PRACTICAL -6

AIM: Write a program to identify that the given grammar is Left recursive or not

Procedure:

If a grammar contains a pair of productions of the form A→**Aα / β**, where **β** does not begin with an A, then the grammar is a "Left Recursive grammar". Left recursion can be eliminated from the grammar by replacing **A→Aα / β** with the productions **A→ βA'** and **A'→αA'/ ε.**

If grammar contains productions:
A->Aa1/Aa2/...../Aam/ β1/β2/...../βn
Then left-recursion can be eliminated in two steps by adding following producations in place of the ones above:
STEP 1: **A->β1B/β2B/……./Bn**
STEP 2: **B->a1B/a2B/……/amB/€**

Code/Method:

```
 #include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char a[10][10],i=-1,flag=0;
clrscr();
printf("Enter the Productions:\n");
do
{
i++;
gets(a[i]);
}
while(a[i][0]!='\0');
printf("Your Productions:\n");
for(i=0;a[i][0]!='\0';i++)
puts(a[i]);
for(i=0;a[i][0]!='\0';i++)
```

```
{
if(a[i][0]==a[i][3])
flag++;
}
if(flag!=0)
printf("Grammar is Left Recursive.");
else
printf("Grammar is not Left Recursive.");
getch();
}
```

Output:

Enter the Productions

A→Ab|c

Grammar is Left Recursive

Enter the Productions

A→B|c

Grammar is not Left Recursive

PRACTICAL -7

AIM: Write a C program for constructing recursive descent parser

Procedure:

A recursive descent parser is a kind of top-down parser built from a set of mutually recursive procedures (or a non-recursive equivalent) where each such procedure usually implements one of the productions of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes. Recursive descent parsing is probably the most well-known and intuitive technique applicable to a subclass of context-free grammars. A subroutine for each non terminal should determine a rule according to which the substring shall be parsed. If such a rule can be determined using k next symbols of the input, then this is a standard LL(k) parser.

Code/Method:

```
#include"stdio.h"
#include"conio.h"
#include"string.h"
#include"stdlib.h"
#include"ctype.h"

char ip_sym[15],ip_ptr=0,op[50],tmp[50];
void e_prime();
void e();
void t_prime();
void t();
void f();
void advance();
int n=0;
void e()
{
 strcpy(op,"TE'");
 printf("E=%-25s",op);
 printf("E->TE'\n");
 t();
 e_prime();
}

void e_prime()
{
int i,n=0,l;
for(i=0;i<=strlen(op);i++)
   if(op[i]!='e')
 tmp[n++]=op[i];
strcpy(op,tmp);
l=strlen(op);
for(n=0;n < l && op[n]!='E';n++);
if(ip_sym[ip_ptr]=='+')
```

```
 {
    i=n+2;
 do
 {
 op[i+2]=op[i];
 i++;
 }while(i<=l);
  op[n++]='+';
  op[n++]='T';
  op[n++]='E';
  op[n++]=39;
  printf("E=%-25s",op);
  printf("E'->+TE'\n");
  advance();
  t();
  e_prime();
 }
 else
 {
    op[n]='e';
  for(i=n+1;i<=strlen(op);i++)
 op[i]=op[i+1];
 printf("E=%-25s",op);
 printf("E'->e");
 }
}
void t()
{
 int i,n=0,l;
 for(i=0;i<=strlen(op);i++)
  if(op[i]!='e')
   tmp[n++]=op[i];
 strcpy(op,tmp);
 l=strlen(op);
for(n=0;n < l && op[n]!='T';n++);

 i=n+1;
 do
 {
  op[i+2]=op[i];
  i++;
 }while(i < l);
 op[n++]='F';
 op[n++]='T';
 op[n++]=39;
 printf("E=%-25s",op);
 printf("T->FT'\n");
 f();
 t_prime();
}

void t_prime()
```

```
{
int i,n=0,l;
for(i=0;i<=strlen(op);i++)
   if(op[i]!='e')
 tmp[n++]=op[i];
strcpy(op,tmp);
l=strlen(op);
for(n=0;n < l && op[n]!='T';n++);
if(ip_sym[ip_ptr]=='*')
 {
    i=n+2;
 do
 {
 op[i+2]=op[i];
 i++;
 }while(i < l);
  op[n++]='*';
  op[n++]='F';
  op[n++]='T';
  op[n++]=39;
  printf("E=%-25s",op);
  printf("T'->*FT'\n");
  advance();
  f();
  t_prime();
 }
 else
 {
   op[n]='e';
  for(i=n+1;i<=strlen(op);i++)
 op[i]=op[i+1];
 printf("E=%-25s",op);
 printf("T'->e\n");
 }
}

void f()
{
int i,n=0,l;
for(i=0;i<=strlen(op);i++)
   if(op[i]!='e')
 tmp[n++]=op[i];
 strcpy(op,tmp);
l=strlen(op);
for(n=0;n < l && op[n]!='F';n++);
 if((ip_sym[ip_ptr]=='i')||(ip_sym[ip_ptr]=='I'))
 {
 op[n]='i';
 printf("E=%-25s",op);
 printf("F->i\n");
 advance();
 }
```

