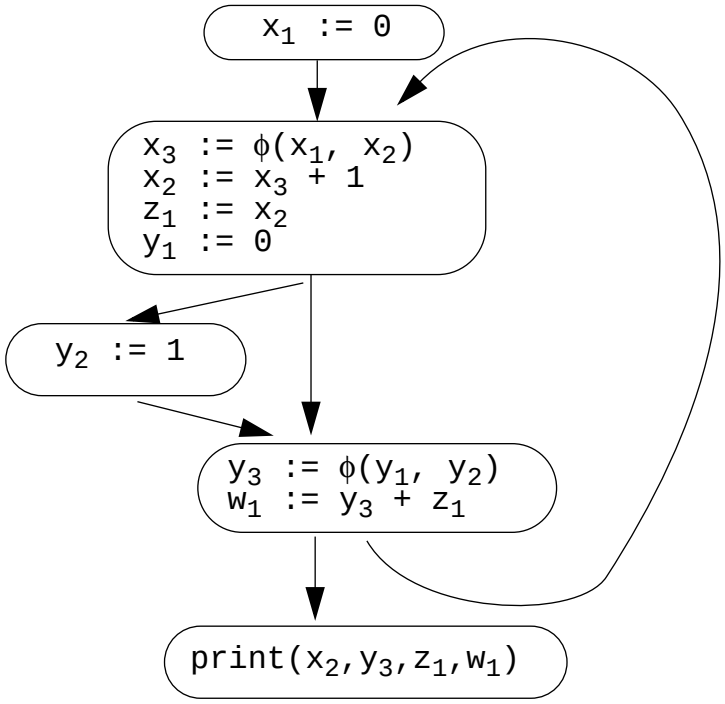
Homework Assignment #2 Sample Solutions

Due Wednesday 2/7, at the start of lecture

1. Put the following program in SSA form (you may draw a control flow graph to illustrate your solution):

```
x := 0;
do {
   x := x + 1;
   z := x;
   y := 0;
   if (...) {
      y := 1;
   }
   w := y + z;
} while (...);
print(x, y, z, w);
```



2. Give an algorithm for constant propagation that exploits def/use chains to work faster than the propagation-based algorithm presented in class. What is the time complexity of your algorithm, assuming def/use chains are already constructed? How, if at all, would converting the program to SSA form before constructing def/use chains help your analysis?

   One can use the generic worklist algorithm, operating over the def/use chain graph. Initialize all edges to $\top$ (top). Put all nodes on the worklist. When removing a node from the worklist, try to constant-fold it based on the info on its incoming def/use edges (each use can have multiple incoming edges, and so the info on each of these edges must be merged, using the lattice meet

operator). Then set the outgoing edge(s) to the info representing the result of the node: $\top$ if any argument is $\top$, a constant if the r.h.s. (after folding) is a constant, and $\bot$ otherwise. Put the downstream node onto the worklist if the result edge's value was changed. A classic optimistic iterative algorithm, with the effect of the resulting transformation (constant folding & propagation) included within the analysis.

Overall time complexity is O(N+U), where N is the number of nodes and U is the number of def/use edges (visit each node and edge at least once, and revisit each node and edge at most twice (the height of the constant-propagation lattice (3) - 1). [There's some disagreement about the worst-case number of def/use edges; Craig can only imagine $O(N^2V)$ such edges, while Tarjan claims that there are up to $O(E^2V)$ such edges. Who would you believe?]

Converting the program to SSA form would eliminate the need to merge multiple edge info's for each use, since each use would have a single incoming def/use edge. The ϕ functions would use the meet operator as their regular flow function. SSA form would also reduce the worst-case time complexity by reducing the worst-case number of def/use edges (O(EV) according to both Craig and Tarjan). [While E is $O(N^2)$, it may be less, so $O(N^2)$ is a worse bound than O(E).]

3. Give an algorithm for dead assignment elimination that exploits def/use chains to work faster than the propagation-based algorithm that used live variables analysis presented in class. Your algorithm should not miss any optimization opportunities found by the best live variables-based algorithm presented in class. What is the time complexity of your algorithm, assuming def/use chains are already constructed? How, if at all, would converting the program to SSA form before constructing def/use chains help your analysis?

