1. **IMPLEMENTATION OF LEXICAL ANALYZER**

**AIM:**  
To develop a lexical analyzer to identify identifiers, constants, comments, operators etc using C program  
  
**ALGORITHM:**  
**Step1:** Start the program.  
**Step2:** Declare all the variables and file pointers.  
 **Step3:** Display the input program.  
**Step4:** Separate the keyword in the program and display it.  
**Step5:** Display the header files of the input program  
**Step6:** Separate the operators of the input program and display it.  
 **Step7:** Print the punctuation marks.  
**Step8:**Print the constant that are present in input program.  
**Step9:** Print the identifiers of the input program

**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;  inti,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**  The Program is : 'float x = a + 1b; '  All Tokens are :  Valid keyword : 'float'  Valid Identifier : 'x'  Valid operator : '='  Valid Identifier : 'a'  Valid operator : '+'  Invalid Identifier : '1b'  **RESULT:**  Thus the C program to Implementation of Lexical Analyzer has been executed and the output has been verified successfully |

1. **CONVERSION OF REGULAR EXPRESSION TO NFA**

**AIM:**

To write a C program to convert the regular expression to NFA.

**ALGORITHM:**

1. Start the program.
2. Declare all necessary header files.
3. Define the main function.
4. Declare the variables and initialize variables r & c to ‘0’.
5. Use a for loop within another for loop to initialize the matrix for NFA states.
6. Get a regular expression from the user & store it in ‘m’.
7. Obtain the length of the expression using strlen() function and store it in ‘n’.
8. Use for loop upto the string length and follow steps 8 to 12.
9. Use switch case to check each character of the expression

10.If case is ‘\*’, set the links as ‘E’ or suitable inputs as per rules.

11.If case is ‘+’, set the links as ‘E’ or suitable inputs as per rules.

12.Check the default case, i.e.,for single alphabet or 2 consecutive alphabets and set the links to respective alphabet.

13.End the switch case.

14.Use for loop to print the states along the matrix.

15.Use a for loop within another for lop and print the value of respective links.

16.Print the states start state as ‘0’ and final state.

17.End the program.

**PROGRAM:**

#include<stdio.h>

#include<conio.h>

void main()

{

char m[20],t[10][10];

intn,i,j,r=0,c=0;

clrscr();

printf("\n\t\t\t\tSIMULATION OF NFA");

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

for(i=0;i<10;i++)

{

for(j=0;j<10;j++)

{

t[i][j]=' ';

}

}

printf("\n\nEnter a regular expression:");

scanf("%s",m);

n=strlen(m);

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

{

switch(m[i])

{

case '|' : {

t[r][r+1]='E';

t[r+1][r+2]=m[i-1];

t[r+2][r+5]='E';

t[r][r+3]='E';

t[r+4][r+5]='E';

t[r+3][r+4]=m[i+1];

r=r+5;

break;

}

case '\*':{

t[r-1][r]='E';

t[r][r+1]='E';

t[r][r+3]='E';

t[r+1][r+2]=m[i-1];

t[r+2][r+1]='E';

t[r+2][r+3]='E';

r=r+3;

break;

}

case '+': {

t[r][r+1]=m[i-1];

t[r+1][r]='E';

r=r+1;

break;

}

default:

{

if(c==0)

{

if((isalpha(m[i]))&&(isalpha(m[i+1])))

{

t[r][r+1]=m[i];

t[r+1][r+2]=m[i+1];

r=r+2;

c=1;

}

c=1;

}

else if(c==1)

{

if(isalpha(m[i+1]))

{

t[r][r+1]=m[i+1];

r=r+1;

c=2;

}

}

else

{

if(isalpha(m[i+1]))

{

t[r][r+1]=m[i+1];

r=r+1;

c=3;

}

}

}

break;

}

}

printf("\n");

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

printf(" %d",j);

printf("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

printf("\n");

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

{

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

{

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

}

printf(" | %d",i);

printf("\n");

}

printf("\nStart state: 0\nFinal state: %d",i-1);

getch();

}

**OUTPUT:**

Enter a regular Expression: a|b

SIMULATION OF NFA

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter a regular expression:a|b

0 1 2 3 4 5

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

E E | 0

a | 1

E | 2

b | 3

E | 4

| 5

Start state: 0

Final state: 5

**RESULT:**

Thus the C program to convert regular expression to NFA has been executed and the output has been verified successfully.

1. **CONVERSION OF NFA TO DFA**

**AIM:**

To write a program to convert NFA to DFA in C.

**ALGORITHM:**

1. Start the program
2. Initially Q' = ϕ
3. The final state is assigned to F
4. Add q0 of NFA to Q'. Then find the transitions from this start state.
5. In Q', find the possible set of states for each input symbol. If this set of states is not in Q', then add it to Q'.
6. In DFA, the final state will be all the states which contain F (final states of NFA)
7. Stop the program

