

**EXP: 01 - LEXICAL ANALYSIS**

* RA1911003010763
* **Date:** 11-01-2022

**Aim:** To identify the keywords, operators & identifiers in a C programs using Lexical Analyzer.

**Algorithm:**

1. Start.
2. Get the input program from the file prog.txt.
3. Read the program line by line and check if each word in a line is a keyword, identifier, constant or an operator.
4. If the word read is an identifier, assign a number to the identifier and make an entry into the symbol table stored in sybol.txt.
5. For each lexeme read, generate a token as follows:
   1. If the lexeme is an identifier, then the token generated is of the form <id, number>.
   2. If the lexeme is an operator, then the token generated is <op, operator>.
   3. If the lexeme is a constant, then the token generated is <const, value>.
   4. If the lexeme is a keyword, then the token is the keyword itself.
6. The stream of tokens generated are displayed in the console output.
7. Stop.

**Source Code:**

#include<stdio.h>

#include<string.h>

#include<stdlib.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], b[30], logical\_op[] = "><", math\_op[]="+-\*/=", number[]=".0123456789", other[]=",;\(){}[]'':";

char kv[32][10],cv[32][10],iv[32][10];

int othc=0,aaa=0,j=0,kvc=0,ovc=0,mvc=0,ivc=0,lvc=0,cvc=0;

char ov[10],mv[10],lv[10];

FILE \*fp;

int mark[1000]={0};

fp=fopen("test","r");

while((ch=fgetc(fp))!=EOF){

for(int i=0;i<12;i++){

if(ch==other[i]){

int aa=ch;

if(mark[aa]!=1){

ov[ovc++]=ch;

mark[aa]=1;

}

}

}

for(int i=0;i<5;i++){

if(ch==math\_op[i]){

int aa=ch;

if(mark[aa]!=1){

mv[mvc++]=ch;

mark[aa]=1;

}

}

}

for(int i=0;i<2;i++){

if(ch==logical\_op[i]){

int aa=ch;

if(mark[aa]!=1){

lv[lvc++]=ch;

mark[aa]=1;

}

}

}

if(ch=='0' || ch=='1' || ch=='2' || ch=='3' || ch=='4' || ch=='5' || ch=='6' || ch=='7' || ch=='8' || ch=='9' || ch=='.' ||ch == ' ' || ch == '\n' || ch == ';'){

if(ch=='0' || ch=='1' || ch=='2' || ch=='3' || ch=='4' || ch=='5' || ch=='6' || ch=='7' || ch=='8' || ch=='9' || ch=='.')

b[aaa++]=ch;

if((ch == ' ' || ch == '\n' || ch == ';') && (aaa != 0)){

b[aaa] = '\0';

aaa = 0;

char arr[30];

strcpy(arr,b);

strcpy(cv[cvc++],arr);

}

}

if(isalpha(ch)){

buffer[j++]=ch;

}

else if((ch==' '||ch=='\n')&&(j!=0)){

buffer[j]='\0';

j=0;

if(isKeyword(buffer)==1){

strcpy(kv[kvc++],buffer);

}

else{

if(mark[buffer[0]-'a']!=1 &&(strcmp(buffer,"printf")!=0 && strcmp(buffer,"includestdioh")!=0)){

if(strcmp(buffer,"main")!=0){

strcpy(iv[ivc++],buffer);

mark[buffer[0]-'a']=1;

}

}

}

}

}

printf("No. of Keywords: %d\n",kvc);

printf("Keywords: ");

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

printf("%s ",kv[i]);

}

printf("\n\n");

printf("No. of Identifiers: %d\n",ivc);

printf("Identifers: ");

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

printf("%s ",iv[i]);

}

printf("\n\n");

printf("No. of Arthematic operators: %d\n",mvc);

printf("Arthematic Operators: ");

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

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

}

printf("\n\n");

printf("No. of Relational operators: %d\n",lvc);

printf("Relational Operators: ");

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

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

}

printf("\n\n");

printf("No. of Constants: %d\n",cvc-1);

printf("Constants: ");

for(int i=1;i<cvc;i++){

printf("%s ",cv[i]);

}

printf("\n\n");

}

**Test Code:**

#include<stdio.h>

int main(){

int a, b, c, d;

a = 10;

b = 20;

if (a + b >= 30){

c = a + b;

}

else{

c = b - a;

}

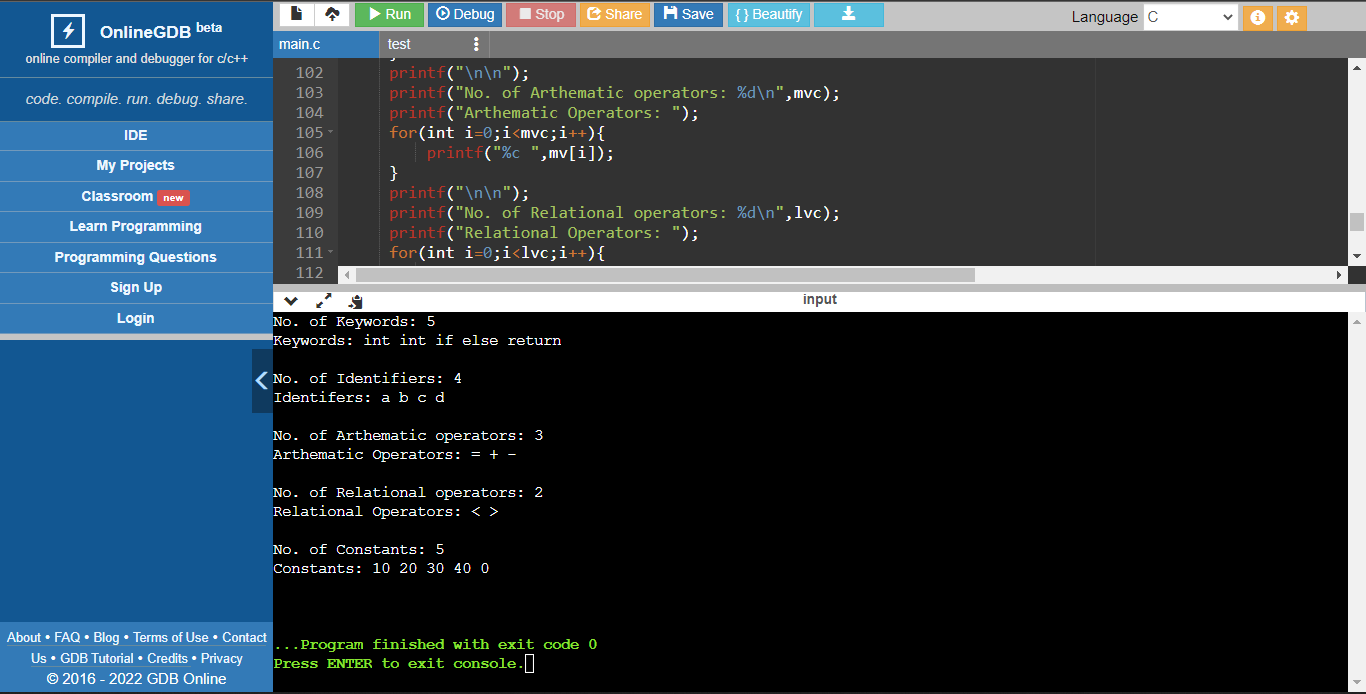
d = 40;

printf ("%d ", c);

return 0;

}

**Output:**

****

**Result:**

Thus, identification the keywords, operators & identifiers in a C programs is done using Lexical Analyzer.

**EXP: 02 - CONVERSION FROM REGULAR EXPRESSION TO NFA**

* RA1911003010763
* **Date:** 19-01-2022

**Aim:** To convert Regular Expressions (RE) to Non-Deterministic Finite Automata (NFA).

**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.

**Source Code:**

#include<stdio.h>

#include<string.h>

int main(){

char reg[20];

int q[20][3],i,j,len,a,b;

for(a=0;a<20;a++){

for(b=0;b<3;b++){

q[a][b]=0;

}

}

printf("Enter the Regular Expression RE: ");

scanf("%s",reg);

len=strlen(reg);

i=0; j=1;

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("Transition function:");

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

if(q[i][0]!=0)

printf("\n q[%d,a]-->%d",i,q[i][0]);

if(q[i][1]!=0)

printf("\n q[%d,b]-->%d",i,q[i][1]);

if(q[i][2]!=0){

if(q[i][2]<10)

printf("\n q[%d,e]-->%d",i,q[i][2]);

else

printf("\n q[%d,e]-->%d & %d",i,q[i][2]/10,q[i][2]%10);

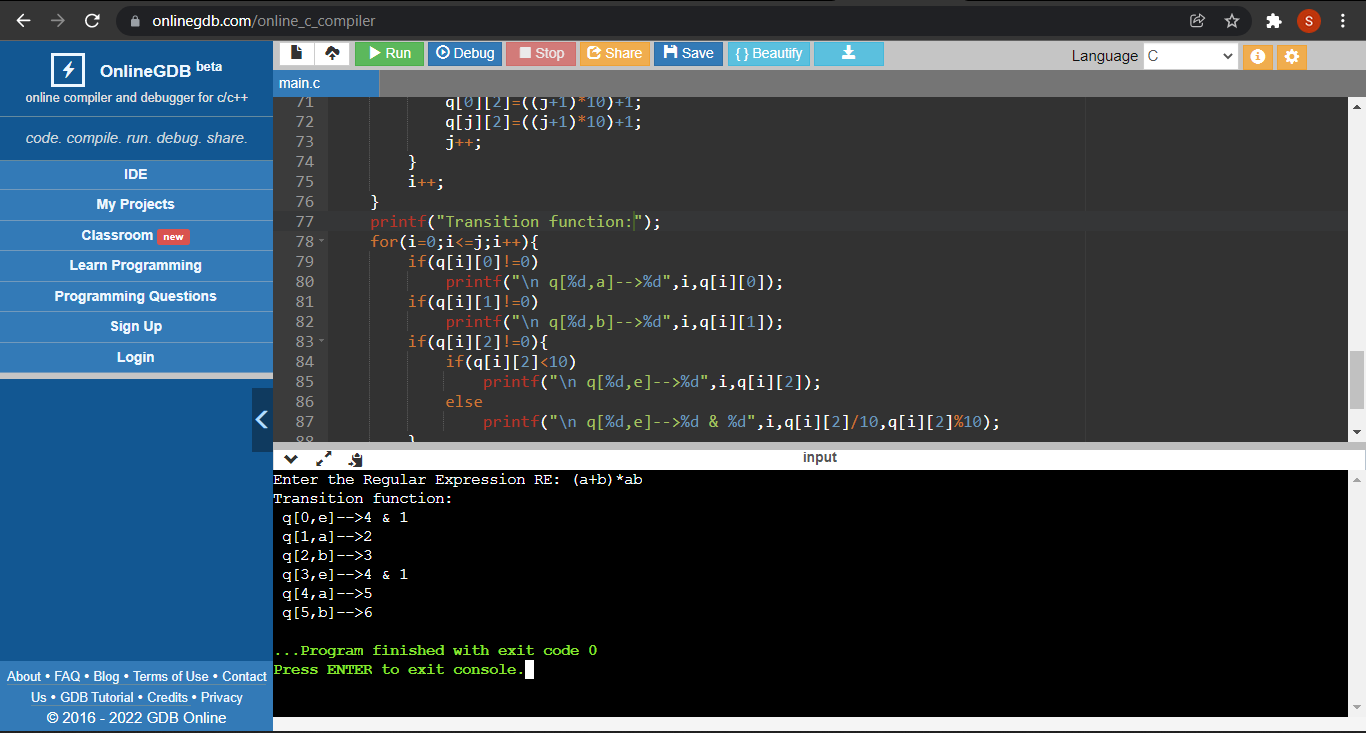
}

}

return 0;

}

**Output:**

****

**Result:**

Thus, conversion of Regular Expressions (RE) to Non-Deterministic Finite Automata (NFA) is done successfully.

**EXP: 03 - CONVERSION FROM NFA TO DFA**

* RA1911003010763
* **Date: 27-01-2022**

**Aim:** To convert Non-Deterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA).

**Algorithm:**

1. Start.
2. Get the input from the user.
3. Set the only state in SDFA to “unmarked”.
4. While SDFA contains an unmarked state do:
   1. Let T be that unmarked state.
   2. for each a in % do S = e-Closure(MoveNFA(T,a)).
   3. if S is not in SDFA already then, add S to SDFA (as an “unmarked” state).
   4. Set MoveDFA(T,a) to S.
5. For each S in SDFA if any s & S is a final state in the NFA then, mark S as a final state in the DFA.
6. Print the result.
7. Stop the program,

**Source Code:**

#include <stdio.h>

int main(){

int nfa[5][2];

nfa[1][1]=12;

nfa[1][2]=1;

nfa[2][1]=0;

nfa[2][2]=3;

nfa[3][1]=0;

nfa[3][2]=4;

nfa[4][1]=0;

nfa[4][2]=0;

int dfa[10][2];

int dstate[10];

int i=1,n,j,k,flag=0,m,q,r;

dstate[i++]=1;

n=i;

dfa[1][1]=nfa[1][1];

dfa[1][2]=nfa[1][2];

printf("\nf(%d,a)=%d",dstate[1],dfa[1][1]);

printf("\nf(%d,b)=%d",dstate[1],dfa[1][2]);

for(j=1;j<n;j++){

if(dfa[1][1]!=dstate[j])

flag++;

}

if(flag==n-1){

dstate[i++]=dfa[1][1];

n++;

}

flag=0;

for(j=1;j<n;j++){

if(dfa[1][2]!=dstate[j])

flag++;

}

if(flag==n-1) {

dstate[i++]=dfa[1][2];

n++;

}

k=2;

while(dstate[k]!=0){

m=dstate[k];

if(m>10){

q=m/10;

r=m%10;

}

if(nfa[r][1]!=0)

dfa[k][1]=nfa[q][1]\*10+nfa[r][1];

else

dfa[k][1]=nfa[q][1];

if(nfa[r][2]!=0)

dfa[k][2]=nfa[q][2]\*10+nfa[r][2];

else

dfa[k][2]=nfa[q][2];

printf("\nf(%d,a)=%d",dstate[k],dfa[k][1]);

printf("\nf(%d,b)=%d",dstate[k],dfa[k][2]);

flag=0;

for(j=1;j<n;j++){

if(dfa[k][1]!=dstate[j])

flag++;

}

if(flag==n-1){

dstate[i++]=dfa[k][1];

n++;

}

flag=0;

for(j=1;j<n;j++){

if(dfa[k][2]!=dstate[j])

flag++;

}

if(flag==n-1){

dstate[i++]=dfa[k][2];

n++;

