

FACULTY OF ENGINEERING AND TECHNOLOGY
SCHOOL OF COMPUTING

COMPUTER SCIENCE & ENGINEERING



LAB CODE: 18CSC304J

LAB NAME: Compiler Design

REGISTER NO: RA1811003010955

NAME: Harsh Vardhan

SRM Institute of Science and Technology
Department of Computer Science and Engineering
18CSC304J-Compiler Design
List of Experiment

| SN | Experiment name | Date of Completion | Page No. |
|----|---|--------------------|----------|
| 1 | Conversion from Regular Expression to NFA | 02-02-2021 | 03 |
| 2 | Conversion from NFA to DFA | 09-02-2021 | 09 |
| 3 | Implementation of Lexical Analyzer | 16-02-2021 | 13 |
| 4a | Elimination of Ambiguity, Left Recursion | 23-02-2021 | 15 |
| 4b | Elimination of Ambiguity, Left Factoring | 02-03-2021 | 19 |
| 5 | FIRST AND FOLLOW computation | 09-03-2021 | 23 |
| 6 | Predictive Parsing Table | 16-03-2021 | 31 |
| 7 | Shift Reduce Parsing | 23-03-2021 | 36 |
| 8 | Computation of LR(0) items | 30-03-2021 | 42 |
| 9 | Computation of LEADING AND TRAILING | 08-04-2021 | 54 |
| 10 | Intermediate code generation – Postfix, Prefix | 16-04-2021 | 64 |
| 11 | Intermediate code generation – Quadruple, Triple, Indirect triple | 23-04-2021 | 69 |
| 12 | Implementation of DAG | 30-04-2021 | 74 |

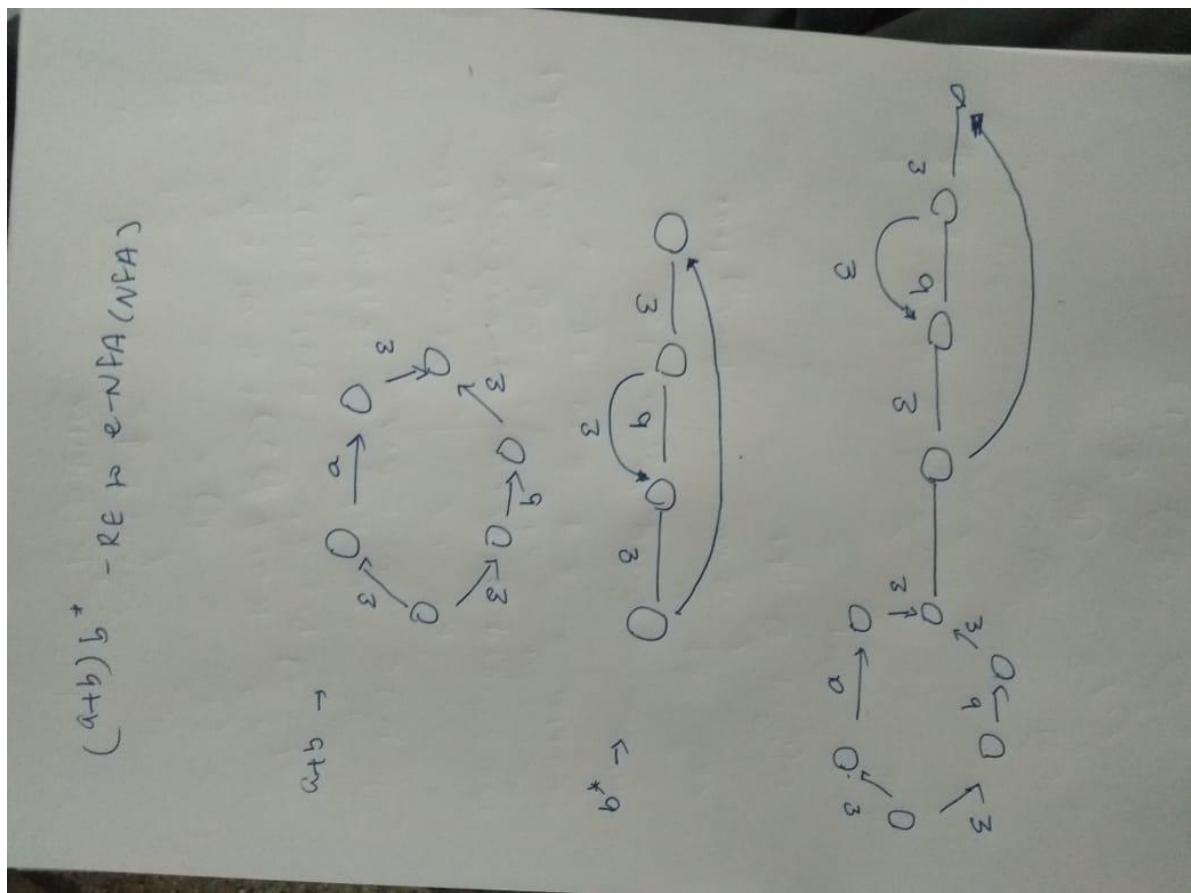
| | |
|------------------|--|
| EX NO:1 | <h2 style="text-align: center;">Conversion from Regular Expression to NFA</h2> |
| DATE: 27/01/2021 | |

AIM : To convert a given regular expression (RE) into NFA

ALGORITHM:

- Step 1: Draw e-NFA for the given expressions individually
- Step 2: Combine both to form NFA for $b(a.b)^*$
- Step 3: Produce NFA transition table for the corresponding expression

Manual Calculation: (Copy paste the calculation done)



PROGRAM :

```

class Type:
    SYMBOL = 1
    CONCAT = 2
    UNION = 3
    KLEENE = 4

class ExpressionTree:

    def __init__(self, _type, value=None):
        self._type = _type
        self.value = value
        self.left = None
        self.right = None

def constructTree(regex):
    stack = []
    for c in regex:
        if c.isalpha():
            stack.append(ExpressionTree(Type.SYMBOL, c))
        else:
            if c == "+":
                z = ExpressionTree(Type.UNION)
                z.right = stack.pop()
                z.left = stack.pop()
            elif c == ".":
                z = ExpressionTree(Type.CONCAT)
                z.right = stack.pop()
                z.left = stack.pop()
            elif c == "*":
                z = ExpressionTree(Type.KLEENE)
                z.left = stack.pop()
                stack.append(z)

    return stack[0]

def inorder(et):
    if et._type == Type.SYMBOL:

```

```

    print(et.value)
elif et._type == Type.CONCAT:
    inorder(et.left)
    print(".")
    inorder(et.right)
elif et._type == Type.UNION:
    inorder(et.left)
    print("+")
    inorder(et.right)
elif et._type == Type.KLEENE:
    inorder(et.left)
    print("*")

def higherPrecedence(a, b):
    p = ["+", ".", "*"]
    return p.index(a) > p.index(b)

def postfix(regex):
    # adding dot "." between consecutive symbols
    temp = []
    for i in range(len(regex)):
        if i != 0\
            and (regex[i-1].isalpha() or regex[i-1] == ")" or regex[i-1] == "*")\
            and (regex[i].isalpha() or regex[i] == "("):
            temp.append(".")
        temp.append(regex[i])
    regex = temp

    stack = []
    output = ""

    for c in regex:
        if c.isalpha():
            output = output + c
            continue

        if c == ")":
            while len(stack) != 0 and stack[-1] != "(":
                output = output + stack.pop()
            stack.pop()
        elif c == "(":
            stack.append(c)
        elif c == "*":
            output = output + c
        elif len(stack) == 0 or stack[-1] == "(" or higherPrecedence(c, stack[-1]):
            stack.append(c)
        else:
            while len(stack) != 0 and stack[-1] != "(" and not higherPrecedence(c, stack[-1]):
                output = output + stack.pop()
            stack.append(c)

```

```

while len(stack) != 0:
    output = output + stack.pop()

return output

class FiniteAutomataState:
    def __init__(self):
        self.next_state = { }

def evalRegex(et):
    # returns equivalent E-NFA for given expression tree (representing a Regular
    # Expression)
    if et._type == Type.SYMBOL:
        return evalRegexSymbol(et)
    elif et._type == Type.CONCAT:
        return evalRegexConcat(et)
    elif et._type == Type.UNION:
        return evalRegexUnion(et)
    elif et._type == Type.KLEENE:
        return evalRegexKleene(et)

def evalRegexSymbol(et):
    start_state = FiniteAutomataState()
    end_state = FiniteAutomataState()

    start_state.next_state[et.value] = [end_state]
    return start_state, end_state

def evalRegexConcat(et):
    left_nfa = evalRegex(et.left)
    right_nfa = evalRegex(et.right)

    left_nfa[1].next_state['epsilon'] = [right_nfa[0]]
    return left_nfa[0], right_nfa[1]

def evalRegexUnion(et):
    start_state = FiniteAutomataState()
    end_state = FiniteAutomataState()

    up_nfa = evalRegex(et.left)
    down_nfa = evalRegex(et.right)

    start_state.next_state['epsilon'] = [up_nfa[0], down_nfa[0]]
    up_nfa[1].next_state['epsilon'] = [end_state]
    down_nfa[1].next_state['epsilon'] = [end_state]

    return start_state, end_state

def evalRegexKleene(et):
    start_state = FiniteAutomataState()
    end_state = FiniteAutomataState()

```

```

sub_nfa = evalRegex(et.left)

start_state.next_state['epsilon'] = [sub_nfa[0], end_state]
sub_nfa[1].next_state['epsilon'] = [sub_nfa[0], end_state]

return start_state, end_state

def printStateTransitions(state, states_done, symbol_table):
    if state in states_done:
        return

    states_done.append(state)

    for symbol in list(state.next_state):
        line_output = "q" + str(symbol_table[state]) + "\t\t" + symbol + "\t\t\t"
        for ns in state.next_state[symbol]:
            if ns not in symbol_table:
                symbol_table[ns] = 1 + sorted(symbol_table.values())[-1]
            line_output = line_output + "q" + str(symbol_table[ns]) + " "

        print(line_output)

        for ns in state.next_state[symbol]:
            printStateTransitions(ns, states_done, symbol_table)

def printTransitionTable(finite_automata):
    print("State\t\tSymbol\t\t\tNext state")
    printStateTransitions(finite_automata[0], [], {finite_automata[0]:0})

r = input("Enter regex: ")
pr = postfix(r)
et = constructTree(pr)

#inorder(et)

fa = evalRegex(et)
printTransitionTable(fa)
OUTPUT

```

```

Enter regex: (a+b)b*
State      Symbol      Next state
q0         epsilon     q1 q2
q1         a           q3
q3         epsilon     q4
q4         epsilon     q5
q5         epsilon     q6 q7
q6         b           q8
q8         epsilon     q6 q7
q2         b           q9
q9         epsilon     q4

```

RESULT: Program to convert RE to NFA was written and executed successfully using the given example.

| | |
|-------------------------|-----------------------------------|
| EX NO:2 | Conversion from NFA to DFA |
| DATE: 03/02/2021 | |

AIM : To convert a given NFA into DFA..

