ASSIGNMENT-6

1.Consider the following grammar rules: E->E+T|T, T->T*F|F, F->(E)|id
where E, T and F are non-terminals and +, *, (,), id are terminals (tokens).
Design a shift-reduce parser for it.

Solution:

```
#include<bits/stdc++.h>

using namespace std;
struct grammer{
   char p[20];
   char prod[20];
}g[10];

int main()
{

   cout<<"\t\t\t SHIFT REDUCE PARSER\t\t\t\n";
   int i,stpos,j,k,l,m,o,p,f,r;
   int np,tspos,cr;

   cout<<"\nEnter Number of productions:";
   cin>>np;

   char sc,ts[10];

   cout<<"\nEnter productions:\n";
   for(i=0;i<np;i++)
   {
```

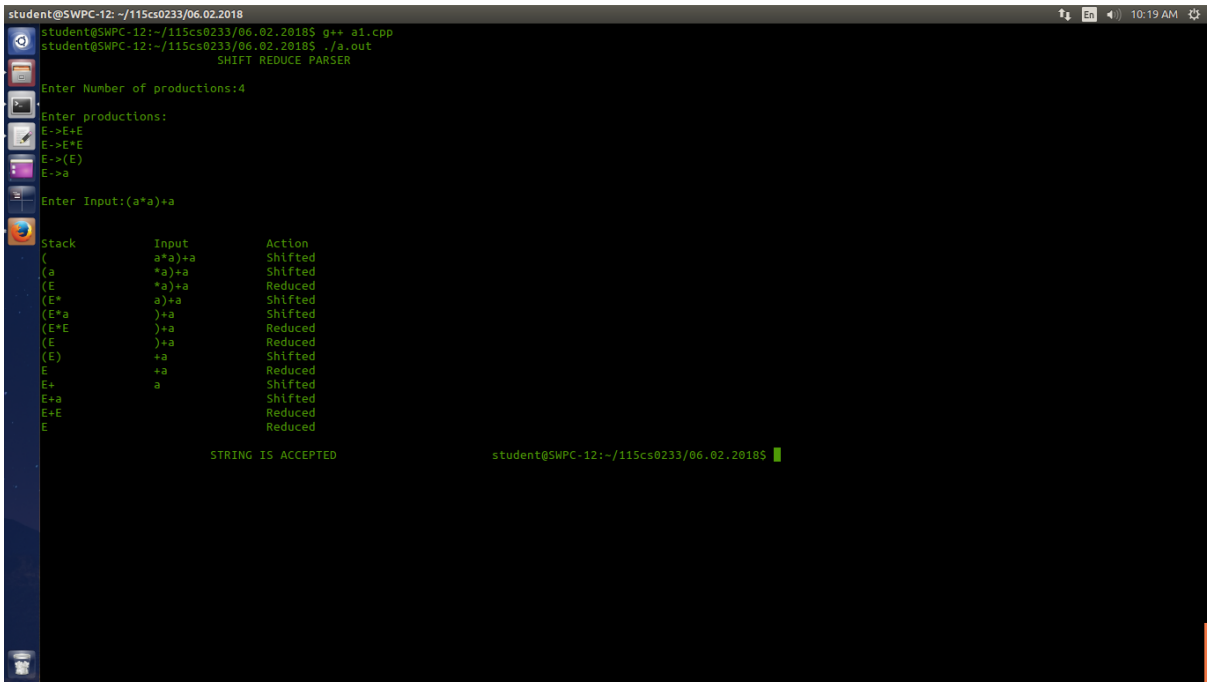
```
    cin>>ts;
    strncpy(g[i].p,ts,1);
    strcpy(g[i].prod,&ts[3]);
}

char ip[10];

cout<<"\nEnter Input:";
cin>>ip;

int lip=strlen(ip);

char stack[10];

stpos=0;
i=0;

//moving input
sc=ip[i];
stack[stpos]=sc;
i++;stpos++;

cout<<"\n\nStack\t\tInput\t\tAction";
do
{
    r=1;
    while(r!=0)
    {
        cout<<"\n";
        for(p=0;p<stpos;p++)
        {
```

```
    cout<<stack[p];
}
cout<<"\t\t";
for(p=i;p<lip;p++)
{
    cout<<ip[p];
}

if(r==2)
{
    cout<<"\t\tReduced";
}
else
{
    cout<<"\t\tShifted";
}
r=0;

//try reducing

for(k=0;k<stpos;k++)
{
    f=0;

    for(l=0;l<10;l++)
    {
        ts[l]='\0';
    }

    tspos=0;
    for(l=k;l<stpos;l++) //removing first caharcter
```

```
        {
          ts[tspos]=stack[l];
          tspos++;
        }

        //now compare each possibility with production
        for(m=0;m<np;m++)
        {
          cr = strcmp(ts,g[m].prod);

          //if cr is zero then match is found
          if(cr==0)
          {
            for(l=k;l<10;l++) //removing matched part from stack
            {
              stack[l]='\0';
              stpos--;
            }

            stpos=k;

            //concatinate the string
            strcat(stack,g[m].p);
            stpos++;
            r=2;
          }
        }
      }
    }

    //moving input
```

```
    sc=ip[i];
    stack[stpos]=sc;
    i++;stpos++;

  }while(strlen(stack)!=1 && stpos!=lip);

  if(strlen(stack)==1)
  {
    cout<<"\n\n \t\t\tSTRING IS ACCEPTED\t\t\t";
  }
  else
      cout<<"\n\n \t\t\tSTRING IS REJECTED\t\t\t";

  return 0;
}
```

