# Experiment no – 09

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

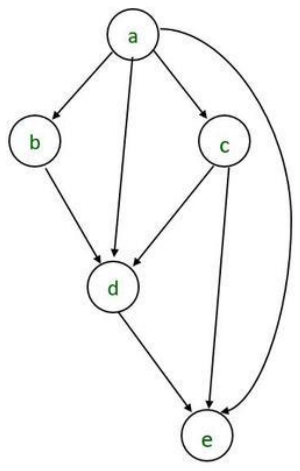
## Theory: -

### Directed Acyclic Graph:

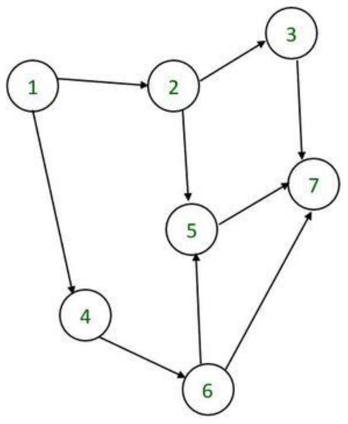
The Directed Acyclic Graph (DAG) is used to represent the structure of basic blocks, to visualize the flow of values between basic blocks, and to provide optimization techniques in the basic block. To apply an optimization technique to a basic block, a DAG is a three-address code that is generated as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



(A)



(B)

### 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 –**

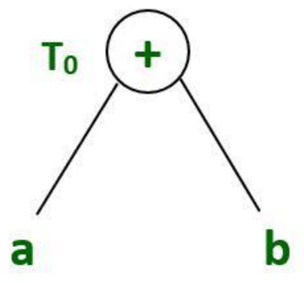
- Create node(OP) for case(1), with node(z) as its right child and node(OP) as its left child (y).
- For the case (2), see if there is a node operator (OP) with one child node (y).
- Node n will be node(y) in case (3).

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

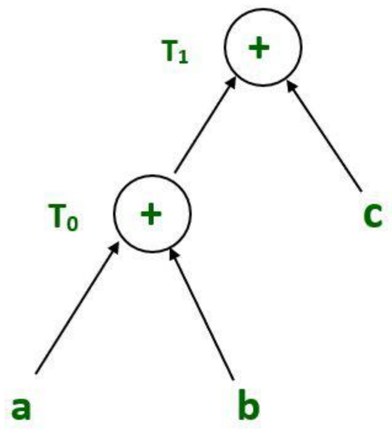
**Example :**

*$T_0 = a + b$ —Expression 1*
*$T_1 = T_0 + c$ —-Expression 2*
*$d = T_0 + T_1$ ––Expression 3*

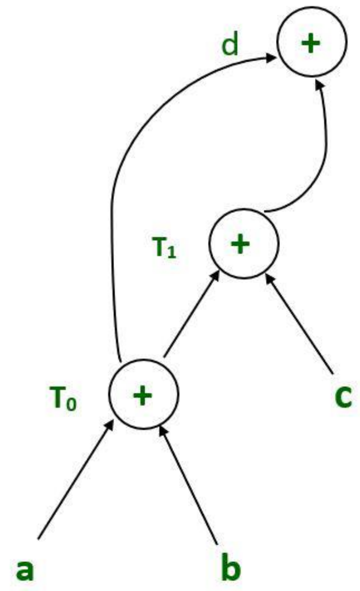
***Expression 1 :*** ***$T_0 = a + b$***



Expression 2: $T_1 = T_0 + c$

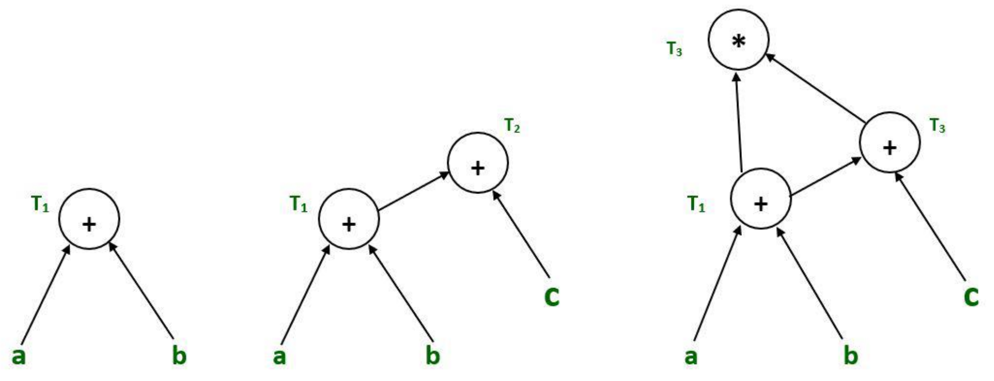


Expression 3 : $d = T_0 + T_1$



Example :

