DIRECTED ACYCLIC GRAPH REPRESENTAION OF BASIC BLOCK

DAG (Directed Acyclic graph) is a useful data structure to implement transformations on basic blocks.

Construction a DAG from three address code is a good way of determining common subexpression within a block.

Algorithm for the construction of DAG:

Input: Basic block
Output: DAG for the basic block containing the following information

1. A label for each node. For the leaves the label is an identifier and for the interior nodes it is an operator symbol.
2. For each node there will be list of identifiers.

Current TAC is of any of the form like the following

(i) x=y op z (ii)x=op y (iii)x=y

1. if node(y) is undefined, create a node labeled y, if node(z) is undefined create a node labeled z.
2. in case (i), determine if there is a node labeled op with the left child is node(y) and right child is node(z), if not create such node. in case (ii), determine if there is a node labeled op with lone child node(y), if not create such node. In case (iii), let n be node(y).
3. append x to the list of attached identifiers for the node n found in step(2).

Example TAC sequence

(1) t1=4*i
(2) t2=a[t1]
(3) t3=4*i
(4) t4=b[t3]
(5) t5=t2*t4
(6) t6=prod+t5
(7) prod=t6
(8) t7=i+1
(9) i=t7

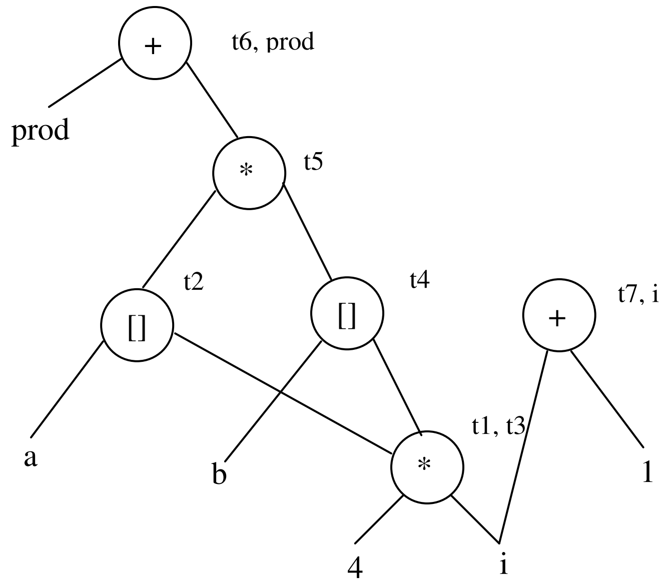

Fig. DAG representation of the above TAC sequence

To start with the first TAC, leaves 4 and i are not there, so create it. First time you are looking for node * with left child 4 and right child i. create and attach the identifier t1 to node *. For the second TAC, node t1 was there. So create a node for a and also create node for [] with a as the left child and t1 as right child and attach the identifier t2 to node []. For the third TAC no need to create any node because subtree exists for that expression. Continuing like this we will get the fig above.