INDEX

NAME: Anshika STD.: TSE SEC.: T2 ROLL NO.: 106 SUB.: C.D practical

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1.	18/1/23	Exp1:- Lexical Analyzer	~~7~~ 17 2+2+6	5/5
2.	25/1/23	Exp2:- Regular exp to NFA	~~8~~ ~~5~~ 5/5	
3	2/2/23	Exp3:- NFA to DFA	~~5+3~~ 5	10/10
4.	8/2/23	Exp4(a): Left factoring	5 } 5+	
~~5~~	9/9/23	Exp4(b): Left recursion		
5	16/9/23	Exp 5: First & Follow	9+3+5	17/3
6	23/2/23	Exp6: Predictive Parsing		
7	6/3/23	Exp 7: Shift Red Parser	2+5+3	
8	14/3/23	Exp 8: Operator precedence Parsing	2+5+3 10	
9.	17/3/23	Exp 9: LR(0) parser	10	AA 17/3/2023
10	27/3/23	Exp10: Postfix Prefix	2+3+5	
11	5/4/23	Exp11: Quadruple, Triple, IT	2+3+5	
12	11/4/23	Exp12: S Code Generation	2+3+3	
13	19/4/23	Exp13: Code Optimization	2+5+3	
14	29/4/23	~~Exp14(a) Stack impl~~ Exp 15(a) Stack impl	2+5+3	AA
15	19/4/23	Exp15(b) Stack Heap	2+2+6	AA

Date: 18/1/23

Experiment: 1

Lexical Analyser

Aim: Implementation of Lexical Analyser

Code:

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

bool isDelimeter (char ch)
{ if (ch == ' ' || ch == ';' || ch == '(' || ch == ')' || ch == '['
||ch == '{' || ch == ']' || ch == '}')
  return (true);
  return (false);
}

bool isOperator (char ch)
{
  if (ch == '+' || ch == '-' || ch == '*' || ch == '/'
  ||ch == '>' || ch == '<' || ch == '=')
      return (true);
      return (false);
}
  bool validIdentifier (char* str)
```

```
{ if (str[0]=='0' || str[0]=='1' || str[0]=='2' ||
  str[0]=='3' || str[0]=='4' || str[0]=='5'
 || str[0]=='6' || str[0]=='7' || str[0]=='8' ||
  || str[0]=='9' || isDelimeter (str[0])==true)
   return (false);
    return (true);
}
bool iskeyword (char *str)
{ if (!strcmp (str, "if") || !strcmp (str, "else") ||
  !strcmp (str, "while") || !strcmp (str, "do") ||
 !strcmp(str,"while") || !strcmp (str, "break") ||
 !strcmp (str,"continue") || !strcmp (str, "double")
 || !strcmp (str,"return") || !strcmp (str,"sizeof") ||
 !strcmp (str,"void") || !strcmp(str,"void") ||
  !strcmp(str,"main") || !strcmp(str,"int") ||
 !strcmp (str,"float") || !(str,"char") || !strcmp
 (str,"static")) || !strcmp(str,"stdio.h"))
    return (true);
   return (false);
}
bool isInteger(char* str)
```

```
{ int i, len = strlen(str);
  for(i=0; i<len; i++)
  { if(str[i]!='0' && str[i]!='1' && str[i]!=
    '2' && str[i]!='3' && str[i]!='4' && str[i]!=
    '5' && str[i]!='6' && str[i]!='7' &&
     str[i]!='8' && str[i]!='9' || (str[i]=='_'
    && i>0))
        return (false);
    }return(true);
  }
  bool isSpecialsymbol(char*str)
  { if(ch=='#' || ch==';' || ch=='(' || ch==
    ')' || ch=='{' || ch=='}' || ch==',')
      return(true);
    return(false);
  }
  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]!='
     && str[i]!='8' && str[i]!='9'
```

```
&& str[i] != '.' || (str[i] == '-' && i > 0))
    return (false);
  if (str[i] == '.')
  { hasDecimal = true;
return (hasDecimal);
}
void parse (char *str)
{ int left = 0, right = 0;
  int len = strlen (str);
while (right <= len && left <= right);
if (isDelimeter(str[right]) == false)
    right++;
if (isDelimeter (str[right]) == true)
  printf ("%c Is operator\n", str[right]);
  right++;
  left = right;
} else if (isDelimeter (str[right]) == true &&
left != right || (right == len && left != right))
{ char *subStr = subString (str, left, right,
    right-1);
if (iskeyword (subStr) == true)
    printf ("keyword\n", subStr);
```

```
else if (isInteger(subStr) == true)
    printf("'%s' Integer \n", subStr)
else if (isSpecialSymbol(subStr) == true)
    printf("'%s' Special Symbol \n", subStr)
else if(isRealNumber(subStr) == true);
    printf("'%s', RealNumber, subStr);
 else if(validIdentifier(subStr) == true && isDelimeter(
(str[right-1] == false);
printf("'%s', Delimeter \n");
else if(validIdentifier(subStr) == false && isDelimeter(
 str[right-1] == false);
printf("'%s', Not valid Identifier \n"),
    left = right;
    }
  }
   return;
}
 int main()
   { char str[100] = "#include <stdio.h>
                      { int a=10, b=5, c;
                        c = a+b;
                      } return; "

    parse(str);
    return(0);
}
```

System Input: (Simple)

```
# include<stdio.h>
{ int a=10, b=5, c ;
  c=a+b;
}return;
```

Output:

Token	Type
#	special symbol
include	keyword
<	operator
stdio.h	keyword
>	operator
{	special symol
int	keyword
a	identifier
=	operator
10	integer
,	special symbol
b	identifier
=	operator
5	integer
c	identifier

;	special symbol
c	identifier
=	operator
a	identifier
+	operator
b	identifier
;	special symbol
}	special symol.
return	keyword
;	special symol.

Result:- Implementation of lexical analyser has been performed successfully.

Date: 25/1/23

Experiment-2:

Regular Expression to NFA

Aim: Implementation of program to perform conversion of regular expression to NFA

Code:

```
# 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; }
```

```
{ sp=sc;
  ret[pos++]=sc;
  ret[pos++]=238;
 ret[pos++]=++sc;
   sp=sc;}
if(*s=='1')
      {sp=st;
      fs[fsc++]=sc;}
   if(*s=='*')
    {ret[pos++]=sc;
     ret[pos++]=238;
     ret[pos++]=sp;
     ret[pos++]=sp;
     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:

ab

Output:

Transition Table

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

Result: Conversion of Regular expression to NFA has been successfully done through the above program.

31/1/23

Experiment - 3

Conversion of NFA to DFA

Aim: To write a program for converting NFA to DFA.

Algorithm:

1] Start

2] Get the input from the user

3] Set the only state in S DFA to "unmarked"

Code:

```
import pandas as pd
nfa = {}
n = int(input("No. of states : "))
t = int(input("No. of transitions : "))
for i in range(n):
    state = input("state name : ")
    nfa[state] = {}
    for j in range(t):
        path = input("path : ")
        print("Enter end state from state {} travelling through path {} : ".format(state, path))
        reaching_state = [x for x in input().split()]
        nfa[state][path] = reaching_state
```

```
print("\n NFA :- \n")
print(nfa)
print("\n Printing 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()]
new_state_list=[]

dfa = { }
keys_list = list(list(nfa.keys())[0])
path_list =list(nfa[keys_list[0]].keys())

dfa[keys_list[0]] ={ }
for y in range(t):
    var ="".join(nfa[keys_list[0]][path_list[y]])

dfa[keys_list[0]][path_list[y]]=var
if var not in keys_list:
    new_states_list.append(var)
    keys_list.append(var)
while len(new_states.list)!=0:
    dfa[new_states_test[0]]={ }
for _ in range(len(new_states_list[0])):
```

```
for i in range (len (new_states_list [0])):
    for i in range (len( path_list)):
        temp = []
    for j in range ( len(new_states_list [0])):
        temp+ = nfa[new_states_list [0][j]][path_list[i]]
        S = " "
    S = S.join(temp)
    if S not in keys_list :
        new_states_list.append(S)
        keys_list.append (S)
    dfa [new_states_list [0]][path_list [0]] = S
    new_states_list.remove( new_states_list [0] )
print ("\n DFA :-\n")
    print (dfa)
print ("\n Printing DFA table :-")
    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_state.append (x)
                    break
print ("\n Final States of the DFA are : ",dfa_final_state)
```

Input :

No. of states = 3
No. of transitions = 2
state name : A
path : 0
Enter end state from state A travelling through path 0 : A
path = 1
Enter end state from state A travelling through path 1 : A B
state name : B
path : 0
Enter end state from state B travelling through path 0 : C
path : 1
Enter end state from state B travelling through path 1 : C
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 1

NFA :- {'A' : {'0' } : ['A'], '1' : ['A', 'B'] }, 'B' :
{'0' : ['C'], '1' : ['C'] }, 'C' : {'0' : [ ], '1' : [ ]}}

Printing NFA table :-

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

Enter final state of NFA : C

Output:

DFA :-

{'A': {'0': 'A', '1': 'AB'}, 'AB': {'0': 'AC', '1': 'ABC'}, 'AC': {'0': 'A', '1': 'AB'}, 'ABC': {'0' = 'AC', '1': 'ABC'}}

Printing DFA table :-

	0	1
A	A	AB
AB	AC	ABC
AC	A	AB
ABC	AC	ABC

Final-states of the DFA are : ['AC', 'ABC']

Result: The given NFA was converted to a DFA using python successfully.

Date 2/2/23

Experiment - 4 (a)

Left Factoring

Aim: To write a program to perform left factoring in C.

Code:

```
#include<stdio.h>
#include<string.h>
int main()
{
  char gram[20], part1[20], part2[20],
  modificationGram[20], newGram[20], tempGram[20];
  int i, j=0, k=0, l=0, pos;
  printf("Enter Production: A->");
  gets(gram);
for(i=0; gram[i]!='|', i++, j++)
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])
  {
    modificationGram[k]=part1[i];
    k++;
    pos=i+1;
} }
```

```
for(i=pos, j=0; part1[i]!='\0'; i++, j++){
    newGram[j]=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];
}
modificationGram[K]='S';
modificatorGram[++k]='\0';
newGram[j]='\0';
printf("\nGrammer without left Factoring : :\n");
Printf("A->%s", modificationGram);
Printf("\nX->%s\n", newGram);
}
```

Input :- Enter production : A -> bSSaaS|bSSaSb|bSb|a

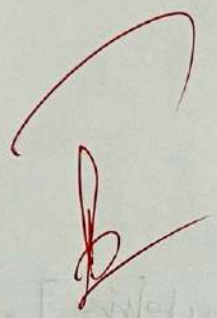
~~bSSaaS|bSSaSb|bSb|a~~

Output :- Grammar without Left Factoring : :

A -> bSSaS

X -> aS|Sb|bSb|a

Result: Left factoring the given expression has been performed successfully using C.



Date: 9/2/23 **Experiment 4(B): Left Recursion**

Aim: To write a program to perform left recursion

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;

printf("Enter the productions as E->E|A:");
scanf("%s", pro);

nonterminal = pro[0];
if(non_terminal == pos[index])
{ for(i = ++index, j = 0, pro[i] != '|';
      i++, j++) {

    alpha[j] = pro[i];
if (pro[i+1] == 0)
{ printf(" This grammar can't
```

```
be reduced.");
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", non_terminal, beta, non-
terminal);
printf("%c' -> %s%c' | # \n", non_termind, alpha,
      non_termind);
}else
  printf("This Grammar Can't be Reduced.\n");
}else
printf("\n This Grammar is not left recursiv
                                    );
}
```

Input/Output : Enter the production as E → E/A

E → E + T | T

Grammar without LEFT recursion :-

E → TE'

E' → + TE' | #

Date: 16/2/23

Experiment - 5

First and Follow

Aim: To write a program to implement First and follow in C++.

Algorithm:

1) If X is a terminal then first (x) = {x}
2) If X is a non-terminal like E → T, then to get first substitute T with other production until you get.
3) If x → ε then add ε to first(x)

Code:

```
# include <stdio.h>
# include <math.h>
# include <ctype.h>
int n, m=0, p, i=0, j=0;
Char a[10][10], f[10];
void follow(char c);
void first (Char c);
int main(){
```

```
for(i=0; i<m; i++)
 printf("continue"(0/1)?");
 } while(z==1);
 return (0);
 }
void first(char c)
 { int k;
  if(!isupper(c))
    f[m++]=c;
    for(k=0; k<n; k++)
  { if(a[k][0]==c)
    { if(a[k][2]=='$')
   follow(a[k][0]);
    f[m++]=a[k][2];
  }
 }
 } void follow(char c)
 { if(a[0][0]==c),
    f[m++]="$";
  { if(a[i][j]==c)
    { if(a[i][j+1]!='\0')
     first[a[i][j+1]);
```

S → aAB | bA | c

A → aAb | ε

B → bB | ε

```
follow(a[i][0]);
}
}
}
}
```

Input/output :

Enter the no. of production : 5

Enter the productions : S = AbCd
A = Cf
A = a
C = gE
E = h

Enter element whose first and follow is to be found : S

first(S) = {ga}
Follow(S) = {$}

Continue(0/1)? 1

Enter the element : A

Frist(A) = {ga}
Follow(A) = {b}

Continue(0/1)? 1

Enter element : C

First ~~Follo~~ (C) = {g}

Follow (C) = {df}

Continue (0/1)? 1

Enter element : E

First (E) = {h}

Follow (E) = {df}

continue (0/1)? 0

Result: First and Follow for the given expression has been performed successfully using C++ program.

S → aAB | bA | ε ⇒ S → aAB

S → bA

S → ε

A → aAb | ε

B → bB | ε

① first (S) = {a, b, ε}

first (A) {a, ε}

first (B) {b, ε}

first (a) = {a}

first (b) = {b}

follow (a) = {a}

follow (b) = {b}

follow (A) = {$, a, b}

follow (B) = {$, a, b}

follow (S) = {$}

Date: 23/2/23

Experiment - 6

Construction of Predictive Parsing Table

Aim: To construct Predictive Parsing Table using C program.

Code:

```
# include <string.h>
# include <conio.h>
Char a[10];
int top = -1, i;
void error(){
printf("Syntax Error");
}
void push(char k[]) //pushes characters on the stack
{ for(i=0; k[i]!='\0'; i++)
  {if (top<9)
     a[++top] = k[i];
  }
}
char Tos() //Returns Top of stack
  {
    return a[top];
}
void pop()
{
```

$S \rightarrow a \mid \uparrow \mid (T)$

$T \rightarrow T, S \mid S$

```
if (top >= 0)
   a[top--] = '\0';
}
void display () // Displays elements of stack
{ for (i=0; i <= top ; i++)
     printf ("%c" ; a[i]);
}
void display1 (char p[], int m) // Display present
                                   input string
{ int l;
 print ("\t");
 for(l=m; p[l] != '\0' ; l++)
    printf ("%c ", p[l]);
}
char * stack () {
    return a;
}
```

```
int main()
{ char ip[20], r[20], st, an;
  int ir, ic, j=0, k;
  char t[5][6][10]={"$", "$", "TH", "$", "TH", "$",
    "TH", "$", "e", "e", "$", "e", "$", "$", "FU",
    "$", "FU", "$", "e", "*FU", "e", "e", "$", "e",
    "(E)", "i", "$"};

  clrscr();
printf("Enter any string(Append with $)");
  gets(ip);
  printf("Stack\tInput\tOutput\n");
push("$E")
  display();
printf("\t%s\n", ip);
for(j=0; ip[j]!='\0';);
  { if(TOS()==an)
    { pop();
      display();
    display1(ip, j+1);
    printf("pop");
} j++;
```

```
an = ip[j];
st = Tos();
if (st == 'E') ir = 0;
else if (st == 'H') ir = 1;
else if (st == 'T') ir = 2;
else if (st == 'U') ir = 3;
else if (st == 'F') ir = 4;
else {
    error();
  break;
}
  if (an == '+') ic = 0;
  else if (an == '*') ic = 1;
  else if (an == '(') ic = 3;
  else if (an >= 'a' && an <= 'z') || (an >= 'A' && an <= 'Z')) {ic = 4; an = 'i';}
else if (an == '$') ic = 5;

strcpy(s, strrev(t[ir][ic]));
strrev(t[ir][ic]);
pop();
 push(r);
 if (Tos() == 'e')
   {pop();
   display();
```

```
display1(ip, j);
printf("\t%c -> %c\n", st, 230);
    }
    else
    {
     display();
    display1(ip, j);
   printf("\t%c -> %s\n", st, t[ir][ic]);
    }
    if(Tos() == '$' && an == '$'),
    break;
    if (Tos() == 'S'){
    error();
    break;
   }
   k = strcmp(stack(), "$");
  if(k == 0 && i == strlen(ip))
 printf("\n Given String is accepted");
  else
 printf("Give Str isn't accepted");
    return 0;
   }
```

Result: Construction of predictive parsing table was performed successfully.

Input/output

Input

$S \rightarrow a \mid \uparrow \mid (T)$

$T \rightarrow T, S \mid S$

① $S \rightarrow .a$
② $S \rightarrow \uparrow$
③ $S \rightarrow (T)$
④ $T \rightarrow .S\,T'$
⑤ $T' \rightarrow , S\,T'$
⑥ $T' \rightarrow \varepsilon$

first(a) = a
first(↑) = ↑
first(,) = ,
first(( ) = (
first( )) =)

First(S) = (a, ↑, ( )
First(T) = (a, ↑, ( )
First(T') = (, , ε)
Follow(S) = ($, , , ε]
Follow(T) = ($,))
Follow(T') = ($,), $, $)

Output:

	a	↑	,	(	)	$
S	S→a	S→↑		S→(T)		
T	T→S T'	T→S T'		T→S T'		
T'			T'→, S T'		T'→ε	T'→ε

Date
6/3/23

Experiment - 7

Shift Reduce Parser

Aim: To perform shift reduce Parser program in python language.

Code:

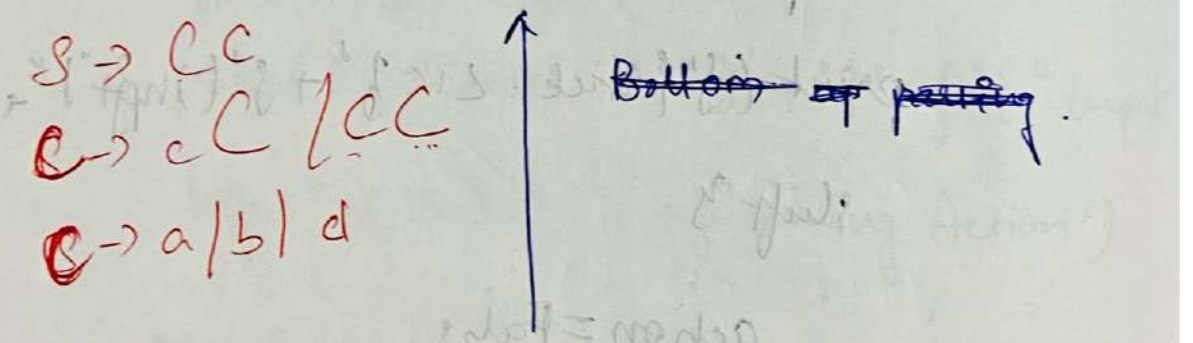
```
gram = {
    "E": ["E+E","E*E","i"]
}

starting terminal = "E"
inp = "i+i*i"

stack = "$"
print(f'{"Stack":<15}'+"|"+f"{"Input Buffer":<15}"+"|"+f'Parsing Action')
print(f'{"-":-<50}')

while True:
    action = True
    i = 0
while i < len(gram[starting terminal]):
```

```
if gram[starting_terminal][i] in stack:
    stack = stack.replace(gram[starting_terminal][i], starting_terminal)
print(f'{stack:<15}'+"|"+f'{inp:<15}'+"|"+
f'Reduce S->
    i = -1
    action False
    i+=1
if len(inp)>1:
    stack+=inp[0]
    inp=inp[1:]
    print(f'{stack:<15}'+f'{inp}'+"|"+f'
    Shift {}
    action=False
if inp=="$" and stack=="$"+starting_
                terminal):
    print(f'{stack:<15}'+"|"+f'{inp:
    +f'Accepted')
```

```
break:
if action:
    print(f'{stack:<15}' + "|" + f'{inp:<15}'
        + "|" + Rejected')
    break.
```

Result: Shift Reduce Parser using python has been succefully performed.

Input | output:



cd ab

Stack	Input String	Action
$	cdab$	Shift
$c	dab$	Shift
$cd	ab$	Shift
$cd	ab$	Reduce $\overset{C}{\rightarrow}$ d
$cC	ab$	Reduce C → cC
$C	ab$	Shift
$Ca	b$	Reduce C → a
$CC	b$	Reduce C → cC
$C	b$	Shift
$Cb	$	Reduce C → b
$CC	$	Reduce S → CC
$S	$	Accept

14/3/23

Experiment-8

Operator Precedence Parsing

Aim:- To implement operator precedence parsing for the grammar.

Code:-

```
import numpy as np
def stringcheck():
    a = list(input("Enter the operator"))
    a.append('$')
    print(a)
    l = list("abcdef---- z")
    o = list('/*%+-)')
    p = list('(/*%+-)')
    n = np.empty([len(a)+1, len(a)+1],
        dtype=str, order='F')
    for j in range(1, len(a)+1):
        n[0][j] = a[j-1]
        n[j][0] = a[j-1]
    for i in range(1, len(a)+1):
        for j in range(1, len(a)+1):
            if((n[i][0] in l) and (n[0][j] in l)):
                n[i][j] = '='
            elif((n[i][0] in l)):
                n[i][j] = '>'
```

```
else :
    u[i][j] = "<"

elif ((u[i][0] in 0) and u[0][j] in 1 ) :
    u[i][j] = "<"

elif ( u[i][0] == '$' and u[0][j] != "$") :
    u[i][j] = "<"

elif ( u[0][j] == '$' and u[i][0] != "$"):

else :
    break

print ( " The operator precedence Relational Table|---")

print (u)
i = list ( input( " Enter the string")
i = append ('$')
s = [none]* len(i)
q = 0
s.insert ( 1, '$')
x = [row (0) from row in u )
y = list [u(0)]
u = 0
while [s[0] != s[1]):
    if ( i[len[i]-2) in y)):
        break.
    elif([s[1] in x and [i[u] in y)):
        if ( u [x. index (s[q]))] [y. index(i(u)] == "<")
            q += 1
        s.insert ( q, i[u])
        u += 1
```

```
elif(n[x.index(s[2]).[y.index(i[n])]] ... ):
        s.pop(2)
        q-=1
elif((n[x.index(s[2])][y.index(i[n])]]=="." and
                            ((s[y]=="$") and
                            (i[n]=="$")):
            s[1]=s[0]
else:
    break
if (s[0]!=s[1])
    return false
else:
    return true
def grammarcheck(i):
    print("enter the"+str(i+1)+"th grammar"):
      b= list(input().split("->"))
      f= list("abcd---z")
      if(b[0]=="" or b[0] "" or b[0] in for
                len(b)==1):
            return false
      else
         b= pop(0)
         h= list(b[0])
         s= list("ABCD.....Z")
         o= list("(abc... z ^/*+-/)")
         sp =['!',@' --- ","  "]
         for in in range(0,len(b),2):
             if(b[i]==","):
                 g=false
             elif(h[i] in sp):
                 g=false
```

```
elif (b[i] in f):
        g = true
elif (b[len(b)-1] in 0):
        g = false
elif (i >= len(b)-1) and (b[i] in ( )):
        g = true
elif ((b[i] in s) and (b[i+1] in s)):
        g = false
        break.
else:
        g = false
        break.
if (g == True)
        return true.
else:
        return false
```

I/P :- E → E+T T → T*F F → (E)
E → T T → F F → i

L(E) = { (, *, +, i }
L(T) = { (, *, i }
L(F) = { (, i }
T(E) = {), +, $ }
T(T) = {), *, +, $ }
T(F) = {), *, +, $ }

```
c = int(input("Enter the number of LHS variable - )n"))
for i in range(c)
        if (grammarcheck(i)):
                t = true
        else:
                t = false
                break
if(t):
        print("Grammar is Accepted")
        if (stringcheck()):
                print("String is accepted")
        else:
                print("String is not accepted")
else:
        print("Grammar is not accepted").
```

17/3/22

Result :- We successfully implemented the operator precedence using python

Date
17/3/23

Experiment 9

LR(0) Parser

Aim: To implement LR(0) parser using C language

Code:

```
#include <iostream>
#<conio>.h
using namespace std;
char prod[20][20], listofvar[26] = "ABCDEF
GHIJKLMNOPQRSTUVWXYZ";
int novar = 1, i=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
```

```
{ if (clos[z][i].rhs[j]=='.' && clos[z][i]
  rh[j+1]=a)
  { clos[noitem][n].lhs = clos[z][i].lhs;
  strcpy(clos[noitem][n].rhs, clos[z][i].
                              rhs);
  char temp = clos[noitem][i];
  clos[noitem][n].rhs[j+1] = temp;
  n=n+1;
  }
}
for(i=0; i<n; i++)
  { void findclosure(int z, char a)
    { int n=0, i=0, j=0, k=0, l=z;
    for(i=0; i<cars[z]; i++)
      { for(k=0; k<novar; k++)
        { if(clos[noitem][i].rhs[j+1]==
          close[0][k].lhs)
          { strcpy(clos[noitem][n].rhs, clos
                                  [0][k].rhs);
          n=n+1;
```

```
for(i=0;i<novar;i++)
  { for(int j = strlen(clos[noitem][i].rhs)
      clos[noitem][i].lhs=g[i].lhs;
      if(strcmp(clos[noitem][i].rhs."ε")==
      0)
    else
    { clos[noitem][i].rhs,".");
  }
}
int main()
 {
    for(int x=0;x<1;x++)
      find closure(z, list[x]);
 }
  cout<<"\n THE SET OF ITEMS
  ARE \n\n";
   { for(int z=0; z<noitems; z++)
     cout<<"\n I"<<z<<"\n\n";
   for(j=0; j<arr[z]; j++)
   cout<< clos[z][j].lhs<<;
}}
```

Result: We succesfully implemented and verified LR(0) parser using C++.

Input / Output

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

$B \rightarrow S$

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

SET OF ITEMS

$B \rightarrow \cdot S$

$S \rightarrow \cdot AA$

$A \rightarrow \cdot aA$

$A \rightarrow \cdot b$

I_1

$B \rightarrow S \cdot$

I_2

$S \rightarrow A \cdot A$

$A \rightarrow a \cdot A$

$A \rightarrow \cdot b$

I_4 $A \rightarrow b \cdot$

I_5

$S \rightarrow AA \cdot$

I_6 $A \rightarrow aA \cdot$

Date:
27/8/23

Experiment - 10

Intermediate code generations - Postfix, Prefix

Aim: A program to implement intermediate code generation - Postfix, Prefix.

Code:

```
OPERATORS = set(['+', '-', '*', '/', '(', ')'])
PRI = {'+': 1, '-': 1, '*': 2, '/': 2}

def infix_to_postfix(formula):
    stack = []
    output = ''
    for ch in formula:
        if ch not in OPERATORS:
            output += ch
        elif ch == '(':
            stack.append('(')
        elif ch == ')':
            while stack and stack[-1] != '(':
                output += stack.pop()
            stack.pop()
        else:
            while stack:
                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.append(op+b+a)
    op_stack.append(ch)
while op_stack
    op = op_stack!
    op = op_stack.pop()
    a = exp_stack.pop()
    b = exp_stack.pop()
    exp_stack.append(op+b+a)
print(f'PREFIX: {exp_stack[-1]}')
return exp_stack[-1]
expres = input("Input the expression")
pre = infix_to_prefix(expres)
pos = infix_to_postfix(expres)
```

Result: We have successfully implemented & verified postfix & prefix using python.

Input / output :

$a + (b+c) \wedge z + y + x$

Postfix :

$a+bc+z\wedge y+x+$

Prefix

$x + y + (c+b) \wedge z + a$

$x+y+cb\wedge$

Sn	input	print
a		a
+	+	
(	+(	
b	+(	ab
+	+(+	ab.
c	+(+	abc
)	+	abc +
∧	+∧	abc+
z	+∧	abc+z
+	++	abc+z∧+
y	++	abc+z∧+y
+	++	abc+z∧+y+
x	+	abc+z∧+y+x

Date: 5/4/23

Experiment - 11

Quadruple, Triple & Indirect Triple

Aim: To implement the intermediate code to form quadruple, triple & indirect triple using C++.

Code:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
void small();
void dove(int i);
int p[5]={0,1,2,3,4},c=1; i,k,l,m,pi;
char sw[5]={'=','-','+','/','*'},
     j[20],q[5],b[5],ch[2];
void main()
{ printf("Enter the expression:");
  scanf("%s",j);
  small();
}
void dove(int i)
{ q[0]=b[0]='10';
```

```
if (!isdigit(j[i+2]) && !isdigit(j[i-2]))
{
  a[0]=j[i-1];
  b[0]=j[i+1];
}
if(isdigit(j[i+2])){
a[0]=j[i-1];
b[0]=j[i+1];
}
if(isdigit(j[i+2])){
  a[0]=j[i-1];
  b[0]=b+1;
  b[1]=j[i+2];
}
  if(isdigit(j[i-2]))
  {
    b[0] j[i+];
    a[0]=b+1;
  }
  if(isdigit(j[i+2]) && isdigit(j[i-2]))
{
  a[0]='t';
  b[0]='t';
  a[1]=j[i-2];
  b[1]=j[i+2];
```

```
}
if(j[i]==k+1)
printf("\t\t%d=%s*%s\n",c,a,b)
if(j[i]=='/')
printf("\t\t%d=%s-%s\n",c,a,b);
if(j[i]=='('-0)
{
sprintf(ch,"%d",c);
j[i]=ch[0];
c++;
small();
}
void small()
{
pi=0; l=0;
for(i=0; i<strlen(j); i++)
{ for(m=0; m<c; m++)
{if(pi<=p[m])
{ pi=p[m];
l=1;
k=i;
}
}
if(l==1)
done(k);
```

else
exit(0); }

Result - We have succefully performed triple, quatraple, indirect triple using C++ programing.

$a = (b+c)\wedge z + y + x$

$t_1 = b+c$

$t_2 = t_1 \wedge z$

$t_3 = t_2 + y$

$t_4 = t_3 + x$

$a = t_4$

Quadraple

	op	op1	op2	result
(1)	+	b	c	t_1
(2)	^	t_1	z	t_2
(3)	+	t_2	y	t_3
(4)	+	t_3	x	t_4
(5)	=	t_y	-	a

Triple

op	op1	op2
+	b	c
^	t_1	z
+	t_2	y
+	t_3	x
=	t_y	-

Indirect Triple

(16)	(1)
(17)	(2)
(18)	(3)
(19)	(4)

Date
11/09/23

Experiment 12

Simple Code Generation

Aim: To write the code for implementation of Simple code Generator.

Code:-

```
#include <bits/stdc++.h>
using namespace std;
typedef struct{
    char var[10];
    int alive;
}regist;
regist preg[10];
void substring(char exp[], int st, int
end){
  int i,j=0;
  char dup[110]="";
  for(i=st;i<end;i++)
  dup[j++]=exp[i];
  dup[j]='\0';
```

```
for (i=0; i<10; i++)
{ if preg[i].aline ==0)
{ strcpy( preg[i], varvar);
  break;
}
} return(i);
}  void getvar ( char exp[ ], char V[ ]){
    int i,j=0
    char var[10] = " ";
    if (isalpha(exp[i]))
        var [j++] = exp[i];
    else
      break;
    var [j] = "10";
    Strcpy(v, var);
  }
  int main( ){
   Char basic [10][10], var[10][10],
    fstr [10]; op;
  int i, j, k, reg, vc, flag=0;
  cout<< "Enter three Address code");
    cin .getline ( basic[i], 10);
```

```
break;
}
cout<<"The Equivalent Assembly code is:\n";
getvar(basic[j], var[vc++]);
strcpy(fstr, var[vc-1]);
getvar(basic[j], var[vc-1]);
if(preg[reg].alive==0){
cout<<"MOVR"<<reg<<","<<var[vc-1];
preg[reg].alive=1;
}
op=basic[j][strlen(var[vc-1])];
substring(basic[j]);
getvar(var[vc++]);
switch(op){
case'+': cout<<"\nAdd"; break;
case'-': cout<<"\nSub"; break;
case'*': cout<<"\nMul"; break;
case'/': cout<<"\nDiv"; break;
}
flag=1;
for(k=0; k<=reg.k++){
preg[k].alive=0;
}}
```

```
if (flag){
cout << "\nMOV" << fst << "R" << reg;
}strcpy ( preg[reg]. var, var[-3]);
}
cout <<"\nMOVR" << reg+1 << ", " << var[0];
    return 0;
}
```

Result: We have successfully performed the implementation of simple code generation in C language.

