Project Documentation

1. Problem Statement

For this project, we had to build a lexical analyzer for the Rat25S language. The lexer reads a Rat25S source file, breaks it into smaller pieces called tokens (like keywords, identifiers, numbers, operators, and separators), and outputs each token with its type and the actual text (called a lexeme). It also ignores comments (written inside [* and *]) and extra spaces or blank lines. The assignment required us to use finite state machines (FSMs) for identifiers, integers, and real numbers, while other tokens like keywords and operators could be handled with simpler methods. Rat25S is a simple language with clear rules, and the lexer is the first step in a compiler, turning raw code into tokens for the following steps to process.

2. How to Use Your Program

To execute the program, follow these steps:

1. Prerequisites: Ensure you have Python 3.x installed on your system. Our program is written in Python and requires no additional libraries or dependencies.

2. Input File: Make a text file containing the Rat25S source code. The file should follow the syntax rules of Rat25S, including proper use of keywords, identifiers, integers, reals, operators, and separators. Example test files (test1.txt, test2.txt, and test3.txt) are provided in the tests folder.

3. Execution: Open a terminal in the project folder and run the program with the following command:

   ```
   python src/main.py tests/test1.txt
   ```

   Replace test1.txt with the path to your Rat25S source file. Our program will process the file and generate an output file to its corresponding test case

4. Output: Our program will generate an output file named output.txt in the same directory as the script. This file will contain a list of tokens and their corresponding lexemes, formatted in a table-like structure. Each line in the output file represents a token, with the token type and lexeme separated by a tab or space.

5. Test Results: We manually created and included the output files for all three test cases (test1.txt, test2.txt, and test3.txt) in the submission. If you run the test command, it will overwrite output.txt with the results for the specific test file you run.

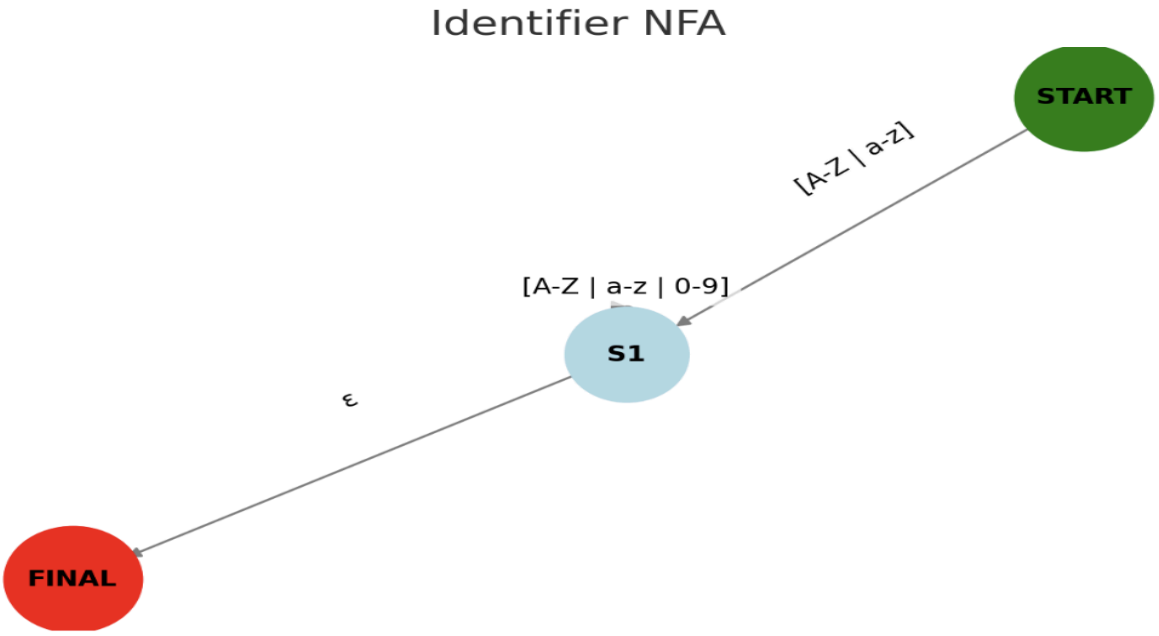
6. Additional Tests: Try running our program with test2.txt and test3.txt for more examples. These test files contain different Rat25S code snippets demonstrating the lexer's ability to handle various token types and edge cases.

