STUDENT PORTFOLIO

Name: D.Bhargav
Register Number: RA2011028010069
Mail ID: db8482@srmist.edu.in
Department: NWC
Specialization: CSE Cloud Computing
Semester: **6**

Subject Title: 18CSC304J Compiler Design
Handled By: Dr. C Fancy

(Write about the assignment questions and how you solved differently)

When working on an assignment related to lex and flex implementation it is important to understand the task learn the basics of the tools define regular expressions for the tokens generate the lexical analyzer code test the implementation and document the code clearly.

Working on an assignment related to types of parsers, it is important to thoroughly understand the grammar of the language being parsed, and to select the appropriate parser type based on the grammar's characteristics and requirements. Each type of parser has its own strengths and weaknesses, and selecting the right one can greatly impact the efficiency and accuracy of the parsing process.

A crossword quiz assignment can involve designing or analyzing a crossword puzzle, or solving an existing one. When analyzing or solving a puzzle, key considerations include reading clues carefully, filling in easier answers first, keeping track of possible answers, and using pattern recognition to narrow down possibilities.

Designing a mini project on command-line calculator was quite challenging and rewarding. It is a software application that allows users to perform mathematical calculations directly from a command-line interface. It is a simple tool that can be used to perform basic arithmetic operations such as addition, subtraction, multiplication, and division.

(**W**hat is the most interesting part in the assignment**)**

The level of the questions was very satisfying and every question was solvable by the knowledge provided in the lectures and lab classes. Each question was based on the basic understanding which I found the most interesting.

SRM INSTITUTE OF SCIENCE & TECHNOLOGY

S.R.M. NAGAR, KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that this portfolio report is the Bonafide work of **D.Bhargav (RA2011028010069)** of III Year/VI Sem B.Tech (CSE) who carried out the Portfolio report(Hackerank, Lab report, Miniproject report) under my supervision for the course **18CSC304J - COMPILER DESIGN** in SRM Institute of Science and Technology during the academic year 2022-2023 (Even sem).

Signature

Dr.Fancy

ASSISTANT PROFESSOR

Department of Networking and communications

Signature

Dr.Annapurani K

Head of Department

Department of Networking and Communications

D. Bhargav (K2) — Compiler Design Assignment — RA2011028010069

1. Illustrate annotated Parse tree with synchronized and inherited attribute for expression 3*5- for the given grammar.

$E \rightarrow TR$

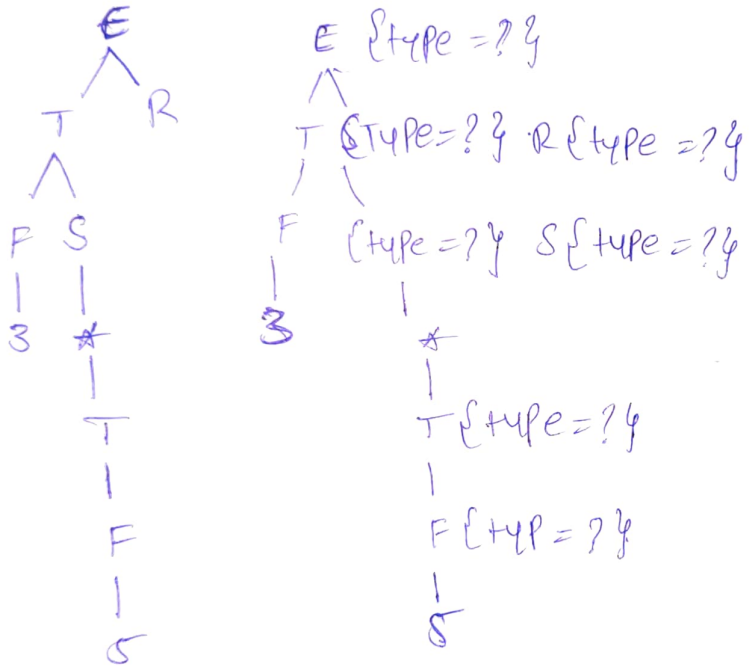
$T \rightarrow FS$

$F \rightarrow n$

$S \rightarrow *T \mid \varepsilon$

$R \rightarrow \varepsilon$

Ans



Rule {E → TR}:-

E {type = T.type}

T {type} R {type}

Rule T → FS:

T {type = F.type}

F {type} S {type}

Rule F → n;

F {type = "int"}

n {type = "int", Value = 3}

Rule S → *T:

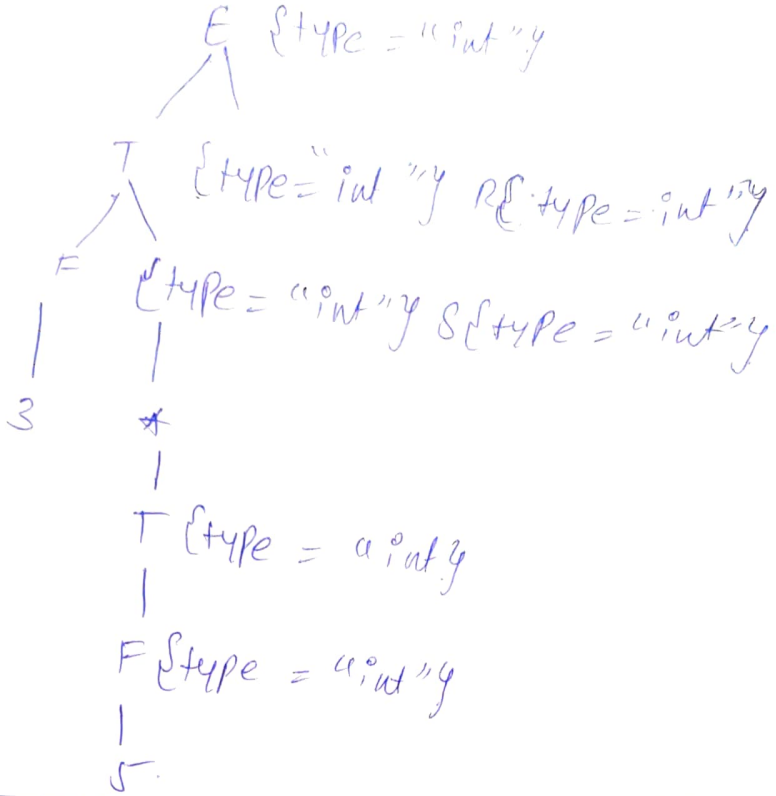
S {type = T.type}

*

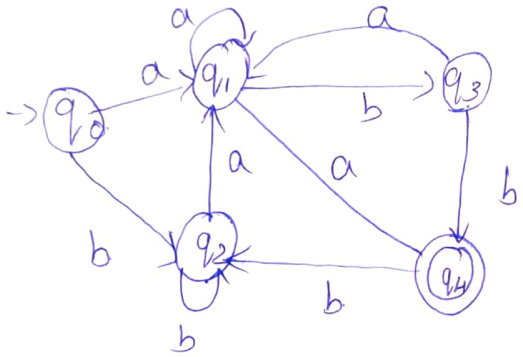
T {type}

Rule R → ε:

