Directed Acyclic Graphs....

(DAGs)

LearnVidFun....

Definitions-

In compiler design, a directed acyclic graph (DAG) is an abstract syntax tree (AST) with a unique node for each value.

OR

A directed acyclic graph (DAG) is a directed graph that contains no cycles.

————xxxxxxxx————

* Use of DAG for optimizing basic blocks :-

- DAG is a useful data structure for implementing transformations on basic blocks.
- A basic block can be optimized by the construction of DAG.
- A DAG can be constructed for a block and certain transformations such as common subexpression elimination and dead code elimination can be applied for performing the local optimization.
- To apply the transformations on basic block, a DAG is constructed from three address statement.

——————xxxxxxxxx——————

Properties of a DAG

① The reachability relation in a DAG forms a partial order and any finite partial order may be represented by a DAG using reachability.

② The transitive reduction and transitive closure are both uniquely defined for DAGs.

③ Every DAG has a topological ordering.

———×××××××———

Applications of a DAG

The DAG is used in-

① determining the common subexpression (expressions computed more than once).

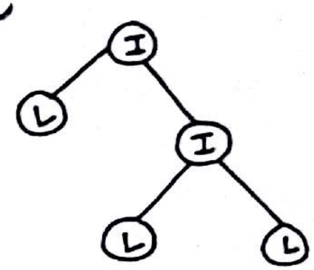
② determining which names are used in the block and computed outside the block.

③ determining which statements of the block could have their computed value outside the block.

④ Simplifying the list of quadruples by eliminating the common subexpressions and not performing the assignment of the form x:=y until and unless it is a must.

xxxxxxxx

Rules for the construction of DAG :-



Rule-1 :- In a DAG,

→ Leaf nodes represent identifiers, names or constants.

→ Interior nodes represent operators.

Rule-2 :- While constructing DAG, there is a check made to find if there is an existing node with the same children. A new node is created only when such a node does not exist. This action allows us to detect common sub-expressions and eliminate the re-computation of the same.

Rule-3 :- The assignment of the form $x := y$ must not be performed until and unless it is a must.

——— xxxxxxx ———

Problems

Problem-1:- Construct DAG for the given expression:-

$$(a+b) * (a+b+c)$$

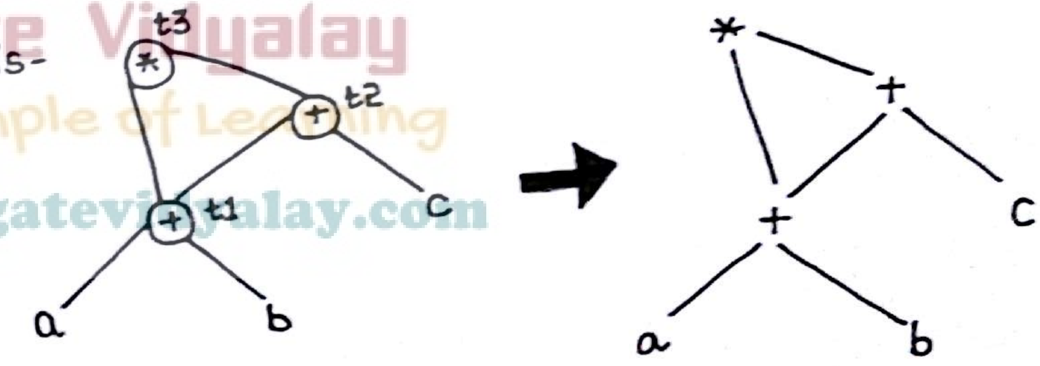
Soln:- Three address code for the given expression is-

$t1 = a + b$

$t2 = t1 + c$

$t3 = t1 * t2$

The DAG is-



Explanation:-

From the constructed DAG, we observe that the common subexpression (a+b) is translated into a single node in the DAG. The computation is carried out only once and stored in the identifier t1 and reused later.

This illustrates how the DAG construction scheme identifies the common sub-expression & helps in eliminating its re-computation later.

