Structured Text Compiler Targeting XML

Jawad Hassan
2010-06-22
Subject: Software Technology
Level: Advanced
Course code: DA4004
Abstract

IEC-61131 is an open standard defined by international electro technical commission for programmable logic controllers. This standard consists of several parts and part 3 deals with programming languages. There are two textual and two graphical languages defined in part 3 of the standard [1].

In this thesis report design and implementation of a compiler is discussed. The compiler is capable of compiling Structured Text based programs and can generate digital circuit layouts equivalent of the source programs. The digital circuits are represented by XML documents. The XML schema for circuit layouts discussed in this thesis report is developed by Sauer-Danfoss, one of the companies manufacturing Programmable Logic Controllers.

The compiler under discussion is developed by using C Language and ANTLR3 as parser generator. Structured Text language specification is translated into Extended Backus Naur form and visitor pattern is adopted to implement semantic analysis and code generation. The XML is validated using the schema provided by Sauer-Danfoss.

The results include a working compiler translating from Structured Text source to XML target and the correctness and performance of the translated Structured Text programs is evaluated.

Complete implementation of the language was not possible due to time constraint and incompatibility of the two standards. Few features exposed by Structured Text are not supported by Sauer Danfoss XML schema at the moment. A subset of the language is implemented as a proof of concept and a base for further development.

Key terms:

Compiler, XML, ANTLR3, Structured Text, PLC, Visitor pattern, Formal languages, Translation.
# Table of Contents

1 **INTRODUCTION** .................................................................................................................................................. 1

1.1 **PROBLEM DESCRIPTION** .............................................................................................................................. 2

1.2 **GOAL CRITERIA** ................................................................................................................................................. 4
  1.2.1 **Correctness** .................................................................................................................................................. 4
  1.2.2 **Maintainability** .............................................................................................................................................. 4
  1.2.3 **Performance** .................................................................................................................................................. 4

1.3 **OUTLINE** ............................................................................................................................................................ 4

2 **BACKGROUND** ...................................................................................................................................................... 5

2.1 **DEFINITIONS** ...................................................................................................................................................... 5
  2.1.1 **Syntax** .......................................................................................................................................................... 5
  2.1.2 **Terminal symbols** ....................................................................................................................................... 5
  2.1.3 **Non terminal symbols** ................................................................................................................................ 5
  2.1.4 **Production rules** .......................................................................................................................................... 5
  2.1.5 **Semantics** ..................................................................................................................................................... 5
  2.1.6 **XML** ............................................................................................................................................................ 5
  2.1.7 **Visual programming languages** .................................................................................................................... 5
  2.1.8 **Abstract syntax tree** .................................................................................................................................... 6

2.2 **FORMAL LANGUAGES** ....................................................................................................................................... 6

2.3 **TRANSLATION OF FORMAL LANGUAGES** ....................................................................................................... 6
  2.3.1 **Formal grammar** .......................................................................................................................................... 6
  2.3.2 **Context Free Grammar** ................................................................................................................................ 6
  2.3.3 **Parsing** .......................................................................................................................................................... 7
  2.3.4 **Compilation** ................................................................................................................................................... 8

2.4 **STRUCTURED TEXT PROGRAMMING LANGUAGE** ............................................................................................. 9
  2.4.1 **Program organization units** ........................................................................................................................... 10
  2.4.2 **Data types** .................................................................................................................................................... 11
  2.4.3 **Variables** ....................................................................................................................................................... 11
  2.4.4 **Expressions** ................................................................................................................................................... 12
  2.4.5 **Statements** ..................................................................................................................................................... 13

3 **APPROACH** .............................................................................................................................................................. 14

3.1 **INTEGRATION WITH THE EXISTING SYSTEM** .................................................................................................. 14

3.2 **IDENTIFIED USE CASES** .................................................................................................................................. 14

3.3 **FILE DIAGRAM** ................................................................................................................................................... 15

3.4 **CALL GRAPH** ...................................................................................................................................................... 17

4 **IMPLEMENTATION** ................................................................................................................................................ 18

4.1 **SOURCE CODE STRUCTURE** ............................................................................................................................ 18
  4.1.1 **Entry point** .................................................................................................................................................... 18

4.2 **STRUCTURED TEXT GRAMMAR** .......................................................................................................................... 19

4.3 **VISITOR** .............................................................................................................................................................. 21

4.4 **DATA TYPES** ...................................................................................................................................................... 22

4.5 **SYMBOL TABLE** ................................................................................................................................................ 22

4.6 **DATA STRUCTURES** .......................................................................................................................................... 23

4.7 **CODE GENERATION LIBRARY** ............................................................................................................................ 25

5 **EVALUATION** .......................................................................................................................................................... 27

5.1 **CORRECTNESS** ................................................................................................................................................. 27
  5.1.1 **Variable declarations** .................................................................................................................................. 27
Chapter 1
Introduction

Embedded systems play vital role in industrial automation, robotics, control systems, and in our daily life appliances. Traditional ways to program these systems require skills in programming languages and hardware knowledge in addition to the understanding of purpose and requirements of the underlying system. Having all these expertise for a single person is not common in real life. To ease the implementation of such systems, visual and simple tools are evolved over time and these tools are designed to target control engineers.

Sauer-Danfoss is a global company; they engineer and develop mobile machines including computer controlled hydraulics, electrical and electronic systems and programmable logic controllers. Most of these systems are preprogrammed and can also be reprogrammed by user as the overall system evolves. To provide more flexibility in these systems Sauer-Danfoss have developed a graphical integrated development environment which completely hides the machine language and exposes standard electrical components to draw complete logic and operation of the system. An example of such drawing is illustrated in Figure 1.1.

Similar approaches are attempted by other companies who work in the same domain and each of them ended up in a slightly different solution. This lead to the problem of incompatibility as large systems usually consists of modules developed by different manufacturers.

IEC-International Electro technical Commission is an organization; they prepare and publish international standards for electrical and electronic technologies. IEC published standard language for programmable logic controllers under document number IEC-61131-3, this language is easy to learn and maintain [1]. Sauer-Danfoss decided to adopt this standard to provide their customers the choice of using both legacy graphical development environment and Structured Text programming language to reprogram the systems.

One possible solution for adding support of this new language is to develop a completely new tool chain only supporting Structured Text programming language and bundle it along with the graphical IDE. But a lot of components are already developed in graphical system, they will not be reused. The second approach is to attach another layer of software on top of existing graphical development system, which can translate code written in Structured Text programming language to graphical design. It is possible to reuse the existing components by using the second approach and is the problem domain of this thesis work.

The IDE developed by Sauer-Danfoss enables the user to pick and drop electronic components such as Boolean logic gates, timers, loops, wires, input output ports, memory buffers, complete subsystems and much more. These components can be connected together to design a complete operational electronic circuit. The designed circuits can be tested, debugged and compiled for different target controllers. Compiled machine code is then transferred to micro-controller unit of the control system for implementation of the system.

Behind this simple and user friendly process, there is a complex set of operations which are executed in a sequence. First of all when a circuit is designed in the graphical IDE, it is saved in an XML file where components and their connections are represented by XML tags. This XML is verified to ensure correctness of the design. After verification, it is translated to C Language code and then it is compiled to binary executable for target hardware.
1.1 Problem description
As described in introduction, our goal is to design and implement a compiler which can translate any given program written in Structured Text language to its equivalent XML program. Stating more specifically, compiler should generate a digital circuit design from the given program. The generated circuit should consist of digital components such as input/output modules, memory modules, logical operations, arithmetic operations, switches, and constant values. Figure 1.1 demonstrates the use of a few components for a simple operation. Component symbols used to draw this diagram and diagrams in coming sections are taken from the Sauer-Danfoss PLUS+1 GUIDE.

![Simple circuit diagram](image)

*Figure 1.1: Simple circuit diagram*

In Figure 1.1 S1 and S2 are two input modules, which can take input from any device such as a key pad or sensors. As every component must have at least one connection in order to attach it with other components, S1 and S2 has connections connected to input of Boolean AND and Boolean XOR operations respectively. In XML layout named IDs are assigned to these connections. Two connections having same IDs are considered connected. For each connection properties such as the direction of flow, data type and the sequence of logic flow can be configured in the respective component tag.

The circuit layout illustrated in Figure 1.1 is saved in XML format by PLUS+1 GUIDE as described in Listing 1.1. Some details are omitted due to copy right restrictions of Sauer-Danfoss.

```xml
<Component Name="IC_MODULE-IN-U1">
  <A Key="V1" Value="P1"/>
  <PIN Name="X1" Net="UN4">
    <A Key="PINDIR" Value="O"/>
    <A Key="PINTYPE" Value=""/>
  </PIN>
</Component>

<Component Name="IC_MODULE-IN-U1">
  <A Key="V1" Value="P2"/>
  <PIN Name="X1" Net="UN5">
    <A Key="PINDIR" Value="O"/>
  </PIN>
</Component>
```
Listing 1.1: XML equivalent of Figure 1.1
The XML layout in Listing 1.1 is taken from PLUS+1 GUIDE IDE from Sauer-Danfoss. In this XML listing, each component is represented by a component tag, component tags contain general and specific properties of components such as the type of component, its name, its id, and connection information. Under the component tag, its connections are represented by PIN tags with PIN name, PIN id (NET) direction of information flow (PINDIR) and data type (PINTYPE) if any.