**PROGRAM:**

|  |  |
| --- | --- |
|  | |
| #include<stdio.h> | |
|  | #include<string.h> |
|  | #include<math.h> |
|  |  |
|  | int ninputs; |
|  | int dfa[100][2][100] = {0}; |
|  | int state[10000] = {0}; |
|  | char ch[10], str[1000]; |
|  | int go[10000][2] = {0}; |
|  | int arr[10000] = {0}; |
|  |  |
|  | int main() |
|  | { |
|  | int st, fin, in; |
|  | int f[10]; |
|  | int i,j=3,s=0,final=0,flag=0,curr1,curr2,k,l; |
|  | int c; |
|  |  |
|  | printf("\nFollow the one based indexing\n"); |
|  |  |
|  | printf("\nEnter the number of states::"); |
|  | scanf("%d",&st); |
|  |  |
|  | printf("\nGive state numbers from 0 to %d",st-1); |
|  |  |
|  | for(i=0;i<st;i++) |
|  | state[(int)(pow(2,i))] = 1; |
|  |  |
|  | printf("\nEnter number of final states\t"); |
|  | scanf("%d",&fin); |
|  |  |
|  | printf("\nEnter final states::"); |
|  | for(i=0;i<fin;i++) |
|  | { |
|  | scanf("%d",&f[i]); |
|  | } |
|  |  |
|  | int p,q,r,rel; |
|  |  |
|  | printf("\nEnter the number of rules according to NFA::"); |
|  | scanf("%d",&rel); |
|  |  |
|  | printf("\n\nDefine transition rule as \"initial state input symbol final state\"\n"); |
|  |  |
|  |  |
|  |  |
|  | for(i=0; i<rel; i++) |
|  | { |
|  | scanf("%d%d%d",&p,&q,&r); |
|  | if (q==0) |
|  | dfa[p][0][r] = 1; |
|  | else |
|  | dfa[p][1][r] = 1; |
|  | } |
|  |  |
|  | printf("\nEnter initial state::"); |
|  | scanf("%d",&in); |
|  |  |
|  | in = pow(2,in); |
|  |  |
|  | i=0; |
|  |  |
|  | printf("\nSolving according to DFA"); |
|  |  |
|  | int x=0; |
|  | for(i=0;i<st;i++) |
|  | { |
|  | for(j=0;j<2;j++) |
|  | { |
|  | int stf=0; |
|  | for(k=0;k<st;k++) |
|  | { |
|  | if(dfa[i][j][k]==1) |
|  | stf = stf + pow(2,k); |
|  | } |
|  |  |
|  |  |
|  | go[(int)(pow(2,i))][j] = stf; |
|  | printf("%d-%d-->%d\n",(int)(pow(2,i)),j,stf); |
|  | if(state[stf]==0) |
|  | arr[x++] = stf; |
|  | state[stf] = 1; |
|  | } |
|  |  |
|  | } |
|  |  |
|  |  |
|  | //for new states |
|  | for(i=0;i<x;i++) |
|  | { |
|  | printf("for %d ---- ",arr[x]); |
|  | for(j=0;j<2;j++) |
|  | { |
|  | int new=0; |
|  | for(k=0;k<st;k++) |
|  | { |
|  | if(arr[i] & (1<<k)) |
|  | { |
|  | int h = pow(2,k); |
|  |  |
|  | if(new==0) |
|  | new = go[h][j]; |
|  | new = new | (go[h][j]); |
|  |  |
|  |  |
|  | } |
|  | } |
|  |  |
|  | if(state[new]==0) |
|  | { |
|  | arr[x++] = new; |
|  | state[new] = 1; |
|  | } |
|  | } |
|  | } |
|  |  |
|  | printf("\nThe total number of distinct states are::\n"); |
|  |  |
|  | printf("STATE     0   1\n"); |
|  |  |
|  | for(i=0;i<10000;i++) |
|  | { |
|  | if(state[i]==1) |
|  | { |
|  | //printf("%d\*\*",i); |
|  | int y=0; |
|  | if(i==0) |
|  | printf("q0 "); |
|  |  |
|  | else |
|  | for(j=0;j<st;j++) |
|  | { |
|  | int x = 1<<j; |
|  | if(x&i) |
|  | { |
|  | printf("q%d ",j); |
|  | y = y+pow(2,j); |
|  | //printf("y=%d  ",y); |
|  | } |
|  | } |
|  | //printf("%d",y); |
|  | printf("       %d   %d",go[y][0],go[y][1]); |
|  | printf("\n"); |
|  | } |
|  | } |
|  |  |
|  |  |
|  |  |
|  | j=3; |
|  | while(j--) |
|  | { |
|  | printf("\nEnter string"); |
|  | scanf("%s",str); |
|  | l = strlen(str); |
|  | curr1 = in; |
|  | flag = 0; |
|  | printf("\nString takes the following path-->\n"); |
|  | printf("%d-",curr1); |
|  |  |
|  | for(i=0;i<l;i++) |
|  | { |
|  | curr1 = go[curr1][str[i]-'0']; |
|  | printf("%d-",curr1); |
|  | } |
|  |  |
|  | printf("\nFinal state - %d\n",curr1); |
|  |  |
|  | for(i=0;i<fin;i++) |
|  | { |
|  | if(curr1 & (1<<f[i])) |
|  | { |
|  | flag = 1; |
|  | break; |
|  | } |
|  | } |
|  |  |
|  | if(flag) |
|  | printf("\nString Accepted"); |
|  | else |
|  | printf("\nString Rejected"); |
|  |  |
|  | } |
|  |  |
|  |  |
|  | return 0; |
|  | } |
|  |  |

**OUTPUT:**

Follow the one based indexing  
Enter the number of states::3  
Give state numbers from 0 to 2  
Enter number of final states 1  
Enter final states::4  
Enter the number of rules according to NFA::4  
Define transition rule as "initial state input symbol final state"  
1 0 1  
1 1 1  
1 0 2  
2 0 4  
Enter initial state::1  
Solving according to DFA1-0-->0  
1-1-->0  
2-0-->6  
2-1-->2  
4-0-->0  
4-1-->0  
for 0 ---- for 0 ----  
The total number of distinct states are::  
STATE 0 1  
q0 0 0  
q0 0 0  
q1 6 2  
q2 0 0  
q1 q2 0 0

Enter stringabbb

String takes the following path-->  
2-0-0-0-0-  
Final state - 0

String Rejected  
Enter stringabba

String takes the following path-->  
2-0-0-0-0-  
Final state - 0

String Rejected  
Enter string b

String Accepted

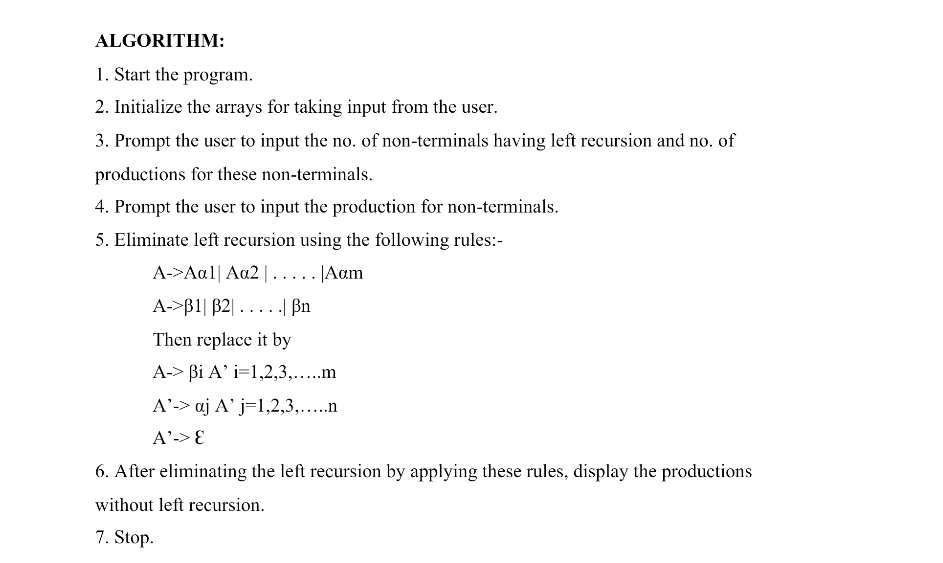
**RESULT:**

Thus, the C program to convert NFA to DFA has been executed and the output has been verified successfully.

1. **ELIMINATION OF AMBIGUITY, LEFT RECURSION AND LEFT FACTORING**

**AIM:**

To study the ambiguity, perform Left recursion and Left factoring.



**PROGRAM:**

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

{

char input[100], l[50],r[50],temp[10],tempprod[20],productions[25][50]; int i=0,j=0,flag=0,consumed=0;

printf(“Enter the Productions:”); scanf(“ %ls->%s”, l, r);

printf(“ %s”, r);

while(sscanf(r+consumed, “ % [^l] s”, temp) == 1 &&consumed<=strlen(r))

{

if(temp[0] == l[0])

{

flag = 1;

sprintf(productions[i++], “%s->%s%s ‘\0”, l,temp+1,1);

}

else

sprintf(productions[i++], “%s->%s%s ‘\0”,l, temp,1); consumed += strlen(temp)+1;

}

if(flag==1)

{

sprintf(productions[i++], “%s->€ \0”, 1);

printf(“the productions after eliminating left recursion are:\n”); for(j=0;j<i;j++)

printf(“%s \n “, productions[j]);

}

else

printf(“ The Given Grammar has no Left Recursion”);

}

**OUTPUT:**

Enter the Productions: E-E+T

The productions after eliminating Left Recursion are:E->+TE’ E->

Enter the Productions:

T->T\*F

The productions after eliminating Left Recursion are: T-> \*FT’ T->

Enter the Productions:

F->id

The Given Grammar has no Left Recursion

**/\* Program to Implement Recursive Descent Parsing \*/**

#include<stdio.h>

//#include<conio.h> #include<string.h> char input[100];

int i,l;

int main()

{

printf("recursive decent parsing for the grammar");

printf("\n E->TEP|\nEP->+TEP|@|\nT->FTP|\nTP->\*FTP|@|\nF-

>(E)|ID\n");

printf("enter the string to check:"); scanf("%s",input);

if(E()){

if(input[i]=='$')

printf("\n string is accepted\n"); else

printf("\n string is not accepted\n");

}

} E(){

if(T()){

if(EP())

return(1); else return(0);

}

else return(0);

} EP(){

if(input[i]=='+'){ i++;

if(T()){

if(EP())

return(1); else return(0);

}

else return(1);

}

} T(){

if(F()){

if(TP())

return(1); else return(0);

}

else return(0);

printf("String is not accpeted\n");

} TP(){

if(input[i]=='\*'){ i++;

if(F()){

if(TP())

return(1); else return(0);

}

else return(0);

printf("The string is not accepted\n");

}

else return(1);

} F(){

if(input[i]=='('){ i++;

if(E()){

if(input[i]==')'){ i++;

return(1);

}

else

{

return(0);

printf("String is not accepted\n");

}

}

else return(0);

}

else if(input[i]>='a'&&input[i]<='z'||input[i]>='A'&&input[i]<='Z')

{ i++;

return(1);

}

else return(0);

}

**OUTPUT**:

$ cc rdp.c

$ ./a.out

recursive decent parsing for the grammar E->TEP|

EP->+TEP|@| T->FTP|

TP->\*FTP|@| F->(E)|ID

enter the string to check:(i+i)\*i string is accepted

**RESULT:**

Thus, the process Elimination of Ambiguity,Left Recursion & Left Factoring have been studied.

1. **FIRST AND FOLLOW COMPUTATION**

**AIM:**

To calculate the first and Follow of the given expression

**ALGORITHM:**

1. Start the program
2. In the production the first terminal on R.H.S becomes the first of it
3. If the first character is non-terminal then its first is taken else Follow of left is taken
4. To find Follow find where all the non terminals appear. the first of its Follows is its Follow
5. If the Follow is t then the Follow of left is taken
6. Finally print first and its Follow
7. Stop the program

**SOURCE CODE:\_**

#include<stdio.h>

#include<ctype.h>

#include<string.h>

void followfirst(char, int, int);

void follow(char c);

void findfirst(char, int, int);

int count, n = 0;

  char calc\_first[10][100];

  char calc\_follow[10][100];

int m = 0;

char production[10][10];

char f[10], first[10];

int k;

char ck;

int e;

  int main(int argc, char \*\*argv)

{

    int jm = 0;

    int km = 0;

    int i, choice;

    char c, ch;

    count = 8;

    strcpy(production[0], "E=TR");

    strcpy(production[1], "R=+TR");

    strcpy(production[2], "R=#");

    strcpy(production[3], "T=FY");

    strcpy(production[4], "Y=\*FY");

    strcpy(production[5], "Y=#");

    strcpy(production[6], "F=(E)");

    strcpy(production[7], "F=i");

    int kay;

    char done[count];

    int ptr = -1;

    for(k = 0; k < count; k++) {

        for(kay = 0; kay < 100; kay++) {

            calc\_first[k][kay] = '!';

        }

    }

    int point1 = 0, point2, xxx;

          for(k = 0; k < count; k++)

    {

        c = production[k][0];

        point2 = 0;

        xxx = 0;

        for(kay = 0; kay <= ptr; kay++)

            if(c == done[kay])

                xxx = 1;

        if (xxx == 1)

            continue;

        findfirst(c, 0, 0);

        ptr += 1;

        done[ptr] = c;

        printf("\n First(%c) = { ", c);

        calc\_first[point1][point2++] = c;

        for(i = 0 + jm; i < n; i++) {

            int lark = 0, chk = 0;

            for(lark = 0; lark < point2; lark++) {

                if (first[i] == calc\_first[point1][lark])

                {

                    chk = 1;

                    break;

                }

            }

            if(chk == 0)

            {

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

                calc\_first[point1][point2++] = first[i];

            }

        }

        printf("}\n");

        jm = n;

        point1++;

    }

    printf("\n");

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

    char donee[count];

    ptr = -1;

    for(k = 0; k < count; k++) {

        for(kay = 0; kay < 100; kay++) {

            calc\_follow[k][kay] = '!';

        }

    }

    point1 = 0;

    int land = 0;

    for(e = 0; e < count; e++)

    {

        ck = production[e][0];

        point2 = 0;

        xxx = 0;

        for(kay = 0; kay <= ptr; kay++)

            if(ck == donee[kay])

                xxx = 1;

        if (xxx == 1)

            continue;

        land += 1;

          follow(ck);

        ptr += 1;

        donee[ptr] = ck;

        printf(" Follow(%c) = { ", ck);

        calc\_follow[point1][point2++] = ck;

        for(i = 0 + km; i < m; i++) {

            int lark = 0, chk = 0;

            for(lark = 0; lark < point2; lark++)

            {

                if (f[i] == calc\_follow[point1][lark])

                {

                    chk = 1;

                    break;

                }

            }

            if(chk == 0)

            {

                printf("%c, ", f[i]);

                calc\_follow[point1][point2++] = f[i];

            }

        }

        printf(" }\n\n");

        km = m;

        point1++;

    }

}

 void follow(char c)

{

    int i, j;

    if(production[0][0] == c) {

        f[m++] = '$';

    }

    for(i = 0; i < 10; i++)

    {

        for(j = 2;j < 10; j++)

        {

            if(production[i][j] == c)

            {

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

                {

                    followfirst(production[i][j+1], i, (j+2));

                }

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

                {

                    follow(production[i][0]);

                }

            }

        }

    }

}

void findfirst(char c, int q1, int q2)

{

    int j;

    if(!(isupper(c))) {

        first[n++] = c;

    }

    for(j = 0; j < count; j++)

    {

        if(production[j][0] == c)

        {

            if(production[j][2] == '#')

            {

                if(production[q1][q2] == '\0')

                    first[n++] = '#';

                else if(production[q1][q2] != '\0'

                          && (q1 != 0 || q2 != 0))

                {

                           findfirst(production[q1][q2], q1, (q2+1));

                }

                else

                    first[n++] = '#';

            }

            else if(!isupper(production[j][2]))

            {

                first[n++] = production[j][2];

            }

            else

            {

                findfirst(production[j][2], j, 3);

            }

        }

    }

}

void followfirst(char c, int c1, int c2)

{

    int k;

          if(!(isupper(c)))

        f[m++] = c;

    else

    {

        int i = 0, j = 1;

        for(i = 0; i < count; i++)

        {

            if(calc\_first[i][0] == c)

                break;

        }

        while(calc\_first[i][j] != '!')

