CPSC 323 Assignment 1: Lexical Analyzer (Documentation)

Jeffrey Liang

1. **Problem Statement**

This assignment focuses on the design and implementation of a lexical analyzer, using a Finite State Machine (FSM) to read and correctly classify the lexemes from an input file/string into general tokens. The lexical analyzer, or lexer for short, outputs each lexeme with its associated token.

1. **How to use this program**

To use this program, one simply needs to run the lexer.py file. Since this program is written in Python, one would need to use a Python shell, interpreter, Jupyter notebook, or similar. The program will prompt the user to enter the directories of an input file and an output file. Then, the program will output to the console the correct tokens with associated lexemes. It will also write the console output to the specified output file, whose pathway is specified as the second argument by the user. An example would be the following, in a Python shell:

# First open lexer.py, and run it

Please input the directory of the input file:

C:/Users/jhlia/Documents/2020-2021/cpsc323/hw1/Test files/testinput1.txt

Please input the directory of the output file:

C:/Users/jhlia/Documents/2020-2021/cpsc323/hw1/Test files/testoutput1.txt

1. **Program Design**

The program was designed so that the user would only have to call one function, lexer(). To begin, input alphabets for the FSM to recognize were created. These include letters/identifier characters, digits, operators, and separators. These were converted from strings into lists to save time on string matching in lexer(). A list of keywords was also created, because the lexer should recognize these as different tokens than identifiers. Next, the FSM state transition table was constructed:

States (Q):

1 - starting state (q­0)

2 - in identifier

3 - end of identifier

4 - in integer

5 - end of integer

6 - in decimal

7 - end of decimal

8 - in block comment

9 - end block comment

10 - operator

11 - separator

Inputs (∑):

l - letter/identifier char (including ‘\_’, ‘$’)

d - digit

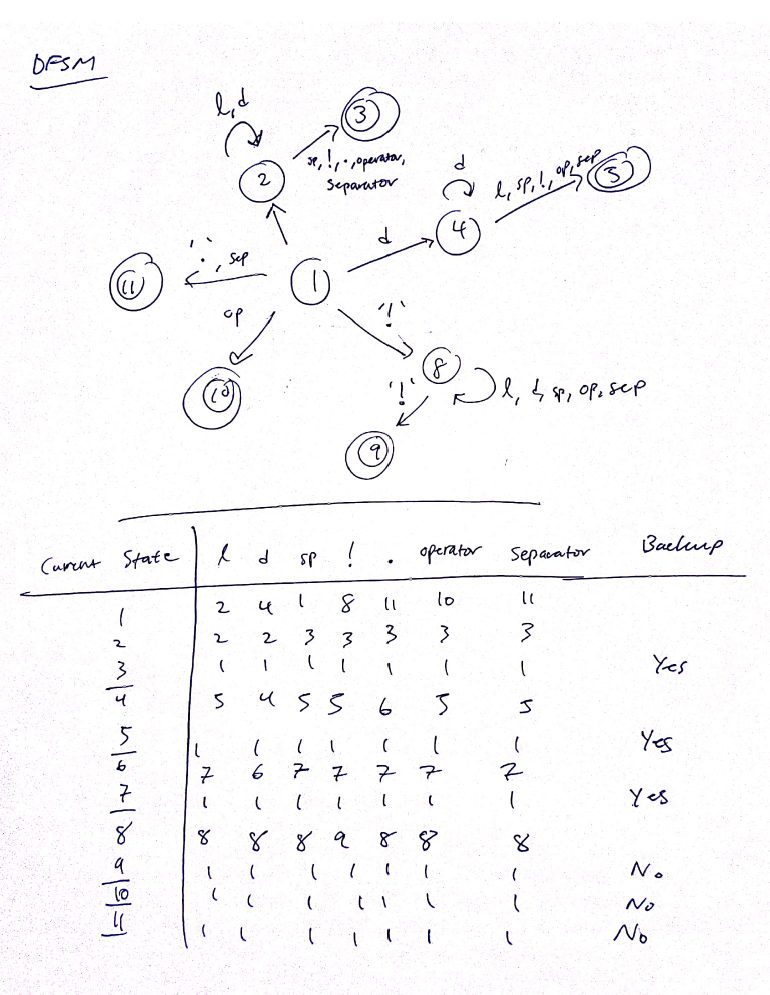
sp - white space

! - comment

. - period

op – operator

sep - separator

The table was implemented as a dictionary, or hash table, in Python, for the ease of use and lookup times. The table is based upon the NFSM obtained from Thompson’s construction from regular expressions. Let *l* denote a letter from the alphabet, or ‘\_’, or ‘$’; *d* denote a digit from 0-9, The regular expressions for integer, reals, and identifiers are the following:

**integer = d+**

**real = d+.d+**

**identifier = *l(l|d*)\***

The associated NSFM’s for these regular expressions are:

**Integer:** *d*

*d* **ε**

*d* *d*

*d* *d*

**Real:**

**ε** *d*

*d* *d*

*“.”*

*d* *d*

*d,l* *d*

**Identifier:**

**ε** *d*

*l* *d*

Finally, the list of accepting states (3, 5, 7, 9, 10, 11) was created. With these constants in place, we move on to defining helper functions.

First, a function get\_token() was written. This function returns a string denoting the appropriate token when an accepting state is passed as an argument. For example, if the accepting state is 3, denoting the end of an identifier, the function outputs “IDENTIFIER”.

Next, a function get\_next\_state() takes two arguments: the current state of the lexer, and the current character, and returns the appropriate state according to the transition table. This is the part where the FSM reads an input and moves to a different state.

Another function file\_to\_str() converts input files into strings for the lexer to read. This is automatically called in lexer() for the user, who supplies the direct file pathway. It also removes newline and tab characters, which this lexer does not recognize.

Finally, the lexer() function incorporates all of the above to carry out the task of interest. Essentially, the function does the following:

Repeat

Pick the next input character.

Find the new state-table entry.

If this is a final state for some token,

isolate the token, and maybe decrement cp (char pointer).

Until input is used up.

(This is the process described in Section 2.9 of Introduction to Compiler Construction, T. Parsons). We note that in the loop, we have to build each lexeme until an accepting state. So a variable lexeme = “” is initialized before the loop, and during the loop, is updated with the input string until an accepting state. Then, it is outputted along with a token, and then reset to an empty string “”, and the process repeats until the input is finished. Any white spaces in the lexeme are removed, unless the lexeme is a block comment.

One more point to note is that to recognize keywords, the lexer initially recognizes them as identifiers. Before outputting the token type, however, the lexer will check an identifier against the list of keywords and output a keyword token if the identifier is in fact a keyword.

1. **Limitations**

This lexer has some limitations:

* Inability to recognize compound operators
* Inability to recognize single-line comments starting with ‘!’
* The limitation on file size is the same as Python’s limit on reading text files

1. **Shortcomings**

To the author’s knowledge, the lexer meets the requirements set forth by the assignment. 3 test cases are included along with their outputs.