Lab Manual Report

of

Compiler Design Laboratory (CSE606)

Bachelor of Technology (CSE)

By

Harsh Sanjaykumar Patel - 22000404

Third Year, Semester 6

Course In-charge: Prof. Vaibhavi Patel

Department of Computer Science and Engineering

School Engineering and Technology

Navrachana University, Vadodara

Spring Semester 2025

TABLE OF CONTENT

Sr. No	Experiment Title
1	1. Write a program to recognize strings starts with ‘a’ over {a, b}. 2. Write a program to recognize strings end with ‘a’. 3. Write a program to recognize strings end with ‘ab’. Take the input from text file. 4. Write a program to recognize strings contains ‘ab’. Take the input from text file.
2	1. Write a program to recognize the valid identifiers. 2. Write a program to recognize the valid operators. 3. Write a program to recognize the valid number. 4. Write a program to recognize the valid comments. 5. Program to implement Lexical Analyzer.
3	To Study about Lexical Analyzer Generator (LEX) and Flex(Fast Lexical Analyzer)
4	Implement following programs using Lex. 1. Write a Lex program to take input from text file and count no of characters, no. of lines & no. of words. 2. Write a Lex program to take input from text file and count number of vowels and consonants. 3. Write a Lex program to print out all numbers from the given file. 4. Write a Lex program which adds line numbers to the given file and display the same into different file. 5. Write a Lex program to printout all markup tags and HTML comments in file.
5	1. Write a Lex program to count the number of C comment lines from a given C program. Also eliminate them and copy that program into separate file. 2. Write a Lex program to recognize keywords, identifiers, operators, numbers, special symbols, literals from a given C program.
6	Program to implement Recursive Descent Parsing in C.
7	1. To Study about Yet Another Compiler-Compiler(YACC). 2. Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, * and / . 3. Create Yacc and Lex specification files are used to generate a calculator which accepts integer type arguments. 4. Create Yacc and Lex specification files are used to convert infix expression to postfix expression.

Practical – 1

AIM:

1. Write a program to recognize strings starts with ‘a’ over {a, b}.
2. Write a program to recognize strings end with ‘a’.
3. Write a program to recognize strings end with ‘ab’. Take the input from text file.
4. Write a program to recognize strings contains ‘ab’. Take the input from text file.

PROGRAM CODE :

1)

```
#include<stdio.h>

void starting_a_over_ab(char []);
void starting_a_over_all(char []);
void main()
{
   // char str[20]="";

   //File Input
   char str[20];
   FILE *filepointer = NULL;
   filepointer=fopen("lab1.txt","r");
   if(filepointer==NULL)
   {
      perror("Error: ");
   }
   fgets(str,60,filepointer);
   fclose(filepointer);
   filepointer=NULL;
```

```
    printf("strings starting with a over (a,b): ");
    starting_a_over_ab(str);

    printf("strings starting with a over all characters: ");
    starting_a_over_all(str);



}

void starting_a_over_ab(char str[])
{
    int state=0,i=0;
    while(str[i]!='\0')
    {
        switch (state)
        {
        case 0:
            if(str[i]=='a'){state=1;}
            else if(str[i]=='b'){state=2;}
            else{state=3;}
            break;

        case 1:
            if(str[i]=='a' || str[i]=='b'){state=1;}
            else{state=3;}

        case 2:
            if(str[i]=='a' || str[i]=='b'){state=2;}
            else{state=3;}

        default:
            break;
        }
        i++;
    }

    if(state==1)
    {
        printf("String %s is Accepted in Language\n\n",str);
    }
    else if(state==2)
    {
        printf("String %s is Not Accepted in Language\n\n",str);
    }
    else if(state==3)
    {
        printf("String (%s) has charecters/symbols not in Language\n\n",str);
    }
    else
    {
        printf("Error");
    }
}

void starting_a_over_all(char str[])
{
    int state=0,i=0;
    while(str[i]!='\0')
    {
        switch (state)
        {
        case 0:
            if(str[i]=='a'){state=1;}
            else{state=2;}
            break;


        default:
            break;
        }
        i++;
    }

    if(state==1)
    {
        printf("String %s is Accepted in Language\n\n",str);
    }
    else if(state==2)
    {
        printf("String %s is Not Accepted in Language\n\n",str);
    }
    else
    {
        printf("Error");
    }
}
```

2)

```
#include <stdio.h>

void ends_with_a(char []);

void main() {

    char str[60];
    FILE *fp = NULL;

    fp = fopen("lab2.txt", "r");
    if (fp == NULL) {
        perror("Error: ");
        return;
    }

    fgets(str, 60, fp);
    fclose(fp);
    fp = NULL;

    printf("Check if string ends with 'a': ");
    ends_with_a(str);
}

void ends_with_a(char str[]) {
    int i = 0;

    while(str[i] != '\0') {
        i++;
    }

    if(i > 0 && str[i - 1] == 'a') {
        printf("String %s ends with 'a' → Accepted\n\n", str);
    } else {
        printf("String %s does NOT end with 'a' → Not Accepted\n\n", str);
    }
}
```

3)

```
#include <stdio.h>

void ends_with_ab(char []);

void main() {
    char str[60];
    FILE *fp = NULL;

    fp = fopen("lab3.txt", "r");
    if (fp == NULL) {
        perror("Error: ");
        return;
    }

    fgets(str, 60, fp);
    fclose(fp);
    fp = NULL;

    printf("Check if string ends with 'ab': ");
    ends_with_ab(str);
}

void ends_with_ab(char str[]) {
    int i = 0;

    while (str[i] != '\0') {
        i++;
    }

    // Remove newline if present
    if(str[i-1] == '\n') {
        str[i-1] = '\0';
        i--;
    }

    if(i >= 2 && str[i - 2] == 'a' && str[i - 1] == 'b') {
        printf("String %s ends with 'ab' → Accepted\n\n", str);
    } else {
        printf("String %s does NOT end with 'ab' → Not Accepted\n\n", str);
    }
}
```

4)

```
#include <stdio.h>

void contains_ab(char []);

void main() {
    char str[60];
    FILE *fp = NULL;

    fp = fopen("lab4.txt", "r");
    if (fp == NULL) {
        perror("Error: ");
        return;
    }

    fgets(str, 60, fp);
    fclose(fp);
    fp = NULL;

    printf("Check if string contains 'ab': ");
    contains_ab(str);
}

void contains_ab(char str[]) {
    int i = 0, found = 0;

    while (str[i] != '\0' && str[i+1] != '\0') {
        if (str[i] == 'a' && str[i + 1] == 'b') {
            found = 1;
            break;
        }
        i++;
    }

    if (found) {
        printf("String %s contains 'ab' → Accepted\n\n", str);
    } else {
        printf("String %s does NOT contain 'ab' → Not Accepted\n\n", str);
    }
}
```

INPUT:

1) abc

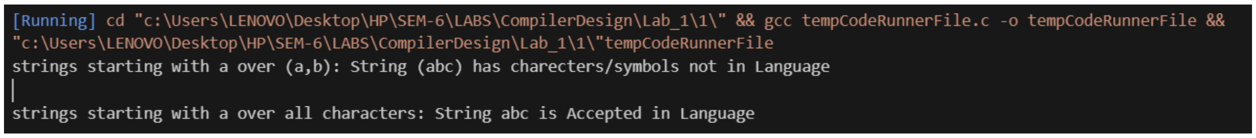
2) bababa

3) ababab

4) aabbbabbbb

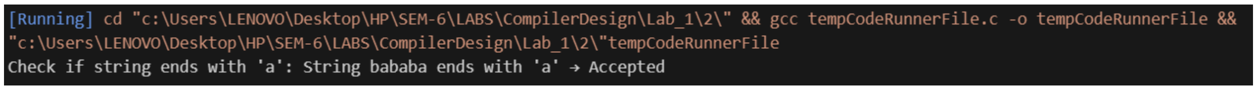
OUTPUT:

1)



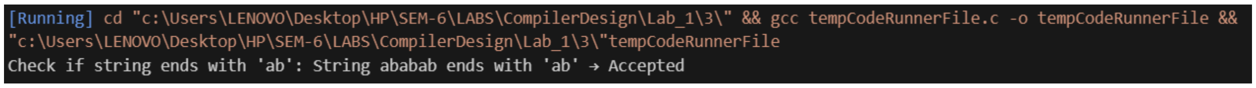
```
[Running] cd "c:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_1\1\" && gcc tempCodeRunnerFile.c -o tempCodeRunnerFile &&
"c:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_1\1\"tempCodeRunnerFile
strings starting with a over (a,b): String (abc) has charecters/symbols not in Language
|
strings starting with a over all characters: String abc is Accepted in Language
```

2)



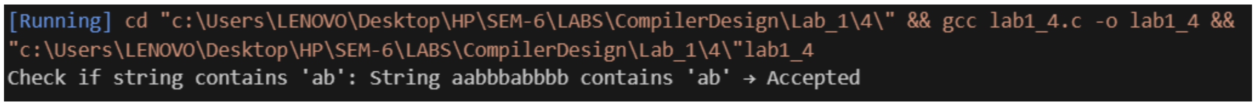
```
[Running] cd "c:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_1\2\" && gcc tempCodeRunnerFile.c -o tempCodeRunnerFile &&
"c:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_1\2\"tempCodeRunnerFile
Check if string ends with 'a': String bababa ends with 'a' → Accepted
```

3)



```
[Running] cd "c:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_1\3\" && gcc tempCodeRunnerFile.c -o tempCodeRunnerFile &&
"c:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_1\3\"tempCodeRunnerFile
Check if string ends with 'ab': String ababab ends with 'ab' → Accepted
```

4)



```
[Running] cd "c:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_1\4\" && gcc lab1_4.c -o lab1_4 &&
"c:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_1\4\"lab1_4
Check if string contains 'ab': String aabbbabbbb contains 'ab' → Accepted
```

Practical – 2

AIM:

1. Write a program to recognize the valid identifiers.
2. Write a program to recognize the valid operators.
3. Write a program to recognize the valid number.
4. Write a program to recognize the valid comments.
5. Program to implement Lexical Analyzer.

