EXPERIMENT NO - 3

Implementation of Lexical Analyzer

AIM:

To implement lexical analyzer to check if a number is odd or even.

CODE:

```
#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","fl
oat","for","goto","if","int","long","register","return","short","signed","sizeof","static","struct","switc
h","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("code.txt","r");

   if(fp == NULL){
```

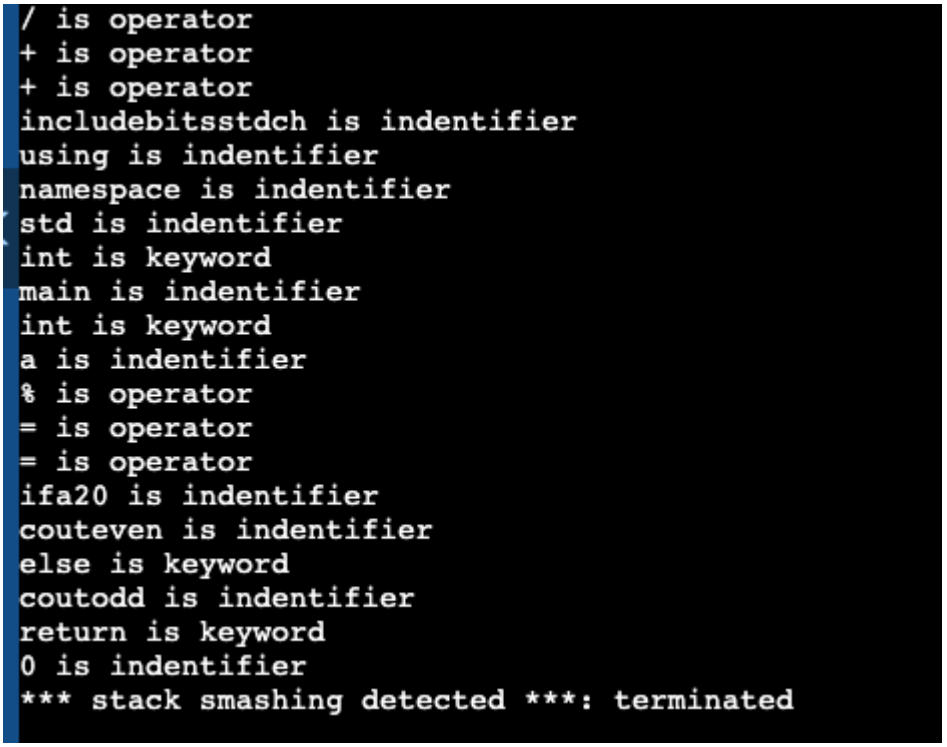
```
        printf("error while opening the file\n");
        exit(0);
    }
    while((ch = fgetc(fp)) != EOF){
        for(i = 0; i < 6; ++i){
            if(ch == operators[i])
                printf("%c is operator\n", ch);
        }
        if(isalnum(ch)){
            buffer[j++] = ch;
        }
        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;
}
```

File:

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int a;
    if(a%2==0)
        cout<<"even";
    else cout<<"odd";
```

```
    return 0;
}
```

OUTPUT:



RESULT:

The above program was created and executed successfully.

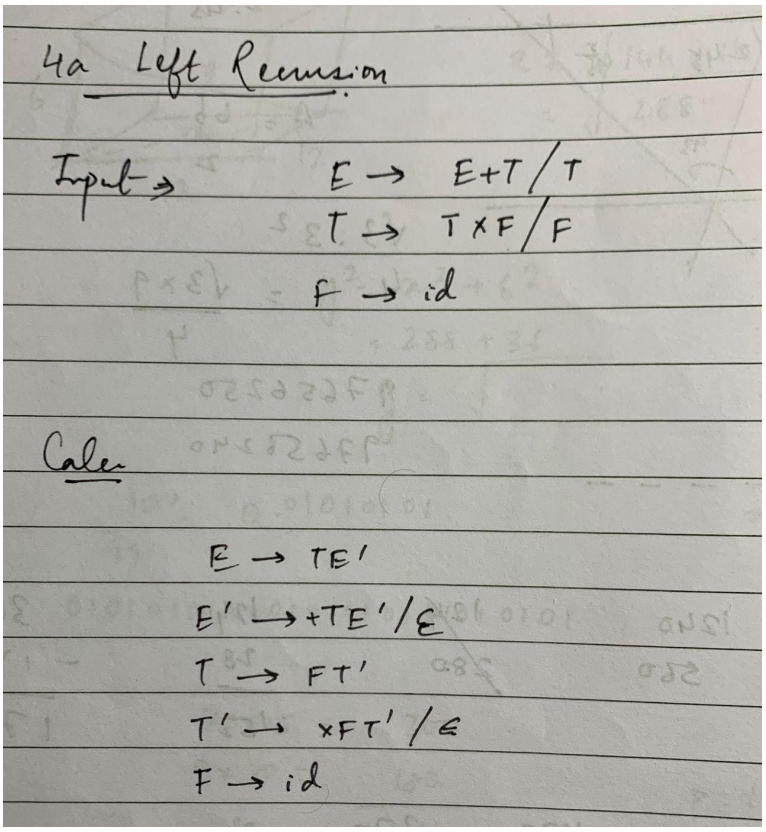
EXPERIMENT NO - 4a

Left Recursion Elimination

AIM:

To write a program to eliminate left recursion from the given grammar using manual calculation and using a C++ program.

MANUAL CALCULATION:



CODE:

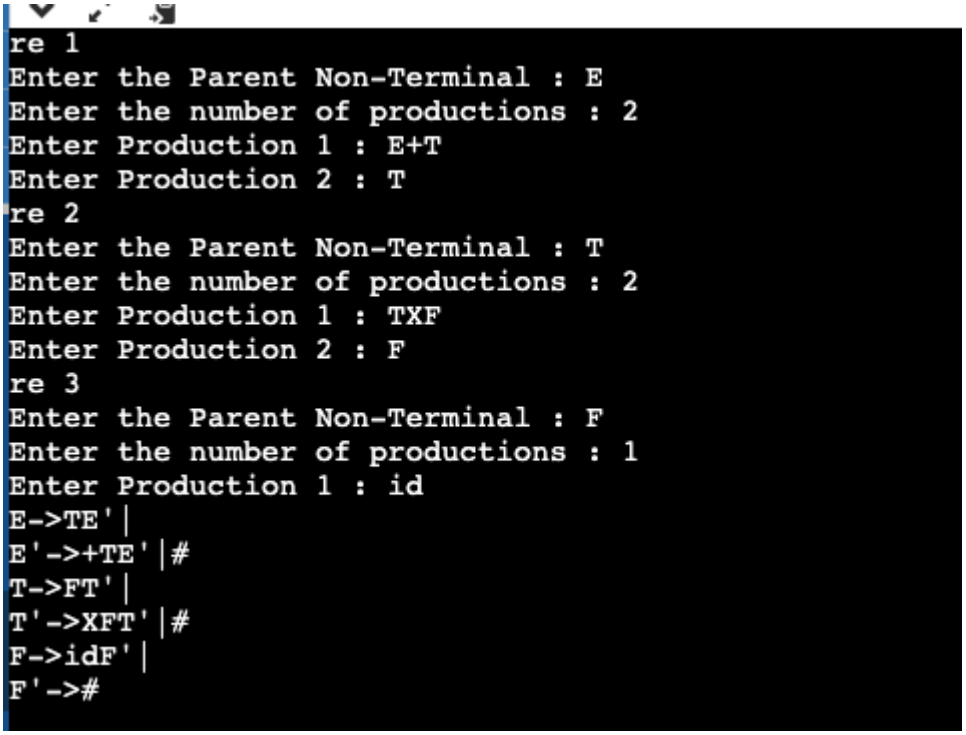
```
#include<iostream>
#include<string>
#include<vector>
using namespace std;
int main()
{
   vector<string> ans;
   for(int tt=0;tt<3;tt++)
```

```
{
cout<<"re "<<tt+1<<"\n";
string ip,op1,op2,temp;
int sizes[10] = {};
char c;
int n,j,l;
cout<<"Enter the Parent Non-Terminal : ";
cin>>c;
ip.push_back(c);
op1 += ip + "\'->";
ip += "->";
op2+=ip;
cout<<"Enter the number of productions : ";
cin>>n;
for(int i=0;i<n;i++)
{   cout<<"Enter Production "<<i+1<<" : ";
   cin>>temp;
   sizes[i] = temp.size();
   ip+=temp;
   if(i!=n-1)
      ip += "|";
}
// cout<<"Production Rule : "<<ip<<endl;
for(int i=0,k=3;i<n;i++)
{
   if(ip[0] == ip[k])
   {
      // cout<<"Production "<<i+1<<" has left recursion."<<endl;
      if(ip[k] != '#')
      {
         for(l=k+1;l<k+sizes[i];l++)
```

