LIST OF EXPERIMENTS

1. The lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value. Develop a lexical Analyzer to identify identifiers, constants, operators using C program.

```
#include <stdio.h>

#include <ctype.h>

#include <string.h>

#define MAX_IDENTIFIER_LENGTH 31

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;

   }

   return 1;

}

int isOperator(char ch) {

   return (ch == '+' || ch == '-' || ch == '*' || ch == '/');

}

void lexicalAnalyzer(char *code) {

   char token[100];

   int index = 0;

   for (int i = 0; code[i] != '\0'; i++) {

      if (isalnum(code[i]) || code[i] == '_') {

         index = 0;

         while (isalnum(code[i]) || code[i] == '_') {

            token[index++] = code[i++];

         }

         token[index] = '\0';

         i--;

         if (isValidIdentifier(token)) {

            printf("Identifier: %s\n", token);
```

```
            } else {
                printf("Constant: %s\n", token);
            }
        } else if (isOperator(code[i])) {
            printf("Operator: %c\n", code[i]);
        }
    }
}
int main() {
    char code[] = "int a = 5; b = a + 2;";
    lexicalAnalyzer(code);
    return 0;
}
```

2. Extend the lexical Analyzer to Check comments, dened as follows in C:
a) A comment begins with // and includes all characters until the end of that line.
b) A comment begins with /* and includes all characters through the next occurrence of the character sequence */Develop a lexical Analyzer to identify whether a given line is a comment or not.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX_LENGTH 100
int is_line_comment(char *line) {
    char *start = strstr(line, "//");
    if (start != NULL) {
        return 1;
    }
    return 0;
}
int is_block_comment(char *line, char *next_line) {
    char start = strstr(line, "/");
    char end = strstr(next_line, "/");
```

```
    if (start != NULL && end != NULL) {
        return 1;
    }
    return 0;
}
int main() {
    char line[MAX_LENGTH];
    char next_line[MAX_LENGTH];
    printf("Enter a line of code: ");
    fgets(line, MAX_LENGTH, stdin);
    if (is_line_comment(line)) {
        printf("This is a line comment.\n");
    } else {
        printf("Enter the next line of code: ");
        fgets(next_line, MAX_LENGTH, stdin);
        if (is_block_comment(line, next_line)) {
            printf("This is a block comment.\n");
        } else {
            printf("This is not a comment.\n");
        }
    }
    return 0;
}
```

3. Design a lexical Analyzer to validate operators to recognize the operators +,-,*,/ using regular Arithmetic operators .

```
#include <stdio.h>
int isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/');
}
void analyzeOperators(char *code) {
    for (int i = 0; code[i] != '\0'; i++) {
```

```
        if (isOperator(code[i])) {
            printf("Operator: %c\n", code[i]);
        }
    }
}
int main() {
    char code[] = "a = b + c * d / e - f;";
    analyzeOperators(code);
    return 0;
}
```

4. Design a lexical Analyzer to find the number of whitespaces and newline characters.

```
#include <stdio.h>
#include <ctype.h>
void countWhitespaces(char *code) {
    int whitespaceCount = 0, newlineCount = 0;
    for (int i = 0; code[i] != '\0'; i++) {
        if (isspace(code[i])) {
            if (code[i] == '\n') newlineCount++;
            else whitespaceCount++;
        }
    }
    printf("Whitespaces: %d, Newlines: %d\n", whitespaceCount, newlineCount);
}
int main() {
    char code[] = "int a = 5; \n b = a + 2; \n";
    countWhitespaces(code);
    return 0;
}
```

5. Develop a lexical Analyzer to test whether a given identifier is valid or not.

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

```
#include <string.h>
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;
   }
   return 1;
}
void checkIdentifier(char *identifier) {
   if (isValidIdentifier(identifier)) {
      printf("\"%s\" is a valid identifier.\n", identifier);
   } else {
      printf("\"%s\" is not a valid identifier.\n", identifier);
   }
}
int main() {
   char id1[] = "var_name";
   char id2[] = "2ndVariable";
   checkIdentifier(id1);
   checkIdentifier(id2);
   return 0;
}
```

6. Implement a C program to eliminate left recursion.

