COMPILER DESIGN

CSA1447

NAME: MONESH

REG NO:192324027

Exp. No. 1

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

```
#include <stdio.h>

#include <ctype.h>

#include <string.h>

#define MAX 30

int main()

{

   char input[MAX], identifiers[MAX], constants[MAX], operators[MAX];

   int ic = 0, cc = 0, oc = 0;

printf("Enter the string: ");

scanf("%[^\n]", input);

for (int i = 0; i < strlen(input); i++) {

   if (isalpha(input[i]))

                  {

      identifiers[ic++] = input[i];

   }

                  else if (isdigit(input[i]))

                  {

   constants[cc++] = input[i];

}

                  else if (strchr("+-=*/", input[i]))

                  {
```

```
        operators[oc++] = input[i];
    }
}
printf("\nIdentifiers: ");
for (int i = 0; i < ic; i++) printf("%c ", identifiers[i]);
printf("\nConstants: ");
    for (int i = 0; i < cc; i++) printf("%c ", constants[i]);
    printf("\nOperators: ");
    for (int i = 0; i < oc; i++) printf("%c ", operators[i]);
    return 0;
}
```

Exp. No. 2

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

```
#include<stdio.h>
#include<conio.h>
int main()
{
char com[30];
int i=2,a=0;
printf("\n Enter comment:");
gets(com);
if(com[0]=='/')
{
if(com[1]=='/')
printf("\n It is a comment");
else if(com[1]=='*')
{
```

```
for(i=2;i<=30;i++)
{
if(com[i]=='*'&&com[i+1]=='/')
{
printf("\n It is a comment");
a=1;
break;
}
else
continue;
}
if(a==0)
printf("\n It is not a comment");
}
else
printf("\n It is not a comment");
}
else
printf("\n It is not a comment");
}
```

Exp. No. 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 <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAX_LEN 100
```

```
int isKeyword(char *word)
{
    char *keywords[] = {"main", "auto", "break", "case", "char", "const", "continue", "default",
                        "do", "double", "else", "enum", "extern", "float", "for", "goto",
                        "if", "int", "long", "register", "return", "short", "signed",
                        "sizeof", "static", "struct", "switch", "typedef",
                        "unsigned", "void", "printf", "while"
                                                                    };
int numKeywords = sizeof(keywords) / sizeof(keywords[0]);
for (int i = 0; i < numKeywords; i++)
          {
   if (strcmp(keywords[i], word) == 0)
                        {
       return 1;
   }
    }
    return 0;
}
int main()
{
    FILE *fp;
    char line[MAX_LEN], *token;
    char operators[] = "+-*/%=";
    fp = fopen("flex_input.txt", "r");
if (fp == NULL) {
    printf("Error opening file\n");
    return 1;
}
while (fgets(line, MAX_LEN, fp))
```

```
        {
token = strtok(line, " \n");
while (token != NULL)
                    {
    if (strchr(operators, token[0]) && strlen(token) == 1)
                                {
    printf("%s is an operator\n", token);
}
else if (isKeyword(token))
                                {
    printf("%s is a keyword\n", token);
}
else
                                {
    printf("%s is an identifier\n", token);
            }
            token = strtok(NULL, " \n");
        }
    }
    fclose(fp);
    return 0;
}
```

Exp. No. 4

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

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
int isOperator(char ch)
{
```

```
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/');
}
int main()
{
    FILE *fp;
    char ch;
    fp = fopen("input.txt", "r");
    if (fp == NULL)
            {
    printf("Error opening file!\n");
    return 1;
}
printf("Recognized Operators:\n");
while ((ch = fgetc(fp)) != EOF)
            {
    if (isOperator(ch))
                    {
        printf("%c is an operator\n", ch);
      }
   }
   fclose(fp);
   return 0;
}
```

Exp. No. 5

Design a lexical Analyzer to find the number of whitespaces and newline characters using C. Exp. No. 5 Design a lexical Analyzer to find the number of whitespaces and newline characters using C.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *fp;
    char ch;
```

```
int whitespaceCount = 0, newlineCount = 0;

fp = fopen("input.txt", "r");

if (fp == NULL) {

    printf("Error opening file!\n");

    return 1;

}

while ((ch = fgetc(fp)) != EOF)

                {

    if (ch == ' ' || ch == '\t')

                                {

        whitespaceCount++;

    } else if (ch == '\n') {

        newlineCount++;

    }

}

fclose(fp);

printf("Number of Whitespaces: %d\n", whitespaceCount);

    printf("Number of Newline Characters: %d\n", newlineCount);

    return 0;