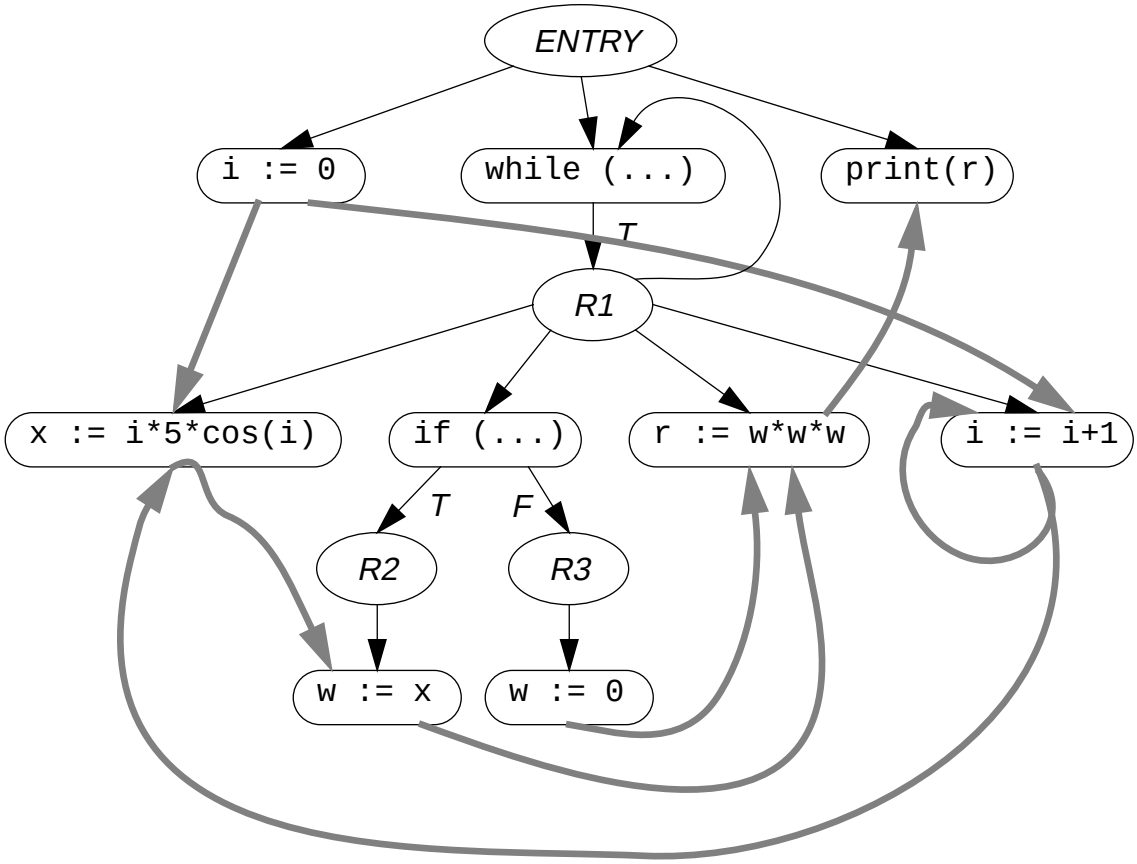
   One can use pretty much the same approach as above, except that 1) the def/use chain graph is traversed in reverse order, 2) the information on each edge is simply a bit saying whether the downstream use is live (initialized to dead), and 3) the flow function for a node sets the info on its uses' edges to live if either the node has side-effects (e.g. a call, a pointer store, a write to a global variable, a return statement, or an instruction that might trap) or if any of the node's def's outgoing edges is marked live. This algorithm will not consider any statement live until some intrinsically live statement requires its result; the example in class of the loop containing x := x + 1 will be handled correctly by this algorithm, never setting the instruction to live. Another classic optimistic iterative algorithm, with the effect of the resulting transformation (dead assignment elimination) included within the analysis.

   Overall time complexity is again O(N+U).

