#1 Develop a lexical Analyzer to identify identifiers, constants, operators using C program.

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

void lex(char *c) {
    while (*c) {
        if (isspace(*c)) c++;
        else if (isalpha(*c) || *c == '_') {
            printf("Identifier: %c\n", *c);
            while (isalnum(*++c) || *c == '_');
        } else if (isdigit(*c)) {
            printf("Constant: %c\n", *c);
            while (isdigit(*++c));
        } else if (strchr("+-*/=<>!", *c)) {
            printf("Operator: %c\n", *c++);
        } else {
            printf("Symbol: %c\n", *c++);
        }
    }
}

int main() {
    char c[] = "int a = 10 + 20;";
    lex(c);
    return 0;
}
```

#2 Develop a lexical Analyzer to identify whether a given line is a comment or not using C

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

void checkComment(char *c) {
    if (c[0] == '/' && c[1] == '/') {
        printf("Single-line comment\n");
```

```
    } else if (c[0] == '/' && c[1] == '*') {
        printf("Multi-line comment\n");
    } else {
        printf("Not a comment\n");
    }
}

int main() {
    char c[] = "// This is a comment";
    checkComment(c);
    return 0;
}
```

#3 Design a lexical Analyzer for given language should ignore the redundant spaces, tabs and new lines and ignore comments using C

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

void lex(char *c) {
    int i = 0;
    while (c[i] != '\0') {
        if (isspace(c[i])) {
            i++;
            continue;
        }
        if (c[i] == '/' && c[i + 1] == '/') {
            while (c[i] != '\0' && c[i] != '\n') i++;
        } else if (c[i] == '/' && c[i + 1] == '*') {
            i += 2;
            while (c[i] != '\0' && !(c[i] == '*' && c[i + 1] == '/')) i++;
            if (c[i] != '\0') i += 2;
        } else {
            putchar(c[i++]);
        }
    }
}
```

```
int main() {
    char c[] = "  int a = 10;  // this is a comment\n  /* multi-line \n  comment */  b =
20; ";
    lex(c);
    return 0;
}
```

#4 Design a lexical Analyzer to validate operators to recognize the operators +,-,*,/ using regular arithmetic operators using C

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

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

void lex(char *c) {
    int i = 0;
    while (c[i] != '\0') {
        if (isOperator(c[i])) {
            printf("Operator: %c\n", c[i]);
        }
        i++;
    }
}

int main() {
    char c[] = "a + b - c * d / e";
    lex(c);
    return 0;
}
```

#5 Design a lexical Analyzer to find the number of whitespaces and newline characters using C.

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

```
void countWhite(char *c) {
    int spaces = 0, newlines = 0, i = 0;
    while (c[i] != '\0') {
        if (c[i] == ' ') spaces++;
        else if (c[i] == '\n') newlines++;
        i++;
    }
    printf("Spaces: %d\nNewlines: %d\n", spaces, newlines);
}

int main() {
    char c[] = "This is a test.\nThis is on a new line.  ";
    countWhite(c);
    return 0;
}
```

#6 Develop a lexical Analyzer to test whether a given identifier is valid or not using C.

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


const char *keywords[] = {
    "auto", "break", "case", "char", "const", "continue", "default", "do",
    "double", "else", "enum", "extern", "float", "for", "goto", "if",
    "inline", "int", "long", "register", "restrict", "return", "short",
    "signed", "sizeof", "static", "struct", "switch", "typedef", "union",
    "unsigned", "void", "volatile", "while", "_Alignas", "_Alignof",
    "_Atomic", "_Bool", "_Complex", "_Generic", "_Imaginary", "_Noreturn",
    "_Static_assert", "_Thread_local"
};
int keywordCount = sizeof(keywords) / sizeof(keywords[0]);


int isKeyword(char *word) {
    for (int i = 0; i < keywordCount; i++) {
        if (strcmp(word, keywords[i]) == 0) {
```

```
            return 1;
        }
    }
    return 0;
}


int isValidIdentifier(char *str) {

    if (!isalpha(str[0]) && str[0] != '_') {
        return 0;
    }


    for (int i = 1; str[i] != '\0'; i++) {
        if (!isalnum(str[i]) && str[i] != '_') {
            return 0;
        }
    }


    if (isKeyword(str)) {
        return 0;
    }

    return 1;
}


int main() {
    char identifier[100];

    printf("Enter an identifier: ");
    scanf("%s", identifier);

    if (isValidIdentifier(identifier)) {
        printf("Valid identifier\n");
    } else {
```

