Andrew Dinh

Joshua Hughes

David Rotter

Andrew Shin

**CS323 Documentation**

**Problem Statement**

The purpose of this assignment was to build the second phase of a compiler known as the syntax analyzer using a top-down parser or a predictive parser. The goal of the syntax analyzer is to take input from the lexical analyzer in the form of token streams. It will analyze the token streams against the production rules to detect any errors in the code and then produce an output in the form of a parse tree.

**How to use your program**

This program was written using the Rust programming language. You can install the necessary tools to run and compile the program on Windows or Linux. Installation information can be found on https://www.rust-lang.org/tools/install. Regardless of which operating system is used (Windows or Linux) the command ‘rustc main.rs’ is used in order to compile the program and then turn the file into an executable using ‘.\main.exe’ or ‘./main’ respectively. For your convenience, the Windows executable is already located under the ’..\target\release’ directory with the filename ‘cpsc\_323\_compiler\_project.exe’. Once the program runs, you will be prompted to enter the path directory of the ‘sample\_input’ and ‘sample\_output’ given in the zip file in the folder ‘src’.The program will then run and will produce the ‘sample\_output.txt’ file that will include the token, lexemes, and production rules along with the parse tree used for analyzing the tokens.

**Design of your program**

The design of our program was created using six main sections: 'main.rs', 'parser.rs', 'token.rs', 'fsm.rs', 'lexer.rs', and 'file\_handling.rs'. The 'main.rs' file includes the main function of the program that allows it to obtain file contents in the form of a string. It will then get a vector of tokens from the lexer module and print it to the file directory specified by the user. The 'parser.rs' file is the primary section of this program as it includes several functions that are structured using a predictive recursive descent parser. This includes the implementation of a constructor for our 'ParseNode' within the parse tree. The 'parse' function within the file will call the recursive functions from which those functions will call other grammar recursion functions as needed. It will start at the root node and call 'parse\_expression', the first recursive function that is called, to parse the expressions. Left recursion is removed through: <E> -> <T> <E'>, <E'> -> +<T><E'> | -<T> <E'>, <T> -> <Factor> <T'>, <T'> -> \*<F> <T'> | / <F><T'>, <F> ->

( <E> ) | <ID> | <Num>, <ID> -> id. Then, it will create a stack of rule strings and check to see if the list has been parsed successfully. The final output of this function will be a tuple of 'ParseNode' or an error message in the case of an unsuccessful parse. The 'token.rs' file includes the working code required to prettyprint the output of tokens and lexemes using enums with match. The 'lexer.rs' file includes a function to parse a given string into a vector of tokens. A 'while-loop' and 'if-statements' are utilized in order to transition through states given the current token. The 'fsm.rs' file includes the finite state machine transitions as well as the corresponding state table. The 'file\_handling.rs' file includes functions that will ask the user for the directory of the input and output files as well as writing a given vector of tokens to the given file.

**Any Limitation**

**; is not supported. Occasional logic breaks can occur with some formulations of input.**

**Any Shortcomings**

**The rule printing is mostly (90%) working.**

**The parse tree can have indentation flaws as well as occasionally being out of order.**