Compiler Design Lab Report

AP20110010290

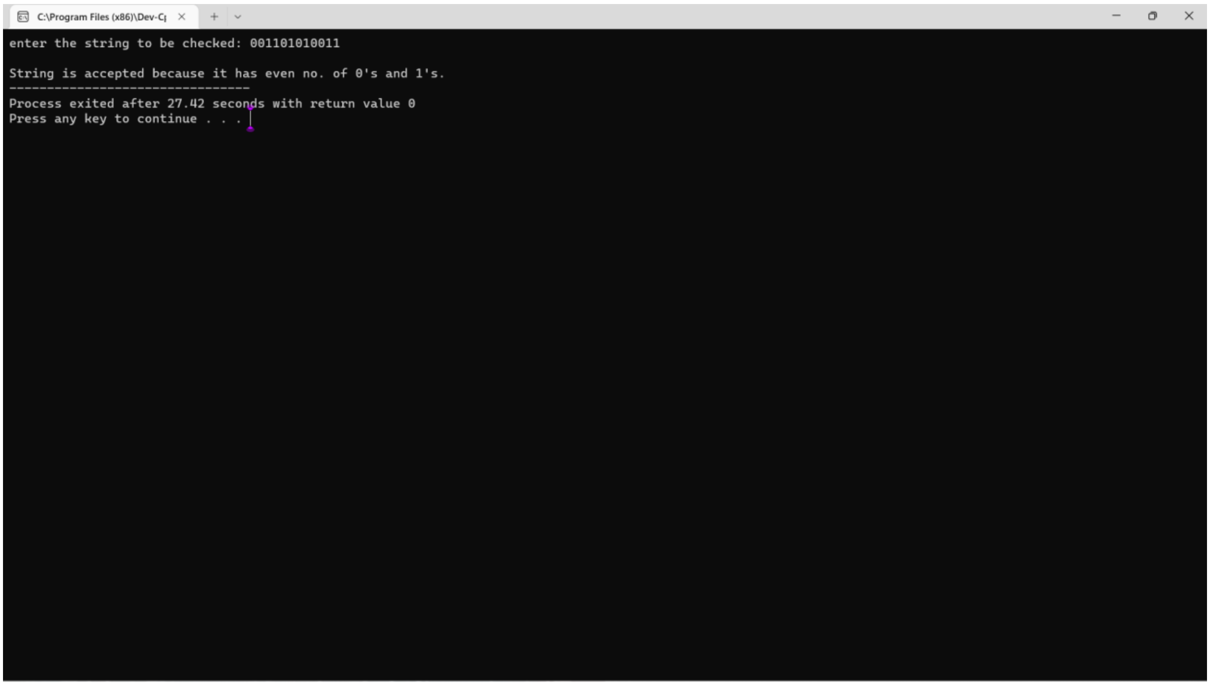
G. Vinay Babu

1. Write a program in C that recognizes the following languages.

a. Set of all strings over binary alphabet containing even number of 0's and even number of 1's.

```
#include<stdio.h>
#define max 100
int main() {
char str[max],f='a';
int i;
printf("enter the string to be checked: ");
scanf("%s",str);
for(i=0;str[i]!='\0';i++) {

switch(f) {
case 'a': if(str[i]=='0') f='b';
else if(str[i]=='1') f='d';
break;
case 'b': if(str[i]=='0') f='a';
else if(str[i]=='1') f='c';
break;
```

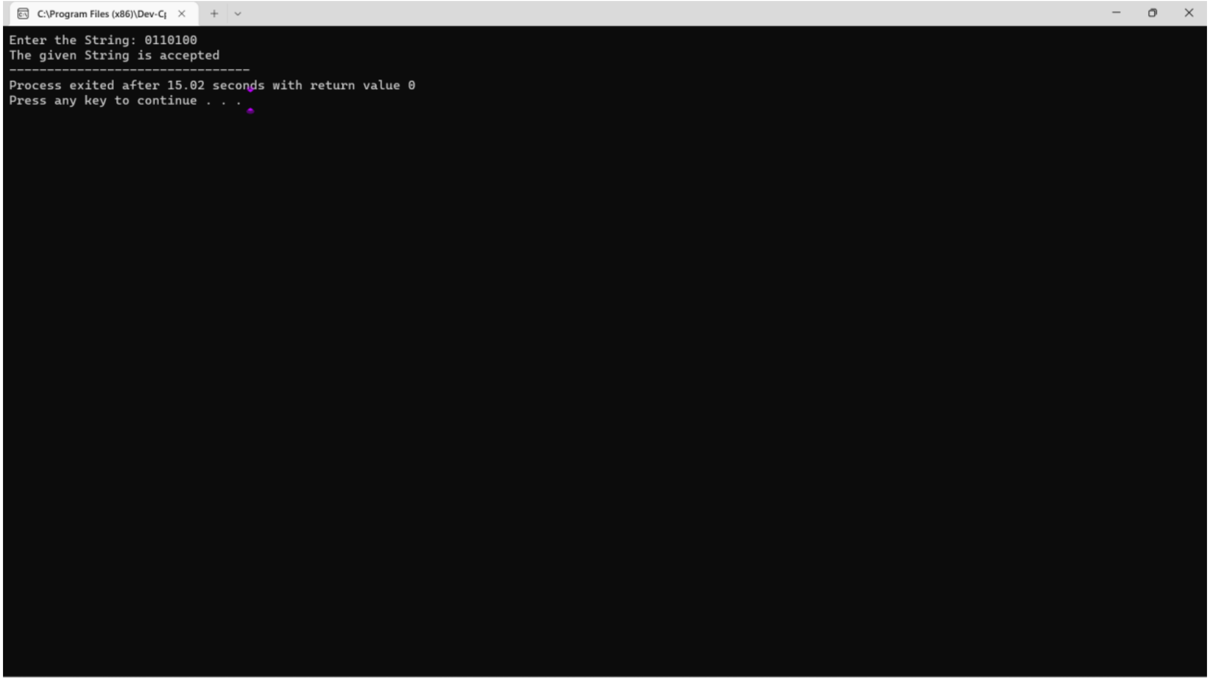
```
case 'c': if(str[i]=='0') f='d';
else if(str[i]=='1') f='b';
break;
case 'd': if(str[i]=='0') f='c';
else if(str[i]=='1') f='a';
break;

}
}
if(f=='a')
printf("\nString is accepted because it has even no. of 0's and 1's.");
else printf("\nString is not accepted");
return 0;
}
```



b. Lab Assignment: Set of all strings ending with two symbols of same type.

```
#include<stdio.h>

int main()
{
char state='a';
int length, i;
char n[20];
printf("Enter the String: ");
scanf("%s",n);
for(i=0;n[i]!='\0';i++)
{
switch(state)
{
case 'a':
if(n[i]=='0'||n[i]=='1')
state='a';
else if(n[i]=='.')
state='b';
else
state='d';
break;
case 'b':
if(n[i]=='0'|n[i]=='1')
state='c';
```

```
else
state='d';
break;
case 'c':
if(n[i]=='0'||n[i]=='1')
state='c';
else
state='d';
break;
case 'd':
state='d';
break; }
}
if(state=='c'||state=='a')
printf("The given String is accepted");
else
printf("The given String is not accepted");
return 0;
}
```



2. Implement lexical analyzer using C for recognizing the following tokens:

- A minimum of 10 keywords of your choice

```c
#include <stdbool.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>


// Returns 'true' if the character is a DELIMITER.

bool isDelimiter(char ch)

{

    if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||

        ch == '/' || ch == ',' || ch == ';' || ch == '>' ||

        ch == '<' || ch == '=' || ch == '(' || ch == ')' ||

        ch == '[' || ch == ']' || ch == '{' || ch == '}')

        return (true);

    return (false);
```

