Directed Acyclic Graph(DAG)

In compiler design, optimization is mainly done to **reduce execution time**, **save memory**, and **improve performance**. One important technique for optimizing code within a **basic block** is the **DAG (Directed Acyclic Graph) representation**.

A DAG representation shows **how expressions inside a basic block are related**, helping the compiler remove unnecessary computations and generate efficient code.

What is a DAG?

A **Directed Acyclic Graph (DAG)** is a graph:

- With **directed edges**
- Having **no cycles**
- Used to represent **expressions and assignments** inside a basic block

In DAG:

- **Leaf nodes** represent **variables or constants**
- **Internal nodes** represent **operators** (+, −, ×, ÷)
- Each node represents a **unique computation**

Or

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

Why DAG is Used for Basic Blocks

A basic block contains straight-line code with no branching inside. Because of this:

- The order of evaluation is fixed
- DAG can safely represent all computations
- Optimizations can be applied locally

DAG Representation of a Basic Block

General Rules for Constructing a DAG

1. **Create leaf nodes** for variables and constants
2. **Create an internal node** for each operator
3. If an expression already exists, **reuse the same node**
4. Assign variable names to nodes instead of creating new nodes
5. Avoid duplicate computation nodes

Steps to Construct a DAG for a Basic Block:

1. Identify Expressions in the basic block.

2. Check for Existing Nodes:

If the same operation with the same operands exists, reuse it.

3. Create New Node:

If no identical node exists, create a new one.

4. Label Nodes:

Assign temporary or variable names to output nodes.

Advantages of DAG:

• Eliminates common subexpressions.

• Removes dead code.

• Improves execution efficiency.

• Provides clear data dependency among computations.

Example 1:

Expression:

ini

Copy code

```ini
a = b + c
d = b + c
e = a + d
```

Normal (Unoptimized) TAC:

ini

Copy code

```ini
t1 = b + c
a = t1
t2 = b + c
d = t2
t3 = a + d
e = t3
```

DAG Representation:

css

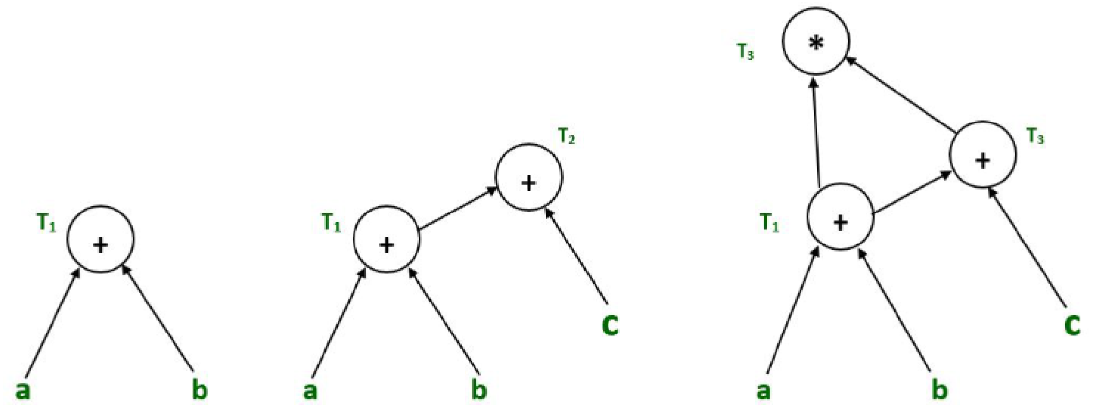
Copy code

```css
      (+)
     /   \
    b     c
     \   /
      (a,d)
        \
         (+)
        /   \
      a,d    -
        \
         e
```

Optimization:
Since b + c is computed twice, the DAG identifies it as a common subexpression — so it's calculated only once.