}
```

Exp. No. 6

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

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

const char *keywords[] =

{

    "auto", "break", "case", "char", "const", "continue", "default",

    "do", "double", "else", "enum", "extern", "float", "for", "goto",

    "if", "int", "long", "register", "return", "short", "signed",

    "sizeof", "static", "struct", "switch", "typedef", "union",

    "unsigned", "void", "volatile", "while"
```

```
};
const int numKeywords = sizeof(keywords) / sizeof(keywords[0]);
int isKeyword(const char *word)
{
    for (int i = 0; i < numKeywords; i++)
                {
        if (strcmp(word, keywords[i]) == 0)
                        {
            return 1;
        }
    }
    return 0;
}
int isValidIdentifier(const char *identifier)
{
    if (!isalpha(identifier[0]) && identifier[0] != '_')
                {
    return 0;
}
for (int i = 1; i < strlen(identifier); i++)
            {
    if (!isalnum(identifier[i]) && identifier[i] != '_')
                        {
        return 0;
    }
}
if (isKeyword(identifier))
                {
        return 0;
    }
    return 1;
}
int main()
{
```

```
char identifier[50];
printf("Enter an identifier: ");
scanf("%s", identifier);
if (isValidIdentifier(identifier))
            {
   printf("\"%s\" is a valid identifier.\n", identifier);
}
            else
            {
      printf("\"%s\" is NOT a valid identifier.\n", identifier);
   }
   return 0;
}
```

Exp. No. 7

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

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAX 10
char productions[MAX][MAX];
int numProductions;
void findFirst(char symbol)
{
   if (!isupper(symbol))
      {
   printf("%c ", symbol);
   return;
}
for (int i = 0; i < numProductions; i++)
```

```
    {
if (productions[i][0] == symbol)
                {
    char nextSymbol = productions[i][2];
    if (!isupper(nextSymbol))
                    {
        printf("%c ", nextSymbol);
    }
                    else
                    {
        findFirst(nextSymbol);
    }
}
    }
}
int main()
{
    printf("Enter number of productions: ");
    scanf("%d", &numProductions);
    getchar();
    printf("Enter grammar rules (Format: A=B):\n");
for (int i = 0; i < numProductions; i++) {
    fgets(productions[i], MAX, stdin);
    productions[i][strcspn(productions[i], "\n")] = '\0';
}
for (int i = 0; i < numProductions; i++) {
```

```
    char nonTerminal = productions[i][0];
    printf("FIRST(%c) = { ", nonTerminal);
    findFirst(nonTerminal);
    printf("}\n");
}
    return 0;
}
```

Exp. No. 8

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

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX 10
char productions[MAX][MAX];
int numProductions;
void findFollow(char nonTerminal)
{
   if (productions[0][0] == nonTerminal)
          {
      printf("$ ");
   }
for (int i = 0; i < numProductions; i++)
       {
   for (int j = 2; productions[i][j] != '\0'; j++)
                {
      if (productions[i][j] == nonTerminal)
```

```
                    {
if (productions[i][j + 1] != '\0')
                                {
    printf("%c ", productions[i][j + 1]);
}
                                else
                                {
                findFollow(productions[i][0]);
                    }
                }
            }
        }
    }
}
int main()
{
    printf("Enter number of productions: ");
    scanf("%d", &numProductions);
    getchar();
    printf("Enter grammar rules (Format: A=B):\n");
    for (int i = 0; i < numProductions; i++)
            {
    fgets(productions[i], MAX, stdin);
    productions[i][strcspn(productions[i], "\n")] = '\0';
}
for (int i = 0; i < numProductions; i++)
        {
```

```
    char nonTerminal = productions[i][0];
    printf("FOLLOW(%c) = { ", nonTerminal);
    findFollow(nonTerminal);
    printf("}\n");
}
    return 0;
}
```

Exp. No. 9

Implement a C program to eliminate left recursion from a given CFG

```
#include <stdio.h>
#include <string.h>
#define MAX 10
void eliminateLeftRecursion(char nonTerminal, char alpha[], char beta[])
{
    char newNonTerminal = nonTerminal + '\';
    printf("After removing left recursion:\n");
    printf("%c -> %s%c\n", nonTerminal, beta, newNonTerminal);
    printf("%c -> %s%c | ε\n", newNonTerminal, alpha, newNonTerminal);
}
int main()
{
char nonTerminal, alpha[MAX], beta[MAX], production[MAX];
printf("Enter a production (Format: A=Aα|β): ");
fgets(production, MAX, stdin);
production[strcspn(production, "\n")] = '\0';
nonTerminal = production[0];
```

```
char *rhs = strchr(production, '=') + 1;
if (rhs[0] == nonTerminal)
        {
    sscanf(rhs + 1, "%[^|]|%s", alpha, beta);
    eliminateLeftRecursion(nonTerminal, alpha, beta);
    } else
            {
        printf("No left recursion detected: %s\n", production);
    }
    return 0;
}
```