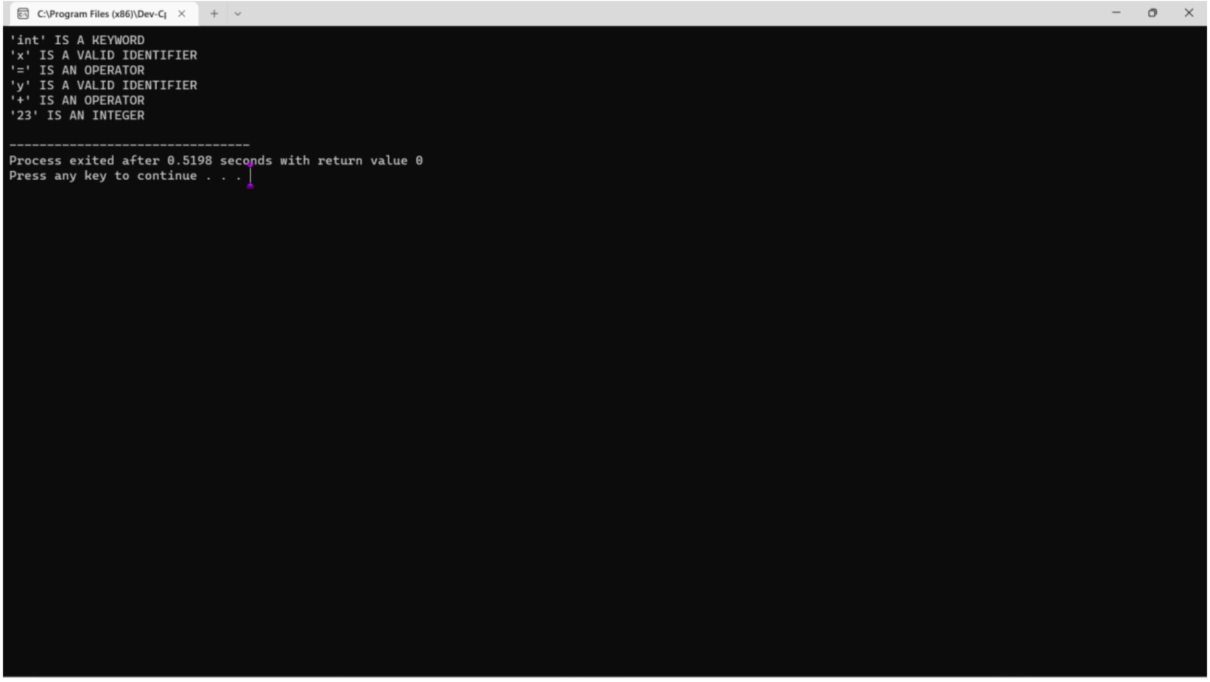
```
}

// Returns 'true' if the character is an OPERATOR.
bool isOperator(char ch)
{
   if (ch == '+' || ch == '-' || ch == '*' ||
      ch == '/' || ch == '>' || ch == '<' ||
      ch == '=')
      return (true);
   return (false);
}

// Returns 'true' if the string is a VALID IDENTIFIER.
bool validIdentifier(char* str)
{
   if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
      str[0] == '3' || str[0] == '4' || str[0] == '5' ||
      str[0] == '6' || str[0] == '7' || str[0] == '8' ||
      str[0] == '9' || isDelimiter(str[0]) == true)
      return (false);
   return (true);
}

// Returns 'true' if the string is a KEYWORD.
bool isKeyword(char* str)
{
   if (!strcmp(str, "if") || !strcmp(str, "else") ||
      !strcmp(str, "while") || !strcmp(str, "do") ||
      !strcmp(str, "break") ||
```

```
        !strcmp(str, "continue") || !strcmp(str, "int")
        || !strcmp(str, "double") || !strcmp(str, "float")
        || !strcmp(str, "return") || !strcmp(str, "char")
        || !strcmp(str, "case") || !strcmp(str, "char")
        || !strcmp(str, "sizeof") || !strcmp(str, "long")
        || !strcmp(str, "short") || !strcmp(str, "typedef")
        || !strcmp(str, "switch") || !strcmp(str, "unsigned")
        || !strcmp(str, "void") || !strcmp(str, "static")
        || !strcmp(str, "struct") || !strcmp(str, "goto"))
        return (true);
    return (false);
}

// Returns 'true' if the string is an INTEGER.
bool isInteger(char* str)
{
    int i, len = strlen(str);

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++) {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
            && str[i] != '3' && str[i] != '4' && str[i] != '5'
            && str[i] != '6' && str[i] != '7' && str[i] != '8'
            && str[i] != '9' || (str[i] == '-' && i > 0))
            return (false);
    }
    return (true);
}
```

```
// Returns 'true' if the string is a REAL NUMBER.
bool isRealNumber(char* str)
{
    int i, len = strlen(str);
    bool hasDecimal = false;

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++) {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
            && str[i] != '3' && str[i] != '4' && str[i] != '5'
            && str[i] != '6' && str[i] != '7' && str[i] != '8'
            && str[i] != '9' && str[i] != '.' ||
            (str[i] == '-' && i > 0))
            return (false);
        if (str[i] == '.')
            hasDecimal = true;
    }
    return (hasDecimal);
}

// Extracts the SUBSTRING.
char* subString(char* str, int left, int right)
{
    int i;
    char* subStr = (char*)malloc(
                sizeof(char) * (right - left + 2));
```

```
    for (i = left; i <= right; i++)
        subStr[i - left] = str[i];
    subStr[right - left + 1] = '\0';
    return (subStr);
}

// Parsing the input STRING.
void parse(char* str)
{
    int left = 0, right = 0;
    int len = strlen(str);

    while (right <= len && left <= right) {
        if (isDelimiter(str[right]) == false)
            right++;

        if (isDelimiter(str[right]) == true && left == right) {
            if (isOperator(str[right]) == true)
                printf("'%c' IS AN OPERATOR\n", str[right]);

            right++;
            left = right;
        } else if (isDelimiter(str[right]) == true && left != right
                   || (right == len && left != right)) {
            char* subStr = subString(str, left, right - 1);

            if (isKeyword(subStr) == true)
                printf("'%s' IS A KEYWORD\n", subStr);
```

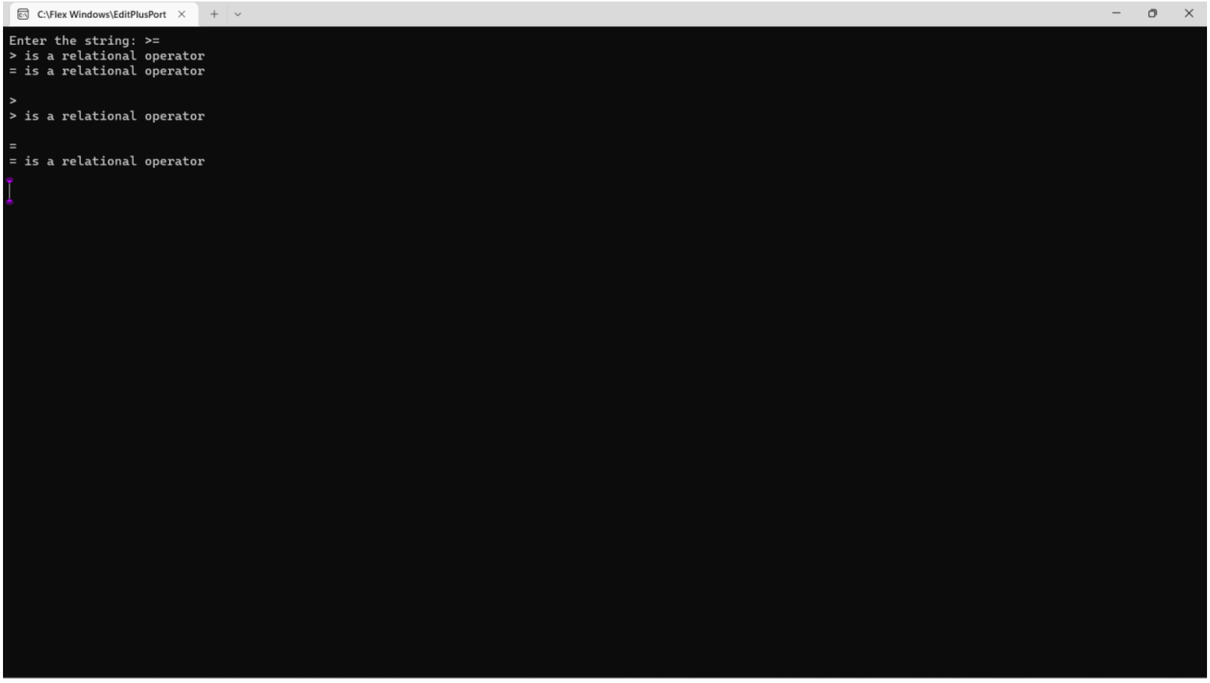
```
        else if (isInteger(subStr) == true)
            printf("'%s' IS AN INTEGER\n", subStr);

        else if (isRealNumber(subStr) == true)
            printf("'%s' IS A REAL NUMBER\n", subStr);

        else if (validIdentifier(subStr) == true
                && isDelimiter(str[right - 1]) == false)
            printf("'%s' IS A VALID IDENTIFIER\n", subStr);

        else if (validIdentifier(subStr) == false
                && isDelimiter(str[right - 1]) == false)
            printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);
        left = right;
      }
    }
    return;
}

// DRIVER FUNCTION
int main()
{
     // maximum length of string is 100 here
    char str[100] = "int a = b + 21821211; ";

    parse(str); // calling the parse function

    return (0);
}
```



- Identifiers with the regular expression : letter(letter | digit)*
- Integers with the regular expression: digit+
- Relational operators: <, >, <=, >=, ==, !=

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


%%


["<" | "<=" | ">" | ">=" | "==" | "!="] {printf("%s is a relational operator\n",yytext);}
%%


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

```
int main()
{
	printf("Enter the string: ");
	yylex();
	return 0;
}
```