   Converting the program to SSA form would reduce the worst-case number of edges, but do nothing else.

4. These questions are about the control dependence graph.
   a. Construct the control dependence graph for the following program. Each assignment statement should have a separate node in the CDG. Also show the data dependence edges (all of flow, anti-, and output dependences) between nodes of the CDG (you need not convert to SSA form or create phi nodes).

```
i := 0;
while (...) {
   x := i * 5 * cos(i);
   if (...) {
      w := x;
   } else {
      w := 0;
   }
   r := w * w * w;
   i := i + 1;
}
print(r);
```



Technically, there are anti- and output dependencies from statements in the loop to statements in the loop for the next loop iteration (e.g. an anti dependence from w := x to x := ..., and an output dependence from x := ... to itself). The above diagram omits these dependences.

b. Using the CDG + DFG, identify all opportunities for code motion, including reordering statements, moving loop-invariant computations out of loops, and moving partially unused computations into conditional branches.

The top three statements cannot be reordered, since data dependences constrain them.

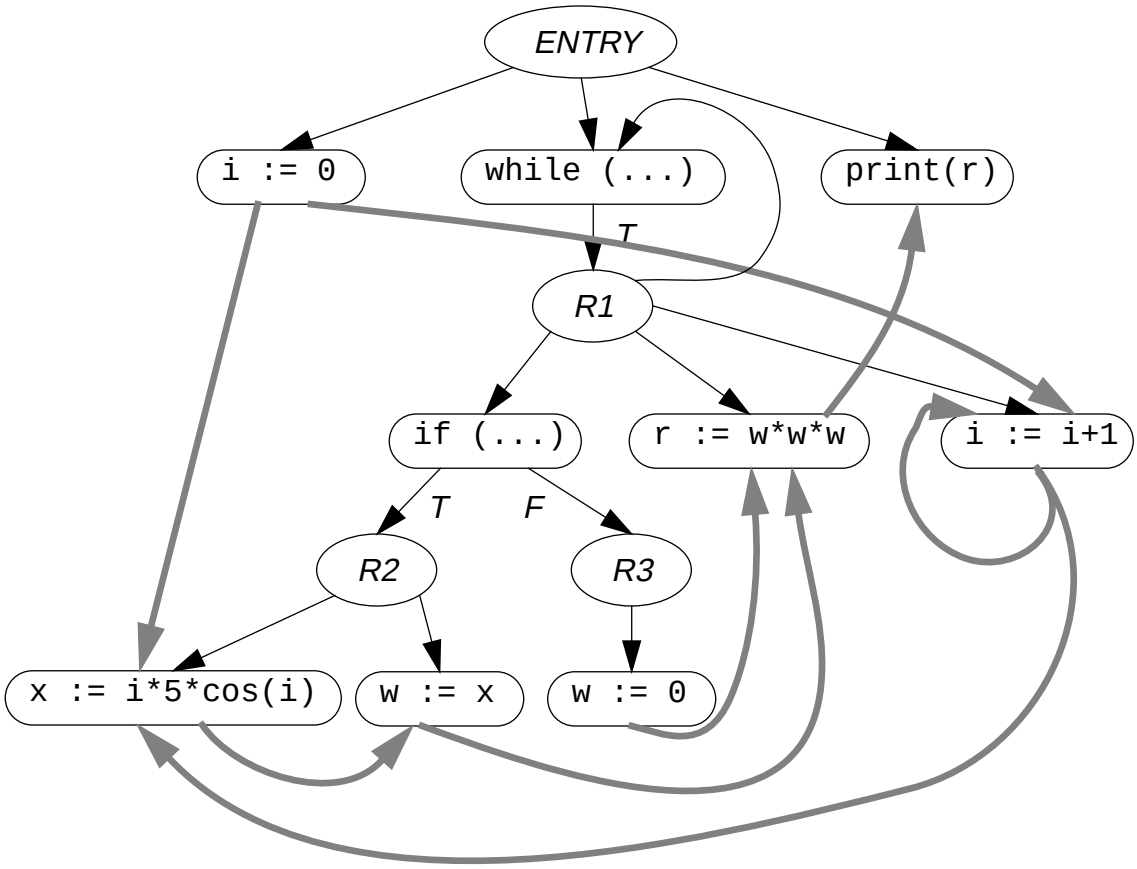
The increment of i can move anywhere after the x assignment. No other reorderings of the statements at the top of the while loop can be reordered.