```
        printf("Invalid identifier\n");
    }

    return 0;
}
```

#7 Write a C program to find FIRST( ) - predictive parser for the given grammar

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

#define MAX_RULES 10
#define MAX_LEN 10

char grammar[MAX_RULES][MAX_LEN];
char first[MAX_RULES][MAX_LEN];
int n;

void find_first(char c, int index) {
    if (!isupper(c)) {
        first[index][strlen(first[index])] = c;
        return;
    }
    for (int i = 0; i < n; i++) {
        if (grammar[i][0] == c) {
            for (int j = 2; grammar[i][j] != '\0'; j++) {
                find_first(grammar[i][j], index);
                if (!isupper(grammar[i][j])) break;
            }
        }
    }
}

int main() {
    printf("Enter number of productions: ");
    scanf("%d", &n);
    printf("Enter productions (e.g., A=Ba):\n");
    for (int i = 0; i < n; i++) scanf("%s", grammar[i]);
```

```
    for (int i = 0; i < n; i++) find_first(grammar[i][0], i);

    printf("\nFIRST sets:\n");
    for (int i = 0; i < n; i++)
        printf("FIRST(%c) = {%s}\n", grammar[i][0], first[i]);

    return 0;
}
```

#8 Write a C program to find FOLLOW( ) - predictive parser for the given grammar

S → AaAb / BbBa

A → ∈

B → ∈

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

#define MAX 10

char productions[MAX][MAX], follow[MAX][MAX];
int prod_count;

void addToSet(char *set, char val) {
    if (strchr(set, val) == NULL) {
        int len = strlen(set);
        set[len] = val;
        set[len + 1] = '\0';
    }
}

void computeFollow(char non_terminal, char *res) {
    if (productions[0][0] == non_terminal)
        addToSet(res, '$');  // Rule 1: FOLLOW(S) = { $ }

    for (int i = 0; i < prod_count; i++) {
```

```
        for (int j = 2; j < strlen(productions[i]); j++) {
            if (productions[i][j] == non_terminal) {
                if (productions[i][j + 1] != '\0') {
                    if (!isupper(productions[i][j + 1]))
                        addToSet(res, productions[i][j + 1]);  // Terminal follows directly
                    else {
                        if (productions[i][j + 1] == 'A' || productions[i][j + 1] == 'B')
                            computeFollow(productions[i][0], res);
                    }
                } else {
                    computeFollow(productions[i][0], res);  // Inherit FOLLOW from LHS
                }
            }
        }
    }
}

int main() {
    printf("Enter number of productions: ");
    scanf("%d", &prod_count);

    printf("Enter productions (e.g., S=AaAb | BbBa):\n");
    for (int i = 0; i < prod_count; i++)
        scanf("%s", productions[i]);

    for (int i = 0; i < prod_count; i++) {
        follow[i][0] = '\0';
        computeFollow(productions[i][0], follow[i]);
    }

    printf("\nFOLLOW sets:\n");
    for (int i = 0; i < prod_count; i++)
        printf("FOLLOW(%c) = {%s}\n", productions[i][0], follow[i]);

    return 0;
}
```

#9 Implement a C program to eliminate left recursion from a given CFG.

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

void eliminateLeftRecursion(char *nonTerminal, char productions[][20], int n) {
    char alpha[10][10], beta[10][10];
    int alphaCount = 0, betaCount = 0;

    for (int i = 0; i < n; i++) {
        if (productions[i][0] == *nonTerminal) {
            strcpy(alpha[alphaCount++], productions[i] + 1);
        } else {
            strcpy(beta[betaCount++], productions[i]);
        }
    }

    if (alphaCount == 0) {
        printf("No left recursion detected.\n");
        return;
    }

    printf("%c -> ", *nonTerminal);
    for (int i = 0; i < betaCount; i++) {
        printf("%s%c'", beta[i], *nonTerminal);
        if (i < betaCount - 1) printf(" | ");
    }
    printf("\n%c' -> ", *nonTerminal);
    for (int i = 0; i < alphaCount; i++) {
        printf("%s%c'", alpha[i], *nonTerminal);
        if (i < alphaCount - 1) printf(" | ");
    }
    printf(" | ε\n");
}

int main() {
    char nonTerminal = 'A';
    char productions[][20] = {"Aa", "b"};
    int n = 2;
    eliminateLeftRecursion(&nonTerminal, productions, n);
```

```
    return 0;
}
```

#10. Implement a C program to eliminate left factoring from a given CFG.