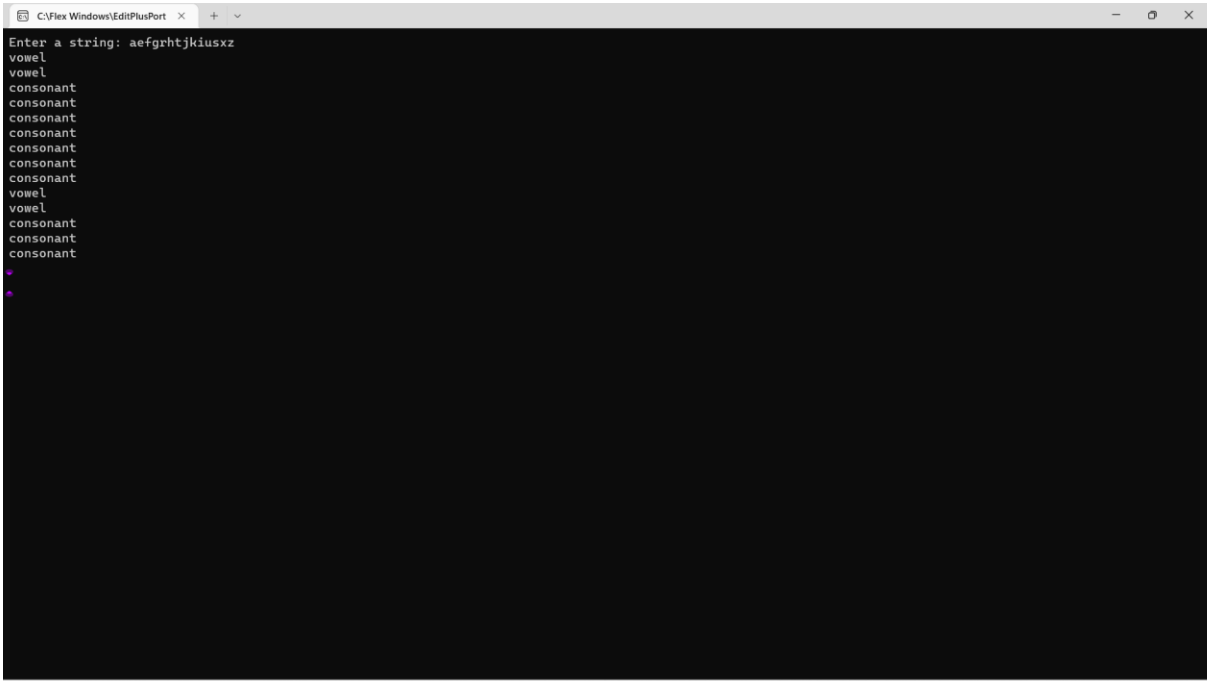


3. Implement the following programs using Lex tool

a. Identification of Vowels and Consonants

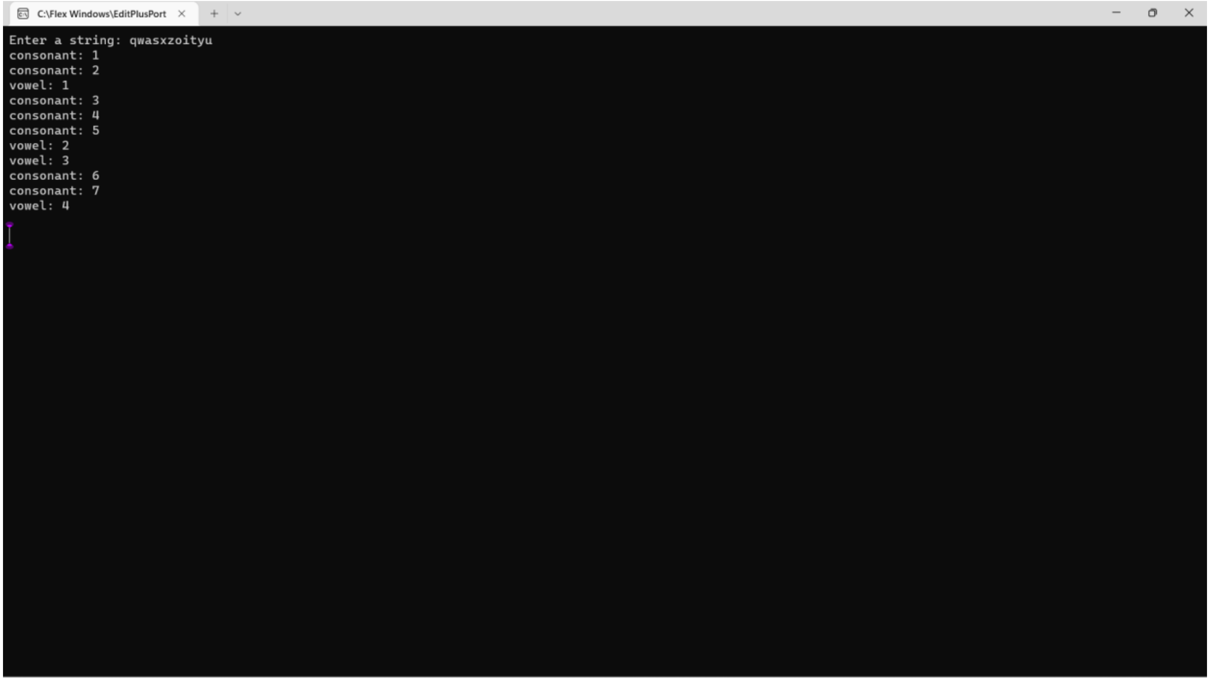
```
%option noyywrap
%{
	#include<stdio.h>

%}
```

```
%%

[aeiouAEIOU] {printf("vowel\n");}

[bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ]
{printf("consonant\n");}


%%

int main()

{

    printf("Enter a string: ");

   yylex();

   return 0;

}
```



b. count number of vowels and consonants

```
%option noyywrap
%{
	#include<stdio.h>
	int vc=0,cc=0;


%}


%%
[aeiouAEIOU] {(vc++);printf("vowel: %d\n",vc);}
[bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ]
{(cc++);printf("consonant: %d\n",cc);}


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

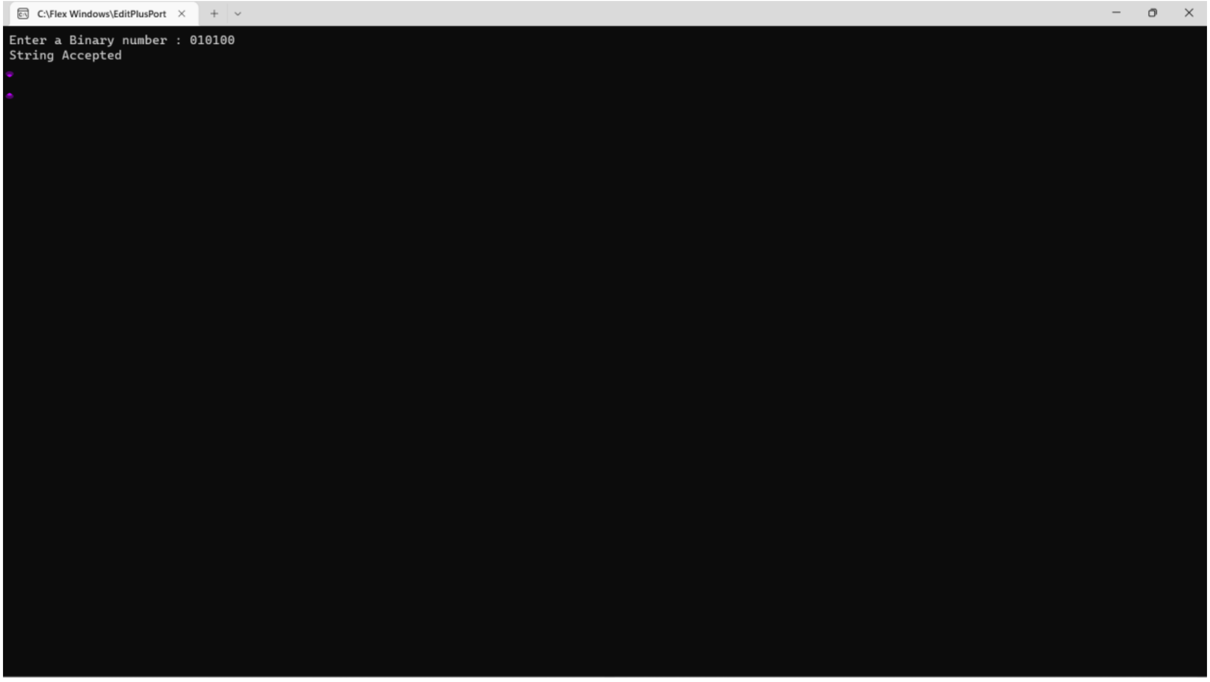


c. Count the number of Lines in given input

```
%option noyywrap
%{
 int num_lines=0;
%}
%%
\n      ++num_lines;
. {}
%%
main()
{
 yylex();
 printf("no. of lines=%d",num_lines);
}
```

d. Recognize strings ending with 00

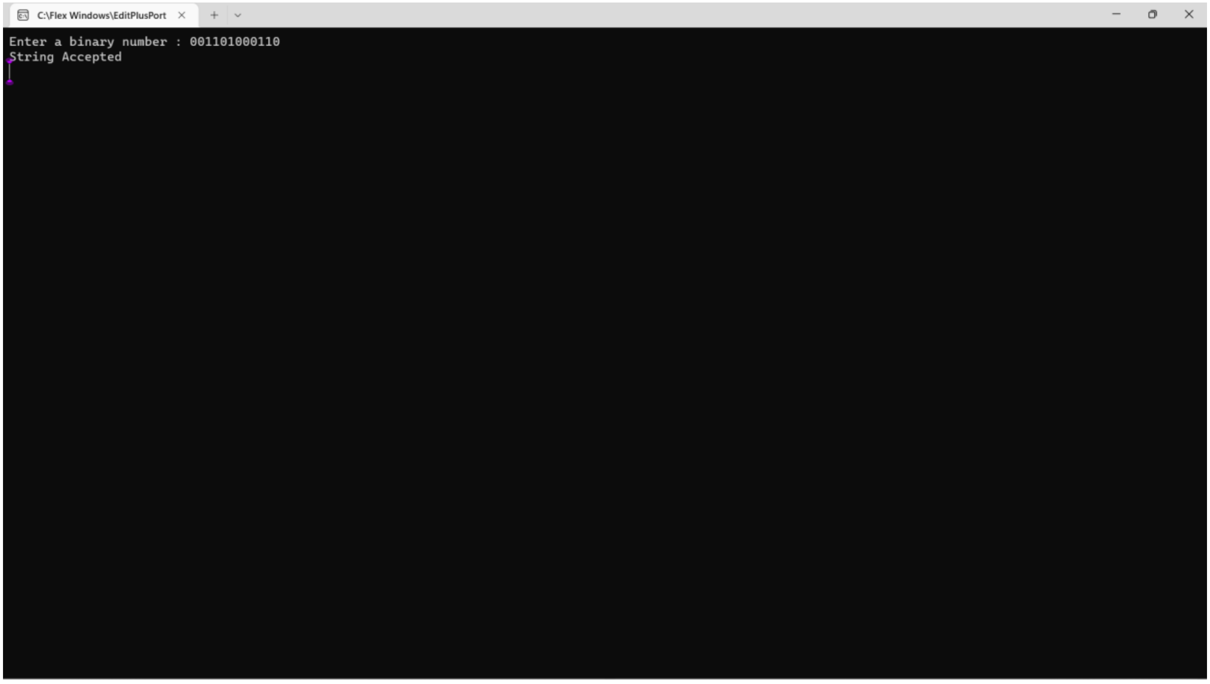
```
%{
#include<stdio.h>
%}