Input / Output :

$y = a + (b + c) \wedge z + x$

a = m
b =
c = l

$a = k + l$ → $t_1 = b + c$

$b = m + a$ → $t_2 = a + t_1$

$c = b * z$ → $t_3 = t_2 \wedge z$

$d = c + y$ → $t_4 = t_3 + y$

$e = d + x$ → $t_5 = t_4 + x$

$y = e$ → $y = t_5$

```
MOV  R0, k
Add  l, R0
MOV  a, R0
MOV  R1, m
Add  a, R1
MOV  b, R1
MOV  R2, b
Mul  z, R2
MOV  c, R2
MOV  R3, c
Add  y, R3
```

```
MOV  a, R3
MOV  R4, d
Add  x, R4
MOV  e, R4
MOV  R5, e  R0, R5
MOV  R6, f
```

Exp 13

$(a+5) * (a/d) \wedge (a+c)$

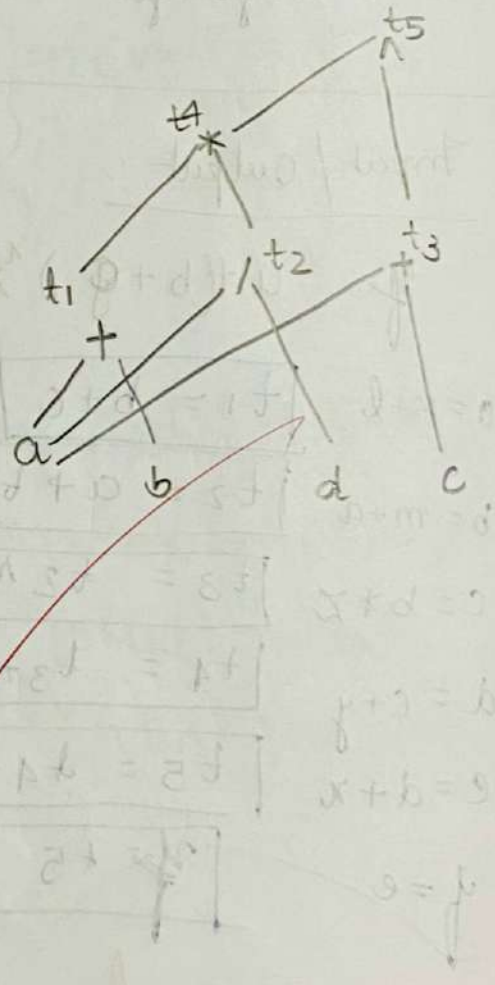
A $t_1 = a+b$

B $t_2 = a/d$

C $t_3 = a+c$

D $t_4 = t_1 * t_2$

E $t_5 = t_4 \wedge t_3$



O/p verified

AIA 19/9/2023

Date: 19/9/23

Experiment - 13

Code Optimization

Aim: To implement code optimization in python programming language.

Code:

```
import network as nx
from matplotlib import pyplot as plt
graph = nx.DiGraph()
graph.add_edges_from([("root","a"),
("a","b"),("a","e"),("b","c"),("b","d"),
("d","e")])
if nx.is_directed(graph):
    print('It is a DAG')
    plt.tight_layout()
plt.savefig("fout=PNG")
    plt.clf()
```

Result: We have successfully executed the code optimization code.

Date: 19/9/23

Experiment 11(a)

Stack implementation

Aim: To imp. stack in python.

Code:

```
stack = []

stack.append('a')
stack.append('b')
stack.append('c')
print('Initial stack')
print(stack)
print(stack.pop())
print(stack.pop())
print(stack.pop())
print('\n stack after elements are popped:')
print('stack')
```

Result: Memory allocation using stack was successfully implemented.

Output

Initial stack

['a', 'b', 'c']

Elements popped from stack

c
b
a

Stack after elements popped

[]

19/6/2025

Date: 19/9/23

Experiment - 15 (b)

Implementation of stack using Heap

Aim: To implement memory allocation using heap for stack.

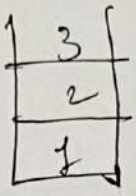
Code:

```
import heap
Class stack
  def __init (self);
    self.cut=0
    self. pq =[]
  def push (self,n)  self init = 1
  heap .heappush (self. pq,(-self.cut, n))
  def pop (self): if not self.pq:
      print ("Nothing to pop!!")
    self. cut  ==1
    return heapq . heap.pop(self.pq)[1]
    def top(self):
      return self pq[0][1]
    return not bool (self.pq)
S = stack()
S . push(1)
S . push(2)
S . push(3)
while not S. isempty(): print (S.top())
```

Result: To implement memory allocation using stack and heap.

Output:

1, 2, 3



Output 3 2 1

O/p verified
7/4/2023