**CS323 Documentation Assignment 2**

Jared Castaneda, Kurt Prutsman, David Estephanian

**1. Problem Statement**

This project’s purpose is to write a syntax analyzer. After rewriting the Rat20SU grammar to remove left recursion, the lexer() is used to print out an output file of tokens and lexemes found. Afterwards, it prints out the production rules used for analyzing each token. The parser also gives error messages indicating the token, lexeme, line number and error type. The error handling feature can exit or continue analysis, that way the program can parse the entire program if it is syntactically correct.

**2. 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.

**3. Design of your program**

Main.cpp, lexical.h, and lexical.cpp are reused from the first assignment. After lexical analysis and pushing everything into their respective vectors (lexVec for lexemes, tokenVec for tokens, parseList for both the lexemes and tokens formatted into one string), we assign the first userLex value to be the first entry in lexVec. This is passed into the Rat20su function to start parsing through every lexeme and token.

The parser.h and lexical.h file (from Project 1) and included in the parser.cpp file. The parser.h file contains the Rat20SU rewrite of the production rules. An integer recIndex is initialized to keep track of the records. The number will match up with lexVec, tokenVec, and parseList respectively. A bool statement is declared to determine if the rules are printed or not, determined by the user in main.

The string function lexerCall() returns the next lexeme. It checks if there’s a record in the lexer. If there is it prints the grammar rules for that record, returns that record and then moves onto the next record.

Here’s a few examples of how some of the rules work:

R2. <Opt Declaration List> ::= <Declaration List> | <Empty>

This function is called if the lexeme input is an integer or boolean, as determined in the declaration list. It displays where it’s being passed in and declares the operation rule. The lexer is not called because it’s nonterminal. The userLex is then put in the DeclarationList function. It also has the option to be empty, in which nothing else happens other than the rule being printed.

R5 <Qualifier> -> integer | boolean

This function is called if the lexeme input is determined to be an integer or boolean. If the input is determined to be an integer, it’s set as a Qualifier. If an integer is not found, the boolean is declared the Qualifier. If neither integer of boolean is determined, an error message is displayed.

R8. <Compound> ::= { <Statement List> }

This function checks if the lexeme input starts with a { bracket. If it does, it will call the lexerCall() function into the userLex to get ready for the next function. It then passes the userLex into the StatementList function, further showing the recursion of the program. After the statement list function is complete, it returns back into the compound function. It then checks the userLex string to find a corresponding “}” bracket.

R11. <Put> ::= put ( <identifier> )

This function checks the lexeme for a put statement. If a put statement is found, the userLex is updated with lexerCall. Then, the program checks for “(“. If one is found, the userLex is set to lexerCall to prepare for an identifier. Once the identifier check is done, it will check if the last character is “).” An error message is displayed if one is not found.

After going through the whole text file, it should double check to make sure that it ends with a $$. If it’s in proper Rat20SU format, then there should be no problems encountered.

**4. Any Limitation**

This program assumes a Rat20SU formatted text file.

**5. Any shortcomings**

We had trouble with some of the functions being passed in, such as with Statement List Prime leading into epsilon when more statements had to be read in. The flow of the project was hard to get properly done, as seen with the longer text files. We could not get some of the invalid tokens to throw an error if they were attempted to be read into our program. We kept getting bugs in our code inside of our lexer. The code was difficult to deal with at times.