INDEX

NAME: Pratyaksh Bhootra STD.: _____ SEC.: T2 ROLL NO.: 109 SUB.: Compiler Design (Lab)

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1.	18/1/23	Lexical Analyzer	1+2+2	b
2.	25/1/23	Regular Expression to NFA	1+2+2/5	b 5/5
3.	02/2/23	NFA to DFA	5+5/10	b 10/10
4.	02/2/23	Elimination of left factoring	5 +5	b
~~5~~	09/2/23	Elimination of left recursion		
5.	16/2/23	First-Follow Computation	2+3+5	b
6.	23/2/23	Construction of Predictive Parsing Table	2+3+4+5	b
7	6/3/23	Shift Reduce Parsing	2+5+8	b
8	14/3/23	Operator Precedence Parsing	5+5	b
9	17/3/23	LR(0) Parsing	10	17/3/2023
10.	27/3/23	Postfix Prefix	2+3+5	b
11	5/4/23	Triple, Quadruple, Indirect triple	2+3+5	b
12.	11/4/23	Simple Code Generation	2+3+5	b
13	19/4/23	Directed Acyclic Graph	2+5+3	b
14				
15	19/4/23	Stack & Heap	2+2+6	b

18/1/23

Experiment-1

Lexical Analyzer

Aim:- Implementation of lexical analyzer

Code:-

```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
bool isDelimiter(char ch)
{
 if (ch == ' ' || ch = '+' || ch == '-' || ch == '*' ||
     ch == '/' || ch == ',' || ch == ';' || ch = '<'
     || ch = '=' || ch = '>' )
    return(true);
  return(false);
}
bool isOperator(char ch)
{ if (ch == '+' || ch == '-' || ch == '*' || ch == '/'
     || ch == '>' || ch == '<' || ch == '=')
    return(true);
  return(false);
}
bool isSpecial(char ch)
 { if(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] ==
        || isDelimiter(str[0]) == true)
        return(false);
    return(true);
}
bool isKeyword(char* str)
{
    if (!strcmp(str, "if") || !strcmp(str, "else") ||
        !strcmp(str, "while") || !strcmp(str, "do") ||
        !strcmp(str, "break") || !strcmp(str, "int") ||
        !strcmp(str, "double") || !strcmp(str, "float") ||
        !strcmp(str, "return") || !strcmp(str, "char") ||
        !strcmp(str, "switch") || !strcmp(str, "unsigned") ||
        !strcmp(str, "void") || !strcmp(str, "static") ||
        !strcmp(str, "struct") || !strcmp(str, "goto"))
        return(true);
    return(false);
}
bool isInteger(char* str)
{
    int i, len = strlen(str);
    if (len == 0)
        return(false);
```

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

Void Parse (char* str)
{ int left = 0, right = 0;
  int len = strlen(str);
  while (right <= len && left <= right) {
     if (isDelimiter(str[right]) == false
       right++;
     if (isDelimiter(str[right]) == true && left == rig
       { if (isOperator(str[right]) == true)
           printf("'%c' IS AN OPERATOR\n", str[
       right++;
       left = right;
     }
        else if (isDelimiter(str[right]) == true &&
        left != right || (right == len && left != right
         char* subStr = subString(str, left, right-1)
```

```
if (isSpecial(subStr) == true)
    printf("'%s' IS A SPECIAL\n", subStr);
else if (isKeyword(subStr) == true)
    printf("'%s' IS A KEYWORD\n", subStr);
else if (isInteger(subStr) == true)
    printf("'%s' IS AN INTEGER\n", subStr);
else if (isRealNumber(subStr) == true)
    printf("'%s' IS A REAL NUMBER\n", subStr);
else if (validIdentifier(subStr) == true &&
        isDelimiter(str[right-1]) == false)
    printf("'%s' IS A VALID IDENTIFIER\n", subStr);
else if (validIdentifier(subStr) == false &&
        isDelimiter(str[right-1]) == false)
    printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);
left = right;
}
}
return;
}
int main() {
char str[100] = "float x = a + 1b;";
parse(str);
return(0);
}
```

Input :-

```
# include <stdio.h>
int main() {
    a = 4, b = 5
    int c;
    c = a+b;
}
print ("%d", c);
return 0;
}
```

Output :-

All tokens are :-

Lexime	Token
#	special symbol
include	identifier
<	operator
stdio.h	identifier
>	operator
int	keyword
main	identifier
a	identifier
=	operator
4	integer
b	identifier
=	operator
5	integer
int	keyword

Lexeme	Token
c	identifier
c	identifer
=	operator
a	identifes
+	operator
b	identifies
print	keyword
%	special
d	identifies
c	identifies
return	keyword
0	integer

25/1/23

Experiment 2

Regular Expression to NFA

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

Input: ab

Output:

Transition Table

Input :- ~~aab~~ $a^* b^*$

Output :-

Transition Table

Current	Input	Next state
q[1]	e	q[2], q[4]
q[2]	a	q[3]
q[3]	e	q[4], q[2]
q[4]	e	q[5], q[7]
q[5]	b	q[6]
q[6]	e	q[7], q[5]