1.2 Goal criteria
The main intention of this work is to solve the above mentioned problem in 1.1 by implementing a compiler. The final product should be of acceptable quality along with this major goal. Critical properties of software should be taken into account to assess the quality of final product; these include correctness, maintainability and performance.

1.2.1 Correctness
Correctness of software is measured with respect to a specification. A software is said be correct if it fulfills both functional and non functional requirements. The software under development should be correct in terms of functionality and should behave correctly for both positive and negative use cases [2].

1.2.2 Maintainability
Technologies evolve and requirements change over time. This evolution results in modification and bug fixes in the software. This change becomes hectic if the software is not well documented or if is not well structured. The goal for the software being developed is to make it maintainable and well documented [2].

1.2.3 Performance
Since the software being developed is a compiler which will be used to translate one programming language into another. The performance of compiler itself is not critical but the performance of translated language is critical. The goal is to achieve good performance for both the compiler software itself and the translated program [2].

1.3 Outline
This thesis report is structured in numbered chapters.
   Chapter 1 provides an introduction to the problem addressed by this report. A simple problem description is provided along with goals of this thesis.
   Chapter 2 is a brief introduction to the key terms, formal languages, theory of compilation and Structured Text programming language.
   Chapter 3 discusses the design theory and the method used to develop the compiler.
   Chapter 4 explains the implementation of the compiler, the grammar of the Structured Text programming language, compiler front end, source code structure, and documentation.
   Chapter 5 contains some experiments and evaluations of the goals.
   Chapter 6 gives conclusion and provides suggestions for future work.
   Appendix A describes the usage of compiler and PLUS+1 GUIDE compilation tool chain.
   Appendix B lists the tools used for development along with their usage and installation guidelines.
Chapter 2
Background

This chapter introduces the theory of formal languages and translation. Key terms are defined, context free grammars, parsing and theory of compilation is briefly explained in this chapter. Structured Text programming language is described and its elements are elaborated. This chapter is intended to be an introduction to the related terms and to provide base for the later chapters.

2.1 Definitions
The definitions are provided in the context of computer science and are closely related to programming languages.

2.1.1 Syntax
Syntax defines rules which are used to decide whether a given combination of symbols is correct or not. It only describes the correctness of the program structure and has no relation to the operation and behavior of the implementation [4].

2.1.2 Terminal symbols
Terminal symbols of a language are characters, words, or string literals which cannot be split further in smaller units, or in other words they no longer belong to the language if broken into parts [5].

2.1.3 Non terminal symbols
Non terminal symbols are rules of the language grammar consisting of sequence of terminal symbols and non terminal symbols themselves [5].

2.1.4 Production rules
Production rules or commonly known as rules are the sets of instructions which guides the expansion of nodes in the language grammar to generate and recognize given language [5].

2.1.5 Semantics
The behavior and meaning of a program is described by the semantics of the programming language. It assigns actions to the symbols and sentences and provides understanding of the behavior of a given piece of code [4].

2.1.6 XML
Extensible Markup Language is a textual data format for encoding documents and data structures. XML documents consist of tags which contain data elements, attributes and or other tags [6].

2.1.7 Visual programming languages
Any programming language which facilitates a user to develop computer software graphically rather than writing in text. Flow Charts and Unified Modeling Language are common visual languages in computer sciences [1].
2.1.8 Abstract syntax tree
Abstract syntax tree is a graph representing the useful structure of a program written in defined programming language. Each node in the graph is a representation of its respective construct in the source code. It is different from concrete syntax tree as it ignores the unnecessary details in the source code such as parenthesis and line terminators etc [5].

2.2 Formal languages
Humans from the very beginning use languages as a medium of communication. These languages are very complex and a complete definition of such languages does not exist. One of the main reasons for not having a concrete definition of these languages is their evolution over time.

A formal language is a collection of words, such as, strings of letters, or symbols lawfully grouped together. A formal language is often defined by means of language grammar (also know as formation rules) [5].

2.3 Translation of formal languages
Translating one formal language into another formal language involves a chain of complex steps. This chain starts with identification of the grammatical model of source and target languages, generating lexer and parser for the source language, implementing the semantic and syntactic analyzer, and finally writing the target language generator.

Software which translates from source language to target language is usually known as translator or compiler. Compilers are categorized as single pass, two pass or three pass. It depends on the number of times the software is executed to translate single piece of source language into target language. Components of compiler, lexer and parser can be generated by front end generating tools such as YACC, Bison and ANTLR3. These tools are also known as compiler compilers and come with support of different programming languages for implementation [7].

2.3.1 Formal grammar
A set of precise rules for combining words and sentences to form a language is called grammar. The grammar addresses the position and valid combination of words. It does not define the meanings of the sentences or semantics of the language. In the context of compilers the grammar rules are also referred to as string re-writing rules [7].

A language is generated by arbitrarily applying these re-write rules starting from a particular symbol. Generating an arbitrary language is not useful at all, grammars are mostly used to recognize given strings to infer whether they belong to given language or not [7].

2.3.2 Context Free Grammar
Context Free Grammar is a set of productions or rules [7]. Each production rule in the set is of the form as described in Listing 2.1.

\[ X \rightarrow y \]

Listing 2.1: Production rule

In Listing 2.1 \( X \) is a non terminal symbol and \( y \) is string of terminals and/or non terminals including the possibility of empty string. Productions provide the blueprint for the words and
sentences to group together in order to form a valid construct or program. Production rules are some times also referred to as phrase structure grammars. Backus-Naur Form (BNF) is closely related to these production rules and compiler compilers usually accept BNF or Extended BNF. Context free grammars are commonly used to represent programming languages and even natural languages for processing.

Consider a programming language which is simple in its nature and consists of a program. Program can only contain a function process and the function process can only contain two types of statements, set and clear. These statements can occur repeatedly many number of times including none. To represent this language one can use the model described in Listing 2.2.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>program</td>
<td>(2.1)</td>
</tr>
<tr>
<td>program</td>
<td>'process' '{' statement* '}</td>
<td>(2.2)</td>
</tr>
<tr>
<td>statement</td>
<td>set_statement</td>
<td>clear_statement</td>
</tr>
<tr>
<td>set_statement</td>
<td>'set'</td>
<td>(2.4)</td>
</tr>
<tr>
<td>clear_statement</td>
<td>'clear'</td>
<td>(2.5)</td>
</tr>
</tbody>
</table>

Listing 2.2: Example Context free grammar

The productions in the listing 2.2 are given numbers in parenthesis and are not included in the production rules. The first rule (2.1) has start state start. It is not the start state because of its name, but because it has no parent. State start expands to program. In rule (2.2) program expands to string literals 'process' followed by curly bracket '{' and then it has a child statement with a star representing infinite number of occurrences including zero. The rule ends in string literal '}'. Now the rule (2.3) statement can expand either in set_statement or clear_statement which expands to their respective children (2.4) and (2.5). If we desire to modify the language so that only one statement can occur, we can modify the program rule (2.2) as described in listing 2.3.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>program</td>
<td>'process' '{' ('set'</td>
<td>'clear')* '}'</td>
</tr>
</tbody>
</table>

Listing 2.3: Usage of plus ‘+’ symbol

In listing 2.3 the plus symbol means statement child can expand infinite number of times but one occurrence is mandatory. There is another thing to note in listing 2.2, rule (2.1) is only there for readability, and rules (2.4) and (2.5) could be merged into rule (2.3). A hard to read but shorter version of the same grammar, describing the exact language, will look like as described in listing 2.4.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>program</td>
<td>'process' '{' ('set'</td>
<td>'clear')* '}'</td>
</tr>
</tbody>
</table>

Listing 2.4: Production rules merged together

This grammar described in listing 2.4 now consists of a single rule, but it traded off its readability, and maintainability.

2.3.3 Parsing

Parsing or syntactic analysis is the process of matching input symbols with production rules to decide if the given input conforms to the language specification. In compilers this technique is used to test whether a given program is syntactically correct or not. The component of compiler
responsible for this task is called parser and the component which breaks down input text into tokens is called lexer [4].

A stream of tokens is provided to parser as input and it tries to fit them on the respective grammatical structure. It consumes tokens one by one tests if same token was expected. If a wrong or unexpected token is found, it simply flags syntax error. If no error is flagged the parser creates a tree from tokens called abstract syntax tree. This abstract syntax tree is traversed later by a compiler, interpreter or a translator. This process is illustrated in figure 2.1.

