# Implementation of DAG

# RA1911003010414

**Saket Kumar Baranwal**

**Aim:** A program to 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 y operand is undefined then create node(y).
5. 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 child 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.

# Program:

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[-1]}')

exp\_stack=exp\_stack[:-2] exp\_stack.append(f't{t}') t+=1

:=

{exp\_stack[-2]} {i}

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}

{2:^4s}".format(i,op1,"(-)"))

x = x+1

if stack != []:

op2 = stack.pop() op1 = stack.pop()

print("{0:^4s}

{2:^4s}".format("+",op1,op2))

stack.append("(%s)" %x) x = x+1

elif i == '=':

op2 = stack.pop() op1 = stack.pop()

print("{0:^4s}

{2:^4s}".format(i,op1,op2))

else:

op1 = stack.pop() if stack != []:

op2 = stack.pop()

|

{1:^4s} |

|

{1:^4s}

|

|

{1:^4s}

|

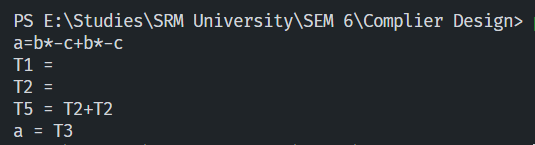
print("{0:^4s} | {1:^4s} |

{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:-



**Result:-**

The program was successfully compiled and run.