R {type = "int"}



2 Minimisation of DFA.



0 - equivalence {A, B, C, D} {E}

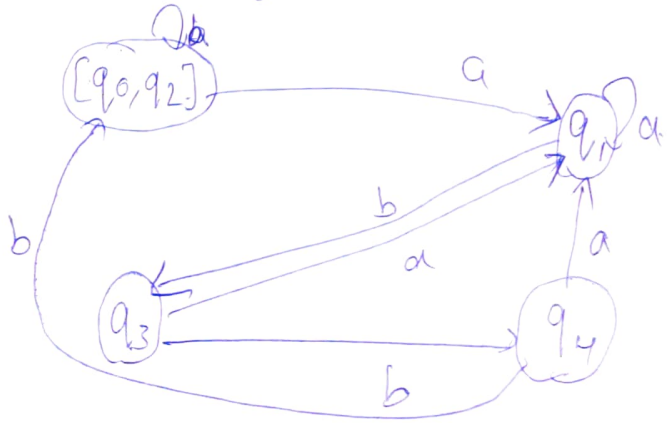
1 - equivalence {A, B, C} {D} {E}

2 - equivalence {A, C} {B} {D} {E}

3 - equivalence {A, C} {B} {D} {E}

	a	b
q_0	q_1	q_2
q_1	q_1	q_3
q_2	q_1	q_2
q_3	q_1	q_4
q_4	q_1	q_3

So new DFA is



VI_sem_K1_K2_CD_1 1680877038 Details ▶

Challenges

Current Rank: 58 View your results

Simple Text Editor

Success Rate: **97.83%** Max Score: **65** Difficulty: **Medium**

Try Again

A Text-Processing Warmup

Success Rate: **61.96%** Max Score: **10** Difficulty: **Medium**

Try Again

Printing Tokens

Success Rate: **100.00%** Max Score: **20** Difficulty: **Medium**

Try Again

Current Leaderboard

Compare Progress

Review Submissions

Message Center

Simple Text Editor

locked

Problem | Submissions | Leaderboard | Discussions

Submitted a month ago • Score: 65.00

Status: Accepted

✔ Test Case #0	✔ Test Case #1	✔ Test Case #2
✔ Test Case #3	✔ Test Case #4	✔ Test Case #5
✔ Test Case #6	✔ Test Case #7	✔ Test Case #8
✔ Test Case #9	✔ Test Case #10	✔ Test Case #11
✔ Test Case #12	✔ Test Case #13	✔ Test Case #14
✔ Test Case #15		

Submitted Code

Language: Python 3

Open in editor

```
1 no_ops = int(input())
2 ops = []
3 s = []
4 for i in range(no_ops):
5     one_op = input().split(' ')
6     ops.append(one_op)
7
8 from copy import copy
9 history = []
10 for op in ops:
11     #print(s)
12     #print(op)
```

A Text-Processing Warmup

locked

Problem | Submissions | Leaderboard | Discussions

Submitted a month ago • Score: 9.63

Status: Processed

✔ Test Case #0 ✔ Test Case #1

Submitted Code

Language: Python 3

Open in editor

```
#!/usr/bin/python
month=['january','february','march','april','may','june','july','august','september','october','november','december']
import sys
if sys.version_info[0]>=3: raw_input=input
for i in range(int(raw_input())):
    txt=''.join(e.lower() for e in raw_input().rstrip() if e.isalnum() or e=='/' or e==' ').split()
    print(sum(e=='a' for e in txt))
    print(sum(e=='an' for e in txt))
    print(sum(e=='the' for e in txt))
    print(sum(e in month for e in txt))
    raw_input()
```

Printing Tokens

locked

Problem | Submissions | Leaderboard | Discussions

Submitted a month ago • Score: 20.00 Status: Accepted

✔ Test Case #0	✔ Test Case #1	✔ Test Case #2
✔ Test Case #3	✔ Test Case #4	✔ Test Case #5
✔ Test Case #6	✔ Test Case #7	✔ Test Case #8
✔ Test Case #9	✔ Test Case #10	

Submitted Code

Language: C Open in editor

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <math.h>
4 #include <stdlib.h>
5
6 int main() {
7
8     char *s;
```

Experiment-1:

Implementation of Lexical Analyzer

Aim:-

To Print the Count of the number of keywords, identifiers, operators, and Symbols. after reading line of code as input, then Store each of these in an array and display them as an output

Algorithm:-

1. Accept String input using getline();
2. Make a function (to vector()) to store element of the input as a vector element.
3. Call each function keyword(), identifier(), operator() and Symbol.
4. Traverse the vector elements for each function call and See if there is a particular match.
5. Increase the Count and display as required.
6. Declare global vector variable to store these details Seperetely.

Code:-

```
#include <bits/stdc++.h>
using namespace std;
vector <string> keys;
vector <string> oper;
vector <string> ident;
vector <string> syms;
vector<string> to-vector(string input)
{
    vector <string> temp;
    int k=0;
    for (int i=0; i<input.length(); i++)
    {
        string part = " "
```

```
if (input[i] == " )
 {
    for (int j = k; j<i; j++)
     {
       part. + = input[j]
     }
      k = i+1;
     temp. push_back (part);
     }
  }
   return temp;
}

int keyword_count (vector<string> parts)
{
   int count =0;
   for (int i = 0; i< parts.size(); i++)
   {
    if (parts[i] == "int" // parts[i] == "float" // parts[i]
== "cout "while // parts[i] == "for")
   {
      count ++;
      keys. push_back (parts[i]);
   }
  }
   return count;
}
int operator_count (vector<string> parts)
{
    int count =0;
    for (int i=0; i< parts. size(); i++)
   { if (parts[i] == "a" // parts[i] == "b" // parts[i]
   == "c" // parts[i] == // parts[j] = "j")
{
```

```
count ++;
  idont.push_back(parts[i]);
 }
}
return count;
}
int operator_count(vector<string> parts)
{
 int count=0;
 for(int i=0;i<parts.size();i++)
 {
  if(parts[i]=="+"||parts[i]=="++"||parts[i]=="-"
||parts[i]=||parts[i]=="=")
 {
  count ++;
   oper.push_back(parts[i]);
 }
}
return count;

}
int separator_count(vector<string> parts)
{ int count=0;
 for(int i=0;i<part.size();i++)
 {if(parts[i]==";"//parts[i]="//")
 {
   count ++;
    syms.push_back(parts[i]);
  }
 }
return count;
}
int main()
{
 string input;
```

```
Count << "How to give input !" << end l;
Count << "1. Give proper space for each & every Syntax" << end
Count << "2. After the input hit space bar and '/' << end l;
Count << "Example -> int a = b+c ; /" << end l;
Count << "Enter proper input" << end l;
getline (cin., input);
Cout << end l;
vector <string> parts;
Parts = to-vector (input);
int key, op, sep, id;
key = keyword-count (Parts);
op = operator - count (Parts);
Sep = separator - count (Parts);
id = identifier - count (Parts);
Count << key << " keywords" << end l;
Count << op << " operators" << end l;
Count << Sep << "separators" << end l;
Count << id << "identifiers" << end l;
Count << end l << "the keywords are ->";
   for (int i = 0; i < keys.size(); i++)
   {
      cout << keys[i] << " ";
   }
Cout << end l << "the operators are ->";
for (int i=0; i < oper.size(); i++)
   {
      Cout << oper[i] << " ";
   }
Cout << end l << " the separators are ->";
For (int i=0; i < syms.size(); i++)
 {
   cout << syms[i] << " ";
}
Cout << end l << "the identifiers are ->";
```

