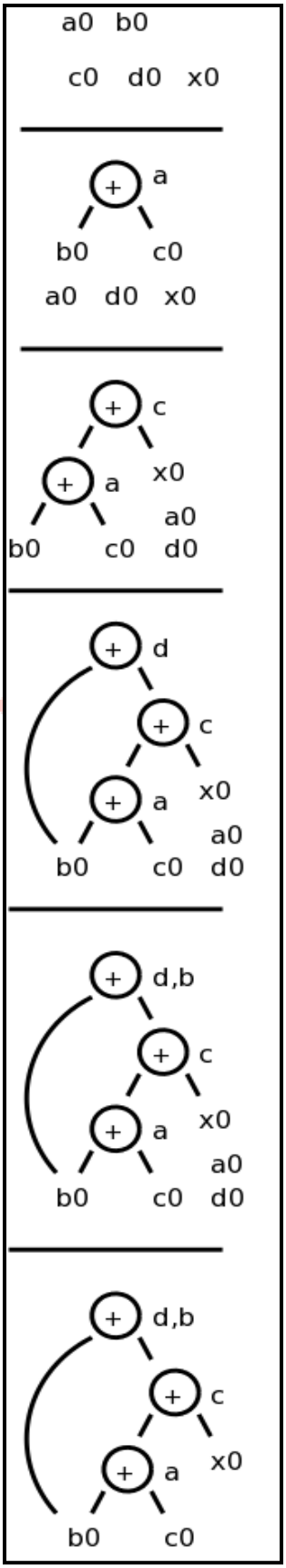
Optimization of Basic Blocks

The DAG Representation of Basic Blocks

The goal is to obtain a visual picture of how information flows through the block. The leaves will show the values entering the block and as we proceed *up* the DAG we encounter uses of these values defs (and redefs) of values and uses of the new values.

Formally, this is defined as follows.

1. Create a leaf for the initial value of each variable appearing in the block. (We do not know what that the value is, not even if the variable has ever been given a value).
2. Create a node N for each statement s in the block.
   i. Label N with the operator of s. This label is drawn inside the node.
   ii. Attach to N those variables for which N is the last def in the block. These additional labels are drawn along side of N.
   iii. Draw edges from N to each statement that is the last def of an operand used by N.
3. Designate as *output nodes* those N whose values are live on exit, an officially-mysterious term meaning values possibly used in another block. (Determining the live on exit values requires global, i.e., inter-block, flow analysis.)



As we shall see in the next few sections various basic-block optimizations are facilitated by using the DAG.

Finding Local Common Sub expressions

As we create nodes for each statement, proceeding in the static order of the statements, we might notice that a new node is just like one already in the DAG in which case we don't need a new node and can use the old node to compute the new value in addition to the one it already was computing.

Specifically, we do not construct a new node if an existing node has the same children in the same order and is labeled with the same operation.

Consider computing the DAG for the following block of code.

```
a = b + c
c = a + x
d = b + c
b = a + x
```

The DAG construction is explain as follows (the movie on the right accompanies the explanation).

1. First we construct leaves with the initial values.
2. Next we process a = b + c. This produces a node labeled + with a attached and having b_0 and c_0 as children.
3. Next we process c = a + x.
4. Next we process d = b + c. Although we have already computed b + c in the first statement, the c's are not the same, so we produce a new node.
5. Then we process b = a + x. Since we have already computed a + x in statement 2, we do not produce a new node, but instead attach b to the old node.
6. Finally, we tidy up and erase the unused initial values.

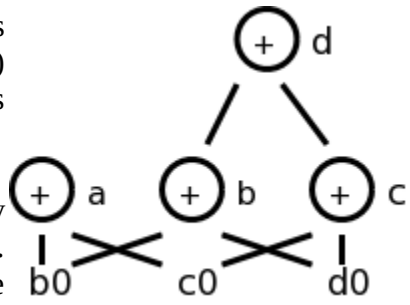
You might think that with only three computation nodes in the DAG, the block could be reduced to three statements (dropping the computation of b). However, this is **wrong**. Only if b is dead on exit can we omit the computation of b. We can, however, replace the last statement with the simpler

b = c.

Sometimes a combination of techniques finds improvements that no single technique would find. For example if a-b is computed, then both a and b are incremented by one, and then a-b is computed again, it will not be recognized as a common sub expression even though the value has not changed. However, when combined with various algebraic transformations, the common value can be recognized.

Dead Code Elimination

Assume we are told (by global flow analysis) that certain values are dead on exit. We examine each root (node with no ancestor) and delete any that have no *live* variables attached. This process is repeated since new roots may have appeared.



For example, if we are told, for the picture on the right, that only a and b are live, then the root d can be removed since d is dead. Then the rightmost node becomes a root, which also can be removed (since c is dead).