Output:



2. Design an operator precedence parser with the help of the following operator-precedence Table:

	+	*	(	)	id	$
+	>	<	<	>	<	>
*	>	>	<	>	<	>
(	<	<	<	=	<	-
)	>	>	-	>	-	>
id	>	>	-	>	-	>
$	<	<	<	-	<	-

Solution:

```
#include<stdio.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;
   }
}
```

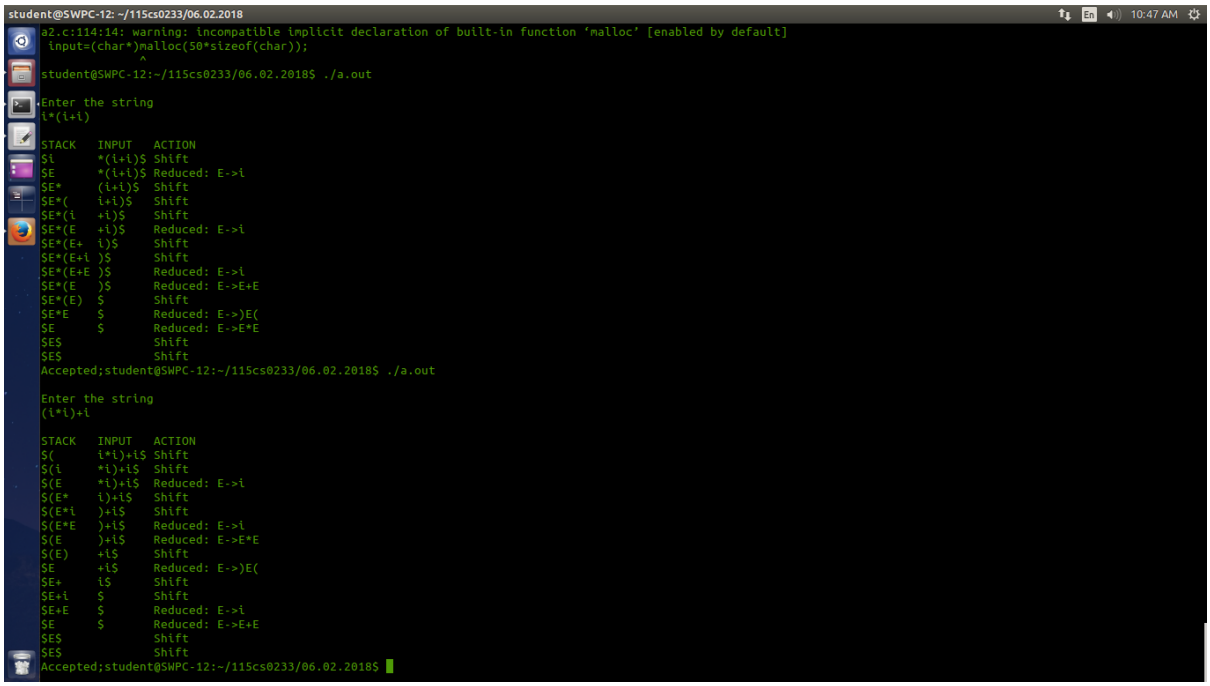
```
int shift()
{
stack[++top]=*(input+i++);
stack[top+1]='\0';
}


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;");
}
```