```
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+2)
        printf("%d  --%c-->  %d\n",
                ret[i], ret[i+1], ret[i+2]);
    printf("\n");
    getch();
}
```

02/02/23

Experiment - 3

Conversion of NFA to DFA

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

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_states_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[key_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_list[0]] = {}
    for - 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]][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[i]] = 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_states.append(x)
            break
print("\n Final states of the DFA are: ", dfa_final_states)
```

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

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 DFA using python successfully

02/02/23

Experiment-4(a)

Elimination of Left Factoring

Aim:- To write a program to perform left factoring

Code:-

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

```
        k++;
        pos = i+1;
    }
}
for (i=pos, j=0; part1[i] != '\0'; i++, j++){
    newGram[j] = part1[i];
}
newGram[j++] = '|';
for (i=pos; part2[i] != '\0'; i++, j++){
    newGram[j] = part2[i];
}
modifiedGram[b] = 'A';
modifiedGram[++c] = '\0';
newGram[j] = '\0';
printf("\n Grammar without Left Factoring : :\n");
printf("\n A -> %s", modifiedGram);
printf("\n A' -> %s\n", newGram);
```

Input:-

Enter Production :-

A → bSSaaS/bSSaSb/

Output:-

Grammar without Left factoring : :

A → bSSaA'

A' → aS/Sb/bSb/a

Results:- Left factoring of the given expression was performed successfully using C

09/02/23

Experiment - 4(6)

Elimination of 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 Production as E->E/A : ");
    scanf("%s", pro);
    non_terminal = pro[0];
    if (non_terminal = pro[index])
    {
        for (i = ++index, j = 0; pro[i] != '|'; i++
            & alpha[j] = pro[i];
            if (pro[i+1] == 0) {
                printf("This Grammar can't
                    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
    printf("This Grammar CAN'T be REDUCED.\n");
}
else
    printf("\n This Grammar is not LEFT RECURSIVE.\n")
```

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

E → E+T | T

Grammar without LEFT recursion:-

E → TE'

E' → +TE' / #.

Result:- Elimination of left recursion was performed successfully using above code in C.

16/02/23

Experiment-5

First and follow

Aim:- To implement the first and follow using C

Code:-

```
#include<stdio.h>
#include<math.h>
#include<string.h>
#include<ctype.h>
#include<stdlib.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(){
int i,z;
char c, ch;
printf("Enter the no. of production:\n");
scanf("%d",&n);
printf("Enter the production:\n");
for(i=0; i<n; i++)
scanf("%c",&a[i]);
scanf("%s %c", a[i], &ch);
do{
  m=0;
printf("Enter the elements whose first & follow is to
        be found:");
```

Algorithm :-

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

Sample Input Output :-

Enter the No. of Productions : 5

Enter the Production

S : AbCd

A = Cf

A : a

C = g ε

t = h

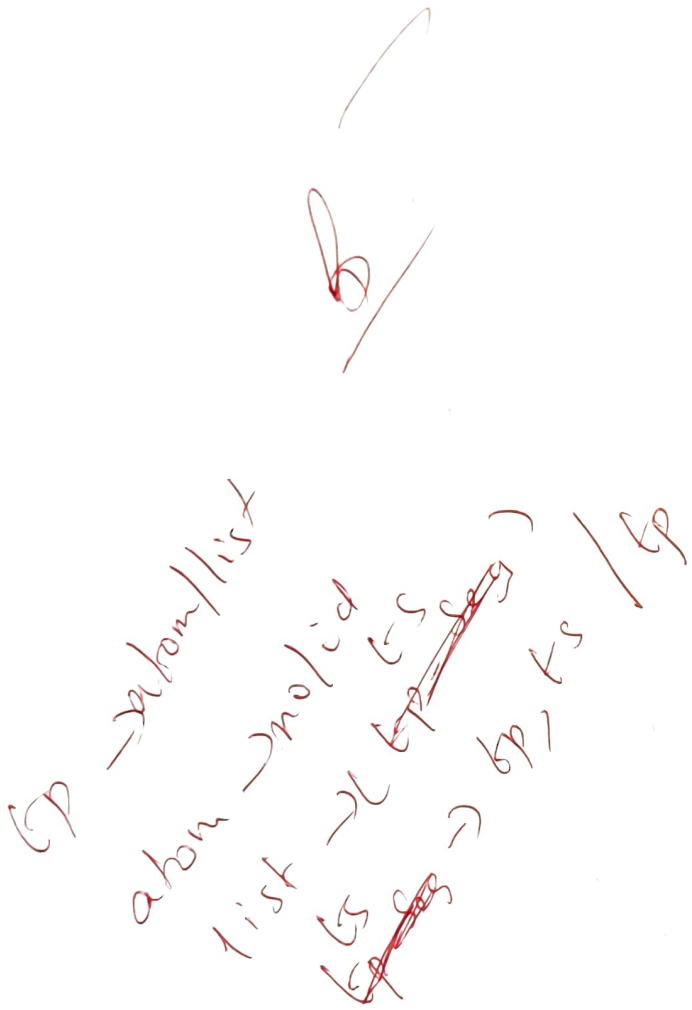
Enter the element whose first & follow you need to be found : S

first (S) = {ga}

follow (S) = {$}

