Samprayash Dahal

Gonzalo Plascencia

Andrew Ta

CPSC 323

**PROJECT DOCUMENTATION**

**1. Problem Statement**

The second assignment is focused on writing a syntax analyzer . The main goal is to create a program that should read a file containing the source code given from class in order to rewrite all the grammar into their respective production rules into an output file.

**2. How to use the program**

1. The program has only been tested to run on windows

2. We used the IDE Visual Studios (not xcode or others)

3. You will need to place a text file in the same directory as the executable file and enter the file name when you run the program (ex: test1.txt)

4. To run the program, just simply double click on the Project2.exe

5. The result will be displayed on the screen, and also be saved in the same directory as the input and executable with name “output.txt”

**3. Design of the program**

**Header File**

The program starts with initializing all the variables through an enum struct. LETTER, ID\_DIGIT, UNDERSCORE, INT\_REAL\_DIGIT, and PERIOD are declared inside two struct enums.

The program will then initialize its final state table in order to find where each final state will take place. It will also create a struct, which defines the necessary variables needed to get the tokens.

The vector lexer function in the header file that is used in the while loop parses the string expression from a text file using the FSM in order to isolate each individual token and lexeme name in the text file. The functionality behind this function involves “pushing” every symbol into the vector until it is finished. It also makes sures that the last token gets saved when the loop ends, ensuring the string to be saved with its respective lexeme. The function also returns the string equivalent of an integer lexeme token type. In this function, for loops and if statements are used to determine if the token is an INTEGER, REAL, OPERATOR, LETTER, SEPARATOR, and UNKNOWN.

The get\_ID\_column and get\_IntReal\_column functions determine the state of the type of character that is being examined and then returns the column variable, which contains what character the token is.

The bool checkifReal function checks if the character is an integer or a real value. Then, the printTokens and printHeader functions treat as a print function, printing all the tokens and lexemes in order to the output file.

**SA\_main.cpp**

The program will start by asking the user for a filename to check if the file exists in the computer’s hardware. If it does, the file will open and the program will start its functionality and display the respective tokens and lexemes within the desired file; however, if it fails and the text file does not exist, an error message will be displayed, ending the program.

In the while loop with the file, it will call the vector “lexer’ and print all of the tokens and lexemes onto the output file while also printing a success message, so the user knows that the program is a success.

Then, the program will use the lex\_Adv function to identify its grammar. The function will move token by token until the file is done.

The rest of the functions in the program are helper functions, in which they help identify the grammar for each token and remove any left recursion that is in place. Using if statements and using multiple functions to test if the conditions of each token are met, we are able to figure out the grammar for all the tokens.

Lastly, we use the Statement function to print out the grammar for its respective token.

DFSM for Identifiers:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | “ “ | D | “.” | $ | L | ! | O |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 2 | 0 | 0 |
| 2 | 3 | 2 | 0 | 2 | 2 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

DFSM for integers(D):

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | “ “ | D | “.” | $ | L | ! | O |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 2 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 2 | 0 | 0 | 0 | 0 | 0 |

DFSM for Real:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | “ “(space) | D(igit) | “.” | $ | L(etter) | ! | O(perator) |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 2 | 3 | 0 | 0 | 0 | 0 |
| 2 | 0 | 2 | 3 | 0 | 0 | 0 | 0 |
| 3 | 0 | 4 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 4 | 0 | 0 | 0 | 0 | 0 |

**REs:**

Identifier: (L(L|D)\*) | (L(L|D)\*$)

Integer: D+

Real: D\* . D+

**4. Any Limitations**

There is a limitation to the program. The program will start by asking the user for the name of the text file. The text file must have correct documentation and format for the program to recognize it. Also, since we have done the program solely on WINDOWS 10, we are not sure if the program will run on different devices such as the Apple MAC or a LINUX operating system.

**5. Shortcomings**

NONE