CS243 Homework 3

Winter 2017

Due: Feb 7, 2017

Directions:

- Complete the following problems and hand them in at the beginning of class with your **name and SUID at the top**. Please do not forget your SUID.
- Remember to complete the corresponding Gradiance quizzes by the start of class on the due date. **There are no late days for Gradiance.**
- If you need to use one late day, hand in your assignment by 3pm Wednesday in the assignment drop-box (not the unlocked handout box) on the first floor of Gates. If you need two days, drop it in the drop-box by 2:50pm or hand it by 3pm at the beginning of class on Thursday.

Problem 1. Lock Analysis

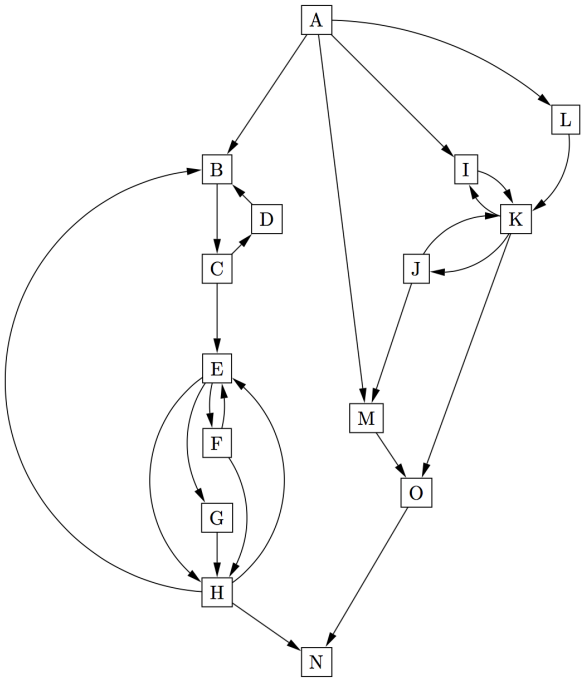
Define a compiler analysis that generates warnings to help programmers detect errors related to the use of locks in the program. There is one lock per variable, which all start unlocked, and two statements: LOCK(v) which takes the lock on variable v, UNLOCK(v) which releases the lock on variable v. Your analysis must allow the compiler to generate the following warnings:

> Warning I: Issued on a LOCK(v) operation if it potentially follows another LOCK(v) operation.
>
> Warning II: Issued if the program may terminate with a lock in the locked state.
>
> Warning III: Issued on an UNLOCK(v) operation that potentially does not follow a LOCK(v) operation.

1. Specify a dataflow analysis that may be used to generate the necessary warnings.
2. Why will your algorithm converge?
3. Is the framework monotone?
4. Is the framework distributive?
5. Describe how to generate the three possible warnings after running your analysis.

Problem 2. Dominators and Natural Loops

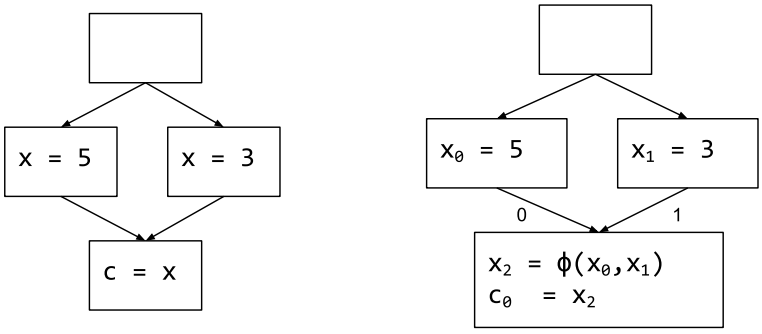
Consider the following graph.



1. Draw the dominator tree for the graph, assuming A is the entry node.
2. Find the back edges and their natural loops in the graph.

Problem 3. Static Single Assignment Form (SSA)

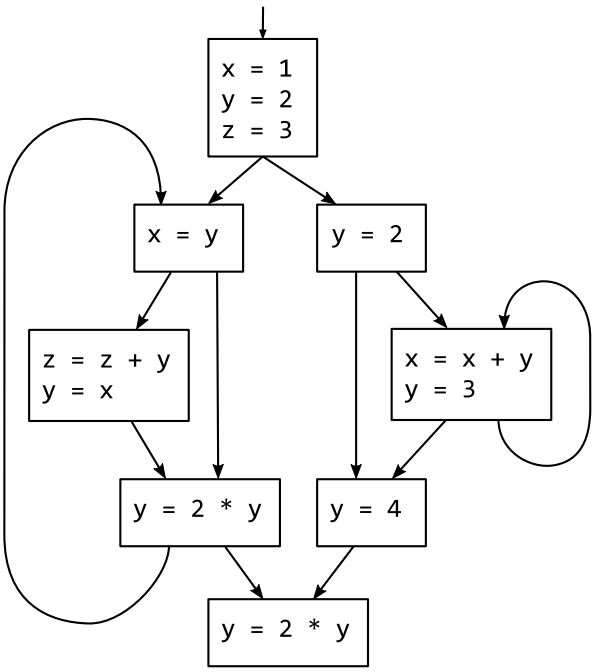
Static single assignment is popular in modern compilers including LLVM. This representation separates multiple definitions of a single variable by numbering each definition and introducing the concept of Φ-nodes to resolve control flow ambiguity. Each use of a variable is rewritten to have a suffix indicating which definition reaches that point. In general, multiple definitions of a variable can all reach a given program point, so there is no unique suffixed variable to use. Phi nodes are special functions at meet points which select from their arguments according to which incoming edge is taken to reach a block, and assign to a new numbered definition of that variable. An example of SSA form is given below:



Here we can see that the use `c = x` can come from either branch, so we need a phi node. The phi nodes always occur at the top of a block with multiple predecessors, and each has a number of arguments exactly equal to the number of incoming edges. We label the edges $0, 1, \ldots$ to indicate which edge corresponds to which phi argument.

There are multiple ways to represent a program in SSA form which vary in how many redundant phi nodes have been inserted. One algorithm for constructing "minimal SSA" places phi nodes for a given variable at the *dominance frontier*[1] of each definition of that variable. These phi nodes are themselves definitions, and may have a dominance frontier which includes more nodes. The algorithm repeatedly adds phi nodes on the dominance frontier until no more are generated for any variable.

a. Using this algorithm, construct the minimal SSA form of the following program. Remember to label incoming edges with their index in the phi node(s).

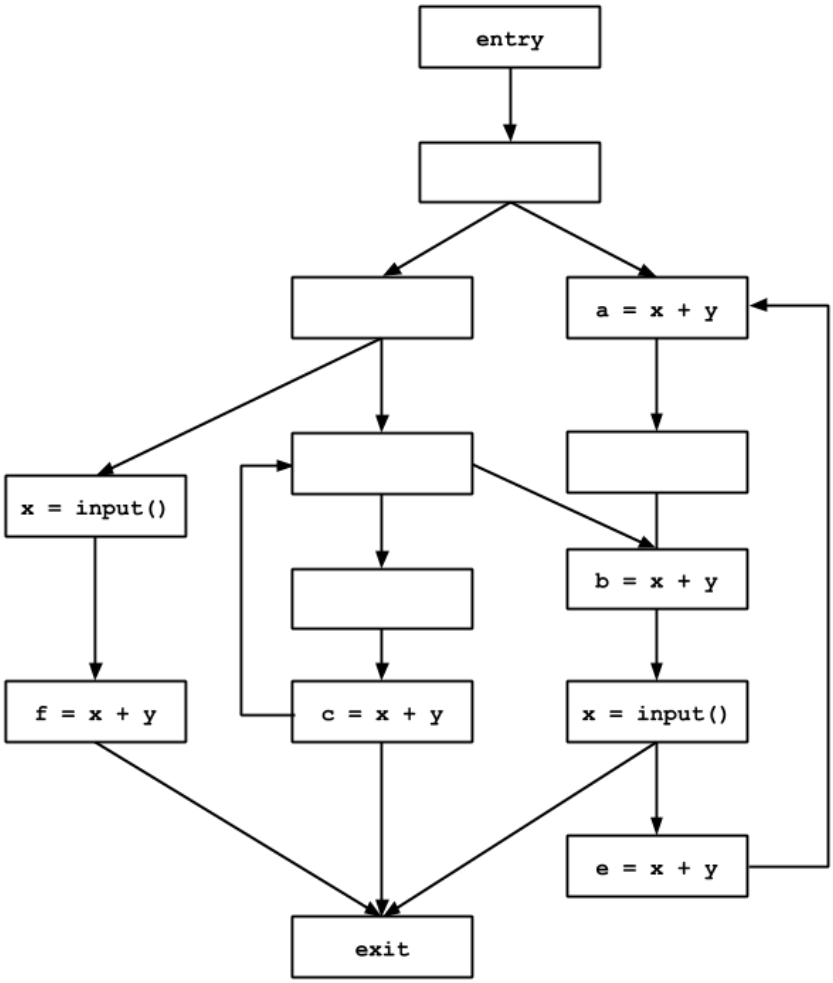


b. Does the given algorithm insert any phi nodes which are not necessary to ensure each use of a variable has exactly one reaching definition? If so, which phi nodes are redundant? If not, will this algorithm add redundant phi nodes for any control flow graphs?

[1] The dominance frontier of a node b is the set of all nodes which have a predecessor dominated by b but are not strictly dominated by b.

Problem 4. Partial Redundancy Elimination

Apply lazy code motion to the following program. You do not need to show the intermediate steps, just show the optimized code. You may add basic blocks to the flow graph, but only show those that are not empty in your solution.



Problem 5. Register Allocation

For the following control flow graph, perform register allocation. Assign each definition and use of a variable to a live range, and draw the resulting flow graph. Draw the interference graph and the assignment of registers (i.e. colors) to live ranges.