```
    for (int i=0; i< ident.size(); i++)
  {
    cout << ident[i] << " ";
  }
}
```

Input and outputs.

How to give Inputs!

1. Give proper space for each and every syntax.
2. After the input hit space bar and add '/' symbol

   Example → int a=b+c;

Enter proper input (or may not yield proper result→

for (int i=0; i<b; i++) /

2 keywords
2 Operators
2 Seperators
4 Identifiers

The keywords are → for int

The operators are → = + +

The seperators are → ; ;

The identifiers are → i i b ;

Result :-

Hence we have successfully Executed the implementation of lexical analyser code.

Exp-2

Conversion from regular Expression to NFA.

Aim: To write a Program from for converting regular expression to NFA.

Algorithm:- ① Start.

2) get the input from the user.

3) Initialize separate variables and functions from postfix, Display and NFA.

4) Create seperate methods for different operators like +, *...

5) By using switch case Initilize different cases for the input.

6) For '.' operator initialize a seperate methods by using various stack functions to 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.

Code:-

```
#include <stdio.h>
#include <string.h>
int main
{
char reg[20]; int q[20][3], i=0, j=1, len, a, b;
for (a=0; a<20; a++) for (b=0; b<3; b++) q[a][b]=0;
Scanf ("%", reg);
Printf ("given regular expression: %s\n", reg);
 len = strlen (reg);
 while (i<len)
 {
if(reg[i] = ='a' && reg [i+1] != '|' reg [i+1] != '*') {q[j][0]=j+1;
                                                       j++;}
```

Ex
viva completed
[signature] 4/2/2023

```
if (reg[i]=='b' && reg[i+1]!='1' && reg[i+1]!='*'){q[j][1]=j+1;
                                                    j++;}
if(reg[i]=='e' && reg[i+1]!='1' && reg[i+1]!='*'){q[j][2]=j+1;
if (reg[i]=='a' && reg[i+1]=='1' && [i+2]=='b')     j++;}
{
 q[j][2] = ((j+1)*10)+(j+3); j++;
 q[j][0]= j+1; j++;
 q[j][2] = j+3; j++;
 q[j][1] = j+1; j++;
    i= i+2;
 }
 if (reg[i]=='b' && reg[i+1]=='1' && reg[i+2]=='a')
{
 q[j][2] = ((j+1)*10)+(j+3); j++;
 q[j][1] = j+1; j++;
 q[j][2] = j+3; j++;
 q[i][0] = j+1; j++;
 q[j][2] = j+1; j++;
     i= i+2;
}
 if (reg[i]=='a' && reg[i+1]=='*')
{
 q[j][2] = ((j+1)*10)+(j+3); j++;
 q[j][0]= j+1; j++;
 q[j][2] = ((j+1)*10)+(j-1); j++;
}
  if (reg[i]=='b' && reg[i+1]=='*')
{
  q[j][2] = ((j+1)*10)+(j+3); j++;
  q[j][1] = j+1; j++;
  q[j][2] = ((j+1)*10)+(j-1); j++;
```

Input and output

(a|b)* a.

Given regular expression: (a|b)*a

Transition Table.

Current State	Input	Next state
q[0]	e	q[4], q[1]
q[1]	a	q[2]
q[2]	b	q[3]
q[3]	e	q[4], q[1]

```
if (reg[i] == 'j' && reg[i+1] == '*')
{
  q[0][2] = ((j+1)*10)+1;
  q[i][2] = ((j+1)*10)+1;
  j++;
}
i++;
}
Printf ("\n Transition table \n");
Printf ("current state \t Input \t Next State);
for (i=0; i<=j; i++).
{ if (q[i][0] != 0) printf ("\n q[%d] \t a \t q[%d]", i, q[i][0]);
if (q[i][1] != 0) printf ("\n q[%d] \t b \t q[%d]", i, q[i][1]);
if (q[i][2] != 0)
{ if (q[i][2] < 10) printf ("\n q[%d] \t e \t q[%d]", i, q[i][2]);
else printf ("\n q[%d] \t e \t q[%d] q[%d]", i, q[i][2], q[i][2] % 10);
}
}
return 0;
```

Program Compiled
8/2/2023

Result:- The implementation of Converting regular expression to NFA in C was Compiled, Executed and Verified Successfully.

Exp-3

Conversion of NFA to DFA

Aim :- To write a Program for Converting NFA to DFA.

Algorithm:-

* Start
* get the input from the user.
* let the only state in SDFA to "unmarked".
* while SDFA Contains an unmarked State do;
* Let J be that unmarked State.
* b. for each a in % do S = e-closure (move NFA (T,a)) c. if S is not in SDFA already then, add S to SDFA (a dn unmarked State)
  d. Set unmarkedDFA (T,a) to S.
* For each S in SDFA if any S & s is a final State in the NFA then mark S an final State in the DFA
* Print the result
* Stop the Program.

Program :-

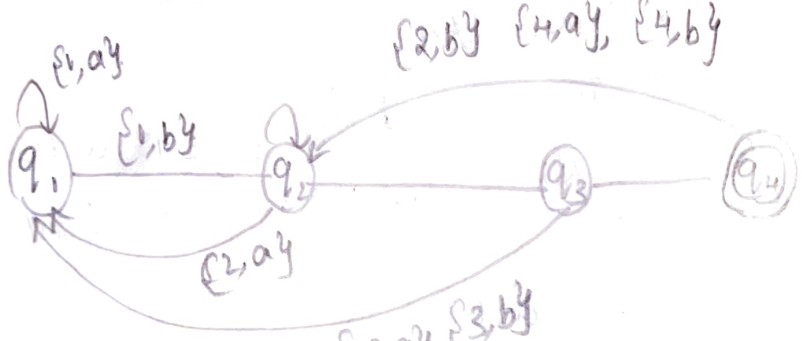
```
#include <vector>
#include <iostream>
using namespace std;
int main()
{
vector < vector <int>> nfa(5, vector <int>(3));
 vector < vector <int> > dfa (10, vector <int> (5));
for (int i=1; i<5; i++){
   for (int j=0; j<=2; j++){
      int n;
```

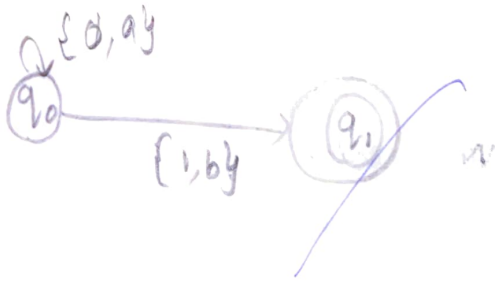
Viva completed

Out Put:

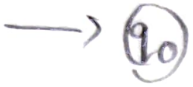
nfa [1,a] = 0
nfa [1,b] = 1
nfa [2,a] = 0
nfa [2, b] = 1
nfa [3,a] = 0
nfa [3, b] = 0
nfa [4,a] = 1
nfa [4,b] = 1

dfa[1,a] : {0,0}
dfa [1,b] : 1





```
        if (j==-1){
        cout<<"nfa ["<<i<<", a]; ";
        }
        else {
        cout<<"nfa ["<<i<<", b]; ";
        }
        cin>>n;
          nfa[i][j]=n;
        }
}
  int dstate[10];
  int i=1, n, j, k, flag=0, m, q, r;
  dstate[i++]=1;
  n=p;
dfa[1][1]=nfa[1][1];
dfa[1][2]= nfa[1][2];
cout<<"\n"<<"dfa["<<dstate[1]<<", a]:{"<<dfa[1][1]/10<<",
 "<<dfa[1][1]%10<<"}";
cout<<"\n"<<"dfa["<<dstate[1]<<", b]: "<<dfa[1][2]
for(j=1; j<n; j++)
 { if(dfa[1][1]!=dstate[j]) flag++;
 }
  if(flag==n-1)
  {
   dstate[i++]=dfa[1][1]; n++;
   }
   flag=0
   for(j=1; j<n; j++)
 {
   if(dfa[1][2]!=dstate[j]) flag++;
}
  if(flag==n-1)
{
  dstate[i++]=dfa[1][2]; n++;
}
```



NFA transition table

	0	1
A	A	{A, B}
B	∅	∅

DFA transition table

	0	1
A	A	[A, B]
[A,B]	A	[A, B]

```
k=2
while(dstate[k]!=0)
{
 m=dstate[k];
 if(m>10)
 {
 q=m/10;
 r=m%10;
}if(nfa[s][1]!=0) dfa[k][1]=nfa[q][1]*10+nfa[r][1];
else
    dfa[k][1]=nfa[q][1]; if(nfa[s][2]!=0)dfa[k][2]=nfa[q]
[2]*10+nfa[r][2];
    else
       dfa[k][2]=nfa[q][2];
 if(dstate[k]>10){  if(dfa[k][1]>10){
cout<<"\n"<<"dfa[{"<<dstate[k]/10<<",";<<dstate[k]%10
<<"},a]:{"<<dfa[k][1]/10<<",";<<dfa[k][1]%10<<"
}
 else{
cout<<"\n"<<"dfa[{"<<dstate[k]/10<<",";<<dstate[k]%
10<<"},a]:";<<dfa[k][1];
    }
  }
  else{
    if(dfa[k][1]>10){
cout<<"\n"<<"dfa["<<dstate[k]<<",a]:{";<<dfa[k][1]
/10<<",";<<dfa[k][1]%10<<"}";
    }
       else{
cout<<"\n"<<"dfa["<<dstate[k]<<",a]:";<<dfa[k][1];
       }
    }
    if(dstate[k]>10){
    if(dfa[k][2]>10){
cout<<"\n"<<"dfa[{"<<dstate[k]/10<<",";<<dstate[k]
%10<<"},b]:{";<<dfa[k][2]/10<<",";<<dfa[k][2]%10
```

```
}
else {
cout<<"\n"<<"dfa[{"<<dstate[k]/10<<","<<dstate[k]%10
<<"}, b]: "<<dfa[k][2];
    }
  }
  else {
    if (dfa[k][1] >10){
cout<<"\n"<<"dfa[{"<<dstate[k]<<",b]:{"<<dfa[k][2]/10
<<","<<dfa[k][2]%10<<"}";
          }
          else {
cout<<"\n"<<"dfa[{"<<dstate[k]<<",b]:"<<dfa[k][2]
        }
    }
    flag=0
    for(j=1;j<n;j++)
    {
      if (dfa[k][1]!=dstate[j])
        flag++;
    }
    if (flag==n-1)
    {
      dstate[i++]=dfa[k][1];
      n++
    }
    flag=0
    for(j=0;j<n;j++)
    {
      if (dfa[k][2]!=dstate[j])
      flag++;
    }
    if (flag=n-1)
    {
```

```
    k++
  }
  return no;
}
```

Result:-

The Program for converting NFA to DFA is Successfully implemented.

Ex1-4(a)

Elimination of left recursion

Aim:- To execute a program for elimination of left recursion.

Algorithm:-

* State the Program.
* Initilize the arrays for taking input from the user.
* Prompt the user to Input the no of terminals having left recursion and no of Productions for these non-terminals
* Prompt the user to input the production for non-terminal
* Eleminate left recursion using following rules

$$A \to A\alpha_1 / A\alpha_2 / \cdots / A\alpha_n$$
$$A \to \beta_1 / \beta_2 / \cdots / \beta_n$$

Then replace it by

$$A \to \beta_i A' \qquad i = 1, 2, 3, \cdots m$$
$$A' \to \alpha_i A' \qquad j = 1, 2, 3, \cdots n.$$
$$A' \to \varepsilon.$$

* After eleminating left recursion by applying these rules display the production without left recursion
* Stop the program.

Code:-

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define SIZE 20

int main()
{
  char pro[SIZE], alpha[SIZE], beta[SIZE];
  int non_terminal, i, j, index = 3;
  Print f ("Enter the Production: (e)
```

Out Put

Enter the production ; E -> E+T/T

Grammar without left recursion;

E -> TE'

E' -> +T.E'/

```
scanf("%s", pro);
non-terminal = = pro[index])
{
  for(i= ++index, j=0; pro[i] != '|'; i++, j++){
    alpha[j] = pro[i];
    if (pro[i+1] == 0){
    Printf("This Grammar cant Be reduced.\n");
     exit(0);
    }
  }
  alpha[j] = "\0";
   if (pro[++i] != 0)
  {
    for(j= i, i=0; pro[j] != '\0'; i++, j++){
          beta[i] = pro[j];
    }
    beta[i] = '\0';
Printf("\n Grammar without left recursion:\n\n");
Printf("%c -> %s %c'\n", non-terminal, beta, non-terminal);
Printf("%c -> %s %c\n", non-terminal, alpha, non-terminal);
}
else print("This Grammar cant be reduced.\n");
}
else
  print("\n This Grammar is not left recursive.\n");
}
```

Result:- Hence, we have successfully created the program for elimination of left recursion.

15/2/2023

EXP-4(b)

Elimination of left factoring

Aim:- To execute a Program for elimination of left factoring.

Algorithm:-

* start
* Ask the user to enter the set of Productions
* Check for common symbols in the given set of Production by comparing with.

$$A \rightarrow \alpha\beta_1 / \alpha\beta_2$$

* If found, replace the particular Problem with.

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 / \beta_2 / \epsilon$$

* Display the output
* exit

Code:-

```
#include <stdio.h>
#include <string.h>
 int main ()
{
  Char gram [20], part 1 [20] part 2 [20], modified gram[20],
 new gram [20], temp gram [20];
int i, j=0, k=0, l=0, pos;
Print f ("Enter Production : A -->");
 gets (gram);
   for (i=0; gram [i] != '|'; i++, j++)
  Part 1 [j] = gram [i];
 Part 1 [j] = \0;
 for (j = ++ i, j = 0; gram [j] != '\0'; j++, i++)
```

Out Put

Enter Production: A->bet acF/betf

Grammar without left factoring ::

A -> bet X

X -> acF/f.

```
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++] = '1';
for(i=Pos; Part2[i] != '\0'; i++ j++){
        newgram[j] = Part2[i];
}
modifiedgram[++k] = 'X';
modifiedgram[++k] = '\0';
newgram[j] = '\0';
Printf("\n grammar without left factoring\n");
Printf("A -> %s", modifiedgram);
Printf("\n X -> %s \n", newgram);
}
```

Result :-

Hence we have successfully executed the elimination of left factoring

Prog 15/2/2023

grammar symbol
Set of rules

Exp-5 First and Follow

Aim:- To write a Program to implement Lexical Analysis using C.

Algorithm :-

First:

To find the first() of the grammar symbol, then we have to apply the following set of rules to the given grammar:-

- If x is a terminal, then First(x) is {x}.
- If x is a non-terminal and x tends to a α is Production, then add 'a' to the first of x. if x -> ε then add null to the First(x).
- If x -> YZ then if First(Y) = ε, then First(X) = {First(Y) - ε} U First(Z).
- If x -> YZ, then if First(X) = Y, then First(Y) = terminal but null then First(X) = First(Y) = terminal

Follow:

To find the follow(A) of the grammar symbol, then we have to apply the following set of rules to the given grammar :-

- $ is a follow of 'S' (Start symbol).
- If A -> αBβ; β != ε, then first(β) is in follow(B).
- If A -> αB or A -> αBβ where First(β) = ε, then everything in Follow(A) is a Follow(B).

Program:

```
//c Program to calculate the first and
// Follow sets of a given Grammar
#include< stdio.h>
#include< ctype.h>
#include< string.h>

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

// function to calculate first void find first (char, int, int);

int count, n=0;
// Stores the final result
// of the first sets.
char calc_first [10][100];
// Stores the final result
// of the first sets.
char calc_first [10][100];
// of the Follow sets
char calc_follow [10][100];
it m = 0;

//stores the production rules
Char production [10][10];
Char f[10], first[10];
int k;
Char ck;
int e;

int main (int argc, char **argv)
{
  int jm = 0;
```

```
int km = 0;
int i, choice;
char c, ch;
count = 8;
// the input grammar
strcpy (production[0], "E = TR");
strcpy (production[1], "R = +TR");
strcpy (production[2], "R = #");
strcpy (production[3], "T = FY");
strcpy (production[4], "Y = FY");
strcpy (production[5], "Y = ##");
strcpy (production[6], "F = (E)");
strcpy (production[7], "F = i");

int kay;
char done[count];
int ptr = -1;
// Initializing the calc first array for(k=0; k<count; k++){
for (kay = 0; kay < 100; kay ++) {
    calc_first[k][kay] = '!';
    }
}
int point1 = 0, point2, xxx;
for (k=0; k< count, k++)
{
    c = production[k][0];
    point2 = 0
    xxx = 0;
// checking if first of c has
// already been calculated for (kay = 0; kay <= ptr; kay++)
        if (c == done[kay])
```

```
if(xxx==1)
    continue;
// Function call.
findfirst(c,0,0);
ptr+=1;
// Adding c to the calculated list done[ptr]=c;
Printf("\n First(%c) = {", c);
calc_first[point1][point2++]=c;
// printing the First sets of the grammar for(i=0+jm; i<n;
  i++){
int lark = 0, chk=0;
    for(lark=0; lark<point2; lark++){
    if(first[i]==calc_first[point1][lark])
    {
        chk=1;
      break;
    }
  }
  if(chk==0)
  {
      Printf("%c, ", first[i]);
      calc_first[point1][point2++]=first[i];
  }
}
  printf("}\n");
  jm=n;
   point1++;
}
Printf("\n");
Printf("-----------\n\n");
char donee[count];
```

```
ptr = -1;

// initialising the calc_follow array for(k=0; k<100; kay++){
        calc_follow[k][kay] = '!';
  }
}
Point1 = 0;
int land = 0;
for (e=0; e< count; e++)
{
  ck = production[e][0];
Point2 = 0;
xxx = 0;
 //checking if follow of ck
//has already been calculated for (kay = 0; kay<= ptr;
kay++)
if (ck == donee[kay])
        xxx = 1;
if (xxx == 1)
        continue;
 land += 1;
 // Function call
   follow(ck);
 ptr += 1;
// Adding ck to the calculated list donee[ptr] = ck;
Printf("Follow (%c) = {", ck);
calc_follow[Point1][Point2] = ck;
// Printing the follow sets of the grammar
for(i=0+km; i<m; i++){
  int lark = 0, chk = 0;
```

```
for (lark=0; lark < point 2; lark ++).
{
    if (f[i] == calc-follow [point 1][lark])
    {
        chk=1;
        break;
    }
}
if (chk == 0)
{
    printf ("%c, ", f[i]);
    calc- follow [point1][point 2++] = f[i];
}
}
printf ("\n\n");
km = m;
point 1 ++;
}
}
void follow (char c)
{
  int i, j;
 // Adding "$" to the follow
 //set of the start symbol
 if (production [0][0] == c) {
     f[m++] = '$';
}
 for (i=0; i<10; i++)
{
  for (j = 2; j<10; j++)
{
   if (production [i][j] == c)
   {
     if (production [i][j+1] != '\0')
```

```
//calculate the first of the next
//Non-Terminal in the production. follow first(production[i][j+1]
p, (j+2));
}
if (production[i][j+1] == '\0' && c!= production[i][0])
{
//Calculate the follow of the Non-Terminal.
//in the LHS of the production. follow (production[i][0]);
                }
            }
        }
    }
}

void findfirst(char c, int q1, int q2)
{
  int j;
  //The case where we
  // encounter a terminal.
if (!(isupper(c))) {
      first[n++] = c;
    }
  for(j = 0; j < count; j++)
{
    if (production[j][0] == 0)
  {
    if (production[j][2] == '#')
  {
    if (production[q1][q2] == '\0')
        first[n++] = '#';
    else if (production[q1][q2] != '\0'.
              && (q1 != 0 || q2 != 0))
```

```
{
//Recursion to Calculate First of
//Non-terminal we encounter
// at the beginning
Find First(Production[i][2], j,3);
        }
    }
}}

Void FollowFirst(Char C, int c1, int c2)
{
  int k;
  // the case where we encounter
//a Terminal
if (!(isupper(c)))
        f[m++] = c;
 else
{ int i=0, j=1;
for(i=0; i<count; i++)
{
  if (calc_first[i][0] == c)
          break;
}
//including the first set of the
// Non-Terminal in the Follow of
// the original query
while (calc_first[i][j] != '!')
{ if (calc_first[i][j] != '#')
{
    f[m++] = calc_first[i][j];
  } else
```

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

16/2/2023

Program Ended

Result: The FIRST and FOLLOW sets of the non-terminals of a grammar were found successfully using ~~Python~~ C.

22/2/23

Predictive Parsing

Exp-6

Aim:- To implement a Program to Construct a Predictive Parsing table for given grammar.

Algorithm:-

1. Start the Program
2. Initate the required Variable
3. Get the number of Co-ordinates and Productives from user
4. Perform the following

for (each Production A -> α in G)
for (each Production in first (A)
Add A -> α to m [A,a]
if (ε is in first (α))
for each Symbol be in follow (A))
Add A -> α to m [A,b]

5. Print the resulting stack.
6. Print the grammar is accepted or not.
7. Exit the Program.

Input & output

Enter no of non terminals

5

Enter the Production in a grammar

E -> TD

D -> +TD / e

T -> FG

G -> *FG / e

~~F -> *FG / e~~ F -> (E) / i

First

FIRST [E] = (i

FIRST [D] = +e

FIRST [T] = (i

FIRST [G] = *e

FIRST [F] = (i

Follow

FOLLOW[E] = $)

FOLLOW [D] = $)

FOLLOW [T] = +$)

FOLLOW [G] = +$)

FOLLOW [F] = *+$

Code:-

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
Void main() {
    char fin[10][20], st[10][20], ft[20][20],
fo[20][20];
    int a=0, e, i, t, b, c, n, k, l=0, j, s, m, p
Printf("Enter the no of non terminals\n");
Scanf("%d", &n);
Printf("Enter the productions in Grammar");
    for(i=0; i<n; i++){
        scanf("%s", st[i]);
    for(i=0; i<n; i++){
        fo[i][0] = '\0';
    for(s=0; s<n; s++){
        for(i=0; i<n; i++){
            j=3
            l=0;
            a=0;
    if((st[i][j]>64)&&(st[i][j]<91))
    {
        for(m=0; m<l; m++){
            if(ft[i][m]==st[a][j])
                goto s1;
        }
    ft[i][0]=st[i][j]
```

M[E,c] = E → TD

M[E,i] = E → TD

M[D,+] = D → +TD

M[D,e] = D → e

M[T,c] = T → FG

M[T,i] = T → FG

M[G,*] = G → *FG

M[G,e] = G → e

M[F,c] = F → (E)

M[F,i] = F → i

o/p verified

Rathi

3/4/23

```
            l= l+1;
            S1: j = j+1;
        }
else if (s>0){
        while (st[i][j] != st[a][a]){
                a++;
        }
            b=0;
        while(ft[a][b] != '\0'){
            for(m=0; m<l; m++){
            if[ft[i][m] == ft[a][b])
                goto S2;
            }
ft[i][j] = ft[a][b];
        l = l+1;
        S2: b = b+1
    }
  }
  }
    while (st[i][j] != '\0'){
        if (st[i][j] == '1'){
            j = j+1;
            goto l;
        }
j = j+1; }
    ft[i][j] = '\0'; } }
Printf("first\n");
  for(i=0; i<n; i++){
  Printf("first[%d] = %s\n", st[i][0] ft[i]);
  for(i[0][0] = '$';
```

```
for (i=0; i<n; i++){
    k=0; j=3;
    if (i==0){
        l=1; }
    else l=0;
k1: while (st[l][0] != st[k][j] && k<n){
        if (st[k][j] == '\0'){
            k++; j=2; j++; }
        j+=1;
    p=0
    while (ft[a][p] != '0'){
        while (ft[a][p] != '@'){
        if (for[i][m] = ft[a][p])
            go to q1;
    }
else
    e=0;
    q2=p++; }
if (e==1){
    e=0; go to a1; }}}
else {
    a1; c=0; a=a;
    while (st[k][0] != st[a][0]){
        a++; }
while ((for[a][c] != '0') && (st[a][0] !=
        st[i][0])){
    for (m=0; m<l; m++){
    if (for[i][m] == for[a][c]){
```

```
goto q1; }
fol[i][l]=fol[a][c]; l++; q1; e++; }}
fol[i][l]=fol[a][c]; l++;
for(i=0; i<n; i++){
printf("follow(%c)= %s\n", st[i][0],
fol[i]);
printf("\n");
s=0;
for(i=0; i<n; i++){
j=3;
while(st[i][j]!='!0'){
for(p=0; p<=2; p++){
for[s][p]=st[i][p]; }
t=j;
while(st[a][0]!=st[i][0]){
a++; }
while(fol[a][b]!='!0'){
printf("m[%c,%c]= %s\n",
b++; }}
else{b=0; a=0;
while(st[a][0]!=st[i][j]){
a++;
j++;
}}
```

Result:-

Hence we successfully executed the predictive Passing

Experiment - 7.

Shift Reduce Parsing

Aim: To write a Program to implement lexical Analysis using C

Algorithm:

* Shift reduce Parsing is a Process of reducing a String to the start ~~symbol~~ Symbol of grammer
* Shift reduce Parsing uses a stack to hold the grammar and an input type to hold the string.
* Shift reduce Parsing perform the two actions, Shift and reduce. That's why it is known as Shift reduce Parsing.
* At the Shift action, the Current Symbol in the input string is Pushed to Stack.
* At each reduction, the Current Symbol will run the input replace by the non-terminal, the Symbol is the right side of the Production and non-terminal is the left side of the Production

Program:

```
#include <Stdio.h>
#include <conio.h>
#include <string.h>
Struct Prodn
{ char R[10];
  char P2[10]; };
void main()
{ char input[20], Stack[50], temp[50], Ch[2], *t1, *t2,
  *t;
int i, j, S1, S2, S, Count=0;
    Stmct Prodn P[10];
```

output:

Input file:

$E \rightarrow E + E$

$E \rightarrow (E)$

$E \rightarrow id$

output:

Enter input String

id * id + id

	Input	action
$ id	* id + id $	Shift -> id
$ E	* id + id $	reduce to E
$ E *	id + id $	Shift + symbols
$ E * i	+ id $	Shift -> id
$ E * E	+ id $	Reduce *
$ E	+ id $	Reduce to E
$ E	id $	Shift -> symbols
$ E + i	$	Shift -> id
$ E + E	$	Reduce to E
$ E	$	Reduce to E

accepted.

O/p verified

```
FILE *fp = fopen("ss-input.txt","r");
Stack[0] = '\0';
Printf("\n Enter the input String \n");
Scanf("%s", &input);
while(!feof(fp))
{ fscanf(fp, "%.s \n", temp);
  t1 = strtok(temp, "->");
  t2 = strtok(temp, "->");
 Strcpy(p[count].p1, t1);
 Strcpy(p[count].p2, t1);
   count++; }
i=0;
while(1)
{ if(i<strlen(input))
 { ch[0] = input[i]; ch[1] = '\0'; i++;
   Strcat(stack, ch);
  Printf("%s\n", stack);
 }
for(j=0; j<count; i++)
{ t = strstr(stack, p[j].p2);
 if(t!=NULL)
{ s1 = strlen(stack); s2 = strlen(t);
   s = s1-s2;
stack[s] = '\0'; strcat(stack, p[j].p1);
Printf("%s\n", stack); j=-1;
}}
```

```
if (strcmp(stack, "E") == 0 && i == strlen(input))
    { printf("\n Accepted");
      break; }
    if (i == strlen (input))
      { printf(" not Accepted");
        break; }
    }
}
```

Result:-

Hence Shift reduce Parsing is Succesfully executed with the above program.

a/3/23

Leading & Trailing

Ex-8

Aim: To implement Program for leading and trailing

Algorithm:

1) If $A \rightarrow \gamma a \beta$ — γ: only non-terminals, a: terminals

Lead(A) = a

2) If $A \rightarrow B$

Lead(A) = Lead(B) {leading}

3) If $A \rightarrow \alpha a \beta$

Lead(A) = a.

} & While statements

Trailing:

1) If $A \rightarrow B \alpha \gamma$

trail(A) = α.

2) If $A \rightarrow B$

trail(A) = trail(B).

3) If $A \rightarrow B \alpha$

trail(A) = α.

Source Code.

```
#include <iostream>
#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
using namespace std;
int vars, terms, i, j, k, m, rep, count, temp = -1;
char var[10], term[10], lead[10][10] trail[10][10];
struct grammar
{
  int prod no;
  char lhs, rhs[10][20];
} gram[20];
void get() {
cout << "\n LEADING AND TRAILING \n";
cout << "\n Enter the no of variables:";
cin >> vars;
cout << "\n Enter the variables: \n";
for (i=0; i< vars; i++){
      cin >> gram[i] lhs;
      var[i] = gram[i].Lhs; }
cout << "\n Enter the no of terminals:";
cin >> terms;
cout << "\n Enter the terminals: ";
for (j=0; j<gram[i].prodo; j++){
```

Out put

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 -> i

F -> (E)

```
        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][i] == term[k])
                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]!='\0')
                count++
```

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

LEADING(T) = *, i, (,

LEADING (T) = i, (,

~~LEAD~~

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

TRAILING (T) = *, i,),

TRAILING (F) = i,).

Gri-

E -> E++

E -> T

T -> T*F

T -> F

F -> id

F -> (E)

o/p ventf

Leading

E = {+, lead(T)}

= {+, * lead(T)}

= {+, *, id, (}

T = {*, lead (F)}

= {*, (, id}

F = {id, (}

Trailing

E = {+, trail(T), *, Trail(F)

= {+, *, id,)}

T = {*, id,)}

F = {id,)}

```
for(k=0; k<terms; k++){
    if(gram[i].rhs[i][count-1]==term[k])
    trail[i][k]=1;
    else{
        if(gram[i].rhs[i][count-2]==term[k],
            trail[i][k]=1;
        }
    }
}
void display(){
    for(i=0; i<vars; i++){
    cout<<"\n LEADING("<<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<<"\n Trailing("<<gram[i].lhs<<")=";
    for(j=0; j<terms; j++){
    if(trail[i][j]==1)
        cout<<term[j]<<", ";
    }
}
}
```

```
int main ( ){
  get ();
leading ();
trailing ()
display();
  }
```

Result :

Hence we successfully executed the Program.

16/8/23 Experiment-9

Computation of LR Items

Aim:- A program to implement LR(0) Items

Algorithm:-

1. Start
2. Create structure for production with LHS, RHS
3. Open file and read input from file
4. Build State 0 from extra grammar law.
   S'->S $ that is add 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 on Left hand side of that law and set Dot in before of first Part of right hand side
6. If State exists, use that instead.
7. Now find Set of terminals and non-terminals in which Dot exist in before.
8. If step 7 set in non-empty go to 4. else go
9. For each terminal / non-terminals 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.

Output:

Enter the productions of the Grammar (0 to End):

E -> E+T

E -> T

T -> T*F

T -> F

F -> (E)

F -> i

0

Augmented grammar

A -> E

E -> E+T

E -> T

T -> T*F

F -> F

F -> (E)

E -> i

The Set of Items OF

I0

A -> . E

E -> . E+T

E -> . T

T -> . T . *F

T -> . F

F -> . (E)

F -> . i

0 point

Program:

```
#include <iostream>
#include <conio.h>
#include <string.h>
char prod[20][20], list of var[20] = "ABCDEFGHIJKLMNOPQ
                                     R";
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][j].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];
```

I_1

A -> E .

E -> E . + T

I_2

E -> T .

T -> T . * F

I_3

T -> F .

I_4

F -> (. E)

E -> . E + T

E -> . T

T -> . T * F

T -> . F

F -> . (E)

F -> . i

I_5

F -> i .

I_6

E -> E + . T

T -> . T * F

T -> . F

F -> . (E)

F -> . i

I_7

T -> T * . F

F -> . (E)

F -> . i

I_8

F -> (E .)

E -> E . + T

I_9

E -> E + T .

T -> T . * F

I_{10}

T -> T * F .

I_{11}

F -> (E)

```
clos[noiut][n].rhs[i]=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]=='.' && is variable.
clos[noitem][i].rhs[j+1]>0){
  for(k=0; k<no var; k+1){
    if(clos[noitem][i].rhs[j+1]==clos[0][k].lhs)
{for(I=0; I<n; I++)
if(clos[noitem][I].Lhs==clos[0][k].Lhs &&
strcmp(clos[noitem][i].rhs, clos[0][k].rhs)==0
break;
if(I==n){

clos[no.item][n].Lhs=clos[0][k].Lhs;
strcpy(clos(no item)[n].rhs, clos[0][k].rhs)
  n=n+1;
}
}
}
}
}
}
}
```

```
arr [no item] = n;
int flag = 0;
for (i=0; i<no-item; i++){
if (arr [i] == n){
for (j=0; j<arr [i]; j++){
int c = 0;
for (k=0; k< arr [i] ; k++)
if (clos [no item] [k].lhs == clos [i][k].lhs && 
strcmp (clos [no item] [k].rhs, clos [i][k].rhs)==0
c = c+1;
if (c == arr[i])
{
flag = 1;
goto exit;
}
}
}
}
exit;;
if (flag == 0)
arr [no-item ++ ] = n; }
```

Result:- Hence we have successfully executed the Program for LR(0).

27/03/23

Experiment - 10

Prgm:

Intermediate code generation: Postfix, prefix.

Aim:- A Program to implement the Intermediate code generation prefix and postfix.

Algorithm:-

* Declare a set of operation.
* Initilize on an empty Stack.
* To convert Infix to postfix follow the following steps.
* Scan the infix expression from left to right
* If the scanned character in the operand, output it.
* Else if the Precedence of the scanned operator is greater in the Stack.
* If the scanned character is an C, push it to the Stack.
* If the Scanned character is an ')' Pop the Stack and output it until a '(' is encountered.
* Scan the expression from left to right
* Whenever the Opperands arise, print them.

Output:-

Expression - A+B^1C/R

Postfix: AB^1CR/+.

Okay

Program:-

```
OPERATION = Set(['+', '-', '*', '/', '(', ')'])

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

def infix to postfix - (formula):
        stack[]
        output = ''
    for ch in formula:
        if ch not in operation
            output += ch
        else if 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()
    print("postfix", output)
    return output
def infix-to-prefix (formula):
    rev = reverse(formula)
    exp = stack[]
```

```
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 ( )
    exp - stack. append (op + b + a)
        op-stack ( )
    else:-
        op = op- stack. pop ( )
        a = exp- stack. pop ( )
        b = exp. stack. pop ( )
    exp - stack. append (
                        op + b + a)
Print (f 'prefix : {exp stack [-1]}').
for ch in formula:-
    if not ch in operators:
        exp - stack. append (ch)
    else if ch = ?
    while op stack [-1] : = '('
```

```
Op = op stack . pop()
a = exp - stack pop()
else.
while op - stack and op stack [-1] != '
                and:
pri [ch] a = pri [op - stack [-1]];
        a = op-stack pop()
        b = exp- stack : pop()
    op - stack append (ch)
Print ( PREfix: { exp- stack [-1] };
    return exp-p-stack [-1].
express = input ("input the expression:")
Pre = infix-to- prefix (express)
post = infix - to- Postfix (express)
```

Result: The Program is successfully executed and Compiled.

Output

Enter the expression: a=b+c-d.

The Intermediate

Code is:-

$t_1 = b + c$

$t_2 = t_1 - d$

$a = t_2$

Done

3/04/23

Experement - 11.

Intermedical · code generation:-

Quadreple ; triple , Indirect triple.

Aim:- Intermedial Code generation -
quadruple, triple, Indirect · triple.

Algorithm:-

The algorithm takes a sequence of three - address statement as input for each three address statement of the form a := b.

* · icancall · the address description · for · y · to determine - y'. If the value · of y currently in memory ang. register both · then prefer the.

* Generate the instruction op z', L where L' is used to show the current location of z. If z is in both then prefer a register memory location.

* If the current · value of y (or) z · have no next use or not live · on exit from the stack · or in register then alter the register descriptor to indicate that after execution of x := y op z. there register will no longer contain y (or) z.

Program:-

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

void small();
    void done(int i);
int P[5] = {0, 1, 2, 3, 2, 4};  C=1, i, k, l, m, p;
char pw[5] = {'-', '-', '+'
    j [20], a[5], b[5], ch[2];
    Void main()
{
  Printf ("Enter the expression: ");
  Scanf ("%s", j);
  Printf ("\t the Intermediate code is : \n");
  small();
}
    Void done (int i)
{
  a[0] = b[0] = '\0'
if ( ) is digit (j[i+2]) && !is digit(j[i+1])
{
  a[0] = j [i-1];
   b[0] = j [i+1];
}
  if (is digit [i+2])
        {
```

```
b[0] = '+';
b[1] = j[i+2];
}
if (isdigit (j[i-2]))
{
    b[0] = j[i+1];
    a[0] = '+';
    a[1] = j[i+2];
    b[1] = '0'
}
if (isdigit (j[i+2]) && isdigit (j[i-2]))
    {
    {
        a[0] = '+';
        b[0] = '+';
        a[1] = j[i+2]
        sprintf (ch, "%d", c);
        j[i+2] = j[i-2] = ch[0];
    }
    if (j[i] == '*')
Printf ("\t+%d = %s * %s\n", c, a, b);
Printf ("%d + %c = + %d", j[i-1] --c);
Printf (ch, "%d", c);
    j[i] = ch[0];
        c++;
        Small(c);
    }
```

```
Void small()
{
  Pi=0; l=0;
  for (i=0; i< strlen(i); i++)
   {
     for (m=0; mas; m++)
       if (i[i]= sw[m])
          if (Pi< P[m])
          {
            Pi = P[m];
            l=1;
       k=i;
       }
     }
     if (l==1)
        done (k);
        else
        exit (0); }
```

Result: The program was successfully compiled and run.

11/4/23 Experiment 12

Design and develop a Simple code generator

Aim: To design and develop a simple code generator

Algorithm:

1. Start
2. get address code sequence
3. Determine current location of B using address
4. If current location is not ready generate more(R,0)
5. Update current location address of A.
6. If current value of B and C is null, Exit
7. If they generate operator, A, B, A opR,
8. Step the move instruction in memory
9. Stop.

Program:

```
#include <stdio.h>
#include <string.h>
Void main()
{
char icode[10][30], str[10], opr[10];
int i=0;
Printf("\n Enter the set of intermediate code
(terminated by exit):\n");
do {
    Scanf("%s", icode[i]);
}
while (Strcmp(icode[i++], "exit")!=0);
Printf("\n Target code generation");
Printf("\n *****");
i=0;
```

Output :-

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

7 - 3

4

exit

Target Code generation

* * * * * *

MOV 3, R0

1, R0

SUB R0, R1

MOV R0 7

MOV, R1

1 1 R1

SUB R0, R1

MOV R1, 4.

```
do{
  strcpy (str, icode[i]);
  switch (str[3])
  {
    case '+':
      strcpy (operator, "ADD");
      break;
    case '-':
      strcpy (operator, "SUB");
      break;
    case '*':
      strcpy (operator, "MUL");
      break;
    case '/':
      strcpy (operator, "DIV");
      break;
  }
  printf ("\n%s\t mov %c, R%d", str[2], i);
  printf ("\n%s\t%c R%d, R%d", opr, str[4], i);
  printf ("\n\t mov R%d, %c", i, str[0]);
  }
  while (strcmp (icode[++i], "exit") != 0);
}
```

Result:- Simple code generator is implemented using C language.

14/11/23 Exp 13

Implementation of DAG

Aim: A Program to implement the DAG.

Algorithm:

-> The leaves of a graph are labeled by unique identifier and that identifier nodes of the graph are labeled by an operator symbol.

-> Nodes are also given a sequence of identifiers for labels to store the computed value.

-> If y operand is undefined then Create node(y)

-> If z operand is undefined then for case i create node (z).

-> For Case (i), Create node (op) whose right child is node (z) and left child is node (y).

-> for Case (ii), check whether there is node (op) with one child node (y).

-> For case(iii), node n will be node (y)

-> Finally set node (x) to

Program:

```
Operator = 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()
    Print f('PostFix : {output}')
    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)
for i in pos:
    if i not in OPERATORS:
        Stack.append(i)
    elif i == '_':
        op1 = Stack.pop()
        Stack.append('%.2f' % (-op1))
print("{0:<5} {1:<5} {2:<5} {3:<5}".
```

Output

Input the expression:

a = b * - c + b * b - c

Prefix :- + - * = * b b c

Postfix = a = b * c - b b * + c -

$t_1 := = * b$

$t_2 := t_1 - c$

$t_3 := b * b$

$t_4 := t_2 + t_3$

$t_5 = t_4 - c$

The quadriple for the expression

op	Arg1	Arg2	Result
*	=	b	t(1)
-	c	(-)	t(2)
+	t(1)	t(2)	t(3)
*	b	b	t(4)
+	t(3)	t(4)	t(5)
-	c	(-)	t(6)
+	t(5)	t(6)	t(7)

The triple for given expression

op	Arg1	Arg2
*	=	b
-	c	(-)
+	t(1)	(1)
*	b	(b)
+	t(3)	(3)
-	c	(-)
+	t(5)	(5)

```
format (i, op2, "(-)", op1)
else:
    op1 = stack.pop()
    op2 = stack.pop()
def triple (pos):
    stack = []
    op = []
    x = 0
for i in pos:
    if i not in OPERATION.
stack.append (i)
elif i == ')':
    op1 = stack.pop()
    stack.append ("t(%s)" % x)
    op2 = stack.pop()
    op1 = stack.pop()
Print ("{i:14s}, {1:14s}, {2:14s}
    format ("t", op1, op2)
    x = x+1
elif i == "=":
    op2 = stack.pop()
    op1 = stack.pop()
    if stack1 = []:
        op2 = stack.pop()
Print ("{i:14s}/{1:14s}/{2:14s}.
    format (i, op2, op1)
    Print ("the triple")
```

Print ("OP1 ARG1 / ARG2 ")

Triple (POS)

Result :-

Hence, the program successfully Compiled & run.