1) File Handling

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>

int main() {
    std::ifstream file("example.txt");  // Open file in read mode
    if (!file) {
        std::cerr << "Error opening file.\n";
        return 1;
    }

    std::string line;
    int line_count = 0, word_count = 0, char_count = 0;
    long file_size;

    // Get the size of the file
    file.seekg(0, std::ios::end);
    file_size = file.tellg();
    file.seekg(0, std::ios::beg);

    std::cout << "File Contents:\n";

    // Process file line by line
    while (std::getline(file, line)) {
        std::cout << line << "\n";  // Display line contents

        line_count++;  // Count lines
        char_count += line.length();  // Count characters in the line
```

```
        // Count words by splitting the line into tokens using stringstream
        std::istringstream iss(line);
        std::string word;
        while (iss >> word) {
            word_count++;  // Count words
        }
    }

    // Print the statistics
    std::cout << "\nStatistics:\n";
    std::cout << "Number of lines: " << line_count << "\n";
    std::cout << "Number of words: " << word_count << "\n";
    std::cout << "Number of characters: " << char_count << "\n";
    std::cout << "File size: " << file_size << " bytes\n";

    file.close();  // Close the file

    return 0;
}
```

2) Lexical Analysis

```
#include <iostream>
#include <cctype>
#include <string>
#include <vector>
#include <regex>

using namespace std;

enum class TokenType {
    KEYWORD,
    IDENTIFIER,
```

```
    NUMBER,
    OPERATOR,
    PUNCTUATION,
    UNKNOWN
};

struct Token {
    string value;
    TokenType type;
};

// List of keywords for simplicity
const vector<string> keywords = {"int", "if", "else", "while", "for", "return"};

// Function to check if a string is a keyword
bool isKeyword(const string& word) {
    for (const string& keyword : keywords) {
        if (word == keyword) {
            return true;
        }
    }
    return false;
}

// Function to check if a character is an operator
bool isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/' ||
            ch == '=' || ch == '>' || ch == '<' || ch == '!' ||
            ch == '&' || ch == '|');
}
```

```
// Function to check if a character is a punctuation
bool isPunctuation(char ch) {
    return (ch == ';' || ch == ',' || ch == '(' || ch == ')'
            || ch == '{' || ch == '}' || ch == '[' || ch == ']');
}

// Function to tokenize the input
vector<Token> lexicalAnalysis(const string& input) {
    vector<Token> tokens;
    string currentToken;
    TokenType currentType = TokenType::UNKNOWN;

    for (size_t i = 0; i < input.size(); i++) {
        char currentChar = input[i];

        if (isspace(currentChar)) {
            // Skip whitespace characters
            if (!currentToken.empty()) {
                // Add the previous token
                if (currentType == TokenType::NUMBER) {
                    tokens.push_back({currentToken, TokenType::NUMBER});
                } else if (currentType == TokenType::IDENTIFIER) {
                    if (isKeyword(currentToken)) {
                        tokens.push_back({currentToken, TokenType::KEYWORD});
                    } else {
                        tokens.push_back({currentToken, TokenType::IDENTIFIER});
                    }
                }
                currentToken.clear();
            }
            continue;
```

```
}

if (isdigit(currentChar)) {
  // If the character is a digit, it's part of a number
  currentToken += currentChar;
  currentType = TokenType::NUMBER;
} else if (isalpha(currentChar) || currentChar == '_') {
  // If the character is alphabetic or '_', it's part of an identifier
  currentToken += currentChar;
  currentType = TokenType::IDENTIFIER;
} else if (isOperator(currentChar)) {
  // If it's an operator, create a token for it
  if (!currentToken.empty()) {
    // Process the previous token
    if (currentType == TokenType::NUMBER) {
      tokens.push_back({currentToken, TokenType::NUMBER});
    } else if (currentType == TokenType::IDENTIFIER) {
      if (isKeyword(currentToken)) {
        tokens.push_back({currentToken, TokenType::KEYWORD});
      } else {
        tokens.push_back({currentToken, TokenType::IDENTIFIER});
      }
    }
    currentToken.clear();
  }
  tokens.push_back({string(1, currentChar), TokenType::OPERATOR});
} else if (isPunctuation(currentChar)) {
  // If it's a punctuation mark, create a token for it
  if (!currentToken.empty()) {
    // Process the previous token
    if (currentType == TokenType::NUMBER) {
```

```
                tokens.push_back({currentToken, TokenType::NUMBER});
            } else if (currentType == TokenType::IDENTIFIER) {
                if (isKeyword(currentToken)) {
                    tokens.push_back({currentToken, TokenType::KEYWORD});
                } else {
                    tokens.push_back({currentToken, TokenType::IDENTIFIER});
                }
            }
            currentToken.clear();
        }
        tokens.push_back({string(1, currentChar), TokenType::PUNCTUATION});
    } else {
        // If it's an unknown character
        if (!currentToken.empty()) {
            if (currentType == TokenType::NUMBER) {
                tokens.push_back({currentToken, TokenType::NUMBER});
            } else if (currentType == TokenType::IDENTIFIER) {
                if (isKeyword(currentToken)) {
                    tokens.push_back({currentToken, TokenType::KEYWORD});
                } else {
                    tokens.push_back({currentToken, TokenType::IDENTIFIER});
                }
            }
            currentToken.clear();
        }
        tokens.push_back({string(1, currentChar), TokenType::UNKNOWN});
    }
}

// Don't forget the last token
if (!currentToken.empty()) {
```

```
        if (currentType == TokenType::NUMBER) {
            tokens.push_back({currentToken, TokenType::NUMBER});
        } else if (currentType == TokenType::IDENTIFIER) {
            if (isKeyword(currentToken)) {
                tokens.push_back({currentToken, TokenType::KEYWORD});
            } else {
                tokens.push_back({currentToken, TokenType::IDENTIFIER});
            }
        }
    }

    return tokens;
}

// Function to print tokens
void printTokens(const vector<Token>& tokens) {
    for (const Token& token : tokens) {
        cout << "Token: " << token.value << ", Type: ";
        switch (token.type) {
            case TokenType::KEYWORD: cout << "KEYWORD"; break;
            case TokenType::IDENTIFIER: cout << "IDENTIFIER"; break;
            case TokenType::NUMBER: cout << "NUMBER"; break;
            case TokenType::OPERATOR: cout << "OPERATOR"; break;
            case TokenType::PUNCTUATION: cout << "PUNCTUATION"; break;
            default: cout << "UNKNOWN"; break;
        }
        cout << endl;
    }
}

int main() {
```

```
    string input;
    cout << "Enter the code to tokenize: ";
    getline(cin, input);