![Diagram of parsing phase]

Figure 2.1: Parsing phase

Depending on the way token stream is consumed parsers are categorized into two types, Top-down parser and Bottom-up parser [4].

Top-down parser tries to match a fetched token with each production rule starting from left-most symbol of the rule. If the symbol is not matched with the token it continues to match it with next symbol of the production rule unless the token is matched or production rule is finished.

Bottom-up parsing on the other hand identifies symbols in the given text and then tries to fit them on the production rules.

2.3.4 Compilation

Compiler is a computer software or set of software components which work together to transforms given program written in source language to a program in target language [7]. Common use of compilers is to transform a human readable program to binary code or machine instructions. A compiler translating a high level language to another high level language is also
known as source to source translator. Enhanced and modern compilers usually facilitate programmers with error detection and target program optimization.

After parsing, the next and usually last phase in the translation process is compilation. In this phase the abstract syntax tree received from the parser is traversed and semantics of the source language are analyzed. Variable data types and their declarations are resolved and this information is saved in a data structure, commonly known as symbol table.

For simplicity this tree is visited again to check compatible assignments and correct use of data types. Strings cannot be used as a condition check of if statement and a Boolean cannot be assigned to an integer variable. In another visit of the tree, when everything is verified and found correct in the source program, target code is generated and is saved in a file. This process is illustrated in figure 2.2.

![Diagram of compilation process]

Figure 2.2: Code generation phase

2.4 Structured Text programming language

Structured Text is textual programming language, its syntax and semantics are defined in IEC-61131-3 standard. A program written in Structured Text is usually composed of functions and statements where statements are composed of predefined keywords and user defined variables. Statements terminate with a semicolon. Structured Text is case insensitive, but it is recommended that the keywords are written in upper case while the user defined variables are written in lower case. This improves the readability of the program. Multi line comments and indentations are also supported and should be used where appropriate [1].

Structured Text programs are intended to run in embedded systems and usually a program executes many times within a second. Each execution is called a loop and this is one of the major
differences between usage of Structured Text and other high level languages. Figure 2.3 describes a simple program written in Structured Text programming language.

Figure 2.3: Simple Structured Text program

A program written in Structured Text gives an impression similar to a program written in Pascal, Ada or C Language. It is especially designed to implement complicated algorithms. Equivalent of a program written in Structured Text can be developed by using Pascal, Ada or C Language. But it is not straight forward to do vice versa.

Many control engineering software packages use Structured Text to create user libraries and packages.

2.4.1 Program organization units
In IEC-61131-3 standard, software consists of units of applications, Functions, Function blocks and Programs. These units are called program organization units, some times abbreviated as POUs. Any POU can be used to perform the same operation, but IEC recommends the following pattern for them [1].

Function is an organized piece of code which can be invoked for execution to accomplish a sub task. Structured text facilitates user to write custom functions and suggests for few standard mathematical functions which should be shipped along with the implementation of the language. These mathematical functions include ADD, ABS, SQRT, SIN and COS [1].

Function blocks are intended to provide abstraction from the internal complexities of an operation by exposing well defined interface. Integrated circuit chips can be seen as function blocks. Function blocks can be written in any programming language, such as C Language and they can retain values of variables to keep track of previous states. This additional feature makes them different from simple functions [1].
Programs in Structured text should implement the control logic of the entire system or subsystem. Functions and Function blocks can be used by programs to simplify the implementation [1].

2.4.2 Data types
Data types and constant literals are most common elements in programming languages. Data types help preventing some runtime errors in the programs by flagging them at compile time. For instance, dividing a string by an integer might not lead to desired behavior of system and neither it is logical to do so. IEC-61131 defines the data types under common element section of the languages. These include both user defined data types and primitive data types. Primitive data types are listed in the table 2.1 [1].

<table>
<thead>
<tr>
<th>Boolean</th>
<th>Integer</th>
<th>Real</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word</td>
<td>Date</td>
<td>Time_of_day</td>
<td>String</td>
</tr>
</tbody>
</table>

*Table 2.1: Primitive data types of Structured Text*

In table 2.1, types are listed in general terms and must not be confused with the keywords for declaring variables of respective types.

2.4.3 Variables
In Embedded systems identifiers or variables are mostly used to communicate with system components through input/output ports. Only temporary and local variables are used to hold intermediate data for processing. In Structured Text, variables can only be defined in the beginning of a program. Once a statement is reached from top, it is not allowed to declare variables anymore. Along with data types, variables have another property associated with them called variable class. A variable class defines the usage and scope of variable [1]. Table 2.2 lists variable classes along with short description.

<table>
<thead>
<tr>
<th>VAR_INPUT</th>
<th>Variables declared as input parameter to POU and value is read only.</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR_OUTPUT</td>
<td>Variables declared as output parameters to POU.</td>
</tr>
<tr>
<td>VAR_IN_OUT</td>
<td>Combines the properties of both INPUT and OUTPUT class.</td>
</tr>
<tr>
<td>VAR</td>
<td>Internally used by POU.</td>
</tr>
<tr>
<td>VAR_EXTERNAL</td>
<td>Variable is defined globally.</td>
</tr>
<tr>
<td>VAR_RETENTIVE</td>
<td>Value retains through different program loops.</td>
</tr>
<tr>
<td>VAR_NON_RETENTIVE</td>
<td>Value does not retain through different program loops.</td>
</tr>
<tr>
<td>VAR_TEMP</td>
<td>Variables to hold temporary data for one</td>
</tr>
</tbody>
</table>
program loop.

<table>
<thead>
<tr>
<th>VAR_INCOLOCATED</th>
<th>Memory is allocated arbitrarily.</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR_LOCATED</td>
<td>Memory is allocated at given address.</td>
</tr>
</tbody>
</table>

Table 2.2: Variable classes

2.4.4 Expressions

Expression is a programming language construct representing a mathematical operation which can be reduced to a value through evaluation. It is composed of an arrangement of operators and operands. It is evaluated by applying the operators on operands with mathematical precedence to yield a result. Listing 2.5 is an example of Structured Text expression.

...}

Result := 4 + 2 - ( value / 2 * 3 );

Listing 2.5: Expression

Operators defined by Structured Text can be categorized into Arithmetic operators and Logical operators. Arithmetic operators operate on numeric values through operands and the result produced by the operation is a numeric value. Addition, subtraction and multiplication operators can be considered as example of arithmetic operators. Logical operators on the other hand take arithmetic values and Boolean values through operands but the result produced by the operation is always a Boolean value, Greater than comparison, Less than comparison, Boolean NOT and Boolean OR operators are example of logical operators [1].

A complete list of operators defined by Structured Text is provided in table 2.3. The list is sorted from top to bottom by the precedence of operation.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(expression)</td>
<td>Parenthesis</td>
</tr>
<tr>
<td>identifier (argument list)</td>
<td>Function evaluation</td>
</tr>
<tr>
<td>**</td>
<td>Exponential operator</td>
</tr>
<tr>
<td>- , NOT</td>
<td>Negation / Complement</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication operator</td>
</tr>
<tr>
<td>/</td>
<td>Division operator</td>
</tr>
<tr>
<td>MOD</td>
<td>Modulus operator</td>
</tr>
<tr>
<td>+</td>
<td>Addition operator</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction operator</td>
</tr>
<tr>
<td>&lt; , &gt; , &lt;=, &gt;=</td>
<td>Comparison operators</td>
</tr>
<tr>
<td>=</td>
<td>Equality comparison operator</td>
</tr>
</tbody>
</table>
<>
<table>
<thead>
<tr>
<th>&lt;&gt;</th>
<th>In-equality comparison operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; , AND</td>
<td>Logical and operator</td>
</tr>
<tr>
<td>XOR</td>
<td>Logical exclusive or operator</td>
</tr>
<tr>
<td>OR</td>
<td>Logical or operator</td>
</tr>
</tbody>
</table>

Table 2.3: Operators

2.4.5 Statements

Statements defined by Structured Text resembles to the statements of Pascal programming language. Each statement ends with a terminator symbol, semicolon ';'. Statements can be grouped in Assignment, Control, Selection and Iteration statements [1].

Assignment statements reset the value of a variable on the left hand side of assignment operator. It consists of an assignment operator with a variable on left hand side and an expression on the right hand side. The data type of variable and expression must be matched an error is reported otherwise. If the right hand side of assignment operator is a multi element variable or structure, left hand side must be a variable of same structure and each element of the target variable structure is assigned the value from its respective element from right hand side variable [1].

Control statements provide the mechanism for invoking functions and function blocks including return and exit statements [1].

Selection statements are used to selectively execute statements or blocks of statements at runtime depending on the state of the system. These include IF, ELSIF, ELSE, and SWITCH CASE statements [1].

