|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **18CSC304J – COMPILER DESIGN**  **SEMESTER – VI**  **2021 – 2022 (EVEN)**     |  |  | | --- | --- | | **Name** | **: Anshika Maheshwari** | | **Register No** | **: RA1911003010697** | | **Branch** | **: CSE** | | **Section** | **: C2** |     **DEPARTMENT OF COMPUTING TECHNOLOIGIES**  **SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**  **(Under Section 3 of UGC Act, 1956)**  **S.R.M. NAGAR, KATTANKULATHUR – 603 203**  **CHENGALPATTU DISTRICT** |

**DEPARTMENT OF COMPUTATING TECHNOLIGIES COLLEGE OF ENGINEERING & TECHNOLOGY**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**(Under Section 3 of UGC Act, 1956)**

**S.R.M. NAGAR, KATTANKULATHUR**

**BONAFIDE CERTIFICATE**

**Register No.: RA1911003010697**

Certified to be the bonafide record of work done by **Anshika Maheshwari** of **CSE, B.Tech.** Degree course in the Practical of **18CSC304J – COMPILER DESIGN** in **SRM IST**, Kattankulathur during the academic year **2021 - 2022**.

**Staff In-Charge Head of the Department**

**Date:**

Submitted for University Examination held on \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ at **SRM IST**, Kattankulathur.

**Date:**

**Internal Examiner I Internal Examiner II**

**CONTENTS**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Ex. No.** | **Date** | **Title** | **Page No.** | **Marks** |
| 1 | 24/01/22 | Lexical Analysis |  |  |
| 2 | 01/02/22 | Regular Expression to NFA |  |  |
| 3 | 08/02/22 | Left Recursive Elimination |  |  |
| 4 | 15/02/22 | Left Factoring |  |  |
| 5 | 22/02/22 | First & Follow Computation |  |  |
| 6 | 01/03/22 | Shift Reduce Parsing |  |  |
| 7 | 08/03/22 | Leading & Trailing |  |  |
| 8 | 15/03/22 | SLR Parsing |  |  |
| 9 | 22/03/22 | Predictive Parsing |  |  |
| 10 | 29/03/22 | Postfix and Prefix Implementation |  |  |
| 11 | 05/03/22 | Quadruple & Triple |  |  |

**Exp 1**

**Lexical Analysis**

Anshika Maheshwari

RA1911003010697

**AIM**:

To write a program to implement a lexical analyser.

**ALGORITHM:**

1. Start.

2. Get the input program from the file prog.txt.

3. Read the program line by line and check if each word in a line is a keyword, identifier,

constant or an operator.

4. If the word read is an identifier, assign a number to the identifier and make an entry into

the symbol table stored in sybol.txt.

5. For each lexeme read, generate a token as follows:

a. If the lexeme is an identifier, then the token generated is of the form <id, number>

b. If the lexeme is an operator, then the token generated is <op, operator>.

c. If the lexeme is a constant, then the token generated is <const, value>.

d. If the lexeme is a keyword, then the token is the keyword itself.

6. The stream of tokens generated are displayed in the console output.

7. Stop.

**PROGRAM:**

#include <stdbool.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

*bool* isDelimiter(*char* *ch*)

{

    if (*ch* == ' ' || *ch* == '+' || *ch* == '-' || *ch* == '\*' ||

*ch* == '/' || *ch* == ',' || *ch* == ';' || *ch* == '>' ||

*ch* == '<' || *ch* == '=' || *ch* == '(' || *ch* == ')' ||

*ch* == '[' || *ch* == ']' || *ch* == '{' || *ch* == '}')

        return (true);

    return (false);

}

*bool* isSpecialCharacter(*char* *ch*)

{

    if (*ch* == ',' || *ch* == ';' || *ch* == '>' ||

*ch* == '<' || *ch* == '=' || *ch* == '(' || *ch* == ')' ||

*ch* == '[' || *ch* == ']' || *ch* == '{' || *ch* == '}')

        return (true);

    return (false);

}

*bool* isOperator(*char* *ch*)

{

    if (*ch* == '+' || *ch* == '-' || *ch* == '\*' ||

*ch* == '/' || *ch* == '>' || *ch* == '<' ||

*ch* == '=')

        return (true);

    return (false);

}

*bool* validIdentifier(*char*\* *str*)

{

    if (*str*[0] == '0' || *str*[0] == '1' || *str*[0] == '2' ||

*str*[0] == '3' || *str*[0] == '4' || *str*[0] == '5' ||

*str*[0] == '6' || *str*[0] == '7' || *str*[0] == '8' ||

*str*[0] == '9' || isDelimiter(*str*[0]) == true)

        return (false);

    return (true);

}

*bool* isKeyword(*char*\* *str*)

{

    if (!strcmp(*str*, "if") || !strcmp(*str*, "else") ||

        !strcmp(*str*, "while") || !strcmp(*str*, "do") ||

        !strcmp(*str*, "break") ||

        !strcmp(*str*, "continue") || !strcmp(*str*, "int")

        || !strcmp(*str*, "double") || !strcmp(*str*, "float")

        || !strcmp(*str*, "return") || !strcmp(*str*, "char")

        || !strcmp(*str*, "case") || !strcmp(*str*, "char")

        || !strcmp(*str*, "sizeof") || !strcmp(*str*, "long")

        || !strcmp(*str*, "short") || !strcmp(*str*, "typedef")

        || !strcmp(*str*, "switch") || !strcmp(*str*, "unsigned")

        || !strcmp(*str*, "void") || !strcmp(*str*, "static")

        || !strcmp(*str*, "struct") || !strcmp(*str*, "goto"))

        return (true);

    return (false);

}

*bool* isInteger(*char*\* *str*)

{

*int* i, len = strlen(*str*);

    if (len == 0)

        return (false);

    for (i = 0; i < len; i++) {

        if (*str*[i] != '0' && *str*[i] != '1' && *str*[i] != '2'

            && *str*[i] != '3' && *str*[i] != '4' && *str*[i] != '5'

            && *str*[i] != '6' && *str*[i] != '7' && *str*[i] != '8'

            && *str*[i] != '9' || (*str*[i] == '-' && i > 0))

            return (false);

    }

    return (true);

}

*bool* isRealNumber(*char*\* *str*)

{

*int* i, len = strlen(*str*);

*bool* hasDecimal = false;

    if (len == 0)

        return (false);

    for (i = 0; i < len; i++) {

        if (*str*[i] != '0' && *str*[i] != '1' && *str*[i] != '2'

            && *str*[i] != '3' && *str*[i] != '4' && *str*[i] != '5'

            && *str*[i] != '6' && *str*[i] != '7' && *str*[i] != '8'

            && *str*[i] != '9' && *str*[i] != '.' ||

            (*str*[i] == '-' && i > 0))

            return (false);

        if (*str*[i] == '.')

            hasDecimal = true;

    }

    return (hasDecimal);

}

*char*\* subString(*char*\* *str*, *int* *left*, *int* *right*)

{

*int* i;

*char*\* subStr = (*char*\*)malloc(

                sizeof(*char*) \* (*right* - *left* + 2));

    for (i = *left*; i <= *right*; i++)

        subStr[i - *left*] = *str*[i];

    subStr[*right* - *left* + 1] = '\0';

    return (subStr);

}

*void* parse(*char*\* *str*)

{

*int* left = 0, right = 0;

*int* len = strlen(*str*);

    while (right <= len && left <= right) {

        if (isDelimiter(*str*[right]) == false)

            right++;

        if (isDelimiter(*str*[right]) == true && left == right) {

            if (isOperator(*str*[right]) == true)

                printf("'%c' IS AN OPERATOR\n", *str*[right]);

            else if (isSpecialCharacter(*str*[right]) == true)

                printf("'%c' IS A SPECIAL CHARACTER\n", *str*[right]);

            right++;

            left = right;