The x assignment is only used in one part of the if, so it can be moved there.

The example was intended to have a loop-invariant computation also, but it didn't! oops.... Well, the w := 0 computation is loop-invariant, but moving it outside the loop will require leaving behind a copy statement like w := t (which is worse since it takes up a register to hold t throughout the execution of the loop).

To make the print(r) computation defined, the compiler can infer that the loop must be executed at least once (or, actually, that the value printed by the print(r) computation can be arbitrary). This could allow a Sufficiently Smart Compiler (TM) to hoist the r assignment out of the loop, after the loop but before the print. If the while loop is executed at least once, w will be defined to the right value (the value on the last iteration), preserving behavior if the loop is executed, and not crashing if the loop isn't executed.

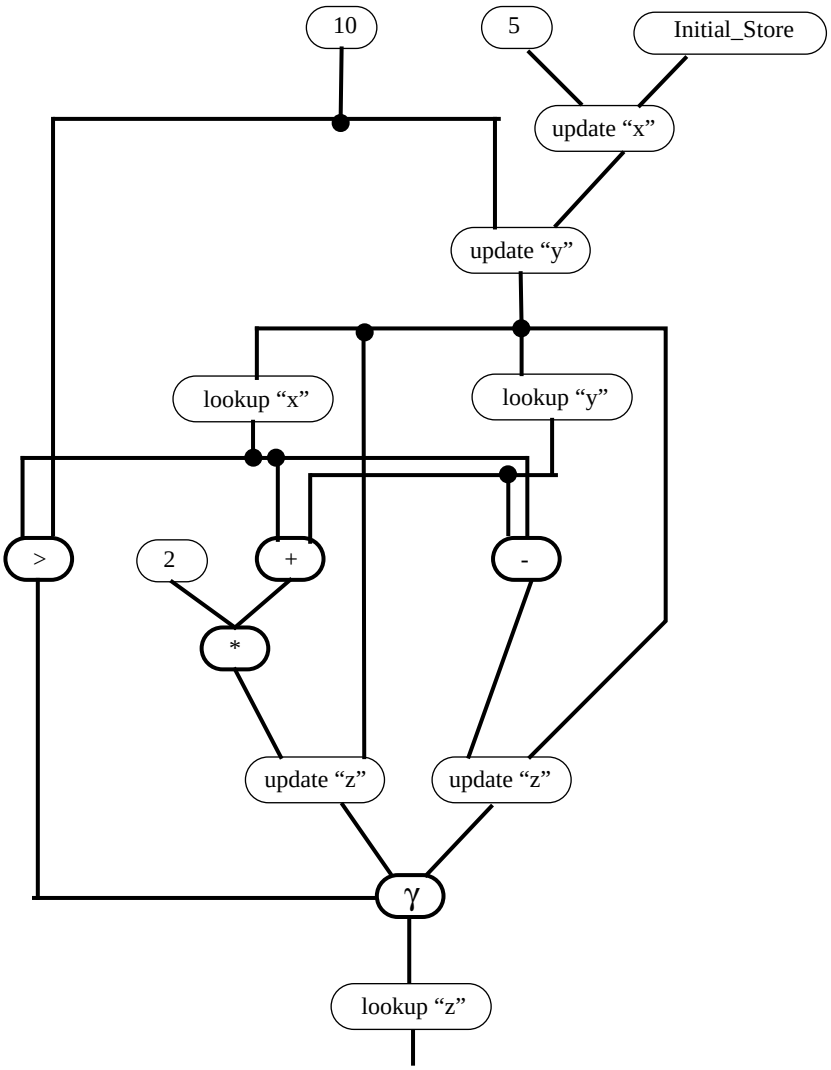
c. Show the CDG + DFG that results after code motion.