Exp. No. 10

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

```
#include <stdio.h>
#include <string.h>
#define MAX 20
void eliminateLeftFactoring(char nonTerminal, char commonPrefix[], char suffix1[],
char suffix2[])
{
    char newNonTerminal = nonTerminal + '\';
    printf("After removing left factoring:\n");
    printf("%c -> %s%c\n", nonTerminal, commonPrefix, newNonTerminal);
    printf("%c -> %s | %s\n", newNonTerminal, suffix1, suffix2);
}
int main()
{
```

```
    char nonTerminal, commonPrefix[MAX], suffix1[MAX], suffix2[MAX],
production[MAX];
    printf("Enter a production (Format: A=αβ|αγ): ");
    fgets(production, MAX, stdin);
    production[strcspn(production, "\n")] = '\0';
    nonTerminal = production[0];
    char *rhs = strchr(production, '=') + 1;
    sscanf(rhs, "%[^|]%*c%s", commonPrefix, suffix1);
    sscanf(suffix1, "%[^|]|%s", suffix1, suffix2);
    eliminateLeftFactoring(nonTerminal, commonPrefix, suffix1, suffix2);
    return 0;
}
```

Exp. No. 12

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

```
#include <stdio.h>
#include <string.h>
#define MAX 10
struct Symbol
{
    char name[20];
    char type[10];
    int value;
};
struct Symbol table[MAX];
int count = 0;
void insert()
```

```
{
    if (count >= MAX)
            {
        printf("Symbol Table Full!\n");
        return;
    }
    printf("Enter Identifier Name: ");
    scanf("%s", table[count].name);
    printf("Enter Type (int/float/char): ");
    scanf("%s", table[count].type);
    printf("Enter Value: ");
    scanf("%d", &table[count].value);
    count++;
    printf("Inserted Successfully!\n");
}
void search()
{
    char name[20];
    printf("Enter Identifier Name to Search: ");
    scanf("%s", name);
    for (int i = 0; i < count; i++)
            {
        if (strcmp(table[i].name, name) == 0)
                    {
        printf("Found: Name = %s, Type = %s, Value = %d\n", table[i].name, table[i].type,
table[i].value);
```

```
            return;
        }
    }
    printf("Identifier Not Found!\n");
}
void display()
{
    if (count == 0)
            {
        printf("Symbol Table is Empty!\n");
        return;
    }
    printf("\nSymbol Table:\n");
    printf("Name\tType\tValue\n");
    for (int i = 0; i < count; i++)
            {
        printf("%s\t%s\t%d\n", table[i].name, table[i].type, table[i].value);
    }
}
int main()
{
    int choice;
    while (1)
            {
        printf("\nSymbol Table Operations:\n");
```

```
printf("1. Insert\n2. Search\n3. Display\n4. Exit\n");
printf("Enter Choice: ");
scanf("%d", &choice);
switch (choice)
            {
              case 1: insert(); break;
              case 2: search(); break;
              case 3: display(); break;
              case 4: return 0;
              default: printf("Invalid Choice!\n");
        }
    }
}
```

Exp. No. 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 <ctype.h>
#include <string.h>
char input[100];
int pos = 0;
void E();
void T();
void F();
```

```
void match(char expected)
{
    if (input[pos] == expected)
            {
        pos++;
    } else
            {
        printf("Error: Expected '%c' at position %d\n", expected, pos);
        exit(1);
    }
}
void E()
{
    T();
    if (input[pos] == '+')
            {
        match('+');
        E();
    }
}
void T()
{
    F();
    if (input[pos] == '*')
            {
        match('*');
```

```
        T();
    }
}
void F()
{
if (input[pos] == '(')
        {
    match('(');
    E();
    match(')');
} else if (isalnum(input[pos]))
        {
    match(input[pos]);
    }
        else
        {
        printf("Error: Unexpected character '%c' at position %d\n", input[pos], pos);
        exit(1);
    }
}
int main()
{
    printf("Enter an expression: ");
    scanf("%s", input);
    E();
    if (input[pos] == '\0')
```

```
        {
    printf("Parsing successful! Input string satisfies the grammar.\n");
}
        else
        {
            printf("Error: Unexpected extra characters in input.\n");
    }
    return 0;
}
```

Exp. No. 14

Implement the concept of Shift reduce parsing in C Programming.