```
#include <stdio.h>
#include <string.h>
typedef struct {
   char lhs;
   char rhs[100];
} Production;
void eliminateLeftRecursion(Production productions[], int numProductions) {
   for (int i = 0; i < numProductions; i++) {
```

```
        char newLhs = productions[i].lhs + 'A' - 'a';
        char* rhs = productions[i].rhs;
        if (rhs[0] == productions[i].lhs) {
            printf("Left recursion detected in production %c -> %s\n", productions[i].lhs, rhs);
            printf("Eliminating left recursion...\n");
            printf("New production: %c -> %s %c'\n", productions[i].lhs, rhs + 1, newLhs);
            printf("New production: %c' -> %s %c' | epsilon\n", newLhs, rhs + 1, newLhs);
        }
    }
}
int main() {
    Production productions[] = {
        {'A', "AB"},
        {'A', "a"},
        {'B', "AB"},
        {'B', "b"}
    };
    int numProductions = sizeof(productions) / sizeof(productions[0]);
    eliminateLeftRecursion(productions, numProductions);
    return 0;
}
```

7. Implement a C program to eliminate left factoring.

```
#include <stdio.h>
#include <string.h>
void eliminateLeftFactoring(char nonTerminal, char alpha[], char beta[]) {
    char newNonTerminal = nonTerminal + '\'';  // Creating new non-terminal (e.g., A')
    printf("Given Grammar:\n");
    printf("%c -> %s%s | %s\n", nonTerminal, alpha, "X", beta);
    printf("\nAfter Eliminating Left Factoring:\n");
    printf("%c -> %s%c\n", nonTerminal, alpha, newNonTerminal);
    printf("%c -> %s | e\n", newNonTerminal, "X");
```

```
}
int main() {
   char nonTerminal, alpha[10], beta[10];
   printf("Enter the non-terminal: ");
   scanf(" %c", &nonTerminal);
   printf("Enter common prefix (alpha): ");
   scanf("%s", alpha);
   printf("Enter remaining part of first production (X): ");
   char X[10];
   scanf("%s", X);
   printf("Enter the second alternative (beta): ");
   scanf("%s", beta);
   eliminateLeftFactoring(nonTerminal, alpha, beta);
   return 0;
}
```

8. Implement a C program to perform symbol table operations.

```
#include <stdio.h>
#include <string.h>
#define MAX_SYMBOLS 10
typedef struct {
   char name[50];
   int type;
} Symbol;
Symbol symbolTable[MAX_SYMBOLS];
int numSymbols = 0;
void insertSymbol(char* name, int type) {
   if (numSymbols < MAX_SYMBOLS) {
      strcpy(symbolTable[numSymbols].name, name);
      symbolTable[numSymbols].type = type;
      numSymbols++;
   }
```

```
}
int searchSymbol(char* name) {
   for (int i = 0; i < numSymbols; i++) {
      if (strcmp(symbolTable[i].name, name) == 0) {
         return i;
      }
   }
   return -1;
}
void displaySymbols() {
   for (int i = 0; i < numSymbols; i++) {
      printf("Name: %s, Type: %d\n", symbolTable[i].name, symbolTable[i].type);
   }
}
int main() {
   insertSymbol("x", 1);
   insertSymbol("y", 2);
   insertSymbol("z", 1);
   printf("Symbols in the symbol table:\n");
   displaySymbols();
   int index = searchSymbol("y");
   if (index != -1) {
      printf("Symbol 'y' found at index %d\n", index);
   } else {
      printf("Symbol 'y' not found\n");
   }
   return 0;
}
```

9. All languages have Grammar. When people frame a sentence we usually say whether the sentence is framed as per the rules of the Grammar or Not. Similarly use the same ideology , implement to check whether the given input string is satisfying the grammar or not .

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

```
#include <string.h>
int checkGrammar(char* input) {
   int length = strlen(input);
   if (input[0] != 'a') {
      return 0;
   }
   for (int i = 1; i < length; i++) {
      if (input[i] != 'b') {
         return 0;
      }
   }
   return 1;
}
int main() {
   char input[100];
   printf("Enter a string: ");
   scanf("%s", input);
   if (checkGrammar(input)) {
      printf("Input string satisfies the grammar.\n");
   } else {
      printf("Input string does not satisfy the grammar.\n");
   }
   return 0;
}
```

10. Write a C program to construct recursive descent parsing.