Example 2:

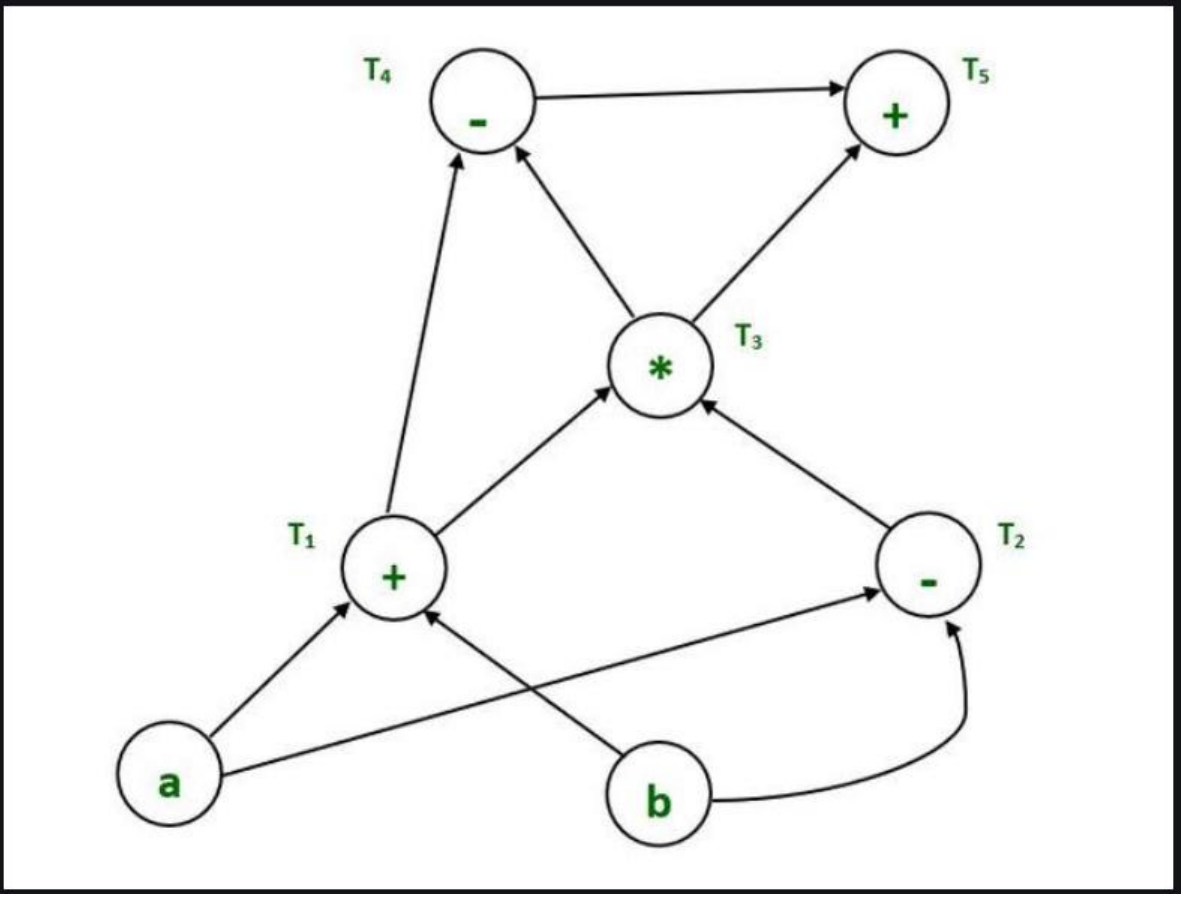
$T_1 = a + b$
$T_2 = T1 + c$
$T_3 = T1 \times T2$



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

Example 3:

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



Application of Directed Acyclic Graph

- **Identification of Common Subexpressions** : A DAG helps detect repeated subexpressions in code. This enables efficient common subexpression elimination.
- **Tracking Variable Usage** : It identifies variables used within the block and those computed externally. This aids in understanding data dependencies.
- **Statement Value Tracking** : DAGs show which statements produce values used outside the block. This supports code optimization and preservation.
- **Code Representation** : Code can be modeled as a DAG showing inputs and outputs of operations. This helps visualize and optimize computations.

- **Reactive Value Systems** : In some languages, values are linked via a DAG. Changing one value triggers updates in all dependent values.