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

#define MAX_RULES 10
#define MAX_LENGTH 100

void eliminateLeftFactoring(char rules[MAX_RULES][MAX_LENGTH], int
ruleCount) {
    for (int i = 0; i < ruleCount; i++) {
        char *production = strstr(rules[i], "->");
        if (!production) continue;

        *production = '\0'; // Split LHS and RHS
        char lhs[MAX_LENGTH], rhs[MAX_LENGTH];
        strcpy(lhs, rules[i]);
        strcpy(rhs, production + 2);

        char *tokens[MAX_RULES];
        int tokenCount = 0;
        char *token = strtok(rhs, "|");
        while (token) {
            tokens[tokenCount++] = token;
            token = strtok(NULL, "|");
        }

        int prefixLen = strlen(tokens[0]);
        for (int j = 1; j < tokenCount; j++) {
            int k = 0;
            while (k < prefixLen && tokens[0][k] == tokens[j][k]) k++;
            prefixLen = k;
        }
```

```
        if (prefixLen == 0) { // No left factoring
            printf("%s->%s\n", lhs, rhs);
            continue;
        }

        char prefix[MAX_LENGTH];
        strncpy(prefix, tokens[0], prefixLen);
        prefix[prefixLen] = '\0';

        printf("%s->%s%c'\n", lhs, prefix, lhs[0]);
        printf("%c'->", lhs[0]);
        int first = 1;
        for (int j = 0; j < tokenCount; j++) {
            if (strncmp(tokens[j], prefix, prefixLen) == 0) {
                if (!first) printf("|");
                printf("%s", tokens[j] + prefixLen);
                first = 0;
            }
        }
        printf("\n");
    }
}

int main() {
    char rules[MAX_RULES][MAX_LENGTH] = {
        "A->ab|ac|ad",
        "B->xyz|xyw|xyt"
    };
    int ruleCount = 2;

    printf("Grammar after removing left factoring:\n");
    eliminateLeftFactoring(rules, ruleCount);

    return 0;
}
```

#12. Write a C program to construct recursive descent parsing for the given grammar

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

char *input;
int position = 0;

void E();
void E_prime();
void T();
void T_prime();
void F();

void error() {
    printf("Parsing failed at position %d: %s\n", position, &input[position]);
    exit(1);
}

void match(char expected) {
    if(input[position] == expected)
        position++;
    else
        error();
}

// Function to check for "id" token
void match_id() {
    if(strncmp(&input[position], "id", 2) == 0)
        position += 2; // Move position forward by 2 (length of "id")
    else
        error();
}

void E() {
    T();
    E_prime();
}
```

```
void E_prime() {
    if(input[position] == '+') {
        match('+');
        T();
        E_prime();
    }
}

void T() {
    F();
    T_prime();
}

void T_prime() {
    if(input[position] == '*') {
        match('*');
        F();
        T_prime();
    }
}

void F() {
    if(strncmp(&input[position], "id", 2) == 0) {
        match_id();  // Match "id" token
    } else if(input[position] == '(') {
        match('(');
        E();
        match(')');
    } else {
        error();
    }
}

int main() {
    char str[100];
    printf("Enter an expression: ");
    scanf("%s", str);
```

```
    input = str;

    E();

    if(input[position] == '\0')
        printf("Parsing successful!\n");
    else
        printf("Parsing failed at position %d\n", position);

    return 0;
}
```

#13. Write a C program to implement either Top Down parsing technique or Bottom Up Parsing technique to check whether the given input string is satisfying the grammar or not.

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

char *input;
int position = 0;

void E();
void E_prime();
void T();
void T_prime();
void F();

void error() {
    printf("Parsing failed at position %d: %s\n", position, &input[position]);
    exit(1);
}

void match(char expected) {
    if(input[position] == expected)
        position++;
    else
```

