EXP 12

A SIMPLE CODE GENERATOR , IMPLEMENTATION OF DAG

AIM: Implementation of DAG

ALGORITHM:

1. The leaves of a graph are labeled by a unique identifier and that identifier can be variable names or constants.
2. Interior nodes of the graph are labeled by an operator symbol.
3. Nodes are also given a sequence of identifiers for labels to store the computed value.
4. If z operand is undefined then for case(i) create node(z).
6. For case(i), create node(OP) whose right child is node(z) and left chIf y operand is undefined then create node(y).
5. ild is node(y).
7. For case(ii), check whether there is node(OP) with one child node(y).
8. For case(iii), node n will be node(y).
9. 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:

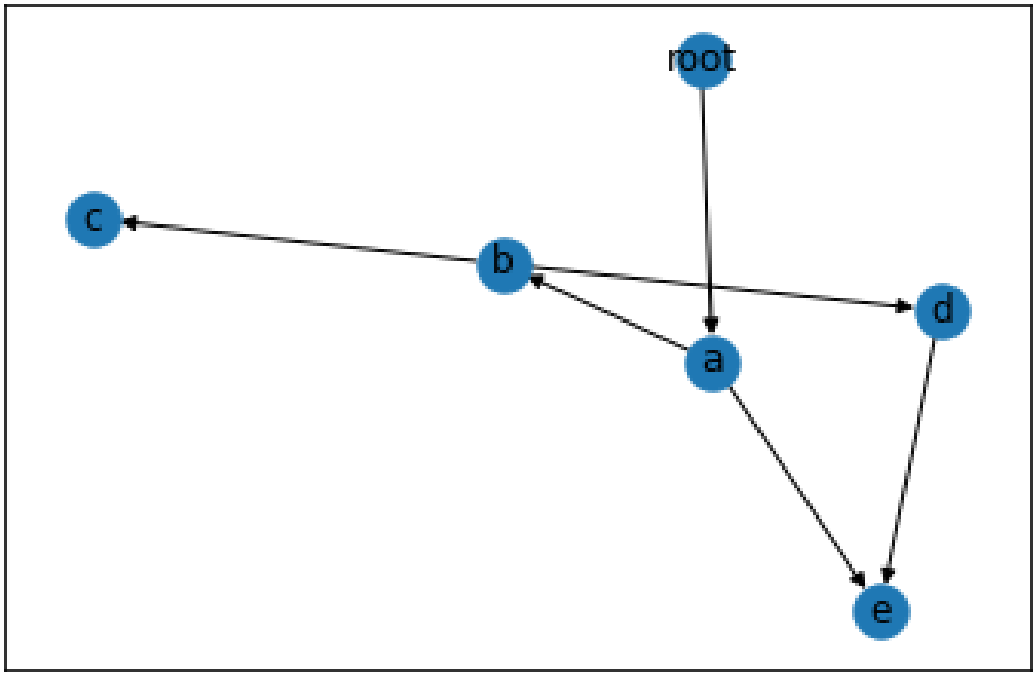
```
OPERATORS = set(['+', '-', '*', '/', '(', ')'])
PRI = {'+':1, '-':1, '*':2, '/':2}

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_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() # 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)
```