```
#include <stdio.h>
#include <string.h>
#define SUCCESS 1
#define FAILED 0
int E(), Edash(), T(), Tdash(), F();
const char *cursor;
```

```
char string[64];
int main()
{
   puts("Enter the string");
   scanf("%s", string);
   cursor = string;
   puts("");
   puts("Input         Action");
   puts("--------------------------------");
   if (E() && *cursor == '\0')
   {
      puts("--------------------------------");
      puts("String is successfully parsed");
      return 0;
   }
   else
   {
      puts("--------------------------------");
      puts("Error in parsing String");
      return 1;
   }
}
int E()
{
   printf("%-16s E -> T E'\n", cursor);
   if (T())
   {
      if (Edash())
      {
         return SUCCESS;
      }
```

```
        else
        {
            return FAILED;
        }
    }
    else
    {
        return FAILED;
    }
}
int Edash()
{
    if (*cursor == '+')
    {
        printf("%-16s E' -> + T E'\n", cursor);
        cursor++;
        if (T())
        {
            if (Edash())
            {
                return SUCCESS;
            }
            else
            {
                return FAILED;
            }
        }
        else
        {
            return FAILED;
        }
```

```
    }
    else
    {
        printf("%-16s E' -> $\n", cursor);
        return SUCCESS;
    }
}
int T()
{
    printf("%-16s T -> F T'\n", cursor);
    if (F())
    {
        if (Tdash())
        {
            return SUCCESS;
        }
        else
        {
            return FAILED;
        }
    }
    else
    {
        return FAILED;
    }
}
int Tdash()
{
    if (cursor == '')
    {
        printf("%-16s T' -> * F T'\n", cursor);
```

```
        cursor++;
        if (F())
        {
            if (Tdash())
            {
                return SUCCESS;
            }
            else
            {
                return FAILED;
            }
        }
        else
        {
            return FAILED;
        }
    }
    else
    {
        printf("%-16s T' -> $\n", cursor);
        return SUCCESS;
    }
}
int F()
{
    if (*cursor == '(')
    {
        printf("%-16s F -> ( E )\n", cursor);
        cursor++;
        if (E())
        {
```

```
        if (*cursor == ')')
        {
            cursor++;
            return SUCCESS;
        }
        else
        {
            return FAILED;
        }
    }
    else
    {
        return FAILED;
    }
  }
  else if (*cursor == 'i')
  {
    printf("%-16s F -> i\n", cursor);
    cursor++;
    return SUCCESS;
  }
  else
  {
    return FAILED;
  }
}
```

11. In a class of Grade 3, Mathematics Teacher asked for the Acronym PEMDAS?. All of them
are thinking for a while. A smart kid of the class Kishore of the class says it is Parentheses, Exponentiation, Multiplication, Division, Addition, Subtraction. Can you write a C Program to help the students to understand about the operator precedence parsing for an expression containing more than one operator, the order of evaluation depends on the order of operations.

```
#include <stdio.h>
void printPrecedenceOrder(char* expression) {
   printf("Expression: %s\n", expression);
   printf("1. Evaluate expressions inside parentheses\n");
   printf("2. Evaluate any exponential expressions\n");
   printf("3. Evaluate any multiplication and division operations from left to right\n");
   printf("4. Finally, evaluate any addition and subtraction operations from left to right\n");
}
int main() {
   char expression1[] = "2 + 3 * 4";
   char expression2[] = "(2 + 3) * 4";
   char expression3[] = "10 / 2 + 3";
   char expression4[] = "10 / (2 + 3)";
   printPrecedenceOrder(expression1);
   printf("Result: %d\n\n", 2 + 3 * 4);
   printPrecedenceOrder(expression2);
   printf("Result: %d\n\n", (2 + 3) * 4);
   printPrecedenceOrder(expression3);
   printf("Result: %d\n\n", 10 / 2 + 3);
   printPrecedenceOrder(expression4);
   printf("Result: %d\n\n", 10 / (2 + 3));
   return 0;
}
```

12. The main function of the Intermediate code generation is producing three address code statements for a given input expression. The three address codes help in determining the sequence in which operations are actioned by the compiler. The key work of Intermediate code generators is to simplify the process of Code Generator. Write a C Program to Generate the Three address code representation for the given input statement.

