GANGISETTY RAJ CHARAN
RA2011028010007

EXPERIMENT-1 Implementation of Lexical Analyser

AIM: To write a C/C++/Java program to implement lexical analyser.

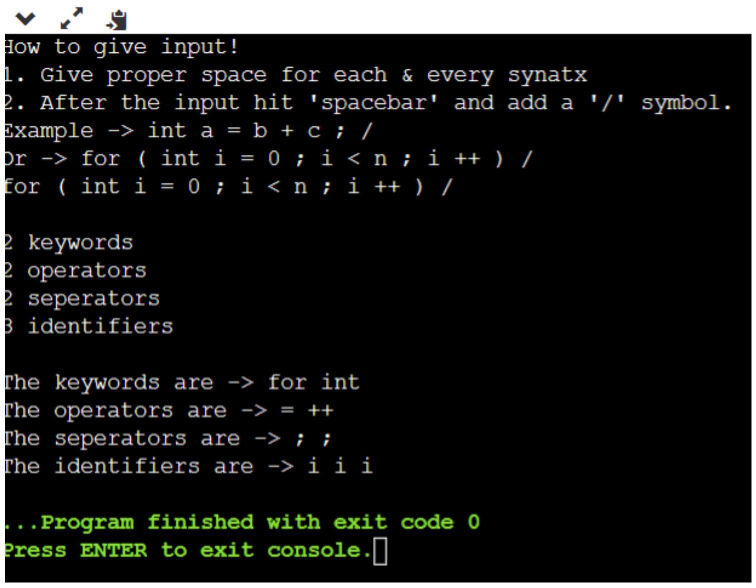
Algorithm:

1. Start
2. Get the input program for source code.
3. Read the program line by line and check if each word in a line is a keyword/operator/identifier/symbol.
4. For each lexeme, read and generate a token as follows:
    a. If the lexeme is a keyword /operator/symbol the token s the lexeme itself. Print it on the console.
    b. If the lexeme is an identifier, print the token generated from it in the console.
5. The stream of the tokens under various categories are displayed in the console.
6. Stop.

CODE:

```
#include <bits/stdc++.h>
using namespace std;
vector<string> keys;
vector<string> oper;
vector<string> ident;
vector<string> syms;
vector<string> to_vector(string input)
{
    vector<string> temp;
    int k = 0;
    for (int i = 0; i < input.length(); i++)
    {
        string part = "";
        if (input[i] == ' ')
        {
            for (int j = k; j < i; j++)
            {
                part += input[j];
            }
            k = i + 1;
            temp.push_back(part);
        }
    }
    return temp;
}
int keyword_count(vector<string> parts)
{
    int count = 0;
    for (int i = 0; i < parts.size(); i++)
    {
         if (parts[i] == "int" || parts[i] == "float" || parts[i] == "while" ||
parts[i] == "for")
        {
            count++;
            keys.push_back(parts[i]);
        }
    }
    return count;
}
int identifier_count(vector<string> parts)
{
    int count = 0;
    for (int i = 0; i < parts.size(); i++)
    {
         if (parts[i] == "a" || parts[i] == "b" || parts[i] == "c" || parts[i]
== "i" || parts[i] == "j")
        {
            count++;
            ident.push_back(parts[i]);
        }
    }
    return count;
}
int operator_count(vector<string> parts)
{
    int count = 0;
    for (int i = 0; i < parts.size(); i++)
    {
         if (parts[i] == "+" || parts[i] == "++" || parts[i] == "-" || parts[i]
== "--" || parts[i] == "=")
        {
            count++;
            oper.push_back(parts[i]);
        }
    }
    return count;
}
int seperator_count(vector<string> parts)
{
    int count = 0;
    for (int i = 0; i < parts.size(); i++)
    {
        if (parts[i] == ";" || parts[i] == "//")
        {
            count++;
            syms.push_back(parts[i]);
        }
    }
    return count;
}
int main() {
  string input;

  cout << "How to give input!" << endl;
  cout << "1. Give proper space for each & every synatx" << endl;
  cout << "2. After the input hit 'spacebar' and add a '/' symbol." << endl;
  cout << "Example -> int a = b + c ; /" << endl;
  cout << "Or -> for ( int i = 0 ; i < n ; i ++ ) /" << endl;
  getline(cin, input);
  cout << endl;
  vector < string > parts;
  parts = to_vector(input);
  int key, op, sep, id;
  key = keyword_count(parts);
  op = operator_count(parts);
  sep = seperator_count(parts);
  id = identifier_count(parts);
  cout << key << " keywords" << endl;
  cout << op << " operators" << endl;
  cout << sep << " seperators" << endl;
  cout << id << " identifiers" << endl;
  cout << endl << "The keywords are -> ";
  for (int i = 0; i < keys.size(); i++) {
    cout << keys[i] << " ";
  }
  cout << endl << "The operators are -> ";
  for (int i = 0; i < oper.size(); i++) {
    cout << oper[i] << " ";
  }
  cout << endl << "The seperators are -> ";
  for (int i = 0; i < syms.size(); i++) {
    cout << syms[i] << " ";
  }
  cout << endl << "The identifiers are -> ";
  for (int i = 0; i < ident.size(); i++) {
    cout << ident[i] << " ";
  }
}
```

OUTPUT:



```
How to give input!
1. Give proper space for each & every synatx
2. After the input hit 'spacebar' and add a '/' symbol.
Example -> int a = b + c ; /
Or -> for ( int i = 0 ; i < n ; i ++ ) /
for ( int i = 0 ; i < n ; i ++ ) /

2 keywords
2 operators
2 seperators
3 identifiers

The keywords are -> for int
The operators are -> = ++
The seperators are -> ; ;
The identifiers are -> i i i

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

Result: Thus, the C/C++/Java program to implement the lexical analyser is completed successfully

GANGISETTY RAJ CHARAN
RA2011028010007

EXPERIMENT-2 program to construct a Non Deterministic Finite Automata (NFA) from Regular Expression using C/ C++/ Java

AIM: To Write A Program to construct a NFA from Regex.

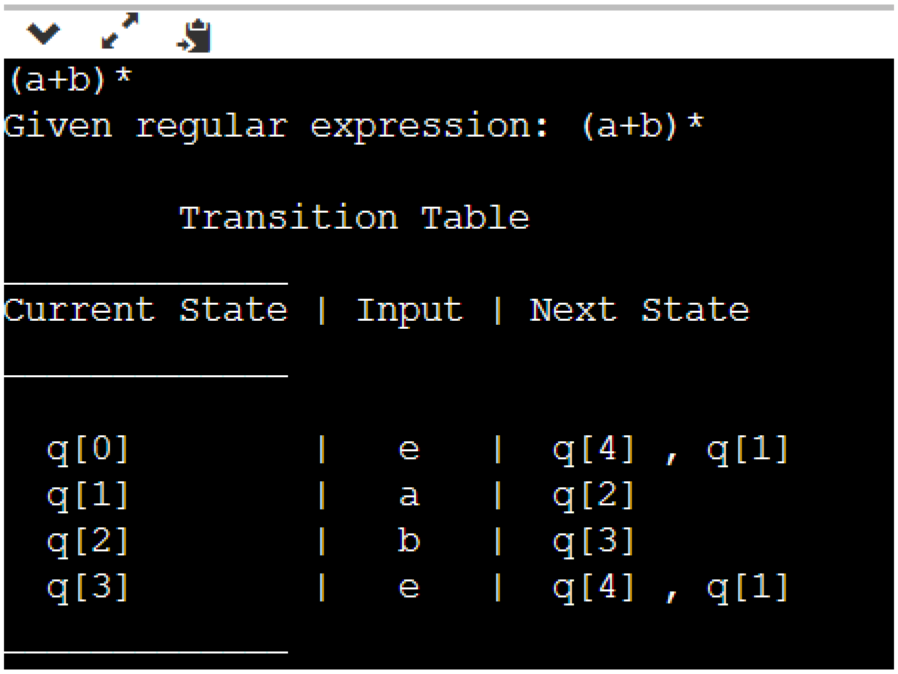
Algorithm:
1. Start
2. Get the input from the user
3. Initialize separate variables and functions for Postfix , Display and NFA
4. Create separate methods for different operators like +,*, .
5. By using Switch case Initialize different cases for the input
6. For ' . ' operator Initialize a separate method by using various stack functions do the same for the other operators like ' * ' and ' + '.
7. Regular expression is in the form like a.b (or) a+b.
8. Display the output
9. Stop

Code:

```
#include<stdio.h>
#include<string.h>
int main()
{
        char reg[20]; int q[20][3],i=0,j=1,len,a,b;
        for(a=0;a<20;a++) for(b=0;b<3;b++) q[a][b]=0;
        scanf("%s",reg);
        printf("Given regular expression: %s\n",reg);
        len=strlen(reg);
        while(i<len)
        {
                if(reg[i]=='a'&&reg[i+1]!='|'&&reg[i+1]!='*') { q[j][0]=j+1; j++; }
                if(reg[i]=='b'&&reg[i+1]!='|'&&reg[i+1]!='*') {   q[j][1]=j+1; j++;          }
                if(reg[i]=='e'&&reg[i+1]!='|'&&reg[i+1]!='*') {   q[j][2]=j+1; j++;          }
                if(reg[i]=='a'&&reg[i+1]=='|'&&reg[i+2]=='b')
                {
                  q[j][2]=((j+1)*10)+(j+3); j++;
                  q[j][0]=j+1; j++;
                        q[j][2]=j+3; j++;
                        q[j][1]=j+1; j++;
                        q[j][2]=j+1; j++;
                        i=i+2;
                }
                if(reg[i]=='b'&&reg[i+1]=='|'&&reg[i+2]=='a')
                {
```

```
                q[j][2]=((j+1)*10)+(j+3); j++;
                q[j][1]=j+1; j++;
                q[j][2]=j+3; j++;
                q[j][0]=j+1; j++;
                q[j][2]=j+1; j++;
                i=i+2;
            }
            if(reg[i]=='a'&&reg[i+1]=='*')
            {
                q[j][2]=((j+1)*10)+(j+3); j++;
                q[j][0]=j+1; j++;
                q[j][2]=((j+1)*10)+(j-1); j++;
            }
            if(reg[i]=='b'&&reg[i+1]=='*')
            {
                q[j][2]=((j+1)*10)+(j+3); j++;
                q[j][1]=j+1; j++;
                q[j][2]=((j+1)*10)+(j-1); j++;
            }
            if(reg[i]==')'&&reg[i+1]=='*')
            {
                q[0][2]=((j+1)*10)+1;
                q[j][2]=((j+1)*10)+1;
                j++;
            }
            i++;
        }
        printf("\n\tTransition Table \n");
        printf("_____________\n");
        printf("Current State |\tInput |\tNext State");
        printf("\n_____________\n");
        for(i=0;i<=j;i++)
        {
            if(q[i][0]!=0) printf("\n  q[%d]\t      |   a   |  q[%d]",i,q[i][0]);
            if(q[i][1]!=0) printf("\n  q[%d]\t      |   b   |  q[%d]",i,q[i][1]);
            if(q[i][2]!=0)
            {
                if(q[i][2]<10) printf("\n  q[%d]\t      |   e   |  q[%d]",i,q[i][2]);
                else printf("\n  q[%d]\t      |   e   |  q[%d] , q[%d]",i,q[i][2]/10,q[i][2]%10);
            }
        }
        printf("\n_____________\n");
        return 0;
}
```

Output:



```
(a+b)*
Given regular expression: (a+b)*

        Transition Table
_____________
Current State | Input | Next State
_____________

  q[0]        |   e   |  q[4] , q[1]
  q[1]        |   a   |  q[2]
  q[2]        |   b   |  q[3]
  q[3]        |   e   |  q[4] , q[1]
_____________
```

GANGISETTY RAJ CHARAN
RA2011028010007

EXPERIMENT-3 **Conversion from NFA to DFA**

AIM: To Convert NFA To DFA

ALGORITHM:

The algorithm outlines the process for constructing a deterministic finite automaton (DFA) from a non-deterministic finite automaton (NFA). The steps are as follows

- Start the algorithm
- Obtain input from the user
- Establish "unmarked" as the only state in the SDFA
- Continue with the following procedure as long as there are unmarked states in the SDFA:
- Identify the next unmarked state (T)
- For each symbol (a) in the alphabet, determine the result of the MoveNFA operation, which results in a set of states (S). Add these states to the SDFA if they are not already present, and label them as "unmarked." Then, set the result of the MoveDFA operation for state T and symbol a.
- Mark any final states in the NFA that are present in the SDFA as final states in the DFA
- Present the outcome of the algorithm
- Terminate the application.

CODE:

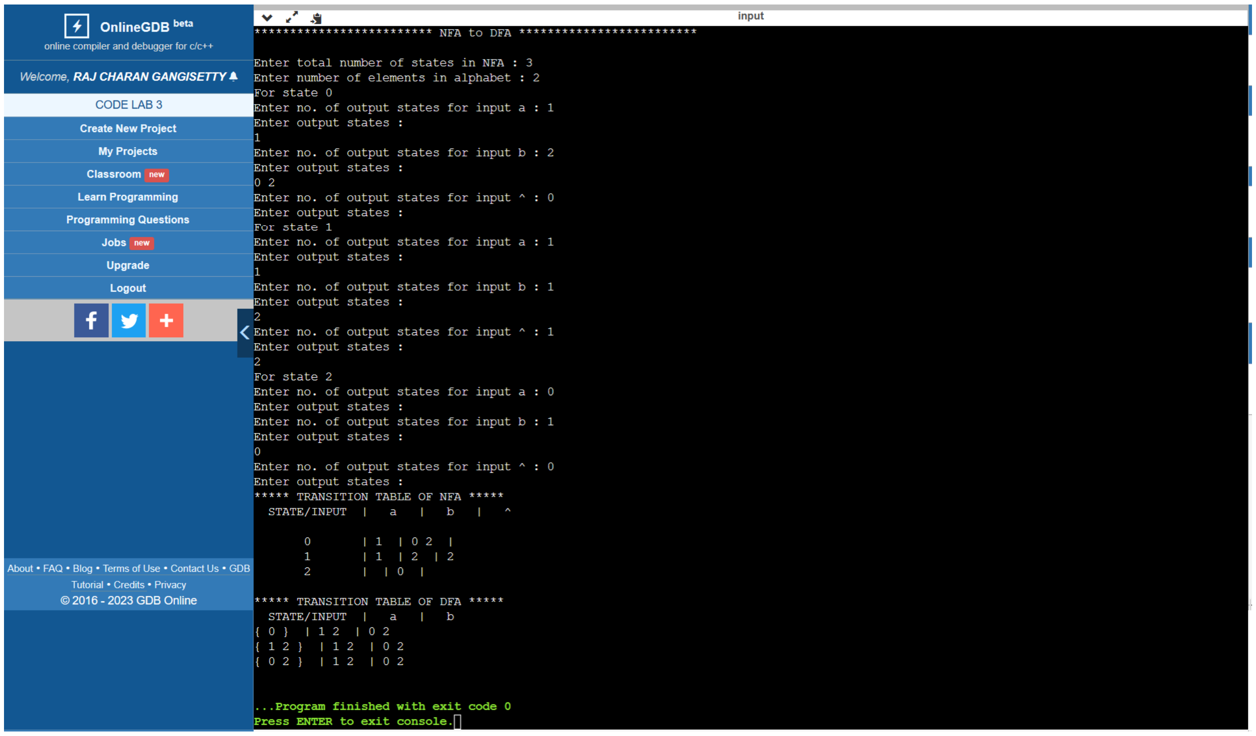
```
#include<iostream>
#include<bits/stdc++.h>
using namespace std;
void print(vector<vector<vector<int> > > table){
    cout<<"  STATE/INPUT  |";
    char a='a';
    for(int i=0;i<table[0].size()-1;i++){
        cout<<"   "<<a++<<"   |";
    }
    cout<<"   ^   "<<endl<<endl;
```

```
	for(int i=0;i<table.size();i++){
		cout<<"      "<<i<<"     ";
		for(int j=0;j<table[i].size();j++){
			cout<<" | ";
			for(int k=0;k<table[i][j].size();k++){
				cout<<table[i][j][k]<<" ";


			}
		}
		cout<<endl;
	}
}

void printdfa(vector<vector<int> > states, vector<vector<vector<int> > > dfa){
	cout<<"  STATE/INPUT  ";
	char a='a';
	for(int i=0;i<dfa[0].size();i++){
		cout<<"|   "<<a++<<"   ";
	}
	cout<<endl;
	for(int i=0;i<states.size();i++){
		cout<<"{ ";
		for(int h=0;h<states[i].size();h++)
			cout<<states[i][h]<<" ";
		if(states[i].empty()){
			cout<<"^ ";
		}
		cout<<"} ";
		for(int j=0;j<dfa[i].size();j++){
			cout<<" | ";
			for(int k=0;k<dfa[i][j].size();k++){
				cout<<dfa[i][j][k]<<" ";
```

```
                }
                if(dfa[i][j].empty()){
                        cout<<"^ ";
                }
            }
            cout<<endl;
        }
}
vector<int> closure(int s,vector<vector<vector<int> > > v){
        vector<int> t;
        queue<int> q;
        t.push_back(s);
        int a=v[s][v[s].size()-1].size();
        for(int i=0;i<a;i++){
                t.push_back(v[s][v[s].size()-1][i]);
                //cout<<"t[i]"<<t[i]<<endl;
                q.push(t[i]);
        }
        while(!q.empty()){
                int f=q.front();
                q.pop();
                if(!v[f][v[f].size()-1].empty()){
                        int u=v[f][v[f].size()-1].size();
                        for(int i=0;i<u;i++){
                                int y=v[f][v[f].size()-1][i];
                                if(find(t.begin(),t.end(),y)==t.end()){
                                        //cout<<"y"<<y<<endl;
                                        t.push_back(y);
                                        q.push(y);
                                }
                        }
            }
        }
        return t;
}
int main(){
        int n,alpha;
        cout<<"************************ NFA to DFA ************************"<<endl<<endl;
        cout<<"Enter total number of states in NFA : ";
        cin>>n;
        cout<<"Enter number of elements in alphabet : ";
        cin>>alpha;
        vector<vector<vector<int> > > table;
        for(int i=0;i<n;i++){
                cout<<"For state "<<i<<endl;
                vector< vector< int > > v;
                char a='a';
                int y,yn;
                for(int j=0;j<alpha;j++){
                        vector<int> t;
                        cout<<"Enter no. of output states for input "<<a++<<" : ";
                        cin>>yn;
                        cout<<"Enter output states :"<<endl;
                        for(int k=0;k<yn;k++){
                                cin>>y;
                                t.push_back(y);
                        }
                        v.push_back(t);
                }
                vector<int> t;
                cout<<"Enter no. of output states for input ^ : ";
                cin>>yn;
                cout<<"Enter output states :"<<endl;
                for(int k=0;k<yn;k++){
                        cin>>y;
                        t.push_back(y);
                }
                v.push_back(t);
                table.push_back(v);
        }
        cout<<"***** TRANSITION TABLE OF NFA *****"<<endl;
        print(table);
        cout<<endl<<"***** TRANSITION TABLE OF DFA *****"<<endl;
        vector<vector<vector<int> > > dfa;
        vector<vector<int> > states;
        states.push_back(closure(0,table));
        queue<vector<int> > q;
        q.push(states[0]);
        while(!q.empty()){
                vector<int> f=q.front();
                q.pop();
                vector<vector<int> > v;
                for(int i=0;i<alpha;i++){
                        vector<int> t;
                        set<int> s;
                        for(int j=0;j<f.size();j++){

                                for(int k=0;k<table[f[j]][i].size();k++){
                                        vector<int> cl= closure(table[f[j]][i][k],table);
                                        for(int h=0;h<cl.size();h++){
                                                if(s.find(cl[h])==s.end())
                                                s.insert(cl[h]);
                                        }
                                }
                        }
```

```
                    for(set<int >::iterator u=s.begin(); u!=s.end();u++)
                              t.push_back(*u);
                    v.push_back(t);
                    if(find(states.begin(),states.end(),t)==states.end())
                    {
                              states.push_back(t);
                              q.push(t);
                    }
          }
          dfa.push_back(v);
     }
     printdfa(states,dfa);
}
```

OUTPUT:



```
************************* NFA to DFA *************************

Enter total number of states in NFA : 3
Enter number of elements in alphabet : 2
For state 0
Enter no. of output states for input a : 1
Enter output states :
1
Enter no. of output states for input b : 2
Enter output states :
0 2
Enter no. of output states for input ^ : 0
Enter output states :
For state 1
Enter no. of output states for input a : 1
Enter output states :
1
Enter no. of output states for input b : 1
Enter output states :
2
Enter no. of output states for input ^ : 1
Enter output states :
2
For state 2
Enter no. of output states for input a : 0
Enter output states :
Enter no. of output states for input b : 1
Enter output states :
0
Enter no. of output states for input ^ : 0
Enter output states :
***** TRANSITION TABLE OF NFA *****
  STATE/INPUT  |   a   |   b   |   ^

       0       | 1  | 0 2  |
       1       | 1  | 2  | 2
       2       |  | 0  |

***** TRANSITION TABLE OF DFA *****
  STATE/INPUT  |   a   |   b
{ 0 }  | 1 2  | 0 2
{ 1 2 }  | 1 2  | 0 2
{ 0 2 }  | 1 2  | 0 2


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

RESULT: The implementation of converting NFA to DFA in CPP was successfully compiled, carried out, and verified.

EXPERIMENT-04

AIM:To implement The Elimination of Left recursion, Left factoring

Algorithm:

1) First:
    a) To find the first() of the grammar symbol, then we have to apply the following set of rules to the given grammar:-
    b) If X is a terminal, then First(X) is {X}.
    c) If X is a non-terminal and X tends to aα is production, then add 'a' to the first of X. if X->ε, then add null to the First(X).
    d) If X_>YZ then if First(Y)=ε, then First(X) = { First(Y)-ε} U First(Z).
    e) If X->YZ, then if First(X)=Y, then First(Y)=teminal but null then First(X)=First(Y)=terminals.
2) Follow:
    a) To find the follow(A) of the grammar symbol, then we have to apply the following set of rules to the given grammar:-
    b) $ is a follow of 'S'(start symbol).
    c) If A->αBβ,β!=ε, then first(β) is in follow(B). • If A->αB or A->αBβ where First(β)=ε, then everything in Follow(A) is a Follow(B)

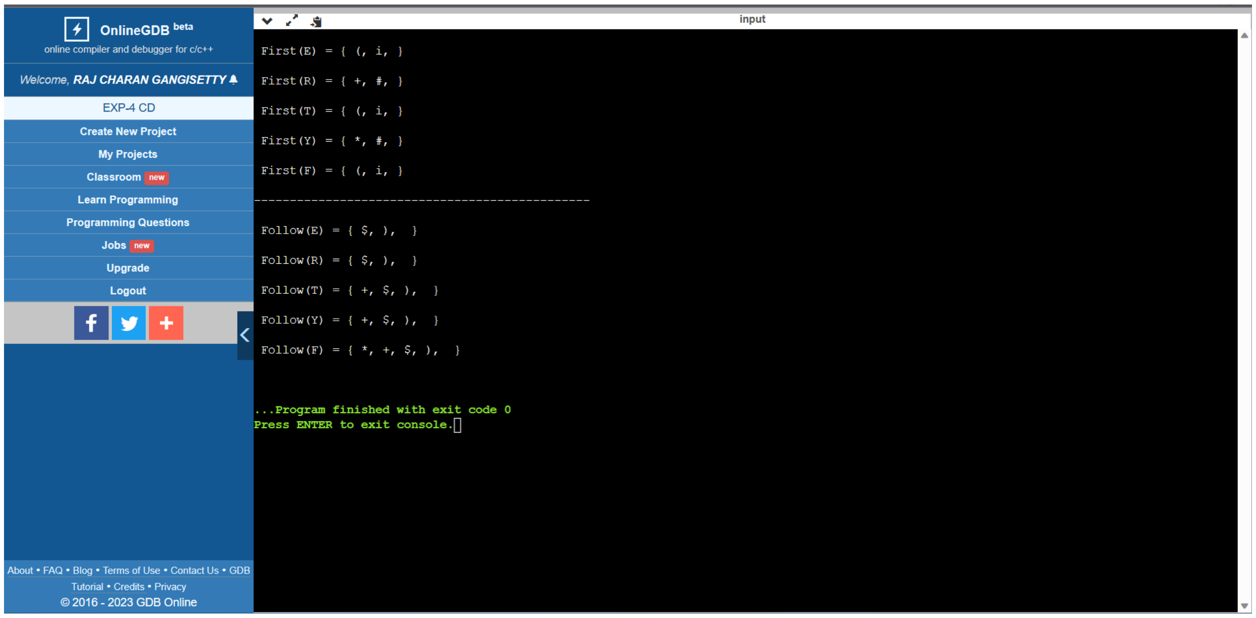
Code:

```
// C program to calculate the First and
// Follow sets of a given grammar
#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];
```

```c
        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 alredy 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++;
            }
        }
}
```

OUTPUT:



```
 First(E) = { (, i, }

 First(R) = { +, #, }

 First(T) = { (, i, }

 First(Y) = { *, #, }

 First(F) = { (, i, }

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

 Follow(E) = { $, ),  }

 Follow(R) = { $, ),  }

 Follow(T) = { +, $, ),  }

 Follow(Y) = { +, $, ),  }

 Follow(F) = { *, +, $, ),  }


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

GANGISETTY RAJ CHARAN
RA2011028010007

EXPERIMENT-05

Aim: To write a program to implement FIRST and FOLLOW in c.

Algorithm:

First:

To find the first() of the grammar symbol, then we have to apply the following set of rules to the given grammar:-

1) If X is a terminal, then First(X) is {X}.
2) If X is a non-terminal and X tends to aα is production, then add ‘a’ to the first of X.
3) if X->ε, then add null to the First(X).
4) If X_>YZ then if First(Y)=ε, then First(X) = { First(Y)-ε} U First(Z). • If X->YZ, then if First(X)=Y, then First(Y)=teminal but null then First(X)=First(Y)=terminals.

Follow:

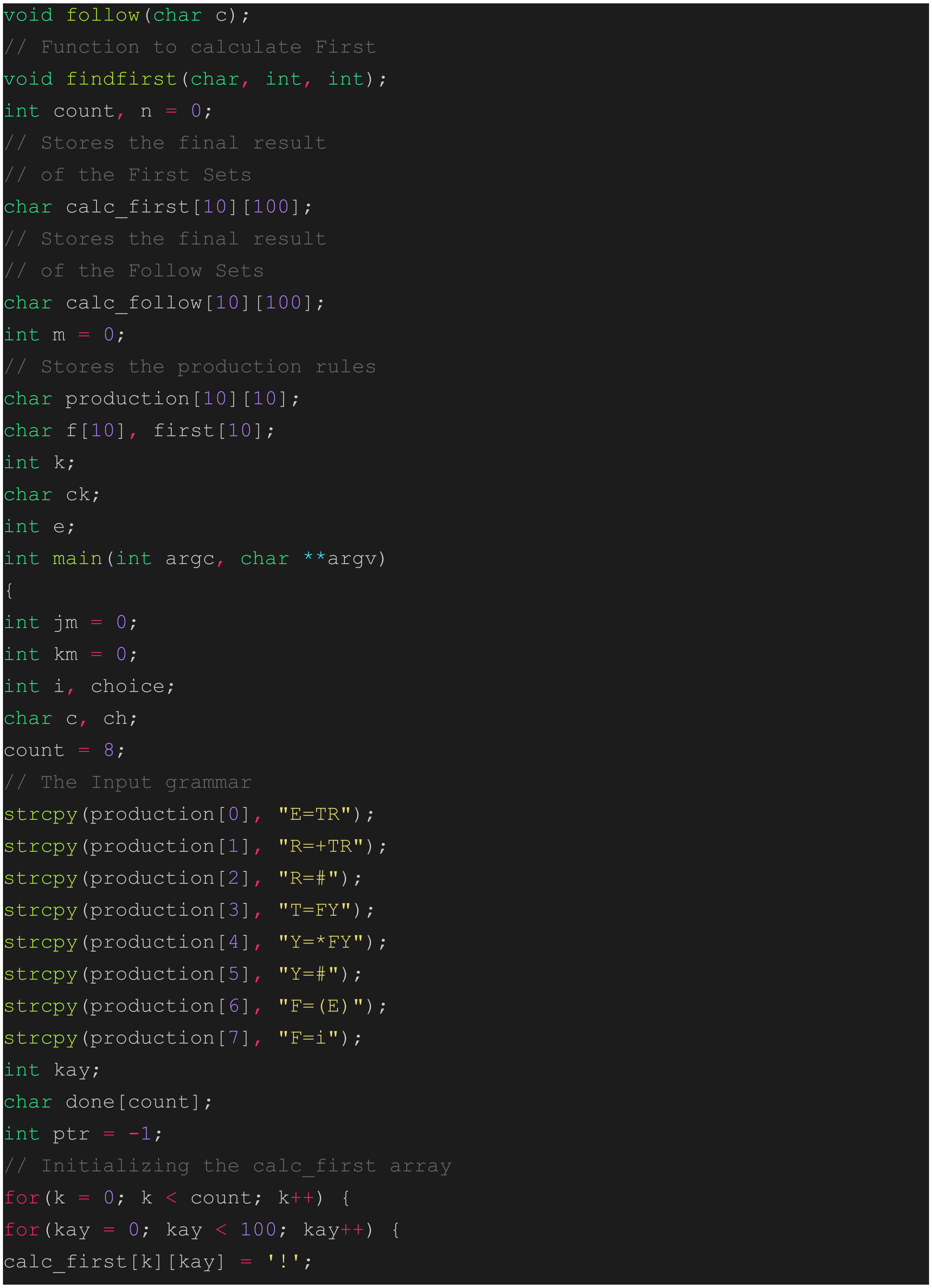
To find the follow(A) of the grammar symbol, then we have to apply the following set of rules to the given grammar:-

1) $ is a follow of ‘S’(start symbol).
2) If A->αBβ,β!=ε, then first(β) is in follow(B).
3) If A->αB or A->αBβ where First(β)=ε, then everything in Follow(A) is a Follow(B)

Code:

```
// C program to calculate the First and
// Follow sets of a given grammar
#include<stdio.h>
#include<ctype.h>
#include<string.h>
// Functions to calculate Follow
void followfirst(char, int, int);
```

GANGISETTY RAJ CHARAN
RA2011028010007

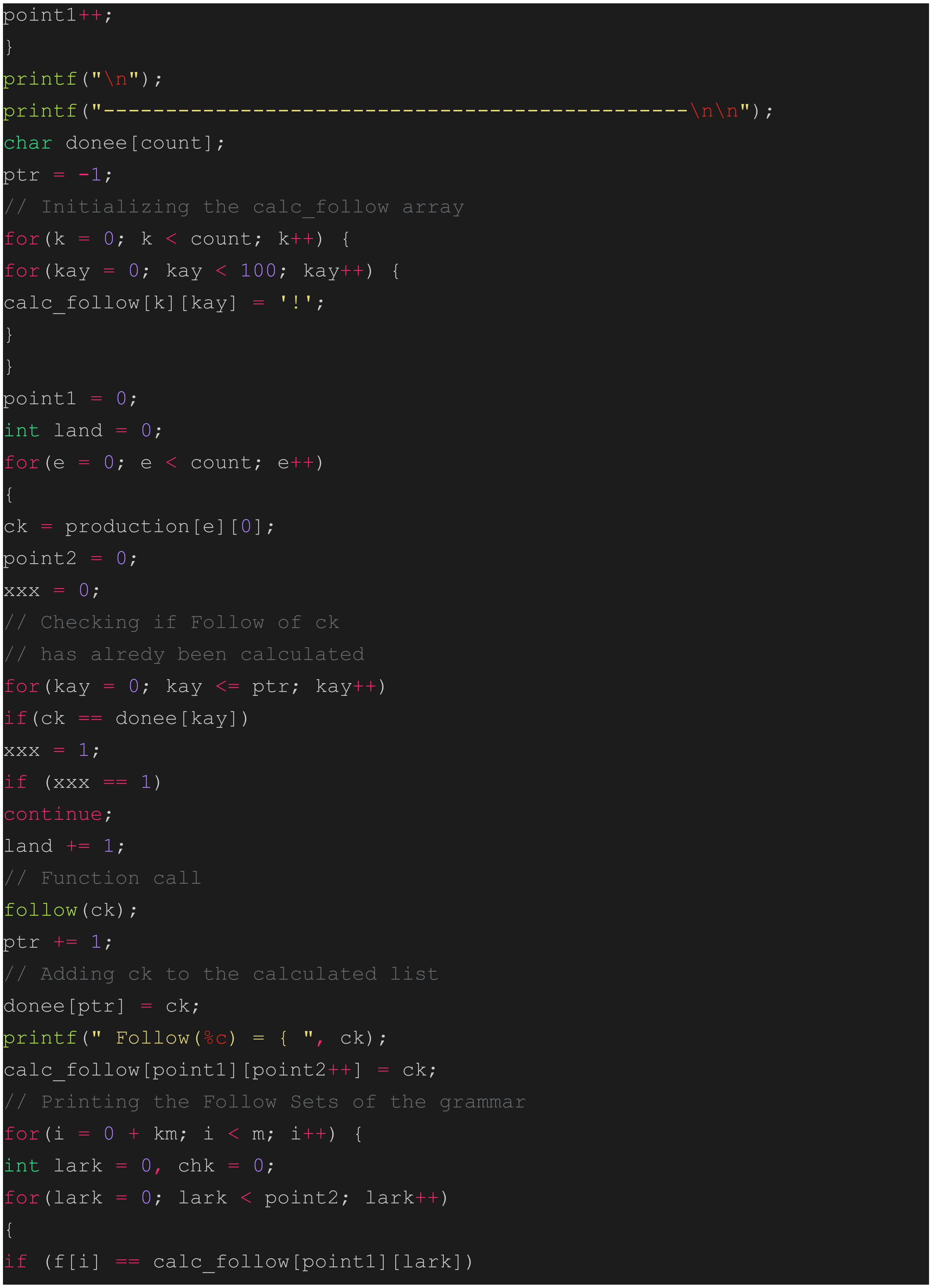


```c
void follow(char c);
// Function to calculate First
void findfirst(char, int, int);
int count, n = 0;
// Stores the final result
// of the First Sets
char calc_first[10][100];
// Stores the final result
// of the Follow Sets
char calc_follow[10][100];
int m = 0;
// Stores the production rules
char 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] = '!';
```

GANGISETTY RAJ CHARAN
RA2011028010007

```
}
}
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;
```

GANGISETTY RAJ CHARAN
RA2011028010007



```c
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 alredy 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])
```

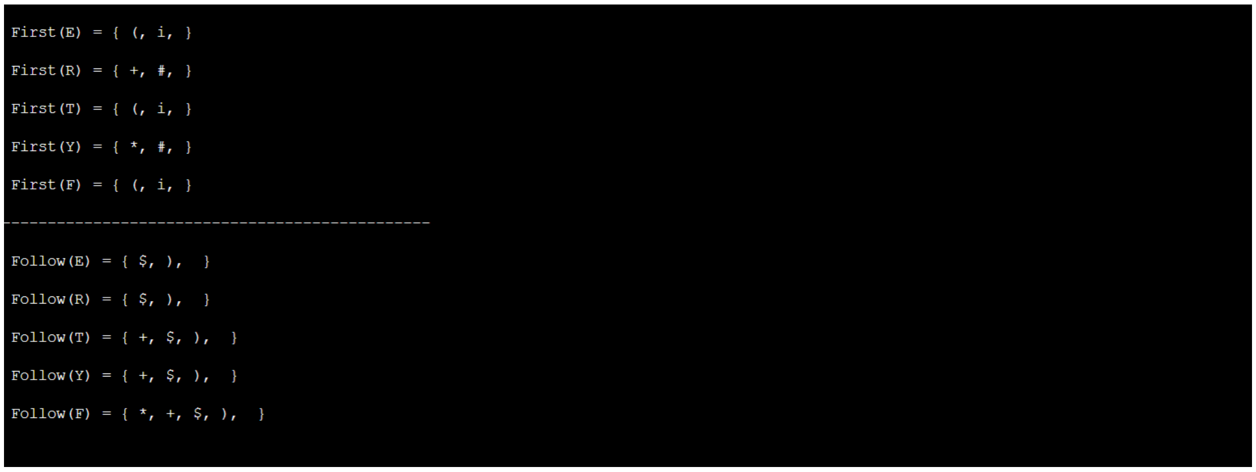
```
{
chk = 1;
break;
}
}
if(chk == 0)
{
printf("%c, ", f[i]);
calc_follow[point1][point2++] = f[i];
}
}
printf(" }\n\n");
km = m;
point1++;
}
}
void follow(char c)
{
int i, j;
// Adding "$" to the follow
// set of the start symbol
if(production[0][0] == c) {
f[m++] = '$';
}
for(i = 0; i < 10; i++)
{
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
```

GANGISETTY RAJ CHARAN
RA2011028010007

```
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++;
}
}
}
```

OUTPUT:



```
First(E) = { (, i, }

First(R) = { +, #, }

First(T) = { (, i, }

First(Y) = { *, #, }

First(F) = { (, i, }

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

Follow(E) = { $, ),  }

Follow(R) = { $, ),  }

Follow(T) = { +, $, ),  }

Follow(Y) = { +, $, ),  }

Follow(F) = { *, +, $, ),  }
```

RESULT:The FIRST and FOLLOW sets of the non-terminals of a grammar were found successfully using python language

GANGISETTY RAJ CHARAN
RA2011028010007

EXPERIMENT-06
AIM: To write a program for Predictive Parsing table.

ALGORITHM:
For the production A → α of Grammar G:
1) For each terminal, a in FIRST (α) add A → α to M [A, a].
2) If ε is in FIRST (α), and b is in FOLLOW (A), then add A → α to M[A, b].
3) If ε is in FIRST (α), and $ is in FOLLOW (A), then add A → α to M[A, $].
4) All remaining entries in Table M are errors.

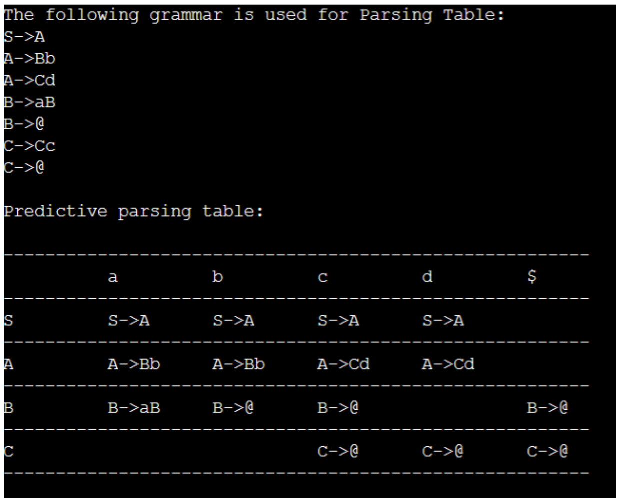
CODE:

```
#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], " ");
```

GANGISETTY RAJ CHARAN
RA2011028010007

```
 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")
;
 }
}
```

Output:



```
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->@
--------------------------------------------------------
```

RESULT:The implementation of predictive parse table was executed successfully.

GANGISETTY RAJ CHARAN
RA2011028010007

EXPERIMENT-07

AIM: To write a program to implement Lexical Analysis using C.

Algorithm:

1) Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.
2) Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string.
3) Sift reduce parsing performs the two actions: shift and reduce. That's why it is known as shift reduces parsing.
4) At the shift action, the current symbol in the input string is pushed to a stack.
5) At each reduction, the symbols will replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production.

CODE:

```c
#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 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];
```

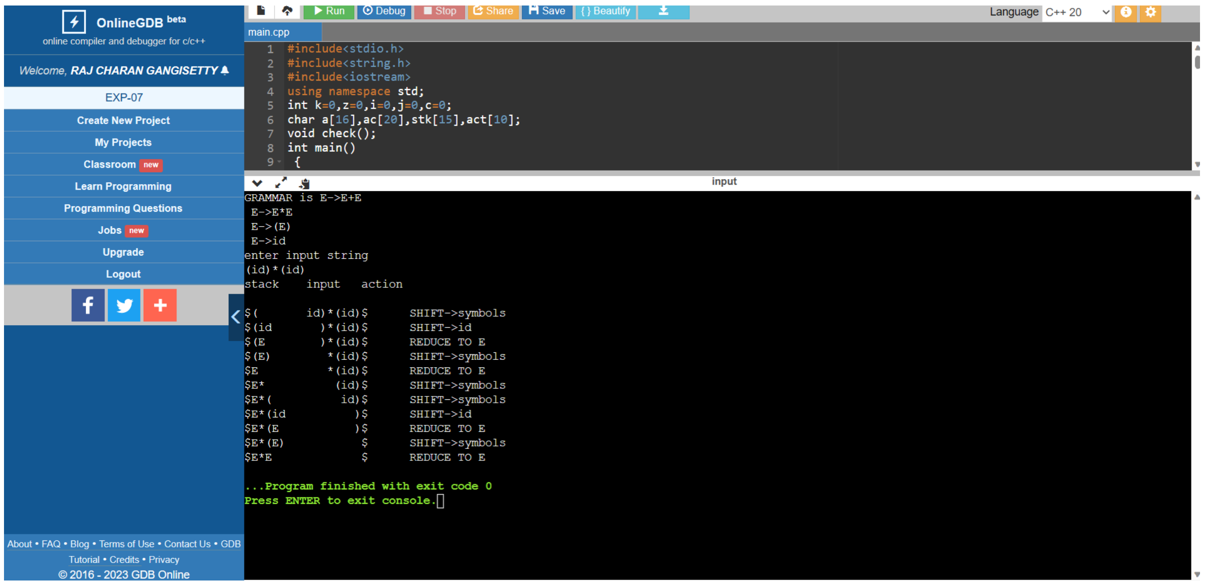
GANGISETTY RAJ CHARAN
RA2011028010007

```
 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();
 }
 }
 }
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';
```

GANGISETTY RAJ CHARAN
RA2011028010007

```
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:



```
#include<stdio.h>
#include<string.h>
#include<iostream>
using namespace std;
int k=0,z=0,i=0,j=0,c=0;
char a[16],ac[20],stk[15],act[10];
void check();
int main()
 {
```

```
GRAMMAR is E->E+E
 E->E*E
 E->(E)
 E->id
enter input string
(id)*(id)
stack    input   action

$(       id)*(id)$      SHIFT->symbols
$(id       )*(id)$      SHIFT->id
$(E        )*(id)$      REDUCE TO E
$(E)        *(id)$      SHIFT->symbols
$E          *(id)$      REDUCE TO E
$E*          (id)$      SHIFT->symbols
$E*(          id)$      SHIFT->symbols
$E*(id          )$      SHIFT->id
$E*(E           )$      REDUCE TO E
$E*(E)           $      SHIFT->symbols
$E*E             $      REDUCE TO E

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

Result: The implementation of shift reduce parsing was executed and verified successfully.

EXPERIMENT-08

AIM:To compute leading and trailing.

ALGORITHM:

1) For Leading, check for the first non-terminal.
2) If found, print it.
3) Look for next production for the same non-terminal.
4) If not found, recursively call the procedure for the single non-terminal present before the comma or End Of Production String.
5) Include it's results in the result of this non-terminal.
6) For trailing, we compute same as leading but we start from the end of the production to the beginning.
7) Stop Program

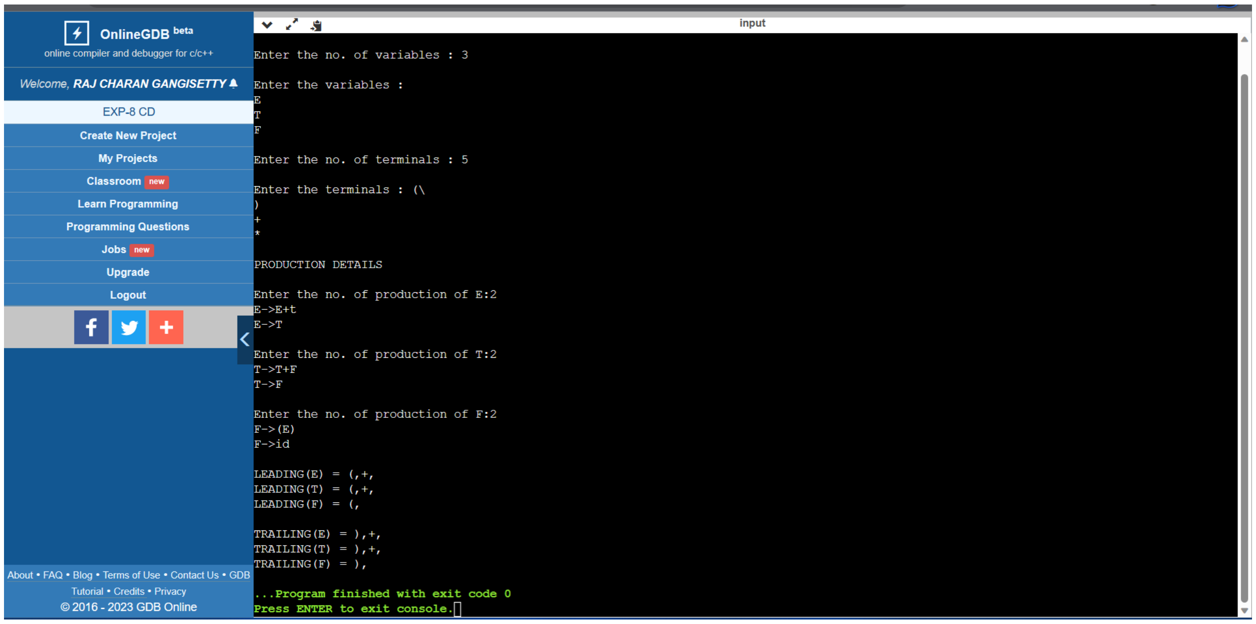
CODE:

```
#include <iostream>
#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
using namespace std;
int vars, terms, i, j, k, m, rep, count, temp = -1;
char var[10], term[10], lead[10][10], trail[10][10];
struct grammar
{
    int prodno;
    char lhs, rhs[20][20];
} gram[50];
void get()
{
    cout << "\nLEADING AND TRAILING\n";
    cout << "\nEnter the no. of variables : ";
    cin >> vars;
    cout << "\nEnter the variables : \n";
```

```cpp
    for (i = 0; i < vars; i++)
    {
        cin >> gram[i].lhs;
        var[i] = gram[i].lhs;
    }
    cout << "\nEnter the no. of terminals : ";
    cin >> terms;
    cout << "\nEnter the terminals : ";
    for (j = 0; j < terms; j++)
        cin >> term[j];
    cout << "\nPRODUCTION DETAILS\n";
    for (i = 0; i < vars; i++)
    {
        cout << "\nEnter the no. of production of " << gram[i].lhs << ":";
        cin >> gram[i].prodno;
        for (j = 0; j < gram[i].prodno; j++)
        {
            cout << gram[i].lhs << "->";
            cin >> gram[i].rhs[j];
        }
    }
}
void leading()
{
    for (i = 0; i < vars; i++)
    {
        for (j = 0; j < gram[i].prodno; j++)
        {
            for (k = 0; k < terms; k++)
            {
                if (gram[i].rhs[j][0] == term[k])
                    lead[i][k] = 1;
                else
                {
                    if (gram[i].rhs[j][1] == term[k])
                        lead[i][k] = 1;
                }
            }
        }
    }
    for (rep = 0; rep < vars; rep++)
    {
        for (i = 0; i < vars; i++)
        {
            for (j = 0; j < gram[i].prodno; j++)
            {
                for (m = 1; m < vars; m++)
                {
                    if (gram[i].rhs[j][0] == var[m])
                    {
                        temp = m;
                        goto out;
                    }
                }
            out:
                for (k = 0; k < terms; k++)
                {
                    if (lead[temp][k] == 1)
                        lead[i][k] = 1;
                }
            }
        }
    }
}
void trailing()
{
    for (i = 0; i < vars; i++)
    {
        for (j = 0; j < gram[i].prodno; j++)
        {
            count = 0;
            while (gram[i].rhs[j][count] != '\x0')
                count++;
            for (k = 0; k < terms; k++)
            {
                if (gram[i].rhs[j][count - 1] == term[k])
                    trail[i][k] = 1;
                else
                {
                    if (gram[i].rhs[j][count - 2] == term[k])
```

```
                        trail[i][k] = 1;
                }
            }
        }
    }
    for (rep = 0; rep < vars; rep++)
    {
        for (i = 0; i < vars; i++)
        {
            for (j = 0; j < gram[i].prodno; j++)
            {
                count = 0;
                while (gram[i].rhs[j][count] != '\x0')
                    count++;
                for (m = 1; m < vars; m++)
                {
                    if (gram[i].rhs[j][count - 1] == var[m])
                        temp = m;
                }
                for (k = 0; k < terms; k++)
                {
                    if (trail[temp][k] == 1)
                        trail[i][k] = 1;
                }
            }
        }
    }
}
void display()
{
    for (i = 0; i < vars; i++)
    {
        cout << "\nLEADING(" << gram[i].lhs << ") = ";
        for (j = 0; j < terms; j++)
        {
            if (lead[i][j] == 1)
                cout << term[j] << ",";
        }
    }
    cout << endl;
    for (i = 0; i < vars; i++)
    {
        cout << "\nTRAILING(" << gram[i].lhs << ") = ";
        for (j = 0; j < terms; j++)
        {
            if (trail[i][j] == 1)
                cout << term[j] << ",";
        }
    }
}
int main()
{
    get();
    leading();
    trailing();
    display();
}
```

OUTPUT:



```
Enter the no. of variables : 3

Enter the variables :
E
T
F

Enter the no. of terminals : 5

Enter the terminals : (\
)
+
*

PRODUCTION DETAILS

Enter the no. of production of E:2
E->E+t
E->T

Enter the no. of production of T:2
T->T+F
T->F

Enter the no. of production of F:2
F->(E)
F->id

LEADING(E) = (,+,
LEADING(T) = (,+,
LEADING(F) = (,

TRAILING(E) = ),+,
TRAILING(T) = ),+,
TRAILING(F) = ),

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

RESULT:SUCCESSFULLY COMPUTED LEADING AND TRAILING VALUES.

GANGISETTY RAJ CHARAN
RA2011028010007

EXPERIMENT-09

AIM: To write a program to implement LR(0) items.

ALGORITHM:

1) Start.
2) Create structure for production with LHS and RHS.
3) Open file and read input from file.
4) Build state 0 from extra grammar Law S' -> S $ that is all start symbol of grammar and one Dot (.) before S symbol.
5) If Dot symbol is before a non-terminal, add grammar laws that this nonterminal is in Left Hand Side of that Law and set Dot in before of first part of Right Hand Side.
6) If state exists (a state with this Laws and same Dot position), use that instead.
7) Now find set of terminals and non-terminals in which Dot exist in before.
8) If step 7 Set is non-empty go to 9, else go to 10.
9) For each terminal/non-terminal in set step 7 create new state by using all grammar law that Dot position is before of that terminal/non-terminal in reference state by increasing Dot point to next part in Right Hand Side of that laws.
10) Go to step 5.
11) End of state building.
12) Display the output.
13) End

PROGRAM:

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

using namespace std;
```

GANGISETTY RAJ CHARAN
RA2011028010007

```
char prod[20][20],listofvar[26]="ABCDEFGHIJKLMNOPQR";
int novar=1,i=0,j=0,k=0,n=0,m=0,arr[30];
int noitem=0;

struct Grammar
{
        char lhs;
        char rhs[8];
}g[20],item[20],clos[20][10];

int isvariable(char variable)
{
        for(int i=0;i<novar;i++)
                if(g[i].lhs==variable)
                        return i+1;
        return 0;
}
void findclosure(int z, char a)
{
        int n=0,i=0,j=0,k=0,l=0;
        for(i=0;i<arr[z];i++)
        {
                for(j=0;j<strlen(clos[z][i].rhs);j++)
                {
                        if(clos[z][i].rhs[j]=='.' && clos[z][i].rhs[j+1]==a)
                        {
                                clos[noitem][n].lhs=clos[z][i].lhs;
                                strcpy(clos[noitem][n].rhs,clos[z][i].rhs);
                                char temp=clos[noitem][n].rhs[j];
                                clos[noitem][n].rhs[j]=clos[noitem][n].rhs[j+1];
                                clos[noitem][n].rhs[j+1]=temp;
                                n=n+1;
                        }
                }
        }
        for(i=0;i<n;i++)
        {
```

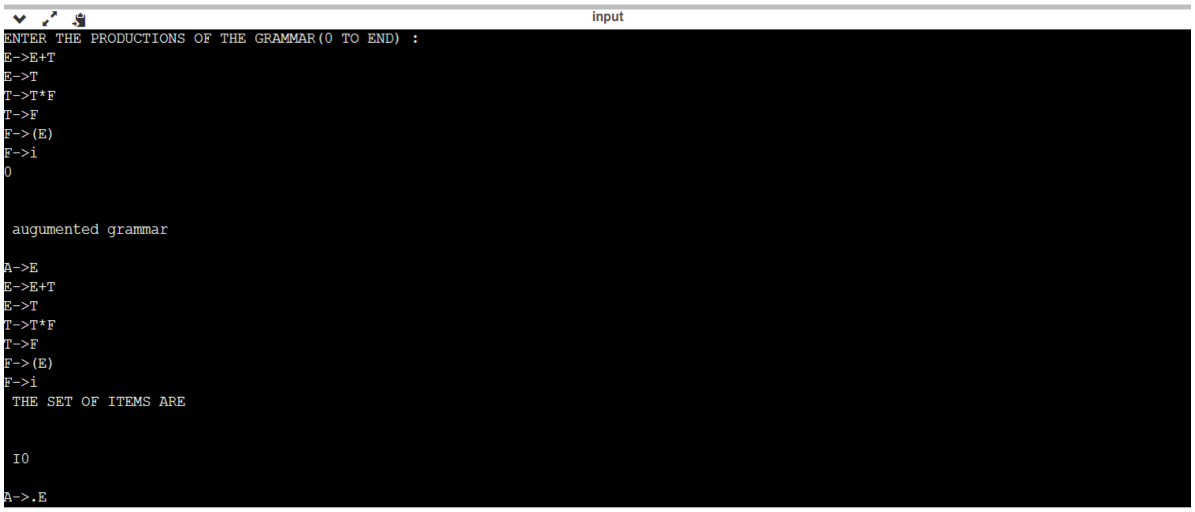
```
		for(j=0;j<strlen(clos[noitem][i].rhs);j++)
		{
			if(clos[noitem][i].rhs[j]=='.'                                   &&
isvariable(clos[noitem][i].rhs[j+1])>0)
			{
				for(k=0;k<novar;k++)
				{
					if(clos[noitem][i].rhs[j+1]==clos[0][k].lhs)
					{
						for(l=0;l<n;l++)
							if(clos[noitem][l].lhs==clos[0][k].lhs
&& strcmp(clos[noitem][l].rhs,clos[0][k].rhs)==0)
								break;
						if(l==n)
						{
							clos[noitem][n].lhs=clos[0][k].lhs;
						strcpy(clos[noitem][n].rhs,clos[0][k].rhs);
							n=n+1;
						}
					}
				}
			}
		}
	}
	arr[noitem]=n;
	int flag=0;
	for(i=0;i<noitem;i++)
	{
		if(arr[i]==n)
		{
			for(j=0;j<arr[i];j++)
			{
				int c=0;
				for(k=0;k<arr[i];k++)
					if(clos[noitem][k].lhs==clos[i][k].lhs                &&
strcmp(clos[noitem][k].rhs,clos[i][k].rhs)==0)
						c=c+1;
```

```
                        if(c==arr[i])
                        {
                                flag=1;
                                goto exit;
                        }
                }
        }
    }
    exit:;
    if(flag==0)
        arr[noitem++]=n;
}

int main()
{
    cout<<"ENTER THE PRODUCTIONS OF THE GRAMMAR(0 TO END)
:\n";
    do
    {
        cin>>prod[i++];
    }while(strcmp(prod[i-1],"0")!=0);
    for(n=0;n<i-1;n++)
    {
        m=0;
        j=novar;
        g[novar++].lhs=prod[n][0];
        for(k=3;k<strlen(prod[n]);k++)
        {
            if(prod[n][k] != '|')
            g[j].rhs[m++]=prod[n][k];
            if(prod[n][k]=='|')
            {
                g[j].rhs[m]='\0';
                m=0;
                j=novar;
                g[novar++].lhs=prod[n][0];
            }
```

```
		}
	}
	for(i=0;i<26;i++)
		if(!isvariable(listofvar[i]))
			break;
	g[0].lhs=listofvar[i];
	char temp[2]={g[1].lhs,'\0'};
	strcat(g[0].rhs,temp);
	cout<<"\n\n augumented grammar \n";
	for(i=0;i<novar;i++)
		cout<<endl<<g[i].lhs<<"->"<<g[i].rhs<<" ";

	for(i=0;i<novar;i++)
	{
		clos[noitem][i].lhs=g[i].lhs;
		strcpy(clos[noitem][i].rhs,g[i].rhs);
		if(strcmp(clos[noitem][i].rhs,"ε")==0)
			strcpy(clos[noitem][i].rhs,".");
		else
		{
			for(int j=strlen(clos[noitem][i].rhs)+1;j>=0;j--)
				clos[noitem][i].rhs[j]=clos[noitem][i].rhs[j-1];
			clos[noitem][i].rhs[0]='.';
		}
	}
	arr[noitem++]=novar;
	for(int z=0;z<noitem;z++)
	{
		char list[10];
		int l=0;
		for(j=0;j<arr[z];j++)
		{
			for(k=0;k<strlen(clos[z][j].rhs)-1;k++)
			{
				if(clos[z][j].rhs[k]=='.')
				{
					for(m=0;m<l;m++)
```

```
                                        if(list[m]==clos[z][j].rhs[k+1])
                                             break;
                                   if(m==l)
                                        list[l++]=clos[z][j].rhs[k+1];
                              }
                         }
                    }
                    for(int x=0;x<l;x++)
                         findclosure(z,list[x]);
               }
               cout<<"\n THE SET OF ITEMS ARE \n\n";
               for(int z=0; z<noitem; z++)
               {
                    cout<<"\n I"<<z<<"\n\n";
                    for(j=0;j<arr[z];j++)
                         cout<<clos[z][j].lhs<<"->"<<clos[z][j].rhs<<"\n";

               }

          }
```

OUTPUT:



```
ENTER THE PRODUCTIONS OF THE GRAMMAR(0 TO END) :
E->E+T
E->T
T->T*F
T->F
F->(E)
F->i
0


 augumented grammar

A->E
E->E+T
E->T
T->T*F
T->F
F->(E)
F->i
 THE SET OF ITEMS ARE


 I0

A->.E
```

RESULT: The program for computation of LR[0] was successfully compiled and run

GANGISETTY RAJ CHARAN
RA2011028010007

EXPERIMENT 10

AIM: To Write code For Intermediate code generation :-> Postfix, Prefix

ALGORITHM:

Infix to Postfix Conversion Algorithm:

Make an empty stack and a blank output string.
Do the following for each token in the infix expression:

a) Add the token to the output string if it is an operand.
b) Place the token on the stack if it is a left parenthesis.
c) If the token is a right parenthesis, remove operators from the stack and replace them in the output string until a left parenthesis is found. Remove the left parenthesis.
d) If the token is an operator, remove operators from the stack and add them to the output string as long as they have equal or greater precedence than the current operator. The current operator is then pushed into the stack.
e) After processing all tokens, remove any lingering operators from the stack and add them to the result string.
f) The postfix expression is the resultant output string.

Infix to Prefix Conversion Algorithm:

1) The infix expression should be reversed.
2) Each left parenthesis should be replaced with a right parenthesis, and each right parenthesis with a left parenthesis.
3) To convert the inverted phrase to postfix notation, use the infix to postfix conversion technique.
4) To retrieve the prefix expression, reverse the resultant postfix expression.
5) Please keep in mind that the preceding methods presume that the infix expression is well-formed and contains only legal operators and operands.

GANGISETTY RAJ CHARAN
RA2011028010007

CODE:

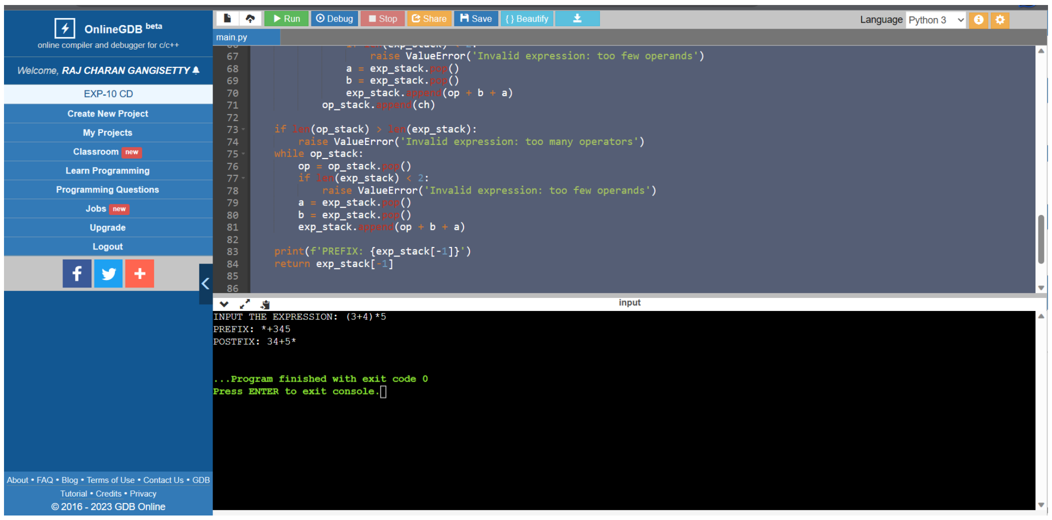
```
OPERATORS = set(['+', '-', '*', '/', '(', ')'])
PRI = {'+': 1, '-': 1, '*': 2, '/': 2}
def infix_to_postfix(formula):
    stack = []  # only pop when the coming op has priority
    output = ''
    for ch in formula:

        if ch not in OPERATORS:

            output += ch

        elif ch == '(':

            stack.append('(')

        elif ch == ')':

            while stack and stack[-1] != '(':
                output += stack.pop()

            stack.pop()  # pop '('

        else:

            while stack and stack[-1] != '(' and PRI[ch] <= PRI[stack[-1]]:
                output += stack.pop()

            stack.append(ch)

            # leftover

    while stack:
        output += stack.pop()

    print(f'POSTFIX: {output}')

    return output
```

GANGISETTY RAJ CHARAN
RA2011028010007

```
### INFIX ===> PREFIX ###

def infix_to_prefix(formula):
    op_stack = []

    exp_stack = []

    for ch in formula:

        if not ch in OPERATORS:

            exp_stack.append(ch)

        elif ch == '(':

            op_stack.append(ch)

        elif ch == ')':

            while op_stack[-1] != '(':
                op = op_stack.pop()

                a = exp_stack.pop()

                b = exp_stack.pop()

                exp_stack.append(op + b + a)

            op_stack.pop()  # pop '('

        else:

                 while op_stack and op_stack[-1] != '(' and PRI[ch] <=
PRI[op_stack[-1]]:
                op = op_stack.pop()

                a = exp_stack.pop()
                b = exp_stack.pop()

                exp_stack.append(op + b + a)

            op_stack.append(ch)

            # leftover

    while op_stack:
        op = op_stack.pop()

        a = exp_stack.pop()

        b = exp_stack.pop()

        exp_stack.append(op + b + a)

    print(f'PREFIX: {exp_stack[-1]}')

    return exp_stack[-1]

expres = input("INPUT THE EXPRESSION: ")

pre = infix_to_prefix(expres)

pos = infix_to_postfix(expres)
```

GANGISETTY RAJ CHARAN
RA2011028010007

OUTPUT:



```python
                    raise ValueError('Invalid expression: too few operands')
                a = exp_stack.pop()
                b = exp_stack.pop()
                exp_stack.append(op + b + a)
            op_stack.append(ch)

    if len(op_stack) > len(exp_stack):
        raise ValueError('Invalid expression: too many operators')
    while op_stack:
        op = op_stack.pop()
        if len(exp_stack) < 2:
            raise ValueError('Invalid expression: too few operands')
        a = exp_stack.pop()
        b = exp_stack.pop()
        exp_stack.append(op + b + a)

    print(f'PREFIX: {exp_stack[-1]}')
    return exp_stack[-1]
```

```
INPUT THE EXPRESSION: (3+4)*5
PREFIX: *+345
POSTFIX: 34+5*


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

RESULT: code For Intermediate code generation Executed Successfully

EXPERIMENT-11

AIM: To Code the intermediate code generation-quadruple,triple,indirect triple

ALGORITHM:

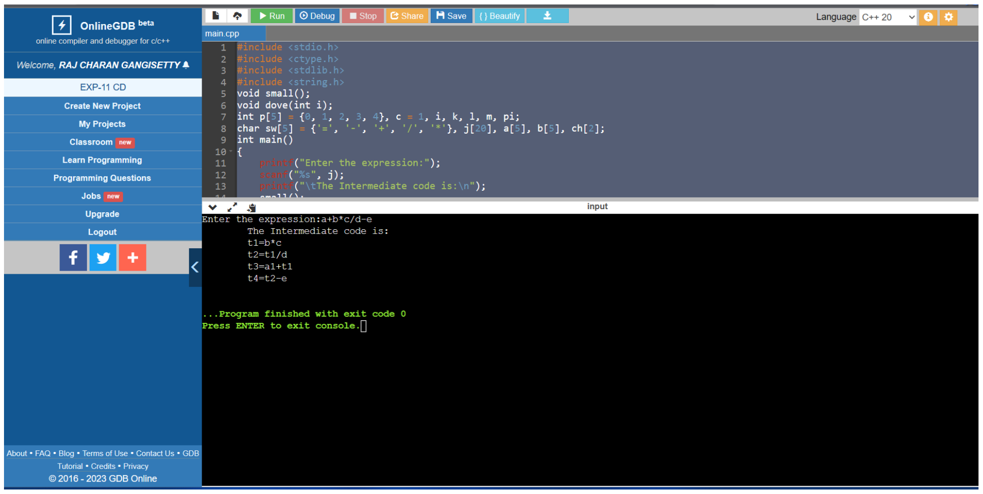
A simple code generator algorithm could look something like this:

1. Parse the input program to build a parse tree or AST.
2. Traverse the AST in a depth-first manner and generate code for each node in the tree.
3. For each node, generate the corresponding intermediate code based on the type of node and its children. For example, for an assignment statement, generate code to evaluate the right-hand side expression and store the result in the left-hand side variable.
4. Assign temporary variables for subexpressions and store intermediate results in these variables.
5. Maintain a symbol table to keep track of the variables and their types. This symbol table should also include information about functions and their parameters.
6. Generate code for control structures such as if-else statements and loops. This involves generating labels and conditional and unconditional jumps to implement the control flow.
7. Implement optimizations like common subexpression elimination and constant folding to reduce the size of the intermediate code.
8. Generate code for function calls and handling of function arguments and return values. This involves generating code to push arguments onto the stack, calling the function, and retrieving the return value.
9. Generate code for memory allocation and deallocation, if required. This involves generating calls to the new and delete operators or their equivalents in the target machine's runtime library.
10. Finally, output the generated intermediate code to a file or data structure for further processing. This intermediate code can then be converted to machine code by a separate code generation phase.

CODE:

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
void small();
void dove(int i);
int p[5] = {0, 1, 2, 3, 4}, c = 1, i, k, l, m, pi;
char sw[5] = {'=', '-', '+', '/', '*'}, j[20], a[5], b[5], ch[2];
void main()
{
    printf("Enter the expression:");
    scanf("%s", j);
    printf("\tThe Intermediate code is:\n");
    small();
}
void dove(int i)
{
    a[0] = b[0] = '\0';
    if (!isdigit(j[i + 2]) && !isdigit(j[i - 2]))
    {
        a[0] = j[i - 1];
        b[0] = j[i + 1];
    }
    if (isdigit(j[i + 2]))
    {
        a[0] = j[i - 1];
        b[0] = 't';
        b[1] = j[i + 2];
    }
    if (isdigit(j[i - 2]))
    {
        b[0] = j[i + 1];
        a[0] = 't';
        a[1] = j[i - 2];
        b[1] = '\0';
    }
    if (isdigit(j[i + 2]) && isdigit(j[i - 2]))
    {
        a[0] = 't';
        b[0] = 't';
        a[1] = j[i - 2];
        b[1] = j[i + 2];
        sprintf(ch, "%d", c);
        j[i + 2] = j[i - 2] = ch[0];
    }
    if (j[i] == '*')
        printf("\tt%d=%s*%s\n", c, a, b);
    if (j[i] == '/')
        printf("\tt%d=%s/%s\n", c, a, b);
    if (j[i] == '+')
        printf("\tt%d=%s+%s\n", c, a, b);
    if (j[i] == '-')
        printf("\tt%d=%s-%s\n", c, a, b);
    if (j[i] == '=')
        printf("\t%c=t%d", j[i - 1], --c);
    sprintf(ch, "%d", c);
    j[i] = ch[0];
    c++;
    small();
}
void small(){
    pi = 0;
    l = 0;
    for (i = 0; i < strlen(j); i++)
    {
        for (m = 0; m < 5; m++)
            if (j[i] == sw[m])
                if (pi <= p[m]){
                    pi = p[m];
                    l = 1;
                    k = i;
                }
    }
    if (l == 1) dove(k);
    else exit(0);
}
```

OUTPUT:



```c
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
void small();
void dove(int i);
int p[5] = {0, 1, 2, 3, 4}, c = 1, i, k, l, m, pi;
char sw[5] = {'=', '-', '+', '/', '*'}, j[20], a[5], b[5], ch[2];
int main()
{
    printf("Enter the expression:");
    scanf("%s", j);
    printf("\tThe Intermediate code is:\n");
```

```
Enter the expression:a+b*c/d-e
        The Intermediate code is:
        t1=b*c
        t2=t1/d
        t3=a1+t1
        t4=t2-e


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

RESULT:simple code Generator is coded successfully

EXPERIMENT-12

AIM:TO CODE SIMPLE CODE GENERATOR

ALGORITHM:

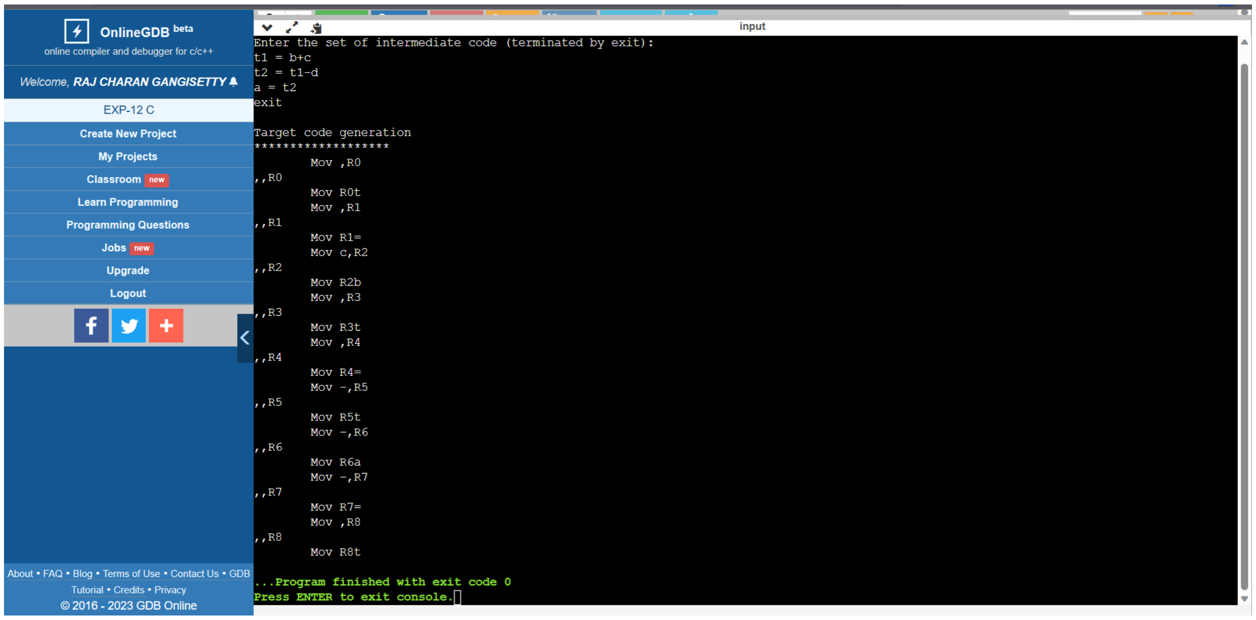
1. Traverse the AST in a depth-first manner, generating code for each node.
2. For each expression, assign a register to store the result. If there are more expressions than available registers, use the memory stack to store intermediate results.
3. For each variable or constant, load the value into a register or use an immediate operand.
4. For arithmetic operations, generate code that performs the operation on the registers holding the operands, and store the result in a register.
5. For assignment statements, generate code that loads the value into a register and stores it in the variable's memory location.
6. For control structures, generate code that handles branching and looping. For example, generate code to jump to a specific memory location if a condition is true or false.
7. For function calls, generate code to push the arguments onto the stack, call the function, and retrieve the return value. If there are not enough registers to hold the arguments, use the stack to store them.
8. When a register is no longer needed, free it up for reuse. This can be done by storing its value in memory or by using a register allocator that keeps track of which registers are currently in use and which are free.
9. Output the generated machine code.

CODE:

```
#include <stdio.h>
#include <string.h>
void main()
{
    char icode[10][30], str[20], opr[10];
    int i = 0;
    printf("\nEnter the set of intermediate code (terminated by
exit):\n");
    do
    {
```

```c
        scanf("%s", icode[i]);
    } while (strcmp(icode[i++], "exit") != 0);
    printf("\nTarget code generation");
    printf("\n******************");
    i = 0;
    do
    {
        strcpy(str, icode[i]);
        switch (str[3])
        {
        case '+':
            strcpy(opr, "ADD");
            break;
        case '-':
            strcpy(opr, "SUB");
            break;
        case '*':
            strcpy(opr, "MUL");
            break;
        case '/':
            strcpy(opr, "DIV");
            break;
        }

        printf("\n\tMov %c,R%d", str[2], i);
        printf("\n\%s%c,,R%d", opr, str[4], i);
        printf("\n\tMov R%d%c", i, str[0]);
    } while (strcmp(icode[++i], "exit") != 0);
}
```

OUTPUT:



```
Enter the set of intermediate code (terminated by exit):
t1 = b+c
t2 = t1-d
a = t2
exit

Target code generation
*******************
        Mov ,R0
,,R0
        Mov R0t
        Mov ,R1
,,R1
        Mov R1=
        Mov c,R2
,,R2
        Mov R2b
        Mov ,R3
,,R3
        Mov R3t
        Mov ,R4
,,R4
        Mov R4=
        Mov -,R5
,,R5
        Mov R5t
        Mov -,R6
,,R6
        Mov R6a
        Mov -,R7
,,R7
        Mov R7=
        Mov ,R8
,,R8
        Mov R8t

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

RESULT:SIMPLE CODE GENERATOR IS CODED SUCCESFULLY

EXPERIMENT-13

AIM:Implementation of DAG

ALGORITHM:

1. Parse the input program and generate an AST.
2. Traverse the AST in a depth-first manner, generating code for each node.
3. Create a directed acyclic graph (DAG) data structure to represent the subexpressions of the input program.
4. For each subexpression encountered, check if it has already been computed and stored in the DAG. If it has, replace the subexpression with a reference to the DAG node that stores its result. If it has not, create a new node in the DAG to represent the subexpression and its result.
5. Assign a unique identifier to each node in the DAG.
6. For each node in the DAG, store the operator (if any), the operands (if any), and the result value.
7. Generate code to evaluate the DAG by performing a depth-first traversal of the DAG and generating code for each node in a bottom-up order.
8. During the traversal, check if a DAG node has already been computed by checking if its result value has already been stored in a register or memory location. If it has, use the existing value. Otherwise, generate code to compute the value of the node.
9. For control structures, generate code that handles branching and looping using the DAG as a guide. For example, generate code to jump to a specific node in the DAG if a condition is true or false.

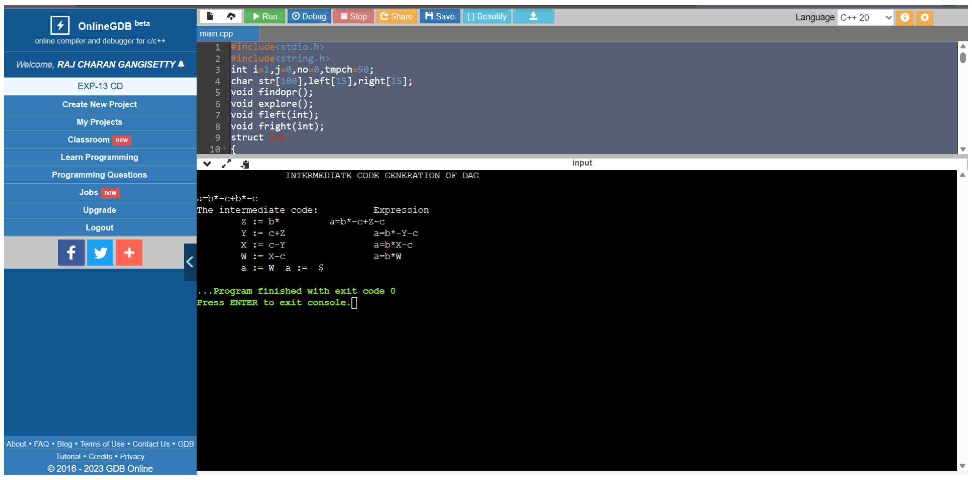
CODE:

```
#include <stdio.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
{
```

```c
    int pos;
    char op;
} k[15];
void main(){

    printf("\t\tINTERMEDIATE CODE GENERATION OF DAG\n\n");
    scanf("%s", str);
    printf("The intermediate code:\t\tExpression\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);
        for (j = 0; j < strlen(str); j++)
            if (str[j] != '$')
                printf("%c", str[j]);
        printf("\n");
        i++;
    }
    fright(-1);
    if (no == 0){
        fleft(strlen(str));
        printf("\t%s := %s", right, left);
    }
    printf("\t%s :=  %c", right, str[k[--i].pos]);
}
void fleft(int x){
    int w = 0, flag = 0;
    x--;
    while (x != -1 && str[x] != '+' && str[x] != '*' && str[x] != '=' &&
str[x] != '\0' && str[x] != '-' && str[x] != '/' && str[x] != ':')
    {
        if (str[x] != '$' && 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:



```
#include<stdio.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
{
```

```
                INTERMEDIATE CODE GENERATION OF DAG

a=b*-c+b*-c
The intermediate code:          Expression
        Z := b*         a=b*-c+Z-c
        Y := c+Z                a=b*-Y-c
        X := c-Y                a=b*X-c
        W := X-c                a=b*W
        a := W  a :=  $

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

RESULT : DAG IS SUCCESSFULLY IMPLEMENTED AND EXECUTED