Output:



ASSIGNMENT-7

1. Write a C program to eliminate left recursion and left factoring in a given grammar.

Solution:

To eliminate left factoring:

```
#include <stdio.h>
#include <string.h>
int main() {
int SIZE = 10;
char non_terminal;
char beta, alpha;
int num;
int i;
char production[10][SIZE];
int index = 3;
printf("\nEnter Number of Production : ");
scanf("%d", &num);
printf("Enter the grammar:\n");
for(i = 0; i < num; i++)
scanf("%s", production[i]);
for(i = 0; i < num; i++) {
printf("\nGRAMMAR : : : %s", production[i]);
non_terminal = production[i][0];
if(non_terminal == production[i][index]) {
alpha = production[i][index+1];
printf(" is left recursive.\n");
while(production[i][index] != 0 && production[i][index] != '|')
index++;
if(production[i][index] != 0) {
```

```
beta = production[i][index+1];
printf("Grammar without left recursion:\n");
printf("%c->%c%c\'", non_terminal, beta, non_terminal);
printf("\n%c\'->%c%c\'|^\n", non_terminal, alpha, non_terminal);
}
else
printf(" can't be reduced\n");
}
else
printf(" is not left recursive.\n");
index = 3;
}return 0;
}
```

To eliminate left recursion:

```
#include <bits/stdc++.h>
using namespace std;

struct production
{
        char lf;
        char rt[10];
        int prod_rear;
        int fl;
};
struct production prodn[20],prodn_new[20];    //Creation of object

//Variables Declaration for left factoring

int b=-1,d,q,f,n,m=0,c=0;
char terminal[20],nonterm[20],alpha[10],extra[10];
```

```
char epsilon='^';

void left_fact(vector<string > &prod, int n)
{
  int cnt, cnt3;
  char lp;
  string d = "->";
  //Input of Productions
  cout<<"Note: Here ^ denotes epsillon\n";
  for(cnt=0;cnt<=n-1;cnt++)
  {
    int pos = prod[cnt].find(d);
    lp = prod[cnt][0];
    prodn[cnt].lf = lp;
    for(int i=pos+2; i<=prod[cnt].length()-1; i++)
    {
          prodn[cnt].rt[i-(pos+2)] = prod[cnt][i];
    }
    prodn[cnt].prod_rear=strlen(prodn[cnt].rt);
    prodn[cnt].fl=0;
  }

  //Condition for left factoring
  int cnt1 = 0, cnt2 = 0;
  for(cnt1=0;cnt1<n;cnt1++)
  {
    for(cnt2=cnt1+1;cnt2<n;cnt2++)
    {
        if(prodn[cnt1].lf==prodn[cnt2].lf)
        {
          cnt=0;
```