```
        error();
}

// Match "id" as a token
void match_id() {
    if(strncmp(&input[position], "id", 2) == 0)
        position += 2; // Move forward by 2 (length of "id")
    else
        error();
}

void E() {
    T();
    E_prime();
}

void E_prime() {
    if(input[position] == '+') {
        match('+');
        T();
        E_prime();
    }
}

void T() {
    F();
    T_prime();
}

void T_prime() {
    if(input[position] == '*') {
        match('*');
        F();
        T_prime();
    }
}

void F() {
```

```
    if(strncmp(&input[position], "id", 2) == 0) {
        match_id();  // Match "id"
    } else if(input[position] == '(') {
        match('(');
        E();
        match(')');
    } else {
        error();
    }
}

int main() {
    char str[100];
    printf("Enter an expression: ");
    scanf("%s", str);
    input = str;

    E();

    if(input[position] == '\0')
        printf("Parsing successful!\n");
    else
        printf("Parsing failed at position %d\n", position);

    return 0;
}
```

#14. Implement the concept of Shift reduce parsing in C Programming.

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

#define MAX 100

char stack[MAX];
int top = -1;
char input[MAX];
```

```
int position = 0;

// Push onto the stack
void push(char symbol) {
    stack[++top] = symbol;
}

// Pop from the stack
void pop() {
    top--;
}

// Display stack and input for debugging
void display() {
    printf("Stack: ");
    for (int i = 0; i <= top; i++)
        printf("%c", stack[i]);
    printf("\t Input: %s\n", &input[position]);
}

// Try to reduce the stack contents based on grammar rules
void reduce() {
    while (1) {
        // Reduction: id → E
        if (top >= 1 && stack[top] == 'd' && stack[top - 1] == 'i') {
            pop();
            stack[top] = 'E'; // Replace 'i' with 'E'
            continue;
        }

        // Reduction: E + E → E
        if (top >= 2 && stack[top] == 'E' && stack[top - 1] == '+' && stack[top - 2] ==
'E') {
            pop();
            pop();
            continue;
        }
```

```
        // Reduction: E * E → E
        if (top >= 2 && stack[top] == 'E' && stack[top - 1] == '*' && stack[top - 2] ==
'E') {
            pop();
            pop();
            continue;
        }

        // Reduction: (E) → E
        if (top >= 2 && stack[top] == ')' && stack[top - 1] == 'E' && stack[top - 2] ==
'(') {
            pop();
            pop();
            stack[top] = 'E'; // Replace '(' with 'E'
            continue;
        }

        break; // No further reduction possible
    }
}

// Shift-Reduce Parsing
void shift_reduce_parsing() {
    printf("\nShift-Reduce Parsing Steps:\n");

    while (position < strlen(input)) {
        push(input[position++]); // Shift
        display();
        reduce(); // Try to reduce
        display();
    }

    // Final check: The stack should contain only 'E'
    if (top == 0 && stack[top] == 'E')
        printf("\nParsing successful!\n");
    else
        printf("\nParsing failed!\n");
}
```

```
int main() {
    printf("Enter an expression: ");
    scanf("%s", input);

    shift_reduce_parsing();

    return 0;
}
```

#15. Write a C Program to implement the operator precedence parsing.

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

#define MAX 100

char stack[MAX];
int top = -1;
char input[MAX];

// Operator precedence table
int precedence(char op) {
    if (op == '*' || op == '/') return 2;
    if (op == '+' || op == '-') return 1;
    return 0; // Default for non-operators
}

// Push onto stack
void push(char symbol) {
    stack[++top] = symbol;
}

// Pop from stack
char pop() {
    return (top >= 0) ? stack[top--] : '\0';
```

```
}

// Get the top of the stack
char peek() {
    return (top >= 0) ? stack[top] : '$'; // '$' is used as a bottom marker
}

// Check if character is an operator
int is_operator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/');
}

// Perform reduction when possible
void reduce() {
    while (top >= 2) {
        // Check if stack contains "E op E" pattern
        if (stack[top] == 'E' && is_operator(stack[top - 1]) && stack[top - 2] == 'E') {
            char op = stack[top - 1]; // Store the operator
            pop(); // Remove 'E'
            pop(); // Remove operator
            pop(); // Remove 'E'
            push('E'); // Replace with single 'E'
            printf("Reduce: E %c E → E\n", op);
        } else {
            break; // Stop reducing if no match
        }
    }
}