}

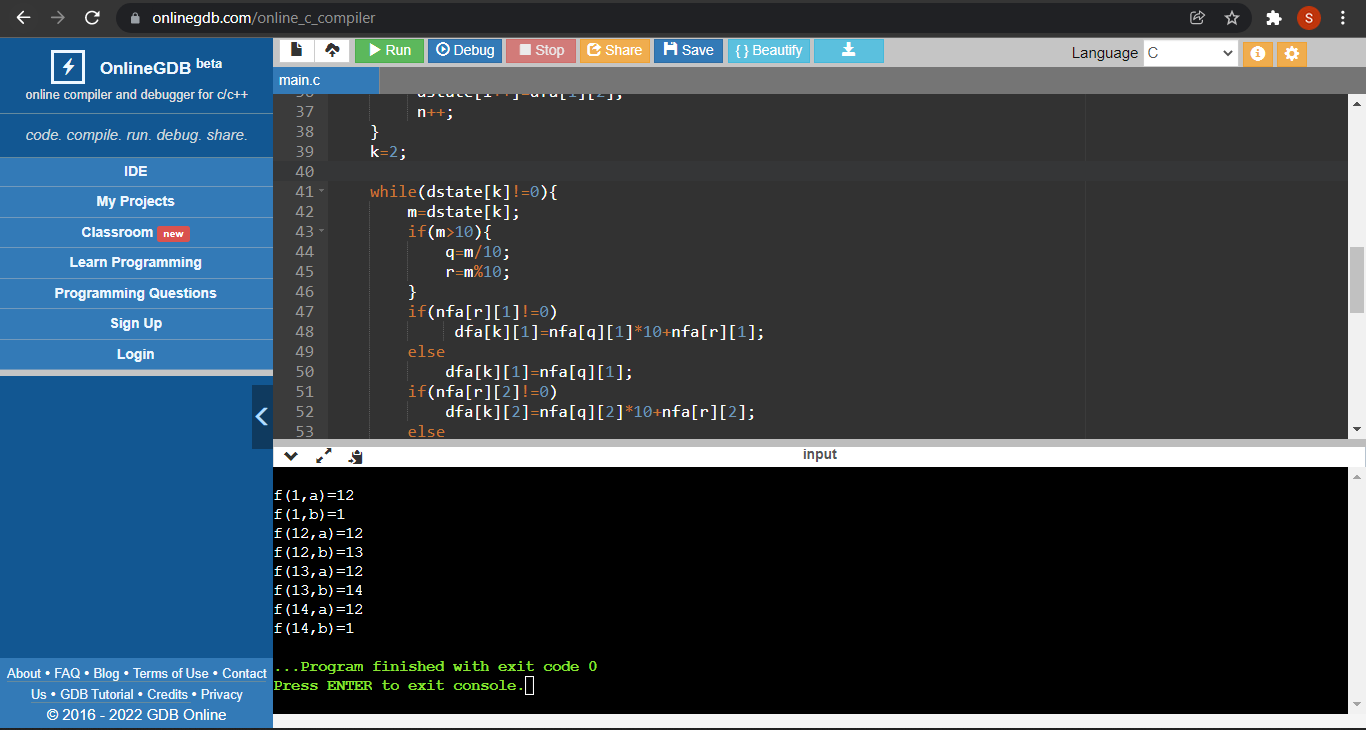
k++;

}

return 0;

}

**Output:**

****

**Result:**

Thus, conversion of Non-Deterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA) is done successfully.

**EXP: 04 - a) ELIMINATING LEFT RECURSION**

* RA1911003010763
* **Date: 10-02-2022**

**Aim:** To eliminate Left Recursion from the given grammar.

**Algorithm:**

1. Start the program.
2. Initialize the arrays for taking input from the user.
3. Prompt the user to input the no. of non-terminals having left recursion and no. of productions for these non-terminals.
4. Prompt the user to input the production for non-terminals.
5. Eliminate left recursion using the following rules:

A->Aα|β

Then replace it by

A-> βA’

A’-> αA’

A’-> Ɛ

1. After eliminating the left recursion by applying these rules, display the productions without left recursion.
2. Stop.

**Source Code:**

#include<stdio.h>

#include<string.h>

#define SIZE 10

int main (){

char non\_terminal;

char beta,alpha;

int num;

char production[10][SIZE];

int index=3;

printf("Enter Number of Production: ");

scanf("%d",&num);

printf("Enter the Production in the form A->Aα|β:\n");

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

printf("Production %d: ", i+1);

scanf("%s",production[i]);

}

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

printf("\nEntered Production no. %d: %s",i+1,production[i]);

non\_terminal=production[i][0];

printf("\nChecking Left Recursivenss...\n");

if(non\_terminal==production[i][index]) {

alpha=production[i][index+1];

printf("Production %s is a Left Recursive Production.\n", production[i]);

while(production[i][index]!=0 && production[i][index]!='|')

index++;

if(production[i][index]!=0) {

beta=production[i][index+1];

printf("Productions after Eliminating the Left Recursion:\n");

printf("%c->%c%c\'",non\_terminal,beta,non\_terminal);

printf("\n%c\'->%c%c\'|E\n",non\_terminal,alpha,non\_terminal);

}

else

printf("Entered Production %s cannot be Reduced\n", production[i]);

}

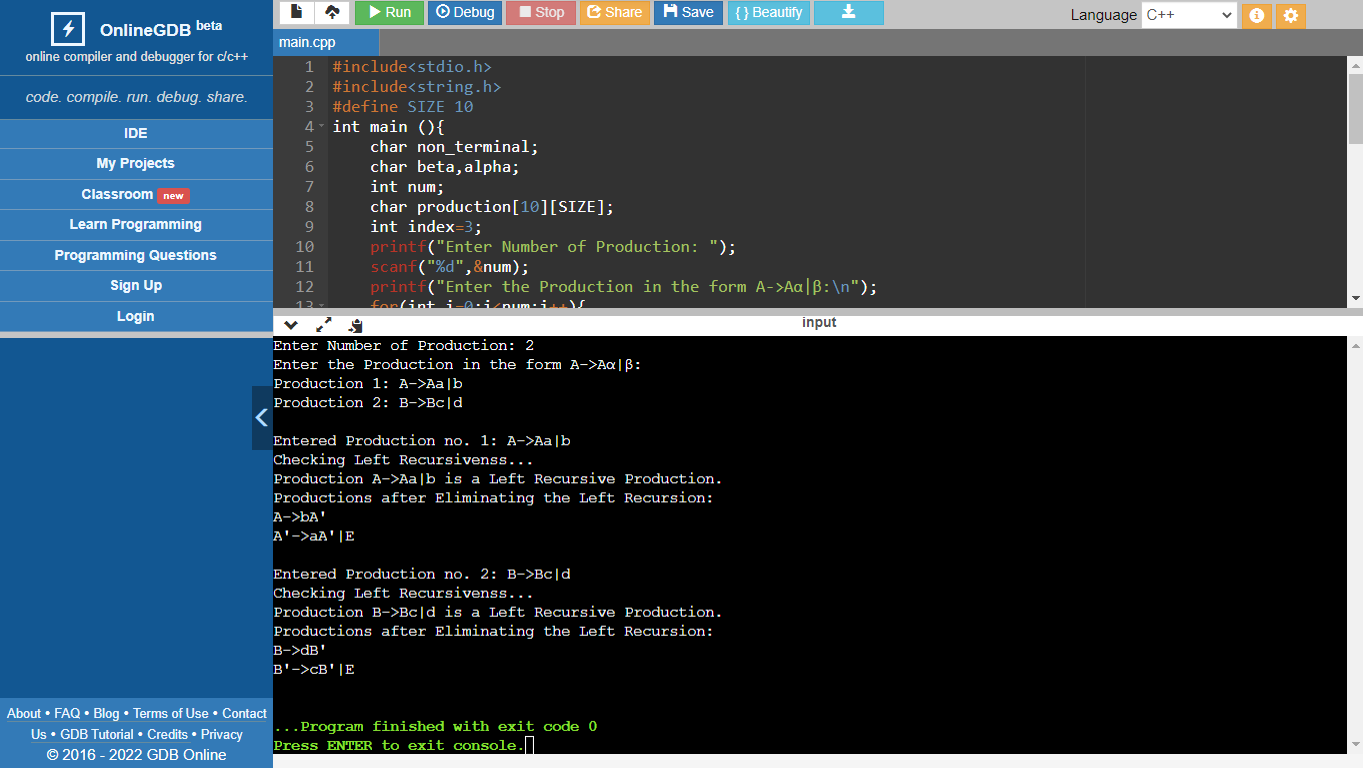
else

printf("Production %s is not a Left Recursive Production\n", production[i]);

index=3;

} }

**Output:**

****

**Result:**

Thus, the left recursion is successfully eliminated from the given grammar.

**EXP: 04 - b) ELIMINATING LEFT FACTORING**

* RA1911003010763
* **Date: 10-02-2022**

**Aim:** To eliminate Left factoring from the given grammar.

**Algorithm:**

1. Start
2. Ask the user to enter the set of productions
3. Check for common symbols in the given set of productions by comparing with:

A->aB1 | aB2

1. If found, replace the particular productions with:

A->aA’

A’->B1 | B2 | ɛ

1. Display the output
2. Exit

**Source Code:**

#include<stdio.h>

#include<string.h>

#define SIZE 20

int main(){

char pro[SIZE],t1[SIZE],t2[SIZE],pro1[SIZE],prof[SIZE],nt;

int i,j=0,k=0,l=0,pos;

printf(" Enter the Non-Terminal: ");

scanf("%c", &nt);

printf(" Enter Production: %c->", nt);

scanf("%s", pro);

for(i=0;pro[i]!='|';i++,j++)

t1[j]=pro[i];

t1[j]='\0';

for(j=++i,i=0;pro[j]!='\0';j++,i++)

t2[i]=pro[j];

t2[i]='\0';

for(i=0;i<strlen(t1)||i<strlen(t2);i++){

if(t1[i]==t2[i]){

pro1[k]=t1[i];

k++;

pos=i+1;

} }

for(i=pos,j=0;t1[i]!='\0';i++,j++)

prof[j]=t1[i];

prof[j++]='|';

for(i=pos;t2[i]!='\0';i++,j++)

prof[j]=t2[i];

pro1[k]='X';

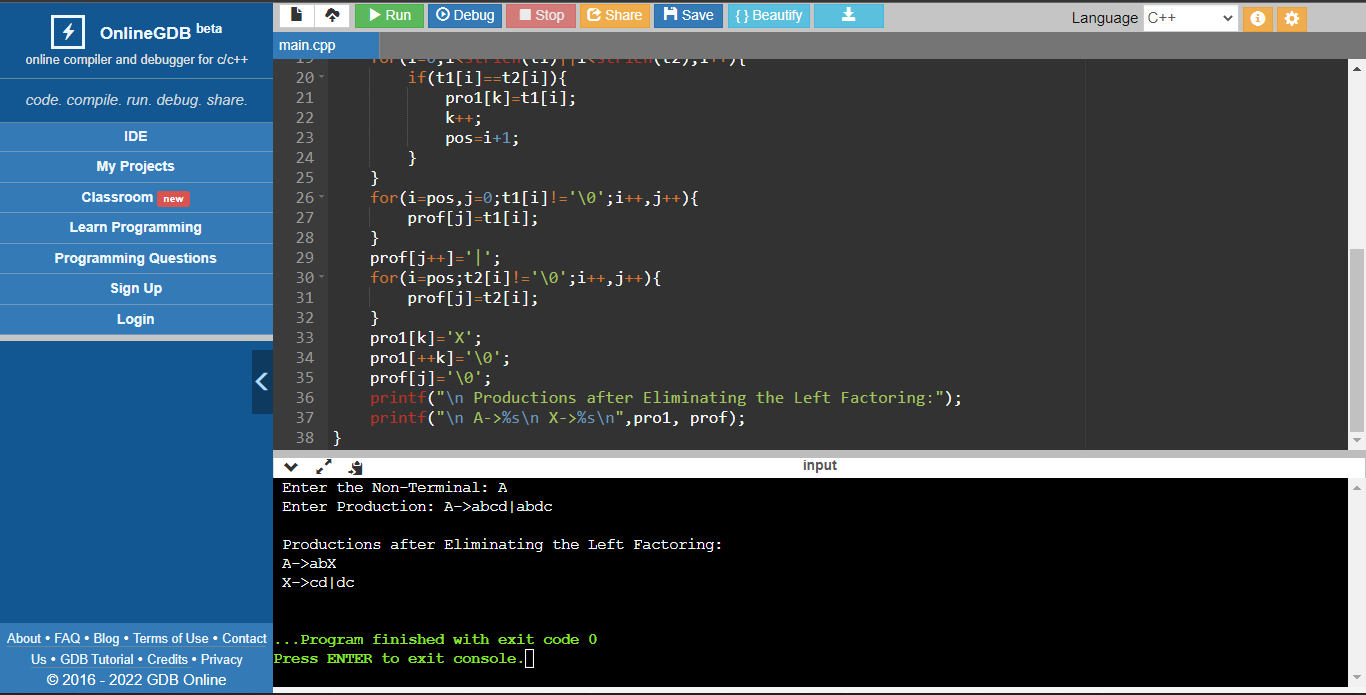
pro1[++k]='\0';

prof[j]='\0';

printf("\n Productions after Eliminating the Left Factoring:");

printf("\n A->%s\n X->%s\n",pro1, prof); }

**Output:**

****

**Result:**

Thus, the left factoring is successfully eliminated from the given grammar.

**EXP: 05 - a) IMPLEMENTATION OF FIRST**

* RA1911003010763
* **Date: 17-02-2022**

**Aim:** To write a program that can implement & find FIRST set for the given grammar.

**Algorithm:**

1. If X is a terminal then FIRST(X) = {X}.

Example: F -> I | id.

1. We can write it as FIRST(F) -> { ( , id }.
2. If X is a non-terminal like E -> T then to get FIRST substitute T with other productions until you get a terminal as the first symbol.
3. If X -> ε then add ε to FIRST(X).

