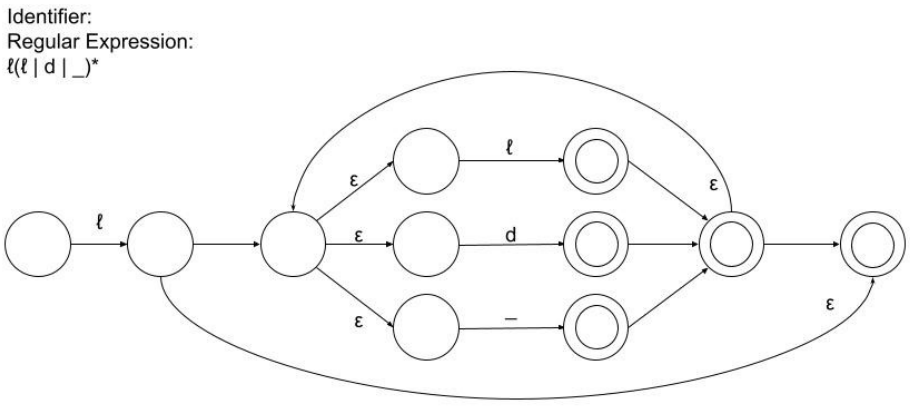
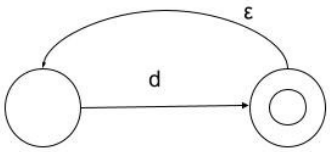
CS323 Documentation

1. **Problem Statement**
   The primary goal for this assignment is to develop a lexer (lexical analyzer, the first step of compilation) which will scan the source code of the Rat24S language program and convert it into a series of tokens; these tokens are meaningful specific to the language. Later these tokens will be used by a parser to understand the structure of the code.
2. **How to use your program**
   Download the zip folder. Unzip the folder. Navigate to the Lexical-Anayzer directory. Run the command 'python3 LexerFSM/main.py'. The lexer will run based off of the 3 input files inside the input folder and corresponding output will be placed in the output folder. If you want to test your own test cases, you can replace one of the three input files that we have and the output will again be placed in the corresponding output folder. The reason for this is that the names of the input and output files are hardcoded.
3. **Design of your program**
   Our project repository is split up into a few different sections, Documents, FSM diagrams, and the code for the lexer. The code for the lexer is broken into classes that hold each finite state machine. We have a file for the identifiers, integers and reals that hold the main finite state machine code that determines if we are in an accepting or not accepting state. Each of these files is built with the main export being a class that holds the states that we are traversing with our fsms. Each of these classes also have specific functions that help determine what state we are in. The main part of all of these classes is their validate_x functions. These functions are passed the current character from the main lexer and return a state and a Boolean value that tells us if the current token was terminated. We update the input_char_terminates_token Boolean if we get to the end of the token and make sure that we still return a proper state that we can use in our lexer to properly identify the token and separate the lexeme. Each of our files that contain our classes also have individual testing code that only gets run if you run those files individually; this code was used for testing the individual finite state machines to verify they were working. In our finite state machines, we included transition tables to determine what state to go to given certain inputs, this is the main driver of all of our functions along with validate_x which uses the

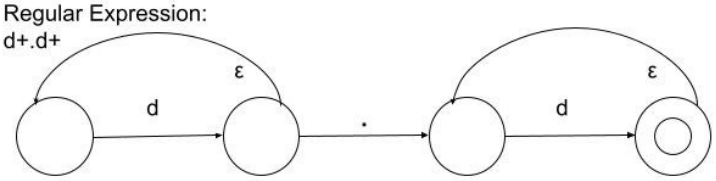
finite state machines to return the state and terminated-token status. These transition tables came from us converting our ndfsms into dfsms for easier code implementations. For separators, operators, and keywords we didn't implement finite state machines, but we did implement a buffer to determine if the input string was one of these. Each of these three internally tracked the characters they were receiving and added them to a buffer, if the elements in the buffer matched with any of the items we had in lists, then it would correctly identify that as the token and lexeme. In our main file, we import all the classes that we need, define the main lexer function, and then call it given the content of the input file as an argument. Our lexer iterates over all the characters in our input file until it reaches the end. One of the first thing that it does is check if there are any comments to skip them, so they don't get fed into our finite state machines. Next, the character is fed through all the finite state machines and the buffer functions. If any of them return true for the state and the input character terminates the token, then the token gets selected and the lexeme is created and both are appended to the output file. We also have code that traverses whitespace after a lexeme has been created so we don't have to feed that into each finite state machine. Images of our ndfsms:



Integers:
Regular Expression: d+



Real Numbers:
Regular Expression:
d+.d+



4. Any Limitation

None

5. Any shortcomings

None