.6] have useful information on this and may be of help.)

2.3.2 Regular Expressions

Regular Expressions are a simple way of specifying the set of tokens of a given language. In order to understand how, it's important to distinguish the concepts of alphabet[5]/vocabulary[6], string and language.

1. **Alphabet/Vocabulary** (Σ) is the set of characters allowed. Examples of this are the binary alphabet: $\{0, 1\}$, the ASCII alphabet (which includes 128 characters) or the English alphabet (which includes all the letters from a to z and A to Z and punctuation).

2. **String** is a sequence of characters from an alphabet. This concept also includes the empty string (usually represented as ε but also as λ) which is the string of length 0.

3. **Language** is the set of strings of a given alphabet. This definition includes the $\{\varepsilon\}$ language and the empty language ($\emptyset$).

The operations on languages defined formally in Table 2.1 must also be considered, to understand how regular expressions are created.

Table 2.1: Operations on Languages (adapted from [5] and [7])

Operation	Definition
Union[5]/Alternation[7]: $L \cup M$[1]	$\{s : s \in L \wedge s \in M\}$
Concatenation: $L \cdot M$	$\{st : s \in L \vee t \in M\}$
Kleene closure: L^*	$\cup_{i=0}^{\infty} L^i$

Note 1: L and M symbolize two different languages.
Note 2: s and t symbolize two strings.
Note 3: L^i represents the concatenation of L with L i times.
If $i = 0$ then $L^i = \varepsilon$, else if $i = 2$ then $L^i = L \cdot L$ and so on.

Another operation may be defined and added to the previous table that is a particular case of the Kleene closure where the ε string is removed. This is a useful and worth mention operation:

Positive closure: L^+	$\cup_{i=1}^{\infty} L^i = L \cdot L^*$

For a better understanding of this: Let L be a language defined by the set of strings $\{a, b, c\}$ and D another language defined by $\{0, 1, 2\}$. Using the previous operations new languages may be defined:

$$
\begin{aligned}
L \mid D &= \{a, b, c, 0, 1, 2\};\\
L \cdot D &= \{a0, b0, c0, a1, b1, c1, a2, b2, c2\}\\
L^2 &= \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}\\
L^* &= \{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, ...\}\\
L^+ &= \{a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, ...\}
\end{aligned}
$$

[1] The Union operation may also be represented with the symbol | (as in L | M) instead of $\cup$. And it's read as *or*.

Finally, regular expressions are defined recursively using smaller regular expressions as it can be seen below as an example:

$$
\begin{aligned}
\text{letter} &= \{a, b, c, ..., z, A, B, C, ..., Z\} \\
\text{digit} &= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
\text{signal} &= \{+, -\} \\
\text{underscore} &= \{\_\} \\
\text{numbers} &= (\varepsilon | signal) \cdot digit^{+} \\
\text{identifier} &= (letter | underscore) \cdot (letter | digit | underscore)^{*}
\end{aligned}
$$

Formally, regular expressions over a given alphabet Σ may be defined ([5] and [7]):

1. ε is a regular expression and L(ε) is the language whose set of strings is $\{\varepsilon\}$.
2. If a is a symbol in Σ, then **a** is a regular expression and L(**a**) = $\{a\}$.
3. r and s are regular expressions denoting L(r) and L(s) respectively, then:
   (a) $(r \mid s)$ is a regular expression denoting L(r) $\cup$ L(s).
   (b) $(r \cdot s)$ is a regular expression denoting L(r) $\cdot$ L(s).
   (c) r^* is a regular expression denoting L(r)*

Simplifying:
A regular expression is an expression that represents a search pattern for the acceptance of a valid set of strings for a determined purpose in the context of a given Language.

2.3.3 Finite Automata

Finite Automata (plural for Finite Automaton) is an algorithm used to *accept* or *reject* certain input. They are essentially transition diagrams where each node represents a state of the algorithm and each transition represents the acquisition of a certain symbol. It is widely used in Scanners to recognize tokens described by regular expressions. Finite Automata must always include ([5]):

1. A finite set of states.
2. An input alphabet Σ. (ε is never a member of Σ).
3. A transition function that gives, for each state and for each symbol in $\Sigma \cup \{\varepsilon\}$, a set of next states.
4. A start state.
5. A set of final states (*accepting states*).

Two different types of Finite Automata can be distinguished ([5]):

1. **Nondeterministic (NFA)**
   - A state may have transitions to different states triggered by the same input symbol
   - A transition may be triggered by ε.
2. **Deterministic (DFA)**
   - A state must have one and only one transition per symbol of the input alphabet.
   - There are no moves on input ε.

Consider a Language defined by $\{a, b\}$. There is a need to accept any string that ends in "*abb*". This can be achieved with the regular expression: $(a|b)^* \cdot a \cdot b \cdot b$ (find any number of a's or b's then an a then a b and finally a b). The NFA that describes this expression is the one in Figure 2.2.

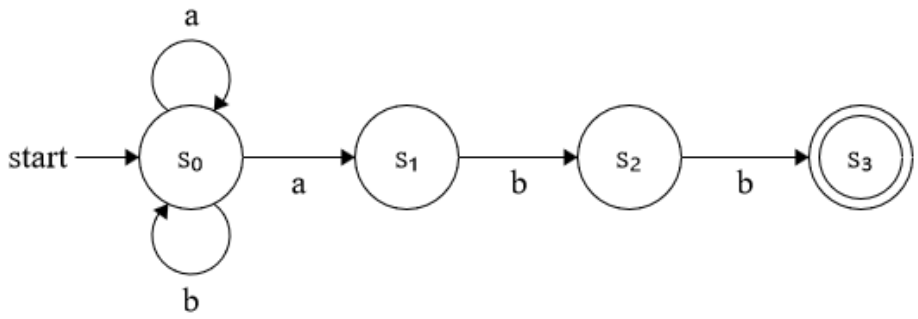

Figure 2.2: NFA that accepts $(a|b)^* \cdot a \cdot b \cdot b$ (example from [5])

The NFA accepts the input if there is at least one path that takes it to the accept state (s_3). For the input string "ababb" a possible sequence of states is: $s_0 \xrightarrow{a} s_0 \xrightarrow{b} s_0 \xrightarrow{a} s_1 \xrightarrow{b} s_2 \xrightarrow{b} s_3$ so this is a valid string.

All NFAs can be converted to a DFA however, they may become bigger and more complex than the original. Figure 2.3 shows the DFA equivalent to the previous example.

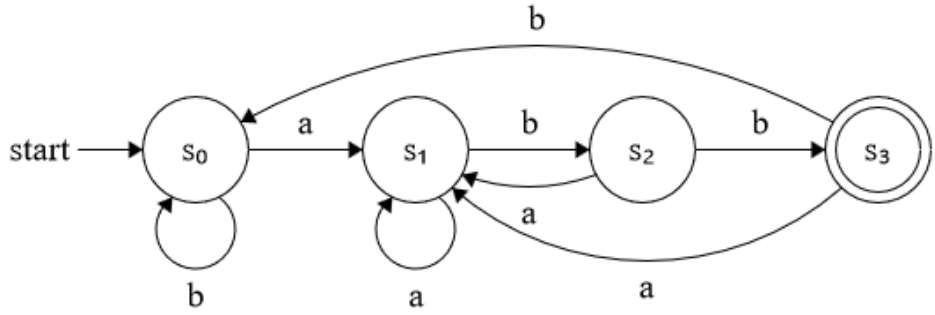

Figure 2.3: DFA corresponding to the NFA on Figure 2.2 ([5])

2.3.4 Types of Lexer generators

A Lexer generator receives as input the regular expressions that define the syntax of the input language, generates an NFA, then transforms it into the equivalent DFA and then generates the executing code for that DFA ([7]). There are three kinds of Lexer Generators worth noticing:

- **Table-Driven**
  This type of Lexer Generator uses a transition table and a "skeleton scanner" ([7]) to create the Lexical Analyser.
- **Direct-Coded**
  The Direct-Coded is a better option because it has the same functionalities of the table-driven but without the overhead of looking to the table. The states are coded directly.
- **Hand-Coded**
  The Hand-Coded Lexer Generators are only useful for small and not complex languages.

2.3.5 Summary of Lexical Analysis

In Language Theory one must say the **Lexical Analysis identifies words** by grouping characters.

Objectives:

- Scan input;
- Remove white space;
- Remove comments;
- Create Token;
- Detect lexical errors.

In this Section:

1. The concepts of Token, Lexeme and symbol table were introduced;
2. There was a brief overview of the lexical errors and how the lexer may recover from them;
3. Regular expressions were explained;
4. The differences between NFA and DFA were clarified;
5. Lastly the different types of lexer generators (table-driven, direct-coded and hand-coded) were instructed.

2.4 Syntax Analysis

input: token stream
output: syntax tree

The Syntax Analysis or *Parsing* is the next compilation phase. The parser must check if the token stream received from the Lexical Analyser is possible considering the syntax rules (*grammar*) of the input language and fit them in a data structure that will be used by the next steps of compilation (*Syntax Tree*).

A grammar is a formal description of a language. It defines what possible structures the code must follow in order to be accepted. There is a widely used formalism for grammar description, the Backus–Naur form (BNF). This notation was created by John Backus in the late 50's and popularized by Peter Naur (hence its name) right after[8][9]. Later the language suffered several extensions and originated variants such as the Extended Backus-Naur Form (EBNF) or the Augmented Backus-Naur Form. These are similar to the way regular expressions are defined so won't be deeply focused here. A BNF expression would look like shown in Grammar 2.1:

Grammar 2.1: BNF example rules

```
<Program> ::= <statement> <Program>
<Statement> ::= <Identifier> ':=' <Expression> ';'
```

This notation is too simple for some operations, for example in the Grammar 2.1 a program is defined as a list of statements, however it must be described with recursion wich makes it harder to read. For this reason the EBNF notation was developed. There are lots of slightly different versions of the EBNF. Pete Jinks' table at [10] may help the reader understand all of these variations.

The EBNF added several symbols to its predecessor. The most common ones are: the optional symbol (`[]` or `?`), the repetition symbol (`{}` or `*`), the grouping symbol (`()`), the end of rule (`;` or `.`), some versions even include a symbol for comments (`(**)`). Grammar 2.2 is a simple grammar written in ISO Extended Pascal EBNF (see [10]) that will be used throughout this:

Grammar 2.2: EBNF example Grammar

```
Program = Statement {Statement};
Statement = Identifier ':=' Expression ';';
Expression = Sum | Sub;
Sum = number '+' number;
Sub = number '-' number;
Identifier = letter {letter | digit};
number = ['-'] digit {digit};
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
letter = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | ... | 'Z';
```

2.4.1 Types of Grammars

According to [11], there are four types of grammars: Types 0 (*Unrestricted grammar*), 1 (*Context-sensitive grammar*), 2 (*Context-free grammar*) and 3 (*Regular grammar*). Each of these types of grammars are subsets of the previous one.

Unrestricted grammars (also known as Recursively Enumerable or Type 0 grammars) are the most general type and include every grammar that can be generated with a Turing machine. There is only one restriction in this type of grammar: the left side of a rule must not be empty. An example can be seen in Grammar 2.3.

Context-sensitive grammars (also known as Turing-decidable or Type 1 grammars) are a set of grammars that can be generated using a linear bounded automaton. They have an additional restriction which states that all rules must be of the form:

$$\alpha A\beta = \alpha\gamma\beta; \alpha \wedge \beta \in \{\varepsilon, teminals, nonterminals\} \wedge \gamma \in \{teminals, nonterminals\}$$

This restriction doesn't allow rules like `AB = BA;`. Grammar 2.4 is a simple example of this type of grammars.

Context-free grammars (also known as Type 2 grammars) are defined by adding another restriction to the previous type:

> The left side of every rule must have exactly one nonterminal and no context.

This means that whenever that nonterminal appears in the right side of a rule you can replace it with on of that nonterminal's right side. An example of a context-free grammar is shown in Grammar 2.5.

Lastly, the **regular grammars** (also known as Type 3 grammars) are a particular set of grammars whose rules right side just have a single terminal (`A = 'a';`) or a single terminal and a single nonterminal (`A = 'a'B;`). See example in Grammar 2.6.

Grammar 2.3: Example of Type 0 Grammar

```
    S = AB'a"b';
AB'a' = BA;
    B = 'b"b';
    B = 'b"b"b';
```

Grammar 2.4: Example of Type 1 Grammar

```
    S = AB'a"b';
   AB = A'c';
'c'B = 'c"b"b';
    B = 'b"b"b';
```

Grammar 2.5: Example of Type 2 Grammar

```
S = AB'a"b';
A = 'a';
A = 'a"a';
B = 'b"b"b';
```

Grammar 2.6: Example of Type 3 Grammar

```
S = 'b'A;
A = 'a'B;
B = 'b"b';
```

For a formal, extensive but not "reader friendly" definition of these types of grammars please see [11]. For a simplified version please see [12, Section 8.6].

2.4.2 Syntax Trees

The output of the parser is the Syntax Tree. The syntax tree is a data structure that describes the syntactic structure of the source code. Syntax trees may be Abstract or Concrete. The difference between these two is the fact that the Concrete Syntax Tree (CST) represent all of the input (including whitespace, keywords, ...) while the Abstract Syntax Tree (AST) uses the tree structure to imply some of the input. Consider Grammar 2.2. The CST and AST for the input "`A:=1+1;B:=1-2;`" would look like Figure 2.4 and Figure 2.4 respectively.

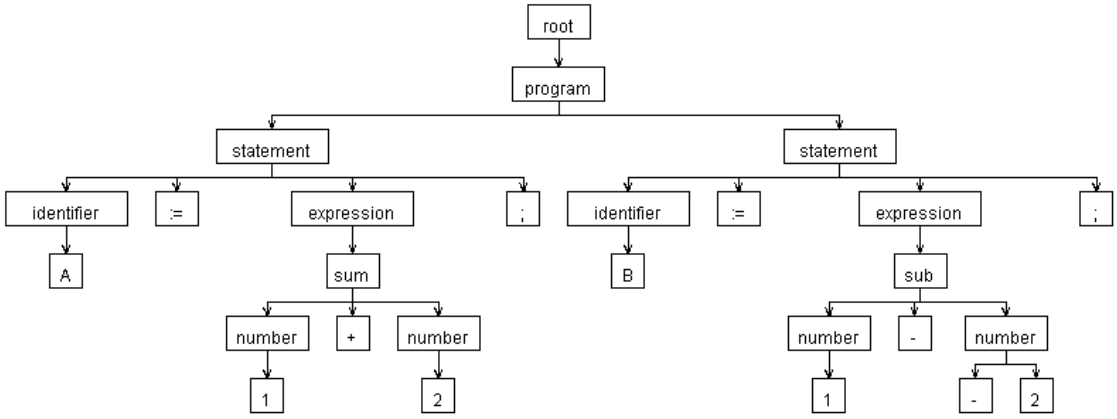

Figure 2.4: CST generated for "`A:=1+1;B:=1-2;`" with respect to Grammar 2.2

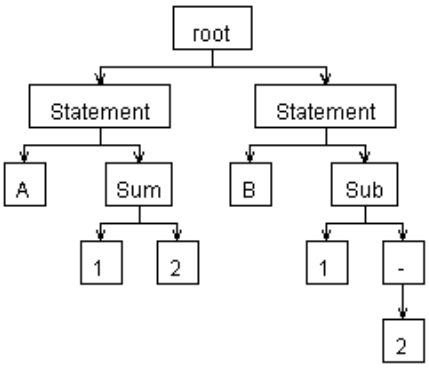

Figure 2.5: Possible AST generated for "`A:=1+1;B:=1-2;`" with respect to Grammar 2.2

In this example, the tokens <;> and <:=> are not needed because if they were missing the parser would produce an error and would not output the syntax tree. In this particular example the node statement as just two leaves: the identifier (left) and the expression (right). Generating an AST instead of a CST results, potentially, in smaller syntax trees (as shown in Figures 2.4 and 2.4) and may reduce compilation time as the next steps won't have to deal with so much information.

2.4.3 Syntactic Errors

A Syntactic error is one that can be detected in the Syntax Analysis phase. These errors may be missing or extra tokens (<+> for example), unbalanced parenthesis or braces... These are errors in nonterminal symbols of the grammar. Using Grammar 2.2 as an example, the input `A : 1+1;` would be a Lexical error (as the lexer wouldn't be able to form a token from `:`) and `A := 1++1;` would be a Syntactic error because the rule `Sum` has an extra <+> which is unexpected.
There are two main error-recovering strategies:

- Panic-Mode Recovery[5] / Error Recovery[6];
- Phrase Recovery[5] / Error Repair[6].

The first consists in skipping input tokens until a unambiguous delimiter appears, for example a <;> or a <}> and then continuing the analysis. The choice of delimiter is in the hands of the compiler designer. [5] calls these delimiters *synchronizing tokens*.

The second strategy uses input correction by replacing, removing or adding a character in order to allow the parser to proceed beyond the point of error detection. Examples of usage are the insertion of a missing <;> or scope corrections (by inserting or removing unbalanced braces). This technique is more difficult than the previous one and may lead the parser to an infinite loop in certain situations[6].

Instead of correcting errors, predicting some of the most common errors and creating specific rules for those in the Grammar itself is extremely helpful (Error Productions[5]). This usually leads to better reports and even suggestions of correction.

In the literature there are some references to Global Correction algorithms which can find the minimum number of alterations to the source code that lead the parser to an acceptable syntax tree. These algorithms require, however, too much space and time to be implemented in a compiler. See [5], [6], [7] and [12].

2.4.4 Types of Parsers

Parsers may be one of three kinds: Universal, top-down and bottom-up [5].

Universal parsers can parse every grammar no matter its complexity. They can find every parse tree for a given input sequence by trying every subtree for the partial input sequences derived from that same input. These are, however powerful, too inefficient to be used and will not be covered by this document. Examples of Universal parsing algorithms are the Cocke-Younger-Kasami algorithm (commonly known as CYK-algorithm) and the Earley parsing algorithm.

If the reader wants to know more on the CYK-algorithm, please see [13], [14], [15], [16].

If the reader wants to know more on the Earley algorithm, please see [17] and [16].

Top-down parsing is a parsing method that creates the parse tree for some input starting in the root node and continuing to the leaves. It takes one token at a time and decides where it fits in the tree before continuing to the next. This approach is almost always not enough because the same token may fit several alternatives at a time.

For a simple explanation of the top-down parsing method consider once again Grammar 2.2. Consider, yet, that the source code to be parsed is `A:=1-2;`. This input would be tokenized by the scanner as: <A>, <:=>, <1>, <->, <2> and <;>. The parser will try to get a `Program` so this is the start rule. It will expect a `Statement` and then an `Identifier`. The first Token will match this rule so the evolution of the syntax tree will look like Figure 2.6.

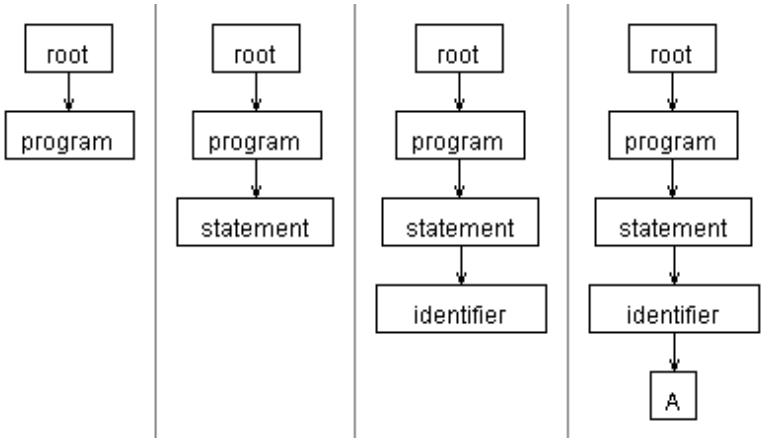

Figure 2.6: Top-down parse tree evolution after token <A> of of the token stream <A> <:=>, <1> <-> <2> <;> with respect to Grammar 2.2

The next token to appear is <:=> which is expected in the `Statement` rule.

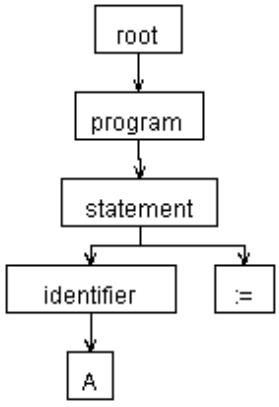

Figure 2.7: Top-down parse tree after token <:=> of of the token stream <A> <:=>, <1> <-> <2> <;> with respect to Grammar 2.2

The next token is a `number` and it fits both in `Sum` as in `Sub` so the parser will try the first to

appear in the `Statement` rule. In this example it will try the `Sum` rule. Figure 2.8 shows exactly this.

