RA2011033010136

INDEX

NAME: HARSHJEET STD.: _____ SEC.: T2 ROLL NO.: _____ SUB.: Compiler Design (LAB)

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1	18/1/2023	Lexical Analyzer	1+1+1	
2	25/1/2023	Regular expression to NFA	3.5/5	
3	2/2/2023	Exp-3 - NFA to DFA	2/2/23.	2.5/5
4.	9/2/2023	Exp-4 - Elimination of left recursion and left factoring	2+3+2	
5.	16/2/2023	first & follow	2+3+2	
6	23/2/2023	Predictive Parsing Table	2+2+3	
7	5/3/2023	Shift Reduce Parser	2+5+3	
8	10/3/2023	Leading & Trailing	2+5+3	
9	17/3/2023	Computation of LR(0)	5+5	
10	27/3/2023	Intermediate Code generation - Postfix, Prefix	2+3+3	
11	3/4/2023	Intermediate Code generation Quadruple, Triple, Indirect Triple	2+5+3	
12	11/4/2023	A simple code generator	2+5+3	

GKC071902 1 C

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
13.	19/4/2023	Exp-13 – Implementation of DAG	2+5+3	10

Completed

22/1/2023

Experiment - 1 - Lexical Analyzer

Lexical Analyzer
- Tokeniser
- Tokenization

Lexical Analyzer program in C.

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

// Returns 'true' if the character
// is a Delimiter.

bool isDelimiter(char ch)
{
  if (ch == ' ' || ch == '+' || ch == '-' ||
ch == '*' || ch == '/' || ch == ',' || ch == ';'
|| ch == '>' || ch == '<' || ch == '=' ||
ch == '(' || ch == ')' || ch == '[' || ch == ']'
|| ch == '{' || ch == '}')
  return (true);
return (false);
}

// Returns true if the character is an
operator.
bool isOperator(char ch)
{
  if (ch == '+' || ch == '-' || ch == '*' ||
ch == '/' || ch == '>' || ch == '<' ||
  ch == '=')
      return (true);
  return (false);
```

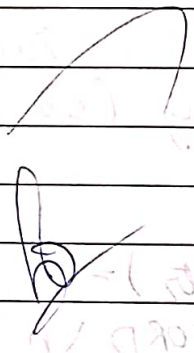
```
}

// Returns 'true' if the string is a Valid
   Identifier.
bool validIdentifier (char * str)
{
  if (str[0]=='0' || str[0]=='1' || str[0]
  =='2' || str[0]=='3' || str[0]=='4' ||
str[0]=='5' || str[0]=='6' || str[0]=='7'
|| str[0]=='8' || str[0]=='9' ||
is Delimiter (str[0]) == true )
     return (false);
     return (true);
}

// Returns 'true' if the string is a Keyword.
  bool isKeyword (char * str)
{
  if (!strcmp (str, "if") || !strcmp(str,
  "else") || !strcmp (str, "while") ||
!strcmp (str, "do") || !strcmp (str, "break")
|| !(str, continue") || !strcmp(str, "int")
|| !strcmp (str, "double") || !strcmp(
str, "float") || !strcmp (str, "return") ||
!strcmp (str, "char") || !strcmp(str, "case") ||
!strcmp (str, "sizeof") || !strcmp (str,
  long") || !strcmp (str "short") ||
!strcmp (str, "typedef") || !strcmp(str,
"switch") || !strcmp (str, "unsigned") ||
strcmp (str, "void") || !strcmp (str,
static") || !strcmp (str, "struct") ||
!strcmp (str, "goto"))
```

```
    return (true);
  return (false);
}

// Returns 'true' if the string is an Integer.
~~bool isInteger~~
bool isInteger (char * str)
{
  int i, len = strlen(str);
  if (len == 0)
      return (false);
  for (i=0; i<len; i++) {
  if (str[i] != '0' && str[i] != '1'
&& str[i] != '2' && str[i] != '3'
&& str[i] != '4' && str[i] != '5'
&& str[i] != '6' && str[i] != '7'
&& str[i] != '8' && str[i] != '9' ||
(str[i] == '-' && i > 0))
    return (false);
  }
  return (true);
}
// Returns 'true' if the string is a Real
  Number.
bool isRealNumber (char* str)
{
    int i, len = strlen(str);
    bool hasDecimal = false;
    if (len == 0)
        return (false);
for (i=0; i<len; i++) {
    if (str[i] != '0' && str[i] != '1'
```

```
&& str[i] != '2' && str[i] != '3' &&
str[i] != '4' && str[i] != '5' &&
str[i] != '6' && str[i] != '7' &&
str[i] != '8' && str[i] != '9' &&
str[i] != '.' ||
(str[i] == '-' && i > 0))
return (false);
if (str[i] == '.')
hasDecimal = true;
}
return (hasDecimal);
}

// Extracts the substring
char * subString (char * str, int left, int right)
{
int i;
char * subStr = (char *) malloc (sizeof (char)
* (right - left + 2));
for (i = left; i <= right; i++)
subStr[i - left] = str[i];
subStr[right - left + 1] = '\0';
return (subStr);
}

// Parsing the input String
void parse (char * str)
{
int left = 0, right = 0;
int len = strlen (str);
while (right <= len && left <= right) {
if (isDelimiter (str[right]) == false)
right++;
```