**Source Code:**

**#include <ctype.h>**

**#include <string.h>**

**#include <stdio.h>**

**char pro[10][10],first[15];**

**int size,k=0;**

**void funcFirst(char nt){**

**int j;**

**if(!(isupper(nt))){**

**first[k++]=nt;**

**}**

**else**

**for(j=0;j<size;j++){**

**if(pro[j][0]==nt){**

**if(pro[j][3]=='$'){**

**first[k++]='$';**

**}**

**else if(islower(pro[j][3])) {**

**first[k++]=pro[j][3];**

**}**

**else{**

**funcFirst(pro[j][3]);**

**}**

**}**

**}**

**}**

**int main(){**

**int i,j=0;**

**printf("Enter the no. of Productions: ");**

**scanf("%d",&size);**

**printf("Enter the Productions: \n");**

**for(i=0;i<size;i++){**

**scanf("%s",pro[i]);**

**}**

**printf("\nEntered Grammar:");**

**for(i=0;i<size;i++) {**

**printf("\n%s",pro[i]);**

**}**

**printf("\n");**

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

**funcFirst(pro[i][0]);**

**printf("\nFIRST of %c: [ ", pro[i][0]);**

**for(j;j<strlen(first);j++) {**

**printf("%c",first[j]);**

**}**

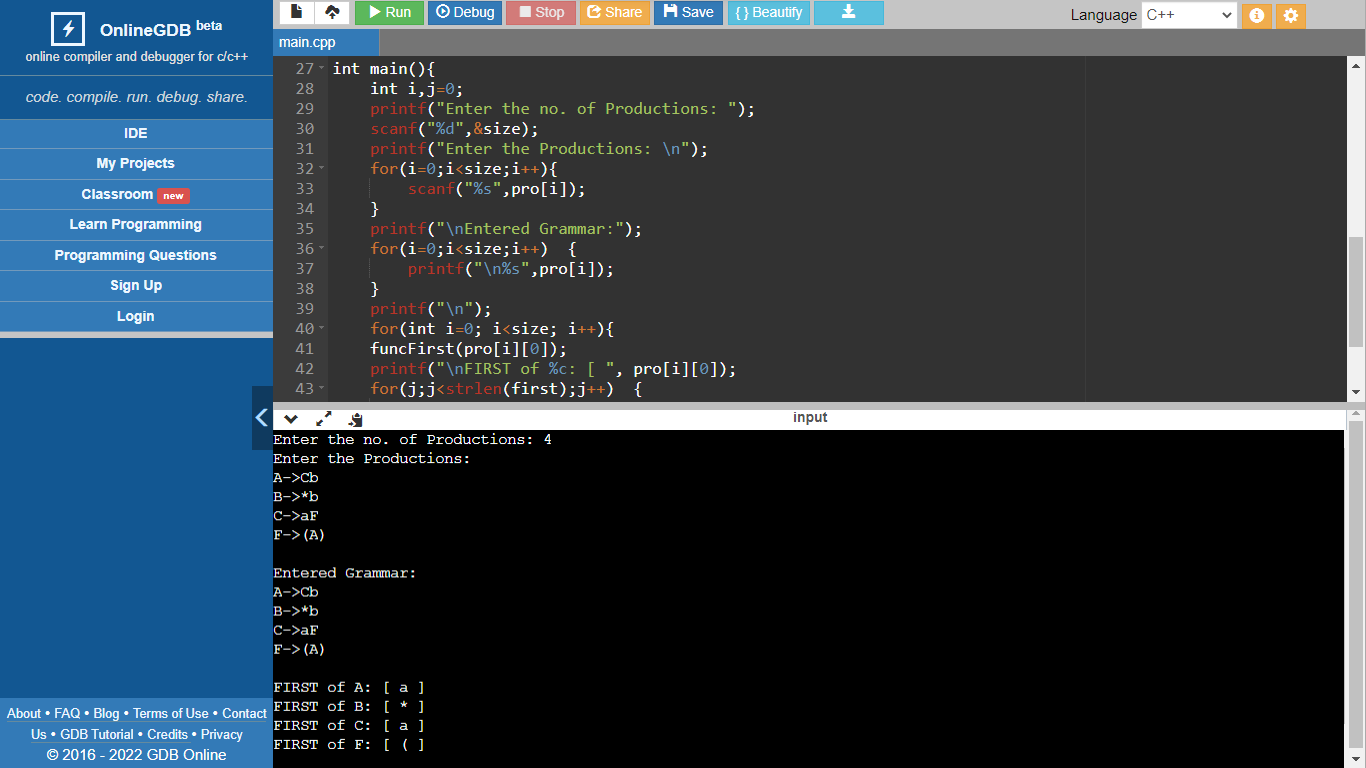
**j = strlen(first);**

**printf(" ]");**

**}**

**}**

**Output:**

****

**Result:**

Thus, the FIRST set is successfully found from the given grammar.

**EXP: 05 - b) IMPLEMENTATION OF FOLLOW**

* RA1911003010763
* **Date: 17-02-2022**

**Aim:** To write a program that can implement & find FOLLOW set for the given grammar.

**Algorithm:**

1. Always check the right side of the productions for a non-terminal, whose FOLLOW set is being found. (never see the left side).
2. If that non-terminal (S,A,B…) is followed by any terminal (a,b…,\*,+,(,)…) , then add that terminal into the FOLLOW set.
3. If that non-terminal is followed by any other non-terminal then add FIRST of other nonterminal into the FOLLOW set.
4. Hence, following these steps will help in finding the FOLLOW set from the grammar.

**Source Code:**

**#include <ctype.h>**

**#include <string.h>**

**#include <stdio.h>**

**char pro[10][10],follow[30];**

**int size,k=0;**

**void funcFirst(char nt)**

**{**

**if(!(isupper(nt)))**

**{**

**follow[k++]=nt;**

**}**

**else**

**for(int j=0;j<size;j++)**

**{**

**if(pro[j][0]==nt)**

**{**

**if(islower(pro[j][3]))**

**{**

**follow[k++]=pro[j][3];**

**}**

**else**

**{**

**funcFirst(pro[j][3]);**

**}**

**}**

**}**

**}**

**void funcFollow(char nt)**

**{**

**if(pro[0][0]==nt)**

**follow[k++]='$';**

**for(int i=0;i<size;i++)**

**{**

**for(int j=3;j<strlen(pro[i]); j++)**

**{**

**if(pro[i][j]==nt)**

**{**

**if(pro[i][j+1]!='\0')**

**funcFirst(pro[i][j+1]);**

**if(pro[i][j+1]=='\0' && nt!=pro[i][0])**

**funcFollow(pro[i][0]);**

**}**

**}**

**}**

**}**

**int main()**

**{**

**int i,j=0;**

**printf("Enter the no. of Productions: ");**

**scanf("%d",&size);**

**printf("Enter the Productions: \n");**

**for(i=0;i<size;i++)**

**scanf("%s",pro[i]);**

**printf("\nEntered Grammar:");**

**for(i=0;i<size;i++)**

**printf("\n%s",pro[i]);**

**printf("\n");**

**for(int i=0; i<size; i++)**

**{**

**funcFollow(pro[i][0]);**

**printf("\nFOLLOW of %c: [ ", pro[i][0]);**

**for(j;j<strlen(follow);j++)**

**printf("%c ",follow[j]);**

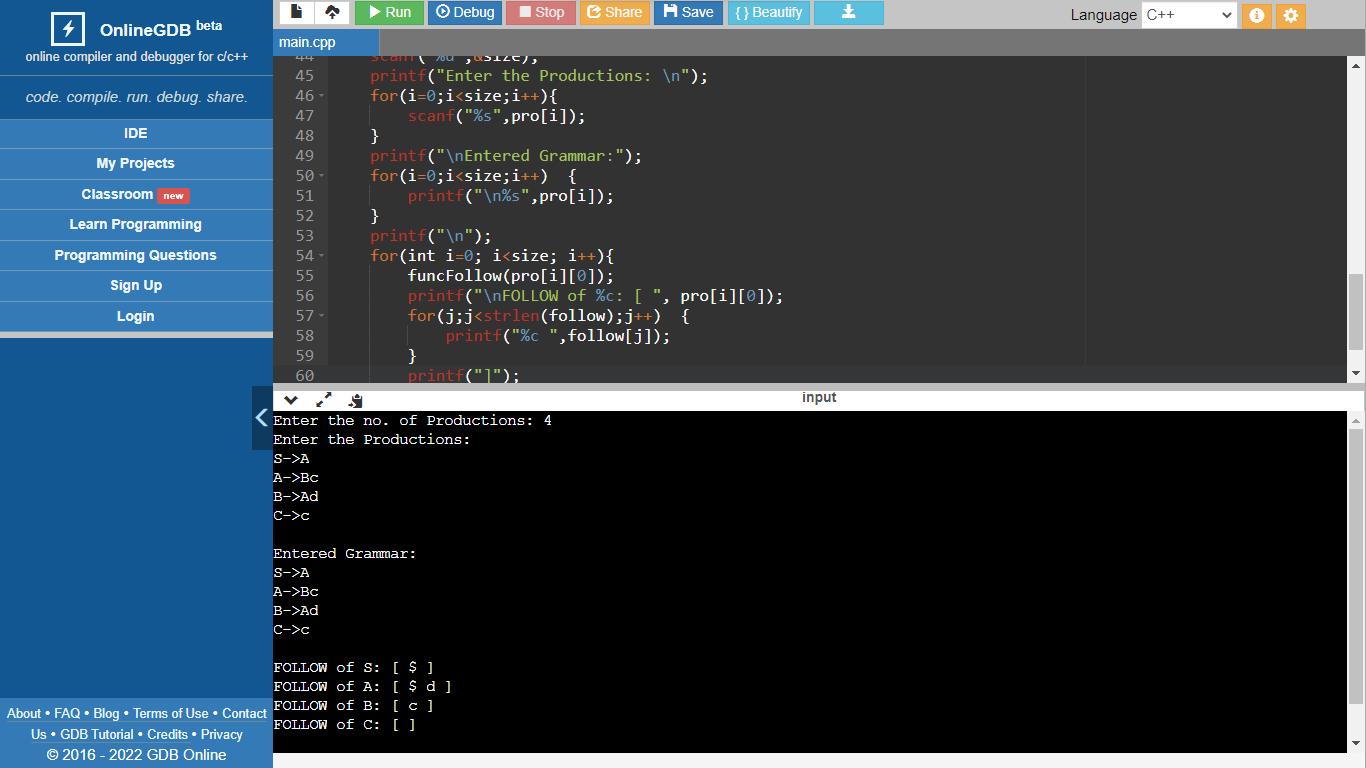
**printf("]");**

**j = strlen(follow);**

**}**

**}**

**Output:**

****

**Result:**

Thus, the FOLLOW set is successfully found from the given grammar.

**EXP: 06 - IMPLEMENTATION OF PREDICTIVE PARSING**

* RA1911003010763
* **Date: 24-02-2022**

**Aim:** To write a program that can implement the Predictive Parsing for the given grammar.

**Algorithm:**

1. If X is a non-terminal like E -> T then to get FIRST substitute T with other productions until you get a terminal as the first symbol.
2. If that non-terminal (S,A,B…) is followed by any terminal (a,b…,\*,+,(,)…) , then add that terminal into the FOLLOW set.
3. If that non-terminal is followed by any other non-terminal then add FIRST of other nonterminal into the FOLLOW set.
4. In this way find the first & follow sets for the given grammar.
5. Then, for each production A –> α. (A tends to alpha)
   1. Find First(α) and for each terminal in First(α), make entry A –> α in the table.
   2. If First(α) contains ε (epsilon) as terminal than, find the Follow(A) and for each terminal in Follow(A), make entry A –> α in the table.
   3. If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> α in the table for the $.