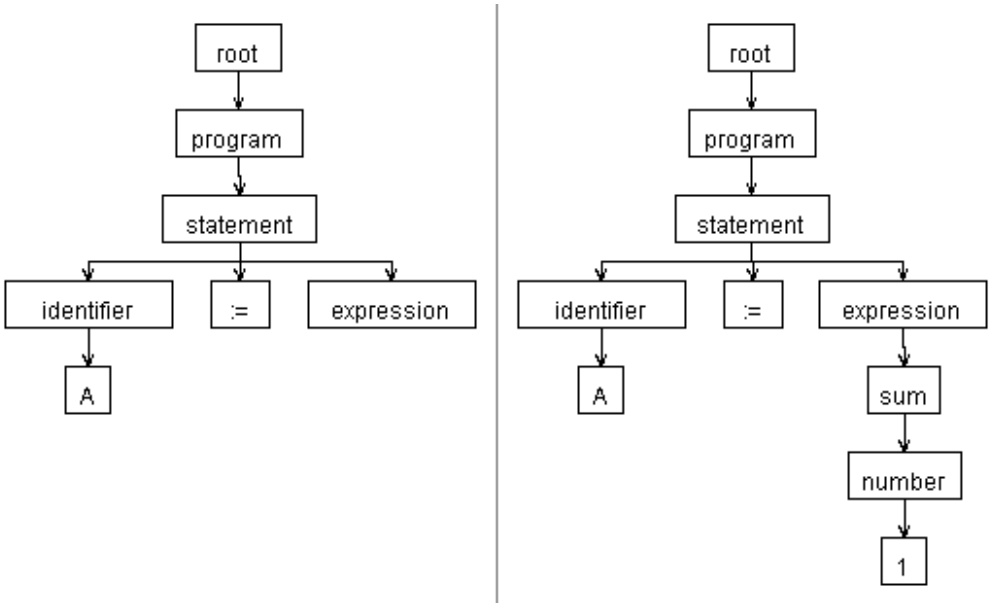

Figure 2.8: Top-down parse tree after token <1> of of the token stream <A> <:=>, <1> <-> <2> <;> with respect to Grammar 2.2

The next token (<->) will not be expected by the parser at this time. So the parser has to go back and try the second alternative (which, in this case, will fit). This technique is called *backtrack* and is used in **Recursive descent parsers**. Alternatively the parser could predict which of the alternatives would fit before generating the next node by using a *lookahead* token. This type of parsers are called **Predictive parsers** and are defined by the number of lookahead tokens they can use. For this example an LL(1) parser would suffice because it just needs to lookahead one token in order to decide which rule is the right one. If it sees a <+> it would decide to go for the `Sum`. If it sees a <-> then it knows to expect a `Sub`. LL stands for *L*eft to right performing *L*eftmost derivation which means the parser will read the input from left to right and will always find the Leftmost symbol (token or non-terminal) to be part of a lower level symbol (token or non-terminal).

Figure 2.9 shows the syntax tree after the parser decides the correct alternative is the `Sub` rule. In a predictive parser the `Sum` rule would never had been selected and the step shown by Figure 2.8 would not actually happen.

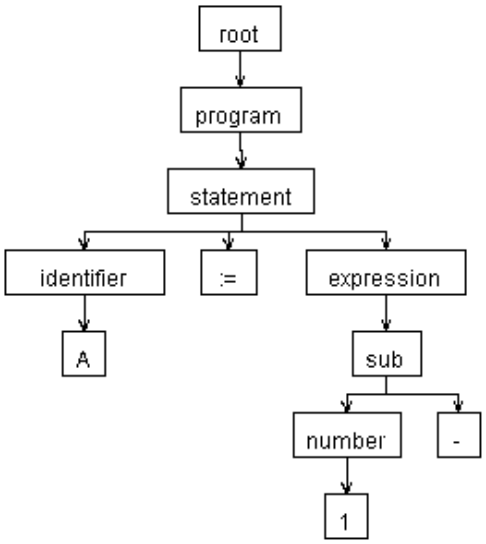

Figure 2.9: Top-down parse tree after token <-> of of the token stream <A> <:=>, <1> <-> <2> <;> with respect to Grammar 2.2

The last two tokens follow the same principles and just complete the `Sub` rule and the `Statement` rule which ends the `Program` correctly:

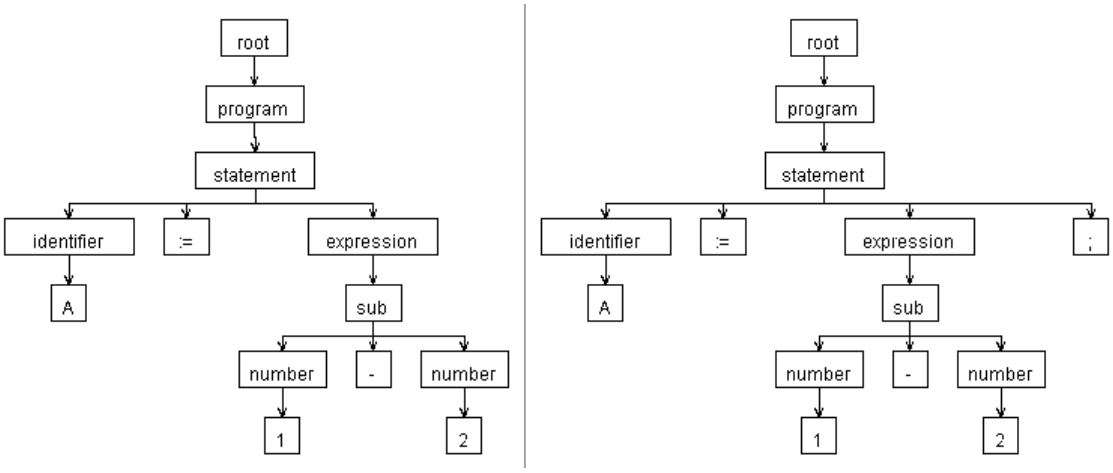

Figure 2.10: Top-down parse trees after tokens <2> (left) and <;> (right) of the token stream <A> <:=>, <1> <-> <2> <;> with respect to Grammar 2.2

A big problem with this type of parsers is *Left Recursion*. Grammars that have a nonterminal rule such as the next symbol to consume is the same as the rule (for example: `S = S a;`) will become trapped in an infinite loop while trying to see what tokens are expected for the `S` rule. This also happens indirectly for example if both `S = B a;` and `B = S b;` are part of the same grammar. For more information on this problem please refer to [6], [7] or the online lectures [18, Chapter 8].

As the name suggests, **Bottom-up parsers** are the opposite of the top-down parsers. They start with the leaves and build their way up to the root. The most general algorithm for this type of parser is called Shift/Reduce. This concept is linked with a type of parser called **LR**(k) parsers which stands for *L*eft to right performing the *R*ightmost derivation and the k is the number of lookahead tokens it supports. This means the tokens are red from left to right and the parser decides at each state whether the rightmost symbol (token or non-terminal rule) can be reduced to a higher symbol or a new token should be shifted. There are several advantages of this type of parser for example[5]: They may be used for almost all context-free grammars and no other left to right parser can detect a syntactic error sooner than the LR parser. On the other hand it is much harder to implement an LR parser than an LL parser and the code is inevitably more difficult to analyse. More advanced types of LR parser are the canonical-LR and the LALR but these will not be covered here.

Parallelly to the top-down parsing methodology consider Grammar 2.2 and the input `A:=1-2;` which is "fed" to the parser as the tokens <A>, <:=>, <1>, <->, <2> and <;>. The parser will now look at the first token and try to reduce it as much as it can. It will in this case produce a syntax tree like the one shown in Figure 2.11.

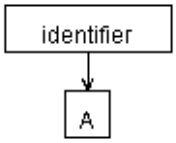

Figure 2.11: Syntax tree created by bottom-up parser after the token <A> of of the token stream <A> <:=>, <1> <-> <2> <;> with respect to Grammar 2.2

The next token (<:=>) will not lead to a reducible state, so the parser will shift the following (<1>) which can be reduced to number. This is shown in Figure 2.12.

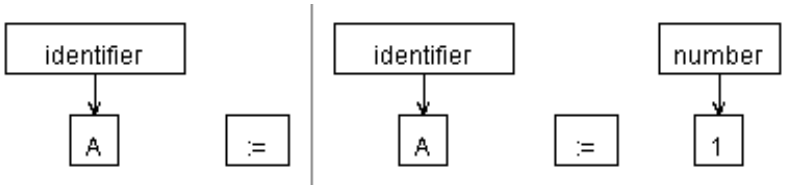

Figure 2.12: Syntax trees created by bottom-up parser after the tokens <:=> (left) and <1> (right) of the token stream <A> <:=>, <1> <-> <2> <;> with respect to Grammar 2.2

At this moment the parser has to shift twice before being able to reduce again (see Figures 2.13 and 2.14).

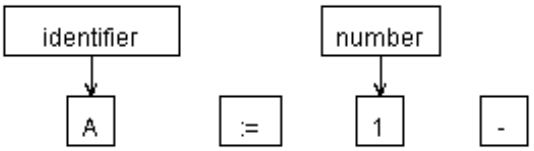

Figure 2.13: Syntax tree created by bottom-up parser after the token <-> of the token stream <A> <:=>, <1> <-> <2> <;> with respect to Grammar 2.2

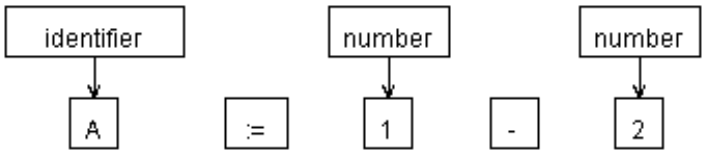

Figure 2.14: Syntax tree created by bottom-up parser after the token <2> of the token stream <A> <:=>, <1> <-> <2> <;> with respect to Grammar 2.2

After reducing <2> to number the rightmost tokens in the tree form a `Sub` which is an `Expression`, so the parser will reduce twice (see Figures 2.15 and 2.16). As it can be seen, this type of parser won't need to backtrack as the its top-down equivalent.

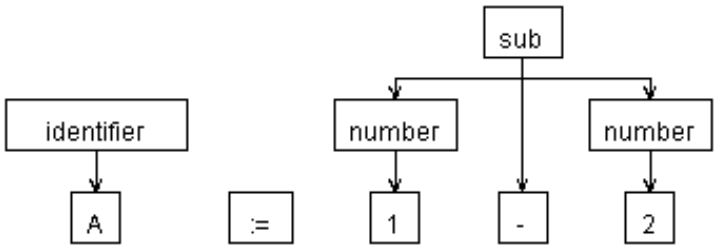

Figure 2.15: Syntax tree of Figure 2.14 after the reduction of `number` <-> `number` to `Sub`

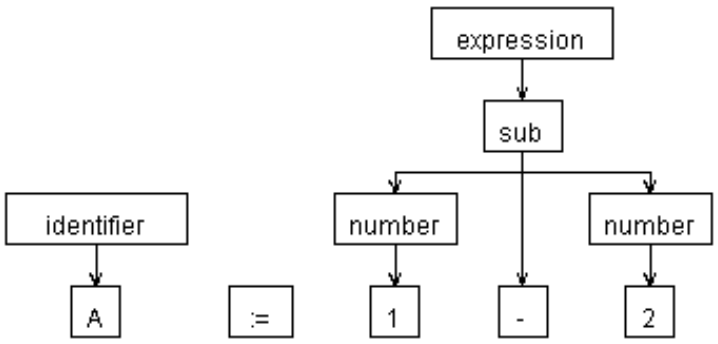

Figure 2.16: Syntax tree of Figure 2.15 after the reduction of `Sub` to `Expression`

The last token (<;>) will correctly end the `Statement` (Figure 2.17) and the `Program` (2.18) rules.

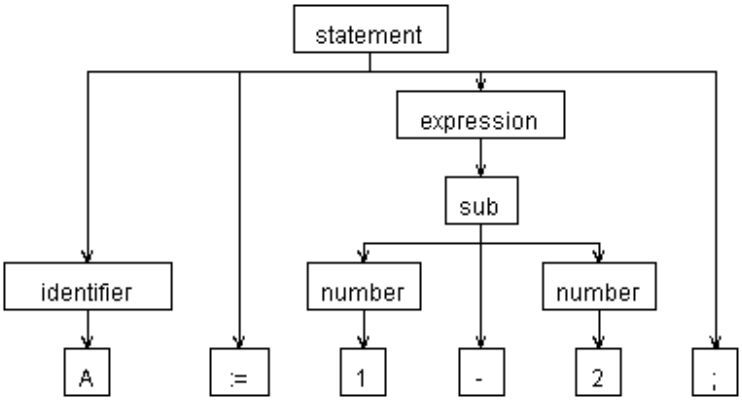

Figure 2.17: Syntax tree created by bottom-up parser after the token <;> of the token stream <A> <:=>, <1> <-> <2> <;> with respect to Grammar 2.2

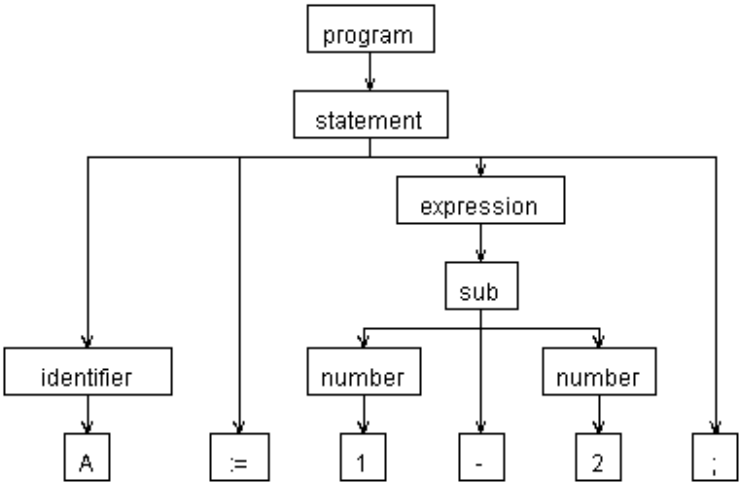

Figure 2.18: Syntax tree of Figure 2.17 after the reduction of `Statement` to `Program`

Despite not needing to backtrack, this methodology has a disadvantages when compared to top-down parsing: *shift-reduce conflicts* and *reduce-reduce conflicts*. The particulars of these problems and their solutions will not be presented in this document. For more information on this topic please refer to [6], [5], [18] which has a great tutorial like explanation of how SLR parsers deal with this by using handles.

2.4.5 Summary of Syntax Analysis

In Language Theory one must say the **Syntactical Analysis identifies sentences** by grouping words.

Objectives:

- Analyse the token stream
- Generate the Syntax Tree
- Detect Syntactical Errors

In this Section:

1. The Grammar writing formalism BNF and its extension EBNF were introduced;
2. The various types of Grammars (Unrestricted grammars, Context-sensitive grammars, Context-free grammars and Regular grammars) were explained;
3. The differences between CST and AST were clarified;
4. There was a brief overview of the syntactic errors and how the parser may recover from them;
5. Lastly the different types of parser (Universal, Top-Down and Bottom-Up) were instructed.

2.5 Compiler Design Tools

There are a large set of tools available for Compiler's Designers that help in the development of a compiler for a given language. In this section several types of these tools will be presented and some examples of each one will be compared.

2.5.1 Lexical Analyser generators

There are a great number of options when speaking of Lexer generators ([6, Chapter 3]):

- **Lex** - table-driven - outputs to C - distributed under proprietary license
- **Flex** - table-driven - outputs to C, C++ - distributed under BSD license
- **JLex** - table-driven - outputs to Java - distributed under GPL license
- **ALex** - table-driven - outputs to Haskell - distributed under BSD license
- **GLA** - direct-coded - outputs to C
- **re2c** - direct-coded - outputs to C - in the public domain

- **COCO/R** - direct-coded/hand-coded - outputs to C, C++, Java and others (is also a Parser Generator) - distributed under GPL license
- **ANTLR 3** - outputs to C, C++, Java and others (is also a Parser Generator) - distributed under BSD license
- **ANTLR 4** - outputs to C#, Java, Python and JavaScript (is also a Parser Generator) - distributed under BSD license

2.5.2 Parser generators

- **YACC** - LALR(1) - outputs to C
- **YACC++** - LR(1), LALR(1) - outputs to C++ and C# - distributed under proprietary license
- **bison** - LALR(1) - outputs to C and C++ - distributed under GPL license
- **Bison++** - LALR(1) - outputs to C++ - distributed under GPL license
- **COCO/R** - LL(k) - outputs to C, C++, Java and others (is also a Lexer Generator) - distributed under GPL license
- **ANTLR 3** - LL(*) - outputs to C, C++, Java and others (is also a Lexer Generator) - distributed under BSD license
- **ANTLR 4** - LL(*) - outputs to C#, Java, Python and JavaScript (is also a Parser Generator) - distributed under BSD license

Chapter 3

The IEC 61131-3 standard

The main assignment of this chapter is to review the IEC 61131-3 standard, starting with a brief introduction to the IEC 61131 series of standards. Afterwords the three versions of the IEC 61131-3 standard will be compared.

3.1 Introduction to the IEC 61131 standards

The International Electrotechnical Commission is an organization that intends to produce International Standards related to all Electrical and Electronic technologies in order to promote international co-operation in this field (from [1] and [19] on the 7^{th} of February of 2016).

The goal of the 61131 standards is to define a global interface to PLCs and their associated peripherals without defining, however, the PLC internal architecture ([20] and [21]). There are eight parts to this series of standards ([21]):

Part 1: Establishes the definitions and identifies the principal characteristics relevant to the selection and application of programmable controllers and their associated peripherals;

Part 2: Specifies equipment requirements and related tests for programmable controllers and their associated peripherals;

Part 3: Defines, for each of the most commonly used programming languages, major fields of application, syntactic and semantic rules, simple but complete basic sets of programming elements, applicable tests and means by which manufacturers may expand or adapt those basic sets to their own programmable controller implementations;

Part 4: Gives general overview information and application guidelines of the standard for the PLC end-user;

Part 5: Defines the communication between programmable controllers and other electronic systems;

Part 6: Is reserved;

Part 7: Defines the programming language for fuzzy control;

Part 8: Gives guidelines for the application and implementation of the programming languages defined in Part 3.

3.1.1 Structure of the standard

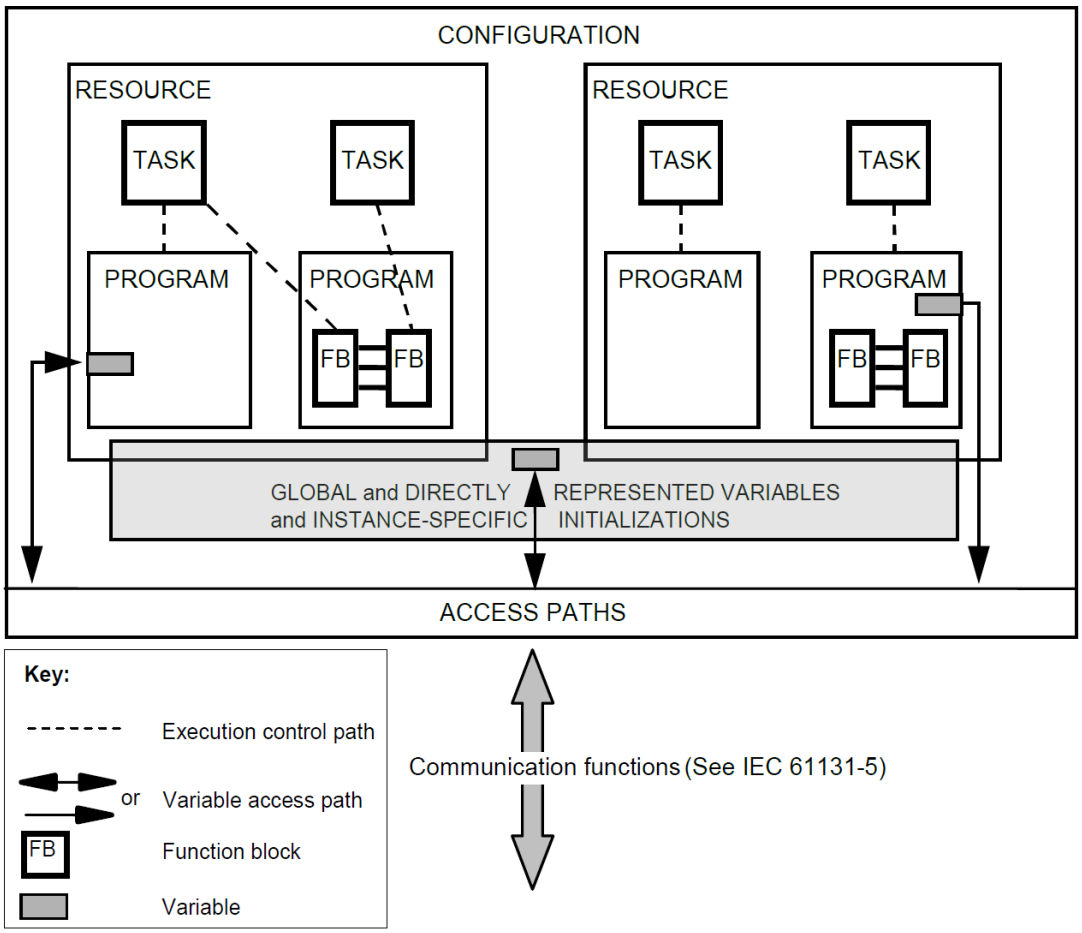

Figure 3.1: Software architectural model of the standard (from [1])

Figure 3.1 is a representation of the hierarchical programming structure defined in [1].

The highest hierarchical element in the structure is the *Configuration*. This language element corresponds to the programmable controller system defined in IEC 61131-1[22]. Here, the global variables for the configuration and the access paths between configurations are defined and directly represented variables are declared. A Configuration must contain at least one Resource.