%%

[01]*00$ {printf("String Accepted\n");}
[01]*    {printf("String Rejected\n");}

%%

int yywrap(){
	return 0;
}

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



e. Recognize a string with three consecutive 0’s

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

%%
[01]*000[01]* {printf("String Accepted");}
[01]*  {printf("String Rejected");}
%%

int yywrap(){
	return 0;
}

int main()
```

```
{
        printf("Enter a binary number : ");
        yylex();
        return 0;
}
```

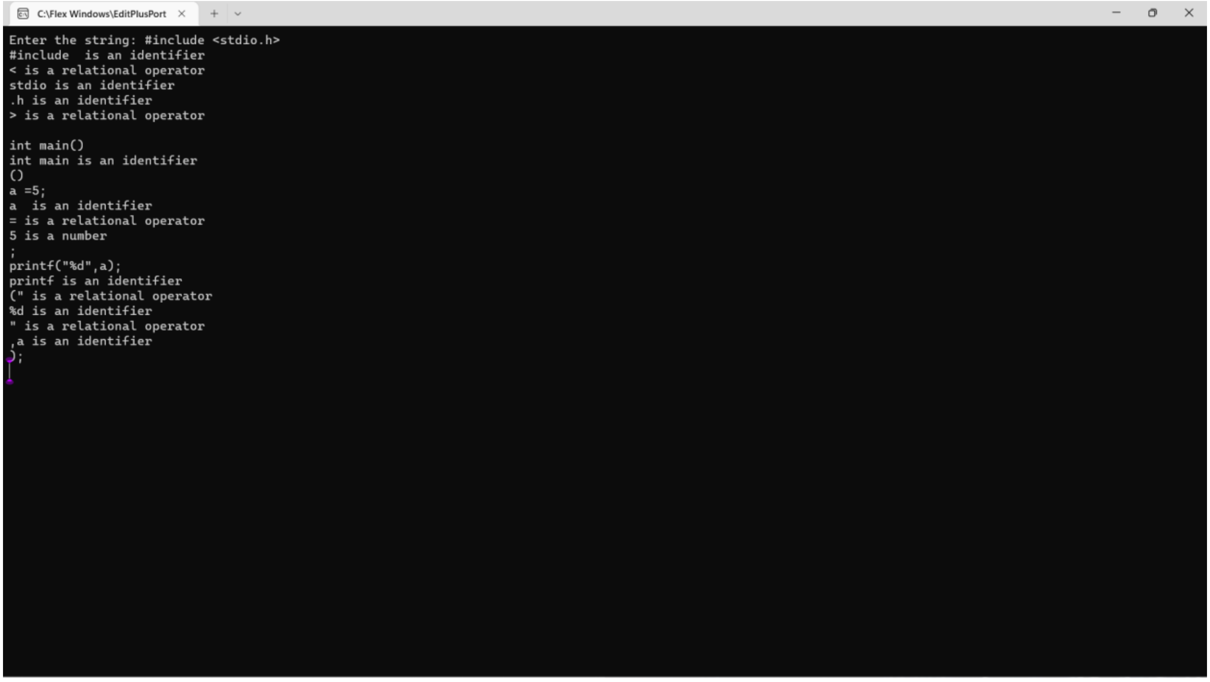


4. Implement lexical analyzer using LEX for recognizing the following tokens:

 A minimum of 10 keywords of your choice

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


%%
auto|double|int|struct|break|else|long|switch|case|enum|register|typedef|char|extern
|return|union|continue|for|signed|void|do|if|static|while|default|goto|sizeof|volatile|c
onst|float|short {printf("%s is a Keyword",yytext);}
```

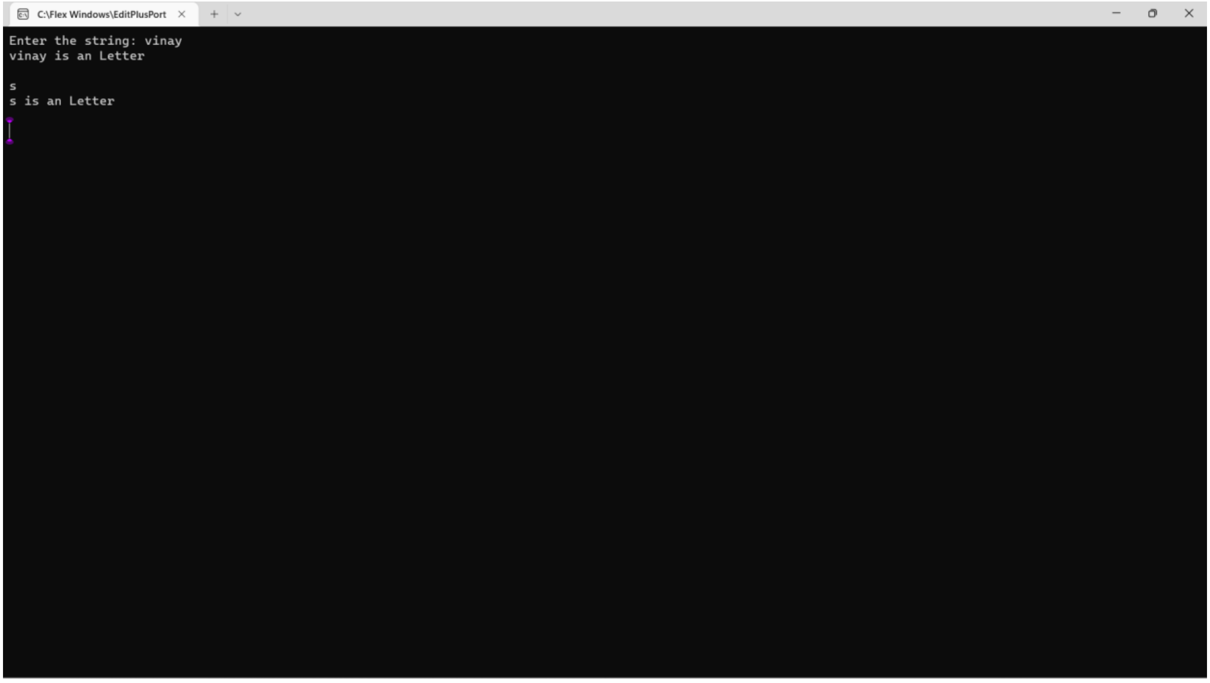
```
[a-zA-Z][a-z A-Z 0-9]* {printf("%s is an identifier\n",yytext);}

[0-9]+ {printf("%s is a number\n",yytext);}

["<" | "<=" | ">" | ">=" | "==" | "!="] {printf("%s is a relational operator\n",yytext);}
%%

int yywrap()
{
	return 1;
}

int main()
{
	printf("Enter the string: ");
	yylex();
	return 0;
}
```



- Identifiers with the regular expression : letter(letter | digit)*

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


%%
[a-zA-Z][a-z A-Z 0-9]* {printf("%s is an Letter\n",yytext);}


%%


int yywrap()
{
	return 1;
}


int main()
{
```

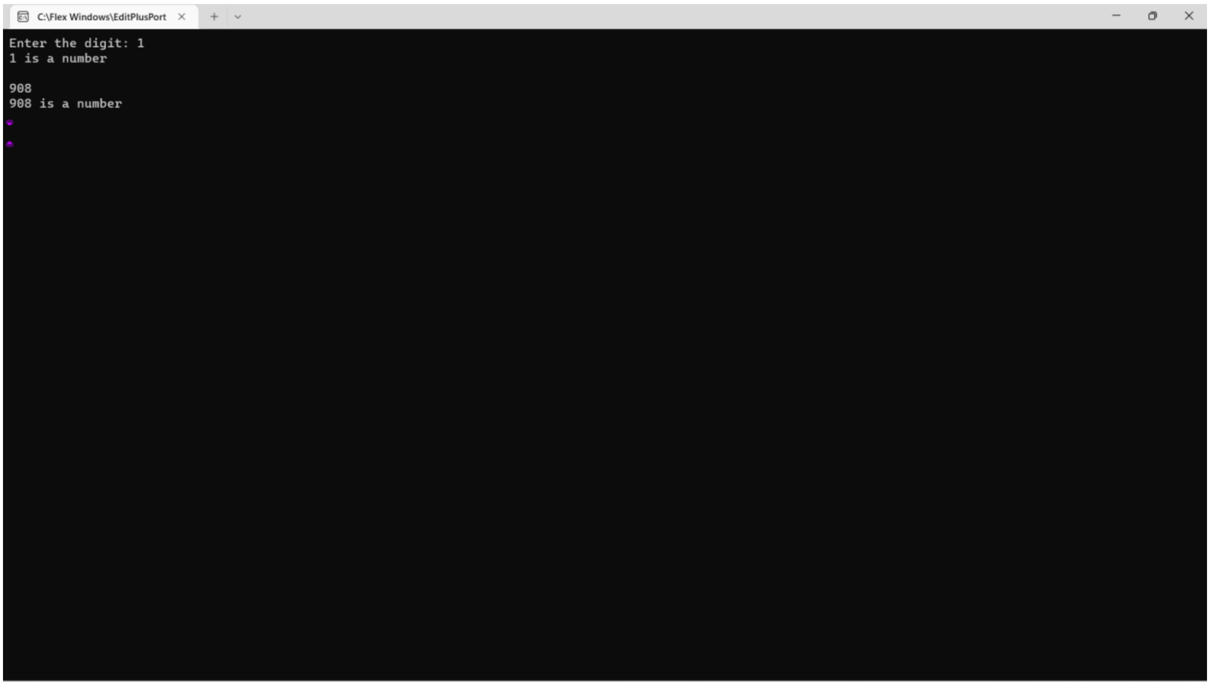
```
	printf("Enter the string: ");
	yylex();
	return 0;
}
```



 Integers with the regular expression: digit+

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

%%

[0-9]+ {printf("%s is a number\n",yytext);}

%%
```