PROGRAM CODE :

1)

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

int main() {
   int state=0,i=0,j=0,z=0,flag=0;
   //char str[20] = {"int a b=int"};
   char token_buffer[20];

   // file input
   char str[20];
   FILE *filePointer = NULL;
   filePointer = fopen("lab4.txt", "r");
   // check there is no error opening the file
   if (filePointer == NULL)
   {
      perror("Error: ");
      return(-1);
```

```
}
// read string from file
if(fgets(str, 60, filePointer) != NULL) {
   // print the return value (aka string read in) to terminal
   printf("Input: %s\n", str);
}
fclose(filePointer);
filePointer = NULL;

// Adding space at end
int len_str=0;
for(j=0;str[j]!='\0';j++)
{
   len_str++;
}
str[len_str]=' ';
str[len_str+1]='\0';

printf("\nTokens in Input are listed below:\n");
while(str[i]!='\0')
{
   switch(state)
   {
      case 0:
         memset(token_buffer, '\0', sizeof(token_buffer));
         z=0;
         token_buffer[z]=str[i];
         z++;
         if(str[i]=='i'){state=1;}
```

```
    else if(isalpha(str[i]) || isdigit(str[i]) || str[i]=='_'){state=5;}
    else{state=0;}
    break;

case 1:
    token_buffer[z]=str[i];
    z++;
    if(str[i]=='n'){state=2;}
    else if(isalpha(str[i]) || isdigit(str[i]) || str[i]=='_'){state=5;}
    else{state=0;}
    break;

case 2:
    token_buffer[z]=str[i];
    z++;
    if(str[i]=='t'){state=3;}
    else if(isalpha(str[i]) || isdigit(str[i]) || str[i]=='_'){state=5;}
    else{state=0;}
    break;

case 3:
    token_buffer[z]=str[i];
    z++;
    if(str[i]==' '){
                    state=4;
                    printf("Input is int: %s\n",token_buffer);
                    memset(token_buffer, '\0', sizeof(token_buffer));
                }
    else if(isalpha(str[i]) || isdigit(str[i]) || str[i]=='_'){state=5;}
```

```
            else{state=0;}
            break;

        case 4:
            z=0;
            token_buffer[z]=str[i];
            z++;
            if(str[i]=='i'){state=1;}
            else if(isalpha(str[i]) || isdigit(str[i]) || str[i]=='_'){state=5;}
            else{state=0;}
            break;
        case 5:
            printf("Input is id: %s\n",token_buffer);
            memset(token_buffer, '\0', sizeof(token_buffer));
            z=0;
            token_buffer[z]=str[i];
            z++;
            if(str[i]=='i'){state=1;}
            else if(isalpha(str[i]) || isdigit(str[i]) || str[i]=='_'){state=5,flag=5;}
            else{state=0;}
            break;

        default:
            break;
    }
    i++;
}
printf("\n\n%d",state);
return 0;
```

```
}
```

2)

```
// Problem Statement
/*
Write a program to identify operators
*/
#include<stdio.h>

void id_operators(char []);
void main(){
    char str[20];

    //Input from File
    FILE *filepointer = NULL;
    filepointer=fopen("lab5.txt","r");
    if(filepointer==NULL)
    {
        perror("Error: ");
    }
    fgets(str,20,filepointer);
    fclose(filepointer);
    filepointer=NULL;

    printf("Program to identify all operators.\n");
    id_operators(str);
}

void id_operators(char str[]){
```

```
int i=0,s=0;

while(str[i]!='\0'){
    switch (s)
    {
    case 0:
        if(str[i]=='+'){s=4;}
        else if(str[i]=='-'){s=5;}
        else if(str[i]=='*'){s=6;}
        else if(str[i]=='/'){s=7;}
        else if(str[i]=='%'){s=8;}
        else if(str[i]=='='){s=9;}
        else if(str[i]=='<'){s=10;}
        else if(str[i]=='>'){s=11;}
        else if(str[i]=='!'){s=12;}
        else if(str[i]=='&'){s=13;}
        else if(str[i]=='|'){s=14;}
        else if(str[i]=='~'){s=15;}
        else if(str[i]=='^'){s=16;}
        else{s=0;}
        break;

    case 4:
        if(str[i]=='+'){s=17;}
        else if(str[i]=='='){s=18;}
        else {s=4;}
        break;

    case 5:
```

```
    if(str[i]=='-'){s=19;}
    else if(str[i]=='='){s=20;}
    else {s=5;}
    break;

case 6:
    if(str[i]=='='){s=21;}
    else {s=6;}
    break;

case 7:
    if(str[i]=='='){s=22;}
    else {s=7;}
    break;

case 8:
    if(str[i]=='='){s=23;}
    else {s=8;}
    break;

case 9:
    if(str[i]=='='){s=24;}
    else {s=9;}
    break;

case 10:
    if(str[i]=='<'){s=25;}
    else if(str[i]=='='){s=26;}
    else {s=10;}
    break;

case 11:
    if(str[i]=='>'){s=27;}
    else if(str[i]=='='){s=28;}
    else {s=11;}
    break;

case 12:
    if(str[i]=='='){s=29;}
    else {s=12;}
    break;

case 13:
    if(str[i]=='&'){s=30;}
    else {s=13;}
    break;

case 14:
    if(str[i]=='|'){s=28;}
    else {s=14;}
    break;

case 15:
    s=15;
    break;

case 16:
    s=16;
```

```
            break;

        default:
            break;
        }
        i++;
    }

    if(s==0){printf("No operator found.\n");}
    else if(s==4){printf("Arithmetic operator: + \n");}
    else if(s==5){printf("Arithmetic operator: - \n");}
    else if(s==6){printf("Arithmetic operator: * \n");}
    else if(s==7){printf("Arithmetic operator: / \n");}
    else if(s==8){printf("Arithmetic operator: %% \n");}
    else if(s==9){printf("Assignment operator: = \n");}
    else if(s==10){printf("Relational operator: < \n");}
    else if(s==11){printf("Relational operator: > \n");}
    else if(s==12){printf("Logical operator: ! \n");}
    else if(s==13){printf("Bitwise operator: & \n");}
    else if(s==14){printf("Bitwise operator: | \n");}
    else if(s==15){printf("Bitwise operator: ~ \n");}
    else if(s==16){printf("Bitwise operator: ^ \n");}
    else if(s==17){printf("Unary operator: ++ \n");}
    else if(s==18){printf("Assignment operator: += \n");}
    else if(s==19){printf("Unary operator: -- \n");}
    else if(s==20){printf("Assignment operator: -= \n");}
    else if(s==21){printf("Assignment operator: *= \n");}
    else if(s==22){printf("Assignment operator: /= \n");}
    else if(s==23){printf("Assignment operator: %%= \n");}
    else if(s==24){printf("Relational operator: == \n");}
    else if(s==25){printf("Bitwise operator: << \n");}
    else if(s==26){printf("Relational operator: <= \n");}
    else if(s==27){printf("Bitwise operator: >> \n");}
    else if(s==28){printf("Relational operator: >= \n");}
    else if(s==29){printf("Relational operator: != \n");}
    else if(s==30){printf("Logical operator: && \n");}
    else if(s==31){printf("Logical operator: || \n");}
}
```

3)

```
// Problem Statement
/*
Write a program to identify Numbers
*/
#include<stdio.h>
#include <ctype.h>
#include <string.h>

void id_numbers(char []);
void is_number(int s);

void main(){
    char str[20];

    //Input from File
    FILE *filepointer = NULL;
    filepointer=fopen("lab6.txt","r");
    if(filepointer==NULL)
    {
       perror("Error: ");
    }
    fgets(str,20,filepointer);
    fclose(filepointer);
    filepointer=NULL;