5. These questions are about the VDG representation.

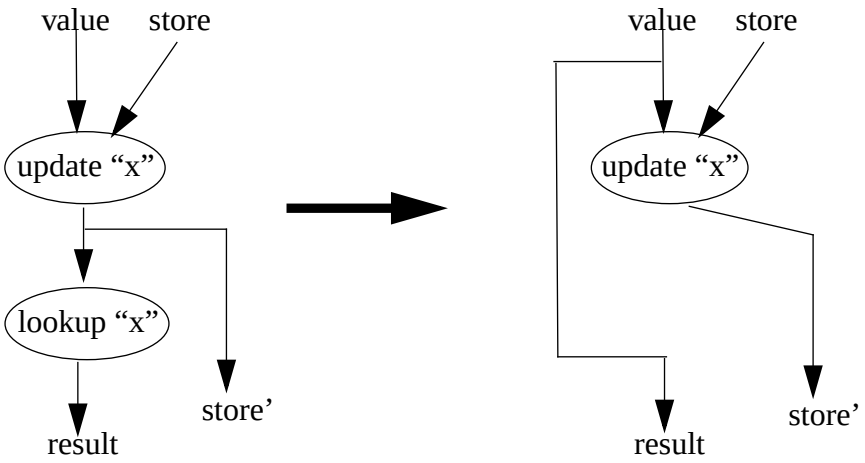
a. Construct the VDG for the following program fragment, assuming that all variables are accessed through a single store. The VDG should take an empty store as input and demand the value of `z` at output (the store is not demanded at output). Each assignment to a variable should be modeled as an update of the store (producing a new store), and each read of a variable should be modeled as a lookup in the store.

```
x := 5;
y := 10;
if x > 10 then
   z := 2 * (x + y);
else
   z := y - x;
endif
```

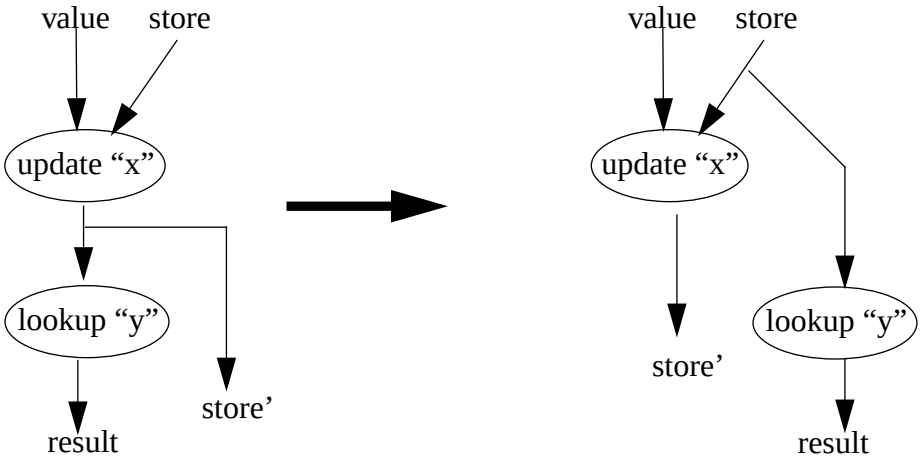


b. Store splitting removes unnecessary reads of the store by noting when a read of a store is preceded by an earlier write of the same variable to the store, separated only by stores to different (non-aliased) variables. Assuming that no two variables are aliased, describe an inductive rule for detecting when a lookup operator on a particular input store can be simplified. For example, the following graph pattern-matching rewrite rule handles the base case where an assignment to a variable immediately precedes a lookup of the same variable, but it doesn’t handle cases where the assignment to the variable is farther back in the store’s history. You should describe (perhaps graphically) such a more general rewrite

rule.