        {

            if(calc\_first[i][j] != '#')

            {

                f[m++] = calc\_first[i][j];

            }

            else

            {

                if(production[c1][c2] == '\0')

                {

                    follow(production[c1][0]);

                }

                else

                {

                    followfirst(production[c1][c2], c1, c2+1);

                }

            }

            j++;

        }

    }

}

**OUTPUT:**

Enter the no. of production:5

E->TE’

E’->+TE’/e

T->FT’

T’->\*FT’/e

F->(E) /id

First

First(E)={(,id}

First(E’)={+,e}

First(T)={(,id}

First(T’)={\*,e}

First(F)={(,id}

Follow

Follow(E)={),$}

Follow(E’)={),+,$}

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

Follow(T’)={\*,+,),$}

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

**RESULT:**

Thus the above computation of FIRST & FOLLOW program is successfully executed.

1. **CONSTRUCTION OF PREDICTIVE PARSING TABLE**

**AIM:**

To write a program for Predictive Parsing Table in C.

**AIM:**

To write a C program for the implementation of predictive parsing table

**ALGORITHM:**

1. start the program
2. Assign an input string and parsing table in for then G.
3. Set ip to point to the first symbol of to $
4. Repeat if X is a terminal of $ then if n=a,then pop X from the stack
5. Push Y into the stack with Y,on top
6. Output the production x->x,y
7. End else error until x=$
8. Terminate the program

**SOURCE CODE:\_**

#include<stdio.h>

#include<ctype.h>

#include<string.h>

void followfirst(char , int , int);

void findfirst(char , int , int);

void follow(char c);

int count,n=0;

char calc\_first[10][100];

char calc\_follow[10][100];

int m=0;

char production[10][10], first[10];

char f[10];

int k;

char ck;

int e;

int main(int argc,char \*\*argv)

{

int jm=0;

int km=0;

int i,choice;

char c,ch;

printf("How many productions ? :");

scanf("%d",&count);

printf("\nEnter %d productions in form A=B where A and B are grammar symbols :\n\n",count);

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

{

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

}

int kay;

char done[count];

int ptr = -1;

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

for(kay=0;kay<100;kay++){

calc\_first[k][kay] = '!';

}

}

int point1 = 0,point2,xxx;

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

{

c=production[k][0];

point2 = 0;

xxx = 0;

for(kay = 0; kay <= ptr; kay++)

if(c == done[kay])

xxx = 1;

if (xxx == 1)

continue;

findfirst(c,0,0);

ptr+=1;

done[ptr] = c;

printf("\n First(%c)= { ",c);

calc\_first[point1][point2++] = c;

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

int lark = 0,chk = 0;

   for(lark=0;lark<point2;lark++){

   if (first[i] == calc\_first[point1][lark]){

   chk = 1;

   break;

}

}

if(chk == 0){

   printf("%c, ",first[i]);

   calc\_first[point1][point2++] = first[i];

}

}

printf("}\n");

jm=n;

point1++;

}

printf("\n");

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

char donee[count];

ptr = -1;

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

for(kay=0;kay<100;kay++){

calc\_follow[k][kay] = '!';

}

}

point1 = 0;

int land = 0;

for(e=0;e<count;e++)

   {

ck=production[e][0];

point2 = 0;

xxx = 0;

for(kay = 0; kay <= ptr; kay++)

if(ck == donee[kay])

xxx = 1;

if (xxx == 1)

continue;

   land += 1;

follow(ck);

   ptr+=1;

donee[ptr] = ck;

   printf(" Follow(%c) = { ",ck);

   calc\_follow[point1][point2++] = ck;

   for(i=0+km;i<m;i++){

   int lark = 0,chk = 0;

   for(lark=0;lark<point2;lark++){

   if (f[i] == calc\_follow[point1][lark]){

   chk = 1;

   break;

}

}

if(chk == 0){

   printf("%c, ",f[i]);

   calc\_follow[point1][point2++] = f[i];

   }

   }

   printf(" }\n\n");

km=m;

point1++;

}

char ter[10];

for(k=0;k<10;k++){

ter[k] = '!';

}

int ap,vp,sid = 0;

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

for(kay=0;kay<count;kay++){

if(!isupper(production[k][kay]) && production[k][kay]!= '#' && production[k][kay] != '=' && production[k][kay] != '\0'){

vp = 0;

for(ap = 0;ap < sid; ap++){

if(production[k][kay] == ter[ap]){

vp = 1;

break;

}

}

if(vp == 0){

ter[sid] = production[k][kay];

sid ++;

}

}

}

}

ter[sid] = '$';

sid++;

printf("\n\t\t\t\t\t\t\t The LL(1) Parsing Table for the above grammer :-");

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

printf("\n\t\t\t=====================================================================================================================\n");

printf("\t\t\t\t|\t");

for(ap = 0;ap < sid; ap++){

printf("%c\t\t",ter[ap]);

}

printf("\n\t\t\t=====================================================================================================================\n");

char first\_prod[count][sid];

for(ap=0;ap<count;ap++){

int destiny = 0;

k = 2;

int ct = 0;

char tem[100];

while(production[ap][k] != '\0'){

if(!isupper(production[ap][k])){

tem[ct++] = production[ap][k];

tem[ct++] = '\_';

tem[ct++] = '\0';

k++;

break;

}

else{

int zap=0;

int tuna = 0;

for(zap=0;zap<count;zap++){

if(calc\_first[zap][0] == production[ap][k]){

for(tuna=1;tuna<100;tuna++){

if(calc\_first[zap][tuna] != '!'){

tem[ct++] = calc\_first[zap][tuna];

}

else

break;

}

break;

}

}

tem[ct++] = '\_';

}

k++;

}

int zap = 0,tuna;

for(tuna = 0;tuna<ct;tuna++){

if(tem[tuna] == '#'){

zap = 1;

}

else if(tem[tuna] == '\_'){

if(zap == 1){

zap = 0;

}

else

break;

}

else{

first\_prod[ap][destiny++] = tem[tuna];

}

}

}

char table[land][sid+1];

ptr = -1;

for(ap = 0; ap < land ; ap++){

for(kay = 0; kay < (sid + 1) ; kay++){

table[ap][kay] = '!';

}

}

for(ap = 0; ap < count ; ap++){

ck = production[ap][0];

xxx = 0;

for(kay = 0; kay <= ptr; kay++)

if(ck == table[kay][0])

xxx = 1;

if (xxx == 1)

continue;

else{

ptr = ptr + 1;

table[ptr][0] = ck;

}

}

for(ap = 0; ap < count ; ap++){

int tuna = 0;

while(first\_prod[ap][tuna] != '\0'){

int to,ni=0;

for(to=0;to<sid;to++){

if(first\_prod[ap][tuna] == ter[to]){

ni = 1;

}

}

if(ni == 1){

char xz = production[ap][0];

int cz=0;

while(table[cz][0] != xz){

cz = cz + 1;

}

int vz=0;

while(ter[vz] != first\_prod[ap][tuna]){

vz = vz + 1;

}

table[cz][vz+1] = (char)(ap + 65);

}

tuna++;

}

}

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

for(kay=0;kay<100;kay++){

if(calc\_first[k][kay] == '!'){

break;

}

else if(calc\_first[k][kay] == '#'){

int fz = 1;

while(calc\_follow[k][fz] != '!'){

char xz = production[k][0];

int cz=0;

while(table[cz][0] != xz){

cz = cz + 1;

}

int vz=0;

while(ter[vz] != calc\_follow[k][fz]){

vz = vz + 1;

