**Brandon Xue, Miguel Pulido, Henry Torres**

# CS323 Documentation

About 2-3 pages

## 1. Problem Statement

Create a lexicographical analyzer (LA) for the Rat20SU programming language. The LA should be able to open a text file containing Rat20SU source code, compare the characters one at a time in sequence, and create a list of valid and invalid tokens.

The LA will have a function `lexer()` that will return one single record. The record will have one field for the textual value of the lexeme, and one field for the token type. The token types are: identifier, integer, keyword, operator, separator, and invalid.

Integers will be unsigned decimal integers. Identifiers must begin with a letter, and may contain letters or underscores. The program shall utilize a finite state automaton to complete lexicographical analysis of at least these two token types. The program shall output a list of lexemes and their token types.

## 2. How to use your program

1. **Python 3 is required for execution of our program.**

- The official Tuffix distribution should come with Python 3. To check if you have Python 3, open your terminal emulator and follow the instructions for your OS.
- Tuffix/Linux: `python3 --version`
- Windows: `py --version`

2. **Make sure all necessary files are present in the same project directory:**

- Lexer.py ---- the Lexer class
- LexerConstants.py ---- set definitions and constants
- Ratify.py ---- the main script file
- test1.txt ---- test file 1, <10 lines
- test2.txt ---- test file 2, <20 lines
- test3.txt ---- test file 3, >20 lines

3. **Navigate your terminal emulator to the directory containing the project files.**

- Tuffix/Linux: `cd ...path/to/project/directory`
- Windows: `cd ...path\to\project\directory`

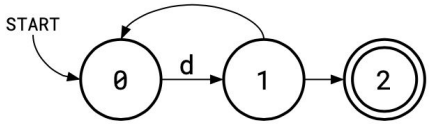
4. **Using Python3, Run the main script file Ratify.py.**

- You can specify the input and output files (in that order) as command line arguments.
- Tuffix/Linux: `python3 Ratify.py test1.txt test1-out.txt`
- Windows: `py -3 Ratify.py test1.txt test1-out.txt`
- If no command line arguments are given, the program will prompt you for file names. Please follow the command prompt. The input file must match your source code file name. The output file must also be provided. For the output file name, be careful not to use the name of an existing file, otherwise its contents will be overwritten.

## 3. Design of your program

The program uses a deterministic finite state automaton (DFSA) to process input characters. The DFSA is derived by using the subset method on a nondeterministic finite state automaton (NFSA). The NFSA is created by using Thompson's Construction on regular expressions. Since Thompson uses reverse Polish regular expressions in his original 1968 paper, we will provide our regular expressions in both forms. Using reverse polish notation makes it easier to follow the NFSA construction process.

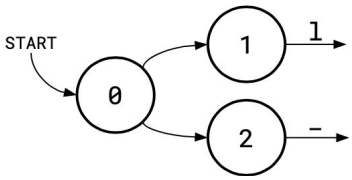
**Integers**




The regular expression for integer tokens is: `d+`, where d represents any of the 10 decimal digits. The reverse Polish expression in this case is also `d+`. (Strictly speaking, some conventions require a starting state that is not reachable from other states. We can simply add a state before state 0.)

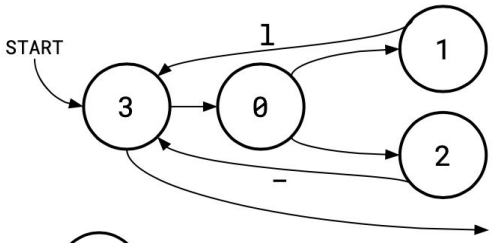
**Identifiers**

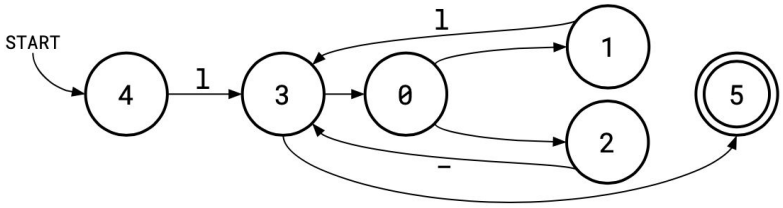
The regular expression for identifiers is `l(l|_)*`. In reverse Polish notation, this is `ll_|*•`, where • represents concatenation. Following reverse Polish, the first operator is `|`. Here is the first step:




The next operator is *, so the Kleene closure will be applied to this NFSA on the left. To apply the Kleene closure to any NFSA `m1` with starting state $q_0$, a new starting state `s` will be created. `s` will have an epsilon transition to $q_0$. The dangling arrows of `m1` will be pointed at `s`.

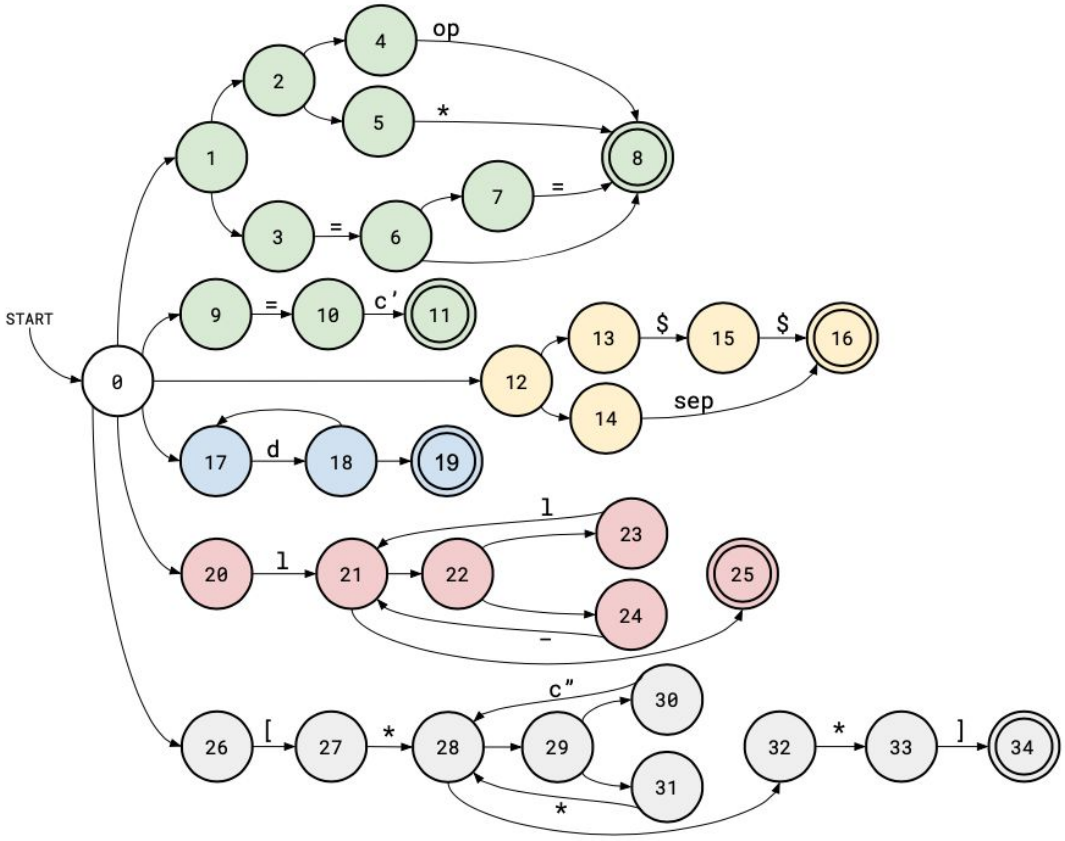
Finally, the new NSFA will have a dangling arrow coming out of `s`. In our case, state 3 is the new state `s`. Next, this NFSA will be concatenated, and the dangling arrow will be connected to an accepting state.