```
    if(a[k][0]==c)
    {
      if(a[k][2]=='$')
        follow(a[k][0]);
      else if(islower(a[k][2]))
        f[m++]=a[k][2];
      else first(a[k][2]);
    }
  }
}
void follow(char c)
{
  if(a[0][0]==c)
    f[m++]='$';
  for(i=0;i<n;i++)
  {
    for(j=2;j<strlen(a[i]);j++)
    {
      if(a[i][j]==c)
      {
        if(a[i][j+1]!='\0')
          first(a[i][j+1]);
        if(a[i][j+1]=='\0' && c!=a[i][0])
          follow(a[i][0]);
      }
    }
  }
}
```

Result :- Thus implementation of first and follow computation was successfully verified.



tp → atom / list

atom → no / id

list → (ts)

ts → tp, ts / tp.

tp → A

atom → B

list → C

ts → D

ts → tp, ts / tp

D → AD'

D' → D

6

① $A \rightarrow B : | C$

② $B \rightarrow no | id$

③ $C \rightarrow (D)$

④ $D \rightarrow AD'$

⑤ $D' \rightarrow D$.

$S \rightarrow aAB$

$S \rightarrow bA$

$A \rightarrow aAb | \varepsilon$

$B \rightarrow bB | \varepsilon$

first { a }

first { b }

first(S) = a, b

first(A) = { a, b }

follow(S) = { $ }

follow(A) = { b.

first(no) = no.

first(id) = id

first(C) = (

first( ) =)

first(,) = ,

first(A) = { no, (}

first(B) = { no, id }

first(C) = { (}

first(D) = { no, id }

first(D') = { no, id }

follow(A) = { no, id }

follow(B) = { no, id }

follow(C) = { $ }

follow(D) = { no, id }

follow(D') = { no, id }

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[])
{ 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()
{
  if(top>=0)
    a[top--]='\0';
}
```

```
Void display ()
{ for (i=0; i<=top ; i++)
   printf ("%c", a[i]);
}
void display1 (char p[], int m)  // display present input
                                    string
{ int l;
   printf ("\t");
   for (l=m; p[l]!='\0'; l++)
       printf ("%c", p[l]);
   char *stack() {
       return a;
   }
int main() {
   char ip[20], a[20], st, an;
   int  iv, ic, j=0, t;
   char t[5][6][10] = { "$", "$", "TH", "$", "TH", "TH", "$",
        "Th", "$", "e", "e", "$", "e", "$", "$", "Fu",
        "$", "Fu", "$", "e", "*Fu", "e", "e", "$", "e",
        "(k)", "i", "$" };
   clrscr();
   printf("Enter any string (Append with $)");
   gets(ip);
   Printf("Stack | Input | Output |\n");
   printf("\t")
   display();
   Printf("\t %s|\n", ip);
   for (j=0; ip[j]!='\0';);
```

```
display1(ip, j);
printf("\t %c → %c\n", st, 238);
} else
  { display();
    display1(ip, j);
  printf("\t %c → %s\n", st, t[ix][ic]);
} if(Tos() == '$' && an == '$');
break;
  if(Tos() == '$') {
     error();
  break;
}
k = strcmp(stack(), "$");
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:-

X → a
Y → Y
Z → d
Z → X Y Z
Y → c

Y → ε

norecursive / NO factoring

first(a) = a
first(d) = d
first(c) = c

first(X) = {a, c, ε}
first(Y) = {c, ε}
first(Z) = {a, c, ε, d}

follow(X) = {$, c}
follow(Y) = {$, c, a, d}
follow(Z) = {$, d}

	a	d	c	$
Y	①		①	
Z	②	③	②	
X	Y→ε	Y→ε	③	Y→ε

6/3/23

Experiment-4

Shift Reducing Parsing

Aim:- To perform shift reducing parsing in the string.

Code:-

