Homework Assignment #2

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?

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?

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);
```

   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.

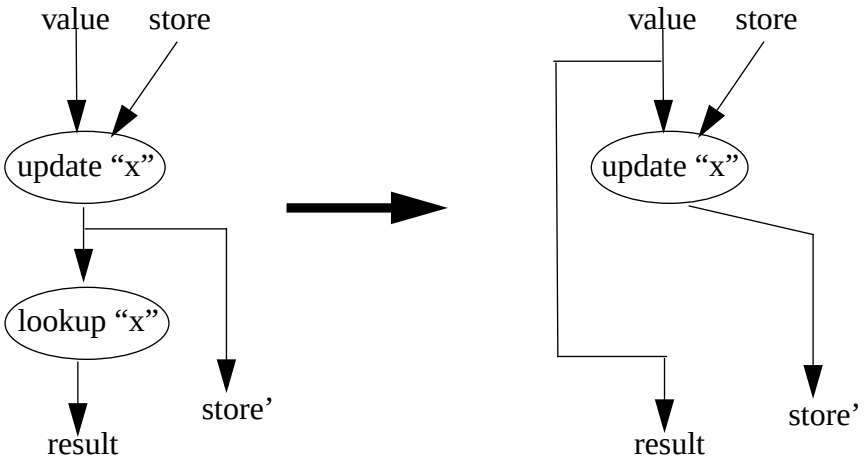
c. Show the CDG + DFG that results after code motion.

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 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.



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

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.

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.