```
            op1.push_back(ip[l]);
        k=l+1;
        op1.push_back(ip[0]);
        op1 += "\'|";
      }
   }
   else
   {
      // cout<<"Production "<<i+1<<" does not have left recursion."<<endl;
      if(ip[k] != '#')
      {
        for(j=k;j<k+sizes[i];j++)
           op2.push_back(ip[j]);
        k=j+1;
        op2.push_back(ip[0]);
        op2 += "\'|";
      }
      else
      {
        op2.push_back(ip[0]);
        op2 += "\'";
      }}}
op1 += "#";
ans.push_back(op2);
ans.push_back(op1);
// cout<<op2<<endl;
// cout<<op1<<endl;
}
for(int i=0;i<ans.size();i++)
   cout<<ans[i]<<"\n";
return 0;
```

}

OUTPUT:



RESULT:

Left Recursion Elimination was successfully calculated manually and the program was successfully executed using an Online C++ compiler.

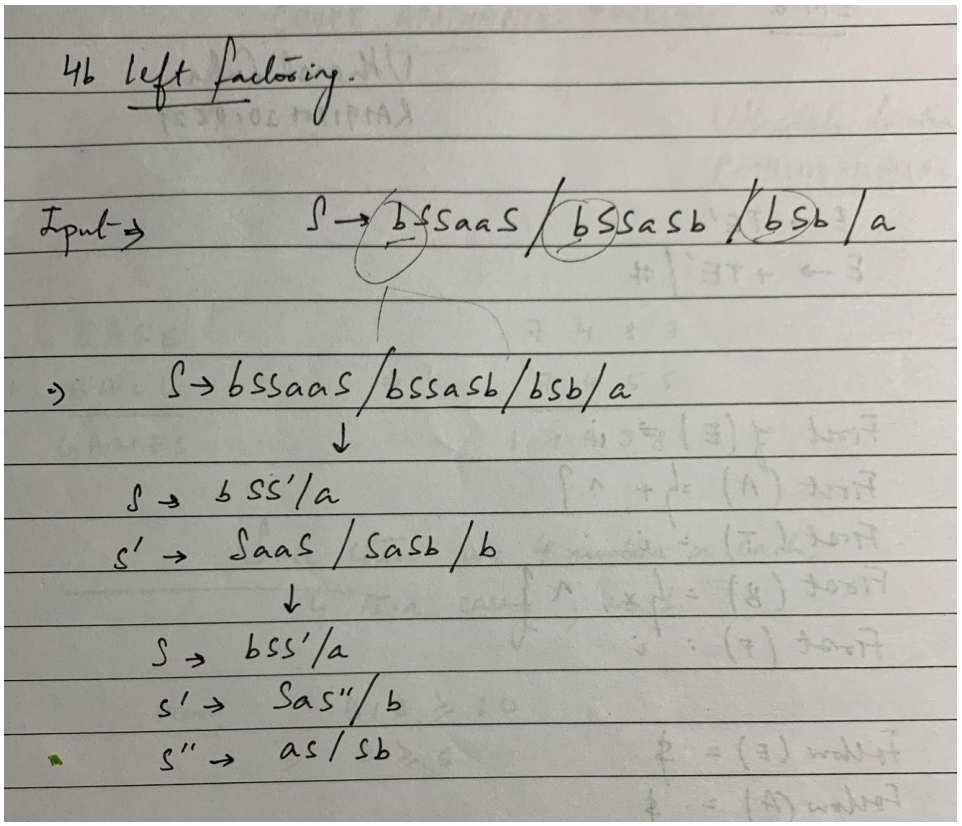
EXPERIMENT NO - 4b

Left Factoring

AIM:

To write a program for performing left factoring to the given grammar using manual calculation and using a C++ program.

MANUAL CALCULATION:



CODE:

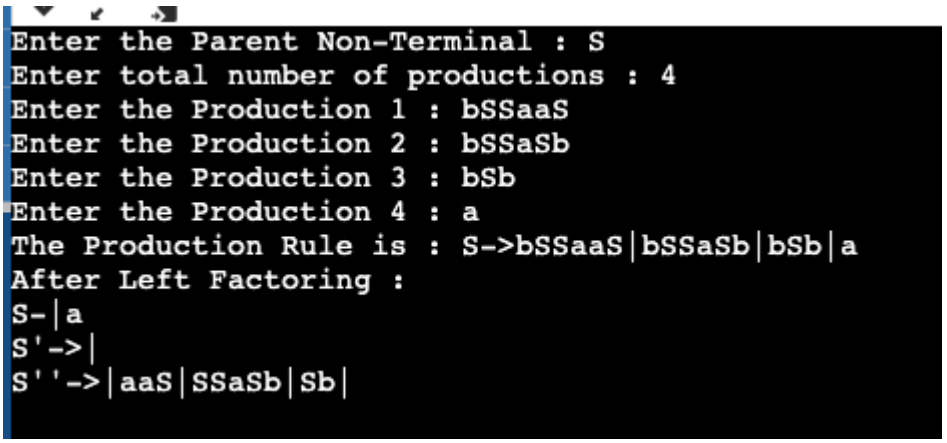
```
#include <iostream>
#include <string>
using namespace std;
int main()
{
        int n,j,l,i,m;
        int len[10] = {};
   string a, b1, b2, flag;
   char c;
```

```
cout << "Enter the Parent Non-Terminal : ";
cin >> c;
a.push_back(c);
b1 += a + "\'->";
b2 += a + "\'\'->";;
a += "->";
cout << "Enter total number of productions : ";
cin >> n;
for (i = 0; i < n; i++)
{
   cout << "Enter the Production " << i + 1 << " : ";
   cin >> flag;
   len[i] = flag.size();
   a += flag;
   if (i != n - 1)
   {
   a += "|";
                    }
}
cout << "The Production Rule is : " << a << endl;
char x = a[3];
for (i = 0, m = 3; i < n; i++)
{
   if (x != a[m])
   {
       while (a[m++] != '|');
   }
   else
   {
         if (a[m + 1] != '|')
      {
```

```
            b1 += "|" + a.substr(m + 1, len[i] - 1);
            a.erase(m - 1, len[i] + 1);
        }
        else
        {
          b1 += "#";
            a.insert(m + 1, 1, a[0]);
            a.insert(m + 2, 1, '\'');
            m += 4;
        }
    }
}
char y = b1[6];
for (i = 0, m = 6; i < n - 1; i++)
{
    if (y == b1[m])
    {
        if (b1[m + 1] != '|')
        {
            flag.clear();
            for (int s = m + 1; s < b1.length(); s++)
            {
                flag.push_back(b1[s]);
                                        }
            b2 += "|" + flag;
            b1.erase(m - 1, flag.length() + 2);
        }
        else
        {
          b1.insert(m + 1, 1, b1[0]);
            b1.insert(m + 2, 2, '\'');
```

```
            b2 += "#";
            m += 5;
          }
        }
    }
    b2.erase(b2.size() - 1);
    cout << "After Left Factoring : " << endl;
    cout << a << endl;
    cout << b1 << endl;
    cout << b2 << endl;
    return 0;
}
```

OUTPUT:



RESULT:

Left Factoring was successfully calculated manually and the program was successfully executed using an Online C++ compiler.

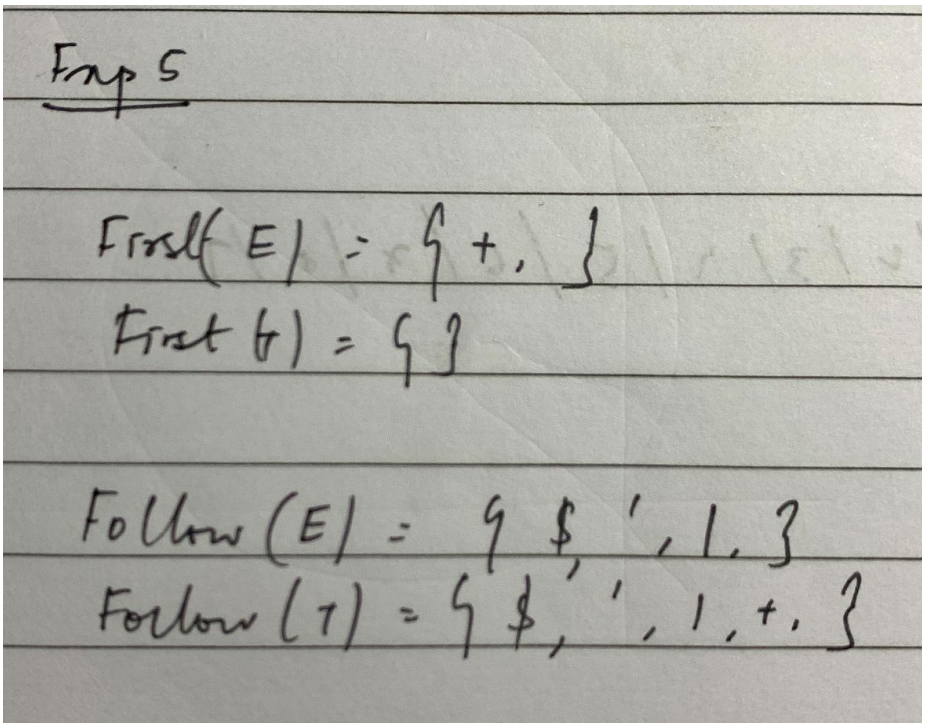
EXPERIMENT NO - 5

FIRST AND FOLLOW computation

AIM:

To write a program for finding first and follow for the given grammar using manual calculation and using a C++ program.

MANUAL CALCULATION:



CODE:

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

// Functions to calculate Follow
void followfirst(char, int, int);
void follow(char c);

// Function to calculate First
void findfirst(char, int, int);
```

```
int count, n = 0;

// Stores the final result
// of the First Sets
char calc_first[10][100];

// Stores the final result
// of the Follow Sets
char calc_follow[10][100];
int m = 0;

// Stores the production rules
char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;

int main(int argc, char **argv)
{
        int jm = 0;
        int km = 0;
        int i, choice;
        char c, ch;
        count = 8;

        // The Input grammar
        strcpy(production[0], "S=01A");
        strcpy(production[1], "A=0S1SA");
        strcpy(production[2], "A=#");
```

```
int kay;
char done[count];
int ptr = -1;

// Initializing the calc_first array
for(k = 0; k < count; k++) {
        for(kay = 0; kay < 100; kay++) {
                calc_first[k][kay] = '!';
        }
}
int point1 = 0, point2, xxx;

for(k = 0; k < count; k++)
{
        c = production[k][0];
        point2 = 0;
        xxx = 0;

        // Checking if First of c has
        // already been calculated
        for(kay = 0; kay <= ptr; kay++)
                if(c == done[kay])
                        xxx = 1;

        if (xxx == 1)
                continue;

        // Function call
        findfirst(c, 0, 0);
        ptr += 1;
```

```
		// Adding c to the calculated list
		done[ptr] = c;
		printf("\n First(%c) = { ", c);
		calc_first[point1][point2++] = c;

		// Printing the First Sets of the grammar
		for(i = 0 + jm; i < n; i++) {
			int lark = 0, chk = 0;

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

				if (first[i] == calc_first[point1][lark])
				{
					chk = 1;
					break;
				}
			}
			if(chk == 0)
			{
				printf("%c, ", first[i]);
				calc_first[point1][point2++] = first[i];
			}
		}
		printf("}\n");
		jm = n;
		point1++;
	}
	printf("\n");
	printf("----------------------------------------------\n\n");
	char donee[count];
```

```
ptr = -1;

// Initializing the calc_follow array
for(k = 0; k < count; k++) {
        for(kay = 0; kay < 100; kay++) {
                calc_follow[k][kay] = '!';
        }
}
point1 = 0;
int land = 0;
for(e = 0; e < count; e++)
{
        ck = production[e][0];
        point2 = 0;
        xxx = 0;

        // Checking if Follow of ck
        // has alredy been calculated
        for(kay = 0; kay <= ptr; kay++)
                if(ck == donee[kay])
                        xxx = 1;

        if (xxx == 1)
                continue;
        land += 1;

        // Function call
        follow(ck);
        ptr += 1;

        // Adding ck to the calculated list
```

```
                donee[ptr] = ck;
                printf(" Follow(%c) = { ", ck);
                calc_follow[point1][point2++] = ck;

                // Printing the Follow Sets of the grammar
                for(i = 0 + km; i < m; i++) {
                        int lark = 0, chk = 0;
                        for(lark = 0; lark < point2; lark++)
                        {
                                if (f[i] == calc_follow[point1][lark])
                                {
                                        chk = 1;
                                        break;
                                }
                        }
                        if(chk == 0)
                        {
                                printf("%c, ", f[i]);
                                calc_follow[point1][point2++] = f[i];
                        }
                }
                printf(" }\n\n");
                km = m;
                point1++;
        }
}

void follow(char c)
{
        int i, j;
```

