MINISTRY OF NATIONAL EDUCATION
TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

SueC - An Editor and Interpreter for Pseudocode

LICENSE THESIS

Graduate: Mihai PÎȚU
Supervisor: dr. eng. Emil Ștefan CHIFU

2019

MINISTRY OF NATIONAL EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

DEAN,
Prof. dr. eng. Liviu MICLEA

HEAD OF DEPARTMENT,
Prof. dr. eng. Rodica POTOLEA

Graduate: **Mihai PÎȚU**

SueC - An Editor and Interpreter for Pseudocode

1. **Project proposal:** *Short description of the license thesis and initial data*
2. **Project contents:** *(enumerate the main component parts) Presentation page, advisor's evaluation, title of chapter 1, title of chapter 2, ..., title of chapter n, bibliography, appendices.*
3. **Place of documentation:** *Example*: Technical University of Cluj-Napoca, Computer Science Department
4. **Consultants:**
5. **Date of issue of the proposal:** November 1, 2017
6. **Date of delivery:** February 18, 2019 *(the date when the document is submitted)*

Graduate: ______________________________

Supervisor: ______________________________

MINISTRY OF NATIONAL EDUCATION

TECHNICAL UNIVERSITY

OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

Declaraţie pe proprie răspundere privind autenticitatea lucrării de licenţă

Subsemnatul(a) ____, legitimat(ă) cu ____________ seria _________ nr. ____________
CNP ____________________________, autorul lucrării ________
____
____
elaborată în vederea susţinerii examenului de finalizare a studiilor de licenţă la Facultatea de Automatică şi Calculatoare, Specializarea ______________________ din cadrul Universităţii Tehnice din Cluj-Napoca, sesiunea ____________ a anului universitar _________, declar pe proprie răspundere, că această lucrare este rezultatul propriei activităţi intelectuale, pe baza cercetărilor mele şi pe baza informaţiilor obţinute din surse care au fost citate, în textul lucrării şi în bibliografie.

Declar, că această lucrare nu conţine porţiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislaţiei române şi a convenţiilor internaţionale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în faţa unei alte comisii de examen de licenţă.

În cazul constatării ulterioare a unor declaraţii false, voi suporta sancţiunile administrative, respectiv, *anularea examenului de licenţă.*

Data   Nume, Prenume

__________   __________________

Semnătura

De citit înainte (această pagină se va elimina din versiunea finală):

1. Cele trei pagini anterioare (foaie de capăt, foaie sumar, declaraţie) se vor lista pe foi separate (nu faţă-verso), fiind incluse în lucrarea listată. Foaia de sumar (a doua) necesită semnătura absolventului, respectiv a coordonatorului. Pe declaraţie se trece data când se predă lucrarea la secretarii de comisie.

2. Pe foaia de capăt, se va trece corect titulatura cadrului didactic îndrumător, în engleză (consultaţi pagina de unde aţi descărcat acest document pentru lista cadrelor didactice cu titulaturile lor).

3. Documentul curent **nu** a fost creat în MS Office. E posibil sa fie mici diferenţe de formatare.

4. Cuprinsul începe pe pagina nouă, impară (dacă se face listare faţă-verso), prima pagină din capitolul *Introducere* tot aşa, fiind numerotată cu 1.

5. E recomandat să vizualizaţi acest document şi în timpul editării lucrării.

6. Fiecare capitol începe pe pagină nouă.

7. Folosiţi stilurile predefinite (Headings, Figure, Table, Normal, etc.)

8. Marginile la pagini nu se modifică.

9. Respectaţi restul instrucţiunilor din fiecare capitol.

Contents

Chapter 1 Introduction - Project Context **11**
1.1 Project Context 11
1.2 Motivation 11

Chapter 2 Project Objectives and Specifications **13**

Chapter 3 Bibliographic research **14**
3.1 C Programming Language 14
3.2 Python Programming Language 15
3.3 Technologies used 16
3.3.1 Lex 16
3.3.2 Yacc 16
3.3.3 .NET Framework 16
3.3.4 Windows Forms 17

Chapter 4 Analysis and Theoretical Foundation **18**
4.1 Editor Application 18
4.1.1 Model-View-Controller(MVC) Architecture 18
4.2 Interpreter 20
4.2.1 Lexical Analyzer 21
4.2.2 Syntactic Analyzer 22
4.2.3 Parse Tree 22