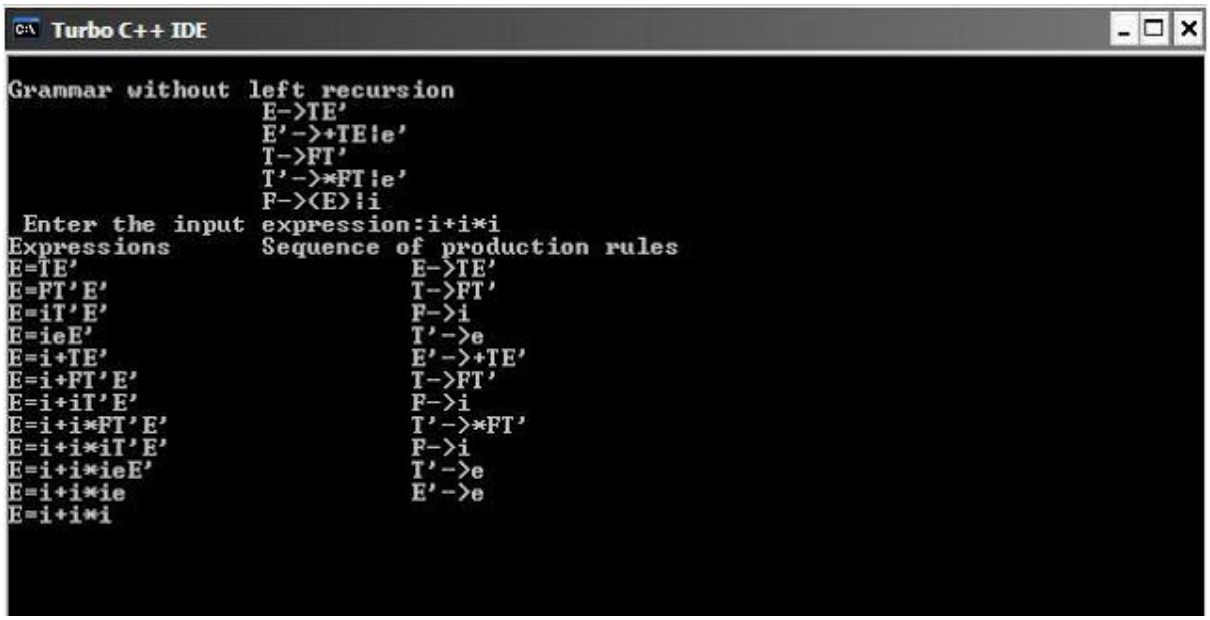
```
else
{
 if(ip_sym[ip_ptr]=='(')
 {
  advance();
  e();
  if(ip_sym[ip_ptr]==')')
  {
   advance();
    i=n+2;
do
{
op[i+2]=op[i];
i++;
}while(i<=l);
 op[n++]='(';
 op[n++]='E';
 op[n++]=')';
 printf("E=%-25s",op);
 printf("F->(E)\n");
  }
 }
 else
 {
  printf("\n\t syntax error");
  getch();
  exit(1);
 }
}
}

void advance()
{
 ip_ptr++;
}

void main()
{
 int i;
 clrscr();
 printf("\nGrammar without left recursion");
 printf("\n\t\t E->TE' \n\t\t E'->+TE'|e \n\t\t T->FT' ");
 printf("\n\t\t T'->*FT'|e \n\t\t F->(E)|i");
 printf("\n Enter the input expression:");
 gets(ip_sym);
 printf("Expressions");
 printf("\t Sequence of production rules\n");
  e();
 for(i=0;i < strlen(ip_sym);i++)
 {
 if(ip_sym[i]!='+'&&ip_sym[i]!='*'&&ip_sym[i]!='('&&
    ip_sym[i]!=')'&&ip_sym[i]!='i'&&ip_sym[i]!='I')
```

```
 {
  printf("\nSyntax error");
  break;
 }
 for(i=0;i<=strlen(op);i++)
   if(op[i]!='e')
 tmp[n++]=op[i];
   strcpy(op,tmp);
   printf("\nE=%-25s",op);
 }
 getch();
}
```

