CS323 Documentation

1. **Problem Statement**
   The primary goal for this assignment is to develop a syntax analyzer. This code takes in a series of tokens that were produced by our lexer, and determines if the code is syntactically correct.
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 syntax analyzer 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**
   The bulk of this section of the project is inside of the production_function.py file. This file houses all the code that we used for our recursive descent parser. The main driver in this file is the syntax_analyzer() function. This function is responsible for calling all of the sub functions which determine if the code is syntactically correct according to our production rules. The syntax_analyzer() function and all sub function all call the lexer() function which points us to the next token we should look at. Our syntax_analyzer() function is imported into our main.py file where it is run for each of our input files and the output is written to the corresponding output files.

   Going into more detail about the design of our functions. They follow the recursive descent parser style. The basic idea is that each non-terminal has an associated parsing procedure that can recognize any sequence of tokens generated by that non-terminal. Basically, each of our non-terminals has a corresponding function that handles validating if that part of the code is syntactically correct. The order in which we call the functions is very important here because it has to match with our production functions. Although this way of determining if the code is syntactically correct is not always the most efficient, it is relatively easy to implement so that is why we chose it. It is also simple to follow the flow of the functions by tracing the function calls. For some of our production functions, they can be epsilon. When we encountered these

occurrences, one way we solved it was by checking the first of the other option. For example, for our functionDefinitions2() function, it can go to <Functions> or epsilon. We check if the current token we are looking at matches the first(<Functions>); if it matches we can call functions() or if it is not, we can pass.

Another part of our code is the helper functions. We have functions to "print" the output to a string and then we finally print this string to our output file at the end. Our lexer() function is also modified here, in this case, it increments a global pointer to get the next token that the lexer gives us. Finally, we have the error() function that prints out a good error message and changes a flag to false. This flag makes sure that we stop printing to the output file after we get to an error.

The production rules that we created are as follows:

1. <Rat24S> ::= $ <Opt Function Definitions> $ <Opt Declaration List> $ <Statement List> $

2. <Opt Function Definitions> ::= <Function Definitions> | <Empty>

3. <Function Definitions> ::= <Function> <Function Definitions'>

4. <Function Definitions'> ::= <Function Definitions> | ε

5. <Function> ::= function <Identifier> (<Opt Parameter List>) <Opt Declaration List> <Body>

6. <Opt Parameter List> ::= <Parameter List> | <Empty>

7. <Parameter List> ::= <Parameter> <Parameter List'>

8. <Parameter List'> ::= , <Parameter List> | ε

9. <Parameter> ::= <IDs> <Qualifier>

10. <Qualifier> ::= integer | boolean | real

11. <Body> ::= { <Statement List> }

12. <Opt Declaration List> ::= <Declaration List> | <Empty>

13. <Declaration List> ::= <Declaration> ; <Declaration List'>

14. <Declaration List'> ::= <Declaration List> | ε

15. <Declaration> ::= <Qualifier> <IDs>

16. <IDs> ::= <Identifier> <IDs'>

17. <IDs'> ::= , <IDs> | ε

18. <Statement List> ::= <Statement> <Statement List'>

19. <Statement List'> ::= <Statement List> | ε

20. <Statement> ::= <Compound> | <Assign> | <If> | <Return> | <Print> | <Scan> | <While>

21. <Compound> ::= { <Statement List> }

22. <Assign> ::= <Identifier> = <Expression> ;

23. <If> ::= if (<Condition>) <Statement> <If'>

24. <If'> ::= endif | else <Statement> endif

25. <Return> ::= return <Return'>

26. <Return'> ::= ; | <Expression>;

27. <Print> ::= print (<Expression>);

28. <Scan> ::= scan (<IDs>);

29. <While> ::= while (<Condition>) <Statement> endwhile

30. <Condition> ::= <Expression> <Relop> <Expression>

31. <Relop> ::= == | != | > | < | <= | =>

32. <Expression> ::= <Term> <Expression'>

33. <Expression'> ::= + <Term> <Expression'> | - <Term> <Expression'> | ε

34. <Term> ::= <Factor> <Term'>

35. <Term'> ::= * <Factor> <Term'> | / <Factor> <Term'> | ε

36. <Factor> ::= - <Primary> | <Primary>

37. <Primary> ::= <Identifier> <Primary'> | <Integer> <Primary'> | <Real> <Primary'> | true <Primary'> | false <Primary'> | (<Expression>) <Primary'>

38. <Primary'> ::= (<IDs>) <Primary'> | ε

39. <Empty> ::= ε

4. **Any Limitation**
   None

5. **Any shortcomings**
   None