// Operator Precedence Parsing Algorithm
void operator_precedence_parsing() {
    int position = 0;
    push('$'); // Push bottom marker
    printf("\nOperator Precedence Parsing Steps:\n");

    while (position < strlen(input)) {
        char current = input[position];
```

```c
        if (strncmp(&input[position], "id", 2) == 0) { // If "id" is encountered
            printf("Shift: id\n");
            push('E'); // Reduce "id" → E immediately
            position += 2; // Move past "id"
        }
        else if (is_operator(current)) { // If operator is encountered
            while (is_operator(peek()) && precedence(peek()) >=
precedence(current)) {
                reduce(); // Reduce before shifting new operator
            }
            printf("Shift: %c\n", current);
            push(current);
            position++;
        }
        else {
            printf("Invalid character detected: %c\n", current);
            exit(1);
        }
    }

    // Final reduction to ensure only 'E' remains
    reduce();

    if (top == 1 && stack[top] == 'E' && stack[0] == '$') {
        printf("\nParsing successful!\n");
    } else {
        printf("\nParsing failed!\n");
    }
}

int main() {
    printf("Enter an arithmetic expression: ");
    scanf("%s", input);

    operator_precedence_parsing();

    return 0;
}
```

#16. Write a C Program to Generate the Three address code representation for the given input statement.

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

#define MAX 100

int tempVarCount = 1; // Temporary variable counter

// Function to generate TAC for an expression
void generateTAC(char expression[MAX]) {
    char tokens[MAX][MAX]; // Token storage
    int tokenCount = 0;
    char *token = strtok(expression, " "); // Tokenize input

    // Tokenizing the input expression
    while (token != NULL) {
        strcpy(tokens[tokenCount++], token);
        token = strtok(NULL, " ");
    }

    printf("\nGenerated Three-Address Code:\n");

    char tempVars[MAX][MAX]; // Store temporary variables
    int tempIndex = 0;

    for (int i = 0; i < tokenCount; i++) {
        if (strcmp(tokens[i], "*") == 0 || strcmp(tokens[i], "/") == 0) {
            // Multiplication and division have higher precedence
            printf("t%d = %s %s %s\n", tempVarCount, tokens[i - 1], tokens[i], tokens[i
+ 1]);
            sprintf(tempVars[tempIndex], "t%d", tempVarCount++);
            strcpy(tokens[i - 1], tempVars[tempIndex]); // Replace left operand with
temp var
```

```
            for (int j = i; j < tokenCount - 2; j++) {
                strcpy(tokens[j], tokens[j + 2]); // Shift remaining tokens left
            }
            tokenCount -= 2;
            i--; // Re-evaluate at same position
        }
    }

    for (int i = 0; i < tokenCount; i++) {
        if (strcmp(tokens[i], "+") == 0 || strcmp(tokens[i], "-") == 0) {
            // Addition and subtraction
            printf("t%d = %s %s %s\n", tempVarCount, tokens[i - 1], tokens[i], tokens[i
+ 1]);
            sprintf(tempVars[tempIndex], "t%d", tempVarCount++);
            strcpy(tokens[i - 1], tempVars[tempIndex]); // Replace left operand with
temp var
            for (int j = i; j < tokenCount - 2; j++) {
                strcpy(tokens[j], tokens[j + 2]); // Shift remaining tokens left
            }
            tokenCount -= 2;
            i--; // Re-evaluate at same position
        }
    }

    // Final assignment
    printf("%s = t%d\n", tokens[0], tempVarCount - 1);
}

int main() {
    char expression[MAX];

    printf("Enter an arithmetic expression (use spaces between operators &
operands):\n");
    fgets(expression, MAX, stdin);
    expression[strcspn(expression, "\n")] = 0; // Remove trailing newline

    generateTAC(expression);
```

```
    return 0;
}
```

#17. Write a C program for implementing a Lexical Analyzer to Scan and Count the number of characters, words, and lines in a file.

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

void analyzeFile(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        printf("Error: Cannot open file %s\n", filename);
        return;
    }

    int characters = 0, words = 0, lines = 0;
    char ch, prev = '\0';

    while ((ch = fgetc(file)) != EOF) {
        characters++;
        if (ch == '\n') {
            lines++;
        }
        if (isspace(ch) && !isspace(prev)) {
            words++;
        }
        prev = ch;
    }

    if (!isspace(prev)) {
        words++; // Counting last word if file doesn't end with space
    }

    printf("Characters: %d\n", characters);
    printf("Words: %d\n", words);
    printf("Lines: %d\n", lines);
```