**Source Code:**

**#include<stdio.h>**

**#include<conio.h>**

**#include<string.h>**

**int main()**

**{**

**char fin[10][20],st[10][20],ft[20][20],fol[20][20];**

**int a=0,e,i,t,b,c,n,k,l=0,j,s,m,p;**

**printf("Enter the no. of Non-Terminals:\n");**

**scanf("%d",&n);**

**printf("\n---Enter @ at the place of epsilon---\n");**

**printf("\nEnter the Productions in the Grammar:\n");**

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

**scanf("%s",st[i]);**

**printf("\n-----------------------------------------------\n\n");**

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

**fol[i][0]='\0';**

**for(s=0;s<n;s++)**

**{**

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

**{**

**j=3;**

**l=0;**

**a=0;**

**l1:if(!((st[i][j]>64)&&(st[i][j]<91)))**

**{**

**for(m=0;m<l;m++)**

**{**

**if(ft[i][m]==st[i][j])**

**goto s1;**

**}**

**ft[i][l]=st[i][j];**

**l=l+1;**

**s1:j=j+1;**

**}**

**else**

**{**

**if(s>0)**

**{**

**while(st[i][j]!=st[a][0])**

**{**

**a++;**

**}**

**b=0;**

**while(ft[a][b]!='\0')**

**{**

**for(m=0;m<l;m++)**

**{**

**if(ft[i][m]==ft[a][b])**

**goto s2;**

**}**

**ft[i][l]=ft[a][b];**

**l=l+1;**

**s2:b=b+1;**

**}**

**}**

**}**

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

**{**

**if(st[i][j]=='|')**

**{**

**j=j+1;**

**goto l1;**

**}**

**j=j+1;**

**}**

**ft[i][l]='\0';**

**}**

**}**

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

**printf("First(%c) = { %s }\n\n",st[i][0],ft[i]);**

**printf("-----------------------------------------------\n\n");**

**fol[0][0]='$';**

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

**{**

**k=0;**

**j=3;**

**if(i==0)**

**l=1;**

**else**

**l=0;**

**k1:while((st[i][0]!=st[k][j])&&(k<n))**

**{**

**if(st[k][j]=='\0')**

**{**

**k++;**

**j=2;**

**}**

**j++;**

**}**

**j=j+1;**

**if(st[i][0]==st[k][j-1])**

**{**

**if((st[k][j]!='|')&&(st[k][j]!='\0'))**

**{**

**a=0;**

**if(!((st[k][j]>64)&&(st[k][j]<91)))**

**{**

**for(m=0;m<l;m++)**

**{**

**if(fol[i][m]==st[k][j])**

**goto q3;**

**}**

**fol[i][l]=st[k][j];**

**l++;**

**q3:;**

**}**

**else**

**{**

**while(st[k][j]!=st[a][0])**

**{**

**a++;**

**}**

**p=0;**

**while(ft[a][p]!='\0')**

**{**

**if(ft[a][p]!='@')**

**{**

**for(m=0;m<l;m++)**

**{**

**if(fol[i][m]==ft[a][p])**

**goto q2;**

**}**

**fol[i][l]=ft[a][p];**

**l=l+1;**

**}**

**else**

**e=1;**

**q2:p++;**

**}**

**if(e==1)**

**{**

**e=0;**

**goto a1;**

**}**

**}**

**}**

**else**

**{**

**a1:c=0;**

**a=0;**

**while(st[k][0]!=st[a][0])**

**a++;**

**while((fol[a][c]!='\0')&&(st[a][0]!=st[i][0]))**

**{**

**for(m=0;m<l;m++)**

**{**

**if(fol[i][m]==fol[a][c])**

**goto q1;**

**}**

**fol[i][l]=fol[a][c];**

**l++;**

**q1:c++;**

**}**

**}**

**goto k1;**

**}**

**fol[i][l]='\0';**

**}**

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

**printf("Follow(%c) = { %s }\n\n",st[i][0],fol[i]);**

**printf("-----------------------------------------------\n");**

**printf("\nThe LL(1) Parsing for the Above Grammer");**

**printf("\n^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n\n");**

**s=0;**

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

**{**

**j=3;**

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

**{**

**if((st[i][j-1]=='|')||(j==3))**

**{**

**for(p=0;p<=2;p++)**

**{**

**fin[s][p]=st[i][p];**

**}**

**t=j;**

**for(p=3;((st[i][j]!='|')&&(st[i][j]!='\0'));p++)**

**{**

**fin[s][p]=st[i][j];**

**j++;**

**}**

**fin[s][p]='\0';**

**if(st[i][k]=='@')**

**{**

**b=0;**

**a=0;**

**while(st[a][0]!=st[i][0])**

**{**

**a++;**

**}**

**while(fol[a][b]!='\0')**

**{**

**printf("M[%c,%c]=%s\n\n",st[i][0],fol[a][b],fin[s]);**

**b++;**

**}**

**}**

**else if(!((st[i][t]>64)&&(st[i][t]<91)))**

**printf("M[%c,%c]=%s\n\n",st[i][0],st[i][t],fin[s]);**

**else**

**{**

**b=0;**

**a=0;**

**while(st[a][0]!=st[i][3])**

**a++;**

**while(ft[a][b]!='\0')**

**{**

**printf("M[%c,%c]=%s\n\n",st[i][0],ft[a][b],fin[s]);**

**b++;**

**}**

**}**

**s++;**

**}**

**if(st[i][j]=='|')**

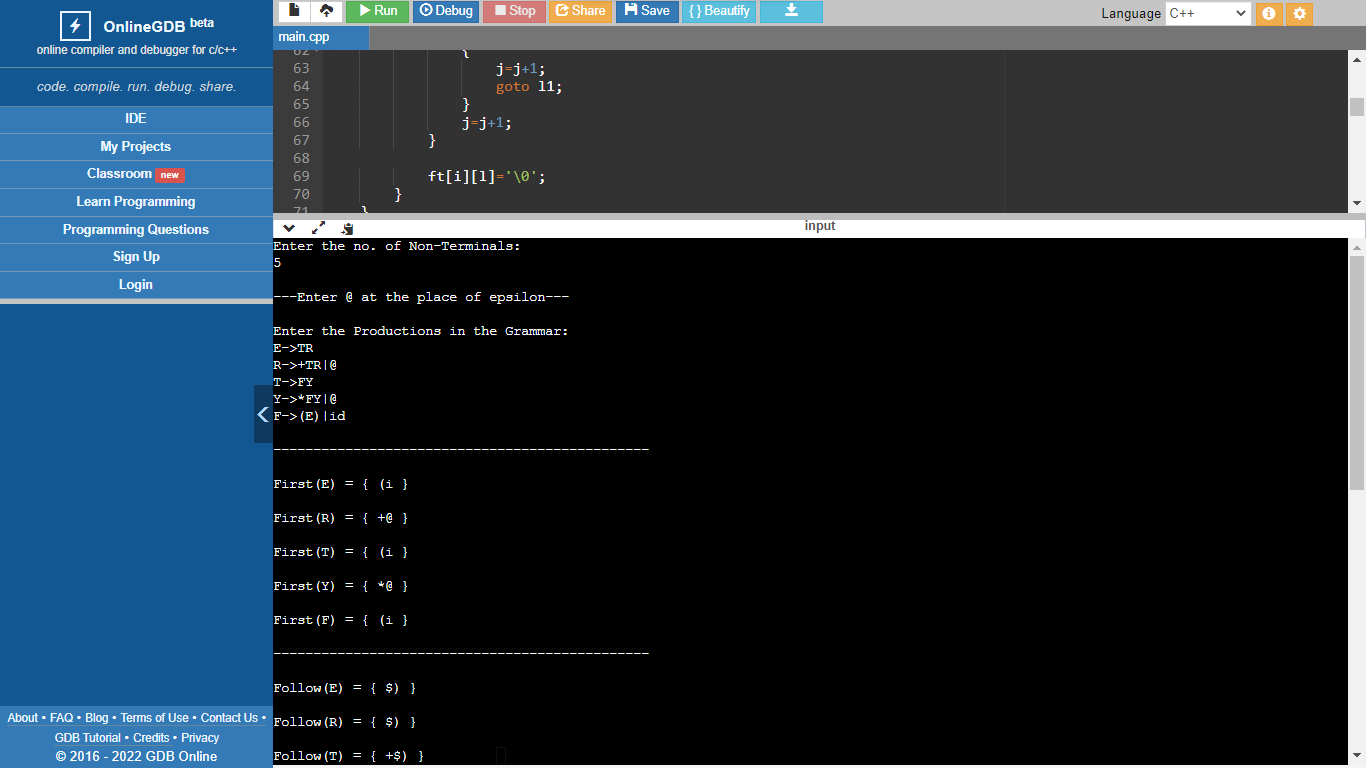
**j++;**

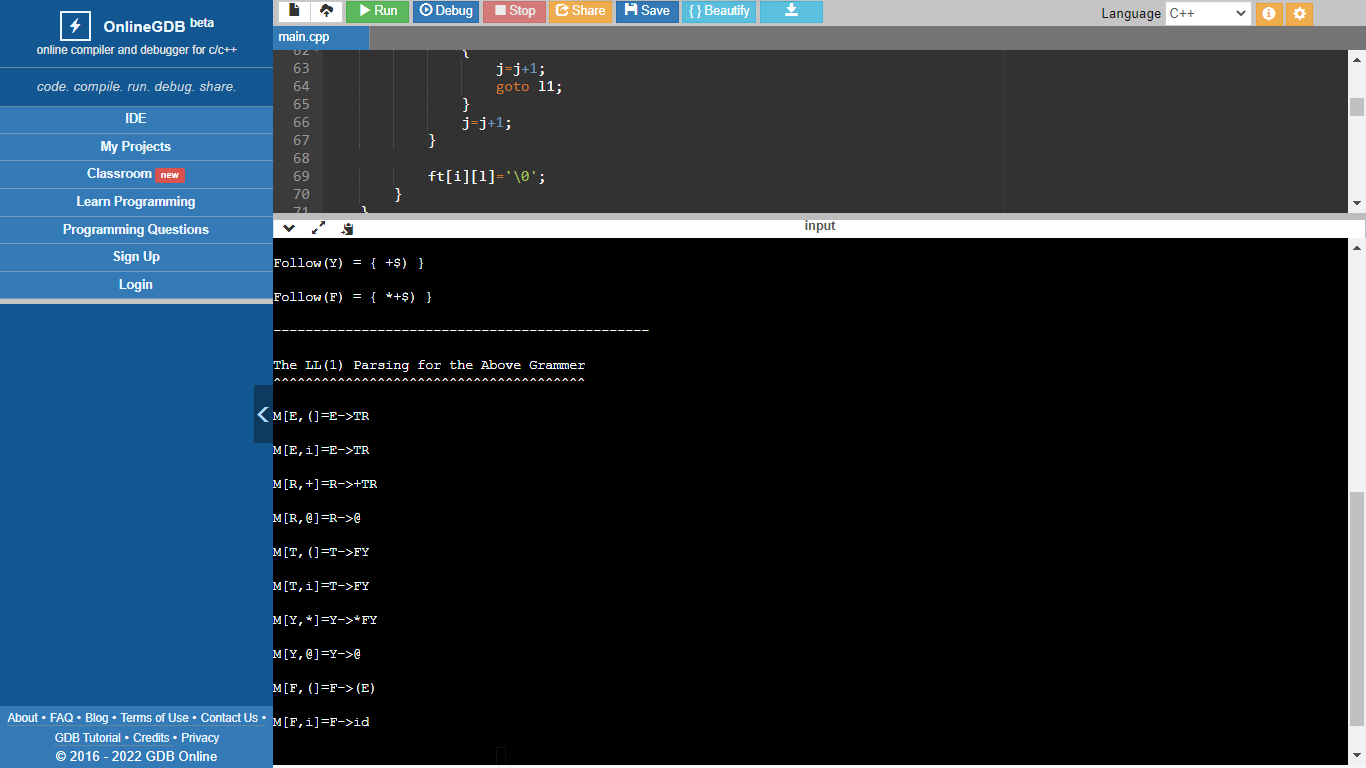
**}**

**}**

**}**

**Output:**

****

****

**Result:**

Thus, the Predictive Parsing is successfully implemented for the given grammar.

**EXP: 07 - IMPLEMENTATION OF SHIFT REDUCE PARSING**

* RA1911003010763
* **Date: 10-03-2022**

**Aim:** To write a program that can implement the Shift Reduce Parsing for the given grammar.

**Algorithm:**

1. First take an input string which needs to tested.
2. Next create a stack for storing and accessing the production rules.
3. Now add each of the input symbol from the input string into the stack created.
4. If handle appears on top of the stack then, its reduction is done by using appropriate production rule i.e. the handle is reduced by the LHS of the rule.
5. After the input string is completely transferred to the stack and all the actions are done then If only the start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept.
6. When accepted action is obtained, it is means successful parsing is done.
7. If there is a situation in which the parser can neither perform shift action nor reduce action and not even accept action then the string is rejected.

**Source Code:**

**#include<stdio.h>**

**#include<stdlib.h>**

**#include<string.h>**

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

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

**void check(){**

**strcpy(ac,"REDUCE TO S -> ");**

**for(z = 0; z < c; z++){**

**if(stk[z] == 'i'){**

**printf("%si", ac);**

**stk[z] = 'S';**

**stk[z + 1] = '\0';**

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

**}**

**}**

**for(z = 0; z < c - 2; z++){**

**if(stk[z] == 'S' && stk[z + 1] == '+' &&**

**stk[z + 2] == 'S'){**

**printf("%sS+S", ac);**

**stk[z] = 'S';**

**stk[z + 1] = '\0';**

**stk[z + 2] = '\0';**

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

**i = i - 2;**

**}**

**}**

**for(z=0; z<c-2; z++){**

**if(stk[z] == 'S' && stk[z + 1] == '\*' &&**

**stk[z + 2] == 'S'){**

**printf("%sS\*S", ac);**

**stk[z]='S';**

**stk[z + 1]='\0';**

**stk[z + 1]='\0';**

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

**i = i - 2;**

**}**

**}**

**return ;**

**}**

**int main(){**

**printf("GRAMMAR is -\nS->S+S \nS->S\*S \nS->i\n");**

**printf("Enter input string: ");**

**scanf("%s", a);**

**c=strlen(a);**

**strcpy(act,"SHIFT");**

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

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

**for(i = 0; j < c; i++, j++){**

**printf("%s", act);**

**stk[i] = a[j];**

**stk[i + 1] = '\0';**

**a[j]=' ';**

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

**check();**

**}**

**check();**

**if(stk[0] == 'S' && stk[1] == '\0'){**

**printf("Accept\n");**

**printf("\nInput String is Accepted");**

**}**

**else{**

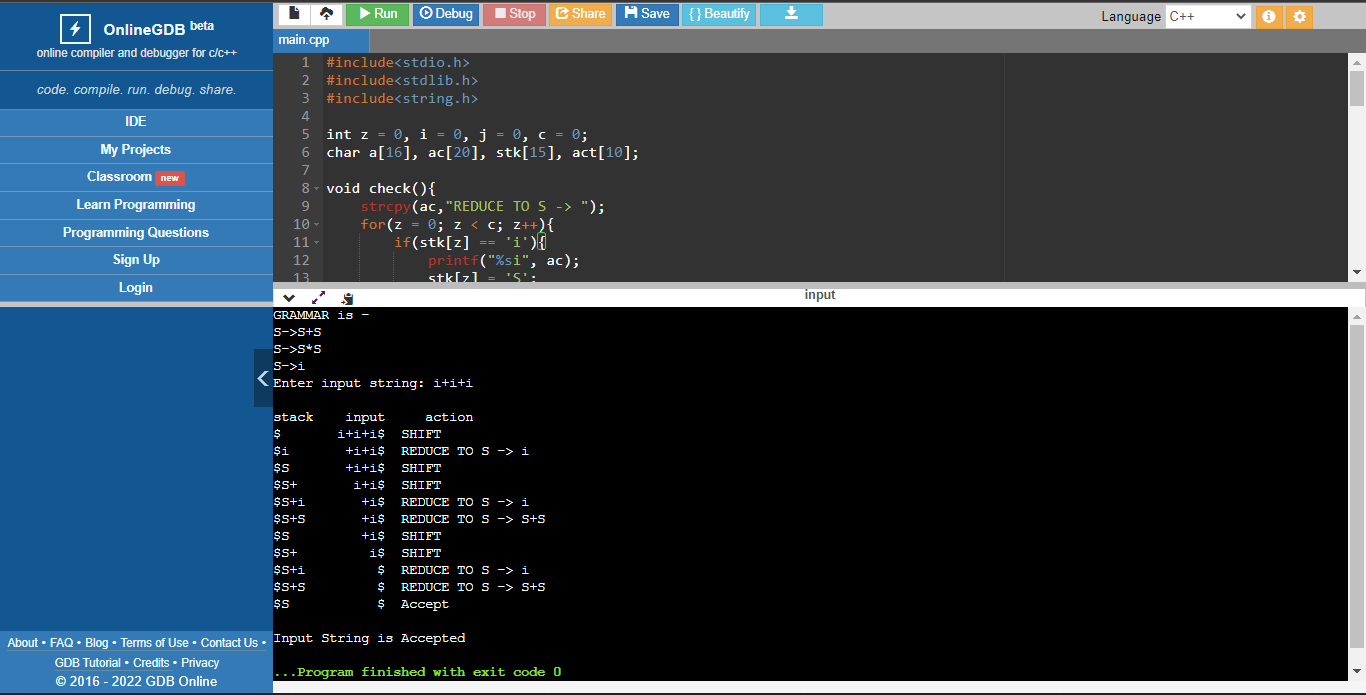
**printf("Reject\n");**

**printf("\nInput String is Rejected");**

**}**

**}**

**Output:**

****

**Result:**

Thus, the Shift Reduce Parsing is successfully implemented for the given grammar.

**EXP: 08 - COMPUTATIONN OF LEADING AND TRAILING**

* RA1911003010763
* **Date: 17-03-2022**

**Aim:** To write a program that can compute the Leading and Trailing for the given grammar.

**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

**Source Code:**

**#include<iostream>**

**#include<string.h>**

**#include<conio.h>**

**using namespace std;**

**int nt,t,top=0;**

**char s[50],NT[10],T[10],st[50],l[10][10],tr[50][50];**

**int searchnt(char a){**

**int count=-1,i;**

**for(i=0;i<nt;i++){**

**if(NT[i]==a)**

**return i;**

**}**

**return count;**

**}**

**int searchter(char a){**

**int count=-1,i;**

**for(i=0;i<t;i++){**

**if(T[i]==a)**

**return i;**

**}**

**return count;**

**}**

**void push(char a){**

**s[top]=a;**

**top++;**

**}**

**char pop(){**

**top--;**

**return s[top];**

**}**

**void installl(int a,int b){**

**if(l[a][b]=='f'){**

**l[a][b]='t';**

**push(T[b]);**

**push(NT[a]);**

**}**

**}**

**void installt(int a,int b){**

**if(tr[a][b]=='f'){**

**tr[a][b]='t';**

**push(T[b]);**

**push(NT[a]);**

**}**

**}**

**int main(){**

**int i,s,k,j,n;**

**char pr[30][30],b,c;**

**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";**

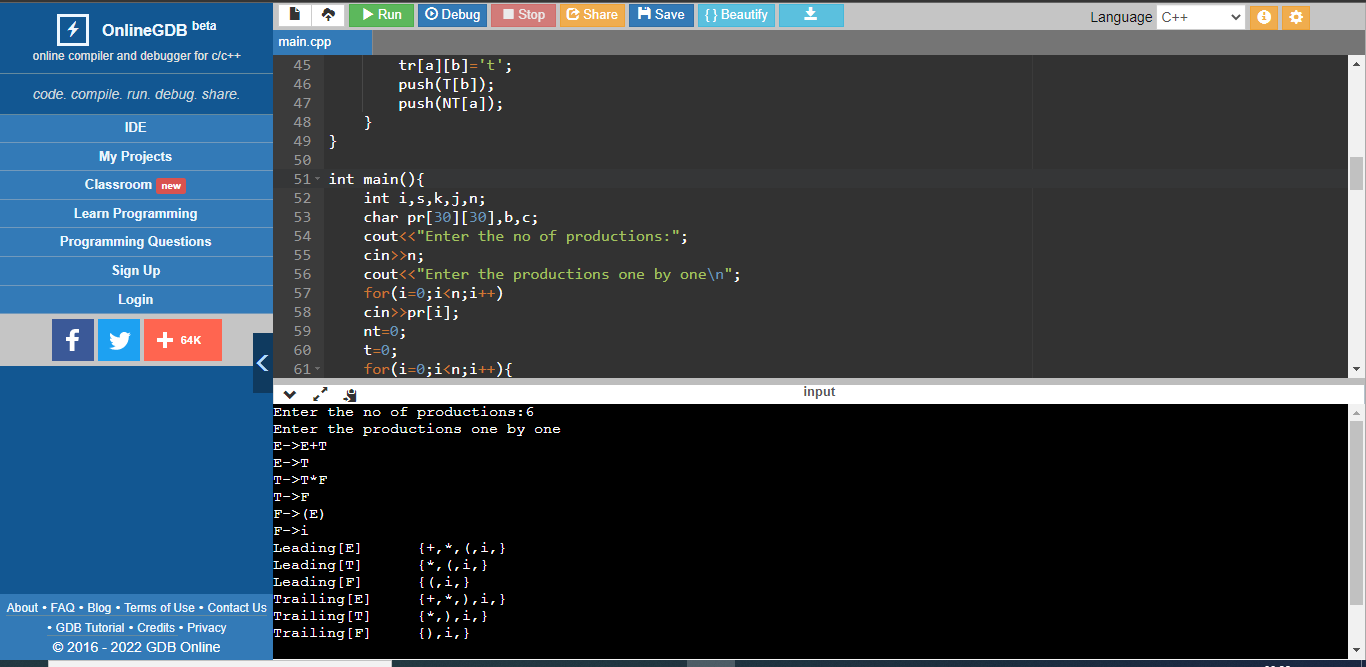
**}**

**getch();**

**return 0;**

**}**

**Output:**



**Result:**

Thus, the Leading and Trailing is successfully computed for the given grammar.