```
#include <stdio.h>
#include <string.h>
#define MAX 100
char stack[MAX];
char input[MAX];
int top = -1;
int ip = 0;
void push(char c)
{
    if (top < MAX - 1)
        {
        stack[++top] = c;
    }
}
void pop()
```

```
{
    if (top >= 0)
            {
        top--;
    }
}
void reduce()
{
    while (top >= 0)
            {
        if (top >= 2 && stack[top] == 'E' && stack[top - 1] == '+' && stack[top - 2] == 'E')
                    {
        printf("Reducing: E + E → E\n");
        pop(); pop(); pop();
        push('E');
    }
    else if (top >= 2 && stack[top] == 'E' && stack[top - 1] == '*' && stack[top - 2] == 'E')
                {
        printf("Reducing: E * E → E\n");
        pop(); pop(); pop();
        push('E');
    }
    else if (top >= 2 && stack[top] == ')' && stack[top - 1] == 'E' && stack[top - 2] == '(')
                {
        printf("Reducing: ( E ) → E\n");
        pop(); pop(); pop();
```

```
    push('E');
}
else if (stack[top] == 'i')
        {
    printf("Reducing: id → E\n");
        pop();
        push('E');
    }
    else
            {
    break;
    }
}
}
int main()
{
    printf("Enter the input string (Example: i+i*i or (i+i)): ");
    scanf("%s", input);
    printf("\nShift-Reduce Parsing Steps:\n");
    while (input[ip] != '\0')
        {
    printf("Shifting: %c\n", input[ip]);
    push(input[ip]);
    ip++;
    reduce();
}
```

```
if (top == 0 && stack[0] == 'E')
        {
    printf("\nParsing successful! The input string is valid.\n");
} else
        {
        printf("\nParsing failed! Invalid input string.\n");
    }
    return 0;
}
```

Exp. No. 15

Write a C Program to implement the operator precedence parsing.

```
#include <stdio.h>
#include <string.h>
#define MAX 100
char stack[MAX] = "$";
char input[MAX];
int top = 0;
int ip = 0;
char precedenceTable[6][6] = {

    {'>', '<', '<', '>', '<', '>'},
    {'>', '>', '<', '>', '<', '>'},
    {'<', '<', '<', '=', '<', 'E'},
     {'>', '>', 'E', '>', 'E', '>'},
     {'>', '>', 'E', '>', 'E', '>'},
     {'<', '<', '<', 'E', '<', 'A'}
```

```
};
int getIndex(char c)
{
    switch (c) {
        case '+': return 0;
        case '*': return 1;
        case '(': return 2;
        case ')': return 3;
        case 'i': return 4;
        case '$': return 5;
        default: return -1;
    }
char getPrecedence(char stackTop, char inputChar)
{
    int row = getIndex(stackTop);
    int col = getIndex(inputChar);
    if (row == -1 || col == -1) return 'E';
    return precedenceTable[row][col];
}
void reduce()
{
while (top >= 0)
        {
   if ((stack[top] == 'E' && stack[top - 1] == '+' && stack[top - 2] == 'E') ||
      (stack[top] == 'E' && stack[top - 1] == '*' && stack[top - 2] == 'E')) {
      printf("Reducing: E %c E → E\n", stack[top - 1]);
```

```
        top -= 2;
        stack[top] = 'E';
    }
    else if (stack[top] == ')' && stack[top - 1] == 'E' && stack[top - 2] == '(')
                {
        printf("Reducing: ( E ) → E\n");
        top -= 2;
        stack[top] = 'E';
    }
    else if (stack[top] == 'i')
                {
        printf("Reducing: id → E\n");
        stack[top] = 'E';
            }
                    else
                    {
            break;
        }
      }
}
int main()
{
    printf("Enter the input string (Example: i+i*i or (i+i)): ");
    scanf("%s", input);
    strcat(input, "$");
    printf("\nOperator Precedence Parsing Steps:\n");
```

```
while (ip < strlen(input)) {
    char stackTop = stack[top];
    char currentInput = input[ip];
    char relation = getPrecedence(stackTop, currentInput);
    if (relation == '<' || relation == '=')
                {
    printf("Shifting: %c\n", currentInput);
    stack[++top] = currentInput;
    ip++;
}
                else if (relation == '>')
                {
    reduce();
}
                else if (relation == 'A')
                {
    printf("\nParsing successful! The input string is valid.\n");
    return 0;
}
                else
                {
        printf("\nParsing failed! Invalid input string.\n");
        return 1;
    }
}
printf("\nParsing failed! Unexpected end of input.\n");
```

```
    return 1;
}
```

Exp. No. 16

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

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
int tempVarCount = 1;
void newTemp(char *temp)
{
    sprintf(temp, "t%d", tempVarCount++);
}
void generateTAC(char expr[])
{
    char op1, op2, op, result, temp[5];
    int len = strlen(expr);
    char tac[10][20];
    int tacCount = 0;
    for (int i = 0; i < len; i++)
        {
if (expr[i] == '+' || expr[i] == '-' || expr[i] == '*' || expr[i] == '/')
            {
    op1 = expr[i - 1];
    op = expr[i];
    op2 = expr[i + 1];
```

```
    newTemp(temp);
    sprintf(tac[tacCount++], "%s = %c %c %c", temp, op1, op, op2);
    expr[i - 1] = temp[0];
    expr[i] = ' ';
    expr[i + 1] = ' ';
    }
}
result = expr[len - 1];
sprintf(tac[tacCount++], "%c = t%d", result, tempVarCount - 1);
printf("\nGenerated Three-Address Code (TAC):\n");
for (int i = 0; i < tacCount; i++) {
    printf("%s\n", tac[i]);
}
}
int main()
{
    char expression[50];
    printf("Enter an expression (e.g., a = b + c * d): ");
    scanf("%s", expression);
    generateTAC(expression);
    return 0;
}
```