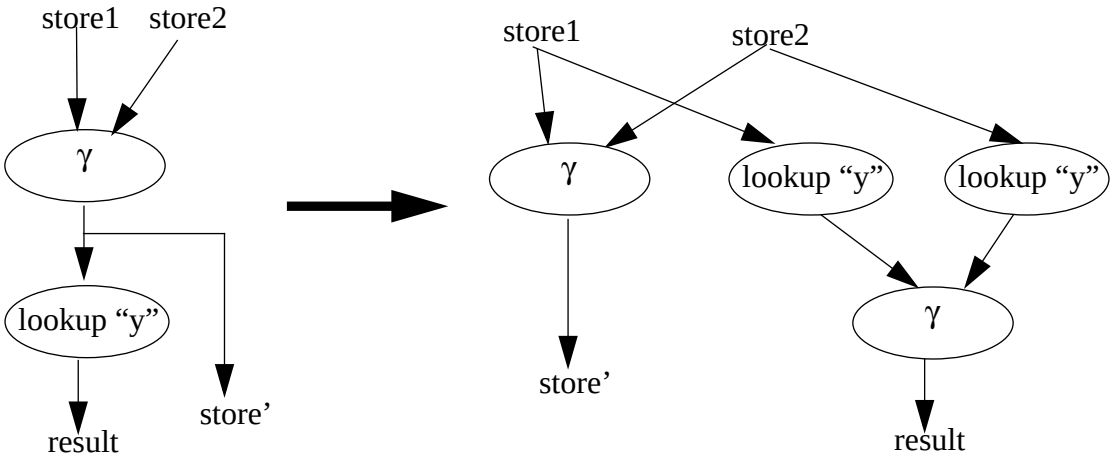


First modify this rule to allow arbitrary update nodes for variables other than “x” between the update of “x” and the lookup of “x.”



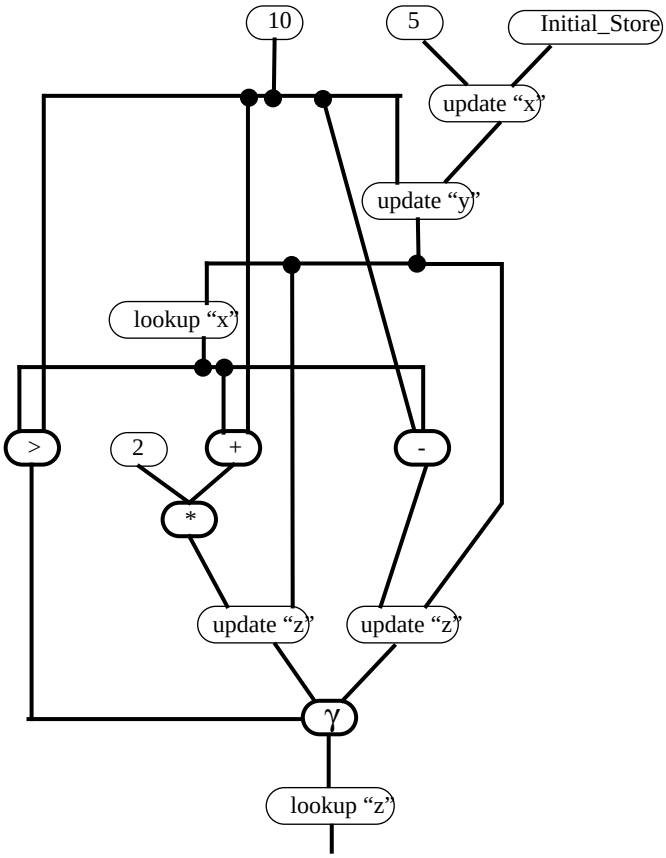
Also extend the rule to distribute lookups over γ nodes, i.e. if given lookup(var, γ(pred, s1, s2)), replace with γ(pred, lookup(var, s1), lookup(var, s2)). This distribution will move lookups

closer to their previous stores.