**EXP: 09 - IMPLEMENTATION OF LR(0) PARSING**

* RA1911003010763
* **Date: 24-03-2022**

**Aim:** To write a program that can implement LR parsing of the (0) items for the given grammar.

**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 non-terminal 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

**Source Code:**

**#include<iostream>**

**#include<conio.h>**

**#include<string.h>**

**using namespace std;**

**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 Grammar (Press 0 To End Input):\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\nAugumented Grammar:";**

**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\nThe Set of LR(0) Items: \n";**

**for(int z=0; z<noitem; z++)**

**{**

**cout<<"\n I"<<z<<"\n";**

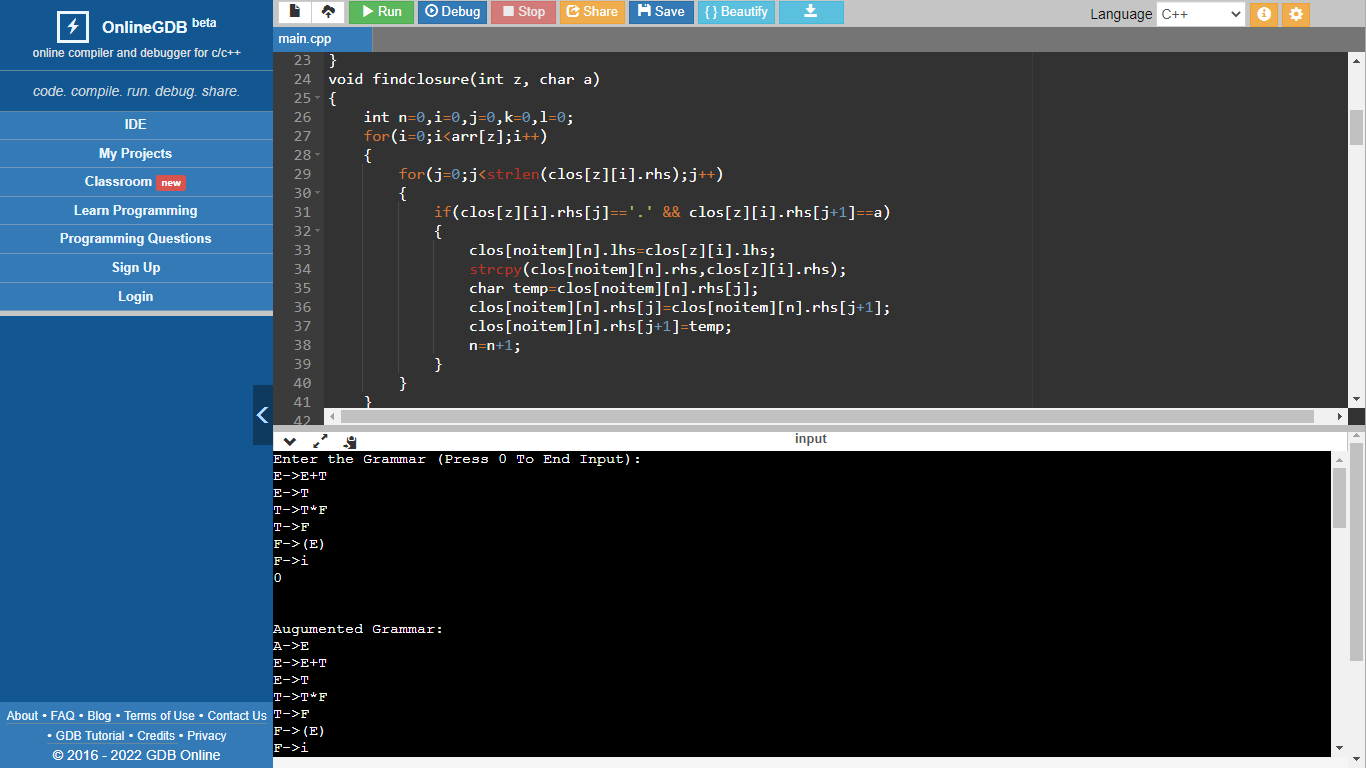
**for(j=0;j<arr[z];j++)**

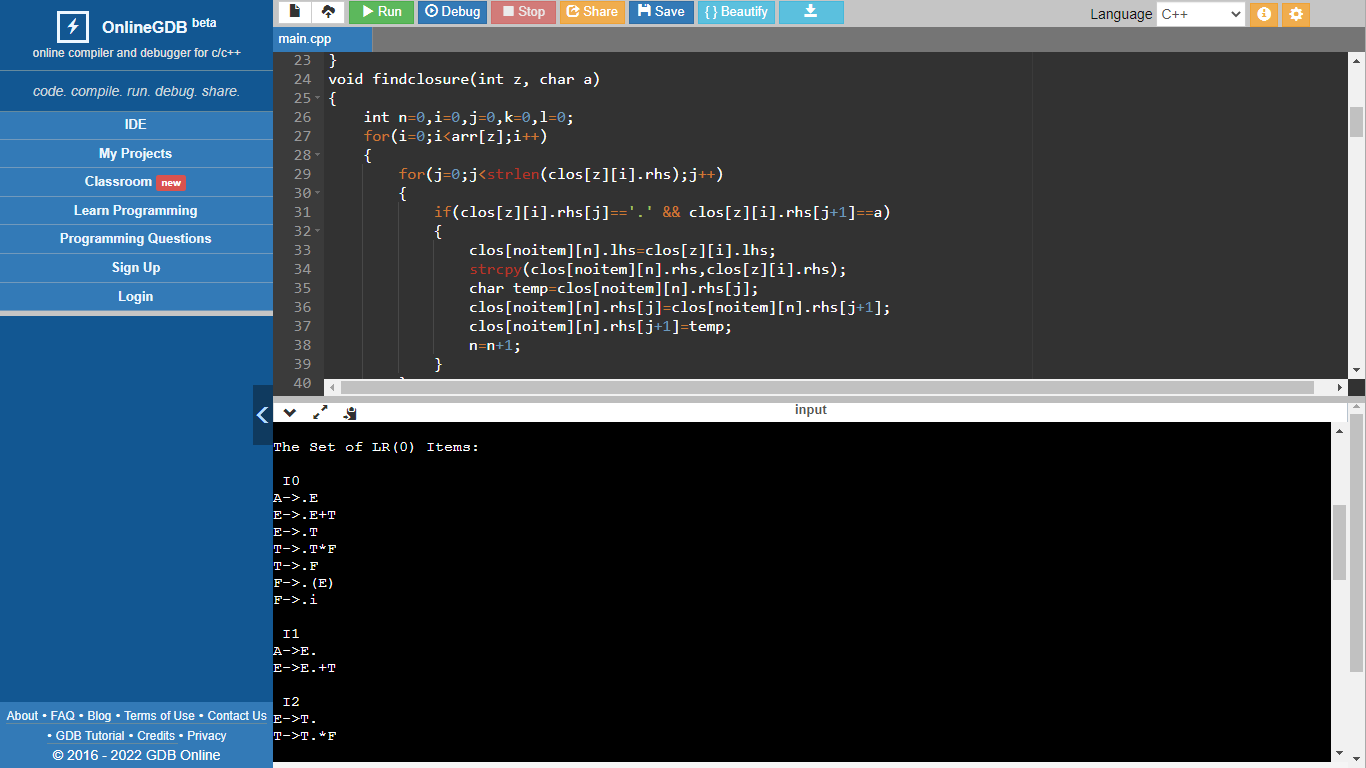
**cout<<clos[z][j].lhs<<"->"<<clos[z][j].rhs<<"\n";**

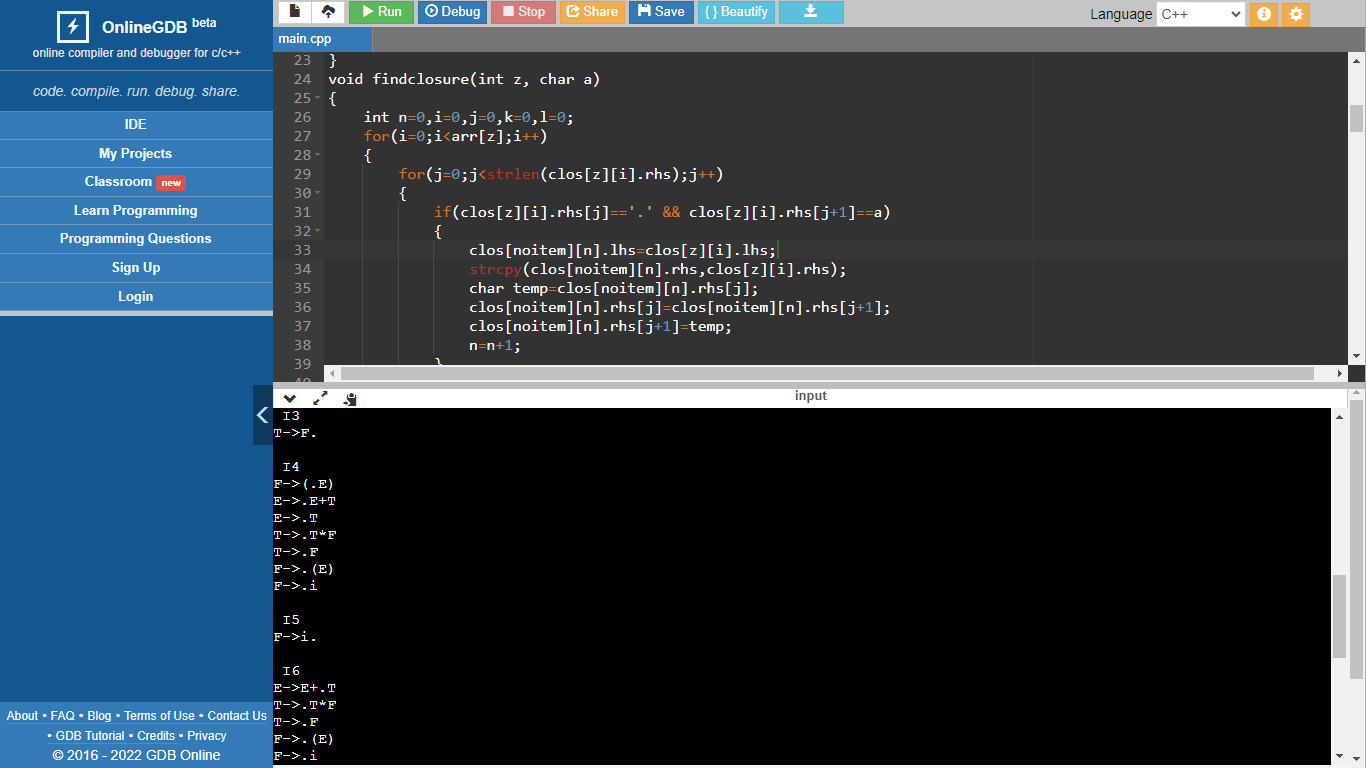
**}**

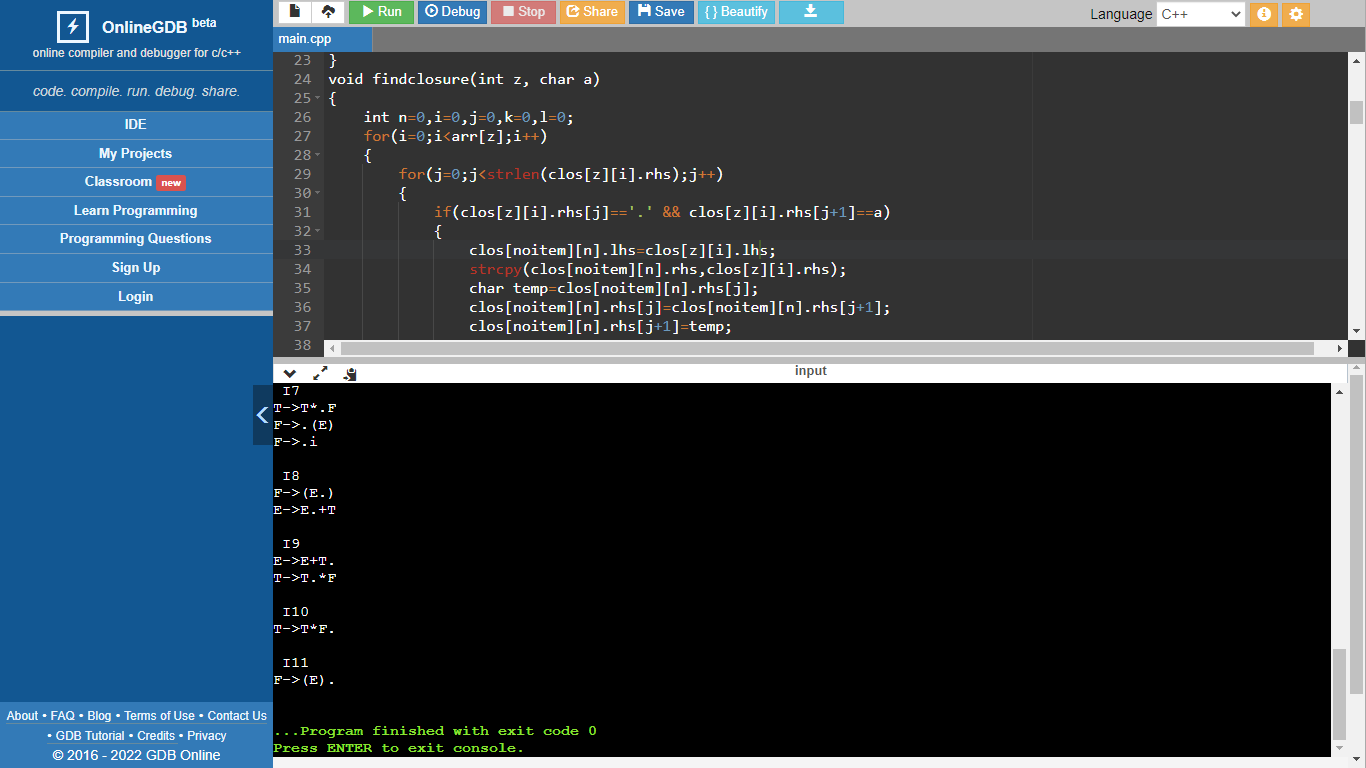
**}**

**Output:**

****

****

****

****

**Result:**

Thus, the LR(0) is successfully implemented for the given grammar.