```
#include <stdio.h>
int main() {
   char expression[] = "a+b*c";
   char* token = expression;
```

```
    while (*token != '\0') {
        if (token == '') {
            printf("t0 = %c * %c\n", *(token-1), *(token+1));
            token += 2;
        }
        else if (*token == '+') {
            printf("t1 = %c + t0\n", *(token-2));
            break;
        }
        token++;
    }
    return 0;
}
```

13. Write a C program for implementing a Lexical Analyzer to Count the number of characters,
words, and lines .

```
#include <stdio.h>
int main() {
    char c;
    int charCount = 0, wordCount = 0, lineCount = 0;
    printf("Enter a text (press Ctrl+Z to stop):\n");
    while ((c = getchar()) != EOF) {
        charCount++;
        if (c == '\n')
            lineCount++;
        if (c == ' ' || c == '\n') {
            wordCount++;
        }
    }
    wordCount = (wordCount > 0) ? wordCount - 1 : 0;
    printf("\nSummary:\n");
    printf("Characters: %d\n", charCount);
```

```
    printf("Words: %d\n", wordCount);
    printf("Lines: %d\n", lineCount);
    return 0;
}
```

14. Write a C Program for code optimization to eliminate common subexpression.

```
#include <stdio.h>
int original_function(int a, int b) {
    int x = a + b;
    int y = x * 2;
    int z = x + 3;
    return y + z;
}
int optimized_function(int a, int b) {
    int x = a + b;
    return (x * 2) + x + 3;
}
int main() {
    int a = 2, b = 3;
    printf("Original Function:\n");
    printf("Result: %d\n", original_function(a, b));
    printf("\nOptimized Function:\n");
    printf("Result: %d\n", optimized_function(a, b));
    return 0;
}
```

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

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct {
    char name[10];
    int value;
```

```
} SymbolTableEntry;
typedef struct {
    char opcode[10];
    char operand1[10];
    char operand2[10];
} CodeGenerationEntry;
int main() {
    SymbolTableEntry symbolTable[2];
    strcpy(symbolTable[0].name, "x");
    symbolTable[0].value = 10;
    strcpy(symbolTable[1].name, "y");
    symbolTable[1].value = 20;
    CodeGenerationEntry codeGeneration[3];
    strcpy(codeGeneration[0].opcode, "LOAD");
    strcpy(codeGeneration[0].operand1, "x");
    strcpy(codeGeneration[0].operand2, "");
    strcpy(codeGeneration[1].opcode, "ADD");
    strcpy(codeGeneration[1].operand1, "y");
    strcpy(codeGeneration[1].operand2, "");
    strcpy(codeGeneration[2].opcode, "STORE");
    strcpy(codeGeneration[2].operand1, "z");
    strcpy(codeGeneration[2].operand2, "");
    printf("Symbol Table:\n");
    for (int i = 0; i < 2; i++) {
        printf("%s %d\n", symbolTable[i].name, symbolTable[i].value);
    }
    printf("\nCode Generation:\n");
    for (int i = 0; i < 3; i++) {
        printf("%s %s %s\n", codeGeneration[i].opcode, codeGeneration[i].operand1,
codeGeneration[i].operand2);
    }
```

```
   return 0;
}
```

16. The lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value. Write a LEX specification file to take input C program from a .c file and count t he number of characters, number of lines & number of words.