Iteration statements depending on the control expression specify to execute a statement or group of statements repeatedly. FOR, REPEAT, and WHILE statements are defined by Structured Text as iteration statements [1].
Chapter 3
Approach

This chapter describes the structure and design theory of IECC compiler application. How it is integrated with the existing development environment. Use cases, interaction of source modules and how different modules are integrated together to form the IECC compiler software is discussed in this chapter.

3.1 Integration with the existing system

Once an application is written in Structured Text programming language it needs to be translated to XML based application. PLUS+1 GUIDE tool chain further processes this XML based application, mean while it is the responsibility of IECC compiler to generate a report listing all errors and warnings corresponding to the given Structured Text application. Figure 3.1 illustrate the PLUS+1 GUIDE IDE system and IECC compiler integrated with PLUS+1 GUIDE tool chain.

![Figure 3.1 - IECC compiler integrated with PLUS+1 GUIDE IDE](image)

PLUS+1 GUIDE’s Structured Text Editor is an external entity which facilitates a system developer to write software in Structured Text programming language. There is nothing special about this editor except syntax highlighting etc. A compile event in the PLUS+1 GUIDE IDE starts the process of compilation. It is the responsibility of the IDE to retrieve the file name from the Editor and pass it on to the IECC compiler. If the generated error report contains no fetal errors, PULS+1 GUIDE Compilation tool chain is invoked by the IDE providing it the target XML file name and path. The PLUS+1 GUIDE Compilation tool chain is explained in Appendix A.

3.2 Identified use cases

It is quite important to define use cases [2], what features must be exposed by IECC compiler which are important to integrate it with PLUS+1 GUIDE tool chain. Figure 3.2 lists the use cases of IECC compiler.
Interaction of the PLUS+1 GUIDE tool chain with IECC compiler is straightforward. The only features IECC compiler expose to the tool chain include compilation of the Structured Text based source code. If the source code is compiled successfully PLUS+1 GUIDE retrieves the XML based target code. Whether the source code is compiled successfully or not is determined by the error report.

3.3 File diagram
For systems developed in object oriented programming languages, Class diagrams are used to illustrate the functionality and dependency among components and classes. Since the IECC compiler is implemented in C Language which is a functional programming language. File diagram is used to illustrate the contents and relations between the source files [2]. Functions are grouped together in files based on their purpose and functionality. Figure 3.3 illustrates the relation among important source files of IECC compiler.
The files mentioned in the Figure 3.3 are in abstract form and are explained in detail in section 4. It is also important to note that there is no built in mechanism for garbage collection in C Language thus each module is responsible for freeing up the memory allocated by it. Functions starting with destroy text are used for this purpose.

a) iecCompiler Module
This is the main module of the compiler software and is responsible for integrating other modules together. It invokes the iecParser module and iecVisitor module to compile the Structured Text source code. After the compilation phase is completed, process of memory cleanup is started. After memory cleanup the execution IECC compiler is stopped.

b) iecParser Module
This module is generated by ANTLR3 tool, when invoked it generates the abstract syntax tree and returns a pointer to the iecCompiler module. Module iecLexer is also auto generated by the Antlr tool and is used internally by iecParser module for the creation of Abstract syntax tree.

c) iecVisitor Module
This module is the core module of the IECC compiler. It implements a visitor pattern to visit the abstract syntax tree. Abstract syntax tree is visited three times for a complete compilation phase, for the first visit, symbol table is populated, and for the second visit the semantics are verified.
target code is generated during the third visit. This module uses `symbolTable` module and `codeGenLib` module for verifying the semantics and generating the target code respectively.

### 3.4 Call graph

Call graphs are used to represent the relations between the functions grouped together to accomplish a particular task. These provide a convenient way to understand the structure of the system [2]. Figure 3.4 illustrates the call graph of IECC compiler.

![Call graph of IECC compiler](image)

The entry point of the IECC compiler is the main function which is defined in `iecCompiler` module. First it initializes the modules which will be used later in the compilation process. Then the visitor is invoked which traverses the abstract syntax tree and generates the target code. After the visit function completes its execution, `exitCleaningUp` function is called to return the allocated memory to the operating system.
Chapter 4
Implementation

In this chapter the directory structure of the project is discussed and a short description of each file is listed. Selective source files, data structures and functions are explained in detail.

The project is implemented in C Language under Linux platform and designed as platform independent. Refer to appendix B for details of tools used to develop the software and build process.

Table 4.1 lists the contents of project source directory.

<table>
<thead>
<tr>
<th>Directory / File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>documentation</td>
<td>Contains this document</td>
</tr>
<tr>
<td>grammar</td>
<td>Contains the grammar file</td>
</tr>
<tr>
<td>src</td>
<td>Contains source and files including those auto generated by ANTLR3</td>
</tr>
<tr>
<td>tools</td>
<td>Contains the ANTLR3 tool, header files and library to link with executable</td>
</tr>
<tr>
<td>makefile</td>
<td>Contains script for grammar processing and project compilation</td>
</tr>
</tbody>
</table>

Table 4.1: Source code directory structure

4.1 Source code structure
Source code of IECC compiler is organized in header and source files. Table 4.2 lists the contents of src directory.

<table>
<thead>
<tr>
<th>codeGenLib.c</th>
<th>iecCompiler.c</th>
<th>symbolTable.c</th>
</tr>
</thead>
<tbody>
<tr>
<td>codeGenLib.h</td>
<td>iecCompiler.h</td>
<td>symbolTable.h</td>
</tr>
<tr>
<td>dataTypeUtil.c</td>
<td>iecVisitor.c</td>
<td>util.c</td>
</tr>
<tr>
<td>dataTypeUtil.h</td>
<td>iecVisitor.h</td>
<td>util.h</td>
</tr>
<tr>
<td>dataStructures.h</td>
<td>iec.h</td>
<td>iecLexer.c*</td>
</tr>
<tr>
<td>iecLexer.h*</td>
<td>iecParser.c*</td>
<td>iecParser.h*</td>
</tr>
</tbody>
</table>

Table 4.2: IECC source files

The files marked with a star are generated by ANTLR3.

4.1.1 Entry point
Programs developed in C Language starts execution from main function. For this project the main function is defined in the file iecCompiler.c. The function first processes arguments provided by operating system’s console. Then it creates input stream from the source file by using ANTLR3 utility function antlr3AsciiFileStreamNew. The language is turned into case insensitive by calling setUcaseLA and providing it the stream of characters. Now the stream is
passed on to lexer and tokens are retrieved into a data structure. These tokens are provided to parser to build a parse tree in memory. Finally the visitor is invoked on the top most node in the tree, which visits every child node one by one. At the end of this process, libraries are cleaned up and memory allocated during the execution is returned back to operating system.

A couple of utility functions are provided with this implementation for debugging purpose. These are intended to be used during the development or maintenance of this software. These functions can be enabled by uncommenting the code in listing 4.1.

```c
/*
 * To view the parse tree, Use this code.
 * printf("%s\n", tree->toStringTree(tree)->chars);
 */

Listing 4.1: Debugging function to print the abstract syntax tree

If the code in listing 4.1 is uncommented it prints parse tree on the console. This helps finding out if the grammar is correctly defined for some particular semantic and to verify the parent-child relations in the tree.

```c
/*
 * To generate yEd viewable markup code of parse tree, Use this code.
 * treeGML(tree, inputFileName);
 */

Listing 4.2: Debugging function to generate yED viewable abstract syntax tree

If the code in listing 4.2 is uncommented it creates a yED (Graph viewing tool) graph of the parse tree. This is almost the same as the parse tree printed on console. But for large trees, it becomes convenient to analyze graphically.

4.2 Structured Text grammar

Grammar of Structured Text language is adopted from the ANNEX-B of IEC-61131 specifications. Syntax used in IEC-61131 grammar specification is different from that of used by ANTLR3. Few changes are made to remove left recursions in provided grammar. The grammar file starts with the code mentioned in listing 4.3.

```c
options
{
   language = C;
   output=AST;
   ASTLabelType = pANTLR3_BASE_TREE;
}

Listing 4.3: Structured Text grammar options

Listing 4.3 are the options which are used by ANTLR3 tool to generate compiler front end, language = C tells the tool to generate parser and lexer in C Language, output=AST specifies that an abstract syntax tree is required to be generated and finally ASTLabelType makes pANTLR3_BASE_TREE become parent class of generated AST. Note that in the generated code, classes are simulated by structures and pointers to functions.
Tokens are defined in this file after ANTLR3 options. These include both real tokens and imaginary tokens. Real tokens are used to identify language key words while imaginary tokens are used to create dummy nodes in parse tree and these simplify traversal of AST. Code samples (4.1) and (4.2) from listing 4.4 are examples of real and imaginary tokens respectively.

\[
\begin{align*}
IF & = 'IF'; \\
POU; & (4.1) \\
\end{align*}
\]

Listing 4.4: Real and Imaginary tokens

After tokens, production rules are defined in this file. Starting rule is goal and rest of the production rules are its children. Code for goal production rule is described in the listing 4.5.