ALGORITHM:

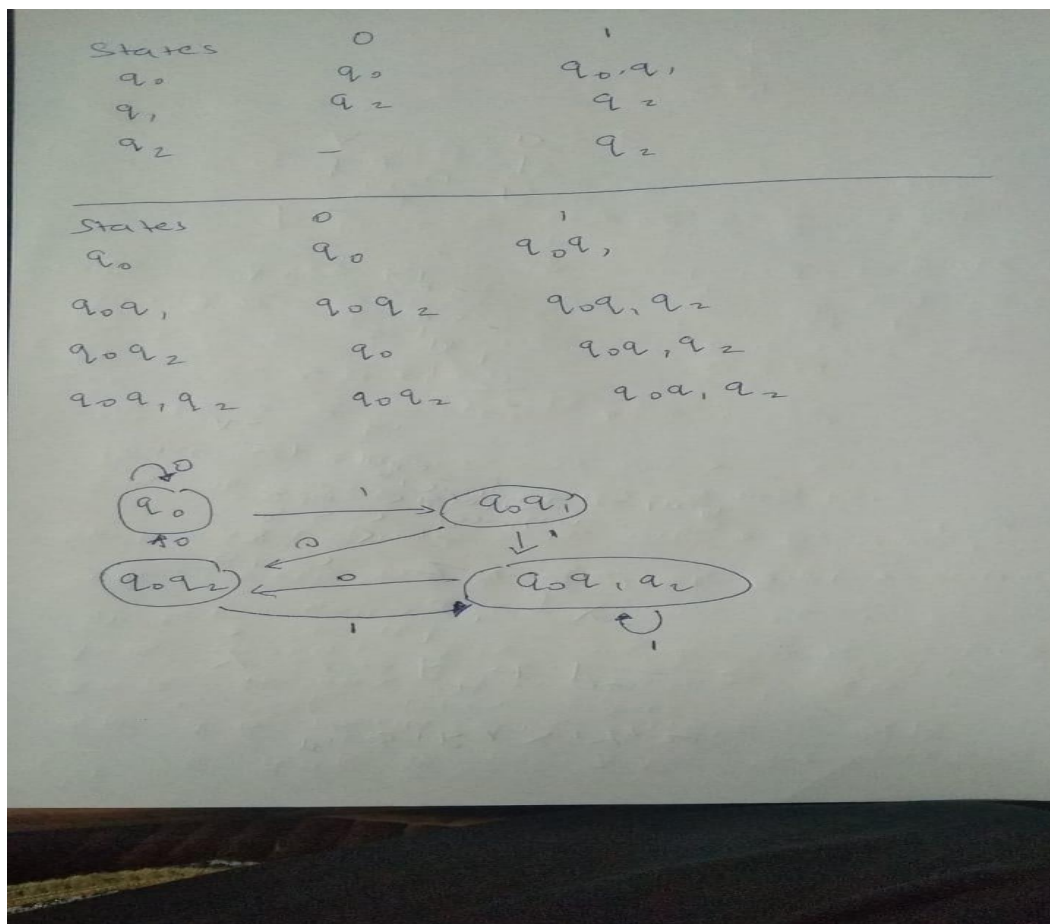
Step 1: Initially $Q' = \phi$.

Step 2: Add q_0 to Q' .

Step 3: For each state in Q' , find the possible set of states for each input symbol using the transition function of NFA. If this set of states is not in Q' , add it to Q' .

Step 4: Final state of DFA will be all states with contain F (final states of NFA)

Manual Calculation: (Copy paste the calculation done)



PROGRAM :

```
import pandas as pd
```

Taking NFA input from User

```
nfa = {}
n = int(input("No. of states : "))      #Enter total no. of states
t = int(input("No. of transitions : "))  #Enter total no. of transitions/paths eg: a,b so input 2 for a,b,c
input 3
for i in range(n):
    state = input("state name : ")      #Enter state name eg: A, B, C, q1, q2 ..etc
    nfa[state] = {}                    #Creating a nested dictionary
    for j in range(t):
        path = input("path : ")        #Enter path eg : a or b in {a,b} 0 or 1 in {0,1}
        print("Enter end state from state { } travelling through path { } : ".format(state,path))
        reaching_state = [x for x in input().split()] #Enter all the end states that
        nfa[state][path] = reaching_state #Assigning the end states to the paths in dictionary
```

```
print("\nNFA :- \n")
print(nfa)                            #Printing NFA
print("\nPrinting NFA table :- ")
nfa_table = pd.DataFrame(nfa)
print(nfa_table.transpose())
```

```
print("Enter final state of NFA : ")
nfa_final_state = [x for x in input().split()] # Enter final state/states of NFA
#####
```

```
new_states_list = []                  #holds all the new states created in dfa
dfa = {}                             #dfa dictionary/table or the output structure we needed
keys_list = list(list(nfa.keys())[0]) #contains all the states in nfa plus the states created in
dfa are also appended further
path_list = list(nfa[keys_list[0]].keys()) #list of all the paths eg: [a,b] or [0,1]
```

#####

Computing first row of DFA transition table

```
dfa[keys_list[0]] = {}               #creating a nested dictionary in dfa
for y in range(t):
    var = "".join(nfa[keys_list[0]][path_list[y]]) #creating a single string from all the elements of the
list which is a new state
    dfa[keys_list[0]][path_list[y]] = var          #assigning the state in DFA table
    if var not in keys_list:                       #if the state is newly created
        new_states_list.append(var)               #then append it to the new_states_list
        keys_list.append(var)                     #as well as to the keys_list which contains all the states
```

#####

Computing the other rows of DFA transition table

```
while len(new_states_list) != 0:      #condition is true only if the new_states_list is not empty
    dfa[new_states_list[0]] = {}      #taking the first element of the new_states_list and
examining it
    for _ in range(len(new_states_list[0])):
        for i in range(len(path_list)):
```

```

temp = []                                #creating a temporay list
for j in range(len(new_states_list[0])):
    temp += nfa[new_states_list[0][j]][path_list[i]] #taking the union of the states
s = ""
s = s.join(temp)                         #creating a single string(new state) from all the elements of the
list
if s not in keys_list:                   #if the state is newly created
    new_states_list.append(s)           #then append it to the new_states_list
    keys_list.append(s)                 #as well as to the keys_list which contains all the states
    dfa[new_states_list[0]][path_list[i]] = s #assigning the new state in the DFA table

new_states_list.remove(new_states_list[0]) #Removing the first element in the new_states_list

print("\nDFA :- \n")
print(dfa)                             #Printing the DFA created
print("\nPrinting DFA table :- ")
dfa_table = pd.DataFrame(dfa)
print(dfa_table.transpose())

dfa_states_list = list(dfa.keys())
dfa_final_states = []
for x in dfa_states_list:
    for i in x:
        if i in nfa_final_state:
            dfa_final_states.append(x)
            break

print("\nFinal states of the DFA are : ",dfa_final_states) #Printing Final states of DFA

```

OUTPUT

```

No. of states : 4
No. of transitions : 2
state name : a
path : 0
Enter end state from state a travelling through path 0 :
a b
path : 1
Enter end state from state a travelling through path 1 :
a c
state name : b
path : 0
Enter end state from state b travelling through path 0 :
d
path : 1
Enter end state from state b travelling through path 1 :

state name : c
path : 0
Enter end state from state c travelling through path 0 :

path : 1
Enter end state from state c travelling through path 1 :
d
state name : d
path : 0
Enter end state from state d travelling through path 0 :

path : 1
Enter end state from state d travelling through path 1 :

NFA :-
{'a': {'0': ['a', 'b'], '1': ['a', 'c']}, 'b': {'0': ['d'], '1': []}, 'c': {'0': [], '1': ['d']}, 'd': {'0': [], '1': []}}
Printing NFA table :-
      0      1
a  [a, b]  [a, c]
b    [d]    []
c     []    [d]
d     []    []
Enter final state of NFA :
d

DFA :-
{'a': {'0': 'ab', '1': 'ac'}, 'ab': {'0': 'abd', '1': 'ac'}, 'ac': {'0': 'ab', '1': 'acd'}, 'abd': {'0': 'abd', '1': 'ac'}, 'acd': {'0': 'ab', '1': 'acd'}}
Printing DFA table :-
      0      1
a    ab    ac
ab  abd    ac
ac   ab   acd
abd abd    ac
acd ab   acd

Final states of the DFA are : ['abd', 'acd']

```

RESULT: Program to convert NFA to DFA was written and executed successfully using the given example.

| | |
|-------------------------|---|
| EX NO:3 | Implementation of Lexical Analyzer |
| DATE: 10/02/2021 | |

AIM : To implement a lexical analyzer based on the given problem. (Printing of lesser of 2 number)

ALGORITHM:

1. Tokenization i.e. Dividing the program into valid tokens.
2. Remove white space characters.
3. Remove comments.
4. It also provides help in generating error messages by providing row numbers and column numbers.