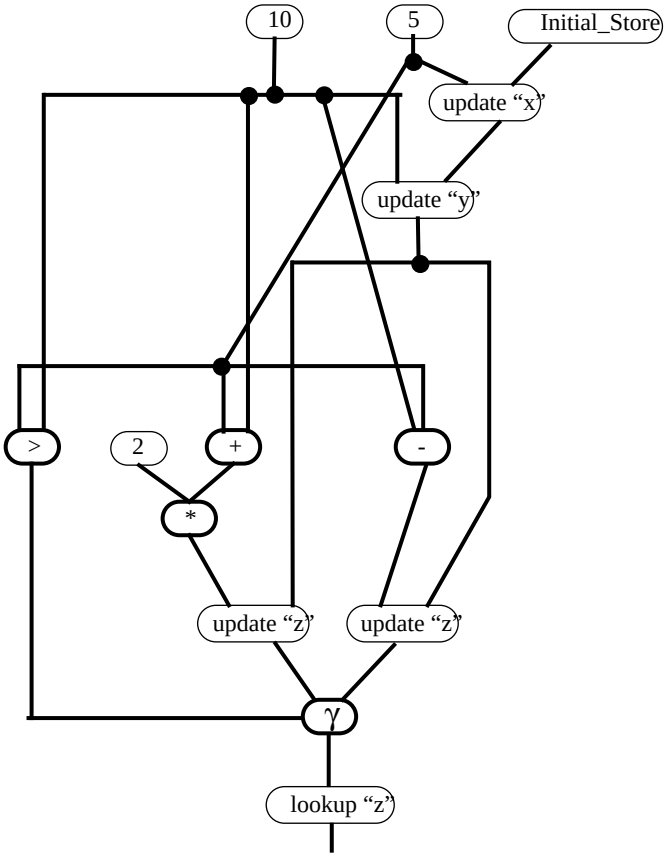


c. Apply your store splitting transformation to your VDG. Be sure to simplify your representation, dropping any undemanded nodes, after the transformation.

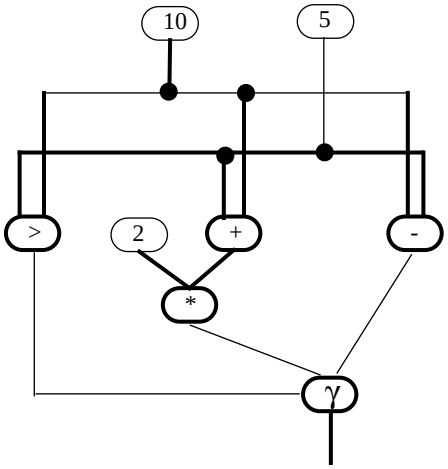
Remove lookup of y.



Remove lookup of x.



Remove lookup of z, after distributing the lookup through the γ node. After the rewrite the final store is no longer demanded, so all update nodes should be removed as dead.



6. None of the common subexpression optimization algorithms presented in class will be able to optimize the `z` calculation of the following simple program, despite its right-hand side having been calculated already on all possible program executions:

```
if (...) {
   ...
   x := a*b;
   ...
} else {
   ...
   y := a*b;
   ...
}
...
z := a*b;
```

a. Explain why none of the algorithms identify the available expression.

*Because the meet function takes the intersection of the available expressions along merging branches, and {a*b → x} ∩ {a*b → y} = {}.*

b. Describe precisely an improved analysis and/or transformation that will enable cases of this general form to be optimized, at least partially. You may rely on a later dead-assignment elimination pass to clean up, if desired, and you may also assume that there is an explicit merge node in your representation for which you give an explicit flow function.

*The flow function for the merge node should attempt to preserve available expressions that would have been lost by the previous merge rule. For example, w.l.o.g., consider a merge node with 2 predecessors, each of which contains a mapping for an expression E, but to different variables, v1 and v2: predecessor1 = { ..., E → v1, ... } and predecessor2 = { ..., E → v2, ... }. In this case we will transform the program by allocating a fresh variable t, adding an assignment t := v1 along predecessor 1, and t := v2 along predecessor 2. The merge node's flow function will simulate the effect of these additional assignments by adding E → t bindings to both predecessors' infos. Then the merge node will take the normal meet and analysis will proceed. When analyzing the later z := a*b expression, a*b→t will be found in the table, and enable the z assignment to be optimized. If it turns out that no uses of the available expression E were generated, i.e., if t has no uses, then dead assignment elimination will clean up by removing the t assignments.*