**Experiment no – 09**

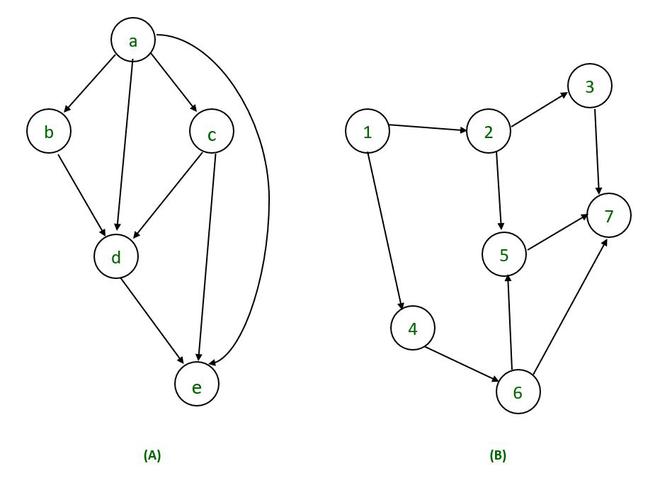
**Aim: Write a code to generate the DAG for the input arithmetic expression.**

**Theory: -**

**Directed Acyclic Graph:**  
The Directed Acyclic Graph (DAG) is used to represent the structure of basic blocks, to visualize the flow of values between basic blocks, and to provide optimization techniques in the basic block. To apply an optimization technique to a basic block, a DAG is a three-address code that is generated as the result of an intermediate code generation.

* Directed acyclic graphs are a type of data structure and they are used to apply transformations to basic blocks.
* The Directed Acyclic Graph (DAG) facilitates the transformation of basic blocks.
* DAG is an efficient method for identifying common sub-expressions.
* It demonstrates how the statement’s computed value is used in subsequent statements.

**Examples of directed acyclic graph :**



**Directed Acyclic Graph Characteristics :**  
A Directed Acyclic Graph for Basic Block is a directed acyclic graph with the following labels on nodes.

* The graph’s leaves each have a unique identifier, which can be variable names or constants.
* The interior nodes of the graph are labelled with an operator symbol.
* In addition, nodes are given a string of identifiers to use as labels for storing the computed value.
* Directed Acyclic Graphs have their own definitions for transitive closure and transitive reduction.
* Directed Acyclic Graphs have topological orderings defined.

**Algorithm for construction of Directed Acyclic Graph :**  
There are three possible scenarios for building a DAG on three address codes:

**Case 1 –**  x = y op z  
**Case 2 –** x = op y  
**Case 3  –**  x = y

Directed Acyclic Graph for the above cases can be built as follows :

**Step 1 –**

* If the y operand is not defined, then create a node (y).
* If the z operand is not defined, create a node for case(1) as node(z).

**Step 2 –**

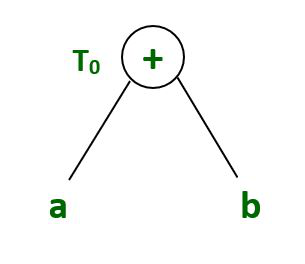
* Create node(OP) for case(1), with node(z) as its right child and node(OP) as its left child (y).
* For the case (2), see if there is a node operator (OP) with one child node (y).
* Node n will be node(y)  in case (3).

**Step 3 –**  
Remove x from the list of node identifiers. Step 2: Add x to the list of attached identifiers for node n.

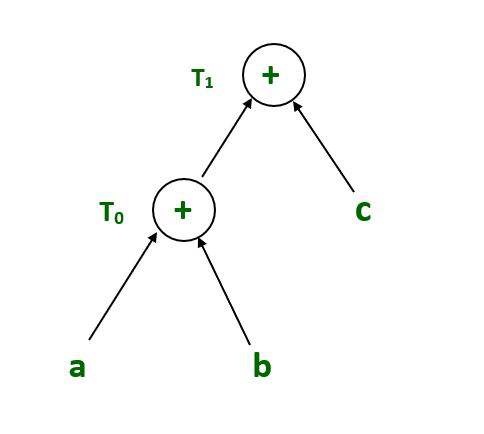
**Example :**

*T0 = a + b         —Expression 1  
T1 = T0 + c       —-Expression 2  
d = T0 + T1—–Expression 3*