    // Adding space at end
    int len_str=0,j;
    for(j=0;str[j]!='\0';j++)
    {
       len_str++;
    }
    str[len_str]=' ';
    str[len_str+1]=' ';
    str[len_str+2]='\0';

    printf("Program to identify all numbers.");
    id_numbers(str);
}

void id_numbers(char str[]){
    int i=0,s=12,z=0;
    char num_buffer[20];

    while(str[i]!='\0'){
       switch (s)
       {
       case 12:
```

```
    memset(num_buffer, '\0', sizeof(num_buffer));
    z=0;
    num_buffer[z]=str[i];
    z++;
    if(isdigit(str[i])){s=13;}
    else{s=12;}
    break;

case 13:

    if(isdigit(str[i])){s=13;num_buffer[z]=str[i];z++;}
    else if(str[i]=='.'){s=14;num_buffer[z]=str[i];z++;}
    else if(str[i]=='E'){s=16;num_buffer[z]=str[i];z++;}
    else{s=20;}
    break;

case 14:
    if(isdigit(str[i])){s=13;num_buffer[z]=str[i];z++;}
    else{printf("DeadState");}
    break;

case 15:
    if(isdigit(str[i])){s=15;num_buffer[z]=str[i];z++;}
    else if(str[i]=='E'){s=16;num_buffer[z]=str[i];z++;}
    else{s=21;}
    break;

case 16:
    if(str[i]=='+' || str[i]=='-'){s=17;num_buffer[z]=str[i];z++;}
    else if(isdigit(str[i])){s=18;num_buffer[z]=str[i];z++;}
    else{printf("DeadState");}
    break;

case 17:
    if(isdigit(str[i])){s=18;num_buffer[z]=str[i];z++;}
    else{printf("DeadState");}
    break;

case 18:
    if(isdigit(str[i])){s=18;num_buffer[z]=str[i];z++;}
    else{s=19;}
    break;

case 19:
    printf("\n\nInput is Number: %s\n",num_buffer);
    memset(num_buffer, '\0', sizeof(num_buffer));
    z=0;
    num_buffer[z]=str[i];
    z++;
    s=12;
    break;

case 20:
    printf("\n\nInput is Number: %s\n",num_buffer);
    memset(num_buffer, '\0', sizeof(num_buffer));
    z=0;
    num_buffer[z]=str[i];
    z++;
```

```
            s=12;
            break;

        case 21:
            printf("\n\nInput is Number: %s\n",num_buffer);
            memset(num_buffer, '\0', sizeof(num_buffer));
            z=0;
            num_buffer[z]=str[i];
            z++;
            s=12;
            break;

        default:
            break;
        }
        i++;
    }


    // is_number(s); // Checking state operator to check wheather the input is number or
not
}

void is_number(int s){
    if(s==12){printf("\nNot a number");}
    else if(s==13){printf("\nNot a number");}
    else if(s==14){printf("\nNot a number");}
    else if(s==15){printf("\nNot a number");}
    else if(s==16){printf("\nNot a number");}
```

```
   else if(s==17){printf("\nNot a number");}
   else if(s==18){printf("\nNot a number");}
   else if(s==19){printf("\nNumber");}
   else if(s==20){printf("\nNumber");}
   else if(s==21){printf("\nNumber");}
}
```

4)

```
//Problem Statement
/*
Write a program to recognize the valid comments.
*/

#include <stdio.h>

int main(){
        char input[100];
        int state = 0, i = 0;

        FILE *file = fopen("lab3.txt", "r");
   if (file == NULL) {
      printf("Error opening file.\n");
      return 1;
   }

   fscanf(file, "%s", input);
   fclose(file);

   while(input[i] != '\0'){
```

```
switch(state){
        case 0:
                if(input[i] == '/'){state = 1;}
                        else{state = 3;}
                        break;

                case 1:
                        if(input[i] == '/'){state = 2;}
                        else if(input[i] == '*'){state = 4;}
                        else{state = 3;}
                        break;

                case 2:
                        if(input[i] != '\0'){state = 2;}
                        break;

                case 3:
                        state = 3;
                        break;

                case 4:
                        if(input[i] == '*'){state = 5;}
                        else{state = 4;}
                        break;

                case 5:
                        if(input[i] == '/'){state = 6;}
                        else{state = 4;}
                        break;
                case 6:
                    state = 3;
                    break;



                default:
                    break;
            }

        i++;
        }

        printf("State is %d\n",state);

        if(state == 2 || state == 6){
            printf("Valid Comment\n");
        }
        else{
            printf("Invalid Comment\n");
        }



        return 0;
}
```

5)

```
// Problem Statement
/*
```

Write a program to identify Lexemes

*/

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#define BUFFER_SIZE 1000
void check(char *lexeme);

void main() {
   FILE *f1;
   char buffer[BUFFER_SIZE], lexeme[50]; // Static buffer for input and lexeme storage
   char c;
   int f = 0, state = 0,  i = 0;
   f1 = fopen("lab7.txt", "r");
   fread(buffer, sizeof(char), BUFFER_SIZE - 1, f1);
   buffer[BUFFER_SIZE - 1] = '\0'; // Null termination
   fclose(f1);

   while (buffer[f] != '\0') {
      switch (state) {
         case 0:
            c = buffer[f];
            if (isalpha(c) || c == '_') {state = 1;lexeme[i++] = c;} // Identifier / Keyword
            else if (c == ' ' || c == '\t' || c == '\n') {state = 0;} // White Space
            else if (c == '/') {state = 2;lexeme[i++] = c;} // Comment
            else if(isdigit(c)){state=3;lexeme[i++]=c;} // Digit/Number
```

```
    // Operators

    else if(c =='+'){state=4;lexeme[i++] = c;}
    else if(c =='-'){state=5;lexeme[i++] = c;}
    else if(c =='*'){state=6;lexeme[i++] = c;}
    else if(c=='/'){state=7;lexeme[i++] = c;}
    else if(c=='%'){state=8;lexeme[i++] = c;}
    else if(c=='='){state=9;lexeme[i++] = c;}
    else if(c=='<'){state=10;lexeme[i++] = c;}
    else if(c=='>'){state=11;lexeme[i++] = c;}
    else if(c=='!'){state=12;lexeme[i++] = c;}
    else if(c=='&'){state=13;lexeme[i++] = c;}
    else if(c=='|'){state=14;lexeme[i++] = c;}
    else if(c=='~'){state=15;lexeme[i++] = c;}
    else if(c =='^'){state=16;lexeme[i++] = c;}

    // Special Symbols
    else if (c == ';' || c == ',' || c == '{' || c == '}' || c == '(' || c == ')' || c == '\''){
        printf("%c is a symbol\n", c);
        state = 0;
    }

    // Unknown Symbol - Loop back and skip symbol
    else {state=0;}
    break;

// Identifier/Keyword
case 1:
    c = buffer[f];
    if (isalpha(c) || isdigit(c) || c == '_') {state = 1;lexeme[i++] = c;
    } else {
        lexeme[i] = '\0'; // Null-terminate the lexeme
        check(lexeme);  // Check if it's a keyword or identifier
        state = 0;
        i=0;
        f--; // Step back to reprocess the current non-alphanumeric character
    }
    break;

// Comment
case 2:
    c = buffer[f];
    int commentTerminate_flag=0, f1 , f2;
    //Single Line comment
    if(c=='/'){
        lexeme[i++] = c;

        //Skip the commented strings
        for(;commentTerminate_flag!=1;f++){
            f1 = f+1;
            if(buffer[f1]=='\n'){
                commentTerminate_flag=1;
            }
        }
        f++;

        // Complete the lexeme for singleline comments
        lexeme[i]='\0';
        i=0;
        state = 0;
        printf("%s is valid comment\n",lexeme);

    }

    // Multiline comment
    else if(c=='*'){
        lexeme[i++] = c;

        //Skip the commented strings
        int commentTerminate_flag=0, f1 , f2;
        for(;commentTerminate_flag!=1;f++){
            f1 = f+1;
            f2 = f+2;
            if(buffer[f1]=='*' && buffer[f2]=='/'){
                lexeme[i++]=buffer[f1];
                lexeme[i++]=buffer[f2];
                commentTerminate_flag=1;
            }
        }
        f+=2;

        // Complete the lexeme for multiline comments
        lexeme[i]='\0';
        i=0;
        state = 0;
        printf("%s is valid comment\n",lexeme);
    }
```

```
    // Not a comment but division symbol or other !!!
    else{
        // Empty the lexeme and backtrack the current symbol
        memset(lexeme, '\0', sizeof(lexeme));
        i=0;
        state=0;
        f--;
    }
    break;

// Digit/Number
case 3:
    c=buffer[f];
    if(isdigit(c)){state=3;lexeme[i++]=c;}
    else if(c=='.') {state=3;lexeme[i++]=c;}
    else if(c=='E'||c=='e') {state=3;lexeme[i++]=c;}
    else {
            lexeme[i]='\0';
            printf("%s is valid number\n",lexeme);
            i=0;
            state=0;
            f--;
        }

// Cases 4 to 16 - Operators
case 4:
    if(buffer[f]=='+'){state=17;lexeme[i++]=c;}
    else if(buffer[f]=='='){state=18;lexeme[i++]=c;}
    else{
            lexeme[i]='\0';
            printf("%s is Operator\n",lexeme);
            i=0;
            state=0;
            f--;
        }
    break;

case 5:
    if(buffer[f]=='-'){state=19;lexeme[i++]=c;}
    else if(buffer[f]=='='){state=20;lexeme[i++]=c;}
    else{
            lexeme[i]='\0';
            printf("%s is Operator\n",lexeme);
            i=0;
            state=0;
            f--;
        }
    break;

case 6:
    if(buffer[f]=='='){state=21;lexeme[i++]=c;}
    else{
            lexeme[i]='\0';
            printf("%s is Operator\n",lexeme);
            i=0;
            state=0;
            f--;
        }
    break;

case 7:
    if(buffer[f]=='='){state=22;lexeme[i++]=c;}
    else{
            lexeme[i]='\0';
            printf("%s is Operator\n",lexeme);
            i=0;
            state=0;
            f--;
        }
    break;

case 8:
    if(buffer[f]=='='){state=23;lexeme[i++]=c;}
    else{
            lexeme[i]='\0';
            printf("%s is Operator\n",lexeme);
            i=0;
            state=0;
            f--;
        }
    break;

case 9:
    if(buffer[f]=='='){state=24;lexeme[i++]=c;}
    else{
            lexeme[i]='\0';
            printf("%s is Operator\n",lexeme);
            i=0;
            state=0;
            f--;
        }
    break;

case 10:
    if(buffer[f]=='<'){state=25;lexeme[i++]=c;}
    else if(buffer[f]=='='){state=26;lexeme[i++]=c;}
    else{
            lexeme[i]='\0';
            printf("%s is Operator\n",lexeme);
            i=0;
            state=0;
            f--;
        }
    break;

case 11:
    if(buffer[f]=='>'){state=27;lexeme[i++]=c;}
    else if(buffer[f]=='='){state=28;lexeme[i++]=c;}
    else{
            lexeme[i]='\0';
            printf("%s is Operator\n",lexeme);
            i=0;
            state=0;
            f--;
        }
    break;
```

```
case 12:
   if(buffer[f]=='='){state=29;lexeme[i++]=c;}
   else{
         lexeme[i]='\0';
         printf("%s is Operator\n",lexeme);
         i=0;
         state=0;
         f--;
      }
   break;

case 13:
   if(buffer[f]=='&'){state=30;lexeme[i++]=c;}
   else{
         lexeme[i]='\0';
         printf("%s is Operator\n",lexeme);
         i=0;
         state=0;
         f--;
      }
   break;

case 14:
   if(buffer[f]=='|'){state=28;lexeme[i++]=c;}
   else{
         lexeme[i]='\0';
         printf("%s is Operator\n",lexeme);
         i=0;
```

```
                    state=0;
                    f--;
                }
                break;

            case 15:
                lexeme[i]='\0';
                printf("%s is Operator\n",lexeme);
                i=0;
                state=0;
                f--;
                break;

            case 16:
                lexeme[i]='\0';
                printf("%s is Operator\n",lexeme);
                i=0;
                state=0;
                f--;
                break;

            default:
                break;
        }
        f++;
    }
}
void check(char *lexeme) {
    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"
};
    for (int i = 0; i < 32; i++) {
        if (strcmp(lexeme, keywords[i]) == 0) {
            printf("%s is a keyword\n", lexeme);
            return;
        }
    }
    printf("%s is an identifier\n", lexeme);
}
```