A *Resource* is as stated in IEC 61131-3, a signal-processing-function, its man-machine interface and its sensor and actuator interface. An example of this would be a CPU in a PLC. Inside a Resource global variables (for that Resource) may be defined and directly represented variables may also be declared. A Resource always includes one ore more Programs controlled by zero or more Tasks[1]. This may seem counterintuitive but a Program may be declared in a Resource

without a Task association (a Program declared this way will have the lowest priority and will be executed cyclically[23]). Note that using the keyword `PROGRAM` in a Resource is not defining a POU (Program Organization Unit) of type Program but rather assigning a Task to trigger a POU of type program.

A *Task* is where run-time properties are defined. A Task may be executed periodically or on the rising edge of a specified boolean variable and may trigger the execution of a POU of types Program or Function Block. The properties of a Task may be changed using the parameters SINGLE (a boolean value that triggers the invocation of this Task associated Programs on its Rising Edge), INTERVAL (a time literal value which represents the time between two invocations of the Programs associated with the Task) and PRIORITY (an unsigned integer that is self explanatory).

A *POU* is either a Function, a Function Block, a Class or a Program (the next four paragraphs will succinctly describe each one of these). The purpose of a POU is to be a structural and modular part of the system that may be compiled independently from other parts of the program (although it needs the prototypes of the other POUs that are called in the being compiled one)23. It has a well defined input/output interface and may be called and executed multiple times [1].

A **Function** is a POU with no memory of its state (has no static variables) which means that for the same input, its output will always be the same.

A **Function Block** is a POU that defines a data structure and a set of methods to operate that data structure. This concept is called in the standard Function Block Type which is different from Function Block Instance. A Function Block Instance corresponds to the usage of a Function Block Type. Each instance has memory of its state. Each call to a Function Block Instance means an execution of that instance (with its previously state).

The concept of **Program** as POU is defined in IEC 61131-1 as: "logical assembly of all the programming language elements and constructs necessary for the intended signal processing required for the control of a machine or process by a PLC-system" [21]. It behaves much like a Function Block apart from a few differences[1]:
Programs must be instantiated in a Resource (Function Blocks must be instantiated in a Program or another Function Block).
A Program may have an access path to any input, output or internal variables of its own.
A directly represented variable may only be declared in a Program, a Resource or a Configuration.

A **Class** is a POU designed for OOP (object oriented programming). This POU type concept is much like the Class concept for C++, C#, Java... It differs from a Function Block because an instance of a Class does not mean an execution of that Classes methods.

3.1.2 IEC 61131-3 Languages

The standard defines two graphical programming languages (**Ladder Diagram** (LD) and **Function Block Diagram** (FBD)), two textual programming languages (**Instruction List** (IL) and **Structured Text** (ST)) and one structural language (**Sequential Function Chart** (SFC)) that may be used both in textual and graphical formats although the former is rarely seen.

The Ladder Diagram programming language was developed to look like a relay system in order to ease its usage to electricians without programming experience, thus a program written in LD contains *contacts* and/or *coils* connected in series or parallel in horizontal lines (LD networks) connecting two vertical lines (Power rails).

Contacts are elements that impose a state to their right link in the LD network. This state is determined by the state of their right link and the boolean variable (may be an input, an output or even a memory variable) associated with them. Standard contacts are Normally open contact, Normally closed contact, Positive transition-sensing contact, Negative transition-sensing contact, Compare contact (typed) and Compare contact (overloaded). For an explanation of each one of these please see [1].

On the other hand, Coils transmit the state from their right link to their left without modifying it (as in electric circuits) and store a given function of that state in their associated boolean variable. Standard coils are Coil, Negated coil, Set (latch) coil, Reset (unlatch) coil Positive transition-sensing coil and Negative transition-sensing coil. For an explanation of each one of these please see [1].

An example of a LD program may be found in Figure 3.2.

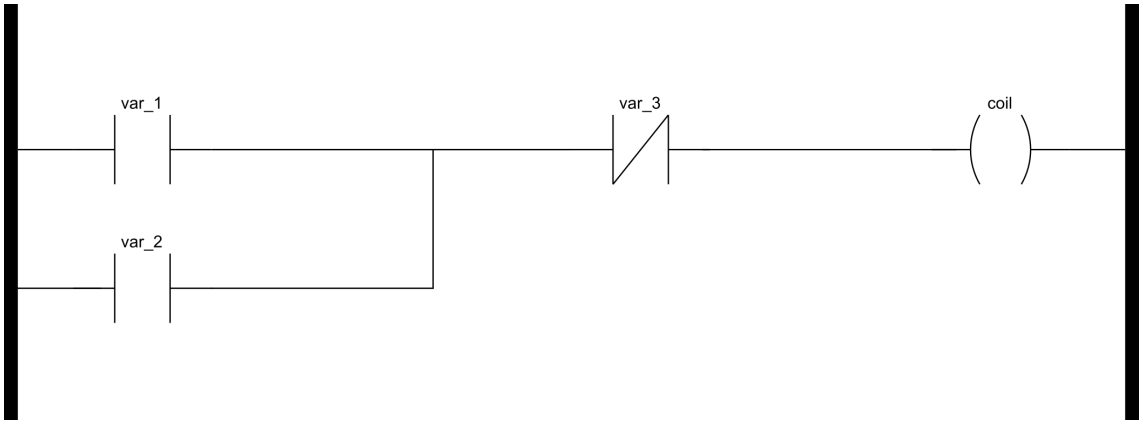

Figure 3.2: Example of LD program

An FBD program is a high level graphical programming language that uses boxes and connections between these boxes to represent Function Blocks/Functions and data connections between these entities. The inputs and outputs of a Function Block/Function are represented respectively on the left and on the right side of the corresponding box. This language resembles a signal processing system.

There are important rules to take into account when programming in FBD[1]: No element of a network shall be evaluated until the states of all of its inputs have been evaluated. The evaluation of a network element shall not be complete until the states of all of its outputs have been evaluated. The evaluation of a network is not complete until the outputs of all of its elements have been evaluated, even if the network contains one of the execution control elements.

Also, in case of a feedback loop there mus exist an implementer specific way to determine the order of execution. In Figure 3.3 the same example as in Figure 3.2 may be seen, this time as an FBD program.

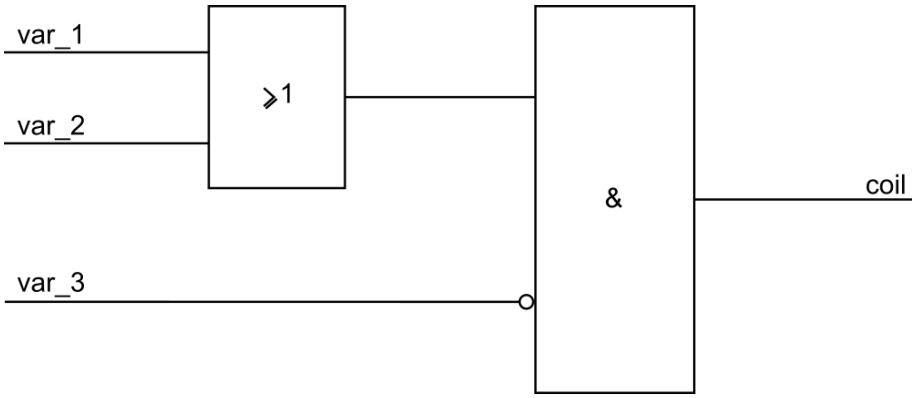

Figure 3.3: Example of Figure 3.2 as an FBD program

The IL language is an Assembler like language that is deprecated and will not appear in subsequent versions of the IEC 61131-3 standard [1]. As its name suggests, aprogram written in this language is a list of instructions. Each new instruction must start in a new line. Instructions are formed with an Operator (with optional modifiers) and one or more Operands separated by `<,>`. The result of the operation replaces the previous result (`result := result OPERATOR OPERAND`)[1]. The accepted Modifiers `N` (which negates the result), `C` (which means the instruction shall only be performed if the previous result is Boolean 1 (or 0 if the `N` modifier is also being used)) and `(` which gives priority to the operation inside it. For a list of the standard instructions and which modifiers they may use please refer to [1, Section 7.2]. It is important to notice that an IL program may call a Function Block with the Operator `CAL` and the name of the Function Block Instance (with the appropriate parameters).

The same example of Figures 3.3 and 3.2 is now expressed in IL in Code Section 3.1.

Code Section 3.1: Example of Figure 3.2 as an IL program

```
1 LD    var_1   (* load var_1: result equal to var_1 *)
  OR    var_2   (* result equal to var_1 OR var_2 *)
3 ANDN  var_3   (* result equal to result AND NOT var_3 *)
  ST    coil    (* coil equal to current result *)
```

The last programming language defined in IEC 61131-3 is the Structured Text language. It is an imperative and procedural programming language derived from PASCAL. Simply stating the ST language is a sequence of statements terminated with `;`. It supports assignment statements (`A := 1;`), call statements (to Functions, Function Blocks Instances or methods (Class methods or Function Block methods)), selection statements (such as `IF`/`ELSIF`/`ELSE` and `CASE`), iteration statements (such as `FOR`, `WHILE`, `REPEAT`, `CONTINUE` and `EXIT`) and even empty statements (`;`).

The example of Figures 3.3 and 3.2 and Code Section 3.1 may be found in ST in code Section 3.2.

Code Section 3.2: Example of Figure 3.2 as an ST program

```
coil := (var_1 OR var_2) AND NOT var_3;
```

The structural language defined in the standard, SFC, may appear, as previously stated, both in its textual version as in its graphical version. It was developed to turn complex projects into smaller and less complex bits. It is based in a previous IEC standard, the IEC 848 which derived from the Grafcet language.

The main elements of this language are Steps and Transitions which are used alternately. Steps are blocks associated with executable code (written in one of the referenced programming languages) that is executed while the step is in its active state. A transition is simply a boolean expression. A Step transits to an Inactive state when it is Active and the immediately succeeding transition is true. This event also activates the step immediately after this transition.

The flow of the program is generally from top to bottom. The only exception to this rule is in the case of a loop where an arrow may appear from the bottom of the last step of the loop to the top of the first.

There is a special Step to mark the beginning of the SFC network. This is called an Initial step and is represented by a double line square instead of a normal square as the other Steps.

Alternative ways are represented by a single horizontal line with the several alternative transitions vertically below it.

Two steps may be active simultaneously. This is represented by a double horizontal line (linking the several steps) below the respective transition.

An example of a graphical SFC network may be found in Figure 3.4. Notice that `step_2` and `step_3` are simultaneously activated if `step_1` is active and `transition_2` is true. Also `step_4` and `step_5` are alternatives to each other (in the particular case of both `transition_3` and `transition_4` are true while `step_3` is active, which step is chosen is implementer specific, therefore it is highly recommended that this type of transitions be mutually exclusive). Finally `step_7` becomes active if both `step_2` and `step_6` are active and `transition_7` is true.

The textual version of the program may be seen in Code Section 3.3.

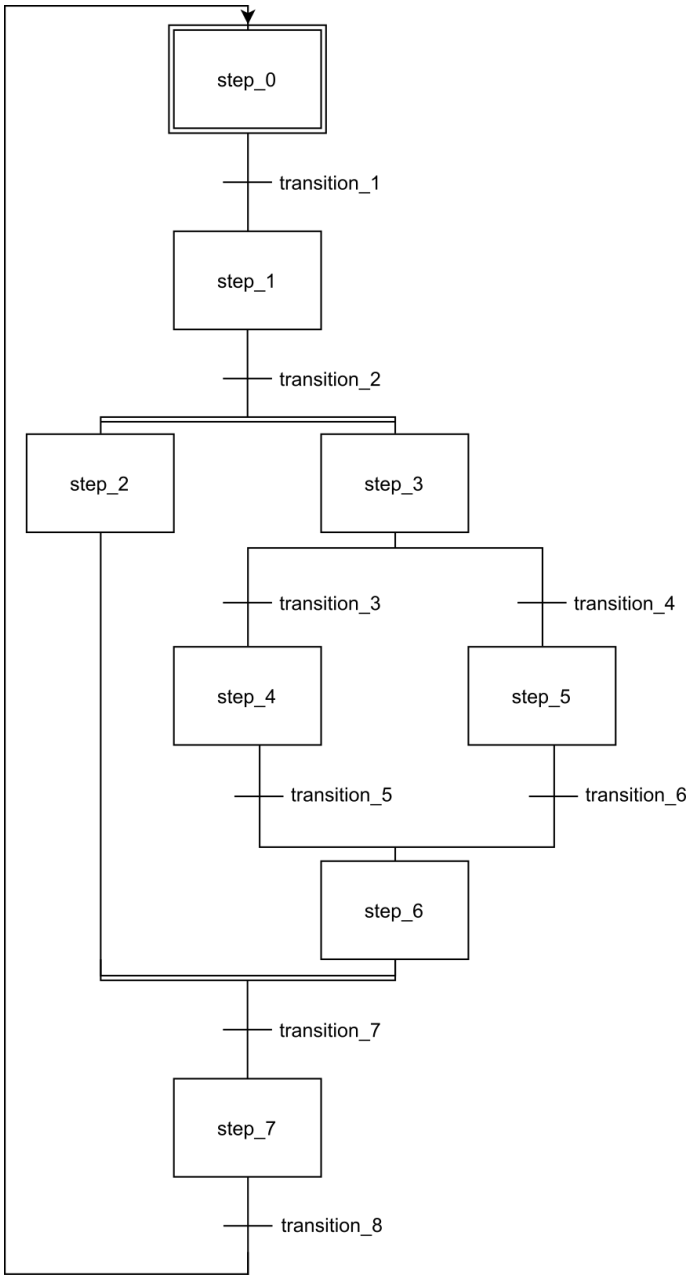

Figure 3.4: Example of a graphical SFC network

Code Section 3.3: Example of Figure 3.4 using SFC textual syntax

```
1 INITIAL_STEP step_0:
      (* code for step_0 *)
3 END_STEP
  STEP step_1:
5     (* code for step_1 *)
  END_STEP
```

```
7  (* declaration of the other steps *)

9  TRANSITION FROM step_0 TO step_1 :=
       (* condition for transition_1 *); END_TRANSITION
11 TRANSITION FROM step_1 TO (step_2, step_3) :=
       (* condition for transition_2 *); END_TRANSITION
13 TRANSITION FROM (step_2, step_6) TO step_7 :=
       (* condition for transition_7 *); END_TRANSITION
15 TRANSITION FROM step_7 TO step_0 :=
       (* condition for transition_8 *); END_TRANSITION
17 (* declaration of other transitions *)
```

3.2 The 3 versions of the IEC 61131-3 standard

3.2.1 The First Version - 1993

The first version of the standard was released on the 22^{nd} of March of 1993.

3.2.2 The Second Version - 2003

The second version of the standard was released on the 21^{st} of January of 2003. Besides small corrections in figures, tables and even descriptions there were amendments to the following items ([24] and [25, Annex A]):

- Numeric Literals - typed literals;
- Elementary data types – double-byte strings;
- Derived data types – enumerated data types;
- Single element variables – “wild-card” direct addresses;
- Declaration – Temporary variables;
- RETAIN and NON_RETAIN Variable attributes;
- Invocations and argument lists of functions;
- Type conversion functions;
- Functions of time data types;
- Function blocks – Extended initialization facilities;
- Pulse action qualifiers;

- Action control;
- Configuration initialization.

3.2.3 The Third Version - 2013

The third version of the standard was released on the 20^{th} of February of 2013. This version is considerably different from its predecessor but it's fully compatible ([1]).

The major features that were introduced can be found in [1, Annex B] and are as follows:

- Data types with explicit layout;
- Type with named values;
- Elementary data types;
- Reference, functions and operations with reference;
- Validate partial access to ANY_BIT;
- Variable-length ARRAY;
- Initial value assignment;
- Type conversion rules: Implicit – explicit;
- Function – call rules, without function result;
- Type conversion functions of numerical, bitwise Data, etc.;
- Functions of concatenate and split of time and date;
- Functions for endianess conversion;
- Class, including method, interface, etc.;
- Object-oriented FB, including method, interface, etc.;
- Namespaces;
- Structured Text: CONTINUE, etc.;
- Ladder Diagram: Contacts for compare (typed and overloaded);
- ANNEX A - Formal specification of language elements.

The next paragraphs will briefly explain these added features.

Regarding the Elementary data types, a number of new additions took place: LTIME, LDATE, LTIME_OF_DAY (or LTOD), LDATE_AND_TIME (or LDT), CHAR and WCHAR.

References were introduced to allow access to a single variable or function block instance that is part of a structure. This is similar to C pointers, however, when an operation is performed on a reference then it happens directly to the referenced variable.

Partial access to ANY_BIT allows the user to access a single bit of a BYTE, WORD, DWORD and LWORD. Its usefulness is obvious. `BOOLEAN_VAR := BYTE_VAR.%X7` means that the seventh bit of the byte variable BYTE_VAR is assigned to the boolean variable BOOLEAN_VAR. Note that the `%X` symbol indicates the size of the access (`%X` or none represents single bit size, `%B` represents byte(8 bits) size, `%W` represents word(16 bits) size, `%D` represents double word(32 bits) size and `%L` represents long word(64 bits) size).

The introduced variable-length ARRAY variables are arrays with no specified length. These may only be used as input, output or in-out variables of a function/method or in-out variables of a function block. The standard also provides a way to handle these ARRAYs: the LOWER_BOUND and UPPER_BOUND functions. They are useful for creating functions that manipulate ARRAYs of different sizes.

The third edition of the standard defines which data type conversions must be explicit and which may be implicit (new). Figures 11 and 12 of the standard (in Section 6.6.1.6) have a really straightforward and graphical way of showing this information.

The TRUNC function for conversion from REAL to INT is deprecated and to replace it typed truncation `*_TRUNC_**` where the `*` is the type of the input variable and the `**` is the type of the output variable (ex: `REAL_TRUNC_INT(1.4)` returns the INT 1) and overloaded truncation `TRUNC_**` (ex: `TRUNC_SINT(1.6)` returns the SINT 1).

Data type conversion functions of bit data types and bit and numeric data types was added to the standard (ex: `LWORD_TO_DWORD`, `BOOL_TO_ULINT` and `SINT_TO_BYTE`).

It is now possible to create a DATE_AND_TIME variable by concatenating a DATE and a TIME_OF_DAY variables or from the required integers (YEAR, MONTH, DAY, HOUR, ...). It is also possible to split a DATE variable, a TOD variable or a DT variable in their constituents integers (YEAR, MONTH, DAY, HOUR, ...).

An interesting addition to the standard are the functions for endianess conversion. `TO_BIG_ENDIAN`, `TO_LITTLE_ENDIAN`, `BIG_ENDIAN_TO` and `LITTLE_ENDIAN_TO` may be used to convert ANY_INT and ANY_BIT with 16 or more bits, ANY_REAL, WCHAR and TIME variables.

As in some high-level languages, the IEC 61131-3's languages now allow the user to use features of OOP. The concepts of Class, Object (instance), Interface, Method, Namespace were added to the standard for a better software structure on the field of Industrial Automation. Although Function Blocks were a basic block structure that mimic some of the OOP features, the new version of the standard specifically states the differences between a Function Block and the new Class concept (acquired from such languages as C++, Java or C#)[1]: `VAR_INPUT`, `VAR_OUTPUT`, `VAR_IN_OUT` and `VAR_TEMP` are not allowed inside Classes (they may be used in methods, however). A Class has no body. It is not possible to call an instance of a Class, only methods of that instance. An example of an IEC 61131-3 Class is represented in Code Section 3.4.

Code Section 3.4: Example of a Class (taken from [1])

```
1  CLASS COUNTER
     VAR
3      CV: UINT;                    // Current value of counter
       Max: UINT:= 1000;
5    END_VAR

7    METHOD PUBLIC UP: UINT              // Method for count up by inc
       VAR_INPUT INC: UINT; END_VAR      // Increment
9      VAR_OUTPUT QU: BOOL; END_VAR      // Upper limit detection

11     IF CV <= Max - INC THEN           // Count up of current value
         CV:= CV + INC;
13       QU:= FALSE;
       ELSE                              // Upper limit reached
15       QU:= TRUE;
       END_IF
17
       UP:= CV;                          // Result of method
19   END_METHOD

21   METHOD PUBLIC UP5: UINT                 // Count up by 5
       VAR_OUTPUT QU: BOOL; END_VAR          // Upper limit reached
23     UP5:= THIS.UP(INC:= 5, QU => QU);     // Internal method call
     END_METHOD
25 END_CLASS
```

Chapter 4

ANTLR vs. COCO/R

This Chapter will focus on explaining the internal structures of the two Lexer and Parser Generators, their strengths, weaknesses and their usage.

4.1 ANTLR

The *ANother Tool for Language Recognition* (ANTLR) is a powerful parser generator created and maintained by Terrence Parr since 1989 [26]. Since its third version (ANTLR v3) it generates LL(*) parsers (contrary to the previous versions which only generated LL(k) parsers). This is one of the most important features in ANTLR.

This tool generates, for a given grammar four files:

- **lexer grammar**:
  The lexer grammar is just an intermediate file and won't be focused in here.

- **tokens file**:
  For a given grammar, a token may be defined explicitly (rules with capital letters) or implicitly (for example in the rule: `add_expr: term '+' term;` + is an implicitly defined token). The tokens file is a text file that assigns an integer value for each token in the input grammar.