```
#include <bits/stdc++.h>
using namespace std;
int z=0, i=0, j=0, c=0;
char a[16], ac[20], stk[15], act[10];
void check()
{
   strcpy(ac, "Reduce to E -> ");
   for (z=0; z<c; z++)
   {
      if (stk[z]=='4')
      {
         printf("%s4", ac);
         stk[z]='E';
         stk[z+1]='\0';
         printf("\n$ %s \t %s$\t", stk, a);
      }
   }
   for(z=0; z<c-2; z++)
   {
      if(stk[z]=='2'&&stk[z+1]=='E'&&stk[z+2]=='2')
```

Input/output :-

$S \rightarrow a \mid \hat{} \mid (T)$

$T \rightarrow T, S \mid S$

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

first (a) = a
first ($\hat{}$) = $\hat{}$
first (,) = ,
first (() = (
first ()) =)

first (S) = (a, $\hat{}$, ()
first (T) = (a, $\hat{}$, ()
First (T') = (, ε)
follow (S) = ($, , , ε)
follow (T) = ($,))
follow (T') = ($,), $, $)

```
        {
            printf("%s2E2",ac);
            stk[z]='E';
            stk[z+1]='10';
            stk[z+2]='10';
            printf("\n$%s\t%s$\t",stk,a);
            i=i-2;
        }
    }

    for(z=0; z<c-2; z++)
    {
        if(stk[z]=='3' && stk[z+1]=='E' &&
                stk[z+2]=='3')
        {
            printf("%s3E3",ac);
            stk[z]='E';
            stk[z+1]='10';
            stk[z+2]='10';
            printf("\n$%s\t%s$\t",stk,a);
            i=i-2;
        }
    }
    return 0;
}
```

```
int main()
{
  printf("Grammar is -\n E -> 2E2 \n E -> 3E3
                      \n E -> 4\n");
  strcpy(a,'32423');
  c= strlen(a);
  strcpy(act,"SHIFT");
  printf("\n stack \t input \t action");
  printf("\n $\t %s $\t",a);

 for(i=0;j<c;i++,j++)
 {
   printf("%s", act);
    stk[i] = a[j];
   stk[i+1] = '\0';
   a[j]= ' ';
   printf("\n $ %s \t %s$ \t", stk, a);
   check();
 }
check();
 if(stk[0]=='E' && stk[1]=='\0')
     printf("Acceptance\n");
 else
     printf("Error\n");
}
```

Result :- We successfully implemented shift parsing reduce using C++.

Input / output :-

E. $E \to E + E$

$E \to E - E$

$E \to a | b | id | \epsilon$

c/p a - b + id.

stack	input	Action
$	a - b + id $	shift
$ a	- b + id $	Reduce $E \to a$
$ E	- b + id $	shift.
$ E -	b + id $	shift
$ E - b	+ id $	shift
$ E - b +	id $	shift
$ E - b + id	$	Reduce $E \to id$
$ E - b + E	$	Reduce $E \to b$
$ E - E + E	$	Reduce $E \to E+E$
$ E - E	$	Reduce $E \to E-E$
$ E	$	Accepted

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 operators used in the
                    grammar"))
    a.append('$')
    print(a)
    l = list("abcdefghi----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] = ">"
```

```
        h += 1
    elif (n[n.index(s[z])][y.index(i[h])] == ">" ):
        s.pop(z)
        q -= 1
    elif((n[n.index(s[z])][y.index(i[h])]=='' and
                                    ((s[z]=='$') and
            s[1] = s[0]                     (i[h]=='$'))):
        else:
            break
    if (s[0] != s[1]):
        return false
    else:
        return true

def grammar check(i):
    print("Enter the", str(i+1)+" th grammar "):
    b = list(input().split("->"))
    f = list("ab---z")
    if(b[0]=="' or b[0]==" or b[0] in for len(b)==1):
        return false
    else
        b = pop(0)
        b = list(b[0])
        s = list("AR---Z")
        o = list("(ab---z"+"/*+-1)*").
        sp = ['!','@'. . . . . ' ']
        for i in range(0, len(b), 2):
            if(b[i]=="*"):
                g = false
            elif(b[i] in sp):
                g = false
                break.
            elif(b[i] in f):
                g = true
```

R

S -> *R

L -> L->S

R -> R+L

```
elif (b[len(b)-1] in O):
        g = false
elif (i == len(b)-1 and (b[i] in s)):
        g = True
elif (b[i] in s) and (b[i+1] in O)):
        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.
c = int(input("enter the number of LHS variable
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").
```