INPUT:

1) int a = b;

2) %=

3) ab 530.500E500 abc

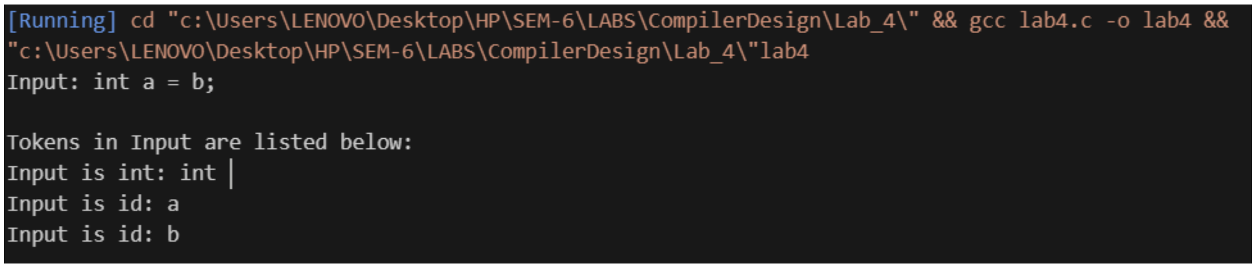
4) //abc

5) void main(){

```
    int a = 10.7e24;
    // This is a comment
    /*
    int b = 11.7;
    */
    char c = 'a';
}
```

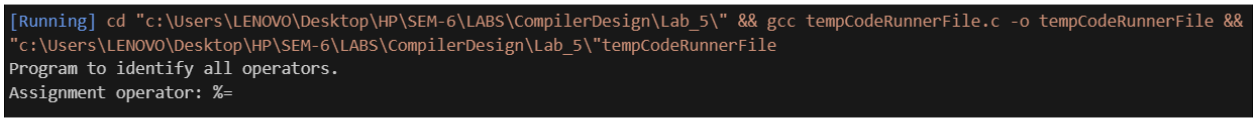
OUTPUT:

1)



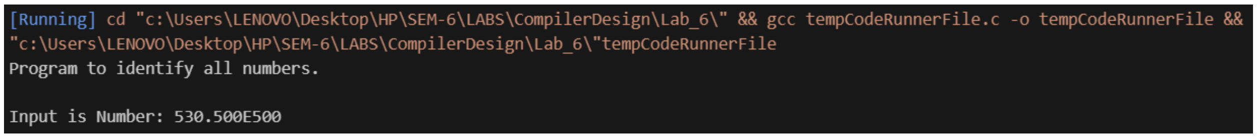
```
[Running] cd "c:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_4\" && gcc lab4.c -o lab4 &&
"c:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_4\"lab4
Input: int a = b;

Tokens in Input are listed below:
Input is int: int
Input is id: a
Input is id: b
```

2)



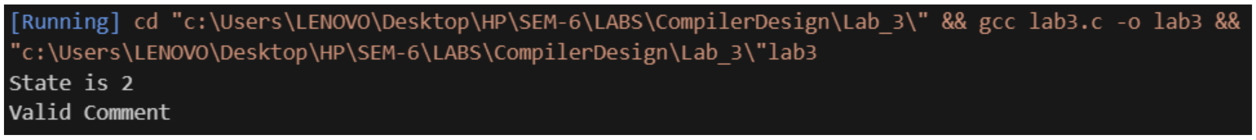
```
[Running] cd "c:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_5\" && gcc tempCodeRunnerFile.c -o tempCodeRunnerFile &&
"c:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_5\"tempCodeRunnerFile
Program to identify all operators.
Assignment operator: %=
```

3)



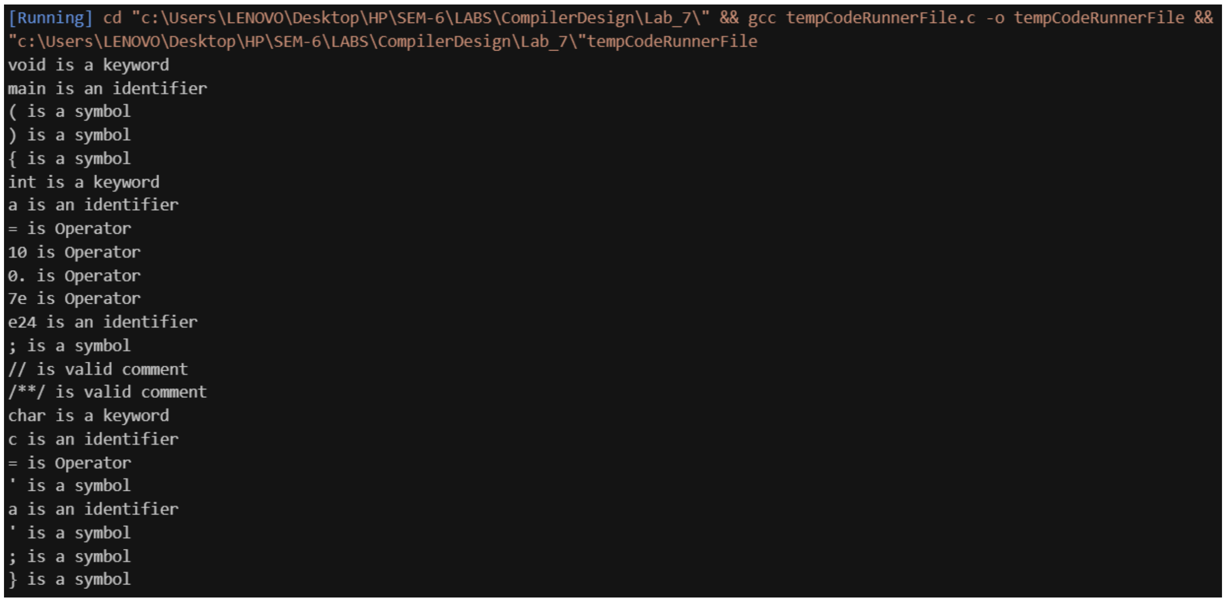
```
[Running] cd "c:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_6\" && gcc tempCodeRunnerFile.c -o tempCodeRunnerFile &&
"c:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_6\"tempCodeRunnerFile
Program to identify all numbers.

Input is Number: 530.500E500
```

4)



```
[Running] cd "c:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_3\" && gcc lab3.c -o lab3 &&
"c:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_3\"lab3
State is 2
Valid Comment
```

5)



```
[Running] cd "c:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_7\" && gcc tempCodeRunnerFile.c -o tempCodeRunnerFile &&
"c:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_7\"tempCodeRunnerFile
void is a keyword
main is an identifier
( is a symbol
) is a symbol
{ is a symbol
int is a keyword
a is an identifier
= is Operator
10 is Operator
0. is Operator
7e is Operator
e24 is an identifier
; is a symbol
// is valid comment
/**/ is valid comment
char is a keyword
c is an identifier
= is Operator
' is a symbol
a is an identifier
' is a symbol
; is a symbol
} is a symbol
```

Practical – 3

AIM: To Study about Lexical Analyzer Generator (LEX) and Flex(Fast Lexical Analyzer)

LEARNING:

What is a Lexical Analyzer?