- **recursive-descent lexer**:
  The recursive-descent lexer is written in the output language and contains a method for each token. These methods consume the largest possible amount of input characters that fit that token.

- **recursive-descent parser**:
  The recursive-descent parser is also written in the output language and contains a method for each rule in the input grammar. A reference to a rule translates to that rule's method call and a reference to a token translates to a call to the method `match` with the current token and the expected one as parameters.

4.1.1 ANTLR prediction algorithms

For solving alternatives in rules, ANTLR first predicts which one is right and only then proceeds. In simple cases it only needs to check a few tokens ahead. This is the LL(k) approach. This is used in many other parser generators and it is simple to understand, consider the following rule [26]:

```
decl :  'int' ID '=' INT ';'
        | 'int' ID ';'
        ;
```

in order to predict which alternative is the correct one, the parser must look three symbols ahead. The generated parser must be at least LL(3). LL(k) parser generators would generate, for the given rule code such as the one in Code Section 4.1. (Assume that the method LA_t returns the lookahead token in the specified depth and the method match returns true if the first argument is of the type specified in the second argument and false otherwise.)

Code Section 4.1: LL(3) example code

```
if (LA_t(1) == 'int' && match(LA_t(2), ID) && LA_t(3) == '=')
    { return alternative1; }
else if (LA_t(1) == 'int' && match(LA_t(2), ID) && LA_t(3) == ';')
    { return alternative2; }
else
    { error(); }
```

There are, although, situations in which LL(k) is not enough. For example, in the following rule ([26]), it doesn't matter how large k is it may not be large enough.

```
decl :  modifier* 'int' ID '=' INT ';'
        | modifier* 'int' ID ';'
        ;
```

As we added a sequence of zero or more modifiers to each alternative it is impossible to say how large the k has to be in order to correctly parse the rule. ANTLR also has a way of dealing with these non-LL(k) grammars: the LL(*). For the previous example ANTLR would generate something like the code in Code Section 4.2.

Code Section 4.2: LL(*) example code

```
while(match(LA_t(k), modifier)) k++;
if (LA_t(k)=='int' && match(LA_t(k+1), ID) && LA_t(k+2)=='=')
    { return alternative1; }
else if (LA_t(k)=='int' && match(LA_t(k+1), ID) && LA_t(k+2)==';')
    { return alternative2; }
else
    { error(); }
```

In this situation, to correctly predict which alternative is the correct one, ANTLR uses a combination of while and if statements such that it arrives at the first token that can distinguish the alternatives. This technique is what makes ANTLR such an appealing tool for the compiler designer.

Despite this, ANTLR can't handle recursive rules because it uses a DFA to scan ahead. This means that it tries to find the sequence of tokens that would get it to an acceptance state. For the following rule it would get stuck as shown in the Figure 4.1:

```
decl :        'int' declarator '=' INT ';'
              | 'int' declarator ';'
              ;
declarator :  ID
              | '*' declarator
              ;
```

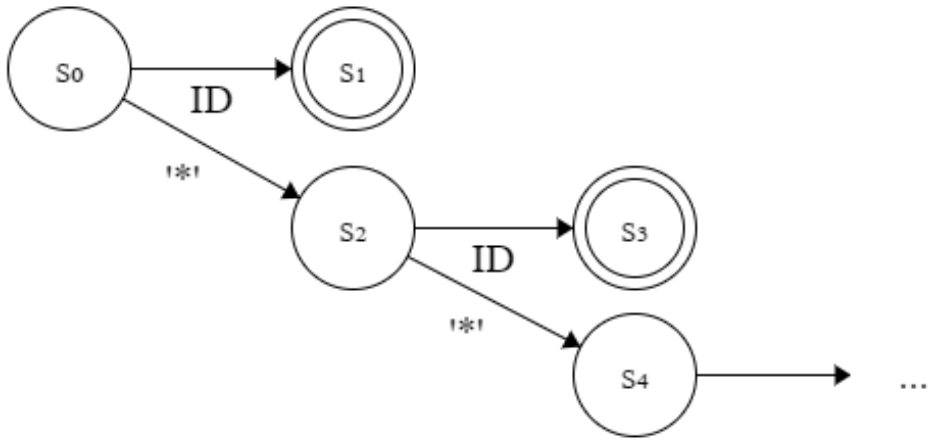

Figure 4.1: DFA for the declarator rule in the previous example

For these situations that can't be predicted with any of the previous algorithms, ANTLR has another solution which is *backtracking*: It chooses the first alternative and if it fails the parser restores to the starting state and tries the next one. This solution is more time consuming. Backtracking is not, although being described in this subsection, a prediction algorithm but rather a way to go back to a previous state and try again. This technique takes longer and is less elegant than the prediction methods and may difficult the error generation process because if all the alternatives lead to an error then the parser as to figure out which of those the user intended by choosing the "most correct" one, but even this may not be the real error.

4.1.2 AST generation

This tool also offers two straightforward ways for AST generation. One consists on telling what is to be considered root by appending a ^. The other way consists in rewriting the rule to fill the needs of the user by using the -> operator. Consider the simple example of Code Section 4.3.

Code Section 4.3: Example of ANTLR grammar for AST generation

```
grammar ANTLR_AST_example;

options {
    output=AST; language=Java;
}
program :   (ID ':='^ expr) | if_expr;
if_expr :   ('IF' (a=expr) 'THEN' (b=expr) ('ELSIF' (c+=expr) 'THEN' (d+=
    expr))* ('ELSE' (e=expr))? 'END_IF')   -> ^('IF' $a $b ^('ELSIF' $c $d)
    * ^('ELSE' $e)?);
expr    :   term (('+' | '-')^ term)*;
term    :   factor (('*' | '/')^ factor)*;
factor  :   Integer
            | '(' expr ')'      -> expr;
Integer :   ('0'..'9')('0'..'9')*;
ID      :   ('A'..'Z'|'_') ('A'..'Z'|'0'..'9'|'_')*;
```

If the input for such example would be `VARIABLE := 1` the AST would simply be the token <:=> as the root node and the leaf nodes would be <VARIABLE> and `expr` which would have a sub-tree of `term` - `factor` - <1> as leaf. A more complex example would be:

```
IF (1+5*8) THEN 1+2
ELSIF 1 THEN 0
ELSE 2 END_IF
```

The AST for this example may be seen in Figure 4.2.

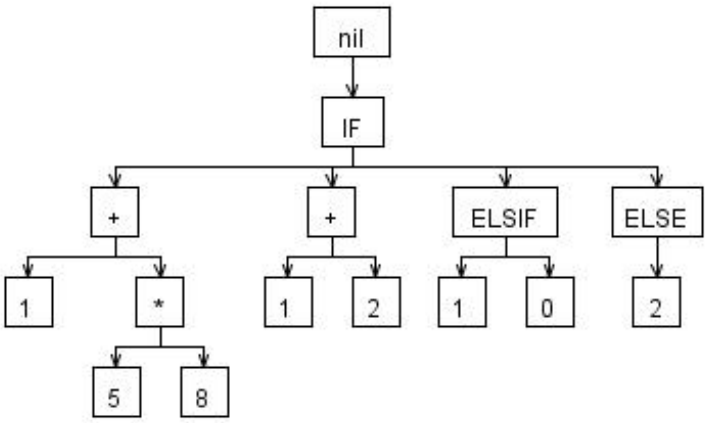

Figure 4.2: AST for the input and ANTLR Grammar described in Code Section 4.3

4.2 COCO/R

The *COmpiler COmpiler generating Recursive descent parsers* (COCO/R) is a scanner and parser generator created in 1990 by Hanspeter Mössenböck [27]. It is distributed under the GNU General Public Licence and generates, for an input LL(k) grammar, a scanner which works as a DFA and a recursive descent parser.

COCO/R is structured in five data structures [28]: The symbol table, the syntax graph, the scanner automaton, the character classes and the literals table.

The **symbol table** is the table that keeps the information of all the symbols (terminals and nonterminals) and pragmas in linear sequence.

The **syntax graph** is the graph that maps a rule (production in COCO/R's notation). Every nonterminal symbol in the symbol table has a pointer for one of these structures. Consider the following example taken from [28]:

Rule: `A = (a b* c | d e* f | ) g;`

Note that the empty string is one of the alternatives inside the parentheses. This would generate the following syntax graph:

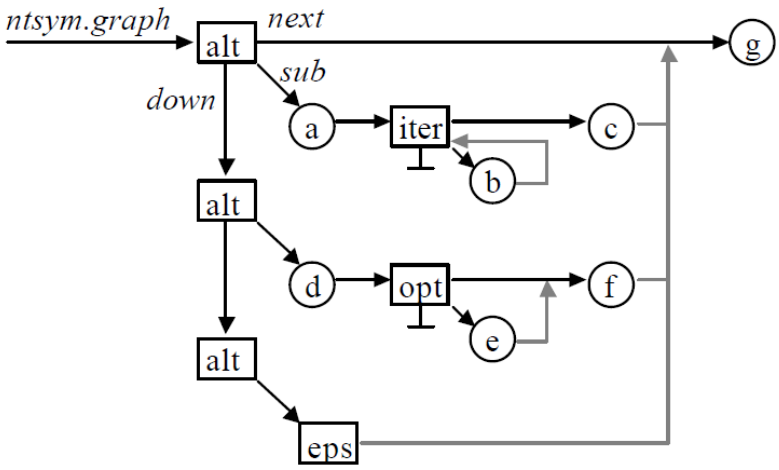

Figure 4.3: Sintax Graph for the rule A (taken from [28])

The **scanner automaton** is a DFA for Token recognition. The process for the generation of this structure starts by the creation of a syntax graph for Tokens as sequence of characters (similarly to the syntax graph for Productions/Rules as sequences of tokens) then a NFA that translates this graph is produced and latter reduced to a DFA. COCO/R's Data Structures document ([28, Section 2]) has a step by step explanation of how this process is made, however the figures for the syntax graph, the NFA, the DFA and the intermediate states are not correct considering the proposed grammar. The figures do not include the `'a'..'f'` in the the *hex* rule. Despite this error it is fairly simple to understand the process and there is no need to show the right figures in this document.

The **character classes** are the valid characters declared in the grammar. These are stored as a linear list (not to be confused with an array or a linked list).

The **literals table** is just an helper structure to map the token names with their literal representation.

In spite of the claim that it accepts LL(k) grammars, COCO/R in fact generates LL(1) parsers and provides a function to lookahead that the compiler designer may use to look further. For the LL(3) example in Section 4.1 a possible COCO/R grammar would be as shown in Code Section 4.4:

Code Section 4.4: COCO/R grammar for the LL(3) example rule in Section 4.1

```
COMPILER TestLL3

CHARACTERS
    letter      = "abcdefghijklmnopqrstuvwxyz".
    underscore  = '_'.
    digit       = "0123456789".
TOKENS
    Identifier  = (letter | (underscore (letter|digit))) {[underscore] (
        letter | digit)}.
PRODUCTIONS
    TestLL3 = decl {decl}.
    decl    =   "int" Identifier '=' Identifier ';'
                | "int" Identifier ';'.
END TestLL3.
```

COCO/R also has a prediction approach to this problem, however if no code from the compiler designer is provided, the result is just an LL(1) parser. The function for the `decl` rule is shown in Code Section 4.5.

Code Section 4.5: COCO/R generated function for `decl` rule using grammar in Code Section 4.4

```
void Parser::decl() {
    if (la->kind == 2 /* "int" */) {
        Get();
        Expect(_Identifier);
        Expect(3 /* "=" */);
        Expect(_Identifier);
        Expect(4 /* ";" */);
    } else if (la->kind == 2 /* "int" */) {
        Get();
        Expect(_Identifier);
        Expect(4 /* ";" */);
    } else SynErr(6);
}
```

To correct these situations COCO/R suggests the use of the `scanner.Peek()` method. This consists in changing the if condition to a call to a boolean function. This function must also be written by the compiler designer. In this case, the if condition was changed to a call to the `FollowedByEqual` function (See Code Section 4.6).

Code Section 4.6: COCO/R grammar of Code Section 4.4 using the `scanner.Peek()` method

```
COMPILER TestLL3_peek
bool FollowedByEqual() {
    Token x = la;
    while (x.kind == _Identifier || x.kind == _int) x = scanner.Peek();
    return x.kind == _equal;
}
CHARACTERS
    letter      = "abcdefghijklmnopqrstuvwxyz".
    underscore  = '_'.
    digit       = "0123456789".
TOKENS
    Identifier  = (letter | (underscore (letter|digit))) {[underscore] (
        letter | digit)}.
PRODUCTIONS
    TestLL3_peek    = decl {decl}.
    decl    =   IF(FollowedByEqual())
                "int" Identifier '=' Identifier ';'
                | "int" Identifier ';'.
END TestLL3_peek.
```

The resulting code is, as expected, the same with the if condition changed (Code Section 4.7).

Code Section 4.7: COCO/R generated function for `decl` rule using grammar in Code Section 4.6

```
void Parser::decl() {
    if (FollowedByEqual()) {
        Expect(2 /* "int" */);
        Expect(_Identifier);
        Expect(3 /* "=" */);
        Expect(_Identifier);
        Expect(4 /* ";" */);
    } else if (la->kind == 2 /* "int" */) {
        Get();
        Expect(_Identifier);
        Expect(4 /* ";" */);
    } else SynErr(6);
}
```

As for AST generation, COCO/R doesn't provide any help except a way for manual code writing. The delimiters of these code writing areas are `(.` and `.)`. In order to compare the two tools, the first two rules of the ANTLR grammar specified in Code Section 4.3 may be seen in COCO/R syntax in Code Section 4.8 with the same AST structure. Consider the class constructor for `ast` receives as first parameter the root and then a variable number of leaf nodes (which may be sub trees themselves). Each function generated for a rule is void thus it must return by reference. The arguments that these functions receive are specified between < and >.

Code Section 4.8: COCO/R syntax similar to `program` and `if_expr` rules of Code Section 4.3

```
COCO_AST_example              (. ast sub_tree1, sub_tree2, id; .)
=    (
         Identifier           (. id = new ast(t->val); .)
         ":="
         expr<&sub_tree2>     (. this->root = new ast(":=", id, sub_tree2);
             .)
     )
     | if_expr<&sub_tree1>    (. this->root = sub_tree1; .)
.
if_expr<ast *root>  (. ast sub_tree1, sub_tree2, else_subtree; ast_list
    elseif_list;  .)
=    "IF" expr<&sub_tree1> "THEN" expr<&sub_tree2>
     {                        (. ast sub_tree3, sub_tree4; .)
          (
              "ELSIF" (expr<&sub_tree3>) "THEN" (expr<&sub_tree4>)
          )                   (. elseif_list->add(new ast("ELSIF",
             sub_tree3, sub_tree4)); .)
     }
     [                        (. ast sub_tree5; .)
          (
              "ELSE" (expr<&sub_tree5>) (. else_subtree = new ast("ELSE",
                  sub_tree5; .)
          )
     ]
     "END_IF" (. root = new ast("IF", sub_tree1, sub_tree2, elseif_list,
         else_subtree); .)
.
```

Considering the described features, COCO/R requires much more work from the compiler designer than the previously explained tool. This is also prone to the occurrence of small typos in the code leading to errors in the generated scanner and parser.

Chapter 5

The IEC 61131-3 standard inconsistencies

5.1 The inconsistencies

During the study of the standard's languages and the exploration of the ANTLR tool the author came across a non-negligible number of inconsistencies between the main body of the standard and the formal definition of the grammar. The identified problematic sections were:

- Variable Declarations
- Constant Function Blocks
- Language Structured Text
- Character String Literals
- Duration Literals
- Comments
- Numeric Literals
- Identifiers

These inconsistencies appeared probably due to the introduced changes in the standard and the conversion of its formal syntax from BNF (Backus Naur Format) to EBNF (Extended Backus Naur Format).

In the next sections of this document those inconsistencies will be addressed and possible corrections to the formal definition of the language elements will be exposed. It is necessary to say that some minor typos were also found in the standard formal syntax (for example the `IL_Return_Operator` rule has 'RT' as a valid return operator instead of 'RET'). These were so obvious that the author decided they were not worth mention.

5.1.1 Variable Declarations

Starting with the inconsistency regarding Variable Declarations, the formal syntax of the third version of the IEC 61131-3standard doesn't accept a variables declaration structure (`VAR ... END_VAR`) which includes directly represented variables and standard variables simultaneously despite the numerous examples throughout the informal part of the standard that use this syntax (for example table 13 no. 1, table 13 no. 3, table 14 no.1, table 14 no.3, example in Section 6.5.6.2, etc.).

Directly represented variables are variables which have their physical or logical location specified by the implementer. Some examples of their syntax are:
`INP_0 AT %I0.0: BOOL;`, `AT %MW12: REAL;` or `C2 AT %Q*: BYTE;`.

Searching for all occurrences of the keyword `AT` the following rules emerge:

Formal Specification:
▷ Located_At: 'AT' Direct_Variable;
▷ Loc_Partly_Var: Variable_Name 'AT' '%' ('I' | 'Q' | 'M') '*' ':' Var_Spec ';' ;

Leaving the Partly Declared Variables aside for now and continuing to look for the occurrences of the *Located_At* rule, four rules are obtained: Declaration of Struct Elements, Declaration of Local Variables, Global Variables Specification and Instance Specific Initialization. For this verification the local variables declaration rule is the only one that matters:

Formal Specification:
▷ Loc_Var_Decl: Variable_Name ? Located_At ':' Loc_Var_Spec_Init;
which only appears in:
▷ Loc_Var_Decls: 'VAR' ('CONSTANT' | 'RETAIN' | 'NON_RETAIN')? (Loc_Var_Decl) ';')*
'END_VAR';

It is clear that it is impossible to declare located variables and non located variables within the same variables declaration structure. This is why the following example taken from table 13 [11, example no.1, table 13, Section 6.5.1.3] will never be considered a valid input:

```
VAR
    MYBIT: BOOL;
    OKAY: STRING[10];
    VALVE_POS AT %QW28: INT;
END_VAR
```

To correct the exposed problem it is only needed an additional alternative for non located variables in the rule:

Formal Specification:
▷ Loc_Var_Decls: 'VAR' ('CONSTANT' | 'RETAIN' | 'NON_RETAIN')? ((Loc_Var_Decl
| Var_Decl_Init) ';')* 'END_VAR';

Although this solution eliminates the error, this entire section of the formal syntax definition needs to be restructured. Its poor hierarchical construction favours the occurrence of inconsistencies.

5.1.2 Constant Function Blocks

In section 6.5.2.1, the standard states that "Function Block instances shall not be declared in variable sections with a `CONSTANT` qualifier." [1, Figure 7, Section 6.5.2.1]. However, the formal syntax definition allows function block instances to be declared with this modifier, as can be seen from the following 2 rules:

Formal Specification:
▷ Var_Decls: 'VAR' 'CONSTANT'? Access_Spec? (Var_Decl_Init ';')* 'END_VAR';
▷ Var_Decl_Init: Variable_List ':' (Simple_Spec_Init | Str_Var_Decl | Ref_Spec_Init)
| Array_Var_Decl_Init | Struct_Var_Decl_Init | FB_Decl_Init | Interface_Spec_Init;

To solve this inconsistency two different rules should replace the *Var_Decls* rule:

Formal Specification:
▷ Var_Decls_non_Constant: 'VAR' Access_Spec? (Var_Decl_Init ';')* 'END_VAR';
▷ Var_Decls_Constant: 'VAR' 'CONSTANT'Access_Spec? (Var_Decl_Init_no_FB ';')* 'END_VAR';
▷ Var_Decl_Init_no_FB: Variable_List ':' (Simple_Spec_Init | Str_Var_Decl | Ref_Spec_Init)
| Array_Var_Decl_Init | Struct_Var_Decl_Init | Interface_Spec_Init;

For this solution a new helper rule (*Var_Decl_Init_no_FB*)was created to simplify the code of the *Var_Decls_Constant* rule. This rule is equivalent to the *Var_Decl_Init* one when the Function Block initialization is not considered.

5.1.3 Language Structured Text

This issue may not be called an inconsistency as nothing is said in the main text about this particular case, however, it seems that there is an undesired behaviour in the standard when there is an occurrence of the *Stmt_List* rule.

Formal Specification:
▷ Stmt_List: (Stmt? ';')*;

A *Stmt_List* is formally defined, as it can be seen in the excerpt above ([1, Annex A]), as a sequence of zero or more statements. This fact has a huge influence in the *IF_Stmt*, *For_Stmt*, *Case_Stmt*, *While_Stmt* and *Repeat_Stmt* that include the *Stmt_List* rule. When the *Stmt_List* is taken as an empty list, then the program extract "`IF TRUE THEN ELSE END_IF`" may be considered as a valid *IF_Stmt*.

Formal Specification:
▷ IF_Stmt: 'IF' Expression 'THEN' Stmt_List ('ELSIF' Expression 'THEN' Stmt_List)* ('ELSE' Stmt_List)?;

The *Stmt_List* rule is also used in the declaration of the body of functions and function blocks (which in turn are used for the body of methods, actions and programs). This means that functions, function blocks, methods, actions and programs with an empty body are allowed. The main body of the standard, however, declares in section 6.6.3.2 that function block declarations must include a body. Although this text does not preclude an empty function block body, usual interpretation of this English construct leads one to conclude that the body must be non-empty. The standard does not however mandate that a function include a function body. Interestingly, version 2 of IEC 61131-3 explicitly stated that both functions and function blocks must include a body in their declaration.