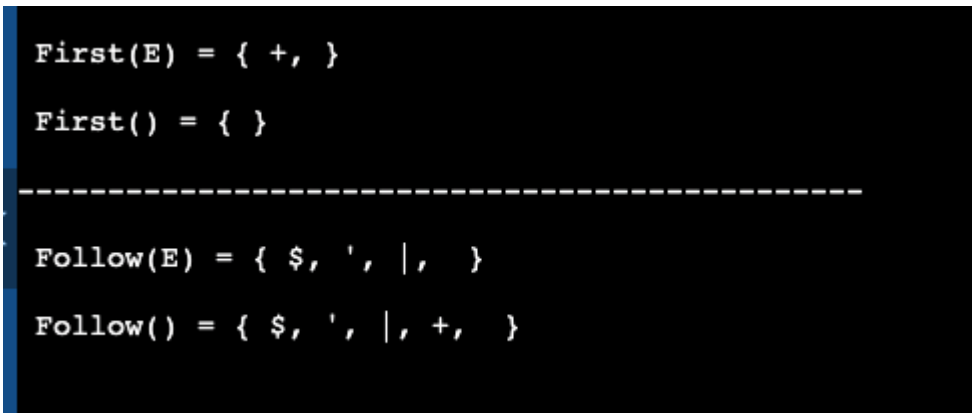
```
	// Adding "$" to the follow
	// set of the start symbol
	if(production[0][0] == c) {
		f[m++] = '$';
	}
	for(i = 0; i < 10; i++)
	{
		for(j = 2;j < 10; j++)
		{
			if(production[i][j] == c)
			{
				if(production[i][j+1] != '\0')
				{
					// Calculate the first of the next
					// Non-Terminal in the production
					followfirst(production[i][j+1], i, (j+2));
				}

				if(production[i][j+1]=='\0' && c!=production[i][0])
				{
					// Calculate the follow of the Non-Terminal
					// in the L.H.S. of the production
					follow(production[i][0]);
				}
			}
		}
	}
}

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

```
int j;

// The case where we
// encounter a Terminal
if(!(isupper(c))) {
        first[n++] = c;
}
for(j = 0; j < count; j++)
{
        if(production[j][0] == c)
        {
                if(production[j][2] == '#')
                {
                        if(production[q1][q2] == '\0')
                                first[n++] = '#';
                        else if(production[q1][q2] != '\0'
                                        && (q1 != 0 || q2 != 0))
                        {
                                // Recursion to calculate First of New
                                // Non-Terminal we encounter after epsilon
                                findfirst(production[q1][q2], q1, (q2+1));
                        }
                        else
                                first[n++] = '#';
                }
                else if(!isupper(production[j][2]))
                {
                        first[n++] = production[j][2];
                }
                else
                {
```

```
                                        // Recursion to calculate First of
                                        // New Non-Terminal we encounter
                                        // at the beginning
                                        findfirst(production[j][2], j, 3);
                                }
                        }
                }
}

void followfirst(char c, int c1, int c2)
{
        int k;

        // The case where we encounter
        // a Terminal
        if(!(isupper(c)))
                f[m++] = c;
        else
        {
                int i = 0, j = 1;
                for(i = 0; i < count; i++)
                {
                        if(calc_first[i][0] == c)
                                break;
                }

                //Including the First set of the
                // Non-Terminal in the Follow of
                // the original query
                while(calc_first[i][j] != '!')
                {
```

```
                        if(calc_first[i][j] != '#')
                        {
                                f[m++] = calc_first[i][j];
                        }
                        else
                        {
                                if(production[c1][c2] == '\0')
                                {
                                        // Case where we reach the
                                        // end of a production
                                        follow(production[c1][0]);
                                }
                                else
                                {
                                        // Recursion to the next symbol
                                        // in case we encounter a "#"
                                        followfirst(production[c1][c2], c1, c2+1);
                                }
                        }
                        j++;
                }
        }
}
```

OUTPUT:



RESULT:

First and follow fwas successfully calculated manually and the program was successfully executed using an Online C++ compiler

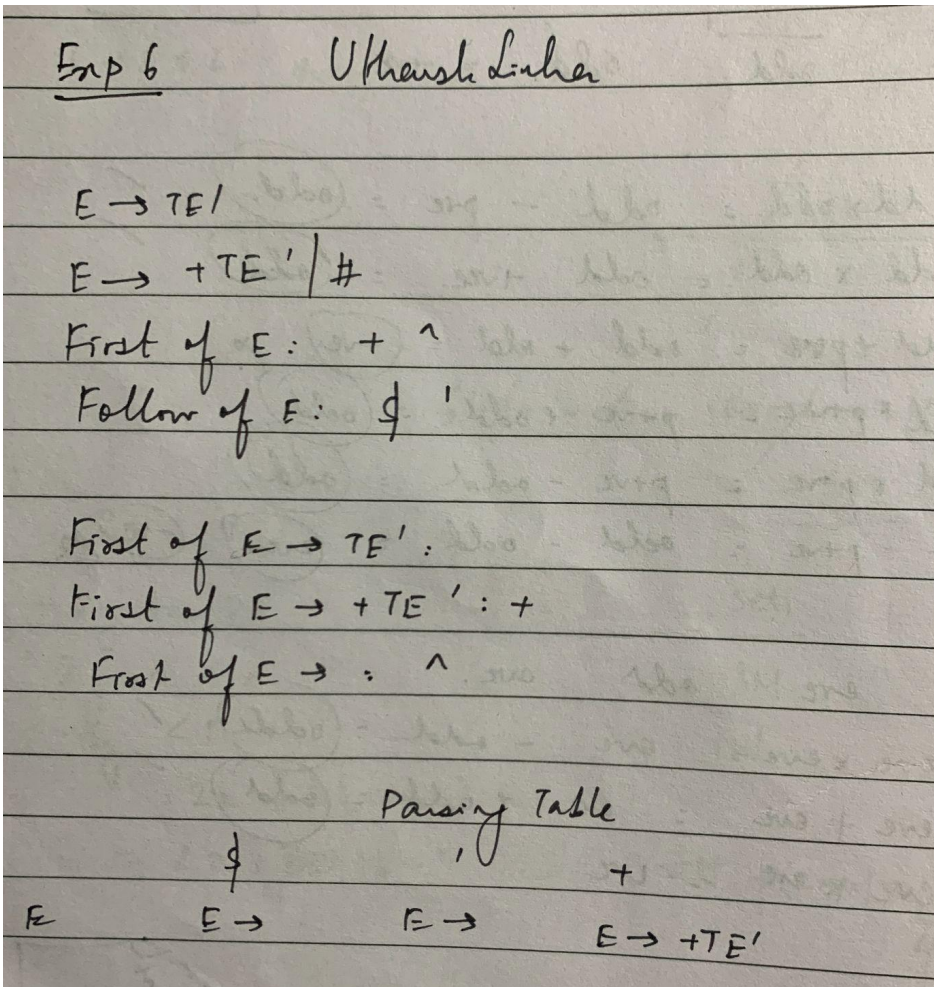
EXPERIMENT NO - 6

Predictive Parsing Table

AIM:

To write a program for making Predictive Parsing Table to the given grammar using manual calculation and using a C++ program.

MANUAL CALCULATION:



CODE:

```
#include<stdio.h>
#include<string.h>
#define TSIZE 128
int table[100][TSIZE];
char terminal[TSIZE];
```

```
char nonterminal[26];
struct product {
   char str[100];
   int len;
}pro[20];
int no_pro;
char first[26][TSIZE];
char follow[26][TSIZE];
char first_rhs[100][TSIZE];
int isNT(char c) {
   return c >= 'A' && c <= 'Z';
}
void readFromFile() {
   FILE* fptr;
   fptr = fopen("TEXT.txt", "r");
   char buffer[255];
   int i;
   int j;
   while (fgets(buffer, sizeof(buffer), fptr)) {
      printf("%s", buffer);
      j = 0;
      nonterminal[buffer[0] - 'A'] = 1;
      for (i = 0; i < strlen(buffer) - 1; ++i) {
         if (buffer[i] == '|') {
            ++no_pro;
            pro[no_pro - 1].str[j] = '\0';
            pro[no_pro - 1].len = j;
            pro[no_pro].str[0] = pro[no_pro - 1].str[0];
            pro[no_pro].str[1] = pro[no_pro - 1].str[1];
            pro[no_pro].str[2] = pro[no_pro - 1].str[2];
            j = 3;
```