Chapter 5 Detailed Design and Implementation **23**

Chapter 6 Testing and Validation **24**
6.1 Title 24
6.2 Other title 24

Chapter 7 User's manual **25**
7.1 Title 25
7.2 Other title 25

Chapter 8 Conclusions **26**
8.1 Title 26
8.2 Other title 26

Bibliography **27**

Appendix A Relevant code **28**

Appendix B Other relevant information (demonstrations, etc.) **29**

Appendix C Published papers **30**

Chapter 1

Introduction - Project Context

1.1 Project Context

Computer science and programming is taught in schools around Romania for at least 30 years, especially in high schools, but also in secondary schools, starting with the 5^{th} grade. Before introducing directly to a programming language, many teachers use a pseudocode language which serves as a mean of understanding programming concepts in a more universal manner, bringing it closer to the natural spoken language. I have decided to make an implementation of this pseudocode by creating an editor and interpreter for it.

The purpose of this project is to create an easier way of learning programming concepts for students who are new into this domain. This will serve as a fresh renewal of software used in schools today, as older tools such as Code::Blocks and/or Free Pascal are still used in schools and programming contests.

SueC is the name of the editor which creates, edits and compiles files which represent pseudocode files. This editor will work also like any other editors, providing some error-checking mechanisms and returning the result after compiling a pseudocode file.

1.2 Motivation

During high school, many of my colleagues have struggled learning programming and computer science as they had issues in understanding the simple paradigms because of C programming language. They have improved throughout the high school due to the teacher using pseudocode as a mean of explaining simple algorithms and paradigms, but there were some struggle shown for some when changing the pseudocode into implementations in C.

Nowadays, this issue is still persistent in schools in Romania as C and Pascal are used as main programming languages for teaching, exams and computer science contests. There are some more interactive programming languages such as Scratch which uses a graphical interface for implementing simple programs, but since the target audience is for primary school students, there is a need for an attractive way of making secondary and high school students for understanding programming at their age group.

At the moment, there are platforms for learning code such as CodeCademy and Udemy which contain basic courses for people at every age, but the main focus is for people who have a little background in programming and computer science.

Chapter 2

Project Objectives and Specifications

As the title of the project suggests - "An Editor and Interpreter for Pseudocode" - this is an application which will serve as an educational tool for using the pseudocode as a programming language.

For the users of this application(students and/or teachers), the main functionalities of this application are:

- Creating/opened a file in which pseudocode can be implemented.
- Writing pseudocode in the file created/opened.
- Compiling the file and obtaining the desired result or error(s) if there are occured.
- Running some step-by-step basic tutorials which are aimed for learning the language.

The main objectives of this project are:

- Developing an user-friendly application which handles the main file handling operations and communicating with the compiler of the pseudocode source files.
- Creating an understandable programming language that resembles the pseudocode used by teachers in schools and/or universities. For a technical point of view, the pseudocode will be created like any other programming languages, having similar elements to existing ones that are used nowadays, but also with specific structural elements bringing it closer to the natural language.
- Developing a compiler for this programming language by defining a lexical and syntactic analyzer respectively. These analyzers contain the set of rules that apply to the programming language.

Chapter 3

Bibliographic research

For this project, my research done was focused on the main components and technologies included in the project:

1. **C Programming Language**
2. **Python Programming Language**
3. **Lex**
4. **Yacc**

3.1 C Programming Language

C is a general-purpose, procedural computer programming language supporting structured programming, lexical variable scope, and recursion, while a static type system prevents unintended operations. This programming language was created between 1972 and 1973 as a way of making utilities work in Unix operating system, later being used for reimplementing the kernel of this OS. Since 1980s, C has gained enough popularity becoming one of the most widely used programming languages in the world. During this time, there were several C compilers created by several vendors for being available for the majority of existing computer architectures and operating systems. Since 1989, C has been standardized by ANSI (American National Standards Institute) and by the International Organization for Standardization (ISO).

Being an imperative procedural language, C was designed to be compiled using a relatively straightforward compiler to provide low-level access to memory and language constructs that map efficiently to machine code instructios all with minimal runtime support. This language supports cross-platform programming, making it available in numerous platforms, from embedded microcontrollers and supercomputers. It also stood as a big influence in the creation of other programming languages, such as:

- C++

- C#
- Java
- Python
- Go

The C programming language syntax is defined by a formal grammar, having specific keywords and rules based on statements to specify different actions. The most common statement is an expression statement, consisting of an expression to be evaluated followed by a semicolon. The main structure of a C program consists of declarations and function definitions, which in turn contain declarations and statements.