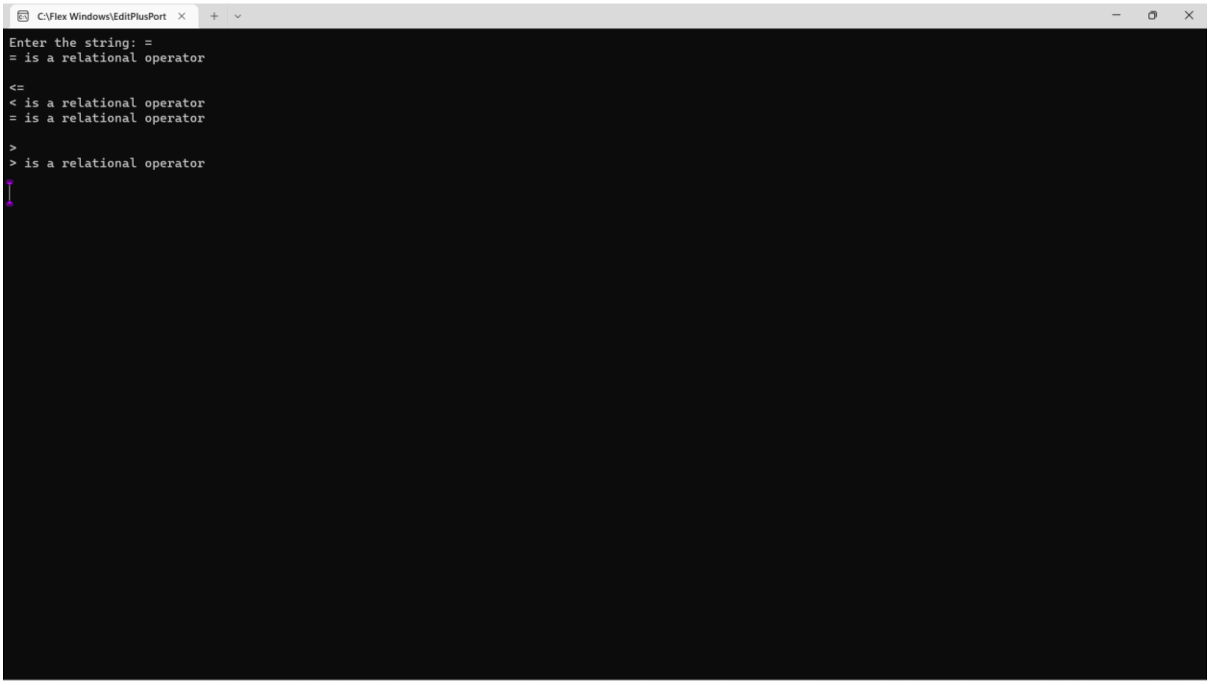
```
int yywrap()
{
	return 1;
}

int main()
{
	printf("Enter the string: ");
	yylex();
	return 0;
}
```



▯ Relational operators: <, >, <=, >=, ==, !=

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

%%
```

```
["<" | "<=" | ">" | ">=" | "==" | "!="] {printf("%s is a relational operator\n",yytext);}
%%

int yywrap()
{
	return 1;
}

int main()
{
	printf("Enter the string: ");
	yylex();
	return 0;
}
```



6. Implement Recursive Descent Parser for the Expression Grammar given below.

E → TE’

E’→ +TE’ | ε

T → FT’

T’→ *FT’ | ε

F → (E) | i

```
#include<stdio.h>
#include<string.h>
int E(),Edash(),T(),Tdash(),F();
char *ip;
char string[50];
int main()
{
   printf("Enter the string:\n");
   scanf("%s",string);
   ip=string;
   printf("\n\nInput\tAction\n--------------------------------\n");
   if(E() && *ip=='\0'){
      printf("\n--------------------------------\n");
      printf("\n String is successfully parsed\n");
   }
   else{
      printf("\n--------------------------------\n");
      printf("Error in parsing String\n");
```

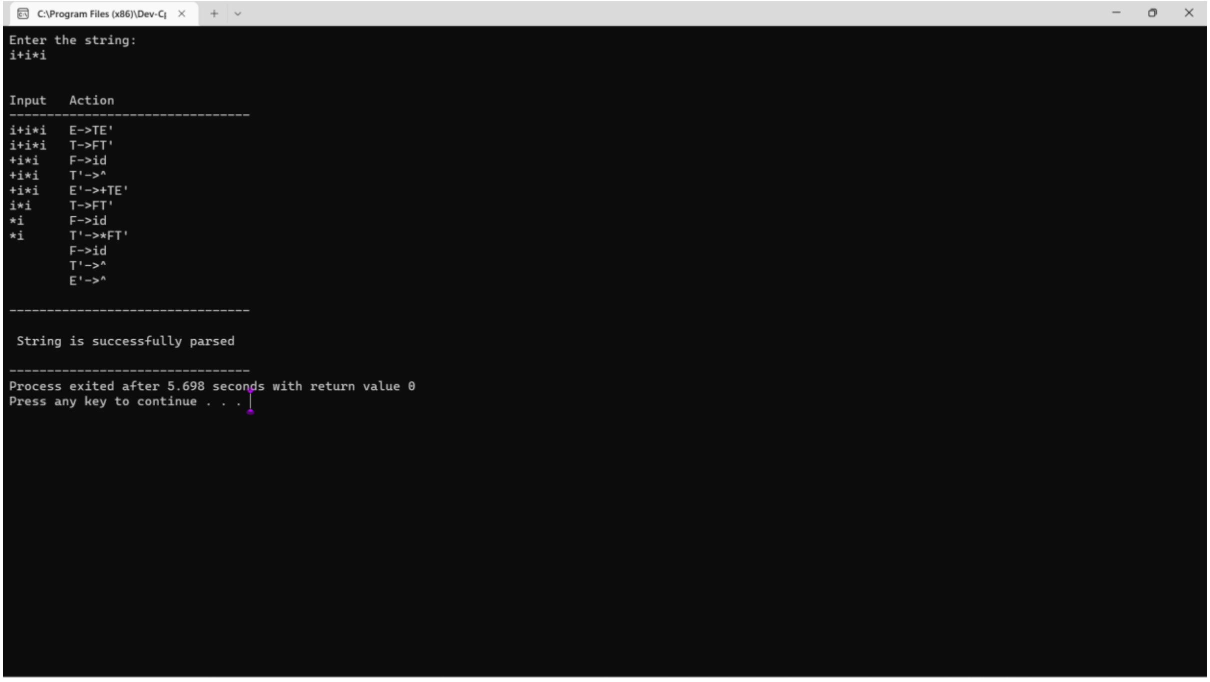
```
    }
}
int E()
{
    printf("%s\tE->TE' \n",ip);
    if(T())
    {
        if(Edash())
        {
            return 1;
        }
        else
            return 0;
    }
    else
        return 0;
}
int Edash()
{
    if(*ip=='+')
    {
        printf("%s\tE'->+TE' \n",ip);
        ip++;
        if(T())
        {
            if(Edash())
```

```
            {
                return 1;
            }
            else
                return 0;
        }
        else
            return 0;
    }
    else
    {
        printf("%s\tE'->^ \n",ip);
        return 1;
    }
}
int T()
{
    printf("%s\tT->FT' \n",ip);
    if(F())
    {
        if(Tdash())
        {
            return 1;
        }
        else
            return 0;
```

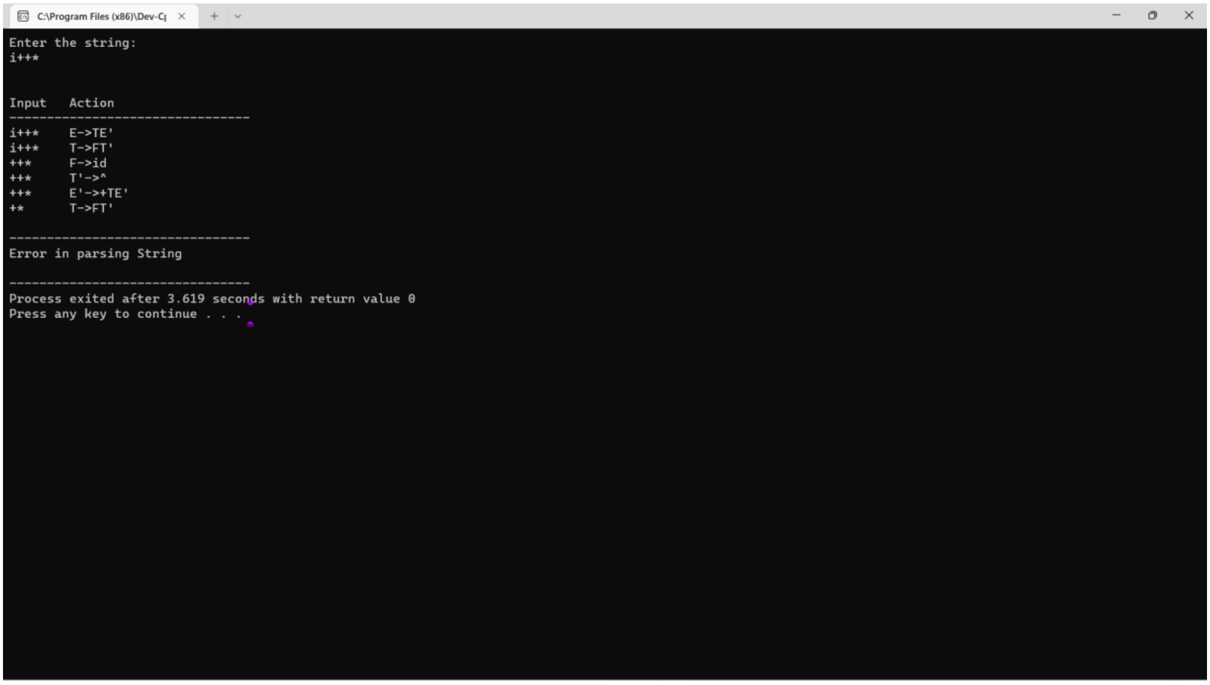
```
    }
    else
        return 0;
}
int Tdash()
{
    if(*ip=='*')
    {
        printf("%s\tT'->*FT' \n",ip);
        ip++;
        if(F())
        {
            if(Tdash())
            {
                return 1;
            }
            else
                return 0;
        }
        else
            return 0;
    }
    else
    {
        printf("%s\tT'->^ \n",ip);
        return 1;
```