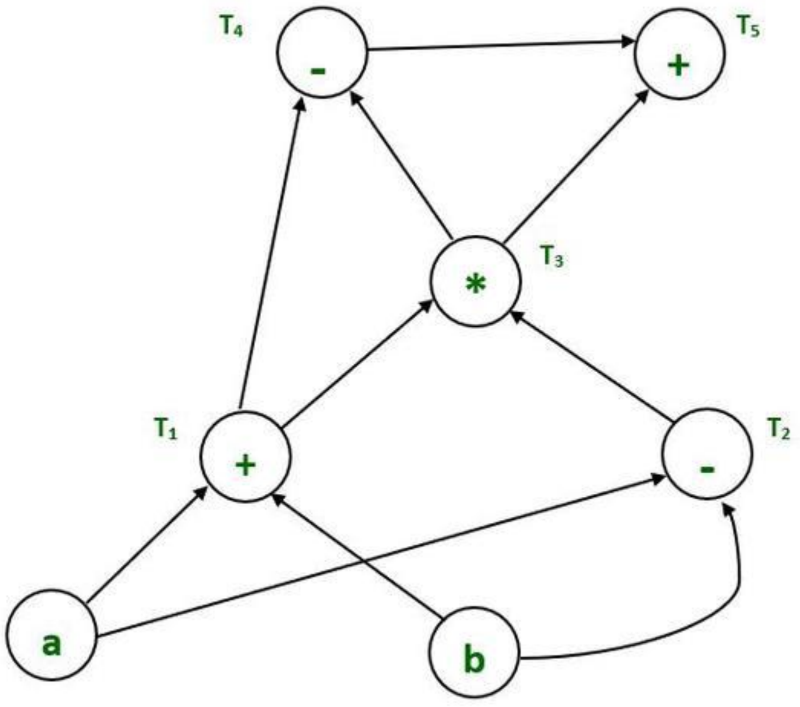
*$T_1 = a + b$*
*$T_2 = T1 + c$*
*$T_3 = T1 \times T2$*



$T_1 = a + b$ $T_2 = T_1 + c$ $T_3 = T_1 * T_2$

Example :

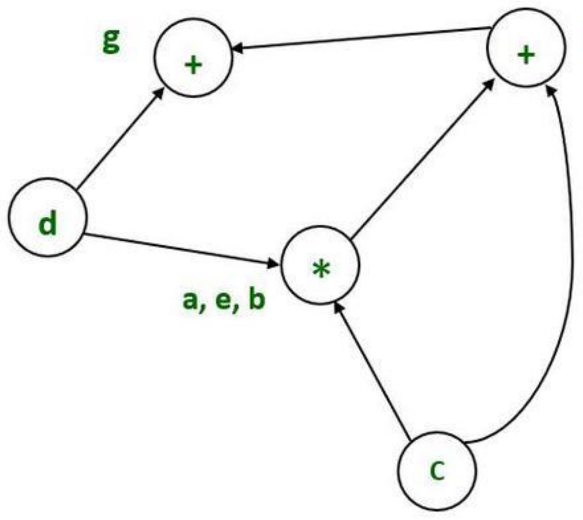
*$T_1 = a + b$*
*$T_2 = a - b$*
*$T_3 = T_1 * T_2$*
*$T_4 = T_1 - T_3$*
*$T_5 = T_4 + T_3$*



Example :

*$a = b \times c$*
*$d = b$*

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



Example :

