Andrew Dinh

Joshua Hughes

David Rotter

Andrew Shin

**CS323 Documentation**

**Problem Statement**

The purpose of this assignment was to build a lexical analyzer using a finite state machine. The goal of the lexical analyzer is to break syntaxes into a series of tokens while removing any whitespace or comments in the source code. Specifically, the program will test the lexer by reading a file containing source code given to generate tokens and write out the results to an output file. Using Rust as our programming language, the main goal is to write a procedure (Function) –lexer (), that returns a token when it is needed. The lexer() should return a record, one field for the token and another field the actual "value" of the token (lexeme), i.e. the instance of a token.

**How to use the Program**

This program is written in 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 the 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 fill the ‘sample\_output.txt’ file with all the tokens and corresponding lexemes. The program can then be run again for the next two sample test cases.

**Design of your program**

The core sections of the program were designed with modularity in mind. Significant components including the finite state machine, lexer, file handling, tokens, and main functions were built with this approach in order to reduce complexity and maintain flexibility. The data structures and algorithms that were used are as follows: the state table in the finite state machine was built using a two-dimensional array. The keyword ‘match’, which is similar to switch statements in C++, is used for matching characters against their current counterparts. A custom algorithm is used within a vector for identifying the current type of string in the iteration. The program uses ‘for loops’ and ‘while loops’ for cycling through the state table and program functions.

Regular Expressions:

Integer – (**^(\+|-)?\d+$)**

Real – **(((\+|-)?[0-9]+\.[0-9]\*)|([0-9]\*\.[0-9]+)|([0-9]+))**

Identifier – **(^[a-zA-Z\_][a-zA-Z0-9\_]\*$)**

**Any Limitation**

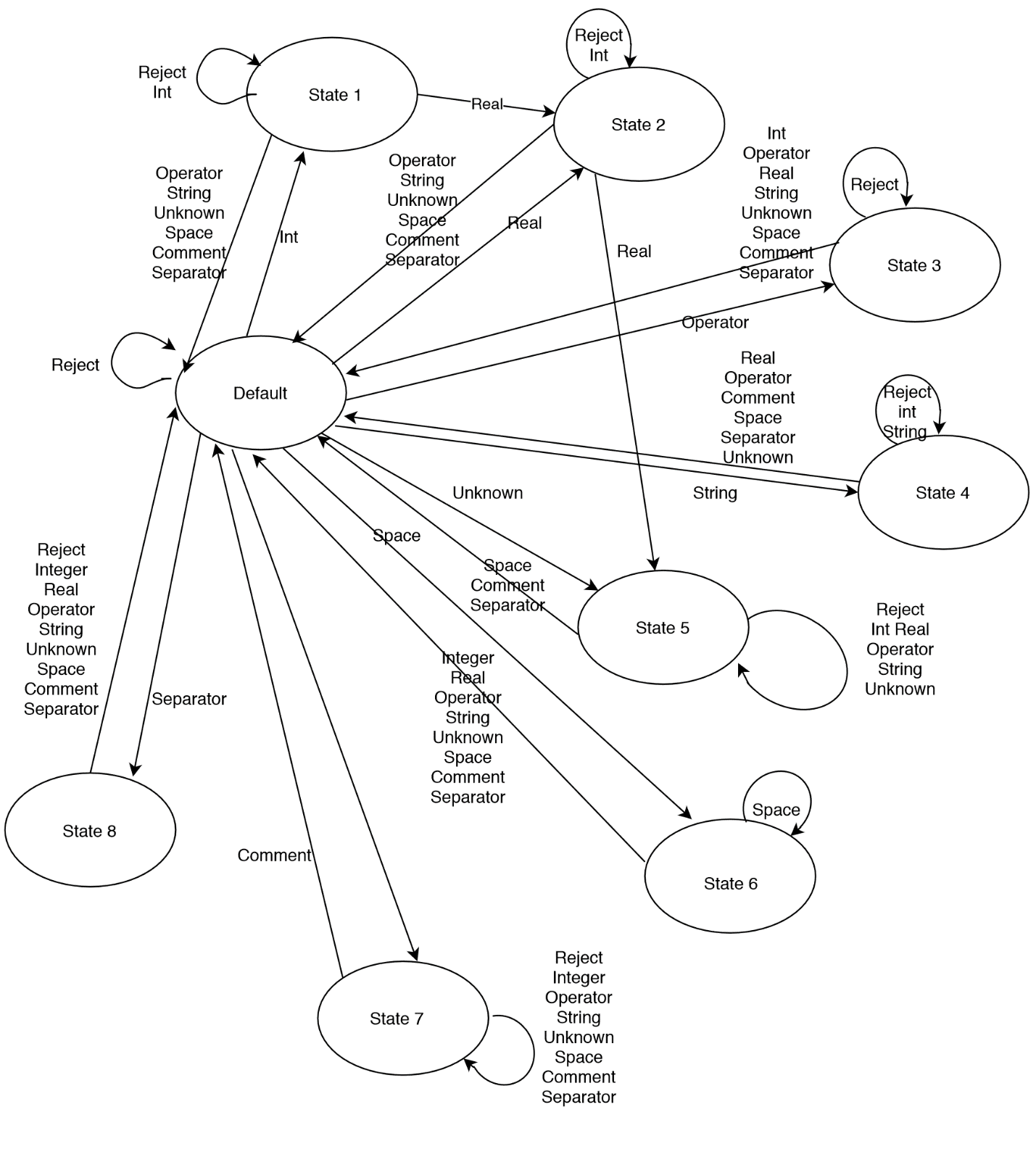
**; is not supported. Occasional logic breaks can occur with some formulations of input.**

**Any Shortcomings**

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

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

**NFSM Thompson:**

****