```
            }
            else {
                pro[no_pro].str[j] = buffer[i];
                ++j;
                if (!isNT(buffer[i]) && buffer[i] != '-' && buffer[i] != '>') {
                    terminal[buffer[i]] = 1;
                }
            }
        }
        pro[no_pro].len = j;
        ++no_pro;
    }
}
void add_FIRST_A_to_FOLLOW_B(char A, char B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^')
            follow[B - 'A'][i] = follow[B - 'A'][i] || first[A - 'A'][i];
    }
}
void add_FOLLOW_A_to_FOLLOW_B(char A, char B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^')
            follow[B - 'A'][i] = follow[B - 'A'][i] || follow[A - 'A'][i];
    }
}
void FOLLOW() {
    int t = 0;
    int i, j, k, x;
    while (t++ < no_pro) {
```

```
        for (k = 0; k < 26; ++k) {
            if (!nonterminal[k])    continue;
            char nt = k + 'A';
            for (i = 0; i < no_pro; ++i) {
                for (j = 3; j < pro[i].len; ++j) {
                    if (nt == pro[i].str[j]) {
                        for (x = j + 1; x < pro[i].len; ++x) {
                            char sc = pro[i].str[x];
                            if (isNT(sc)) {
                                add_FIRST_A_to_FOLLOW_B(sc, nt);
                                if (first[sc - 'A']['^'])
                                    continue;
                            }
                            else {
                                follow[nt - 'A'][sc] = 1;
                            }
                            break;
                        }
                        if (x == pro[i].len)
                            add_FOLLOW_A_to_FOLLOW_B(pro[i].str[0], nt);
                    }
                }
            }
        }
    }
}
void add_FIRST_A_to_FIRST_B(char A, char B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^') {
            first[B - 'A'][i] = first[A - 'A'][i] || first[B - 'A'][i];
```

```
        }
    }
}
void FIRST() {
    int i, j;
    int t = 0;
    while (t < no_pro) {
        for (i = 0; i < no_pro; ++i) {
            for (j = 3; j < pro[i].len; ++j) {
                char sc = pro[i].str[j];
                if (isNT(sc)) {
                    add_FIRST_A_to_FIRST_B(sc, pro[i].str[0]);
                    if (first[sc - 'A']['^'])
                        continue;
                }
                else {
                    first[pro[i].str[0] - 'A'][sc] = 1;
                }
                break;
            }
            if (j == pro[i].len)
                first[pro[i].str[0] - 'A']['^'] = 1;
        }
        ++t;
    }
}
void add_FIRST_A_to_FIRST_RHS__B(char A, int B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^')
            first_rhs[B][i] = first[A - 'A'][i] || first_rhs[B][i];
```

```
    }
}
// Calculates FIRST(ß) for each A->ß
void FIRST_RHS() {
    int i, j;
    int t = 0;
    while (t < no_pro) {
        for (i = 0; i < no_pro; ++i) {
            for (j = 3; j < pro[i].len; ++j) {
                char sc = pro[i].str[j];
                if (isNT(sc)) {
                    add_FIRST_A_to_FIRST_RHS__B(sc, i);
                    if (first[sc - 'A']['^'])
                        continue;
                }
                else {
                    first_rhs[i][sc] = 1;
                }
                break;
            }
            if (j == pro[i].len)
                first_rhs[i]['^'] = 1;
        }
        ++t;
    }
}
int main() {
    readFromFile();
    follow[pro[0].str[0] - 'A']['$'] = 1;
    FIRST();
    FOLLOW();
```

```
FIRST_RHS();
int i, j, k;

// display first of each variable
printf("\n");
for (i = 0; i < no_pro; ++i) {
   if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
      char c = pro[i].str[0];
      printf("FIRST OF %c: ", c);
      for (j = 0; j < TSIZE; ++j) {
         if (first[c - 'A'][j]) {
            printf("%c ", j);
         }
      }
      printf("\n");
   }
}

// display follow of each variable
printf("\n");
for (i = 0; i < no_pro; ++i) {
   if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
      char c = pro[i].str[0];
      printf("FOLLOW OF %c: ", c);
      for (j = 0; j < TSIZE; ++j) {
         if (follow[c - 'A'][j]) {
            printf("%c ", j);
         }
      }
      printf("\n");
   }
```

```
}
// display first of each variable ß
// in form A->ß
printf("\n");
for (i = 0; i < no_pro; ++i) {
   printf("FIRST OF %s: ", pro[i].str);
   for (j = 0; j < TSIZE; ++j) {
      if (first_rhs[i][j]) {
         printf("%c ", j);
      }
   }
   printf("\n");
}

// the parse table contains '$'
// set terminal['$'] = 1
// to include '$' in the parse table
terminal['$'] = 1;

// the parse table do not read '^'
// as input
// so we set terminal['^'] = 0
// to remove '^' from terminals
terminal['^'] = 0;

// printing parse table
printf("\n");
printf("\n\t********** LL(1) PARSING TABLE **********\n");
printf("\t-------------------------------------------------------\n");
printf("%-10s", "");
for (i = 0; i < TSIZE; ++i) {
```

```
    if (terminal[i])    printf("%-10c", i);
}
printf("\n");
int p = 0;
for (i = 0; i < no_pro; ++i) {
    if (i != 0 && (pro[i].str[0] != pro[i - 1].str[0]))
        p = p + 1;
    for (j = 0; j < TSIZE; ++j) {
        if (first_rhs[i][j] && j != '^') {
            table[p][j] = i + 1;
        }
        else if (first_rhs[i]['^']) {
            for (k = 0; k < TSIZE; ++k) {
                if (follow[pro[i].str[0] - 'A'][k]) {
                    table[p][k] = i + 1;
                }
            }
        }
    }
}
k = 0;
for (i = 0; i < no_pro; ++i) {
    if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
        printf("%-10c", pro[i].str[0]);
        for (j = 0; j < TSIZE; ++j) {
            if (table[k][j]) {
                printf("%-10s", pro[table[k][j] - 1].str);
            }
            else if (terminal[j]) {
                printf("%-10s", "");
            }
```

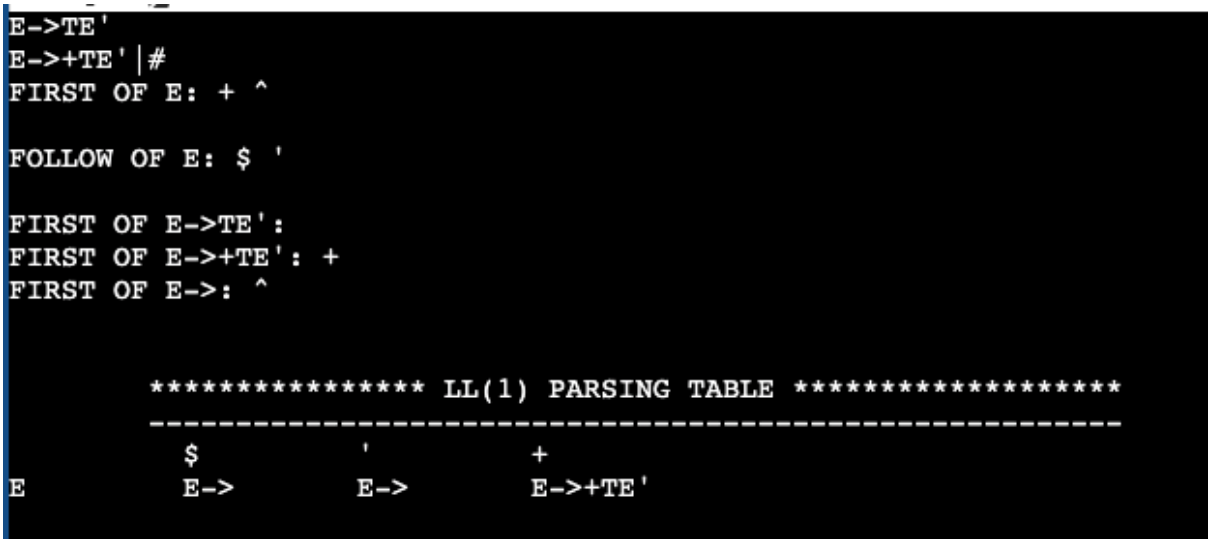
```
        }
        ++k;
        printf("\n");
      }
    }
}
```

TEXT.txt FILE:

E->TE'

E->+TE'|#

OUTPUT:



RESULT:

Predictive Parsing Table was successfully calculated manually and the program was successfully executed using an Online C++ compiler.

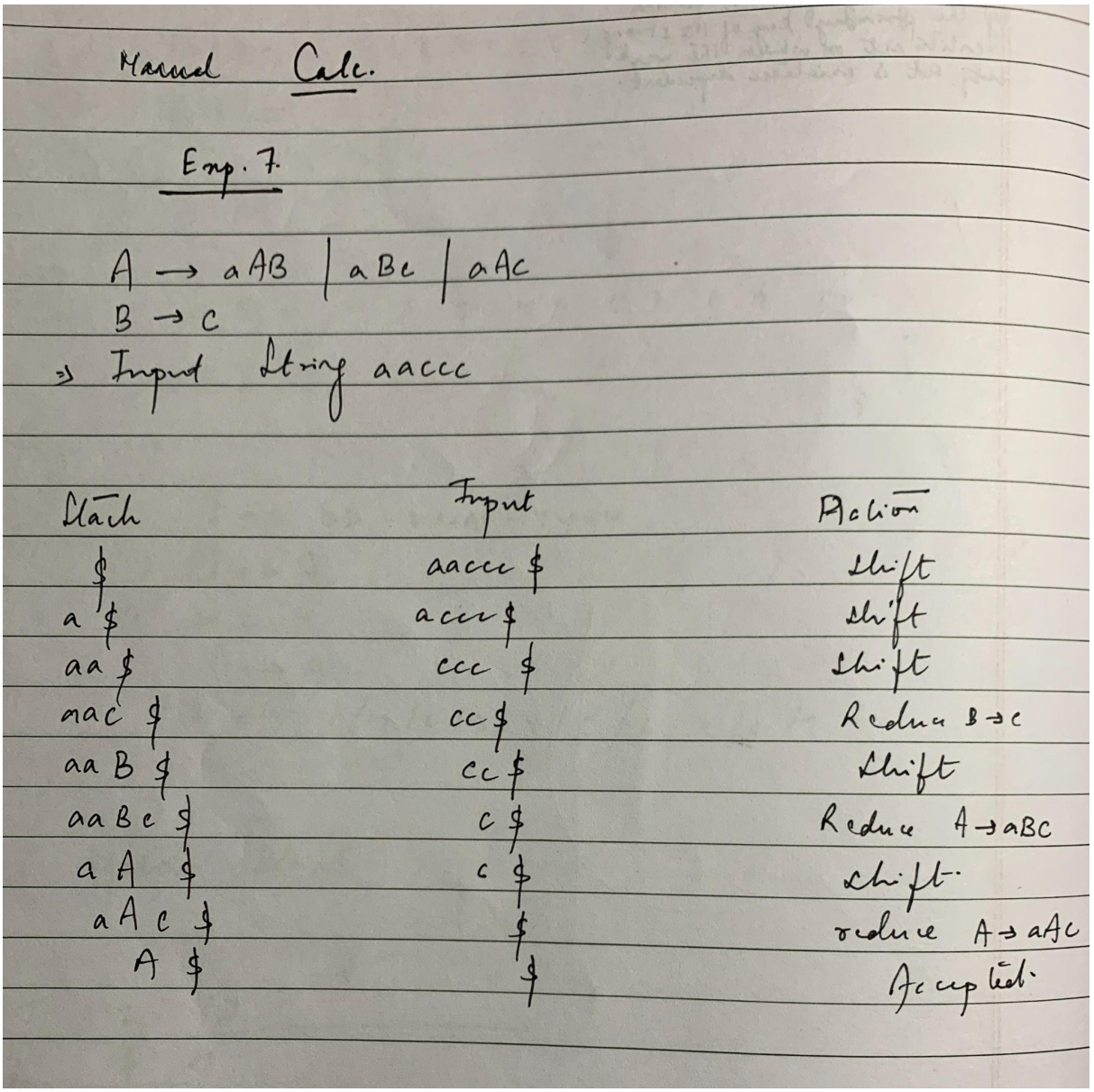
EXPERIMENT NO - 7

Shift Reduce Parsing

AIM:

To write a program for performing Shift Reduce Parsing to the given grammar using manual calculation and using a C++ program.

MANUAL CALCULATION:



CODE:

```
#include<iostream>
```

```
#include<string.h>
using namespace std;
struct prodn
{
	char p1[10];
	char p2[10];
};
int main()
{
	char input[20],stack[50],temp[50],ch[2],*t1,*t2,*t;
	int i,j,s1,s2,s,count=0;
	struct prodn p[10];
	FILE *fp=fopen("input.txt","r");
	stack[0]='\0';
	cout<<"Enter the Input String:\n";
	cin>>input;
	while(!feof(fp))
	{
		fscanf(fp,"%s\n",temp);
		t1=strtok(temp,"->");
		t2=strtok(NULL,"->");
		strcpy(p[count].p1,t1);
		strcpy(p[count].p2,t2);
		count++;
	}
	i=0;
	while(1)
	{
		if(i<strlen(input))
		{
```

```
            ch[0]=input[i];
            ch[1]='\0';
            i++;
            strcat(stack,ch);
            cout<<"\n"<<stack;
    }
    for(j=0;j<count;j++)
    {
            t=strstr(stack,p[j].p2);
            if(t!=NULL)
            {
                    s1=strlen(stack);
                    s2=strlen(t);
                    s=s1-s2;
                    stack[s]='\0';
                    strcat(stack,p[j].p1);
                    cout<<"\n"<<stack;
                    j=-1;
            }
    }
    if(strcmp(stack,"E")==0&&i==strlen(input))
    {
            cout<<"\n\nAccepted";
            break;
    }
    if(i==strlen(input))
    {
            cout<<"\n\nNot Accepted";
            break;
    }
```

```
    }
    return 0;
}
```

TEXT FILE:

A=>aAB|aBc|aAc

B->C

OUTPUT:

RESULT:

Shift Reduce Parsing was successfully calculated manually and the program was successfully executed using an Online C++ compiler.

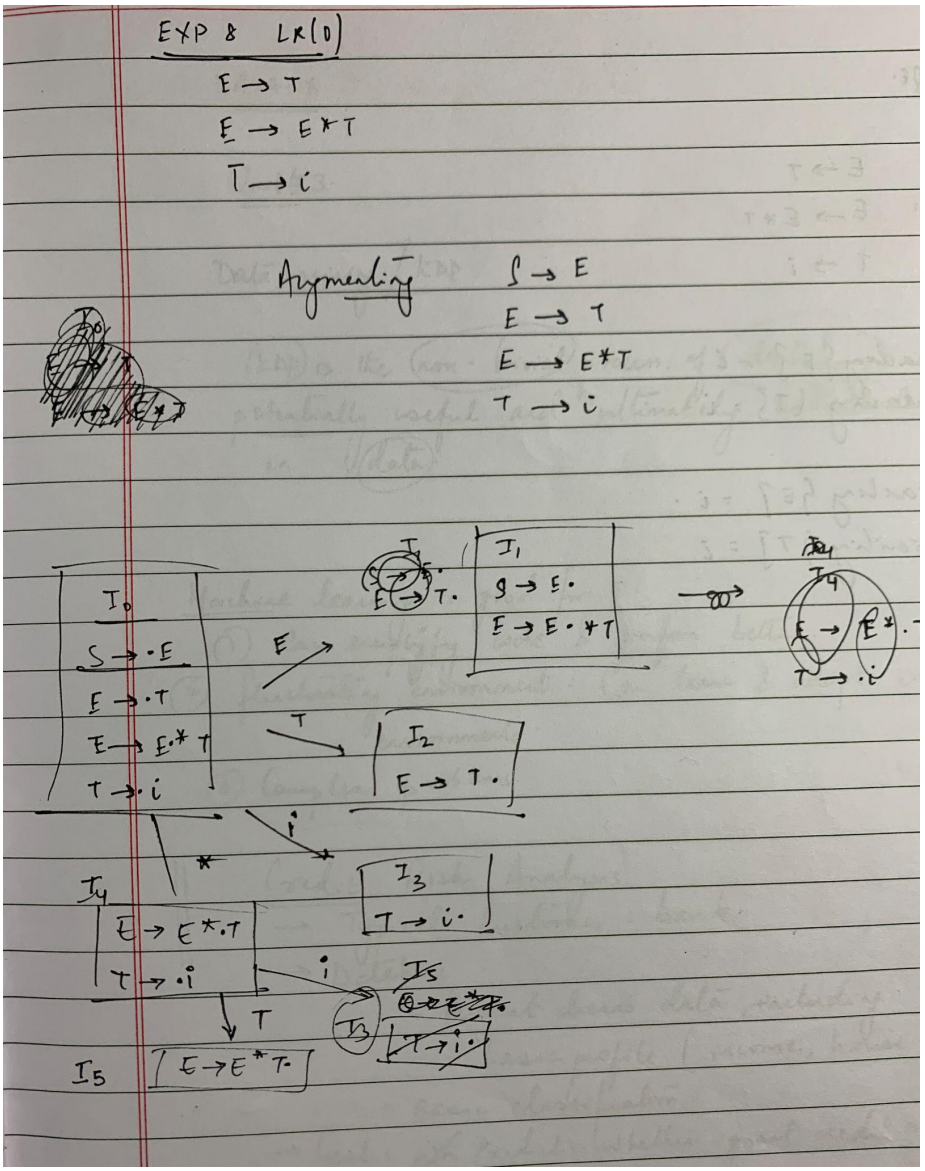
EXPERIMENT NO - 8

Computation of LR(0) items

AIM:

To write a program for finding LR(0) items for the given grammar using manual calculation and using a C++ program.

MANUAL CALCULATION:



CODE:

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

using namespace std;

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

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

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

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

```
                                        }
                                }
                        }
                }
        }
        arr[noitem]=n;
        int flag=0;
        for(i=0;i<noitem;i++)
        {
                if(arr[i]==n)
                {
                        for(j=0;j<arr[i];j++)
                        {
                                int c=0;
                                for(k=0;k<arr[i];k++)
                                        if(clos[noitem][k].lhs==clos[i][k].lhs &&
strcmp(clos[noitem][k].rhs,clos[i][k].rhs)==0)
                                                c=c+1;
                                if(c==arr[i])
                                {
                                        flag=1;
                                        goto exit;
                                }
                        }
                }
        }
        exit:;
        if(flag==0)
                arr[noitem++]=n;
}
```

```
int main()
{
	cout<<"ENTER THE PRODUCTIONS OF THE GRAMMAR(0 TO END) :\n";
	do
	{
		cin>>prod[i++];
	}while(strcmp(prod[i-1],"0")!=0);
	for(n=0;n<i-1;n++)
	{
		m=0;
		j=novar;
		g[novar++].lhs=prod[n][0];
		for(k=3;k<strlen(prod[n]);k++)
		{
			if(prod[n][k] != '|')
			g[j].rhs[m++]=prod[n][k];
			if(prod[n][k]=='|')
			{
				g[j].rhs[m]='\0';
				m=0;
				j=novar;
				g[novar++].lhs=prod[n][0];
			}
		}
	}
	for(i=0;i<26;i++)
		if(!isvariable(listofvar[i]))
			break;
	g[0].lhs=listofvar[i];
```

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

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

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

OUTPUT:

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


 augumented grammar

A->E
E->T
E->E*T
T->i
 THE SET OF ITEMS ARE


 I0

A->.E
E->.T
E->.E*T
T->.i

 I1

A->E.
E->E.*T

 I2

E->T.

 I3

T->i.

 I4

E->E*.T
T->.i

 I5

E->E*T.
```

RESULT:

LR(0) items was successfully calculated manually and the program was successfully executed using an Online C++ compiler.

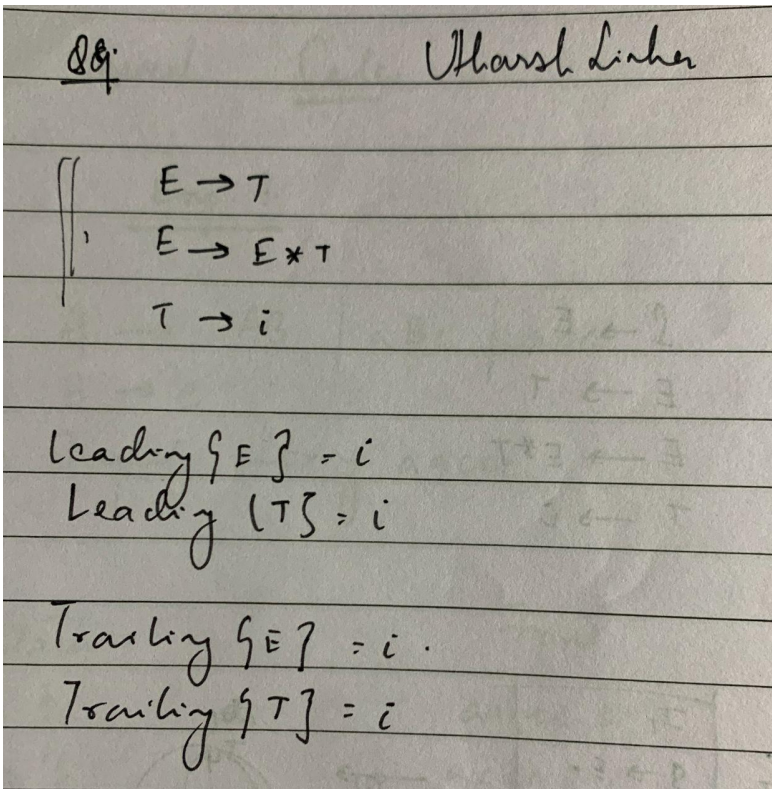
EXPERIMENT NO - 9

Computation of LEADING AND TRAILING

AIM:

To write a program for finding Leading and Trailing to the given grammar using manual calculation and using a C++ program.

MANUAL CALCULATION:



CODE:

```
#include<iostream>
#include<conio.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
using namespace std;

int vars,terms,i,j,k,m,rep,count,temp=-1;
```

```
char var[10],term[10],lead[10][10],trail[10][10];
struct grammar
{
	int prodno;
	char lhs,rhs[20][20];
}gram[50];
void get()
{
	cout<<"\nLEADING AND TRAILING\n";
	cout<<"\nEnter the no. of variables : ";
	cin>>vars;
	cout<<"\nEnter the variables : \n";
	for(i=0;i<vars;i++)
	{
		cin>>gram[i].lhs;
		var[i]=gram[i].lhs;
	}
	cout<<"\nEnter the no. of terminals : ";
	cin>>terms;
	cout<<"\nEnter the terminals : ";
	for(j=0;j<terms;j++)
		cin>>term[j];
	cout<<"\nPRODUCTION DETAILS\n";
	for(i=0;i<vars;i++)
	{
		cout<<"\nEnter the no. of production of "<<gram[i].lhs<<":";
		cin>>gram[i].prodno;
		for(j=0;j<gram[i].prodno;j++)
		{
			cout<<gram[i].lhs<<"->";
```

```
                cin>>gram[i].rhs[j];
            }
        }
}
void leading()
{
        for(i=0;i<vars;i++)
        {
            for(j=0;j<gram[i].prodno;j++)
            {
                for(k=0;k<terms;k++)
                {
                    if(gram[i].rhs[j][0]==term[k])
                        lead[i][k]=1;
                    else
                    {
                        if(gram[i].rhs[j][1]==term[k])
                            lead[i][k]=1;
                    }
                }
            }
        }
        for(rep=0;rep<vars;rep++)
        {
            for(i=0;i<vars;i++)
            {
                for(j=0;j<gram[i].prodno;j++)
                {
                    for(m=1;m<vars;m++)
                    {
```

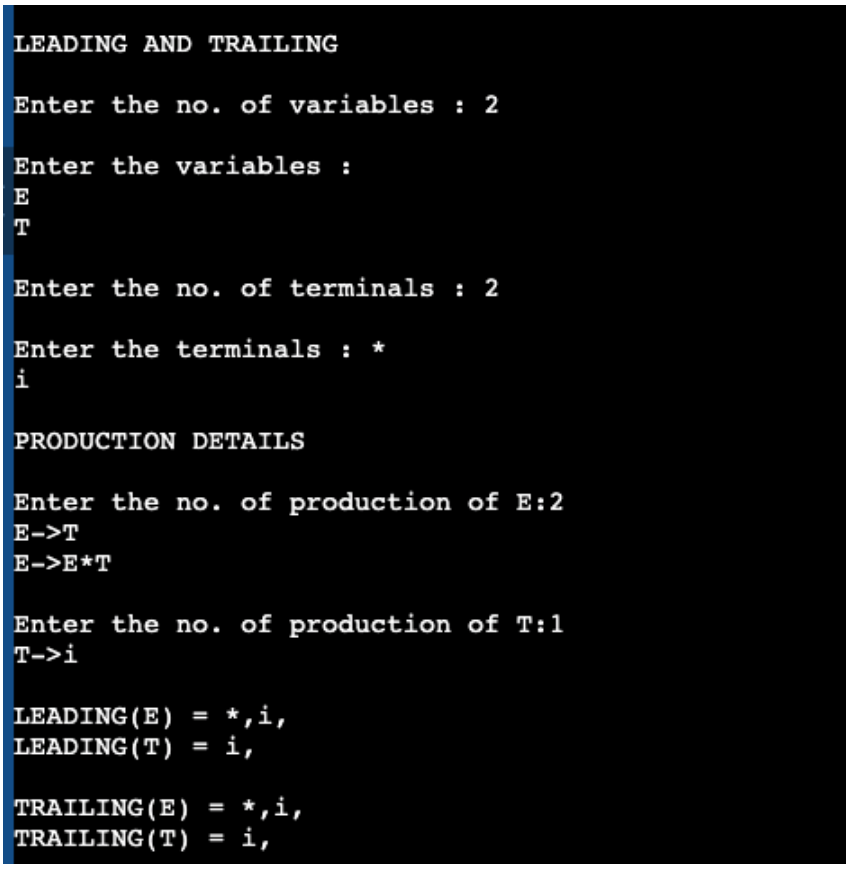
```
                        if(gram[i].rhs[j][0]==var[m])
                        {
                                temp=m;
                                goto out;
                        }
                }
                out:
                for(k=0;k<terms;k++)
                {
                        if(lead[temp][k]==1)
                                lead[i][k]=1;
                }
        }
    }
}
void trailing()
{
    for(i=0;i<vars;i++)
    {
        for(j=0;j<gram[i].prodno;j++)
        {
            count=0;
            while(gram[i].rhs[j][count]!='\x0')
                count++;
            for(k=0;k<terms;k++)
            {
                if(gram[i].rhs[j][count-1]==term[k])
                    trail[i][k]=1;
                else
```

```
                        {
                                if(gram[i].rhs[j][count-2]==term[k])
                                        trail[i][k]=1;
                        }
                }
        }
        for(rep=0;rep<vars;rep++)
        {
                for(i=0;i<vars;i++)
                {
                        for(j=0;j<gram[i].prodno;j++)
                        {
                                count=0;
                                while(gram[i].rhs[j][count]!='\x0')
                                        count++;
                                for(m=1;m<vars;m++)
                                {
                                        if(gram[i].rhs[j][count-1]==var[m])
                                                temp=m;
                                }
                                for(k=0;k<terms;k++)
                                {
                                        if(trail[temp][k]==1)
                                                trail[i][k]=1;
                                }
                        }
                }
        }
}
```

```
void display()
{
	for(i=0;i<vars;i++)
	{
		cout<<"\nLEADING("<<gram[i].lhs<<") = ";
		for(j=0;j<terms;j++)
		{
			if(lead[i][j]==1)
				cout<<term[j]<<",";
		}
	}
	cout<<endl;
	for(i=0;i<vars;i++)
	{
		cout<<"\nTRAILING("<<gram[i].lhs<<") = ";
		for(j=0;j<terms;j++)
		{
			if(trail[i][j]==1)
				cout<<term[j]<<",";
		}
	}
}
int main()
{

	get();
	leading();
	trailing();
	display();
```

}

OUTPUT:



RESULT:

Leading and Trailing was successfully calculated manually and the program was successfully executed using an Online C++ compiler.

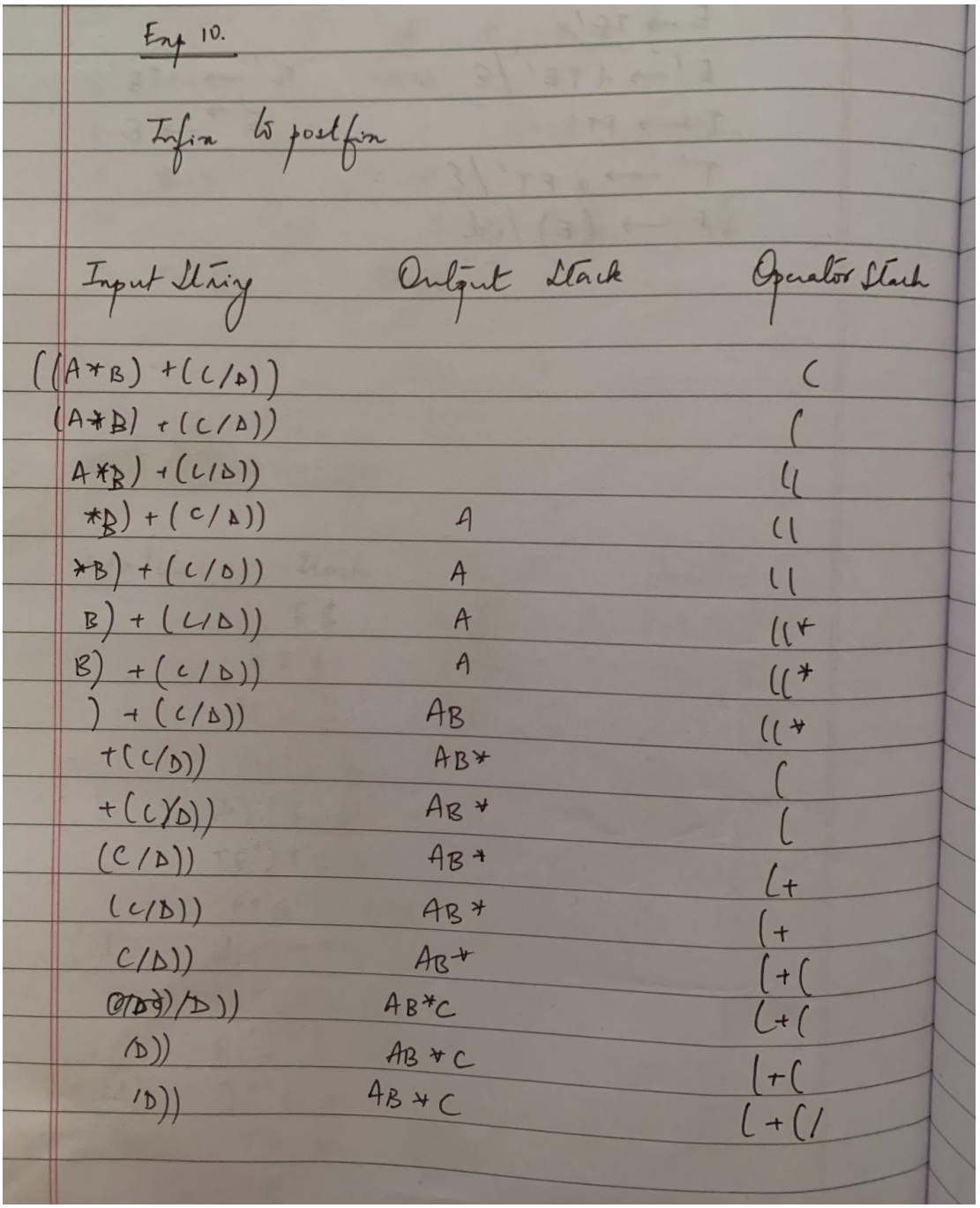
EXPERIMENT NO - 10

Intermediate code generation – Postfix, Prefix

AIM:

To generate the prefix and postfix for the given expression.

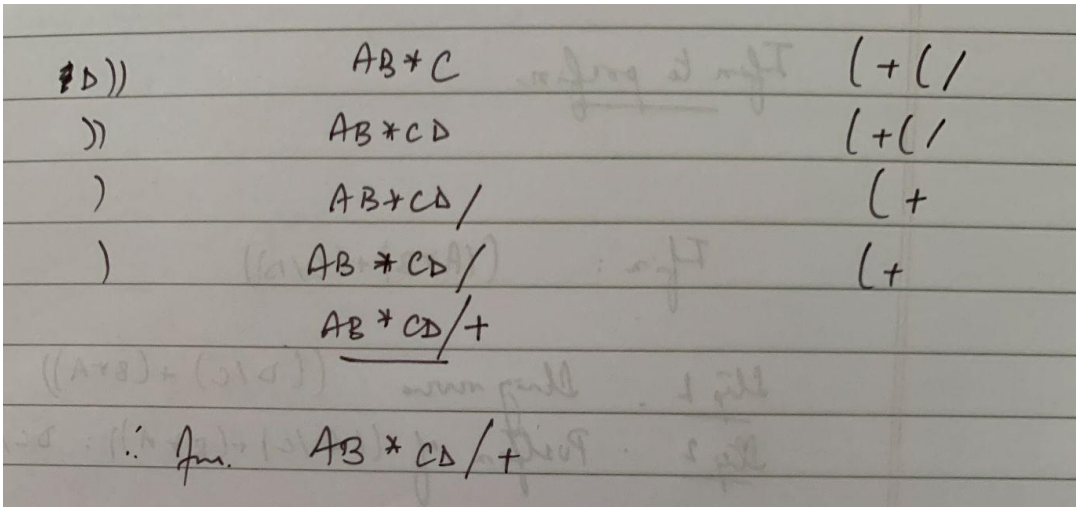
MANUAL CALCULATION:

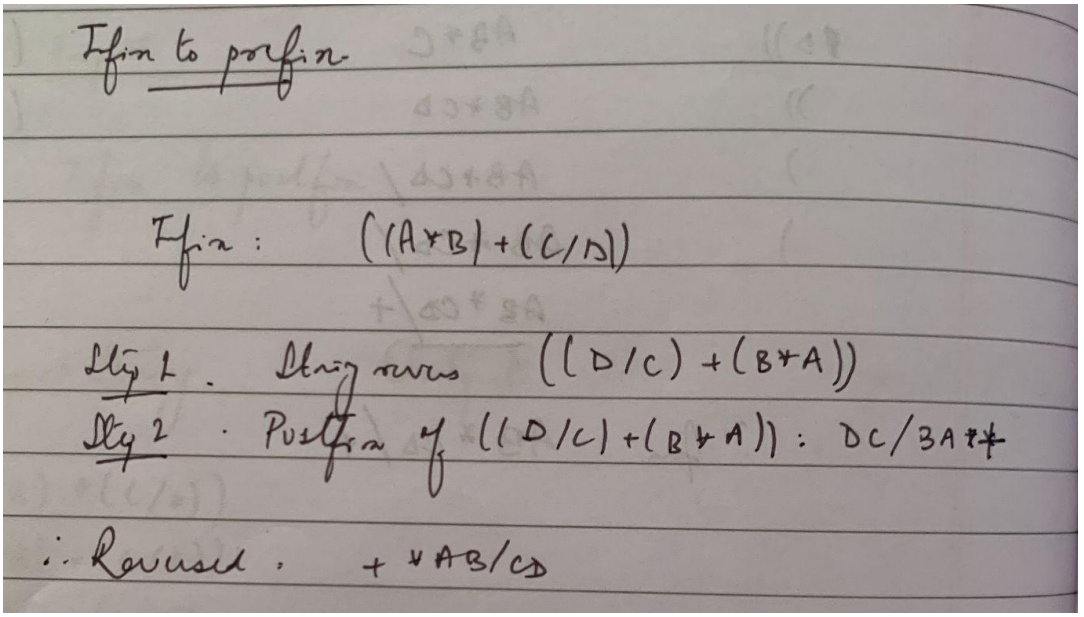


Exp 10.

Infix to postfix

Input String	Output Stack	Operator Stack
((A*B)+(C/D))		(
(A*B)+(C/D))		((
A*B)+(C/D))		(((
*B)+(C/D))	A	((
*B)+(C/D))	A	((
B)+(C/D))	A	((*
B)+(C/D))	A	((*
)+(C/D))	AB	((+
+(C/D))	AB*	(
+(C/D))	AB*	(
(C/D))	AB*	(+
(C/D))	AB*	(+
C/D))	AB*	(+(
/D))	AB*C	(+(
/D))	AB*C	(+(
D))	AB*C	(+(/
D))	AB*C	(+(/





CODE:

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


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


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


def infix_to_postfix(formula):
    stack = []  # only pop when the coming op has priority
```

```
output = ''

for ch in formula:

    if ch not in OPERATORS:

        output += ch

    elif ch == '(':

        stack.append('(')

    elif ch == ')':

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

        stack.pop()  # pop '('

    else:

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

        stack.append(ch)

        # leftover

while stack:
```

```
        output += stack.pop()

    print(f'POSTFIX: {output}')

    return output


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

def infix_to_prefix(formula):
    op_stack = []

    exp_stack = []

    for ch in formula:

        if not ch in OPERATORS:

            exp_stack.append(ch)

        elif ch == '(':

            op_stack.append(ch)

        elif ch == ')':

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

                a = exp_stack.pop()
```

```
            b = exp_stack.pop()

            exp_stack.append(op + b + a)

        op_stack.pop()  # pop '('

    else:

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

            a = exp_stack.pop()

            b = exp_stack.pop()

            exp_stack.append(op + b + a)

        op_stack.append(ch)

        # leftover

while op_stack:
    op = op_stack.pop()

    a = exp_stack.pop()

    b = exp_stack.pop()

    exp_stack.append(op + b + a)
```

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

    return exp_stack[-1]

expres = input("INPUT THE EXPRESSION: ")

pre = infix_to_prefix(expres)

pos = infix_to_postfix(expres)
```

OUTPUT:

```
INPUT THE EXPRESSION: ((A*B)+(C/D))
PREFIX: +*AB/CD
POSTFIX: AB*CD/+
```

RESULT:

The program was successfully executed and output was verified.

