Implementation of Lexical Analyzer

Experiment No : 1
Date : 19/01/2021

Aim	Implementation of Lexical Analyzer
Software Requirements	C Compiler

Algorithms :-

- 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.

Code

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
int isKeyword(char buffer[]){
char keywords[32][10] =
{"auto","break","case","char","const","continue","default","do","double","else","enum","extern","float","for","goto",
"if","int","long","register","return","short","signed",
"sizeof","static","struct","switch","typedef","union",
"unsigned","void","volatile","while"};
```

```
int i, flag = 0;
for(i = 0; i < 32; ++i){
if(strcmp(keywords[i], buffer) == 0){
flag = 1;
break;
}
}
return flag;
}
int main(){
char ch, buffer[15], operators[] = "+-*/%=";
FILE *fp;
int i,j=0;
fp = fopen("program.txt","r");
if(fp == NULL){
printf("error while opening the file\n");
exit(0);
}
while((ch = fgetc(fp)) != EOF){
 for(i = 0; i < 6; ++i){
 if(ch == operators[i])
 printf("%c is operator\n", ch);
 }

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

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

}
fclose(fp);
return 0;
}
```

Output

```
void is keyword
main is indentifier
a is indentifier
= is operator
a is indentifier
+ is operator
c is indentifier
include is indentifier

--------------------------------
Process exited after 0.2687 seconds with return value 0
Press any key to continue . . .
```

RESULT: **Thus, the C program to implement lexical analyzer has been executed and the output has been verified successfully.**

Conversion from Regular Expression to NFA

Experiment No : 2
Date : 27/01/2021

Aim	Conversion from Regular Expression to NFA
Software Requirements	C Compiler

Algorithms :-

- 1. Start the program
- 2. Include the header files.
- 3. Allocate memory for the variable by dynamic memory allocation function.
- 4. Use the file accessing functions to read the file.
- 5. Get the input file from the user.
- 6. Separate all the file contents as tokens and match it with the functions.
- 7. Define all the keywords in a separate file and name it as key.c
- 8. Define all the operators in a separate file and name it as open.c
- 9. Give the input program in a file and name it as input.c
- 10. Finally print the output after recognizing all the tokens.
- 11. Stop the program

Code

```
#include<stdio.h>
#include <string.h>
int main()
{
char m[20],t[10][10];
int n, i , j , r=0, c=0;
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);

}
```

Output

```
                                SIMULATION OF NFA
                                *******

Enter a regular expression:a+b*c

 0 1 2 3 4
_____________

   E       | 0
 E   E   E | 1
       b   | 2
     E   E | 3
           | 4

Start state: 0
Final state: 4
--------------------------------
Process exited after 7.534 seconds with return value 30
Press any key to continue . . .
```

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

Conversion from NFA to DFA

Experiment No : 3
Date : 03/02/2021

Aim	Conversion from NFA to DFA
Software Requirements	C Compiler

Algorithms :-

- 1. Start the program
- 2. Assign an input string terminated by end of file, DFA with start
- 3. The final state is assigned to F
- 4. Assign the state to S
- 5. Assign the input string to variable C
- 6. While C!=e of do S=move(s,c) C=next char
- 7. If it is in ten return yes else no
- 8. Stop the program

Code

```
#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include<math.h>

int n[11], i, j, c, k, h, l, h1, f, h2, temp1[12], temp2[12], count = 0, ptr = 0;
char a[20][20], s[5][8];
int tr[5][2], ecl[5][8], st[5], flag;

void ecls(int b[10], int x) {
  i = 0;
  k = -1;
  flag = 0;

  while (l < x) {
    n[++k] = b[l];
    i = b[l];
    h = k + 1;

    a:
```

```
      for (j = i; j <= 11; j++) {

        if (a[i][j] == 'e') {
          n[++k] = j;
        }

        if (j == 11 && h <= k) {
          i = n[h];
          h++;

          goto a;
        }
      }

    l++;
  }

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

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

      if (n[i] > n[j])

  {

    c = n[i];

    n[i] = n[j];

    n[j] = c;

  }

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

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

  {

    if (ecl[i][j] != n[j])

    {

      if (i < count)

      {

        i++;
```

```
        j = 0;
      } else
        goto b;
    } else if ((ecl[i][j] == n[j]) && (j == k))
    {
      tr[ptr][f] = st[i];
      flag = 1;
      break;
    }
  }
  b: if (flag == 0)
  {
    for (i = 0; i <= k; i++)
      ecl[count][i] = n[i];
    st[count] = count + 65;
    tr[ptr][f] = st[count];
    count++;
  }
}
void
mova(int g) {
  h1 = 0;
  for (i = 0; i < 7; i++)
  {
```

```
    if (ecl[g][i] == 3)
      temp1[h1++] = 4;
    if (ecl[g][i] == 8)
      temp1[h1++] = 9;
  }
  printf("\n move(%c,a):", st[g]);
  for (i = 0; i < h1; i++)
    printf("%d", temp1[i]);
  f = 0;
  ecls(temp1, h1);
}
void
movb(int g) {
  h2 = 0;
  for (i = 0; i < 7; i++)
  {
    if (ecl[g][i] == 5)
      temp2[h2++] = 6;
    if (ecl[g][i] == 9)
      temp2[h2++] = 10;
    if (ecl[g][i] == 10)
      temp2[h2++] = 11;
  }
  printf("move(%c,b):", st[g]);
```

```

  for (i = 0; i < h2; i++)

    printf("%d", temp2[i]);

  f = 1;

  ecls(temp2, h2);

}

int main() {
  printf("\n the no. of states in nfa (a/b)*abb are:11");

  for (i = 0; i <= 11; i++)
    for (j = 0; j <= 11; j++)
      a[i][j] = '\0';

  a[1][2] = 'e';
  a[1][6] = 'e';
  a[2][4] = 'e';
  a[2][8] = 'e';
  a[3][4] = 'a';
  a[4][8] = 'e';
  a[5][8] = 'b';
  a[5][10] = 'e';
  a[6][10] = 'e';
  a[7][4] = 'e';
  a[7][10] = 'e';
  a[9][10] = 'a';
  a[10][10] = 'b';
  a[11][10] = 'b';
  printf("\n the transmission table is as follows");
  printf("\n states  1 2 3 4 5 6 7 8 9 10 11");

  for (i = 1; i <= 11; i++) {
    printf("\n %d \t", i);

    for (j = 1; j <= 11; j++) {
      if (a[i][j]) {
        printf("%c", a[i][j]);
      } else {
        printf(" ");
      }
    }
  }
}
```

Output

