**COVER PAGE**

CS323 Programming Assignments

**Fill out all entries 1 - 7. If not, there will be deductions!**

**Check one**

**1. Names [ 1. Jarrett Chien ], (MW [ ] or R class [ X ] )**

**[ 2. Eric Edelman ], (MW [ ] or R class [ X ] )**

**[ if 3. Michael Li ], (MW [ ] or R class [ X ] )**

**2. Assignment Number [ 2 ]**

**3. Due Dates Softcopy [ 11/12 ], Hardcopy [ 11/15 ]**

**4. Turn-In Dates Softcopy [ 11/12 ], Hardcopy [ 11/15 ]**

**5. Executable FileName [ SyntaxAnalyser.exe ]**

**(A file that can be executed without compilation by the instructor)**

**6. LabRoom [ CS-200 ]**

**(Execute your program in a lab in the CS building before submission)**

**7. Operating System [ Windows 10 ]**

**To be filled out by the Instructor:**

**GRADE:**

**COMMENTS:**

**CS323 Assignment 2 Documentation**

**1. Problem Statement**

Construct a syntax analyzer for the RAT18F language. Remove the left recursion and backtracking from the syntax rules to create at least 34/35 rules in total.

**2. How to use your program**

When you first press the executable (“SyntaxAnalyser.exe”), it asks for the file name. Enter the full filename, including extension. Once you have entered it in, it will either tell you the results of our lexer or that it’s an invalid filename that can’t be opened for reading. The results are recorded into “parseOutput.txt”. The compiler must be run in “Release” mode instead of “Debug” mode if compiling in Visual Studio.

**3. Design of your program**

The Syntax class contains a constructor that accepts a vector of Tokens from our Lexer/Parser function from the first project. It initializes our own Token vector, opens a output file (“parseOutput.txt”), prints out the first token, and initializes our private variables. In the private section, it contains an index variable for counting the position of the vector, a string vector for the rules that one token goes across, and a current Token variable. It also contains function prototypes for all rules. The functions are based on the syntax Backus-Naur form rules in the project guideline. The rules were rewritten to remove left recursion and backtracking.The Syntax class also has a variety of helper functions to the help the syntax functions operate.

One of them is a void function called nextToken(), which increments the index, updates the current Token, and outputs Token Name and Lexeme Name like in the example. This is only if the index is less than Token Vector size -1. This function is only called when we find a Match, which brings me to two other helper functions. A global string vector called numberLinesVec is used to keep track of the line number.

The first one being a boolean function called Match, which receives a string that is a expected output. It compares it to the current token’s Lexeme Name. If it’s the same, then it prints all of the corresponding rules in the token’s route. It also prints out a Match statement for the expected lexeme name and the token lexeme name. Then it calls nextToken(). Or if it’s not a match, it just returns false. This is used with terminal symbols. We have a similar function called Match\_t, which does the same thing, but compares TokenNames. Sometimes, we have to push\_back the rule before we check.

We also have an error function called syn\_error that takes in the expected value and current nonterminal. It prints out all the corresponding rules leading up to the syntax error. Then it outputs the expected value and the current Lexeme Name. It tells you which corresponding did the error occur in. It also prints out the line number, then it exits the entire program.

The Empty function handles the epsilon case and returns true. The Primary and Primary\_prime functions handles if a token is an ID, integer, expression, real, true, or false. Primary\_prime handles if the ID has epsilon. If the token is invalid, the error is printed and program terminates. The Factor function utilizes the Primary function to check validity of the input. The Term and TermPrime functions check if the token is a valid multiply or divide operator. The Expression and ExpressionPrime functions check if the token is a valid addition or subtraction operator. The Relop function checks if the current token is a valid comparison operator. The Condition function checks the syntax of the operation; if must be <Expression> <Relop> <Expression>. The functions Return and Return\_prime handle if the token is “return”, a semicolon, or an Expression. The If and If\_prime functions handle “if”, “ifend”, and “else” tokens. The Statement function calls the appropriate function (Compound, Assign, If, Return, Print, Scan, or While) based on what the token is.

The main function transfers the collection of Tokens inside a vector into the constructor of the Syntax Object. Then, it calls RAT18F() and the rest is history.

**4. Any Limitation**

Visual Studio 2017 must be run in Release mode for code to compile. Program will terminate at the first finding of a syntax error.

**5. Any shortcomings**

None.