1. Construct the Regular Expression for
    a. Identifying C, Java and Python variable names.
    b. Transition Diagram for the above regular expressions.
    c. Each variable name Suffix of '#' and prefix of '$'

2. Analyze the different phases involved in developing a machine understandable code for a C program that verifies whether the sum of the cubes of its individual digits is equal to the original number? Please describe each phase's role in transforming the C source code into executable code.
3. Analyze the concept of regular expressions and how they are used in lexical analysis. Provide the regular expressions for
    a. Identifying common language constructs like identifiers, literals, and keywords.
    b. Avoiding the comments and white spaces in the program.
4. Design a simple input buffering mechanism for a c programming language. Describe the data structures and algorithms used, and provide best mechanism for decision statements.
    i. Implement a simple input buffering mechanism for finding the greatest number among 3 in c programming language.
    ii. Implement a C code for creating a symbol table capable of storing various elements like keywords, literals, valid identifiers, invalid identifiers, integer numbers, and real numbers from any C p. rogram?

5. Elaborate the Error Handling in Compiler Phases:
    a. Discuss the various error handling techniques employed in different phases of a compiler.
    b. Analyze how error recovery mechanisms can be implemented in the lexical analysis, syntax analysis, semantic analysis, and code generation phases.
    c. Compare the advantages and disadvantages of different error handling strategies in each phase.
6. A. Significance of regular expressions in LEX.

B. Provide examples of regular expressions commonly used for recognizing tokens in C programming languages.

C. Develop the LEX code to recognization of all tokens of C language file.

7. A. Consider the following Context-Free Grammar (CFG) for a simple programming language.
<program> → <statement_list>
<statement_list> → <statement> | <statement_list> <statement>
<statement> → <assignment> | <if_statement> | <loop_statement>
<assignment> → <identifier> "=" <expression> ";"
<if_statement> → "if" "(" <condition> ")" <statement> "else" <statement>
<loop_statement> → "while" "(" <condition> ")" <statement>
<expression> → <term> | <expression> "+" <term> | <expression> "-" <term>
<term> → <factor> | <term> "*" <factor> | <term> "/" <factor>
<factor> → <identifier> | <number> | "(" <expression> ")"
<condition> → <expression> "<" <expression> | <expression> ">" <expression> | <expression>

"==" <expression>.

a. Compute the First(A) and Follow(A) sets.
b. Construct the parse tree for the "a = b + c"
c. Construct the annotated Parse tree for evaluation of x = 4 * 4 + 5 * 5 – 50.

B. Consider the context-free grammar:

S → SS+ | SS* | a and the string aa + a*.

a. Compute the leftmost derivation for the string.
b. Compute the rightmost derivation for the string.
c. Design a parse tree for the string.
d. Is the grammar ambiguous or unambiguous? Justify your answer.
e. Describe the language generated by this grammar.

8. Consider the following expression grammar: Terminals = {num, +, *, $}
   Non-Terminals = {E' , E} Rules = E' → E$; E → num; E → E + E; E → E * E
   Start Symbol = E'
   a. Develop Augmented Grammar
   b. Create the LR(0) states and transitions for this grammar
   c. Since this grammar is ambiguous, it is not LR(k), for any k. We can, however, create a parse table for the grammar, using precedence rules to determine when to shift, and when to reduce. Assuming that * binds more strongly than +, and both are left-associative, create a parse table using the LR(0) states and transitions.
   d. Test your parse table with the following strings:
      num * num + num num + num * num num + num + num
   e. Now create an LR parse table for this grammar, this time assuming that + binds more strongly than *, and both are right-associative. Test your parse table with the following strings:

9. Implement a CLR(1) parser for a simple arithmetic expression grammar in the programming language of your choice. Test the parser with different input expressions and display the parsing steps.
   B. Discuss the role of the LR(0) and SLR(1) parsers in the construction of CLR(1) parsers.

10. Consider the below grammar S→E;E→E+T|T; T→T*F|F; F→(E)|num|id
    a. Design a SDD for the given G.
    b. Design graphical representation for the Evaluation order for the expression (5+1)*((2+3)*4.5)n by above SDD.
    c. Design a suitable SDT for the expression