```
    fclose(file);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    analyzeFile(argv[1]);
    return 0;
}
```

#18. Write a C program to implement the back end of the compiler.

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

#define MAX_LEN 100

void generateAssembly(char *expr) {
    char *token = strtok(expr, " ");
    int regCount = 0;

    while (token != NULL) {
        if (isdigit(token[0])) {
            printf("MOV R%d, %s\n", regCount, token);
        } else if (strcmp(token, "+") == 0) {
            printf("ADD R%d, R%d\n", regCount - 2, regCount - 1);
            regCount -= 1;
        } else if (strcmp(token, "-") == 0) {
            printf("SUB R%d, R%d\n", regCount - 2, regCount - 1);
            regCount -= 1;
        } else if (strcmp(token, "*") == 0) {
            printf("MUL R%d, R%d\n", regCount - 2, regCount - 1);
```

```
        regCount -= 1;
    } else if (strcmp(token, "/") == 0) {
        printf("DIV R%d, R%d\n", regCount - 2, regCount - 1);
        regCount -= 1;
    }
    token = strtok(NULL, " ");
    regCount++;
  }
}

int main() {
  char expr[MAX_LEN];
  printf("Enter postfix expression: ");
  fgets(expr, MAX_LEN, stdin);
  expr[strcspn(expr, "\n")] = 0;

  printf("Generated Assembly Code:\n");
  generateAssembly(expr);
  return 0;
}
```

#19. Write a C program to compute LEADING( ) – operator precedence parser for the given grammar

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX 10
char productions[MAX][MAX];
char leading[MAX][MAX];
int numProductions;
void findLeading(char nonTerminal, int index) {
for (int i = 0; i < numProductions; i++) {
if (productions[i][0] == nonTerminal) {
for (int j = 2; productions[i][j] != '\0'; j++) {
if (!isupper(productions[i][j])) { // If terminal, add to LEADING
int len = strlen(leading[index]);
if (!strchr(leading[index], productions[i][j])) {
```

```
leading[index][len] = productions[i][j];
leading[index][len + 1] = '\0';
}
break; // Stop after the first terminal
} else { // If non-terminal, recursively find LEADING
findLeading(productions[i][j], index);
}
}
}
}
}
int main() {
printf("Enter the number of productions: ");
scanf("%d", &numProductions);
getchar(); // Clear newline buffer
printf("Enter the productions (E -> aE format):\n");
for (int i = 0; i < numProductions; i++) {
fgets(productions[i], MAX, stdin);
productions[i][strcspn(productions[i], "\n")] = 0; // Remove newline
}
// Initialize leading array
for (int i = 0; i < MAX; i++) {
leading[i][0] = '\0';
}
printf("\nComputing LEADING():\n");
for (int i = 0; i < numProductions; i++) {
char nonTerminal = productions[i][0];
if (leading[nonTerminal - 'A'][0] == '\0') {
leading[nonTerminal - 'A'][0] = nonTerminal;
leading[nonTerminal - 'A'][1] = ':';
leading[nonTerminal - 'A'][2] = '\0';
findLeading(nonTerminal, nonTerminal - 'A');
}
}
for (int i = 0; i < MAX; i++) {
if (leading[i][0] != '\0') {
printf("LEADING(%c) = { %s }\n", leading[i][0], leading[i] + 2);
}
```

```
}
return 0;
}
```

#20. Write a C program to compute TRAILING( ) – operator precedence parser for the given grammar

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX 10
typedef struct {
char nonTerminal;
char production[MAX][MAX];
int prodCount;
} Grammar;
Grammar grammar[MAX];
int grammarCount = 0;
void computeTrailing(char symbol, char trailingSet[MAX], int *index) {
for (int i = 0; i < grammarCount; i++) {
if (grammar[i].nonTerminal == symbol) {
for (int j = 0; j < grammar[i].prodCount; j++) {
char *prod = grammar[i].production[j];
int len = strlen(prod);
if (len > 0) {
char lastChar = prod[len - 1];
if (!isupper(lastChar)) { // If last char is a terminal
trailingSet[(*index)++] = lastChar;
} else { // If last char is a non-terminal
computeTrailing(lastChar, trailingSet, index);
}
}
}
}
}
}
void printTrailing() {
for (int i = 0; i < grammarCount; i++) {
```

```
char trailingSet[MAX] = {0};
int index = 0;
computeTrailing(grammar[i].nonTerminal, trailingSet, &index);
printf("TRAILING(%c) = { ", grammar[i].nonTerminal);
for (int j = 0; j < index; j++) {
printf("%c ", trailingSet[j]);
}
printf("}\n");
}
}
int main() {
printf("Enter the number of productions: ");
scanf("%d", &grammarCount);
getchar(); // Consume newline
for (int i = 0; i < grammarCount; i++) {
printf("Enter production (e.g., E->E+T): ");
char input[MAX];
fgets(input, MAX, stdin);
input[strcspn(input, "\n")] = 0; // Remove newline
grammar[i].nonTerminal = input[0];
strcpy(grammar[i].production[0], input + 3);
grammar[i].prodCount = 1;
}
printTrailing();
return 0;
}
```

############## LEX PROGRAMS ################

#21. Write a LEX program to identify the capital words from the given input.