```
    }
}
int F()
{
    if(*ip=='(')
    {
        printf("%s\tF->(E) \n",ip);
        ip++;
        if(E())
        {
            if(*ip==')')
            {
                ip++;
                return 1;
            }
            else
                return 0;
        }
        else
            return 0;
    }
    else if(*ip=='i')
    {
        ip++;
        printf("%s\tF->id \n",ip);
        return 1;
```

```
    }
    else
        return 0;
}
```



```
Enter the string:
i+i*i


Input   Action
--------------------------------
i+i*i   E->TE'
i+i*i   T->FT'
+i*i    F->id
+i*i    T'->^
+i*i    E'->+TE'
i*i     T->FT'
*i      F->id
*i      T'->*FT'
        F->id
        T'->^
        E'->^

--------------------------------

 String is successfully parsed

--------------------------------
Process exited after 5.698 seconds with return value 0
Press any key to continue . . .
```



```
Enter the string:
i++*


Input   Action
--------------------------------
i++*    E->TE'
i++*    T->FT'
++*     F->id
++*     T'->^
++*     E'->+TE'
+*      T->FT'

--------------------------------
Error in parsing String

--------------------------------
Process exited after 3.619 seconds with return value 0
Press any key to continue . . .
```

7. Lab Assignment: Construct Recursive Descent Parser for the grammar

G = ({S, L}, {(,), a, ,}, {S → (L) | a ; L→ L, S | S}, S) and verify the acceptability of the

following strings:

i. (a,(a,a))

ii. (a,((a,a),(a,a)))

You can manually eliminate Left Recursion if any in the grammar.

```
#include<stdio.h>
#include<string.h>
 int S(),L(),Ldash();
 char *ip;
 char string[50];
 int main()
{
    printf("Enter the string:\n");
    scanf("%s" ,string);
    ip=string;
    printf("\n\nInput\tAction\n\n");
    if(S() && *ip=='\0'){
        printf("\n---------\n");
        printf("\n string is successfully parsed\n");}
    else{
        printf("\n\n");
        printf("Error in parsing String\n");


      }
```

```
    return 0;
}

int S()
{
    if(*ip=='(')
    {
        printf("%s\tS->(L) \n",ip);
        ip++;
        if(L())
        {
            if(*ip==')')
            {

                ip++;
                return 1;
            }
            else
                return 0;
```

```
        }
        else
            return 0;
     }
     else if(*ip=='a')
     {
        ip++;
        printf("%s\tS->a \n",ip);
        return 1;
     }
     else
         return 0;
}
int L()
{
    printf("%s\tL->SL' \n",ip);
    if(S())
    {
      if(Ldash())
      {


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

```
    else
        return 0;
}
int Ldash()
{

    if(*ip==',')
    {
        printf("%s\tL'->,SL' \n",ip);
        ip++;
        if(S())
        {
            if(Ldash())
            {
                return 1;
            }
            else
                return 0;
        }
        else
            return 0;
    }
    else
    {
         printf("%s\tL'->? \n",ip);
         return 1;
```

```
    }
}
```

```
C:\Program Files (x86)\Dev-C

Enter the string:
(a,((a,a),(a,a)))


Input   Action

(a,((a,a),(a,a)))       S->(L)
a,((a,a),(a,a)))        L->SL'
,((a,a),(a,a))) S->a
,((a,a),(a,a))) L'->,SL'
((a,a),(a,a)))  S->(L)
(a,a),(a,a)))   L->SL'
(a,a),(a,a)))   S->(L)
a,a),(a,a)))    L->SL'
,a),(a,a)))     S->a
,a),(a,a)))     L'->,SL'
),(a,a)))       S->a
),(a,a)))       L'->?
,(a,a)))        L'->,SL'
(a,a))) S->(L)
a,a)))  L->SL'
,a)))   S->a
,a)))   L'->,SL'
)))     S->a
)))     L'->?
))      L'->?
)       L'->?

---------

 string is successfully parsed

--------------------------------
Process exited after 27.64 seconds with return value 0
Press any key to continue . . .
```

```
C:\Program Files (x86)\Dev-C

Enter the string:
(a,a


Input   Action

(a,a    S->(L)
a,a     L->SL'
,a      S->a
,a      L'->,SL'
        S->a
        L'->?


Error in parsing String

--------------------------------
Process exited after 3.599 seconds with return value 0
Press any key to continue . . .
```

8. Write a C program for the computation of FIRST and FOLLOW for a given CFG

```
#include<stdio.h>

#include<ctype.h>

#include<string.h>


// Functions to calculate Follow

void followfirst(char, int, int);

void follow(char c);


// Function to calculate First

void findfirst(char, int, int);


int count, n = 0;


// Stores the final result

// of the First Sets

char calc_first[10][100];


// Stores the final result

// of the Follow Sets

char calc_follow[10][100];

int m = 0;


// Stores the production rules

char production[10][10];

char f[10], first[10];
```

```
int k;
char ck;
int e;

int main(int argc, char **argv)
{
    int jm = 0;
    int km = 0;
    int i, choice;
    char c, ch;
    count = 8;

    // The Input grammar
    strcpy(production[0], "E=TR");
    strcpy(production[1], "R=+TR");
    strcpy(production[2], "R=#");
    strcpy(production[3], "T=FY");
    strcpy(production[4], "Y=*FY");
    strcpy(production[5], "Y=#");
    strcpy(production[6], "F=(E)");
    strcpy(production[7], "F=i");


    int kay;
    char done[count];
    int ptr = -1;
```

```
// Initializing the calc_first array
for(k = 0; k < count; k++) {
    for(kay = 0; kay < 100; kay++) {
        calc_first[k][kay] = '!';
    }
}
int point1 = 0, point2, xxx;

for(k = 0; k < count; k++)
{
    c = production[k][0];
    point2 = 0;
    xxx = 0;

    // Checking if First of c has
    // already been calculated
    for(kay = 0; kay <= ptr; kay++)
        if(c == done[kay])
            xxx = 1;

    if (xxx == 1)
        continue;

    // Function call
    findfirst(c, 0, 0);
```

```
ptr += 1;

// Adding c to the calculated list
done[ptr] = c;
printf("\n First(%c) = { ", c);
calc_first[point1][point2++] = c;

// Printing the First Sets of the grammar
for(i = 0 + jm; i < n; i++) {
    int lark = 0, chk = 0;

    for(lark = 0; lark < point2; lark++) {

        if (first[i] == calc_first[point1][lark])
        {
            chk = 1;
            break;
        }
    }
    if(chk == 0)
    {
        printf("%c, ", first[i]);
        calc_first[point1][point2++] = first[i];
    }
}
printf("}\n");
```

```
    jm = n;
    point1++;
}
printf("\n");
printf("-----------------------------------------------\n\n");
char donee[count];
ptr = -1;

// Initializing the calc_follow array
for(k = 0; k < count; k++) {
    for(kay = 0; kay < 100; kay++) {
        calc_follow[k][kay] = '!';
    }
}
point1 = 0;
int land = 0;
for(e = 0; e < count; e++)
{
    ck = production[e][0];
    point2 = 0;
    xxx = 0;

    // Checking if Follow of ck
    // has already been calculated
    for(kay = 0; kay <= ptr; kay++)
        if(ck == donee[kay])
```

```
        xxx = 1;

if (xxx == 1)
    continue;
land += 1;

// Function call
follow(ck);
ptr += 1;

// Adding ck to the calculated list
donee[ptr] = ck;
printf(" Follow(%c) = { ", ck);
calc_follow[point1][point2++] = ck;

// Printing the Follow Sets of the grammar
for(i = 0 + km; i < m; i++) {
    int lark = 0, chk = 0;
    for(lark = 0; lark < point2; lark++)
    {
        if (f[i] == calc_follow[point1][lark])
        {
            chk = 1;
            break;
        }
    }
```

```
            if(chk == 0)
            {
                printf("%c, ", f[i]);
                calc_follow[point1][point2++] = f[i];
            }
        }
        printf(" }\n\n");
        km = m;
        point1++;
    }
}

void follow(char c)
{
    int i, j;

    // Adding "$" to the follow
    // set of the start symbol
    if(production[0][0] == c) {
        f[m++] = '$';
    }
    for(i = 0; i < 10; i++)
    {
        for(j = 2;j < 10; j++)
        {
            if(production[i][j] == c)
```