        }

        else if (isDelimiter(*str*[right]) == true && left != right

                || (right == len && left != right)) {

*char*\* subStr = subString(*str*, left, right - 1);

            if (isKeyword(subStr) == true)

                printf("'%s' IS A KEYWORD\n", subStr);

            else if (isInteger(subStr) == true)

                printf("'%s' IS AN INTEGER\n", subStr);

            else if (isRealNumber(subStr) == true)

                printf("'%s' IS A REAL NUMBER\n", subStr);

            else if (validIdentifier(subStr) == true

                    && isDelimiter(*str*[right - 1]) == false)

                printf("'%s' IS A VALID IDENTIFIER\n", subStr);

            else if (validIdentifier(subStr) == false

                    && isDelimiter(*str*[right - 1]) == false)

                printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);

            left = right;

        }

    }

    return;

}

*int* main()

{

*char str[100];*

*printf("Enter Input : ");*

*fgets(str, 100 , stdin);*

printf("\nLEXICAL ANALYSIS:\n\n");

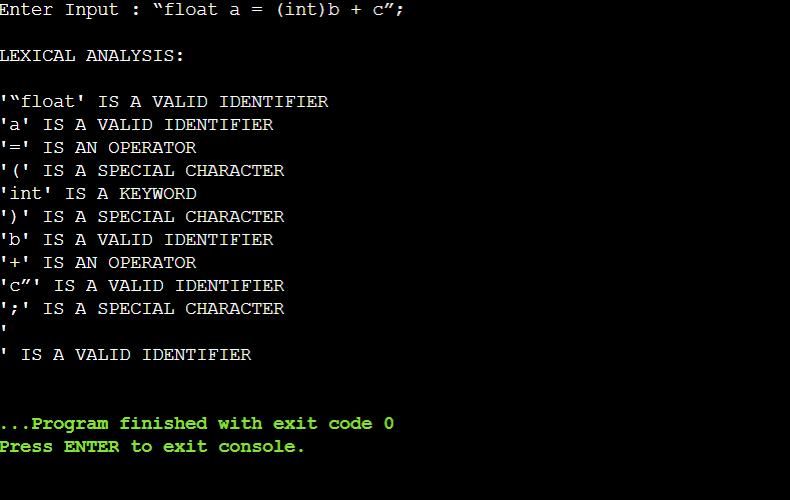
    parse(str);

    return (0);

}

**Input -** “float a = (int)b + c”;

**Output –**



**Result:**

Hence the output for lexical analysis was successfully achieved.

**EXP 2**

**CONVERSION FROM REGULAR EXPRESSION TO NFA**

Anshika Maheshwari

RA1911003010697

**AIM:**

To write a program for converting Regular Expression to NFA.

**ALGORITHM**:

1. Start
2. Get the input from the user
3. Initialize separate variables and functions for Postfix , Display and NFA
4. Create separate methods for different operators like +,\*, .
5. By using Switch case Initialize different cases for the input
6. For ' . ' operator Initialize a separate method by using various stack functions do the same for the other operators like ' \* ' and ' + '.
7. Regular expression is in the form like a.b (or) a+b
8. Display the output
9. Stop

**PROGRAM:**

transition\_table = [ [0]\*3 for \_ in range(20) ]

re = input("Enter the regular expression : ")

re += " "

i = 0

j = 1

while(i<len(re)):

    if re[i] == 'a':

        try:

            if re[i+1] != '|' and re[i+1] !='\*':

                transition\_table[j][0] = j+1

                j += 1

            elif re[i+1] == '|' and re[i+2] =='b':

                transition\_table[j][2]=((j+1)\*10)+(j+3)

                j+=1

                transition\_table[j][0]=j+1

                j+=1

                transition\_table[j][2]=j+3

                j+=1

                transition\_table[j][1]=j+1

                j+=1

                transition\_table[j][2]=j+1

                j+=1

                i=i+2

            elif re[i+1]=='\*':

                transition\_table[j][2]=((j+1)\*10)+(j+3)

                j+=1

                transition\_table[j][0]=j+1

                j+=1

                transition\_table[j][2]=((j+1)\*10)+(j-1)

                j+=1

        except:

            transition\_table[j][0] = j+1

    elif re[i] == 'b':

        try:

            if re[i+1] != '|' and re[i+1] !='\*':

                transition\_table[j][1] = j+1

                j += 1

            elif re[i+1]=='|' and re[i+2]=='a':

                transition\_table[j][2]=((j+1)\*10)+(j+3)

                j+=1

                transition\_table[j][1]=j+1

                j+=1

                transition\_table[j][2]=j+3

                j+=1

                transition\_table[j][0]=j+1

                j+=1

                transition\_table[j][2]=j+1

                j+=1

                i=i+2

            elif re[i+1]=='\*':

                transition\_table[j][2]=((j+1)\*10)+(j+3)

                j+=1

                transition\_table[j][1]=j+1

                j+=1

                transition\_table[j][2]=((j+1)\*10)+(j-1)

                j+=1

        except:

            transition\_table[j][1] = j+1

    elif re[i]=='e' and re[i+1]!='|'and re[i+1]!='\*':

        transition\_table[j][2]=j+1

        j+=1

    elif re[i]==')' and re[i+1]=='\*':

        transition\_table[0][2]=((j+1)\*10)+1

        transition\_table[j][2]=((j+1)\*10)+1

        j+=1

    i +=1

print ("Transition function:")

print("s  a  b  e\n")

for i in range(j):

    if(transition\_table[i][0]!=0):

        print("q[{0},a]-->{1}".format(i,transition\_table[i][0]))

    if(transition\_table[i][1]!=0):

        print("q[{0},b]-->{1}".format(i,transition\_table[i][1]))

    if(transition\_table[i][2]!=0):

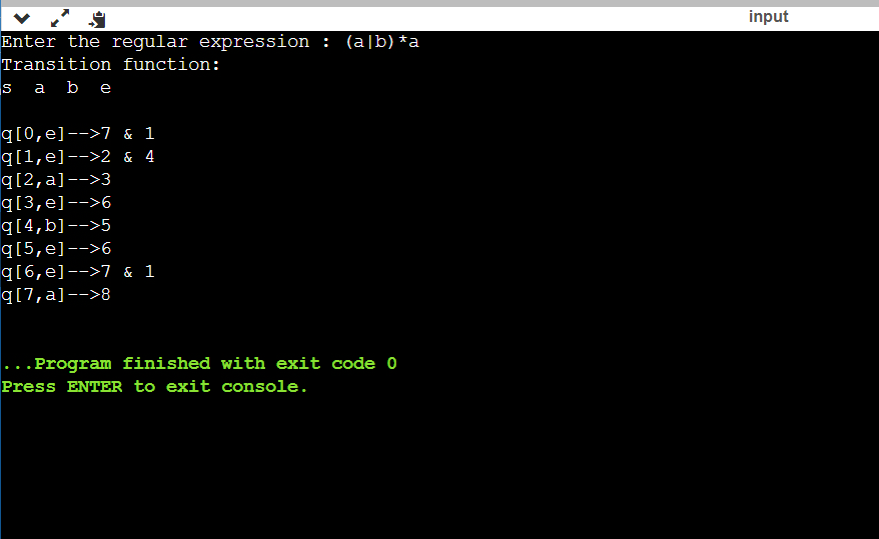
        if(transition\_table[i][2]<10):

            print("q[{0},e]-->{1}".format(i,transition\_table[i][2]))

        else:

            print("q[{0},e]-->{1} & {2}".format(i,int(transition\_table[i][2]/10),transition\_table[i][2]%10))

**Output:**

****

**Result:** Hence theprogram for converting Regular Expression to NFA was successfully executed.

**Exp-3**

**LEFT RECURSION ELIMINATION**

Anshika Maheshwari

RA1911003010697

**AIM:**

A program for Elimination of Left Recursion.