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

%%
```

```
[A-Z]+     { printf("Capital Word: %s\n", yytext); }
.          { /* Ignore other characters */ }

%%

int main() {
    yylex();
    return 0;
}

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

#22. Write a LEX Program to check the email address is valid or not.

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

%%
^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$  { printf("Valid Email: %s\n",
yytext); }
.*  { printf("Invalid Email: %s\n", yytext); }
%%

int main() {
    printf("Enter an email: ");
    yylex();
    return 0;
}

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

#23. Implement a LEX program to check whether the mobile number is valid or not.

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

%%
[789][0-9]{9} { printf("Valid Mobile Number: %s\n", yytext); }
.             { /* Ignore other characters */ }

%%

int main() {
    yylex();
    return 0;
}

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

#24. Write a LEX program to count the number of vowels in the given sentence.

```
%{
#include <stdio.h>
int vowel_count = 0;
%}

%%
[AEIOUaeiou] { vowel_count++; }
.            { /* Ignore other characters */ }
\n           { printf("Number of vowels: %d\n", vowel_count); vowel_count = 0; }

%%

int main() {
    yylex();
    return 0;
}
```

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

#25. Write a LEX program to check whether the given input is digit or not.

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

%%
[0-9] { printf("Digit: %s\n", yytext); }
.     { printf("Not a digit: %s\n", yytext); }

%%

int main() {
    yylex();
    return 0;
}

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

#26. Write a LEX specification file to take input C program from a .c file and count tthe number of characters, number of lines & number of words.

```
%{
#include <stdio.h>
int c_count = 0, l_count = 0, w_count = 0;
%}

%%
.    { c_count++; }      /* Count characters */
\n   { l_count++; }      /* Count lines */
[ \t]+   { /* Ignore spaces and tabs */ }
[a-zA-Z0-9_]+ { w_count++; } /* Count words */
%%
```

```
int main() {
    yylex();
    printf("Characters: %d\nLines: %d\nWords: %d\n", c_count, l_count, w_count);
    return 0;
}

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

#27. Write a LEX program to print all the constants in the given C source program file.

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

%%
[0-9]+(\.[0-9]+)? { printf("Constant: %s\n", yytext); } /* Match integers and floats */
.|\n { /* Ignore everything else */ }
%%

int main() {
    yylex();
    return 0;
}

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

#28. Write a LEX program to count the number of Macros defined and header files included in the C program.

```
%{
#include <stdio.h>
int macro_count = 0, header_count = 0;
%}

%%
```

```
^#define { macro_count++; }
^#include { header_count++; }
.|\n { /* Ignore everything else */ }
%%

int main() {
    yylex();
    printf("Number of Macros: %d\nNumber of Header Files: %d\n", macro_count,
header_count);
    return 0;
}

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

#29. Write a LEX program to print all HTML tags in the input file.

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

%%

"<"[^">"]+">" { printf("HTML Tag: %s\n", yytext); }  /* Match HTML tags */
.|\n { /* Ignore everything else */ }

%%

int main() {
    yylex();
    return 0;
}

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

#30. Write a LEX program which adds line numbers to the given C program file and display the same in the standard output.

```
%{
#include <stdio.h>
int line_no = 1;
%}

%%
.* { printf("%d: %s\n", line_no++, yytext); }  /* Add line numbers to each line */
%%

int main() {
    yylex();
    return 0;
}

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

#31. Write a LEX program to count the number of comment lines in a given C program and eliminate them and write into another file.