A Lexical Analyzer (Lexer or Scanner) is the first phase of a compiler. Its main job is to:

- Read the source code character by character.
- Group characters into tokens (keywords, identifiers, operators, etc.).
- Remove whitespaces and comments.
- Send tokens to the parser for further analysis.

What is LEX?

LEX is a Lexical Analyzer Generator developed in the 1970s at Bell Labs. It generates a C program that can perform lexical analysis based on user-defined patterns (typically written using regular expressions).

Key Points:

- You write rules in a .l file.
- LEX converts those rules into a C program (lex.yy.c).
- That C program, when compiled, scans input text and performs actions based on the matched patterns.

What is Flex (Fast Lexical Analyzer)?

Flex is an enhanced version of LEX that is:

- Faster.
- More portable.
- Widely used in modern systems.

It is often used with Bison (a parser generator like Yacc).

Structure of a LEX/Flex Program

A .l file (LEX source) has three sections:

```
%{
   /* C declarations and headers */
%}


%%
   /* Pattern   Action */
   "int"     { printf("Keyword: int\n"); }
   [a-zA-Z]+ { printf("Identifier: %s\n", yytext); }
   [0-9]+    { printf("Number: %s\n", yytext); }
   \n        { /* Ignore newline */ }
%%
   /* Additional C code (main function etc.) */
int main() {
   yylex();  // Start lexical analysis
   return 0;
}
```

Compilation & Execution (Using Flex)

Assuming your file is example.l

- flex example.l   # Generates lex.yy.c
- gcc lex.yy.c -o example # Compiles the C code
- ./example   # Run the lexer

CONCLUSION

LEX and Flex are powerful tools for automating the process of lexical analysis, which is the foundational phase in compiler design. They help transform regular expression-based token definitions into efficient C code capable of recognizing patterns in input text.

- LEX laid the groundwork for lexical analyzer generators by allowing programmers to define patterns and actions easily.
- Flex, its modern alternative, offers better performance, portability, and integration with C-based compiler tools like Bison.

Both tools significantly reduce manual coding effort in lexical analyzers and are widely used in compiler construction, interpreters, and many text-processing applications. Mastery of LEX/Flex provides a strong foundation for building custom language parsers, interpreters, and compilers.

Practical – 4

AIM: Implement following programs using Lex.

1. Write a Lex program to take input from text file and count no of characters, no. of lines & no. of words.
2. Write a Lex program to take input from text file and count number of vowels and consonants.
3. Write a Lex program to print out all numbers from the given file.
4. Write a Lex program which adds line numbers to the given file and display the same into different file.
5. Write a Lex program to printout all markup tags and HTML comments in file.

PROGRAM CODE :

1)

```
%{
#include<stdio.h>
int characters=0;
int words=1;
int lines=1;
%}
%%
[\t ]+ {words++;}
\n  {lines++;words++;}
.  {characters++;}
%%
void main(){
yyin=fopen("lab8_2.txt","r");
yylex();
printf("This file is containing %d characters\n",characters);
printf("This file is containing %d words\n",words);
printf("This file is containing %d lines\n",lines);
}
int yywrap(){return(1);}
```

2)

```
%{
#include<stdio.h>
int vowels = 0;
int consonants = 0;
%}
%%
[aAeEiIoOuU] { vowels++; }
[a-zA-Z] { consonants++; }
.|\n|\t|[ ] { }
%%

void main() {
   yyin = fopen("lab9_1.txt", "r");
   yylex();
   printf("The file contains %d vowels\n", vowels);
   printf("The file contains %d consonants\n", consonants);
   fclose(yyin);
}
int yywrap() { return 1; }
```

3)

```
%{
#include<stdio.h>
%}
%%
[0-9]+(\.[0-9]+)?([Ee][+=]?[0-9]+)? {printf("This is a valid number: %s\n", yytext);}
.|\n|\t|[ ] { }
%%

void main() {
   yyin = fopen("lab9_2.txt", "r");
   yylex();
   fclose(yyin);
}
int yywrap() { return 1; }
```

4)

```
%{
#include<stdio.h>
int line_number =1;
%}
%%
.+ {fprintf(yyout,"%d : %s",line_number,yytext);line_number++;}
%%

void main() {
   yyin = fopen("lab9_3_in.txt", "r");
   yyout=fopen("lab9_3_out.txt","w");
   yylex();
   printf("Done");
   fclose(yyin);
}
```

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

5)

```
%{
#include<stdio.h>
%}
%%
"<!--"(.|\n)*"-->"      { printf("This is a HTML comment: %s\n", yytext); }
"<"[A-Za-z0-9]+">"      { printf("This is a markup opening tag: %s\n", yytext); }
"</"[A-Za-z0-9]+">"    { printf("This is a markup closing tag: %s\n", yytext); }
.|\n|\t|[ ]             { }
%%

int main() {
   yyin = fopen("lab9_4.txt", "r");
   yylex();
   fclose(yyin);
   return 0;
}

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

INPUT:

1)

```
Compiler   Compiler
123 Compiler
```

2)

Compiler Design Lab

3)

Compiler Design Lab 123 Compiler Design Lab 123.123 Compiler Design Lab 123.123e123

4)

Compiler Design Lab

123

Compiler Design Lab

123.123

Compiler Design Lab

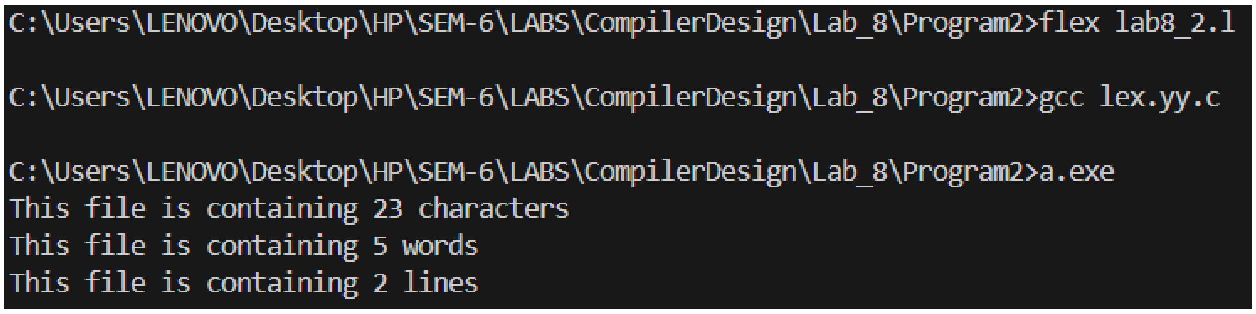
123.123e123

5)

```
<html>
  <body>
    <!-- This is a comment -->
  </body>
</html>
```

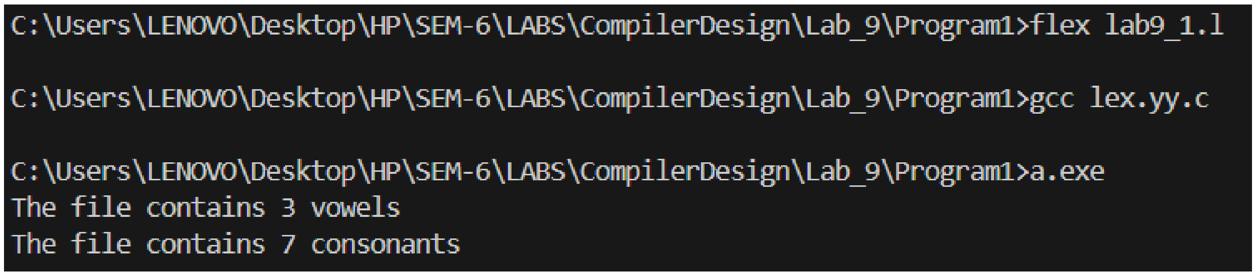
OUTPUT:

1)



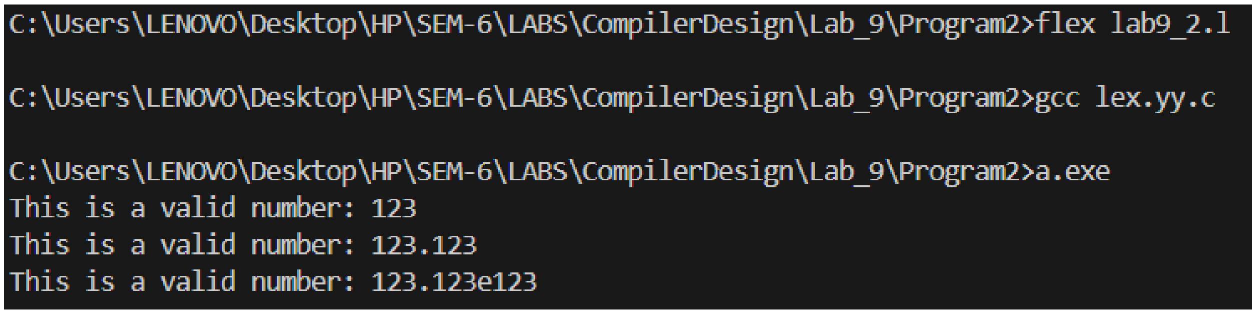
```
C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_8\Program2>flex lab8_2.l