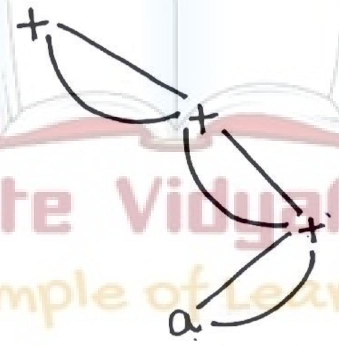
xxxxxxxx

Problem-02: Construct DAG for the given expression -

$$(((a+a) + (a+a)) + ((a+a) + (a+a)))$$

Soln:-

DAG for the given expression is-



———— xxxxxxx ————

Problem-03: Construct the DAG for the following block -

$$a = b * c$$

$$d = b$$

$$e = d * c$$

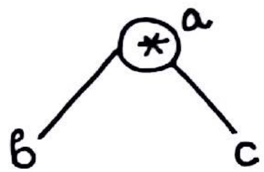
$$b = e$$

$$f = b + c$$

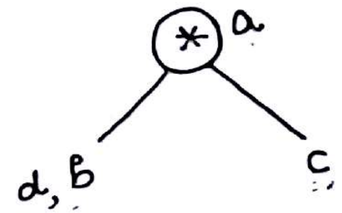
$$g = f + d$$

Soln:-

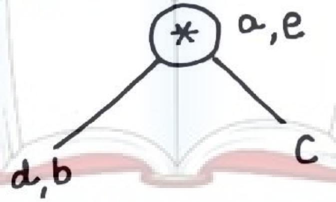
Step-1:



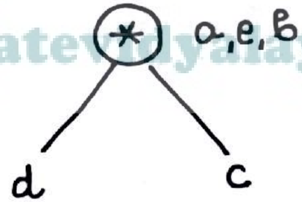
Step-2 :-



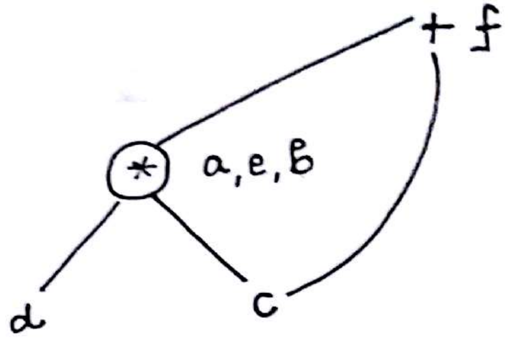
Step-3 :-



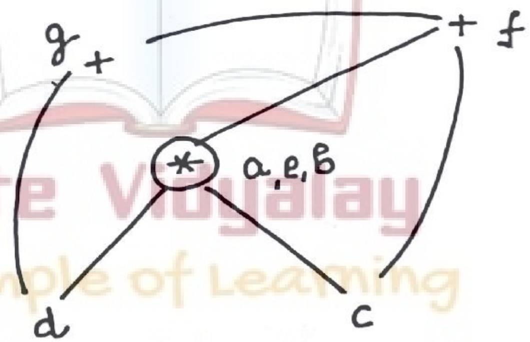
Step-4 :-



Step-5 :-



Step-6 :-



——— xxxxxxxx ———

Problem-4 :- optimitize the block given in problem-3.

Soln:-

Step-1:- First construct the DAG for the given block.

Step-2:- Now, the optimized code can be generated by traversing the DAG.