Exp. No. 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>

int main()
{
    FILE *file;

    char filename[50], ch;

    int characters = 0, words = 0, lines = 0;

    int inWord = 0;

    printf("Enter the filename: ");

    scanf("%s", filename);

    file = fopen(filename, "r");

    if (file == NULL) {

        printf("Error: File not found!\n");

        return 1;

    }

    while ((ch = fgetc(file)) != EOF)

            {

        characters++;

        if (ch == '\n')

            lines++;

        if (isspace(ch))

                    {

        inWord = 0;
```

```
}
            else if (!inWord)
            {
    inWord = 1;
    words++;
    }
}
fclose(file);
printf("\nFile Analysis:\n");
printf("Characters: %d\n", characters);
printf("Words: %d\n", words);
printf("Lines: %d\n", lines);
return 0;
}
```

Exp. No. 18

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

```
#include <stdio.h>
#include <string.h>
void generateCode(char expression[])
{
    char op1, op2, op, result;
    int tempCount = 1;
    printf("\nGenerated Assembly-like Code:\n");
    for (int i = 0; i < strlen(expression); i++)
        {
    if (expression[i] == '+' || expression[i] == '-' || expression[i] == '*' || expression[i] == '/')
```

```
            {
op1 = expression[i - 1];
op = expression[i];
op2 = expression[i + 1];
printf("MOV R%d, %c\n", tempCount, op1);
        printf("%s R%d, %c\n", (op == '+') ? "ADD" :
                        (op == '-') ? "SUB" :
                        (op == '*') ? "MUL" : "DIV", tempCount, op2);
        printf("MOV %c, R%d\n", expression[0], tempCount);
        tempCount++;
        break;
    }
}
}
int main()
{
    char expression[50];
    printf("Enter an arithmetic expression (e.g., a=b+c): ");
    scanf("%s", expression);
    generateCode(expression);
    return 0;
}
#include <stdio.h>
#include <string.h>
void generateCode(char expression[])
{
```

```
    char op1, op2, op, result;
    int tempCount = 1;
    printf("\nGenerated Assembly-like Code:\n");
    for (int i = 0; i < strlen(expression); i++) {
        if (expression[i] == '+' || expression[i] == '-' || expression[i] == '*' || expression[i] == '/')
{
op1 = expression[i - 1];
op = expression[i];
op2 = expression[i + 1];
printf("MOV R%d, %c\n", tempCount, op1);
printf("%s R%d, %c\n", (op == '+') ? "ADD" :
                       (op == '-') ? "SUB" :
                       (op == '*') ? "MUL" : "DIV", tempCount, op2);
            printf("MOV %c, R%d\n", expression[0], tempCount);
            tempCount++;
            break;
        }
    }
}
int main()
{
char expression[50];
printf("Enter an arithmetic expression (e.g., a=b+c): ");
scanf("%s", expression);
generateCode(expression);
return 0;
```

}

Exp. No. 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_RULES 10

#define MAX_LENGTH 20

char productions[MAX_RULES][MAX_LENGTH];

char leading[MAX_RULES][MAX_LENGTH];

int numRules;

int isTerminal(char symbol)

{

    return (!isupper(symbol));

}

void findLeading(char nonTerminal, int ruleIndex)

{

    for (int i = 0; i < numRules; i++)

            {

        if (productions[i][0] == nonTerminal)

                {

for (int j = 2; productions[i][j] != '\0'; j++)

                        {

    if (isTerminal(productions[i][j]))

                                {
```

```
            strncat(leading[ruleIndex], &productions[i][j], 1);
            break;
        }
                                else
                                {
                                    findLeading(productions[i][j], ruleIndex);
                                }
                            }
                        }
                    }
                }

int main()
{
    printf("Enter the number of production rules: ");
    scanf("%d", &numRules);
    getchar();
    printf("Enter the grammar rules (e.g., E=+T or T=*F):\n");
    for (int i = 0; i < numRules; i++) {
        printf("Rule %d: ", i + 1);
        fgets(productions[i], MAX_LENGTH, stdin);
        productions[i][strcspn(productions[i], "\n")] = '\0';
    }

    printf("\nComputing LEADING( ) for each non-terminal...\n");
    for (int i = 0; i < numRules; i++) {
```

```
    char nonTerminal = productions[i][0];
    findLeading(nonTerminal, i);
}
printf("\nLEADING( ) Sets:\n");
for (int i = 0; i < numRules; i++) {
    printf("LEADING(%c) = { %s }\n", productions[i][0], leading[i]);
  }
  return 0;
}
```