}

table[k][vz+1] = '#';

fz++;

}

break;

}

}

}

for(ap = 0; ap < land ; ap++){

printf("\t\t\t   %c\t|\t",table[ap][0]);

for(kay = 1; kay < (sid + 1) ; kay++){

if(table[ap][kay] == '!')

printf("\t\t");

else if(table[ap][kay] == '#')

printf("%c=#\t\t",table[ap][0]);

else{

int mum = (int)(table[ap][kay]);

mum -= 65;

printf("%s\t\t",production[mum]);

}

}

printf("\n");

printf("\t\t\t---------------------------------------------------------------------------------------------------------------------");

printf("\n");

}

int j;

printf("\n\nPlease enter the desired INPUT STRING = ");

char input[100];

scanf("%s%c",input,&ch);

printf("\n\t\t\t\t\t===========================================================================\n");

printf("\t\t\t\t\t\tStack\t\t\tInput\t\t\tAction");

printf("\n\t\t\t\t\t===========================================================================\n");

int i\_ptr = 0,s\_ptr = 1;

char stack[100];

stack[0] = '$';

stack[1] = table[0][0];

while(s\_ptr != -1){

printf("\t\t\t\t\t\t");

int vamp = 0;

for(vamp=0;vamp<=s\_ptr;vamp++){

printf("%c",stack[vamp]);

}

printf("\t\t\t");

vamp = i\_ptr;

while(input[vamp] != '\0'){

printf("%c",input[vamp]);

vamp++;

}

printf("\t\t\t");

char her = input[i\_ptr];

char him = stack[s\_ptr];

s\_ptr--;

if(!isupper(him)){

if(her == him){

i\_ptr++;

printf("POP ACTION\n");

}

else{

printf("\nString Accepted by LL(1) Parser !!\n");

exit(0);

}

}

else{

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

if(ter[i] == her)

break;

}

char produ[100];

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

if(him == table[j][0]){

if (table[j][i+1] == '#'){

printf("%c=#\n",table[j][0]);

produ[0] = '#';

produ[1] = '\0';

}

else if(table[j][i+1] != '!'){

int mum = (int)(table[j][i+1]);

mum -= 65;

strcpy(produ,production[mum]);

printf("%s\n",produ);

}

else{

printf("\nString Not Accepted by LL(1) Parser !!\n");

exit(0);

}

}

}

int le = strlen(produ);

le = le - 1;

if(le == 0){

continue;

}

for(j=le;j>=2;j--){

s\_ptr++;

stack[s\_ptr] = produ[j];

}

}

}

printf("\n\t\t\t=======================================================================================================================\n");

if (input[i\_ptr] == '\0'){

printf("\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN ACCEPTED !!\n");

}

else

printf("\n\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN REJECTED !!\n");

printf("\t\t\t=======================================================================================================================\n");

}

void follow(char c)

{

int i ,j;

if(production[0][0]==c){

  f[m++]='$';

  }

  for(i=0;i<10;i++)

  {

   for(j=2;j<10;j++)

   {

    if(production[i][j]==c)

    {

     if(production[i][j+1]!='\0'){

followfirst(production[i][j+1],i,(j+2));

  }

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

      follow(production[i][0]);

}

    }

   }

  }

}

void findfirst(char c ,int q1 , int q2)

{

int j;

if(!(isupper(c))){

first[n++]=c;

}

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

{

if(production[j][0]==c)

{

if(production[j][2]=='#'){

if(production[q1][q2] == '\0')

first[n++]='#';

else if(production[q1][q2] != '\0' && (q1 != 0 || q2 != 0))

{

findfirst(production[q1][q2], q1, (q2+1));

}

else

first[n++]='#';

}

else if(!isupper(production[j][2])){

first[n++]=production[j][2];

}

else {

findfirst(production[j][2], j, 3);

}

}

}

}

void followfirst(char c, int c1 , int c2)

{

    int k;

    if(!(isupper(c)))

f[m++]=c;

else{

int i=0,j=1;

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

{

if(calc\_first[i][0] == c)

break;

}

while(calc\_first[i][j] != '!')

{

if(calc\_first[i][j] != '#'){

f[m++] = calc\_first[i][j];

}

else{

if(production[c1][c2] == '\0'){

follow(production[c1][0]);

}

else{

followfirst(production[c1][c2],c1,c2+1);

}

}

j++; } } }

**OUTPUT:**

How many productions ? :5

Enter 8 productions in form A=B where A and B are grammar symbols :

S=iEtSD

S=a

D=eS

D=$

E=b

**RESULT:**

Thus the Predictive Parser program is executed successfully.

1. **SHIFT REDUCE PARSING**

**AIM:**

To write a program for implementing Shift Reduce Parsing using C.

**ALGORITHM:**

1. Get the input expression and store it in the input buffer.
2. Read the data from the input buffer one at the time.
3. Using stack and push & pop operation shift and reduce symbols with respect to production rules available.
4. Continue the process till symbol shift and production rule reduce reaches the start symbol.
5. Display the Stack Implementation table with corresponding Stack actions with input symbols.

**PROGRAM:**

#include<stdio.h>

#include<string.h>

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

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

void check();

int main()

   {

      puts("GRAMMAR is E->E+E \n E->E\*E \n E->(E) \n E->id");

      puts("enter input string ");

      gets(a);

      c=strlen(a);

      strcpy(act,"SHIFT->");

      puts("stack \t input \t action");

      for(k=0,i=0; j<c; k++,i++,j++)

       {

         if(a[j]=='i' && a[j+1]=='d')

           {

              stk[i]=a[j];

              stk[i+1]=a[j+1];

              stk[i+2]='\0';

              a[j]=' ';

              a[j+1]=' ';

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

              check();

           }

         else

           {

              stk[i]=a[j];

              stk[i+1]='\0';

              a[j]=' ';

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

              check();

           }

       }

   }

void check()

   {

     strcpy(ac,"REDUCE TO E");

     for(z=0; z<c; z++)

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

         {

           stk[z]='E';

           stk[z+1]='\0';

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

           j++;

         }

     for(z=0; z<c; z++)

       if(stk[z]=='E' && stk[z+1]=='+' && stk[z+2]=='E')

         {

           stk[z]='E';

           stk[z+1]='\0';

           stk[z+2]='\0';

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

           i=i-2;

         }

     for(z=0; z<c; z++)

       if(stk[z]=='E' && stk[z+1]=='\*' && stk[z+2]=='E')

         {

           stk[z]='E';

           stk[z+1]='\0';

           stk[z+1]='\0';

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

           i=i-2;

         }

     for(z=0; z<c; z++)

       if(stk[z]=='(' && stk[z+1]=='E' && stk[z+2]==')')

         {

           stk[z]='E';

           stk[z+1]='\0';

           stk[z+1]='\0';

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

           i=i-2;

         }

   }

**OUTPUT :**

GRAMMAR is E->E+E

E->E\*E

E->(E)

E->id enter input string

id+id\*id+id

stack               input            action

$id              +id\*id+id$         SHIFT->id

$E               +id\*id+id$         REDUCE TO E

$E+               id\*id+id$         SHIFT->symbols

$E+id               \*id+id$         SHIFT->id

$E+E                \*id+id$         REDUCE TO E

$E                  \*id+id$         REDUCE TO E

$E\*                  id+id$         SHIFT->symbols

$E\*id                  +id$         SHIFT->id

$E\*E                   +id$         REDUCE TO E

$E                     +id$         REDUCE TO E

$E+                     id$         SHIFT->symbols

$E+id                     $         SHIFT->id

$E+E                      $         REDUCE TO E

$E                        $         REDUCE TO E

# RESULT:

## Thus the program for implementation of Shift Reduce parsing algorithm is executed and verified.

1. **COMPUATION OF LEADING AND TRAILING**

**AIM:**

To write a C program to implement the concept of operator precedence.