Input Source Program: (sample.c)

```
#include <stdio.h>
int main()
{
int number1, number2, sum;
printf("Enter two integers: ");
scanf("%d %d", &number1, &number2);
sum = number1 + number2;
printf("%d + %d = %d", number1, number2, sum);
return 0;
}
```

Lex

```
%{

int nchar, nword, nline;

%}

%%

\n { nline++; nchar++; }

[^ \t\n]+ { nword++, nchar += yyleng; }

. { nchar++; }

%%

int yywrap(void) {

return 1;

}

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

yyin = fopen(argv[1], "r");

yylex();

printf("Number of characters = %d\n", nchar);

printf("Number of words = %d\n", nword);

printf("Number of lines = %d\n", nline);
```

```
fclose(yyin);

}
```

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

Input Source Program: (sample.c)

```
#define PI 3.14
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c = 30;
printf("hello");
}
```

Lex

```
%{

#include<stdio.h>

#include<stdlib.h>

%}

digit [0-9]

number {digit}+

floatnum {digit}+\.({digit}+)?

string \"([^\"]|\\\")*\"

%%

{number} { printf("Integer constant: %s\n", yytext); }

{floatnum} { printf("Floating-point constant: %s\n", yytext); }

{string} { printf("String constant: %s\n", yytext); }

. { /* Ignore other characters */ }

%%

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

if(argc != 2) {

printf("Usage: %s <filename>\n", argv[0]);

return 1;

}

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

if (!file) {
```

```
printf("Error opening file: %s\n", argv[1]);

return 1;

}

yyin = file;

yylex();

fclose(file);

return 0;

}

int yywrap() {

return 1;

}
```

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

Input Source Program: (sample.c)

```
#define PI 3.14
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c = 30;
printf("hello");
}
```

Lex

```
%{

#include<stdio.h>

#include<stdlib.h>

int macro_count = 0, header_count = 0;

%}

macro \#define[ ]+[a-zA-Z_][a-zA-Z0-9_]*

header \#include[ ]+<[^>]+>

%%

{macro} { macro_count++; }

{header} { header_count++; }

. { /* Ignore other characters */ }

%%
```

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

if(argc != 2) {

printf("Usage: %s <filename>\n", argv[0]);

return 1;

}

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

if (!file) {

printf("Error opening file: %s\n", argv[1]);

return 1;

}

yyin = file;

yylex();

fclose(file);

printf("Number of Macros: %d\n", macro_count);

printf("Number of Header files: %d\n", header_count);

return 0;

}

int yywrap() {

return 1;

}
```

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

Input Source Program: (sample.html)

```
<html>
<body>
<h1>My First Heading</h1>
<p>My first paragraph.</p>
</body>
</html>
```

Lex

```
%{

int tags;

%}

%%

"<"[^>]*> { tags++; printf("%s \n", yytext); }
```

```
.|\n { }

%%

int yywrap(void) {

return 1; }

int main(void)

{

FILE *f;

char file[10];

printf("Enter File Name : ");

scanf("%s",file);

f = fopen(file,"r");

yyin = f;

yylex();

printf("\n Number of html tags: %d",tags);

fclose(yyin);

}
```

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

Input Source Program: (sample.c)

```
#define PI 3.14
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c = 30;
printf("hello");
}
```

Lex

```
%{
int yylineno;
%}
%%
^(.*)\n printf("%4d\t%s", ++yylineno, yytext);
%%
int yywrap(void) {
return 1;
}
int main(int argc, char *argv[]) {
yyin = fopen(argv[1], "r");
```

```
yylex();
fclose(yyin);
}
```

21. Write a LEX specification count the number of characters, number of lines & number of words. IV

```
%{

int nchar, nword, nline;

%}

%%

\n { nline++; nchar++; }

[^ \t\n]+ { nword++, nchar += yyleng; }

. { nchar++; }

%%

int yywrap(void) {

return 1;

}

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

yyin = fopen(argv[1], "r");

yylex();

printf("Number of characters = %d\n", nchar);

printf("Number of words = %d\n", nword);

printf("Number of lines = %d\n", nline);

fclose(yyin);

}
```

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

Input Source File: (input.c)

```
#include<stdio.h>
int main()
{
int a,b,c; /*varible declaration*/
printf(“enter two numbers”);
scanf(“%d %d”,&a,&b);
c=a+b;//adding two numbers
printf(“sum is %d”,c);
return 0;
```

