CIS 341 Final Examination
3 May 2011

1	/16
2	/20
3	/40
4	/28
5	/16
Total	/120

- Do not begin the exam until you are told to do so.
- You have 120 minutes to complete the exam.
- There are 12 pages in this exam.
- Please acknowledge the following statement:

My signature below certifies that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

____

Name (printed)

____

Signature / Date

1. True or False? (16 points)

a. T F The subtyping relation between mutable structures such as arrays or ML-style references should be *invariant*, meaning that if `Array<T> <: Array<S>` then `T = S` (although Java gets this wrong and compensates via dynamic checks).

b. T F Optimizing by *inlining functions* is guaranteed to produce a performance improvement because it eliminates the overhead (allocating a stack frame, copying arguments, etc.) of doing a function all.

c. T F An example of *constant folding* is replacing the expression `300 + 41` by `341`.

d. T F Consider optimizing the following program, where `a` is an `int array` and `b`, `i`, `j`, and `t` are `int`s:

```
a[j] = a[i] + 1;
b = a[i];
```

It can always safely be rewritten to:

```
t = a[i];
a[j] = t + 1;
b = t;
```

e. T F Because C exposes low-level pointers, if you want to use a garbage collector, you must use a conservative copying collector.

f. T F One beneficial side effect of Cheney's garbage collection algorithm is that it compacts the heap, thereby eliminating fragmentation.

g. T F The Low Level Virtual Machine (LLVM) uses a static single assignment intermediate representation, which facilitates many state-of-the-art optimizations.

h. T F To support *abstract datatypes*, in which clients of a module cannot violate the abstraction, it is necessary to have a type safe language.

2. Object-oriented Language Implementation (20 points)

a. (8 points) OAT, like most other object-oriented languages, uses constructors to initialize the object's dispatch vector and fields. As we saw in the class projects, the memory associated with the newly created object is usually allocated in the `new` statement and a pointer to the fresh space is passed to the class's constructor for initialization. Briefly (in one or two sentences) explain what would change if we tried to do the `malloc` inside the constructor instead of at the `new`. Would making such a change be a good idea?

b. (12 points) Consider the following class declarations in a variant of OAT whose `Object` class has no fields or methods.

```
class A <: Object {                    class B <: A {
  int a;                                 int b;
  new ()() {this.a = 0;}                 new ()(){this.b = 1;}
  int f() { code$_1$ }                     int f() { code$_3$ }
  int g() { code$_2$ }                     int h() { code$_4$ }
}                                      }
```

Assuming the simple dispatch table scheme with sequentially assigned method indices, draw the state of memory after executing the following program:

```
A x = new A();
B y = new B();
```

Include the objects themselves, their dispatch tables and indicate the code pointers of the the tables for all methods. (You do not need to show the constructor code.)

3. Data-flow Analysis (40 points)

Warning: read all of this problem before tackling the earlier questions — the later parts give you clues to the earlier ones.

In this problem, we'll explore a data-flow analysis that could help eliminate array-bounds checks. The idea is to compute conservative *intervals* bounding the values of all integer variables. For example, if the analysis determines that a variable x is statically approximated by the interval $[0, 9]$, then at run-time, x might take on any value in the range $0 \dots 9$ (inclusive).

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, pred[n] is the set of predecessor nodes, $\mathtt{F_n}$ 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]);
}
```

a. (6 points) Recall that the flow functions $\mathtt{F_n}$ and the $\sqcap$ operator work over elements of a *lattice* $\mathcal{L}$. To create a suitable lattice for interval analysis, we extend the set $\mathbb{Z}$ of integers with plus and minus infinity: $\mathbb{Z}^* = \mathbb{Z} \cup \{\infty, -\infty\}$, such that $-\infty < n$ and $n < \infty$ for any integer n.

We define the lattice of intervals by $\mathcal{L} = \{[a, b] \mid a, b \in \mathbb{Z}^* \wedge a \leq b\} \cup \{\top\}$ ordered such that $\forall \ell \in \mathcal{L}.\ \ell \sqsubseteq \top$ and $[a, b] \sqsubseteq [c, d] \iff a \leq c \wedge d \leq b$. Here the element $\top$ indicates an "impossible" interval (see part **d** to see why).

Give the definition for the $\sqcap$ operation on this lattice. Taking into account symmetry, here are these cases you need to complete:

$$\ell \sqcap \top = \top \sqcap \ell \ =$$

$$[a, b] \sqcap [c, d] \ =$$

b. (4 points) Unfortunately, the lattice defined in part **a** will not work for general program analysis. In one sentence, explain why. Hint: consider the the interval computed for x by iteratively analyzing the following program