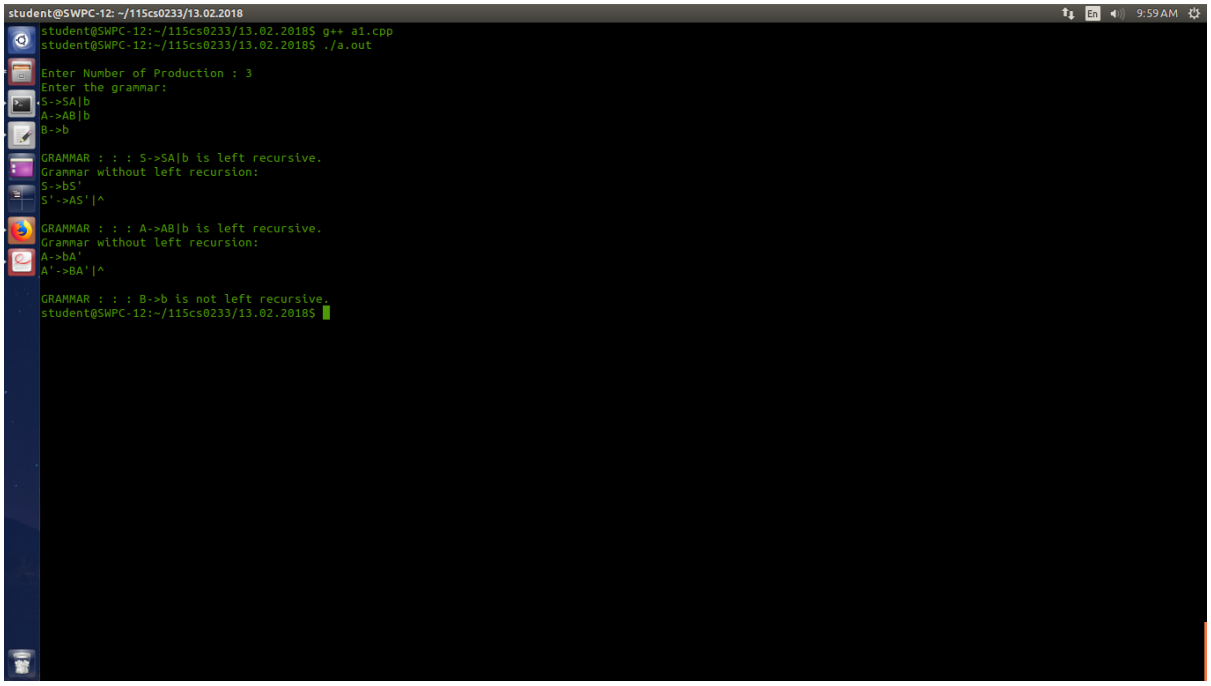
```
int p=-1;
while((prodn[cnt1].rt[cnt]!='\0')&&(prodn[cnt2].rt[cnt]!='\0'))
{
 if(prodn[cnt1].rt[cnt]==prodn[cnt2].rt[cnt])
 {
  extra[++p]=prodn[cnt1].rt[cnt];
  prodn[cnt1].fl=1;
  prodn[cnt2].fl=1;
 }
 else
 {
  if(p==-1)
        break;
  else
  {
        int h=0,u=0;
        prodn_new[++b].lf=prodn[cnt1].lf;
        strcpy(prodn_new[b].rt,extra);
        //prodn_new[b].rt[p+1]=alpha[c];
        prodn_new[b].rt[p+1]=prodn[cnt1].lf;
        //prodn_new[++b].lf=alpha[c];
        prodn_new[++b].lf=prodn[cnt1].lf;
        for(q=cnt;q<prodn[cnt2].prod_rear;q++)
         prodn_new[b].rt[h++]=prodn[cnt2].rt[q];
         //prodn_new[++b].lf=alpha[c];
         prodn_new[++b].lf=prodn[cnt1].lf;
        for(q=cnt;q<=prodn[cnt1].prod_rear;q++)
         prodn_new[b].rt[u++]=prodn[cnt1].rt[q];
         m=1;
         break;
   }
```