```
}
Lex
%{
int com=0;
%}
%s COMMENT
%%
"/*" {BEGIN COMMENT;}
<COMMENT>"*/" {BEGIN 0; com++;}
<COMMENT>\n {com++;}
<COMMENT>. {;}
\/\/.* {; com++;}
.|\n {fprintf(yyout,"%s",yytext);}
%%
void main(int argc, char *argv[])
{
if(argc!=3)
{
printf("usage : a.exe input.c output.c\n");
exit(0);
}
yyin=fopen(argv[1],"r");
yyout=fopen(argv[2],"w");
yylex();
printf("\n number of comments are = %d\n",com);
}
int yywrap()
{
return 1;
}
```

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

```
%%
;
[A-Z]+[\t\n ] { printf("%s is a capital word\n",yytext); }
.
%%
int main( )
{
printf("Enter String :\n");
yylex();
}
int yywrap( )
{
return 1;
}
```

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

```
%{
#include <stdio.h>
#include <string.h>
%}
%%
[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,} { printf("Valid email: %s\n", yytext); }
. ;
%%
int main() {
printf("Enter email address: \n");
yylex();
return 0;
}
int yywrap() {
return 1;
}
```

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

```
%{
#include <stdio.h>
%}
%%
abc { printf("ABC"); }
. { printf("%s", yytext); }
%%
int main() {
printf("Enter text: \n");
yylex();
return 0;
}
int yywrap() {
return 1;
}
```

26. The Company ABC runs with employees with several departments. The Organization manager had all the mobile numbers of employees. Assume that you are the manager and need to verify the valid mobile numbers because there may be some invalid numbers present. Implement a LEX program to check whether the mobile number is valid or not.

```
%%
[1-9][0-9]{9} {printf("\nMobile Number Valid\n");}
.+ {printf("\nMobile Number Invalid\n");}
%%
int main()
{
printf("\nEnter Mobile Number : ");
yylex();
printf("\n");
```

```
return 0;

}

int yywrap()

{ }
```

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

Input Source Program: (sample.c)

```
#include<stdio.h>
void main()
{
int a,b,c = 30;
IV
printf("hello");
}
```

```
Lex

digit [0-9]

letter [A-Za-z]

%{

int count_id,count_key;

%}

%%

(stdio.h|conio.h) { printf("%s is a standard library\n",yytext); }

(include|void|main|printf|int) { printf("%s is a keyword\n",yytext); count_key++; }

{letter}({letter}|{digit})* { printf("%s is a identifier\n", yytext); count_id++; }

{digit}+ { printf("%s is a number\n", yytext); }

\"(\\.|[^"\\])*\" { printf("%s is a string literal\n", yytext); }

.|\n { }

%%

int yywrap(void) {

return 1;

}

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

yyin = fopen(argv[1], "r");

yylex();
```

```
printf("number of identifiers = %d\n", count_id);

printf("number of keywords = %d\n", count_key);

fclose(yyin);

}
```

28. In a class, an English teacher was teaching the vowels and consonants to the students. She says “Vowel sounds allow the air to flow freely, causing the chin to drop noticeably, whilst consonant sounds are produced by restricting the air flow”. As a class activity the students are asked to identify the vowels and consonants in the given word/sentence and count the number of elements in each. Write an algorithm to help the student to count the number of vowels and consonants in the given sentence.

```
%{

int vow_count=0;

int const_count =0;

%}

%%

[aeiouAEIOU] {vow_count++;}

[a-zA-Z] {const_count++;}

%%

int yywrap(){}

int main()

{

printf("Enter the string of vowels and consonants:");

yylex();

printf("Number of vowels are: %d\n", vow_count);

printf("Number of consonants are: %d\n", const_count);

return 0;

}
```

29. Keywords are predefined, reserved words used in programming that have special meanings to the compiler. Keywords are part of the syntax and they cannot be used as an identifier. In general there are 32 keywords. The prime function of Lexical Analyser is token Generation. Among the 6 types of tokens, differentiating Keyword and Identifier is a challenging issue. Thus write a LEX program to separate keywords and identifiers.

Input Source File(sample8.c):

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

