Name: ____________________________

CIS 341 Final Examination
10 December 2008

1	/8
2	/12
3	/18
4	/18
5	/14
Total	/70

- Do not begin the exam until you are told to do so.
- You have 120 minutes to complete the exam.
- There are 11 pages in this exam.
- Make sure your name is on the top of this page.

1. True or False? (8 points)

a. T $\boxed{\text{F}}$ Control may enter a basic block at more than one location.

b. $\boxed{\text{T}}$ F Abstract data types ensure that *clients* of a module cannot violate representation invariants by hiding the representation type from the client.

c. $\boxed{\text{T}}$ F In C++ implementations, multiple inheritance requires that some type cast operations have an effect at runtime.

d. T $\boxed{\text{F}}$ An example of *strength reduction* optimization is replacing `x << 4` by `x * 16`.

e. $\boxed{\text{T}}$ F It is possible to implement graph-coloring register allocation in such a way that only one attempt at graph-coloring is needed.

f. T $\boxed{\text{F}}$ The *roots* of garbage collection include pointers in registers and every live pointer in the heap.

g. $\boxed{\text{T}}$ F The Cheney copying garbage collection algorithm compacts heap objects as it works, thereby enabling an implementation of `malloc` that is as cheap as just incrementing a pointer.

h. T $\boxed{\text{F}}$ The Deutsch-Schorr-Waite in-place implementation of depth-first traversal is suitable for implementing incremental or concurrent garbage collectors.

2. Object-oriented Language Implementation (12 points)

Consider this Java code we saw in class:

```
interface Shape {
  void setCorner(int w, Point p);
}
interface Color {
  float get(int rgb);
  void  set(int rgb, float value);
}
class Blob implements Shape, Color {
  void setCorner(int w, Point p) {...}
  float get(int rgb) {...}
  void  set(int rgb, float value) {...}
}
```

a. (2 points) Which of the following two loops would you expect to have better performance in an implementation of Java that supports separate compilation? In one sentence, describe why.

```
1.   Color c = new Blob();
     for(int i = 0; i<10000; i++) {
       c.set(3, 1.0);
     }

2.   Blob b = new Blob();
     for(int i = 0; i<10000; i++) {
       b.set(3, 1.0);
     }
```

2 would be faster because dispatch through an interface is more expensive because it uses an extra level of indirection of some form (either a map or a hash table).

b. (6 points) Pick an implementation strategy for multiple inheritance. Draw the corresponding memory layout for an object of type `Blob`, including the object record and dispatch vector(s). Briefly(!) describe the implementation strategy.

c. (4 points) For the implementation strategy you picked in part **b**, (briefly!) describe one benefit and one drawback to the approach.

3. Data-flow Analysis (18 points)

In this problem, we'll explore how data-flow analysis can be used for purposes other than optimization. As one example, data-flow analysis can help detect bugs in concurrent programs by determining whether locks have been acquired appropriately. Here, we simplify the model and assume that there is a single, global lock that might be used in the program.

Consider a cut-down version of the language of quadruples seen in class—it doesn't have memory read/write operations and it omits function calls. This language has two new primitive operations: `lock` (which acquires the unique global lock) and `unlock` (which releases it). The quadruples that make up the language are given below:

`a = b op c`	Basic arithmetic
`jump L`	Jump to label L
`if a then` L_1 `else` L_2	Conditional branch
`lock`	Acquire the unique global lock
`unlock`	Release the unique global lock
`return a`	Exit the procedure

The purpose of the data-flow analysis is to determine, for each edge of the control flow graph, whether the lock is *locked* (meaning that the program has acquired the lock exactly once along the path prior to the program point), whether the lock is *unlocked* (meaning that the lock has not been acquired yet or has been previously released appropriately), or whether the lock is in an *error* state (meaning that the lock is acquired more than once along the path without an intervening `unlock` command). It is OK to use `unlock` when the lock is not held (so two `unlock` commands in sequence are not an error).

