COMPILER DESIGN

Sub. Code : 18CSC304J

Anubhav Sharma(RA2011028030025)

B.Tech CSE Cloud Computing III Year , VI Semester

Submitted to: Dr. Akash Punhani Sir

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

FACULTY OF ENGINEERING TECHNOLOGY SRM INSTITUTE OF SCIENCE & TECHNOLOGY, DELHI-NCR CAMPUS, MODINAGAR

www.srmup.in

LAB 1

Aim - Implementation of Lexical Analyzer

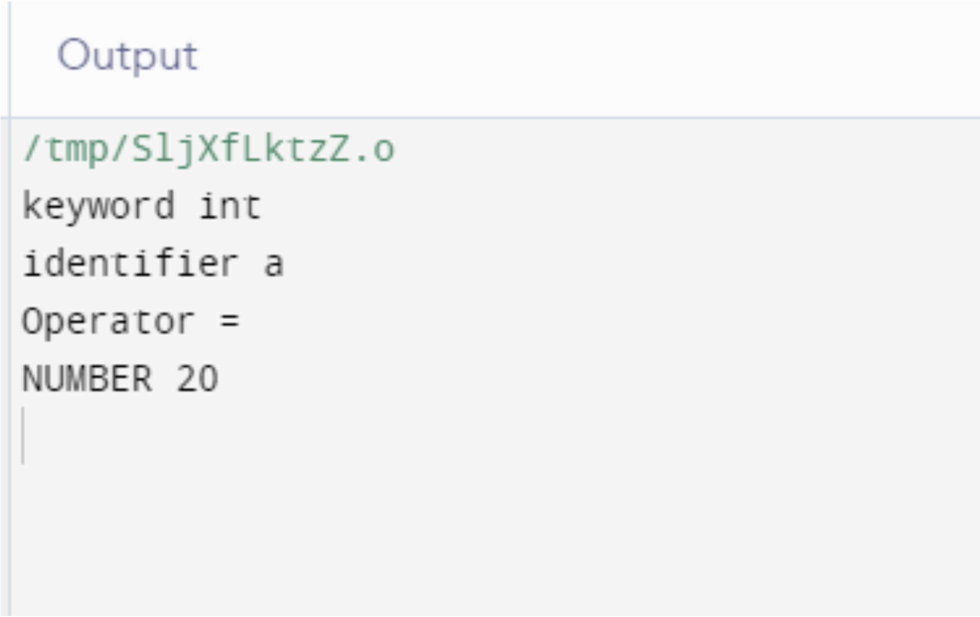
Source Code –

```
#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <ctype.h>


const char *keywords[] = { "if", "else", "while", "for", "do", "int", "float", "char", "double", "return" };

const int num_keywords = 10;


int is_keyword(char *str) {

   for (int i = 0; i < num_keywords; i++) {

      if (strcmp(str, keywords[i]) == 0) {

         return 1;

      }

   }

   return 0;

}

char *operators[] = {"+","-","*","/","%","^"};

int is_operator(char *str) {

   for (int i = 0; i < 6; i++) {

      if (strcmp(str, operators[i]) == 0) {

         return 1;
```

```
        }
    }
    return 0;
}

void lexicalanalyser(char *code){
    int i =0;
    char *p = " ";
    char *p1 = "\0";
    char *p2 = ";";

    while(code[i]){
    if(isdigit(code[i])){
            char num[100];
            int j=0;
            num[j++] = code[i];
            i++;
            while (isdigit(code[i])) {
                num[j++] = code[i];
                i++;
            }
            num[j] = '\0';
            printf("NUMBER %s\n", num);
        }
    else if (isalpha(code[i])) {
            char word[100];
            int j = 0 ;
            word[j++] = code[i];
            i++;
            while (isalpha(code[i])) {
                word[j++] = code[i];
                i++;
            }
            word[j]= '\0';
            if(is_keyword(word)){
                printf("keyword %s\n", word);
            }else{
                printf("identifier %s\n", word);
            }
    }
    else if((code[i] != *p) && (code[i] != *p1) && (code[i] != *p2)){
        char Operator[10];
        int j = 0;
        Operator[j++] = code[i];
        i++;
        while(code[i] != *p){
            Operator[i++] = code[i];
            i++;
        }
        Operator[j] = '\0';
        printf("Operator %s\n",Operator);
    }
    else{
        i++;
    }
    }
```

```
}
int main()
{
    char *p = "int a = 20;";
    lexicalanalyser(p);
    return 0;
}
```

Output –

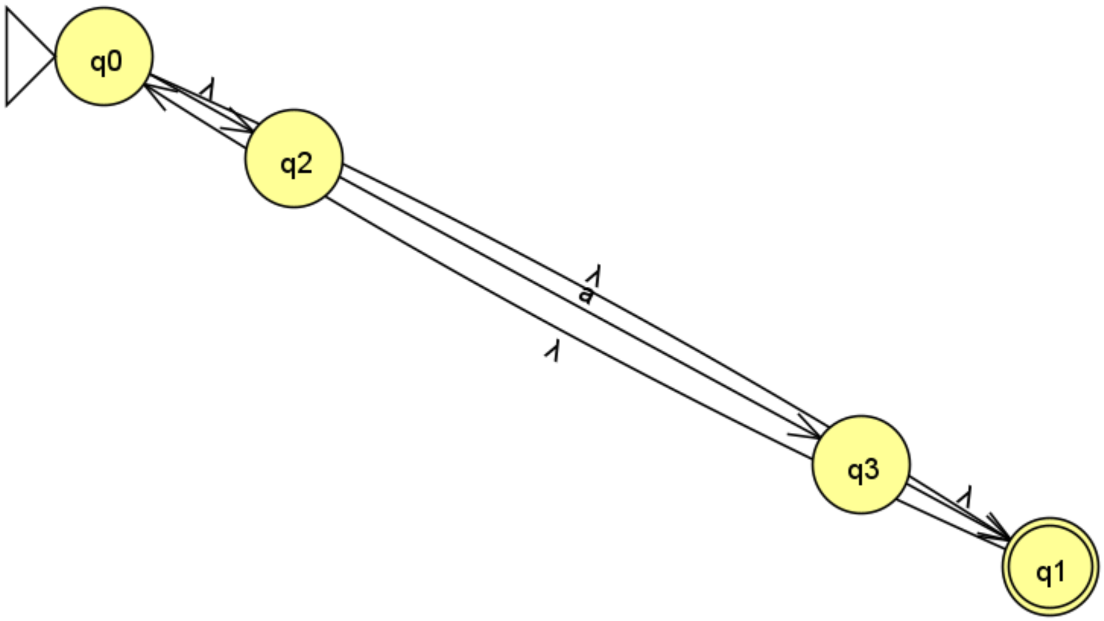


```
/tmp/SljXfLktzZ.o
keyword int
identifier a
Operator =
NUMBER 20
```

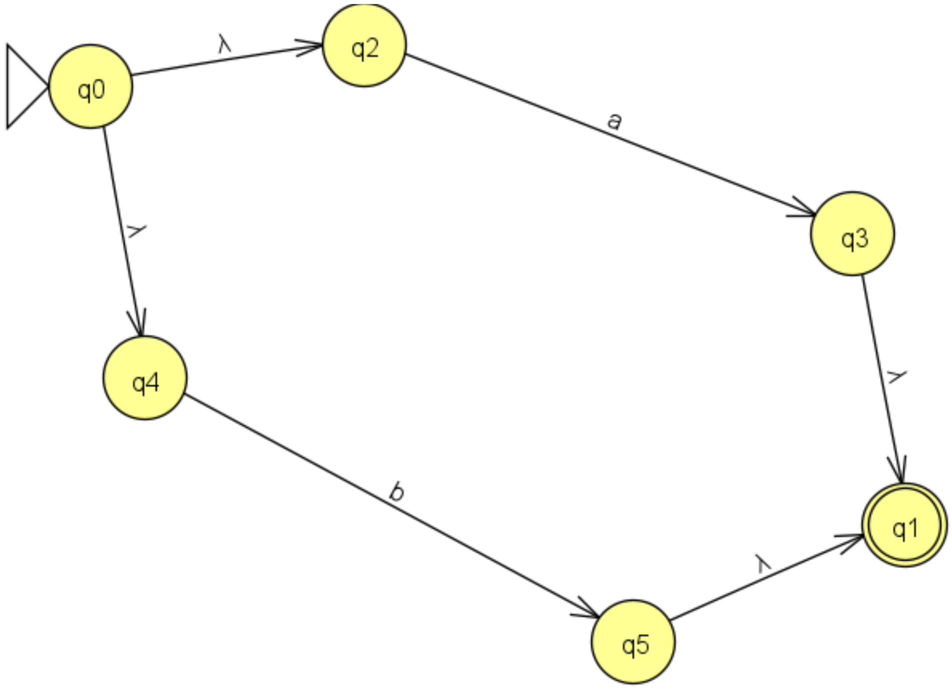
LAB 2

Aim - Conversion from Regular Expression to NFA

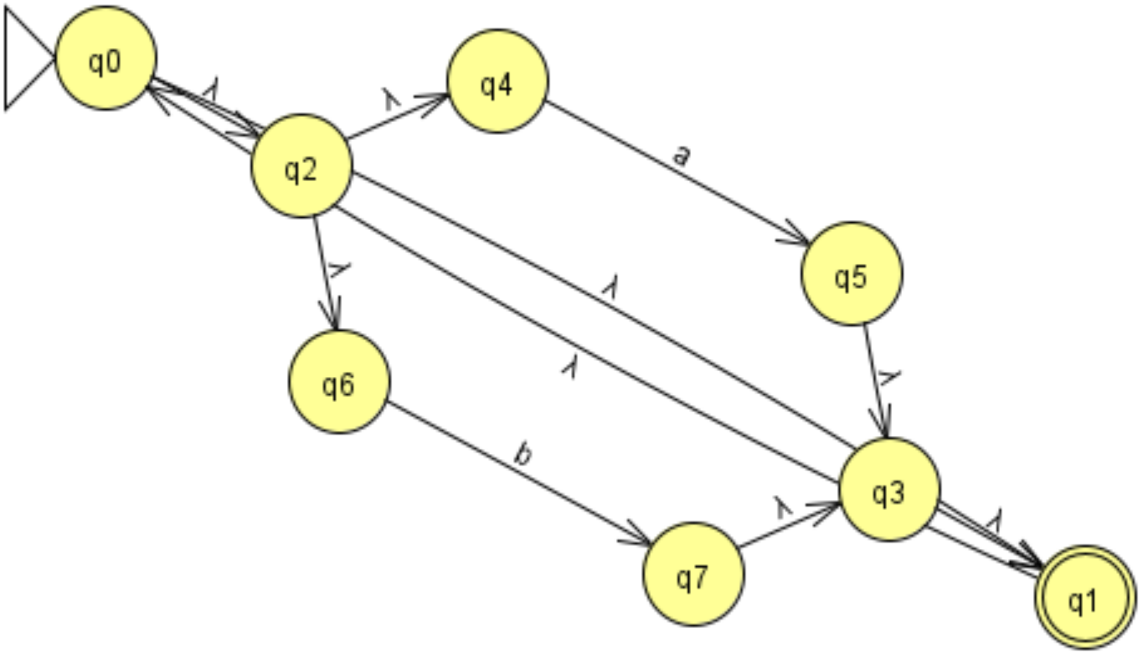
1. **a***



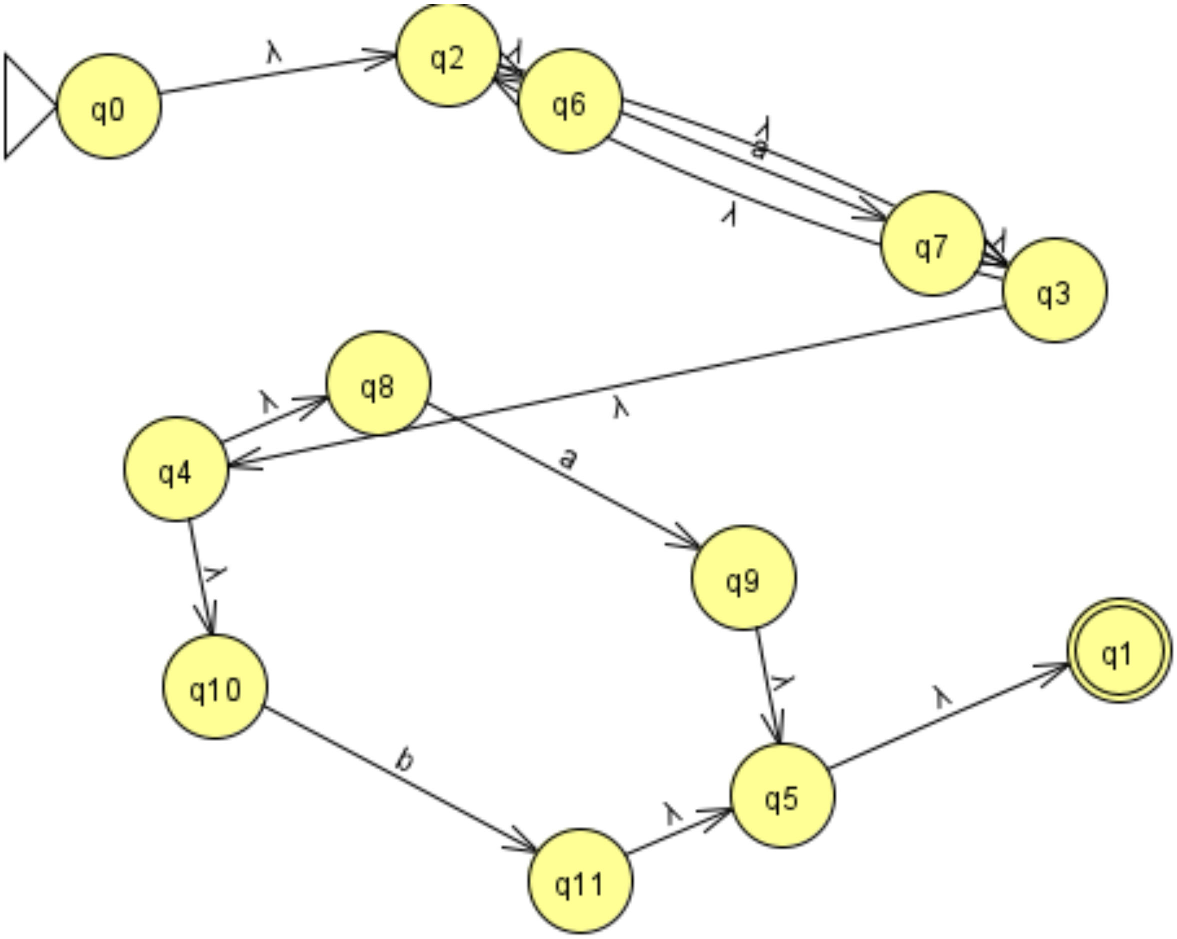
2. **a+b**



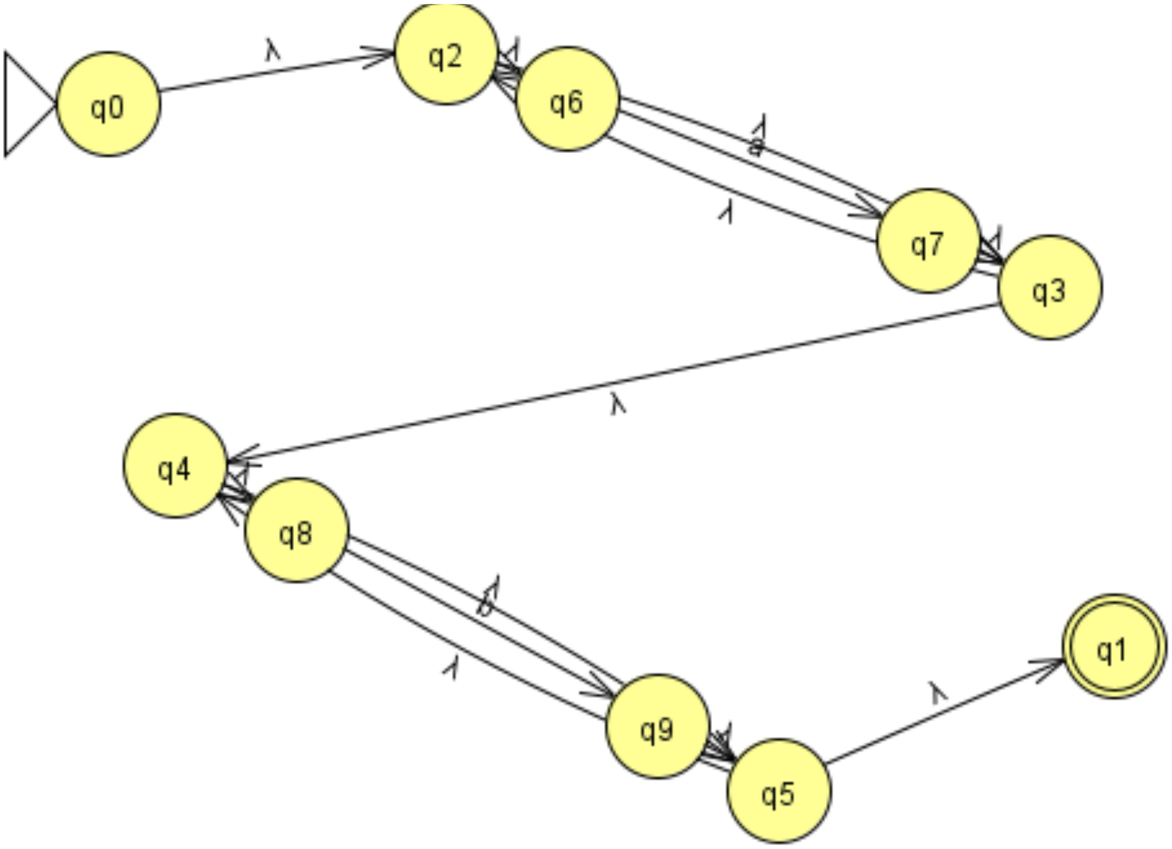
3. (a+b)*


q0
λ
q2
λ
q4
a
q5
λ
λ
q6
λ
λ
b
q3
λ
λ
q7
q1

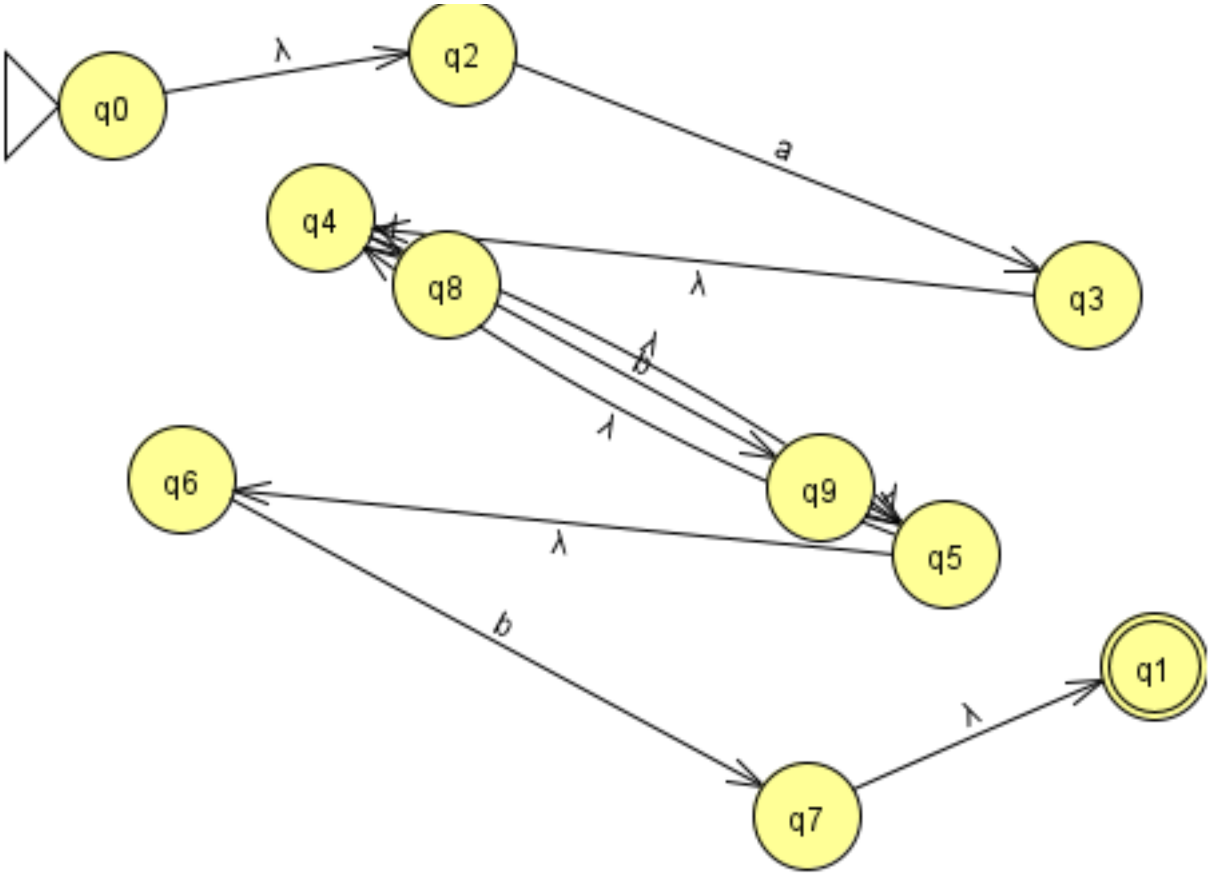
4. a*(a+b)


q0
λ
q2
λ
q6
λ
a
λ
q7
λ
q3
q8
λ
λ
q4
a
q9
λ
q10
λ
q1
λ
b
q5
λ
q11

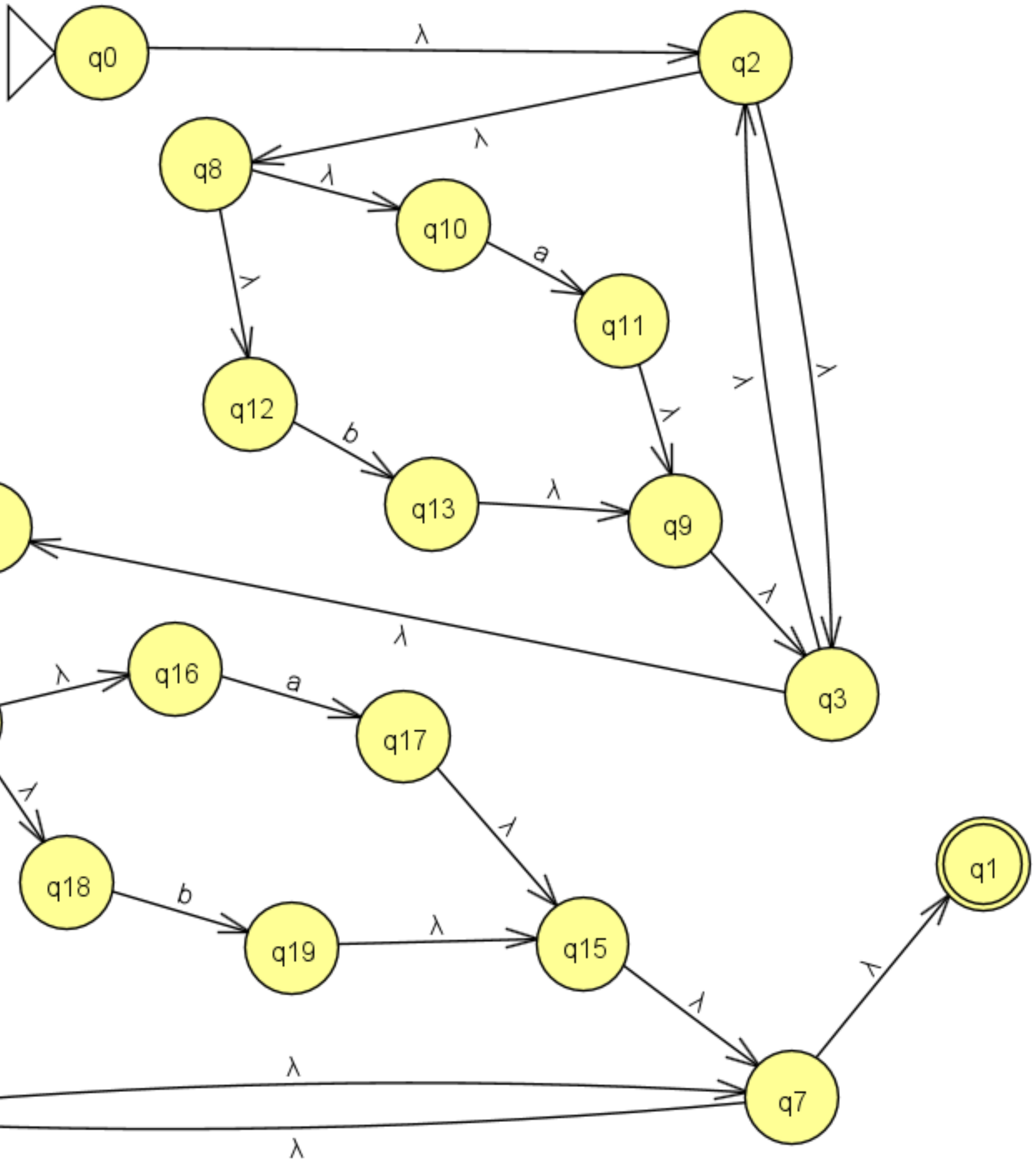
5. **a*b***


q0
λ
q2
λ
q6
λ
a
λ
q7
λ
q3
λ
q4
λ
q8
λ
b
λ
q9
λ
q5
λ
q1

6. **ab*b**


q0
λ
q2
a
q3
λ
q4
λ
q8
λ
b
λ
q9
λ
q5
λ
q6
b
q7
λ
q1

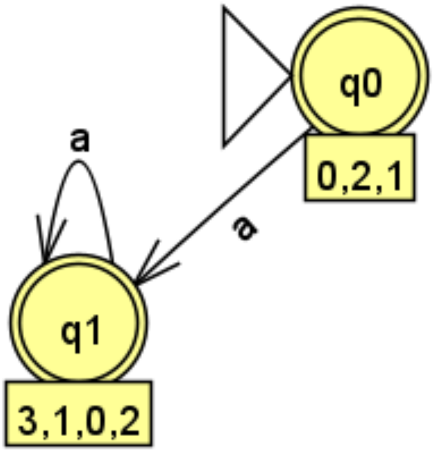
7. (a+b)*a(a+b)*


q0
λ
q2
λ
q8
λ
q10
a
q11
λ
λ
λ
λ
q12
b
q13
λ
q9
λ
q4
q5
a
λ
q3
q14
λ
q16
a
q17
λ
λ
λ
q18
b
q19
λ
q15
λ
q1
λ
q6
λ
q7
λ

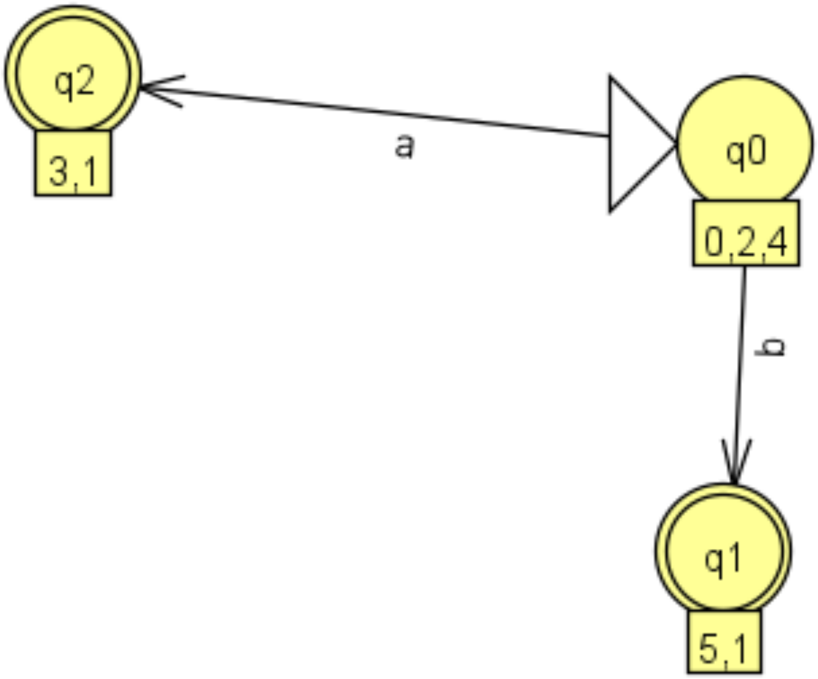
LAB 3

Aim - Conversion from NFA to DFA

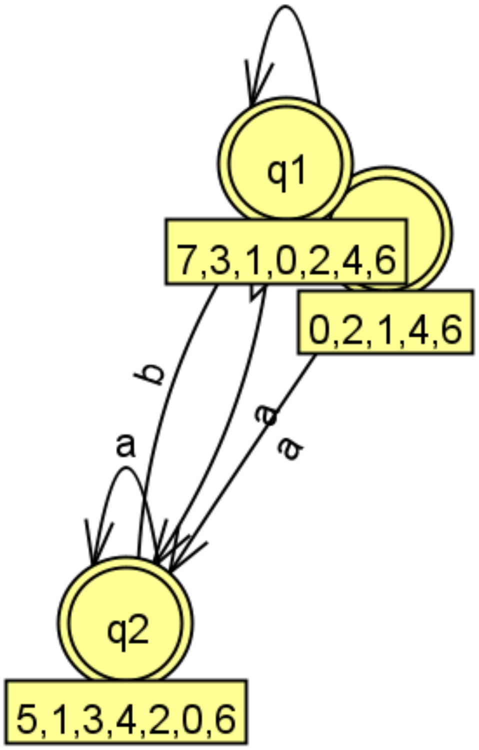
1. **a***



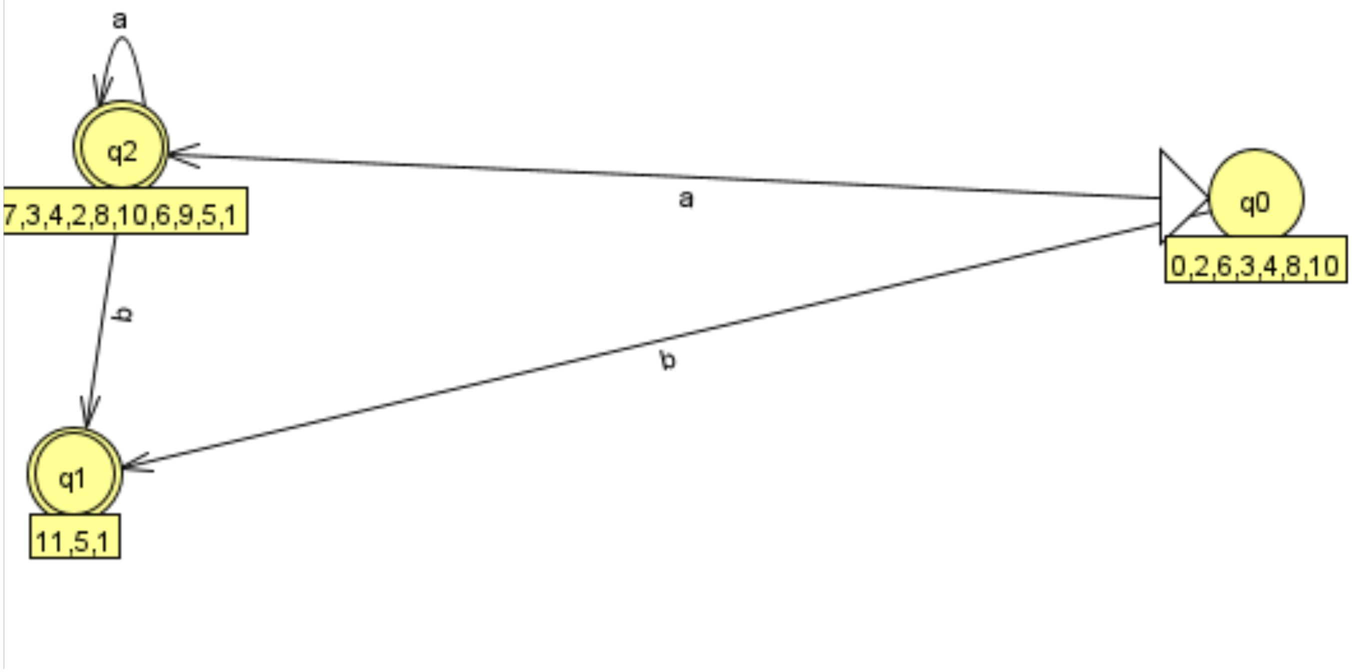
2. **a+b**



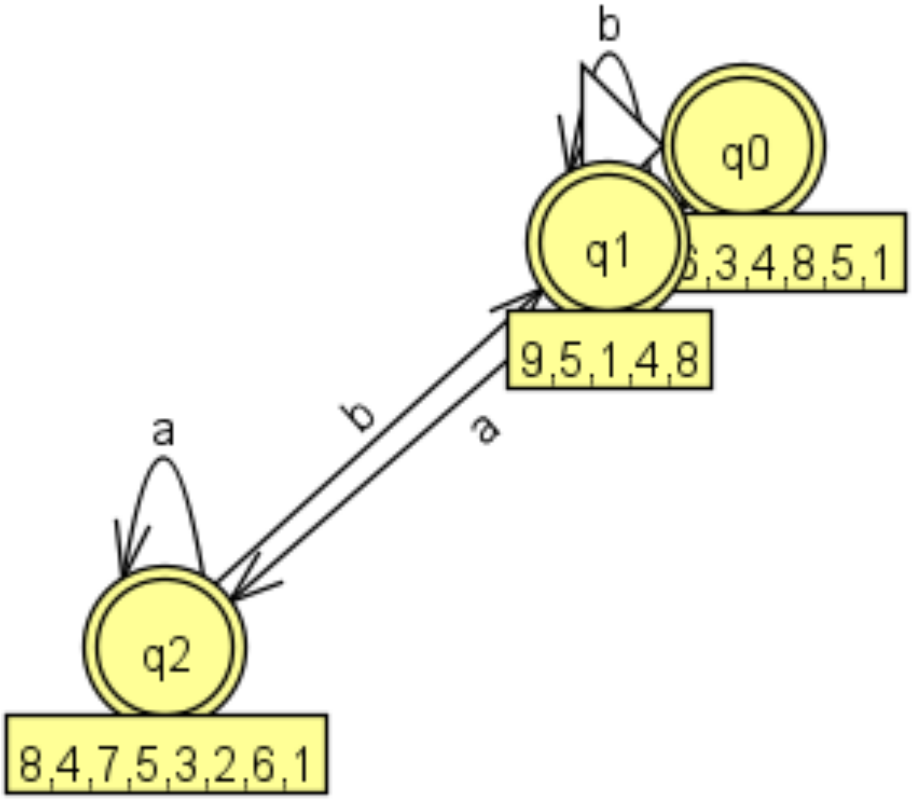
3. (a+b)*


q1
7,3,1,0,2,4,6
0,2,1,4,6
b
a
a
a
q2
5,1,3,4,2,0,6

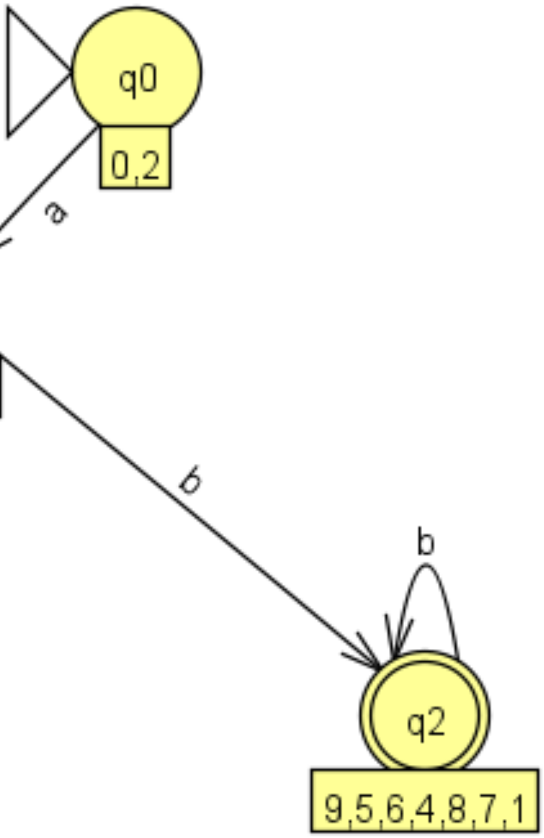
4. a*(a+b)


a
q2
7,3,4,2,8,10,6,9,5,1
a
q0
0,2,6,3,4,8,10
b
b
q1
11,5,1

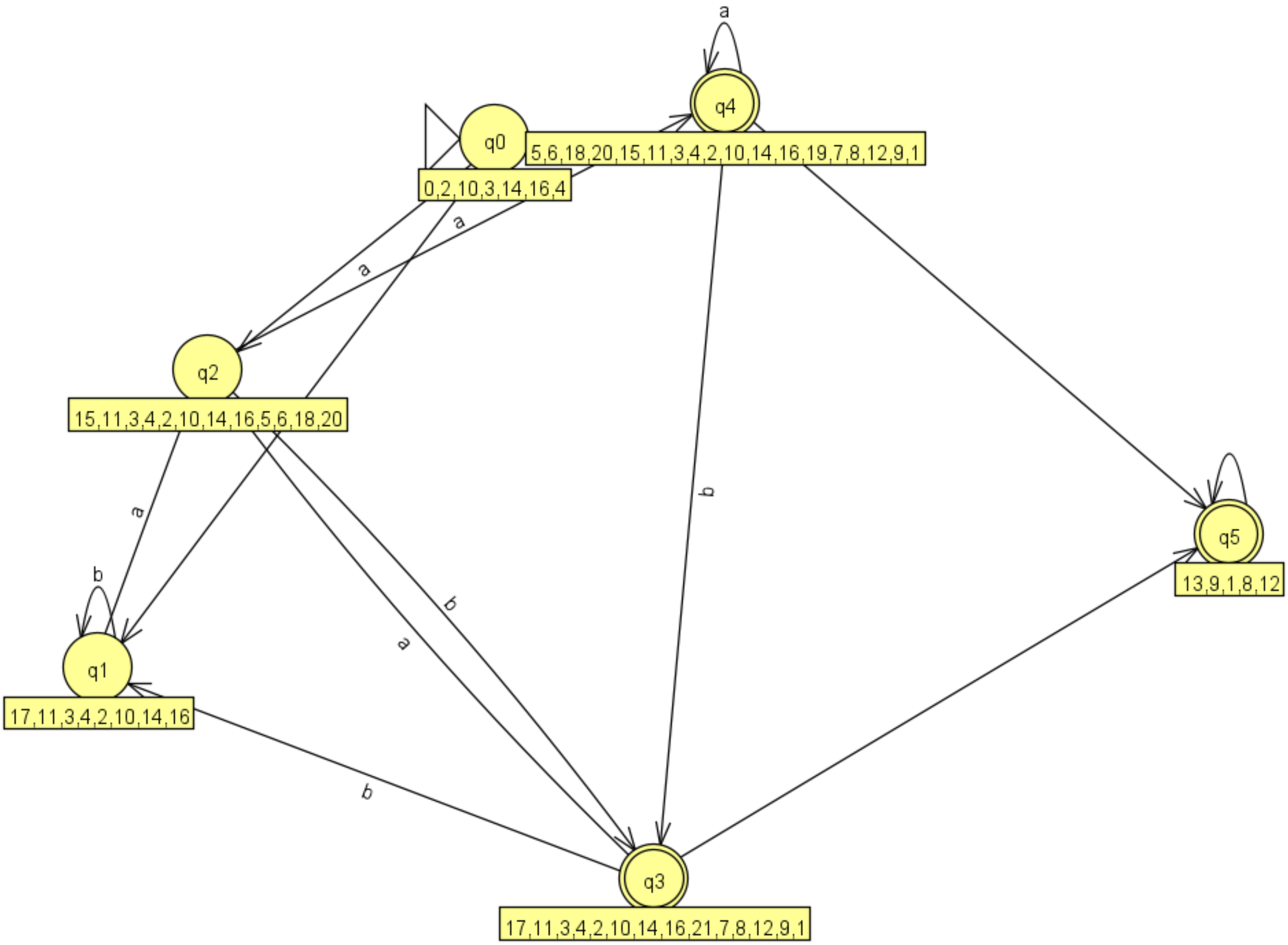
5. a*b*


b
q0
6,3,4,8,5,1
q1
9,5,1,4,8
b
a
a
q2
8,4,7,5,3,2,6,1

6. **ab*b**


q0
0,2
a
q1
3,4,8,5,6
b
b
q2
9,5,6,4,8,7,1

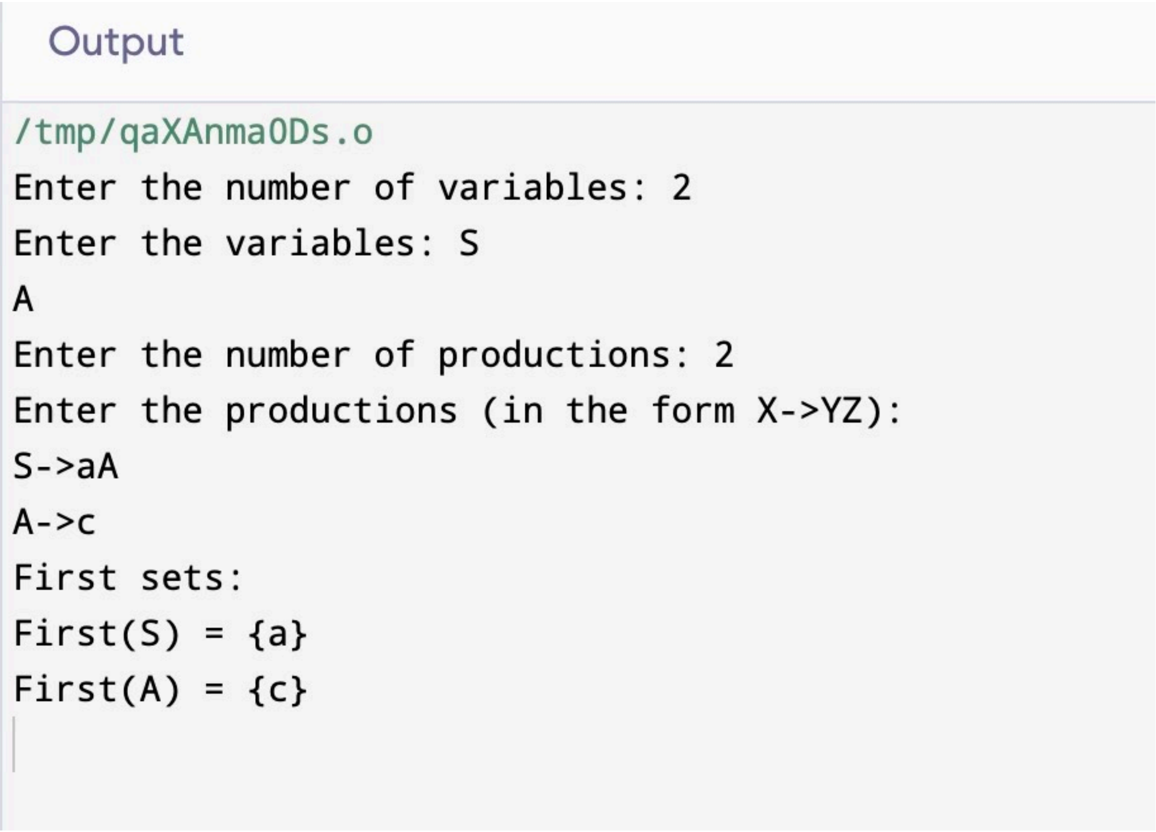
7. (a+b)*a(a+b)*


a
q4
q0
5,6,18,20,15,11,3,4,2,10,14,16,19,7,8,12,9,1
0,2,10,3,14,16,4
a
a
q2
15,11,3,4,2,10,14,16,5,6,18,20
b
q5
13,9,1,8,12
a
b
b
q1
a
17,11,3,4,2,10,14,16
b
q3
17,11,3,4,2,10,14,16,21,7,8,12,9,1

LAB 4

Aim - Write a program to find first of given productions

```
#include <iostream>

#include <string>

#include <map>

#include <vector>


#define MAX_VARIABLES 26


using namespace std;


map<char, vector<string>> productions;

vector<char> variables;

map<char, vector<char>> first;


bool is_variable(char c) {

  for (int i = 0; i < variables.size(); i++) {

    if (variables[i] == c) {

      return true;

    }

  }
  return false;
}

void find_first(char c) {
  if (!is_variable(c)) {
    first[c].push_back(c);
    return;
  }
  for (int i = 0; i < productions[c].size(); i++) {
    string production = productions[c][i];
    for (int j = 0; j < production.length(); j++) {
      char symbol = production[j];
      if (!is_variable(symbol)) {
        first[c].push_back(symbol);
        break;
      } else {
        find_first(symbol);
        for (int k = 0; k < first[symbol].size(); k++) {
          if (first[symbol][k] != '#') {
            first[c].push_back(first[symbol][k]);
            break;
          }
        }
      }
    }
  }
}

int main() {
  int num_variables;
  cout<< "Enter the number of variables: ";  cin >> num_variables;
  cout << "Enter the variables: ";
  for (int i = 0; i < num_variables; i++) {
    char variable;
    cin >> variable;
    variables.push_back(variable);
  }
  int num_productions;
  cout<< "Enter the number of productions: "; cin >> num_productions;
  cout << "Enter the productions (in the form X->YZ):" << endl;
  for (int i = 0; i < num_productions; i++) {
    char variable;
    string production;
    cin >> variable >> production;
    productions[variable].push_back(production.substr(3));
  }
  for (int i = 0; i < variables.size(); i++) {
    find_first(variables[i]);
  }
  cout << "First sets:" << endl;
  for (int i = 0; i < variables.size(); i++) {
    cout << "First(" << variables[i] << ") = {";
    for (int j = 0; j < first[variables[i]].size(); j++) {
      cout << first[variables[i]][j] << ", ";
    }
    cout << "}" << endl;
  }return 0;}
```



```
/tmp/qaXAnma0Ds.o
Enter the number of variables: 2
Enter the variables: S
A
Enter the number of productions: 2
Enter the productions (in the form X->YZ):
S->aA
A->c
First sets:
First(S) = {a}
First(A) = {c}
```

LAB 5

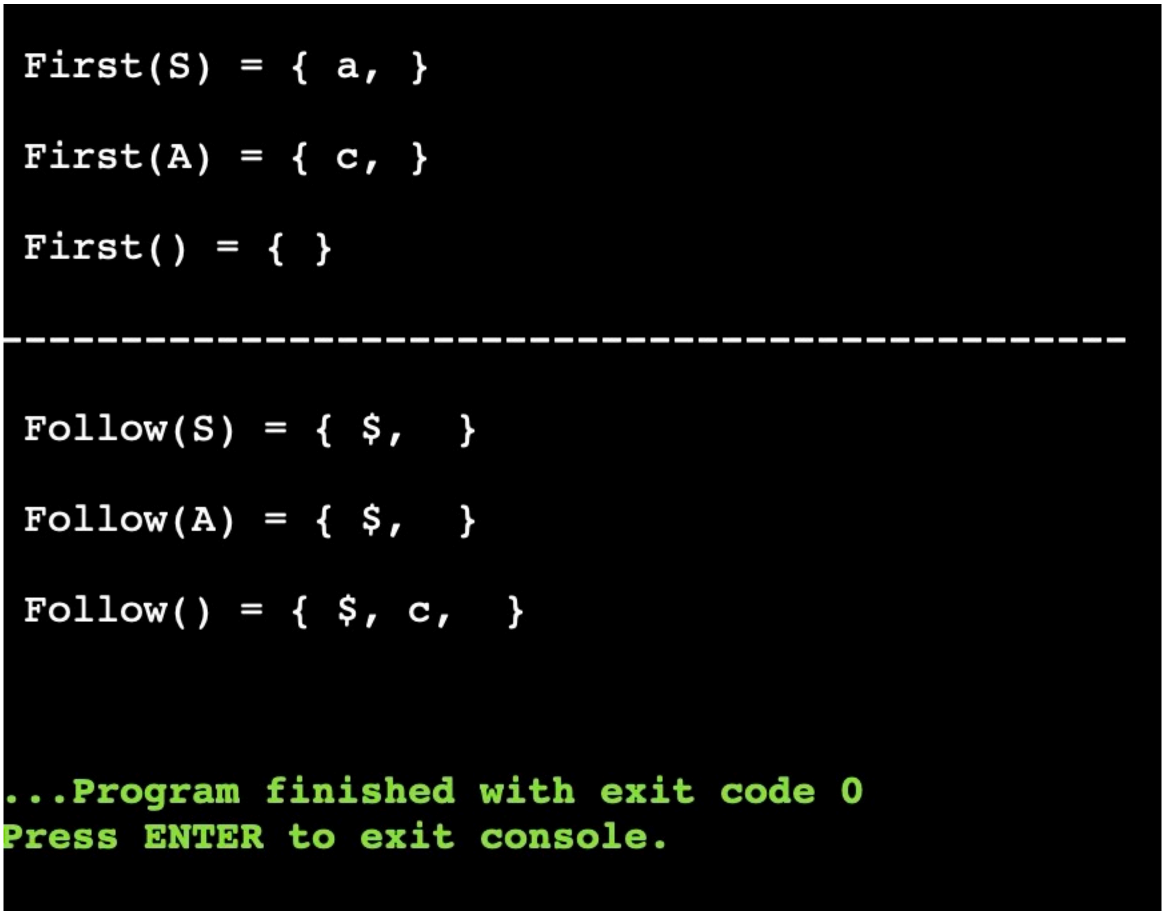
Aim - Write a program to find follow of given productions

```
#include <ctype.h>
#include <stdio.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], "S=aA");
        strcpy(production[1], "A=c");
        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;
    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;
        }
        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++;
        }
    }
}
```



```
First(S) = { a, }

First(A) = { c, }

First() = { }

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

Follow(S) = { $,  }

Follow(A) = { $,  }

Follow() = { $, c,  }



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

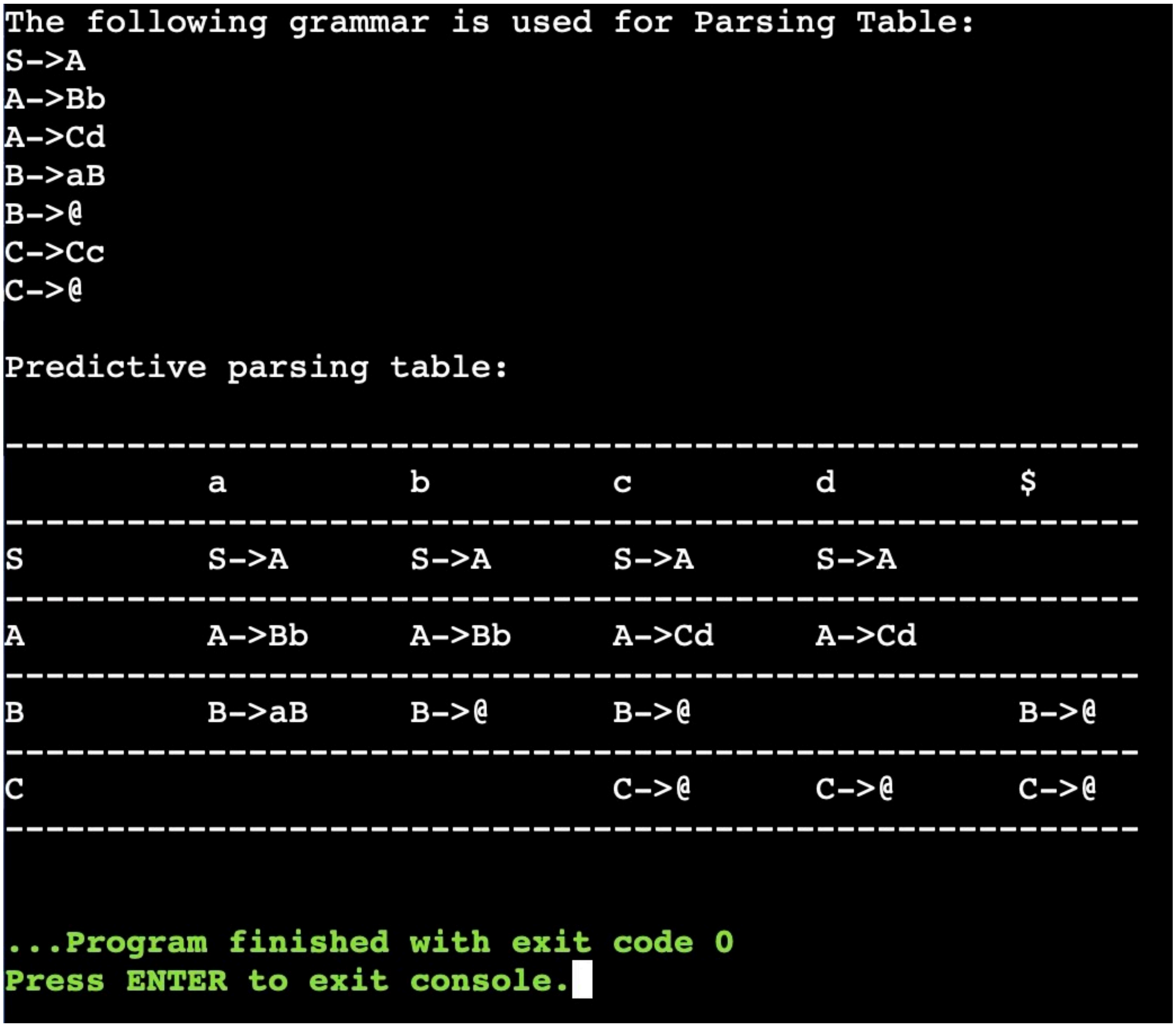
LAB 6

Aim - Write a program for predictive parser table of given productions

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

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("The following grammar is used for Parsing Table:\n");

  for (i = 0; i < 7; i++)
    printf("%s\n", prod[i]);

  printf("\nPredictive parsing table:\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");
      }
}
```

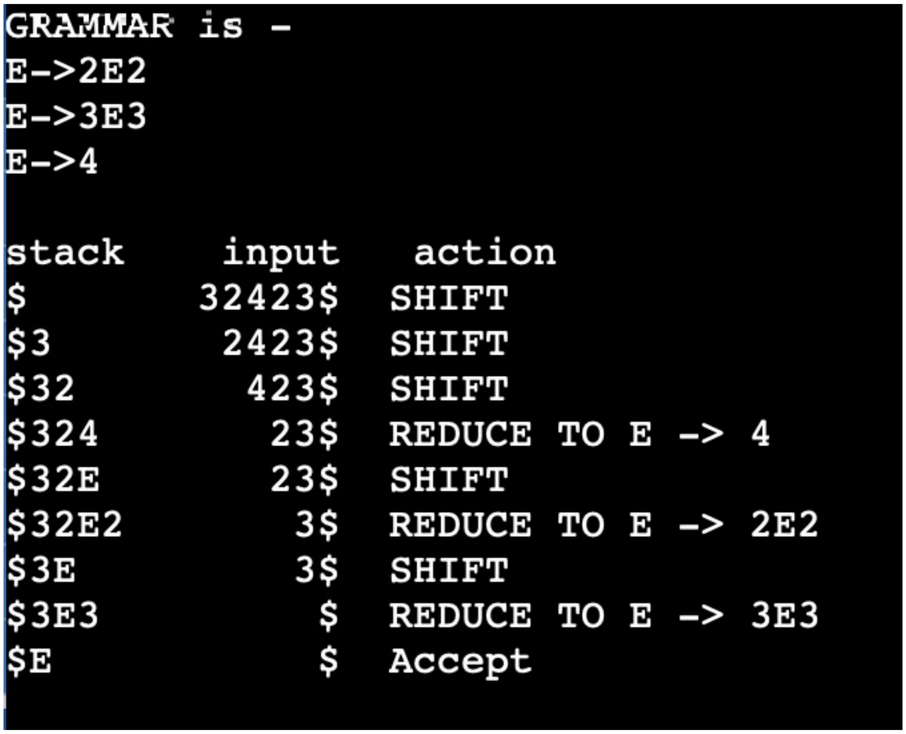


```
The following grammar is used for Parsing Table:
S->A
A->Bb
A->Cd
B->aB
B->@
C->Cc
C->@

Predictive parsing table:

--------------------------------------------------------
          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->@
--------------------------------------------------------


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

LAB 7

Aim - Write a program for Shift Reduce Parsing of given productions

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

int z = 0, i = 0, j = 0, c = 0;

char a[16], ac[20], stk[15], act[10];

void check()
{
	strcpy(ac,"REDUCE TO E -> ");

	for(z = 0; z < c; z++)
	{
		if(stk[z] == '4')
		{
			printf("%s4", ac);
			stk[z] = 'E';
			stk[z + 1] = '\0';

			printf("\n$%s\t%s$\t", stk, a);
		}
	}

	for(z = 0; z < c - 2; z++)
	{
		if(stk[z] == '2' && stk[z + 1] == 'E' &&
							stk[z + 2] == '2')
		{
			printf("%s2E2", ac);
			stk[z] = 'E';
			stk[z + 1] = '\0';
			stk[z + 2] = '\0';
			printf("\n$%s\t%s$\t", stk, a);
			i = i - 2;
		}

	}

	for(z = 0; z < c - 2; z++)
	{
		if(stk[z] == '3' && stk[z + 1] == 'E' &&
                                stk[z + 2] == '3')
        {
            printf("%s3E3", ac);
            stk[z]='E';
            stk[z + 1]='\0';
            stk[z + 2]='\0';
            printf("\n$%s\t%s$\t", stk, a);
            i = i - 2;
        }
    }
    return ;
}

int main()
{
    printf("GRAMMAR is -\nE->2E2 \nE->3E3 \nE->4\n");

    strcpy(a,"32423");

    c=strlen(a);

    strcpy(act,"SHIFT");

    printf("\nstack \t input \t action");

    printf("\n$\t%s$\t", a);

    for(i = 0; j < c; i++, j++)
    {
        printf("%s", act);

        stk[i] = a[j];
        stk[i + 1] = '\0';

        a[j]=' ';

        printf("\n$%s\t%s$\t", stk, a);

        check();
    }

    check();


    if(stk[0] == 'E' && stk[1] == '\0')
        printf("Accept\n");
        else
                printf("Reject\n");
}
```



```
GRAMMAR is -
E->2E2
E->3E3
E->4

stack     input    action
$        32423$   SHIFT
$3        2423$   SHIFT
$32        423$   SHIFT
$324        23$   REDUCE TO E -> 4
$32E        23$   SHIFT
$32E2        3$   REDUCE TO E -> 2E2
$3E          3$   SHIFT
$3E3          $   REDUCE TO E -> 3E3
$E            $   Accept
```

LAB 8

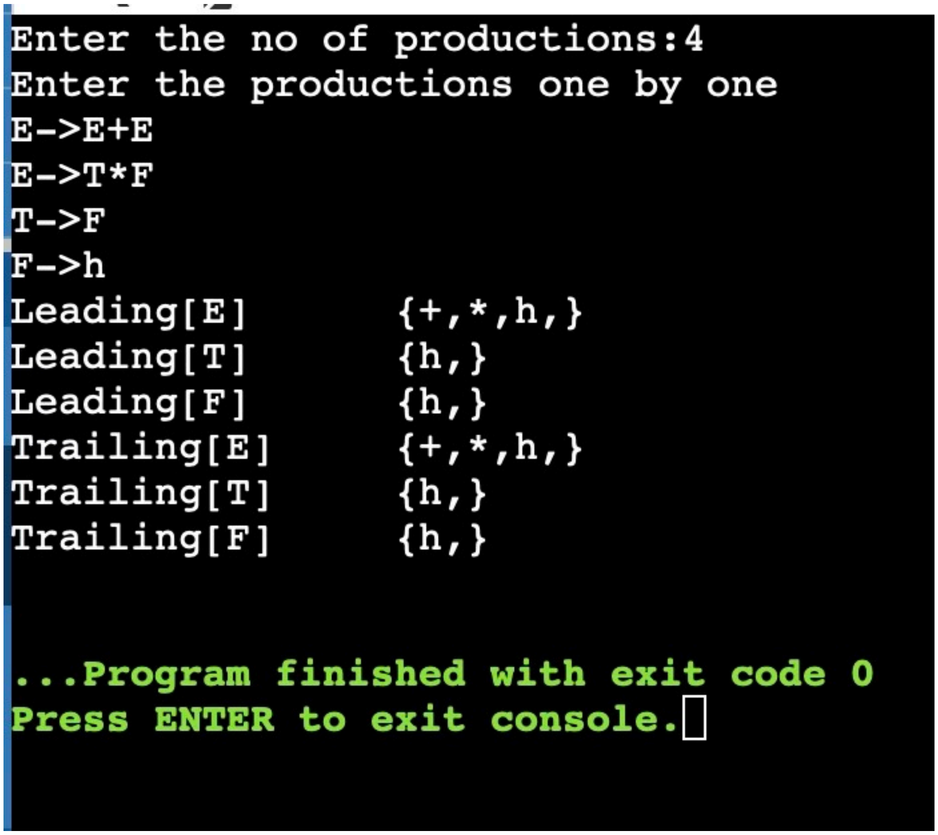
Aim - Computation Of Leading And Trailing

```
#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;}
```



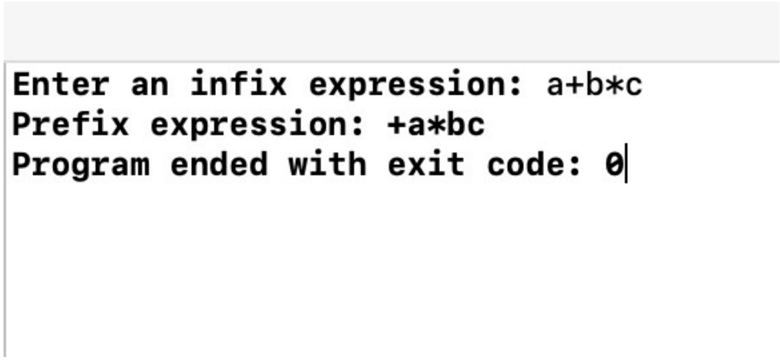
```
Enter the no of productions:4
Enter the productions one by one
E->E+E
E->T*F
T->F
F->h
Leading[E]      {+,*,h,}
Leading[T]      {h,}
Leading[F]      {h,}
Trailing[E]     {+,*,h,}
Trailing[T]     {h,}
Trailing[F]     {h,}


...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;
}
```



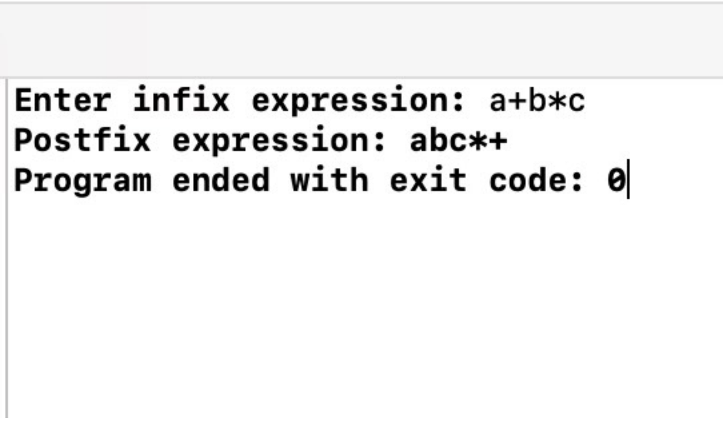
```
Enter an infix expression: a+b*c
Prefix expression: +a*bc
Program ended with exit code: 0
```

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
```