C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_8\Program2>gcc lex.yy.c

C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_8\Program2>a.exe
This file is containing 23 characters
This file is containing 5 words
This file is containing 2 lines
```

2)



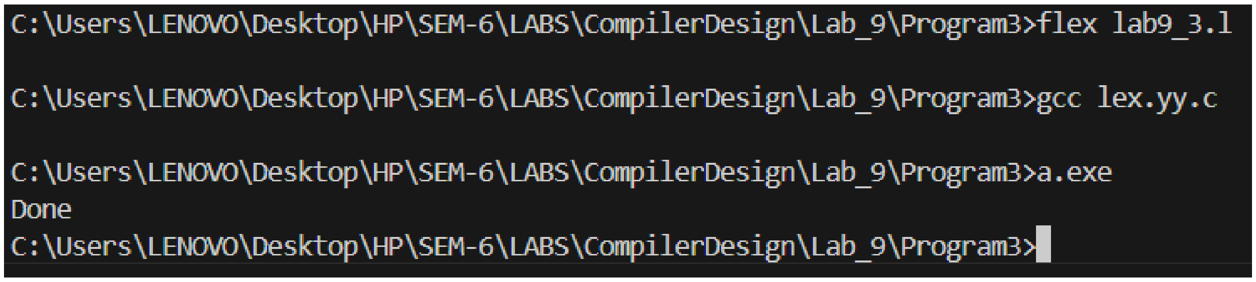
```
C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_9\Program1>flex lab9_1.l

C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_9\Program1>gcc lex.yy.c

C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_9\Program1>a.exe
The file contains 3 vowels
The file contains 7 consonants
```

3)

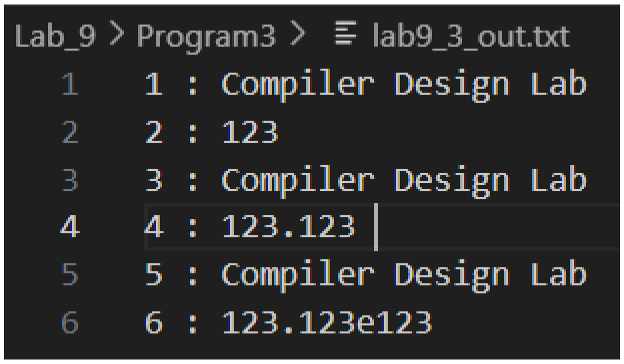


```
C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_9\Program2>flex lab9_2.l

C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_9\Program2>gcc lex.yy.c

C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_9\Program2>a.exe
This is a valid number: 123
This is a valid number: 123.123
This is a valid number: 123.123e123
```

4)



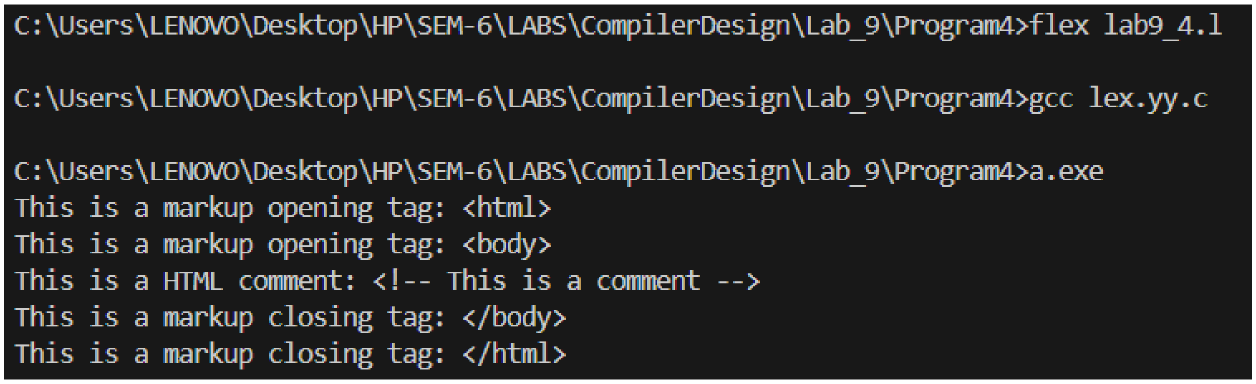
```
C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_9\Program3>flex lab9_3.l

C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_9\Program3>gcc lex.yy.c

C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_9\Program3>a.exe
Done
C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_9\Program3>
```



Lab_9 > Program3 > lab9_3_out.txt

```
1 : Compiler Design Lab
2 : 123
3 : Compiler Design Lab
4 : 123.123
5 : Compiler Design Lab
6 : 123.123e123
```

5)



```
C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_9\Program4>flex lab9_4.l

C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_9\Program4>gcc lex.yy.c

C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_9\Program4>a.exe
This is a markup opening tag: <html>
This is a markup opening tag: <body>
This is a HTML comment: <!-- This is a comment -->
This is a markup closing tag: </body>
This is a markup closing tag: </html>
```

Practical – 5

AIM:

1. Write a Lex program to count the number of C comment lines from a given C program. Also eliminate them and copy that program into separate file.
2. Write a Lex program to recognize keywords, identifiers, operators, numbers, special symbols, literals from a given C program.

PROGRAM CODE :

1)

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

%%
"/*"(.|\n)*"*/" {fprintf(yyout, " ");comment_count++;}
"//".*          {fprintf(yyout, " ");comment_count++;}
.             {fprintf(yyout,"%s",yytext);}
%%
void main() {
   yyin = fopen("input.txt", "r");
   yyout = fopen("output.txt", "w");
   yylex();

   printf("Total comment lines found: %d\n", comment_count);

   fclose(yyin);
   fclose(yyout);
}
int yywrap() { return 1; }
```

2)

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

int keyword_count = 0;
int identifier_count = 0;
int operator_count = 0;
int number_count = 0;
int symbol_count = 0;
int literal_count = 0;

char *keywords[] = {
    "auto", "break", "case", "char", "const", "continue", "default", "do",
    "double", "else", "enum", "extern", "float", "for", "goto", "if", "inline",
    "int", "long", "register", "return", "short", "signed", "sizeof", "static",
    "struct", "switch", "typedef", "union", "unsigned", "void", "volatile", "while"
};

int is_keyword(char *str) {
    for (int i = 0; i < 32; i++) {
        if (strcmp(str, keywords[i]) == 0)
            return 1;  // It's a keyword
    }
    return 0;
}
%}

%%
int|float|if|else|while  { printf("Keyword: %s\n", yytext); keyword_count++; }

[A-Za-z_][A-Za-z0-9_]*  {
    if (is_keyword(yytext)) {
        printf("Keyword: %s\n", yytext);
        keyword_count++;
    } else {
        printf("Identifier: %s\n", yytext);
        identifier_count++;
    }
}

"+"|"-"|"*"|"/"|"%"|"="|"<"|"|"|"!"|"&"|"^"|"~"|"?"|":"|"."  { printf("Operator: %s\n",
yytext); operator_count++; }

[0-9]+                  { printf("Number: %s\n", yytext); number_count++; }

\"[^\"]*\"              { printf("String Literal: %s\n", yytext); literal_count++; }

\'[^\'\n]\'             { printf("Character Literal: %s\n", yytext); literal_count++; }

[ \t\n]  /* Ignore whitespace */

[;{}()[]<>.,]  { printf("Special Symbol: %s\n", yytext); symbol_count++; }
```

```
%%

int main() {
    FILE *in = fopen("input.txt", "r");  // Open the input file (input.txt)
    if (!in) {
        perror("Error opening input file");
        return 1;
    }

    yyin = in;  // Assign the input file to yyin
    yylex();  // Start lexical analysis

    printf("\nTotal Keywords: %d\n", keyword_count);
    printf("Total Identifiers: %d\n", identifier_count);
    printf("Total Operators: %d\n", operator_count);
    printf("Total Numbers: %d\n", number_count);
    printf("Total Special Symbols: %d\n", symbol_count);
    printf("Total Literals: %d\n", literal_count);

    fclose(in);  // Close the input file
    return 0;
}
int yywrap() { return 1; }
```

INPUT:

1)

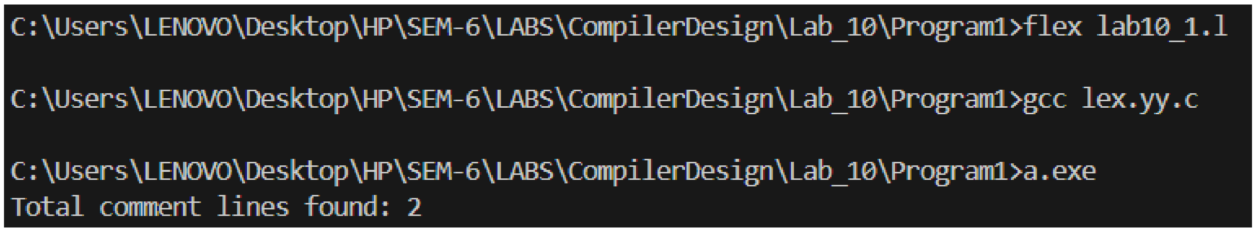
```
int main() {
    // This is a single-line comment
    printf("Hello world");
    harsh /* multi-line
    comment here
    spans 3 lines */
    return 0;
}
```

2)

