CS323 Assignment 1 Documentation

by Natalie Ottolia, Angel Soto

1. Problem Statement

The problem can be defined as creating an early-stage compiler that reads in a file of text and creates tokens out of it. Then, we must identify each token as either one of the following: Keywords, Identifiers, Separators, Operators, Integers, or Reals. To do so, we must use Finite State Machines, at the very least, on the identifiers, integers, and reals. Finally, the program must output the tokens and their corresponding lexeme as single-line records.

2. How to use the program

In order to use the program, a file must be used as input as a command line argument. Since this is a python script, a sample usage of how to use the code looks like the following:

python Lexer.py SampleInputFile.txt

The first argument *python* because it is the interpreter. Secondly, the name of the script; in this case, *Lexer.py.* Finally, we provide the name of the input file. **Note:** If the file is not on the same directory as the *Lexer.py* script, then provide the full path for the file.

3. Design of the program

The program is broken up into multiple functions. At the top level, we have a simple call that reads the input argument given through the command line (the name of the file), and calls a function called *process_file().* This returns a list of tokens. Then, we iterate through each, sending it to the *lexer()* function to do the heavy lifting. Each line gets appended to a final output string, which gets written to a file called *output.txt.* In greater detail, this is how the program's functionality is broken down:

process_file(): This function reads in a file and does the following things:

- *Splits the text into a list of sentences*
- *Removes any sentence that starts with a block comment identifier*
- *Creates a single string out of the remaining text*
- *Adds space between separators and other identifiers*
- *Creates a list out of all tokens and returns it*

lexer(): This function takes in a single lexeme, and returns a string with the name of the token. It does it in the following order:

- *Using list iteration, it checks if the token is a separator*
- *If not, checks if the token is an operator*

- *If not, it checks if the token is a keyword*

If none of the above apply, it then calls up to 3 other functions that behave like FSMs by doing the following:

- *Iterates through each individual character on the token*
- *It determines which column of the transition function it belongs to*
- *It changes state given the current state and the transition function*
- *Returns true if it ends in an accepting state, false otherwise*

This is done for all 3: identifiers, reals, and integers. To briefly explain what each of the FMS do:

Identifier_fsm(): *This FSM has alpha, num, '$', and other as its possible inputs. In order to arrive to an accepting state, the string must start with an alpha character. Otherwise, it moves to a permanent rejection state. Then, it will allow any combination of '$', num, and alphas. If at any point it sees any other input, it will move to the permanent rejection state.*

real_fsm(): *This FSM has num, '.', and other as its possible inputs. In order to arrive to an accepting state, the string must have any number of num characters with a '.' and any number of nums after it. If at any point it sees a non-numeric or '.' character, it moves to a rejection state. It will also reject if it sees any number of '.''s other than exactly 1.*

integer_fsm(): *This FSM has num and other as its only inputs. It will move to an accepting state and remain there as long as it continues to see numeric values. If at any point it sees a non-numeric value, it moves to a permanent rejection state.*

The equivalent RE's for these FSMs are as follows:

Identifier: '[a-zA-Z][a-zA-Z0\d\$]*'

Real: '[\d]*\.[\d]+' (Allows the string to start with a . but not end in one)

Integer: '[\d]+'

4. Limitations

The program works as expected with the provided specification, as well as input examples. The code currently is limited to not being able to detect block comments if they don't begin on their own line or span multiple lines, although this may be outside of the scope of this assignment.

5. Shortcomings

None.