Exp. No. 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_RULES 10
#define MAX_LENGTH 20
char productions[MAX_RULES][MAX_LENGTH];
char trailing[MAX_RULES][MAX_LENGTH];
int numRules;
int isTerminal(char symbol)
{
  return (!isupper(symbol));
}
void findTrailing(char nonTerminal, int ruleIndex)
{
```

```
for (int i = 0; i < numRules; i++)
        {
    if (productions[i][0] == nonTerminal)
                {
        int len = strlen(productions[i]);
  for (int j = len - 1; j >= 2; j--)
                    {
     if (isTerminal(productions[i][j]))
                                {
        strncat(trailing[ruleIndex], &productions[i][j], 1);
        break;
     }
                                else
                                {
                    findTrailing(productions[i][j], ruleIndex);
                }
            }
        }
    }
}
int main()
{
    printf("Enter the number of production rules: ");
    scanf("%d", &numRules);
    getchar();
    printf("Enter the grammar rules (e.g., E=+T or T=*F):\n");
```

```
for (int i = 0; i < numRules; i++)
        {
    printf("Rule %d: ", i + 1);
    fgets(productions[i], MAX_LENGTH, stdin);
    productions[i][strcspn(productions[i], "\n")] = '\0';
}
printf("\nComputing TRAILING( ) for each non-terminal...\n");
for (int i = 0; i < numRules; i++)
        {
    char nonTerminal = productions[i][0];
    findTrailing(nonTerminal, i);
}
printf("\nTRAILING( ) Sets:\n");
    for (int i = 0; i < numRules; i++)
            {
        printf("TRAILING(%c) = { %s }\n", productions[i][0], trailing[i]);
    }
    return 0;
}
```

Exp. No. 21

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 char_count = 0;   // Number of characters
int word_count = 0;   // Number of words
```

```
int line_count = 0;   // Number of lines
%}
%%
\n          { line_count++; char_count++; }
[^\t\n ]+   { word_count++; char_count += yyleng; }
.           { char_count++; }


%%
int main(int argc, char *argv[])
{
    if (argc != 2)
            {
    printf("Usage: %s <input_file.c>\n", argv[0]);
    return 1;
}
FILE *fp = fopen(argv[1], "r");
if (!fp)
        {
    printf("Error opening file: %s\n", argv[1]);
    return 1;
}
yyin = fp;
yylex();
printf("\nStatistics for %s:\n", argv[1]);
printf("Number of characters: %d\n", char_count);
```

```
    printf("Number of words: %d\n", word_count);
    printf("Number of lines: %d\n", line_count);
    fclose(fp);
    return 0;
}
```

Exp. No. 22

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

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


%}
%%
[0-9]+              { printf("Integer constant: %s\n", yytext); }
[0-9]+\.[0-9]+      { printf("Float constant: %s\n", yytext); }
'\'.\'              { printf("Character constant: %s\n", yytext); }
\"(\\.|[^\\"])*\"   { printf("String constant: %s\n", yytext); }
[ \t\n]             ;
[a-zA-Z_][a-zA-Z0-9_]* ;
.                   ;


%%
int main(int argc, char *argv[])
{
    if (argc != 2)
            {
        printf("Usage: %s <input_file.c>\n", argv[0]);
```

```
    return 1;
}
FILE *fp = fopen(argv[1], "r");
if (!fp)
        {
        printf("Error opening file: %s\n", argv[1]);
        return 1;
    }
    yyin = fp;
    yylex();
    fclose(fp);
    return 0;
}
```

Exp. No. 23

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;
int header_count = 0;
%}
%%
^#include[ \t]+[<"].*[>"]  { header_count++; }
^#define[ \t]+[a-zA-Z_][a-zA-Z0-9_]*  { macro_count++; }
.|\n  ;
%%
```

```
int main(int argc, char *argv[])
{
    if (argc != 2)
            {
        printf("Usage: %s <input_file.c>\n", argv[0]);
        return 1;
    }
    FILE *fp = fopen(argv[1], "r");
    if (!fp)
            {
        printf("Error opening file: %s\n", argv[1]);
        return 1;
    }
        yyin = fp;
        yylex();
        printf("\nStatistics for %s:\n", argv[1]);
        printf("Number of header files: %d\n", header_count);
        printf("Number of macros: %d\n", macro_count);
        fclose(fp);
        return 0;
}
```

