INDEX

EXPERIMENT 1

AIM : Write a program for the implementation of Lexical Analyser.

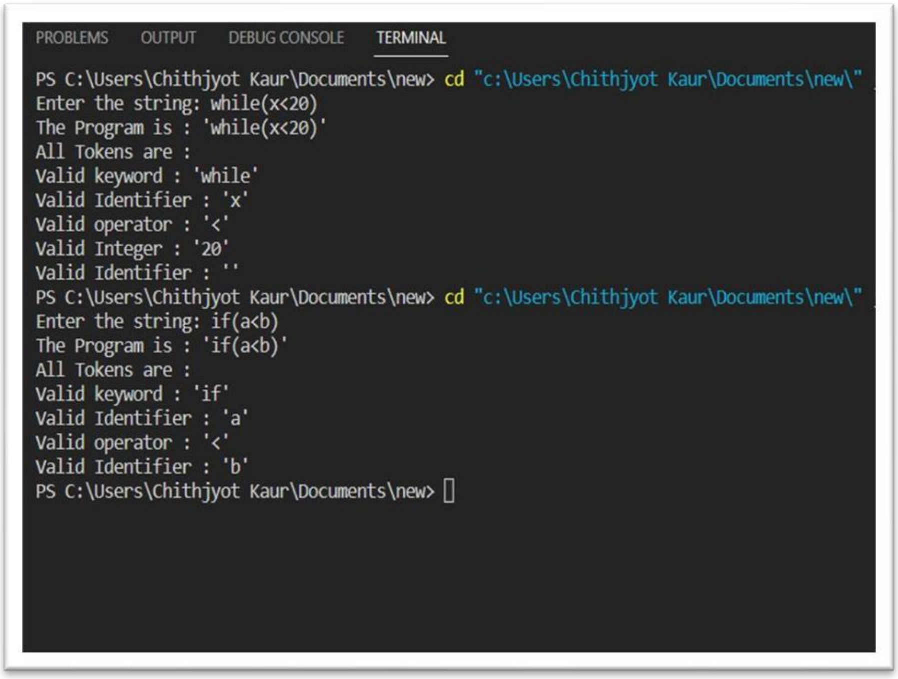
SOURCE CODE FOR LEXICAL ANALYSER:

```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
bool isValidDelimiter(char ch) {
  if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
  ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
  ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
  ch == '[' || ch == ']' || ch == '{' || ch == '}')
  return (true);
  return (false);
}
bool isValidOperator(char ch){
  if (ch == '+' || ch == '-' || ch == '*' ||
  ch == '/' || ch == '>' || ch == '<' ||
  ch == '=')
  return (true);
  return (false);
}
bool isvalidIdentifier(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' || isValidDelimiter(str[0]) == true)
  return (false);
  return (true);
}
```

```
bool isValidKeyword(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);
}
bool isValidInteger(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);
}
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);
}
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);
}
void detectTokens(char* str) {
  int left = 0, right = 0;
  int length = strlen(str);
  while (right <= length && left <= right) {
    if (isValidDelimiter(str[right]) == false)
    right++;
    if (isValidDelimiter(str[right]) == true && left == right) {
      if (isValidOperator(str[right]) == true)
      printf("Valid operator : '%c' \n", str[right]);
      right++;
      left = right;
    }
else if (isValidDelimiter(str[right]) == true && left != right || (right == length && left !=       right)) {
      char* subStr = subString(str, left, right - 1);
      if (isValidKeyword(subStr) == true)
        printf("Valid keyword : '%s' \n", subStr);
      else if (isValidInteger(subStr) == true)
        printf("Valid Integer : '%s' \n", subStr);
      else if (isRealNumber(subStr) == true)
        printf("Real Number : '%s' \n", subStr);
      else if (isvalidIdentifier(subStr) == true
          && isValidDelimiter(str[right - 1]) == false)
        printf("Valid Identifier : '%s' \n", subStr);
        else if (isvalidIdentifier(subStr) == false
          && isValidDelimiter(str[right - 1]) == false)
        printf("Invalid Identifier : '%s' \n", subStr);
        left = right;
      } }
  return;
}
int main(){
  char str[100];
  printf("Enter the string: ");
  scanf("%s", str);
  printf("The Program is : '%s' ", str);
  printf("\n");
  printf("All Tokens are : \n");
  detectTokens(str);
  return (0);
}
```

OUTPUT:

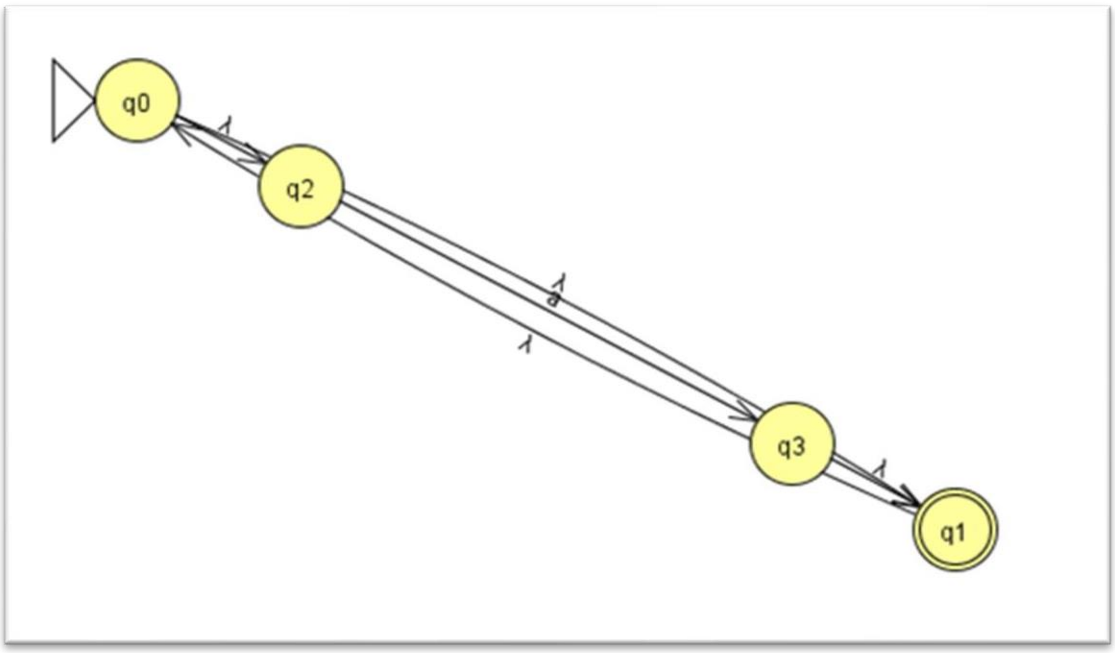


```
PS C:\Users\Chithjyot Kaur\Documents\new> cd "c:\Users\Chithjyot Kaur\Documents\new\"
Enter the string: while(x<20)
The Program is : 'while(x<20)'
All Tokens are :
Valid keyword : 'while'
Valid Identifier : 'x'
Valid operator : '<'
Valid Integer : '20'
Valid Identifier : ''
PS C:\Users\Chithjyot Kaur\Documents\new> cd "c:\Users\Chithjyot Kaur\Documents\new\"
Enter the string: if(a<b)
The Program is : 'if(a<b)'
All Tokens are :
Valid keyword : 'if'
Valid Identifier : 'a'
Valid operator : '<'
Valid Identifier : 'b'
PS C:\Users\Chithjyot Kaur\Documents\new>
```

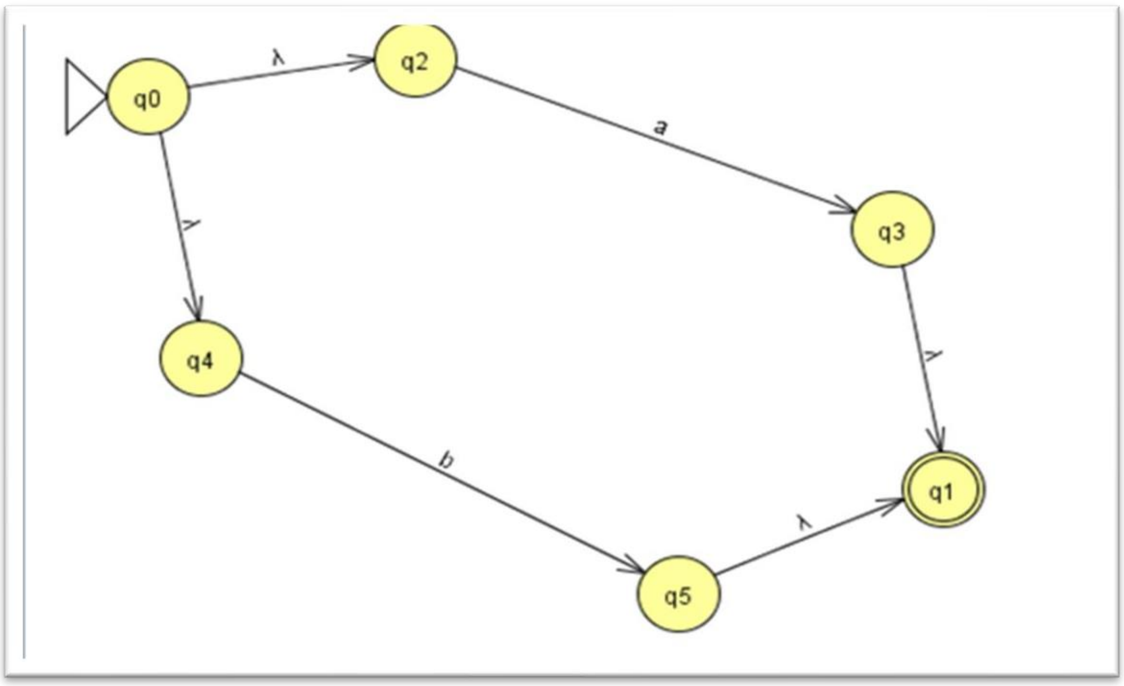
EXPERIMENT 2

AIM : Conversion from Regular Expression to NFA.