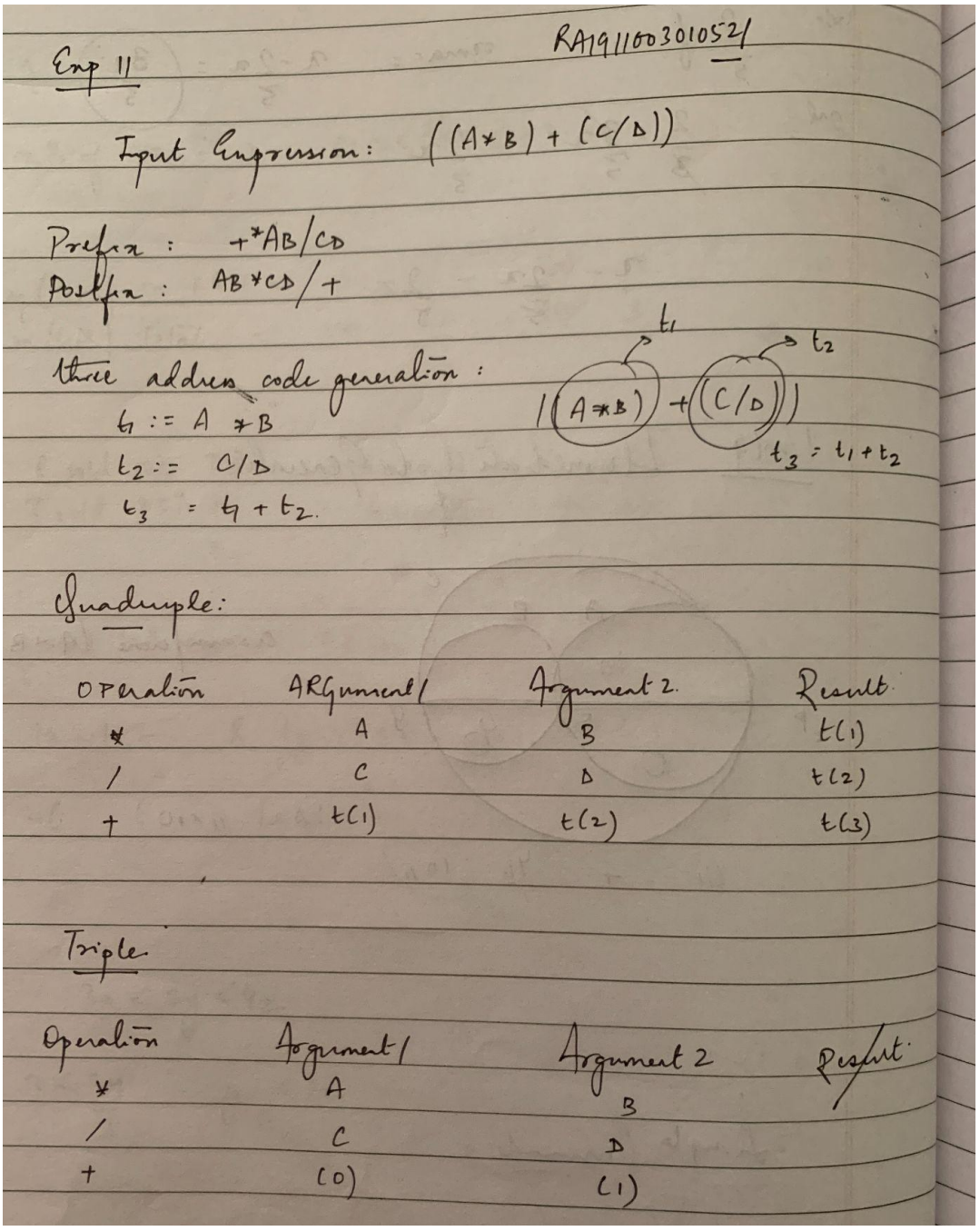
EXPERIMENT NO - 11

Intermediate code generation – Quadruple, Triple, Indirect triple

AIM:

To find the intermediate code generation - Quadruple, triple, indirect.

MANUAL CALCULATION:



Exp 11 RA1911003010521

Input Expression: ((A*B)+(C/D))

Prefix : +*AB/CD

Postfix : AB*CD/+

three address code generation :

$t_1 := A * B$

$t_2 := C / D$

$t_3 = t_1 + t_2$

Quadruple:

Operation	Argument 1	Argument 2	Result
*	A	B	t(1)
/	C	D	t(2)
+	t(1)	t(2)	t(3)

Triple

Operation	Argument 1	Argument 2	Result
*	A	B	
/	C	D	
+	(0)	(1)	

CODE:

```
OPERATORS = set(['+', '-', '*', '/', '(', ')'])
PRI = {'+':1, '-':1, '*':2, '/':2}
### INFIX ===> POSTFIX ###
def infix_to_postfix(formula):
    stack = [] # only pop when the coming op has priority
    output = ''
    for ch in formula:
        if ch not in OPERATORS:
            output += ch
        elif ch == '(':
            stack.append('(')
        elif ch == ')':
            while stack and stack[-1] != '(':
                output += stack.pop()
            stack.pop() # pop '('
        else:
            while stack and stack[-1] != '(' and PRI[ch] <= PRI[stack[-1]]:
                output += stack.pop()
            stack.append(ch)
    # leftover
    while stack:
            output += stack.pop()
    print(f'POSTFIX: {output}')
    return output
### INFIX ===> PREFIX ###
def infix_to_prefix(formula):
    op_stack = []
    exp_stack = []
```

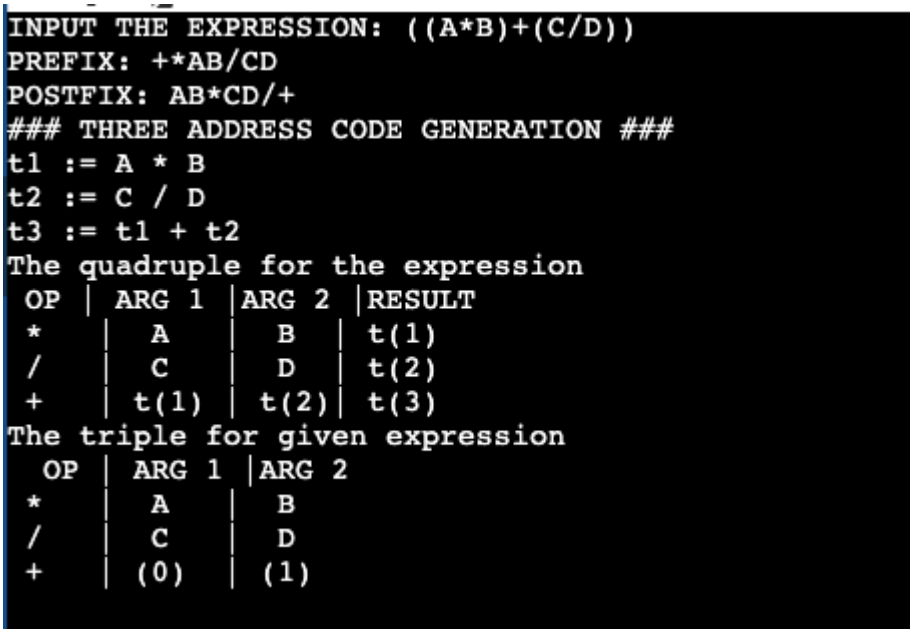
```
    for ch in formula:
        if not ch in OPERATORS:
            exp_stack.append(ch)
        elif ch == '(':
            op_stack.append(ch)
        elif ch == ')':
            while op_stack[-1] != '(':
                op = op_stack.pop()
                a = exp_stack.pop()
                b = exp_stack.pop()
                exp_stack.append( op+b+a )
            op_stack.pop() # pop '('
        else:
            while op_stack and op_stack[-1] != '(' and PRI[ch] <= PRI[op_stack[-1]]:
                op = op_stack.pop()
                a = exp_stack.pop()
                b = exp_stack.pop()
                exp_stack.append( op+b+a )
            op_stack.append(ch)
    # leftover
    while op_stack:
        op = op_stack.pop()
        a = exp_stack.pop()
        b = exp_stack.pop()
        exp_stack.append( op+b+a )
    print(f'PREFIX: {exp_stack[-1]}')
    return exp_stack[-1]
### THREE ADDRESS CODE GENERATION ###
def generate3AC(pos):
            print("### THREE ADDRESS CODE GENERATION ###")
```

```
        exp_stack = []
        t = 1
        for i in pos:
            if i not in OPERATORS:
                exp_stack.append(i)
            else:
                print(f't{t} := {exp_stack[-2]} {i} {exp_stack[-1]}')
                exp_stack=exp_stack[:-2]
                exp_stack.append(f't{t}')
                t+=1
expres = input("INPUT THE EXPRESSION: ")
pre = infix_to_prefix(expres)
pos = infix_to_postfix(expres)
generate3AC(pos)
def Quadruple(pos):
  stack = []
  op = []
  x = 1
  for i in pos:
    if i not in OPERATORS:
      stack.append(i)
    elif i == '-':
      op1 = stack.pop()
      stack.append("t(%s)" %x)
      print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format(i,op1,"(-)"," t(%s)" %x))
      x = x+1
      if stack != []:
        op2 = stack.pop()
        op1 = stack.pop()
        print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format("+",op1,op2," t(%s)" %x))
```

```
            stack.append("t(%s)" %x)
            x = x+1
        elif i == '=':
          op2 = stack.pop()
          op1 = stack.pop()
          print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format(i,op2,"(-)",op1))
        else:
          op1 = stack.pop()
          op2 = stack.pop()
          print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format(i,op2,op1," t(%s)" %x))
          stack.append("t(%s)" %x)
          x = x+1
    print("The quadruple for the expression ")
    print(" OP | ARG 1 |ARG 2 |RESULT  ")
    Quadruple(pos)
    def Triple(pos):
            stack = []
            op = []
            x = 0
            for i in pos:
              if i not in OPERATORS:
                stack.append(i)
              elif i == '-':
                op1 = stack.pop()
                stack.append("(%s)" %x)
                print("{0:^4s} | {1:^4s} | {2:^4s}".format(i,op1,"(-)"))
                x = x+1
                if stack != []:
                  op2 = stack.pop()
                  op1 = stack.pop()
```

```
            print("{0:^4s} | {1:^4s} | {2:^4s}".format("+",op1,op2))
            stack.append("(%s)" %x)
            x = x+1
        elif i == '=':
          op2 = stack.pop()
          op1 = stack.pop()
          print("{0:^4s} | {1:^4s} | {2:^4s}".format(i,op1,op2))
        else:
          op1 = stack.pop()
          if stack != []:
            op2 = stack.pop()
            print("{0:^4s} | {1:^4s} | {2:^4s}".format(i,op2,op1))
            stack.append("(%s)" %x)
            x = x+1
print("The triple for given expression")
print("  OP | ARG 1 |ARG 2  ")
Triple(pos)
```

OUTPUT:



RESULT: The program was successfully executed and output was verified.