Some tests were made using CODESYS V3.5 SP7 (IEC 61131-3 3rd edition compliant) software in order to check how these cases were handled (see Table 5.1). All of the previously mentioned statements generated an error when the Stmt_List rule was used as a sequence of zero statements apart from the case statement. (For the complete code used please see Table I). What is interesting is that the compiler no longer generated an error if the empty statements were replaced by a single comment, with no code whatsoever.

To fix this the *Stmt_List* rule should be reverted to what is used in version 2 of IEC61131-3, namely that it be defined as a sequence of one or more statements:

Formal Specification:
▷ Stmt_List: (Stmt? ';')+;

Table 5.1: Tests on CODESYS

IF_Stmt	IF a=0 THEN
	ELSIF a=1 THEN
	ELSE
	END_IF
		ERROR
		ERROR
		ERROR

For_Stmt	FOR a:=0 TO 20 BY 1 DO END_FOR	ERROR
While_Stmt	WHILE TRUE DO END_WHILE	ERROR
Repeat_Stmt	REPEAT UNTIL TRUE END_REPEAT	ERROR
Case_Stmt	CASE a OF 1: ELSE END_CASE	NO ERROR ERROR

It is noteworthy that the current definition also overlooks the "overflow rule":

> ""Overflow" of the most significant unit of a duration literal is permitted."

(from [1, Section 6.3.4]).

However, this rule is much easier corrected in the Semantic Analysis phase of the compiler and it is not necessary to specifically write it in the grammar.

5.1.4 Character String Literals

Tables 6 and 7 respectively define the rules of Character String Literals and Two-Character combination in character string. There are 20 rules in this table distributed by 4 groups: Single-Byte characters or characters strings with ' ' (7 rules), Double-Byte characters or characters strings with " " (7 rules), Single-Byte typed characters or string literals with # (2 rules), Double-Byte typed string literals with # (4 rules). Only 15 of these are met by the Formal specification:

Formal Specification:
▷ Char_Literal: ('STRING#')? Char_Str;
▷ Char_Str: S_Byte_Char_Str | D_Byte_Char_Str;
▷ S_Byte_Char_Str: '\" S_Byte_Char_Value+ '\';
▷ D_Byte_Char_Str: '"' D_Byte_Char_Value+ '"';
▷ S_Byte_Char_Value: Common_Char_Value | '$\" | '"' | '$' Hex_Digit Hex_Digit;
▷ Common_Char_Value: // any printable character except $, " and '

It can be clearly seen that the Char_Literal doesn't support rules 3b (`CHAR#'X'`) nor any of the rules of the 4th group (`WSTRING#"OK"`, `WCHAR#"X"`, `WSTRING#'OK'`, `WCHAR#'X'`).

Section 6.3.3 (Character String Literals) also states that a single-byte character string literal is a sequence of zero or more characters prefixed and terminated by ' and that a double-byte character string literal is a sequence of zero or more characters prefixed and terminated by ".

Solving this inconsistency is a matter of adding the missing pieces to the rule:

Formal Specification:
▷ Char_Literal: ('STRING#' | 'CHAR#' | 'WCHAR#' | 'WSTRING#')? Char_Str;

Besides these, in section 6.3.3, a single-byte character string literal is defined as "a sequence of zero or more characters prefixed and terminated by the single quote character (')" [1, Section 6.3.3] and a double-byte character string literal is defined as "a sequence of zero or more characters prefixed and terminated by the double quote character (")" [1, Section 6.3.3]. As it can be seen in the Formal Specification of S_Byte_Char_Str and D_Byte_Char_Str they are a sequence of one and more characters prefixed and terminated as stated. They should be defined as:

Formal Specification:
▷ S_Byte_Char_Str: '\" S_Byte_Char_Value* '\';
▷ D_Byte_Char_Str: '"' D_Byte_Char_Value* '"';

5.1.5 Duration Literals

The Duration Literals, specified in section 6.3.4 and in table 8, are another source of error in the standard. This section explains that these literals have the format that allows the following literal: *TIME#4d_22h_53m_25ms_95us_43.5ns*. Each unit is optional, for example TIME#6h_43m is

correct syntax. The most significant unit may overflow (TIME#4d_25h is incorrect), and only least significant unit is allowed to include a decimal point (TIME#6.5h_3m is incorrect).

The formal specification of duration literals in annex A specifies that the non-terminal elements *Days*, *Hours*, *Minutes*, *Seconds*, *Milliseconds* and *Microseconds* follow the same structure:

Formal Specification:
▷ Days: (Fix_Point 'd') | (Unsigned_Int 'd' '_'?)? Hours?;
▷ Hours:(Fix_Point 'h') | (Unsigned_Int 'h' '_'?)? Minutes?;
...
Description: `Fix_point followed by a character indicating the unit OR an optional sequence of an Unsigned_Int, a character indicating the unit and an optional underscore followed by an optional literal of lower rate.`

By this definition, and as the second alternative in every one of these elements is made only of optional parts, the empty input can be considered an Interval (Interval: Days | Hours | Minutes | ... | Nanoseconds;) which is part of Duration (Duration: (Time_Type_Name | 'T' | 'LT') '#' ('+' | '-')? Interval;) which is itself a Time_Literal (Time_Literal: Duration | ...;). This means that TIME# is considered a valid time literal.

This cannot be declared an inconsistency as both the informal textual description of the syntax, as well as the formal definition, both seem to allow the empty value TIME# to be considered a duration literal. The standard, however, fails to declare the semantic of this value; does this represent a value of 0ns duration, or some other value?

Comparing to version 2 of the standard, it once again can be seen that the informal text is essentially identical, having only been changed to take into account the new LTIME datatype, and to allow the us and ns time units. Nevertheless, the formal definition in version 2 is substantially different in that the empty TIME# sequence is not allowed as a duration literal, and it correctly matches the informal description.

A slightly different formal specification to that used in version 2 is also possible and perhaps simpler:

Formal Specification:
▷ Days: (Fix_Point 'd') | (Unsigned_Int 'd' '_'?) Hours?;
▷ Hours:(Fix_Point 'h') | (Unsigned_Int 'h' '_'?) Minutes?;
...
Description: `Fix_point followed by a character indicating the unit OR a sequence of an Unsigned_Int, a character indicating the unit and an optional underscore followed by an optional literal of lower rate.`

Note that none of the above mentioned alternatives to the formal description take into account the rule that only the most significant unit can overflow. Although it is possible to create a formal definition that takes this into account (the syntax definition used in the MatIEC compiler [2] is such an example), it is probably best to leave this rule to be checked by the semantic analyser.

5.1.6 Comments

Table 3 in section 6.1.5 specifies the acceptable syntax for writing comments. In this same section, the informal text clearly states that multi-line comments may be nested. In spite of this the formal specification shown below does not support this feature:

Formal Specification:
▷ Comment: '//' ~('\n' | '\r')* '\r'? '\n' {$channel=HIDDEN;}
| '(*' (options{greedy=false;}: .)* '*)' {$channel=HIDDEN;}
| '/*' (options{greedy=false;}: .)* '*/' {$channel=HIDDEN;};
Description: `A sequence of two / followed by any number of characters until the first newline (\n) OR the sequence of characters (* followed by any number of characters until the first occurence of the sequence *) OR the sequence of characters /* followed by any number of characters until the first occurence of the sequence */.`

It is interesting to note that this definition includes the text *options{greedy=false;}*. This does not belong to EBNF, and unlike all the other rules of the EBNF that were used in the formal grammar definition, its semantics is not defined in the beggining of the annex A containing the formal specification.

Having had access to a working version of the IEC 61131-3 v3 standard, it is straightforward for us to guess that this option follows the syntax allowed by the ANTLR [26] tool for generating syntax parsers, as in the working versions of IEC 61131-3 v3 it was noted that the formal syntax of annex A had been tested using this tool.

It is this option that makes the above definition support nested comments. However, since it does not use the EBNF rules for defining syntax, we propose the following alternative definition. The approach distinguishes one line comments from multi-line comments, and these from multi-line comments. Two rules are therefore added to the formal syntax specification:

Formal Specification:
▷ One_Line_Comment: '//' ~('\n' | '\r')* '\r'? '\n';
Description: `A sequence of two / followed by any number of characters until the first \n or \r\n.`

Formal Specification:
▷ Nested_par: ('(*' (Nested_par | .)*) '*)');
Description: `The sequence of characters (* followed by any number of characters until the first occurence of the sequence *) or until another Nested_par occurence.`

Formal Specification:
▷ Nested_bar: ('/*' (Nested_bar | .)*) '*/');
Description: `The sequence of characters /* followed by any number of characters until the first occurence of the sequence */ or until another Nested_bar occurence.`

5.1.7 Numeric Literals

The allowed syntax for numeric literals is defined in table 5 of section section 6.3.2. This section further clarifies that "single underscore characters "_" inserted between the digits of a numeric literal shall not be significant. No other use of underscore characters in numeric literals is allowed." [1, Section 6.3.2]

This sentence is overlooked when the formal specifications of *Binary_Int*, *Octal_Int* and *Hex_Int* appear in annex A:

Formal Specification:
▷ Binary_Int: '2#' ('_'? Bit)+;
▷ Octal_Int: '8#' ('_'? Octal_Digit)+;
▷ Hex_Int: '16#' ('_'? Hex_Digit)+;
Description: `The sequence of characters 2# folowed by at least one occurence of the structure: optional _ and Bit (Similar to the Octal_Int and Hex_Int).`

Assuming this formal specification is correct then the input "2#_1110_1000" is a valid input and should be considered a Binary_Int, which is an Int_Literal, and which in turn is a Numeric_Literal. This goes against the written description of a Numeric Literal because it allows an underscore without it being inserted between the numeric literal's digits.

It is once again interesting to note that, besides two extra sentences that clarify the situations when numeric literals may be used as bit string literals, the informal text defining numeric literals is unchanged from versions 2 and 3 of the stdandard. However, the formal specification was changed, and as a consequence an inconsistency was introduced in the standard.

In order to eliminate this inconsistency, we propose reverting to the formal definition used in version 2:

Formal Specification:
▷ Binary_Int: '2#' Bit ('_'? Bit)*;
▷ Octal_Int: '8#' Octal_Digit ('_'? Octal_Digit)*;
▷ Hex_Int: '16#' Hex_Digit ('_'? Hex_Digit)*;
Description: `The sequence of characters 2# folowed by a Bit followed by zero or more structures as: optional _ and Bit (Similar to the Octal_Int and Hex_Int).`

5.1.8 Identifiers

Tables 1 (in section 6.1.1 - *Character Set*) and 2 (in section 6.1.2 *Identifiers*) of [1, Annex A] respectively specify the character sets and the acceptable sequence of characters that may compose the identifier lexical element. Here, an Identifier is defined as a sequence of letters, digits and underscores that match the following the rules:

- An Identifier must not start with a Digit

- Multiple consecutive leading underscores are not allowed
- Multiple consecutive embedded underscores are not allowed
- Trailing underscores are not allowed

For example, *ABC*, *_ABC*, *A_BC*, *A_B_C*, *A123*, *_123*, and *_1_2_3* are all legal and distinct identifiers, whereas *1ABC*, *__ABC*, *A__BC*, and *ABC_* are not considered identifiers. Reading this informal text, it is not absolutely clear whether the single underscore _ should be considered a legal identifier. The only rule that may potentially invalidate it as an identifier is rule 4, as it may itself be considered a trailing underscore.

The formal specification of the *Letter* and *Identifier* lexical elements, however, only enforces the first rule to met:

Formal Specification: ▷ Letter: 'A'..'Z' | '_';
Description: `Any character between A and Z or the _ character.`

Formal Specification: ▷ Identifier: Letter (Letter | Digit)*;
Description: `A character from Letter followed by a sequence of 0 or more characters from Letter or Digit.`

According to this specification, the following sequence of characters are all considered legal and distinct *Identifiers*: __, ___, *__abc*, *a__bc*, *abc_* and *abc__*, even though they all fail one of the 4 above listed rules. This specification also allows the single underscore _ to be considered an identifier, even though it may be argued that it fails rule 4.

Comparing v2 and v3 of the standard, it is interesting to note that, other than changing the word *underline* to *underscore*, the informal definition of the *Identifier* lexical element has not suffered any changes. However, the formal definition is different in the fact that it does not allow consecutive or trailing underscores. Taking into account that it seems that there was no intention of introducing changes to the definition of *Identifier* in version 3, the obvious solution is to adopt the formal definition from version 2.

Formal Specification: ▷ Letter: 'A'..'Z';
Description: `Any character between A and Z.`

Formal Specification:
▷ Identifier: (Letter | ('_' (Letter | Digit))) ('_'? (Letter | Digit))*;
Description: `A Letter or, the _ character followed by a Letter or a Digit. Followed by a sequence of 0 or more characters from Letter or Digit, with an optional leading _ character for each one.`

It is interesting to note that according to this definition the single underscore is not considered a legal identifier.

5.2 Written Article for the IEEE ETFA'2016 conference

The 21st IEEE international conference on Emerging Technologies and Factory Automation is an international conference that took place in Berlin from the 6th to the 9th of September of 2016. It was organized by the Fraunhofer IOSB-INA and inIT- Institute Industrial IT. This conference goes back to 1992 and, apart from a few years, it is held annually. It usually has over 200 attendees (even in its first editions).

The paper focuses on the previously described inconsistencies, was presented in session *Information Technology in Automation* and can be found in the following url: http://ieeexplore.ieee.org/document/7733649/.

Chapter 6

The Lexical and Syntactical Analysis

This is the chapter where the development of the Compiler will be detailedly explained. In a first phase the description of development method of the project, secondly it is important to report the ANTLR approach. Finally the COCO/R modifications will be stated and the results will be commented.

6.1 Development Method

The complexity of the desired results ruled out immediately the Hand-Coded option for the Lexer/Parser generators and, instead of using one tool for each of the generators, it was decided that a Lexer and Parser generator would be used in order to reduce the amount of time needed to learn those tools.

The first option was the ANTLR tool. Some of the reasons behind this choice were the fact that the standard was tested with this tool (this is stated in the document 65B/807/CDV of 2011-05-03 which is a draft version of the standard)[29], some productions included configuration options for this tool (comments) and it generates LL(*) parsers. Several Lexical and Syntactic analysers were generated to test the tool's capabilities. The same analysers were later generated including AST. The Syntax Diagram and the Debug tools in ANTLRWorks were used throughout this phase to validate the results as they offer respectively a graphical DFA for each of the selected productions and a good AST display for the desired tests.

The COCO/R generator was secondly chosen due to the fact that the ANTLR's C++ target didn't support the TreeParser feature (for AST generation) that the Java target did. This tool is, unfortunately, unable to generate LL(*) parsers as stated in Chapter 4. The same test analysers were generator for learning purposes. COCO/R doesn't offer a debugger so the development methodology suffered some alterations: each new version introduced just one small feature to its predecessor in order to facilitate the detection of errors. These new versions were compared with the older ones side-by-side with the aid of Kdiff3 software. This method allowed the detection of situations beyond the capabilities of the tool. The GPL licensed software was modified to generate parsers able to surpass these otherwise impossible situations.

6.2 ANTLR approach

As previously stated the ANTLR tool was the first choice for parser generator. In order to test the limits of this tool the author first generated a test grammar that accepted input as a calculator (addition, subtraction, multiplication and division operators, parenthesised expressions and integers (both positive and negative)). This first step was fairly easy and the author promptly began building the Abstract Syntax Tree for the calculator.

Code Section 6.1: ANTLR grammar for AST generation for the calculator test

```
grammar calculator;
options {
    language=Java;
    output=AST;
    backtrack=true;
}

program : expr (EOL expr)*;
expr    : term (('+' | '-')^ term)*;
term    : factor (('*' | '/')^ factor)*;
factor  : integer
        | '(' expr ')' -> expr
        ;
integer : '-'? Digit;

Digit   : ('0'..'9')+;
EOL     : '\n' | '\r\n';
```

After this, several tests were made on the generated lexer and parser using the built-in debugger in the ANTLRWorks tool. This tool has a good graphical display of the output AST. The test file had the following test expressions:

Table 6.1: Tests on the generated lexer and parser for the Calculator grammar

Test	Reason
`1+2*-3`	validate multiplication priority over addition and negative numbers.
`(1+2)*-3`	validate parenthesis priority over multiplication.
`1+(1+2*-2)--1`	validate priority inside parenthesis.
`21*(1+5)*3*(2+2)/(2*3)`	validate multiple parenthesis in an expression.
`(1+(2+3))+2`	validate parenthesis inside parenthesis
`((2+5*3+-1)-(1*(2+3)-6)+8)/91`	validate complex expression with every feature
`((1+1)`	validate syntax error
`1---2`	validate another syntax error

All of the results for the tests in Table 6.1 were as expected. A graphical representation of the resulting AST for each test can be seen in Appendix A.

This grammar grew to accept if statements, for statements, comments and assignment expressions. The reason for this improvement was to test different and slightly more complex abstract syntax trees. The resulting grammar is presented in Code Section 6.2.

Code Section 6.2: ANTLR grammar for AST generation for the calculator with extras test

```
grammar calculator_with_extras;
options {
    language=Java;
    output=AST;
    backtrack=true;
}

tokens {
    NEGATIVE;
}

program     :   valid_expr (EOS valid_expr)* EOS EOF -> valid_expr+;
valid_expr  : (Identifier ':='^ expr) | if_expr | for_expr;

if_expr  :  ('IF' (a=expr) 'THEN' (b=valid_expr) ('ELSIF' (c+=expr) 'THEN'
     (d+=valid_expr))* ('ELSE' (e=valid_expr))? 'END_IF') -> ^('IF' $a $b
    ^('ELSIF' $c $d)* ^('ELSE' $e)?);

for_expr :  ('FOR' Identifier ':=' a=expr 'TO' b=expr ('BY' c=expr)? 'DO'
    (d+=valid_expr)+ 'END_FOR') -> ^('FOR' ^(':=' Identifier $a $b $c?)
    ^('DO' $d*));

expr    :   term (('+' | '-')^ term)*;
term    :   factor (('*' | '/')^ factor)*;
factor  :   integer
            | '(' expr ')'      -> expr;
integer :   '-' Digit           -> ^(NEGATIVE Digit)
            | Digit;

Digit       : ('0'..'9')+;
Identifier  : 'A'..'Z' ('_'? ('A'..'Z'|'0'..'9'))*;

EOS         : ';';
WS          : (' ' | '\t' | '\n' | '\r\n') {$channel=HIDDEN;};

OneLineCOMMENT  : ('//' ~('\n'|'\r')* ('\n' | '\r\n')){$channel=HIDDEN;};
MultiLineNESTED : ('/*' (options{greedy=false;}:(MultiLineNESTED | .))*
       '*/'){$channel=HIDDEN;};
```

The tests on this new grammar included the previous ones (assigned to an identifier such as: `A_B`) as well as the ones on Table 6.2

Table 6.2: Tests on the generated lexer and parser for the calculator with extras grammar

Test	Reason

`IF(21+(2-3)*5) THEN`
`  IF(1+2) THEN`
`  A:=00`
`  END_IF`
`END_IF;`
	Validate Nested If Statements

`IF(01) THEN`
`  A:=01`
`ELSIF(02) THEN`
`  A:=02`
`ELSE`
`  A:=03`
`END_IF;`
	Validate If / Elsif / Else Statement

`IF(11) THEN`
`  A:=11`
`ELSIF(12) THEN`
`  A:=12`
`ELSIF 13 THEN`
`  A:=13`
`END_IF;`
	Check the AST for several Elsif in an If / Elsif Statement

`FOR I:=0 TO (5-3*1) DO`
`  A:=0`
`  B:=5`
`END_FOR;`
	Validate For Statement

In parallel with these experiments there was an effort to import the formal syntax as written in Annex A of the IEC 61131-3 standard to an ANTLR grammar as well as the study of ANTLR in order to make it produce c++ lexers and parsers instead of Java ones as in the previous examples.

6.3 COCO/R approach

The second choice for parser generator was the COCO/R. Similarly to the previous approach, the limitations of the program were tested with a Calculator grammar (Code Section 6.3).

Code Section 6.3: COCO/R grammar for AST generation for the calculator test

```
COMPILER calculator_AST
CHARACTERS
    letter      = "abcdefghijklmnopqrstuvwxyz".
    underscore  = '_'.          digit       = "0123456789".
    operators   = "+-*/".       cr          = '\r'.
    lf          = '\n'.         tab         = '\t'.
TOKENS
    Identifier  = (letter | (underscore (letter | digit))) {[underscore] (
        letter | digit)}.
    number      = (['-'] digit{digit}).     eol = cr lf | lf.
COMMENTS FROM "/*" TO "*/" NESTED
COMMENTS FROM "//" TO cr lf
PRODUCTIONS
    calculator  =
        Identifier ":="             (. Expr e; .)
        Expression<e>
        { eol (Identifier ":=" Expression<e>) }.
    op_add_sub  <Operator &op> =
        '+'                         (. op = new addOperator(); .)
        | '-'                       (. op = new subOperator(); .)
        .
    op_mult_div <Operator &op> =
        '*'                         (. op = new mulOperator(); .)
        | '/'                       (. op = new divOperator(); .)
        .
    Expression <Expr &e> =          (. Expr e1; Operator op; Expr e2; .)
        Term            <e1>
        {   op_add_sub  <op>
            Term        <e2>        (. e = new Expr(e1, op, e2); .)
        }.
    Term <Expr &e> =                (. Expr e1; Operator op; Expr e2; .)
        Factor          <e1>
        {   op_mult_div <op>
            Factor      <e2>        (. e = new Expr(e1, op, e2); .)
        }.
    Factor <Expr &e> =
        number                      (. e = new Expr(t.val); .)
        | '(' Expression<e> ')'.
END calculator_AST.
```

This code, successfully generated the desired C++ code without any errors or warnings and the author promptly began working in the IEC 61131-3 grammar.