```
void main()
{
int a,b,c = 30;
printf("hello");
}
```

Code (Lex):

```
digit [0-9]
letter [A-Za-z]
%{
int count_id,count_key;
%}
%%
(stdio.h|conio.h) { printf("%s is a standard library\n",yytext); }
(include|void|main|printf|int) { printf("%s is a keyword\n",yytext); count_key++; }
{letter}({letter}|{digit})* { printf("%s is a identifier\n", yytext); count_id++; }
{digit}+ { printf("%s is a number\n", yytext); }
\"(\\.|[^"\\])*\" { printf("%s is a string literal\n", yytext); }
.|\n { }
%%
int yywrap(void) {
return 1;
}
int main(int argc, char *argv[]) {
yyin = fopen(argv[1], "r");
yylex();
printf("number of identifiers = %d\n", count_id);
printf("number of keywords = %d\n", count_key);
fclose(yyin);
}
```

30. Write a LEX program to recognise numbers and words in a statement. Pooja is a small girl of age 3 always fond of games. Due to the pandemic, she was not allowed to play

outside. So her mother designs a gaming event by showing a flash card. Pooja has to separate the numbers in one list and words in another list shown in the flash card.

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
int words = 0, numbers = 0;
%}
%%
[0-9]+ {
printf("NUMBER: %s\n", yytext);
numbers++;
}
[a-zA-Z]+ {
printf("WORD: %s\n", yytext);
words++;
}
[ \t\n] ; /* Ignore whitespace */
. ; /* Ignore other characters */
%%
int main() {
printf("Enter a statement:\n");
yylex();
printf("\nTotal Words: %d\n", words);
printf("Total Numbers: %d\n", numbers);
return 0;
}
int yywrap() {
return 1;
}
```

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

```
%{
#include <stdio.h>
int pos_count = 0, neg_count = 0;
%}
%%
[+-]?[0-9]+(\.[0-9]+)? {
if(yytext[0] == '-')
neg_count++;
else
pos_count++;
}
\n { printf("Positive Numbers: %d\nNegative Numbers: %d\n", pos_count, neg_count); }
. ;
%%
int main() {
printf("Enter numbers (Ctrl+D to stop):\n");
yylex();
return 0;
}
int yywrap() {
return 1;
}
```

32. A networking company wants to validate the URL for their clients. Write a LEX program to implement the same.

```
%%
((http)|(ftp))s?:\/\/[a-zA-Z0-9](.[a-z])+(.[a-zA-Z0-9+=?]*)* {printf("\nURL Valid\n");}
.+ {printf("\nURL Invalid\n");}
%%
void main()
{
printf("\nEnter URL : ");
```

```
yylex();
printf("\n");
}
int yywrap()
{
}
```

33. School management wants to validate DOB of all students. Write a LEX program to implement it.

```
%{
#include <stdio.h>
%}
%%
((0[1-9])|([1-2][0-9])|(3[0-1]))\/((0[1-9])|(1[0-2]))\/(19[0-9]{2}|2[0-9]{3}) {
printf("Valid DoB: %s\n", yytext);
}
.* {
printf("Invalid DoB: %s\n", yytext);
}
%%
int main() {
printf("Enter DOB (DD/MM/YYYY) to validate (Ctrl+D to stop):\n");
yylex();
return 0;
}
int yywrap() {
return 1;
}
```

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

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

```
%%
[0-9]+ { printf("\nValid digit\n"); }
.* { printf("\nInvalid digit\n"); }
%%
int yywrap() {
return 1;
}
int main() {
printf("Enter input (Ctrl+D to stop):\n");
yylex();
return 0;
}
```