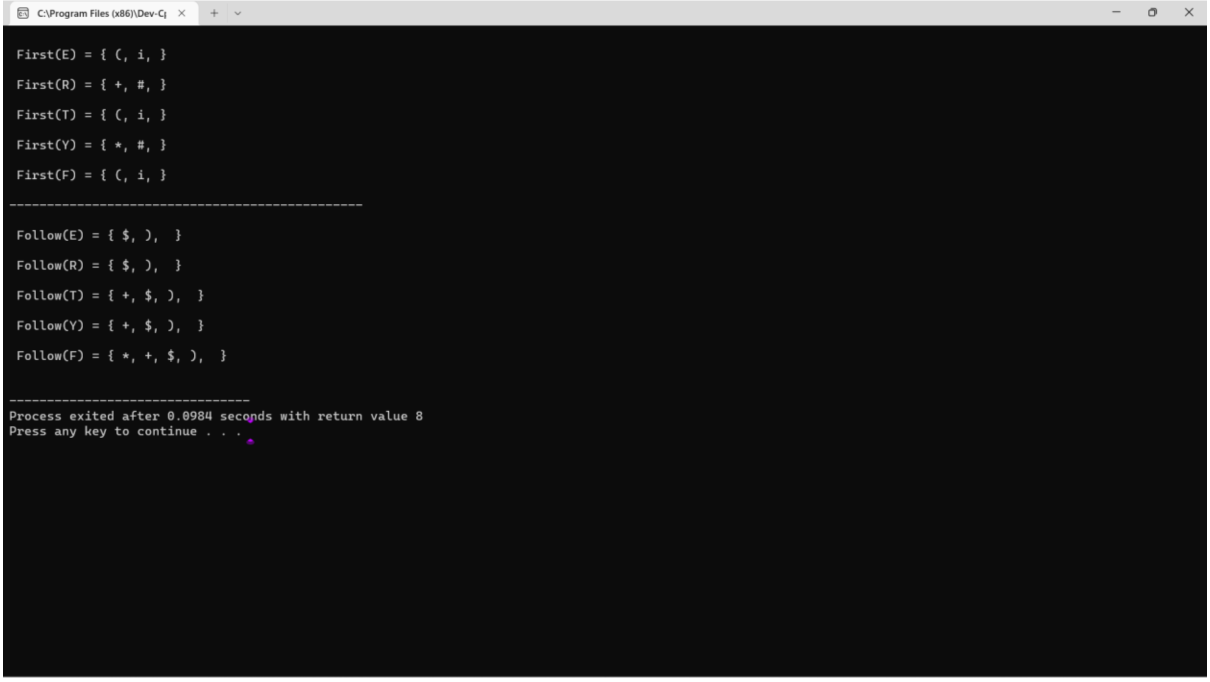
```
            {
                if(production[i][j+1] != '\0')
                {
                    // Calculate the first of the next
                    // Non-Terminal in the production
                    followfirst(production[i][j+1], i, (j+2));
                }

                if(production[i][j+1]=='\0' && c!=production[i][0])
                {
                    // Calculate the follow of the Non-Terminal
                    // in the L.H.S. of the production
                    follow(production[i][0]);
                }
            }
        }
    }
}

void findfirst(char c, int q1, int q2)
{
    int j;

    // The case where we
    // encounter a Terminal
    if(!(isupper(c))) {
```

```
    first[n++] = c;
}
for(j = 0; j < count; j++)
{
    if(production[j][0] == c)
    {
        if(production[j][2] == '#')
        {
            if(production[q1][q2] == '\0')
                first[n++] = '#';
            else if(production[q1][q2] != '\0'
                    && (q1 != 0 || q2 != 0))
            {
                // Recursion to calculate First of New
                // Non-Terminal we encounter after epsilon
                findfirst(production[q1][q2], q1, (q2+1));
            }
            else
                first[n++] = '#';
        }
        else if(!isupper(production[j][2]))
        {
            first[n++] = production[j][2];
        }
        else
        {
```

```
            // Recursion to calculate First of
            // New Non-Terminal we encounter
            // at the beginning
            findfirst(production[j][2], j, 3);
         }
      }
   }
}

void followfirst(char c, int c1, int c2)
{
   int k;

   // The case where we encounter
   // a Terminal
   if(!(isupper(c)))
      f[m++] = c;
   else
   {
      int i = 0, j = 1;
      for(i = 0; i < count; i++)
      {
         if(calc_first[i][0] == c)
            break;
      }
```

```
//Including the First set of the
// Non-Terminal in the Follow of
// the original query
while(calc_first[i][j] != '!')
{
   if(calc_first[i][j] != '#')
   {
      f[m++] = calc_first[i][j];
   }
   else
   {
      if(production[c1][c2] == '\0')
      {
         // Case where we reach the
         // end of a production
         follow(production[c1][0]);
      }
      else
      {
         // Recursion to the next symbol
         // in case we encounter a "#"
         followfirst(production[c1][c2], c1, c2+1);
      }
   }
   j++;
}
```

```
   }
}
```



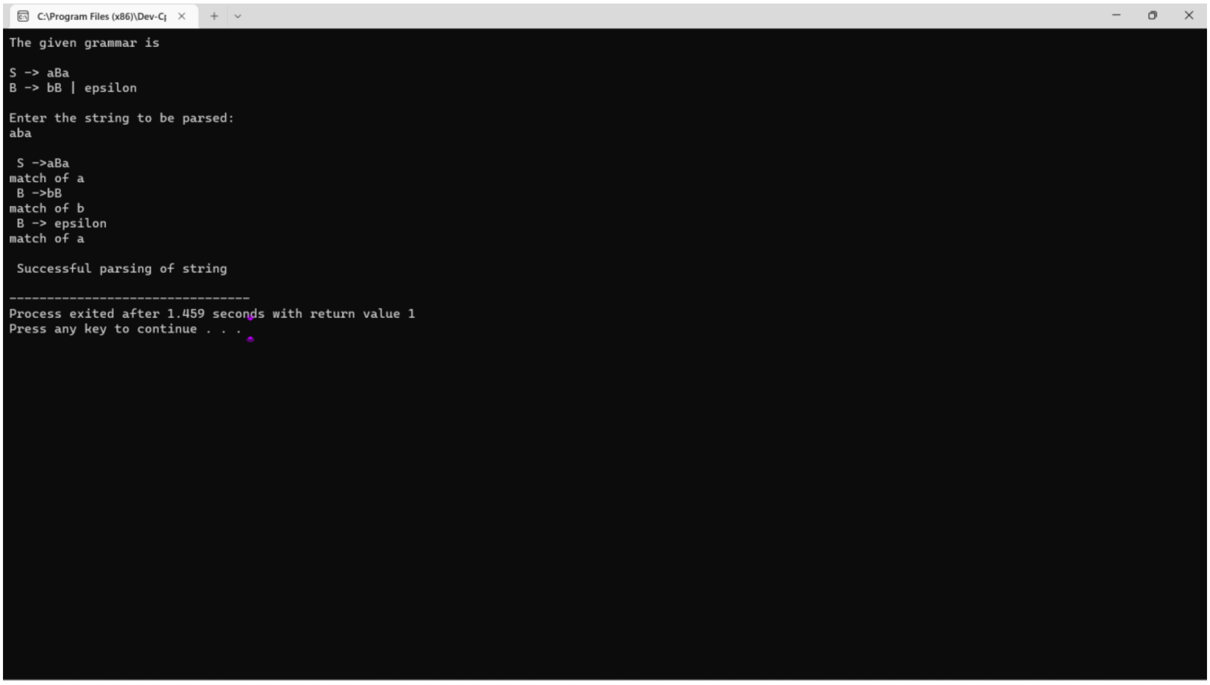
9. Implement non-recursive Predictive Parser for the grammar

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
int i=0,top=0;
char stack[20],ip[20];
void push(char c)
{
   if (top>=20)
      printf("Stack Overflow");
   else
```

```
        stack[top++]=c;
}
void pop(void)
{
    if(top<0)
        printf("Stack underflow");
    else
        top--;
}
void error(void)
{
    printf("\n\nSyntax Error!!!! String is invalid\n");
    exit(0);
}
int main()
{
    int n;
    printf("The given grammar is\n\n");
    printf("S -> aBa\n");
    printf("B -> bB | epsilon \n\n");
    printf("Enter the string to be parsed:\n");
    scanf("%s",ip);
    n=strlen(ip);
    ip[n]='$';
    ip[n+1]='\0';
    push('$');
```

```
push('S');
while(ip[i]!='\0')
{
    if(ip[i]=='$' && stack[top-1]=='$')
    {
        printf("\n\n Successful parsing of string \n");
        return 1;
    }
    else if(ip[i]==stack[top-1])
    {
        printf("\nmatch of %c ",ip[i]);
        i++;pop();
    }
    else
    {
        if(stack[top-1]=='S' && ip[i]=='a')
        {
            printf(" \n S ->aBa");
            pop();
            push('a');
            push('B');
            push('a');
        }
        else if(stack[top-1]=='B' && ip[i]=='b')
        {
            printf("\n B ->bB");
```

```
          pop();push('B');push('b');
        }
        else if(stack[top-1]=='B' && ip[i]=='a')
        {
          printf("\n B -> epsilon");
          pop();
        }
        else
          error();
      }
    }
}
```



10. Lab Assignment: Implement Predictive Parser using C for the Expression Grammar

E → TE’

E’→ +TE’ | ε

T → FT’

T’→ *FT’ | ε

F → (E) | d

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
int i=0,top=0;
char stack[20],ip[20];

void push(char c)
{
   if (top>=20)
      printf("Stack Overflow");
   else
      stack[top++]=c;
}

void pop(void)
{
   if(top<0)
```