Output:



PRACTICAL -8

AIM: Write a program in C to implement shift-reduce parser

Procedure:

A shift-reduce parser is a class of efficient, table-driven bottom-up parsing methods for computer languages and other notations formally defined by a grammar. The parsing methods most commonly used today, LR parsing and its variations, are shift-reduce methods.

A shift-reduce parser works by doing some combination of Shift steps and Reduce steps, hence the name.

- A **Shift** step advances in the input stream by one symbol. That shifted symbol becomes a new single-node parse tree.
- A **Reduce** step applies a completed grammar rule to some of the recent parse trees, joining them together as one tree with a new root symbol.

The parser continues with these steps until all of the input has been consumed and all of the parse trees have been reduced to a single tree representing an entire legal input

Code/Method:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int k=0,z=0,i=0,j=0,c=0;
char a[16],ac[20],stk[15],act[10];
void check();
void main()
  {
    clrscr();
    puts("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->id");
    puts("enter input string ");
    gets(a);
    c=strlen(a);
    strcpy(act,"SHIFT->");
    puts("stack \t input \t action");
    for(k=0,i=0; j<c; k++,i++,j++)
     {
      if(a[j]=='i' && a[j+1]=='d')
       {
         stk[i]=a[j];
         stk[i+1]=a[j+1];
         stk[i+2]='\0';
```

```
          a[j]=' ';
          a[j+1]=' ';
          printf("\n$%s\t%s$\t%sid",stk,a,act);
          check();
        }
       else
        {
          stk[i]=a[j];
          stk[i+1]='\0';
          a[j]=' ';
          printf("\n$%s\t%s$\t%ssymbols",stk,a,act);
          check();
        }
     }
     getch();
  }
void check()
  {
    strcpy(ac,"REDUCE TO E");
    for(z=0; z<c; z++)
      if(stk[z]=='i' && stk[z+1]=='d')
       {
         stk[z]='E';
         stk[z+1]='\0';
         printf("\n$%s\t%s$\t%s",stk,a,ac);
         j++;
       }
    for(z=0; z<c; z++)
      if(stk[z]=='E' && stk[z+1]=='+' && stk[z+2]=='E')
       {
         stk[z]='E';
         stk[z+1]='\0';
         stk[z+2]='\0';
         printf("\n$%s\t%s$\t%s",stk,a,ac);
         i=i-2;
       }
    for(z=0; z<c; z++)
      if(stk[z]=='E' && stk[z+1]=='*' && stk[z+2]=='E')
       {
         stk[z]='E';
         stk[z+1]='\0';
         stk[z+1]='\0';
         printf("\n$%s\t%s$\t%s",stk,a,ac);
         i=i-2;
       }
    for(z=0; z<c; z++)
      if(stk[z]=='(' && stk[z+1]=='E' && stk[z+2]==')')
       {
         stk[z]='E';
         stk[z+1]='\0';
         stk[z+1]='\0';
         printf("\n$%s\t%s$\t%s",stk,a,ac);
```