35. A School student was asked to do basic mathematical operations. Implement a LEX program to implement the same.

```
%{
#include <stdio.h>
#include <stdlib.h>
#undef yywrap
#define yywrap() 1
int f1 = 0, f2 = 0;
char oper;
float op1 = 0, op2 = 0, ans = 0;
void eval();
%}
DIGIT [0-9]
NUM {DIGIT}+(\.{DIGIT}+)?
OP [*/+-]
%%
{NUM} {
if (f1 == 0) {
op1 = atof(yytext);
```

```
f1 = 1;
} else {
op2 = atof(yytext);
f2 = 1;
}
}
{OP} {
oper = yytext[0];
}
\n {
if (f1 && f2) {
eval();
printf("Result: %.2f\n", ans);
f1 = f2 = 0; // Reset for next input
}
}
%%
void eval() {
switch(oper) {
case '+': ans = op1 + op2; break;
case '-': ans = op1 - op2; break;
case '*': ans = op1 * op2; break;
case '/':
if (op2 != 0)
ans = op1 / op2;
else
printf("Error: Division by zero\n");
break;
default:
printf("Invalid operator\n");
}
```

```
}
int main() {
printf("Enter arithmetic expression (e.g., 5 + 3). Press Enter to evaluate:\n");
yylex();
return 0;
}
```

36. Write a LEX program to accept string starting with vowel.

```
%{
#include <stdio.h>
%}
%%
^[AEIOUaeiou][a-zA-Z]* { printf("Valid String: %s\n", yytext); }
.* { printf("Invalid String: %s\n", yytext); }
%%
int main() {
printf("Enter a string (Ctrl+D to stop):\n");
yylex();
return 0;
}
int yywrap() {
return 1;
}
```

37. Write a LEX program to find the length of the longest word.

```
%{
#include <stdio.h>
#include <string.h>
int max_length = 0;
char longest_word[100]; // Assuming words won't exceed 100 characters
%}
%%
[a-zA-Z]+ {
```

```
int len = strlen(yytext);
if (len > max_length) {
max_length = len;
strcpy(longest_word, yytext);
}
}
[^a-zA-Z]+ { /* Ignore non-word characters */ }
%%
int main() {
printf("Enter text (Ctrl+D to stop):\n");
yylex();
printf("Longest Word: %s (Length: %d)\n", longest_word, max_length);
return 0;
}
int yywrap() {
return 1;
}
```

38. Write a LEX program to count the frequency of the given word in a given sentence.

```
%{
#include <stdio.h>
#include <string.h>
int count = 0;
char target[100]; // Word to search for
%}
%%
[a-zA-Z]+ {
if (strcmp(yytext, target) == 0) {
count++;
}
}
.|\n { /* Ignore other characters */ }
```

```
%%
int main() {
printf("Enter the word to search: ");
scanf("%s", target);
printf("Enter the sentence (Ctrl+D to stop):\n");
yylex();
printf("The word '%s' appears %d times.\n", target, count);
return 0;
}
int yywrap() {
return 1;
}
```

39. Write a LEX code to replace a word with another word in a file.

```
#include <stdio.h>
#include <string.h>
char old_word[100], new_word[100]; // Words for replacement
%}
%%
[a-zA-Z]+ {
if (strcmp(yytext, old_word) == 0) {
printf("%s", new_word); // Replace old word with new word
} else {
printf("%s", yytext); // Print the word as it is
}
}
. { printf("%c", yytext[0]); } // Print other characters (punctuation, spaces, etc.)
%%
int main() {
printf("Enter the word to be replaced: ");
scanf("%s", old_word);
printf("Enter the new word: ");
```

```
scanf("%s", new_word);
printf("Enter the file content (Ctrl+D to stop):\n");
yylex(); // Process the file
return 0;
}
int yywrap() {
return 1;
}
```

40. Write a LEX program to recognize a word and relational operator.

```
%{
#include <stdio.h>
%}
%%
[a-zA-Z_][a-zA-Z0-9_]* { printf("Word: %s\n", yytext); }
(<=|>=|==|!=|<|>) { printf("Relational Operator: %s\n", yytext); }
[ \t\n] { /* Ignore whitespace */ }
. { printf("Other: %s\n", yytext); } // Print other symbols if needed
%%
int main() {
printf("Enter input (Ctrl+D to stop):\n");
yylex();
return 0;
}
int yywrap() {
return 1;
}
```