**ALGORITHM:**

1. Start the program.

2. Initialize the arrays for taking input from the user.

3. Prompt the user to input the no. of non-terminals having left recursion and no. of productions for these non-terminals.

4. Prompt the user to input the production for non-terminals.

5. Eliminate left recursion using the following rules:- A->Aα1| Aα2 | . . . . . |Aαm A->β1| β2| . . . . .| βn Then replace it by A-> βi A’ i=1,2,3,…..m A’-> αj A’ j=1,2,3,…..n A’-> Ɛ 6. After eliminating the left recursion by applying these rules, display the productions without left recursion.

7. Stop.

**Code:-**

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<conio.h>

void main()

{

    char a[10],b[50][10]={""},d[50][10]={""},ch;

    int i,n,c[10]={0},j,k,t,n1;

    printf("\nEnter the left production(s) (NON TERMINALS) : ");

    scanf("%s",a);

    n=strlen(a);

    for(i=0;i<n;i++)

    {

        printf("\nEnter the number of productions for %c : ",a[i]);

        scanf("%d",&c[i]);

    }

    t=0;

    for(i=0;i<n;i++)

    {

        printf("\nEnter the right productions for %c",a[i]);

        k=t;

        for(j=0;j<c[i];j++)

        {

            printf("\n%c->",a[i]);

            do

            {

                scanf("%s",b[k]);

                k++;

            }while(k<j);

        }

        t=t+10;

    }

    t=0;

    for(i=0;i<n;i++)

    {

        if(a[i]==b[t][0])

        {

            n1=strlen(b[t]);

            for(k=1;k<n1;k++)

            {

                d[t][k-1]=b[t][k];

            }

        }

        t=t+10;

    }

    t=0;

    printf("\n\nThe resulting productions after eliminating Left Recursion are : \n");

    for(i=0;i<n;i++)

    {

        if(a[i]==b[t][0])

        {

            for(j=1;j<c[i];j++)

            {

                printf("\n%c -> %s%c'",a[i],b[t+j],a[i]);

            }

        }

        t=t+10;

    }

    t=0;

    for(i=0;i<n;i++)

    {

        if(a[i]==b[t][0])

            printf("\n%c' -> %s%c'|%c",a[i],d[t],a[i],(char)238);

        else

            for(j=0;j<c[i];j++)

                printf("\n%c -> %s",a[i],b[t+j]);

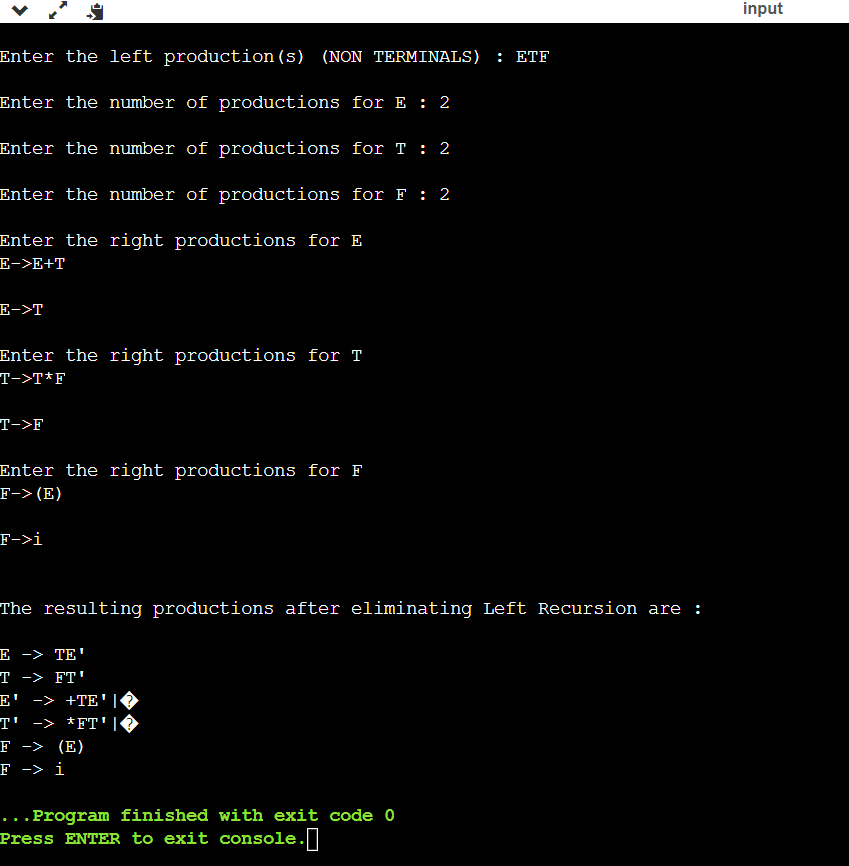
        t=t+10;

    }

    getch();

}

**Output**



**RESULT** :

Hence, program for Elimination of Left Recursion was run successfully and executed.

**EXP-4**

**LEFT FACTORING**

Anshika Maheshwari

RA1911003010697

**AIM :**

A program for implementation Of Left Factoring

**ALGORITHM :**

1. Start

2. Ask the user to enter the set of productions

3. Check for common symbols in the given set of productions by comparing with: A->aB1|aB2

4. If found, replace the particular productions with: A->aA’ A’->B1 | B2|ɛ

5. Display the output

6. Exit

**CODE:**

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<conio.h>

void main()

{

    char ch,lhs[20][20],rhs[20][20][20],temp[20],temp1[20];

    int n,n1,count[20],x,y,i,j,k,c[20];

    printf("\nEnter the no. of productions : ");

    scanf("%d",&n);

    n1=n;

    for(i=0;i<n;i++)

    {

        printf("\nProduction %d \nEnter the no. of productions : ",i+1);

        scanf("%d",&c[i]);

        printf("\nEnter LHS : ");

        scanf("%s",lhs[i]);

        for(j=0;j<c[i];j++)

        {

            printf("%s->",lhs[i]);

            scanf("%s",rhs[i][j]);

        }

    }

    for(i=0;i<n;i++)

    {

        count[i]=1;

        while(memcmp(rhs[i][0],rhs[i][1],count[i])==0)

            count[i]++;

    }

    for(i=0;i<n;i++)

    {

        count[i]--;

        if(count[i]>0)

        {

            strcpy(lhs[n1],lhs[i]);

            strcat(lhs[i],"'");

            for(k=0;k<count[i];k++)

                temp1[k] = rhs[i][0][k];

            temp1[k++] = '\0';

            for(j=0;j<c[i];j++)

            {

                for(k=count[i],x=0;k<strlen(rhs[i][j]);x++,k++)

                    temp[x] = rhs[i][j][k];

                temp[x++] = '\0';

                if(strlen(rhs[i][j])==1)

                    strcpy(rhs[n1][1],rhs[i][j]);

                strcpy(rhs[i][j],temp);

            }

            c[n1]=2;

            strcpy(rhs[n1][0],temp1);

            strcat(rhs[n1][0],lhs[n1]);

            strcat(rhs[n1][0],"'");

            n1++;

        }

    }

    printf("\n\nThe resulting productions are : \n");

    for(i=0;i<n1;i++)

    {

        if(i==0)

            printf("\n %s -> %c|",lhs[i],(char)238);

        else

            printf("\n %s -> ",lhs[i]);

        for(j=0;j<c[i];j++)

        {

            printf(" %s ",rhs[i][j]);

            if((j+1)!=c[i])

                printf("|");