Exp. No. 24

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

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

```
%%

"<"[^>]+">"   { printf("HTML Tag: %s\n", yytext); }

.|\n          ;

%%

int main(int argc, char *argv[])

{

    if (argc < 2)

            {

        fprintf(stderr, "Usage: %s <input HTML file>\n", argv[0]);

        return 1;

    }

    FILE *fp = fopen(argv[1], "r");

    if (fp == NULL)

            {

        perror("Error opening file");

        return 1;

}

yyin = fp;

yylex();

fclose(fp);

return 0;

}
```

Exp. No. 25

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_number = 1;

%}

%%

^(.*)    { printf("%d: %s\n", line_number++, yytext); }

%%

int main(int argc, char *argv[])

{

    if (argc < 2)

            {

        fprintf(stderr, "Usage: %s <input C file>\n", argv[0]);

        return 1;

    }

    FILE *fp = fopen(argv[1], "r");

    if (fp == NULL)

            {

        perror("Error opening file");

        return 1;

    }

    yyin = fp;

    yylex();

        fclose(fp);

        return 0;

    }
```

Exp. No. 26

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>

int comment_count = 0;

FILE *cleaned_file;

%}

%%

"//".*  { comment_count++; }

"/*"([^*]|\*+[^*/])*\*+"/"

{

    int i;

    for (i = 0; yytext[i] != '\0'; i++)

        {

        if (yytext[i] == '\n') comment_count++;

    }

}

.|\n    { fputc(yytext[0], cleaned_file); }

%%

int main(int argc, char *argv[])

{

 if (argc < 3)

        {

    fprintf(stderr, "Usage: %s <input C file> <output C file>\n", argv[0]);

    return 1;

}
```

```
FILE *fp = fopen(argv[1], "r");
if (fp == NULL)
        {
    perror("Error opening input file");
    return 1;
}
cleaned_file = fopen(argv[2], "w");
if (cleaned_file == NULL)
        {
    perror("Error opening output file");
    return 1;
}
    yyin = fp;
    yylex();
    fclose(fp);
    fclose(cleaned_file);
    printf("Total comment lines removed: %d\n", comment_count);
    printf("Cleaned code written to: %s\n", argv[2]);
    return 0;
}
```

Exp. No. 27

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

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

```
%%

[A-Z]+   { printf("Capital Word: %s\n", yytext); }

.|\n     ;

%%

int main()

{

    printf("Enter text (Press Ctrl+D to end input):\n");

    yylex();

    return 0;

}
```

Exp. No. 28

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]{2,4}$

{

    printf("Valid Email: %s\n", yytext);

}

.*    { printf("Invalid Email: %s\n", yytext); }

%%

int main()

{

    printf("Enter an email address: ");

    yylex();
```

```
    return 0;
}
```

Exp. No. 29

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

```
%{
#include <stdio.h>
%}
%%
abc   { printf("ABC"); }
.|\n  { printf("%s", yytext); }
%%
int main()
{
    printf("Enter the input string: ");
    yylex();
    return 0;
}
```

Exp. No. 30

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); }
.*    { printf("Invalid Mobile Number: %s\n", yytext); }
%%
```

```
int main()
{
    printf("Enter a mobile number: ");
    yylex();
    return 0;
}
```

Exp. No. 31

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>
#include <stdlib.h>
#include <string.h>
int line = 1;
%}
%%
"int"|"float"|"char"|"double"|"return"|"if"|"else"|"for"|"while"|"void"
{
    printf("<KEYWORD> %s\n", yytext);
}
[a-zA-Z_][a-zA-Z0-9_]*
{
    printf("<IDENTIFIER> %s\n", yytext);
}
[0-9]+(\.[0-9]+)?
{
```

```
    printf("<NUMBER> %s\n", yytext);
}
"=="|"<="|">="|"!="|"="|"<"|">"|"+"|"-"|"*"|"/"
{
    printf("<OPERATOR> %s\n", yytext);
}
"(" | ")" | "{" | "}" | ";" | ","
{
    printf("<SYMBOL> %s\n", yytext);
}
\n { line++; }
[ \t]+ ;
"//".* { printf("<COMMENT> %s\n", yytext); }
"/*"([^*]|[\r\n]|(\*+([^*/]|[\r\n])))*"*/" { printf("<COMMENT> MULTILINE\n"); }
%%
int main()
{
    printf("Enter C program (Press Ctrl+D to end input):\n");
    yylex();
    return 0;
}
```

Exp. No. 32

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

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

```
%}

%%

[aAeEiIoOuU] { vowel_count++; }

.|\n ;

%%

int main()

{

    printf("Enter a sentence: ");

    yylex();

    printf("\nTotal number of vowels: %d\n", vowel_count);

    return 0;

}
```

Exp. No. 33

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

```
%{

#include <stdio.h>

int vowel_count = 0;

%}

%%

[aAeEiIoOuU] { vowel_count++; }

.|\n ;

%%

int main()

{

    printf("Enter a sentence: ");

    yylex();
```

```
    printf("\nTotal number of vowels: %d\n", vowel_count);
    return 0;
}
```

Exp. No. 34

Write a LEX program to separate the keywords and identifiers.