1. The common subexpression $e = d * c$ which is actually $b * c$ ($\because d = b$) is eliminated.

2. The dead code $b = e$ is eliminated.

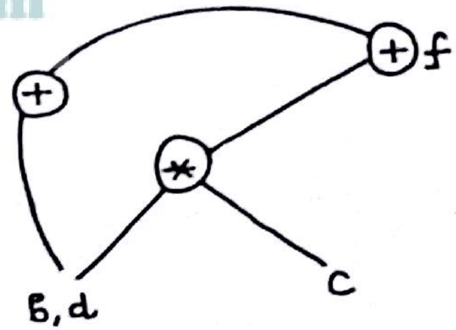
The optimized basic block is-

$a = b * c$

$d = b$

$f = a + c$

$g = f + d$



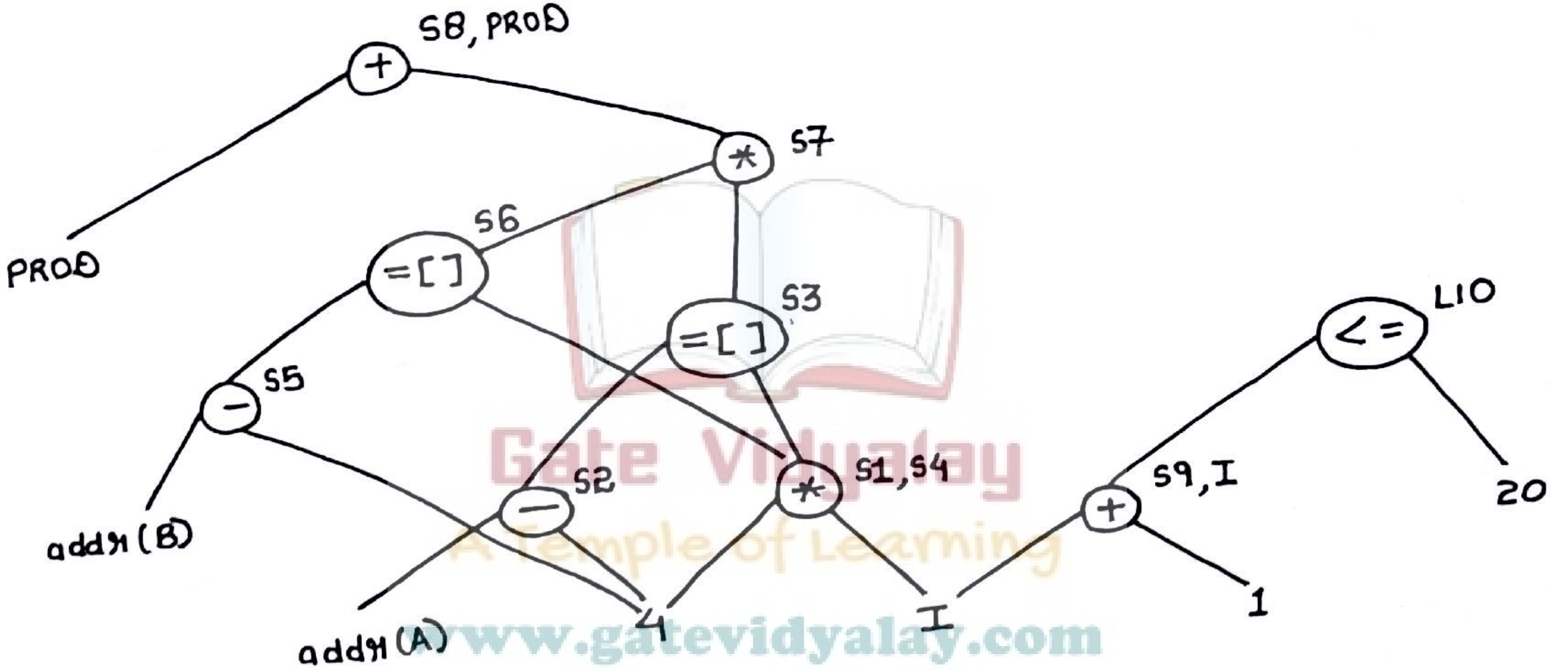
———— xxxxxxxx ————

Problem-5 :- Consider the following Basic Block. Draw the DAG representation of the Block and identify local common sub-expressions. Eliminate the common expressions and rewrite the Basic Block.

```
L10 :  S1 = 4 * I
       S2 = addr(A) - 4
       S3 = S2[S1]
       S4 = 4 * I
       S5 = addr(B) - 4
       S6 = S5[S4]
       S7 = S3 * S6
       S8 = PROD + S7
       PROD = S8
       S9 = I + 1
       I = S9
       If I <= 20 goto L10
```

<u>Solution:-</u>

<u>DAG representation for the Block is :-</u>



In this code fragment,

- 4*I is a common subexpression. Hence, we can eliminate S4 because S1 = S4.

- S8 = PROD + S7
  PROD = S8
  can be optimized as → PROD = PROD + S7

- S9 = I + 1
  I = S9
  can be optimized as → I = I + 1

After eliminating S4, S8 and S9, we get -

```
L10 :
        S1 = 4*I
        S2 = addr(A)-4
        S3 = S2[S1]
        S5 = addr(B)-4
        S6 = S5[S1]
        S7 = S3 * S6
        PROD = PROD + S7
        I = I+1
        IF I <= 20 GOTO L10
```


xxxxxxxx