```
//***************************************************************
//  Starting point
//***************************************************************
goal :
  program_declaration -> ^(POU program_declaration)
  | function_declaration -> ^(POU function_declaration)
  | function_block_declaration -> ^(POU function_block_declaration)
;
```

Listing 4.5: Root production rule

Compiler triggers parser when a program is provided to compiler for translation. First thing the parser test is the input source. Program must consist of program_declaration, function_declaration or function_block_declaration. These productions are matched by being replaced with their definitions recursively until unique tokens are identified. If all tokens are matched successfully, a root node with the id POU is created in the abstract syntax tree, otherwise a syntax error is reported. ANTLR3 grammar editor facilitates easy navigation between the production rules and their definitions.

There are two important things which are potential bugs in the ANTLR3 implementation and are not documented. First, consider the token definitions in listing 4.6.

\[
\begin{align*}
EQ & = '='; \\
LT_OP & = '<'; \\
GT & = '>'; \\
\end{align*}
\]

Listing 4.6: Token definitions

In listing 4.6 tokens define the operator symbols, and most of them have resemblance except the (4.4), this was defined as LT as a convention to (4.3) and (4.5) but ANTLR3 without reporting any warning or error, ignored LT which caused problems handling the expressions using '<' operator.

Second, leaving an empty line or comment at the end of grammar file makes ANTLR3 tool to report arbitrary errors and grammar processing is failed.
4.3 Visitor
Visitor for abstract syntax tree is implemented in the file iecVisitor.c. Usually three visitors are required for compilation or language translation. These include symbol table visitor, semantic analysis visitor and code generation visitor. In this implementation all three visitors are merged into one visitor which can be invoked to perform concerned operation according to visit type provided as a parameter [9].

Important functions contained in this visitor are visit, visitChildren, and visit(nodes) where nodes are the tree nodes of abstract syntax tree. Visitor is invoked by calling visit function on root node of the abstract syntax tree with appropriate visit type provided as parameter. This visit function consists of a long list of switch case statements where each case is matched against its relative tree node and a visit handler is invoked to perform appropriate task.

If none of these cases match with the given node, a default case is executed which moves current node to one of its children and recursively invokes visit function on the child node. Eventually complete tree is traversed in this manner. Consider listing 4.7 for code snippet from visit function. If there are no more child left in the tree to visit, the visit function returns NULL and visiting process is terminated.

```c
void* visit(pANTLR3_BASE_TREE tree, void* data, int visitType)
{
    switch(tree->getType(tree))
    {
        case POU:
            return visitPou(tree, data, visitType);
        break;
        case FUNCTION_DECL:
            return visitFunctionDecl(tree, data, visitType);
        break;
        .
        .
        default:
            visitChildren(tree, data, visitType);
    }
    return NULL;
}
```

Listing 4.7: Visit function code snippet

For symbol table visit type, the identifiers are recognized and are saved in symbol table data structure. For type checking visit cycle, recognized errors and warnings are saved in errorMessages string buffer. Similarly for code generation visit XML code is appended to generatedCode string buffer which is later saved to a file.

Visitor needs to be initialized before it can perform its operation. Initialization of visitor can be done by invoking a helper function initializeVisitor and when the visitor is no longer required, destroyVisitor can be invoked to free up memory resources occupied by visitor.
4.4 Data types
Data types defined by PLUS+1 GUIDE and Structured Text are slightly different. In this report these are referred to as signal types and data types respectively. Intermediate types are defined in file dataStructures.h to map signal types to data types. Consider listing 4.8 for code snippet as an example of intermediate types.

```c
/**
 * Data types
 */
#define __TYPE_U8 0
#define __TYPE_U16 1
#define __TYPE_U32 2
```

Listing 4.8: Intermediate types

Identifiers are saved in symbol table using signal types. Errors are reported using Structured Text data types during type checking visit cycle. Similarly type compatibility is analyzed at various places. For these recurring operations utility functions are added to provide abstraction and ease of use. These utility functions increase maintainability, for instance, at the moment all data types cannot be mapped to signal types but in future more signal types will be supported. This will only require updating the related type functions. These functions are grouped together in file dataTypeUtil.c and are exposed by dataTypeUtil.h. A function from each sub group is listed in listing 4.9.

```c
int dataTypeNameToSignalValue (pANTLR3_UINT8 name);
pANTLR3_UINT8 signalValueToDataTypeName(int signalType);
pANTLR3_STRING toSignalName(int signalType);
int isTypeU8Compatible(int type);
```

Listing 4.9: Data type utility functions

4.5 Symbol Table
Symbol table is used to keep track of all identifiers found in given source program. Identifiers are stored in a hash table data structure which is provided by ANTLR3. Wrapper functions are written on top of native functions of hash table interface for easy usage in the program. Symbol table related functions are grouped in file symbolTable.c and are exposed by the interface symbolTable.h. Important functions exposed by this library are described in listing 4.10, 4.11, 4.12, 4.13 and 4.14.

```c
void initializeSymbolTable();
```

Listing 4.10: Symbol table initializer

The function in listing 4.10 initialize data structures used for symbol table and it must be invoked before calling any other function related to symbol table operation.
void destroySymbolTable();

Listing 4.11: Symbol table destructor

The function in listing 4.11 frees up any resources allocated to the symbol table library. This should only be called at the end of program execution when symbol table is no longer required.

pVariableRecord getFromSymbolTable(pANTLR3_UINT8 key);

Listing 4.12: Symbol table lookup function

The function in listing 4.12 looks up for a variable record from symbol table. The key it uses to find existing record is a pointer to a field of unsigned bytes. The key is in fact is the name of the variable.

void putIntoSymbolTable(pVariableRecord variableRecord);

Listing 4.13: Symbol table store function

The function in listing 4.13 stores a variable record in symbol table. It automatically extracts identifier name from provided variable record and use this identifier name as a key to store record in symbol table.

void printSymbolTable(pANTLR3_HASH_TABLE symbolTable);

Listing 4.14: Symbol table printing utility

The function in listing 4.14 is a debugging function. It is useful during program development. Calling this function simply prints information attached with each variable record on console.

4.6 Data structures

Common data objects which are mostly used together are grouped into complex types. These are used to provide easy manipulation of data and to improve readability. Implemented in the file dataStructures.h and important structures are described in listing 4.15, 4.16 and 4.17.

typedef struct PIN_struct
{
    pANTLR3_STRING net;
    Int  signalType;
} PIN;

Listing 4.15: Connection PIN data structure

In XML a PIN represents a connection. There are at least two properties associated with a connection, first, type of signal it can handle, and UN-ID, which connects it with other connections having same UN-IDs. The structure is defined to represent the PIN and is listed in listing 4.15.

typedef struct COMPONENT_CONNECTION_struct
{
    pPIN inPinA;
}
\[ \text{pPIN inPinB;} \]
\[ \text{pPIN outPin;} \]
\[ \text{// Write enable pin is for memory component, if this pin is connected to} \]
\[ \text{// true, memory will be over written.} \]
\[ \text{pPIN writeEnable;} \]
\[ \text{// Block pin is for conditional and iterating statements} \]
\[ \text{pPIN blockPin;} \]
\[ \} \text{ COMPONENT_CONNECTION;} \]

Listing 4.16: Component connection data structure

On abstract level the circuit in XML consists of components which are connected together using PIN connections. The structure in listing 4.16 represents a component with a maximum of two inputs and one output. Support of memory components and control switches is also added in this structure. The \textit{writeEnable} PIN can be connected to either Boolean \textit{true} or Boolean \textit{false} component to make memory component writeable or read only respectively. Similarly \textit{blockPin} can be connected to \textit{true} or \textit{false} component to block flow of signals from input to output. \textit{blockPin} is used to handle conditional statements, where some statements can only execute if attached condition circuit evaluates to Boolean true.

typedef struct VariableRecord_struct
{
    pANTLR3_UINT8 name;
    pCOMPONENT_CONNECTION componentConnection;
    int variableClass;
    int retainability;
    int dimensions;
    int subrange;
    int edgeType;
    int constant;
    int dataType;
    int isUsed;
} VariableRecord;

Listing 4.17: Variable record data structure

VariableRecord data structure is used to hold information of a variable defined in Structured Text program. The data structure is described in listing 4.17. This include name of identifier, component connection information, class to which this variable belongs such as VAR_INPUT, VAR_OUTPUT etc. \textit{Retainability} is not implemented for now but the information is collected in this field. \textit{Retainability} is used to retain the value contained in variable over cold restart of the system. \textit{Dimensions} field is used to specify array property of variable. Currently two and more than two dimensional arrays are not supported in this implementation. \textit{Constant} is a Boolean flag, if set marks the variable read only. \textit{DataType} holds the intermediate type of the variable. Refer to section 4.4 for intermediate types. \textit{IsUsed} is a Boolean flag which infers if a variable is assigned a value yet or not. This is used to check that output variables are not written more than once during a program loop. \textit{Subrange} restricts the variable to only contain a value between specified ranges of numbers.
To create objects of these above mentioned structures respective factory methods must be used. These methods keeps track of all created objects and can free up resources allocated to them at the end of program execution. These factory methods are listed in listing 4.18 and 4.19.

