**CS323 Documentation**

**Problem Statement**

Write a lexer (lexical analyzer). A finite state machine must be implemented for identifier, integer, and real tokens (the rest can be written ad-hoc). A major component of this assignment 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 for the lexeme, i.e. the instance of a token. The main program should test the lexer, i.e. the program should read a file containing the source code of Rat15S to generate tokens and write out the results to a file.

**How to use your program**

The program has only been tested on the MacOSX operating system running the newest version of Python3.4, any incompatibility will be due to the user not using the correct version of python.

The data.txt file, which contains the input script, must be in the same directory as the Lexer.py python script. To run the program, enter the directory of the data file and python script and type: $ python3.4 Lexer.py

Once the program is opened, it will prompt the user for input:

‘Enter file you would like to open (type “quit” to exit): ’

You may enter the name of a file you would like to input such as: testcase1.txt

DO NOT use quotations around the filename.

If a file is not recognized the program will respond with:

‘Your file was not found!’

Then it will prompt you to enter another filename. To quit the program type: quit into the terminal and the program will exit.

Successful runs will output to the terminal and be written into a file with the name of the input (user defined) file and the extension “.RAT”.

Once the program is called, it will run on a loop until the user enters: quit

Example run:

Arts-MacBook-Pro:Lexer Arty$ python3.4 Lexer.py

Enter file you would like to open (type "quit" to exit): testcase1.txt

File open!

File data

Your Tokens and Lexemes have been saved as testcase1.RAT in the working directory.

**Design of your program**

The program ‘Lexer.py’ has 10 functions. Initially the ‘main()’ function is called. The ‘main()’ function runs the appropriate functions to read, analyze, and write files if necessary. Once the user is finished with her input the ‘main()’ function exits. The following will describe the program flow of the code and the responsibilities of the 10 functions.

1. main():

The ‘main()’ function initializes the tokens, lexemes, todo, and user variables. It then enters a while loop which will continue until the user enters ‘quit’. First, the main() loop asks the user to enter the filename of the input file. Then, the file is checked against the users input to see if the program should continue or break and exit. If user enters anything except quit, the ‘process\_file()’ function is called and the ‘todo’ and ‘user\_fh’ are returned and assigned. The ‘lexer()’ function is then called with ‘todo’ as its input. If tokens are returned, the ‘write\_tokens\_lexemes()’ function is called. The loop is then repeated.

1. process\_file():

This function takes a string input called ‘user\_file’. This is the user specified file that the program should use as its input. The try/except operations are used to test if the file exists and is opened. If the file is opened then each line of the file is read in and appended to a deque known as ‘todo’ which will contain every character needed to be processed. The contents of the file are also printed to the screen so that the user can see the contents of the file. If the file could not be opened (i.e. the user enters file that does not exist) the *except* operation will be called and the user will get an error message saying the file was not found.

1. lexer():

This function takes an input of ‘todo’, that is a deque data structure. A deque data structure was chosen due to its versatility as a stack and a queue. This deque contains all characters needed to process. A character is popped from ‘todo’ and appended to ‘token’. The token then travels through the lexer. First, double-character operators (e.g. :=) and separators (e.g. @@) are checked. The token is then checked for separators and operators through the check\_separator() and check\_operator() functions. Then, the token is processed by the fsm\_digits() function. If no match is found then the token is processed through the fsm\_identifier() function. Finally, if no match then the token moves back to the top of the loop and another character is appended from ‘todo’ to the token and the process repeats. If the ‘todo’ list is empty and there is a token not matched, the token is appended to unknown and the function returns tokens and lexemes to the caller.

1. fsm\_identifier():

This function takes two inputs: omega (token string) and state (machine state). A deterministic final state machine table is initialized (see attached for graph representation). For each element in omega, if the element is an alphabetical character, then its column number is equal to 0, else if the element is a digit, then its column number is equal to 1, else if the element is an ‘\_’ the column is equal to 2, else the column number is 3. The new state of the machine is updated by looking up the table[state][col] where state is the row and col is the column. This is done for every element in omega. The final state is returned to the caller.

Regular expression:

1. fsm\_digits():

This function takes two inputs: omega (token string) and state (machine state). A deterministic final state machine table is initialized (see attached for graph representation). For each element in omega, if the element is a digit then its value is equal to the column number, else if the element is a ‘.’ the column is equal to 10, else the column number is 11. The new state of the machine is updated by looking up the table[state][col] where state is the row and col is the column. This is done for every element in omega. The final state is returned to the caller.

Regular expression:

1. write\_tokens\_lexemes():

This function takes an input of a list of tokens, lexemes, and a string ‘fh’ for the file handle. The function formats and prints all tokens and corresponding lexemes to the terminal. Then, the output file is opened and the ‘outputFilename()’ function is called. Then, the same format and output as done to the terminal is written to the output file. Finally, the output file is closed and the function exits.

1. outputFilename():

This function takes a string of a filename, removes the ‘.extension’ (e.g. .txt) part of the file and appends a new extension. This function is used to convert the filename of the users input file to the output of the source file. An example of this function would be an input of ‘foo.txt’ and an output of ‘foo.RAT’ returned to the caller.

1. check\_keyword():

This function takes a string input ‘token’, which it checks against defined keywords. If the ‘token’ matches a defined keyword, a True boolean expression is returned. Otherwise, False is returned to the caller. The defined keywords are:

int, boolean, real, if, else, endif, while, return, read, write, true, false, function

1. check\_operator():

This function takes a string type as input. The string is checked against the possible single character operators. If the string matches an operator, a True Boolean expression is returned. Otherwise, False is returned. The following operators are checked in this function:

< > + \* - / =

Note: these may not be all the possible operators. Operators of length more than 1 are handled in the ‘lexer()’ function.

1. check\_seperator():

This function takes a string type as input. That string is checked against the possible single character separators. If the string matches a separator a True Boolean expression is returned. Otherwise the return is False.

The following possible separators that are checked in this function are:

( ) { } [ ] : ; ,

Note: these may not be all the possible separators. Separators of length more than 1 are handled in the ‘lexer()’ function.

**Any Limitation**

None

**Any shortcomings**

None