$T_1 := 4 * I_0$

$T_2 := a[T_1]$

$T_3 := 4 * I_0$

$T_4 := b[T_3]$

$T_5 := T_2 * T_4$

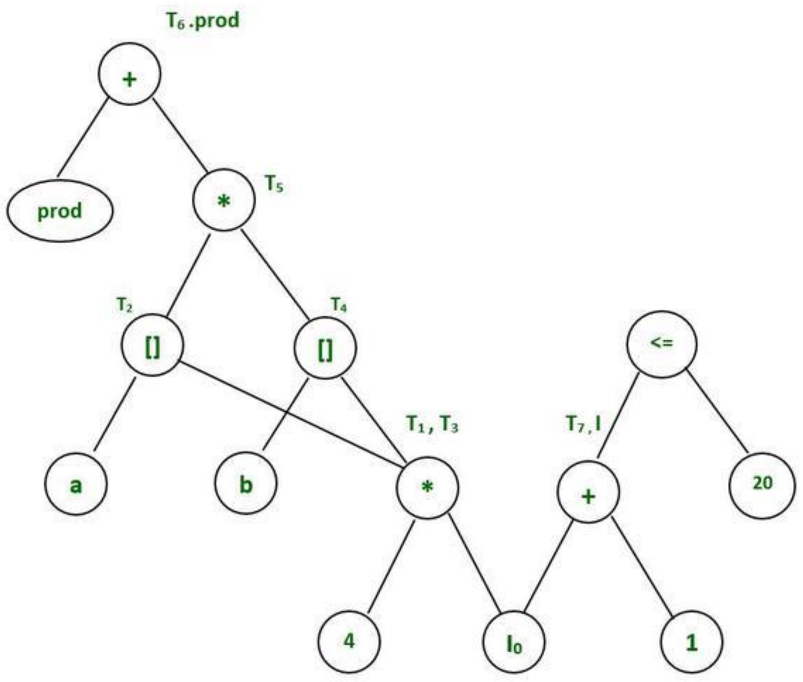
$T_6 := prod + T_5$

$prod := T_6$

$T_7 := I_0 + 1$

$I_0 := T_7$

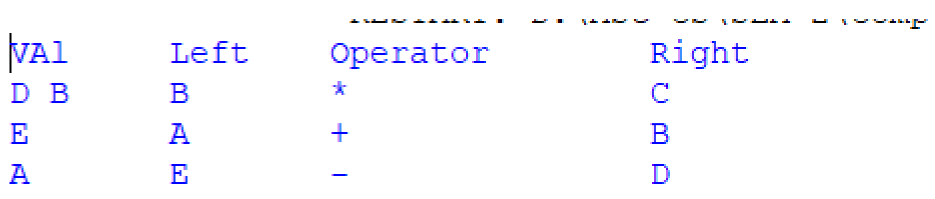
*if* $I_0 <= 20$ *goto 1*



Code:

```
import re
grammer = ["D=B*C", "E=A+B", "B=B*C", "A=E-D"]
x=[]

opr = []
left =[]
right =[]
temp=[]
val =[]
tempNew=[]
valNew =[]

for i in grammer:
    a = i.split("=")
    val.append(a[0])
    temp.append(a[1])


loopTime = len(temp)
i = 0;
j = 0;
while i < loopTime:
    while j < loopTime:
        if i == j:
            j=j+1
        if temp[j] == temp[i]:
            x= val[i] + " " + val[j]
            val.pop(j)
            temp.pop(j)
            val[i] = x
            j+=1
        j+=1
    i+=1


count = 0
for i in temp:
    if len(i) == 3:
        re.split('[+-]{1}', i)
        opr.append(i[1])
        left.append(i[0])
        right.append(i[2])
        count += 1
        continue

    if len(i) == 2:
        i.split("+")
        opr.append(i[0])
        left.append("-")
        right.append(i[1])
        count += 1
        continue
    if len(i) == 1:
        x = val[count]
        x = x + " " + i
        for k in (0,len(val)-1):
            if val[k] == i:
                temp2 = k
                val[temp2] = x
        count += 1
        continue



print("VAl\tLeft\tOperator\tRight")
for i in range(0,count):
    print(f"{val[i]}\t{left[i]}\t{opr[i]}\t\t{right[i]}")
```

Output:



```
VAl     Left    Operator        Right
D B     B       *               C
E       A       +               B
A       E       -               D
```

Conclusion:-

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