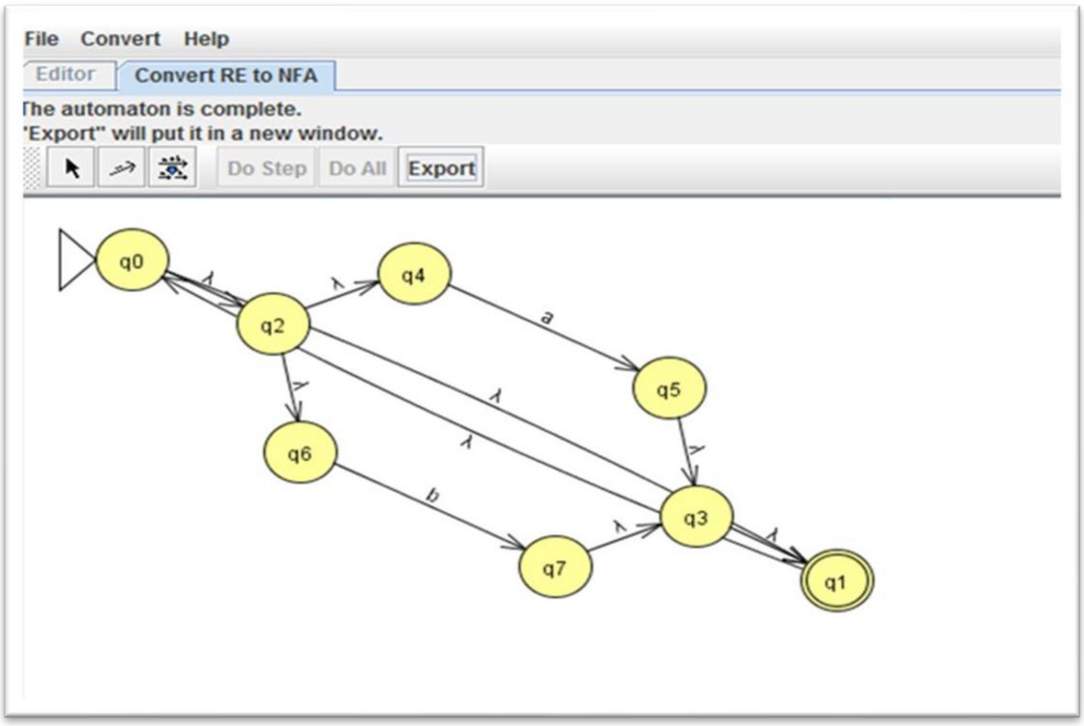
1) a*



2) (a+b)

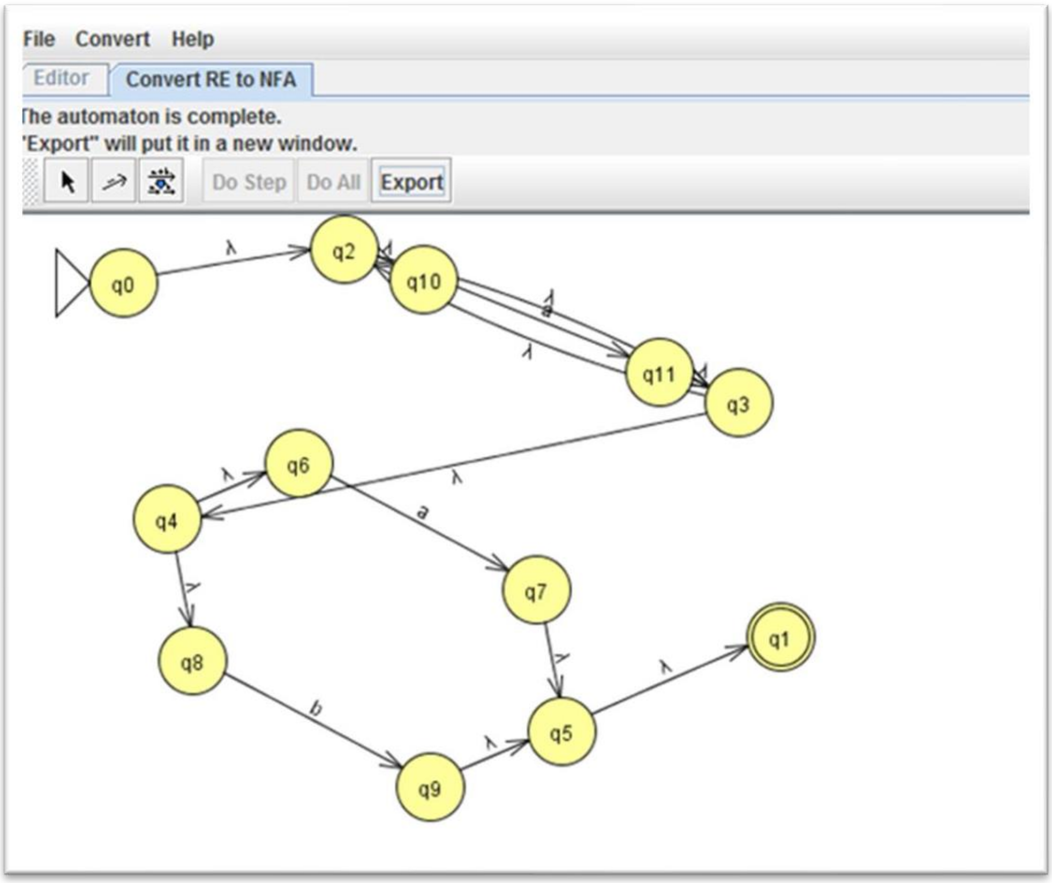


3) (a+b)*



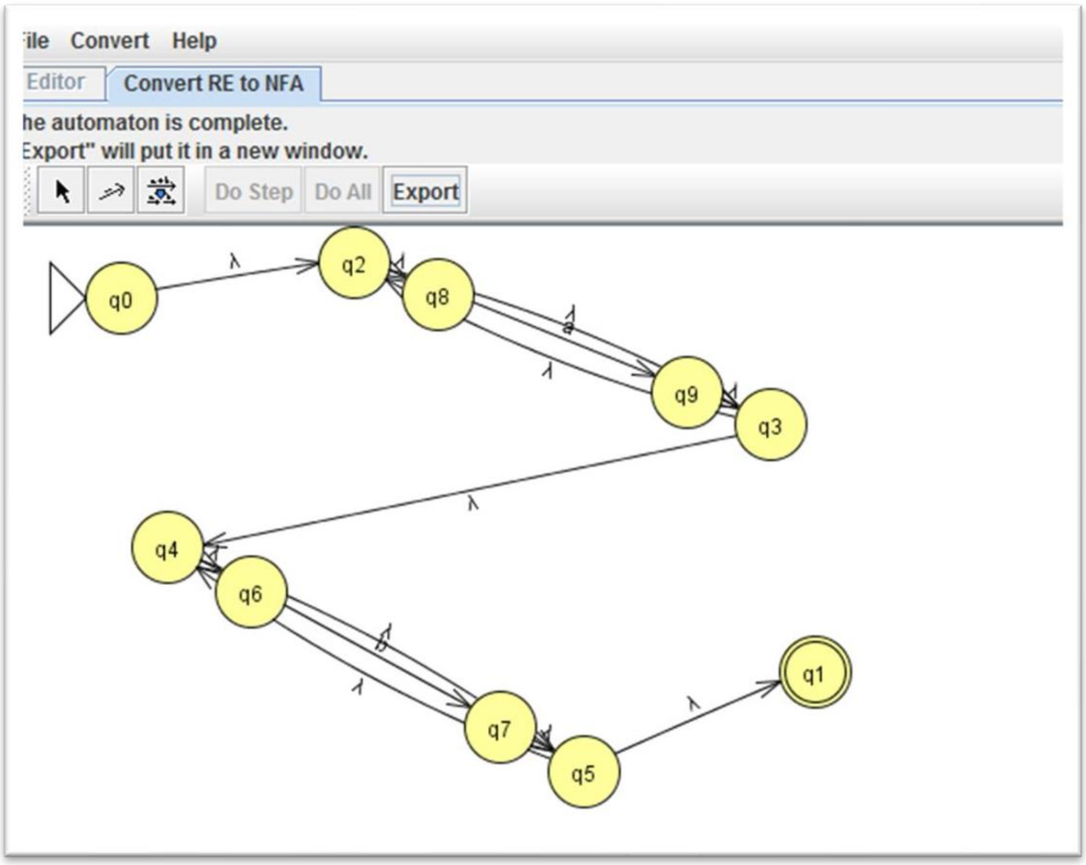
The automaton is complete.
"Export" will put it in a new window.

4) a*(a+b)



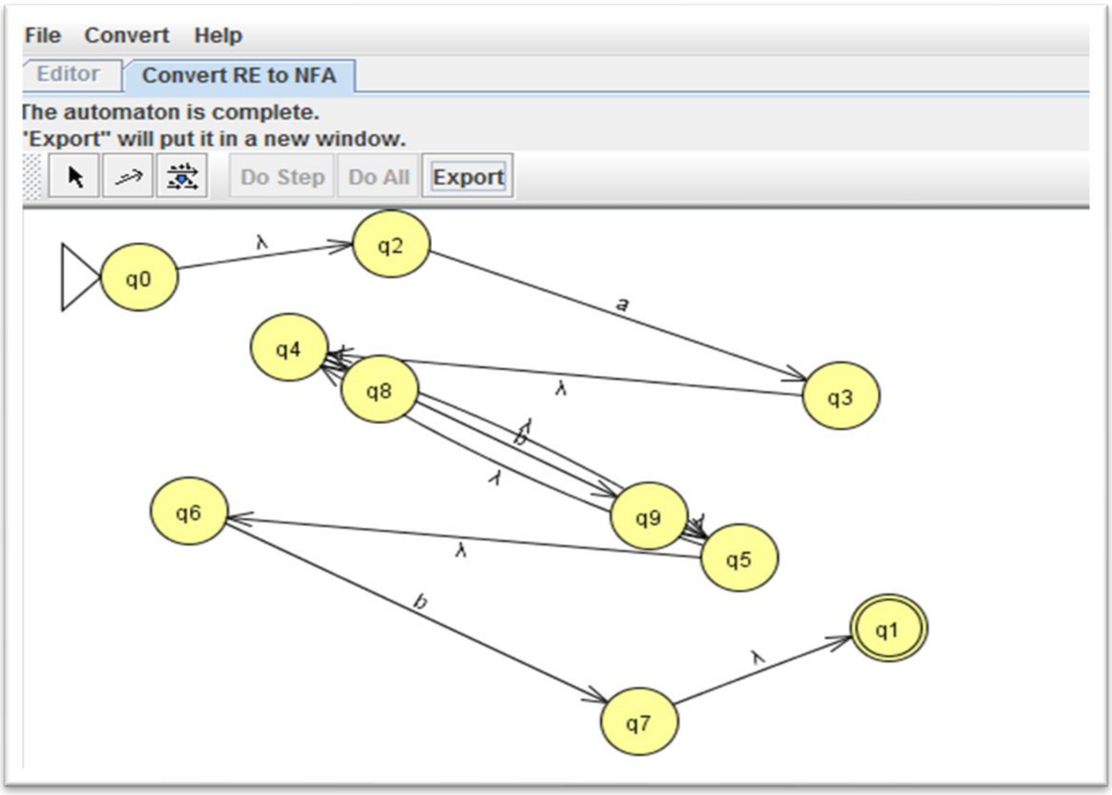
The automaton is complete.
"Export" will put it in a new window.

5) a*b*



he automaton is complete.
Export" will put it in a new window.