6.3.1 COCO/R modifications

After the manual translation of the 323 rules defined in the Formal Specification of languages elements of the standard (a transcription of these rules is available in Annex A of the standard) to the equivalent COCO/R syntax as explained in Chapter 4, the first results produced several warnings. When studying these warnings, errors were encountered in the code that appeared due to the limitations of the chosen tool. For the other_var_decls rule specified below, COCO/R generates the function in Code Section 6.4.

Formal Specification:
▷ other_var_decls: retain_var_decls | no_retain_var_decls | loc_partly_var_decl;
▷ retain_var_decls: 'VAR' 'RETAIN' Access_Spec? (var_decl_init ';')* 'END_VAR';
▷ no_retain_var_decls: 'VAR' 'NON_RETAIN' Access_Spec? (var_decl_init ';')* 'END_VAR';
▷ loc_partly_var_decl: 'VAR' ('RETAIN' | 'NON_RETAIN')? loc_partly_var* 'END_VAR';

Code Section 6.4: function before COCO/R modifications (generated by COCO/R)

```
void Parser::other_var_decls() {
    if (la->kind == 141 /* "VAR" */) {
        retain_var_decls();
    } else if (la->kind == 141 /* "VAR" */) {
        no_retain_var_decls();
    } else if (la->kind == 141 /* "VAR" */) {
        loc_partly_var_decl();
    } else SynErr(399);
}
```

As COCO/R just looks to the next token to determine the alternative paths for a given production, errors such as the one in Code Section 6.4 are prone to appear. When the Parser decides that the current token fits in the production *other_var_decls* it cannot continue to the next token as its three possible paths all start with `VAR` and will eventually produce an error if the input should be interpreted as *no_retain_var_decls* or *loc_partly_var_decl*. To overcome this limitation, the program needed a way of testing the several alternatives. This was achieved with some alterations to COCO/R's source code.

6.3.1.1 COCO/R structure

COCO/R source code is composed by 49 files that had to be studied in order to understand where the program needed alterations that would produce a parser powerful enough to the standard syntax.

The Parser class (defined in files Parser.h and Parser.cpp) has three singletons that are used to interpret the source code and generate the desired parser: A singleton of the class Tab, another of the class DFA and another of the class ParserGen. The Tab class (defined in files Tab.h and Tab.cpp) has three ArrayList (defined in files ArrayList.h and ArrayList.cpp): one stores the terminal symbols (tokens), another one stores the pragmas and the last one stores the non-terminal symbols defined in the source code. While parsing the source code, COCO/R uses a singleton of the class Tab to store each new symbol in the proper ArrayList. At the same time it generates a Syntax Graph that describes the syntax for that symbol according to the specification. The Syntax Graph is composed by objects of the Class Node (defined in files Node.h and Node.cpp). There are 14 types of Nodes:

Node::t	- a terminal symbol	**Node::pr**	- a pragma
Node::nt	- a non-terminal symbol	**Node::clas**	- a character class
Node::chr	- a character	**Node::wt**	- a weak terminal symbol
Node::any	- any symbol	**Node::eps**	- the empty string
Node::sync	- a synchronization symbol	**Node::sem**	- a semantic action
Node::alt	- an alternative	**Node::iter**	- an iteration
Node::opt	- an option	**Node::rslv**	- a resolver

Each Node has a pointer to its successor (`Node *next`). Nodes of type Node::alt also use a pointer (`Node *down`) to the next alternative. Finally Nodes of type Node::alt, Node::iter and Node::opt use the pointer `Node *sub` to access the substructure that defines that alternative/iteration/option. A typical graph may be seen in Figure 4.3.

The singleton of ParserGen (defined in files ParserGen.h and ParserGen.cpp) is responsible for the parser generation. After the parsing of the entire source code, this object takes the singleton of Tab and generates the parser. It takes a skeleton of the Parser (defined in Parser.frame) to generate the static code and, for every non-terminal in the non-terminals ArrayList of the Tab object, the ParserGen checks its type and generates the appropriate code.

To better understand the simple example of the other_var_decls (see Code Section 6.4) the excerpt of COCO/R's source code that generates the code for a Node::alt type Node is presented in Code Section 6.5. This code is part of the GenCode method of the ParseGen class (ParseGen.cpp file). Note that this excerpt is inside a cycle that goes through every Node of a substructure. `p` represents the currently being analysed Node. The UseSwitch method returns true if there are 5 or more alternatives and none of those start with a resolver (in this example it returns `false`). It is clear from the code below that the *if statements* are generated inside the cycle that explores the several alternatives by using the auxiliary Node `p2` and the `Node *down` pointer (explained above).

Code Section 6.5: excerpt of the `GenCode` function from ParserGen.cpp before the modifications

```
if (p->typ == Node::alt) {
  s1 = tab->First(p);
  bool equal = Sets::Equals(s1, isChecked);
  bool useSwitch = UseSwitch(p);
  if (useSwitch) { Indent(indent); fwprintf(gen, L"switch␣(la->kind)
     ␣{\n"); }
  p2 = p;
  while (p2 != NULL) {
    s1 = tab->Expected(p2->sub, curSy);
    Indent(indent);
    if (useSwitch) { PutCaseLabels(s1); fwprintf(gen, L"{\n");
    } else if (p2 == p) { fwprintf(gen, L"if␣("); GenCond(s1, p2->
       sub); fwprintf(gen, L")␣{\n");
    } else if (p2->down == NULL && equal) { fwprintf(gen, L"}␣else␣
       {\n");
    } else { fwprintf(gen, L"}␣else␣if␣("); GenCond(s1, p2->sub);
       fwprintf(gen, L")␣{\n");
    }
    GenCode(p2->sub, indent + 1, s1);
    if (useSwitch) {
      Indent(indent); fwprintf(gen, L"\tbreak;\n");
      Indent(indent); fwprintf(gen, L"}\n");
    }
    p2 = p2->down;
  }
  Indent(indent);
  if (equal) { fwprintf(gen, L"}\n");
  } else {
    GenErrorMsg(altErr, curSy);
    if (useSwitch) {
      fwprintf(gen, L"default:␣SynErr(%d);␣break;\n", errorNr);
      Indent(indent); fwprintf(gen, L"}\n");
    } else { fwprintf(gen, L"}␣"); fwprintf(gen, L"else␣SynErr(%d);\
       n", errorNr); }
  }
}
```

6.3.1.2 Modifications

The first modification was to change the `else if` statements (as seen in Code Section 6.4) to `if` statements. This alone could not solve the problem as it could enter in all of the and produce errors for all the wrong ones even if one of them is the expected one. This was dealt with the creation of a singleton to temporarily store the generated error while testing the next alternative. If there's one way that doesn't produce an error then the error is discarded. If the previous is not true then the last error is the one that is shown to the user. This must be modified in order to provide the programmer a better bug detection interface. The error that should be shown must be the one produced in the alternative which progressed further.

It was also necessary to create a way for the Scanner to go back to the original state. For this to be achieved three functions were added to the original COCO/R code and another one had to be slightly modified. The modifications allow the scanner to provide the current token and lookahead variables to the main sequence of the program and restore them to a previous state by returning those variables to the scanner. These variables were also worked in order to become elements of a linked list and therefore the tokens already scanned don't need to be scanned again (this would be harder to implement as the scanner had to go back in the source code and not only to a previous state).

The *other_var_decls* function became, after these modifications, like the one shown in Code Section B.1 in Appendix B.

To link this section with the previous one the cycle that explores the several alternatives by using the auxiliary Node `p2` and the `Node *down` pointer is shown in Appendix B in its final form (see Code Section B.2).

The second modification concerns the optional blocks of the grammar. When one of these structures begins with the same token as what follows the optional block the parser will try the optional block and will not try the possibility of that block being unused. To exemplify this consider the `Enum_Value` rule which expects an optional `Enum_Type_Name` which is an Identifier:

Formal Specification:
▷ Enum_Value: (Enum_Type_Name '#')? Identifier;
▷ Enum_Type_Name: Identifier;

in this particular case, the COCO/R tool generated the code provided in Code Section 6.6.

Code Section 6.6: function before COCO/R modifications (generated by COCO/R)

```
void Parser::enum_Value() {
    if (la->kind == _Identifier) {
        enum_Type_Name();
        Expect(50 /* "#" */);
    }
    Expect(_Identifier);
}
```

It is clear that the parser would generate an error if it doesn't find a # after the Identifier. These cases were overcome as previously. The parser will try the optional block and in case of error it will restore the previous state and continue the main sequence of the program allowing this portion of the code to not interfere with the rest. the resulting code for the `Enum_value` rule may be found in Code Section B.3 in Appendix B, as well as the excerpt of the `GenCode` method of the `ParserGen` class for the Nodes of the type `Node::opt` (see Code Section B.4).

The last modification is similar to the previously explained one for sequences of zero or more tokens that match the following one. Consider for example the `Enum_Type_Access` rule.

Formal Specification:
▷ Enum_Type_Access: (Namespace_Name '.')* Enum_Type_Name;
▷ Enum_Type_Name: Identifier;

The modified excerpt of the `GenCode` method of the `ParserGen` class for the Nodes of the type `Node::iter` (see Code Section B.5 in Appendix B).

To validate these alterations the following test files were created to be the input of the parser:

Code Section 6.7: ST test file provided as input to the parser

```
PROGRAM CDE
    VAR RETAIN
        ABC AT %I* : INT;
    END_VAR
    ABC := 1;
END_PROGRAM
```

For debugging purposes the code was modified so that each entry in a function increments the indentation and writes the function name and each exit of a function reduces the indentation. This produces a visual sequence of the program that was really helpful. The output before and after the modifications to COCO/R may be seen in Code sections 6.8 and 6.9 respectively.

Code Section 6.8: Parser output before COCO/R modifications

```
IEC61131---
  prog_Decl---
    prog_Type_Name---
    func_Var_Decls---
      var_Decls---
-- line 3 col 1: "VAR_IN_OUT" expected
    fB_Body---
Errors Detected: 1
```

Code Section 6.9: Parser output after COCO/R modifications

```
IEC61131---
  prog_Decl---
    prog_Type_Name---
    func_Var_Decls---
      var_Decls---
    other_Var_Decls---
      retain_Var_Decls---
        var_Decl_Init---
          variable_List---
            variable_Name---
      no_Retain_Var_Decls---
        loc_Partly_Var_Decl---
          loc_Partly_Var---
            variable_Name---
              var_Spec---
                simple_Spec---
                  elem_Type_Name---
                    numeric_Type_Name---
                      int_Type_Name---
                        Sign_Int_Type_Name---
    fB_Body---
      instruction_List---
        iL_Instruction---
          iL_Label---
          iL_Simple_Operation---
            func_Access---
              namespace_Name---
              func_Name---
                derived_Func_Name---
      stmt_List---
        stmt---
          assign_Stmt---
            variable---
              symbolic_Variable---
                namespace_Name---
                var_Access---
                  variable_Name---
            expression---
              xor_Expr---
```

```
                and_Expr---
                  compare_Expr---
                    equ_Expr---
                      add_Expr---
                        term---
                          power_Expr---
                            unary_Expr---
                              primary_Expr---
                                constant---
                                  Bool_Literal---
Errors Detected: 0
```

6.3.2 AST generation

The idea behind this project was to update the MatIEC - IEC 61131-3 compiler[2] in order to include the new IEC 61131-3 - third edition syntax. With this goal in mind, it was of the utmost interest to maximize the quantity of code from the already existing compiler. Thus is obvious that the AST for the unchanged rules should remain exactly as it was developed by De Sousa. Despite this, the amount of necessary changes and the fact that COCO/R was being used instead of the flex/bison utilities which were chosen for the MatIEC - IEC 61131-3 compiler[2] proved that the creation of the required AST was no trivial task.

The idea behind MatIEC's AST is that each node of the tree is represented by an instance of the an automatically generated class with the node type name which inherits from the major class `symbol_c`. This is accomplished by using the `#define` directive to create a macro for the declaration of generic classes (file: absyntax.hh). There are macros for classes with 0, 1, 2, 3, 4, 5, 6, 7 and 8 leaves.

The number of leaves of a node type class is defined by the complexity of the rule. For a rule that only includes fixed tokens, for this example:

Formal Specification:

```
▷Sign_Int_Type_Name:  "SINT" | "INT" | "DINT" | "LINT";
```

the macro for the 0 leaves class is called (in this case: `SYM_REF0(sint_type_name_c)`[1], ...).

For a rule that includes several other non-terminals, for example:

Formal Specification:

```
▷repeat_stmt:  "REPEAT" stmt_List "UNTIL" expression "END_REPEAT";
```

the macro for that rules number of non-terminals in its composition is called (in this case: `SYM_REF2(repeat_statement_c, statement_list, expression)`[2]).

[1] the argument for the `#define` directive is the class name.

[2] the first argument for the define directive is the class name and the next are the names of the leaves.

MatIEC uses a really interesting way to call these macros. All these calls are written in a single file (absyntax.def) that is included after the `#define` directives in file absyntax.hh. These macros are then *undefined* and *redefined* with the definition (file: absyntax.cc) of the general classes and the absyntax.def file is included once again! The modifications to the AST were, therefore, only made in the absyntax.def file.

It was always privileged to preserve the paths for existing classes and create new classes for the ones that were added than to modify existing classes. For example the rule:

Formal Specification:

```
▷enum_type_decl:  enum_type_name ':'  (([elem_type_name]named_spec_init) |
enum_spec_init);
```

previously existed as:

Formal Specification:

```
▷enumerated_type_declaration:  enumerated_type_name ':'  named_spec_init;
```

the new one was subdivided in two rules: the one that is equal to the previous version and the new one. In terms of COCO/R code, this rule looks like Code Section 6.10.

Code Section 6.10: enum_type_decl rule

```
enum_type_decl  <symbol_c **sym>    (. symbol_c *enum_tn, *elem, *nsi, *
    esi; int type_flag = 0; .)
    =
        enum_Type_Name<&enum_tn>
        ':'
        (
            (
                [
                    elem_Type_Name<&elem>   (. type_flag = 1; .)
                ]
                named_Spec_Init<&nsi>   (. *sym = new
                    enumerated_type_declaration2_c(enum_tn, ((!type_flag)?
                    NULL : elem), nsi, fl, fc, ffile, first_order, ll, lc,
                    lfile, last_order); .)  // Added to matiec -
                    enumerated_type_declaration2_c
            )
            | enum_Spec_Init<&esi>      (. *sym = new
                enumerated_type_declaration_c(enum_tn, esi, fl, fc, ffile,
                first_order, ll, lc, lfile, last_order); .)    // Equal to
                matiec's enumerated_type_declaration_c
        )
    .
```

This means that the enum_type_decl function passes by reference a `symbol_c` pointer that is either pointing to an object of `enumerated_type_declaration_c` (inherits from `symbol_c`)

if it's parsing code compliant with the second version of the standard or it is pointing to an object of `enumerated_type_declaration2_c` (inherits from `symbol_c`) if the code meets the new specification.

Another important class is the `list_c`. It also inherits from `symbol_c` and it's really useful for sequences. This class implements a dynamic array of objects of `symbol_c`. An example of its usage in COCO/R syntax is the IL_simple_inst_list rule:

Formal Specification:

```
▷IL_Simple_Inst_List:  IL_Simple_Instruction IL_Simple_Instruction;
```

The code for this rule is simpler and may be seen in Code Section 6.11.

The function passes by reference a `symbol_c` pointer that is pointing to a an object of `simple_inst_list_c` (inherits from `list_c` which inherits from `symbol_c`). The number of elements in the list is incremented each time the parser finds an IL_Simple_Instruction.

Code Section 6.11: IL_simple_inst_list rule

```
IL_simple_inst_list <symbol_c **sym>     (. symbol_c *isi; .)
    =
        iL_Simple_Instruction<&isi>      (. simple_instr_list_c *list = new
             simple_instr_list_c(); list->add_element(isi); .)
        {
            iL_Simple_Instruction<&isi> (. list->add_element(isi); .)
        }                                (. *sym = list; .)
    .
```

The last major class that also inherits from `symbol_c` is the `token_c` and it's used to store tokens such as *identifiers*, *integers/reals*, *chars* or *strings*. An example of its usage is the Signed_Int rule:

Formal Specification:

```
▷Signed_Int:  ['+' | '-']Unsigned_Int;
▷Unsigned_Int:  Digit  ['_'] Digit ;
```

The Signed_Int rule passes by reference a `symbol_c` pointer that is pointing to a an object of `integer_c` (inherits from `token_c` which inherits from `symbol_c`) or an object of `neg_integer_c` in case the parser finds a `-`. When a Unsigned_Int is found in this context a new `integer_c` is created with the value from the COCO/R current token (`t->val`).

Code Section 6.12: Signed_Int rule

```
Signed_Int <symbol_c **sym> (. symbol_c* value; int neg_flag = 0; .)
    =
        [
            '+'
            | '-'        (. neg_flag = 1; .)
        ]
        Unsigned_Int     (. value = new integer_c((const char*) t->val, fl,
            fc, ffile, first_order, ll, lc, lfile, last_order); .)
                         (. (neg_flag == 1)? *sym = new neg_integer_c(
                             value, fl, fc, ffile, first_order, ll, lc,
                             lfile, last_order) : *sym = value; .)
    .
```

For a better understanding of the previously explained structure please see the UML (Unified Modeling Language) class diagram in Figure 6.1. Note that the diagram is incomplete due to the quantity of classes that had to be represented to complete it (these missing classes are represented in gray).

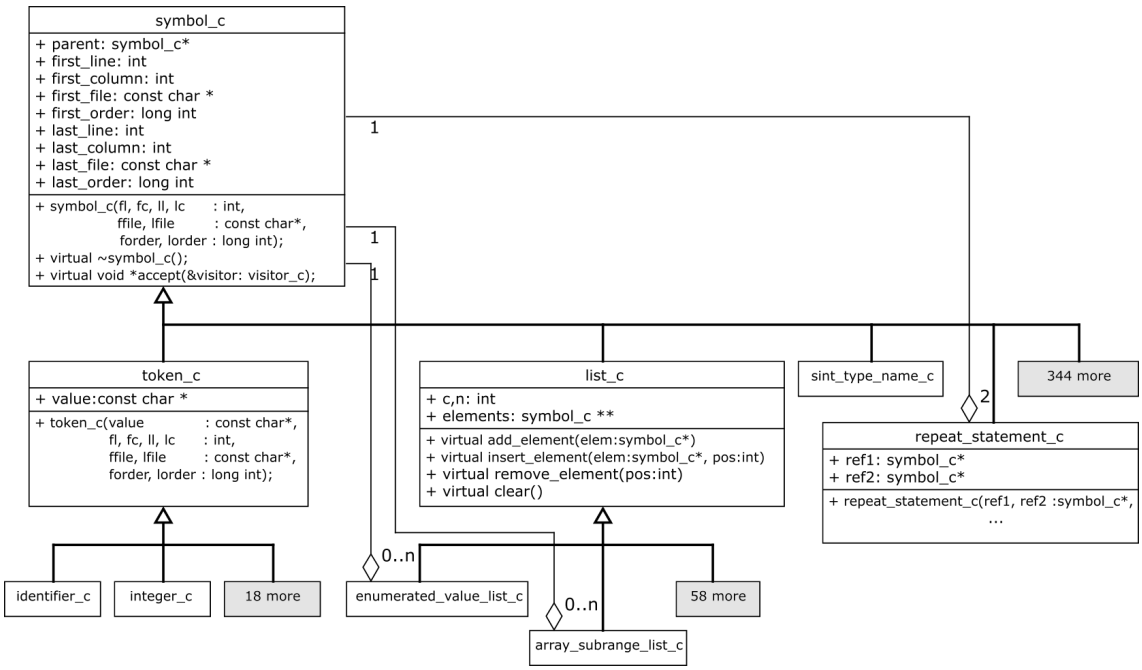

Figure 6.1: MatIEC's UML Class Diagram

A list of all the added classes as well as the modified ones may be found in Appendix C.

6.3.3 MatIEC integration and Tests

After this exhaustive process the `stage4` and `generate_iec` files from MatIEC were used to validate the AST. These functions get the AST as input and work backwards to generate code that must be similar to the Input code. A diagram of this integration may be seen in Figure 6.2.

