Date

EXPERIMENT - 13

IMPLEMENTATION OF DIRECTED - ACYCLIC GRAPH

AIM: To compute a program for running the implementation of DAG.

ALGORITHM:

1. Interior nodes are labeled by operation symbol.
2. Nodes are given sequence of identifier lables to store the computed value.
3. If y operand is undefined then create node(y).
4. If z operand is undefined then for case(i) create node (z).
5. for case (i) create node (op) whose right child is node(z). and left child is node is(y).
6. For case (ii) check whether there is node op with one child node.
7. For node (y) delete x from the list of identifier.

PROGRAM:

```
OPERATORS = set(['+','-','*','/','(',')',])
 PRI = {'+':1,'-':1,'*:2,'/':2}
def infix_to_postfix (formula):
    output =''
  for ch in formula:
      if ch not in operator
          output +=ch
    elif ch = = '(':
        stack . append ['(']
     elif   ch = = ')':
  while stack and stack [-1] ! = '(':
            output + = stack . pop ()
              stack . pop () # opp '('
       else:
         while stack and stalk [-1]! = '('
           and  pri [ch] < =
           PRI ( stack [-1]]:
     Output += stak - pop ()
       stack . append (ch)
           # left owes
    while stack!
      output  += stack . pop()
  printf ( f'portfix : { output }'}
      oretmun  output
   ## Infix --> Prefix ##
    def  infix _to- prefix ( formula)
  op - stack = [ ]
       exp - stack [ ]
           for  ch  in  formulae:
  if not ch in  operator:
           exp - stack . append (ch)
             elif ch = = '('
         op - stuck . append (ch)
           elif ch = = ')':
  while op- stack .[-1]! = '(':
      if not  ch  in  operator
              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_stuck . append (op+b+a)
      op - stack . append ( ch)
 # left over
    while op - stack :
   op = op - stack . pop()
        a = exp - stack . pop ()
        b = exp - stuck . pop ()
exp_ stack $. append (op + b+a)
print ( f' prefix : { exp_stack. [-1] }')
.return   exp - stack [-1]
## three address ~~ode~~ code generation ###
 def generate 3AC [pos);
 printf ('# three address code generator #)
   exp - stack = []
          t = 1
  for i in pos:
       if i not in operator:
           exp - stuck . append (i)
       else :
printf (f'{t}: {exp - [-2]} {i}
      { exp - stuck [-1] }')
exp - stack = exp. stack [:-2]
   exp - stack. append ( f'(f {t}')
            t +=1
  express = input (" Input the expression:"
 pre = infix _ to _ postfix ( express)
     pos = infix _ to _ postfix ( express)
       generate 3AC (pos)
```

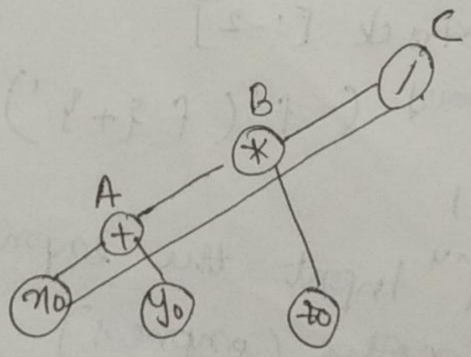
Sample Output :

$A = x + y$

$B = A * z$

$C = B / x$

lable	ptr	LeftPtr	rightPtr
A	+	x	y
B	*	A	z
C	/	B	x



DAG

```
def Quadrapule (pos):
                stack = []
                  op = []
                    x = 1
for i  in pos:
  for i not   in operator:
                stack append (i)
               elfi i == '_':
             opd = stack.pop()
{ 2: ^49}/ {3: 4s}".format (i, op2, "(-)", op1))
else:   OP1 = stack.pop()
            OP 2 = stack.pop()
( 2: ^4 8)". format (i, op1, op2))
          . else:
       OP1 = stack.pop()
            if   stack != []:
            op2 = stack.pop()
  print.("{0: ^4s}|{1: ^4s})
  print (" The triple for given expression")
print ("op| ArG1 /ArG2")
      triple ( pos)
```

RESULT: the program was sucessfully compiled and run.