```
        i=i-2;
      }
  }
```

Output:

Input: 2+3*5

Output: The value of expression is 17

PRACTICAL -9

AIM: Write a program in C for SLR parsing.

Procedure:

Parser: A parser for grammar G is a program that takes as input a string w and produces as output either a parse tree for w, if w is a sentence of G, or an error message indicating that w is not a sentence of G. often the parse tree is produced in only a figurative sense; in reality, the parse tree exists only as a sequence of actions made by stepping through the tree construction process. There are two basic types of parsers for the context free grammars-bottom up and top down. As indicated by their names, bottom-up parsers build parse trees from the bottom (leaves) to the top(root),while top down parsers start with the root and work down to the leaves. in both the cases the input to the parser is being scanned from left to right ,one symbol at a time.

Efficient bottom up parsers for a large class of context free grammars, these parsers are called LR parsers because they scan the input from left to right and construct a right most derivation in reverse. Logically, an LR parser consists of two parts, a driver routine and a parsing table. The driver routine is the same for all LR parsers; only the parsing table changes from one parser to another.

Code/Method:

```
#include<stdio.h>
#include<string.h>
int axn[][6][2]={
    {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
    {{-1,-1},{100,6},{-1,-1},{-1,-1},{-1,-1},{102,102}},
    {{-1,-1},{101,2},{100,7},{-1,-1},{101,2},{101,2}},
    {{-1,-1},{101,4},{101,4},{-1,-1},{101,4},{101,4}},
    {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
    {{-1,-1},{101,6},{101,6},{-1,-1},{101,6},{101,6}},
    {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
    {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},

    {{-1,-1},{100,6},{-1,-1},{-1,-1},{100,1},{-1,-1}},

    {{-1,-1},{101,1},{100,7},{-1,-1},{101,1},{101,1}},

    {{-1,-1},{101,3},{101,3},{-1,-1},{101,3},{101,3}},

    {{-1,-1},{101,5},{101,5},{-1,-1},{101,5},{101,5}}

    };//Axn Table

int gotot[12][3]={1,2,3,-1,-1,-1,-1,-1,-1,-1,-1,8,2,3,-1,-1,-1,
```

```
  -1,9,3,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};  //GoTo table

int a[10];

char b[10];

int top=-1,btop=-1,i;

void push(int k)

{

 if(top<9)

   a[++top]=k;

}
void pushb(char k)
{

 if(btop<9)
   b[++btop]=k;
}
char TOS()

{
  return a[top];
}
void pop()
{

  if(top>=0)
    top--;
}
void popb()
{

  if(btop>=0)
    b[btop--]='\0';
}
void display()

{

 for(i=0;i<=top;i++)

   printf("%d%c",a[i],b[i]);

}
```

```
void display1(char p[],int m) //Displays The Present Input String
{
 int l;
 printf("\t\t");

 for(l=m;p[l]!='\0';l++)
  printf("%c",p[l]);
 printf("\n");
}
void error()
{

 printf("Syntax Error");
}
void reduce(int p)

{

  int len,k,ad;

  char src,*dest;

  switch(p)

  {

case 1:dest="E+T";

     src='E';

     break;

case 2:dest="T";

     src='E';

     break;

case 3:dest="T*F";

     src='T';

     break;

case 4:dest="F";

     src='T';

     break;

case 5:dest="(E)";
```

```
      src='F';
      break;
case 6:dest="i";
      src='F';
      break;
default:dest="\0";
 src='\0';
 break;
 }
 for(k=0;k<strlen(dest);k++)
 {
  pop();
  popb();
 }
 pushb(src);
 switch(src)
 {
case 'E':ad=0;
 break;
case 'T':ad=1;
 break;
case 'F':ad=2;
 break;
default: ad=-1;
 break;
 }
```