Besides exxpressions, the main sequence execution of statements can contain several control-flow statements defined by reserved keywords:

- Conditional execution

  This is defined by *if* and *else* statements. These statements contain an expression that the *if* checks if it is true or not and execute statements based on a condition.

  Alongside those statements, there exists the *switch* statement in which it displays a *case* based on the expression given.

- Iterative execution (Looping)

  This is defined by *while* , *do-while* and *for* statements which can loop through a certain set. The *for* statement contains separate expressions for initialization, testing and reinitialization, any of which can be omitted.

3.2 Python Programming Language

Python is an interpreted, high-level, general-purpose programming language with the aim to help programmers with clear, logical code for small and large-scale projects. It was conceived in the late 1980s as a successor to ABC language, but it was released in 1991. Python is dynamically typed and garbage-collected, supporting multiple programming paradigms, such as: procedural, object-oriented and functional. Due to this and its comprehensive standard library, Python is often described as a "batteries included" language.

Due to supporting multiple programming paradigms, Python is used in a lot of domains. Object-oriented programming and structured programming are fully supported, but it also includes features from functional programming and aspect-oriented programming. Other paradigms can be supported by Python via extensions, including even logic programming and design by contract.

Python is meant to be an easily readable language due to its syntax and semantics. The format is visually uncluttered, using English keywords more often than punctuation. Curly brackets are not used to delimit blocks and semicolons are optional. For block delimitation, whitespace indentation is used. A decrease in indentation shows that the current block of code is finished. With this method, it is shown that the program's visual structure accurately represents the program's semantic structure.

3.3 Technologies used

3.3.1 Lex

Lex/Flex is a computer program designed for generating lexical analyzers. It is the standard lexical analyzer generator on many Unix systems, having an equivalent tool as part of the POSIX standard. It is commonly used with *yacc* parser generator.

Lex reads an input stream specifying the lexical analyzer and outputs source code implementing a lexer in C. Older versions of Lex could also generate lexers in Ratfor.

3.3.2 Yacc

Yacc*(Yet Another Compiler-Compiler)* is a computer program for the Unix operating system. It is a LALR*(Look Ahead Left-to-Right)* parser generator, which generates a parser, the part of a compiler that tries to make syntactic sense of the source code.

3.3.3 .NET Framework

.NET Framework is a software development framework created by Microsoft and used for creating, compiling and distributing Windows and Web applications. It includes several technologies, such as ASP, LINQ, SQL, WPF, UWP that can be implemented in different programming languages, mainly: C#, C++, F#, Visual Basic, showing that there are implementations for objected oriented and functional programming. Besides the technologies, .NET framework holds a set of APIs and a runtime environment similar with Java Virtual Machine (JVM), called Common Language Runtime(CTR).

All the programming languages included in the framework respect the OOP rules via the introduction of a Common Type System (CTS) which defines all possible data types and programming constructs supported by CTR and how they may or may not interact according to Common Language Infrastructure (CLI) specification. These rules include: the creation of classes, interfaces, references, data types, SOLID principles, exception handling etc.

3.3.4 Windows Forms

Windows Forms represents the graphical interface API included in .NET framework. A Form in .NET is represented in Windows applications as a window.

A Form is an object which show properties that define the graphical interface and methods of responding to the events of user interface. By creating code for handling the events and setting the properties, objects are created based on the requirements needed for the application. One form contains a title bar, navigation bars, a menu, toolbars and a client zone for defining the event operations.

Chapter 4

Analysis and Theoretical Foundation

4.1 Editor Application

4.1.1 Model-View-Controller(MVC) Architecture

Model-View-Controller (MVC) is a software design pattern commonly used for developing user interfaces which divides the related program logic into three interconnected elements. It is used mainly for designing the layout of a page (Desktop or Web). Although it has been traditionally used for desktop applications, this pattern has become popular in designing web applications. There are specified MVC frameworks for web and mobile application development in popular programming languages like: Java, C#, Swift, Python, Ruby, JavaScript and PHP.

The main components of this design pattern are:

- **Model**

  It represents the central component of the pattern, being the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application.

- **View**

  It represents the visual element of the pattern, being any representation of information possible, like: charts, tables, diagrams, pages, forms etc.

- **Controller**

  Accepts input from the view and converts the data obtained into commands that are directed to the model and/or view.