```
%{
#include <stdio.h>
FILE *fout;
int comment_count = 0;
%}

%%
"//".* { comment_count++; } /* Count single-line comments */
"/*"([^*]|\*+[^*/])*\*+"/" { comment_count++; } /* Count multi-line comments */
.|\n { fputc(yytext[0], fout); } /* Write everything else to output file */
%%

int main() {
    fout = fopen("output.c", "w");
    if (!fout) {
        printf("Error opening file!\n");
        return 1;
```

```
    }
    yylex();
    fclose(fout);
    printf("Number of comment lines removed: %d\n", comment_count);
    return 0;
}

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

#32. Write a LEX Program to convert the substring abc to ABC from the given input string

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

%%
abc { printf("ABC"); }  /* Convert 'abc' to 'ABC' */
.   { putchar(yytext[0]); }  /* Print other characters as they are */
%%

int main() {
    yylex();
    return 0;
}

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

#33. Implement Lexical Analyzer using FLEX (Fast Lexical Analyzer). The program should separate the tokens in the given C program and display with appropriate caption.

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

```
DIGIT   [0-9]+
ID      [a-zA-Z_][a-zA-Z0-9_]*
KEYWORD "int"|"float"|"if"|"else"|"return"|"while"

%%
{KEYWORD}  { printf("Keyword: %s\n", yytext); }
{ID}       { printf("Identifier: %s\n", yytext); }
{DIGIT}    { printf("Number: %s\n", yytext); }
.|\n       { /* Ignore other characters */ }
%%

int main() {
    yylex();
    return 0;
}

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

#34. Write a LEX program to separate the keywords and identifiers.

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

KEYWORD "int"|"float"|"if"|"else"|"return"|"while"
ID      [a-zA-Z_][a-zA-Z0-9_]*

%%
{KEYWORD} { printf("Keyword: %s\n", yytext); }
{ID}      { printf("Identifier: %s\n", yytext); }
.|\n      { /* Ignore other characters */ }
%%

int main() {
    yylex();
    return 0;
}
```

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

#35. Write a LEX program to recognise numbers and words in a statement.

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

DIGIT   [0-9]+
WORD    [a-zA-Z]+

%%
{DIGIT} { printf("Number: %s\n", yytext); }
{WORD}  { printf("Word: %s\n", yytext); }
.|\n    { /* Ignore other characters */ }
%%

int main() {
    yylex();
    return 0;
}

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

#36. Write a LEX program to identify and count positive and negative numbers.

```
%{
#include <stdio.h>
int pos_count = 0, neg_count = 0;
%}

POSITIVE   [0-9]+
NEGATIVE   -[0-9]+

%%
```

```
{POSITIVE} { printf("Positive Number: %s\n", yytext); pos_count++; }
{NEGATIVE} { printf("Negative Number: %s\n", yytext); neg_count++; }
.|\n      { /* Ignore other characters */ }
%%

int main() {
    yylex();
    printf("Total Positive Numbers: %d\n", pos_count);
    printf("Total Negative Numbers: %d\n", neg_count);
    return 0;
}

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

#37. Write a LEX program to validate the URL.

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

%%
"http://"[a-zA-Z0-9.-]+(\.[a-zA-Z]{2,6})("/"[a-zA-Z0-9?=&._-]*)? {
    printf("Valid URL: %s\n", yytext);
}
"https://"[a-zA-Z0-9.-]+(\.[a-zA-Z]{2,6})("/"[a-zA-Z0-9?=&._-]*)? {
    printf("Valid URL: %s\n", yytext);
}
.|\n { /* Ignore other characters */ }

%%

int main() {
    yylex();
    return 0;
}

int yywrap() {
```

```
    return 1;
}
```

#38. Write a LEX program to validate DOB of students.

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

%%
([0-2][0-9]|3[0-1])[-/](0[1-9]|1[0-2])[-/](19[0-9]{2}|20[0-2][0-9]) {
    printf("Valid DOB: %s\n", yytext);
}
.|\n { /* Ignore other characters */ }
%%

int main() {
    yylex();
    return 0;
}

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

#39. Write a LEX program to implement basic mathematical operations.

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

DIGIT   [0-9]+

%%
{DIGIT}      { printf("Number: %s\n", yytext); }
"+"          { printf("Operator: Addition (+)\n"); }
"-"          { printf("Operator: Subtraction (-)\n"); }
"*"          { printf("Operator: Multiplication (*)\n"); }
"/"          { printf("Operator: Division (/)\n"); }
"%"          { printf("Operator: Modulus (%)\n"); }
```

```
.|\n        { /* Ignore other characters */ }
%%

int main() {
    yylex();
    return 0;
}

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