```
if (isDelimiter(str[right]) == true &&
 left == right) {
  if (isOperator(str[right]) == true)
  printf("'%c' is an operator \n",
     str[right]);
     right++;
  left = right; }

else if (isDelimiter(str[right]) ==
   true && left != right
     || (right == len && left != right)) {
char* subStr = subString(str, left,
  right - 1);

if (isKeyword(subStr) == true)
 printf("'%s' IS A KEYWORD \n", subStr);
else if (isInteger(subStr) == true)
printf("%s is a integer \n", subStr);
 else if (isRealNumber(subStr) == true)
printf("%s is a Real Number \n", subStr);
 else if (validIdentifier(subStr) == true
  && isDelimiter(str[right-1]) == false)
  printf("%s is a Valid identifier \n", subStr);

else if (validIdentifier(subStr) == false
&& isDelimiter(str[right-1]) == false)
printf("'%s is not a valid identifier \n",
    subStr);
  left = Right;
    }
  }
  return;
}
```

```
// Driver Function
int main()
{
    // maximum length of string is 100 here
char str[100] = "int a = b + 1 c;";
 parse(str); // calling the pars
// Calling the parse function

 return(0);
}
```



25/1/2023

Experiment - 2 - Regular Expression to NFA

Aim → Write a C++ program to convert Regular Expression to NFA

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

int ret[100];
static int pos=0;
static int sc=0;
void nfa(int st, int p, char *s)
{
  int i, sp, fs[15], fsc=0;
  sp=st; pos=p; sc=st;
  while(*s!=NULL)
{ if(isalpha(*s))
  { ret[pos++]=sp;
    ret[pos++]=*s;
    ret[pos++]=++sc;}

if(*s=='.')
  { sp=sc;
    ret[pos++]=sc;
    ret[pos++]=238;
    ret[pos++]=++sc;
    sp=sc;}

  if(*s=='|')
    { sp=st;
      fs[fsc++]=sc;}
```

```
if (*s == '*')
 { ret[pos++] = sc;
   ret[pos++] = 238;
   ret[pos++] = sp;
   ret[pos++] = sr;
   ret[pos++] = 238;
   ret[pos++] = sc;
  }
if (*s = '(')
 { char ps[50];
   int i=0, flag = 1;
    s++;
   while (flag != 0)
   { ps[i++] = *s;
     if (*s = '(')
            flag ++;
     if (*s == ')')
            flag --;
      s++; }
   ps[--i] = '\0';
   nfa (sc, pos, ps);
    s--;
   }
s++;
}
sc++
for (i=0; i<fsc; i++)
 { ret[pos++] = fs[i];
   ret[pos++] = 238;
   ret[pos++] = sc;
 }
 ret[pos++] = sc-1;
 ret[pos++] = 238;
```

```
ret [pos++] = sc;
}
void main()
{ int i;
  char * inp;
  clrscr()
  printf("enter the regular expression:");
  gets(inp);
  nfa(1,0,inp);
  printf("\n state input state \n");
  for(i=0; i<pos; i=i+3)
   printf("%d  -- %c -->
%d \n", ret[i], ret[i+1], ret[i+2]);
  printf("\n");
   getch();
}
```

Sample Input / output

Sample input → ab

Sample output :→

Transition Table

Current	Input	Next State
q[1]	a	q[2]
q[2]	b	q[3]

Result → The program to convert RE to NFA is implemented successfully.

2/2/2023

Experiment - 3 - NFA to DFA

Aim — To convert NFA to DFA

```
#include <vector>
#include <iostream>
using namespace std;
int main()
{
  vector < vector <int>> nfa (5, vector<int>
      (3));
  vector <vector <int >> dfa (10, vector <int>
    (3));
for (int i = 1; i<5; i++){
   for (int j = 1; j <=2; j++){
      int h;
      if (j==1){
        cout<<" nfa [" <<i <<",a]: ";
        }
      else {
      cout <<"nfa [" <<i <<",b]: ";
      }
    cin >>h;
    nfa [i][j] = h;
    }
} 
int dstate [10];
int i=1, n, j, k, flag =0, m, q, r;
dstate [i++] = 1;
  n = i;
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++;
}
K = 2;
While (dstate[K] != 0)
{
   m = dstate[K];
   if (m > 10)
   {
      q = m / 10;
      r = m % 10;
   }
```

```
if (nfa[r][1] != 0)
   dfa[k][1] = nfa[q][1] * 10 + nfa
     [r][1];
else
   dfa[k][1] = nfa[q][1];
if (nfa[r][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]/
<<","<<dstate[k]%10<<"},b=a];
{"<<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]:a"<<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=1; j<n; j++)
{
  if (dfa[k][2] != dstate [j])
    flag++;
}
if (flag == n-1)
{
  dstate [i++] = dfa[k][2];
   n++;
}
k++;
}
return 0;
```

Input

```
nfa[1,a] = 1 2
nfa[1,b] : 1
nfa[2,a] : 1 2
nfa[2,b] : 1 3
nfa[3,a] : 1 2
nfa[3,b] : 1 4
nfa[4,a] : 1 2
nfa[4,b] : 1
```