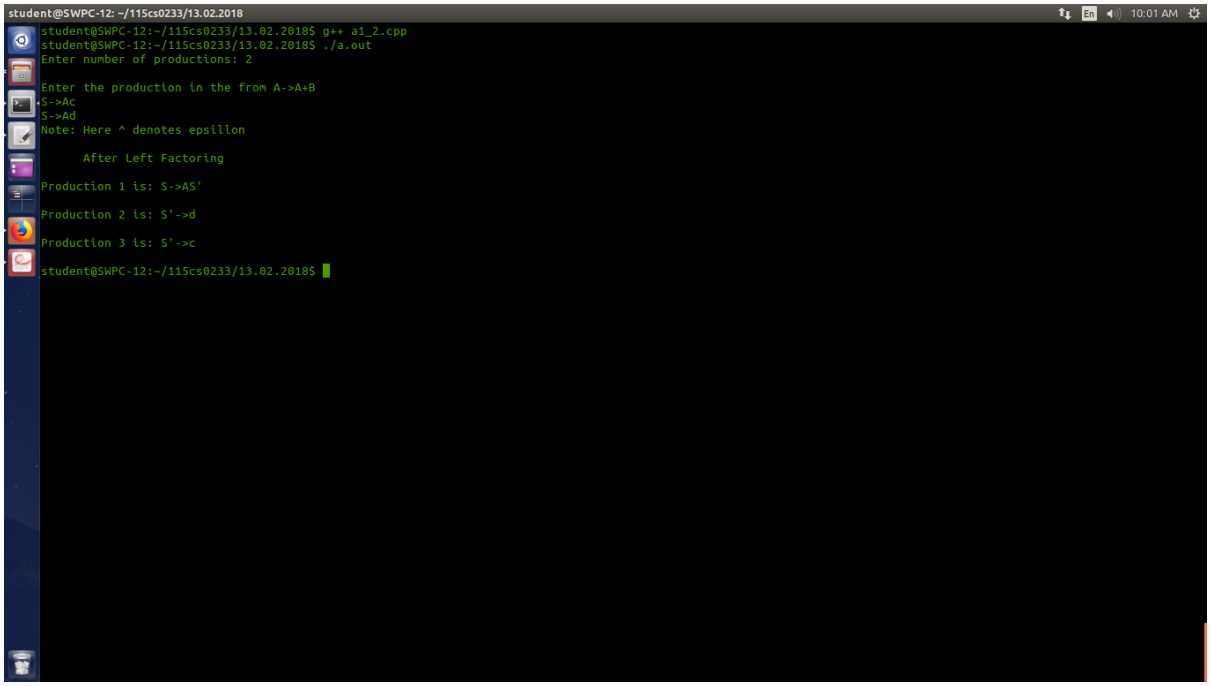
```
  }
  cnt++;
}
if((prodn[cnt1].rt[cnt]==0)&&(m==0))
{
        int h=0;
        prodn_new[++b].lf=prodn[cnt1].lf;
        strcpy(prodn_new[b].rt,extra);
        //prodn_new[b].rt[p+1]=alpha[c];
        prodn_new[b].rt[p+1]=prodn[cnt1].lf;
        //prodn_new[++b].lf=alpha[c];
        prodn_new[++b].lf=prodn[cnt1].lf;
        prodn_new[b].rt[0]=epsilon;
        //prodn_new[++b].lf=alpha[c];
        prodn_new[++b].lf=prodn[cnt1].lf;
        for(q=cnt;q<prodn[cnt2].prod_rear;q++)
        prodn_new[b].rt[h++]=prodn[cnt2].rt[q];
}
if((prodn[cnt2].rt[cnt]==0)&&(m==0))
{
  int h=0;
  prodn_new[++b].lf=prodn[cnt1].lf;
  strcpy(prodn_new[b].rt,extra);
  //prodn_new[b].rt[p+1]=alpha[c];
  prodn_new[b].rt[p+1]=prodn[cnt1].lf;
  //prodn_new[++b].lf=alpha[c];
  prodn_new[++b].lf=prodn[cnt1].lf;
  prodn_new[b].rt[0]=epsilon;
  //prodn_new[++b].lf=alpha[c];
  prodn_new[++b].lf=prodn[cnt1].lf;
  for(q=cnt;q<prodn[cnt1].prod_rear;q++)
```

```
            prodn_new[b].rt[h++]=prodn[cnt1].rt[q];
          }
          c++;
          m=0;
        }
    }
}

//Display of Output

 cout<<"\n      After Left Factoring       \n";
 cout<<endl;
          for(cnt3=0;cnt3<=0;cnt3++)
        {
                cout<<"Production "<<cnt3+1<<" is: ";
                cout<<prodn_new[cnt3].lf;
                cout<<"->";
                cout<<prodn_new[cnt3].rt<<"'";
                cout<<endl<<endl;
        }
 for(cnt3=1;cnt3<=b;cnt3++)
        {
                cout<<"Production "<<cnt3+1<<" is: ";
                cout<<prodn_new[cnt3].lf<<"'";
                cout<<"->";
                cout<<prodn_new[cnt3].rt;
                cout<<endl<<endl;
        }
          cnt3 = b+2;
 for(int cnt4=0;cnt4<n;cnt4++)
 {
```

```
    if(prodn[cnt4].fl==0)
    {
    cout<<"Production  "<<cnt3<<" is: ";
    cout<<prodn[cnt4].lf;
    cout<<"->";
    cout<<prodn[cnt4].rt;
    cout<<endl<<endl;
    cnt3++;
    }
    }
}
int main()
{
        int n;
        cout<<"Enter number of productions: ";
        cin>>n;
        vector<string > prod(n, "0");
        cout<<"\nEnter the production in the from A->A+B\n";
        int i;
        for(i=0; i<n; i++)
        {
                cin>>prod[i];
        }
        //remove_left_recur(prod, n);
        left_fact(prod, n);
        return 0;
}
```