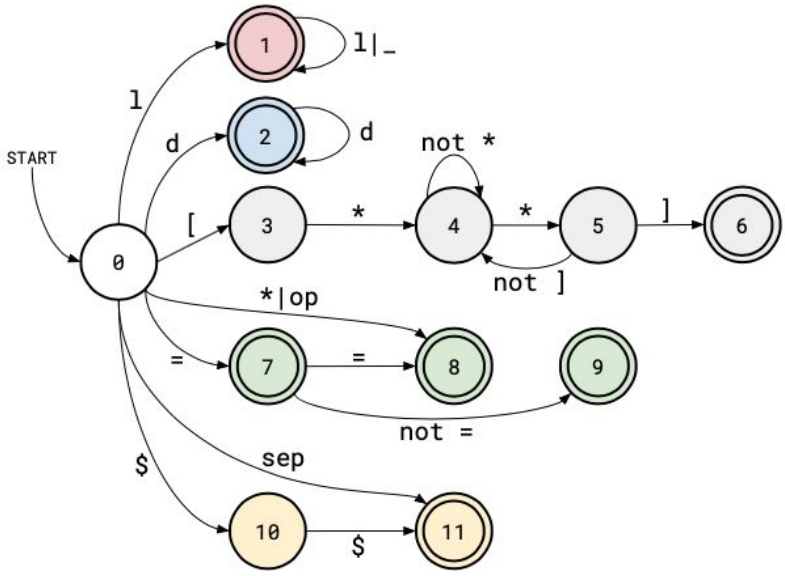




Please note, we have used Thompson's original methodology rather than the method provided in the textbook. Thompson demonstrates `a(b|c)*` here:
www.import.io/wp-content/uploads/2016/01/p419-thompson.pdf
A more applicable reference is found here:
https://swtch.com/~rsc/regexp/regexp1.html

Next, we will show the entire NSFA used for our Lexer, as well as the corresponding DFSA derived by using the subset method.




We used Thompson's Construction on each individual pattern, but we simply joined the patterns together at state 0 without Thompson's Construction.
Green: operators. Yellow: separators. Blue: digits. Red: identifiers (and keywords). Gray: comments. A graveyard state (not shown) was used to handle invalid tokens. Note that `c'` represents any character that is not an equal sign, and `c"` represents any character that is not an asterisk. Here is the derived deterministic finite state automaton used in our program:



Detailed work for our NFSA to DFSA conversion is shown here, with short written explanations and transition tables:
docs.google.com/drawings/d/1S7PYk6Tb5ep1jDAgNVHbH2cIL9_3kzXqFlOy_iohViE/edit?usp=sharing

**Column Lookup Dictionary (Hash Table)**
Since we have decided to use a transition table, input characters must be grouped into columns. To facilitate O(1) lookup, our program creates a dictionary at the beginning of the runtime. The key of the dictionary is an input character, and the value is the column of the transition table that the character corresponds to. The dictionary is created in the function: `char_col_dict_init()` of the `Lexer` class. The execution time for creating this lookup table is negligible, and produces time savings when the program is run on longer source code files.

**Token Class**
The return type of the `lexer()` function is an object of class Token. Token has two fields, `type` and `value`. `type` stores the token type as an integer, and `value` stores the textual value of the lexeme as a string.

**Sets**
The accepting states of our program are stored in a set. We also have a set for what we will call "final" states. "Final" states are states that tell the lexer to stop analyzing any characters. Because the file is treated as one giant stream, the lexer needs this information to know when to stop and create a token. Otherwise it would evaluate the entire file as one string. Certain characters encountered in certain states will also stop the finite state machine. These characters are stored in a "distinguishing" set. We use sets for their O(1) search time. Further explanation is provided as comments in the source code.

## 4. Any Limitation

**Time**: The program is written in Python, so it is slower than a traditional compiler, and execution speed is a limitation. The same concepts can be implemented in a different language for better performance.

**Convenience**: Traditional compilers do not require users to specify an output file name. Our program has this requirement. This improvement would not be difficult to implement.

## 5. Any shortcomings

**None** that we are aware of.