**EXP: 10 - IMPLEMENTATION OF INTERMEDIATE CODE GENERATION -**

**PREFIX & POSTFIX**

* RA1911003010763
* **Date: 31-03-2022**

**Aim:** To write a program that can implement intermediate code generation - prefix & postfix.

**Algorithm:**

* Declare a set of operators.
* Initialise an empty stack.
* To convert INFIX to POSTFIX follow the following steps
* Scan the infix expression from left to right.
* If the scanned character is an operand, output it.
* Else, If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a ‘(‘ ), push it.
* Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack.
* If the scanned character is an ‘(‘, push it to the stack.
* If the scanned character is an ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.
* Pop and output from the stack until it is not empty.
* To convert INFIX to PREFIX follow the following steps
* First, reverse the infix expression given in the problem.
* Scan the expression from left to right.
* Whenever the operands arrive, print them.
* If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
* Repeat steps 6 to 9 until the stack is empty

**Source Code:**

**OPERATORS = set(['+', '-', '\*', '/', '(', ')'])**

**PRI = {'+': 1, '-': 1, '\*': 2, '/': 2}**

**### INFIX ===> POSTFIX ###**

**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**

**### 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)**

**Output:**



**Result:**

Thus, the program for the Implementation of Intermediate code generation, prefix & Postfix was compiled and the output was verified successfully.

**EXP: 11 - IMPLEMENTATION OF INTERMEDIATE CODE GENERATION -**

**TRIPLES, QUADRUPLES AND INDIRECT TRIPLES**

* RA1911003010763
* **Date: 07-04-2022**

**Aim:** To write a program that can implement intermediate code generation - triples, quadruples, indirect triples

**Algorithm:**

The algorithm takes a sequence of three-address statements as input. For each three address statements of the form a:= b op c perform the various actions. These are as follows:

1. Invoke afunction getreg to find out the location L where the result of computation b op c should be stored.

2. Consult the address description for y to determine y'. If the value of y currently in memory and register both then prefer the register y' . If the value of y is not already in L then generate the instruction MOV y' , L to place a copy of y in L.

3. Generate the instruction OP z' , L where z' is used to show the current location of z. if z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all other descriptors.

4. If the current value of y or z have no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of x : = y op z those register will no longer contain y or z.

**Source 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**

**### 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]**

**### THREE ADDRESS CODE GENERATION ###**

**def generate3AC(pos):**

**print("### THREE ADDRESS CODE GENERATION ###")**

**exp\_stack = []**

**t = 1**

**for i in pos:**

**if i not in OPERATORS:**

**exp\_stack.append(i)**

**else:**

**print(f't{t} := {exp\_stack[-2]} {i} {exp\_stack[-1]}')**

**exp\_stack=exp\_stack[:-2]**

**exp\_stack.append(f't{t}')**

**t+=1**

**expres = input("INPUT THE EXPRESSION: ")**

**pre = infix\_to\_prefix(expres)**

**pos = infix\_to\_postfix(expres)**

**generate3AC(pos)**

**def Quadruple(pos):**

**stack = []**

**op = []**

**x = 1**

**for i in pos:**

**if i not in OPERATORS:**

**stack.append(i)**

**elif i == '-':**

**op1 = stack.pop()**

**stack.append("t(%s)" %x)**

**print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format(i,op1,"(-)"," t(%s)" %x))**

**x = x+1**

**if stack != []:**

**op2 = stack.pop()**

**op1 = stack.pop()**

**print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format("+",op1,op2," t(%s)" %x))**

**stack.append("t(%s)" %x)**

**x = x+1**

**elif i == '=':**

**op2 = stack.pop()**

**op1 = stack.pop()**

**print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format(i,op2,"(-)",op1))**

**else:**

**op1 = stack.pop()**

**op2 = stack.pop()**

**print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format(i,op2,op1," t(%s)" %x))**

**stack.append("t(%s)" %x)**

**x = x+1**

**print("The quadruple for the expression ")**

**print(" OP | ARG 1 |ARG 2 |RESULT ")**

**Quadruple(pos)**

**def Triple(pos):**

**stack = []**

**op = []**

**x = 0**

**for i in pos:**

**if i not in OPERATORS:**

**stack.append(i)**

**elif i == '-':**

**op1 = stack.pop()**

**stack.append("(%s)" %x)**

**print("{0:^4s} | {1:^4s} | {2:^4s}".format(i,op1,"(-)"))**

**x = x+1**

**if stack != []:**

**op2 = stack.pop()**

**op1 = stack.pop()**

**print("{0:^4s} | {1:^4s} | {2:^4s}".format("+",op1,op2))**

**stack.append("(%s)" %x)**

**x = x+1**

**elif i == '=':**

**op2 = stack.pop()**

**op1 = stack.pop()**

**print("{0:^4s} | {1:^4s} | {2:^4s}".format(i,op1,op2))**

**else:**

**op1 = stack.pop()**

**if stack != []:**

**op2 = stack.pop()**

**print("{0:^4s} | {1:^4s} | {2:^4s}".format(i,op2,op1))**

**stack.append("(%s)" %x)**

**x = x+1**

**print("The triple for given expression")**

**print(" OP | ARG 1 |ARG 2 ")**

**Triple(pos)**

**Output:**



**Result:**

Thus, the program for the Implementation of Intermediate code generation - triples, quadruples, indirect triples was compiled and the output was verified successfully.

**EXP: 12 - A SIMPLE CODE GENERATOR**

**Date: 07/04/22**

**RA1911003010763**

**Aim:**

To write a program for Implementation of a simple code generator.

**Algorithm:**

1. Invoke a function getreg to find out the location L where the result of computation b op c should be stored.
2. Consult the address description for y to determine y'. If the value of y is currently in memory and register both then prefer the register y' . If the value of y is not already in L then generate the instruction MOV y' , L to place a copy of y in L.
3. Generate the instruction OP z' , L where z' is used to show the current location of z. if z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all other descriptors.
4. If the current value of y or z has no next uses or does not live on exit from the block or in register then alter the register descriptor to indicate that after execution of x : = y op z those registers will no longer contain y or z.

**Code:**

#include<stdio.h>

#include<conio.h>

#include<string.h>

char op[2],arg1[5],arg2[5],result[5];

void main()

{

FILE \*fp1,\*fp2;

fp1=fopen("input.txt","r");

fp2=fopen("output.txt","w");

while(!feof(fp1))

{

fscanf(fp1,"%s%s%s%s",op,arg1,arg2,result);

if(strcmp(op,"+")==0)

{

fprintf(fp2,"\nMOV R0,%s",arg1); fprintf(fp2,"\nADD R0,%s",arg2); fprintf(fp2,"\nMOV %s,R0",result); }

if(strcmp(op,"\*")==0)

{

fprintf(fp2,"\nMOV R0,%s",arg1); fprintf(fp2,"\nMUL R0,%s",arg2); fprintf(fp2,"\nMOV %s,R0",result); }

if(strcmp(op,"-")==0)

{

fprintf(fp2,"\nMOV R0,%s",arg1); fprintf(fp2,"\nSUB R0,%s",arg2); fprintf(fp2,"\nMOV %s,R0",result); }

if(strcmp(op,"/")==0)

{

fprintf(fp2,"\nMOV R0,%s",arg1); fprintf(fp2,"\nDIV R0,%s",arg2); fprintf(fp2,"\nMOV %s,R0",result); }

if(strcmp(op,"=")==0)

{

fprintf(fp2,"\nMOV R0,%s",arg1); fprintf(fp2,"\nMOV %s,R0",result); }

}

fclose(fp1);

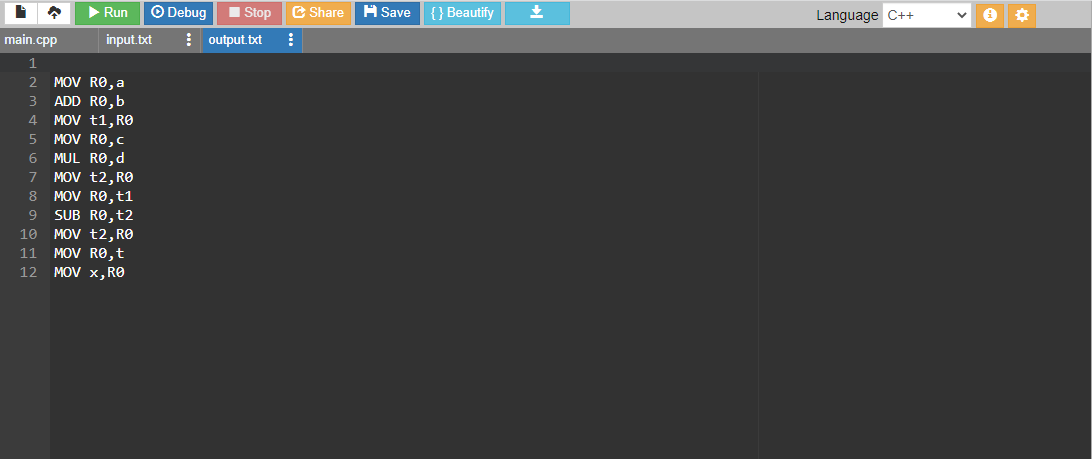
fclose(fp2);

getch();

}

**Output:**

****

****

**Result:**

Program for the Implementation of a simple code generator was compiled and the output was verified successfully.

**EXP: 13 - IMPLEMENTATION OF DAG**

* RA1911003010763
* **Date: 12-04-2022**

**Aim:** To write a program that can implement DAG.

**Algorithm:**

1. The leaves of a graph are labeled by a unique identifier and that identifier can be variable names or constants.

2. Interior nodes of the graph are labeled by an operator symbol.

3. Nodes are also given a sequence of identifiers for labels to store the computed value. 4. If y operand is undefined then create node(y).

5. If z operand is undefined then for case(i) create node(z).

6. For case(i), create node(OP) whose right child is node(z) and left child is node(y). 7. For case(ii), check whether there is node(OP) with one child node(y). 8. For case(iii), node n will be node(y).

9. For node(x) delete x from the ;list of identifiers. Append x to attached identifiers list for the node n found in step 2. Finally set node(x) to n.

**Source 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**

**{**

**int pos;**

**char op;**

**}k[15];**

**int main()**

**{**

**printf("INTERMEDIATE CODE GENERATION OF DAG\n");**

**printf("Enter Input Expression: \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;**

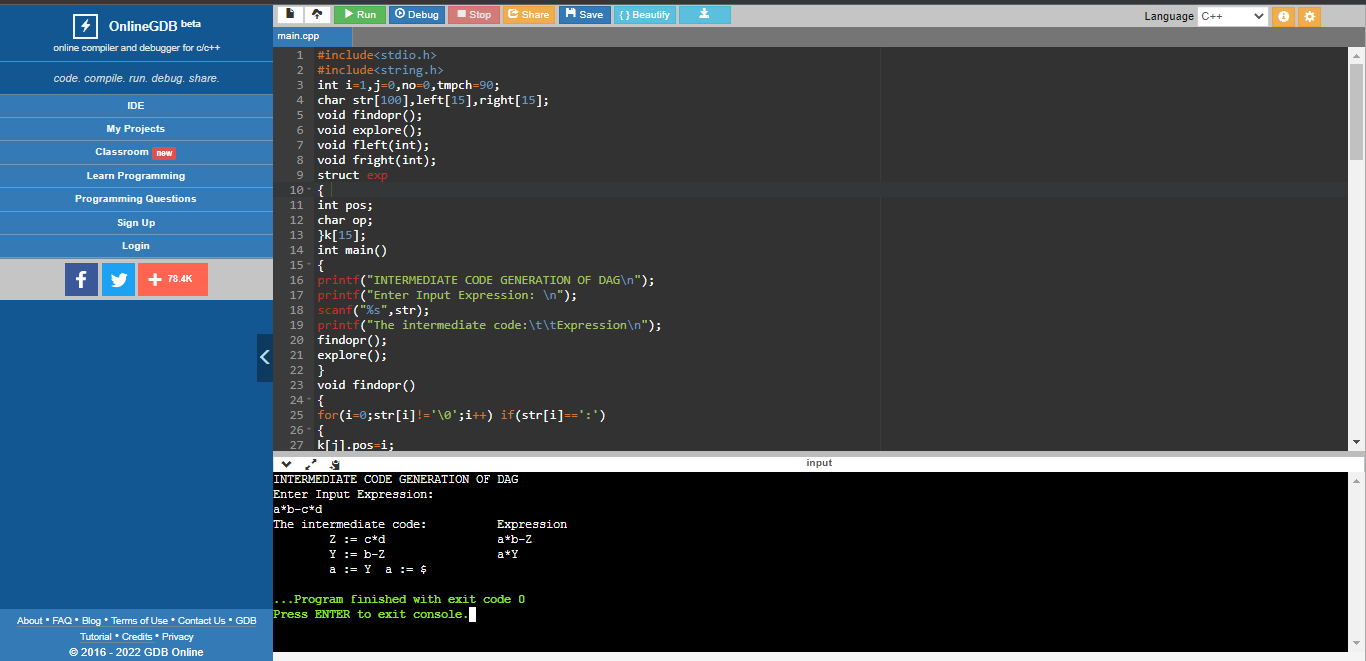
**}**

**x++;**

**}**

**}**

**Output:**



**Result:**

Thus, the program for the Implementation of DAG was compiled and the output was verified successfully.

**EXP: 14 – IMPLEMENTATION OF GLOBAL DATA FLOW ANALYSIS**

**Date: 12/04/22**

**RA1911003010763**

**Aim:**

To write a program for Implementation of Global data flow Analysis.

**Algorithm:**

1. Start the Program Execution.
2. Read the total Numbers of Expression
3. Read the Left and Right side of Each Expressions
4. Display the Expressions with Line No
5. Display the Data flow movement with Particular Expressions
6. Stop the Program Execution.

**Source Code:**

#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;

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)

{

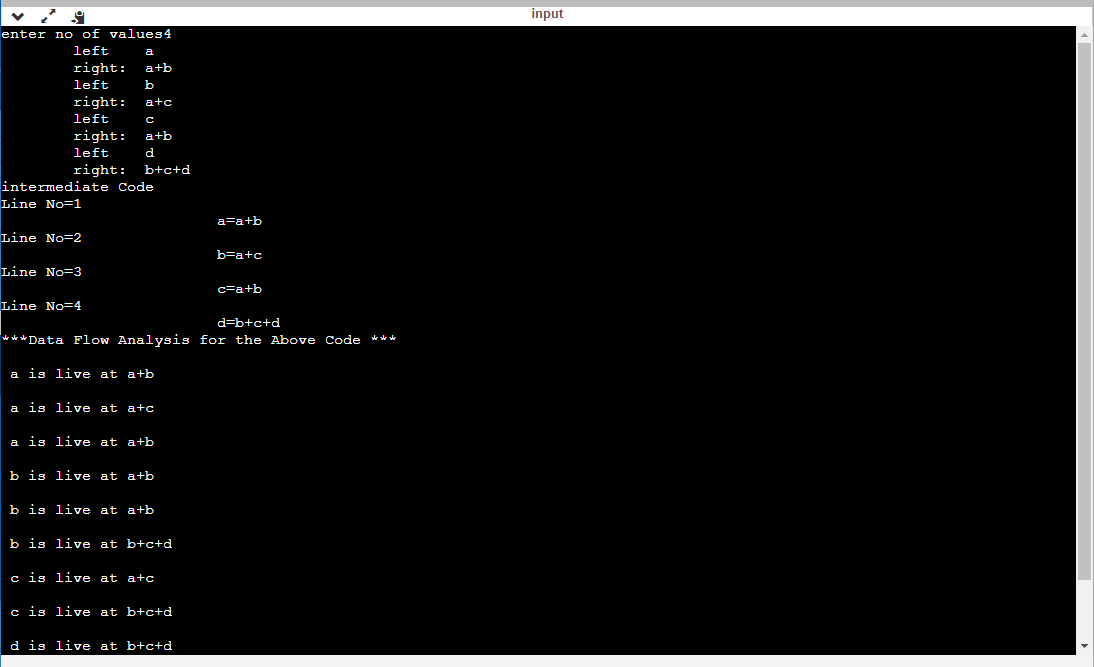
printf("\n %s is live at %s \n ", op[i].l,op[j].r); }

}

}

}

**Output:**

****

**Result:**

Program for the Implementation of Global data flow Analysis was compiled and the output was verified successfully.