Output

dfa [1, a] : {1, 2}

dfa [1, b] : 1

dfa [{1, 2}, a] : {13, 2}

dfa [{1, 2}, b] : {2, 3}

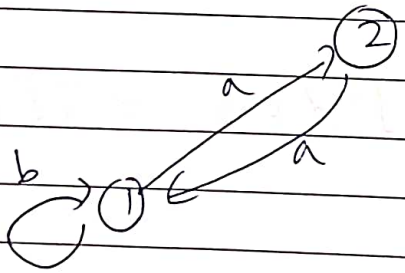
dfa [{13, 2}, a] : {13, 2}

dfa [{13, 2}, b] : {2, 3}

dfa [{2, 3}, a] : {13, 2}

dfa [{2, 3}, b] : {4, 4}

dfa [{4, 4}, a] : {133, 2}



4

3

NFA

Aim – To convert NFA to DFA

Algorithm

1. ~~First~~ Convert the given NFA to its equivalent transition table.
2. Create the DFA's start state
3. Create the DFA's transition table
4. Create the DFA's final state
5. Simplify the DFA ? ? ? ?
6. Repeat the steps 3-5 until no further simplification is possible.

Result – The conversion from NFA to DFA was executed and verified successfully.

9/2/2023

Experiment -4 - Elimination of left recursion and Left Factoring

Aim → To implement elemination of left ~~recurs~~ recursion and Left factoring

Algorithm →
Left Recursion can be eliminated by introducing new non-terminal A ~~such~~ such that.

$A \rightarrow A\alpha \mid \beta$	Removal of left recursion →	$A \rightarrow \beta A'$ $A' \rightarrow \alpha A' \mid \varepsilon$

Left recursive grammer.

$A \rightarrow \beta A'$
$A \rightarrow \alpha A' \mid \varepsilon$

Algorithm to left factor a grammer :-

(1) For each non terminal A find the longest prefix α common to two or more of its ~~alternate~~ alternatives.

(2) If $\alpha \neq \varepsilon$, i.e, there is a non trivial common prefix, replace all the A ~~productio~~ production

Left recursion

$E \rightarrow \underbrace{E}_{A} \underbrace{+ T}_{\alpha} \mid \underbrace{T}_{\beta}$

$E \rightarrow TE'$
$E' \rightarrow + TE' \mid \varepsilon$

Left Factoring

$A \rightarrow a\alpha' \mid a\alpha^2$

~~$A \rightarrow \alpha A'$~~ $A \rightarrow a A'$

$A' \rightarrow \alpha' \mid \alpha^2$

$S \rightarrow Aa \mid Aab \mid Aabb$

$S \rightarrow A a S', \quad S' \rightarrow \epsilon \mid b \mid bb$

$S' \rightarrow \epsilon \mid bS''$

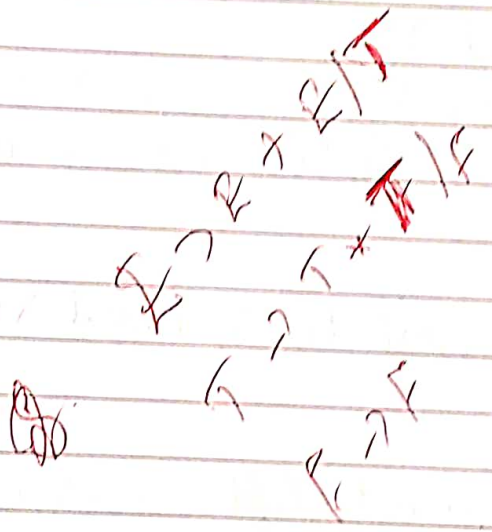
$S'' \rightarrow \epsilon \mid b$

Program

```
# include <stdio.h>
# include <string.h>
# define SIZE 10
~~int main {}~~
int main () {
    char non_terminal;
    char beta, alpha;
    int num;
    char production [10][SIZE];
    int index = 3; /* starting of the string
    following "->" */
    printf ("Enter Number of Production : ");
    scanf ("%d", &num);
    printf ("Enter the grammer as E->E-A : \n");
    for (int i = 0; i < num; i++) {
        scanf ("%s", production [i]);
    }
    for (int i=0; i<num; i++) {
        printf ("\n GRAMMAR : : : %s", production [i]
        );
    non_terminal = = production [i] [index]) {
    { alpha = production [i] [index + 1];
```

```
printf (" is left recursive. \n ");
while ( production [i][index] != 0 && production
[i ][index ] != '|')
index++;

if (production [i ][index ] != 0) {
beta = production [i ][index +1];
printf (" Grammar without left recursion: \n");
printf ("%c -> %c%c\", non_terminal, beta,
non_terminal );
printf ("\n%c' -> %c%c' | E\n",
non_terminal , alpha , non_terminal );
}
else
printf (" can't be reduced \n");
}
else
printf (" is not left recursive. \n");
index =3;
}
}
```

b/ Result – Elemination of left recursion and factoring was executed and verified successfully.



$E \rightarrow E + E \mid T$

$T \rightarrow T * T \mid F$

$F \rightarrow F$

First (E) = ~~{T}~~ {E, T}

First (T) = {T, F}

First (F) = {F, }

Follow (E) = {+, $}

Follow (T) = {*, $}

Follow (F) = {F}

16/02/2023

Exp-5 - First and Follow

Aim → To write a program to perform first and follow using any language.

Algorithm →

For Computing the first :

1) If X is a terminal then FIRST(X) = {X}
Example : F → I | id
We can write it as FIRST(F) → {(, id)
2) If X is a non terminal like E → T then to get FIRST substitute T with other productions until you get a terminal as the first symbol.
3) If X → ε then add ε to FIRST(X).

For Computing the follow :

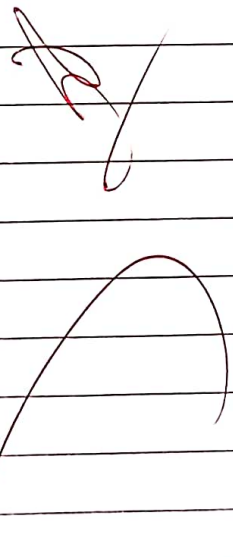
1) Always check the right side of the productions for a non-terminal, whose FOLLOW set is being found. (never see the left side).
2) (a) If that non terminal (S, A, B, ...) is ~~follo~~ followed by any terminal (a, b, ... *, +, (,), ...), then add that terminal into the FOLLOW set.
(b) If that non-terminal is followed by any other non-terminal then add FIRST of ~~other~~ other non terminal into the follow set.

Code →

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

void follow first (char, int, int);
void follow (char c);

void ffidfirst (char, int, int);
int count, n=0;
char calc_follow [10][100];
 int m=0;
char production [10][10];
char f[10], first [10];
int k;

int main (int argg argc, char **argv)
{
   int jm=0;
    int km=0
int i, choice;
 char c, ch;
  count = 8;
// The input grammar
 strcpy (production [0], "E=TK");

int key
char done [count];
int ptr =-1;
for (k=0; k< count ; k++){
 for (kay=0; kay <1000 ; kay++){
    calc_ first[k][kay]='!';
    }
  }
```

```
Void follow (char c)
{
    int i, j;
    if (production [0][0] = c) {
        f [n++] = '$';
    }
for (i=0; i<10; i++)
  for (j=2; j<10; j++)
  {
      if (production [i][j] == c)
      {
          first follow (production [i][j+1],
                  i, (j+2));
      }
if (production [i][j+1] == '\0' && c != 
   production [i][0])
{
    follow (production [i][0]);
        }
        }
    }
}
Void find first (char c, int q1, int q2)
{
    int j;
    if (!(isupper (c))) {
        first [n++] = c;
    }
for (j=0; j<count; j++)
  find first (production [q1][q2], q1,
     (I2+1));
}
```

```
void firstfollow ( 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;
  }

}
```



Result - The first and follow program was executed & verified successfully.

S → iEtS

S → iEtSeS

S → a

E → b

23/02/2022

Exercise - 6 - Computation of Parsing Table

☆ Aim - To construct the parsing table

☆ Algorithm →

Step 1 → First check all the essential conditions mentioned above and go to step 2.

Step 2 → Calculate first and follow for all non-terminals.

(1) First ( ): If there is a variable, and from that variable, if we try to drive all the strings then the Begining Terminal symbol is called the first.

(2) Follow → What is the terminal symbol which follows a variable in the process of derivation.

Step - 3 - For each production $A \rightarrow \alpha$. (A tends to alpha).

(1) Find first (α) and for each terminal in First (α), make entry $A \rightarrow \alpha$ in the table.

(2) If first (α) contains (ε) epsilon as terminal, then find the Follow (A) and for each terminal in Follow (A), make entry $A \rightarrow \varepsilon$ in the table.

(3) If the First (α) contains ε and Follow (A) contains $ as terminal, then make entry $A \rightarrow \varepsilon$ in the table for the $. To construct the parsing table, we have two functions.

$S \rightarrow i\,E\,t\,S$

$S \rightarrow i\,E\,t\,S\,e\,S$

$S \rightarrow a$

$E \rightarrow b$

Removal of left recursion

① $S \rightarrow a\,S'$

$S' \rightarrow e\,S\,S' \mid S' \mid \varepsilon$

$E \rightarrow b$

② Removal of left factoring

$S \rightarrow i\,E\,t\,S\,S'$

$S' \rightarrow \varepsilon \mid e\,S$

First

$S \rightarrow \{S', \varepsilon\}$

$a \rightarrow \{S',\}$

$b \rightarrow \{\varepsilon\}$

Follow

$S \rightarrow \{\$, S'\}$

$a \rightarrow$

$b \rightarrow$

In the table, the rows will contain the Non-Terminals and the columns will contain the Terminal Symbols. All the Null Productions of the grammers will go under the Follow elements and the remaining productions will lie under the elements of the first set.

★ Code.

```
# include <stdio.h>
# include <conio.h>
# include <string.h>
Void main()
{
  char fin[10][20], st[10][20], ft[20][20],
  fol[20][20];
  int a=0, e, i, t, b, c, n, k, l=0, j, s, m, p;
  printf("Enter the no. of nonterminals \n");
  scanf("%d", &n);
  printf("enter the productions in a grammar \n");
  for(i=0; i<n; i++)
    fol[i][0] = '\0';
  for(s=0; s<n; s++)
  {
    for(i=0; i<n; i++)
    {   j=3;
        l=0;
        a=0;
    l1: if (!((st[i][j]>64)&&(st[i][j]<91)))
    {
      for(m=0; m<l; m++)
      {
```

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

```
While (st[i][j] != '\0')
{
  if (st[i][j] == ' ')
  {
    j = j + 1;
    goto l1;
  }
  j = j + 1;
}
ft[i][l] = '\0';
  }
}
printf("first \n");
for (i=0; i<n; i++)
printf("First @[%c]=%s)n",
   st[i][0], ft[i]);
fol[0][0] = '$';
for (i=0; i<n; i++)
{ k=0;
  j=3;
  if (i==0)
    l=1;
  else
    l=0;
K1: while ((st[i][0] != st[k][j]
  && (k<n))
{
  if (st[k][j]=='\0')
  { k++;
    j=2;
  }
  j++;
}
```

```
j = j + 1;
if (st[i][0] == st[k][j-1])
{
    if ((st[k][j] != '|') && (st[k][j]
!= '\0'))
{ a=0;
  if (!((st[k][j] > 64) && (st[k][j]
        < 91)))
{ for (m=0; m < l; m++)
  {
    if (fol[i][m] == st[k][j])
      goto q3;
  }
}
}
```

b)

Result - The construction of parsing table was executed & verified Successfully.

5/3/2022

Exp-7 - Shift Reduce Parser

Aim → To construct Shift Reduce Parser

Algorithm →
Shift → This involves moving symbols from the input buffer onto the stack.
Reduce → If the handle appears on the top of the stack then, its reduction by using appropriate production rule is done. i.e RHS of a production Rule is popped out of stack and LHS of the production rule is pushed onto the stack.
Accept → If only the start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accepted action is obtained, it is means successful parsing is done.
Error → This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

Example - Consider The grammar
$S \rightarrow S+S$
$S \rightarrow S*S$
$S \rightarrow id$
Perform Shift Reduce Parser Parsing for input String "id + id + id".

P.T.O

Stack	Input Buffer	Parsing Action
$	id + id + id $	Shift
$ id	+ id + id $	Reduce S → id
$ S	+ id + id $	Shift
$ S+	id + id $	Shift
$ S+id	+ id $	Reduce S → id
$ S+S	+ id $	Reduce S → S+S
$ S	+ id $	Shift
$ S+	id $	Shift
$ S+id	$	Reduce S → id
$ S+S	$	Reduce S → S+S
$S	$	Accept

Code →

```
# include <stdio.h>
# include <conio.h>
# include <string.h>
 Struct prodn
{ char P1[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;
Struct prodn p[10];
File *fp = fopen("sr_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 (NULL, "->");
        strcpy (p[count].p1, t1);
        strcpy (p[count].p2, t2);
        count++;
    }
    i=0
    while(1)
    {   if (i<strlen (input))
        {   ch[0] = input[i];
            ch[1] = '\0';
            i++;
            strcat (stack, ch);
            printf ("%s\n", stack);
        }
        for (j=0; j<count; j++)
        {   t = strstr (stack, p[j].p2);
            if (t != NULL)
            {   s1 = strlen (stack);
                s2 = strlen (t);
                s = s1 - s2;
                stack[s] = '\0';
                strcat (stack, p[j].p1);
                printf ("%s\n", stack);
                j = -1;
            }
    }
    if (strcmp (stack, "E")==0 && i==
    strlen (input))
    {   printf ("\n Accepted");
        break;
    }
```

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow id / no / a$

$no + a * id$

Stack	Input Buffer	Parsing Action
$	no + a * id $	Shift
$ no	+ a * id $	Reduce $E \rightarrow$ no
$ E	+ a * id $	Shift
$ E+	a * id $	Shift
$ E+ a	* id $	Reduce $E \rightarrow a$
$ E + E	* id $	Reduce $E \rightarrow E+E$
$ E	* id $	Shift
$ E *	id $	Shift
$ E* id	$	Reduce $E \rightarrow id$
$ E* E	$	Reduce $E \rightarrow E*E$
$ E	$	

```
        if (i = strlen(input))
        { printf("\n Not Accepted");
          break;
        }
    }
}
```

Result → Hence, The Shift Reduce parsing was implemented & verified Successfully.

10/3/2023

Exp - 8 - Leading and Trailing

Aim → To demonstrate Leading and Trailing.

Algorithm →

For Leading

Input - Context free grammar G.

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

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

```
begin
For each non-terminal A and terminal a
    L[A, a] = false;
For each production of form A → aα
 or A → Baα
    Install (A, a);
While the stack not empty
  Pop top pair (B, a) from Stack;
  For each production of form A → Bα
Install (A, a);
end
Procedure Install (A, a)
begin
if not L[A, a] then
  L[A, a] = true
    push (A, a) onto Stack.
end
```

For Trailing
Input - Context free Grammar G
Output - TRAILING (A) = {a} iff Boolean Array T[A, a] = true
Method.
begin
For each non terminal A and Terminal a
T[A, a] = false;
For each production of form A → αa or A → αaB
Intall (A, a);
While the stack not empty
Pop top pair (B, a) from Stack;
For each production of form A → α∅B
Install (A, a);
end.
Procedure Install (A, a)
begin
if not T[A, a] then
T[A, a] = true
push (A, a) onto Stack
end.

Code :-

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
void f()
{ printf("Not Operator Grammar");
  exit(0);
}
```

Terminals → +, *, -, no, id

E → E+E

E → E*E

T → T-F

F → no | id

Leading(E) = { +, * }
Leading(T) = { - }
Leading(F) = { no, id }

Trailing(E) = { +, * }
Trailing(F) = { id, no }
Trailing(T) = { -, Trailing(F) }
= { -, id, no }

```
Void main()
{
  char ~~gra~~ gram [20][20], c;
  int i, n, j=2, flag=0;
  scanf("%d", &n);
  for(i=0; i<n; i++)
   scanf("%s", gram[i]);
  for(i=0; i<n; i++){
      c = gram[i][2];
  While(c != '\0'){
  if(gram[i][3]=='+' || gram[i][3]=='-'
  || gram[i][3]=='*' || gram[i][3]=='/')
     flag=1;
  else{
        flag=0;
        }
     }
  if(c=='$'){
     flag=0;
     }
     c= gram[i][~~++~~ ++j];
     }
}
if(flag==1)
printf("Operator grammar");
```

Result → The Operator precedence parsing was executed & ~~verfo~~ verified successfully

Input →

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

17/3/2022

Exp-9 - Computation of LR(0) items

Aim → To write a program to implement LR(0) items.

Algorithm :

1. Start
2. Create structure for production with LHS and RHS.
3. Open file and read input from file.
4. Build State 0 from extra grammar Law S' → S $ that is all start symbol of grammar and one Dot (.) before S symbol.
5. If Dot Symbol is before a non terminal, add grammar laws that this non-terminal is in Left Hand side of that law and set Dot in ~~left~~ 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 goto 9, else goto 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. Goto Step 5
11. End of State Building
12. Display the output
13. End

Program:

```
#include <iostream>
#include <conio.h>
#include <string.h>
using namespace std;
char prod[20][20], listofvar[26] = "ABCDEFGH
IJKLMNOPQR";
int novar = 1, i = 0, j = 0, k = 0, n = 0, m = 0, arr[30];
int noitem = 0;
struct Grammar
{
    char lhs;
    char rhs[8];
} g[20], item[20], clos[20][10];
int isvariable(char variable)
{
  for(int i=0; i<novar; i++)
    if(g[i].lhs == variable)
      return i+1;
    return 0;
}
void findclosure(int z, char a)
{
  int n=0, i=0, j=0, k=0, l=0;
  for(i=0; i<arr[z]; i++)
  {
    for(j=0; j<strlen(clos[z][i].rhs);
    j++)
{   if(clos[z][i].rhs[j] == '.' &&
    clos[z][i].rhs[j+1] == a)
    { clos[noitem][n].lhs = clos[z][i].lhs;
    strcpy(clos[noitem][n].rhs[j]);
    clos[noitem][n].rhs[j] = clos[noitem][n]
                            .rhs[j+1];
```

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

```
for(k=0; k<arr[i]; k++)
if (clos[noitem][k].lhs == clos[i][k].lhs &&
strcmp(clos[noitem][k].rhs, clos[i][k].rhs)
==0)
  c=c+1;
if (c==arr[i])
{   flag = 1;
    goto exit;
}}}}
exit:;
int (flag==0)
  arr[noitem++]=n;
}
int main()
{ cout<<"Enter the productions of the Grammar (0 to END)
\n";
do { cin>>prod[i++];
  }
while (strcmp(prod[i-1], "0")!=0);
for(n=0; n<i-1; n++)
{
  m=0;
  j = novar;
  g[novar++].lhs = prod[n][0];
for (k=3; k<strlen(prod[n]); k++)
{ if (prod[n][k] != '|')
g[j].rhs[m++] = prod[n][k];
if (prod[n][k] == '|')
{  g[j].rhs[m] = '\0';
     m=0;
     j = novar;
   g[novar++].lhs = prod[n][0];
}}}
```

```
for (i=0; i<26; i++)
if (! isvariable (listofvar[i]))
    break;
g[0].lhs = listofvar[i];
char temp[2] = {g[1].lhs, '\0'};
Strcat (g[0].rhs, temp);
Cout << "\n\n augumented grammar \n";
for (i=0; i< novar; i++)
cout << endl << g[i].lhs << "->" << g[i].rhs
<< " ";
for (i=0; i< novar; i++)
{ clos[noitem][i].lhs = g[i].lhs;
Strcpy (clos[noitem][i].rhs, g[i].rhs);
if (strcmp (clos[noitem][i].rhs, "ε") == 0)
Strcpy (clos[noitem][i].rhs, ".");
else
{ for (int j = Strlen (clos[noitem][i].rhs)+1;
j >=0; j--)
clos[noitem][i].rhs[j] = clos[noitem][i].
rhs[j-1]; clos[noitem][i].rhs[0] = '.';
}}
```

Result → The program for computation of LR[0] was successfully compiled and run.

27/3/2023

Lab Exp - 10 - Intermediate Code Generation - Postfix, Prefix

Aim → A program to implement Intermediate code generation - Postfix, Prefix.

Algorithm -

(1) Declare Set of operators.
(2) Initialize an empty stack.
(3) To convert Infix to Postfix follow the ~~folling~~ following steps.
(4) If the scanned character is an operand, output it.
(5) If the scanned character is an operand, output it.
(6) Else, 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.
(7) 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.
(8) If the scanned character is an '(', push it to the stack.
(9) If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
(10) Pop and output from the stack until it is not empty.
(11) To convert Infix to PREFIX follow the following steps.

$(a+b)+c*(D+E)$

~~$(ab+)+c*(DE+)$~~
~~$(ab+c+)*(DE+)$~~
~~$(ab+c+DE+*)$~~

$(ab+)+c*(DE+)$
$(ab+c+)*(DE+)$
$ab+c+DE+*$

Input	Stack	Print
(	(	
a	(a	a
+	(+	a
b	(+	ab
)	)	ab +
+	+	
c		ab + c
*	+ *	ab + c
~~(~~	~~+ * (~~	~~ab + c~~
(	+ * (	
D		ab + c D
+	+ * (+	
E		ab + c DE +
)		ab + c DE + * +

ab + c DE + * +

Final Answer

(12) First, reverse the infix expression given in the problem.
(13) Scan the expression from left to right.
(14) Whenever the operands arrive, print them.
(15) If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
(16) Repeat steps 6 to 9 until the stack is empty.

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_stad = []
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("PREFIX: ",exp_stack[-1])
return exp_stack[-1]
express = input("Input The Expression: ")
pre = infix_to_prefix(express)
pos = infix_to_postfix(express)
```

Result → The program was successfully compiled and run.

$a = (-b+c) * c / d$

Quadruple

$a = (-b+c) * c / d$

$t_1 = \text{uminus } b$

$t_2 = t_1 + c$

~~$t_3 = d$~~

~~$t_4 = c / t_3$~~

$t_3 = c / d$

$t_4 = t_2 * t_3$

$a = t_2 * t_3$

	Op	arg 1	arg 2	result
(0)	uminus	b		t_1
(1)	+	t_1	c	t_2
(2)	/	c	d	t_3
(3)	*	t_2	t_3	t_4
(4)	=	t_2	t_3	

Triple

Op, arg 1, arg 2

	Op	arg 1	arg 2
(0)	uminus	b	
(1)	+	(0)	c
(2)	/	c	d
(3)	*	(1)	(2)
(4)	=	(1)	(2)

3/4/2023

Lab-11- Intermediate Code Generation - Quadruple, Triple, Indirect triple

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

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

1. Invoke a function getreg to find out the location L where the result of compolation b op c should be stored.
2. Consult the address description for y to determine y'. If the value of y currently in memory and register both then prefer the register y'. If the value of y is not already in L then generate the instruction MOV y', L to place a copy of y in L.

3. Generate the instruction OP z', L where z' is used to show the current location of z. If z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all other descriptors.

Indirect Triple

(0)	(40)
(1)	(41)
(2)	(42)
(3)	(43)
(4)	(44)

	Op	arg 1	arg 2
(40)	uminus	b	
(41)	+	(40)	c
(42)	/	c	d
(43)	*	(41)	(42)
(44)	=	(41)	(42)

$a = (-b + c) * c / d$

$t_1 = \text{uminus } b$

$t_2 = t_1 + c$

$t_3 = t_2 * c$

$t_4 = t_3 / d$

$a = t_3 / d$

	Op	arg 1	arg 2	result
(0)	uminus	b		t_1
(1)	+	t_1	c	t_2
(2)	*	t_2	c	t_3
(3)	/	t_3	d	t_4
(4)	= a	t_3	d	a

(4) If the current value of y or z have no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of X := y op z those register will no longer contain y or z.

Program:

```
# include <stdio.h>
# include <ctype.h>
# include <stdlib.h>
# include <string.h>
Void Small ();
 void dove (int i);
 int p[5] = {0,1,2,3,4}, c=1, ik, l, m, P, i;
char sw[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 ();
  }
  a[0] = b[0] = '\0';
if (! is digit [j[i+2]) && ! is digit (j[i-2])
{
  a[0] = j[i-1];
  b[0] = j[i+1];
  }
  if (is digit (j[i+2])) {
   a[0] = j[i-1];
```

Triple

OP, arg 1, arg 2

	OP	arg 1	arg 2
(0)	uminus	b	
(1)	+	(0)	c
(2)	*	(1)	c
(3)	/	(2)	d
(4)	=	(2)	d

Indirect Triple

(0)	(40)
(1)	(41)
(2)	(42)
(3)	(43)
(4)	(44)

	OP	arg 1	arg 2
(40)	uminus	b	
(41)	+	(40)	c
(42)	*	(41)	c
(43)	/	(42)	d
(44)	=	(42)	d

```
b[0] = g[i+1];
a[0] = 'x';
a[1] = g[i-2];
b[1] = '0';
}
if (isdigit(g[i+2]) && isdigit(g[i-2]))
{
a[0] = 'x';
b[0] = 'x';
a[1] = g[i-2];
b[1] = g[i+2]

for (i=0; i<strlen(g); i++)
{ for m=0; m<5; m++)
if (g[i] == sw[m])
if [pi <= p[m])
{
pi = p[m];
l=1
k=i;
}
}
if(l==1)
dove(k);
else
exit(0); }
```

Result — The program was successfully compiled & Run

11/4/

Viva Ques

$a = (-b + c) * c / d$

$t_1 = $ uminus b

$t_2 = t_1 + c$

$t_3 = t_2 * c$

$t_4 = t_3 / d$

$a = t_3 / d$

R_0, R_1

Registers are empty

Statements	Code generated	Register Descriptor	Address Descriptor
$t_1 =$ uminus b	SUB b, R_0	R_0 contains t_1	t_1 is R_0
$t_2 = t_1 + c$	MOV c, R_1 ADD R_0, R_1	R_0 contains t_2 R_1 Contains c	t_2 in R_0 c in R_1
$t_3 = t_2 * c$	MUL R_0, R_1	R_0 Contains t_3 R_1 Contains c	t_3 in R_0 c in R_1
$t_4 = t_3 / d$	MOV d, R_1 DIV R_0, R_1	R_0 Contains t_4	t_4 in R_0 t_4 in R_0 and Memory.

11/4/2023

Exp - 12 - A simple code generator

Aim → To design and develop a simple code generator (back end of the compiler).

Algorithm:

- Start
- Get address code sequence
- Determine current location of 3 using address
- If current location is not ready generate more (R,0)
- Update the ~~address~~ address of A (for 2nd operand)
- If current value of B and () is null, Exit.
- If they generate operator (), A, 3, ADR.
- Stop the move instruction in memory.
- Stop.

Program

```
# include <stdio.h>
# include <string.h>
 void main()
{
char icode[10][30], str[20], opr[10];
int i=0;
Printf("\n Enter the set of intermediate code
(terminated by exit): \n");
do {
    Scanf("%s", icode[i]);
  }
```

```
where (strcmp(icode[i++], "Edit") != 0);
Printf("In target code generation");
Printf("\n ********");
i = 0;
do {
Strcpy Strcpy(str, icode[i]);
Switch(Str[3]){
    case '+':
    Strcpy(opr : "ADD");
    break;
    Case '-':
    Strcpy(opr "SUB");
    break;
    Case '*':
    Strcpy(opr "MUL");
    break;
    Case '/':
    Strcpy(opr "DIV");
    break
}
Printf("\n\t MOV %c, R%d ; Str[2], i);
Printf("\n %s %c, R%d ; opr, Str[4], i);
Printf("\n\t MOV R%d %c", i, Str[0]);
}
While(Strcmp(icode[++P], "Exit") != 0);
}
```

Result - Simple Code generation is implemented using C language.

Case (i) x := y OP z

Case (ii) x := OP y

Case (iii) x := y

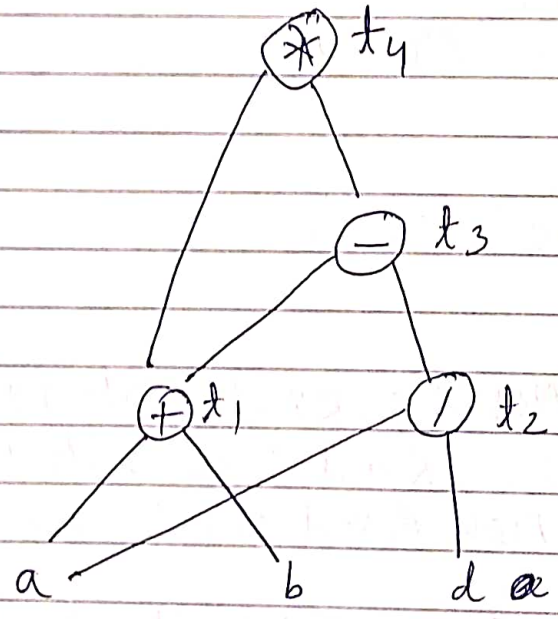
$(a+b) - (a/d) * (a+b)$

$t_1 = a + b$

$t_2 = a / d$

$t_3 = t_1 - t_2$

$t_4 = t_3 * t_1$



$A = a + b$

$B = a / d$

$C = A - B$

$D = C * A$

19/4/2023

Exp - 13 - Implementation of DAG

Aim - ~~Code~~ Write a program to Implement DAG

Algorithm →
Input : It contains a basic block.
Output : It contains the following information :
- Each node contains a label : For leaves, the label is an identifier.
- Each node contains a list of attached identifiers to hold the computed ~~values~~ values.

Method →
Step - 1
If y operand is unidentified then create node(y).
If z operand is unidentified then for case(i) create node(z)
Step - 2
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).
Output →
For node(x) delete x from the list of identifiers. Append x to attached identifiers list for the node n found in step 2. Finally set node(x) to n.

Code →

```
#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] = "A = x+y";
st[1] = "B = A*z";
st[2] = "C = B/x";
unordered_map <char, DAG* > labelDAGNode;
```

```
for ( int i=0; i<3; i++){
    String St Temp = St [i];
for ( int j=0; j<5; j++){
    char temp Label = St Temp [0];
    char temp Left = St Temp [2];
    char temp Data = St Temp [3];
    char temp Right = St Temp [4];
    DAG * left Ptr;
    DAG * right Ptr;
if ( label DAG Node . count ( temp left )==0){
    left Ptr = new DAG ( temp left );
}
else {
    left Ptr = label DAG Node [temp left];
}
if (label DAG Node . count (temp Right )==0){
    right Ptr = new DAG ( temp Right );
}
else {
right Ptr = label DAG Node [ temp Right ];
}
DAG * nn = new DAG (temp Label, temp Data, left Ptr, right Ptr);
label DAG Node . insert ( make_pair ( temp Label, nn));
    }
}
Cout <<" Label ptr left ptr right ptr " << endl;
for ( int i=0; i<n; i++ ){
    DAG * x = label DAG Node [St [1][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;
}
```

Result → Implementation of DAG is executed & verified successfully.