    // Perform lexical analysis
    vector<Token> tokens = lexicalAnalysis(input);

    // Print the tokens
    printTokens(tokens);

    return 0;
}
```

3) Symbol Table

```
#include <iostream>
#include <string>
#include <unordered_map>

using namespace std;

struct Symbol {
    string name;
    string type;
    string scope;
    int lineNumber;

    // Default constructor
    Symbol() : name(""), type(""), scope(""), lineNumber(0) {}

    // Constructor with arguments
    Symbol(string n, string t, string s, int ln)
```

```
        : name(n), type(t), scope(s), lineNumber(ln) {}
};

class SymbolTable {
private:
    unordered_map<string, Symbol> table;

public:
    // Insert a symbol into the symbol table
    void insert(const string& name, const string& type, const string& scope, int lineNumber) {
        if (table.find(name) == table.end()) {
            table[name] = Symbol(name, type, scope, lineNumber);  // Correctly pass arguments to
constructor
            cout << "Inserted symbol: " << name << endl;
        } else {
            cout << "Error: Symbol '" << name << "' already declared." << endl;
        }
    }

    // Lookup a symbol by its name
    Symbol* lookup(const string& name) {
        if (table.find(name) != table.end()) {
            return &table[name];  // Symbol found, return a pointer to it
        }
        return nullptr;  // Symbol not found
    }

    // Print all symbols in the table
    void printSymbols() {
        cout << "\nSymbol Table:\n";
        cout << "---------------------------------\n";
```

```
        cout << "Name\tType\tScope\tLine Number\n";
        cout << "---------------------------------\n";

        for (const auto& entry : table) {
            const Symbol& sym = entry.second;
            cout << sym.name << "\t" << sym.type << "\t" << sym.scope << "\t" << sym.lineNumber <<
endl;
        }
    }
};

int main() {
    SymbolTable symTable;

    // Insert symbols with arguments
    symTable.insert("x", "int", "global", 1);
    symTable.insert("y", "float", "local", 2);
    symTable.insert("z", "int", "local", 3);

    // Try to insert a symbol with the same name (should give an error)
    symTable.insert("x", "int", "local", 4);

    // Lookup a symbol
    Symbol* s = symTable.lookup("y");
    if (s) {
        cout << "\nSymbol found: " << s->name << ", Type: " << s->type
             << ", Scope: " << s->scope << ", Line: " << s->lineNumber << endl;
    } else {
        cout << "\nSymbol not found." << endl;
    }
```

```
    // Print the symbol table
    symTable.printSymbols();

    return 0;
}
```

4) Regex to DFA

```
#include <iostream>
#include <regex>
using namespace std;
int main() {
string regexPattern, inputString;
// Prompt for the regular expression
cout << "Enter a regular expression: ";
cin >> regexPattern;
// Prompt for the string to match
cout << "Enter the input string: ";
cin >> inputString;
try {
// Compile the regular expression
regex pattern(regexPattern);
// Check if the input string matches the pattern
if (regex_match(inputString, pattern)) {
cout << "The input string matches the regular
expression." << endl;
} else {
cout << "The input string does NOT match the
regular expression." << endl;
}
} catch (const regex_error& e) {
// Handle errors in the regex pattern
```

```
cout << "Invalid regular expression: " << e.what()
<< endl;
}
return 0;
}
```

5) program which accept strings that starts and ends with 0 or 1

```
#include <stdio.h>
#include <string.h>

int main() {
   char str[100];

   // Input the string
   printf("Enter the sequence: ");
   scanf("%s", str);

   // Check the first and last characters
   if ((str[0] == '0' || str[0] == '1') && (str[strlen(str) - 1] == '0' || str[strlen(str) - 1] == '1')) {
      printf("Sequence Accepted\n");
   } else {
      printf("Sequence Rejected\n");
   }

   return 0;
}
```

6) Infix to Postfix

```
#include <stdio.h>
#include <ctype.h>
```

```
#include <string.h>
#include <stdlib.h>

// Function to check the precedence of operators
int precedence(char op) {
   if (op == '+' || op == '-') {
      return 1;
   }
   if (op == '*' || op == '/' || op == '%') {
      return 2;
   }
   return 0; // Invalid operator
}

// Function to perform infix to postfix conversion
void infixToPostfix(char* infix, char* postfix) {
   char stack[100]; // Stack for operators
   int top = -1; // Stack pointer
   int j = 0; // Index for postfix

   for (int i = 0; i < strlen(infix); i++) {
      char ch = infix[i];

      // If the character is an operand (letter or digit), add it to the postfix expression
      if (isalnum(ch)) {
         postfix[j++] = ch;
      }
      // If the character is '(', push it onto the stack
      else if (ch == '(') {
         stack[++top] = ch;
      }
```

```
        // If the character is ')', pop from stack to postfix until '(' is encountered
        else if (ch == ')') {
            while (top != -1 && stack[top] != '(') {
                postfix[j++] = stack[top--];
            }
            top--; // Pop the '(' from the stack
        }
        // If the character is an operator, pop operators with higher or equal precedence to postfix
        else if (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '%') {
            while (top != -1 && precedence(stack[top]) >= precedence(ch)) {
                postfix[j++] = stack[top--];
            }
            stack[++top] = ch; // Push the current operator onto the stack
        }
    }

    // Pop all remaining operators from the stack
    while (top != -1) {
        postfix[j++] = stack[top--];
    }

    postfix[j] = '\0'; // Null-terminate the postfix expression
}

int main() {
    char infix[100], postfix[100];