**EXP: 15 - IMPLEMENT ANY STORAGE ALLOCATION STRATEGY**

**RA1911003010763**

**Date: 19.4.22**

**AIM:** To implement Stack storage allocation strategies (heap, stack, static) using a C program.

**ALGORITHM:**

1. Initially check whether the stack is empty
2. Insert an element into the stack using push operation
3. Insert more elements onto the stack until the stack becomes full
4. Delete an element from the stack using pop operation
5. Display the elements in the stack
6. Stop the program by exit

**Code:**

#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);

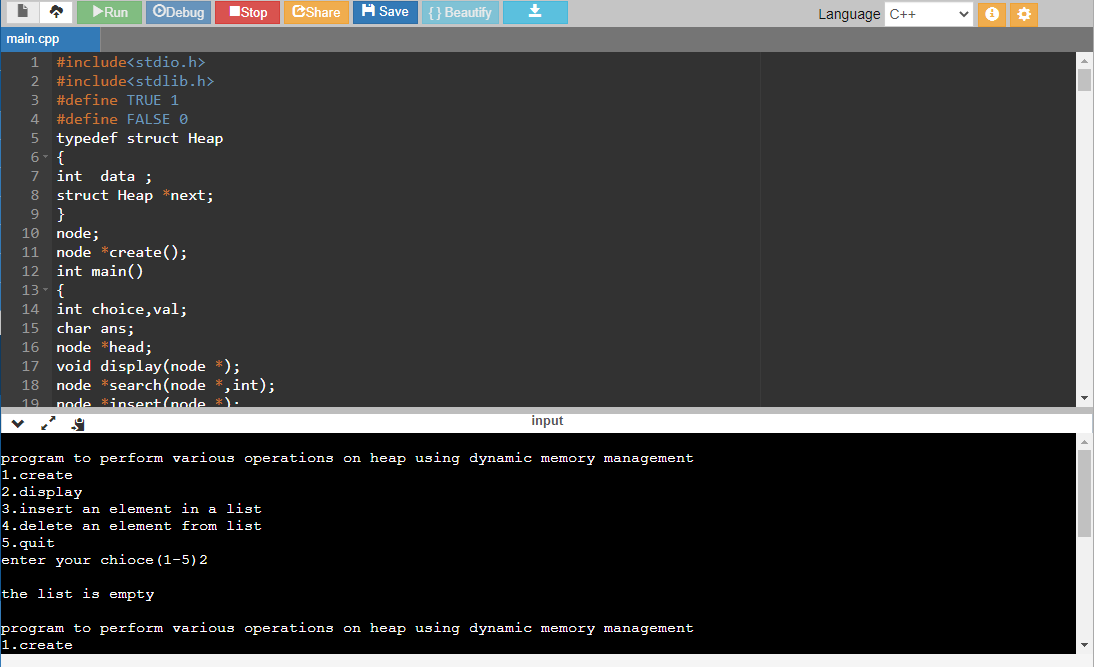
}

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

}

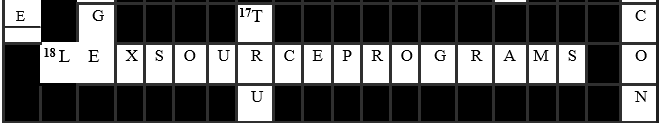
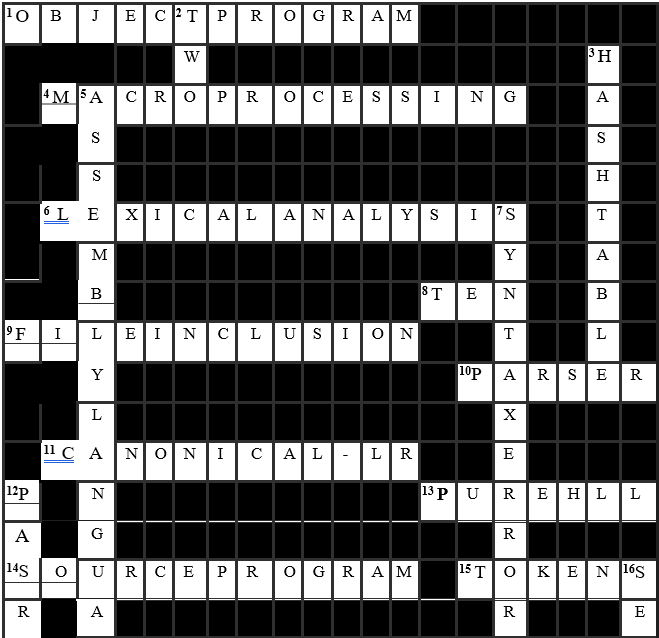
}

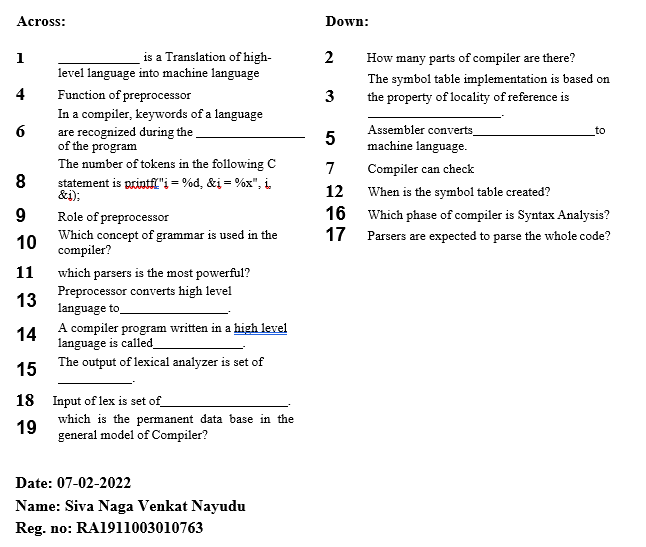
**OUTPUT:**

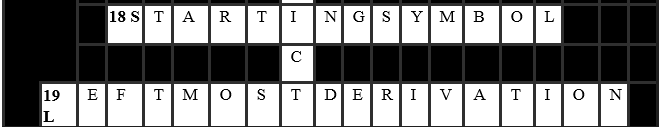
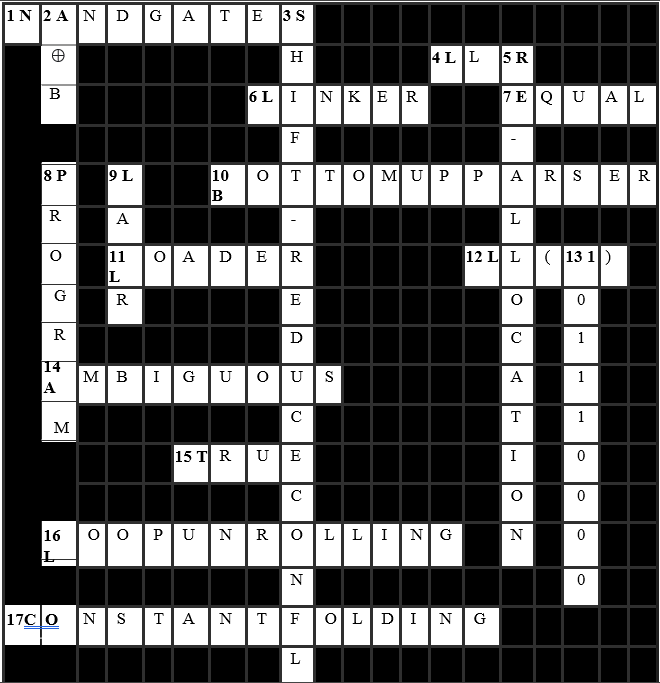
****

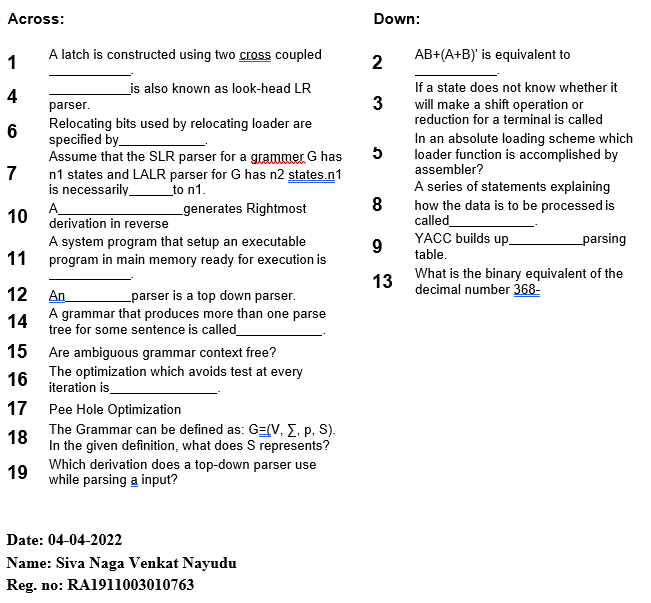
**Result:**

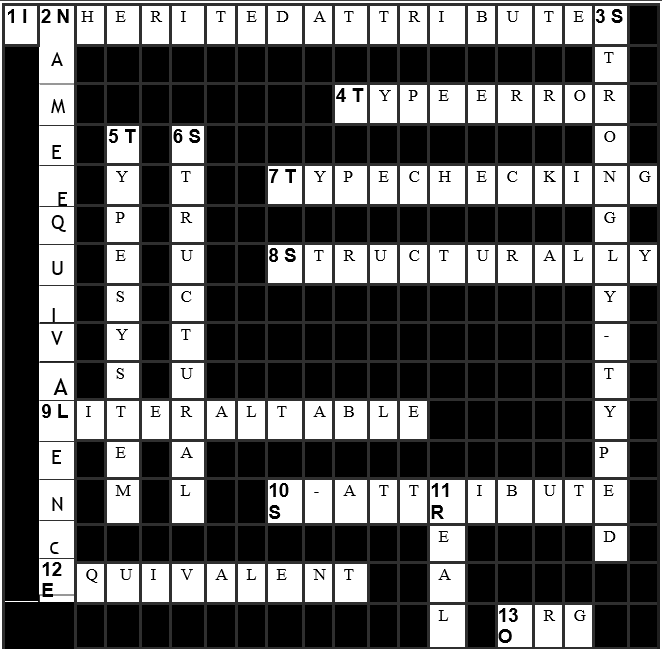
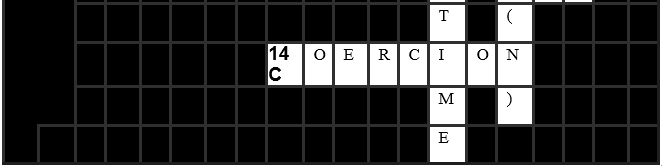
Successful implementation of Stack storage allocation strategies.

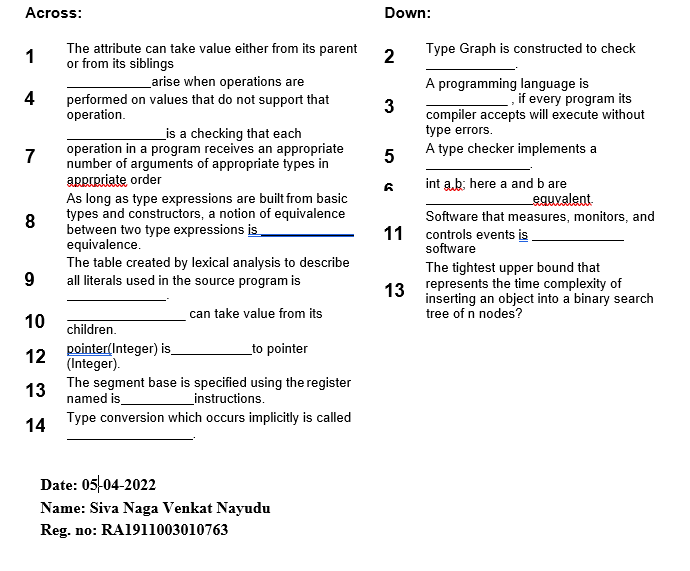
**18CSC304J / Compiler Design / Dr. Shiny Irene D** - **UNIT I**

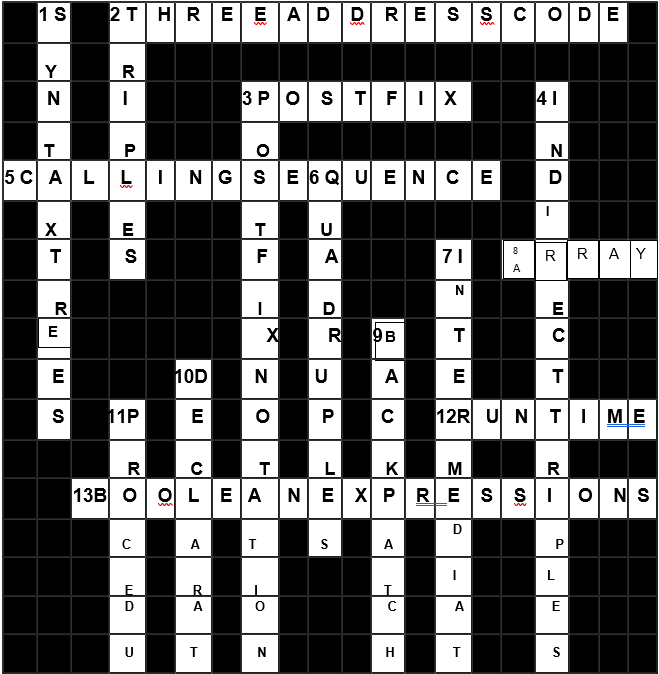
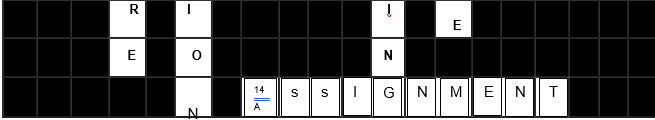
****