3. Design of Our Program

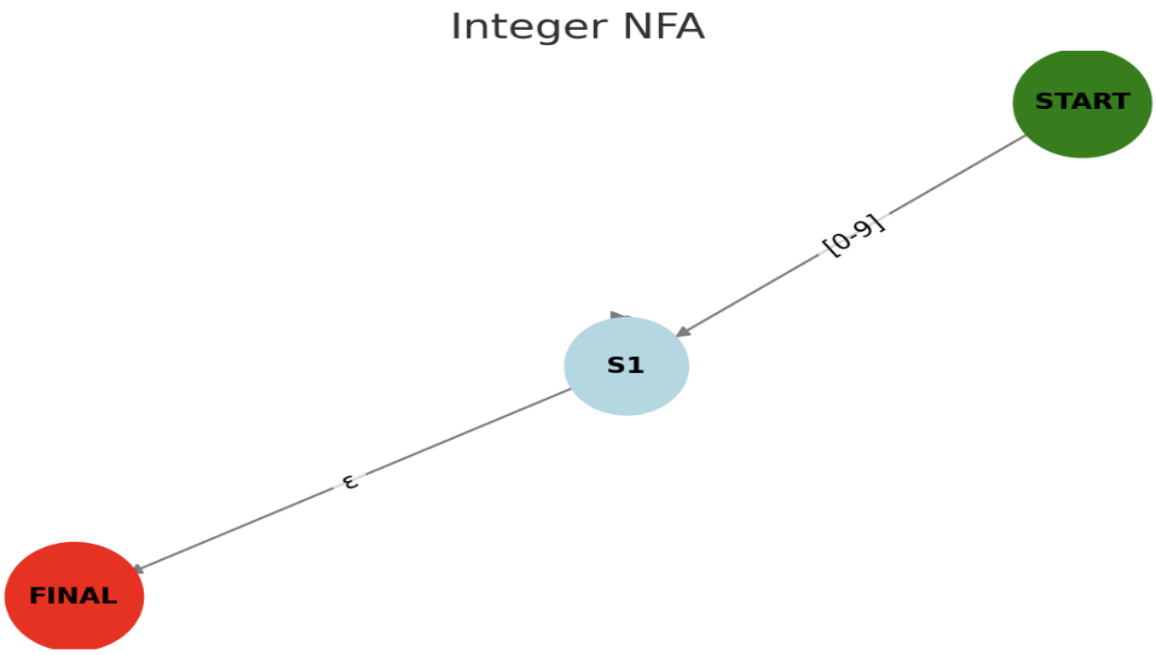
Our program is organized into different parts, each responsible for a specific aspect of the lexical analysis process. Below is a detailed description of the design:

A. Lexical Analyzer (lexer):

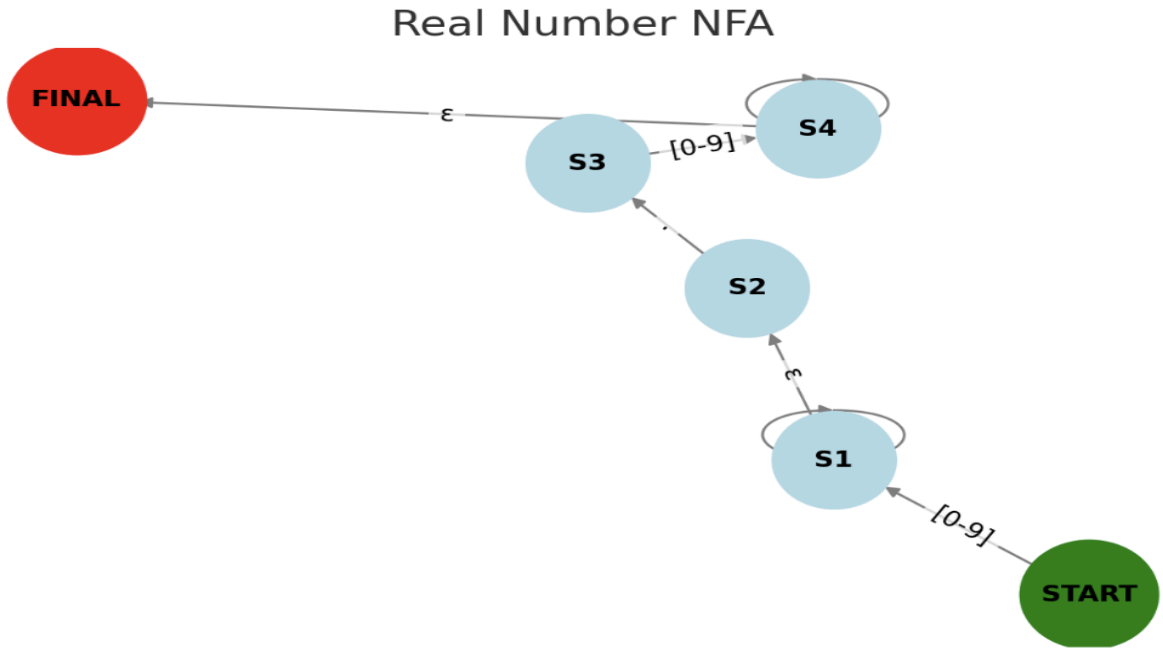
- The lexer(source, index) function is the core of our program. It processes the source code starting at a given position and returns the next token with the new position.
- For identifiers, integers, and reals, we used deterministic finite state machines (DFSMs):
    - Identifiers: Start with a letter, followed by letters, digits, or underscores.
        - Regular Expression (RE): [A-Za-z][A-Za-z0-9_]*



    - Integers: A sequence of digits.
        - RE: [0-9]+



- Reals: Digits, followed by a dot, and then more digits.
  - RE: [0-9]+\.[0-9]+



- We first designed these as non-deterministic finite state machines (NFSMs) using Thompson's construction and then converted them to deterministic processes. This approach ensures that our lexer can efficiently recognize these token types.

- We used more straightforward logic for other tokens (keywords, operators, separators). Keywords are matched against a predefined list, operators are checked against a set of valid operators, and separators are matched against a set of valid separators.

- The lexer returns a Token object containing the token type and the actual lexeme. This object is used by the main program to generate the output.

B. Main Program:

- The main program reads the entire input file into a string and initializes the lexer.
- It then loops through the source code, repeatedly calling the lexer() function until it reaches the end of the file. Each call to lexer() returns a token, which is added to a list of tokens.
- After processing the entire file, our program writes all tokens to the output.txt file. The output file is formatted as a table, with each line showing the token type and lexeme.

C. Error Handling:

- Our program includes basic error handling to deal with invalid tokens. If an invalid token is encountered (e.g., a poorly formatted real number), the program reports an error message and continues processing the rest of the file. This ensures that our lexer can handle minor errors without crashing.

4. Any Limitation

- When there's an invalid token (like a poorly formatted real number), Our program reports an error message

5. Any Shortcomings

- None