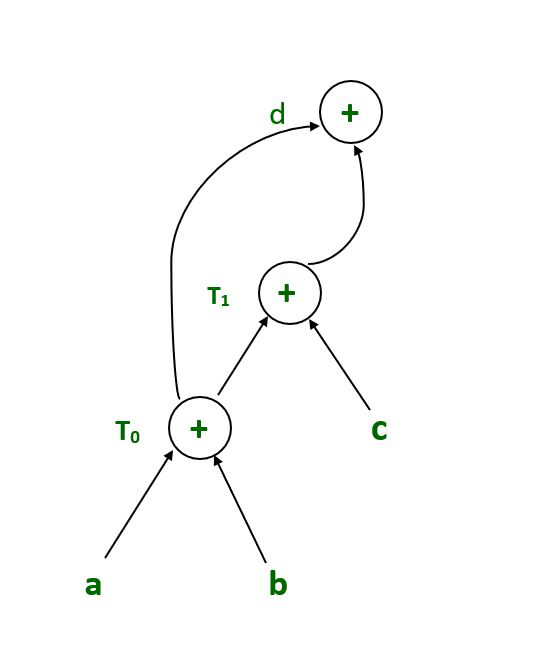
***Expression 1 :                   T0 = a + b***



**Expression 2:                    T1 = T0 + c**

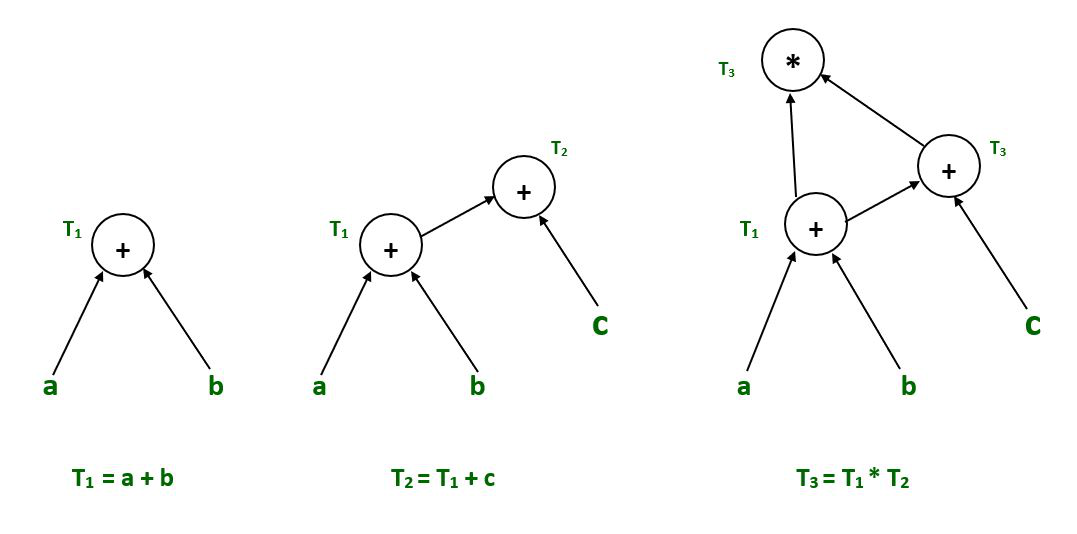


**Expression 3 :                          d = T0 + T1**



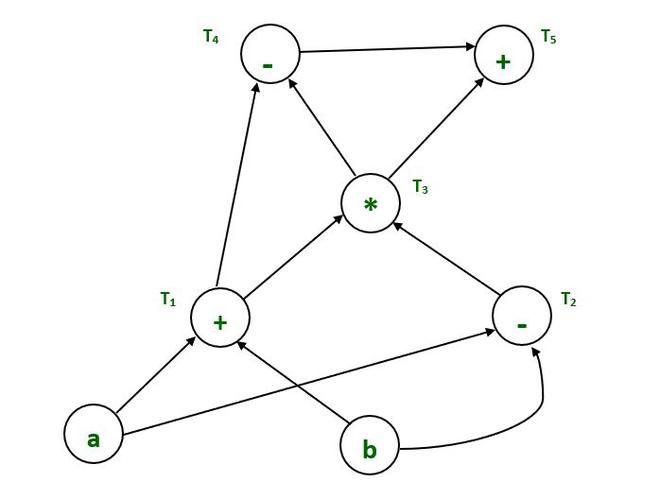
**Example :**

*T1 = a + b        
T2 = T1 + c       
T3= T1 x T2*



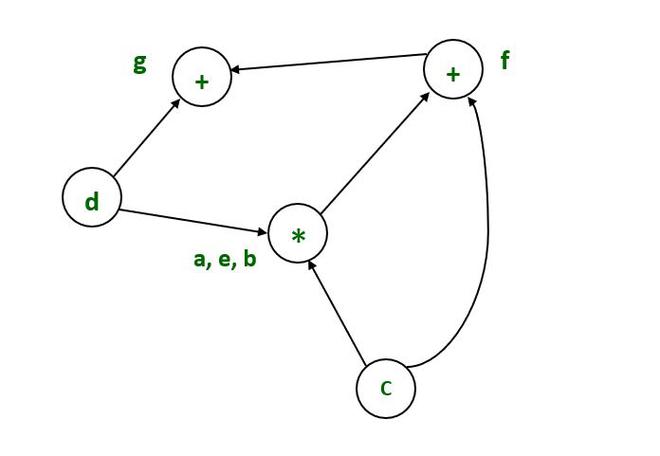
**Example :**

*T1= a + b  
T2= a – b  
T3 = T1 \* T2  
T4 = T1 – T3  
T5 = T4 + T3*



**Example :**

*a = b x c  
d = b  
e = d x c  
b = e  
f = b + c  
g = f + d*



**Example :**

*T1:= 4\*I0*

*T2:= a[T1]*

*T3:= 4\*I0*

*T4:= b[T3]*

*T5:= T2 \* T4*

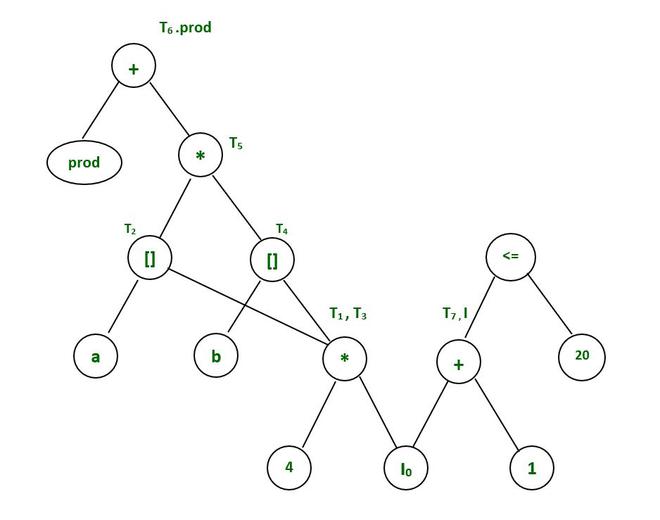
*T6:= prod + T5*

*prod:= T6*

*T7:= I0 + 1*

*I0:= T7*

*if I0<= 20 goto 1*



**Code:**

# **Code to generate the DAG for the input arithmetic expression**

class Graph:

# Constructor

def \_\_init\_\_(self, edges, n):

# A list of lists to represent an adjacency list

self.adjList = [[] for \_ in range(n)]

# add edges to the directed graph

for (src, dest) in edges:

self.adjList[src].append(dest)

# Perform DFS on the graph and set the departure time of all vertices of the graph

def DFS(graph, v, discovered, departure, time):

# mark the current node as discovered

discovered[v] = True

# do for every edge (v, u)

for u in graph.adjList[v]:

# if `u` is not yet discovered

if not discovered[u]:

time = DFS(graph, u, discovered, departure, time)

# ready to backtrack

# set departure time of vertex `v`

departure[v] = time

time = time + 1

return time