```
 the no. of states in nfa (a/b)*abb are:11
 the transmission table is as follows
 states  1 2 3 4 5 6 7 8 9 10 11
 1       e   e
 2         e   e
 3         a
 4             e
 5             b e
 6               e
 7         e     e
 8
 9               a
 10              b
 11              b
--------------------------------
Process exited after 1.919 seconds with return value 12
Press any key to continue . . .
```

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

Elimination of Ambiguity, Left Recursion and Left Factoring

Experiment No : 4
Date :10/02/2021

Aim	Elimination of Ambiguity, Left Recursion and Left Factoring
Software Requirements	C Compiler

Algorithms :-

- 1. For each nonterminal :
  - a. Repeat until an iteration leaves the grammar unchanged:
  - b. For each rule , being a sequence of terminals and non terminals.
- 2. If begins with a nonterminal and :
  - a. Let be without its leading.
  - b. Remove the rule .
  - c. For each rule : i. Add the rule .
- 3. Remove direct left recursion for as described above.

Code

```
#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("%1s->%s", l, r);
   printf("%s", r);
   while (sscanf(r + consumed, "%[^|]s", temp) == 1 && consumed <= strlen(r))
   {
      if (temp[0] == l[0])
      {
         flag = 1;
         sprintf(productions[i++], "%s->%s%s'\0", l, temp + 1, l);
      }
      else
         sprintf(productions[i++], "%s'->%s%s'\0", l, temp, l);
      consumed += strlen(temp) + 1;
   }
   if (flag == 1)
   {
```

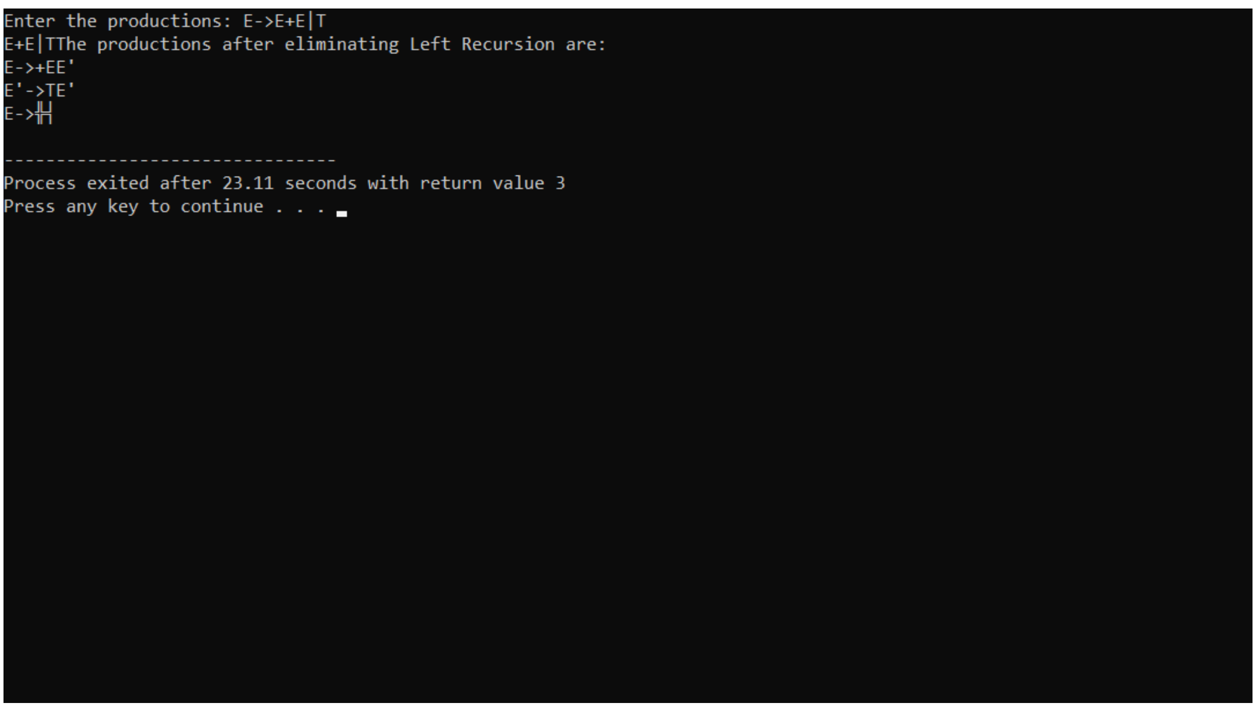
```
      sprintf(productions[i++], "%s->ε\0", l);

      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");
}
// Left factoring
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string ip, op1, op2, temp;
   int sizes[10] = {};
   char c;
   int n, j, l;
   cout << "Enter the Parent Non-Terminal : ";
   cin >> c;
   ip.push_back(c);
   op1 += ip + "\'->";
   op2 += ip + "\'\'->";;
   ip += "->";
   cout << "Enter the number of productions : ";
   cin >> n;
   for (int i = 0; i < n; i++)
   {
      cout << "Enter Production " << i + 1 << " : ";
      cin >> temp;
      sizes[i] = temp.size();
      ip += temp;
      if (i != n - 1)
         ip += "|";
   }

   cout << "Production Rule : " << ip << endl;
   char x = ip[3];
   for (int i = 0, k = 3; i < n; i++)
   {
      if (x == ip[k])
      {
         if (ip[k + 1] == '|')
         {
            op1 += "#";
            ip.insert(k + 1, 1, ip[0]);
            ip.insert(k + 2, 1, '\'');
```

```
            k += 4;
        }
        else
        {
            op1 += "|" + ip.substr(k + 1, sizes[i] - 1);
            ip.erase(k - 1, sizes[i] + 1);
        }
    }
    else
    {
        while (ip[k++] != '|');
    }
}

char y = op1[6];
for (int i = 0, k = 6; i < n - 1; i++)
{
    if (y == op1[k])
    {
        if (op1[k + 1] == '|')
        {
            op2 += "#";
            op1.insert(k + 1, 1, op1[0]);
            op1.insert(k + 2, 2, '\'');
            k += 5;
        }
        else
        {
            temp.clear();
            for (int s = k + 1; s < op1.length(); s++)
                temp.push_back(op1[s]);
            op2 += "|" + temp;
            op1.erase(k - 1, temp.length() + 2);
        }
    }
}

op2.erase(op2.size() - 1);
cout << "After Left Factoring : " << endl;
cout << ip << endl;
cout << op1 << endl;
cout << op2 << endl;
return 0;
}
```

Output