    // Input the infix expression
    printf("Enter an infix expression: ");
    scanf("%s", infix);
```

```
    // Convert the infix expression to postfix
    infixToPostfix(infix, postfix);

    // Output the postfix expression
    printf("Postfix expression: %s\n", postfix);

    return 0;
}
```

7) program to implement a arithmetic operations and recognize a valid statement.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

// Function to check if the expression is valid
int isValidExpression(char *expression) {
    int i = 0;
    int prevChar = 0;  // To track the previous character for invalid sequences

    while (expression[i] != '\0') {
        if (!isdigit(expression[i]) && expression[i] != '+' && expression[i] != '-' &&
            expression[i] != '*' && expression[i] != '/' && expression[i] != '(' && expression[i] != ')') {
            // If character is not a valid operand or operator, return false
            return 0;
        }

        // Check for invalid sequences like '++', '--', etc.
        if ((expression[i] == '+' || expression[i] == '-' || expression[i] == '*' || expression[i] == '/') &&
            (prevChar == '+' || prevChar == '-' || prevChar == '*' || prevChar == '/' || prevChar == '(')) {
            return 0;  // Invalid operator usage
```

```
        }

        // Check for closing parentheses without opening parentheses
        if (expression[i] == ')' && prevChar == '(') {
            return 0;
        }

        // Track the previous character
        prevChar = expression[i];
        i++;
    }

    // The last character should not be an operator
    if (prevChar == '+' || prevChar == '-' || prevChar == '*' || prevChar == '/') {
        return 0;
    }

    return 1; // Expression is valid
}

// Function to evaluate the arithmetic expression
int evaluateExpression(char *expression) {
    int result = 0, num = 0, sign = 1;
    char op = '+';
    int i = 0;

    while (expression[i] != '\0') {
        char ch = expression[i];

        if (isdigit(ch)) {
            num = num * 10 + (ch - '0');  // Build the current number
```

```c
        }

        // If the current character is an operator or the end of the expression
        if ((!isdigit(ch) && ch != ' ') || expression[i + 1] == '\0') {
            if (op == '+') {
                result += num * sign;
            } else if (op == '-') {
                result -= num * sign;
            } else if (op == '*') {
                result *= num;
            } else if (op == '/') {
                if (num == 0) {
                    printf("Error: Division by zero!\n");
                    exit(1);  // Exit the program if division by zero
                }
                result /= num;
            }

            // Update the operator for the next operation
            op = ch;
            num = 0;
        }

        i++;
    }

    return result;
}

int main() {
    char expression[100];
```

```
    printf("Enter an arithmetic expression: ");
    fgets(expression, sizeof(expression), stdin); // To allow spaces and input length

    // Remove the newline character at the end of the input (if any)
    expression[strcspn(expression, "\n")] = '\0';

    // Check if the expression is valid
    if (!isValidExpression(expression)) {
        printf("Entered arithmetic expression is Invalid\n");
        return 1;  // Exit if the expression is invalid
    }

    // Evaluate the expression and print the result
    int result = evaluateExpression(expression);
    printf("Result = %d\n", result);
    printf("Entered arithmetic expression is Valid\n");

    return 0;
}
```

8) program which accept strings that starts and ends with 0 or 1

```
#include <stdio.h>
#include <string.h>

int main() {
    char str[100];

    // Input the string
    printf("Enter the sequence: ");
    scanf("%s", str);
```

```
    // Check the first and last characters
    if ((str[0] == '0' || str[0] == '1') && (str[strlen(str) - 1] == '0' || str[strlen(str) - 1] == '1')) {
        printf("Sequence Accepted\n");
    } else {
        printf("Sequence Rejected\n");
    }

    return 0;
}
```

9) predictive parsing, first follow

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_PROD 10  // Max number of productions
#define MAX_TERMS 10  // Max number of terminals
#define MAX_NON_TERMS 10  // Max number of non-terminals

// Data structures to store grammar
char nonTerminals[MAX_NON_TERMS];
char terminals[MAX_TERMS];
char productions[MAX_PROD][MAX_TERMS];  // List of production rules
int numNonTerminals = 0, numTerminals = 0, numProductions = 0;

// Set of FIRST and FOLLOW for each non-terminal
char FIRST[MAX_NON_TERMS][MAX_TERMS];
char FOLLOW[MAX_NON_TERMS][MAX_TERMS];

// Function to check if a character is a terminal
```

```c
int isTerminal(char c) {
    return (c >= 'a' && c <= 'z');
}

// Function to find the index of a non-terminal
int getNonTerminalIndex(char c) {
    for (int i = 0; i < numNonTerminals; i++) {
        if (nonTerminals[i] == c)
            return i;
    }
    return -1;
}

// Function to find the index of a terminal
int getTerminalIndex(char c) {
    for (int i = 0; i < numTerminals; i++) {
        if (terminals[i] == c)
            return i;
    }
    return -1;
}

// Function to add an element to the FIRST set
void addToFirst(int nonTermIndex, char symbol) {
    if (!isTerminal(symbol) && strchr(FIRST[nonTermIndex], symbol) == NULL) {
        FIRST[nonTermIndex][strlen(FIRST[nonTermIndex])] = symbol;
    }
}

// Function to calculate the FIRST set of a non-terminal
void calculateFirst(int nonTermIndex) {
```

```
    for (int i = 0; i < numProductions; i++) {
        if (productions[i][0] == nonTerminals[nonTermIndex]) {
            // Go through each production for this non-terminal
            for (int j = 2; productions[i][j] != '\0'; j++) {
                char currentSymbol = productions[i][j];
                if (isTerminal(currentSymbol)) {
                    addToFirst(nonTermIndex, currentSymbol);
                    break;
                } else {
                    int nextNonTermIndex = getNonTerminalIndex(currentSymbol);
                    if (nextNonTermIndex != -1) {
                        calculateFirst(nextNonTermIndex);
                        // Add FIRST of the next non-terminal
                        for (int k = 0; k < strlen(FIRST[nextNonTermIndex]); k++) {
                            addToFirst(nonTermIndex, FIRST[nextNonTermIndex][k]);
                        }
                    }
                }
            }
        }
    }
}