PROGRAM :

code.c

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

int isKeyword(char buffer[]){
char keywords[32][10] = {"auto","break","case","char","const","continue","default",
"do","double","else","enum","extern","float","for","goto",
"if","int","long","register","return","short","signed",
"sizeof","static","struct","switch","typedef","union",
"unsigned","void","volatile","while"};
int i, flag = 0;

for(i = 0; i < 32; ++i){
if(strcmp(keywords[i], buffer) == 0){
flag = 1;
break;
}
}

return flag;
}

int main(){
char ch, buffer[15], operators[] = "+-*/%=";
FILE *fp;
int i,j=0;

fp = fopen("c.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 identifier\n", buffer);
    }
}
fclose(fp);
return 0;

}

c.txt:
void main ( ){
    int x;
    scanf("%d",&x);
    if(x%2)==0{
        printf("even number");
    }
    else
        printf("odd number");
}

```

OUTPUT

```
~$ gcc code.c
~$ ./a.out
void is keyword
main is identifier
int is keyword
x is identifier
% is operator
scanfdx is identifier
% is operator
= is operator
= is operator
ifx20 is identifier
printfeven is identifier
number is identifier
else is keyword
printfodd is identifier
number is identifier
~$ █
```

RESULT: Program to implement a lexical analyzer was written and executed successfully using the given example.

| | |
|-------------------------|--------------------------------------|
| EX NO: 04 -a | ELIMINATION OF LEFT RECURSION |
| DATE: 16-02-2021 | |

AIM : To implement a C++ program to eliminate left recursion from a context free grammar.

ALGORITHM:

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

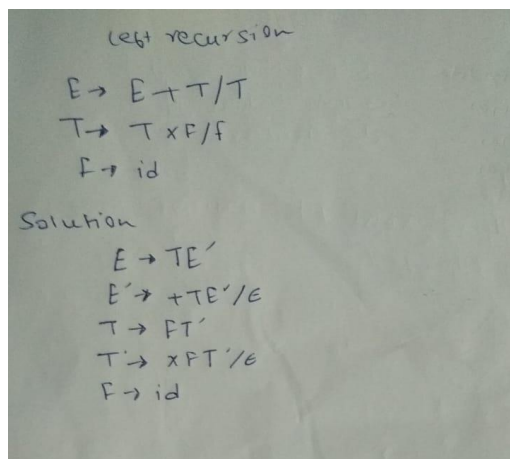
A-
 $\rightarrow A\alpha_1 \mid A\alpha_2$
 $\mid \dots$
 $\mid A\alpha_m$ A-
 $\rightarrow \beta_1 \mid \beta_2 \dots$
 $\dots \mid \beta_n$

Then replace it by

$A' \rightarrow \beta_i A' \quad i=1,2,3,\dots,m$
 $A' \rightarrow \alpha_j A' \quad j=1,2,3,\dots,n$
 $A' \rightarrow \epsilon$

6. After eliminating the left recursion by applying these rules, display the productions without left recursion.
7. Stop.

Manual Calculation:



PROGRAM :


```

#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main()
{
    int n;
    cout<<"\nEnter number of non terminals: ";
    cin>>n;
    cout<<"\nEnter non terminals one by one: ";
    int i;
    vector<string> nonter(n);
    vector<int> leftrecr(n,0);
    for(i=0;i<n;++i) {
        cout<<"\Non terminal "<<i+1<<" : ";
        cin>>nonter[i];
    }
    vector<vector<string> > prod;
    cout<<"\nEnter '^' for null";
    for(i=0;i<n;++i) {
        cout<<"\nNumber of "<<nonter[i]<<" productions: ";
        int k;
        cin>>k;
        int j;
        cout<<"\nOne by one enter all "<<nonter[i]<<"
productions";
        vector<string> temp(k);
        for(j=0;j<k;++j) {
            cout<<"\nRHS of production "<<j+1<<": ";
            string abc;
            cin>>abc;
            temp[j]=abc;
        }
        if(nonter[i].length()<=abc.length()&&nonter[i].compare(abc
.substr(0,nonter[i].length()))==0)
            leftrecr[i]=1;
        prod.push_back(temp);
    }
    for(i=0;i<n;++i) {
        cout<<leftrecr[i];
    }
}

```

```

}
for(i=0;i<n;++i) {
    if(leftrecr[i]==0)
        continue;
    int j;
    nonter.push_back(nonter[i]+"");
    vector<string> temp;
    for(j=0;j<prod[i].size();++j) {

if(nonter[i].length()<=prod[i][j].length()&&nonter[i].compare
(prod[i][j].substr(0,nonter[i].length()))==0) {
        string
abc=prod[i][j].substr(nonter[i].length(),prod[i][j].length()-
nonter[i].length()+nonter[i]+"");
        temp.push_back(abc);
        prod[i].erase(prod[i].begin()+j);
        --j;
    }
    else {
        prod[i][j]+=nonter[i]+"";
    }
}
    temp.push_back("^");
    prod.push_back(temp);
}
cout<<"\n\n";
cout<<"\nNew set of non-terminals: ";
for(i=0;i<nonter.size();++i)
    cout<<nonter[i]<<" ";
cout<<"\n\nNew set of productions: ";
for(i=0;i<nonter.size();++i) {
    int j;
    for(j=0;j<prod[i].size();++j) {
        cout<<"\n"<<nonter[i]<<" -> "<<prod[i][j];
    }
}
return 0;
}

```

OUTPUT

```
One by one enter all T productions
RHS of production 1: TxF

RHS of production 2: F

Number of F productions: 1

One by one enter all F productions
RHS of production 1: i
110

New set of non-terminals: E T F E' T'

New set of productions:
E -> TE'
T -> FT'
F -> i
E' -> +TE'
E' -> ^
T' -> xFT'
T' -> ^

Process returned 0 (0x0)   execution time : 106.687 s
Press any key to continue.
```

RESULT: Left Recursion was eliminated from the given grammar successfully using C++..

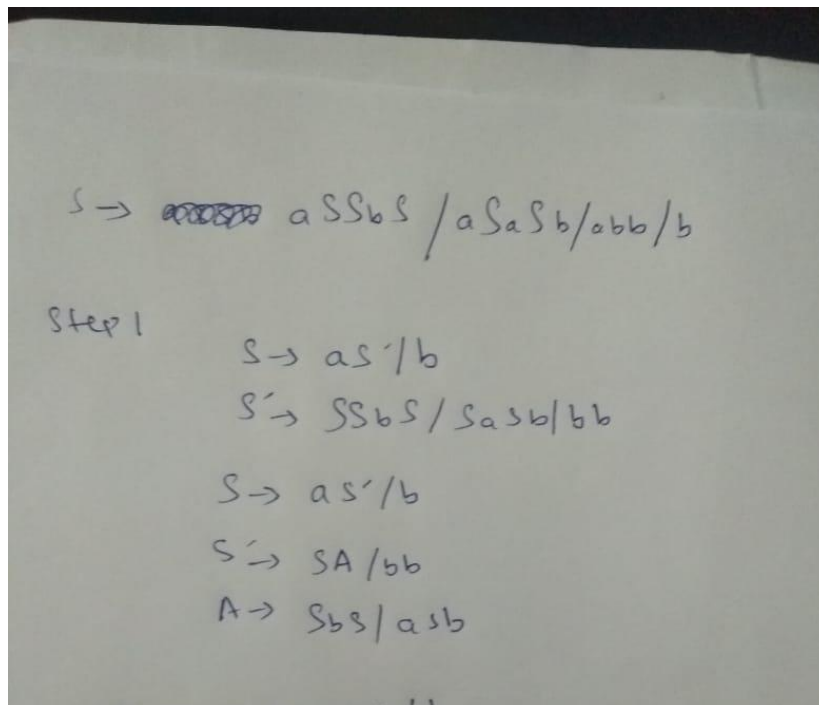
| | | |
|-----------------|-------------------------------|--|
| EX NO: 04-b | ELIMINATION OF LEFT FACTORING | |
| DATE:16-02-2021 | | |

AIM : To Write a C++ Program to eliminate Left Factoring in the given grammar.

ALGORITHM:

1. Start
2. Get productions from the user
3. Check for common left factors in the production
4. Group all like productions
5. Simplify original production
6. Create the new production
7. Display all productions
8. Stop

Manual Calculation:



PROGRAM :

```

#include<stdio.h>

#include<string.h>

int main()
{

```

```

char
gram[20],part1[20],part2[20],modifiedGram[20],newGram[20],
tempGram[20];

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

printf("Enter Production : A->");

fgets(gram,20,stdin);

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

    part1[j]=gram[i];

part1[j]='\0';

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

    part2[i]=gram[j];

part2[i]='\0';

for(i=0;i<strlen(part1)||i<strlen(part2);i++)

{

    if(part1[i]==part2[i])

    {

        modifiedGram[k]=part1[i];

        k++;

        pos=i+1;

    }

}

for(i=pos,j=0;part1[i]!='\0';i++,j++){

    newGram[j]=part1[i];

}

newGram[j++]='|';

```

```

for(i=pos;part2[i]!='\0';i++,j++){
    newGram[j]=part2[i];
}

modifiedGram[k]='X';
modifiedGram[++k]='\0';
newGram[j]='\0';

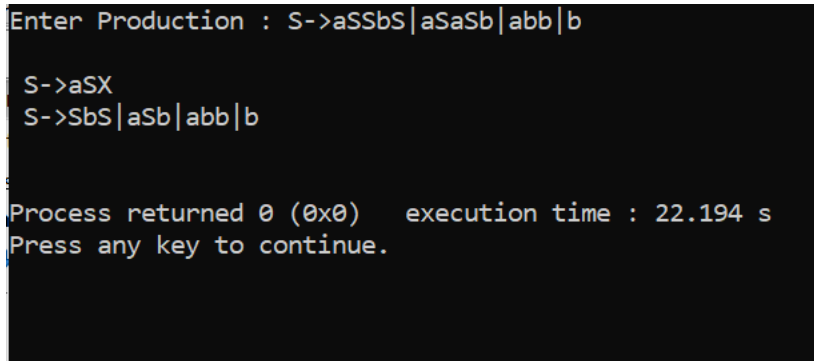
printf("\n S->%s",modifiedGram);

printf("\n x->%s\n",newGram);

}

```

OUTPUT



```

Enter Production : S->aSSbS|aSaSb|abb|b
S->aSX
S->SbS|aSb|abb|b
S
Process returned 0 (0x0)   execution time : 22.194 s
Press any key to continue.

```

RESULT: Left factoring was performed on the given grammar successfully using C++.

| | |
|------------------------|--------------------------------|
| EX NO: 05 | <u>FIRST AND FOLLOW</u> |
| DATE:23-02-2021 | |

AIM : To write a program to perform first and follow .

ALGORITHM:

For computing the first:

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

Example: $F \rightarrow I \mid id$

We can write it as $\text{FIRST}(F) \rightarrow \{ (, id)$

2. If X is a non-terminal like $E \rightarrow T$ then to get $\text{FIRST}T$

substitute T with other productions until you get a terminal as the first symbol

3. If $X \rightarrow \epsilon$ then add ϵ to $\text{FIRST}(X)$.

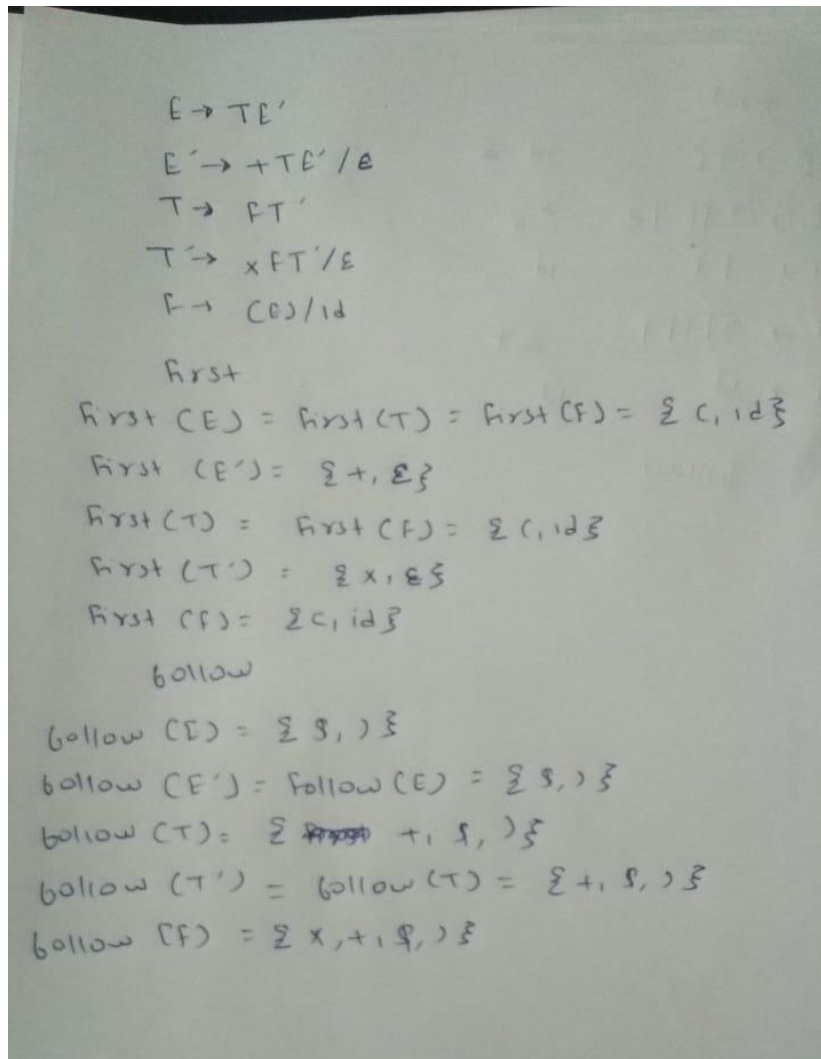
For computing the follow:

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.

Manual Calculation:

**PROGRAM :**

```

#include <bits/stdc++.h>

#define max 20

using namespace std;

char prod[max][10], ter[10], nt[10], first[10][10],
follow[10][10];

int eps[10], c=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)

{

    return (c >= 'A' && c <= 'Z') ? 1 : 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[c][0] = s[0];
```

```

for (i = 3; s[i] != '\0'; i++)
{
    if (!IsCap(s[i]))
        add(ter, s[i]);

    prod[c][i - 2] = s[i];
}

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

add(nt, s[0]);

c++;
}

void findfirst()
{
    int i, j, n, k, e, n1;

    for (i = 0; i < c; i++)
    {
        for (j = 0; j < c; 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 < c; i++)
{
    for (j = 0; j < c; j++)
    {
        for (k = strlen(prod[j]) - 1; k > 0; k--)
        {
            if (IsCap(prod[j][k]))
            {
                n = findpos(prod[j][k]);
                if (prod[j][k + 1] == '\0') // A -> aB
                {
                    n1 = findpos(prod[j][0]);
                    addarr(follow[n], follow[n1]);
                }
                if (IsCap(prod[j][k + 1])) // A -> aBb
                {
                    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]);
    }
}
}
}
}

int main()
{
    char s[max], i;

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

    cin >> s;

    while (strcmp("end", s))
    {
        addprod(s);

        cin >> s;
    }

    findfirst();

    findfollow();

    for (i = 0; i < strlen(nt); i++)
    {
        cout << "\nFIRST[" << nt[i] << "]: " << first[i];

        if (eps[i] == 1)

```

```

        cout << (char)238 << "\t";

    else

        cout << "\t";

    cout << "FOLLOW[" << nt[i] << "]: " << follow[i];

}

return 0;

}

```

OUTPUT

```

Enter the productions (type 'end' at the last of the production)
E->TR
R->+TR/ε
T->FD
D->xFD/ε
F->(E)/i
end

FIRST[E]: (      FOLLOW[E]: $)
FIRST[R]: +      FOLLOW[R]: $/)
FIRST[T]: (      FOLLOW[T]: +
FIRST[D]: x      FOLLOW[D]: +/
FIRST[F]: (      FOLLOW[F]: x

Process returned 0 (0x0)   execution time : 153.681 s
Press any key to continue.

```

RESULT:

The FIRST and FOLLOW sets of the non-terminals of a grammar were found successfully using C++.

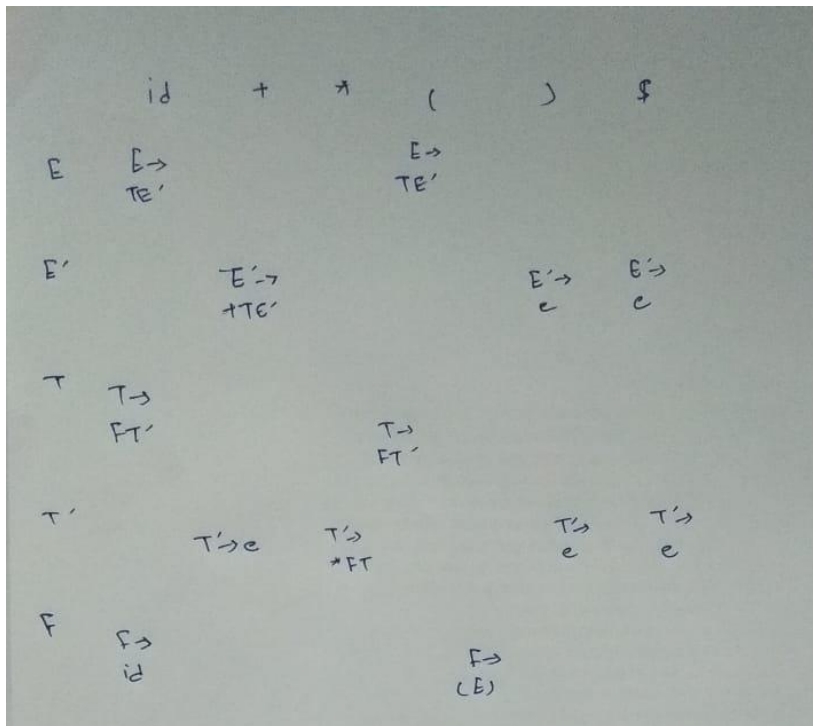
| | |
|------------------------|--------------------------|
| EX NO: 06 | PREDICTIVE PARSER |
| DATE:02-03-2021 | |

AIM : To construct a predictive parser in C++.

ALGORITHM:

1. Start the program.
2. Initialize the required variables.
3. Get the number of coordinates and productions from the user.
4. Perform the following for (each production $A \rightarrow \alpha$ in G)
 - {
 - for (each terminal a in $FIRST(\alpha)$)
 - add $A \rightarrow \alpha$ to $M[A, a]$;
 - if (ϵ is
 - in $FIRST(\alpha)$)
 - for (each
 - symbol b in
 - $FOLLOW(A)$)
 - add A
 - $\rightarrow \alpha$ to
 - $M[A, b]$;
5. Print the resulting stack.
6. Print if the grammar is accepted or not.
7. Exit the program.

Manual Calculation:

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

```



```

        break;
    }
}
tem[ct++] = '_';
}
k++;
}
int zap = 0,tuna;
for(tuna = 0;tuna<ct;tuna++){
    if(tem[tuna] == '#'){
        zap = 1;
    }
    else if(tem[tuna] == '_'){
        if(zap == 1){
            zap = 0;
        }
        else
            break;
    }
    else{
        first_prod[ap][destiny++] = tem[tuna];
    }
}
}
char table[land][sid+1];
ptr = -1;
for(ap = 0; ap < land ; ap++){
    for(kay = 0; kay < (sid + 1) ; kay++){
        table[ap][kay] = '!';
    }
}
for(ap = 0; ap < count ; ap++){
    ck = production[ap][0];
    xxx = 0;
    for(kay = 0; kay <= ptr; kay++)
        if(ck == table[kay][0])
            xxx = 1;
    if (xxx == 1)
        continue;
    else{
        ptr = ptr + 1;
        table[ptr][0] = ck;
    }
}

```

```

}
}
for(ap = 0; ap < count ; ap++){
int tuna = 0;
while(first_prod[ap][tuna] != '\0'){
    int to,ni=0;
    for(to=0;to<sid;to++){
        if(first_prod[ap][tuna] == ter[to]){
            ni = 1;
        }
    }
    if(ni == 1){
        char xz = production[ap][0];
        int cz=0;
        while(table[cz][0] != xz){
            cz = cz + 1;
        }
        int vz=0;
        while(ter[vz] != first_prod[ap][tuna]){
            vz = vz + 1;
        }
        table[cz][vz+1] = (char)(ap + 65);
    }
    tuna++;
}
}
for(k=0;k<sid;k++){
for(kay=0;kay<100;kay++){
    if(calc_first[k][kay] == '!'){
        break;
    }
    else if(calc_first[k][kay] == '#'){
        int fz = 1;
        while(calc_follow[k][fz] != '!'){
            char xz = production[k][0];
            int cz=0;
            while(table[cz][0] != xz){
                cz = cz + 1;
            }
            int vz=0;
            while(ter[vz] != calc_follow[k][fz]){
                vz = vz + 1;
            }
        }
    }
}
}

```

```

        }
        table[k][vz+1] = '#';
        fz++;
    }
    break;
}
}
for(ap = 0; ap < land ; ap++){
printf("\t\t\t %c\t\t",table[ap][0]);
for(kay = 1; kay < (sid + 1) ; kay++){
    if(table[ap][kay] == '!')
        printf("\t\t");
    else if(table[ap][kay] == '#')
        printf("%c=#\t\t",table[ap][0]);
    else{
        int mum = (int)(table[ap][kay]);
        mum -= 65;
        printf("%s\t\t",production[mum]);
    }
}
printf("\n");
printf("\t\t\t-----
-----");
printf("\n");
}
int j;
printf("\n\nPlease enter the desired INPUT STRING =
");
char input[100];
scanf("%s%c",input,&ch);
printf("\n\t\t\t\t\t=====
=====
\n");
printf("\t\t\t\t\tStack\t\t\tInput\t\t\tAction");
printf("\n\t\t\t\t\t=====
=====
\n");
int i_ptr = 0,s_ptr = 1;
char stack[100];
stack[0] = '$';
stack[1] = table[0][0];

```



```

while(s_ptr != -1){
printf("\t\t\t\t\t");
int vamp = 0;
for(vamp=0;vamp<=s_ptr;vamp++){
    printf("%c",stack[vamp]);
}
printf("\t\t\t\t\t");
vamp = i_ptr;
while(input[vamp] != '\0'){
    printf("%c",input[vamp]);
    vamp++;
}
printf("\t\t\t\t\t");
char her = input[i_ptr];
char him = stack[s_ptr];
s_ptr--;
if(!isupper(him)){
    if(her == him){
        i_ptr++;
        printf("POP ACTION\n");
    }
    else{
        printf("\nString Not Accepted by LL(1)
Parser !!\n");
        exit(0);
    }
}
else{
    for(i=0;i<sid;i++){
        if(ter[i] == her)
            break;
    }
    char produ[100];
    for(j=0;j<land;j++){
        if(him == table[j][0]){
            if (table[j][i+1] == '#'){
                printf("%c=#\n",table[j][0]);
                produ[0] = '#';
                produ[1] = '\0';
            }
            else if(table[j][i+1] != '!'){
                int mum = (int)(table[j][i+1]);

```

```

        mum -= 65;

strcpy(produ,production[mum]);
        printf("%s\n",produ);
    }
    else{
        printf("\nString Not
Accepted by LL(1) Parser !!\n");
        exit(0);
        // break;
    }
}

}
int le = strlen(produ);
le = le - 1;
if(le == 0){
    continue;
}
for(j=le;j>=2;j--){
    s_ptr++;
    stack[s_ptr] = produ[j];
}

}
}

printf("\n\t\t\t=====
=====
=====\\n
");
if (input[i_ptr] == '\0'){
printf("\t\t\t\t\tYOUR STRING HAS BEEN
ACCEPTED !!\n");
}
else
printf("\n\t\t\t\t\tYOUR STRING HAS BEEN
REJECTED !!\n");
printf("\t\t\t=====
=====
=====\\n");
}

void follow(char c)
{

```

```

int i ,j;
if(production[0][0]==c){
    f[m++]='$';
}
for(i=0;i<10;i++)
{
    for(j=2;j<10;j++)
    {
        if(production[i][j]==c)
        {
            if(production[i][j+1]!='\0'){
followfirst(production[i][j+1],i,(j+2));
            }

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

void findfirst(char c ,int q1 , int q2)
{
    int j;
    if(!(isupper(c))){
        first[n++]=c;
    }
    for(j=0;j<count;j++)
    {
        if(production[j][0]==c)
        {
            if(production[j][2]=='#{'){
                if(production[q1][q2] == '\0')
                    first[n++]='#';
                else if(production[q1][q2] != '\0' && (q1 !=
0 || q2 != 0))
                {
                    findfirst(production[q1][q2], q1,
(q2+1));
                }
            }
        }
    }
}

```

```

        else
            first[n++]='#';
    }
    else if(!isupper(production[j][2])){
        first[n++] = production[j][2];
    }
    else {
        findfirst(production[j][2], j, 3);
    }
}
}
}

void followfirst(char c, int c1 , int c2)
{
    int k;
    if(!(isupper(c)))
    f[m++] = c;
    else{
        int i=0,j=1;
        for(i=0;i<count;i++)
        {
            if(calc_first[i][0] == c)
                break;
        }
        while(calc_first[i][j] != '!')
        {
            if(calc_first[i][j] != '#'){
                f[m++] = calc_first[i][j];
            }
            else{
                if(production[c1][c2] == '\0'){
                    follow(production[c1][0]);
                }
                else{
                    followfirst(production[c1][c2],c1,c2+1);
                }
            }
            j++;
        }
    }
}
}

```

RESULT: The C++ program to implement the predictive parser was compiled, executed and verified successfully.

| | |
|------------------------|----------------------------|
| EX NO: 07 | SHIFT REDUCE PARSER |
| DATE:09-03-2021 | |

AIM : To implement Shift Reduce Parser in C.

ALGORITHM:

1. Start the program.
2. Initialize the required variables.
3. Enter the input symbol.
4. Perform the following:

for top-of-stack symbol, s , and next input symbol, a

Shift x : (x is a

STATE number)

Push a , then x on the top of the stack

Advance ip to point to the next input symbol.

Reduce y : (y is a

PRODUCTION number)

Assume that the production is of the

*form $A \rightarrow \beta$ Pop $2 * |\beta|$*

symbols of the stack.

At this point the top of the stack should be a state number, say s' .

Push A , then goto of $T[s', A]$ (a state number) on the top of the stack.

Output the production $A \rightarrow \beta$.

5. Print if string is accepted or not.
6. Stop the program.

Manual Calculation:

| Stack | input | action |
|--------|-------|----------------------------|
| \$id | +E\$ | Shift \rightarrow id |
| \$E | +id\$ | Reduce $T \rightarrow E$ |
| \$E+ | id\$ | Shift \rightarrow symbol |
| \$E+id | \$ | Shift \rightarrow id |
| \$E+E | \$ | Reduce $T \rightarrow E$ |
| \$E | \$ | Reduce $T \rightarrow E$ |

955 - Harsh

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$s\t%s\t%s\t%sid",stk,a,act);  
        check();  
    }  
    else  
    {  
        stk[i]=a[j];  
        stk[i+1]='\0';  
        a[j]=' '  
        printf("\n$s\t%s\t%s\t%symbols",stk,a,act);  
        check();  
    }  
}  
}
```



```

void check()
{
    strcpy(ac,"REDUCE TO E");
    for(z=0; z<c; z++)
        if(stk[z]=='i' && stk[z+1]=='d')
        {
            stk[z]='E';
            stk[z+1]='\0';
            printf("\n%s\t%s\t%s",stk,a,ac);
            j++;
        }
    for(z=0; z<c; z++)
        if(stk[z]=='E' && stk[z+1]=='+' && stk[z+2]=='E')
        {
            stk[z]='E';
            stk[z+1]='\0';
            stk[z+2]='\0';
            printf("\n%s\t%s\t%s",stk,a,ac);
            i=i-2;
        }
    for(z=0; z<c; z++)
        if(stk[z]=='E' && stk[z+1]=='*' && stk[z+2]=='E')
        {
            stk[z]='E';

```

```

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

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

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

        i=i-2;

    }

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

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

        {

            stk[z]='E';

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

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

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

            i=i-2;

        }

    }

```

OUTPUT

```

GRAMMAR is E->E+E
E->E*E
E->(E)
E->id
enter input string
id+id
stack   input   action
$id      +id$  SHIFT->id
$E       +id$  REDUCE TO E
$E+      id$   SHIFT->symbols
$E+id     $    SHIFT->id
$E+E      $    REDUCE TO E
$E        $    REDUCE TO E
Process returned 0 (0x0)   execution time : 14.318 s
Press any key to continue.

```

RESULT: The C implementation of Shift Reduce Parser was compiled, executed and verified successfully.

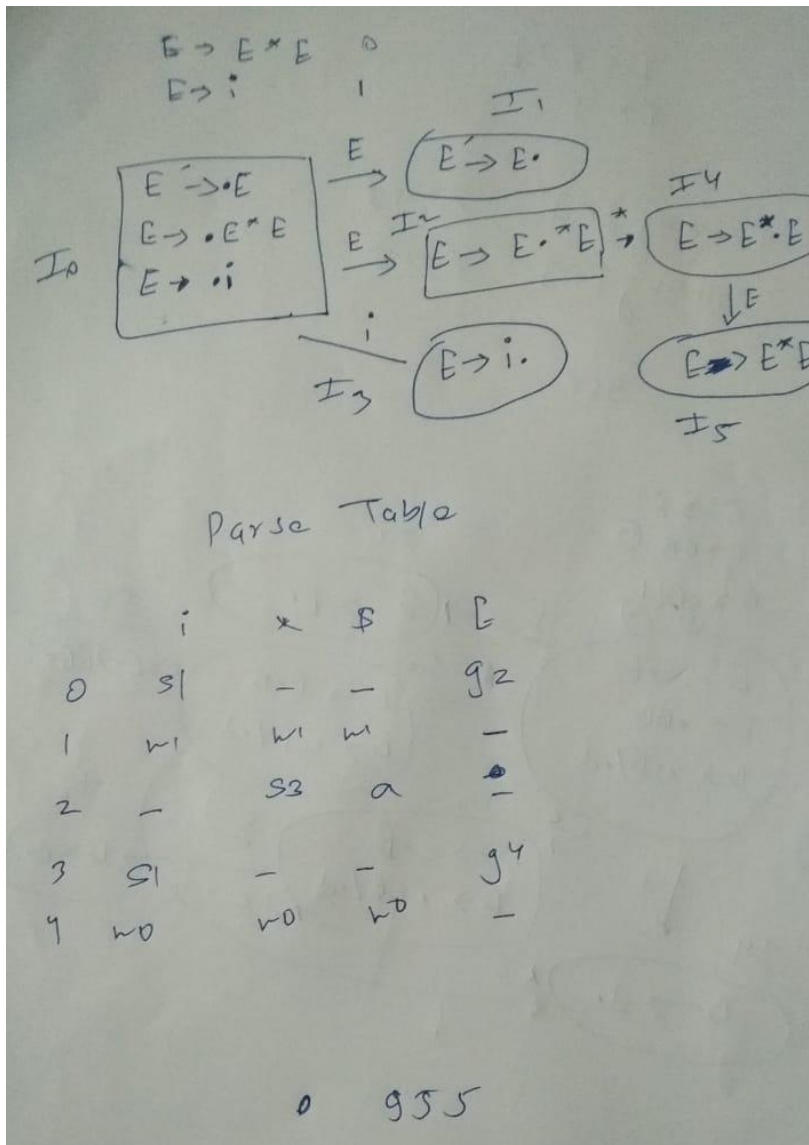
| | |
|------------------------|---------------------|
| EX NO: 08 | LR(0) PARSER |
| DATE:16-03-2021 | |

AIM : To implement LR(0) Parser in Python.

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 $S' \rightarrow S \$$ 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.

Manual Calculation:

**PROGRAM :**

```
def closure(I, nonT):
```

```
    J = I
```

```
    for item in J :
```

```
        #print(item)
```

```
        index = item[1].index('.')
```

```
        if(index < (len(item[1])-1) and item[1][index+1] in nonT):
```

```

    #print('item : ',item[1][index+1])

    for production in nonT[item[1][index+1]]:

        if( [item[1][index+1],str('.')+str(production)] not in J):

J.append([item[1][index+1],str('.')+str(production)])

        #print([item[1][index+1],str('.')+str(production)])

    return J

# ----- Ends -----

```

2. ----- Set of Canonical Items -----

```

state = []

I = []

def setOfItems(start,nonTer,ter):

    I.append(closure(['start','.'+start+'$'],nonTer))

    #print(I)

    ter += list(nonTer.keys())

    #print("list of inputs : " , ter)

    for conI in I:

        for grammar in ter:

```

```

if(grammar is '$'):
    continue

#print("grammar : ",grammar)

goto = False

goto1 = False

shift = False

shift1 = False

reduce = False

close = []

for item in conI:

    #print("item : ",item)

    if(item[1].index('.') < (len(item[1])-1) and
item[1][item[1].index('.')+1] is grammar):

close.append([item[0],item[1][:item[1].index('.')] + grammar + '.' +
item[1][item[1].index('.')+2:]])

    #else:

    # print(item)

#print("close : ",close)

l = closure(close,nonTer)

if(len(l) == 0):

    continue

#print("closure : ", l)

if(grammar in nonTer.keys()):

    goto1 = True

```

```

else:

    shift1 = True

if(l not in I):

    if(goto1):

state.append(['g',I.index(conI)+1,len(I)+1,grammar])

        goto = True

    elif(shift1):

        shift = True

state.append(['s',I.index(conI)+1,len(I)+1,grammar])

        I.append(l)

else:

    if(goto1):

        goto = True

state.append(['g',I.index(conI)+1,I.index(l)+1,grammar])

    elif(shift1):

        shift = True

state.append(['s',I.index(conI)+1,I.index(l)+1,grammar])

# -----

```


3. -----Create a Parse Table -----

```

reduce = []
accept = -1
def toReduce(rule,accept,start):
    s = ['start',start+'.']
    for parState in I:
        #print(s,parState)
        if(s in parState):
            #print("here;")
            accept = I.index(parState)
            for item in parState:
                if( item in rule):
                    reduce[I.index(parState)].append(rule.index(item))

    return accept

```

4. ----- To Parse -----

```
symbolMap = dict()
```

```
parseTable = []
```

```
def createParseTable(ter):
```

```
    for i in state:
```

```
        parseTable[i[1]-1][symbolMap[i[3]]] = i[0]+str(i[2]-1)
```

```
    parseTable[accept][symbolMap['$']] = 'a'
```

```
    for i in reduce:
```

```
        if(len(i)>0):
```

```
            for j in ter:
```

```
                parseTable[reduce.index(i)][symbolMap[j]] =  
'r'+str(i[0])
```

(i) Stack -----

```
class Stack:
```

```
    def __init__(self):
```

```
        self.__storage = []
```

```
    def isEmpty(self):
```

```
return len(self.__storage) == 0
```

```
def push(self,p):
```

```
    self.__storage.append(p)
```

```
def pop(self):
```

```
    return self.__storage.pop()
```

```
def top(self):
```

```
    return self.__storage[len(self.__storage) - 1]
```

```
def length(self):
```

```
    return len(self.__storage)
```

```
def __str__(self):
```

```
    """
```

Because of using list as parent class for stack, our last element will

be first for stack, according to FIFO principle. So, if we will use

parent's implementation of str(), we will get reversed order of

elements.

```
    """
```

#: You can reverse elements and use supper `__str__` method, or

#: implement it's behavior by yourself.

#: I choose to add 'stack' in the begging in order to differ list and

```

#: stack instances.

return 'stack [{ }]'.format(', '.join([ str(i) for i in
reversed(self.__storage) ]))

#-----Stack Defn ENDS -----
-----

def parseString(rule,string):

    index = 0

    flag = False

    st = Stack()

    st.push('0')

    while(index < len(string)):

        print(st , string , index , sep = '\t\t ')

        c = parseTable[int(st.top())][symbolMap[string[index]]][0]

        if(c is 'a'):

            flag = True

            break

        pt =
parseTable[int(st.top())][symbolMap[string[index]]][1:]

        pt = int(pt)

        if( c is 'r'):

            l = len(rule[pt][1])

            l *= 2

            l -= 2 #'.' is also considered

            if(l >= st.length()):

```

```

        break
    else:
        for i in range(l):
            st.pop()
            top = int(st.top())
            st.push(rule[pt][0])
            st.push(parseTable[top][symbolMap[st.top()]] [1:])
        else:
            st.push(string[index])
            st.push(str(pt))
            index+=1
    return flag

# -----

# ----- Driver Program -----

terminals = []
nonTerminals = dict()
terminals = input("Enter Terminals (|) : ").split("|")
n = int(input("No. of Non - Terminals : "))

```

```

for i in range(n):

    ch = input("NonTerminals : ").strip()

    rules = input("Productions (|) : ").split("|")

    nonTerminals[ch] = rules

```

```

# --- Old Rules-----

```

```

S = input("Start Symbol : ")

```

```

terminals+=['$']

```

```

print("Productions : ")

```

```

for i in nonTerminals.keys():

```

```

    print(i,"-->",end=' ')

```

```

    for j in nonTerminals[i]:

```

```

        print(j,end=' | ')

```

```

    print()

```

```

setOfItems(S,nonTerminals,terminals)

```

```

print("canonicals Items : ")

```

```

for count , i in enumerate(I):

```

```

    print(count+1 , i)

```

```

print("state Transitions : ")

```

```

for count , i in enumerate(state):

```

```

    print(count+1, i)

rule = []

accept = -1

for i in nonTerminals.keys():
    for j in nonTerminals[i]:
        rule.append([i,j+str('.')])

print('rule :')
for i in rule:
    print(i)

# ----- To find the reduction rules - - - - -
reduce = [ [] for i in range(len(I)) ]
accept = toReduce(rule,accept,S)

print("reduce")
for count,i in enumerate(reduce):
    print(count+1,i)

print("accept : ",accept+1)

# --- - - - parse Table - - - - -

```

```

symbols = []

symbols += terminals

for count , i in enumerate(symbols):

    symbolMap[i] = count

print(symbols)


parseTable = [ ['-'] for i in range(len(symbols))] for j in
range(len(I)) ]


for i in nonTerminals.keys():

    terminals.remove(i)


createParseTable(terminals)


# ---Parse Table-----

print('Parse Table')

print("\t\t",end="")

for i in symbols:

    print(i,end= '\t')

print()

for count,j in enumerate(parseTable):

    print(count,end="\t\t")

    for i in j:

```



```
print(i,end='\t')
```

```
print()
```

OUTPUT

```
Enter Terminals (|) : i|*
No. of Non - Terminals : 1
NonTerminals : E
Productions (|) : E*E|i
Start Symbol : E
Productions :
E --> E*E | i |
canonicals Items :
1 [['start', '.E$'], ['E', '.E*E'], ['E', '.i']]
2 [['E', 'i.']]
3 [['start', 'E.$'], ['E', 'E.*E']]
4 [['E', 'E*.E'], ['E', '.E*E'], ['E', '.i']]
5 [['E', 'E*E.'], ['E', 'E.*E']]
state Transitions :
1 ['s', 1, 2, 'i']
2 ['g', 1, 3, 'E']
3 ['s', 3, 4, '*']
4 ['s', 4, 2, 'i']
5 ['g', 4, 5, 'E']
6 ['s', 5, 4, '*']
rule :
['E', 'E*E.']
['E', 'i.']
reduce
1 []
2 [1]
3 []
4 []
5 [0]
accept : 3
['i', '*', '$', 'E']
Parse Table
```

| | i | * | \$ | E |
|---|----|----|----|----|
| 0 | s1 | - | - | g2 |
| 1 | r1 | r1 | r1 | - |
| 2 | - | s3 | a | - |
| 3 | s1 | - | - | g4 |
| 4 | r0 | r0 | r0 | - |

RESULT: The Python implementation of LR(0) Parser was compiled, executed and verified successfully.

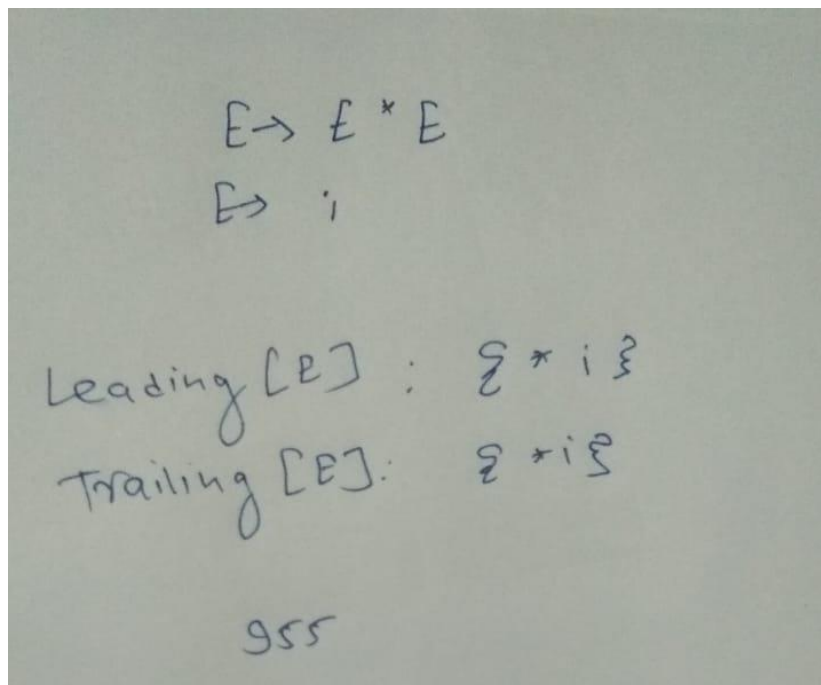
| | |
|-----------------|----------------------|
| EX NO: 09 | LEADING AND TRAILING |
| DATE:23-02-2021 | |

AIM : To implement a C++ program to find the LEADING and TRAILING sets of variables of a given CFG.

ALGORITHM:

1. For Leading, check for the first non-terminal.
2. If found, print it.
3. Look for next production for the same non-terminal.
4. If not found, recursively call the procedure for the single non-terminal present before the comma or End Of Production String.
5. Include it's results in the result of this non-terminal.
6. For trailing, we compute same as leading but we start from the end of the production to the beginning.
7. Stop

Manual Calculation:



PROGRAM :

```
#include<iostream>

#include<string.h>

#include<conio.h>
```

```

using namespace std;

int nt,t,top=0;

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

int searchnt(char a)
{
    int count=-1,i;
    for(i=0;i<nt;i++)
    {
        if(NT[i]==a)
            return i;
    }
    return count;
}

int searchter(char a)
{
    int count=-1,i;
    for(i=0;i<t;i++)
    {
        if(T[i]==a)
            return i;
    }
    return count;
}

void push(char a)

```

```
{  
s[top]=a;  
top++;  
}  
  
char pop()  
{  
top--;  
return s[top];  
}  
  
void installl(int a,int b)  
  
{  
if(l[a][b]=='f')  
{  
l[a][b]='t';  
push(T[b]);  
push(NT[a]);  
}  
}  
  
void installt(int a,int b)  
  
{  
if(tr[a][b]=='f')  
{  
tr[a][b]='t';
```

```
push(T[b]);  
push(NT[a]);  
  
}  
  
}  
  
int main()  
{  
    int i,s,k,j,n;  
    char pr[30][30],b,c;  
    cout<<"Enter the no of productions:";  
    cin>>n;  
    cout<<"Enter the productions one by one\n";  
    for(i=0;i<n;i++)  
        cin>>pr[i];  
    nt=0;  
    t=0;  
    for(i=0;i<n;i++)  
    {  
        if((searchnt(pr[i][0]))==-1)  
            NT[nt++]=pr[i][0];  
    }  
    for(i=0;i<n;i++)  
    {  
        for(j=3;j<strlen(pr[i]);j++)
```

```

{
if(searchnt(pr[i][j])==-1)
{
if(searchter(pr[i][j])==-1)
T[t++]=pr[i][j];
}
}
}
for(i=0;i<nt;i++)
{
for(j=0;j<t;j++)
l[i][j]='f';
}
for(i=0;i<nt;i++)
{
for(j=0;j<t;j++)

tr[i][j]='f';
}
for(i=0;i<nt;i++)
{
for(j=0;j<n;j++)
{
if(NT[(searchnt(pr[j][0]))]==NT[i])

```

```

{
if(searchter(pr[j][3])!=-1)
installl(searchnt(pr[j][0]),searchter(pr[j][3]));
else
{
for(k=3;k<strlen(pr[j]);k++)
{
if(searchnt(pr[j][k])==-1)
{
installl(searchnt(pr[j][0]),searchter(pr[j][k]));
break;
}
}
}
}
}
}
while(top!=0)
{
b=pop();
c=pop();
for(s=0;s<n;s++)
{
if(pr[s][3]==b)

```

```

installl(searchnt(pr[s][0]),searchter(c));

}

}

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

{

cout<<"Leading["<<NT[i]<<"]"<<": "<<"\t{ "<<" ";

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

{

if(l[i][j]=='t')

cout<<T[j]<<" ";

}

cout<<" }\n";

}


top=0;

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

{

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

{

if(NT[searchnt(pr[j][0])]==NT[i])

{

if(searchter(pr[j][strlen(pr[j])-1])!=-1)

installt(searchnt(pr[j][0]),searchter(pr[j][strlen(pr[j])-1]));

```



```
else
{
for(k=(strlen(pr[j])-1);k>=3;k--)
{
if(searchnt(pr[j][k])== -1)
{
installt(searchnt(pr[j][0]),searchter(pr[j][k]));
break;
}
}
}
}
}
}
}
while(top!=0)
{
b=pop();
c=pop();
for(s=0;s<n;s++)
{
if(pr[s][3]==b)
installt(searchnt(pr[s][0]),searchter(c));
}
}
```

```

for(i=0;i<nt;i++)
{
cout<<"Trailing["<<NT[i]<<"]"<<": "<<"\t{"<<" ";
for(j=0;j<t;j++)
{
if(tr[i][j]=='t')
cout<<T[j]<<" ";
}
cout<<"}\n";
}
getch();
}

```

OUTPUT

```

Enter the no of productions:2
Enter the productions one by one
E->E*E
E->i
Leading[E]:    { * i }
Trailing[E]:  { * i }

```

RESULT: The C++ implementation to find the LEAD and TRAIL sets of variables of a CFG was compiled, executed and verified successfully.

| | |
|-------------------------|------------------------------------|
| EX NO:08-04-2021 | INTERMEDIATE CODE GENERATER |
| | |

```

{
if(pr[s][3]==b)

installt(searchnt(pr[s][0]),searchter(c));

}

}

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

{

cout<<"Trailing["<<NT[i]<<"]"<<":"<<"\t{"<<" ";

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

{

if(tr[i][j]=='t')

cout<<T[j]<<" ";

}

cout<<"}\n";

}

getch();

```

OUTPUT

```

Enter the no of productions:2
Enter the productions one by one
E->E÷E
E->i
Leading[E]:  {+i }
Trailing[E]: {+i }

```

RESULT:The C++ implementation to find the LEAD and TRAIL sets of variables of a CFG was compiled, executed and verified successfully.

AIM: To implement a program to convert infix expression to postfix, prefix.

ALGORITHM:

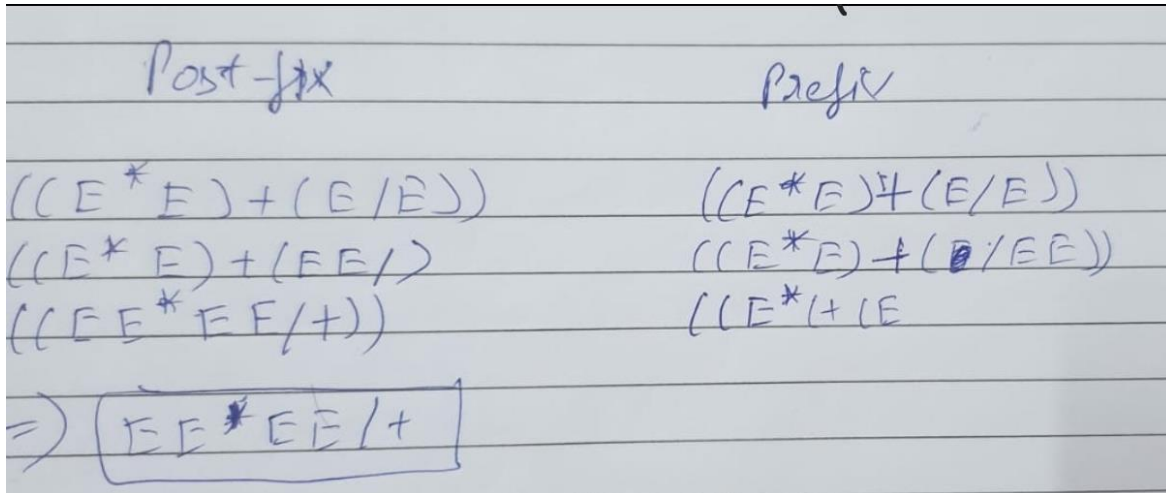
Postfix:

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
 - 1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.
 - 2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

Postfix:

1. Scan the infix expression from right to left.
2. If the scanned character is an operand, output it.
3. Else,
 - 1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.
 - 2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an ')', push it to the stack.
5. If the scanned character is an '(', pop the stack and output it until a ')' is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Reverse the infix expression ,Print the output
8. Pop and output from the stack until it is not empty.

Manual Calculation:



PROGRAM :

```
class infix_to_postfix:
    precedence={'^':5, '*':4, '/':4, '+':3, '-':3, '(':2, ')':1}
    def __init__(self):
        self.items=[]
        self.size=-1
    def push(self,value):
        self.items.append(value)
        self.size+=1
    def pop(self):
        if self.isempty():
            return 0
        else:
            self.size-=1
            return self.items.pop()
    def isempty(self):
        if(self.size== -1):
            return True
        else:
            return False
    def seek(self):
        if self.isempty():
            return false
        else:
            return self.items[self.size]
```

```

def isOperand(self,i):
    if i.isalpha() or i in '1234567890':
        return True
    else:
        return False
def infixtopostfix (self,expr):
    postfix=""

    for i in expr:
        if(len(expr)%2==0):
            print("Incorrect infix expr")
            return False
    elif(self.isOperand(i)):
        postfix +=i
    elif(i in '+-*/^'):
        while(len(self.items)and self.precedence[i]<=self.precedence[self.seek()]):
            postfix+=self.pop()
        self.push(i)
    elif i is '(':
        self.push(i)
    elif i is ')':
        o=self.pop()
        while o!='(':
            postfix +=o
            o=self.pop()
        print(postfix)
        #end of for
    while len(self.items):
        if(self.seek()=='('):
            self.pop()
        else:
            postfix+=self.pop()

    return postfix
s=infix_to_postfix()
expr=input('enter the expression ')
result=s.infixtopostfix(expr)
if (result!=False):
    print("the postfix expr of :",expr,"is",result)

class infix_to_prefix:

```

```

precedence={'^':5,'*':4,'/':4,'+':3,'-':3,'(':2,')':1}
def __init__(self):
self.items=[]
self.size=-1
    def push(self,value):
self.items.append(value)
self.size+=1
    def pop(self):
    if self.isempty():
        return 0
    else:
self.size-=1
        return self.items.pop()
def isempty(self):
    if(self.size== -1):
        return True
    else:
        return False
def seek(self):
    if self.isempty():
        return False
    else:
        return self.items[self.size]
def isOperand(self,i):
    if i.isalpha() or i in '1234567890':
        return True
    else:
        return False
def reverse(self,expr):
    rev=""
    for i in expr:
        if i is '(':
i=')'
        elif i is ')':
i='('
            rev=i+rev
    return rev
def infixtoprefix (self,expr):
    prefix=""
print('\nprefix expression after every iteration is:')
    for i in expr:

```

| | |
|------------------|---|
| EX NO: 11 | INTERMEDIATE CODE GENERATER Quadruple, Triple, Indirect triple |
|------------------|---|

```

        if(len(expr)%2==0):
            print("Incorrect infix expr")
            return False
        elif(self.isOperand(i)):
            prefix +=i
        elif(i in '+-*/^'):
            while(len(self.items)and self.precedence[i] <self.precedence[self.seek()]):
                prefix+=self.pop()
            self.push(i)
        elif i is '(':
            self.push(i)
        elif i is ')':
            o=self.pop()
            while o!='(':
                prefix +=o
            o=self.pop()
            print(prefix)
            #end of for
        while len(self.items):
            if(self.seek()=='('):
                self.pop()
            else:
                prefix+=self.pop()
                print(prefix)
        return prefix
V=infix_to_prefix()
rev=""
rev=V.reverse(expr)
#print(rev)
result=V.infixtoprefix(rev)
if (result!=False):
    prefix=V.reverse(result)
print("the prefix expr of :",expr,"is",prefix)

```

OUTPUT

PREFIX NOTATION: +*EE/EE
 POSTFIX NOTATION: EE*EE/+

RESULT:The python implementation to convert infix to postfix, prefix has done successfully.

DATE: 16-04-2021

AIM: To implement a program to convert infix expression to quadruple, triple, indirect triple.

ALGORITHM:

1. Start the program
2. Convert the expression to three address code
3. Using three address code construct quadruple
4. Using quadruple construct triple
5. Construct indirect triple
6. Print output
7. End program

Manual Calculation:

$\Rightarrow ((E * E) + (E / E))$

$\Rightarrow t_1 = E * E \quad t_2 = E / E \quad t_3 = t_1 + t_2$

Quadruple

| # | OP | Arg1 | Arg2 | Result |
|-----|----|-------|-------|--------|
| (0) | * | E | E | t_1 |
| (1) | / | E | E | t_2 |
| (2) | + | t_1 | t_2 | t_3 |

Triple

| # | OP | Arg1 | Arg2 |
|-----|----|------|------|
| (0) | * | E | E |
| (1) | / | E | E |
| (2) | + | (0) | (1) |

| # | statement |
|-----|-----------|
| (0) | 11 |
| (1) | 12 |
| (2) | 13 |

Indexed graph

| # | OP | Arg1 | Arg2 |
|-----|----|------|------|
| (0) | * | E | E |
| (1) | / | E | E |
| (2) | + | (0) | (1) |

PROGRAM :

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]
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(f'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 == '=':
    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 == '-':
            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 == '=':
            op2 = stack.pop()
```

| | |
|------------------|------------------------------|
| EX NO: 12 | IMPLEMENTATION OF DAG |
|------------------|------------------------------|

```
        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

```
INPUT THE EXPRESSION: (E*E)+(E/E)
PREFIX: +*EE/EE
POSTFIX: EE*EE/+
### THREE ADDRESS CODE GENERATION ###
t1 := E * E
t2 := E / E
t3 := t1+t2
The quadruple for the expression
  OP | ARG 1 |ARG 2 |RESULT
*|  E   |  E   | t(1)
/|  E   |E   | t(2)
+|  t(1)| t(2)| t(3)
The triple for given expression
  OP | ARG 1 |ARG 2
*|  E   |  E
/|  E   |  E
+   | (0)  | (1)
```

RESULT:The python implementation to convert infix expression to quadruple, triple, indirect triple has done successfully.

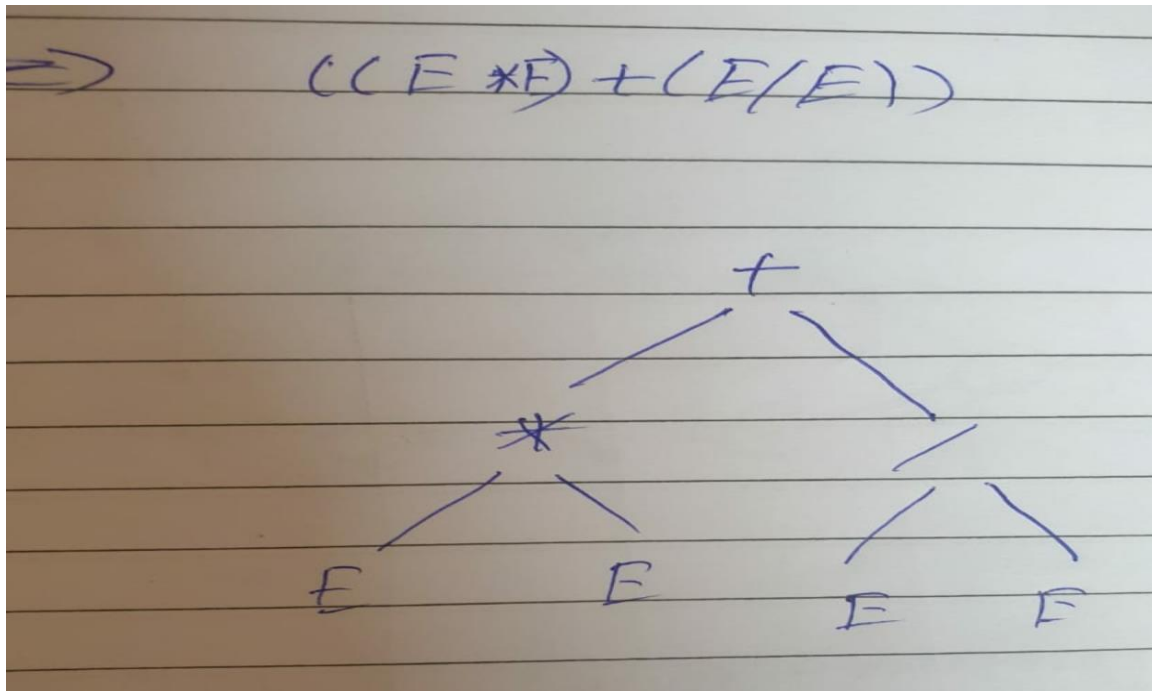
DATE: 23-04-2021

AIM: To implement a program to convert infix expression to DAG.

ALGORITHM:

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

Manual Calculation:



PROGRAM :

```
#include<iostream>
#include<string>
#include<unordered_map>
using namespace std;
class DAG
{ public:
    char label;
```

```

char data;
DAG* left;
DAG* right;

DAG(char x){
    label='_';
    data=x;
    left=NULL;
    right=NULL;
}
DAG(char lb, char x, DAG* lt, DAG* rt){
    label=lb;
    data=x;
    left=lt;
    right=rt;
}
};

int main(){
    int n;
    n=3;
    string st[n];
    st[0]="x=C/D";
    st[1]="y=B+x";
    st[2]="z=A*y";
    unordered_map<char, DAG*>labelDAGNode;

    for(int i=0;i<3;i++){
        string stTemp=st[i];
        for(int j=0;j<5;j++){
            char tempLabel = stTemp[0];
            char tempLeft = stTemp[2];
            char tempData = stTemp[3];
            char tempRight = stTemp[4];
            DAG* leftPtr;
            DAG* rightPtr;
            if(labelDAGNode.count(tempLeft) == 0){
                leftPtr = new DAG(tempLeft);
            }
            else{
                leftPtr = labelDAGNode[tempLeft];
            }
            if(labelDAGNode.count(tempRight) == 0){

```

```

rightPtr = new DAG(tempRight);
    }
else{
rightPtr = labelDAGNode[tempRight];
    }
    DAG* nn = new DAG(tempLabel,tempData,leftPtr,rightPtr);
labelDAGNode.insert(make_pair(tempLabel,nn));
    }
}
cout<<"Label   ptrleftPtrrightPtr"<<endl;
for(int i=0;i<n;i++){
    DAG* x=labelDAGNode[st[i][0]];
cout<<st[i][0]<<"      "<<x->data<<"      ";
    if(x->left->label=='_')cout<<x->left->data;
    else cout<<x->left->label;
cout<<"      ";
    if(x->right->label=='_')cout<<x->right->data;
    else cout<<x->right->label;
cout<<endl;
}
return 0;
}

```

OUTPUT

```

Label      ptrleftPtrrightPtr
x          *   EE
y          /   EE
z          +   x           y

```

RESULT:The C++ implementation to convert infix expression to dag representation has done successfully.