```c
createComponentConnection();
```

*Listing 4.18: Factory method to create component connection object*

This function creates an object of type *component* and returns a pointer to created object. It is declared in the file *CodeGenLib.h*.

```c
variableRecordNew();
```

*Listing 4.19: Factory method to create variable record object*

This function creates an object of type *variableRecord* and returns a pointer to created object. It is declared in the file *symbolTable.h*.

### 4.7 Code Generation Library

Functions used to generate XML layout from Structured Text elements are grouped together in the file *CodeGenLib.c* and are exposed by interface defined in the file *CodeGenLib.h*. Important functions defined by this library are discussed in listing 4.20, 4.21, 4.22, 4.23, 4.24, 4.25, 4.26 and 4.27.

```c
void initializeCodeGenLib();
```

*Listing 4.20: Code generation library initializer*

The function in listing 4.20 initializes data structures and creates buffers used for code generation. This function must be invoked before any other operation is performed for code generation.

```c
void destroyCodeGenLib();
```

*Listing 4.21: Code generation library destroyer*

The function in listing 4.21 frees up any resources occupied by code generator. This function can be invoked to clean up memory after generation of code is successful and target code is saved in a file.

```c
pANTLR3_STRING getGeneratedCode();
```

*Listing 4.22: Function to retrieve generated code*

Generated code is appended to a string buffer which is declared in code generation library and the function in listing 4.22 retrieves contents of this buffer.

```c
void varDeclarationCode(pVariableRecord variableRecord);
```

*Listing 4.23: XML code generator from variable record object*
The function in listing 4.23 is capable of generating XML component code for given variable record. It can extract variable class type and depending on the variable class, it uses the appropriate component in the generated code.

```c
void constXXXXCode(pCOMPONENT_CONNECTION compCon);
```

**Listing 4.24: XML code generator from component connection object**

The function in listing 4.24 can generate code for single PIN components such as Constant True and Constant False values.

```c
void setPinTrueCode(pPIN pin);
```

**Listing 4.25: XML code generator from PIN connection object**

The function in listing 4.25 can be used to generate code for setting given PIN to Constant True value. Same operation can be performed by using a Constant True component and an assignment component together but this function reduces generated code.

```c
void constXXXXCode(pCOMPONENT_CONNECTION compCon, pANTLR3_STRING value);
```

**Listing 4.26: XML code generator for numerical literals**

The function in listing 4.26 can be used to generate code for numerical literals found in the Structured Text program.