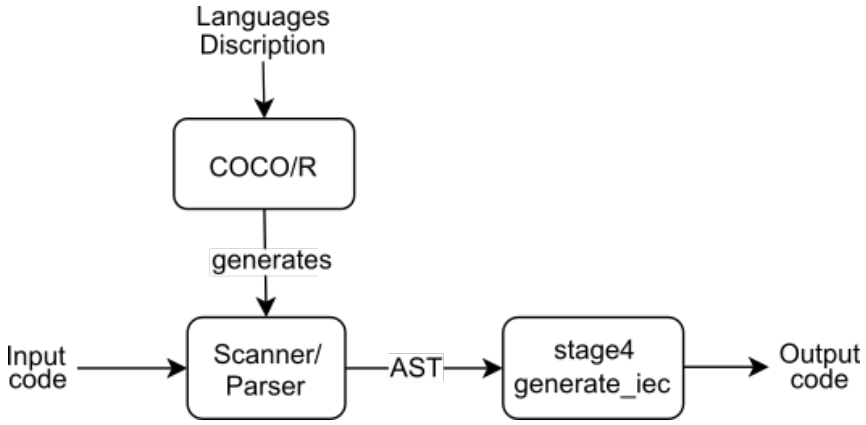

Figure 6.2: Diagram of the integration with MatIEC's stage4

The program used to test the project was the one shown in Code Section 6.13. The output of the parser is shown in Code Section 6.14.

Code Section 6.13: Test file used to validate the AST

```
1  PROGRAM TEST_NAME
       VAR RETAIN
3          ABC AT % I* : INT;
           //ABC: SINT;
5      END_VAR

7      ABC := 10+(2-3)+2;
   END_PROGRAM
```

Code Section 6.14: output of the parser for the test file of Code Section 6.13

```
   PROGRAM T
2      VAR RETAIN
           A  AT I : INT;
4      END_VAR

6      A := ((1 + (2 - 3)) + 2);
   END_PROGRAM
```

There is a clear problem with this result and that is the tokens do not totally appear in the result. A little digging in the code showed that the problem is the conversion of wide char (used by COCO/R) to char (used in matiec). Further tests were not conducted due to lack of time.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Considering the numerous inconsistencies found in the IEC 61131-3 standard it is clear that the formal specification of the languages elements deserves a major review and a restructuring.

Comparing the two tools for Lexer/Parser generation ANTLR is much more powerful than COCO/R. It also required less time to learn. In spite of this the fact that the C++ target for ANTLR was not as developed as the Java one and the need for this project to be compatible with MatIEC made COCO/R the tool of choice. This tool requires a lot more work but as it allows insertion of C++ code in a simple way there was a door for generation of an AST compatible with MatIEC.

The modifications made to COCO/R for this tool to allow backtracking are functional but the added code could be better organized.

Despite the little testing the AST results were the expected ones which means the main goal of this dissertation was achieved.

7.2 Future Work

The foremost task yet to do is to complete the `generate_iec_c` class in order to be able to test the generated parser intensively. It is also of the utmost importance to correct the minor bugs that were found while testing with the `generate_iec_c` class (the conversion from wide char (COCO/R) to char(matiec) and others that may appear when the proper tests conducted). The next step would be to generate better error messages as the ones being displayed at the moment are the COCO/R generated ones which may not be correct due to the modifications to this tool. An interesting work that the author would like to see done is the update of the COCO/R modifications to more elegant code. Lastly this project should be linked with the C++ code generation classes from matiec to make it compliant with the third edition of the IEC 61131-3 standard.

Appendix A

Results of the ANTLR approach

This appendix will show the graphical results for the AST tests done with ANTLRWorks and described in Chapter 6.

A.1 Syntax Tree resuts for the tests on the generated lexer and parser for the calculator grammar in Chapter 6

In this section the ASTs obtained for the calculator grammar (Code Section 6.1) are presented. Each subsection inside this section represents the generated AST when the grammar has one line of Table 6.1 in Chapter 6 as input.

A.1.1 Test: `1+2*-3`

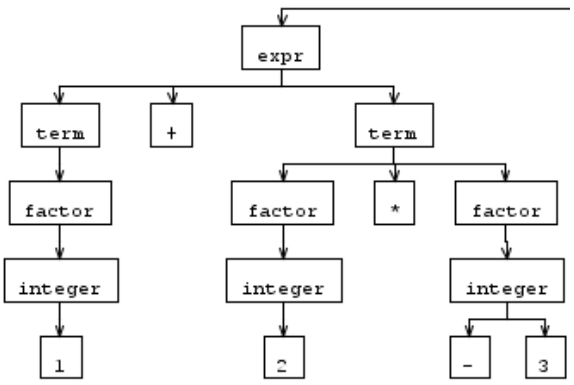

Figure A.1: AST diagram for the first test

A.1.2 Test: `(1+2)*-3`

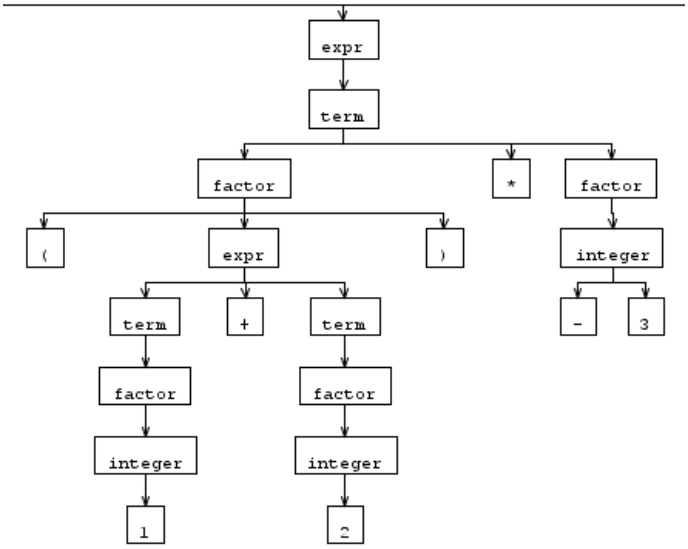

Figure A.2: AST diagram for the second test

A.1.3 Test: `1+(1+2*-2)--1`

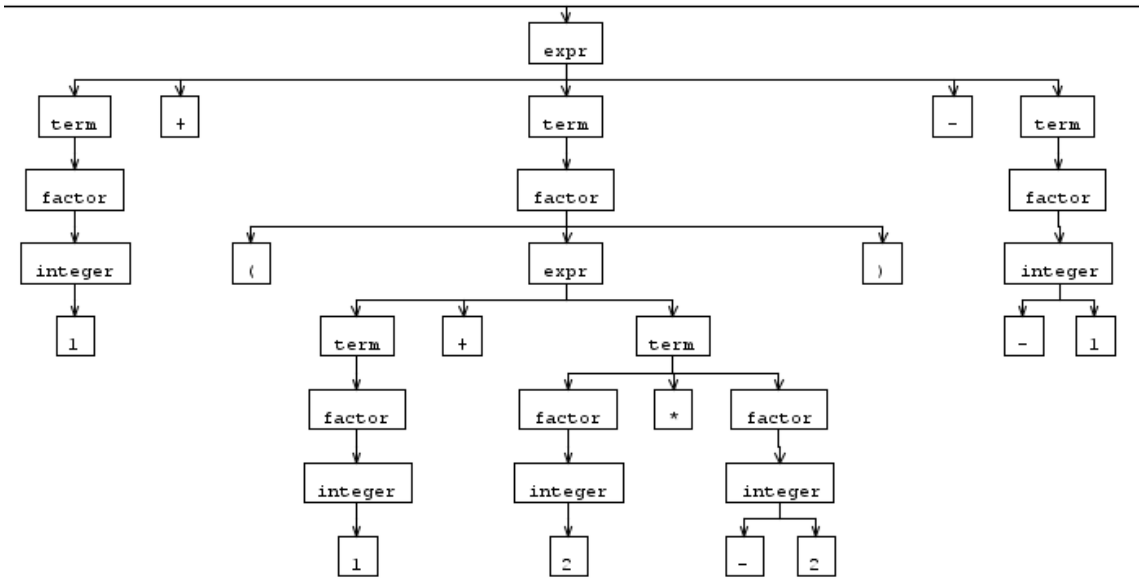

Figure A.3: AST diagram for the third test

A.1.4 Test: `21*(1+5)*3*(2+2)/(2*3)`

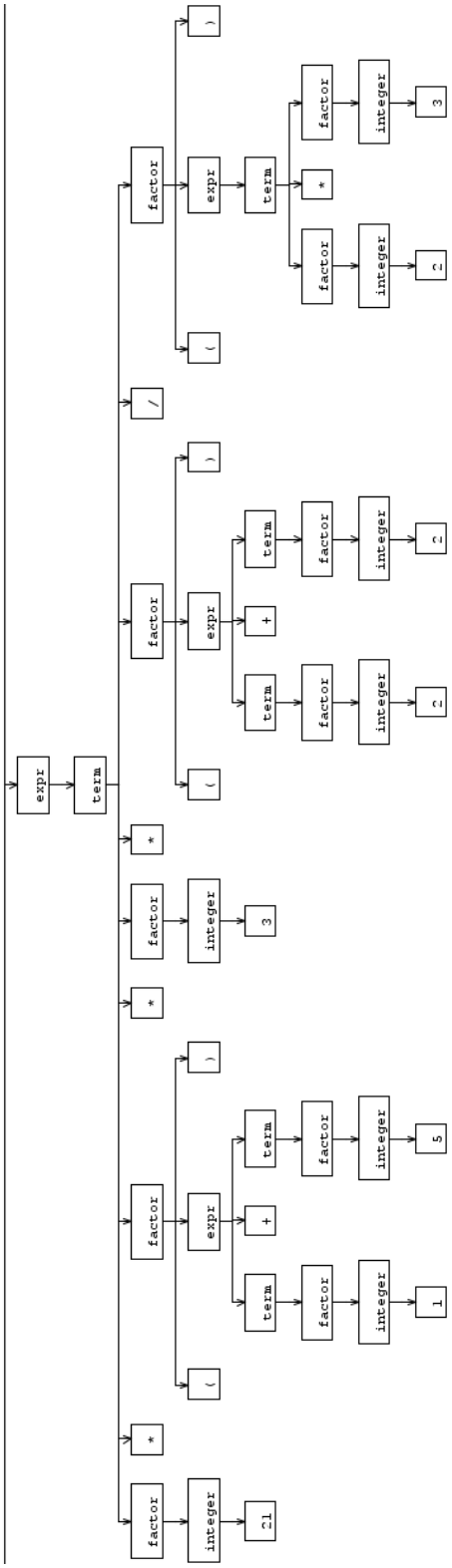

Figure A.4: AST diagram for the forth test

A.1.5 Test: `(1+(2+3))+2`

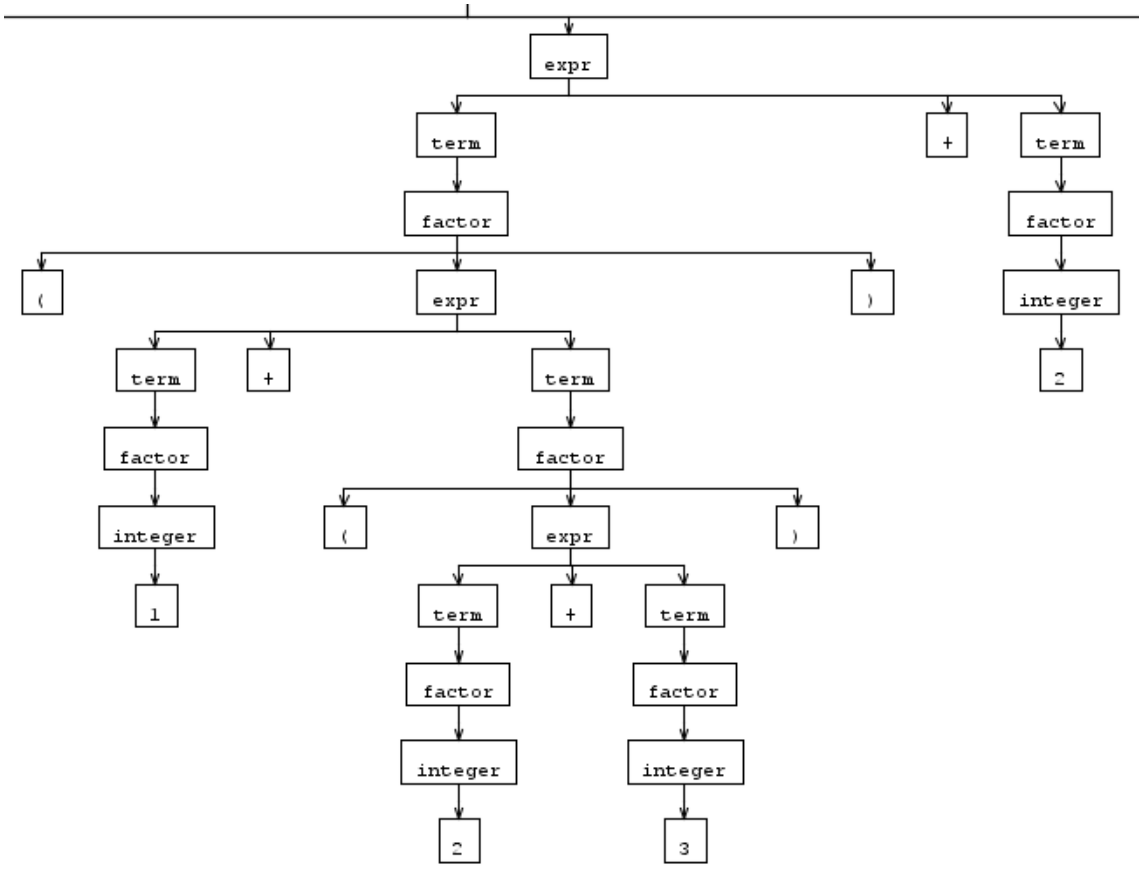

Figure A.5: AST diagram for the fifth test

A.1.6 Test: `((2+5*3+-1)-(1*(2+3)-6)+8)/91`

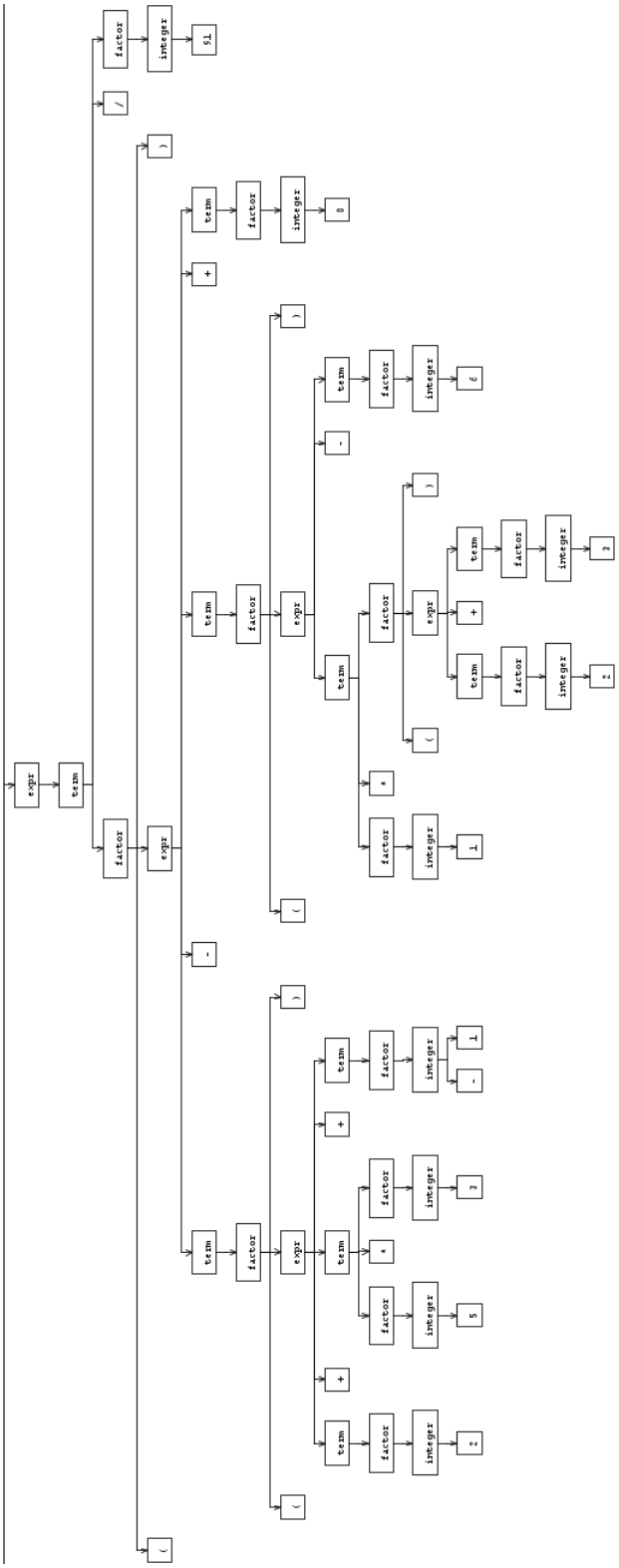

Figure A.6: AST diagram for the sixth test

A.1.7 Test: `((1+1)`

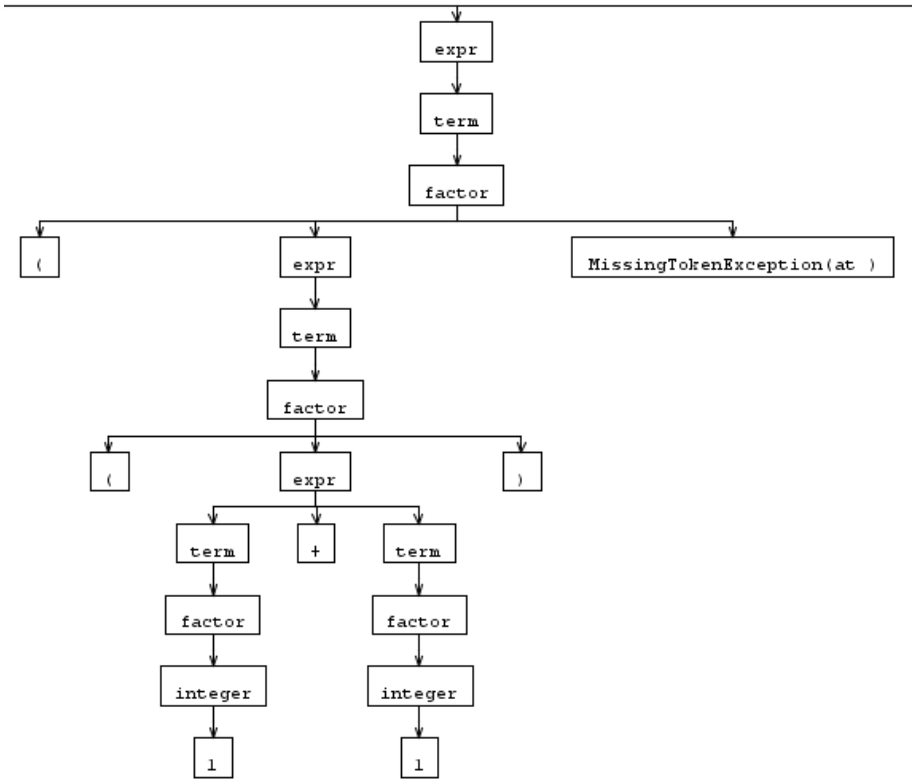

Figure A.7: AST diagram for the seventh test

A.1.8 Test: `1---2`

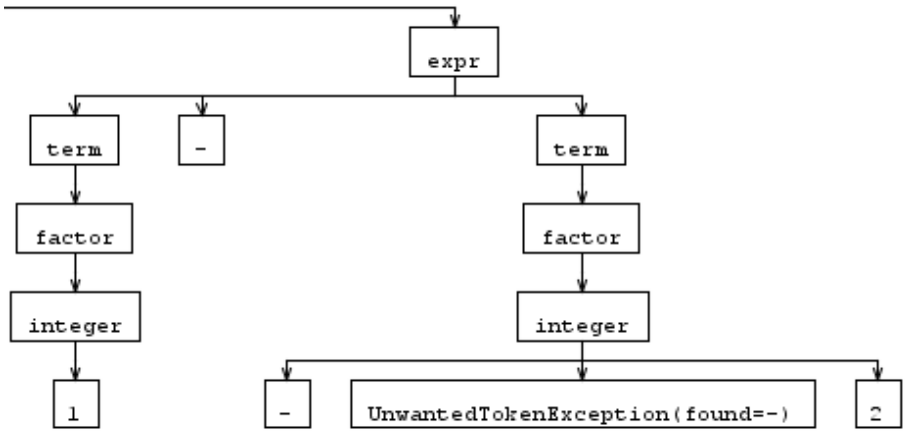

Figure A.8: AST diagram for the eight test

A.2 AST resuts for the tests on the generated lexer and parser for the calculator with extras grammar in Chapter 6

A.2.1 Test:

```
IF(21+(2-3)*5) THEN
IF(1+2) THEN A:=00 END_IF
END_IF;
```

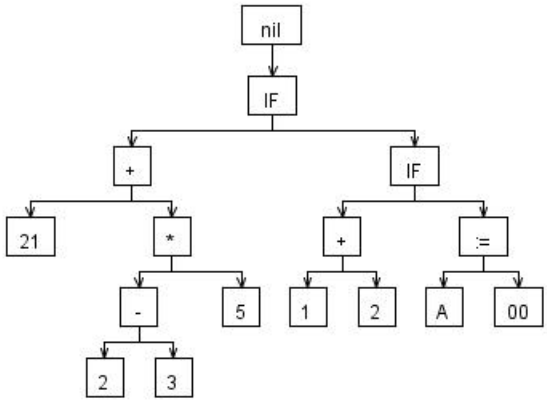

Figure A.9: AST diagram for the first test on the ANTLR Grammar on Code Section 6.2