# Returns true if the given directed graph is DAG

def isDAG(graph, n):

# keep track of whether a vertex is discovered or not

discovered = [False] \* n

# keep track of the departure time of a vertex in DFS

departure = [None] \* n

time = 0

# Perform DFS traversal from all undiscovered vertices

# to visit all connected components of a graph

for i in range(n):

if not discovered[i]:

time = DFS(graph, i, discovered, departure, time)

# check if the given directed graph is DAG or not

for u in range(n):

# check if (u, v) forms a back-edge.

for v in graph.adjList[u]:

# If the departure time of vertex `v` is greater than equal

# to the departure time of `u`, they form a back edge.

# Note that `departure[u]` will be equal to `departure[v]`

# only if `u = v`, i.e., vertex contain an edge to itself

if departure[u] <= departure[v]:

return False

# no back edges

return True

if \_\_name\_\_ == '\_\_main\_\_':

# List of graph edges as per the above diagram

edges = [(0, 1), (0, 3), (1, 2), (1, 3), (3, 2), (3, 4), (3, 0), (5, 6), (6, 3)]

# total number of nodes in the graph (labelled from 0 to 6)

n = 7

# build a graph from the given edges

graph = Graph(edges, n)

# check if the given directed graph is DAG or not

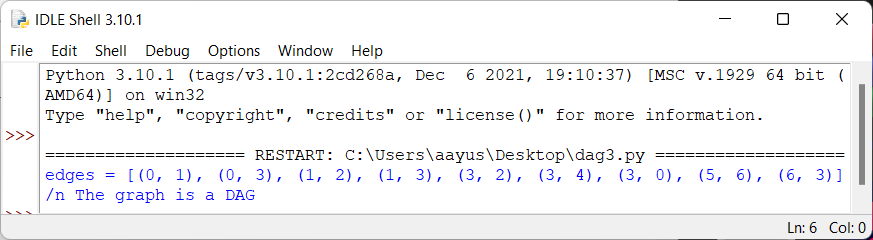
if isDAG(graph, n):

print('The graph is a DAG')

else:

print('The graph is not a DAG')

**Output:**

****

**Conclusion**:-

We successfully constructed and checked DAG for the input arithmetic expression.

**Reference:**

https://www.techiedelight.com/check-given-digraph-dag-directed-acyclic-graph-not/