```
x = 0;
while ( 0 < 1 ) { x = x + 1; }
```

c. (6 points) To fix the problem identified in part **b**, we define a new "clipped" lattice $\mathcal{L}_k$, where $k \in \mathbb{Z}^+$ is a clipping parameter, by:

$$\mathcal{L}_k = \{[a,b] \mid a,b \in \{-\infty, -k, \ldots, -1, 0, 1, \ldots, k, \infty\} \wedge a \leq b\} \cup \{\top\}$$

This lattice has the same ordering as $\mathcal{L}$, but computes precise ranges only between $-k$ and k. It is helpful to define a "clipping" operator that takes an arbitrary $m \in \mathbb{Z}^*$ and approximates it with the available precision:

$$\lfloor m \rfloor = \begin{cases} -\infty & \text{if } m < -k \\ m & \text{if } -k \leq m \leq k \\ \infty & \text{if} m > k \end{cases}$$

We can now define the flow functions for each program statement. As usual, because we want to compute intervals for each program variable, we treat the lattice elements ℓ as finite maps from program variables to elements of $\mathcal{L}_k$. For example, $\ell(\mathtt{x}) = [2,4]$ means that the approximation for x in ℓ is the interval $[2,4]$. The notation $\ell(\mathtt{x}) \mapsto [a,b]$ means "update the value of x in ℓ to be the interval $[a,b]$, but leave the mappings for other variables alone".

Complete the following flow functions to achieve the goals of the analysis described above. Here, x, y, and z are program variables and $n \in \mathbb{Z}$ is an integer. The second case uses ML-like syntax to do case analysis on the two possible kinds of lattice elements. Hint: you should use the $\lfloor - \rfloor$ function defined above and be sure to treat $\top$ properly.

- $\mathtt{F}_{(\mathtt{x} \; \mathtt{=} \; n)}(\ell) =$
