Andrew Dinh

Joshua Hughes

David Rotter

Andrew Shin

CS323 Documentation

1. Problem Statement

The purpose of this assignment was to create a Top-Down Syntax Analyzer using stack. We integrated functions such as scope checking, type checking, error handling, and symbol handling.

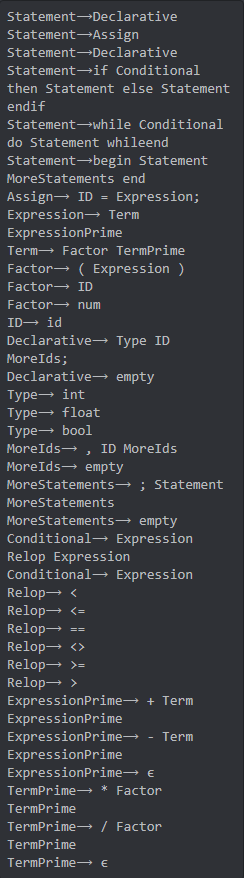
2. How to use your program

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 ‘sample\_input’ **must include** the keywords **‘begin’** at the start of the file and **‘end’** at the end of the file in order for the program to process successfully. The program will then run and will produce the ‘sample\_output.txt’ file that will include the rules, identifiers, and symbol table.

3. Design of your program

The design of our program is based off of our Assignment 2. We've maintained the same six main sections including: 'main.rs', 'parser.rs', 'token.rs', 'fsm.rs', 'lexer.rs', and 'file\_handling.rs'. The main changes we've made in order to bring the assignment inline with goals of Assignment 3 were converting our assignment 2 which used recursive descent parsing and created a top-down parser using the stack implementation. Within the parser.rs we created a hash map and symbol stack that will compare the lexer at the pointer to stack. We increment the token pointer and pop off the front of the vector stack. Functions were also included in order to handle the rules that were implemented including handling if statements, while statements, conditionals, and rules such as: Type -> float | bool | int; and Statement -> Expression | Assign. Other important functions were added such that multiple comparison operators in lexer.rs can be used. Our lexer.rs function includes a function such that we can parse a given string into a vector of tokens via the use of FSM. The function will check precursor states for validity and if it is invalid then it will continue on to the next state. We implement an if statement in order to check if the string is an identifier or keyword and then add the valid token. Our function 'fn get\_string\_type' is used to pick out keywords and return the correct transition to the FSM. We've also added functions in order to make our output easy on the eyes such as the use of enums with match in functions such as fn get\_col and fn get\_lexeme\_name.

Our Rules:



4. Any Limitations

-None-

5. Any shortcomings for each iterations

-None-