Assignment 1 - Lexer

John Lee

Brandon K Nguyen

Jerry Huang

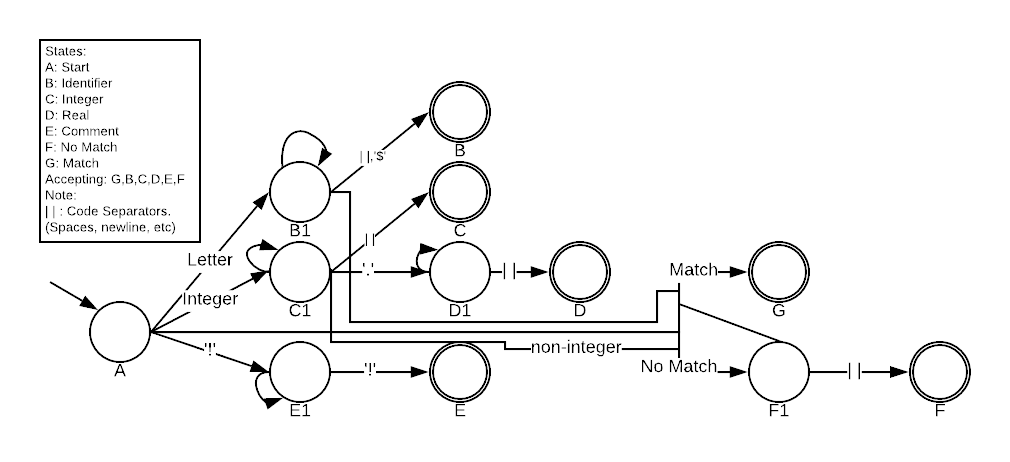
Problem Statement

This program creates a lexer() function that will tokenize source code for the Rat18S programming language. It will be able to read a file containing source code written in Rat18S and generate tokens which will then be written to an output file. Each record in the output file will consist of two parts: the token, and the lexeme, which is the actual value of the token.

How to use

To use, run the exe. “tokens.txt” must be in the same folder as the exe, as well any text files containing Rat18S source code to be scanned. When the program prompts, input the name of the txt file. After runtime, a new file called “lexeme.txt” will be created with the tokens scanned from Rat18S source code.

Program Design



|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Match | Letter | Int | ‘!’ | ‘.’ | $ | | | | Other |
| A | G | B1 | C1 | E1 | D1 | F1 | A | F1 |
| B1 | G | B1 | B1 | E1 | F1 | B | B | F1 |
| B | G | B1 | B1 | E1 | F1 | B | B | F1 |
| C1 | G | F1 | C1 | E1 | D1 | F1 | C | F1 |
| C | G | F1 | C1 | E1 | D1 | F1 | C | F1 |
| D1 | G | F1 | D1 | E1 | F1 | F1 | D | F1 |
| D | G | F1 | D1 | E1 | F1 | F1 | D | F1 |
| E1 | E1 | E1 | E1 | E | E1 | E1 | E1 | E1 |
| E | E1 | E1 | E1 | E | E1 | E1 | E1 | E1 |
| F1 | G | F1 | F1 | E | F1 | F1 | F1 | F1 |
| F | G | F1 | F1 | E | F1 | F1 | F1 | F1 |
| G | G | G | G | G | G | G | G | G |

The basic principle behind the code design is built around the dictionary. The language’s tokens are loaded into a dictionary via text file. The dictionary’s key value pair is <Token,LexorType>. The source code is then loaded in character by character, and passed into the Finite State Machine. As each character enters the FSM, it checks against the dictionary key to see whether the character had a match. (i.e tokens[“Keyword”]) If it did, then it returns the corresponding Lexor type, otherwise, it continues down the state machine. The diagram above, demonstrates this behavior.

If the first character does not result in a match, it will move to state B1, if it was a letter, and C1 if it was an integer. From B1 and C1, it will continue to loop into itself, until the proper conditions are satisfied, and allow the move to their final states. From C1, if a ‘.’ is passed into the state, it is considered real, and will move to state D1, which will loop until a code separator is reached.

From B1, if during the loop, it matches a key in the dictionary, it will Move to state G, which will return the matched lexor type. There are several other sub cases specified in the table, that was excluded from the diagram for the sake of visual cleanliness. The benefit of using the dictionary is that all Keyword, Separators, and operator matches can be passed off as a single “match” state with no further effort necessary.

Limitations

None

Shortcomings

The lexer accepts the comment separator correctly, but takes the entire comment from separator to separator (!), and takes the entire lexeme as a comment token without spaces.

Example: ! this is a comment ! would be output as !thisisacomment! with token ‘Comment’ in the output.