**CPSC323 Assignment 1 Documentation**

**Lexical Analyzer**

1. **Problem Statement**

This first assignment is to write a lexical analyzer. A major component of this assignment is to write a procedure, lexor(), that returns a token when it is needed. The lexor() should return a record, one field for the token and another field for the actual value. The program should read a file containing the source code given from class to generate tokens and write out the results to an output file. Do at least three test cases and make sure you turn in proper documentation using the documentation template.

1. **How to Use the Program**

This project contains two submissions written in different languages, C++ and C#. Here are instructions for using both versions of the program.

**C++:**

1. Extract the compressed folder to a location of your choice.

2. Open the Windows 10 Command Prompt

3. Navigate to the directory where the files are located.

4. Navigate inside the "exe" directory (ex. LexAnalyzerEarlySubmission/C++/exe)

5. Run the exe file from the command line and pass in the input and output files as parameters.

USAGE: "CPSC 323 Lexical Analyzer.exe" [InputFileName] [OutputFileName]

Example: "CPSC 323 Lexical Analyzer.exe" sample.txt output.txt

**C#**

1. Extract the compressed folder to the directory of your choosing.

2. Open the Windows 10 Command Prompt

3. Navigate to the location where the folder was extracted.

4. Navigate inside the exe folder.

5. Place input file into this directory (input file must be name input.txt).

6. Run the following command to run the executable file: LexicalAnalyzer.exe

-Note: In the current state, the program will look for an input file named "input.txt" within the same directory. The program will not run correctly if there input file is not in the directory or named differently

-Note: There are no parameters that are passed when running the program.

-Notes: The output will be written to a file named "outputFile.txt" in the same directory.

1. **Program Design**

This section covers the basic design of the program. Both versions of the program (C++ and C#) are identical regarding the major components, data structures, and algorithms used so this section will cover both versions.

**Before Parsing for Lexors:**

Before the program begins parsing the lexors, the source file is opened and copied into a character vector. The file is copied into a character vector to avoid any possible file stream related issues. Additionally, a vector can access elements at a specific location and traverse through elements to make navigation easier.

**Parsing Lexors:**

Parsing for lexors begin after the source file has been loaded into the character vector. The parsing component is comprised of several states to replicate a finite state machine: Neutral, Comment, Integer, Real, and Identifier. Operators and Separators do not have their own state because they are only comprised of a single character, and usually used to indicate the end of the current state. The program will loop through the character vector, taking one character per iteration, and processes it based on its current state. A string data type is used to build integers, reals, keywords, and identifiers as the characters are processed. Once a lexor token has been processed, a structure called Record is used to save the results. The Record structure contains a string for the Token name and another string for the Token value. After a Record is created, it is stored into a Records vector and can be retrieved using the lexor() method.

**Neutral State:**

The Neutral state indicates that no other states are valid. This state is used for the first iteration, and when another state is done processing a lexor token. The purpose of the neutral state is to take the next character from the vector and determine what the next state will be. If the machine is in the Neutral state, it can transition to the Comment, Int, or Identifier states. If an Operator or Separator is read while in the Neutral state, it will be processed and remain in the Neutral state. The Neutral state will transition to the comment state if a ‘!’ is read, will transition to the Int state if a numerical value is read, and will transition to the Identifier state if an alphabetic character is read.

**Comment State:**

The Comment state indicates that a comment is being processed and no inputs are saved until a closing ‘!’ is inputted. The program returns to the Neutral state once the comment is finished.

**Int State:**

The Int state indicates that a numeric string is being built. During each valid iteration, the numeric character will be appended to a string to save that tokens value. The string will only accept numeric values or a ‘.’ character. If a ‘.’ character is read, the character will be appended to the string, and the machine will transition to the Real state. If an Operator or Separator character is read, the string will be processed into a token and saved. Then the Operator or Separator character will be processed into its correct token and the machine transitions to the Neutral state. If any other types of characters are read while in the Int state, an error will be thrown.

**Real State:**

The Real state is identical to the Int state except that it treats ‘.’ as a Separator instead of a character, ending the string building loop and returning to the Neutral state. The ‘.’ will be processed as a Separator token and saved.

**Identifier State:**

The Identifier state indicates that either a keyword or identifier token. The program will read characters from the char vector until an Operator or Separator are inputted. Once an Operator or Separator are read, the program will compare the string to the array of keywords and check for a match. If a match is found, a Keyword token is created, otherwise an Identifier token is created. Then the Operator or Separator that was inputted is processed into a token before returning to the Neutral state.

1. **Project Limitations**

C++: None.

C#: None.

1. **Shortcomings**

C++: None.

C#: None.