Software Construction
Lab # 6

Practice Session

Steps involved in developing a Compiler

1. Defining language specifications
    a. Character set
    b. Keywords
2. Defining Tokens using Regular Expressions
    a. Identifiers
    b. Constants
3. Storing other types of tokens in arrays or text files
    a. Operators
    b. Special characters
    c. Allowable digits and alphabets
4. Converting R.E's into deterministic FSA's.
    a. Make FSA's manually and store them in the form of a 2-D array to store states and transitions.
5. Take the source code as input.
6. Read the input character by character.
7. Identify all valid tokens.
    a. This is done by reading the characters till whitespace and matching with the list of keywords. If match found, then display the keyword as a valid token.
    b. If the token is not a keyword, check it by reading the 2-D array of FSA and see if you reach the final state correctly. This ensures a valid token as identifier or constant (depending on which FSA accepts it).
    c. If the token is neither an identifier nor a constant, then it can be an operator or a special character. If yes, display the valid token.
    d. Otherwise, there is a lexical error in the input. Display the error message.
8. Store the valid tokens in a text file along with their name and type.
9. For simplicity, you will be provided with a CFG including the starting symbol for the given language.
10. Make a parse table from the given CFG manually.
11. Store the parse table in a matrix or a text file.
12. Read the tokens one by one in sequence and match in the parse table.
13. You can use a stack for this purpose or alternately you can devise a logic to parse the input within your code.
14. In any case, the sequence of tokens should match with any of the language statement or expression.
15. If it matches, then parsing is successful. Otherwise display the syntax error.

16. Once parsing has been done, the next step is to analyze the semantics of the input code.
17. Construct a symbol table and store all the identifiers in it.
18. Now check for the scope and other static semantics of the code using symbol table.
19. Display semantic errors, if any.
20. Next step is to generate intermediate code.
    a. For each production in CFG, you will have to write the corresponding intermediate code in assembly language (preferably).
    b. Store this assembly code in some file.
    c. Use the parsing information collected previously and start replacing the productions with their corresponding assembly code.
21. Once all productions have been transformed into assembly code, analyze the code to optimize it.
    a. Use any of the optimizing techniques.
22. Generating machine code from the intermediate code is a complicated task. Therefore, it is out of scope currently.

MODULES

The above mentioned steps are broken down into modules as follows:

- Lexical Analyzer Module:
  - Step 1 to 8.
- Syntax Analyzer Module:
  - Step 9 to 15.
- Semantic Analyzer Module:
  - Step 16 to 19.
- Intermediate Code Generation Module:
  - Step 20 & 21.

TASK

So far, you have developed the first two modules. Today's task is to develop the third module which largely uses the information acquired during first two modules.