6) ab*b

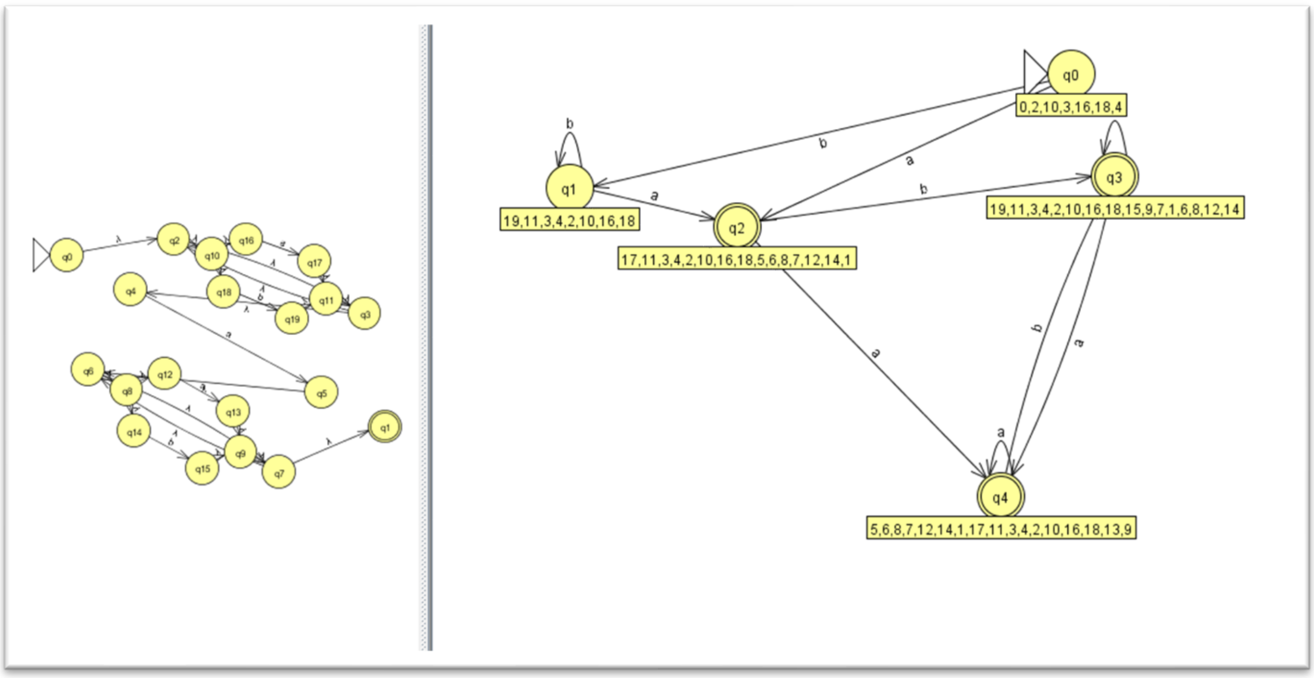


The automaton is complete.
'Export" will put it in a new window.

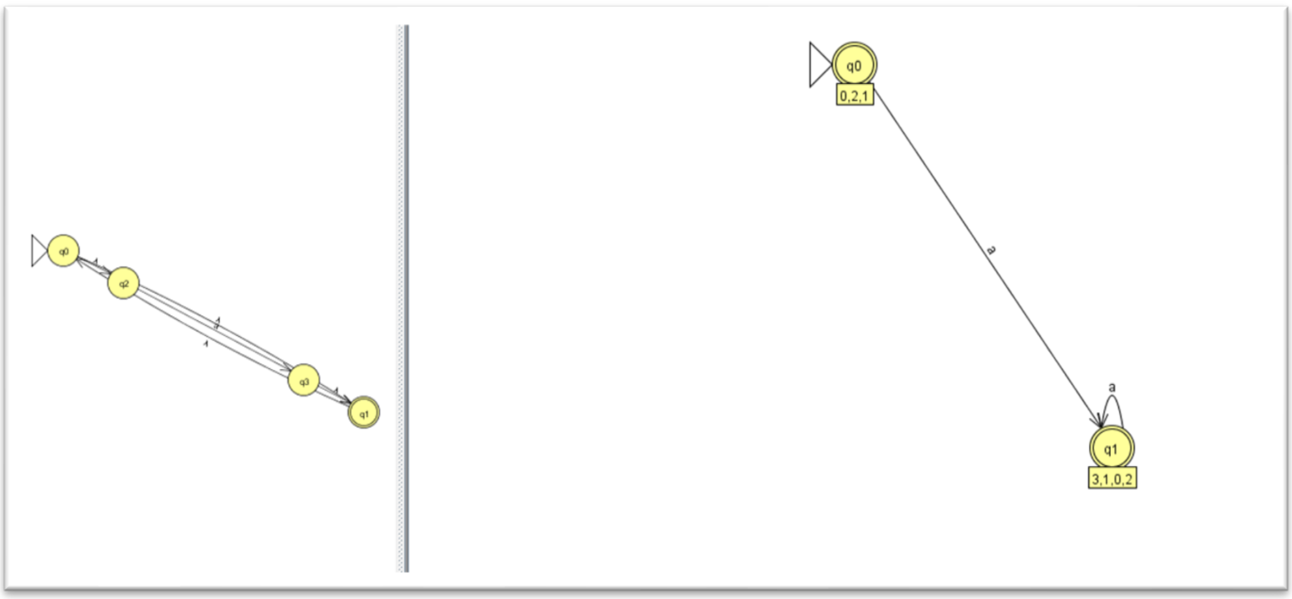
EXPERIMENT 3

AIM : Conversion from NFA to DFA.

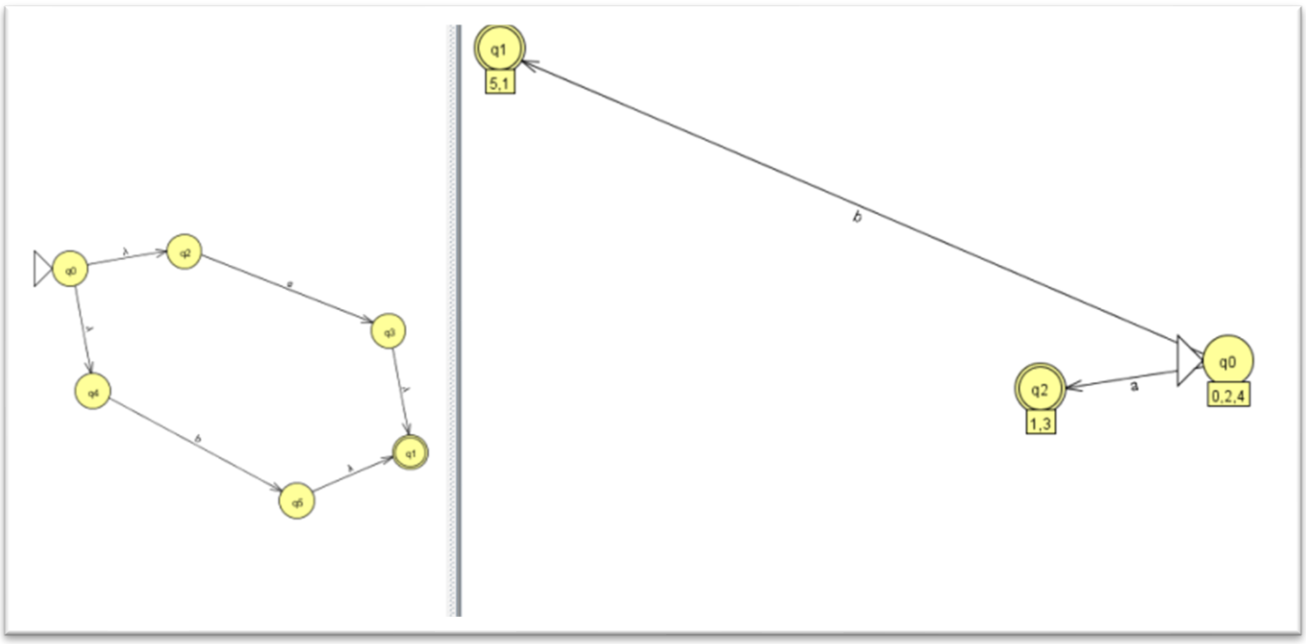
1) (a+b)* a(a+b)*



2) a*

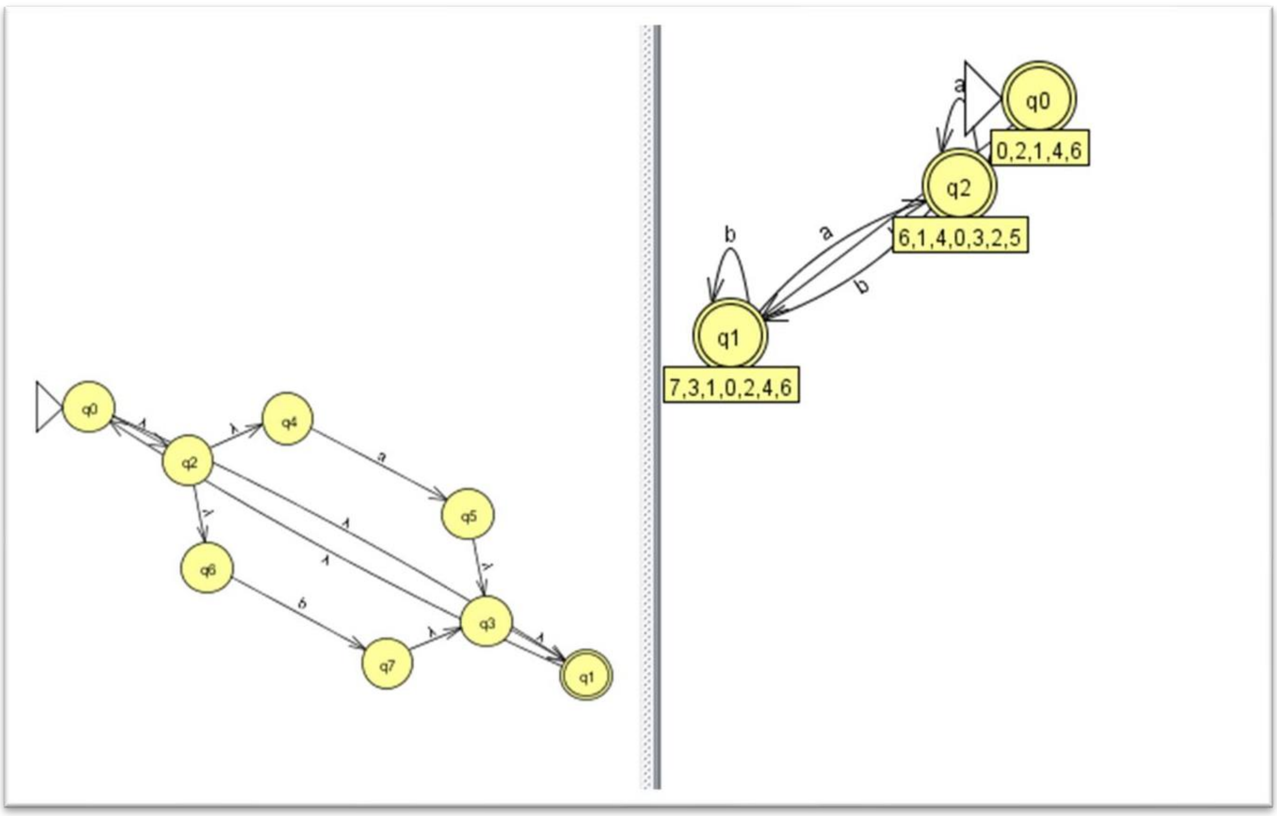


3) (a+b)