Output:


student@SWPC-12: ~/115cs0233/13.02.2018
9:59 AM
student@SWPC-12:~/115cs0233/13.02.2018$ g++ a1.cpp
student@SWPC-12:~/115cs0233/13.02.2018$ ./a.out
Enter Number of Production : 3
Enter the grammar:
S->SA|b
A->AB|b
B->b
GRAMMAR : : : S->SA|b is left recursive.
Grammar without left recursion:
S->bS'
S'->AS'|^
GRAMMAR : : : A->AB|b is left recursive.
Grammar without left recursion:
A->bA'
A'->BA'|^
GRAMMAR : : : B->b is not left recursive.
student@SWPC-12:~/115cs0233/13.02.2018$


student@SWPC-12: ~/115cs0233/13.02.2018
10:01 AM
student@SWPC-12:~/115cs0233/13.02.2018$ g++ a1_2.cpp
student@SWPC-12:~/115cs0233/13.02.2018$ ./a.out
Enter number of productions: 2
Enter the production in the from A->A+B
S->Ac
S->Ad
Note: Here ^ denotes epsillon
After Left Factoring
Production 1 is: S->AS'
Production 2 is: S'->d
Production 3 is: S'->c
student@SWPC-12:~/115cs0233/13.02.2018$

2. Write a C program to compute FIRST and FOLLOW sets for a given grammar, and check whether the grammar is LL(1).

Solution:

```
#include "ctype.h"
#include "string.h"
#include "stdio.h"
char gram[10][10];
char vFirst[10];
char nonT[10];
char vFollow[10];
int m = 0;
int p;
int i = 0;
int j = 0;
int elem[10];
int size;
int fPt;
int k = 0;
int getGram() {
char ch;
int i;
int j;
int k;
printf("\nEnter Number of Rule : ");
scanf("%d", &size);
printf("\nEnter Grammar as E=E+B \n");
for(i = 0; i < size; i++){
scanf("%s%c", gram[i], &ch);
elem[i] = strlen(gram[i]);
}
```

```
}
int funcFirst(char victim){
int j;
int i;
if(!(isupper(victim)))
vFirst[k++] = victim;
else {
for(j = 0; j < size; j++) {if(gram[j][0] == victim) {
if(gram[j][2] == '$')
vFirst[k++] = '$';
else if(islower(gram[j][2]))
vFirst[k++] = gram[j][2];
else
funcFirst(gram[j][2]);
}
}
}
}
void funFollow(char);
void first(char victim) {
int k;
if(!(isupper(victim)))
vFollow[m++] = victim;
for(k = 0; k < size; k++) {
if(gram[k][0] == victim) {
if(gram[k][2] == '$')
funFollow(gram[i][0]);
else if(islower(gram[k][2]))
vFollow[m++] = gram[k][2];
else
first(gram[k][2]);
```

```
}
}
}
void funFollow(char victim) {
if(gram[0][0] == victim)
vFollow[m++] = '$';
for(i = 0; i < size; i++) {
for(j = 2; j < strlen(gram[i]); j++) {
if(gram[i][j] == victim) {
if(gram[i][j+1] != '\0')
first(gram[i][j+1]);if(gram[i][j+1] == '\0' && victim != gram[i][0])
funFollow(gram[i][0]);
}
}
}
}
int main() {
int i;
int j;
int l = 0;
getGram();
for(i = 0; i < size; i++) {
for (j = 0; j < i; j++) {
if (gram[i][0] == gram[j][0])
break;
}
if (i == j) {
nonT[l] = gram[i][0];
l++;
}
}
```

```
for(i = 0; i < l; i++) {
k = 0;
funcFirst(nonT[i]);
printf("\nFIRST(%c){ ", nonT[i]);
for(j = 0; j < strlen(vFirst); j++)
printf(" %c", vFirst[j]);
printf(" }\n");
}
int s = 0;
for(s = 0; s < l; s++) {
m = 0;
printf("\nFOLLOW(%c){ ", nonT[s]);
funFollow(nonT[s]);
for(i = 0; i < m; i++)
printf("%c ", vFollow[i]);
printf(" }\n");
}
}
```

Output: