**CS323 Documentation**

Jared Castaneda, Kurt Prutsman, David Estephanian

1. **Problem Statement**

The purpose of this assignment is to build a functional lexical analyzer using a Finite State Machine. Additionally, Regular Expressions for the identifiers and integers are provided, as well as a Non-Deterministic Finite State Machine using Thompson Algorithm Method.

1. **How to use your program**

To use the program, the user must double click on the main.exe application located in the folder. Users must take note of the given .txt files inside of the folder containing the test cases. Users are also welcome to create their own test cases with Notepad.

1. **Design of your program**

Starting in main.cpp, an ifstream testFile gets initialized. This will be used to pass into two functions, getFile and scanFile. Starting with the getFile function, where the user enters the name of the file they wish to analyze. If the file exists, the file will open. If the file does not exist, the user is prompted to enter a valid file name.

Once the program is opened, it will work its way into the scanFile function. There is a temporary string to read in parts of the file. This temporary string will have any given amount of records containing tokens and lexemes, depending on how it is formatted. The vector of strings called recordList calls the lexer function with the temp string to return the records. Once the temp string has been completely analyzed and the records are in place inside recordList, they are outputted until there is nothing left in the vector recordList. This happens in a while loop, allowing one line to be read into temp until the end of the file.

The lexer() function is the biggest part of the lexical.cpp file. The lexer() function uses five other functions in the cpp file, so we will describe those first, and then describe the lexer() itself.

There is a bool function called isSeperator() that is used in the lexer function to determine if a character is a separator or not. The function takes in a character, runs it through a for loop, and using an if statement, compares the character to known symbols. Similarly, there is another bool function, isOperator(), that takes in a character and compares it to known operators.

There are also two int functions char\_to\_col\_int() and char\_to\_col\_id(). A character gets passed inside. They are used to return the column number of character in the table. This gets used inside the lexer() later on.

**The regular expressions we used were (d)^+ for integers and ( l | \_ ) ^+ for identifiers. This is important to how we designed the tables inside of the DFSM functions.**

The lexer is what will return the vector of records back into the main function. It stores strings and takes in a string input. There are four strings declared at the beginning of the function: token, lexeme, result, temp. Additionally, there is a string vector declared named records. A for loop is used to go through each character of the vector. The lexer will start by analyzing the string by each character.

There are two bool checks initialized, separatorCheck and operatorCheck. It stores its result based on isSeparator and isOperator. Depending on the result, it will set the given token to its proper result (separator or operator). The lexeme is the current character being analyzed. The result adds them together into one string, separated by some whitespace. The result is pushed back onto the records vector.

`Next, we check for integers. The intDFSM returns a 1 if the string being passed in is detected to be an integer after being looked up in the given table. The starting and accepting states need to match for this to be correct. The state table was initialized by converting the NFSM to the DFSM shown at the end of the document. They will be pushed to the records as an integer token.

The last check will see if it’s an identifier or keyword. If it is not inside the list of keywords, i will be an identifier. This has a much bigger state table being involved. Inside the identifierDFSM, it will analyze each character that isn’t an operator or separator. The char\_to\_col\_id once again returns the column number needed. The starting state is compared against the accepting states with an additional boolean check. It sees if the string is a keyword or not. This involves a similar check to isOperator and isSeparator, going through multiple keywords. It will return true if the string matches. If it matches an accepting state and has this check as false, it returns 1. This signifies it is an identifier. However, if there is a match and the check is true, it will return 2, a sign for a keyword.

As things are being sent back to the scanFile function, they are printed to the terminal. They are also sent to the ofstream file userOutput. This writes the output to a text file, output.txt. The user can view the results here after running the program through an IDE or the executable file.

1. **Any Limitation**

None

1. **Any shortcomings**

Some test cases were hard to implement due to Rat20SU. Was not sure on some of the operators or keywords.

**(d)^+ for integers and ( l | \_ ) ^+ for identifiers.**