```
 push(gotot[TOS()][ad]);
}
int main()
{
  int j,st,ic;
  char ip[20]="\0",an;
 // clrscr();
  printf("Enter any String\n");
+
  scanf("%s",ip);
  push(0);
  display();
  printf("\t%s\n",ip);
  for(j=0;ip[j]!='\0';)
  {
 st=TOS();
 an=ip[j];
 if(an>='a'&&an<='z') ic=0;
 else if(an=='+') ic=1;
 else if(an=='*') ic=2;
 else if(an=='(') ic=3;
 else if(an==')') ic=4;
 else if(an=='$') ic=5;
 else {
   error();
   break;
```

```
}
 if(axn[st][ic][0]==100)
  {
    pushb(an);
    push(axn[st][ic][1]);
    display();
    j++;
    display1(ip,j);
  }
 if(axn[st][ic][0]==101)
  {
    reduce(axn[st][ic][1]);
    display();
    display1(ip,j);
  }
 if(axn[st][ic][1]==102)
  {
 printf("Given String is accepted \n");
//  getch();
    break;
  }
/*  else
  {
 printf("Given String is rejected \n");
    break;
  }*/
```

```
 }
return 0;
}
```

Output:

```
deepti@Inspiron-3542:~$ gcc slr.c
deepti@Inspiron-3542:~$ ./a.out
Enter any String
a+a*a$
0 a+a*a$
0a5 +a*a$
0F3 +a*a$
0T2 +a*a$
0E1 +a*a$
0E1+6 a*a$
0E1+6a5 *a$
0E1+6F3 *a$
0E1+6T9 *a$
0E1+6T9*7 a$
0E1+6T9*7a5 $
0E1+6T9*7F10 $
0E1+6T9 $
0E1 $
Given String is accepted
```

PRACTICAL -10

AIM: Write a program in C to implement operator precedence parser

Procedure:

Operator precedence parser –

An operator precedence parser is a one of the bottom-up parser that interprets an operator-precedence grammar. This parser is only used for operator grammars. *Ambiguous grammars are not allowed* in case of any parser except operator precedence parser.

This parser relies on the following three precedence relations: ⋖ , ≐ , ⋗

a ⋖ **b** This means a "yields precedence to" b.

a ⋗ **b** This means a "takes precedence over" b.

a ≐ **b** This means a "has precedence as" b.

Code/Method:

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

char *input;
int i=0;
char lasthandle[6],stack[50],handles[][5]={")E(","E*E","E+E","i","E^E"};
//(E) becomes )E( when pushed to stack

int top=0,l;
char prec[9][9]={

                /*input*/

        /*stack   +   -   *   /   ^   i   (   )   $ */

        /*  + */  '>', '>','<','<','<','<','<','>','>',

        /*  - */  '>', '>','<','<','<','<','<','>','>',

        /*  * */  '>', '>','>','>','<','<','<','>','>',

        /*  / */  '>', '>','>','>','<','<','<','>','>',

        /*  ^ */  '>', '>','>','>','<','<','<','>','>',

        /*  i */  '>', '>','>','>','>','e','e','>','>',

        /*  ( */  '<', '<','<','<','<','<','<','>','e',

        /*  ) */  '>', '>','>','>','>','e','e','>','>',
```

```
        /*  $ */  '<', '<','<','<','<','<','<','<','>',

            };

int getindex(char c)
{
switch(c)
   {
   case '+':return 0;
   case '-':return 1;
   case '*':return 2;
   case '/':return 3;
   case '^':return 4;
   case 'i':return 5;
   case '(':return 6;
   case ')':return 7;
   case '$':return 8;
   }
return 0;
}

void shift()
{
stack[++top]=*(input+i++);
stack[top+1]='\0';
}


int reduce()
{
int i,len,found,t;
for(i=0;i<5;i++)//selecting handles
   {
   len=strlen(handles[i]);
   if(stack[top]==handles[i][0]&&top+1>=len)
      {
      found=1;
      for(t=0;t<len;t++)
         {
         if(stack[top-t]!=handles[i][t])
            {
            found=0;
            break;
            }
         }
      if(found==1)
         {
         stack[top-t+1]='E';
         top=top-t+1;
         strcpy(lasthandle,handles[i]);
         stack[top+1]='\0';
         return 1;//successful reduction
```

```
        }
      }
  }
return 0;
}



void dispstack()
{
int j;
for(j=0;j<=top;j++)
   printf("%c",stack[j]);
}



void dispinput()
{
int j;
for(j=i;j<l;j++)
   printf("%c",*(input+j));
}



void main()
{
int j;

input=(char*)malloc(50*sizeof(char));
printf("\nEnter the string\n");
scanf("%s",input);
input=strcat(input,"$");
l=strlen(input);
strcpy(stack,"$");
printf("\nSTACK\tINPUT\tACTION");
while(i<=l)
          {
          shift();
          printf("\n");
          dispstack();
          printf("\t");
          dispinput();
          printf("\tShift");
          if(prec[getindex(stack[top])][getindex(input[i])]=='>')
                    {
                    while(reduce())
                              {
                              printf("\n");
                              dispstack();
                              printf("\t");
```

```
                        dispinput();
                        printf("\tReduced: E->%s",lasthandle);
                        }
                }
        }

if(strcmp(stack,"$E$")==0)
   printf("\nAccepted;");
else
   printf("\nNot Accepted;");
getch();
}
```