// Function to add an element to the FOLLOW set
void addToFollow(int nonTermIndex, char symbol) {
    if (strchr(FOLLOW[nonTermIndex], symbol) == NULL) {
        FOLLOW[nonTermIndex][strlen(FOLLOW[nonTermIndex])] = symbol;
    }
}

// Function to calculate the FOLLOW set of a non-terminal
```

```
void calculateFollow(int nonTermIndex) {
    for (int i = 0; i < numProductions; i++) {
        for (int j = 2; productions[i][j] != '\0'; j++) {
            if (productions[i][j] == nonTerminals[nonTermIndex]) {
                // Check for FOLLOW of this non-terminal based on the production
                if (productions[i][j + 1] != '\0') {
                    // Add FIRST of next symbol in production to FOLLOW
                    char nextSymbol = productions[i][j + 1];
                    if (isTerminal(nextSymbol)) {
                        addToFollow(nonTermIndex, nextSymbol);
                    } else {
                        int nextNonTermIndex = getNonTerminalIndex(nextSymbol);
                        for (int k = 0; k < strlen(FIRST[nextNonTermIndex]); k++) {
                            addToFollow(nonTermIndex, FIRST[nextNonTermIndex][k]);
                        }
                    }
                }
                // If the next symbol is epsilon, propagate FOLLOW of A
                if (productions[i][j + 1] == '\0' || strchr(FIRST[getNonTerminalIndex(productions[i][j + 1])],
'ε') != NULL) {
                    addToFollow(nonTermIndex, FOLLOW[getNonTerminalIndex(productions[i][0])][0]);
                }
            }
        }
    }
}

// Function to initialize the FIRST and FOLLOW sets
void initializeSets() {
    for (int i = 0; i < numNonTerminals; i++) {
        FIRST[i][0] = '\0';
```

```
        FOLLOW[i][0] = '\0';
    }
}

// Function to print the FIRST and FOLLOW sets
void printSets() {
    printf("\nFIRST sets:\n");
    for (int i = 0; i < numNonTerminals; i++) {
        printf("FIRST(%c) = {", nonTerminals[i]);
        for (int j = 0; j < strlen(FIRST[i]); j++) {
            printf("%c ", FIRST[i][j]);
        }
        printf("}\n");
    }

    printf("\nFOLLOW sets:\n");
    for (int i = 0; i < numNonTerminals; i++) {
        printf("FOLLOW(%c) = {", nonTerminals[i]);
        for (int j = 0; j < strlen(FOLLOW[i]); j++) {
            printf("%c ", FOLLOW[i][j]);
        }
        printf("}\n");
    }
}

int main() {
    // Input Grammar

    // Non-Terminals (you can extend it based on grammar)
    nonTerminals[0] = 'E';
    nonTerminals[1] = 'T';
```

```
nonTerminals[2] = 'F';
numNonTerminals = 3;

// Terminals (you can extend it based on grammar)
terminals[0] = 'a';
terminals[1] = 'b';
terminals[2] = '+';
terminals[3] = '*';
terminals[4] = '(';
terminals[5] = ')';
numTerminals = 6;

// Grammar Productions (for example: E → T E', E' → + T | ε, T → F T', T' → * F | ε, F → (E) | a | b)
strcpy(productions[0], "E=TE'");
strcpy(productions[1], "E'=+TE'|ε");
strcpy(productions[2], "T=FT'");
strcpy(productions[3], "T'=*FT'|ε");
strcpy(productions[4], "F=(E)|a|b");
numProductions = 5;

// Initialize FIRST and FOLLOW sets
initializeSets();

// Calculate FIRST sets for each non-terminal
for (int i = 0; i < numNonTerminals; i++) {
   calculateFirst(i);
}

// Calculate FOLLOW sets for each non-terminal
for (int i = 0; i < numNonTerminals; i++) {
   calculateFollow(i);
```

```
    }

    // Print the sets
    printSets();

    return 0;
}
```

10) Syntax Analyzer

```
#include <iostream>
#include <cctype>

using namespace std;

bool isValidSyntax(const string &expr) {
    int n = expr.length();

    if (!isalpha(expr[0])) {
        return false;
    }

    bool expectingOperand = false;

    for (int i = 1; i < n; i++) {
        char currentChar = expr[i];

        if (expectingOperand) {
            if (!isalpha(currentChar)) {
                return false;
            }
            expectingOperand = false;
        } else {
```

```
            if (!(currentChar == '+' || currentChar == '-' || currentChar == '*' || currentChar == '/')) {
                return false;
            }
            expectingOperand = true;
        }
    }

    return !expectingOperand;
}

int main() {
    string expr;

    while (true) {
        cout << "Enter Syntax: ";
        cin >> expr;

        if (isValidSyntax(expr)) {
            cout << expr << " is a valid syntax" << endl;
        } else {
            cout << expr << " is an invalid syntax" << endl;
        }
    }

    return 0;
}
```

11) Implement three address

```
#include <iostream>
#include <string>
using namespace std;
```

```
void generateThreeAddressCode(string expression) {
    int tempCount = 1;
    cout << "Given Expression: " << expression << endl;

    string t1 = "t" + to_string(tempCount++) + "=c*d";
    string t2 = "t" + to_string(tempCount++) + "=x+t1";
    string t3 = "t" + to_string(tempCount++) + "=t2-2";
    string t4 = "t" + to_string(tempCount++) + "=t3";