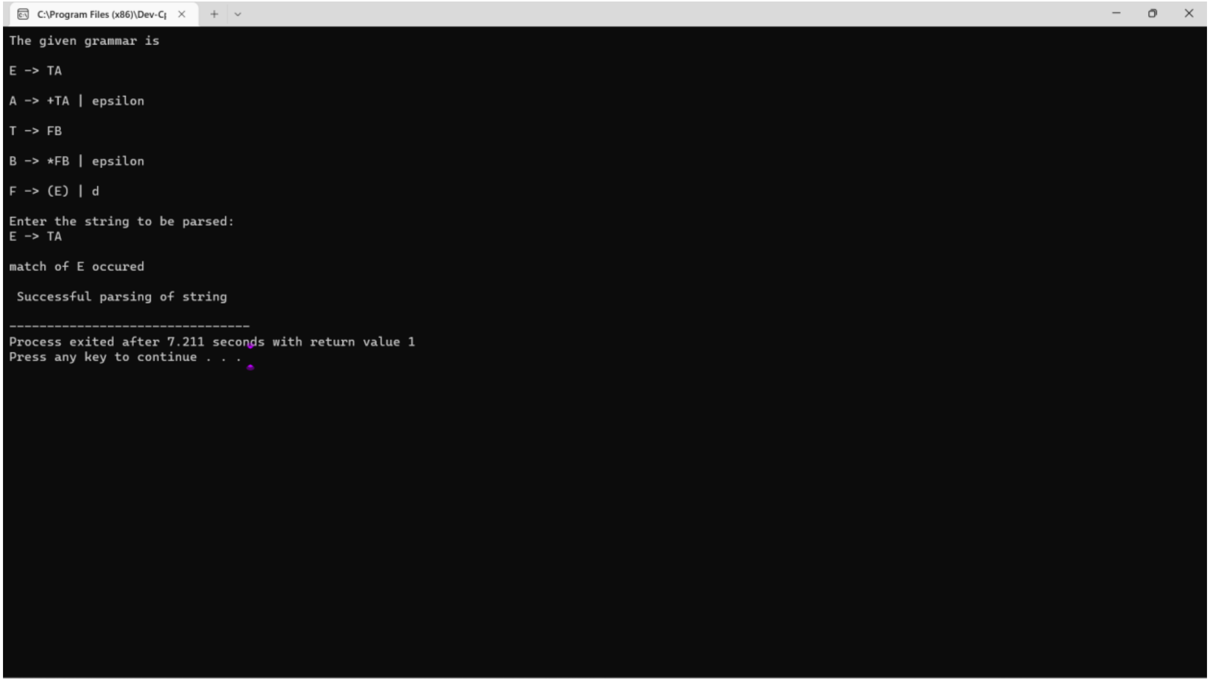
```
        printf("Stack underflow");
    else
        top--;
}

void error(void)
{
printf("\n\nSyntax Error!!!! String is invalid\n");
getch();
exit(0);
}

int main()
{
int n;

printf("The given grammar is\n\n");
printf("E -> TA\n\n");
printf("A -> +TA | epsilon \n\n");
printf("T -> FB\n\n");
printf("B -> *FB | epsilon \n\n");
printf("F -> (E) | d \n\n");
printf("Enter the string to be parsed:\n");
scanf("%s",ip);
n=strlen(ip);
ip[n]='$';
```

```
ip[n+1]='\0';
push('$');
push('E');
while(ip[i]!='\0')
{ if(ip[i]=='$' && stack[top-1]=='$')
  {
    printf("\n\n Successful parsing of string \n");
    return(1);
  }
  else
    if(ip[i]==stack[top-1])
    {
      printf("\nmatch of %c occured ",ip[i]);
      i++;pop();
    }
  else
    {
       if(stack[top-1]=='E' && ip[i]=='d')
       {
             printf(" \n E ->TA");
             pop();
             push('A');
             push('T');
       }
       else
       if(stack[top-1]=='A' && ip[i]=='+')
```

```
{
   printf("\n A ->+TA");
   pop();
   push('A');
   push('T');
   push('+');
}
else
if(stack[top-1]=='A' && ip[i]=='$')
{
   printf("\n A -> epsilon");
   pop();
}
else
if(stack[top-1]=='T' && ip[i]=='d')
{
     printf(" \n T ->FB");
     pop();
     push('B');
     push('F');
}
else if(stack[top-1]=='B' && ip[i]=='*')
{
   printf("\n B ->*FB");
   pop();
   push('B');
```

```
            push('F');
            push('*');
        }
        else if(stack[top-1]=='B' && (ip[i]=='+'||ip[i]=='$'))
        {
            printf("\n B -> epsilon");
            pop();
        }
        else if(stack[top-1]=='F' && ip[i]=='(')
        {
            printf("\n F ->(E)");
            pop();
            push(')');
            push('E');
            push('(');
        }
        else if(stack[top-1]=='F' && ip[i]=='d')
        {
            printf("\n F ->d");
            pop();
            push('d');
        }
        else
            error();
    }
}
```

}



11. Implementation of Shift Reduce parser using C for the following grammar and illustrate

the parser’s actions for a valid and an invalid string.

E→E+E

E→E*E

E→(E)

E→d

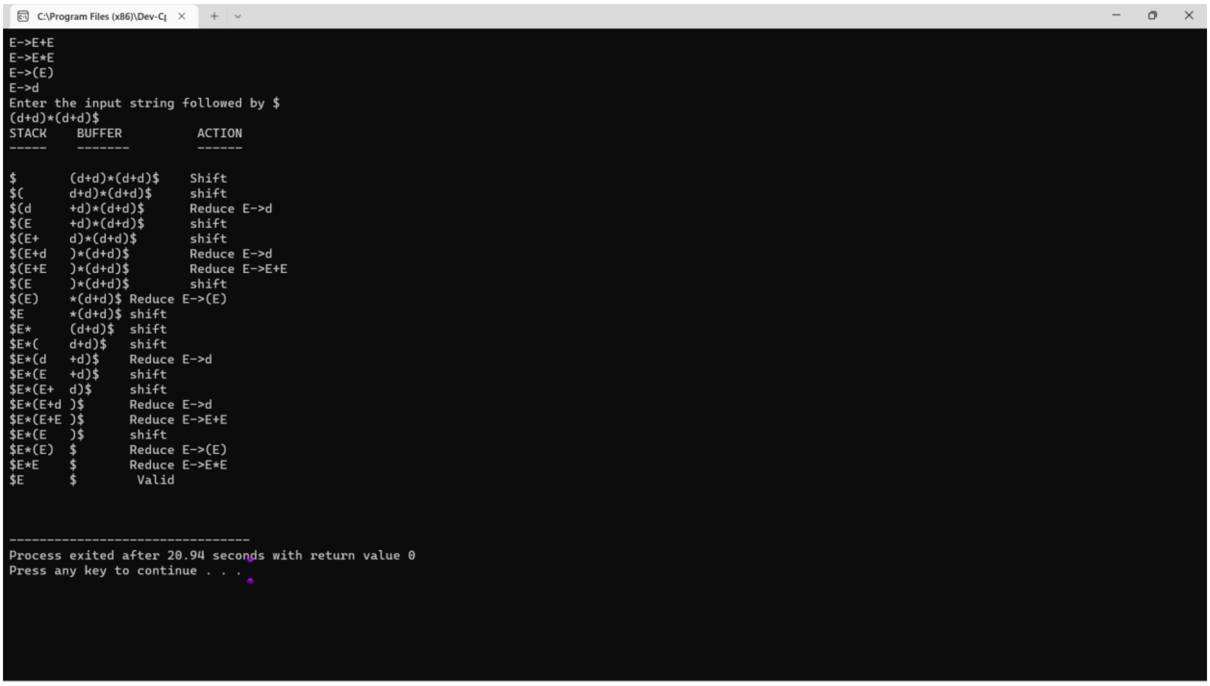
```
#include<stdio.h>

#include<stdlib.h>


char stack[100]="\0", input[100], *ip;

int top=-1;

void push(char c)

{
```

```
    top++;
    stack[top]=c;
}
void pop()
{
    stack[top]='\0';
    top--;
}
void display()
{
    printf("\n%s\t%s\t",stack,ip);
}
void main()
{
    printf("E->E+E\n");
    printf("E->E*E\n");
    printf("E->(E)\n");
    printf("E->d\n");
    printf("Enter the input string followed by $ \n");
    scanf("%s",input);
    ip=input;
    push('$');
    printf("STACK\t BUFFER \t ACTION\n");
    printf("-----\t ------- \t ------\n");
    display();
    if(stack[top]=='$' && *ip=='$')
```

```
  {
      printf("Null Input");
      exit(0);
  }
  do
  {
      if((stack[top]=='E' && stack[top-1]=='$') && (*(ip)=='$'))
      {
          display();
          printf(" Valid\n\n\n");
          break;
      }
      if(stack[top]=='$')
      {
          push(*ip);
          ip++;
          printf("Shift");
      }
      else if(stack[top]=='d')
      {
          display();
          pop();
          push('E');
          printf("Reduce E->d");
      }
      else if(stack[top]=='E' && stack[top-1]=='+' && stack[top-2]=='E'&&
*ip!='*')
```

```
{
   display();
   pop();
   pop();
   pop();
   push('E');
   printf("Reduce E->E+E");
}
else if(stack[top]=='E' && stack[top-1]=='*' && stack[top-2]=='E')
{
   display();
   pop();
   pop();
   pop();
   push('E');
   printf("Reduce E->E*E");
}
else if(stack[top]==')' && stack[top-1]=='E' && stack[top-2]=='(')
{
   display();

   pop();
   pop();
   pop();
   push('E');
   printf("Reduce E->(E)");
```

```
        }
        else if(*ip=='$')
        {
            printf(" Invalid\n\n\n");
            break;
        }
        else
        {
            display();
            push(*ip);
            ip++;
            printf("shift");
        }
    }while(1);
}
```



13. Implement LALR parser using LEX and YACC for the following Grammar:

E → E+T |T

E’→ T*F | F

F → (E) | d

Yacc code:

```
%{
#include <ctype.h>
#include<stdio.h>
#include<stdlib.h>
%}
%token digit
%%
S: E {printf("Reached\n\n"); getch();}
;
E: E '+' T
| E '-' T
| T
;
T: T '*' P
| T '/' P
| P
;
P: F '^' P
| F
;
```

```
F: '(' E ')'
| digit
;
%%
int main()
{
printf("Enter infix expression: ");
yyparse();
}
int yyerror(s)
char *s;
{
printf("Error");
getch();
}
```

Lex Code:

```
%{
#include <stdio.h>
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+ {
yylval=atoi(yytext);
return (digit);
}
```

```
[\t];
[\n] return 0;
.return yytext[0];
%%
int yywrap(){
return 1;
}
```