Recall the generic frameworks for forward iterative data-flow analysis that we discussed in class. Here, `n` ranges over the nodes of the control-flow graph (which are assumed to be single quadruple instructions), `pred[n]` is the set of predecessor nodes, F_n is the *flow function* for the node `n`, and $\sqcap$ is the *meet* combining operator.

```
for all nodes n: in[n] = $\top$, out[n] = $\top$;
repeat until no change {
  for all nodes n:
    in[n]  := $\bigsqcap_{n' \in pred[n]}$ out[n'];
    out[n] := $F_n$(in[n]);
}
```

Recall that the flow functions F_n and the $\sqcap$ operator work over elements of a *lattice* $\mathcal{L}$. The lattice for our lock analysis is just:

$$\begin{tikzcd}
U \arrow[d, dash] & = \text{unlocked} \\
L \arrow[d, dash] & = \text{locked} \\
E & = \text{error}
\end{tikzcd}$$

a. (3 points) The lattice element U is the *top* element ($\top$). How do we interpret the $\sqcap$ operation? Complete the following table (each entry should be a lattice element computing the $\sqcap$ of the row and column):

$\sqcap$	U	L	E
U	U		
L		L	
E			E

b. (6 points) How do we implement the following flow functions to achieve the goals of the analysis described above?

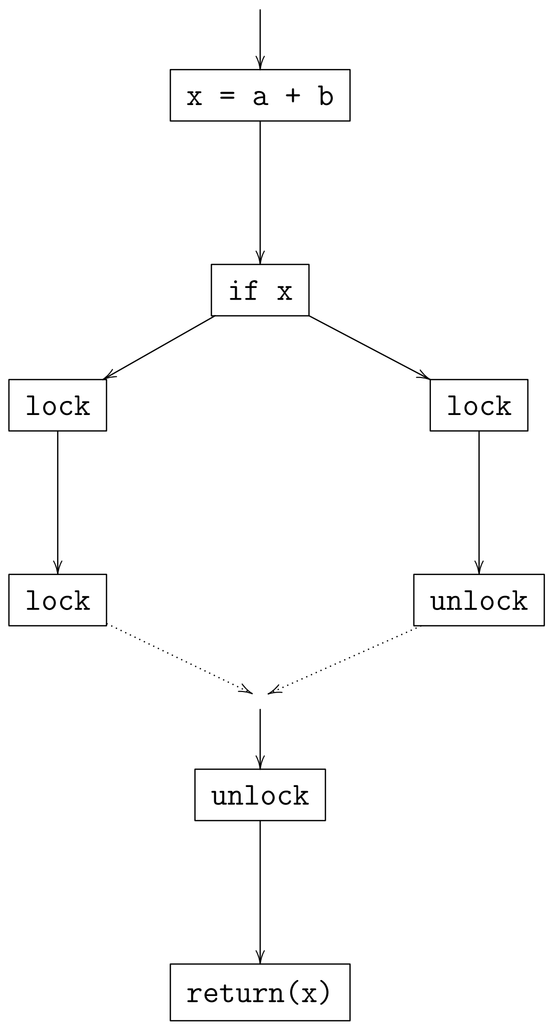
- $\texttt{F}_{\texttt{(a = b op c)}}(\ell) = \ell$

- $\texttt{F}_{\texttt{lock}}(\ell) =$ if $\ell = U$ then L else E

- $\texttt{F}_{\texttt{unlock}}(\ell) =$ if $\ell = E$ then E else U

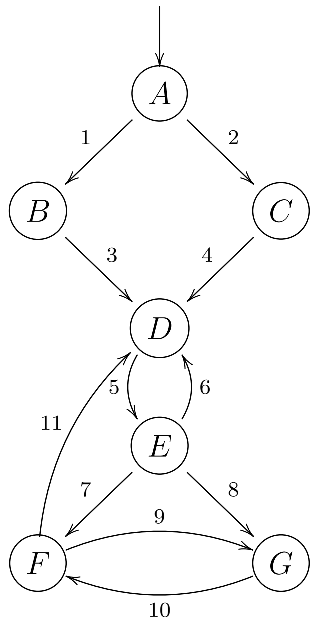
c. (4 points) Assuming that all of the other flow functions not mentioned in part **b** are just the identity: $\texttt{F}_n(\ell) = \ell$, does your solution have the *meet over paths* property? Why or why not?

d. (5 points) Consider the following control-flow graph in which the "in" edges are solid and the "out" edges are dotted. Label each edge with the lattice element (E, L, or U) that should be computed for the edge once a fixpoint is reached by a correct implementation of the lock-analysis.