Result: We successfully implemented the operator precedence using python

---

Viva :-

$S \rightarrow S * R$

$L \rightarrow L, S$

$R \rightarrow R + L$

Leading(S) = { *, Leading(R) } = { *, + }

Leading(L) = { , , Leading(S) } = { Leading(L), , } = { , , }

Leading(R) = { +, Leading(L) }

{ Leading(R), + } = { + }

Trailing (S) = {, * ∪ Trailing (R)} = {*, , .+}

Trailing (L) = {, ∪ Trailing (S)} = {, , *, +}

Trailing (R) = { + ∪ Trailing (L)}

= = {+, , *}

17/03/22

Experiment-9

LR(0)

Computation of LR(0) item

Aim:- To implement the LR(0) / simple LR parsing in C.

Code:-

```
#include <iostream>
#include <conio.h>
#include <string.h>
using namespace std;
char prod[20][20], listofvar[26] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
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<a[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].lh
            strcpy(clos[noitem][n].rhs, clos[z][i].rh
            char temp = clos[noitem][n].rhs[j];
            clos[noitem][n].rhs[j]= clos[noitem][n].
                                    rhs[j+1];
            clos[noitem][n].rhs[j+1]=temp;
            n=n+1;
        }
    }
}
for(i=0; i<n; i++)
{
    for(j=0; j<strlen(clos[noitem][i].rhs); j++)
    {
        if(clos[noitem][i].rhs[j]=='/' && isvariable
                (clos[noitem][i].rhs[j+1])>0)
        {
            for(k=0; k<novar; k++)
            {
                if(clos[noitem][i].rhs[j+1]==
                    closs[0][k].lh
                {
                    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(e==n)
      { clos[noitem][n].lhs = clos[0][k].lhs;
        strcpy(clos[noitem][n].rhs,clos[0][k].rhs);
          n=n+1;
        }
      }
    }
  }
}
}
arr[noitem]=n;
  int flag=0;
  for(i=0;i<noitem;i++)
    { if(arr[i]==n)
        { for(j=0;j<arr;j++)
            { int c=0;
              for(k=0;k<arr[i];k++)
                if(clos[noitem][k].lhs && strcmp
(clos[noitem][k].rhs, clos[i][k].rhs)==0)
                    c=c+1;
              if(c==arr[i])
                  flag=1;
                  goto exit;
}}}}
  exit;
  if(flag==0)
        arr[noitem]++]=n;
}
int main()
 {
```

```
clos[noitem][i].lhs = g[i].lhs;
strcpy(clos[noitem][i].rhs, g[i].rhs);
if(strcmp(clos[noitem][i].rhs, "ε")==0)
        strcpy(clos[noitem][i].rhs, ".");
else
{
    for(int j=strlen(clos[noitem][i].rhs)+1;j>=0;j--
        clos[noitem][i].rhs[j]=clos[noitem][i].rhs[j-1];
    clos[noitem][i].rhs[0]='.';
}}
arr[noitem++]=novar;
for(int z=0;z<noitem;z++)
{   char list[10];
    int l=0;
    for(j=0;j<arr[z];j++)
    {  for(k=0;k<(strlen(clos[I][j].rhs)-1;k++)
       {  if(clos[z][j].rhs[k]=='.')
          {  for(m=0;m<l;m++)
                if(list[m]==clos[z][j].rhs[k+1])
                    break;
             if(m==l)
                list[l++]=clos[z][j].rhs[k+1]
       }}
    }
    for(int x=0;x<l;x++)
        findclosure(z,list[x]);
}
```

```
cout << "\n The set of items are \n\n";
for (int i=0; i < no item ; i++)
{
    cout << "\n  I" << i << "\n\n";
    for (j=0; j< arr [i]; j++)
        cout << clos [i] [j]. lhs << "->" << clos [i]
    rhs << "\n";
}
}
```

17/3/2023

Result :- We successfully implemented and verified LR(0) using C++.