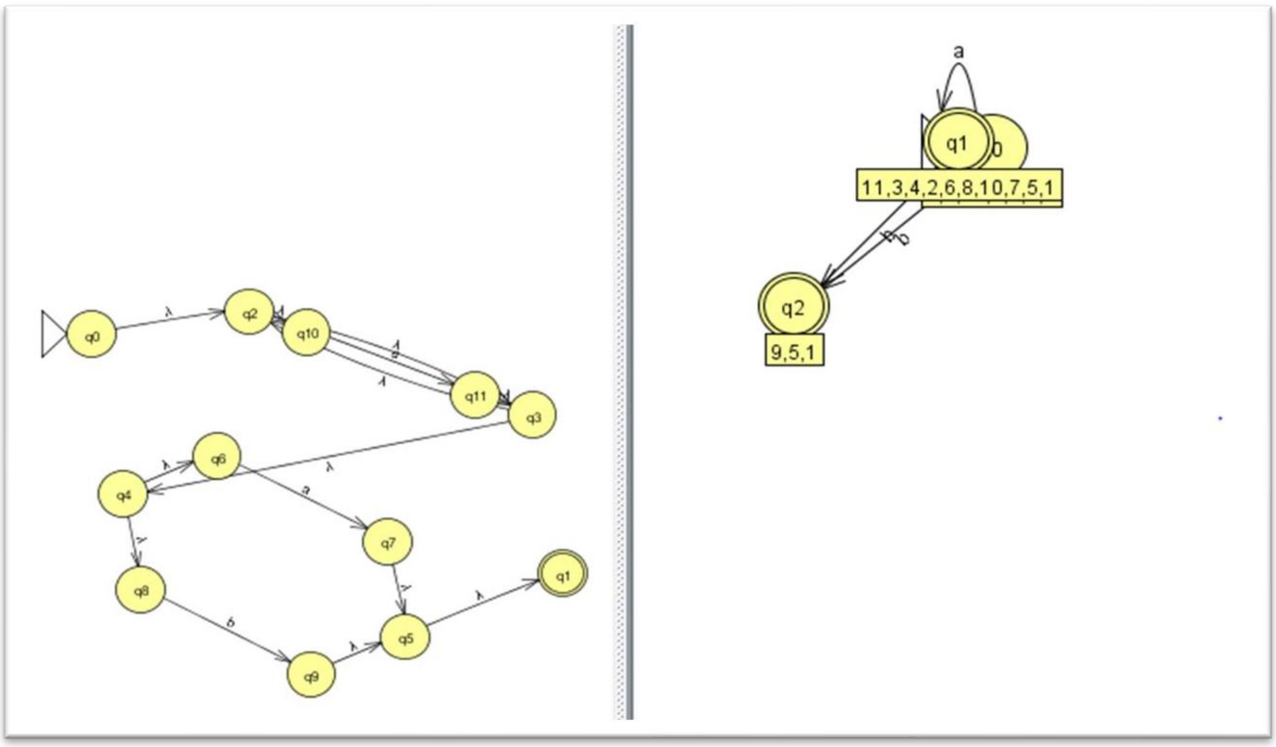
q0
q2
q3
q4
q5
q1
λ
a
b
q1
5,1
b
q2
1,3
a
q0
0,2,4

4) (a+b)*

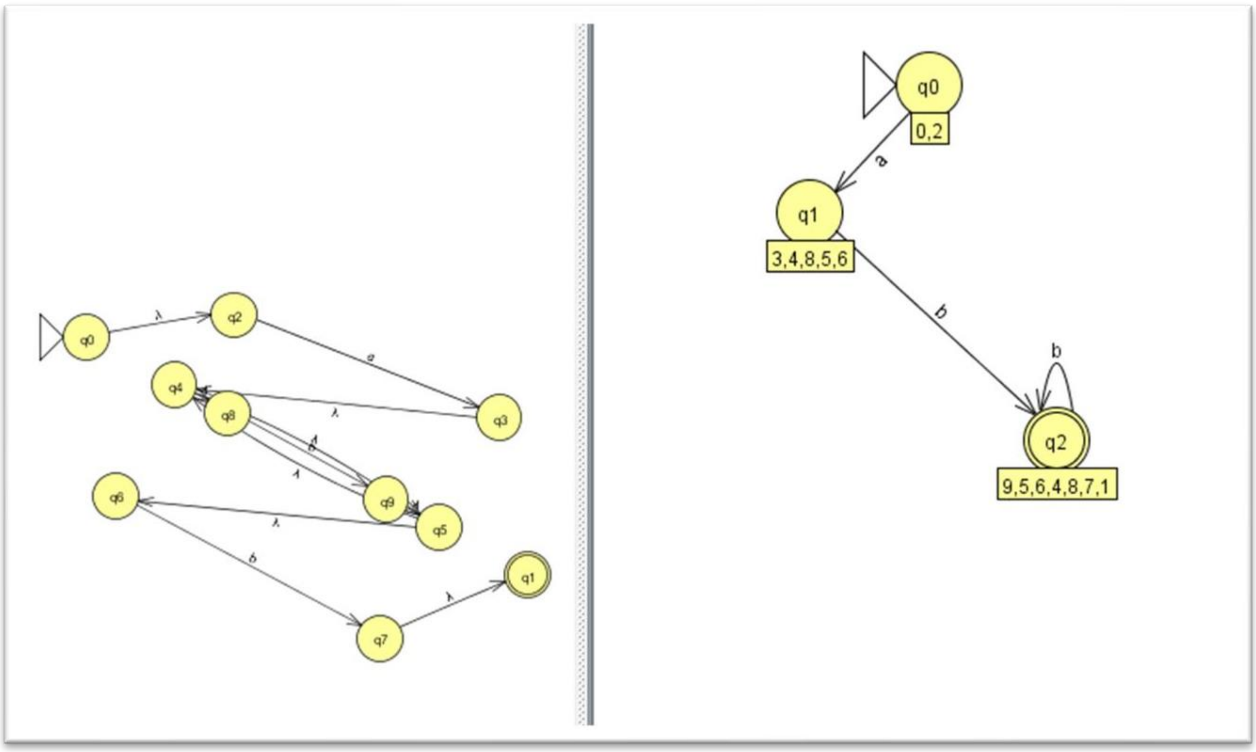


q0
q2
q4
q5
q6
q7
q3
q1
λ
a
b
a
q0
0,2,1,4,6
q2
6,1,4,0,3,2,5
b
a
b
q1
7,3,1,0,2,4,6

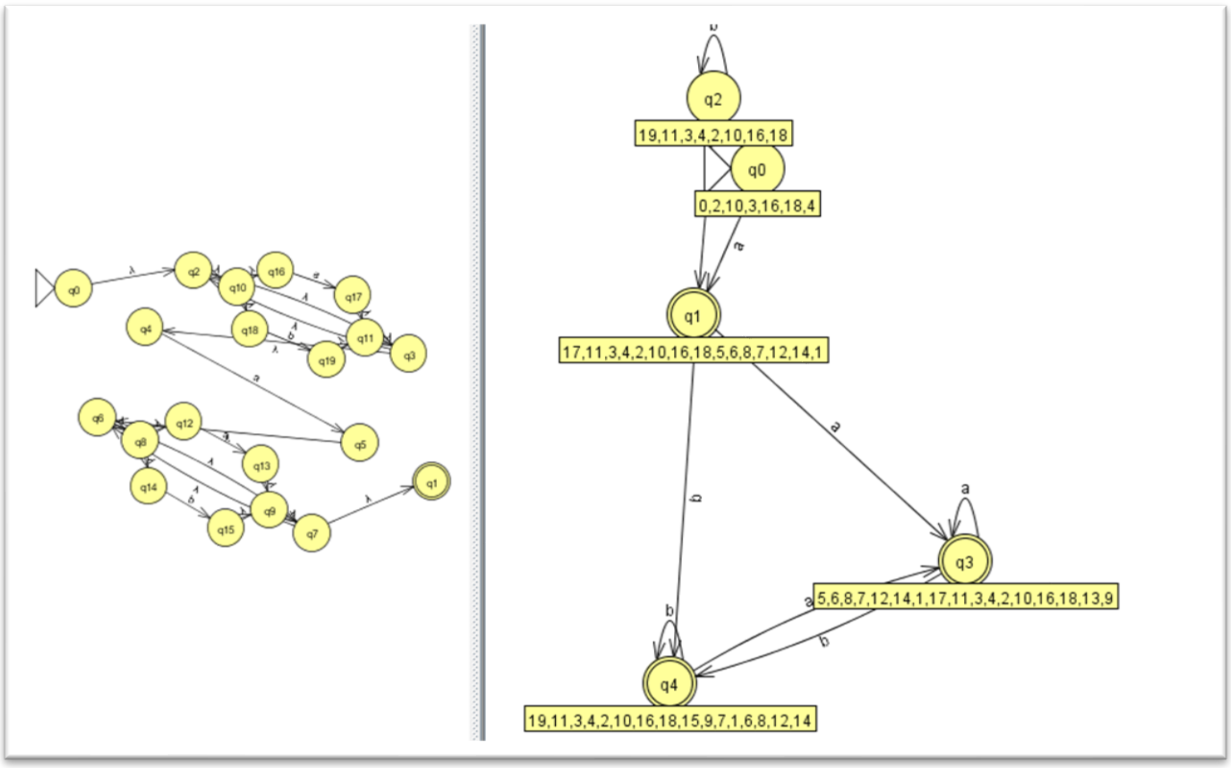
5) a*(a+b)


q0
q2
q10
q11
q3
q4
q6
q7
q8
q9
q5
q1
λ
a
b
q1
0
11,3,4,2,6,8,10,7,5,1
q2
9,5,1

6) a*b*


q0
q2
q3
q4
q8
q9
q5
q6
q7
q1
λ
a
b
q0
0,2
q1
3,4,8,5,6
q2
9,5,6,4,8,7,1

7) ab*b


q0
q2
q10
q16
q17
q4
q18
q11
q19
q3
q6
q12
q8
q13
q5
q14
q9
q15
q7
q1
λ
a
b
q2
19,11,3,4,2,10,16,18
q0
0,2,10,3,16,18,4
q1
17,11,3,4,2,10,16,18,5,6,8,7,12,14,1
q3
5,6,8,7,12,14,1,17,11,3,4,2,10,16,18,13,9
q4
19,11,3,4,2,10,16,18,15,9,7,1,6,8,12,14