```
Enter the Parent Non-Terminal : L
Enter the number of productions : 4
Enter Production 1 : {
Enter Production 2 : iL
Enter Production 3 : (L)
Enter Production 4 : iL+L
Production Rule : L->{|iL|(L)|iL+L

--------------------------------
Process exited after 34.45 seconds with return value 3221225477
Press any key to continue . . .
```

Left Factoring



```
Enter the productions: E->E+E|T
E+E|TThe productions after eliminating Left Recursion are:
E->+EE'
E'->TE'
E->╟╢

--------------------------------
Process exited after 23.11 seconds with return value 3
Press any key to continue . . .
```

Left Recursion

RESULT: Thus, the C programs to eliminate left factoring and Left recursion has been executed and the output has been verified successfully

FIRST AND FOLLOW computation

Experiment No : 5
Date : 17/02/2021

Aim	FIRST AND FOLLOW computation
Software Requirements	C Compiler

Algorithms :-

- 1. For each terminal symbol Z FIRST([Z] « {Z}
- 2. repeat
- 3. For each production X —Y, ... ¥, if ¥, ... Y, are all nullable (orif k=0) then
  nullable| X | < true
- 4. For each i from 1 to k,each j from i+l to k
  if Y, ... Y, are all nullable (or if i=1) then
  FIRST[X | < FIRST[X |U FIRSTIY,]
  if Y, ... Y, areall nullable (or if i=k) then
  FOLLOW|Y,] <~ FOLLOW|Y,|u FOLLOW|X |
  if Y., ... Y, areall nullable (or if i+1=j) then
  FOLLOWLY, | <~ FOLLOW[Y,]1U FOLLOW[Y,]
- 5. until FIRST, FOLLOW and nullable no longer change.

Code

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

void FIRST(char[], char);
void addToResultSet(char[], char);
int numOfProductions;
char productionSet[10][10];
main()
{
    int i;
    char choice;
    char c;
    char result[20];
    printf("How many number of productions ? :");
    scanf(" %d", &numOfProductions);
    for (i = 0; i < numOfProductions; i++)//read production string eg: E=E+T
    {
        printf("Enter productions Number %d : ", i + 1);
        scanf(" %s", productionSet[i]);
```

```
    }

    do {
        printf("\n Find the FIRST of  :");
        scanf(" %c", &c);
        FIRST(result, c);     //Compute FIRST; Get Answer in 'result' array
        printf("\n FIRST(%c)= { ", c);
        for (i = 0; result[i] != '\0'; i++)
            printf(" %c ", result[i]);    //Display result
        printf("}\n");
        printf("press 'y' to continue : ");
        scanf(" %c", &choice);
    }

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

void FIRST(char *Result, char c)
{
    int i, j, k;
    char subResult[20];
    int foundEpsilon;
    subResult[0] = '\0';
    Result[0] = '\0';

    if (!(isupper(c)))
    {
        addToResultSet(Result, c);
        return;
    }

    for (i = 0; i < numOfProductions; i++)
    {
        if (productionSet[i][0] == c)
        {
            if (productionSet[i][2] == '$') addToResultSet(Result, '$');

            else
            {
                j = 2;
                while (productionSet[i][j] != '\0')
                {
                    foundEpsilon = 0;
                    FIRST(subResult, productionSet[i][j]);
                    for (k = 0; subResult[k] != '\0'; k++)
                        addToResultSet(Result, subResult[k]);
                    for (k = 0; subResult[k] != '\0'; k++)
                        if (subResult[k] == '$')
```

```
                        {
                            foundEpsilon = 1;
                            break;
                        }

      //No ε found, no need to check next element
                    if (!foundEpsilon)
                        break;
                    j++;
                }
            }
        }
    }

    return;
}

void addToResultSet(char Result[], char val)
{
    int k;
    for (k = 0; Result[k] != '\0'; k++)
        if (Result[k] == val)
            return;
    Result[k] = val;
    Result[k + 1] = '\0';
}
```

Output

```
How many number of productions ? :4
Enter productions Number 1 : e=TD
Enter productions Number 2 : D=+TD
Enter productions Number 3 : D=$
Enter productions Number 4 : T=FS

 Find the FIRST of  :e

 FIRST(e)= {  e }
press 'y' to continue : y

 Find the FIRST of  :D

 FIRST(D)= {  +  $ }
press 'y' to continue : n

--------------------------------
Process exited after 40.89 seconds with return value 110
Press any key to continue . . .
```

First

```
Enter the no.of productions: 4
 Enter 4 productions
Production with multiple terms should be give as separate productions
E=TD
D+=TD
D=$
T=FS
Find FOLLOW of -->E
FOLLOW(E) = { $  }
Do you want to continue(Press 1 to continue....)?1
Find FOLLOW of -->D
FOLLOW(D) = {  }
Do you want to continue(Press 1 to continue....)?
2

--------------------------------
Process exited after 72.53 seconds with return value 2
Press any key to continue . . .
```

Follow

RESULT: Thus, the C program for First and Follow Computation has been executed and the output has been verified successfully.

Predictive Parsing Table

Experiment No : 6
Date : 03/03/2021

Aim	Predictive Parsing Table
Software Requirements	C Compiler

Algorithms :-

- 1. In the beginning, the pushdown stack holds the start symbol of the grammar G.
- 2. At each step a symbol X is popped from the stack:
  if X is a terminal then it is matched with the lookahead and lookahead is advanced one step,
  if X is a nonterminal symbol, then using lookahead and a parsing table (implementing the FIRST sets) a production is chosen and its right-hand side is pushed into the stack.
- 3. This process repeats until the stack and the input string become null (empty).

Code

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

void 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;
      //clrscr();
    printf("enter the no. of coordinates\n");
    scanf("%d", &n);
    printf("enter the productions in a grammar\n");
    for (i = 0; i < n; i++)
        scanf("%s", st[i]);
    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';
        }
```

```
    }

    printf("first pos\n");
    for (i = 0; i < n; i++)
        printf("FIRS[%c]=%s\n", st[i][0], ft[i]);
    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: j++;
                }
                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';
    }

    printf("follow pos\n");
    for (i = 0; i < n; i++)
        printf("FOLLOW[%c]=%s\n", st[i][0], fol[i]);
    printf("\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", 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", 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", st[i][0], ft[a][b], fin[s]);
                        b++;
                    }
                }

                s++;
            }

            if (st[i][j] == '|')
                j++;
        }
    }

    getch();

}
```

Output

```
enter the no. of coordinates
2
enter the productions in a grammar
s->cc
c->ec|d
first pos
FIRS[s]=c
FIRS[c]=ed
follow pos
FOLLOW[s]=$
FOLLOW[c]=c