**ALGORITHM**:

1. Start the program.
2. Include the required header files and start the declaration of main method.
3. Declare the required variable and define the function for pushing and poping the characters.
4. The operators are displayed in column and row wise and stored it in a queue.
5. Using a switch case find the values of the operators.
6. Display the precedence of the operator and generate the code for precedence of operator for the given expression.
7. Compile and execute the program for the output.
8. Stop the program

**PROGRAM**

#include<stdlib.h>

#include<stdio.h>

#include<string.h>

// function f to exit from the loop

// if given condition is not true

void f()

{

    printf("Not operator grammar");

    exit(0);

}

void main()

{

    char grm[20][20], c;

    // Here using flag variable,

    // considering grammar is not operator grammar

    int i, n, j = 2, flag = 0;

    // taking number of productions from user

    scanf("%d", &n);

    for (i = 0; i < n; i++)

        scanf("%s", grm[i]);

    for (i = 0; i < n; i++) {

        c = grm[i][2];

        while (c != '\0') {

            if (grm[i][3] == '+' || grm[i][3] == '-'

                || grm[i][3] == '\*' || grm[i][3] == '/')

                flag = 1;

            else {

                flag = 0;

                f();

            }

            if (c == '$') {

                flag = 0;

                f();

            }

            c = grm[i][++j];

        }

    }

    if (flag == 1)

        printf("Operator grammar");

}

**Input :3**

**A=A\*A**

**B=AA**

**A=$**

**Output : Not operator grammar**

**Input :2**

**A=A/A**

**B=A+A**

**Output : Operator grammar**

**RESULT:**   Thus the C program implementation for operator precedence is executed and verified.

1. **COMPUATION OF LR(0) ITEMS**

**AIM:**

To write a C program to implement LR(0) Parser.

**ALGORITHM:**

1.Initialize the stack with the start state.

2. Read an input symbol

3.Using the top of the stack and the input symbol determine the next state.

4 If the next state is a stack state then stack the state get the next input symbol else if the next state is a reduce state

5.Output reduction number, k

6. POP  RHSk -1 states from the stack where RHSk is the right hand side of production k.

7. Set the next input symbol to the LHSk else if the next state is an accept state

8. Output valid sentence else output invalid sentence

**PROGRAM:**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

#include<string.h>

void push(char \*,int \*,char);

char stacktop(char \*);

void isproduct(char,char);

int ister(char);

int isnter(char);

int isstate(char);

void error();

void isreduce(char,char);

char pop(char \*,int \*);

void printt(char \*,int \*,char [],int);

void rep(char [],int);

struct action

{

char row[6][5];

};

const struct action A[12]={

{"sf","emp","emp","se","emp","emp"},

{"emp","sg","emp","emp","emp","acc"},

{"emp","rc","sh","emp","rc","rc"},

{"emp","re","re","emp","re","re"},

{"sf","emp","emp","se","emp","emp"},

{"emp","rg","rg","emp","rg","rg"},

{"sf","emp","emp","se","emp","emp"},

{"sf","emp","emp","se","emp","emp"},

{"emp","sg","emp","emp","sl","emp"},

{"emp","rb","sh","emp","rb","rb"},

{"emp","rb","rd","emp","rd","rd"},

{"emp","rf","rf","emp","rf","rf"}

};

struct gotol

{

char r[3][4];

};

const struct gotol G[12]={

{"b","c","d"},

{"emp","emp","emp"},

{"emp","emp","emp"},

{"emp","emp","emp"},

{"i","c","d"},

{"emp","emp","emp"},

{"emp","j","d"},

{"emp","emp","k"},

{"emp","emp","emp"},

{"emp","emp","emp"},

};

char ter[6]={'i','+','\*',')','(','$'};

char nter[3]={'E','T','F'};

char states[12]={'a','b','c','d','e','f','g','h','m','j','k','l'};

char stack[100];

int top=-1;

char temp[10];

struct grammar

{

char left;

char right[5];

};

const struct grammar rl[6]={

{'E',"e+T"},

{'E',"T"},

{'T',"T\*F"},

{'T',"F"},

{'F',"(E)"},

{'F',"i"},

};

void main()

{

char inp[80],x,p,dl[80],y,bl='a';

int i=0,j,k,l,n,m,c,len;

printf(" Enter the input :");

scanf("%s",inp);

len=strlen(inp);

inp[len]='$';

inp[len+1]='\0';

push(stack,&top,bl);

printf("\n stack \t\t\t input");

printt(stack,&top,inp,i);

do

{

x=inp[i];

p=stacktop(stack);

isproduct(x,p);

if(strcmp(temp,"emp")==0)

error();

if(strcmp(temp,"acc")==0)

break;

else

{

if(temp[0]=='s')

{

push(stack,&top,inp[i]);

push(stack,&top,temp[1]);

i++;

}

else

{

if(temp[0]=='r')

{

j=isstate(temp[1]);

strcpy(temp,rl[j-2].right);

dl[0]=rl[j-2].left;

dl[1]='\0';

n=strlen(temp);

for(k=0;k<2\*n;k++)

pop(stack,&top);

for(m=0;dl[m]!='\0';m++)

push(stack,&top,dl[m]);

l=top;

y=stack[l-1];

isreduce(y,dl[0]);

for(m=0;temp[m]!='\0';m++)

push(stack,&top,temp[m]);

}

}

}

printt(stack,&top,inp,i);

}while(inp[i]!='\0');

if(strcmp(temp,"acc")==0)

printf(" \n accept the input ");

else

printf(" \n do not accept the input ");

getch();

}

void push(char \*s,int \*sp,char item)

{

if(\*sp==100)

printf(" stack is full ");

else

{

\*sp=\*sp+1;

s[\*sp]=item;

}

}

char stacktop(char \*s)

{

char i;

i=s[top];

return i;

}

void isproduct(char x,char p)

{

int k,l;

k=ister(x);

l=isstate(p);

strcpy(temp,A[l-1].row[k-1]);

}

int ister(char x)

{

int i;

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

if(x==ter[i])

return i+1;

return 0;

}

int isnter(char x)

{

int i;

for(i=0;i<3;i++)

if(x==nter[i])

return i+1;

return 0;

}

int isstate(char p)

{

int i;

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

if(p==states[i])

return i+1;

return 0;

}

void error()

{

printf(" error in the input ");

exit(0);

}

void isreduce(char x,char p)

{

int k,l;

k=isstate(x);

l=isnter(p);

strcpy(temp,G[k-1].r[l-1]);

}

char pop(char \*s,int \*sp)

{

char item;

if(\*sp==-1)

printf(" stack is empty ");

else

{

item=s[\*sp];

\*sp=\*sp-1;

}

return item;

}

void printt(char \*t,int \*p,char inp[],int i)