        }

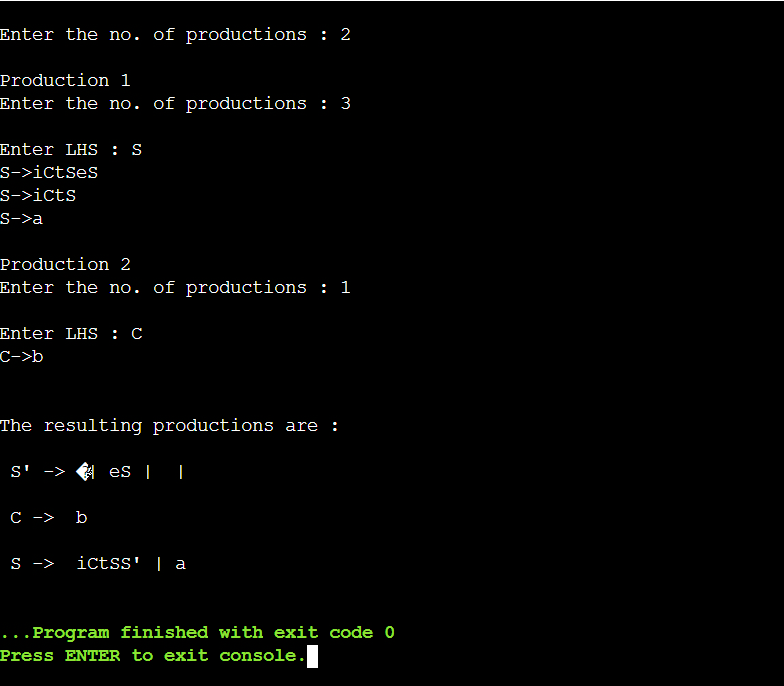
        printf("\b\b\b\n");

    }

    getch();

}

**OUTPUT:**

****

**Result:**

Hence the program for left factoring was successfully executed.

**EXP-5**

**FIRST AND FOLLOW**

Anshika Maheshwari

RA1911003010697

**Aim:**

A program to implement First and Follow

**Algorithm:-**

For computing the first:

1. If X is a terminal then FIRST(X) = {X} Example: F -> (E) | id We can write it as FIRST(F) -> { ( , id }

2. If X is a non terminal like E -> T then to get FIRST(E) substitute T with other productions until you get a terminal as the first symbol

3. If X -> ε then add ε to FIRST(X).

For computing the follow:

1. Always check the right side of the productions for a non-terminal, whose FOLLOW set is being found. ( never see the left side ).

2. (a) If that non-terminal (S,A,B…) is followed by any terminal (a,b…,\*,+,(,)…) , then add that “terminal” into FOLLOW set.

(b) If that non-terminal is followed by any other non-terminal then add “FIRST of other nonterminal” into FOLLOW set.

**Code**

#include<stdio.h>

#include<string.h>

#include<conio.h>

#define max 20

char prod[max][10];

char ter[10],nt[10];

char first[10][10],follow[10][10];

int eps[10];

int count=0;

int findpos(char *ch*)

{

int n;

for(n=0;nt[n]!='\0';n++)

if(nt[n]==*ch*)

break;

if(nt[n]=='\0')

return 1;

return n;

}

int IsCap(char *c*)

{

if(*c* >= 'A' && *c*<= 'Z')

return 1;

return 0;

}

void add(char \**arr*,char *c*)

{

int i,flag=0;

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

{

if(*arr*[i] == *c*)

{

flag=1;

break;

}

}

if(flag!=1)

*arr*[strlen(*arr*)] = *c*;

}

void addarr(char \**s1*,char \**s2*)

{

int i,j,flag=99;

for(i=0;*s2*[i]!='\0';i++)

{

flag=0;

for(j=0;;j++)

{

if(*s2*[i]==*s1*[j])

{

flag=1;

break;

}

if(j==strlen(*s1*) && flag!=1)

{

*s1*[strlen(*s1*)] = *s2*[i];

break;

}

}

}

}

void addprod(char \**s*)

{

int i;

prod[count][0] = *s*[0];

for(i=3;*s*[i]!='\0';i++)

{

if(!IsCap(*s*[i]))

add(ter,*s*[i]);

prod[count][i-2] = *s*[i];

}

prod[count][i-2] = '\0';

add(nt,*s*[0]);

count++;

}

void findfirst()

{

int i,j,n,k,e,n1;

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

{

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

{

n = findpos(prod[j][0]);

if(prod[j][1] == (char)238)

eps[n] = 1;

else

{

for(k=1,e=1;prod[j][k]!='\0' && e==1;k++)

{

if(!IsCap(prod[j][k]))

{

e=0;

add(first[n],prod[j][k]);

}

else

{

n1 = findpos(prod[j][k]);

addarr(first[n],first[n1]);

if(eps[n1] == 0)

e=0;

}

 }

if(e==1)

eps[n]=1;

}

}

}

}

void findfollow()

{

int i,j,k,n,e,n1;

n = findpos(prod[0][0]);

add(follow[n],'#');

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

{

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

{

k = strlen(prod[j])-1;

for(;k>0;k--)

{

if(IsCap(prod[j][k]))

{

n=findpos(prod[j][k]);

if(prod[j][k+1] == '\0')

{

n1 = findpos(prod[j][0]);

addarr(follow[n],follow[n1]);

}

if(IsCap(prod[j][k+1]))

{

n1 = findpos(prod[j][k+1]);

addarr(follow[n],first[n1]);

if(eps[n1]==1)

{

n1=findpos(prod[j][0]);

addarr(follow[n],follow[n1]);

}

}

else if(prod[j][k+1] != '\0')

add(follow[n],prod[j][k+1]);

}

}

}

}

}

void main()

{

char s[max],i;

printf("\nEnter the productions(type 'end' at the last of the production)\n");

scanf("%s",s);

while(strcmp("end",s))

{

addprod(s);

scanf("%s",s);

}

findfirst();

findfollow();

for(i=0;i<strlen(nt);i++)

{

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

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

if(eps[i]==1)

printf("%c\t",(char)238);

else

printf("\t");

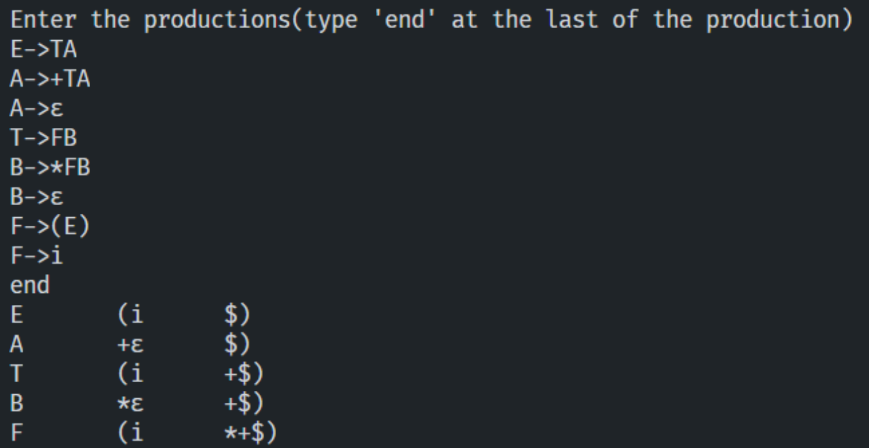
printf("%s\n",follow[i]);

}

getch();

}

**Output**

****

**Result**

Hence, the first and follow sets of non-terminals of a grammar were found successfully.

**EXP-6**

**Shift Reduce Parsing**

**Anshika Maheshwari**

**RA1911003010697**

**Aim:**

To implement shift reduce parser using any programming language.

**Process:**

* Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.
* Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string.
* Sift reduce parsing performs the two actions: shift and reduce. That's why it is known as shift reduces parsing.
* At the shift action, the current symbol in the input string is pushed to a stack.
* At each reduction, the symbols will be replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production.