M[s,c]=s->cc
M[c,e]=c->ec
M[c,d]=c->d

--------------------------------
Process exited after 41.49 seconds with return value 13
Press any key to continue . . .
```

RESULT: Thus, the C program for Predictive Parsing Table has been executed and the output has been verified successfully..

Shift Reduce Parsing

Experiment No : 7
Date : 03/03/2021

Aim	Shift Reduce Parsing
Software Requirements	C Compiler

Algorithms :-

- Initialize the parse stack to contain a single state s0, where s0 is the distinguished initial state of the parser.
- 2. Use the state s on top of the parse stack and the current lookahead t to consult the action table entry action[s][t]:

  If the action table entry is shift s' then push state s' onto the stack and advance the input so that the lookahead is set to the next token.

  If the action table entry is reduce r and rule r has m symbols in its RHS, then pop m symbols off the parse stack. Let s' be the state now revealed on top of the parse stack and N be the LHS nonterminal for rule r. Then consult the goto table and push the state given by goto[s'][N] onto the stack. The lookahead token is not changed by this step.

  If the action table entry is accept, then terminate the parse with success.
  If the action table entry is error, then signal an error.
- 3. Repeat step (2) until the parser terminates.

Code

```
#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()
{
      //clrscr();
    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();
        }
    }

    getch();
    return 0;
}

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
a+b-c
stack    input   action

$a       +b-c$  SHIFT->symbols
$a+       b-c$  SHIFT->symbols
$a+b       -c$  SHIFT->symbols
$a+b-       c$  SHIFT->symbols
$a+b-c       $  SHIFT->symbols
```

RESULT: Thus, the C program for Shift Reduce Parsing been executed and the output has been verified successfully..

Computation of LEADING AND TRAILING

Experiment No : 8
Date : 09/03/2021

Aim	Computation of LEADING AND TRAILING
Software Requirements	C Compiler

Algorithms :-

- 1. 'a' is in LEADING(A) is A→ γaδ where γ is ε or any non-terminal.
- 2. If 'a' is in LEADING(B) and A→B, then 'a' is in LEADING(A).
- 3. 'a' is in TRAILING(A) is A→ γaδ where δ is ε or any non-terminal.
- 4. If 'a' is in TRAILING(B) and A→B, then 'a' is in TRAILING(A)

Code

```
#include <conio.h>

#include <stdio.h>

char arr[18][3] = {
  {
    'E',
    '+',
    'F'
  },
  {
    'E',
    '*',
    'F'
  },
  {
    'E',
    '(',
    'F'
  },
  {
    'E',
    ')',
    'F'
  },
  {
    'E',
    'i',
```

```
      'F'
   },
   {
      'E',
      '$',
      'F'
   },
   {
      'F',
      '+',
      'F'
   },
   {
      'F',
      '*',
      'F'
   },
   {
      'F',
      '(',
      'F'
   },
   {
      'F',
      ')',
      'F'
   },
   {
      'F',
      'i',
      'F'
   },
   {
      'F',
      '$',
      'F'
   },
   {
      'T',
      '+',
      'F'
   },
   {
      'T',
      '*',
      'F'
   },
```

```
  {
    'T',
    '(',
    'F'
  },
  {
    'T',
    ')',
    'F'
  },
  {
    'T',
    'i',
    'F'
  },
  {
    'T',
    '$',
    'F'
  },
};
char prod[6] = "EETTFF";
char res[6][3] = {
  {
    'E',
    '+',
    'T'
  },
  {
    'T',
    '\0'
  },
  {
    'T',
    '*',
    'F'
  },
  {
    'F',
    '\0'
  },
  {
    '(',
    'E',
    ')'
  },
  {
```

```
    'i',
    '\0'
  },
};
char stack[5][2];
int top = -1;
void install(char pro, char re) {
  int i;
  for (i = 0; i < 18; ++i) {
    if (arr[i][0] == pro && arr[i][1] == re) {
      arr[i][2] = 'T';
      break;
    }
  }

  ++top;
  stack[top][0] = pro;
  stack[top][1] = re;
}

void main() {
  int i = 0, j;
  char pro, re, pri = ' ';
  //clrscr();
  for (i = 0; i < 6; ++i) {
    for (j = 0; j < 3 && res[i][j] != '\0'; ++j) {
      if (res[i][j] ==
        '+' || res[i][j] == '*' || res[i][j] == '(' || res[i][j] == ')' || res[i][j] == 'i' || res[i][j] == '$') {
        install(prod[i], res[i][j]);
        break;
      }
    }
  }

  while (top >= 0) {
    pro = stack[top][0];
    re = stack[top][1];
    --top;
    for (i = 0; i < 6; ++i) {
      if (res[i][0] == pro && res[i][0] != prod[i]) {
        install(prod[i], re);
      }
    }
  }

  for (i = 0; i < 18; ++i) {
```

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

  for (j = 0; j < 3; ++j)
    printf("%c\t", arr[i][j]);
}

getch();
//clrscr();
printf("\n\n");
for (i = 0; i < 18; ++i) {
  if (pri != arr[i][0]) {
    pri = arr[i][0];
    printf("\n\t%c -> ", pri);
  }

  if (arr[i][2] == 'T')
    printf("%c ", arr[i][1]);
}

getch();

}
```

Output

```
        E       +       T
        E       *       T
        E       (       T
        E       )       F
        E       i       T
        E       $       F
        F       +       F
        F       *       F
        F       (       T
        F       )       F
        F       i       T
        F       $       F
        T       +       F
        T       *       T
        T       (       T
        T       )       F
        T       i       T
        T       $       F


        E -> + * ( i
        F -> ( i
        T -> * ( i
--------------------------------
Process exited after 7.696 seconds with return value 42
Press any key to continue . . .
```

RESULT: Thus, the C program for Computation of Leading and Trailing has been executed and the output has been verified successfully

Computation of LR(0) items

Experiment No : 9
Date : 17/03/2021

Aim	Computation of LR(0) items
Software Requirements	C Compiler

Algorithms :-

- 1. Initialize the stack with the start state.
- 2. Read an input symbol
- 3. while true do
- 4. Using the top of the stack and the input symbol determine the next state.
- 5. If the next state is a stack state
- 6. Then
- 7. stack the state
- 8. get the next input symbol
- 9. else if the next state is a reduce state
- 10.then
- 11.output reduction number, k
- 12.pop RHSk -1 states from the stack where RHSk is the right hand side of production k.
- 13.set the next input symbol to the LHSk
- 14.else if the next state is an accept state
- 15.then
- 16.output valid sentence
- 17.return
- 18.else
- 19.output invalid sentence
- 20.return

Code

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

int axn[][6][2]={
 {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
 {{-1,-1},{100,6},{-1,-1},{-1,-1},{-1,-1},{102,102}},
 {{-1,-1},{101,2},{100,7},{-1,-1},{101,2},{101,2}},
 {{-1,-1},{101,4},{101,4},{-1,-1},{101,4},{101,4}},
```

```
 {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
 {{100,5},{101,6},{101,6},{-1,-1},{101,6},{101,6}},
 {{100,5},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1}},
 {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
 {{-1,-1},{100,6},{-1,-1},{-1,-1},{100,11},{-1,-1}},
 {{-1,-1},{101,1},{100,7},{-1,-1},{101,1},{101,1}},
 {{-1,-1},{101,3},{101,3},{-1,-1},{101,3},{101,3}},
 {{-1,-1},{101,5},{101,5},{-1,-1},{101,5},{101,5}}
};

int gotot[12][3]={1,2,3,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,2,3,-1,-1,-1,-1,
 9,3,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};

int a[10];
char b[10];
int top=-1,btop=-1,i;
void push(int k)
{
    if(top<9)
    a[++top]=k;
}
void pushb(char k)
{
    if(btop<9)
    b[++btop]=k;
}
char TOS()
{
    return a[top];
}
void pop()
{
    if(top>=0)
    top--;
}
void popb()
{
    if(btop>=0)
    b[btop--]='\0';
}
void display()
{
    for(i=0;i<=top;i++)
    printf("%d%c",a[i],b[i]);
}
void display1(char p[],int m)
{
```

```
    int l;

    printf("\t\t");
    for(l=m;p[l]!='\0';l++)
    printf("%c",p[l]);
    printf("\n");
}
void error()
{
    printf("\n\nSyntax Error");
}
void reduce(int p)
{
    int len,k,ad;
    char src,*dest;
    switch(p)
    {
        case 1:
            dest="E+T";
            src='E';
            break;
        case 2:
            dest="T";
            src='E';
            break;
        case 3:
            dest="T*F";
            src='T';
            break;
        case 4:
            dest="F";
            src='T';
            break;
        case 5:
            dest="(E)";
            src='F';
            break;
        case 6:
            dest="i";
            src='F';
            break;
        default:
            dest="\0";
            src='\0';
            break;
    }
    for(k=0;k<strlen(dest);k++)
    {
```

```
        pop();
        popb();
    }
    pushb(src);
    switch(src)
    {
        case 'E':
            ad=0;
            break;
        case 'T':
            ad=1;
            break;
        case 'F':
            ad=2;
            break;
        default:
            ad=-1;
            break;
    }
    push(gotot[TOS()][ad]);
}
int main()
{
    int j,st,ic;
    char ip[20]="\0",an;

    printf("Enter any String :- ");
    gets(ip);
    push(0);
    display();
    printf("\t%s\n",ip);
    for(j=0;ip[j]!='\0';)
    {
        st=TOS();
        an=ip[j];
        if(an>='a'&an<='z')
            ic=0;
        else if(an=='+')
            ic=1;
        else if(an=='*')
            ic=2;
        else if(an=='(')
            ic=3;
        else if(an==')')
            ic=4;
        else if(an=='$')
            ic=5;
```

```
        else
        {
            error();
            break;
        }
        if(axn[st][ic][0]==100)
        {
            pushb(an);
            push(axn[st][ic][1]);
            display();
            j++;
            display1(ip,j);
        }
        if(axn[st][ic][0]==101)
        {
            reduce(axn[st][ic][1]);
            display();
            display1(ip,j);
        }
        if(axn[st][ic][1]==102)
        {
            printf("Given String is Accepted");
            break;
        }
    }
    getch();
    return 0;
}
```

Output

```
Enter any String :- a+b*c
0       a+b*c
0a5             +b*c
0F3             +b*c
0T2             +b*c
0E1             +b*c
0E1+6           b*c
0E1+6b5                 *c
0E1+6F3                 *c
0E1+6T9                 *c
0E1+6T9*7               c
0E1+6T9*7c5

--------------------------------
Process exited after 10.86 seconds with return value 0
Press any key to continue . . .
```

RESULT: Thus, the C program for Computation of LR(0) items has been executed and the output has been verified successfully...

Intermediate code generation – Postfix, Prefix

Experiment No : 10

Date : 25/03/2021

Aim	Intermediate code generation – Postfix, Prefix
Software Requirements	C Compiler

Algorithms :-

- 1.Start.
- 2. Enter the three address codes.
- 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.
- 4. If the code constitutes immediate operands then the code will have a # symbol proceeding the number in code.
- 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.
- 6. Appropriate functions and other relevant display statements are executed.
- 7. Stop.

Code

```
#include <stdio.h>
#include <conio.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;
```

```
    //clrscr();

    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());
    getch();
    return 0;
}
```

Output

```
Enter the expression :: 69*+420

The result of expression 69*+420 = 0


--------------------------------
Process exited after 10.11 seconds with return value 0
Press any key to continue . . .
```

RESULT: The Intermediate Code is Generated for Postfix and Prefix Operations.

Intermediate code generation – Quadruple, Triple, Indirect triple

Experiment No : 11
Date : 31/03/2021

Aim	Intermediate code generation – Quadruple, Triple, Indirect triple
Software Requirements	C Compiler

Algorithms :-

- 1. Start.
- 2. Enter the three address codes.
- 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.
- 4. If the code constitutes immediate operands then the code will have a # symbol proceeding the number in code.
- 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.
- 6. Appropriate functions and other relevant display statements are executed.
- 7. Stop

Code

```
#include"stdio.h"
#include"conio.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];

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 : w:a*b+c/d-e/f+g*h
The intermediate code:          Expression
        Z := c/d                w:a*b+Z-e/f+g*h
        Y := e/f                w:a*b+Z-Y+g*h
        X := a*b                w:X+Z-Y+g*h
        W := g*h                w:X+Z-Y+W
        V := X+Z                w:V-Y+W
        U := Y+W                w:V-U
        T := V-U                w:T
        w := T
```

RESULT: Thus, the C program for Intermediate code generation – Quadruple, Triple, Indirect triple has been executed and the output has been verified successfully.

A simple code Generator

Experiment No : 12
Date : 17/04/2021

Aim	A simple code Generator
Software Requirements	C Compiler

Algorithms :-

- 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.

Code

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

void main()
{
    char icode[10][30], str[20], opr[10];
    int i = 0;
    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);
}
```

Output

```
 Enter the set of intermediate code (terminated by exit):
y = a + b
x = y
exit

 target code generation
************************
        Mov  ,R0
        @ ,R0
        Mov R0,y
        Mov  ,R1
        @ ,R1
        Mov R1,=
        Mov  ,R2
        @ ,R2
        Mov R2,a
        Mov  ,R3
        @ ,R3
        Mov R3,+
        Mov  ,R4
        @ ,R4
        Mov R4,b
        Mov  ,R5
        @ ,R5
        Mov R5,x
        Mov  ,R6
        @ ,R6
        Mov R6,=
        Mov  ,R7
        @ ,R7
        Mov R7,y
--------------------------------
Process exited after 32.27 seconds with return value 0
Press any key to continue . . .
```

RESULT: Target Code is Generated for required Operations.