EXPERIMENT 4

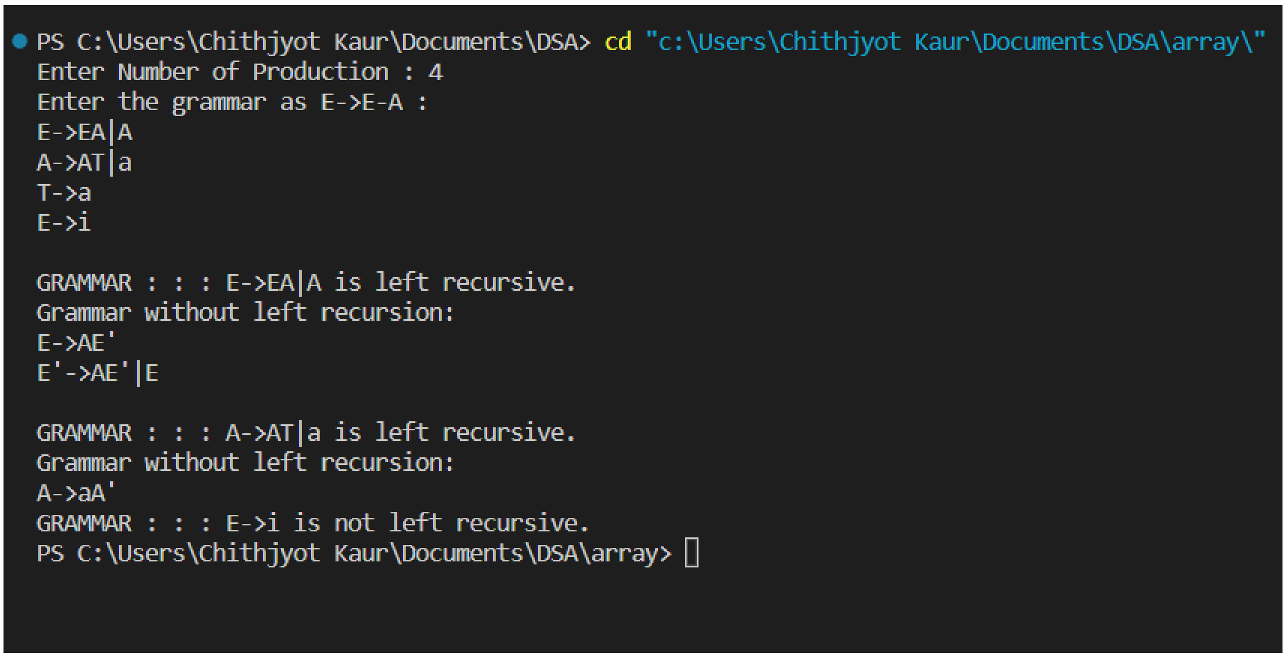
AIM : Write a program to remove left recursion and left factoring.

Left Recursion Removal -

```
#include<stdio.h>
#include<string.h>
#define SIZE 10
int main () {
    char non_terminal;
    char beta,alpha;
    int num;
    char production[10][SIZE];
    int index=3; /* starting of the string following "->" */
    printf("Enter Number of Production : ");
    scanf("%d",&num);
    printf("Enter the grammar as E->E-A :\n");
    for(int i=0;i<num;i++){
        scanf("%s",production[i]);
    }
    for(int 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\'|E\n",non_terminal,alpha,non_terminal);

                }

                else

                    printf(" can't be reduced\n");

            }

            else

      printf(" is not left recursive.\n");

      index=3;

   }

  }
```

OUTPUT:



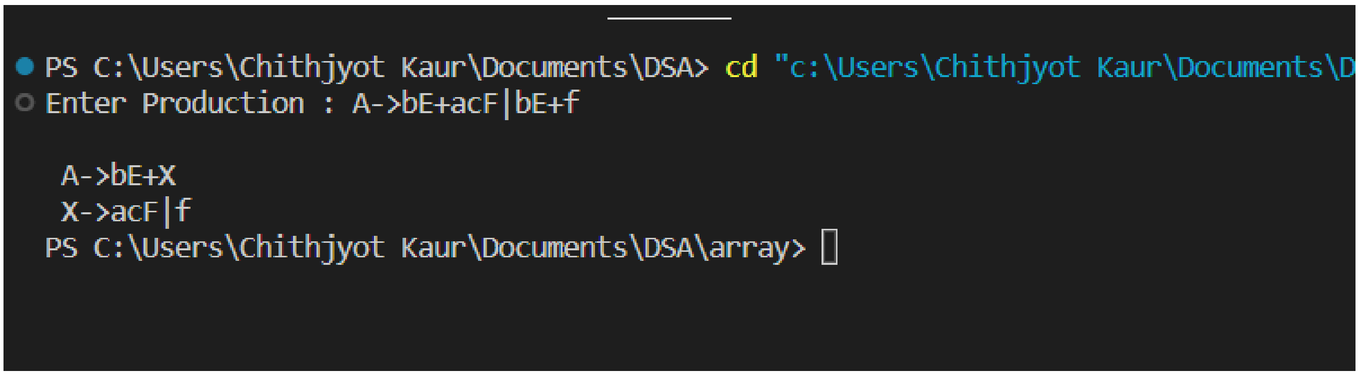
```
PS C:\Users\Chithjyot Kaur\Documents\DSA> cd "c:\Users\Chithjyot Kaur\Documents\DSA\array\"
Enter Number of Production : 4
Enter the grammar as E->E-A :
E->EA|A
A->AT|a
T->a
E->i

GRAMMAR : : : E->EA|A is left recursive.
Grammar without left recursion:
E->AE'
E'->AE'|E

GRAMMAR : : : A->AT|a is left recursive.
Grammar without left recursion:
A->aA'
GRAMMAR : : : E->i is not left recursive.
PS C:\Users\Chithjyot Kaur\Documents\DSA\array> 
```

Left Factoring –

```
#include<stdio.h>
#include<string.h>
int main(){
  chargram[20],part1[20],part2[20],modifiedGram[20],newGram[20],tempGram[20];
   int i,j=0,k=0,l=0,pos;
   printf("Enter Production : A->");
   gets(gram);
   for(i=0;gram[i]!='|';i++,j++)
      part1[j]=gram[i];
   part1[j]='\0';
   for(j=++i,i=0;gram[j]!='\0';j++,i++)
      part2[i]=gram[j];
   part2[i]='\0';
   for(i=0;i<strlen(part1)||i<strlen(part2);i++){
      if(part1[i]==part2[i]){
         modifiedGram[k]=part1[i];
         k++;
         pos=i+1;
      }
   }
   for(i=pos,j=0;part1[i]!='\0';i++,j++){
      newGram[j]=part1[i];
   }
   newGram[j++]='|';
   for(i=pos;part2[i]!='\0';i++,j++){
```

```
            newGram[j]=part2[i];
        }
        modifiedGram[k]='X';
        modifiedGram[++k]='\0';
        newGram[j]='\0';
        printf("\nGrammar Without Left Factoring : : \n");
        printf(" A->%s",modifiedGram);
        printf("\n X->%s\n",newGram);
    }
```

OUTPUT:



```
● PS C:\Users\Chithjyot Kaur\Documents\DSA> cd "c:\Users\Chithjyot Kaur\Documents\D
○ Enter Production : A->bE+acF|bE+f

   A->bE+X
   X->acF|f
  PS C:\Users\Chithjyot Kaur\Documents\DSA\array> 
```

EXPERIMENT 5

AIM : Write a program to find first and follow of given productions.

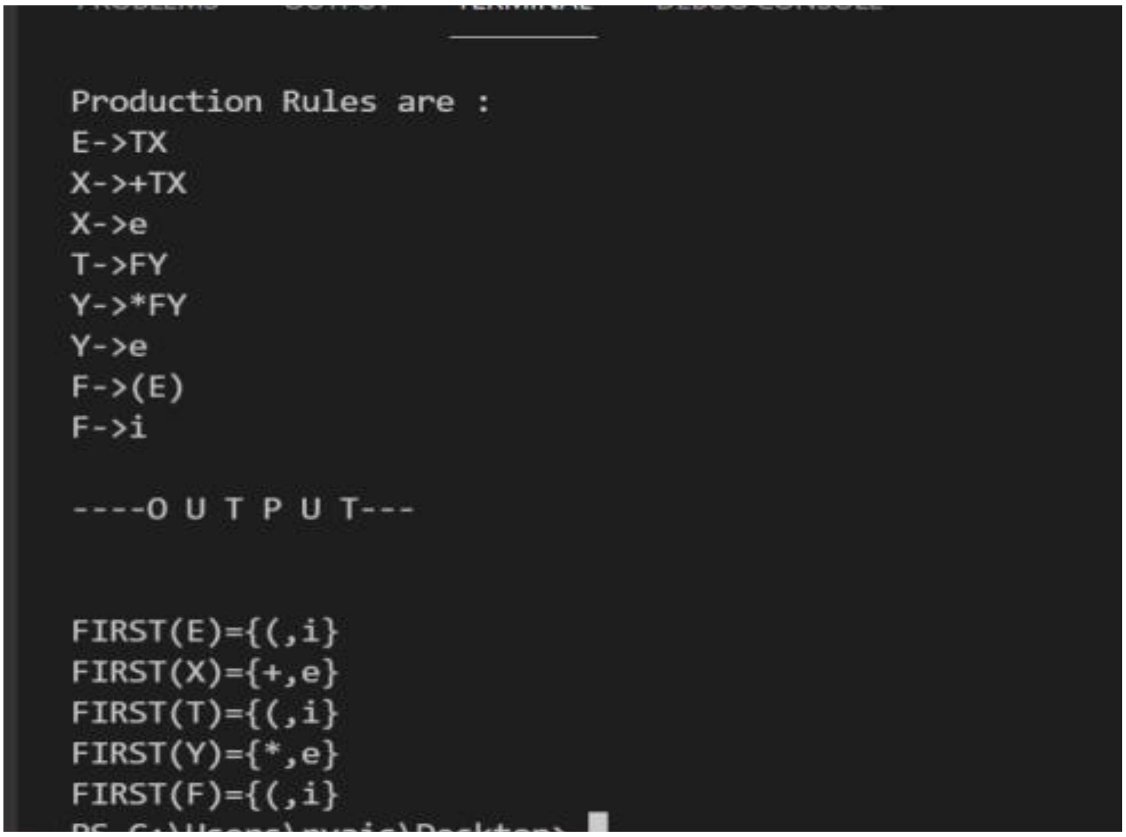
Algorithm:

```
1. Input the number of production N.
2. Input all the production rule PArray
3. Repeat steps a, b, c until process all input production rule i.e. PArray[N]
    a. If $X_i \neq X_{i+1}$ then i. Print Result array of $X_i$ which contain FIRST($X_i$)
    b. If first element of $X_i$ of PArray is Terminal or ε Then i. Add Result = Result U first element
    c. If first element of $X_i$ of PArray is Non-Terminal Then i. searchFirst(i, PArray, N)
4. End Loop
5. If N (last production) then a. Print Result array of $X_i$ which contain FIRST($X_i$)
6. End

Procedure searchFirst(i, PArray, N)
1. Repeat steps Loop j=i+1 to N
    a. If first element of $X_j$ of PArray is Non-Terminal then
        i. searchFirst(j, of PArray, N)
    b. If first element of $X_j$ of PArray is Terminal or ε Then
        i. Add Result = Result U first element
        ii. Flag=0
2. End Loop
3. If Flag = 0 Then
    a. Print Result array of $X_j$ which contain FIRST($X_j$)
4. End
```