**Program:**

#include<stdio.h>

#include<string.h>

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

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

*void* check();

*int* main()

{

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

    puts("enter input string ");

    gets(a);

    c=strlen(a);

    strcpy(act,"SHIFT->");

    puts("stack \t input \t action");

    for(k=0,i=0; j<c; k++,i++,j++)

    {

        if(a[j]=='i' && a[j+1]=='d')

        {

            stk[i]=a[j];

            stk[i+1]=a[j+1];

            stk[i+2]='\0';

            a[j]=' ';

            a[j+1]=' ';

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

            check();

        }

        else

        {

            stk[i]=a[j];

            stk[i+1]='\0';

            a[j]=' ';

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

            check();

        }

    }

}

*void* check()

{

    strcpy(ac,"REDUCE TO E");

    for(z=0; z<c; z++)

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

        {

            stk[z]='E';

            stk[z+1]='\0';

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

            j++;

        }

    for(z=0; z<c; z++)

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

        {

            stk[z]='E';

            stk[z+1]='\0';

            stk[z+2]='\0';

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

            i=i-2;

        }

    for(z=0; z<c; z++)

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

        {

            stk[z]='E';

            stk[z+1]='\0';

            stk[z+1]='\0';

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

            i=i-2;

        }

    for(z=0; z<c; z++)

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

        {

            stk[z]='E';

            stk[z+1]='\0';

            stk[z+1]='\0';

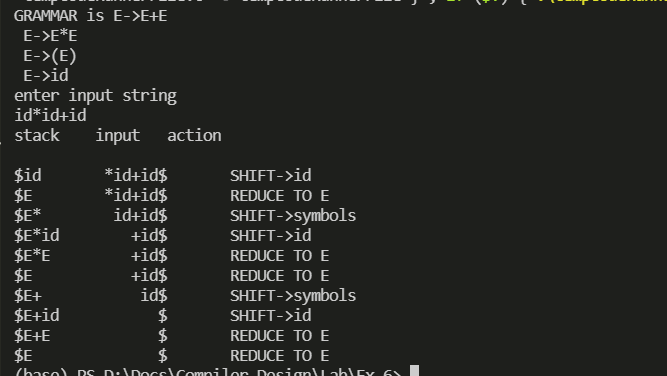
            printf("\n$%s\t%s$\t%s",stk,a,ac);

            i=i-2;

        }

}

**Output:**



**Result:**

Hence, the shift reduce parser is implemented and tested successfully.

**EXP-7**

**LEADING AND TRAILING**

**RA1911003010697**

**Aim:**

To write a program to solve leading and trailing problems.

**Algorithm**

**Input** − Context Free Grammar G

**Output** − LEADING (A) = {a} iff Boolean Array L [A, a] = true

**Method** − Procedure Install (A, a) will make L (A, a) to true if it was not true earlier.

* begin
* For each non-terminal A and terminal a :  L [A, a] = false ;
* For each production of form A ⟶ aα or A → B a α

                            Install (A, a) ;

* While the stack not empty

                            Pop top pair (B, a) form Stack ;

                             For each production of form A → B α

                             Install (A, a);

* end

**Procedure Install (A, a)**

* begin
* If not L [A, a] then

                    L [A, a] = true

                    push (A, a) onto stack.

* end

**Algorithm to compute TRAILING**

**Input** − Context Free Grammar G

**Output** − TRAILING (A) = {a} iff Boolean Array T [A, a] = true

**Method**

* begin
* For each non-terminal A and terminal a

                 T [A, a] = false ;

* For each production of form A ⟶ αa or A → α a B

                  Install (A, a) ;

* While the stack not empty

                Pop top pair (B, a) form Stack ;

                For each production of form A → αB

                Install (A, a);

* end

**Procedure Install (A, a)**

* begin
* If not T [A, a] then

                 T [A, a] = true

                 push (A, a) onto stack.

* end

**Program:-**

#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<<"\n------------- LEADING 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<<"\n------------- PRODUCTION 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();

getch();

}

**Output:-**

------------- LEADING AND TRAILING ---------------

Enter the no. of variables : 3

Enter the variables :

E

T

F

Enter the no. of terminals : 5

Enter the terminals :

(

)

\*

+

i

------------- PRODUCTION DETAILS -----------------

Enter the no. of production of E:2

E->E+T

E->T

Enter the no. of production of T:2

T->T\*F

T->F

Enter the no. of production of F:2

F->(E)