Besides the division of the application into these components, this pattern defines the interactions between the components:

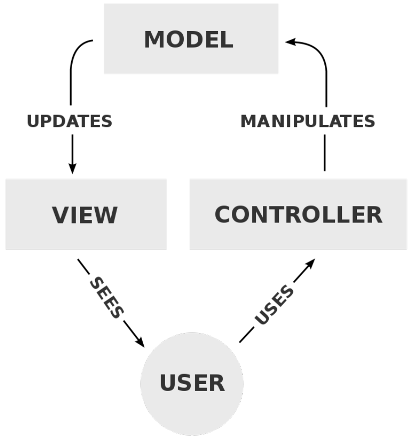

Figure 4.1: Model-View-Controller Diagram

- The model of the application is responsible for managing the data of the application. The user input is received from the controller.
- The view represents a presentation of the model having a particular format.
- The controller responds to the user input given in the view(s) and performs interactions on the data model objects.

Model-View-Controller pattern offers different advantages in regards to developing applications, such as:

- **Simultaneous development over the code**

  Multiple developers can work simultaneously on the model, controller and views, having no change in the actual structure of the system.

- **High cohesion**

  The cohesion is defined in computer programming as the degree to which the elements inside a module belong together. In other terms, it acts as a measure of the strength of relationship between the methods and data of one class and some unifying purpose or concept served by the class.

  In object-oriented programming, the term ”cohesion” is used quite frequently together with coupling. The methods that serve in a class tend to be similar in many aspects, then that class is said to have high cohesion. A highly cohesive system has a manageable complexity, due to the increase of code readability and reusability.

  This pattern presents this characteristic by having logical groupings of related actions in one controller altogether. Also, multiple views that are associated to a model can be grouped together.

- **Loose coupling**

  Besides the cohesion, which serves the degree of "togetherness" of elements inside a module, the coupling is described as the degree of independence between software modules i.e. the strength of the relationships between modules between a system.

  A loosely coupled system is one which each of its components makes use of little or no knowledge of the definitions of other separated components. The main subareas include the coupling of classes, interfaces, data and services. Loose coupling goes hand-in-hand with high cohesion by having an manageable complexity and the ease of use of alternative implementations that provide the same services.

  In the case of Model-View-Controller pattern, its nature and workflow shows the existence of a loose coupling between the Model, View and Controller of the application.

4.2 Interpreter

An interpreter is a computer program that directly executes instructions written in a programming or scripting language, without requiring them to have been compiled into a machine code language program. For program execution, an interpreter uses one of these strategies:

- Parsing the source code and performing its behavior directly;
- Translating the source code into efficient intermediate representation and immediately execute this;
- Explicitly execute stored precompiled code made by a compiler which is part of the interpreter system.

Historically speaking, interpreters have been used since 1952 to ease programming within the limitations of computers existing at that time. Another usage of them was to translate between low-level machine languages, allowing code to be written for machines that were still under development and tested on computers that already existed. The first interpreted high-level language was Lisp. Nowadays, programs written in a high-level language are either directly executed by some kind of interpreter or converted into machine code by a compiler for the CPU to execute.

There are some differences between a compiler and interpreter, mainly in the functionality. A compiler works most of the time with an assembler and linker. It produces machine code most of the time to be executed by the computer hardware, but it can often produce object code, an intermediate form. An object code is the same machine code created before, but with the addition of a symbol table containing names and tags to make executable blocks (or modules) identifiable and relocatable. The linker comes into working by combining these object file(s) with libraries that are included in the compiler in order to create a single executable file. On the other hand, a interpreter written in a low-level language may have similar machine code blocks implementing functions of the high level language stored and executed when a function's entry in a look up table points to that code. However, an interpreter written in a high level language uses another approach, such as: generating and walking a parse tree, generating and executing intermediate software-defined instructions or both approaches.

Both compilers and interpreters generally turn source code into tokens, generate a parse tree. The basic difference is that a compiler system (along with a linker) generates a stand-alone machine code program, whereas an interpreter system performs the actions described by the high level program.

For this project, this interpreter is written in C, a high level language, being split into three parts:

4.2.1 Lexical Analyzer

Lexical analysis (or tokenization) is the process of converting a sequence of characters into a sequence of tokens. A program that performs lexical analysis is called a lexer or tokenizer.