```c
void assignmentCode (pCOMPONENT_CONNECTION compConA, pCOMPONENT_CONNECTION compConB);
```

**Listing 4.27: XML code generator for assignment operator**

Assignment operator is used to set the value of a variable. A direct connection between two memory components cannot be established because it can overwrite the value contained in right hand side component from left hand side component. Also assignments performed under IF, ELS IF blocks are disabled if statement block is not supposed to be executed. As variables are declared in early stage and UNs of their PINs are assigned before a statement can occur in the Structured Text program. These UNs of two connected variables must be same in order to assign a value. This problem is solved by introducing a controlled wire component which consists of a Boolean control switch, one input connection and one output connection. This component is used to establish connection for each assignment operation. Function in listing 4.27 is used to generate code for such component.
Chapter 5
Evaluation

In this chapter experiments are performed to evaluate the correctness, maintainability and performance of the developed IECC compiler.

5.1 Correctness
To evaluate the correctness of the compiler, various programs are written in Structured Text programming language and are compiled using developed compiler. Error messages and translated programs are observed.

5.1.1 Variable declarations
In listing 5.1 a simple Structured Text program is presented. This program is used to test the basic correctness of the IECC compiler by declaring and using four variables.

(*
* Simple program demonstrating variable declarations.
*)

PROGRAM MainProgram

VAR_INPUT
i_b_switch_1, i_b_switch_2 : BOOL ;
i_b_sensor_1, i_b_sensor_2 : DINT ;
END_VAR

VAR_OUTPUT
o_b_valve_1, o_b_valve_2 : BOOL ;
END_VAR

(* At least one statement is required *)
o_b_valve_1 := false;

END_PROGRAM

Listing 5.1: Simple Structured Text program

After compiling the program in listing 5.1 the compiler generates the output listed in listing 5.2.
Listing 5.2: Compilation result of simple program form listing 5.1

5.1.2 Assignment and addition

The program in listing 5.3 demonstrates the use of input variable, output variable, addition operator and assignment operation.
(*)
* Variable declaration, assignment and addition.
*)

PROGRAM MainProgram

VAR_INPUT
  i_d_sensor_1 : INT;
END_VAR

VAR_OUTPUT
  o_d_valve_1 : INT;
END_VAR

  o_d_valve_1 := i_d_sensor_1 + 50;

END_PROGRAM

Listing 5.3: Structured Text program demonstrating the use of addition and assignment

After generating the program in listing 5.3 the compiler generates the XML document listed in listing 5.4.

<?xml version="1.0" encoding="ISO-8859-1"?>
<NOB_FLAT FileName="MAINPROGRAM" Version="1.0"> <LIBRARY/> <DESIGN> 
<BUSSTRUCTURE> </BUSSTRUCTURE>
<EXTENDS X0="-9000" Y0="-21600" X1="4291220" Y1="2728204"/>
<IC Name="IC_MODULE-IN-U1" X="1" Y="1">
  <A Key="V2" Value="_"/>
  <A Key="V1" Value="I_D_SENSOR_1"/>
  <A Key="VER" Value="101"/>
  <A Key="UGVER" Value="1.1.1"/> 
  <PIN Name="X1" Net="UN1" X="2" Y="2">
    <A Key="PINDIR" Value="O"/>
    <A Key="PINTYPE" Value="S16"/>
  </PIN>
</IC>

<IC Name="IC_MODULE-OUT" X="3" Y="3">
  <A Key="V2" Value="_"/>
  <A Key="V1" Value="O_D_VALVE_1"/>
  <A Key="VER" Value="101"/>
  <A Key="UGVER" Value="1.1.1"/> 
  <PIN Name="X1" Net="UN2" X="4" Y="4">
    <A Key="PINDIR" Value="O"/>
    <A Key="PINTYPE" Value="S16"/>
  </PIN>
</IC>

<IC Name="C__CONST-T1" X="5" Y="5">
  <A Key="V1" Value="50"/>
  <A Key="VER" Value="101"/>
  <A Key="UGVER" Value="4.83"/>
</IC>
Listing 5.4: XML document generated by compiling program in listing 5.3
5.1.3 If condition
The Structured Text program in listing 5.5 demonstrates the use of IF - ELSE statement.

/*
 * If then, elseif, greater than operator,
 * less than operator and logical and operator
 * demonstration.
 */

PROGRAM MainProgram

VAR_INPUT
   i_d_sensor_1 : INT;
END_VAR

VAR_OUTPUT
   o_d_valve_1 : BOOL;
   o_d_valve_2 : BOOL;
   o_d_valve_3 : BOOL;
END_VAR

IF( i_d_sensor_1 > 0 AND i_d_sensor_1 < 80 )THEN
   o_d_valve_1 := true;
ELSIF( i_d_sensor_1 >= 80 )THEN
   o_d_valve_2 := true;
ELSE
   o_d_valve_3 := false;
END_IF;

END_PROGRAM

Listing 5.5: Structured Text program demonstrating IF-ELSE statement

Compilation of the program in listing 5.5 results in the XML listed in 5.6.

<?xml version="1.0" encoding="ISO-8859-1"?>
<NOB_FLAT FileName="MAINPROGRAM" Version="1.0"> <LIBRARY/> <DESIGN>
<BUSSTRUCTURE/></BUSSTRUCTURE>
<EXTENDS X0="-9000" Y0="-21600" X1="4291220" Y1="2728204"/>
<1 Name="IC_MODULE-IN-U1" X="1" Y="1">
   <A Key="V2" Value="_"/>
   <A Key="V1" Value="I_D_SENSOR_1"/>
   <A Key="VER" Value="101"/>
   <A Key="UGVER" Value="1.1.1"/>
   <PIN Name="X1" Net="UN1" X="2" Y="2">
      <A Key="PINDIR" Value="O"/>
      <A Key="PINTYPE" Value="S16"/>
   </PIN>
</1>
<1 Name="IC_MODULE-OUT" X="3" Y="3"/>
<I Name="LG_NOT" X="27" Y="27">
   <A Key="VER" Value="102"/>
   <A Key="UGVER" Value="4.83"/>
   <PIN Name="A1" Net="UN10" X="28" Y="28">
      <A Key="PINDIR" Value="I"/>
   </PIN>
   <PIN Name="X1" Net="UN11" X="29" Y="29">
      <A Key="PINDIR" Value="O"/>
      <A Key="PINTYPE" Value="CBOOL"/>
   </PIN>
</I>

<I Name="LG_AND2" X="30" Y="30">
   <A Key="VER" Value="102"/>
   <A Key="UGVER" Value="4.83"/>
   <PIN Name="A1" Net="UN11" X="31" Y="31">
      <A Key="PINDIR" Value="I"/>
   </PIN>
   <PIN Name="A2" Net="UN9" X="32" Y="32">
      <A Key="PINDIR" Value="I"/>
   </PIN>
   <PIN Name="X1" Net="UN12" X="33" Y="33">
      <A Key="PINDIR" Value="O"/>
      <A Key="PINTYPE" Value="CBOOL"/>
   </PIN>
</I>

<I Name="C_CONST-TRUE1" X="34" Y="34">
   <A Key="V1" Value="1"/>
   <A Key="VER" Value="101"/>
   <A Key="UGVER" Value="4.83"/>
   <PIN Name="X1" Net="UN13" X="35" Y="35">
      <A Key="PINDIR" Value="O"/>
      <A Key="PINTYPE" Value="CBOOL"/>
   </PIN>
</I>

<I Name="GF_CONNECT1" X="36" Y="36">
   <A Key="VER" Value="102"/>
   <A Key="UGVER" Value="4.83"/>
   <PIN Name="A1" Net="UN13" X="37" Y="37">
      <A Key="PINDIR" Value="I"/>
   </PIN>
   <PIN Name="A2" Net="UN12" X="38" Y="38">
      <A Key="PINDIR" Value="I"/>
   </PIN>
   <PIN Name="X1" Net="UN2" X="39" Y="39">
      <A Key="PINDIR" Value="O"/>
      <A Key="PINTYPE" Value="CBOOL"/>
   </PIN>
</I>

<I Name="C_CONST-T1" X="40" Y="40">
   <A Key="VER" Value="102"/>
   <A Key="UGVER" Value="4.83"/>
   <PIN Name="A1" Net="UN13" X="37" Y="37">
      <A Key="PINDIR" Value="I"/>
   </PIN>
</I>

<I Name="C_CONST-T1" X="40" Y="40">
   <A Key="VER" Value="102"/>
   <A Key="UGVER" Value="4.83"/>
   <PIN Name="A1" Net="UN13" X="37" Y="37">
      <A Key="PINDIR" Value="I"/>
   </PIN>
</I>
<A Key="V1" Value="80"/>
<A Key="VER" Value="101"/>
<A Key="UGVER" Value="4.83"/>
  <PIN Name="X1" Net="UN14" X="41" Y="41">
    <A Key="PINDIR" Value="O"/>
    <A Key="PINTYPE" Value="CU32"/>
  </PIN>
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
  
<br>
  
</I>
<A Key="PINDIR" Value="I"/>
</PIN>

<PIN Name="X1" Net="UN18" X="54" Y="54">
   <A Key="PINDIR" Value="O"/>
   <A Key="PINTYPE" Value="CBOOL"/>
</PIN>

<I Name="LG_AND2" X="55" Y="55">
   <A Key="VER" Value="102"/>
   <A Key="UGVER" Value="4.83"/>
   <PIN Name="A1" Net="UN18" X="56" Y="56">
      <A Key="PINDIR" Value="I"/>
   </PIN>
   <PIN Name="A2" Net="UN15" X="57" Y="57">
      <A Key="PINDIR" Value="I"/>
   </PIN>
   <PIN Name="X1" Net="UN19" X="58" Y="58">
      <A Key="PINDIR" Value="O"/>
      <A Key="PINTYPE" Value="CBOOL"/>
   </PIN>
</I>

<I Name="C__CONST-TRUE1" X="59" Y="59">
   <A Key="VI" Value="1"/>
   <A Key="VER" Value="101"/>
   <A Key="UGVER" Value="4.83"/>
   <PIN Name="X1" Net="UN20" X="60" Y="60">
      <A Key="PINDIR" Value="O"/>
      <A Key="PINTYPE" Value="CBOOL"/>
   </PIN>
</I>

<I Name="GF_CONNECT1" X="61" Y="61">
   <A Key="VER" Value="102"/>
   <A Key="UGVER" Value="4.83"/>
   <PIN Name="A1" Net="UN20" X="62" Y="62">
      <A Key="PINDIR" Value="I"/>
   </PIN>
   <PIN Name="A2" Net="UN19" X="63" Y="63">
      <A Key="PINDIR" Value="I"/>
   </PIN>
   <PIN Name="X1" Net="UN3" X="64" Y="64">
      <A Key="PINDIR" Value="O"/>
      <A Key="PINTYPE" Value="CBOOL"/>
   </PIN>
</I>

<I Name="C__CONST-FALSE1" X="65" Y="65">
   <A Key="VI" Value="0"/>
   <A Key="VER" Value="101"/>
   <A Key="UGVER" Value="4.83"/>
   <PIN Name="X1" Net="UN21" X="66" Y="66">
   </PIN>
</I>
5.1.4 Incompatible assignment

For negative test cases the program in listing 5.7 attempts to perform an illegal assignment operation by assigning an integer variable a Boolean value. No XML is generated for the programs containing syntax or logical errors.

Listing 5.7: Demonstration of incompatible assignment

Compiling the program in listing 5.7 generated the error message listed in listing 5.8.
5.1.5 Undeclared variable usage
The Structured Text program in listing 5.9 attempts to assign a value to an unknown or undeclared variable. This must end up in an error.

Listing 5.9: Structured Text program using undeclared variable

Compilation of program in listing 5.9 results in an error message listed in listing 5.10.

5.1.6 Incorrect use of output variables
In Structured Text programming language the output variables cannot be assigned values more than once. This feature is demonstrated by the program in listing 5.11. One can use temporary variables if an output variable is required to be assigned values in different control paths and at the end of the program the value of temporary variable can be assigned to the output variable.

Listing 5.11: Assignments to the output variable
When the program from listing 5.11 is compiled the error message listed in listing 5.12 is generated.

| Line 16, Column 1 : Output variables can only be assigned once! |
| Errors found in program! |

Listing 5.12: Error message from compiler for program in listing 5.11

5.2 Maintainability

Maintenance is more expensive than original development effort of a software system [2]. At the moment it is not possible to translate all semantics of the Structured Text programming language to PLUS+1 GUIDE XML based programs. It is planned to support such missing features in the future and thus the IECC compiler is designed in a way to easily adopt changes and support the missing features.

Software is said to be maintainable if it is easy to add new features or modify the behavior of the existing ones. To achieve high maintainability first step is to make the software understandable. Comments in the source code serve this purpose well. For implementing missing features in the future, the grammar to recognize the Structured Text programming language is fully implemented and stubs are provided in the implementation.

Figure 5.1: Code breakdown of IECC compiler
### Table 5.1: Source code statistics

<table>
<thead>
<tr>
<th>Source code</th>
<th>Comments</th>
<th>Blank Lines</th>
<th>Inactive</th>
<th>Preprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>30175</td>
<td>9461</td>
<td>9120</td>
<td>76</td>
</tr>
</tbody>
</table>

Figure 5.1 and table 5.1 shows some statistics which assess the maintainability of the IECC compiler software.

#### 5.3 Performance

In normal circumstances, performance of a compiler in terms of compilation time is not critical except when the results are intended to be displayed back to the user in real time. But in this case as the IECC compiler is intended to be used along with the PLUS+1 GUIDE tool chain, the compilation time consumed by the IECC compiler is added to the overall time of the PLUS+1 GUIDE tool chain, thus very little time is desired for the compilation process.

Good performance is achieved by implementing the IECC compiler in C Language. In table 5.2, compilation time is measured for a range of programs with different sizes.

<table>
<thead>
<tr>
<th>Lines of code</th>
<th>Compilation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0m0.007s</td>
</tr>
<tr>
<td>100</td>
<td>0m0.013s</td>
</tr>
<tr>
<td>1000</td>
<td>0m0.102s</td>
</tr>
<tr>
<td>10,000</td>
<td>0m0.592s</td>
</tr>
<tr>
<td>100,000</td>
<td>0m6.085s</td>
</tr>
</tbody>
</table>

*Table 5.2: Compilation time measured for different sizes of programs*

The memory consumption of the IECC compiler is measured for a large program in the domain of embedded systems. Figure 5.1 illustrates a graph of memory usage of IECC compiler while compiling a program of length 100,000 LOC (Lines of Code) on a system having 2 GB (Giga Bytes) of total memory.

*Figure 5.1: Memory usage for compiling 100,000 LOC - Total system memory 2GB*
Chapter 6
Conclusion and future work

This thesis has evaluated the implementation of a domain specific programming language. Outcome from this work includes a compiler capable of parsing and compiling a program written in Structured Text. The compiler performs semantic analysis, and translates it to another program written in XML. From a birds eye view this work can be divided into two major tasks; first, to implement the language and second, to simulate the semantics by using basic operations which are supported by target XML schema.

Programming languages with the passage of time evolves towards simple and generic semantics as compared to earlier attempts. Instead of embedding every thing in the language itself, software libraries are provided along with implementations. The biggest motivation behind this trend is to improve quality by keeping the language simple and to increase portability of the language for different platforms. These languages are intended to be used by professional programmers. Structured Text on the other hand has a different target of users, possibly mechanical or control engineers. This difference in the intended audience is really reflected in the design of Structured Text. Increasing easiness for user resulted in a complex language, which is hard to implement.

It was not possible to implement all elements of the Structured Text programming language due to limited number of digital components available in PLUS+1 GUIDE. Increasing available choices provided by XML schema can optimize size and increase performance of translated program. Code generation library can be improved by making it an independent component, such that, if replaced with another library exposing the same interface, completely changes the outcome; for instance Java byte code.
References


Available at: http://www.w3.org/standards/xml/core
[Accessed: June, 2009].


[8] R. Mark Volkmann, Software Engineer at Object Computing Inc. ANTLR3. [Internet]
[Accessed: June, 2009].

Available at: http://www.antlr.org/article/1170602723163/treewalkers.html
[Accessed: March, 2009].

[10] C language target - ANTLR 3.0 prerelease [Internet]
Available at: https://winczent.com/wiki/C_language_target_(ANTLR_3.0_prerelease)
[Accessed: March, 2009].

Available at: http://www.antlr.org/wiki/pages/viewpage.action?pageId=1782
[Accessed: May, 2009].
Appendices

Appendix A Usage and Compilation Chain
Software developed for this thesis project can compile a program written in Structured Text programming language to XML based circuit layout. Generated circuit layout is used to program micro-controllers. But these micro-controller chips require programs in binary format. To translate generated XML program for native hardware few tools are required. Tools required to translate Structured Text programs all the way to C code can be installed automatically by installing PLUS+1 GUIDE software and a valid license must be activated. Note that successful C code translation is only possible with a correct XML program.

First create a project in PLUS+1 GUIDE, use default names for this example to keep things simple. The module MAINMODULE is created at the same time as this project is created. Now add a new module manually, it will have a default name Module1. Input and output signals of the Structured Text program can be defined in MAINMODULE, and the Structured Text program itself can be placed as Module1.

All module input signals must have a module output defined but module outputs do not need any module inputs. Therefore it makes sense to put any needed module outputs to the Structured Text code in another module, for example the main module.

Now save the project. Saving this project will create a directory on the file system containing many files. Browse this directory and locate MAINMODULE.NOB file. This file contains the XML layout of circuit designed in PLUS+1 GUIDE but it is saved in encrypted form and can only be viewed by first decrypting with a utility program NOBDECryPTER which is provided by Sauer-Danfoss.

Copy the generated XML program in project directory and name it MAINMODULE.NOB such that the existing MAINMODULE.NOB file is overwritten by generated XML. Now perform the following steps in a sequence to translate XML program to C program, assuming PLUS+1 GUIDE is installed on default location and if not modify the paths used in commands accordingly.

Step 1) Open shell console of operating system and change current directory to the project directory.

Step 2) Type the command text from listing A.1 and hit enter.