A.2.2 Test:

```
IF(01) THEN A:=01
ELSIF(02)THEN A:=02
ELSE A:=03
END_IF;
```

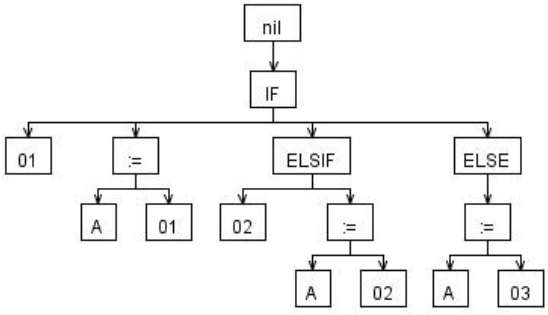

Figure A.10: AST diagram for the second test on the ANTLR Grammar on Code Section 6.2

A.2.3 Test:

```
IF(11) THEN A:=11
ELSIF(12) THEN A:=12
ELSIF 13 THEN A:=13
END_IF;
```

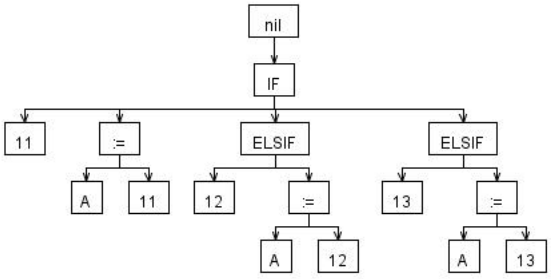

Figure A.11: AST diagram for the third test on the ANTLR Grammar on Code Section 6.2

A.2.4 Test:

```
FOR I:=0 TO (5-3*1) DO
A:=0
B:=5
END_FOR;
```

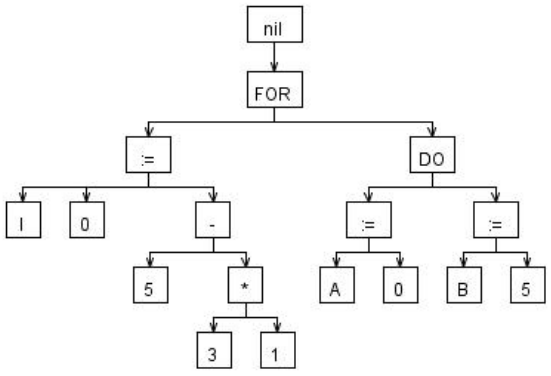

Figure A.12: AST diagram for the fourth test on the ANTLR Grammar on Code Section 6.2

Appendix B

Results after COCO/R modifications

Code Section B.1: function after COCO/R modifications (generated by COCO/R)

```
void Parser::other_var_decls() {
    int count = 0;
    bool no_error_flag = false;
    bool entry_flag = false;

    Token *pt = NULL;
    Token *tokens = NULL;
    Token *save_cur_token = NULL;
    Token *save_cur_la = NULL;
    Token *save_error_la = NULL;

    count = TempErrors::errorsCount;

    no_error_flag = false;
    pt = scanner->SavePt();
    tokens = scanner->SaveTokens();
    save_cur_token = t;
    save_cur_la = la;
    save_error_la = la;

    if ((!no_error_flag) && (la->kind == 142 /* "VAR" */)) {
        entry_flag = true;

        retain_Var_Decls();
        if(count == TempErrors::errorsCount) {
            no_error_flag = true;
            ListErrors->RemoveAllAfter(count);
        }
        else {
            save_error_la = la;
```

```
31              scanner->Restore(pt, tokens);
                t = save_cur_token;
33              la = save_cur_la;
            }
35      }

37      if ((!no_error_flag) && (la->kind == 142 /* "VAR" */)) {
            entry_flag = true;
39          ListErrors->RemoveAllAfter(count);
            count = TempErrors::errorsCount;
41
            no_Retain_Var_Decls();
43          if(count == TempErrors::errorsCount) {
                no_error_flag = true;
45              ListErrors->RemoveAllAfter(count);
            }
47          else {
                save_error_la = la;
49              scanner->Restore(pt, tokens);
                t = save_cur_token;
51              la = save_cur_la;
            }
53      }

55      if ((!no_error_flag) && (la->kind == 142 /* "VAR" */)) {
            entry_flag = true;
57          ListErrors->RemoveAllAfter(count);
            count = TempErrors::errorsCount;
59
            loc_Partly_Var_Decl();
61          if(count == TempErrors::errorsCount) {
                no_error_flag = true;
63              ListErrors->RemoveAllAfter(count);
            }
65          else {
                save_error_la = la;
67              scanner->Restore(pt, tokens);
                t = save_cur_token;
69              la = save_cur_la;
            }
71      }

73      if(!entry_flag) ListErrors->AddError(-1,-1,400);

75      if(!no_error_flag) {
```

```
        la = save_error_la;
77    }
}
```

Code Section B.2: excerpt of the `GenCode` function from ParserGen.cpp after the first modification

```
while (p2 != NULL) {
2   s1 = tab->Expected(p2->sub, curSy);

4   if (useSwitch) {
      PutCaseLabels(s1, indent); fwprintf(gen, L"␣{\n");
6   } else if (p2 == p) {
      /*Initialization Here*/
8     Indent(indent); fwprintf(gen, L"count␣=␣TempErrors::errorsCount;\n\n")
          ;
      Indent(indent); fwprintf(gen, L"no_error_flag␣=␣false;\n");
10    Indent(indent); fwprintf(gen, L"pt␣=␣scanner->SavePt();\n");
      Indent(indent); fwprintf(gen, L"tokens␣=␣scanner->SaveTokens();\n");
12    Indent(indent); fwprintf(gen, L"save_cur_token␣=␣t;\n");
      Indent(indent); fwprintf(gen, L"save_cur_la␣=␣la;\n");
14    Indent(indent); fwprintf(gen, L"save_error_la␣=␣la;\n\n");

16    Indent(indent);  fwprintf(gen, L"if␣((!no_error_flag)␣&&␣(");
      GenCond(s1, p2->sub);
18    fwprintf(gen, L"))␣{\n");
      Indent(indent+1); fwprintf(gen, L"entry_flag␣=␣true;\n\n");
20  } else if (p2->down == NULL && equal) {
      Indent(indent);
22    fwprintf(gen, L"}\n\n");
      Indent(indent); fwprintf(gen, L"if␣((!no_error_flag)␣&&␣(");
24    GenCond(s1, p2->sub);
      fwprintf(gen, L"))␣{\n");
26    Indent(indent+1); fwprintf(gen, L"entry_flag␣=␣true;\n");
      Indent(indent+1); fwprintf(gen, L"ListErrors->RemoveAllAfter(count);\n
          ");
28    Indent(indent+1); fwprintf(gen, L"count␣=␣TempErrors::errorsCount;\n\n
          ");
  } else {
30    Indent(indent);
      fwprintf(gen, L"}\n\n");
32    Indent(indent); fwprintf(gen, L"if␣((!no_error_flag)␣&&␣(");
      GenCond(s1, p2->sub);
34    fwprintf(gen, L"))␣{\n");
      Indent(indent+1); fwprintf(gen, L"entry_flag␣=␣true;\n");
```

```
36      Indent(indent+1); fwprintf(gen, L"ListErrors->RemoveAllAfter(count);\n
            ");
        Indent(indent+1); fwprintf(gen, L"count = TempErrors::errorsCount;\n\n
            ");
38    }
    GenCode(p2->sub, indent + 1, s1);
40
    if (useSwitch) {
42      Indent(indent); fwprintf(gen, L"\tbreak;\n");
        Indent(indent); fwprintf(gen, L"}\n");
44    }
    else {
46      Indent(indent+1); fwprintf(gen, L"if(count == TempErrors::errorsCount)
            {\n");
        Indent(indent+2); fwprintf(gen, L"no_error_flag = true;\n");
48      Indent(indent+2); fwprintf(gen, L"ListErrors->RemoveAllAfter(count);\n
            ");
        Indent(indent+1); fwprintf(gen, L"}\n");
50      Indent(indent+1); fwprintf(gen, L"else {\n");
        Indent(indent+2); fwprintf(gen, L"save_error_la = la;\n");
52      Indent(indent+2); fwprintf(gen, L"scanner->Restore(pt, tokens);\n");
        Indent(indent+2); fwprintf(gen, L"t = save_cur_token;\n");
54      Indent(indent+2); fwprintf(gen, L"la = save_cur_la;\n");
        Indent(indent+1); fwprintf(gen, L"}\n");
56    }
    p2 = p2->down;
58  }
```

Code Section B.3: function after COCO/R modifications (generated by COCO/R)

```
    void Parser::enum_Value(symbol_c **sym) {
2
        if (la->kind == _Identifier) {
4           int    opt_errors_count = TempErrors::errorsCount;
            Token *opt_pt           = scanner->SavePt();
6           Token *opt_tokens       = scanner->SaveTokens();
            Token *opt_save_token   = t;
8           Token *opt_save_la      = la;

10          enum_Type_Name();
            Expect(14 /* "#" */);
12
            if(opt_errors_count != TempErrors::errorsCount) {
14              scanner->Restore(opt_pt, opt_tokens);
                t = opt_save_token;
16              la = opt_save_la;
```

```
        }
    }

    Expect(_Identifier);
}
```

Code Section B.4: excerpt of the `GenCode` function from ParserGen.cpp after the second modification

```
if (p->typ == Node::opt) {
  s1 = tab->First(p->sub);
  Indent(indent);
  fwprintf(gen, L"if␣("); GenCond(s1, p->sub); fwprintf(gen, L")␣{\n");

  Indent(indent+1); fwprintf(gen, L"int␣␣␣␣opt_errors_count␣=␣TempErrors::
      errorsCount;\n");
  Indent(indent+1); fwprintf(gen, L"Token␣*opt_pt␣␣␣␣␣␣␣␣␣␣␣=␣scanner->
      SavePt();\n");      /* B - Inside an optional structure if in the end
       there is an error      */
  Indent(indent+1); fwprintf(gen, L"Token␣*opt_tokens␣␣␣␣␣␣␣=␣scanner->
      SaveTokens();\n");    /* then possibly the optional was not used.
                        */
  Indent(indent+1); fwprintf(gen, L"Token␣*opt_save_token␣␣␣=␣t;\n");
                   /* These variables are used to restore the parser to the
       initial state.    */
  Indent(indent+1); fwprintf(gen, L"Token␣*opt_save_la␣␣␣␣␣=␣la;\n");

  GenCode(p->sub, indent + 1, s1);

  Indent(indent+1); fwprintf(gen, L"if(opt_errors_count␣!=␣TempErrors::
      errorsCount)␣{\n");
  Indent(indent+2); fwprintf(gen, L"ListErrors->RemoveAllAfter(
      opt_errors_count);\n");
  Indent(indent+2); fwprintf(gen, L"scanner->Restore(opt_pt,␣opt_tokens);\
      n");
  Indent(indent+2); fwprintf(gen, L"t␣=␣opt_save_token;\n");
  Indent(indent+2); fwprintf(gen, L"la␣=␣opt_save_la;\n");
  Indent(indent+1); fwprintf(gen, L"}\n");

  Indent(indent); fwprintf(gen, L"}\n\n");
}
```

Code Section B.5: excerpt of the `GenCode` function from ParserGen.cpp after the third modification

```
if (p->typ == Node::iter) {
  Indent(indent);
  p2 = p->sub;

  fwprintf(gen, L"{\n");                        /* B - open new scope */

  /* 1. Initialize variables to backtrack */  /* B */
  Indent(indent+1); fwprintf(gen, L"int␣␣␣␣iter_errors_count␣=␣TempErrors
      ::errorsCount;\n");
  Indent(indent+1); fwprintf(gen, L"Token␣*iter_pt␣␣␣␣␣␣␣␣␣␣=␣scanner->
      SavePt();\n");
  Indent(indent+1); fwprintf(gen, L"Token␣*iter_tokens␣␣␣␣␣␣=␣scanner->
      SaveTokens();\n");
  Indent(indent+1); fwprintf(gen, L"Token␣*iter_save_token␣␣␣=␣t;\n");
  Indent(indent+1); fwprintf(gen, L"Token␣*iter_save_la␣␣␣␣␣=␣la;\n\n");

  p2 = p->sub;
  Indent(indent+1); fwprintf(gen, L"while␣(");
  if (p2->typ == Node::wt) {
    s1 = tab->Expected(p2->next, curSy);
    s2 = tab->Expected(p->next, curSy);
    fwprintf(gen, L"WeakSeparator(");
    WriteSymbolOrCode(gen, p2->sym);
    fwprintf(gen, L",%d,%d)␣", NewCondSet(s1), NewCondSet(s2));
    s1 = new BitArray(tab->terminals->Count);  // for inner structure
    if (p2->up || p2->next == NULL) p2 = NULL; else p2 = p2->next;
  } else {
    s1 = tab->First(p2);
    GenCond(s1, p2);
  }
  fwprintf(gen, L")␣{\n");
  /* 2. run code */
  GenCode(p2, indent + 2, s1);

  /* 3. Test if generated an error */  /* B */
  Indent(indent+2); fwprintf(gen, L"if(iter_errors_count␣==␣TempErrors::
      errorsCount)␣{\n");

  /* 3.NO save variables go to while */  /* B */
  Indent(indent+3); fwprintf(gen, L"iter_pt␣␣␣␣␣␣␣␣␣␣=␣scanner->SavePt()
      ;\n");
  Indent(indent+3); fwprintf(gen, L"iter_tokens␣␣␣␣␣␣=␣scanner->
      SaveTokens();\n");
```

```
38    Indent(indent+3); fwprintf(gen, L"iter_save_token␣␣␣=␣t;\n");
      Indent(indent+3); fwprintf(gen, L"iter_save_la␣␣␣␣␣␣=␣la;\n\n");
40    Indent(indent+2); fwprintf(gen, L"}\n");    /* end of if of error test
          */

42    /* 3.YES restore last saved variables and break! */  /* B */
      Indent(indent+2); fwprintf(gen, L"else␣{\n");
44    Indent(indent+3); fwprintf(gen, L"ListErrors->RemoveAllAfter(
          iter_errors_count);\n");
      Indent(indent+3); fwprintf(gen, L"scanner->Restore(iter_pt,␣iter_tokens)
          ;\n");
46    Indent(indent+3); fwprintf(gen, L"t␣=␣iter_save_token;\n");
      Indent(indent+3); fwprintf(gen, L"la␣=␣iter_save_la;\n");
48    Indent(indent+3); fwprintf(gen, L"break;\n");
      Indent(indent+2); fwprintf(gen, L"}\n");    /* end of else of error test
           */
50
      Indent(indent+1); fwprintf(gen, L"}\n\n");  /* end of while */
52    Indent(indent); fwprintf(gen, L"}\n\n");    /* end of scope */
  }
```

Appendix C

Changes to matiec's AST

C.1 List of added classes

ltime_type_name_c	ldate_type_name_c
ltod_type_name_c	ldt_type_name_c
char_type_name_c	wchar_type_name_c
namespace_name_list_c	type_access_c
enumerated_type_declaration2_c	named_spec_init_c
enumerated_value_spec_list_c	enum_value_spec_c
structure_declaration_c	overlap_option_c
ref_type_access_c	ref_address_c
ref_assign_c	ref_deref_c
deref_list_c	deref_c
ref_var_decl_c	interface_var_decl_c
array_conformand_c	star_list_c
star_c	array_conform_decl_c
fb_instance_name_c	func_access_c
std_func_name_c	fb_type_access_c
std_fb_name_c	method_decl_c
override_modifier_c	fb_input_decls_c
fb_input_decl_list_c	fb_output_decls_c
fb_output_decl_list_c	class_decl_c
final_modifier_c	abstract_modifier_c
method_decl_list_c	class_instance_name_c
interface_decl_c	method_prototype_list_c
method_prototype_c	io_var_decls_list_c
interface_spec_init_c	interface_name_list_c
public_access_spec_c	protected_access_spec_c
private_access_spec_c	internal_access_spec_c
prog_access_decls_c	prog_access_decl_c
access_decl_list_c	access_decl_c
access_path_c	access_path2_c
read_write_c	read_only_c

namespace_decl_c
internal_option_c
namespace_h_name_c
declarations_list_c
il_fb_call_super_c
no_this_call_c
instance_list_c
multibit_part_access2_c
partial_access_ommited_c
partial_access_W_c
partial_access_L_c
continue_statement_c
using_directive_list_c
namespace_elements_c
pou_decl_c
il_simple_operation2_c
this_call_c
this_instance_method_c
variable_access_c
partial_access_X_c
partial_access_B_c
partial_access_D_c
assignment_attempt_c
super_statement_c

C.2 List of modified classes

interval_c
structure_element_declaration_c
var_declarations_c
function_declaration_c
function_block_declaration_c
fb_initialization_c
enumerated_spec_init_c
symbolic_variable_c
retentive_var_declarations_c
function_var_decls_c
non_retentive_var_decls_c

Bibliography

[1] International Electrotechnical Commission. *IEC 61131-3:2013 International Standard*. IEC, 3.0 edition, 2013. Cited on pages xi, xv, 1, 23, 24, 25, 26, 27, 31, 32, 33, 45, 46, 47, and 50.

[2] Mário Jorge Rodrigues de Sousa and Adriano da Silva Carvalho. An IEC 61131-3 compiler for the MatPLC. Cited on pages 2, 48, and 64.

[3] Andrew Ferguson. The history of computer programming languages. *Princeton New Jersey: http://www. princeton. edu/˜ ferguson/adw/programming_languages*, 2000. Cited on page 3.

[4] Richard L. Wexelblat. *History of Programming Languages*. Acm Monograph Series. Academic Press, 1st edition, 1981. Cited on page 3.

[5] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Pearson Education Inc., second edition, 2006. Cited on pages xi, xiii, 3, 4, 5, 6, 7, 8, 13, 17, and 19.

[6] C N Fisher and R J L Blanc. *Crafting a Compiler*. The Benjamin-Cummings Series in Computer Science. Benjamin/Cummings, 1988. Cited on pages 5, 6, 13, 16, 19, and 20.

[7] K Cooper and L Torczon. *Engineering a Compiler*. Elsevier Science, second edition, 2011. Cited on pages xiii, 6, 7, 9, 13, and 16.

[8] Lars Marius Garshol. Bnf and ebnf: What are they and how do they work. 16, 2003. Cited on page 10.

[9] Richard Feynman. Ebnf: A notation to describe syntax. Cited on page 10.

[10] Pete Jinks. Bnf/ebnf variants, 2014. Cited on page 10.

[11] Noam Chomsky. On certain formal properties of grammars. *Information and control*, 2(2):137–167, 1959. Cited on page 11.

[12] Patrick D Terry. Compilers and compiler generators: an introduction with c++. 2000. Cited on pages 11 and 13.

[13] Larry Ullman. Introduction to automata theory, languages, and computation, 2006. Cited on page 13.

[14] Tadao Kasami. An efficient recognition and syntaxanalysis algorithm for context-free languages. Technical report, DTIC Document, 1965. Cited on page 13.

[15] Martin Lange and Hans Leiß. To cnf or not to cnf? an efficient yet presentable version of the cyk algorithm. *Informatica Didactica*, 8:2008–2010, 2009. Cited on page 13.

[16] Te Li and Devi Alagappan. A comparison of cyk and earley parsing algorithms. Cited on page 13.

[17] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970. Cited on page 13.

[18] Alex Aiken. Compilers stanford. Online Lecture, 2012. Cited on pages 16 and 19.

[19] International Electrotechnical Commission. About the IEC page, 2016. Cited on page 23.

[20] Mário Jorge Rodrigues de Sousa. *Desenvolvimento de Plataforma para Apoio de Aplicações de Controlo Industrial*. PhD thesis, Faculdade de Engenharia da Universidade do Porto, 2004. Cited on page 23.

[21] International Electrotechnical Commission. *IEC 61131-1:2003 Internation Standard*. IEC, 2.0 edition, 2003. Cited on pages 23 and 25.

[22] Dag H Hanssen. *Programmable Logic Controllers: A Practical Approach to IEC 61131-3 Using CODESYS*. John Wiley & Sons, 2015. Cited on page 24.

[23] Karl-Heinz John and Michael Tiegelkamp. *IEC 61131-3: programming industrial automation systems: concepts and programming languages, requirements for programming systems, decision-making aids*. Springer Science & Business Media, 2010. Cited on page 25.

[24] International Electrotechnical Commission. *IEC 61131-3:2003 International Standard*. IEC, 2.0 edition, 2003. Cited on page 30.

[25] International Electrotechnical Commission. *Technical Report IEC TR 61131-8*. IEC, 2.0 edition, 2003. Cited on page 30.

[26] Terrence Parr. *The Definitive ANTLR Reference*. Pragmatic Bookshelf, 2007. Cited on pages 35, 36, and 49.

[27] Hanspeter Mössenböck. *The Compiler Generator Coco / R*. 2010. Cited on page 39.

[28] Hanspeter Mössenböck. Data Structures in Coco / R. Technical Report April, 2005. Cited on pages xi and 39.

[29] International Electrotechnical Commission. Iec 61131-3 draft - document 65b/807/cdv, 2011. Cited on page 53.