**Members:**

1. *Vinh Tran*
2. *Quang Nguyen*

**Assignment 2 - Documentation**

1. **Problem Statement:**

The problem is to create a program which mainly analyzes the syntax of a partial source code from user’s input or the source file. The result will be written/displayed either on the console or on the output file or both.

1. **How to use the program:**
2. Select to run the program either on Windows or on Linux
3. Extract the zip file into the folder
4. The folder includes some source files:

* Main.cpp
* SyntaxAnalyzer.h
* LexicalChecking.h
* LinkedList.h
* Tools.cpp
* Tools.h
* input.txt

1. **Windows - Microsoft Visual Studio (Recommended)**

* Open Microsoft Visual Studio (Our program is coded in Microsoft Visual Studio 2019, so it is better if the program is executed on the same version)
* Choose “Create a new project”
* Click “Empty Project” in C++ Language, and then click “Next”
* Name your project (e.g. Assignment2), and then click “Create”
* Press “Shift + Alt + A” to add existing items into the project
* Select all source files (except ‘input.txt’ file) from the folder, and then click “Add”
* Press “Ctrl + Shift + A” to add a new item into the project
* Click “Utility”, and then click “Text File (.txt)”
* Name the text file (e.g. test.txt) and click “Add”
* Open “test.txt” file from “Source Files” in “Solution Explorer” section. Write a block of the code or just copy the content of ‘input.txt’ file
* Open “Main.cpp” file (the executable file) from “Source Files” in “Solution Explorer” section
* Either press F5 or click on “Local Windows Debugger” to run the program

1. **Linux - Tuffix**

* (Optional) Create a text file in the folder and write a block of the code. Or, use ‘input.txt’ file when the program is executing to do the syntax analyzer
* Open Terminal
* Change the current directory to the directory which contains the folder of the program (e.g: /home/student → /home/student/Assignment2)
* To compile the files, command: $ c++ -std=c++17 Main.cpp Tools.cpp
* To run the program, command: $ ./a.out

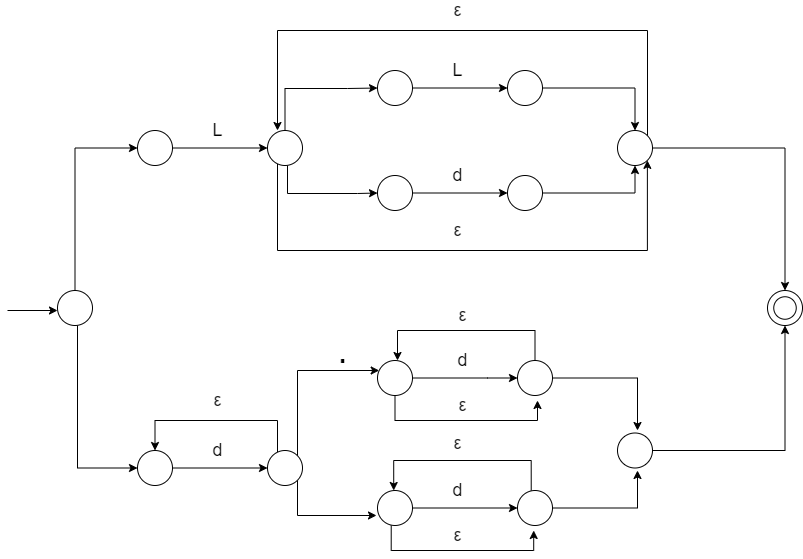
1. **Design of the program**

- The program was designed to be able to accept the source code (except for a block of the comment if any) by either reading the user’s input from the keyboard or reading from the input file. The program starts doing the lexical analyzer first, and then it starts doing the syntax analyzer.

1. **Lexical Analyzer was designed by the following process:**

- Since the lexical analyzer returns two values (a token and its relevant lexeme) as the output, we designed and implemented Linked List, which stores them at the same time.

- We also designed and implemented a Finite State Machine (FSM) to analyze tokens such as IDENTIFIER, INTEGER, and REAL for some lexemes based on Regular Expressions (RE).