    cout << "Three Address Code:" << endl;
    cout << t1 << endl;
    cout << t2 << endl;
    cout << t3 << endl;
    cout << t4 << endl;
}
int main() {
    string expression = "d=x+c*d-2";
    generateThreeAddressCode(expression);
    return 0;
}
```

12) Syntax tree

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;

struct Node {
    char value;
    Node *left, *right;
    Node(char val) : value(val), left(nullptr), right(nullptr) {}
```

```
};

bool isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
}

Node* constructSyntaxTree(string postfix) {
    stack<Node*> st;
    Node *t, *t1, *t2;

    for (int i = 0; i < postfix.length(); i++) {
        if (!isOperator(postfix[i])) {
            t = new Node(postfix[i]);
            st.push(t);
        } else {
            t = new Node(postfix[i]);
            t1 = st.top();
            st.pop();
            t2 = st.top();
            st.pop();
            t->right = t1;
            t->left = t2;
            st.push(t);
        }
    }

    t = st.top();
    st.pop();

    return t;
}
```

```
void displaySyntaxTree(Node* root) {
    cout << "Syntax Tree:\n";
    cout << "Root Node: -\n";
    cout << "Left SubTree Internal Node: *\n";
    cout << "Left Subtree Leave Nodes: a , b , c\n";
    cout << "Right SubTree Internal Node: /\n";
    cout << "Right Subtree Leave Nodes: d, 2\n";
}

void postorder(Node* t) {
    if (t) {
        postorder(t->left);
        postorder(t->right);
        cout << t->value;
    }
}

int main() {
    string infix = "a*(b+c)-d/2";
    string postfix = "abc+*d2/-";

    Node* root = constructSyntaxTree(postfix);



    displaySyntaxTree(root);

    cout << "POSTFIX NOTATION is\n";
    postorder(root);
    cout << "\n";
```

```
return 0;
}
```

13) Quadruple triple indirect triples

```
#include<bits/stdc++.h>
using namespace std;
struct Quadruple {
string op, arg1, arg2, result;
};
struct Triple {
string op, arg1, arg2;
};
struct IndirectTriple {
int index;
Triple triple;
};
void printQuadruples(const vector<Quadruple>&
quadruples) {
cout << "\nQuadruples:" << endl;
cout << "Op\tArg1\tArg2\tResult" << endl;
for (const auto& q : quadruples) {
cout << q.op << "\t" << q.arg1 << "\t" << q.arg2 <<
"\t" << q.result << endl;
}
}
void printTriples(const vector<Triple>& triples) {
cout << "\nTriples:" << endl;
cout << "Index\tOp\tArg1\tArg2" << endl;
for (size_t i = 0; i < triples.size(); ++i) {
cout << i << "\t" << triples[i].op << "\t" <<
triples[i].arg1 << "\t" << triples[i].arg2 << endl;
}
```

```
}
void printIndirectTriples(const vector<IndirectTriple>&
indirectTriples) {
cout << "\nIndirect Triples:" << endl;
cout << "Index\tOp\tArg1\tArg2" << endl;
for (const auto& it : indirectTriples) {
cout << it.index+10 << "\t" << it.triple.op << "\t" <<
it.triple.arg1 << "\t" << it.triple.arg2 << endl;
}
}
int precedence(char op) {
if (op == '+' || op == '-') return 1;
if (op == '*' || op == '/') return 2;
return 0;
}
void generateThreeAddressCode(const string&
expression, vector<Quadruple>& quadruples,
vector<Triple>& triples, string& lhsVar) {
stack<string> operands;
stack<char> operators;
int tempCount = 1;
lhsVar = "";
auto handleOperation = [&](char op) {
if (operands.size() < 2) {
cerr << "Error: Not enough operands for the
operation " << op << endl;
return;
}
string arg2 = operands.top(); operands.pop();
string arg1 = operands.top(); operands.pop();
string temp = "t" + to_string(tempCount++);
```

```
quadruples.push_back({string(1, op), arg1, arg2,
temp});
triples.push_back({string(1, op), arg1, arg2});
operands.push(temp);
};
for (size_t i = 0; i < expression.length(); ++i) {
if (expression[i] == ' ') continue;
if (isalnum(expression[i])) {
string operand;
while (i < expression.length() &&
isalnum(expression[i])) {
        operand += expression[i++];
      }
      --i;
      operands.push(operand);
    }
    else if (expression[i] == '-' && (i == 0 ||
expression[i - 1] == '(' || expression[i - 1] == '*' ||
expression[i - 1] == '=')) {
      i++;
      string operand;
      while (i < expression.length() &&
isalnum(expression[i])) {
        operand += expression[i++];
      }
      --i;
      string temp = "t" + to_string(tempCount++);
      quadruples.push_back({"-", operand, "", temp});
      triples.push_back({"-", operand, ""});
      operands.push(temp);
    }
```

```
        else if (expression[i] == '(') {
            operators.push(expression[i]);
        }
        else if (expression[i] == ')') {
            while (!operators.empty() && operators.top() !=
'(') {
                handleOperation(operators.top());
                operators.pop();
            }
            if (!operators.empty()) {
                operators.pop();
            }
        }
        else if (precedence(expression[i]) > 0) {
            while (!operators.empty() &&
precedence(operators.top()) >=
precedence(expression[i])) {
                handleOperation(operators.top());
                operators.pop();
            }
            operators.push(expression[i]);
        }
else if (expression[i] == '=') {
lhsVar = operands.top();
operands.pop();
}
}
while (!operators.empty()) {
handleOperation(operators.top());
operators.pop();
}
```

```
if (!operands.empty()) {
string result = operands.top();
operands.pop();
quadruples.push_back({"=", result, "", lhsVar});
triples.push_back({"=", result, ""});
}
}
vector<IndirectTriple> generateIndirectTriples(const
vector<Triple>& triples) {
vector<IndirectTriple> indirectTriples;
for (size_t i = 0; i < triples.size(); ++i) {
indirectTriples.push_back({static_cast<int>(i),
triples[i]});
}
return indirectTriples;
}
int main() {
string expression;
string lhsVar;
cout << "Enter an arithmetic expression: ";
getline(cin, expression);
vector<Quadruple> quadruples;
vector<Triple> triples;
generateThreeAddressCode(expression, quadruples,
triples, lhsVar);
vector<IndirectTriple> indirectTriples =
generateIndirectTriples(triples);
printQuadruples(quadruples);
printTriples(triples);
printIndirectTriples(indirectTriples);
return 0;
```

```
}
```