```
#include <stdio.h>

int main() {
    int x = 10;
    float y = 20.5;
    printf("Hello, World!");
    return 0;
}
```

OUTPUT:

1)



```
C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_10\Program1>flex lab10_1.l

C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_10\Program1>gcc lex.yy.c

C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_10\Program1>a.exe
Total comment lines found: 2
```

2)

```
C:\Users\LENOVO\Desktop\HP\SEM-6\LABS\CompilerDesign\Lab_10\Program2>a.ex
e
#Identifier: include
```

```
Operator: <
Identifier: stdio
Operator: .
Identifier: h
>Keyword: int
Identifier: main
(){Keyword: int
Identifier: x
Operator: =
Number: 10
;Keyword: float
Identifier: y
Operator: =
Number: 20
Operator: .
Number: 5
;Identifier: printf
(String Literal: "Hello, World!"
);Keyword: return
(String Literal: "Hello, World!"
);Keyword: return
Number: 0
(String Literal: "Hello, World!"
);Keyword: return
(String Literal: "Hello, World!"
);Keyword: return
(String Literal: "Hello, World!"
);Keyword: return
(String Literal: "Hello, World!"
);Keyword: return
(String Literal: "Hello, World!"
);Keyword: return
Number: 0
(String Literal: "Hello, World!"
);Keyword: return
Number: 0
(String Literal: "Hello, World!"
);Keyword: return
(String Literal: "Hello, World!"
);Keyword: return
);Keyword: return
Number: 0
;}
Total Keywords: 4
Total Identifiers: 7
Total Operators: 5
Total Numbers: 4
Total Special Symbols: 0
Total Literals: 1
```

Practical – 6

AIM: Program to implement Recursive Descent Parsing in C.

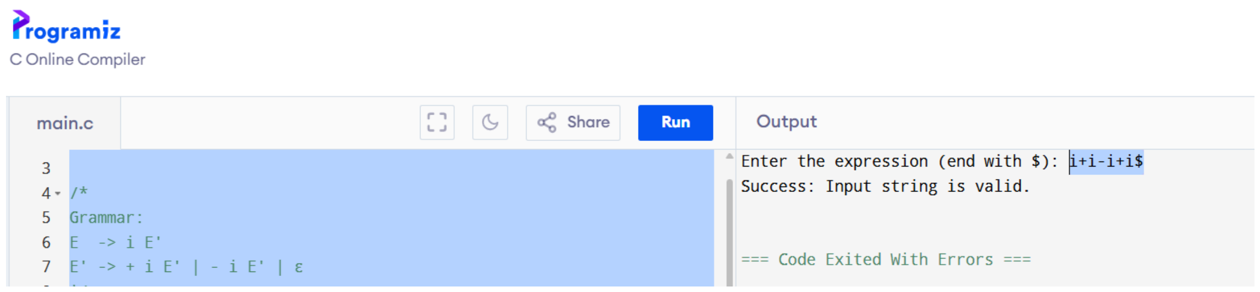
PROGRAM CODE :

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

/*
Grammar:
E  -> i E'
E' -> + i E' | - i E' | ε
*/

char s[20];
int i = 0;
char l;

void match(char t);
void E();
void E_();

void main() {
   printf("Enter the expression (end with $): ");
   scanf("%s", s);
   l = s[i];
   E();

   if (l == '$') {
       printf("Success: Input string is valid.\n");
   } else {
       printf("Syntax error: Unexpected character '%c'\n", l);
   }

}

void match(char t) {
   if (l == t) {
       i++;
       l = s[i];
   } else {
       printf("Syntax error: Expected '%c' but found '%c'\n", t, l);
       exit(1);
   }
}

void E() {
   if (l == 'i') {
       match('i');
       E_();
   } else {
       printf("Syntax error in E: Expected 'i'\n");
       exit(1);
   }
}

void E_() {
   if (l == '+') {
       match('+');
       match('i');
       E_();
   } else if (l == '-') {
       match('-');
       match('i');
       E_();
   }
}
```

INPUT:

i+i-i+i$

OUTPUT:



```
/*
Grammar:
E  -> i E'
E' -> + i E' | - i E' | ε
```

```
Enter the expression (end with $): i+i-i+i$
Success: Input string is valid.


=== Code Exited With Errors ===
```

Practical – 7

AIM:

1. To Study about Yet Another Compiler-Compiler(YACC).
2. Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, * and / .
3. Create Yacc and Lex specification files are used to generate a calculator which accepts integer type arguments.
4. Create Yacc and Lex specification files are used to convert infix expression to postfix expression.

LEARNING:

What is YACC?

YACC (Yet Another Compiler-Compiler) is a tool used to generate parsers for context-free grammars, typically used in compiler construction. It works in combination with Lex, which performs lexical analysis (tokenization).

- Lex: Generates a scanner (tokenizer).
- YACC: Generates a parser that interprets the token stream according to grammar rules.

Purpose of YACC

YACC simplifies the creation of a syntax analyzer (parser) by:

- Allowing you to define grammar rules in a readable format.
- Automatically generating C code for the parser.
- Managing syntax errors and precedence rules.

How YACC Works

- Input: YACC takes grammar rules written in a .y file.
- Output: Generates a C file y.tab.c containing a parser.
- Integration: This is compiled with:
  - The output from Lex (lex.yy.c)
  - Any supporting C code (e.g., for semantic actions)

Structure of a YACC Program

A YACC program has three sections:

```
%{
/* C Declarations */
#include <stdio.h>
int yylex();
void yyerror(const char *);
%}

%token IDENTIFIER NUMBER PLUS MINUS MUL DIV

%%
/* Grammar Rules and Semantic Actions */
expression: expression PLUS term    { printf("PLUS\n"); }
        | expression MINUS term   { printf("MINUS\n"); }
        | term;


term: term MUL factor              { printf("MUL\n"); }
    | term DIV factor              { printf("DIV\n"); }
    | factor;

factor: NUMBER
```

```
      | IDENTIFIER;

%%

/* Auxiliary C Code */
int main() {
   return yyparse();
}

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

Key Components

%token: Declares tokens coming from Lex.

Grammar rules: Define the syntax structure.

Actions ({}): C code that runs when the rule matches.

yyparse(): The parsing function generated by YACC.

yyerror(): Called when a syntax error occurs.

Integration with Lex

Lex handles input and identifies tokens, passing them to YACC:

Lex file (lex.l):

```
%{
#include "y.tab.h"
%}

%%
[0-9]+      { yylval = atoi(yytext); return NUMBER; }
```

```
[a-zA-Z_]+  { return IDENTIFIER; }
"+"         { return PLUS; }
"-"         { return MINUS; }
"*"         { return MUL; }
"/"         { return DIV; }
[ \t\n]     ;
.           { return yytext[0]; }
%%
```

How to Compile and Run

- lex lex.l
- yacc -d yacc.y
- gcc lex.yy.c y.tab.c -o parser
- ./parser

CONCLUSION

YACC is a powerful tool for automating the creation of parsers from formal grammar definitions. Combined with Lex, it allows for quick construction of interpreters and compilers by handling tokenization and syntax parsing

PROGRAM CODE :

2)

arith.l:

```
%{
#include "arith.tab.h"
#include <stdlib.h>
%}
```

```
%%

[0-9]+          { yylval = atoi(yytext); return NUM; }
[+\-*/\n()]     { return *yytext; }
[ \t]           { /* skip whitespace */ }
.               { return yytext[0]; }

%%

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

arith.y

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

void yyerror(const char *s);
int yylex(void);
%}

%token NUM

%left '+' '-'
%left '*' '/'
%left UMINUS

%%
```

```
input:
    /* empty */
  | input expr '\n'  { printf("Valid expression\n"); }
  ;

expr:
    expr '+' expr
  | expr '-' expr
  | expr '*' expr
  | expr '/' expr
  | '-' expr %prec UMINUS
  | '(' expr ')'
  | NUM
  ;

%%

void yyerror(const char *s) {
    fprintf(stderr, "Invalid expression\n");
}

int main() {
    return yyparse();
}
```

3)

lex_2.l

```
%{
```

```
#include <stdlib.h>
void yyerror(char *);
#include "bison_2.tab.h"
%}
%%
[0-9]+ {yylval = atoi(yytext); return NUM;}
[-+*\n] {return *yytext;}
[ \t] { }
. yyerror("invalid character");
%%
int yywrap() {
 return 0;
}
```

bison_2.y

```
%{
 #include <stdio.h>
 int yylex(void);
 void yyerror(char *);
%}
%token NUM
%%
S: E '\n' { printf("%d\n", $1); return(0); }
E: E '+' T   { $$ = $1 + $3; }
 | E '-' T  { $$ = $1 - $3; }
 | T       { $$ = $1; }
T : T '*' F { $$ = $1 * $3; }
 | F       { $$ = $1; }
F:NUM   { $$ = $1; }
```