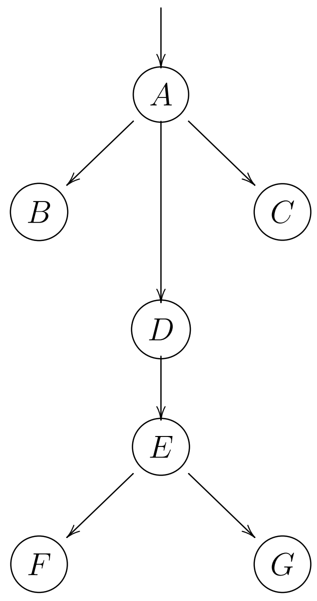


4. Control-flow Analysis (18 points)

Consider the following control-flow graph $\mathbb{G}$ with nodes labeled A through G and edges labeled 1 through 11. Node A is the entry point.



a. (6 points) Draw the dominance tree for the control-flow graph $\mathbb{G}$, making sure to label nodes appropriately (there is no need to label the edges).



b. (6 points) For each *back edge* e of $\mathbb{G}$, identify the set of nodes appearing e's *natural loop*. Each answer should be of the form "e, $\{nodes\}$" where e is a back edge and $nodes$ is the set of nodes that make up the loop.

6, {D, E} and 11, {D, E, F, G}

c. (3 points) Which nodes are in the dominance frontier of the node B?

{D}

d. (3 points) Which nodes are in the dominance frontier of the node E?

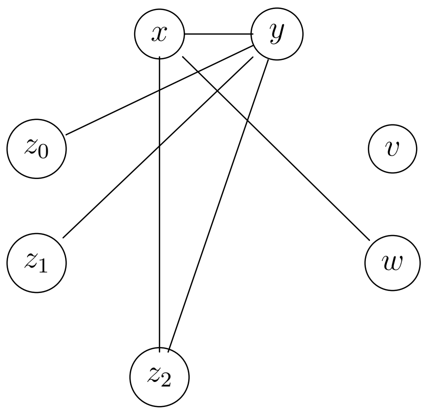
{D,E }

5. Optimization (14 points)

Consider the following program that has already been put into SSA form.

```
lbl₁:
  x = 0;
  y = x + 1;
  if (y = 3) lbl₂ else lbl₃
lbl₂:
  z₀ = x + 2;
  jump lbl₄;
lbl₃:
  z₁ = x + y;
  jump lbl₄;
lbl₄:
  z₂ = φ(z₀, z₁);
  w = z₂ + y;
  v = w + x;
  return(v);
```

a. (6 points) Complete the interference graph generated from this program for graph-coloring register allocation. Assume that there are no precolored registers, so you *do not* have to include EAX, etc. There are seven nodes—one node for each temporary variable mentioned.



b. (4 points) Give an assignment of temporary variables to colors $\{C_0, C_1, C_2, C_3, \ldots\}$ that is a minimal coloring of your graph from part **a**. Assume you have as many colors as you like, so no spilling is necessary. Assign colors in such a way as to make the resulting register-assigned code as optimal as possible (i.e. eliminate as many moves as you can).

Temporary	Color
x	C_0
y	C_1
z_0	C_2
z_1	C_2
z_2	C_2
v	C_0
w	C_1

c. (4 points) Suppose you implemented *all* of the optimizations we discussed in class (not just register allocation) and iterated them as many times as necessary to generate the best code possible from the above input. What would be the resulting optimized program?

```
return(2)
```