In modern processing, a lexer forms the first phase of a compiler frontend, occuring mostly in one pass of the source code. A lexer is used alongside with a parser in most compilers and interpreters. It splits the source code, sentence by sentence, into tokens. Tokens are strings with an assigned meaning, being structured as a pair consisting of a token name and an optional token value. The token names can generally be split into:

- *Identifiers* - Names that a programmer/developer chooses.
- *Keywords* - Names that are already defined in the programming language
- *Separators/Punctuators* - Punctuation characters and paired-delimiters
- *Operators* - Symbols that operate on arguments and produce results
- *Literals* - Numeric, logical, textual, reference literals
- *Comments* - Line, block comments

When a statement is given as a source, the lexer splits every element of the statement, creating one token per element. For example, we consider a C expression:

*c = a * (b + 5);*

The lexer gets this statement, passes through its all defined lexems (the source program that matches the pattern for a token) and creates the token based on their lexem appartenance. In this case, the statement becomes:

- (identifier, c)
- (operator,=)
- (identifier,a)
- (operator,*)
- (separator,()
- (identifier,b)
- (operator,+)
- (literal,5)
- (separator,))
- (separator,;)

The tokens obtained are, then, passed to the parser.

4.2.2 Syntactic Analyzer

Syntactic analysis (or parsing) is the process of analyzing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar. In computer science, a parser is a software component that takes input data and builds a data structure - often a parse tree, abstract syntax tree or other hierarchical structure, giving a structural representation of the input while checking for correct syntax. In most cases, a lexer works hand-in-hand with a parser.

After obtaining the tokens from the lexer, each token is analyzed based on its token name and creates a parse tree which is needed for passing through in order to get an order of operation handling, giving the response.

4.2.3 Parse Tree

The parse tree is the result of parsing the tokens from the lexer. Passing the tree determines the order of the operations done, giving the results. There are two ways of performing the pass of the tree:

- **Top-down parsing**

  This method can be viewed as an attempt to find the left-most derivations of an input stream by searching for parse trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right.

- **Bottom-up parsing**

  A parser can start with the input and attempt to rewrite it to the start symbol. In this case, the input is represented by the leaves of the parse tree, being the most basic elements. The best example of this case are the LR(Left-to-right Rightmost) parsers, which analyze deterministic context-free languages in linear time.

Chapter 5

Detailed Design and Implementation

Together with the previous chapter takes about 60% of the paper.

The purpose of this chapter is to document the developed application such a way that it can be maintained and developed later. A reader should be able (from what you have written here) to identify the main functions of the application.

The chapter should contain (but not limited to):

- a general application sketch/scheme,
- a description of every component implemented, at module level,
- class diagrams, important classes and methods from key classes.

Chapter 6

Testing and Validation

About 5% of the paper

6.1 Title

6.2 Other title

Chapter 7

User's manual

In the installation description section your should detail the hardware and software resources needed for installing and running the application, and a step by step description of how your application can be deployed/installed. An administrator should be able to perform the installation/deployment based on your instructions.

In the user manual section you describe how to use the application from the point of view of a user with no inside technical information; this should be done with screen shots and a stepwize explanation of the interaction. Based on user's manual, a person should be able to use your product.

7.1 Title

7.2 Other title

Chapter 8

Conclusions

About. 5% of the whole
Here your write:

- a summary of your contributions/achievements,
- a critical analysis of the achieved results,
- a description of the possibilities of improving/further development.

8.1 Title

8.2 Other title

Bibliography

[1] E. Bellucci, A. Lodder, and J. Zeleznikow, "Integrating artificial intelligence, argumentation and game theory to develop an online dispute resolution environment." in *16th International Conference on Tools with Artificial Intelligence*, 2004, pp. 749–754.

[2] G. Antoniou, T. Skylogiannis, A. Bikakis, M. Doerr, and N. Bassiliades, "Dr-brokering: A semantic brokering system." *Knowledge-Based Systems*, vol. 20, no. 1, pp. 61–72, 2007.

[3] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards, *Artificial intelligence: a modern approach*. Prentice hall Englewood Cliffs, 1995, vol. 2.

[4] W. Strunk, Jr. and E. B. White, *The Elements of Style*, 3rd ed. Macmillan, 1979.

Appendix A

Relevant code

```
 /** Maps are easy to use in Scala. */
object Maps {
  val colors = Map("red" -> 0xFF0000,
                   "turquoise" -> 0x00FFFF,
                   "black" -> 0x000000,
                   "orange" -> 0xFF8040,
                   "brown" -> 0x804000)
  def main(args: Array[String]) {
    for (name <- args) println(
      colors.get(name) match {
        case Some(code) =>
          name + " has code: " + code
        case None =>
          "Unknown color: " + name
      }
    )
  }
}
```

Appendix B

Other relevant information (demonstrations, etc.)

Appendix C

Published papers