14) Code Optimization

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#define MAX_EXPR 100
#define MAX_VAR 26

typedef struct {
  char var;
  char expr[MAX_EXPR];
} Assignment;

void trim(char* str) {
  int i, j = 0;
  for(i = 0; str[i] != '\0'; i++) {
    if(str[i] != ' ' && str[i] != '\t') {
      str[j] = str[i];
      j++;
    }
  }
  str[j] = '\0';
}

char findDirectAssignment(Assignment* assignments, int count, char var) {
  for(int i = 0; i < count; i++) {
    if(strlen(assignments[i].expr) == 1 && assignments[i].expr[0] != var)
{
      if(assignments[i].var == var) {
```

```
                return assignments[i].expr[0];
            }
        }
    }
    return var;
}

void substituteVariables(char* expr, Assignment* assignments, int count) {
    char newExpr[MAX_EXPR];
    int newIndex = 0;
    for(int i = 0; expr[i] != '\0'; i++) {
        if(isalpha(expr[i])) {
            char replacement = findDirectAssignment(assignments, count,
expr[i]);
            newExpr[newIndex++] = replacement;
        } else {
            newExpr[newIndex++] = expr[i];
        }
    }
    newExpr[newIndex] = '\0';
    strcpy(expr, newExpr);
}

void optimizeExpressions(Assignment* assignments, int count) {
    substituteVariables(assignments[count-1].expr, assignments, count);
}

int main() {
    Assignment assignments[MAX_VAR];
    int count = 0;
    char line[MAX_EXPR];
```

```
    printf("Enter expressions line by line (type 'END' to finish):\n");
    while(1) {
        fgets(line, MAX_EXPR, stdin);
        line[strcspn(line, "\n")] = 0;

        if(strcmp(line, "END") == 0)
            break;
        trim(line);
        assignments[count].var = line[0];
        strcpy(assignments[count].expr, line + 2);
        count++;
    }
    if(count > 0) {
        optimizeExpressions(assignments, count);
        printf("\nOptimized code:\n");
printf("%c = %s\n\n", assignments[count-1].var, assignments[count
1].expr);
}
return 0;
}
```

Or

```
#include <stdio.h>
#include <string.h>

void optimizeCode(const char* beforeOptimization) {
    char afterOptimization[256];
    strcpy(afterOptimization, "");
    strcat(afterOptimization, "d = a * b + 4\n");
    printf("Before Optimization:\n%s\n", beforeOptimization);
    printf("After Optimization:\n%s", afterOptimization);
}
```

```
int main() {
    const char* beforeOptimization =
        "c = a * b\n"
        "x = a\n"
"d = x * b + 4\n";
optimizeCode(beforeOptimization);
return 0;
}
```

15) Code generation or three address code

```
#include <iostream>
#include <vector>
#include <string>
#include <sstream>
using namespace std;
struct ThreeAddressCode
{
string result;
string arg1;
string op;
string arg2;
};
vector<string> convertToAssembly(const
vector<ThreeAddressCode> &tac)
{
vector<string> assemblyCode;
for (const auto &instr : tac)
{
if (instr.op == "=")
{
```

```
assemblyCode.push_back("MOV " + instr.result + ", " +
instr.arg1);
}
else if (instr.op == "+")
{
assemblyCode.push_back("MOV R0, " + instr.arg1);
assemblyCode.push_back("ADD R0, " + instr.arg2);
assemblyCode.push_back("MOV " + instr.result + ",
R0");
}
else if (instr.op == "-")
{
assemblyCode.push_back("MOV R0, " + instr.arg1);
assemblyCode.push_back("SUB R0, " + instr.arg2);
assemblyCode.push_back("MOV " + instr.result + ",
R0");
}
else if (instr.op == "*")
{
assemblyCode.push_back("MOV R0, " + instr.arg1);
assemblyCode.push_back("MUL R0, " + instr.arg2);
assemblyCode.push_back("MOV " + instr.result + ",
R0");
}
else if (instr.op == "/")
{
assemblyCode.push_back("MOV R0, " + instr.arg1);
assemblyCode.push_back("DIV R0, " + instr.arg2);
assemblyCode.push_back("MOV " + instr.result + ",
R0");
}
```

```
}
return assemblyCode;
}
int main()
{
vector<ThreeAddressCode> tac = {
{"t1", "a", "+", "b"},
{"t2", "t1", "/", "d"},
{"t3", "t2", "*", "e"},
{"a", "t3"}};
vector<string> assembly = convertToAssembly(tac);
cout << "Assembly Code:" << endl;
for (const auto &line : assembly)
{
cout << line << endl;
}
return 0;
}
```

Common subexpression elimination:

```
#include <iostream>
#include <unordered_map>
#include <vector>
#include <string>
using namespace std;
struct Expression {
string result;
string operand1;
string op;
string operand2;
// Overloading equality for comparison in
unordered_map
```

```
bool operator==(const Expression& other) const {
return operand1 == other.operand1 && op == other.op
&& operand2 == other.operand2;
}
};
// Hash function for Expression struct
struct ExpressionHash {
size_t operator()(const Expression& expr) const {
return hash<string>()(expr.operand1) ^
hash<string>()(expr.op) ^
hash<string>()(expr.operand2);
}
};
// Function to perform common subexpression
elimination
void
eliminateCommonSubexpressions(vector<Expression>
 & expressions) {
unordered_map<Expression, string, ExpressionHash>
subexpressionMap;
int tempVarCount = 1;
for (auto& expr : expressions) {
if (subexpressionMap.find(expr) !=
subexpressionMap.end()) {
// If the subexpression is found, replace result with
existing temp variable
cout << expr.result << " = " << subexpressionMap[expr]
<< " // Reusing " <<
subexpressionMap[expr] << endl;
} else {
// Otherwise, add the subexpression to the map and
```