```
"C:\Program Files\Sauer-Danfoss\PLUS14.3P1Tools\P1Compiler\ChParGU1.exe" Module1 2
"ugp1part.txt" "UNTITLED" "UNTITLED" ccode
```

Listing A.1: Basic test command

This command performs basic checks on NOB file and appends recognized errors in the file Screen.tmp.

Step 3) Proceed with the command from listing A.2 if no errors are found in the XML program.

```
"C:\Program Files\Sauer-Danfoss\PLUS14.3P1Tools\P1Compiler\Glink.exe"
"PROJECT=UNTITLED,APPLICATION=UNTITLED"
```

Listing A.2: Linker operations validation command
This command validates linker operations and any errors found are reported on the console.

**Step 4)** Execute the command from listing A.3 to produce C code if there are no linker errors.

```
"C:\Program Files\Sauer-Danfoss\PLUS1\P1Tools\P1Compiler\ChParGU2.exe" Module1 2
"ugp1partxt" "UNTITLED" "UNTITLED" ccode
```

*Listing A.3: CCode generation command*

Any errors encountered during this step are also reported on the console. After this step *Screen.tmp* file can be cleaned up for future use.
Appendix B – Development tools

Implementing a compiler require tools such as source code editor, parser generator, and of course a compiler. There is a variety of such software tools available including both licensed and open source. Open source software usually provide same power as of licensed but these are hard to setup. For this implementation only open source software is used.

B.1 GNU Make

Build process of compiler under development involves the ANTLR3 grammar processing tool which is written in Java and requires Java Virtual Machine to process grammar files. Lexer and parser files are generated as a result, which are purely C source and header files. This generated code is then used with actual implementation of software which is a collection of header and source files. These are very tightly integrated with each other, so that, a minor change in grammar file requires rebuilding of all source code. This build process is automated by using Make utility, which is widely used in software development. GNU Make is a scripting utility which can execute a given set of commands repeatedly and with the ease of a single click or a single command. Setting up GNU Make is a simple task, on Linux platform it comes with the default installation and for Windows platform, MinGW can be installed to simulate Linux OS environment. A script is written specifically for this project which includes the following important targets.

<table>
<thead>
<tr>
<th>process-grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing B.1: Make target for processing grammar</td>
</tr>
</tbody>
</table>

Invoking Make with target in listing B.1 runs ANTLR3 tool to generate Lexer and Parser code. Copies the generated code to source code directory, and removes them from grammar file directory. This target uses Java Virtual Machine, so it must be installed on the system and should be available on system class path.

<table>
<thead>
<tr>
<th>build-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing B.2: Make target for building source code</td>
</tr>
</tbody>
</table>

The Make target in listing B.2 invokes C compiler to compile all source code into one executable file. As GNU Compiler is used for this project, it must be installed on the system for this target to work properly. Errors and warning are reported on the console from which the Make utility is executed.

<table>
<thead>
<tr>
<th>build-all</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing B.3: Make target for processing grammar and building source code</td>
</tr>
</tbody>
</table>

This target invokes process-grammar and build-code in a sequence to build software all the way from grammar file to executable file.
The target in listing B.4 deletes intermediate and generated files so that newly generated files can be copied to the same directory over and over again.

B.2 Eclipse CDT
IDE's help organize software projects under development and facilitates with source code editors to write programs. These also facilitate with easy navigation throughout the project to speed up the development process. Eclipse CDT is used during development of this project and no doubt it is a powerful IDE. It easily integrates with Make utility for build process and GDB (GNU-Debugger) for debugging software. It can be downloaded from http://www.eclipse.org.

B.3 ANTLR3
ANTLR (Another Tool for Language Recognizer) is not just a parser generator; it is a framework for building compilers and language translators. Provides complete support for implementation in C Language and is well documented. It is quite popular in compiler forums for its flexibility and user friendliness. The most important tool bundled with it is parser generator. The collection of data structures and grammar development environment cannot be ignored.

Data structures such as hash tables, lists, vectors and strings processors are extensively used in compilers. As these data structures are used by the generated parser and lexer, for re-usability, they are used in visitor and code generating library as well.

For C Language as target, ANTLR3 package only comes in source distribution, one need to build the library to link with actual software being developed. This process is very tricky and time taking because the guidelines and tutorials are very generic and does not include minor details. A wrong configuration of the system might end up in a quit termination of the process. Once ANTLR3 library is built, it becomes simple to link with any C project. The only purpose to let user build this library is to port software on different platforms.

For Linux platform it comes with install script to automatically configure, build and install libraries. But for windows platform it requires a manual build from source code. In listing B.5 exact version of ANTLR3 distribution is listed which is used with this project.

\texttt{antlr-3.1.2.tar.gz} downloaded from \url{http://www.antlr.org/}

\texttt{Listing B.5: ANTLR3 version used with this project}

The tar file listed in listing B.5 is a complete package of ANTLR3 distribution, if extracted; there will be many subdirectories and files. Runtime library source can be found under the following subdirectory.

\texttt{runtime/C/dist/libantlr3c-3.1.2.tar.gz}

\texttt{Listing B.6: ANTLR3 runtime library source location}

Extract the tar file listed in listing B.6 to get src and include directories. Build it as shared or static library according to choice and link it with the project. There are many tutorials available on home page of ANTLR3, especially a topic with title \textit{How to build this damn thing}. 

48
B.4 GNU C Compiler

To compile the generated and handwritten source code, we need a compiler. A standard and popular compiler among C programmers is GNU C compiler also known as GCC, which can process a variety of source programs and of course it is very concrete when it comes to C Language. It can compile source code with debugging information which is very useful in finding the problems and fixing runtime bugs.