{

int r;

printf("\n");

for(r=0;r<=\*p;r++)

rep(t,r);

printf("\t\t\t");

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

printf("%c",inp[r]);

}

void rep(char t[],int r)

{

char c;

c=t[r];

switch(c)

{

case 'a': printf("0");

break;

case 'b': printf("1");

break;

case 'c': printf("2");

break;

case 'd': printf("3");

break;

case 'e': printf("4");

break;

case 'f': printf("5");

break;

case 'g': printf("6");

break;

case 'h': printf("7");

break;

case 'm': printf("8");

break;

case 'j': printf("9");

break;

case 'k': printf("10");

break;

case 'l': printf("11");

break;

default :printf("%c",t[r]);

break;

}

}

Enter the input :i\*i+i\*i

 stack                   input

0                       i\*i+i\*i$

0i5                     \*i+i\*i$

0F3                     \*i+i\*i$

0T2                     \*i+i\*i$

0T2\*7                   i+i\*i$

0T2\*7i5                 +i\*i$

0T2\*7F10                +i\*i$

0E1                     +i\*i$

0E1+6                   i\*i$

0E1+6i5                 \*i$

0E1+6F3                 \*i$

0E1+6T9                 \*i$

0E1+6T9\*7               i$

0E1                     +i\*i$

0E1+6                   i\*i$

0E1+6i5                 \*i$

0E1+6F3                 \*i$

0E1+6T9                 \*i$

0E1+6T9\*7                       i$

0E1+6T9\*7i5                     $

0E1+6T9\*7F10                    $

0E1+6T9                 $

0E1                     $

 accept the input

**RESULT:**

Thus the    LR(0) Program  was executed and verified Successfully.

1. **INTERMEDIATE CODE GENERATION- POSTFIX, PREFIX**

**AIM:**

To write a C Program to Implement Intermediate code generation – Postfix notation

**ALGORITHM:**

**1.** Scan the infix expression from left to right.   
**2.** If the scanned character is an operand, output it.   
**3.**Else,   
**1** 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.   
**2** 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 you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)   
**4.** If the scanned character is an ‘(‘, push it to the stack.   
**5.** If the scanned character is an ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.   
**6.** Repeat steps 2-6 until infix expression is scanned.   
**7.** Print the output   
**8.**Pop and output from the stack until it is not empty.

**PROGRAM:**

#include<stdio.h>

int stack[20];

int top = -1;

void push(int x)

{

    stack[++top] = x;

}

int pop()

{

    return stack[top--];

}

int main()

{

    char exp[20];

    char \*e;

    int n1,n2,n3,num;

    printf("Enter the expression :: ");

    scanf("%s",exp);

    e = exp;

    while(\*e != '\0')

    {

        if(isdigit(\*e))

        {

            num = \*e - 48;

            push(num);

        }

        else

        {

            n1 = pop();

            n2 = pop();

            switch(\*e)

            {

            case '+':

            {

                n3 = n1 + n2;

                break;

            }

            case '-':

            {

                n3 = n2 - n1;

                break;

            }

            case '\*':

            {

                n3 = n1 \* n2;

                break;

            }

            case '/':

            {

                n3 = n2 / n1;

                break;

            }

            }

            push(n3);

        }

        e++;

    }

    printf("\nThe result of expression %s  =  %d\n\n",exp,pop());

    return 0;

}

OUTPUT:

Enter the expression :: 245+\*

The result of expression 245+\*  =  18

**RESULT:**

Thus, the C Program to Implement POSTFIX Notation in Intermediate Code Generation was executed successfully.

**11. Intermediate code generation – Quadruple, Triple, Indirect triple**

**AIM:**

To write a C Program to Implement Intermediate code generation – Quadruple, Triple, Indirect triple.

**ALGORITHM:**

step 1: Start.

Step 2: Enter the three address codes.

Step 3: If the code constitutes only memory operands they are moved to

the register and according to the operation the corresponding

assembly code is generated.

Step 4: If the code constitutes immediate operands then the code will have

a # symbol proceeding the number in code.

Step 5: If the operand or three address code involve pointers then the code

generated will constitute pointer register. This content may be

stored to other location or vice versa.

Step 6: Appropriate functions and other relevant display statements are

executed.

Step 7: Stop.

**SOURCE CODE**

#include"stdio.h"

#include"conio.h"

#include"string.h"

#include"stdlib.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];

void main()

{

 printf("\t\tINTERMEDIATE CODE GENERATION\n\n");

 printf("Enter the Expression :");

 scanf("%s",str);

 printf("The intermediate code:\t\tExpression\n");

 findopr();

 explore();

 getch();

}

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);

  getch();

  exit(0);

 }

 printf("\t%s :=  %c",right,str[k[--i].pos]);

 getch();

}

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:**

                INTERMEDIATE CODE GENERATION

Enter the Expression :x=a\*b+c/d

The intermediate code:          Expression

        Z := a\*b

x=Z+c/d

        Y := Z+c

        x=Y/d

        x := d

**RESULT:**  Thus the implementation of intermediate code generation in C has been successfully executed and output is verified.

**12.** **A SIMPLE CODE GENERATOR**

**AIM:**

To write a C program to implement a simple code generation

**ALGORITHM:**

1. Start the program

2. Open the source file and store the contents as quadruples.

3. Check for operators, in quadruples, if it is an arithmetic operator generator it or if assignment operator generates it, else perform unary minus on register C.

 4. Write the generated code into output definition of the file in outp.c

5. Print the output.

6. Stop the program.

**SOURCE CODE:**

#include<stdio.h>

#include<stdio.h>

//#include<conio.h>

#include<string.h>

void main()

{

char icode[10][30],str[20],opr[10];

int i=0;

//clrscr();

printf("\n Enter the set of intermediate code (terminated by exit):\n");

do

{

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

} while(strcmp(icode[i++],"exit")!=0);

printf("\n target code generation");

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

i=0;

do

{

strcpy(str,icode[i]);

switch(str[3])

{

case '+':

strcpy(opr,"ADD");

break;

case '-':

strcpy(opr,"SUB");

break;

case '\*':

strcpy(opr,"MUL");

break;

case '/':

strcpy(opr,"DIV");

break;

}

printf("\n\tMov %c,R%d",str[2],i);

printf("\n\t%s%c,R%d",opr,str[4],i);

printf("\n\tMov R%d,%c",i,str[0]);

}while(strcmp(icode[++i],"exit")!=0);

//getch();

}

**OUTPUT:**

Enter the set of intermediate code (terminated by exit):

t=a+b

x=t

exit

 target code generation

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

        Mov a,R0

        ADDb,R0

        Mov R0,t

        Mov t,R1

        ADDb,R1

        Mov R1,x

**RESULT:**

Thus the program to implement the code generation has been successfully executed.