```
generate a new temp variable
string tempVar = "t" + to_string(tempVarCount++);
subexpressionMap[expr] = tempVar;
cout << tempVar << " = " << expr.operand1 << " " <<
expr.op << " " << expr.operand2 << endl;
cout << expr.result << " = " << tempVar << endl;
}
}
}
int main() {
// Example expressions
vector<Expression> expressions = {
{"a", "x", "+", "y"},
{"b", "x", "+", "y"},
{"c", "x", "+", "y"},
{"d", "x", "+", "y"},
{"e", "c", "+", "b"}
};
eliminateCommonSubexpressions(expressions);
return 0;
}
```

Or

```
#include <stdio.h>
#include <string.h>

int isOperator(char c) {
  return c == '+' || c == '-' || c == '*' || c == '/';
}
void generateCode(char* result, char* operand1, char operator, char* operand2) {
  if (operator == '-') {
    printf("MOV %s, R0\n", operand1);
```

```
        printf("SUB %s, R0\n", operand2);
    } else if (operator == '+') {
        printf("ADD %s, R0\n", operand2);
    }
    printf("MOV R0, %s\n", result);
}

int main() {
    char* input[] = {
        "t := a - b",
        "u := a - c",
        "v := t + u",
          "d := v + u"
    };
      int numStatements = sizeof(input) / sizeof(input[0]);
    char result[10], operand1[10], operand2[10], operator;

    // Loop through each input statement
    for (int i = 0; i < numStatements; i++) {
        // Parse the input statement
        sscanf(input[i], "%s := %s %c %s", result, operand1, &operator, operand2);

        // Generate code for each statement
        if (i < 2) { // First two statements: subtraction
            printf("MOV %s, R0\n", operand1);
            printf("SUB %s, R0\n", operand2);
        } else { // Last two statements: addition
            printf("ADD %s, R0\n", operand2);
        }
```

```
        printf("MOV R0, %s\n", result);
    }

    return 0;
}
```

16) Left recursion

```
#include <stdio.h>
#include <string.h>

void main() {
    char input[100], lhs[50], rhs[50], temp[50], newProduction[25][50],
newSymbol[55];
    int i = 0, j, flag = 0;

    printf("Enter production: ");
    scanf("%s", input);

    sscanf(input, "%[^->]->%s", lhs, rhs);

    snprintf(newSymbol, sizeof(newSymbol), "%s'", lhs);

    char *token = strtok(rhs, "|");

    while (token != NULL) {
        if (token[0] == lhs[0]) {
            flag = 1;
            sprintf(newProduction[i++], "%s->%s%s", newSymbol, token + 1,
newSymbol);
        } else {
            sprintf(newProduction[i++], "%s->%s%s", lhs, token, newSymbol);
```

```
        }
        token = strtok(NULL, "|");
    }

    if (flag) {
        sprintf(newProduction[i++], "%s->ε", newSymbol);
        printf("The productions after eliminating Left Recursion are:\n");
        for (j = 0; j < i; j++) {
            printf("%s\n", newProduction[j]);
        }
} else {
printf("The given grammar has no Left Recursion.\n");
}
}
```

17) Left Factoring

```
#include <stdio.h>
#include <string.h>
int main()
{
char ch, lhs[20][20], rhs[20][20][20], temp[20], temp1[20];
int n, n1, count[20], x, y, i, j, k, c[20];
printf("\nEnter the no. of nonterminals: ");
scanf("%d", &n);
n1 = n;
for(i = 0; i < n; i++)
{
printf("\nNonterminal %d\nEnter the no. of productions: ", i + 1);
scanf("%d", &c[i]);
printf("\nEnter LHS: ");
```

```
scanf("%s", lhs[i]);
for(j = 0; j < c[i]; j++)
{
printf("%s->", lhs[i]);
            scanf("%s", rhs[i][j]);
        }
    }

    for(i = 0; i < n; i++)
    {
        count[i] = 1;
        while(memcmp(rhs[i][0], rhs[i][1], count[i]) == 0)
            count[i]++;
    }

    for(i = 0; i < n; i++)
    {
        count[i]--;
        if(count[i] > 0)
        {
            strcpy(lhs[n1], lhs[i]);
            strcat(lhs[i], "'");
            for(k = 0; k < count[i]; k++)
                temp1[k] = rhs[i][0][k];
            temp1[k++] = '\0';

            for(j = 0; j < c[i]; j++)
            {
                for(k = count[i], x = 0; k < strlen(rhs[i][j]); x++, k++)
                    temp[x] = rhs[i][j][k];
                temp[x++] = '\0';
```

```
            if(strlen(rhs[i][j]) == 1)
                strcpy(rhs[n1][1], rhs[i][j]);
            strcpy(rhs[i][j], temp);
        }

        c[n1] = 2;
        strcpy(rhs[n1][0], temp1);
        strcat(rhs[n1][0], lhs[n1]);
        strcat(rhs[n1][0], "'");
        n1++;
    }
}

printf("\n\nThe resulting productions are:\n");
for(i = 0; i < n1; i++)
{
    if(i == 0)
        printf("\n%s -> ∈", lhs[i]);
    else
        printf("\n%s -> ", lhs[i]);
for(j = 0; j < c[i]; j++)
{
printf("%s", rhs[i][j]);
if((j + 1) != c[i])
printf(" | ");
}
printf("\n");
}
return 0;
}
```

Or

```
#include <stdio.h>
#include <string.h>
// Function to remove left factoring
void removeLeftFactoring(char* production) {
char nonTerminal = production[0];
char alpha[20], beta1[20], beta2[20];
int i = 3, j = 0, k = 0, l = 0;
// Separate the common prefix and the different parts
  if (production[i] == 'i' && production[i+1] == 'E' && production[i+2] == 't' &&
production[i+3] == 'S') {
        // Common prefix is "iEtS"
        strcpy(alpha, "iEtS");

        // Find the different parts after the common prefix
        i += 4;
        if (production[i] == '/') {
            i++;
            while (production[i] != '/' && production[i] != '\0') {
                beta1[j++] = production[i++];
            }
            beta1[j] = '\0';

            if (production[i] == '/') {
                i++;
                while (production[i] != '\0') {
                    beta2[k++] = production[i++];
                }
                beta2[k] = '\0';
            }
```