Implementation of DAG

Experiment No : 13

Date : 21/04/2021

Aim	Implementation of DAG
Software Requirements	C Compiler

Algorithms :-

- 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 progra

Code

```
#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->4;
1->3;
1->4;
2->4;
0->8;
2->8;
2->9;
3->8;
3->9;
5->9;
6->9;
0->10;
3->11;
4->10;
5->10;
5->12;
7->12;
8->11;

--------------------------------
Process exited after 2.086 seconds with return value 5
Press any key to continue . . .
```

RESULT: Thus, the C program for Implementation of DAG has been executed and the output has been verified successfully.

IMPLEMENTATION OF GLOBAL DATA FLOW ANALYSIS

Experiment No : 14
Date :

Aim	To study about Global Data Flow Analysis in Complier Design
Software Requirements	----

GLOBAL DATA FLOW ANALYSIS:

In order to do code optimization and a good job of code generation , compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph. A compiler could take advantage of “reaching definitions” , such as knowing
where a variable like debug was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.

Data-flow information can be collected by setting up and solving systems of equations of the form :

out [S] = gen [S] U (in [S] - kill [S])

This equation can be read as “ the information at the end of a statement is either generated within the statement , or enters at the beginning and is not killed as control flows through the statement.” Such equations are called data-flow equation.

1. The details of how data-flow equations are set and solved depend on three factors. The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[S] in terms of in[S], we need to proceed backwards and define in[S] in terms of out[S].

2. Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write out[s] we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.

3. There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

Points and Paths:
Within a basic block, we talk of the point between two adjacent statements, as well as the point before the first statement and after the last. Thus, block B1 has four points: one before any of the assignments and one after each of the three assignments.

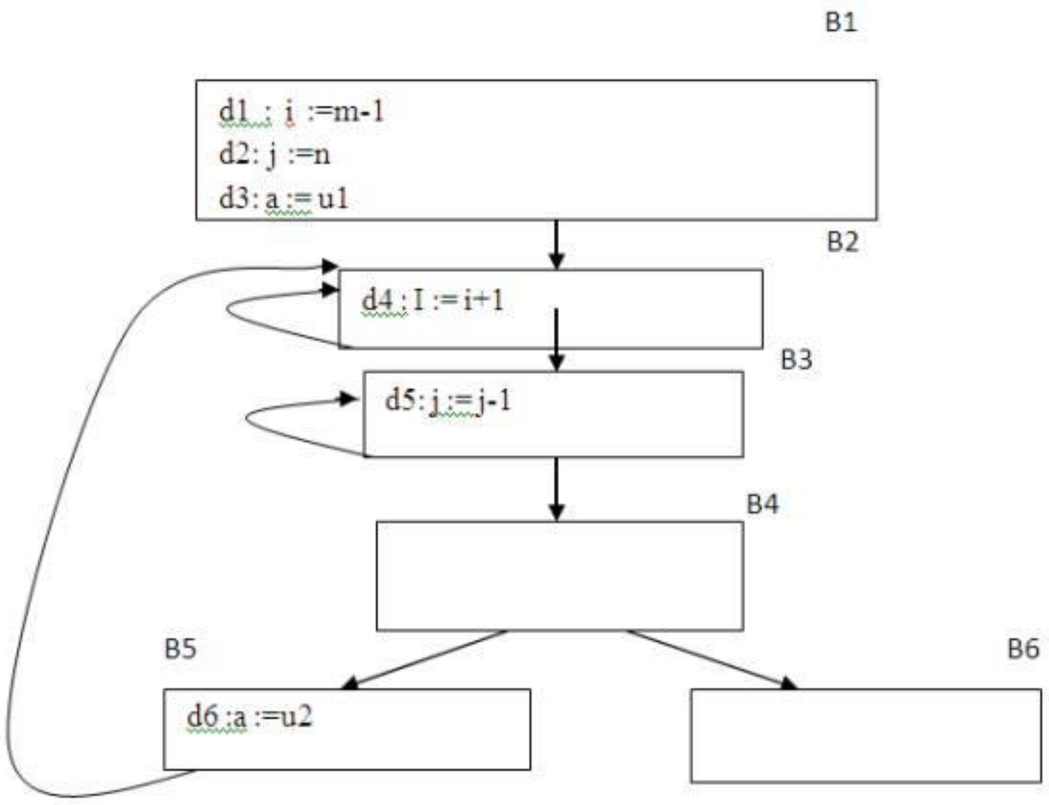


Fig. 5.6 A flow graph

Now let us take a global view and consider all the points in all the blocks. A path from p1 to pn is a sequence of points p1, p2,….,pn such that for each i between 1 and n-1, either

1. Pi is the point immediately preceding a statement and pi+1 is the point immediately following that statement in the same block, or
2. Pi is the end of some block and pi+1 is the beginning of a successor block.

Reaching definitions:

A definition of variable x is a statement that assigns, or may assign, a value to x. The most common forms of definition are assignments to x and statements that read a value from an i/o device and store it in x. These statements certainly define a value for x, and they are referred to as unambiguous definitions of x. There are certain kinds of statements that may define a value for x; they are called ambiguous definitions.

The most usual forms of ambiguous definitions of x are:

1. A call of a procedure with x as a parameter or a procedure that can access x because x is in the scope of the procedure.
2. An assignment through a pointer that could refer to x. For example, the assignment *q:=y is a definition of x if it is possible that q points to x. we must assume that an assignment through a pointer is a definition of every variable.

We say a definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not “killed” along that path. Thus a point can be reached by an unambiguous definition and an ambiguous definition of the appearing later along one path.

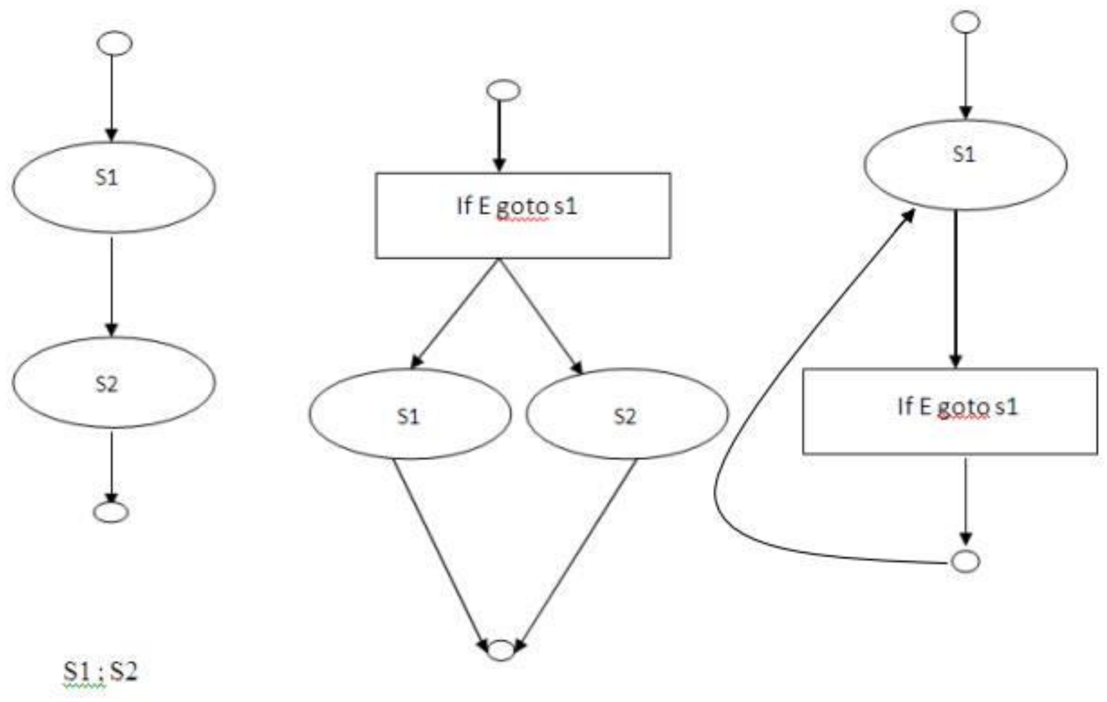


Fig. 5.7 Some structured control constructs

Data-flow analysis of structured programs:

Flow graphs for control flow constructs such as do-while statements have a useful property: there is a single beginning point at which control enters and a single end point that control leaves from when execution of the statement is over. We exploit this property when we talk of the definitions reaching the beginning and the end of statements with the following syntax.

S->id: = E| S; S | if E then S else S | do S while E
E->id + id| id

Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restricted forms.

We define a portion of a flow graph called a region to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the header. The portion of flow graph corresponding to a statement S is a region that obeys the further restriction that control can flow to just one outside block when it leaves the region.

We say that the beginning points of the dummy blocks at the statement's region are the beginning and end points, respective equations are inductive, or syntax-directed, definition of the sets in[S], out[S], gen[S], and kill[S] for all statements S. gen[S] is the set of definitions "generated" by S while kill[S] is the set of definitions that never reach the end of S.

• Consider the following data-flow equations for reaching definitions :

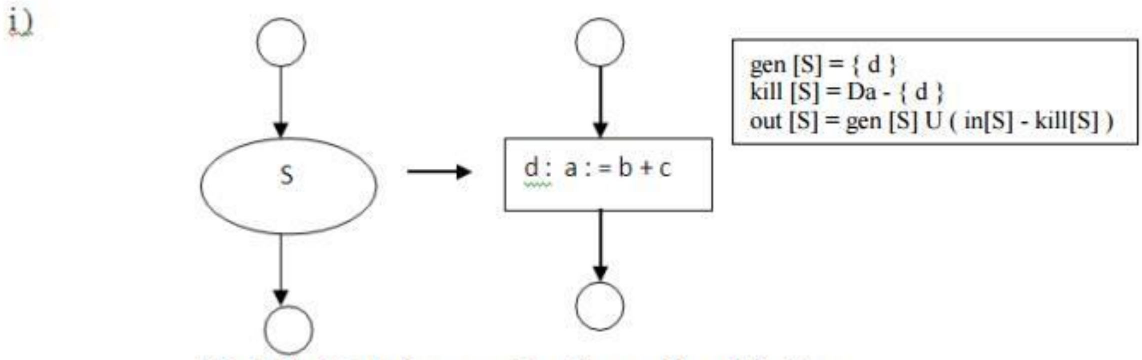


Fig. 5.8 (a) Data flow equations for reaching definitions

Fig. 5.8 (a) Data flow equations for reaching definitions

Observe the rules for a single assignment of variable a. Surely that assignment is a definition of a, say d. Thus

gen[S]={d}

On the other hand, d "kills" all other definitions of a, so we write

Kill[S] = Da - {d}

Where, Da is the set of all definitions in the program for variable a.

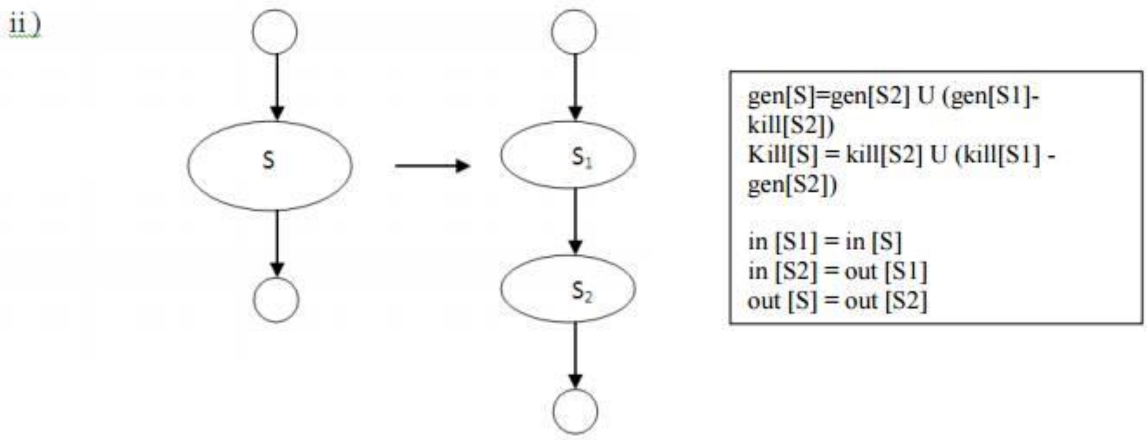


Fig. 5.8 (b) Data flow equations for reaching definitions

Under what circumstances is definition d generated by S=S1; S2? First of all, if it is generated by S2, then it is surely generated by S. if d is generated by S1, it will reach the end of S provided it is not killed by S2. Thus, we write

gen[S]=gen[S2] U (gen[S1]-kill[S2])
Similar reasoning applies to the killing of a definition, so we have

Kill[S] = kill[S2] U (kill[S1] - gen[S2])

Conservative estimation of data-flow information:

There is a subtle miscalculation in the rules for gen and kill. We have made the assumption that the conditional expression E in the if and do statements are "uninterpreted"; that
is, there exists inputs to the program that make their branches go either way.

We assume that any graph-theoretic path in the flow graph is also an execution path, i.e., a path that is executed when the program is run with least one possible input. When we compare the computed gen with the "true" gen we discover that the true gen is always a subset of the computed gen. on the other hand, the true kill is always a superset of the computed kill.

These containments hold even after we consider the other rules. It is natural to wonder whether these differences between the true and computed gen and kill sets present a serious obstacle to data-flow analysis. The answer lies in the use intended for these data.

Overestimating the set of definitions reaching a point does not seem serious; it merely stops us from doing an optimization that we could legitimately do. On the other hand, underestimating the set of definitions is a fatal error; it could lead us into making a change in the program that changes what the program computes. For the case of reaching definitions, then, we call a set of definitions safe or conservative if the estimate is a superset of the true set of reaching definitions. We call the estimate unsafe, if it is not necessarily a superset of the truth. Returning now to the implications of safety on the estimation of gen and kill for reaching definitions, note that our discrepancies, supersets for gen and subsets for kill are both in the safe direction. Intuitively, increasing gen adds to the set of definitions that can reach a point, and cannot prevent a definition from reaching a place that it truly reached. Decreasing kill can only increase the set of definitions reaching any given point.

Computation of in and out:

Many data-flow problems can be solved by synthesized translation to compute gen and kill. It can be used, for example, to determine computations. However, there are other kinds of data-flow information, such as the reaching-definitions problem. It turns out that in is an inherited attribute, and out is a synthesized attribute depending on in. we intend that in[S] be the set of definitions reaching the beginning of S, taking into account the flow of control throughout the entire program, including statements outside of S or within which S is nested.

The set out[S] is defined similarly for the end of s. it is important to note the distinction between out[S] and gen[S]. The latter is the set of definitions that reach the end of S without following paths outside S. Assuming we know in[S] we compute out by equation, that is

Out[S] = gen[S] U (in[S] - kill[S])

Considering cascade of two statements S1; S2, as in the second case. We start by observing in[S1]=in[S]. Then, we recursively compute out[S1], which gives us in[S2], since a definition reaches the beginning of S2 if and only if it reaches the end of S1. Now we can compute out[S2], and this set is equal to out[S].
Consider the if-statement. we have conservatively assumed that control can follow either branch, a definition reaches the beginning of S1 or S2 exactly when it reaches the beginning of S. That is,

in[S1] = in[S2] = in[S]

If a definition reaches the end of S if and only if it reaches the end of one or both substatements; i.e,

out[S]=out[S1] U out[S2]

Representation of sets:

Sets of definitions, such as gen[S] and kill[S], can be represented compactly using bit vectors. We assign a number to each definition of interest in the flow graph. Then bit vector representing a set of definitions will have 1 in position I if and only if the definition numbered I is in the set.

The number of definition statement can be taken as the index of statement in an array holding pointers to statements. However, not all definitions may be of interest during global data-flow analysis. Therefore the number of definitions of interest will typically be recorded in a separate table.

A bit vector representation for sets also allows set operations to be implemented efficiently. The union and intersection of two sets can be implemented by logical or and logical and, respectively, basic operations in most systems-oriented programming languages. The difference A-B of sets A and B can be implement complement of B and then using logical and to compute A

Local reaching definitions:

Space for data-flow information can be traded for time, by saving information only at certain points and, as needed, recomputing information at intervening points. Basic blocks are usually treated as a unit during global flow analysis, with attention restricted to only those points that are the beginnings of blocks.

Since there are usually many more points than blocks, restricting our effort to blocks is a significant savings. When needed, the reaching definitions for all points in a block can be calculated from the reaching definitions for the beginning of a block.

Use-definition chains:
It is often convenient to store the reaching definition information as" use-definition chains" or "ud-chains", which are lists, for each use of a variable, of all the definitions that reaches that use. If a use of variable a in block B is preceded by no unambiguous definition of a, then ud-chain for that use of a is the set of definitions in in[B] that are definitions of a.in addition, if there are ambiguous definitions of a ,then all of these for which no unambiguous definition of a lies between it and the use of a are on the ud-chain for this use of a.

Evaluation order:

The techniques for conserving space during attribute evaluation, also apply to the computation of data-flow information using specifications. Specifically, the only constraint on the evaluation order for the gen, kill, in and out sets for statements is that imposed by dependencies between these sets. Having chosen an evaluation order, we are free to release the space for a set after all uses of it have occurred. Earlier circular dependencies between attributes were not allowed, but we have seen that data-flow equations may have circular dependencies.

General control flow:

Data-flow analysis must take all control paths into account. If the control paths are evident from the syntax, then data-flow equations can be set up and solved in a syntax directed manner. When programs can contain goto statements or even the more disciplined break and continue statements, the approach we have taken must be modified to take the actual control paths into account.

Several approaches may be taken. The iterative method works arbitrary flow graphs. Since the flow graphs obtained in the presence of break and continue statements are reducible, such constraints can be handled systematically using the interval-based methods. However, the syntax-directed approach need not be abandoned when break and continue statements are allowed.

RESULT:

Thus, Implementation of Global Data Flow Analysis has been studied.

Implement any one storage allocation strategies (heap, stack, static)

Experiment No : 15
Date : 28/04/2021

Aim	Implement any one storage allocation strategies (heap, stack, static)
Software Requirements	C Compiler

Algorithms :-

- 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

Code

```
#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

```
10
Do You want To Continue?y
Main Menu
1.Push
2.Pop
3.Display
4.exit
Enter Your Choice1

Enter The item to be pushed5

Do You want To Continue?y
Main Menu
1.Push
2.Pop
3.Display
4.exit
Enter Your Choice8

Do You want To Continue?y
Main Menu
1.Push
2.Pop
3.Display
4.exit
Enter Your Choice3

5
10
Do You want To Continue?
```

RESULT: Thus, the C program for Implementation of DAG has been executed and the output has been verified successfully.