* + 1. **IMPLEMENTATION OF DAG**

**AIM:** To write a C program to construct DAG (Directed Acyclic Graph)

**ALGORITHM:**

1. Start the program
2. Include all the header files
3. Check for postfix expression and construct the in order DAG representation
4. Print the output
5. Stop the program

**PROGRAM:**

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#define MIN\_PER\_RANK 1

#define MAX\_PER\_RANK 5

#define MIN\_RANKS 3

#define MAX\_RANKS 5

#define PERCENT 30

void main()

{

int i,j,k,nodes=0;

srand(time(NULL));

int ranks=MIN\_RANKS+(rand()%(MAX\_RANKS-MIN\_RANKS+1));

printf("DIRECTED ACYCLIC GRAPH\n");

for(i=1;i<ranks;i++)

{

int new\_nodes=MIN\_PER\_RANK+(rand()%(MAX\_PER\_RANK-MIN\_PER\_RANK+1));

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

for(k=0;k<new\_nodes;k++)

if((rand()%100)<PERCENT)

printf("%d->%d;\n",j,k+nodes);

nodes+=new\_nodes;

}

}

**OUTPUT:**

DIRECTED ACYCLIC GRAPH

0->6;

1->5;

1->8;

2->5;

2->7;

4->5;

4->7;

**...Program finished with exit code 3**

**Press ENTER to exit console.**

**RESULT:**

Thus the program for implementation of DAG has been successfully executed and output is verified.

**14. IMPLEMENTATION OF GLOBAL DATAFLOW ANALYSIS**

**AIM:** To write a C program to implement global datflow analysis.

**ALGORITHM:**

1. Get the maximum number of expressions from the user
2. Get the expressions as string from the user
3. Find the constant from the string and convert it into integer
4. Find the operator and do the necessary operation such as addition, multiplication, subtraction, division
5. Change the redundant variables to get the optimized code
6. Stop the program by exit

**PROGRAM CODE:**

//*Data Flow Analysis implementation*

#include<stdio.h>

#include<string.h>

#include<ctype.h>

void input();

void output();

void change(int p,int q,char \*res);

void constant();

void expression();

struct expr

{

char op[2],op1[5],op2[5],res[5];

int flag;

}arr[10];

int n;

int main()

{

int ch=0;

input();

constant();

expression();

output();

}

void input()

{

int i;

printf("\n\nEnter the maximum number of expressions:");

scanf("%d",&n);

printf("\nEnter the input : \n");

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

{

scanf("%s",arr[i].op);

scanf("%s",arr[i].op1);

scanf("%s",arr[i].op2);

scanf("%s",arr[i].res);

arr[i].flag=0;

}

}

void constant()

{

int i;

int op1,op2,res;

char op,res1[5];

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

{

if(isdigit(arr[i].op1[0]) && isdigit(arr[i].op2[0]))

{

op1=atoi(arr[i].op1);

op2=atoi(arr[i].op2);

op=arr[i].op[0];

switch(op)

{

case '+':

res=op1+op2;

break;

case '-':

res=op1-op2;

break;

case '\*':

res=op1\*op2;

break;

case '/':

res=op1/op2;

break;

}

sprintf(res1,"%d",res);

arr[i].flag=1;

change(i,i,res1);

}

}

}

void expression()

{

int i,j;

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

{

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

{

if(strcmp(arr[i].op,arr[j].op)==0)

{

if(strcmp(arr[i].op,"+")==0||strcmp(arr[i].op,"\*")==0)

{

if(strcmp(arr[i].op1,arr[j].op1)==0&&strcmp(arr[i].op2,arr[j].op2)==0 || strcmp(arr[i].op1,arr[j].op2)==0&&strcmp(arr[i].op2,arr[j].op1)==0)

{

arr[j].flag=1;

change(i,j,NULL);

}

}

else

{

if(strcmp(arr[i].op1,arr[j].op1)==0&&strcmp(arr[i].op2,arr[j].op2)==0)

{

arr[j].flag=1;

change(i,j,NULL);

}          }

}          }

}          }

void output()

{

int i=0;

printf("\nOptimized code is : ");

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

{

if(!arr[i].flag)

{

printf("\n%s %s %s %s\n",arr[i].op,arr[i].op1,arr[i].op2,arr[i].res);

}

}

}

void change(int p,int q,char \*res)

{

int i;

for(i=q+1;i<n;i++)

{

if(strcmp(arr[q].res,arr[i].op1)==0)

if(res == NULL)

strcpy(arr[i].op1,arr[p].res);

else

strcpy(arr[i].op1,res);

else if(strcmp(arr[q].res,arr[i].op2)==0)

if(res == NULL)

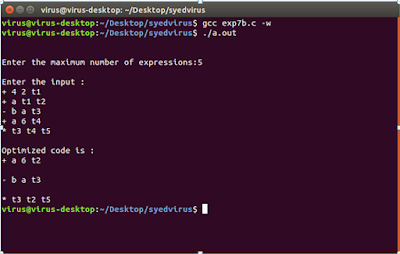
strcpy(arr[i].op2,arr[p].res);

else

strcpy(arr[i].op2,res);

}

}

**OUTPUT**

**RESULT:**

Thus the program to implement global data flow analysis has been executed and the output has been verified successfully.

.

**15. IMPLEMENTATION OF ONE STORAGE ALLOCATION STRATEGY - STACK**

**AIM:**

To write a program to Implement storage allocation strategy (Stack) in C.

**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 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

**PROGRAM:**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

#define size 5

struct stack

{

int s[size];

int top;

} st;

int stfull()

{

if (st.top >= size - 1)

return 1;

else

return 0;

}

void push(int item)

{

st.top++;

st.s[st.top] = item;

}

int stempty()

{

if (st.top == -1)

return 1;

else

return 0;

}

int pop()

{

int item;

item = st.s[st.top];

st.top--;

return (item);

}

void display()

{

int i;

if (stempty())

printf("\nStack Is Empty!");

else

{

for (i = st.top; i >= 0; i--)

printf("\n%d", st.s[i]);

}

}

int main()

{

int item, choice;

char ans;

st.top = -1;

printf("\n\tImplementation Of Stack");

do {

printf("\nMain Menu");

printf("\n1.Push \n2.Pop \n3.Display \n4.exit");

printf("\nEnter Your Choice");

scanf("%d", &choice);

switch (choice)

{

case 1:

printf("\nEnter The item to be pushed");

scanf("%d", &item);

if (stfull())

printf("\nStack is Full!");

else

push(item);

break;

case 2:

if (stempty())

printf("\nEmpty stack!Underflow !!");

else

{

item = pop();

printf("\nThe popped element is %d", item);

}

break;

case 3:

display();

break;

case 4:

goto halt;

}

printf("\nDo You want To Continue?");

ans = getche();

} while (ans == 'Y' || ans == 'y');

halt:

return 0;

}

**OUTPUT:**

Implementation Of Stack

Main Menu

1.Push

2.Pop

3.Display

4.exit

Enter Your Choice 1

Enter The item to be pushed 10

Do You want To Continue?y

Main Menu

1.Push

2.Pop

3.Display

4.exit

Enter Your Choice 1

Enter The item to be pushed 20

Do You want To Continue?y

Main Menu

1.Push

2.Pop

3.Display

4.exit

Enter Your Choice1

Enter The item to be pushed 30

Do You want To Continue?y

Main Menu

1.Push

2.Pop

3.Display

4.exit

Do You want To Continue?y

Main Menu

1.Push

2.Pop

3.Display

4.exit

Enter Your Choice 2

The popped element is 30

Do You want To Continue?y

Main Menu

1.Push

2.Pop

3.Display

4.exit

Enter Your Choice3

20

10

Do You want To Continue? n

**RESULT:**

Thus, the C program for Implementation of storage allocation strategy (Stack) has been executed and the output has been verified successfully.