```
%{
#include <stdio.h>
#include <string.h>
int isKeyword(char *word)
{
    char *keywords[] =
            {
        "int", "float", "double", "char", "if", "else", "while", "for", "return",
    "void", "break", "continue", "switch", "case", "struct", "typedef"
};
int i;
for (i = 0; i < sizeof(keywords) / sizeof(keywords[0]); i++)
        {
    if (strcmp(word, keywords[i]) == 0)
        return 1;
}
    return 0;
}
%}
%%
[a-zA-Z_][a-zA-Z0-9_]*
```

```
{
    if (isKeyword(yytext))
        printf("Keyword: %s\n", yytext);
    else
        printf("Identifier: %s\n", yytext);
}
[0-9]+  ;
.       ;
%%
int main()
{
    printf("Enter a C program (Ctrl+D to stop input):\n");
    yylex();
    return 0;
}
```

Exp. No. 35

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

```
%{
#include <stdio.h>
%}
%%
[a-zA-Z_][a-zA-Z0-9_]*   { printf("Word: %s\n", yytext); }
[0-9]+(\.[0-9]+)?        { printf("Number: %s\n", yytext); }
[ \t\n]                  ;
.                        ;
%%
```

```
int main()
{
    printf("Enter a statement:\n");
    yylex();
    return 0;
}
```

Exp. No. 36

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

```
%{
#include <stdio.h>
int positive_count = 0, negative_count = 0;
%}
%%
-[0-9]+(\.[0-9]+)?
{
    printf("Negative Number: %s\n", yytext);
    negative_count++;
}
[0-9]+(\.[0-9]+)?
{
    printf("Positive Number: %s\n", yytext);
    positive_count++;
}
[ \t\n]             ;
.                   ;
%%
```

```
int main()
{
    printf("Enter a statement:\n");
    yylex();
    printf("\nTotal Positive Numbers: %d\n", positive_count);
    printf("Total Negative Numbers: %d\n", negative_count);
    return 0;
}
```

Exp. No. 37

Write a LEX program to validate the URL.

```
%{
#include <stdio.h>
#include <string.h>
%}
%%
^https?:\/\/(www\.)?[a-zA-Z0-9\-]+\.[a-zA-Z]{2,6}(/[a-zA-Z0-9\-._?=&]*)?$
{
    printf("Valid URL: %s\n", yytext);
}
.* {
    printf("Invalid URL: %s\n", yytext);
}
%%
int main()
{
    printf("Enter a URL to validate:\n");
```

```
    yylex();
    return 0;
}
```

Exp. No. 38

Write a LEX program to validate DOB of students.

```
%{
#include <stdio.h>
%}
%%
^(0[1-9]|[12][0-9]|3[01])[-/](0[1-9]|1[0-2])[-/](19[0-9]{2}|20[0-2][0-9])$
{
    printf("Valid DOB: %s\n", yytext);
}
.* {
    printf("Invalid DOB: %s\n", yytext);
}
%%
int main()
{
    printf("Enter a Date of Birth (DD-MM-YYYY or DD/MM/YYYY):\n");
    yylex();
    return 0;
}
```

Exp. No. 39

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

```
%{
```

```
#include <stdio.h>

%}

%%

[0-9]

{

    printf("'%s' is a digit.\n", yytext);

}

. {

    printf("'%s' is NOT a digit.\n", yytext);

}

%%

int main()

{

    printf("Enter a character:\n");

    yylex();

    return 0;

}
```

Exp. No. 40

Write a LEX program to implement basic mathematical operations.

```
%{

#include <stdio.h>

#include <stdlib.h>

float num1, num2, result;

char operator;

%}

%%
```

```
[0-9]+(\.[0-9]+)?   { sscanf(yytext, "%f", &num1); }

[+\-*/]   { operator = yytext[0]; }

[0-9]+(\.[0-9]+)?

{

    sscanf(yytext, "%f", &num2);

switch(operator)

        {

    case '+': result = num1 + num2; break;

    case '-': result = num1 - num2; break;

    case '*': result = num1 * num2; break;

    case '/':

        if(num2 != 0)

            result = num1 / num2;

        else

                        {

            printf("Error: Division by zero!\n");

            exit(1);

        }

        break;

}

printf("Result: %.2f %c %.2f = %.2f\n", num1, operator, num2, result);

}

%%

int main()

{

    printf("Enter a basic arithmetic expression (e.g., 5 + 3):\n");
```

```
    yylex();
    return 0;
}
```