**RE = L(L | n)\* | n+(.n\* | n\*)**

|  |  |  |  |
| --- | --- | --- | --- |
| **FINITE STATE MACHINE**  **(FOR IDENTIFIER, INTEGER, AND REAL)** | | | |
| **N:** | **Letter**  **[a...z][A...Z][$]** | **Digit**  **[0...9]** | **Other**  **[.]** |
| → **0** | 1 | 2 | 0 |
| **1** | 1 | 1 | 0 |
| **2** | 0 | 2 | 3 |
| **3** | 0 | 3 | 0 |

(Note: In the program, there are two string variables named ‘*testChar*’ which is used to go through each character of a line of the code and ‘*word*’ which combines one by one character of ‘*testChar*’ into a string.)

1. Starts analyzing one character at a time of a line of code.
2. The character will be assigned to the ‘*testChar*’ variable.
3. ‘*testChar*’ will be checked to see if it is a white space, an operator, or a separator.

If no, ‘*testChar*’ will be put into the ‘*word*’ variable, and then move to the next character.

If yes, before that character will be stored into the Linked List, ‘*word*’ will be checked if it contains any string, integer, or real. If yes, the program calls lexer() function (FSM) and passes ‘*word*’ as a parameter.

1. In the lexer() function, ‘*word*’ will first check if it is a keyword or not. If no, it goes through the finite state machine (FSM) to do the lexical analyzer to check if it is an integer, a real or an identifier. In FSM, state 0 is implied as the starting state. State 1 is implied as an identifier. State 2 is implied as an integer, and state 3 is implied as a real number.
2. The machine will start at state 0 (ENTRY) then it will enter a for loop to go through each character of the string variable ‘word’. Therefore, the state will be changed by depending on the character of the string.
3. After going through the string and determining tokens and lexemes, the program will put two values into the Linked List called ‘*list*’.

(Lexical Analyzer repeats the process from 1 to 6 until it reaches the end of a line of the code or the end of a file.)

1. **Syntax Analyzer was designed by the following process:**

- We defined the grammar for this assignment by G = (T, N, S, R), where

T = {+, -, \*, /, =, >, <, id, int, float, bool, true, false, while, if, else, then, do, whileend, endif}

N = {<Expresson>, <Term>, <ExpressionPrime>, <TermPrime>, <Factor>, <ID>, <Num>, <BoolType>, <Statement>, <Assign>, <Declarative>, <Type>, <Conditional>, <Relop>}

S = <Statement>

For R, we used several rules for this assignment:

1. Rules for the arithmetic expressions after removing left recursions:

<Expression> → <Term> <ExpressionPrime>

<ExpressionPrime> → + <Term> <ExpressionPrime> | - <Term> <ExpressionPrime> | Epsilon

<Term> → <Factor> <TermPrime>

<TermPrime> → \* <Factor> <TermPrime> | / <Factor> <TermPrime> | Epsilon

<Factor> → <ID> | <Num> | <BoolType>

<ID> → id

<Num> → int | float

<BoolType> → true | false

1. Rules for the assignment statement:

<Statement> → <Assign>

<Assign> → <ID> = <Expression>;

1. Rules for the single declarative statement:

<Statement> → <Declarative>

<Declarative> → <Type> <ID>

<Type> → int | float | bool

1. Rules for the while and if else statement:

<Statement> → if <Conditional> then <Statement> else <Statement> endif

<Statement> → while <Conditional> do <Statement> whileend

<Conditional> → ( <Expression> <Relop> <Expression> ) | ( <Expression> )

<Relop> → > | <

- In the program, we constructed Recursive Descent Parser (RDP) to do the syntax analyzer; therefore, we implemented some functions, such as statement(), expression(), term(), expression\_prime(), factor(), term\_prime(). Each function will have its own task by checking if the source code follows the rules. If yes, then display the rule(s) of each token. Otherwise, display the syntax error at that token.

(The program analyzes the syntax until it reaches the last element of a list which contains two values from Lexical Analyzer or encounters the syntax error of the source code. Finally, the program will print out the result of syntax analyzers on screen and copy and write it into a destination file called ‘*output.txt*’ as default.)

1. **Limitation & Shortcoming:**

- For doing the syntax analyzer about “while” and “if else” statement, there are several bugs occurring while the program is executing. Also, the program only works if <Relop> is “>” or “<”. Otherwise, the program will give the bad result or crash if <Relop> is “>=”, “<=” or “==”.