Input :-

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b.$

5/4/23

Experiment-10

Intermediate Code Generation - Postfix, Prefix

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

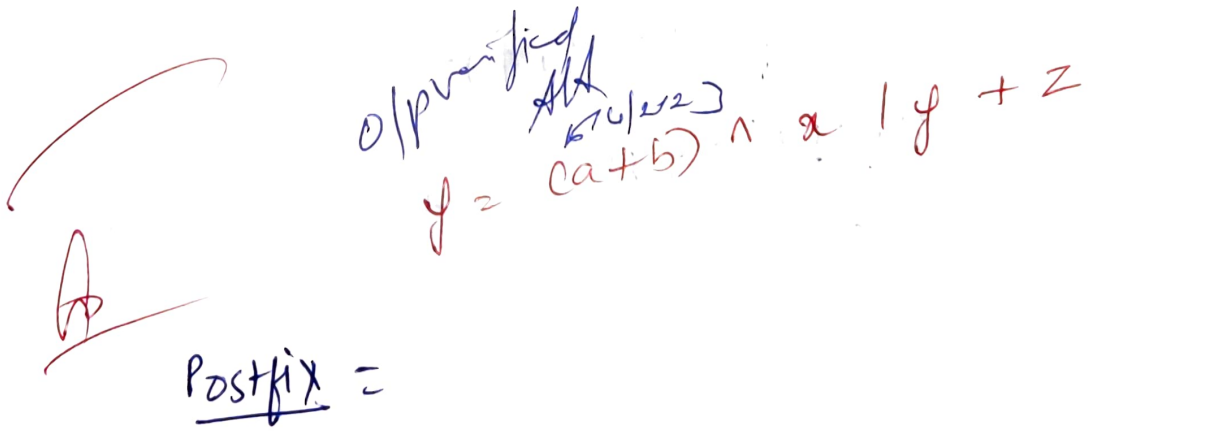
Code:-

```
OPERATORS = set(['+', '-', '*', '/', '(', ')'])

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

def infix-to-postfix (formula):
    stack[0] = []
    output = ''
    for ch in formula:
        if 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 == 'C':
            op = stack.append(ch)
        elif ch == 'D':
            while op stack[-1] != 'C':
                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] != 'C' and
                PRI[ch]C = [op_stack[-1]].
```

```
        op = op_stack.pop()
        a = exp_stack.pop()
        b = exp_stack.pop()
        exp_stack.append(op+b+a)
    while op_stack:
        op = op_stack.pop()
        a = exp_stack.pop()
        b = exp_stack.pop()
        exp_stack.append(op+b+a
    print(f'PREFIX : {exp_stack[-1]}')
    return exp_stack[-1]
expres = input("INPUT THE EXPRESSION: ")
pre = infix_to_prefix(expres)
pos = infix_to_postfix(expres)
```

Result:- We successfully implemented & verified postfix & prefix using python



5/4/23

Experiment-11

Implement- Quadruple, Triple & Indirect Triple

Aim:- To implement quadruple, triple and ~~Triple~~ Indirect-triple.

Code:-

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
void small();
void done(int i);
int p[5] = {0, 1, 2, 3, 4}, c=1, i, k, l, m, pi;
char sw[5] = {'=', '-', '+', '/', '*'}, j[20], c[5],
                                        b[5], a[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(!digit(j[i+2]) && !isdigit(j[i-2]))
{ a[0] = j[i-1];
```

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

```
for (i=0; i<strlen(j); j++)
{
    for (m=0; m<5; m++)
    if (j[i]==sw[m])
      if (pi<p=p[m])

    {
      pi=p[m];
      k=1;
       k=i;
    }
}
  if (k==1)
  down(k);
else
  exit(0); }
```

Result:- We succesfully implemented the concep
quadraples, triple and indirect-triple using c

input/output :-

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

$t_1 = a+b$

$t_2 = t_1 \wedge x$

$t_3 = t_2 / y$

$t_4 = t_3 + z$

$y = t_4$

2 oper. 2 L
1 op.
1.1 =

i- Quadruple

#	op	op1	op2	Result
(1)	+	a	b	t_1
(2)	^	t_1	x	t_2
(3)	/	t_2	y	t_3
(4)	+	t_3	z	t_4
(5)	=	t_4		y

o/p verified
6/4/2023

Triple :-

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

Indirect

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

11/04/25

Experiment-12

Simple Code Generation

Aim:- To implement the simple code generation.

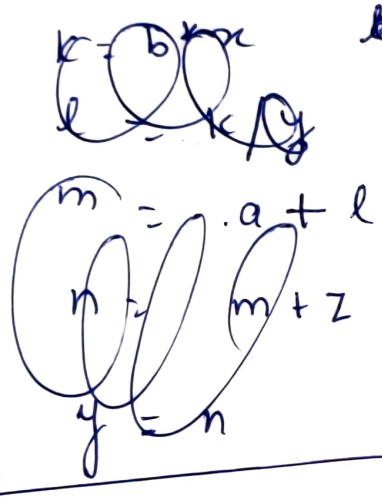
Code:-

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

input = a+b*x/y+z

$y = a + b * x / y + z$

three address code ;—



$k = x / y$

$l = b * k$

$m = a + l$

$n = m + z$

$y = n$

```
MOV  R1, b
MUL  c; R1
STO  c, R1
MOV  R2, k
DIV  y, R2
STO  R2, l
MOV  R3, a
ADD  l, R3
STO  R3, m
MOV  R4, m
ADD  z, R4
STO  n, R5
MOV  n, y
```