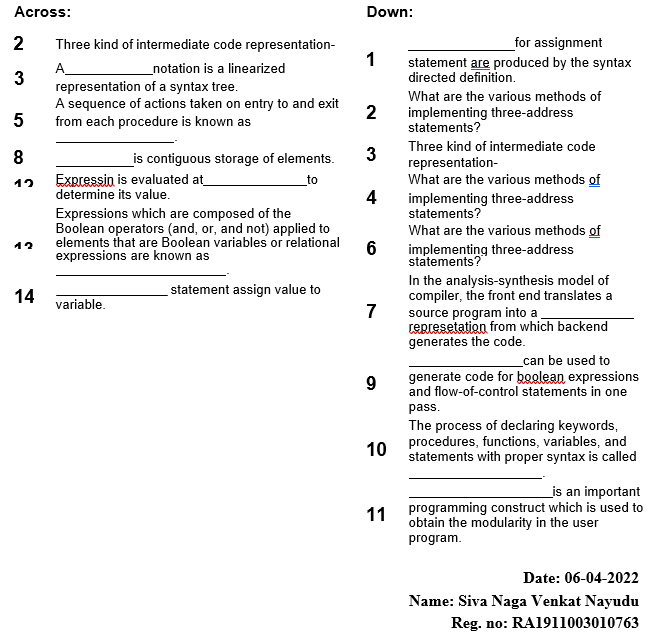
**18CSC304J / Compiler Design / Dr. Shiny Irene D** - **UNIT II**

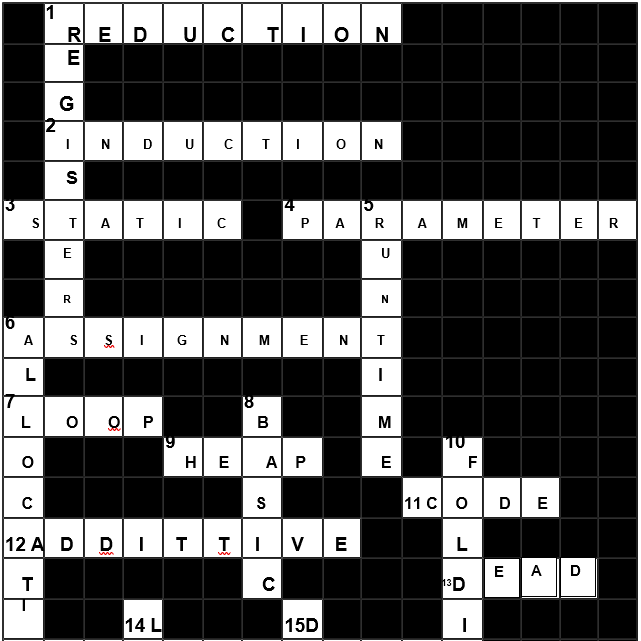
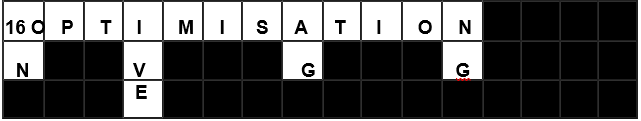
****

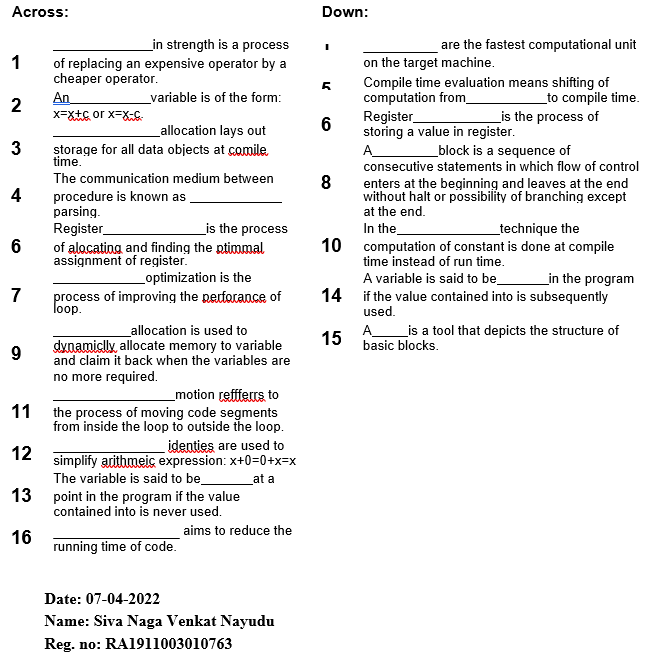
**18CSC304J / Compiler Design / Dr. Shiny Irene D** - **UNIT III** **** 

****

**18CSC304J / Compiler Design / Dr. Shiny Irene D** - **UNIT IV** **** 

****

**18CSC304J / Compiler Design / Dr. Shiny Irene D** - **UNIT V** **** 

****

**Understanding APmap Methodologies**

**Authors:**

* **Naveed Shaik (RA1911003010760)**
* **Siva Nayudu (RA1911003010763)**
* **Shivang Tripathi (RA1911003010765)**

**Abstract:**

Automata processors (APs), a new form of hardware accelerator, had been proposed to hurry up finite-nation automata. A hierarchical routing matrix that connects a massive quantity of memory arrays is called AP's bone shape. An AP may also technique an enter image each clock cycle with this arrangement, ensuing in drastically more overall performance than conventional systems. However, the layout automation for APs has now no longer been very well investigated. This have a look at shows using APmap, a totally computerized device for mapping automata to APs with a two-stage routing matrix. APmap divides a big automaton into small graphs earlier than mapping them. To meet hardware limits, APmap applies many adjustments to the automaton. Experiments on a not unusualplace benchmark suite display that once in comparison to the nation of the art, our method makes use of more or less 19% much less storage.

**Introduction:**

In fields including bioinformatics, network security and artificial intelligence, FINITE-state automata (FSA) are commonly employed. RRAM-AP, Cache Automaton and Micron Automata Processor (MAP) are examples of creative hardware architectures that repurpose memory arrays for faster FSA execution.

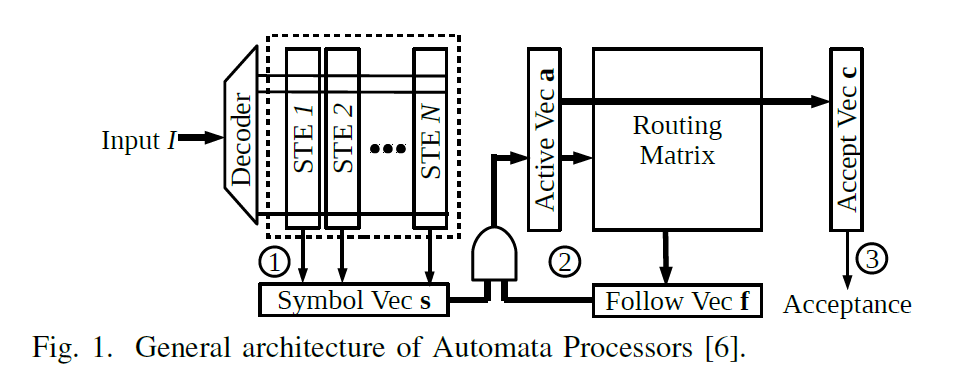
For the time being, there aren't any open-supply layout equipment for APs. Cache Automaton's creators defined their approach for mapping FSA to hardware. Because to the boundaries at the routing matrix, a few FSAs cannot be mapped without delay the usage of their methods. As a result, those FSA have to be transformed into other kinds which are similar. This is an iterative system that necessitates previous experience. As for different associated works, For MAP, Micron gives a industrial software program improvement kit (SDK). This SDK cannot be prolonged for different architectures, along with Cache Automaton, due to the fact it's miles closed-supply. By reproducing a preliminary result, RAPID's compiler can produce mapping. As a result, it cannot be utilised through itself. While open-supply equipment like REAPR and Grapefruit were advanced for mapping packages to FPGAs, an equal device for APs remains needed.

This work solves the aforementioned concerns and introduces APmap1 which is an open-source compiler for APs like RRAM-AP and Cache Automaton. A two-level routing matrix is used by these APs. Due to APmap's algorithm limitations, it cannot be used with MAP. APmap employs a variety of algorithms to convert provided FSAs into equivalent forms that satisfy the requirements.

**Existing System:**

An input symbol entered (I) will be processed in 3 key phases throughout each clock cycle:

1. **Matching the Input Symbols:** The total states which have incoming transitions occuring on I can be recognized on this step. Each state is represented by state transition elements (STE), which can be pre-configured primarily constructed on the automaton in question. With respect to the input symbol I, the decoder turns on one of the word lines. If incoming transition does not occur on I, the output of a STE is logic 0; otherwise, it is logic 1. All STEs' outputs are transferred to a vector known as S (Symbol Vector).
2. **Active State Processing:** Depending on the transition function, which is stored in the routing matrix, it creates all potential states that may be reached from the presently active states and saves the result in the Follow Vector f. By bit-wise ANDing s and f, this phase also creates the following active states.

****

1. **Output Identification:** c, an accept vector, is pre-configured depending on the accept states, C, of the automaton. This particular stage determines if the input sequence is acceptable by looking at the intersection of a and c.

The ideal FSA requires the design of numerous parts, including STEs, the directing grid, and Accept Vector c. APmap will make the arrangement for you.

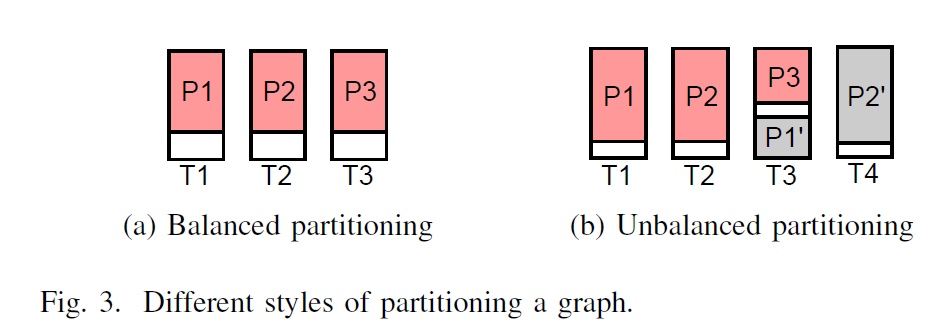
**Objective:**

The objective of this paper is to analyze the current existing mapping tools and to propose a new/more efficient tool for mapping automata to AP chips.

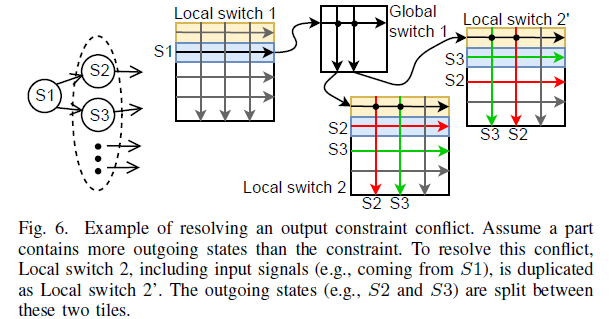
**APmap Methodologies:**

We recommend combining APmap with a few additional tools to create apps that target APs. RAPID may be used to create the application. As result, RAPID's compiler produces Automata Network Markup Language (ANML) documents. VASim, an instrument that impersonates the execution of a homogeneous machine, can parse ANML. Some key automata changes, for example, prefix blending, are likewise upheld. We changed VASim to preprocess the automata and make the APmap-viable record designs. The automata are depicted as an assortment of associated parts (CCs), which are nonoverlapping subsets of the first automata. At long last, APmap creates AP design records. APmap organizes the CCs as per their state numbers prior to planning them individually. APmap chooses the greatest unmapped CC in every cycle and guides it to at least one tiles. This CC may not possess all of the region of these tiles in certain conditions. As a result, APmap searches for some tiny CCs to fill in the gaps. This procedure is repeated until all CCs have been mapped. Steps involved are briefly discussed below.

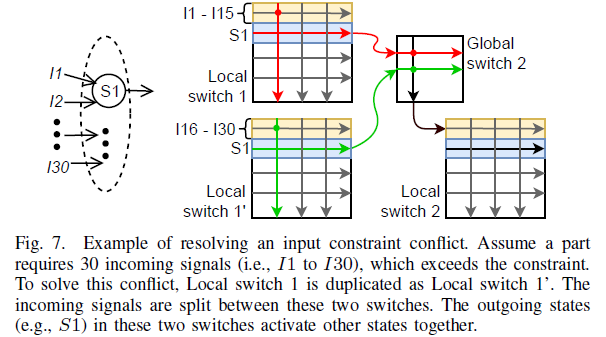
1. **CC Partitioning:** With the aid of METIS, APmap splits a CC. METIS separates an undirected graph into k non-overlapping sections, cutting as few edges as possible. When an edge is divided, the two nodes connected by the edge are assigned to distinct sections. To make the CC suitable for METIS, we first convert it to an undirected graph. This transformation eliminates self-loops. Then, given two types of input parameters, we activate METIS with the number of pieces and the size limits on those parts. A size constraint is the intended size of a component, and a size constraint is the number of nodes included in that part. This dividing technique is iterated with fluctuated settings to decrease capacity use, otherwise called the quantity of tiles to which the automata is planned. It is impacted by the partitioning outcome in three ways. To begin, each partitioned component will be mapped to a distinct tile; hence, the number of partitioned parts is the most essential aspect in the ultimate usage. The cutting edges will then be mapped to global wires. The wire supply is restricted. As a result, a partitioning result with several cutting edges may result in overhead for resolving the constraint conflict. Finally, the balance of partitioned sections may have an impact on usage. Figure 3 depicts two alternative CC division strategies. The first two portions of the imbalanced partitioning are roughly the size of a tile, while the third component is rather tiny. This one is a significant option since T3's whitespace may be utilised to map a tiny CC. It may also be used to map a section of another huge CC. We like the imbalanced partitioning technique since it gives more chances for optimization.



1. **Mapping Method:** The arrangement of the routing matrix, for example the global and local switches, is the accentuation of this part. VAsim creates the setup of STEs, including their connected information images and whether they are in the beginning or last stages. This information is just copied to the final configuration file by APmap. In two processes, the transitions between states in distinct tiles are mapped to numerous global and local switches: first, the global switches, then the local switches. The stages are illustrated in Fig. 4 with an example. The global switch component is successfully mapped if all of the preceding conditions are met. In the event that any of the circumstances aren't met, APmap continues on toward the following worldwide switch and checks once more. The planning method fizzles in the event that none of the worldwide switches satisfy the necessity.
2. **Meeting Constraints:** Hardware resource restrictions and poor partitioning cause constraint violations. The number of transitions between portions of a large CC is unrestricted when it is divided into many parts. METIS makes every effort to keep the impact to a minimum. The strategies for tending to yield requirement clashes are introduced first, trailed by the info. Simply 24 result wires connect to worldwide switches on a tile. Any express that travels to states in different tiles, known as an active state, should be planned in those 24 openings. all transition numbers, the change number might outperform the quantity of wires associating a tile to worldwide switches. We should address this issue prior to planning it to the tile in this situation.



In Fig. 6, these two segments are planned to Tile 2 and 2', separately, as a potential planning result. APmap copies input requirements to address input limitation clashes. When contrasted with yield imperative goal, be that as it may, another arrangement is fundamental. The copies initiate the states in different parts mutually for the active states in this part (e.g., S1), as represented in Fig. 7. This arrangement guarantees that automata execution is right. Expect to be that (one of) the info signals I1 to I30 can set off S1. Following the duplication, the S1 in either Local switch 2 or 3 is dynamic (or the two of them are), and it (or they) will enact different states through the red or green diverts in the picture.



**Conclusion:**

We introduced an open-source program/instrument called APmap for planning automata to AP contributes this work. Different enhancements are utilized to robotize the planning system and decrease stockpiling utilization. An assessment of APmap utilizing the ANMLzoo benchmark suite uncovers that it has a negligible upward and outflanks the state of the art.