Program:

```
#include<iostream>
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
using namespace std;
```

```
void searchFirst(int n, int i, char pl[], char r[], char result[], int k) {
int j,flag;
for(j=i+1;j<n;j++)
{
if(r[i]==pl[j])
{
if(isupper(r[j]))
{
searchFirst(n,j,pl,r,result,k);
}
if(islower(r[j]) || r[j]== '+' || r[j]=='*' || r[j]==')' || r[j]=='(')
{
result[k++]=r[j];
result[k++]=','; flag=0;
}
}
}
if(flag==0)
{
for(j=0;j<k-1;j++)cout<<result[j];
}
}
int main() {
char pr[10][10],pl[10],r[10],prev,result[10];
int i,n,k,j;
cout<<"\nHow many production rule : ";
cin>>n;
if(n==0) exit(0);
for(i=0;i<n;i++)
{
cout<<"\nInput left part of production rules : ";
cin>>pl[i];
cout<<"\nInput right part of production rules : ";
cin>>pr[i];
r[i]=pr[i][0];
}
cout<<"\nProduction Rules are : \n";
for(i=0;i<n;i++)
{
cout<<pl[i]<<"->"<<pr[i]<<"\n";//<<";"<<r[i]<<"\n";
}
cout<<"\n----O U T P U T---\n\n";
prev=pl[0];k=0;
for(i=0;i<n;i++)
{
if(prev!=pl[i])
{
cout<<"\nFIRST("<<prev<<")={";
for(j=0;j<k-1;j++)cout<<result[j];
cout<<"}";
k=0;prev=pl[i];
//cout<<"\n3";
}
if(prev==pl[i])
{
if(islower(r[i]) || r[i]== '+' || r[i]=='*' || r[i]==')' || r[i]=='(')
{
result[k++]=r[i];
result[k++]=',';
}
if(isupper(r[i]))
{
cout<<"\nFIRST("<<prev<<")={";
searchFirst(n,i,pl,r,result,k);
cout<<"}";
k=0;prev=pl[i+1];
}
}
}
if(i==n)
{
cout<<"\nFIRST("<<prev<<")={";
for(j=0;j<k-1;j++)cout<<result[j];
cout<<"}";
k=0;prev=pl[i];
}
return 0;
}
```



```
Production Rules are :
E->TX
X->+TX
X->e
T->FY
Y->*FY
Y->e
F->(E)
F->i

----O U T P U T---


FIRST(E)={(,i}
FIRST(X)={+,e}
FIRST(T)={(,i}
FIRST(Y)={*,e}
FIRST(F)={(,i}
```

Algorithm:

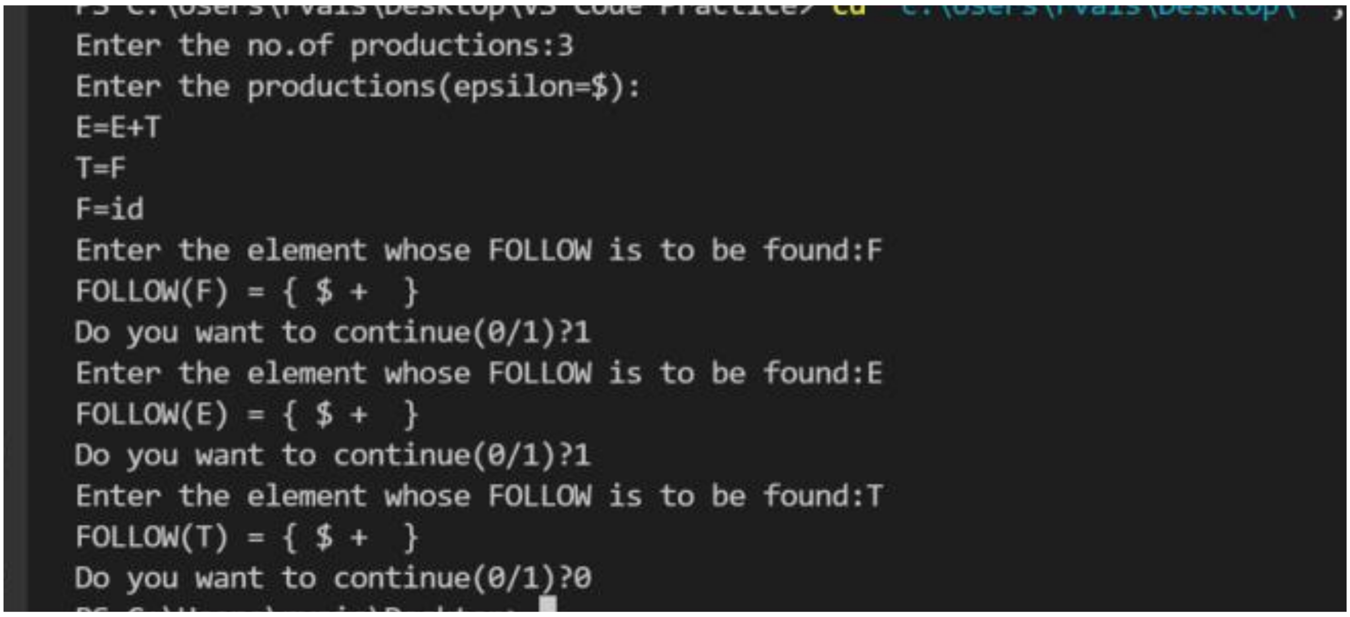
1. Declare the variables.
2. Enter the production rules for the grammar.
3. Calculate the FOLLOW set for each element call the user defined function follow().
4. If x->aBb a. If x is start symbol then FOLLOW(x)={$}.
b. If b is NULL then FOLLOW(B)=FOLLOW(x).
c. If b is not NULL then FOLLOW(B)=FIRST(b).

Program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
using namespace std;

int n,m=0,p,i=0,j=0;
char a[10][10],f[10];
void follow(char c);
void first(char c);

int main()
{
int i,z;
char c,ch;
```

```
printf("Enter the no.of productions:");
scanf("%d",&n);
printf("Enter the productions(epsilon=$):\n");
for(i=0;i<n;i++)
scanf("%s%c",a[i],&ch);
do
{
m=0;
printf("Enter the element whose FOLLOW is to be found:");
16
scanf("%c",&c);
follow(c);
printf("FOLLOW(%c) = { ",c);
for(i=0;i<m;i++)
printf("%c ",f[i]);
printf(" }\n");
printf("Do you want to continue(0/1)?");
scanf("%d%c",&z,&ch);
}
while(z==1);
}
void follow(char c)
{
if(a[0][0]==c)f[m++]='$';
for(i=0;i<n;i++)
{
for(j=2;j<strlen(a[i]);j++)
{
if(a[i][j]==c)
{
if(a[i][j+1]!='\0')first(a[i][j+1]);
if(a[i][j+1]=='\0'&&c!=a[i][0])
follow(a[i][0]);
}
}
}
}
void first(char c)
{
int k;
if(!(isupper(c)))f[m++]=c;
for(k=0;k<n;k++)
{
if(a[k][0]==c)
{
if(a[k][2]=='$') follow(a[i][0]);
else if(islower(a[k][2]))f[m++]=a[k][2];
else first(a[k][2]);
}
}
}
```

OUTPUT:



```
Enter the no.of productions:3
Enter the productions(epsilon=$):
E=E+T
T=F
F=id
Enter the element whose FOLLOW is to be found:F
FOLLOW(F) = { $ +  }
Do you want to continue(0/1)?1
Enter the element whose FOLLOW is to be found:E
FOLLOW(E) = { $ +  }
Do you want to continue(0/1)?1
Enter the element whose FOLLOW is to be found:T
FOLLOW(T) = { $ +  }
Do you want to continue(0/1)?0
```

Result: The Program Executed successfully.

EXPERIMENT 6

AIM : Write a program for predictive parser table of given productions.

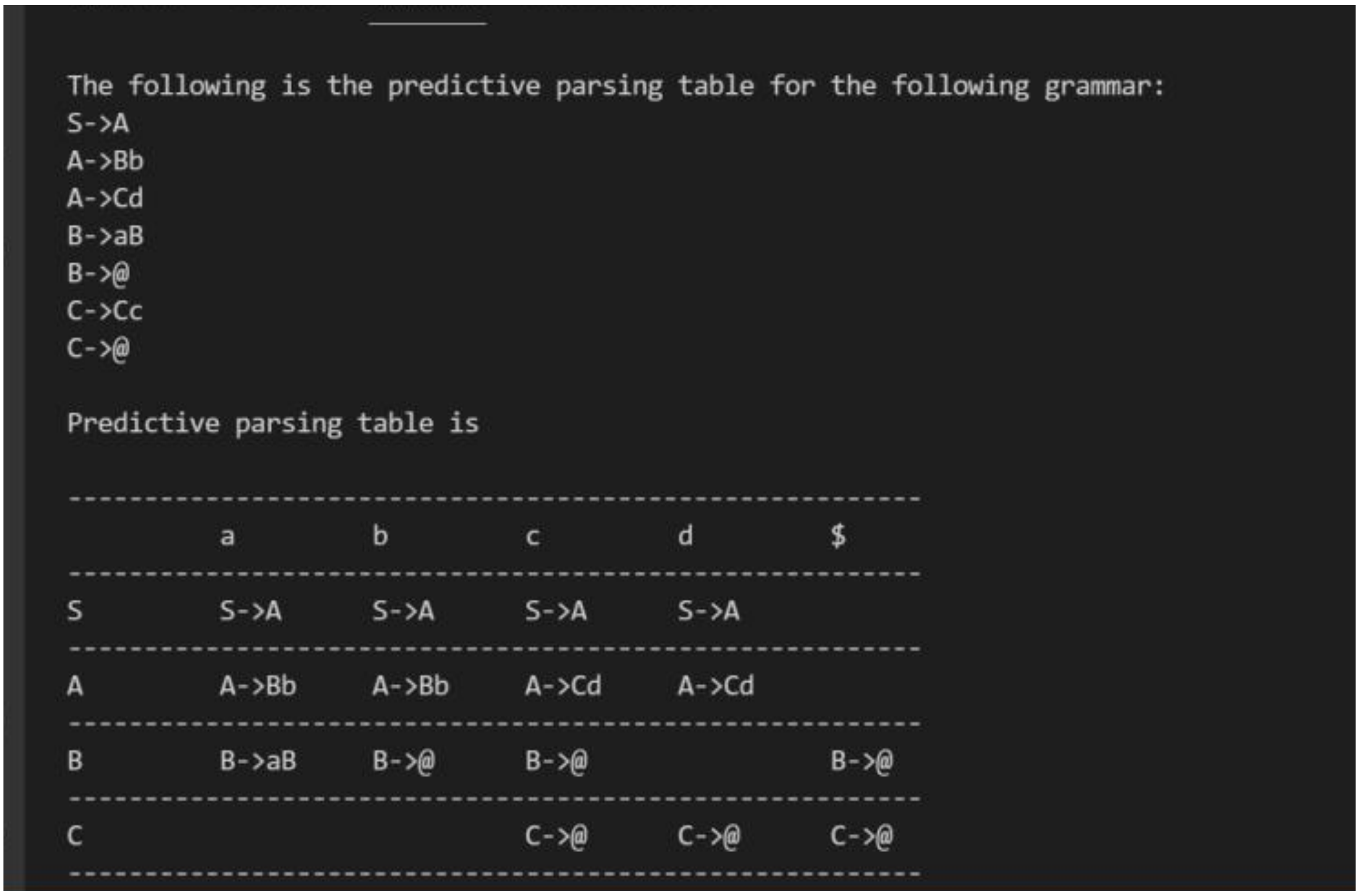
Program:

```
#include<stdio.h>

#include<conio.h>

#include<string.h>

using namespace std;

char prol[7][10] ={"S","A","A","B","B","C","C"};

char pror [7][10] ={"A","Bb","Cd","aB","@","Cc","@"};

char prod [7][10] ={"S->A","A->Bb","A->Cd","B->aB","B->@","C->Cc","C->@"}; 
char

first [7][10] ={"abcd","ab","cd","a@","@","c@","@"}; char

follow [7][10] ={"$","$","$","a$","b$","c$","d$"};

char table [5][6][10];

int numr (char c)

{

switch(c)

{

case 'S': return 0;

case 'A': return 1;

case 'B': return 2;

case 'C': return 3;

case 'a': return 0;

case 'b': return 1;

case 'c': return 2;
case 'd': return 3;
case '$': return 4;
}
return (2);
}
int main ()
{
int i,j,k;
for (i=0; i<5; i++)
for (j=0; j<6; j++)
strcpy(table[i][j]," ");
printf ("\nThe following is the predictive parsing table for the following grammar:\n");
for (i=0; i<7; i++)
19
printf ("%s\n",prod[i]);
printf ("\nPredictive parsing table is\n");
fflush (stdin);
for (i=0; i<7; i++)
{
k=strlen(first[i]);
for (j=0; j<10; j++)
if(first[i][j] !='@')
strcpy(table[numr(prol[i][0])+1][numr(first[i][j])+1],prod[i]);
}
```

```
for(i=0;i<7;i++)
{
if(strlen(pror[i])==1)
{
if(pror[i][0]=='@')
{
k=strlen(follow[i]);
for(j=0;j<k;j++)
strcpy(table[numr(prol[i][0])+1][numr(follow[i][j])+1],prod[i]);
}
}
}
strcpy(table[0][0]," ");
strcpy(table[0][1],"a");
strcpy(table[0][2],"b");
strcpy(table[0][3],"c");
strcpy(table[0][4],"d");
strcpy(table[0][5],"$");
strcpy(table[1][0],"S");
strcpy(table[2][0],"A");
strcpy(table[3][0],"B");
strcpy(table[4][0],"C");
printf("\n--------------------------------------------------------\n");
for(i=0;i<5;i++)
for(j=0;j<6;j++)
```

```
{
printf("%-10s",table[i][j]);
if(j==5)
printf("\n-------------------------------------------------------\n");
}
return 0;
}
```

OUTPUT:



```
The following is the predictive parsing table for the following grammar:
S->A
A->Bb
A->Cd
B->aB
B->@
C->Cc
C->@

Predictive parsing table is

-------------------------------------------------------
          a         b         c         d         $
-------------------------------------------------------
S         S->A      S->A      S->A      S->A
-------------------------------------------------------
A         A->Bb     A->Bb     A->Cd     A->Cd
-------------------------------------------------------
B         B->aB     B->@      B->@                B->@
-------------------------------------------------------
C                             C->@      C->@      C->@
-------------------------------------------------------
```

Result: The Program Executed successfully.

EXPERIMENT 7

AIM : Write a program for Shift Reduce Parser.

Algorithm:

1. Start the Process.
2. Symbols from the input are shifted onto stack until a handle appears on top of the stack.
3. The Symbols that are the handle on top of the stack are then replaces by the left-hand side of the production (reduced).
4. If this result in another handle on top of the stack, then another reduction is done, otherwise we go back to shifting.
5. This combination of shifting input symbols onto the stack and reducing productions when handles appear on the top of the stack continues until all of the input is consumed and the goal symbol is the only thing on the stack - the input is then accepted.
6. If we reach the end of the input and cannot reduce the stack to the goal symbol, the input is rejected.
7. Stop the process.

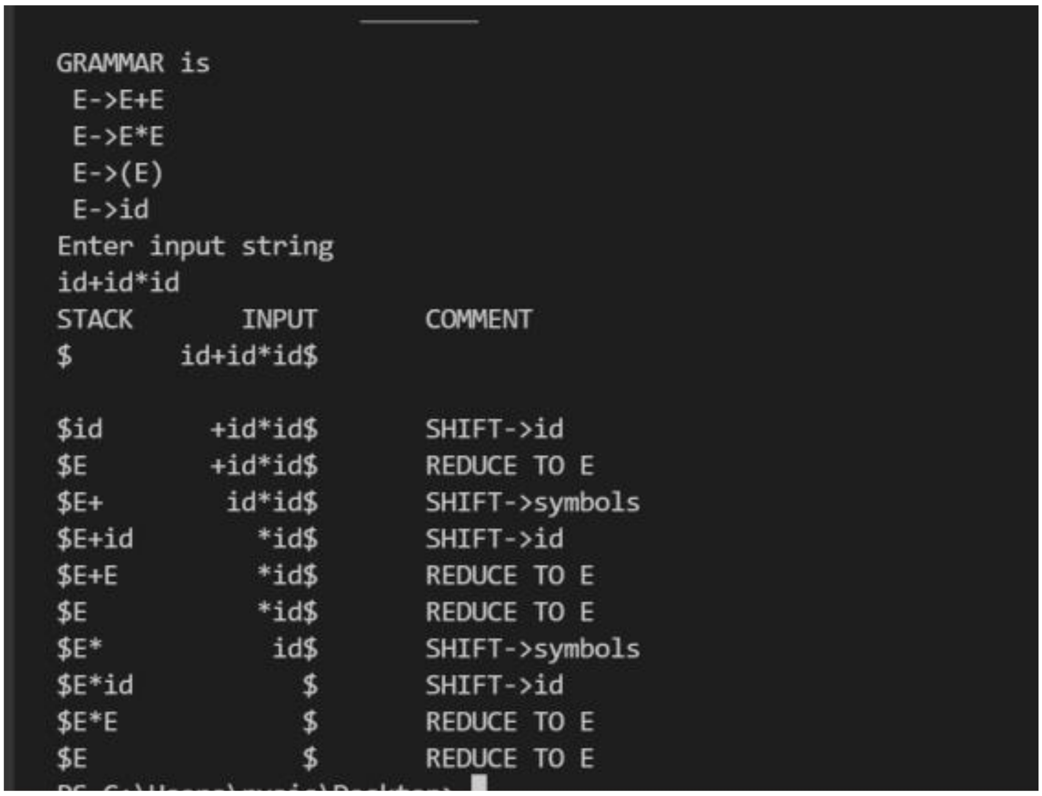
Program:

```
#include<stdio.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();

int main()
{
puts("GRAMMAR is \n 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 \tCOMMENT");
//puts("$ \t");
//puts(a);
printf("$ \t%s$\n",a);
```

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



```
GRAMMAR is
 E->E+E
 E->E*E
 E->(E)
 E->id
Enter input string
id+id*id
STACK       INPUT       COMMENT
$       id+id*id$

$id       +id*id$       SHIFT->id
$E        +id*id$       REDUCE TO E
$E+        id*id$       SHIFT->symbols
$E+id        *id$       SHIFT->id
$E+E         *id$       REDUCE TO E
$E           *id$       REDUCE TO E
$E*           id$       SHIFT->symbols
$E*id           $       SHIFT->id
$E*E            $       REDUCE TO E
$E              $       REDUCE TO E
```

Result: The Program Executed successfully.

LAB 8

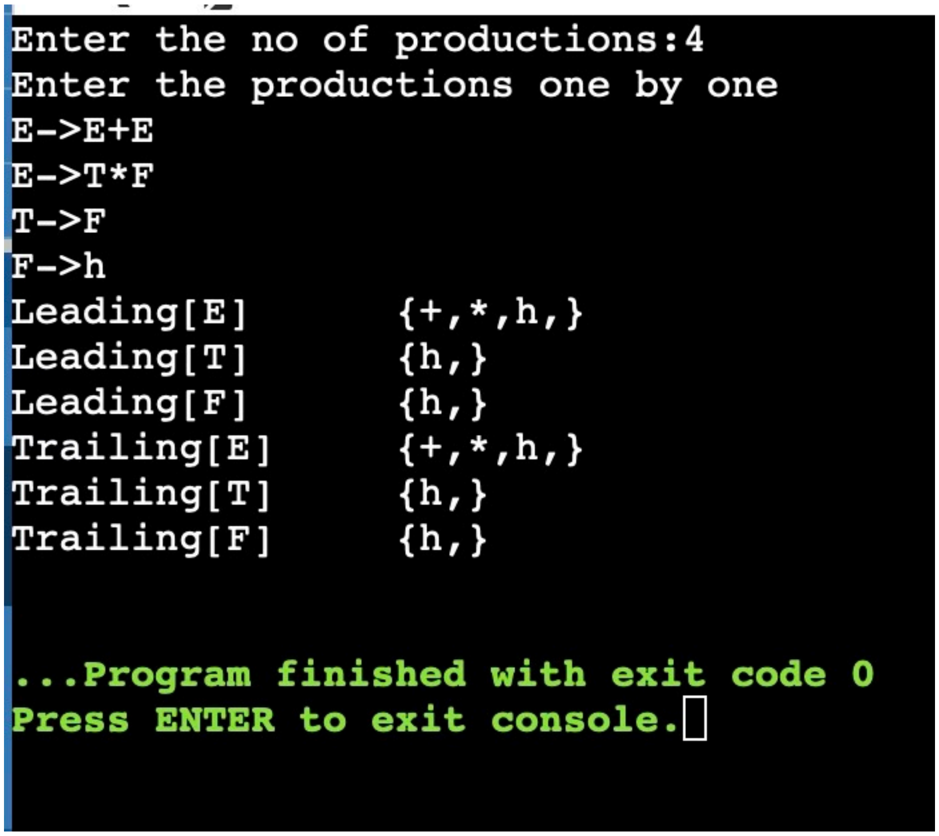
Aim - Computation Of Leading And Trailing

```
#include<iostream>
using namespace std;
#include<string.h>
#include<conio.h>
int nt,t,top=0;
char s[50],NT[10],T[10],st[50],l[10][10],tr[50][50]; int searchnt(char a)
{
int count=-1,i;
for(i=0;i<nt;i++) {
if(NT[i]==a) return i;
} return count;
}
int searchter(char a) {int count=-1,i; for(i=0;i<t;i++) {
if(T[i]==a) return i;
}
return count;
}
void push(char a)
{
s[top]=a;
top++;
}
char pop()
{
top--;
return s[top];
}
void installl(int a,int b) {
if(l[a][b]=='f')
{
l[a][b]='t';
push(T[b]); push(NT[a]);
}
}
void installt(int a,int b) {
if(tr[a][b]=='f')
{
tr[a][b]='t';
push(T[b]); push(NT[a]); }}
int main()
{
int i,s,k,j,n;
char pr[30][30],b,c;
//clrscr();
cout<<"Enter the no of productions:"; cin>>n;
cout<<"Enter the productions one by one\n"; for(i=0;i<n;i++)
cin>>pr[i];
nt=0;
t=0;
for(i=0;i<n;i++)
{
if((searchnt(pr[i][0]))==-1)
NT[nt++]=pr[i][0];
}
for(i=0;i<n;i++)
{
for(j=3;j<strlen(pr[i]);j++)
{
if(searchnt(pr[i][j])==-1)
{
if(searchter(pr[i][j])==-1)
T[t++]=pr[i][j];
}
}
}
for(i=0;i<nt;i++)
```

```
{
for(j=0;j<t;j++) l[i][j]='f';
} for(i=0;i<nt;i++) { for(j=0;j<t;j++) tr[i][j]='f';
}
for(i=0;i<nt;i++)
{
for(j=0;j<n;j++)
{
if(NT[(searchnt(pr[j][0]))]==NT[i])
{
if(searchter(pr[j][3])!=-1) installl(searchnt(pr[j][0]),searchter(pr[j][3])); else
{
for(k=3;k<strlen(pr[j]);k++)
{ if(searchnt(pr[j][k])==-1)
{ installl(searchnt(pr[j][0]),searchter(pr[j][k])); break;
}
}
}
}
}
}
while(top!=0)
{
b=pop();c=pop();
for(s=0;s<n;s++)
{
if(pr[s][3]==b)
installl(searchnt(pr[s][0]),searchter(c));
}
}
for(i=0;i<nt;i++)
{
cout<<"Leading["<<NT[i]<<"]"<<"\t{";
for(j=0;j<t;j++)
{
```

```
if(l[i][j]=='t')
cout<<T[j]<<",";
}
cout<<"}\n";
}
top=0;
for(i=0;i<nt;i++)
{
for(j=0;j<n;j++)
{
if(NT[searchnt(pr[j][0])]==NT[i])
{
if(searchter(pr[j][strlen(pr[j])-1])!=-1)   installt(searchnt(pr[j][0]),searchter(pr[j][strlen(pr[j])-1]));
else
{
for(k=(strlen(pr[j])-1);k>=3;k--)
{
if(searchnt(pr[j][k])==-1)
{ installt(searchnt(pr[j][0]),searchter(pr[j][k])); break;
}
}
}
}
}
}
while(top!=0)
{
b=pop();
c=pop();
for(s=0;s<n;s++)
{
if(pr[s][3]==b) installt(searchnt(pr[s][0]),searchter(c));
}
}
for(i=0;i<nt;i++)
```

```
{
cout<<"Trailing["<<NT[i]<<"]"<<"\t{"; for(j=0;j<t;j++)
{
if(tr[i][j]=='t')
cout<<T[j]<<",";
}
cout<<"}\n";
}
return 0;}
```



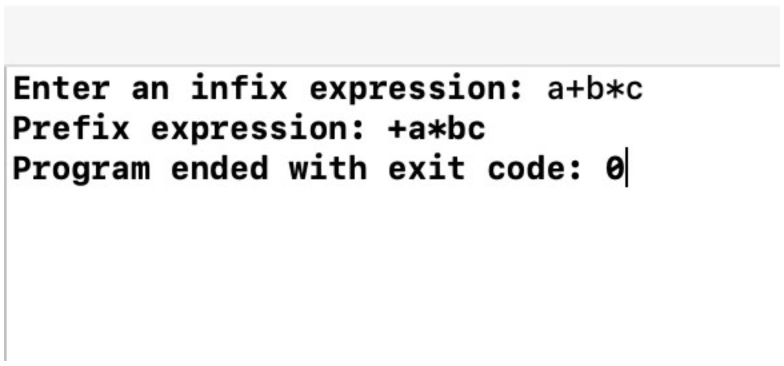
```
Enter the no of productions:4
Enter the productions one by one
E->E+E
E->T*F
T->F
F->h
Leading[E]      {+,*,h,}
Leading[T]      {h,}
Leading[F]      {h,}
Trailing[E]     {+,*,h,}
Trailing[T]     {h,}
Trailing[F]     {h,}


...Program finished with exit code 0
Press ENTER to exit console.
```

LAB 9

Aim - Intermediate Code Generation : Prefix

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

#define MAX_SIZE 20

char stack[MAX_SIZE];
int top = -1;

void reverse_string(char* str) {
    int length = strlen(str);
    int i, j;
    char temp;

    for (i = 0, j = length - 1; i < j; i++, j--) {
        temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }
}


void push(char ch) {
    if (top == MAX_SIZE - 1) {
        printf("Stack overflow\n");
        return;
    }
    top++;
    stack[top] = ch;
}

char pop() {
    if (top == -1) {
        printf("Stack underflow\n");
        return '\0';
    }
    char ch = stack[top];
    top--;
    return ch;
}

int isOperator(char ch) {
    return ch == '+' || ch == '-' || ch == '*' || ch == '/' ||
ch == '^';
}

int precedence(char ch) {
    switch (ch) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
        default:
            return 0;
    }
}

void infixToPrefix(char infix[], char prefix[]) {
    reverse_string(infix);
    int len = strlen(infix);
    for (int i = 0; i < len; i++) {
        char ch = infix[i];
        if (isOperator(ch)) {
            while (top != -1 && precedence(stack[top]) >=
precedence(ch)) {
                prefix[strlen(prefix)] = pop();
            }
            push(ch);
        } else if (ch == ')') {
            push(ch);
        } else if (ch == '(') {
            while (top != -1 && stack[top] != ')') {
                prefix[strlen(prefix)] = pop();
            }
            pop();
        } else {
            prefix[strlen(prefix)] = ch;
        }
    }
    while (top != -1) {
        prefix[strlen(prefix)] = pop();
    }
    reverse_string(prefix);
}
```

```c
int main() {
    char infix[20], prefix[20];
    printf("Enter an infix expression: ");
    scanf("%s", infix);
    infixToPrefix(infix, prefix);
    printf("Prefix expression: %s\n", prefix);
    return 0;
}
```



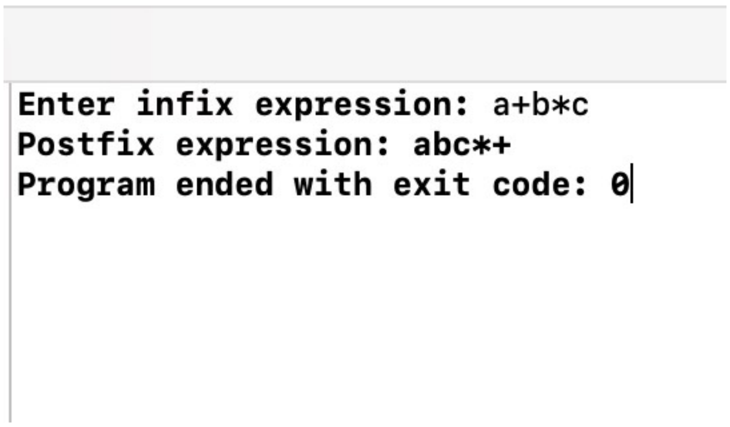
```
Enter an infix expression: a+b*c
Prefix expression: +a*bc
Program ended with exit code: 0
```

LAB 10

Aim - Intermediate Code Generation : Postfix

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

#define MAX_SIZE 20

int top = -1;
char infix[MAX_SIZE], postfix[MAX_SIZE], stack[MAX_SIZE];

void push(char);
char pop();
int is_operator(char);
int precedence(char, int);
void infix_to_postfix();

int main() {
    printf("Enter infix expression: ");
    scanf("%s", infix);

    infix_to_postfix();

    printf("Postfix expression: %s\n", postfix);

    return 0;
}

void push(char symbol) {
    if (top == MAX_SIZE - 1) {
        printf("Stack overflow\n");
        exit(1);
    }
    stack[++top] = symbol;
}

char pop() {
    if (top == -1) {
        printf("Stack underflow\n");
        exit(1);
    }
    return stack[top--];
}
```

```
int is_operator(char symbol) {
    if (symbol == '+' || symbol == '-' || symbol == '*' ||
symbol == '/' || symbol == '^') {
        return 1;
    }
    return 0;
}

int precedence(char symbol, int is_operator) {
    int precedence = 0;

    switch (symbol) {
        case '+':
        case '-':
            precedence = 1;
            break;
        case '*':
        case '/':
            precedence = 2;
            break;
        case '^':
            precedence = 3;
            break;
        default:
            if (is_operator) {
                printf("Invalid operator: %c\n", symbol);
                exit(1);
            }
            break;
    }

    return precedence;
}

void infix_to_postfix() {
    int i = 0, j = 0;
    char symbol;

    while (infix[i] != '\0') {
        symbol = infix[i];

        if (isalnum(symbol)) {
            postfix[j++] = symbol;
        }
        else if (is_operator(symbol)) {
            while (top != -1 && precedence(stack[top], 1) >=
precedence(symbol, 0)) {
                postfix[j++] = pop();
            }
            push(symbol);
        }
        else if (symbol == '(') {
            push(symbol);
        }
        else if (symbol == ')') {
            while (stack[top] != '(') {
                postfix[j++] = pop();
            }
            pop();
        }
        else {
            printf("Invalid symbol: %c\n", symbol);
            exit(1);
        }

        i++;
    }

    while (top != -1) {
        postfix[j++] = pop();
    }

    postfix[j] = '\0';
}
```



```
Enter infix expression: a+b*c
Postfix expression: abc*+
Program ended with exit code: 0
```