F->i

LEADING(E) = (,\*,+,i,

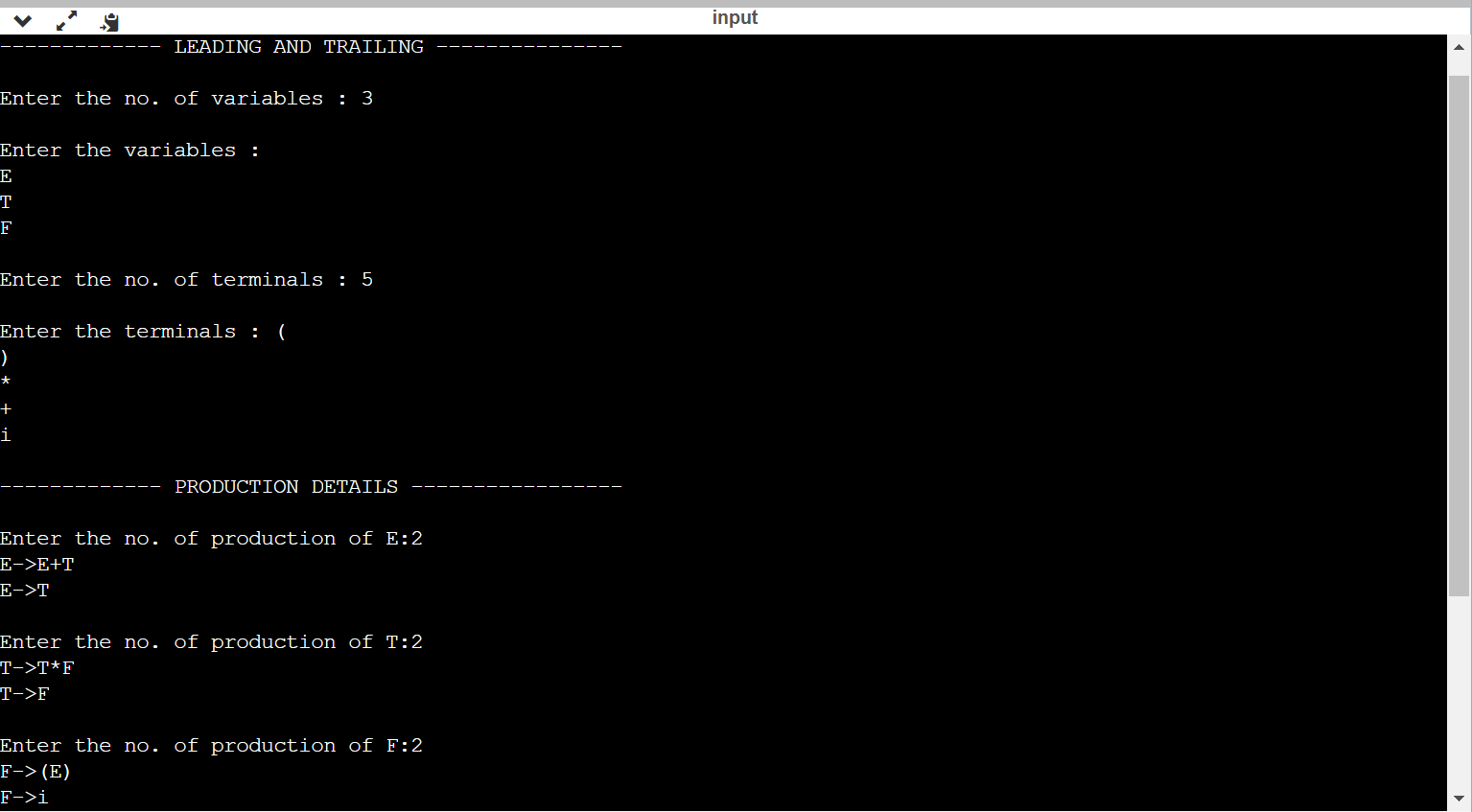
LEADING(T) = (,\*,i,

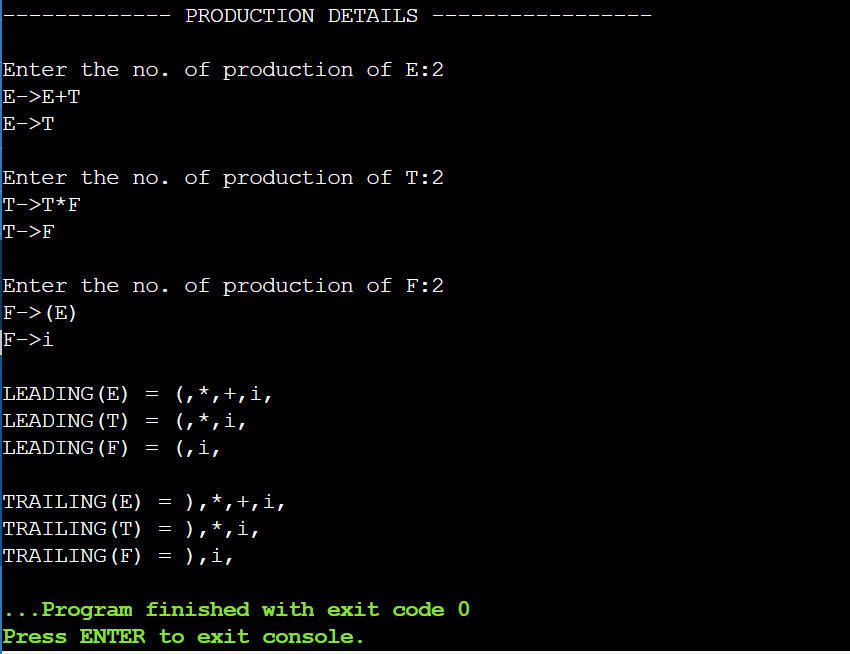
LEADING(F) = (,i,

TRAILING(E) = ),\*,+,i,

TRAILING(T) = ),\*,i,

TRAILING(F) = ),i,





**Result:**

Hence, the program of leading and trailing was successfully executed and implemented.

**EXP-8**

**Computation of LR (0) Items**

**Anshika Maheshwari**

**RA1911003010697**

**Aim:**

A program to implement LR (0) items

**Algorithm:**

1. Start.
2. Create structure for production with LHS and RHS.
3. Open file and read input from file.
4. Build state 0 from extra grammar Law S' -> S $ that is all start symbol of grammar and one Dot (.) before S symbol.
5. If Dot symbol is before a non-terminal, add grammar laws that this non-terminal is in Left Hand Side of that Law and set Dot in before of first part of right-hand Side.
6. If state exists (a state with this Laws and same Dot position), use that instead.
7. Now find set of terminals and non-terminals in which Dot exist in before.
8. If step 7 Set is non-empty go to 9, else go to 10.
9. For each terminal/non-terminal in set step 7 create new state by using all grammar law that Dot position is before of that terminal/non-terminal in reference state by increasing Dot point to next part in Right Hand Side of that laws.
10. Go to step 5.
11. End of state building.
12. Display the output.
13. End.

**Program:**

#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";

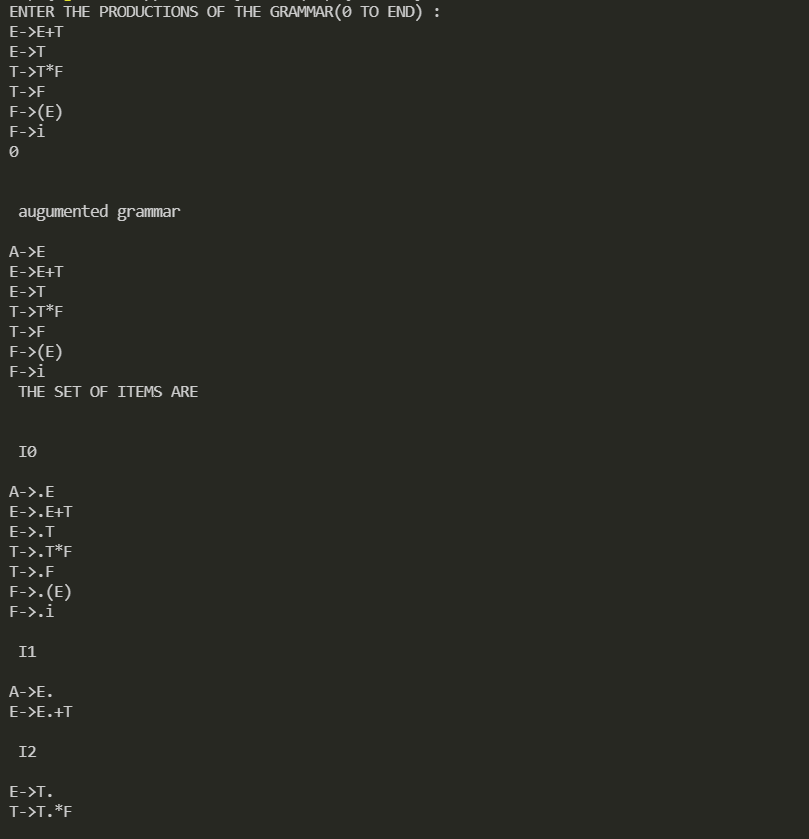
        for(j=0;j<arr[z];j++)

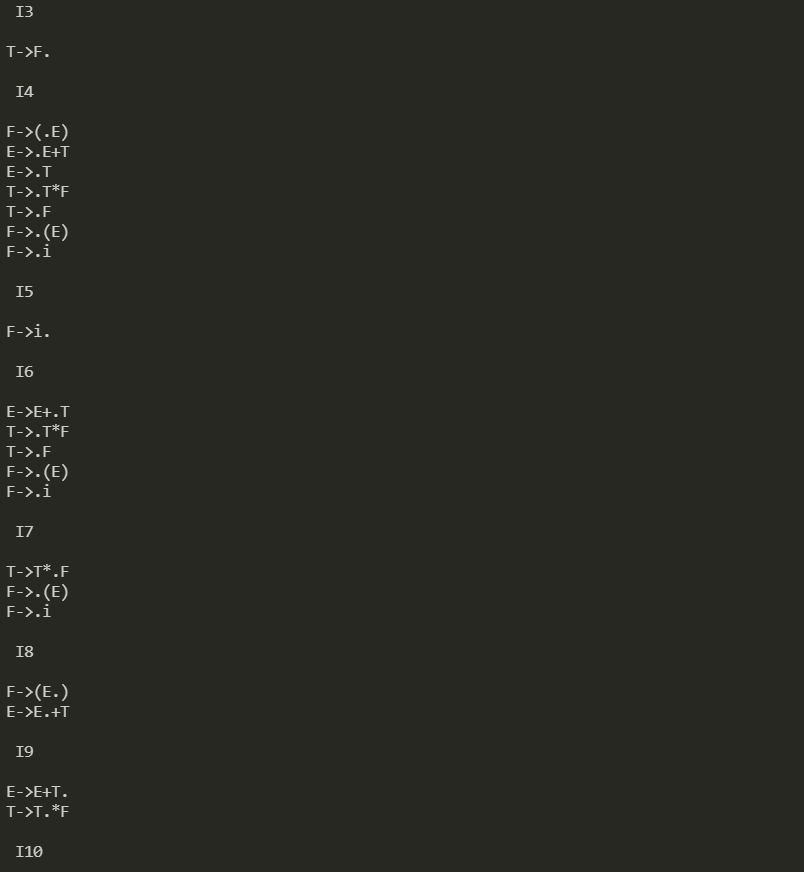
            cout<<clos[z][j].lhs<<"->"<<clos[z][j].rhs<<"\n";

    }

}

**Output:**

****

****

**Result:**

Hence, program for SLR parsing was successfully compiled and run.

**EXP-9**

**PREDICTIVE PARSING**

Anshika Maheshwari

RA1911003010697

**Aim:**

Design a program to construct predictive parsing table.

**Procedure:**

1. Start the program.
2. Initialise required variables.
3. Take CFG as input with each production on a new line in the format A=B.
4. Epsilon should be represented as ‘#’.
5. Compute first and follow for all the non-terminals.
6. For each terminal ‘a’ in FIRST(A), add A->alpha to M[A, a].
7. If epsilon is in FIRST(alpha) and then for each terminal ‘b’ in FOLLOW(A), add A->alpha to M[A, b].
8. Repeat 6 and 7 for all the non-terminals.
9. Stop the program.

**Program:**

#include<stdio.h>

#include<ctype.h>

#include<string.h>

*void* followfirst(*char* , *int* , *int*);

*void* findfirst(*char* , *int* , *int*);

*void* follow(*char* *c*);

*int* count,n=0;

*char* calc\_first[10][100];

*char* calc\_follow[10][100];

*int* m=0;

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

*char* f[10];

*int* k;

*char* ck;

*int* e;

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

{

*int* jm=0;

*int* km=0;

*int* i,choice;

*char* c,ch;

    printf("How many productions ? :");

    scanf("%d",&count);

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

    for(i=0;i<count;i++)

    {

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

    }

*int* kay;

*char* done[count];

*int* ptr = -1;

    for(k=0;k<count;k++){

        for(kay=0;kay<100;kay++){

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

        }

    }

*int* point1 = 0,point2,xxx;

    for(k=0;k<count;k++)

    {

        c=production[k][0];

        point2 = 0;

        xxx = 0;

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

            if(c == done[kay])

                xxx = 1;

        if (xxx == 1)

            continue;

        findfirst(c,0,0);

        ptr+=1;

        done[ptr] = c;

        printf("\n First(%c)= { ",c);

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

        for(i=0+jm;i<n;i++){

*int* lark = 0,chk = 0;

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

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

                    chk = 1;

                    break;

                }

            }

            if(chk == 0){

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

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

            }

        }

        printf("}\n");

        jm=n;

        point1++;

    }

    printf("\n");

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

*char* donee[count];

    ptr = -1;

    for(k=0;k<count;k++){

        for(kay=0;kay<100;kay++){

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

        }

    }

    point1 = 0;

*int* land = 0;

    for(e=0;e<count;e++)

    {

        ck=production[e][0];

        point2 = 0;

        xxx = 0;

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

            if(ck == donee[kay])

                xxx = 1;

        if (xxx == 1)

            continue;

        land += 1;

        follow(ck);

        ptr+=1;

        donee[ptr] = ck;

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

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

        for(i=0+km;i<m;i++){

*int* lark = 0,chk = 0;

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

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

                    chk = 1;

                    break;

                }

            }

            if(chk == 0){

                printf("%c, ",f[i]);

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

            }

        }

        printf(" }\n\n");

        km=m;

        point1++;

    }

*char* ter[10];

    for(k=0;k<10;k++){

        ter[k] = '!';

    }

*int* ap,vp,sid = 0;

    for(k=0;k<count;k++){

        for(kay=0;kay<count;kay++){

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

                vp = 0;

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

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

                        vp = 1;

                        break;

                    }

                }

                if(vp == 0){

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

                    sid ++;

                }

            }

        }

    }

    ter[sid] = '$';

    sid++;

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

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

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

    printf("\t\t\t\t|\t");

    for(ap = 0;ap < sid; ap++){

        printf("%c\t\t",ter[ap]);

    }

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

*char* first\_prod[count][sid];

    for(ap=0;ap<count;ap++){

*int* destiny = 0;

        k = 2;

*int* ct = 0;

*char* tem[100];

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

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

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

                tem[ct++] = '\_';

                tem[ct++] = '\0';

                k++;

                break;

            }

            else{

*int* zap=0;

*int* tuna = 0;

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

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

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

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

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

                            }

                            else

                                break;

                        }

                    break;

                    }

                }

                tem[ct++] = '\_';

            }

            k++;

        }

*int* zap = 0,tuna;

        for(tuna = 0;tuna<ct;tuna++){

            if(tem[tuna] == '#'){

                zap = 1;

            }

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

                if(zap == 1){

                    zap = 0;

                }

                else

                    break;

            }

            else{

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

            }

        }

    }

*char* table[land][sid+1];

    ptr = -1;

    for(ap = 0; ap < land ; ap++){

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

            table[ap][kay] = '!';

        }

    }

    for(ap = 0; ap < count ; ap++){

        ck = production[ap][0];

        xxx = 0;

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

            if(ck == table[kay][0])

                xxx = 1;

        if (xxx == 1)

            continue;

        else{

            ptr = ptr + 1;

            table[ptr][0] = ck;

        }

    }

    for(ap = 0; ap < count ; ap++){

*int* tuna = 0;

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

*int* to,ni=0;

            for(to=0;to<sid;to++){

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

                    ni = 1;

                }

            }

            if(ni == 1){

*char* xz = production[ap][0];

*int* cz=0;

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

                    cz = cz + 1;

                }

*int* vz=0;

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

                    vz = vz + 1;

                }

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

            }

            tuna++;

        }

    }

    for(k=0;k<sid;k++){

        for(kay=0;kay<100;kay++){

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

                break;

            }

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

*int* fz = 1;

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

*char* xz = production[k][0];

*int* cz=0;

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

                        cz = cz + 1;

                    }

*int* vz=0;

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

                        vz = vz + 1;

                    }

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

                    fz++;

                }

                break;

            }

        }

    }

    for(ap = 0; ap < land ; ap++){

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

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

            if(table[ap][kay] == '!')

                printf("\t\t");

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

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

            else{

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

                mum -= 65;

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

            }

        }

        printf("\n");

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

        printf("\n");

    }

}

*void* follow(*char* *c*)

{

*int* i ,j;

    if(production[0][0]==*c*){

        f[m++]='$';

    }

    for(i=0;i<10;i++)

    {

        for(j=2;j<10;j++)

        {

            if(production[i][j]==*c*)

            {

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

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

                }

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

                    follow(production[i][0]);

                }

            }

        }

    }

}

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

{

*int* j;

    if(!(isupper(*c*))){

        first[n++]=*c*;

    }

    for(j=0;j<count;j++)

    {

        if(production[j][0]==*c*)

        {

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

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

                    first[n++]='#';

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

                {

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

                }

                else

                    first[n++]='#';

            }

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

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

            }

            else {

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

            }

        }

    }

}

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

{

*int* k;

    if(!(isupper(*c*)))

        f[m++]=*c*;

    else{

*int* i=0,j=1;

        for(i=0;i<count;i++)

        {

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

                break;

        }

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

        {

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

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

            }

            else{

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

                    follow(production[*c1*][0]);

                }

                else{

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

                }

            }

            j++;

        }

    }

}

**Output:**

**Text

Description automatically generated**

**Result:**

Hence, a program is implemented to construct predictive parsing table.

**Experiment – 10**

**Postfix and Prefix Evaluation**

**Anshika Maheshwari**

**RA1911003010697**

**Postfix Evaluation**

**Aim:**

To evaluate a postfix expression.

**Algorithm:**

1) Create a stack to store operands (or values).

2) Scan the given expression and do the following for every scanned element.

…..a) If the element is a number, push it into the stack

…..b) If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack

3) When the expression is ended, the number in the stack is the final answer

**Program:**

#include <bits/stdc++.h>

using namespace std;

int evaluatePostfix(string *exp*) {

    stack<int> st;

    for(int i = 0; exp[i] != '\0'; i++) {

        if(isdigit(exp[i]))

            st.push(exp[i]-'0');

        else{

            int op2 = st.top();

            st.pop();

            int op1 = st.top();

            st.pop();

            switch (exp[i]) {

                case '+': st.push(op1+op2); break;

                case '-': st.push(op1-op2); break;

                case '\*': st.push(op1\*op2); break;

                case '/': st.push(op1/op2); break;

                case '^': st.push(op1^op2); break;

            }

        }

    }

    return st.top();

}

int main()

{

    string exp;

    cout << "Postfix expression: ";

    cin >> exp;

    cout << "Result: ";

    cout << evaluatePostfix(exp);

    return 0;

}

**Output:**



**Result:**

Hence, the given postfix expression is evaluated.

**Prefix Evaluation**

**Aim:**

To evaluate a postfix expression.

**Algorithm:**

Step 1: Put a pointer P at the end of the end

Step 2: If character at P is an operand push it to Stack

Step 3: If the character at P is an operator pop two

elements from the Stack. Operate on these elements

according to the operator, and push the result

back to the Stack

Step 4: Decrement P by 1 and go to Step 2 as long as there

are characters left to be scanned in the expression.

Step 5: The Result is stored at the top of the Stack,

return it

Step 6: End

**Program:**

#include <bits/stdc++.h>

using namespace std;

int evaluatePrefix(string *exp*) {

    stack<int> st;

    for(int i = *exp*.size()-1; i >= 0; i--) {

        if(isdigit(*exp*[i]))

            st.push(*exp*[i]-'0');

        else {

            int op1 = st.top();

            st.pop();

            int op2 = st.top();

            st.pop();

            switch (*exp*[i]) {

                case '+': st.push(op1+op2); break;

                case '-': st.push(op1-op2); break;

                case '\*': st.push(op1\*op2); break;

                case '/': st.push(op1/op2); break;

                case '^': st.push(op1^op2); break;

            }

        }

    }

    return st.top();

}

int main()

{

    string exp;

    cout << "Prefix expression: ";

    cin >> exp;

    cout << "Result: ";

    cout << evaluatePrefix(exp);

    return 0;

}

**Output:**



**Result:**

Hence, the given prefix expression is evaluated.

**Exp-11**

**Intermediate code generation- Quadruple, Triple, Indirect Triple**

Anshika Maheshwari

RA1911003010697

**Aim:**

A program to implement Intermediate code generation – Quadruple, Triple, Indirect Triple.

**Algorithm:**

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

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

Code:

OPERATORS = set(['+', '-', '\*', '/', '(', ')']) PRI = {'+': 1, '-': 1, '\*': 2, '/': 2} def infix\_to\_postfix(formula):

stack = [] 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()

else:

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

stack.append(ch)

while stack:

output += stack.pop()

return output

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()

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)

while op\_stack:

op = op\_stack.pop() a = exp\_stack.pop() b = exp\_stack.pop() exp\_stack.append(op+b+a)

return exp\_stack[-1] 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

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

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

def IndirectTriple(pos):

stack = [] op = [] x = 0 c = 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} | {3:^5d}".format(i, op1, "(-)", c)) x = x+1 if stack != []:

op2 = stack.pop() op1 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s} | {3:^5d}".format(

"+", op1, op2, c)) stack.append("(%s)" % x) x = x+1 c = c+1 elif i == '=':

op2 = stack.pop() op1 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s} | {3:^5d}".format(i, op1, op2, c)) c = c+1

else:

op1 = stack.pop() if stack != []:

op2 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s} | {3:^5d}".format(

i, op2, op1, c))

stack.append("(%s)" % x) x = x+1 c = c+1

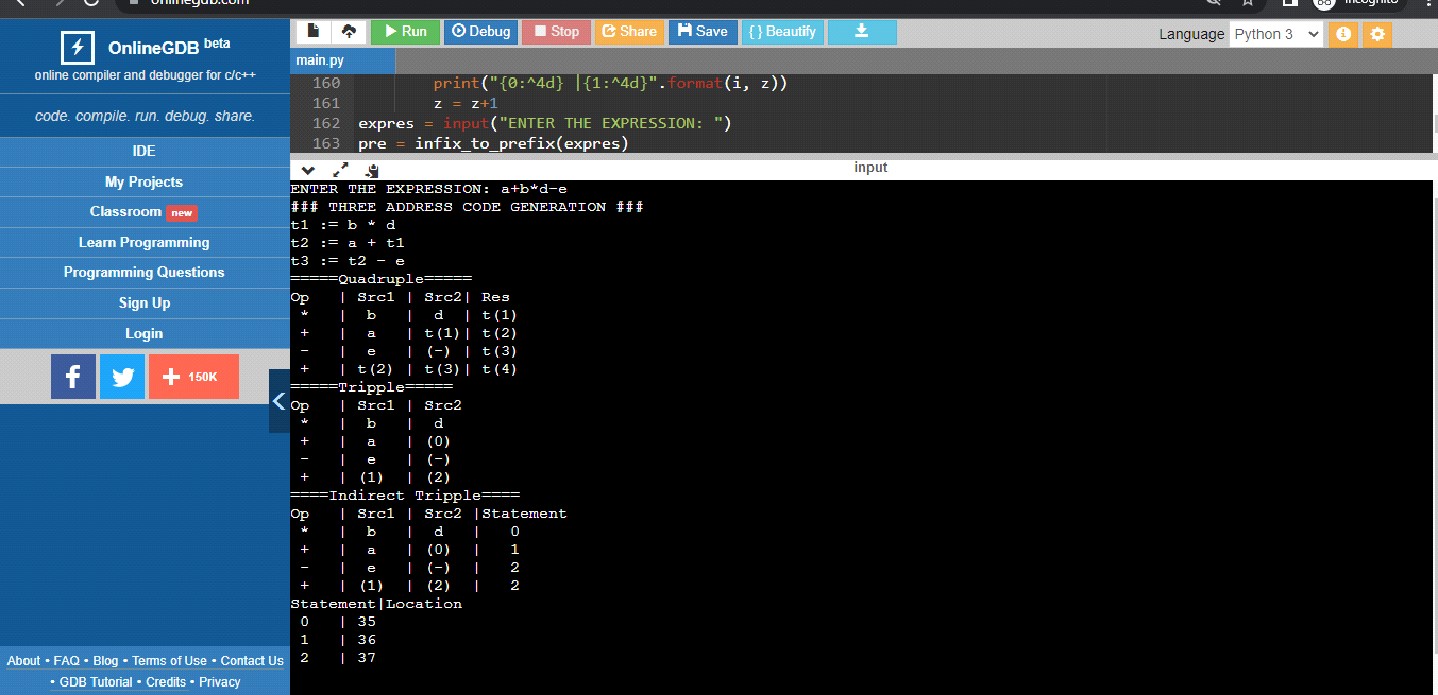
z = 35 print("Statement|Location") for i in range(0, c): print("{0:^4d} |{1:^4d}".format(i, z)) z = z+1

expres = input("ENTER THE EXPRESSION: ") pre = infix\_to\_prefix(expres) pos = infix\_to\_postfix(expres) generate3AC(pos) print("=====Quadruple=====")

print("Op | Src1 | Src2| Res") Quadruple(pos) print("=====Tripple=====") print("Op | Src1 | Src2") Triple(pos) print("====Indirect Tripple====") print("Op | Src1 | Src2 |Statement")

IndirectTriple(pos)

**Output:**



Result:

The program was successfully compiled and run.