Output:

```
A=A*A
B=AA
A=$

Not Accepted
Input :2
A=A/A
B=A+A

Accepted
```

PRACTICAL -11

C program to implement intermediate code generation for simple expression.

Code/Method:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int i=1,j=0,no=0,tmpch=90;
char str[100],left[15],right[15];
void findopr();
void explore();
void fleft(int);
void fright(int);
struct exp
{
int pos;
char op;
}k[15];
void main()
{
printf("\t\tINTERMEDIATE CODE GENERATION\n\n");
printf("Enter the Expression :");
scanf("%s",str);
printf("The intermediate code:\n");
findopr();
explore();
}
void findopr()
{
for(i=0;str[i]!='\0';i++)
if(str[i]==':')
{
k[j].pos=i;
k[j++].op=':';
}
for(i=0;str[i]!='\0';i++)
if(str[i]=='/')
{
k[j].pos=i;
k[j++].op='/';
}
for(i=0;str[i]!='\0';i++)
if(str[i]=='*')
{
k[j].pos=i;
k[j++].op='*';
```

```
}
for(i=0;str[i]!='\0';i++)
if(str[i]=='+')
{
k[j].pos=i;
k[j++].op='+';
}
for(i=0;str[i]!='\0';i++)
if(str[i]=='-')
{
k[j].pos=i;
k[j++].op='-';
}
}
void explore()
{
i=1;
while(k[i].op!='\0')
{
fleft(k[i].pos);
fright(k[i].pos);
str[k[i].pos]=tmpch--;
printf("\t%c := %s%c%s\t\t",str[k[i].pos],left,k[i].op,right);
printf("\n");
i++;
}
fright(-1);
if(no==0)
{
fleft(strlen(str));
printf("\t%s := %s",right,left);
getch();
exit(0);
}
printf("\t%s := %c",right,str[k[--i].pos]);
getch();
}
void fleft(int x)
{
int w=0,flag=0;
x--;
while(x!= -1 &&str[x]!= '+' &&str[x]!='*'&&str[x]!='='&&str[x]!='\0'&&str[x]!='-
'&&str[x]!='/'&&str[x]!=':')
{
if(str[x]!='$'&& flag==0)
{
left[w++]=str[x];
left[w]='\0';
```

```
str[x]='$';
flag=1;
}
x--;
}
}
void fright(int x)
{
int w=0,flag=0;
x++;
while(x!= -1 && str[x]!= '+'&&str[x]!='*'&&str[x]!='\0'&&str[x]!='='&&str[x]!=':'&&str[x]!='-'&&str[x]!='/')
{
if(str[x]!='$'&& flag==0)
{
right[w++]=str[x];
right[w]='\0';
str[x]='$';
flag=1;
}
x++;
}
}
```

Output: