**INDEX**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Exp. No.** | **Name of Experiment** | **Page No.** | **Date** | **Signature** |
| 1. | Implementation of Lexical Analyzer | 1-2 |  |  |
| 2. | Regular Expression to NFA | 3-6 |  |  |
| 3. | NFA to DFA | 7-12 |  |  |
| 4. | Elimination of Ambiguity Left Recursion and Left Factoring | 13-17 |  |  |
| 5. | First and Follow Computation | 18-23 |  |  |
| 6. | Computation of Predictive Parsing table | 24-30 |  |  |
| 7. | Computation of Shift Reduce Parsing | 31-34 |  |  |
| 8. | Program for finding the leading and trailing. | 35-40 |  |  |
| 9. | Computation of LR(0) items | 41-43 |  |  |
| 10. | Intermediate Code Generation Postfix and prefix expression | 44-51 |  |  |
| 11. | Intermediate Code Generation Quadruple, Triple, Indirect triple | 52-54 |  |  |
| 12. | A simple code Generator | 55-58 |  |  |
| 13. | Implementation of DAG | 59-62 |  |  |
| 14. | Implementation of Global Data flow Analysis | 63-65 |  |  |
| 15. | Implement any one storage  allocation strategies (heap, stack, static ) | 66-73 |  |  |

# EXPERIMENT 1

**Implementation of Lexical Analyzer**

**Aim:** Write a program in C/C++ to implement a lexical analyzer.

### Algorithm:

1. Start
2. Get the input expression from the user.
3. Store the keywords and operators.
4. Perform analysis of the tokens based on the ASCII values. 5.

ASCII Range TOKEN TYPE

97-122 Keyword else identifier 48-57 Constant else operator Greater than 12 Symbol

1. Print the token types.
2. Stop

**Program:-** #include<stdio.h> #include<stdlib.h> #include<string.h> #include<ctype.h>

int isKeyword(char buffer[]){

char keywords[32][10] = {"auto","break","case","char","const","continue","default",

"do","double","else","enum","extern","float","for","goto",

"if","int","long","register","return","short","signed",

"sizeof","static","struct","switch","typedef","union",

"unsigned","void","volatile","while"}; int i, flag = 0;

for(i = 0; i < 32; ++i){ if(strcmp(keywords[i], buffer) == 0){ flag = 1;

break;

}

}

return flag;

}

int main(){

char ch, buffer[15], operators[] = "+-\*/%=";

FILE \*fp; int i,j=0;

fp = fopen("program.txt","r"); if(fp == NULL){

printf("error while opening the file\n"); exit(0);

}

while((ch = fgetc(fp)) != EOF){ for(i = 0; i < 6; ++i){

if(ch == operators[i]) printf("%c is operator\n", ch);

}

if(isalnum(ch)){ buffer[j++] = ch;

}

else if((ch == ' ' || ch == '\n') && (j != 0)){ buffer[j] = '\0';

j = 0;

if(isKeyword(buffer) == 1) printf("%s is keyword\n", buffer); else

printf("%s is indentifier\n", buffer);

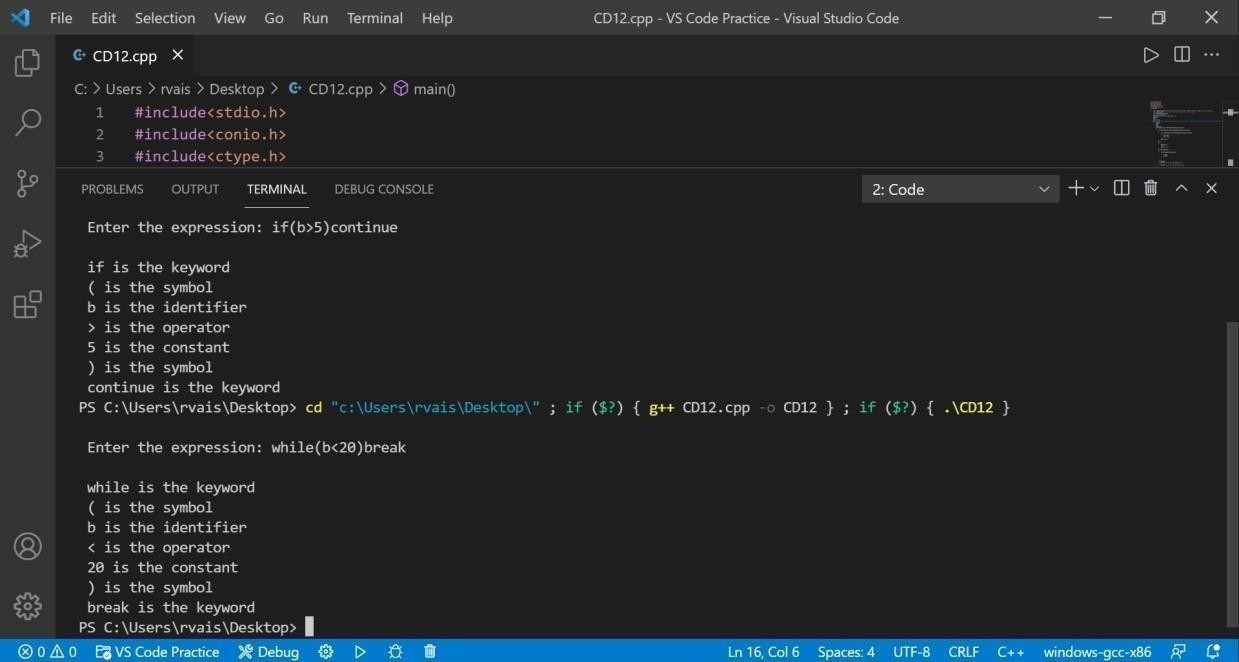
}

}

fclose(fp); return 0;

}

### OUTPUT:

****

**Result:** The Program Executed successfully.

# EXPERIMENT 2

**Regular Expression to NFA**

**AIM:-** Program to convert Regular Expression(R.E.) to Non-Deterministic FiniteAutomata(N.F.A.)

### Algorithm:-

1. The NFA representing the empty string is:

\begin{picture}(77.00,35.00)  \put(10.00,10.00){\vector(1,0){3.00}}  \put(20.00,10...  ...ctor(3,-1){2}}  \put(45.00,22.00){\makebox(0,0)[cc]{$\varepsilon$}}  \end{picture}

1. If the regular expression is just a character, eg. a, then the correspondingNFA is :

\begin{picture}(77.00,35.00)  \put(10.00,10.00){\vector(1,0){3.00}}  \put(20.00,10...  ...0,15.50){\vector(3,-1){2}}  \put(45.00,22.00){\makebox(0,0)[cc]{a}}  \end{picture}

1. The union operator is represented by a choice of transitions from anode; thus a|b can be represented as:

\begin{picture}(77.00,35.00)  \put(10.00,20.00){\vector(1,0){3.00}}  \put(20.00,20...  ....00,14.50){\vector(3,1){2}}  \put(45.00,8.00){\makebox(0,0)[cc]{b}}  \end{picture}

1. Concatenation simply involves connecting one NFA to the other; eg. abis:

\begin{picture}(77.00,35.00)  \put(10.00,10.00){\vector(1,0){3.00}}  \put(20.00,10...  ...0,16.00){\vector(2,-1){2}}  \put(65.00,22.00){\makebox(0,0)[cc]{b}}  \end{picture}

1. The Kleene closure must allow for taking zero or more instances of theletter from the input; thus a\* looks like:

\begin{picture}(77.00,55.00)  \put(10.00,30.00){\vector(1,0){3.00}}  \put(20.00,30...  ...vector(2,1){2}}  \put(65.00,8.00){\makebox(0,0)[cc]{$\varepsilon$}}  \end{picture}

### Program Code:-

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

int ret[100]; static int pos = 0; static int sc = 0;

void nfa(int st, int p, char\*s)

{

int i,sp,fs[15],fsc=0; sp=st:pos=p;sc=st; while(\*s!=NULL)

{

if(isalpha(\*s))

{ret[pos++]=sp; ret[pos++]=\*s; ret[pos++]=++sc;} if(\*s=='.')

{

sp=sc; ret[pos++]=sc; ret[pos++]=238; ret[pos++]=++sc; sp=sc;}

if(\*s=='|')

{sp=st; fs[fsc++]=sc;} if(\*s=='\*')

{ret[pos++]=sc; ret[pos++]=238; ret[pos++]=sp; ret[pos++]=sp; ret[pos++]=238;

ret[pos++]=sc;

}

if(\*s=='(')

{

char ps[50]; int i=0,flag=1; s++;

while(flag!=0)

{

ps[i++]=\*s; if(\*s=='(') flag++; if(\*s==')') flag--; s++;}

ps[--i]='\0';

nfa(sc,pos,ps); s--;

} s++;

}

sc++; for(i=0;i<fsc;i++)

{

ret[pos++]=fs[i]; ret[pos++]=238; ret[pos++]=sc;

}

ret[pos++]=sc-1;

ret[pos++]=238; ret[pos++]=sc;

}

void main()

{

int i;

char \*inp; clrscr();

printf("enter the regular expression :"); gets(inp);

nfa(1,0,inp);

printf("\nstate intput state\n"); for(i=0;i<pos;i=i+3)

print("%d --%c--> %d\n", ret[i], ret[i+1,ret[i+2]); printf("\n");

getch();

}

### Output:-

****

**Result:-** The program to convert R.E to N.F.A. was successfully executed and the output was verified.

# EXPERIMENT 3

**NFA to DFA**

**AIM:-** Program to convert NFA to Deterministic Finite Automata(D.F.A.)

### Algorithm:-

1. Convert into NFA using above rules for operators (union, concatenationand closure) and precedence.
2. Find Ɛ -closure of all states.
3. Start with epsilon closure of start state of NFA.
4. Apply the input symbols and find its epsilon closure. Dtran[state,

input symbol] = Ɛ -closure(move(state, input symbol)) where Dtran àtransitionfunction of DFA

1. Analyze the output state to find whether it is a new state.
2. If new state is found, repeat step 4 and step 5 until no more new statesare found.
3. Construct the transition table for Dtran function.
4. Draw the transition diagram with start state as the Ɛ -closure (start state of NFA) and final state is the state that contains final state of NFA drawn.

**Program Code:-** #include<iostream.h> #include<string.h> #include<stdio.h> #include<conio.h> #include<stdlib.h>

char nfa[50][50],s[20],st[10][20],eclos[20],input[20]; int x,e,top=0,topd=0,n=0,ns,nos,in;

int checke(char a)

{

int i; for(i=0;i<e;i++)

{

if(eclos[i]==a) return i;

}

return -1;

}

int check(char a)

{

int i; for(i=0;i<in;i++)

{

if(input[i]==a) return i;

}

return -1;

}

void push(char a)

{

s[top]=a; top++;

}

char pop()

{

top--;

return s[top];

}

void pushd(char \*a)

{

strcpy(st[topd],a); topd++;

}

char \*popd()

{

topd--;

return st[topd];

}

int ctoi(char a)

{

int i=a-48; return i;

}

char itoc(int a)

{

char i=a+48; return i;

}

char \*eclosure(char \*a)

{

int i,j; char c;

for(i=0;i<strlen(a);i++) push(a[i]); e=strlen(a); strcpy(eclos,a); while(top!=0)

{

c=pop(); for(j=0;j<ns;j++)

{

if(nfa[ctoi(c)][j]=='e')

{

if(check(itoc(j))==-1)

{

eclos[e]=itoc(j); push(eclos[e]); e++;

}

}

}

}

eclos[e]='\0'; return eclos;

}

void main()

{

int i,j,k,count;

char ec[20],a[20],b[20],c[20],dstates[10][10]; clrscr();

cout<<"Enter the number of states"<<endl; cin>>ns;

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

{

for(j=0;j<ns;j++)

{

cout<<"Move["<<i<<"]["<<j<<"]";

cin>>nfa[i][j];

if(nfa[i][j]!='-'&&nfa[i][j]!='e')

{

if((check(nfa[i][j]))==-1) input[in++]=nfa[i][j];

}

}

}

topd=0; nos=0; c[0]=itoc(0);

c[1]='\0';

pushd(eclosure(c)); strcpy(dstates[nos],eclosure(c)); for(x=0;x<in;x++) cout<<"\t"<<input[x]; cout<<"\n";

while(topd>0)

{

strcpy(a,popd()); cout<<a<<"\t"; for(i=0;i<in;i++)

{

int len=0; for(j=0;j<strlen(a);j++)

{

int x=ctoi(a[j]); for(k=0;k<ns;k++)

{

if(nfa[x][k]==input[i]) ec[len++]=itoc(k);

}

}

ec[len]='\0'; strcpy(b,eclosure(ec)); count=0; for(j=0;j<=nos;j++)

{

if(strcmp(dstates[j],b)==0)

count++;

}

if(count==0)

{ if(b[0]!='\0')

{

nos++; pushd(b);

strcpy(dstates[nos],b);

}

}

cout<<b<<"\t";

}

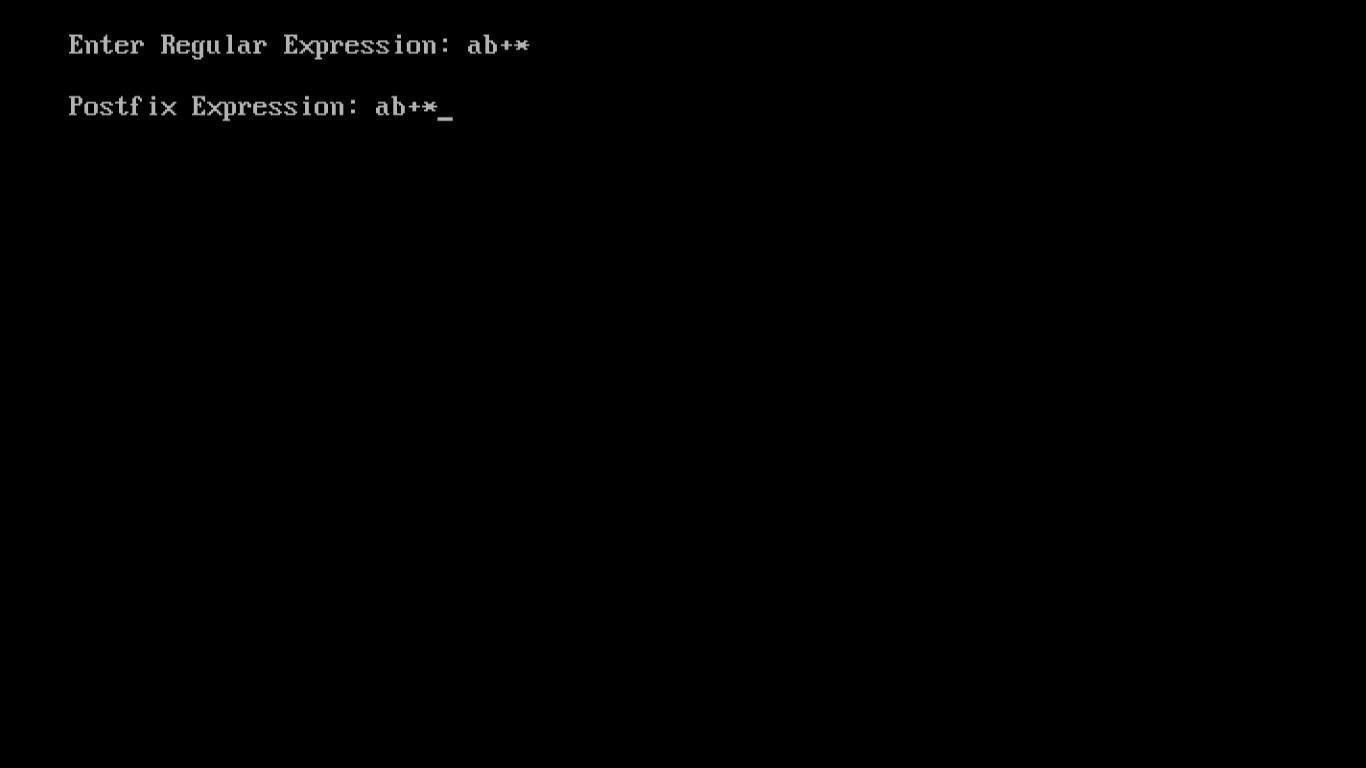
cout<<endl;

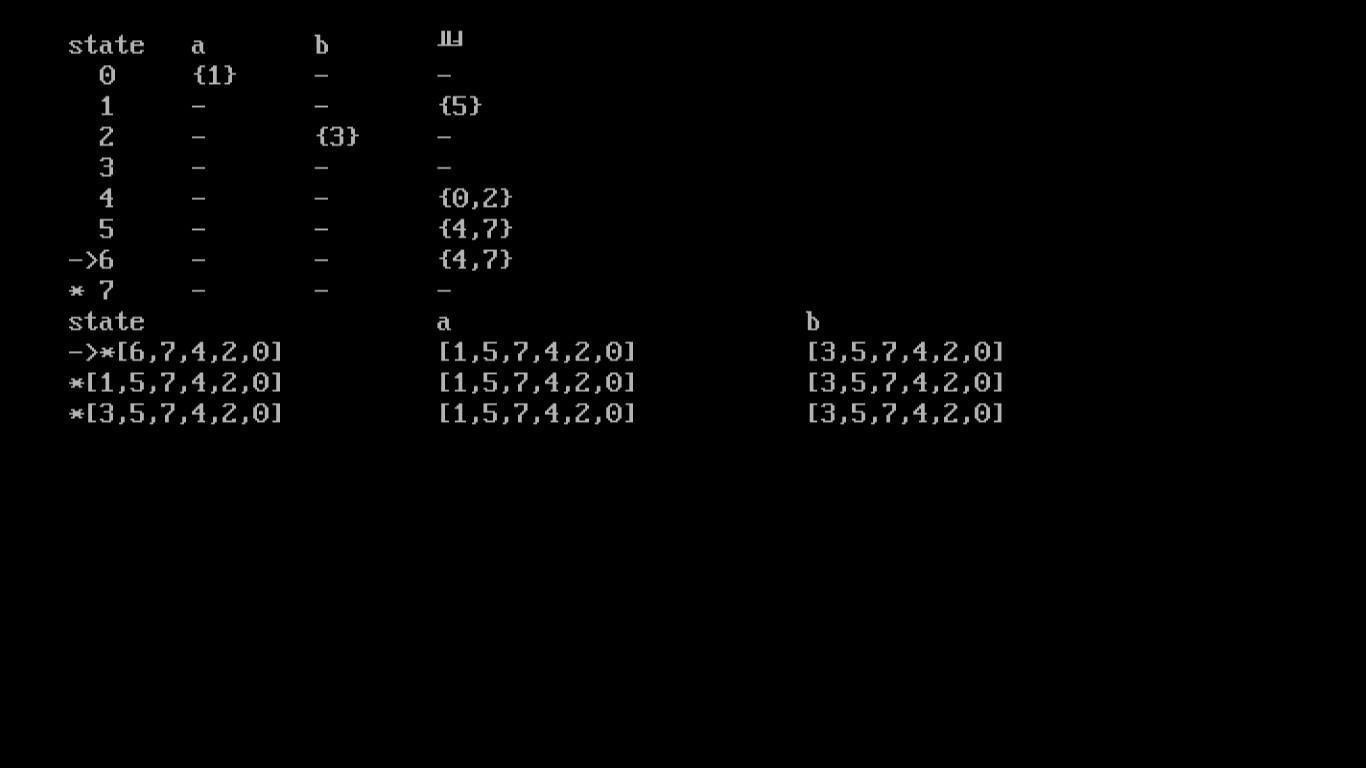
}

getch();

}

### Output:-

****



**Result:-** The program to convert NFA to Deterministic Finite Automata(D.F.A.)was successfully executed and the output was verified.

# EXPERIMENT 4

**Left Recursion and Left Factoring**

## **Aim-**Elimination of Ambiguity Left Recursion and Left Factoring

### Algorithm:

1. **Left Recursion-**

* A production of grammar is said to have **left recursion** if the leftmost variable of its RHS is same as variable of its LHS.
* A grammar containing a production having left recursion is called as Left Recursive Grammar.

1. **Left factoring**
   * For each non terminal A find the longest prefix α common to two or more of its alternatives.
   * If α!= E,i. e., there is a non trivial common prefix, replace all the A productions

# Program –

1. **Left Recursion-** #include<iostream> #include<string> using namespace std; int main()

{ string ip,op1,op2,temp; int sizes[10] = {};

char c; int n,j,l;

cout<<"Enter the Parent Non-Terminal : "; cin>>c;

ip.push\_back(c); op1 += ip + "\'->";

ip += "->";

op2+=ip;

cout<<"Enter the number of productions : "; cin>>n;

for(int i=0;i<n;i++)

{ cout<<"Enter Production "<<i+1<<" : "; cin>>temp;

sizes[i] = temp.size(); ip+=temp;

if(i!=n-1)

ip += "|";

}

cout<<"Production Rule : "<<ip<<endl; for(int i=0,k=3;i<n;i++)

{

if(ip[0] == ip[k])

{

cout<<"Production "<<i+1<<" has left recursion."<<endl; if(ip[k] != '#')

{

for(l=k+1;l<k+sizes[i];l++) op1.push\_back(ip[l]);

k=l+1; op1.push\_back(ip[0]); op1 += "\'|";

}

}

else

{

cout<<"Production "<<i+1<<" does not have left recursion."<<endl; if(ip[k] != '#')

{

for(j=k;j<k+sizes[i];j++) op2.push\_back(ip[j]);

k=j+1; op2.push\_back(ip[0]); op2 += "\'|";

}

else

{

op2.push\_back(ip[0]); op2 += "\'";

}}}

op1 += "#";

cout<<op2<<endl; cout<<op1<<endl; return 0;}

1. **Left factoring** #include<iostream> #include<string> using namespace std; int main()

{ string ip,op1,op2,temp; int sizes[10] = {};

char c; int n,j,l;

cout<<"Enter the Parent Non-Terminal : "; cin>>c;

ip.push\_back(c); op1 += ip + "\'->";

op2 += ip + "\'\'->";; ip += "->";

cout<<"Enter the number of productions : "; cin>>n;

for(int i=0;i<n;i++)

{

cout<<"Enter Production "<<i+1<<" : "; cin>>temp;

sizes[i] = temp.size(); ip+=temp;

if(i!=n-1)

ip += "|";

}

cout<<"Production Rule : "<<ip<<endl; char x = ip[3];

for(int i=0,k=3;i<n;i++)

{

if(x == ip[k])

{

if(ip[k+1] == '|')

{

op1 += "#";

ip.insert(k+1,1,ip[0]);

ip.insert(k+2,1,'\''); k+=4;

}

else

{

op1 += "|" + ip.substr(k+1,sizes[i]-1); ip.erase(k-1,sizes[i]+1);

}

}

else

{

while(ip[k++]!='|');

}

}

char y = op1[6];

for(int i=0,k=6;i<n-1;i++)

{

if(y == op1[k])

{

if(op1[k+1] == '|')

{

op2 += "#";

op1.insert(k+1,1,op1[0]);

op1.insert(k+2,2,'\''); k+=5;

}

else

{

temp.clear();

for(int s=k+1;s<op1.length();s++) temp.push\_back(op1[s]);

op2 += "|" + temp;

op1.erase(k-1,temp.length()+2);

} }}

op2.erase(op2.size()-1);

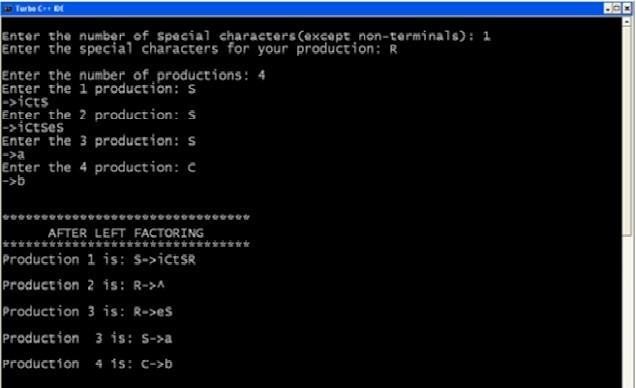
cout<<"After Left Factoring : "<<endl; cout<<ip<<endl;

cout<<op1<<endl; cout<<op2<<endl; return 0;

}

### C:\Users\91898\Downloads\Untitled (1).pngOutput:1 Left Recursion

1. **Left factoring**



**Result:** Left recursion and left factoring is done successfully.

# EXPERIMENT 5

**First and Follow Computation**

**AIM:-** Program to compute FIRST and FOLLOW sets

### Algorithm:-

FIRST(X) for all grammar symbols X:

* 1. If X is terminal, FIRST(X) = {X}.
  2. If X → ε is a production, then add ε to FIRST(X).
  3. If X is a non-terminal, and X → Y1 Y2 … Yk is a production, and ε is in allof FIRST(Y1 ), …, FIRST(Yk ), then add ε to FIRST(X).
  4. If X is a non-terminal, and X → Y1 Y2 … Yk is a production, then add a toFIRST(X) if for some i, a is in FIRST(Yi), and ε is in all of FIRST(Y1 ), …, FIRST(Yi-1 ).

FOLLOW(A) for all non-terminals A:

1. If $ is the input end-marker, and S is the start symbol, $ ∈ FOLLOW(S).
2. If there is a production, A → αBβ, then (FIRST(β) – ε) ⊆ FOLLOW(B).
3. If there is a production, A → αB, or a production A → αBβ, where ε ∈

FIRST(β), then FOLLOW(A) ⊆ FOLLOW(B).

### Program:- First-

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

void FIRST(char[],char );

void addToResultSet(char[],char); int numOfProductions;

char productionSet[10][10]; main()

{

int i;

char choice; char c;

char result[20];

printf("How many number of productions ? :"); scanf(" %d",&numOfProductions);

for(i=0;i<numOfProductions;i++)//read production string eg: E=E+T

{

printf("Enter productions Number %d : ",i+1); scanf(" %s",productionSet[i]);

}

do

{

printf("\n Find the FIRST of :"); scanf(" %c",&c);

FIRST(result,c); //Compute FIRST; Get Answer in 'result' array printf("\n FIRST(%c)= { ",c);

for(i=0;result[i]!='\0';i++)

printf(" %c ",result[i]); //Display result printf("}\n");

printf("press 'y' to continue : "); scanf(" %c",&choice);

}

while(choice=='y'||choice =='Y');

}

/\*

\*Function FIRST:

\*Compute the elements in FIRST(c) and write them

\*in Result Array.

\*/

void FIRST(char\* Result,char c)

{

int i,j,k;

char subResult[20]; int foundEpsilon; subResult[0]='\0'; Result[0]='\0';

//If X is terminal, FIRST(X) = {X}. if(!(isupper(c)))

{

addToResultSet(Result,c); return ;

}

//If X is non terminal

//Read each production for(i=0;i<numOfProductions;i++)

{

//Find production with X as LHS if(productionSet[i][0]==c)

{

//If X → ε is a production, then add ε to FIRST(X). if(productionSet[i][2]=='$') addToResultSet(Result,'$');

//If X is a non-terminal, and X → Y1 Y2 … Yk

//is a production, then add a to FIRST(X)

//if for some i, a is in FIRST(Yi),

//and ε is in all of FIRST(Y1), …, FIRST(Yi-1). else

{

j=2;

while(productionSet[i][j]!='\0')

{

foundEpsilon=0; FIRST(subResult,productionSet[i][j]); for(k=0;subResult[k]!='\0';k++)

addToResultSet(Result,subResult[k]); for(k=0;subResult[k]!='\0';k++)

if(subResult[k]=='$')

{

foundEpsilon=1; break;

}

//No ε found, no need to check next element if(!foundEpsilon)

break; j++;

}

}

}

}

return ;

}

/\* addToResultSet adds the computed

\*element to result set.

\*This code avoids multiple inclusion of elements

\*/

void addToResultSet(char Result[],char val)

{

int k;

for(k=0 ;Result[k]!='\0';k++) if(Result[k]==val)

return; Result[k]=val;

Result[k+1]='\0';

}

**Follow-** #include<stdio.h> #include<string.h> int n,m=0,p,i=0,j=0;

char a[10][10],followResult[10]; void follow(char c);

void first(char c);

void addToResult(char); int main()

{

int i;

int choice;

char c,ch;

printf("Enter the no.of productions: "); scanf("%d", &n);

printf(" Enter %d productions\nProduction with multiple terms should be give as separate productions \n", n);

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

scanf("%s%c",a[i],&ch);

// gets(a[i]);

do

{

m=0;

printf("Find FOLLOW of -->");

scanf(" %c",&c);

follow(c);

printf("FOLLOW(%c) = { ",c);

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

printf("%c ",followResult[i]);

printf(" }\n");

printf("Do you want to continue(Press 1 to continue. )?");

scanf("%d%c",&choice,&ch);

}

while(choice==1);

}

void follow(char c)

{

if(a[0][0]==c)addToResult('$');

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

{

for(j=2;j<strlen(a[i]);j++)

{

if(a[i][j]==c)

{

if(a[i][j+1]!='\0')first(a[i][j+1]);

if(a[i][j+1]=='\0'&&c!=a[i][0])

follow(a[i][0]);

}

}

}

}

void first(char c)

{

int k;

if(!(isupper(c)))

//f[m++]=c;

addToResult(c);

for(k=0;k<n;k++)

{

if(a[k][0]==c)

{

if(a[k][2]=='$') follow(a[i][0]);

else if(islower(a[k][2]))

//f[m++]=a[k][2];

addToResult(a[k][2]);

else first(a[k][2]);

}

}

}

void addToResult(char c)

{

int i;

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

if(followResult[i]==c)

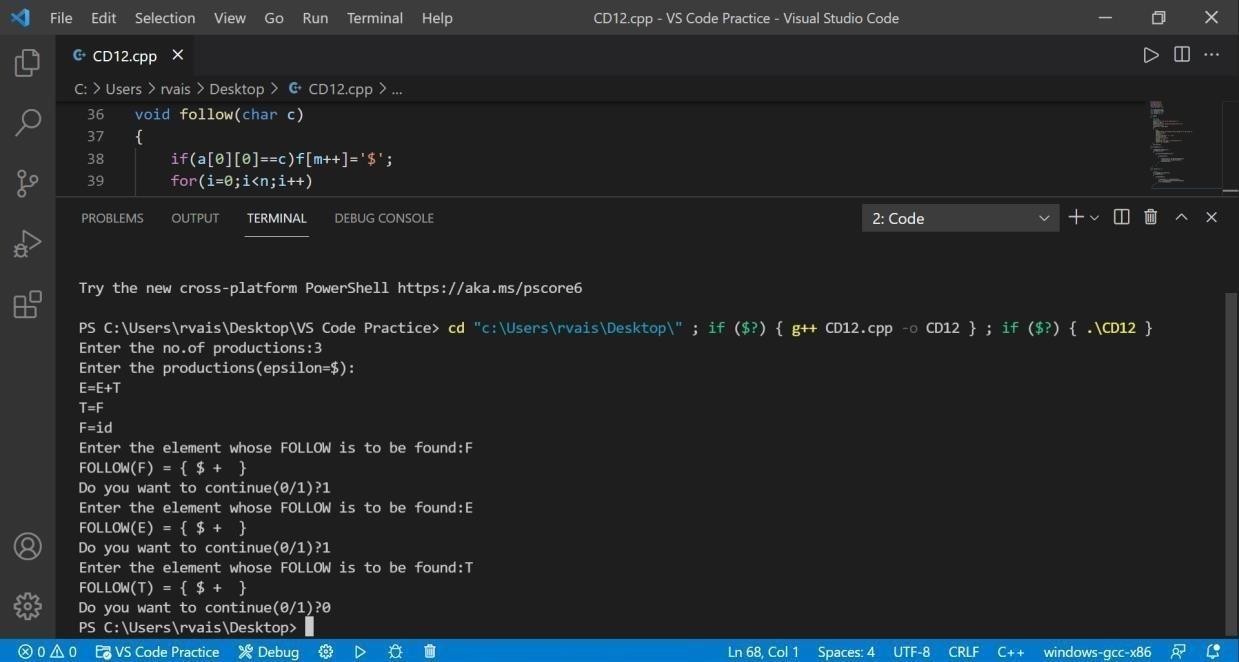
return;

followResult[m++]=c;

}

### Output:- 1 First-

**2 Follow -**



**Result:-**First and Follow computed successfully.

# EXPERIMENT 6

**Predictive Parsing table**

**Aim:** Write a program in c for construction of predictive parser table.

**Algorithm:-** Repeat: For each production A  of the grammar doFor each terminal in FIRST( )

add A   to M[A, a]if FIRST( ) contains 

add A   to M[A, b] for each bin FOLLOW(A)if is in FIRST( ) and $ is in FOLLOW(A)

add A   to M[A,$ ]

make each undefined entry of M be error

**Program:** #include<stdio.h> #include<string.h> #define TSIZE 128

int table[100][TSIZE]; char terminal[TSIZE]; char nonterminal[26]; struct product {

char str[100]; int len;

}pro[20]; int no\_pro;

char first[26][TSIZE]; char follow[26][TSIZE];

char first\_rhs[100][TSIZE]; int isNT(char c) {

return c >= 'A' && c <= 'Z';

}

void readFromFile() { FILE\* fptr;

fptr = fopen("text.txt", "r");

char buffer[255]; int i;

int j;

while (fgets(buffer, sizeof(buffer), fptr)) { printf("%s", buffer);

j = 0;

nonterminal[buffer[0] - 'A'] = 1;

for (i = 0; i < strlen(buffer) - 1; ++i) { if (buffer[i] == '|') {

++no\_pro;

pro[no\_pro - 1].str[j] = '\0'; pro[no\_pro - 1].len = j;

pro[no\_pro].str[0] = pro[no\_pro - 1].str[0]; pro[no\_pro].str[1] = pro[no\_pro - 1].str[1]; pro[no\_pro].str[2] = pro[no\_pro - 1].str[2]; j = 3;

}

else {

pro[no\_pro].str[j] = buffer[i];

++j;

if (!isNT(buffer[i]) && buffer[i] != '-' && buffer[i] != '>') { terminal[buffer[i]] = 1;

}

}

}

pro[no\_pro].len = j;

++no\_pro;

}

}

void add\_FIRST\_A\_to\_FOLLOW\_B(char A, char B) { int i;

for (i = 0; i < TSIZE; ++i) {

if (i != '^')

follow[B - 'A'][i] = follow[B - 'A'][i] || first[A - 'A'][i];

}

}

void add\_FOLLOW\_A\_to\_FOLLOW\_B(char A, char B) { int i;

for (i = 0; i < TSIZE; ++i) {

if (i != '^')

follow[B - 'A'][i] = follow[B - 'A'][i] || follow[A - 'A'][i];

}

}

void FOLLOW() { int t = 0;

int i, j, k, x;

while (t++ < no\_pro) { for (k = 0; k < 26; ++k) {

if (!nonterminal[k]) continue; char nt = k + 'A';

for (i = 0; i < no\_pro; ++i) { for (j = 3; j < pro[i].len; ++j) {

if (nt == pro[i].str[j]) {

for (x = j + 1; x < pro[i].len; ++x) { char sc = pro[i].str[x];

if (isNT(sc)) { add\_FIRST\_A\_to\_FOLLOW\_B(sc, nt); if (first[sc - 'A']['^'])

continue;

}

else {

follow[nt - 'A'][sc] = 1;

}

break;

}

if (x == pro[i].len) add\_FOLLOW\_A\_to\_FOLLOW\_B(pro[i].str[0], nt);

}

}

}

}

}

}

void add\_FIRST\_A\_to\_FIRST\_B(char A, char B) { int i;

for (i = 0; i < TSIZE; ++i) {

if (i != '^') {

first[B - 'A'][i] = first[A - 'A'][i] || first[B - 'A'][i];

}

}

}

void FIRST() { int i, j;

int t = 0;

while (t < no\_pro) {

for (i = 0; i < no\_pro; ++i) {

for (j = 3; j < pro[i].len; ++j) { char sc = pro[i].str[j];

if (isNT(sc)) { add\_FIRST\_A\_to\_FIRST\_B(sc, pro[i].str[0]); if (first[sc - 'A']['^'])

continue;

}

else {

first[pro[i].str[0] - 'A'][sc] = 1;

}

break;

}

if (j == pro[i].len) first[pro[i].str[0] - 'A']['^'] = 1;

}

++t;

}

}

void add\_FIRST\_A\_to\_FIRST\_RHS B(char A, int B) { int i;

for (i = 0; i < TSIZE; ++i) {

if (i != '^')

first\_rhs[B][i] = first[A - 'A'][i] || first\_rhs[B][i];

}

}

void FIRST\_RHS() { int i, j;

int t = 0;

while (t < no\_pro) {

for (i = 0; i < no\_pro; ++i) { for (j = 3; j < pro[i].len; ++j) { char sc = pro[i].str[j];

if (isNT(sc)) { add\_FIRST\_A\_to\_FIRST\_RHS B(sc, i); if (first[sc - 'A']['^'])

continue;

}

else { first\_rhs[i][sc] = 1;

}

break;

}

if (j == pro[i].len)

first\_rhs[i]['^'] = 1;

}

++t;

}

}

int main() { readFromFile();

follow[pro[0].str[0] - 'A']['$'] = 1; FIRST();

FOLLOW(); FIRST\_RHS();

int i, j, k;

for (i = 0; i < no\_pro; ++i) {

if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) { char c = pro[i].str[0];

printf("FIRST OF %c: ", c); for (j = 0; j < TSIZE; ++j) {

if (first[c - 'A'][j]) { printf("%c ", j);

}

}

printf("\n");

}

}

// display follow of each variable printf("\n");

for (i = 0; i < no\_pro; ++i) {

if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) { char c = pro[i].str[0];

printf("FOLLOW OF %c: ", c); for (j = 0; j < TSIZE; ++j) {

if (follow[c - 'A'][j]) { printf("%c ", j);

}

}

printf("\n");

}

}

printf("\n");

for (i = 0; i < no\_pro; ++i) { printf("FIRST OF %s: ", pro[i].str); for (j = 0; j < TSIZE; ++j) {

if (first\_rhs[i][j]) {

printf("%c ", j);

}

}

printf("\n");

}

terminal['$'] = 1;

terminal['^'] = 0;

printf("\n");

printf("\n\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* LL(1) PARSING TABLE

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\t \n"); printf("%-10s", "");

for (i = 0; i < TSIZE; ++i) {

if (terminal[i]) printf("%-10c", i);

}

printf("\n"); int p = 0;

for (i = 0; i < no\_pro; ++i) {

if (i != 0 && (pro[i].str[0] != pro[i - 1].str[0])) p = p + 1;

for (j = 0; j < TSIZE; ++j) {

if (first\_rhs[i][j] && j != '^') { table[p][j] = i + 1;

}

else if (first\_rhs[i]['^']) {

for (k = 0; k < TSIZE; ++k) {

if (follow[pro[i].str[0] - 'A'][k]) { table[p][k] = i + 1;

}

}

}

}

}

k = 0;

for (i = 0; i < no\_pro; ++i) {

if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {

printf("%-10c", pro[i].str[0]); for (j = 0; j < TSIZE; ++j) {

if (table[k][j]) {

printf("%-10s", pro[table[k][j] - 1].str);

}

else if (terminal[j]) { printf("%-10s", "");

}

}

++k;

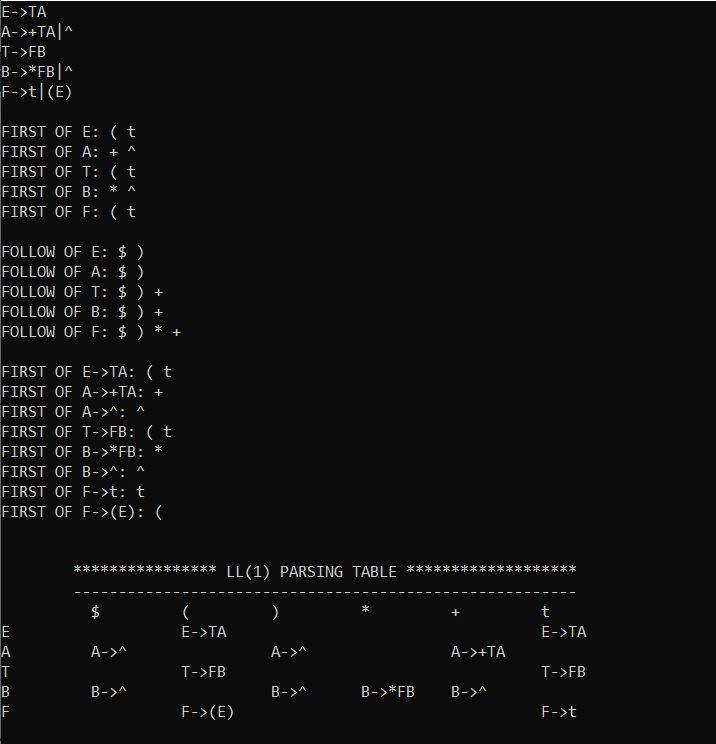
printf("\n");

}

}

}

### Output:-

****

**Result:-** The Program Executed successfully.

# EXPERIMENT 7

**Shift ReduceParsing**

**Aim:-**Computation of Shift ReduceParsing.

### Algorithm:-

* Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.
* Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string.
* Sift reduce parsing performs the two actions: shift and reduce. That's why it is known as shift reduces parsing.
* At the shift action, the current symbol in the input string is pushed to a stack.
* 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.

**Program:-** #include<stdio.h> #include<iostream.h> #include<ctype.h> #include<string.h> #include<conio.h> struct stru1

{

char non\_ter[1],pro[25];

}cfg[25];

int n,st=-1,j,i,t=-1,m; int v,c,p=1;

char str[20],stack[20],ch,tmp[10]; void match(int k);

void matchl(int k); void main()

{

clrscr();

cprintf("Enter the number of productions:\n\r"); cscanf("%d",&n);

cprintf("\n\r");

cprintf("Enter the productions on LEFT and RIGHT sides:\n\r");

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

{

cscanf("%s",cfg[i].non\_ter); cprintf("\n\r");

cprintf("->\n\r"); cscanf("%s",cfg[i].pro); cprintf("\n\r");

}

cprintf("Enter the input string:\n\r"); cscanf("%s",str);

cprintf("\n\r"); i=0;

do

{

ch=str[i]; stack[++st]=ch; tmp[0]=ch; match(1);

i++;

}while(str[i]!='\0'); c=st;

v=st; cputs(stack); cprintf("\n\r"); while(st!=0)

{

v=--st; t=-1; p=0;

while(v<=c)

{

tmp[++t]=stack[v++]; p++;

}

matchl(p);

}

cfg[0].non\_ter[1]='\0'; if(strcmp(stack,cfg[0].non\_ter)==0) cprintf("String is present in Grammar G\n\r"); else

cprintf("String is not present in Grammar G\n\r");

}

void match(int k)

{

for(j=0;j<n;j++)

{

if(strlen(cfg[j].pro)==k)

{

if(strcmp(tmp,cfg[j].pro)==0)

{

stack[st]=cfg[j].non\_ter[0]; break;

}

}

}

}

void matchl(int k)

{

int x=1,y; y=k-1;

for(j=0;j<n;j++)

{

if(strlen(cfg[j].pro)==k)

{

if(strcmp(tmp,cfg[j].pro)==0)

{

k=c-k+1; stack[k]=cfg[j].non\_ter[0]; do

{

stack[k+x]='\0';

tmp[t--]='\0'; c--;

x++;

}while(x<=y);

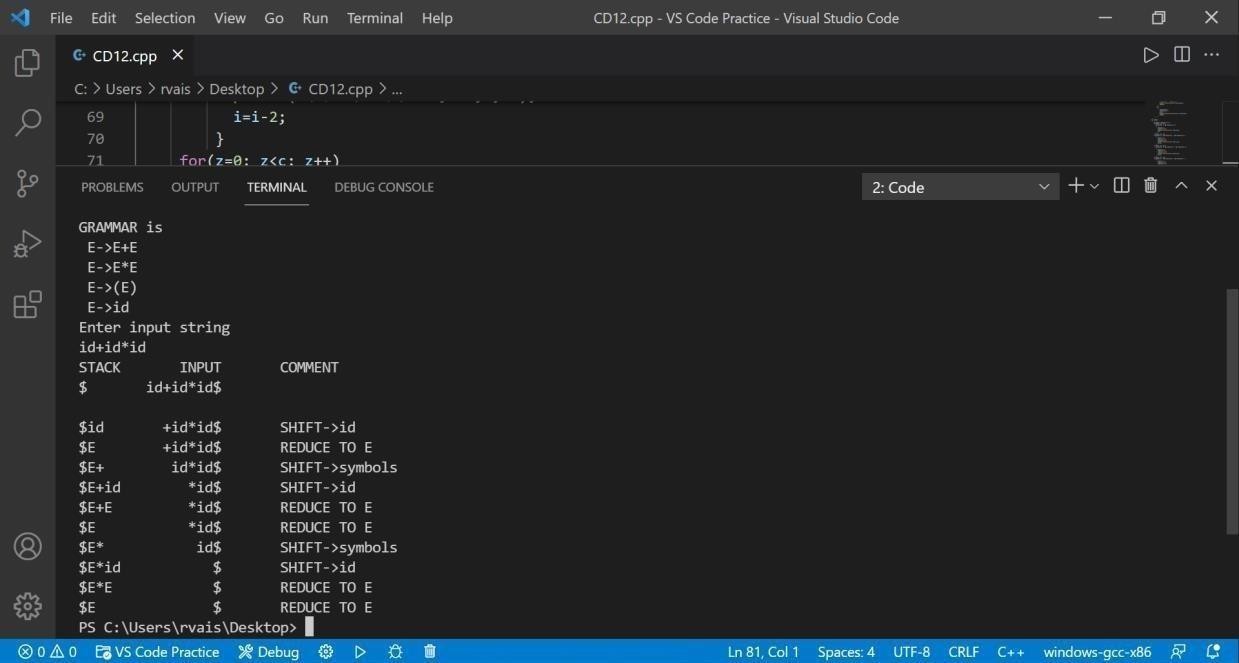
tmp[t]='\0'; cputs(stack); cprintf("\n\r"); break;}

}

}

}

### Output:-



**Result:** The Program Executed successfully.

# EXPERIMENT 8

**Leading and Trailing Aim:** Write a program for finding the leading and trailing. **Algorithm:-**

**Lead** :- Lead is a list of all those terminals symbols("operators") which can

appear first on any right hand side of a production.

For each non-terminal i.e. left hand side,a Lead list containing the first terminalin each production for that non-terminal.Where a non-terminal is the first symbol on the right hand side, include both it and the first terminal following.e.g. for

X → a. / Bc

includes a,c and B in X's Lead List.

**Trail** :- Trail or Last is a similar list of those terminals which can appear Last.

For each non-terminal i.e. left hand side,a Last or trail list containing the last terminal in each production for that non terminal.Where a non-terminal is the last symbol on the right hand side, include both it and the last terminal

.e.g. for

Y → ....u / vW

includes u,v and W in Last or Trail list.

**Program:-** #include<iostream.h> #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]);

}

}

void 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]<<",";

}

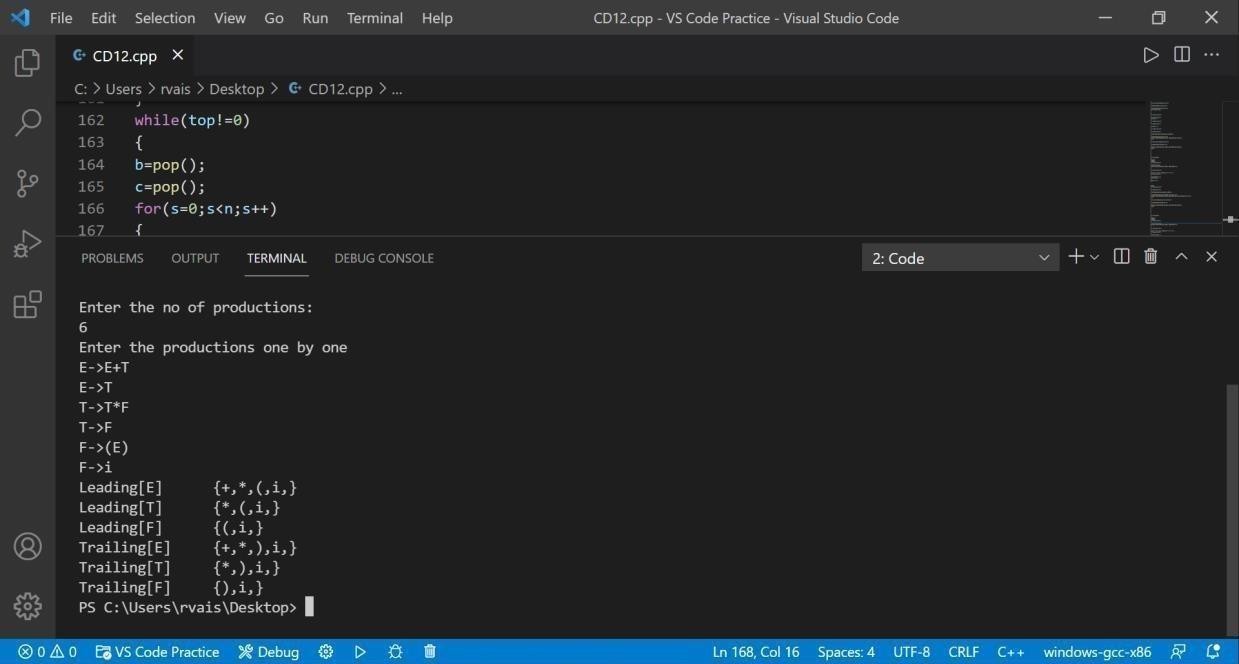
cout<<"}\n";

}

getch();

}

### OUTPUT:



**Result:** The Program Executed successfully

# EXPERIMENT 9

**Computation of LR(0) items Aim:-** Computation of LR(0) items.

### Algorithm:-

* + An LR (0) item is a production G with dot at some position on the right side of the production.
* LR(0) items is useful to indicate that how much of the input has been scanned up to a given point in the process of parsing.
* In the LR (0), we place the reduce node in the entire row.

**Program:-** #include<stdio.h> #include<conio.h> char stack[30];

int top=-1;

void push(char c)

{

top++; stack[top]=c;

}

char pop()

{

char c; if(top!=-1)

{

c=stack[top]; top--;

return c;

}

return'x';

}

void printstat()

{

int i; printf("\n\t\t\t $");

for(i=0;i<=top;i++) printf("%c",stack[i]);

}

void main()

{

int i,j,k,l;

char s1[20],s2[20],ch1,ch2,ch3; clrscr();

printf("\n\n\t\t LR PARSING");

printf("\n\t\t ENTER THE EXPRESSION"); scanf("%s",s1);

l=strlen(s1); j=0;

printf("\n\t\t $"); for(i=0;i<l;i++)

{

if(s1[i]=='i' && s1[i+1]=='d')

{ s1[i]=' ';

s1[i+1]='E';

printstat(); printf("id"); push('E');

printstat();

}

else if(s1[i]=='+'||s1[i]=='-'||s1[i]=='\*' ||s1[i]=='/' ||s1[i]=='d')

{

push(s1[i]); printstat();

}

}

printstat(); l=strlen(s2); while(l)

{

ch1=pop(); if(ch1=='x')

{

printf("\n\t\t\t $"); break;

}

if(ch1=='+'||ch1=='/'||ch1=='\*'||ch1=='-')

{

ch3=pop(); if(ch3!='E')

{

printf("errror"); exit();

}

else

{

push('E'); printstat();

}

}

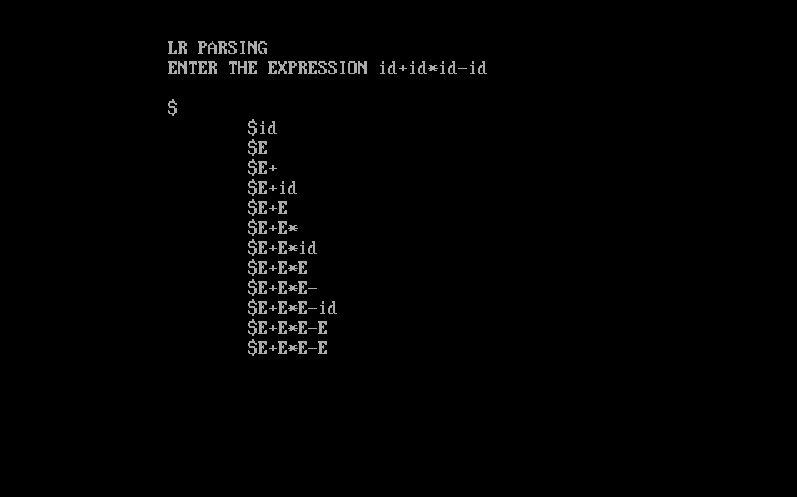
ch2=ch1;

}

getch();

}

### OUTPUT:-

****

**Result:-** The Program Executed successfully.

# EXPERIMENT 10

**Postfix and Prefix Aim:-**Intermediate code generation Postfix, Prefix.

### Algorithm:-

1. **Postfix**

|  |  |
| --- | --- |
| 1. | Scan the Infix string from left to right. |
| 2. | Initialize an empty stack. |
| 3. | If the scanned character is an operand, add it to the Postfix string. |
| 4. | If the scanned character is an operator and if the stack is empty push the character to stack. |
| 5. | If the scanned character is an Operator and the stack is not empty, compare the precedence of the character with the element on top of the stack. |
| 6. | If top Stack has higher precedence over the scanned character pop the  stack else push the scanned character to stack. Repeat this step until the stack is not empty and top Stack has precedence over the character. |
| 7. | Repeat 4 and 5 steps till all the characters are scanned. |
| 8. | After all characters are scanned, we have to add any character that the  stack may have to the Postfix string. |
| 9. | If stack is not empty add top Stack to Postfix string and Pop the stack. |
| 10. | Repeat this step as long as stack is not empty. |

1. **Prefix**
   1. Push “)” onto STACK, and add “(“ to end of the A
   2. Scan A from right to left and repeat step 3 to 6 for each element of A until theSTACK is empty
   3. If an operand is encountered add it to B
   4. If a right parenthesis is encountered push it onto STACK
   5. If an operator is encountered then:
   6. Repeatedly pop from STACK and add to B each operator (on the top ofSTACK) which has same or higher precedence than the operator.
   7. Add operator to STACK
   8. If left parenthesis is encontered then:
      1. A.Repeatedly pop from the STACK and add to B (each operator on top of stackuntil a left parenthesis is encounterd)
      2. B.Remove the left parenthesis
   9. Exit

### Program:-

1. **Postfix** #include<bits/stdc++.h> using namespace std; int prec(char ch) {

if (ch == '^')

return 3;

else if (ch == '/' || ch == '\*') return 2;

else if (ch == '+' || ch == '-') return 1;

else

}

{

return -1;

stack<char> st; string ans = “”“”;

for (int i = 0; i < s.length(); i++) { char ch = s[i];

If ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') || (ch >= '0' && ch <= '9'))

ans += ch; else if (ch == '(')

st.push('(');

else if (ch == ')') { while (st.top() != '(')

{

ans += st.top(); st.pop();

}

st.pop();

}

else {

while (!st.empty() && prec(s[i]) <= prec(st.top())) { ans += st.top();

st.pop();

}

st.push(ch); }

}

while (!st.empty()) {

ans += st.top(); st.pop();

}

return ans;

}

int main() {

string s; cin >> s;

cout << infixToPostfix(s); return 0;

}

### Prefix

#include <stdio.h> #include <conio.h> #include <string.h> #include <ctype.h> #define MAX 50 struct infix

{

char target[MAX] ; char stack[MAX] ; char \*s, \*t ;

int top, l ;

} ;

void initinfix ( struct infix \* ) ;

void setexpr ( struct infix \*, char \* ) ; void push ( struct infix \*, char ) ; char pop ( struct infix \* ) ;

void convert ( struct infix \* ) ; int priority ( char c ) ;

void show ( struct infix ) ;

void main( )

{

struct infix q ; char expr[MAX] ; clrscr( ) ;

initinfix ( &q ) ;

printf ( "\nEnter an expression in infix form: " ) ; gets ( expr ) ;

setexpr ( &q, expr ) ; convert ( &q ) ;

printf ( "The Prefix expression is: " ) ; show ( q ) ;

getch( ) ;

}

/\* initializes elements of structure variable \*/ void initinfix ( struct infix \*pq )

{

pq -> top = -1 ;

strcpy ( pq -> target, "" ) ;

strcpy ( pq -> stack, "" ) ; pq -> l = 0 ;

}

/\* reverses the given expression \*/

void setexpr ( struct infix \*pq, char \*str )

{

pq -> s = str ; strrev ( pq -> s ) ;

pq -> l = strlen ( pq -> s ) ;

\*( pq -> target + pq -> l ) = '\0' ;

pq -> t = pq -> target + ( pq -> l - 1 ) ;

}

/\* adds operator to the stack \*/ void push ( struct infix \*pq, char c )

{

if ( pq -> top == MAX - 1 )

printf ( "\nStack is full.\n" ) ;

else

{

}

}

pq -> top++ ;

pq -> stack[pq -> top] = c ;

/\* pops an operator from the stack \*/ char pop ( struct infix \*pq )

{

if ( pq -> top == -1 )

{

}

else

{

}

}

printf ( "Stack is empty\n" ) ; return -1 ;

char item = pq -> stack[pq -> top] ; pq -> top-- ;

return item ;

/\* converts the infix expr. to prefix form \*/ void convert ( struct infix \*pq )

{

char opr ;

while ( \*( pq -> s ) )

{

if ( \*( pq -> s ) == ' ' || \*( pq -> s ) == '\t' )

{

pq -> s++ ; continue ;

}

if ( isdigit ( \*( pq -> s ) ) || isalpha ( \*( pq -> s ) ) )

{

while ( isdigit ( \*( pq -> s ) ) || isalpha ( \*( pq -> s ) ) )

{

\*( pq -> t ) = \*( pq -> s ) ;

pq -> s++ ; pq -> t-- ;

}

}

if ( \*( pq -> s ) == ')' )

{

push ( pq, \*( pq -> s ) ) ; pq -> s++ ;

}

if ( \*( pq -> s ) == '\*' || \*( pq -> s ) == '+' || \*( pq -> s ) == '/' || \*(

pq -> s ) == '%' || \*( pq -> s ) == '-' || \*( pq -> s ) == '$' )

{

if ( pq -> top != -1 )

{

opr = pop ( pq ) ;

while ( priority ( opr ) > priority ( \*( pq -> s ) ) )

{

\*( pq -> t ) = opr ;

pq -> t-- ;

opr = pop ( pq ) ;

}

}

else

}

push ( pq, opr ) ;

push ( pq, \*( pq -> s ) ) ;

push ( pq, \*( pq -> s ) ) ; pq -> s++ ;

if ( \*( pq -> s ) == '(' )

{

opr = pop ( pq ) ; while ( opr != ')' )

{

\*( pq -> t ) = opr ;

pq -> t-- ;

opr = pop ( pq ) ;

}

pq -> s++ ;

}

}

while ( pq -> top != -1 )

{

opr = pop ( pq ) ;

\*( pq -> t ) = opr ;

pq -> t-- ;

}

pq -> t++ ;

}

/\* returns the priotity of the operator \*/ int priority ( char c )

{

if ( c == '$' )

return 3 ;

if ( c == '\*' || c == '/' || c == '%' ) return 2 ;

else

{

if ( c == '+' || c == '-' ) return 1 ;

else

}

}

return 0 ;

/\* displays the prefix form of given expr. \*/ void show ( struct infix pq )

{

while ( \*( pq.t ) )

{

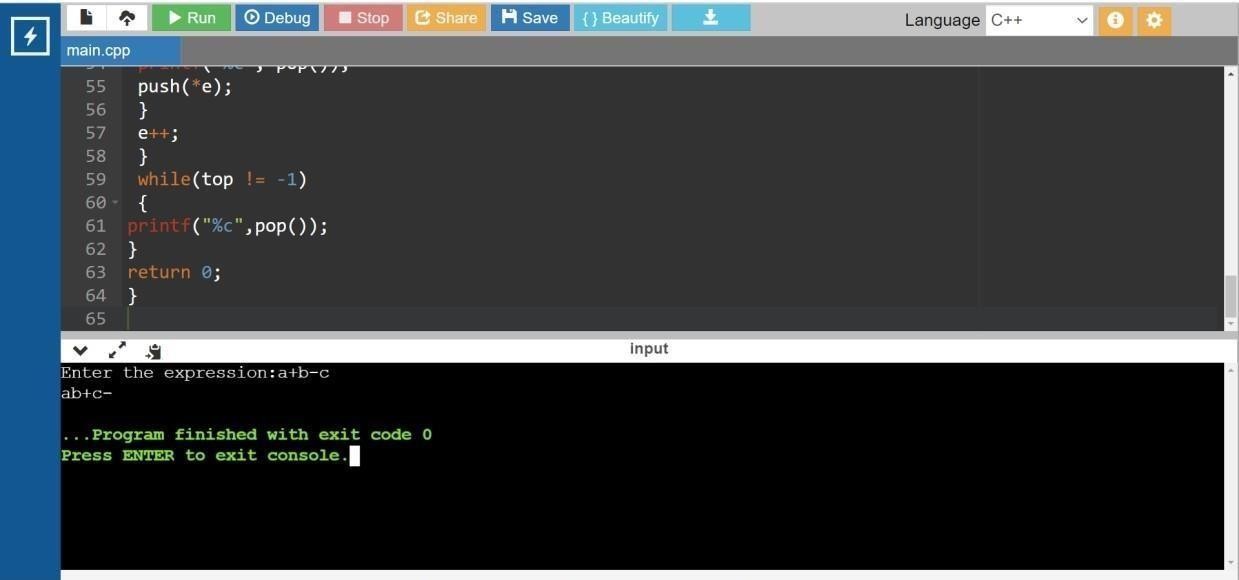
printf ( " %c", \*( pq.t ) ) ; pq.t++ ;

}

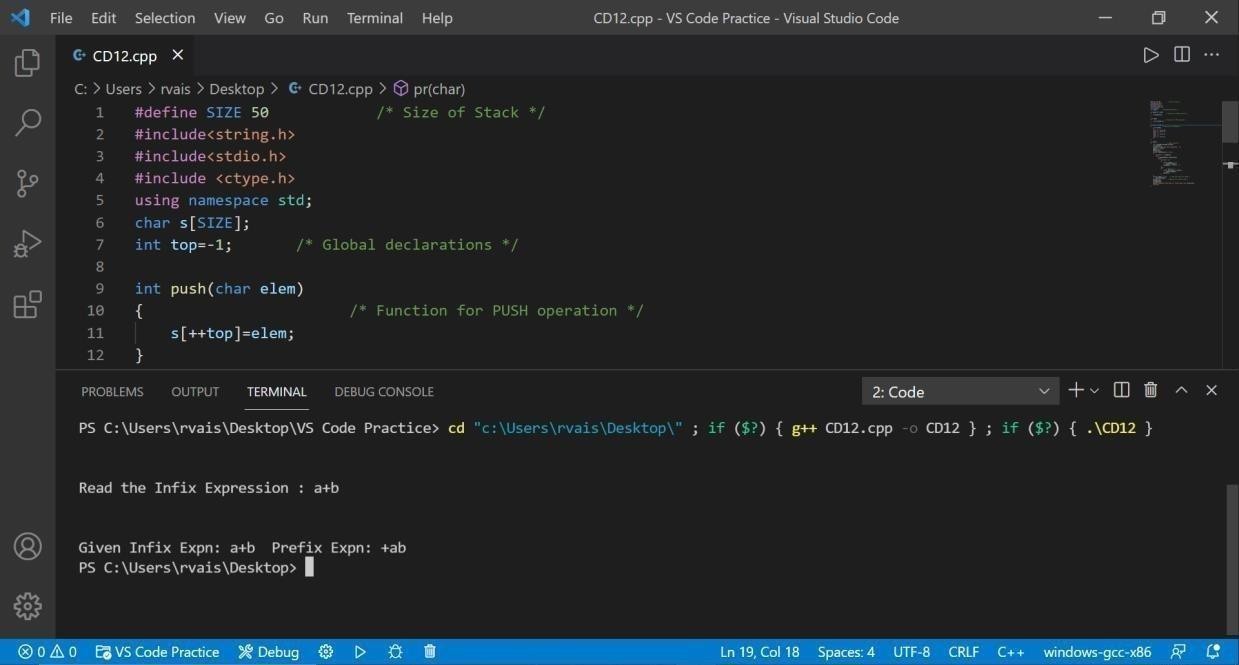
}

### Output:-

* 1. **Postfix**



* 1. **Prefix**



**Result:** The Program Executed successfully.

# EXPERIMENT 11

### Quadruple, Triple, Indirect triple

**Aim:-**Intermediate Code Generation Quadruple,Triple,Indirect triple

### Algorithm:-

* + 1. **Quadruple**
       - Using quadruple representation, the three-address statement x = y op z is represented by placing op in the operator field, y in the operand1 field, z in the operand2 field, and x in the result field.
       - The statement x = op y , where op is a unary operator, is represented by placing op in the operator field, y in the operand1 field, and x in the result field; the operand2 field is not used.
       - A statement like param t 1 is represented by placing param in the operator field and t 1 in the operand1 field; neither operand2 nor the result field are used.
       - Unconditional and conditional jump statements are represented by placing the target labels in the result field.

### Triple

* + - * The contents of the operand1, operand2, and result fields are therefore normally the pointers to the symbol records for the names represented by these fields.
      * Hence, it becomes necessary to enter temporary names into the symbol table as they are created.
      * This can be avoided by using the position of the statement to refer to a temporary value.
      * If this is done, then a record structure with three fields is enough to represent the three-address statements:
      * The first holds the operator value, and the next two holding values for the operand1 and operand2, respectively. Such a representation is called a "triple representation".
      * The contents of the operand1 and operand2 fields are either pointers to the symbol table records, or they are pointers to records (for temporary names) within the triple representation itself.

### Indirect triple

* + - * Another representation uses an additional array to list the pointers to the triples in the desired order.
      * This is called an indirect triple representation. For example, a triple representation of the three-address code for the statement x = ( a + b )\* ˆ’ c/d

### Program:-

**A.Quadruple**

#include<stdio.h> #include<string.h> main()

{

char line[20]; int s[20];

int t=1;

int i=0; printf("Enter string.. :");

gets(line); for(i=0;i<20;i++)s[i]=0; printf("op\ta1\ta2\tres\n"); for(i=2;line[i]!='\0';i++)

{

if(line[i]=='/' || line[i]=='\*')

{

printf("\n");

if(s[i]==0)

{

if(s[i+1]==0)

{

printf(":=\t%c\t\t t%d\n",line[i+1],t); s[i+1]=t++;

}

printf("%c\t",line[i]);

(s[i-1]==0)?printf("%c\t",line[i-1]):printf("t%d\t",s[i-1]); printf("t%d \t t%d",s[i+1],t);

s[i-1]=s[i+1]=t++; s[i]=1;

}

}

}

for(i=2;line[i]!='\0';i++)

{

if(line[i]=='+' || line[i]=='-')

{

printf("\n");

if(s[i]==0)

{

if(s[i+1]==0)

{

printf(":=\t%c\t\t t%d\n",line[i+1],t); s[i+1]=t++;

}

printf("%c\t",line[i]);

(s[i-1]==0)?printf("%c\t",line[i-1]):printf("t%d\t",s[i-1]); printf("t%d \t t%d",s[i+1],t);

s[i-1]=s[i+1]=t++; s[i]=1;

}

}

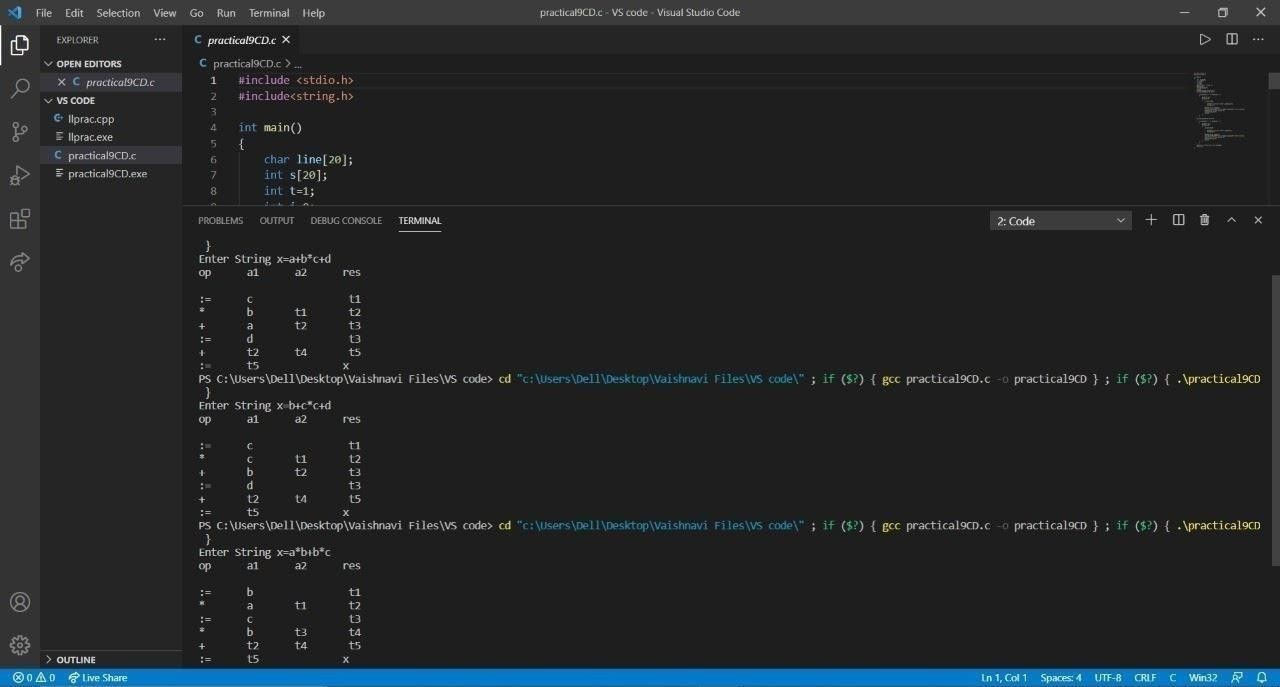
}

printf("\n:=\tt%d\t\t%c",t-1,line[0]);

getch();

}

**Output-**

****

**Result:** The Program Executed successfully.

# EXPERIMENT 12

**Simple Code Generator**

## **Aim:-**A simple code Generator.

### Algorithm:-

1. Start
2. Get address code sequence.
3. Determine current location of 3 using address (for 1st operand).
4. If current location not already exist generate move (B,O).
5. Update address of A(for 2nd operand).
6. If current value of B and () is null,exist.
7. If they generate operator () A,3 ADPR.
8. Store the move instruction in memory
9. Stop

**Program:-** #include<stdio.h> #include<string.h> void pm();

void plus(); void div();

int i,ch,j,l,addr=100;

char ex[10], exp[10] ,exp1[10],exp2[10],id1[5],op[5],id2[5]; void main()

{

clrscr(); while(1)

{

printf("\n1.assignment\n2.arithmetic\n3.relational\n4.Exit\nEnter the choice:");

scanf("%d",&ch); switch(ch)

{

case 1:

printf("\nEnter the expression with assignment operator:"); scanf("%s",exp);

l=strlen(exp); exp2[0]='\0'; i=0;

while(exp[i]!='=')

{ i++;

}

strncat(exp2,exp,i); strrev(exp); exp1[0]='\0';

strncat(exp1,exp,l-(i+1)); strrev(exp1);

printf("Three address code:\ntemp=%s\n%s=temp\n",exp1,exp2); break;

case 2:

printf("\nEnter the expression with arithmetic operator:"); scanf("%s",ex);

strcpy(exp,ex); l=strlen(exp); exp1[0]='\0';

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

{

if(exp[i]=='+'||exp[i]=='-')

{

if(exp[i+2]=='/'||exp[i+2]=='\*')

{

pm(); break;

}

else

{

plus(); break;

}

}

else if(exp[i]=='/'||exp[i]=='\*')

{

div(); break;

}

}

break;

case 3:

printf("Enter the expression with relational operator");

scanf("%s%s%s",&id1,&op,&id2); if(((strcmp(op,"<")==0)||(strcmp(op,">")==0)||(strcmp(op,"<=")==0)||(strcmp

(op,">=")==0)||(strcmp(op,"==")==0)||(strcmp(op,"!=")==0))==0) printf("Expression is error");

else

{

printf("\n%d\tif %s%s%s goto %d",addr,id1,op,id2,addr+3); addr++;

printf("\n%d\t T:=0",addr); addr++;

printf("\n%d\t goto %d",addr,addr+2); addr++;

printf("\n%d\t T:=1",addr);

}

break; case 4:

exit(0);

}

}

}

void pm()

{

strrev(exp); j=l-i-1;

strncat(exp1,exp,j); strrev(exp1); printf("Three address

code:\ntemp=%s\ntemp1=%c%ctemp\n",exp1,exp[j+1],exp[j]);

}

void div()

{

strncat(exp1,exp,i+2); printf("Three address

code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3]);

}

void plus()

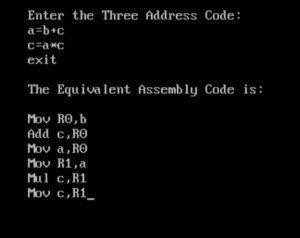
{

strncat(exp1,exp,i+2); printf("Three address

code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3]);

}

### OUTPUT:

****

**Result:** The Program Executed successfully

# EXPERIMENT 13

### Implementation of DAG

**Aim:-** Implementation of DAG

### Algorithm:-

* + Start the program
  + Include all the header files
  + Check for postfix expression and construct the in order DAG representation
  + Print the output
  + Stop the program

### Program:-

#include<stdio.h> main()

{

struct da

{

int ptr,left,right;char label;

}

dag[25];

int ptr,l,j,change,n=0,i=0,state=1,x,y,k;

char store,\*input1,input[25],var; clrscr();

for(i=0;i<25;i++)

{

dag[i].ptr=NULL;

dag[i].left=NULL; dag[i].right=NULL; dag[i].label=NULL;

}

printf("\n\nENTER THE EXPRESSION\n\n"); scanf("%s",input1); for(i=0;i<25;i++)input[i]=NULL; l=strlen(input1);

a: for(i=0;input1[i]!=')';i++);

for(j=i;input1[j]!='(';j--); for(x=j+1;x<i;x++)

if(isalpha(input1[x])) input[n++]=input1[x]; else

if(input1[x]!='0') store=input1[x]; input[n++]=store; for(x=j;x<=i;x++) input1[x]='0'; if(input1[0]!='0')goto a; for(i=0;i<n;i++)

{

dag[i].label=input[i]; dag[i].ptr=i;

if(!isalpha(input[i])&&!isdigit(input[i]))

{

dag[i].right=i-1; ptr=i; var=input[i-1]; if(isalpha(var)) ptr=ptr-2;

else

{

ptr=i-1;b: if(!isalpha(var)&&!isdigit(var))

{

ptr=dag[ptr].left;

var=input[ptr]; goto b;

}

else ptr=ptr-1;

}

dag[i].left=ptr;

}

}

printf("\n SYNTAX TREE FOR GIVEN EXPRESSION\n\n");

printf("\n\n PTR \t\t LEFT PTR \t\t RIGHT PTR \t\t LABEL\n\n"); for(i=0;i<n;i++) printf("\n%d\t%d\t%d\t%c\n",dag[i].ptr,dag[i].left,dag[i].right,dag[i].label); getch();

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

{

for(j=0;j<n;j++)

{

if((dag[i].label==dag[j].label&&dag[i].left==dag[j].left)&&dag[i].right==dag[j]

.right)

{

for(k=0;k<n;k++)

{

if(dag[k].left==dag[j].ptr)dag[k].left=dag[i].ptr; if(dag[k].right==dag[j].ptr)dag[k].right=dag[i].ptr;

}

dag[j].ptr=dag[i].ptr;

}

}

}

printf("\n DAG FOR GIVEN EXPRESSION\n\n"); printf("\n\nPTR \t LEFT PTR \t RIGHT PTR \t LABEL \n\n"); for(i=0;i<n;i++)

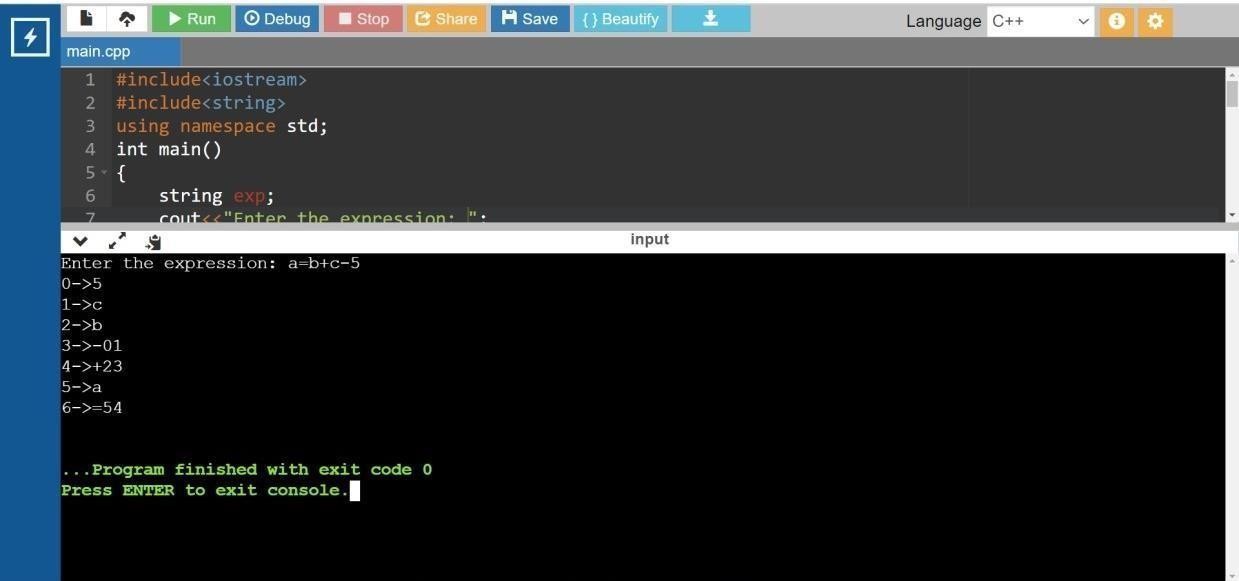
printf("\n%d\t\t%d\t\t%d\t\t%c\n",dag[i].ptr,dag[i].left,dag[i].right,dag[i].label)

;

getch();

}

### OUTPUT:-



**Result:** The Program Executed successfully.

**EXPERIMENT 14**

**Global Data flow Analysis Aim:-** Implementation of Global Data flow Analysis.

### Algorithm:-

Step-1: Start the Program Execution.

Step-2: Read the total Numbers of Expression

Step-3: Read the Left and Right side of Each Expressions Step-4: Display the Expressions with Line No

Step-5: Display the Data flow movement with Particular Expressions Step-6: Stop the Program Execution.

**Program:-** #include <stdio.h> #include <conio.h> #include <string.h > struct op

{

char l[20]; char r[20];

}

op[10], pr[10];

void main()

{

int a, i, k, j, n, z = 0, m, q,lineno=1; char \* p, \* l;

char temp, t;

char \* tem;char \*match; clrscr();

printf("enter no of values"); scanf("%d", & n);

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

{

printf("\tleft\t"); scanf("%s",op[i].l); printf("\tright:\t"); scanf("%s", op[i].r);

}

printf("intermediate Code\n");

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

{ printf("Line No=%d\n",lineno); printf("\t\t\t%s=", op[i].l);

printf("%s\n", op[i].r); lineno++;

}

printf("\*\*\*Data Flow Analysis for the Above Code \*\*\*\n"); for(i=0;i<n;i++)

{

for(j=0;j<n;j++)

{

match=strstr(op[j].r,op[i].l); if(match)

{

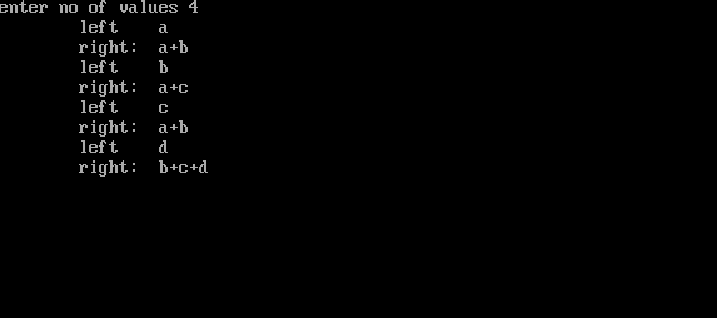
printf("\n %s is live at %s \n ", op[i].l,op[j].r);

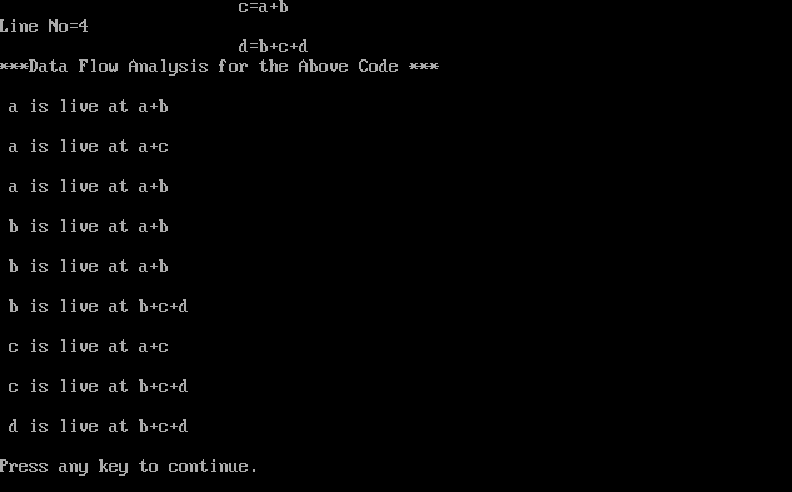
}

}

}}

### OUTPUT:-

****



**Result:-** The Program Executed successfully.

**EXPERIMENT 15**

**Storage Allocation Strategies**

**Aim:-**Implement any one storage allocation strategies (heap, stack, static)

### Algorithm:-

Step1: Initially check whether the stack is empty

Step2: Insert an element into the stack using push operation

Step3: Insert more elements onto the stack until stack becomes full Step4: Delete an element from the stack using pop operation Step5: Display the elements in the stack

Step6: Top the stack element will be displayed

### Program:-

//implementation of heap allocation storage strategies// #include<stdio.h>

#include<stdlib.h> #define TRUE 1

#define FALSE 0 typedef struct Heap

{

int data;

struct Heap \*next;

}

node;

node \*create(); void main()

{

int choice,val; char ans; node \*head;

void display(node \*); node \*search(node \*,int); node \*insert(node \*); void dele(node \*\*); head=NULL;

do

{

printf("\nprogram to perform various operations on heap using dynamic memory management");

printf("\n1.create");

printf("\n2.display");

printf("\n3.insert an element in a list"); printf("\n4.delete an element from list"); printf("\n5.quit");

printf("\nenter your chioce(1-5)"); scanf("%d",&choice); switch(choice)

{

case 1:head=create(); break;

case 2:display(head); break;

case 3:head=insert(head); break;

case 4:dele(&head); break;

case 5:exit(0); default:

printf("invalid choice,try again");

}

}

while(choice!=5);

}

node\* create()

{

node \*temp,\*New,\*head; int val,flag;

char ans='y';

node \*get\_node(); temp=NULL; flag=TRUE;

do

{

printf("\n enter the element:"); scanf("%d",&val); New=get\_node(); if(New==NULL)

printf("\nmemory is not allocated"); New->data=val;

if(flag==TRUE)

{

head=New; temp=head;

flag=FALSE;

}

else

{

temp->next=New; temp=New;

}

printf("\ndo you want to enter more elements?(y/n)");

}

while(ans=='y');

printf("\nthe list is created\n"); return head;

}

node \*get\_node()

{

node \*temp; temp=(node\*)malloc(sizeof(node)); temp->next=NULL;

return temp;

}

void display(node \*head)

{

node \*temp; temp=head; if(temp==NULL)

{

printf("\nthe list is empty\n"); return;

}

while(temp!=NULL)

{

printf("%d->",temp->data); temp=temp->next;

}

printf("NULL");

}

node \*search(node \*head,int key)

{

node \*temp; int found; temp=head;

if(temp==NULL)

{

printf("the linked list is empty\n"); return NULL;

}

found=FALSE;

while(temp!=NULL && found==FALSE)

{

if(temp->data!=key) temp=temp->next; else

found=TRUE;

}

if(found==TRUE)

{

printf("\nthe element is present in the list\n"); return temp;

}

else

{

printf("the element is not present in the list\n"); return NULL;

}

}

node \*insert(node \*head)

{

int choice;

node \*insert\_head(node \*); void insert\_after(node \*); void insert\_last(node \*);

printf("n1.insert a node as a head node"); printf("n2.insert a node as a head node");

printf("n3.insert a node at intermediate position in t6he list"); printf("\nenter your choice for insertion of node:"); scanf("%d",&choice);

switch(choice)

{

case 1:head=insert\_head(head); break;

case 2:insert\_last(head); break;

case 3:insert\_after(head); break;

}

return head;

}

node \*insert\_head(node \*head)

{

node \*New,\*temp; New=get\_node();

printf("\nEnter the element which you want to insert"); scanf("%d",&New->data);

if(head==NULL) head=New;

else

{

temp=head;

New->next=temp; head=New;

}

return head;

}

void insert\_last(node \*head)

{

node \*New,\*temp; New=get\_node();

printf("\nenter the element which you want to insert"); scanf("%d",&New->data);

if(head==NULL) head=New;

else

{

temp=head;

while(temp->next!=NULL) temp=temp->next;

temp->next=New; New->next=NULL;

}

}

void insert\_after(node \*head)

{

int key;

node \*New,\*temp; New=get\_node();

printf("\nenter the elements which you want to insert"); scanf("%d",&New->data);

if(head==NULL)

{

head=New;

}

else

{

printf("\enter the element which you want to insert the node"); scanf("%d",&key);

temp=head; do

{

if(temp->data==key)

{

New->next-temp->next; temp->next=New; return;

}

else

temp=temp->next;

}

while(temp!=NULL);

}

}

node \*get\_prev(node \*head,int val)

{

node \*temp,\*prev; int flag; temp=head; if(temp==NULL) return NULL; flag=FALSE; prev=NULL;

while(temp!=NULL && ! flag)

{

if(temp->data!=val)

{

prev=temp; temp=temp->next;

}

else flag=TRUE;

}

if(flag) return prev; else

return NULL;

}

void dele(node \*\*head)

{

node \*temp,\*prev; int key; temp=\*head; if(temp==NULL)

{

printf("\nthe list is empty\n"); return;

}

printf("\nenter the element you want to delete:"); scanf("%d",&key);

temp=search(\*head,key); if(temp!=NULL)

{

prev=get\_prev(\*head,key); if(prev!=NULL)

{

prev->next=temp->next; free(temp);

}

else

{

\*head=temp->next; free(temp);

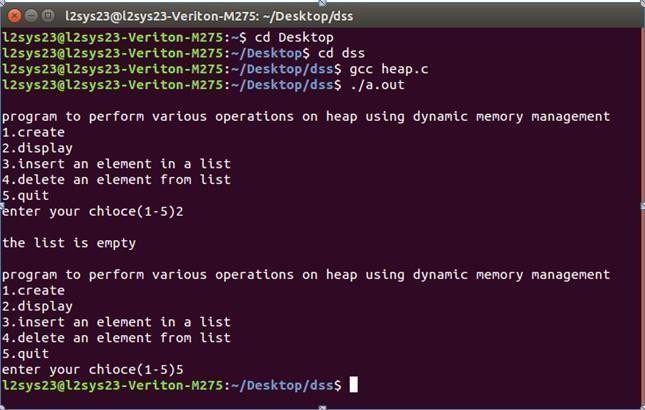
}

printf("\nthe element is deleted\n");

}

}

### OUTPUT:-



**Result:-** The Program Executed successfully.