```
MOV  R0, x
Div  y, R0
MOV  k, R0
MOV  R1, b
MUL  k, R1
MOV  l, R1
MOV  R2, a
Add  l, R2
MOV  m, R2
MOV  R3, m
Add  z, R3
MOV  n, R3
MOV  R4, n
MOV  R5, y
```

```
void getvar (char exp[], char v[]) {
    int i, j = 0;
    char var[10] = "";
    for (i = 0; exp[i] != '\0'; i++)
        if (isalpha(exp[i]))
            var[j++] = exp[i];
        else
            break;
    var[j] = '\0';
    strcpy(v, var);
}
int main() {
    char basic[10][10], var[10][10], fstr[10], op;
    int i, j, k, reg, vc, flag = 0;
}
cout << "\n Enter the Three Address Code: \n";
    for (i = 0;; i++) {
        cin.getline(basic[i], 10);
        if (strcmp(basic[i], "exit") == 0)
            break;
}
cout << "\n The Equivalent Assembly Code is: \n";
for (j = 0; j < i; j++) {
    vc = 0;
    getvar(basic[j], var[vc++]);
    strcpy(fstr, var[vc-1]);
    substring(basic[j], strlen(var[vc-1])+1, strlen(basic[j]));
    getvar(basic[j], var[vc++]);
    reg = getregister(var[vc-1]);
    if (greg[reg].alive == 0) {
```

```
cout<<"\n MOV R"<<reg<<", "<<var[vc-1];
preg[reg].alive=1;
}
op=basic[j][strlen(var[vc-1])];
getvar(basic[j], var[vc++]);
switch(op){
    case '+': cout<<"\n Add"; break;
    case '-': cout<<"\n Sub"; break;
    case '/': cout<<"\n Div"; break;
    case '*': cout<<"\n Mul"; break;
}
flag=1;
for(k=0; k<=reg; k++){
    if(strcmp(preg[k].var, var[vc-1])==0){
        cout<<"R"<<k<<", R"<<reg;
        preg[k].alive=0;
        flag=0;
        break;
    }}
if(flag){
    cout<<"R"<<reg<<", "<<var[vc-1];
    cout<<"\n MOV "<<str<<", R"<<reg;}
strcpy(preg[reg].var, var[-3]);
}
cout<<"\n MOV R"<<reg+1<<", "<<var[0];
return 0;
}
```

Result:- We successfully implemented the concept of simple code generation

19/04/23

Experiment-13

Implementation of Directed Acyclic Graph (DAG)

Aim:- To implement the directed acyclic graph using c++.

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

```
DAG* nn = new DAG(tempLabel, tempData, leftPtr, rightPtr);

Label DAG.Node.insert(make_pair(templabel, nn));

}
}
cout << "Label ptr leftPtr rightPtr" << endl;
for (int i=0; i<n; i++)
    DAG* x = labelDAG.Node
    cout << H[i][0] << "  " << x->data <<
    "  ";