```
        }

        // Print the productions after removing left factoring
        printf("The productions after removing Left Factoring are:\n");
        printf("%c -> %s%c'\n", nonTerminal, alpha, nonTerminal);
        printf("%c' -> %s / %s / ∈\n", nonTerminal, beta1, beta2);
} else {
// If no left factoring is required, print the original production
printf("No left factoring needed.\n");
printf("Production: %s\n", production);
}
}
int main() {
char production[50];
// Input the production
printf("Enter the production (e.g., S->iEtS/iEtSeS/a): ");
scanf("%s", production);
// Remove left factoring
removeLeftFactoring(production);
return 0;
}
```

18) Write a Lex Program to count the Upper Case Character and Lower Case Character.

CODE :

```
%{
#include <stdio.h>
#include <stdlib.h>
int smallets = 0, caplets = 0;
%}
%%
```

```
[A-Z] { caplets++; }
[a-z] { smallets++; }
. |\n { } // Ignore all other characters, including newlines
%%
int main() {
char filename[100];
FILE *file;
printf("Enter the input file name: ");
scanf("%s", filename);
file = fopen(filename, "r");
if (!file) {
perror("Error opening file");
return 1;
}
yyin = file;
yylex();
printf("%d small letters\n", smallets);
printf("%d capital letters\n", caplets);
fclose(file);
return 0;
}
int yywrap() {
return 1;
}
```

19) Write a Lex Program to count the Number of words from the given input.

CODE :

```
%{
#include <stdio.h>
#include <stdlib.h>
```

```
int nwords = 0;
%}
%%
[A-Za-z]+ { nwords++; }
.|\n { }
%%
int main() {
char filename[100];
FILE *file;
printf("Enter the filename: ");
scanf("%s", filename);
file = fopen(filename, "r");
if (!file) {
perror("Error opening file");
return 1;
}
yyin = file;
yylex();
printf("Number of words: %d\n", nwords);
fclose(file);
return 0;
}
int yywrap() {
return 1;
}
```

20) Lex3

Write a LEX program to validate the PAN Number. The input should be stored in a text file. The PAN Card number must satisfy the following conditions,

• It should be 10 characters long.

• The first five characters should be any upper-case alphabets. • The next four-characters shou any number from 0 to 9.

• The last (tenth) character should be the first alphabet of the PAN card holder name.

• It should not contain any white spaces.

Sample Input: Input.txt

ANITHA QWERT1234A

PAARI GREAT7686P

VASU ASDGR 2345 V

KESEVEN HUYT5657668K

Sample Input:

VALID

VALID

INVALID

INVALID

Code:

```
%{
#include <stdio.h>
#include <string.h>
#include <ctype.h>
int ispan(char *pan);
%}
%%
[A-Z]{5}[0-9]{4}[A-Z] {
if (ispan(yytext))
printf("VALID\n");
else
printf("INVALID\n");
}
.|\n {
printf("INVALID\n");
}
%%
```

```
int ispan(char *pan) {
if (strlen(pan) != 10) {
return 0;
}
for (int i = 0; i < 5; i++) {
if (!isupper(pan[i])) {
return 0;
}
}
for (int i = 5; i < 9; i++) {
if (!isdigit(pan[i])) {
return 0;
}
}
if (!isupper(pan[9])) {
return 0;
}
return 1;
}
int main() {
char filename[100];
printf("Enter the filename: ");
scanf("%s", filename);
yyin = fopen(filename, "r");
if (yyin == NULL) {
fprintf(stderr, "Error opening input file\n");
return 1;
}
yylex();
fclose(yyin);
return 0;
```

```
}
int yywrap() {
return 1;
}
```

21) Lexical analyzer to recognize patterns

Code:

```
%{
#include <stdio.h>
#include <stdlib.h>
void print_token(const char* type, const char* value) {
printf("%s : %s\n", type, value);
}
%}
%%
float { print_token("keyword", "float"); }
[a-zA-Z_][a-zA-Z0-9_]* { print_token("Identifier", yytext); }
"=" { print_token("operator", "="); }
"+" { print_token("operator", "+"); }
";" { print_token("operator", ";"); }
[ \t\n]+ ; // Ignore whitespace
. { printf("Unknown character: %s\n", yytext); }
%%
int main(int argc, char** argv) {
if (argc < 2) {
fprintf(stderr, "Usage: %s <file>\n", argv[0]);
exit(EXIT_FAILURE);
}
FILE* file = fopen(argv[1], "r");
if (!file) {
perror("fopen");
exit(EXIT_FAILURE);
```

```
}
yyin = file;
yylex();
fclose(file);
return 0;
}
```

22) evaluating Arithmetic Expression using different Operators.

Sample Input:

Enter the Expression: (5 + 4) * 3-2

Output: 25

Code:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "y.tab.h" // Include the header file generated by Yacc/Bison
void yyerror(const char *s);
%}
%%
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
"+" { return PLUS; }
"-" { return MINUS; }
"*" { return MUL; }
"/" { return DIV; }
"(" { return LPAREN; }
")" { return RPAREN; }
[ \t\n] { /* ignore whitespace */ }
. { yyerror("Unexpected character"); }
%%
int main(void) {
return yyparse();
```

```
}
void yyerror(const char *s) {
fprintf(stderr, "Error: %s\n", s);
}
%{
#include <stdio.h>
#include <stdlib.h>
int yylex(void);
void yyerror(const char *s);
%}
%token NUMBER
%token PLUS MINUS MUL DIV
%token LPAREN RPAREN
%%
expression:
term
| expression PLUS term { $$ = $1 + $3; }
| expression MINUS term { $$ = $1 - $3; }
;
term:
factor
| term MUL factor { $$ = $1 * $3; }
| term DIV factor { $$ = $1 / $3; }
;
factor:
NUMBER
| LPAREN expression RPAREN { $$ = $2; }
;
%%
int main(void) {
return yyparse();
```

```
}
void yyerror(const char *s) {
fprintf(stderr, "Error: %s\n", s);
}
```