CS323 Documentation

1. **Problem Statement**
   The primary goal for this assignment is to adjust our current code in our syntax analyzer to be able to generate assembly code for our modified version of Rat24S. We also need to generate a symbol table and handle errors such as type matching and improper declarations.
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 compiler will run based off 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 additions to our program are split up into 2 parts, additions to the symbol table and addition to our main functions that generate the object code.

   Our symbol table is implemented as a python dictionary. We use this to hold the identifier, memory address, and type of the symbols we encounter. The way that we implemented it, the identifier is the key, and the memory address and type are mapped in their own dictionary as the value corresponding to the keys in the symbol table. The initial memory address is set by a variable in our code and is incremented whenever we need to. The type is determined by functions we have written to properly evaluate the identifier. In order to use our symbol table to handle any errors in our code, we have written functions to handle any improper code. One thing we check for is if a particular identifier is already in the symbol table and it gets declared again; in this case we will print an error. Another thing we check for is if an identifier is used without being declared in the declaration list. In this case, we also print out a descriptive error. If an identifier is already in the table and we try to declare it for a second time, we provide an error message. Another big part of the symbol table is type matching. This is important when we try to do any operation. For example, if we try to multiply the variables x and y together, we need to make sure that x and y are the same type. If x is an integer and y is a Boolean, we would print an error because we can’t do any operations on variables of different types. The way we implement this is by checking the left and right side of any operation or assignment to see if both variables are of the same type.

The second part of the project is to generate assembly code. We altered the production function code to add function calls that generate assembly instructions and add them to our instruction table. We also implemented jump stacks to handle certain cases. Our instruction table is a list of 1000 elements, each element is a python dictionary holding the address, operation, and operand. For the implementation of the code that generates the assembly instructions, I will detail the additions that we made to the partial solutions. For the operations, we made sure to account for + and -, * and /, when generating those instructions. We also made sure to add a label generation instruction inside of our if’ function so that we can go to “else”. For print, we added generate_instruction("SOUT", "nil"). For scan we added generate_instruction("SIN", "nil") at the beginning and generate_instruction("POPM", get_address({lexeme})) for each item inside of the scan function. For compound we left that the way it is because it doesn’t need any instructions.

Our Production_Functions.py file has all of this new code. We call our syntax analyzer in main which uses all the code I mentioned above. In main, we write our output to the output files. The output contains our symbol table, instruction table, and any error messages.

4. **Any Limitation**
   None

5. **Any shortcomings**
   None