    if (x->left->label=='-') cout << x->left->data;
    else cout << x->left->label;
    cout << "  ";
    if (x->right->label=='-') cout << x->left->rig
    else cout << x->right->label;
    cout << endl;
}
return 0;
}
```

Result :- We successfully implemented DAG using C++.

BODMAS

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

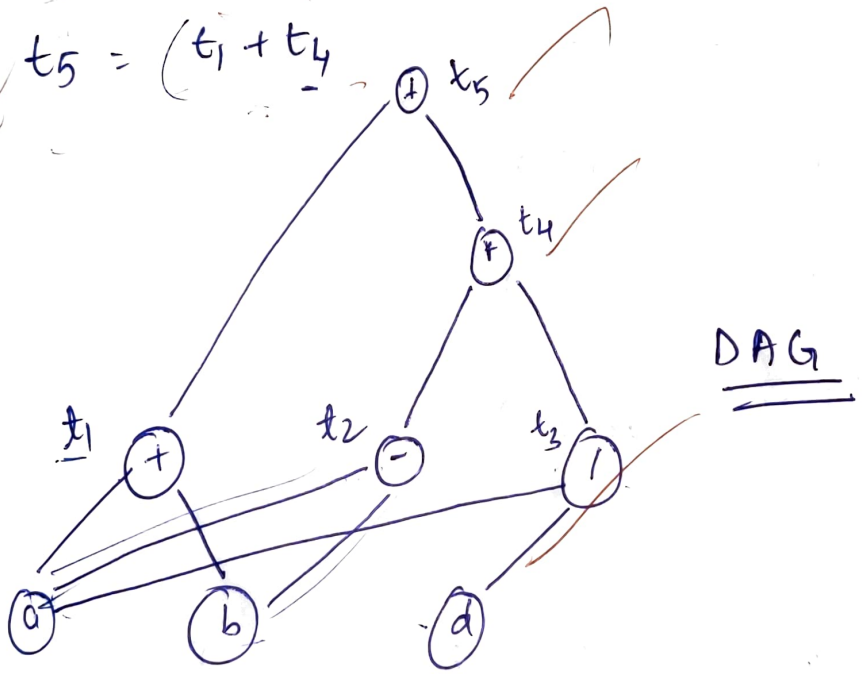
a $t_1 = a+b$

b $t_2 = a-b$

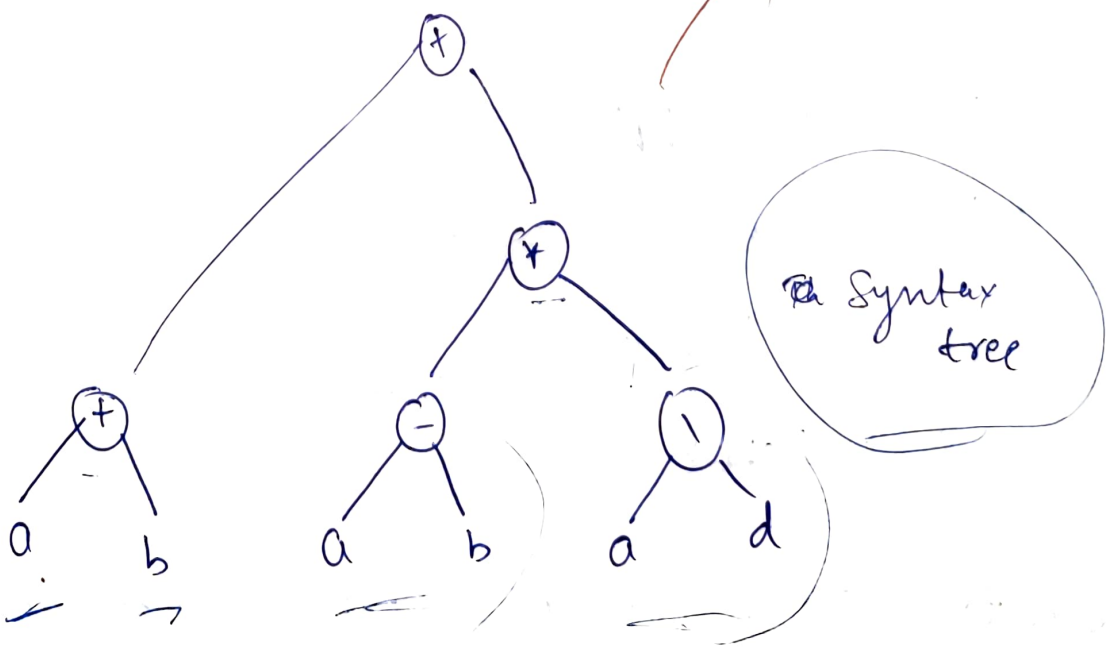
c $t_3 = a/d$

d $t_4 = t_2 * t_3$

$t_5 = (t_1 + t_4$



DAG



Syntax tree

19/04/23

Experiment - 13 (a)

Implementation of stack

Aim:- To implement the concept of stack in python.

Code:-

```
stack = []
stack.append('a')
stack.append('b')
stack.append('c')
print('Initial stack')
print(stack)

print('\n elements popped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())

print('\n Stack after elements are popped')
print(stack)
```

Result:- We successfully implemented the concept of stack

Input/Output :-

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

Elements popped from stack :

c
b
a

Stack are popped.

19/04/23

Experiment 15 (b)

Implementation of Stack using Heaps

Aim:- To implement stack using heaps.

Code:-

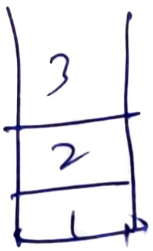
```
import heapq
class Grex
    def __init__(self):
        self.cnt = 0
        self.pq = []
    def push(self, n)
        self.cnt += 1
        heapq.heappush(self.pq, (-self.cnt, n))
    def pop(self):
        if not self.pq
            print("Nothing to pop!!!")
        self.cnt -= 1
        return heapq.heappop(self.pq)[1]
    def top(self):
        return self.pq[0][1]
    def isEmpty(self):
        return not bool(self.pq)
```

```
s = stack()
s.push(1)
s.push(2)
s.push(3)
while not s.isEmpty():
    print(s.top())
    s.pop()
```

Result: We successfully implemented the stack using heap.

Input/output :-

1, 2, 3



Output :-

3 2 1