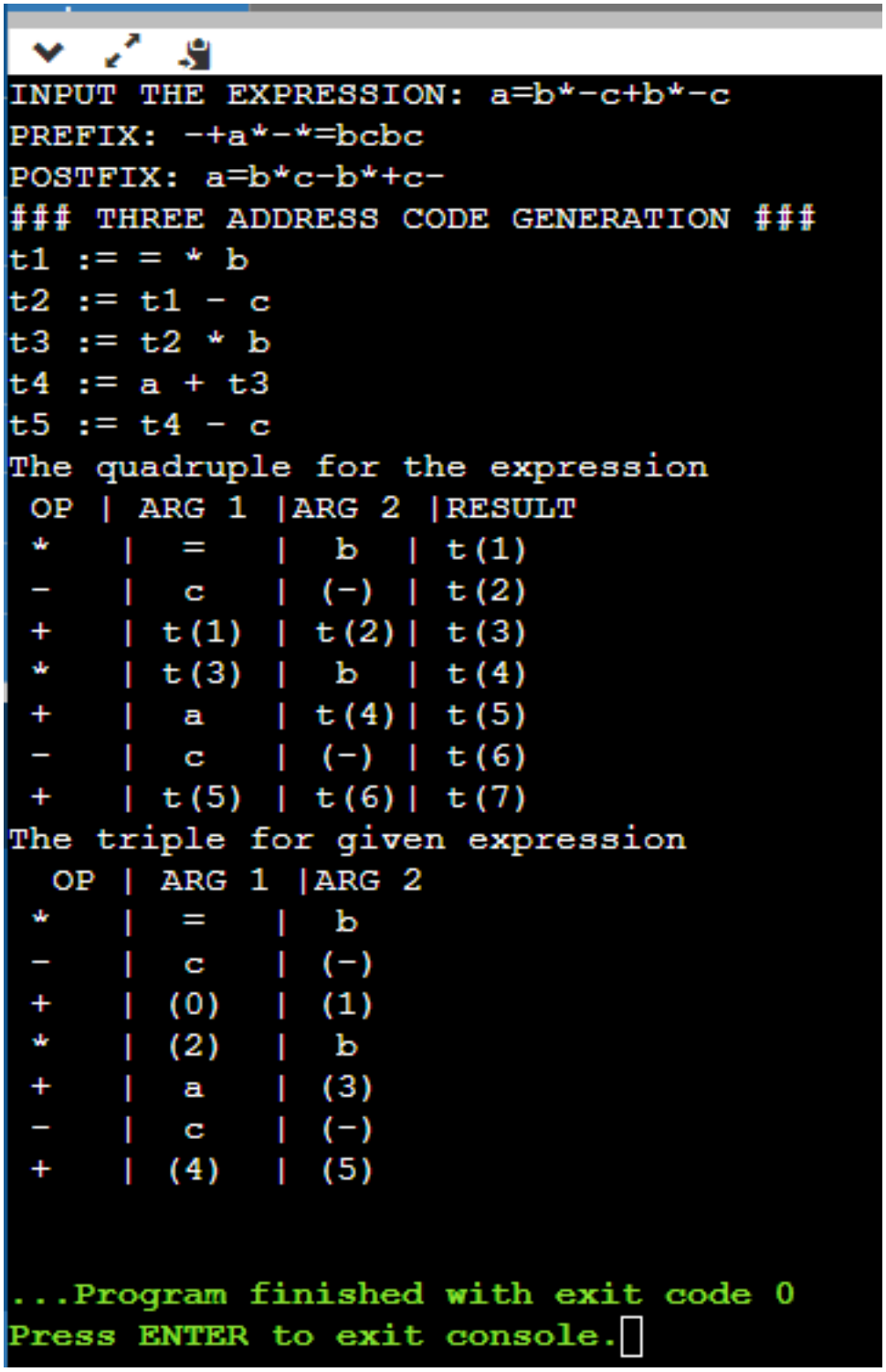
```
    # leftover
    while op_stack:
        op = op_stack.pop()
        a = exp_stack.pop()
        b = exp_stack.pop()
        exp_stack.append( op+b+a )
    print(f'PREFIX: {exp_stack[-1]}')
    return exp_stack[-1]

### THREE ADDRESS CODE GENERATION ###
def generate3AC(pos):
        print("### THREE ADDRESS CODE GENERATION ###")
        exp_stack = []
        t = 1

        for i in pos:
                if i not in OPERATORS:
                        exp_stack.append(i)
                else:
                        print(f't{t} := {exp_stack[-2]} {i} {exp_stack[-1]}')
                        exp_stack=exp_stack[:-2]
                        exp_stack.append(f't{t}')
                        t+=1

expres = input("INPUT THE EXPRESSION: ")
pre = infix_to_prefix(expres)
pos = infix_to_postfix(expres)
generate3AC(pos)
def Quadruple(pos):
  stack = []
  op = []
  x = 1
  for i in pos:
    if i not in OPERATORS:
      stack.append(i)
    elif i == '-':
        op1 = stack.pop()
        stack.append("t(%s)" %x)
        print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format(i,op1,"(-)"," t(%s)" %x))
```

```
        x = x+1
        if stack != []:
          op2 = stack.pop()
          op1 = stack.pop()
          print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format("+",op1,op2," t(%s)" %x))
          stack.append("t(%s)" %x)
          x = x+1
    elif i == '=':
      op2 = stack.pop()
      op1 = stack.pop()
      print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format(i,op2,"(-)",op1))
    else:
      op1 = stack.pop()
      op2 = stack.pop()
      print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format(i,op2,op1," t(%s)" %x))
      stack.append("t(%s)" %x)
      x = x+1
print("The quadruple for the expression ")
print(" OP | ARG 1 |ARG 2 |RESULT  ")
Quadruple(pos)

def Triple(pos):
        stack = []
        op = []
        x = 0
        for i in pos:
          if i not in OPERATORS:
            stack.append(i)
          elif i == '-':
            op1 = stack.pop()
            stack.append("(%s)" %x)
            print("{0:^4s} | {1:^4s} | {2:^4s}".format(i,op1,"(-)"))
            x = x+1
            if stack != []:
              op2 = stack.pop()
              op1 = stack.pop()
              print("{0:^4s} | {1:^4s} | {2:^4s}".format("+",op1,op2))
              stack.append("(%s)" %x)
              x = x+1
          elif i == '=':
```

```
            op2 = stack.pop()
            op1 = stack.pop()
            print("{0:^4s} | {1:^4s} | {2:^4s}".format(i,op1,op2))
          else:
            op1 = stack.pop()
            if stack != []:
              op2 = stack.pop()
              print("{0:^4s} | {1:^4s} | {2:^4s}".format(i,op2,op1))
              stack.append("(%s)" %x)
              x = x+1
print("The triple for given expression")
print("  OP | ARG 1 |ARG 2  ")
Triple(pos)
```



OUTPUT:



```
INPUT THE EXPRESSION: a=b*-c+b*-c
PREFIX: -+a*-*=bcbc
POSTFIX: a=b*c-b*+c-
### THREE ADDRESS CODE GENERATION ###
t1 := = * b
t2 := t1 - c
t3 := t2 * b
t4 := a + t3
t5 := t4 - c
The quadruple for the expression
 OP | ARG 1 |ARG 2 |RESULT
 *    |   =    |   b   | t(1)
 -    |   c    | (-)  | t(2)
 +    | t(1)  | t(2)| t(3)
 *    | t(3)  |   b   | t(4)
 +    |   a    | t(4)| t(5)
 -    |   c    | (-)  | t(6)
 +    | t(5)  | t(6)| t(7)
The triple for given expression
  OP | ARG 1 |ARG 2
 *    |   =    |   b
 -    |   c    | (-)
 +    |  (0)   |  (1)
 *    |  (2)   |   b
 +    |   a    |  (3)
 -    |   c    | (-)
 +    |  (4)   |  (5)


...Program finished with exit code 0
Press ENTER to exit console.
```

RESULT: The program was successfully compiled and executed

CHITRALEKHA.CH
RA1911003010387