```
%%
void yyerror(char *s) {
 fprintf(stderr, "%s\n", s);
}
int main() {
 yyparse();
 return 0;
}
```

4)

lex.l

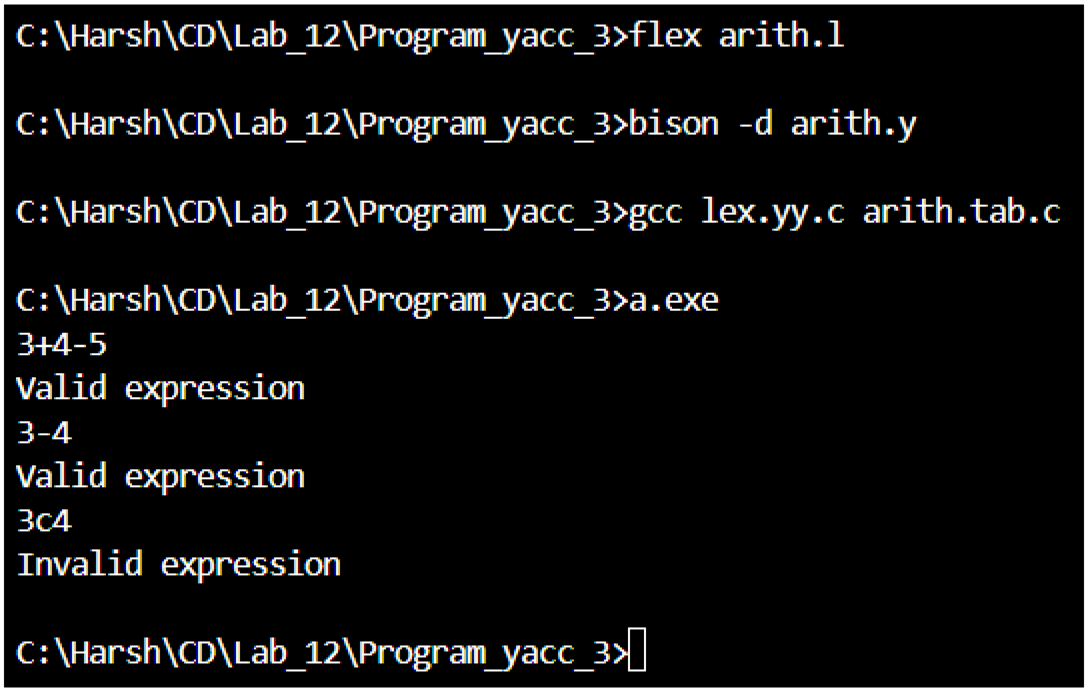
```
%{
#include <stdlib.h>
void yyerror(char *);
#include "bison.tab.h"
%}
%%
[0-9]+ {yylval.num = atoi(yytext); return INTEGER;}
[a-zA-Z_][a-zA-Z0-9_]* {yylval.str = yytext; return ID;}
[-+*\n] {return *yytext;}
[ \t] { }
. yyerror("invalid character");
%%
int yywrap() {
 return 0;
}
```

bison.y

```
%{
 #include <stdio.h>
 int yylex(void);
 void yyerror(char *);
%}
%union{
   char *str;
   int num;
}
%token <num> INTEGER
%token <str> ID
%%
S: E '\n' { printf("\n"); }
E: E '+' T   { printf("+"); }
 | E '-' T  { printf("-"); }
 | T        { }
T : T '*' F { printf("*"); }
 | F        { }
F: INTEGER  { printf("%d",$1); }
 | ID       { { printf("%s",$1); }}
%%
void yyerror(char *s) {
 fprintf(stderr, "%s\n", s);
}
int main() {
 yyparse();
 return 0;
}
```

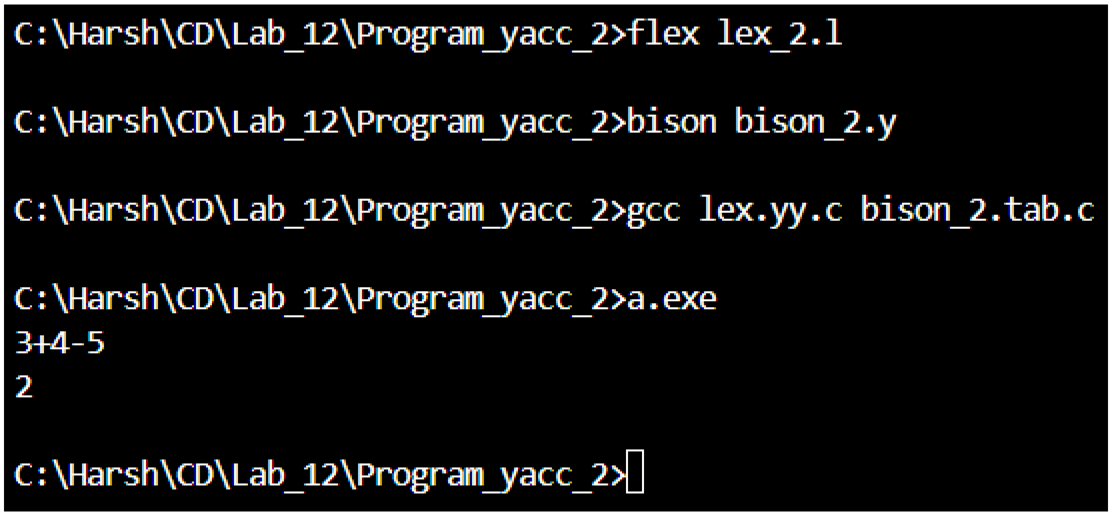
OUTPUT:

2)



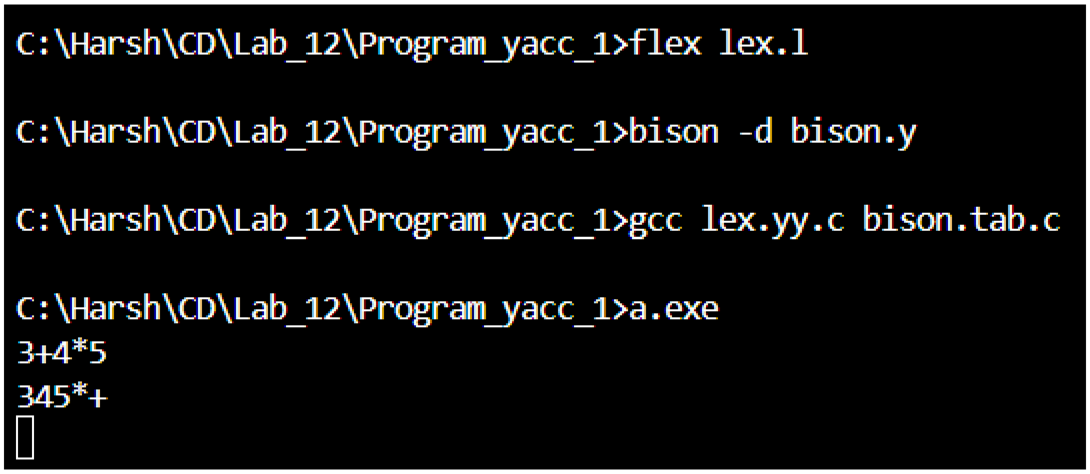
```
C:\Harsh\CD\Lab_12\Program_yacc_3>flex arith.l

C:\Harsh\CD\Lab_12\Program_yacc_3>bison -d arith.y

C:\Harsh\CD\Lab_12\Program_yacc_3>gcc lex.yy.c arith.tab.c

C:\Harsh\CD\Lab_12\Program_yacc_3>a.exe
3+4-5
Valid expression
3-4
Valid expression
3c4
Invalid expression

C:\Harsh\CD\Lab_12\Program_yacc_3>
```

3)



```
C:\Harsh\CD\Lab_12\Program_yacc_2>flex lex_2.l

C:\Harsh\CD\Lab_12\Program_yacc_2>bison bison_2.y

C:\Harsh\CD\Lab_12\Program_yacc_2>gcc lex.yy.c bison_2.tab.c

C:\Harsh\CD\Lab_12\Program_yacc_2>a.exe
3+4-5
2

C:\Harsh\CD\Lab_12\Program_yacc_2>
```

4)



```
C:\Harsh\CD\Lab_12\Program_yacc_1>flex lex.l

C:\Harsh\CD\Lab_12\Program_yacc_1>bison -d bison.y

C:\Harsh\CD\Lab_12\Program_yacc_1>gcc lex.yy.c bison.tab.c

C:\Harsh\CD\Lab_12\Program_yacc_1>a.exe
3+4*5
345*+
```

CONCLUSION

In these practicals, we explored the fundamentals of lexical analysis and parsing using Lex (Flex) and YACC (Yet Another Compiler Compiler). Through the tasks, we learned how to recognize and process different types of strings, such as those starting or ending with specific characters, containing substrings, or matching patterns like valid identifiers, operators, and numbers. We also implemented programs to analyze and manipulate text files, count characters, lines, words, vowels, consonants, and recognize markup tags. Additionally, we worked with C language features like comments, keywords, and operators, honing our skills in handling the intricacies of language syntax. By implementing recursive descent parsing and converting infix expressions to postfix, we gained insight into building simple compilers and calculators. Overall, these practicals provided hands-on experience with lexical analysis, parsing techniques, and tools like Lex and YACC, which are essential for building parsers and compilers in various programming languages.