  $\ell(\mathtt{x}) \mapsto$

- $\mathtt{F}_{(\mathtt{x} \; \mathtt{=} \; \mathtt{y} \; \mathtt{+} \; \mathtt{z})}(\ell) =$
  $\ell(\mathtt{x}) \mapsto$ $\mathtt{match}\ \ell(\mathtt{y})\ \mathtt{with}$
  $\mid \top \rightarrow \top$
  $\mid [a,b] \rightarrow$

d. (8 points) For this kind of analysis, it is more precise to associate *different* information (i.e. lattice elements) to the two out-edges of conditional statements. For example, suppose that $\ell(\mathtt{x}) = [1, 5]$ and we execute the test `if (x < 3) then lbl`$_1$ `else lbl`$_2$. In the "then" branch (at $\mathtt{lbl}_1$), we can refine the approximation to: $\ell(\mathtt{x}) \mapsto [1, 2]$ and in the "else" branch (at $\mathtt{lbl}_2$), we can refine it to: $\ell(\mathtt{x}) \mapsto [3, 5]$ — the test narrows down the set of possible values for `x`.

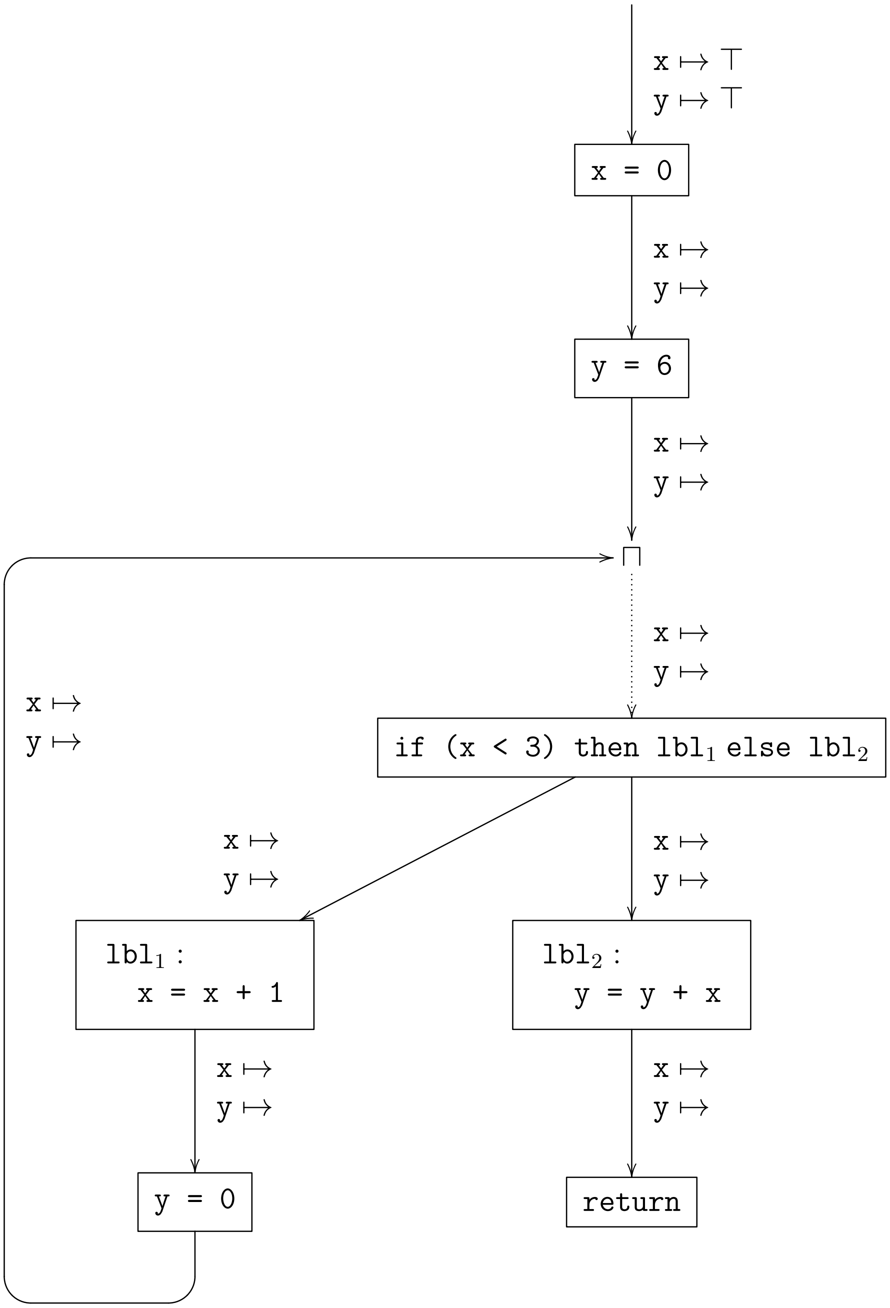
Complete the following transfer function for such conditional tests, where n is an integer constant:

$$\mathtt{F}_{(\mathtt{if\ (x\ <\ }n\mathtt{)\ then\ lbl}_1\ \mathtt{else\ lbl}_2)}(\ell) =$$

$$\begin{aligned}
\mathtt{lbl}_1 :\ \ell(\mathtt{x}) &\mapsto \mathtt{match}\ \ell(\mathtt{x})\ \mathtt{with} \\
&\quad \mid\ \top \to \top \\
&\quad \mid\ [a, b] \to
\end{aligned}$$

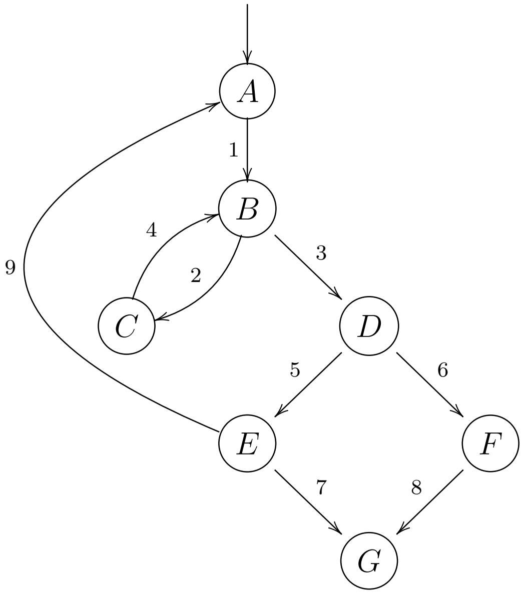
$$\begin{aligned}
\mathtt{lbl}_2 :\ \ell(\mathtt{x}) &\mapsto \mathtt{match}\ \ell(\mathtt{x})\ \mathtt{with} \\
&\quad \mid\ \top \to \top \\
&\quad \mid\ [a, b] \to
\end{aligned}$$

e. (16 points) Consider the following control-flow graph. Label each variable with the lattice element of $\mathcal{L}_5$ that should be computed for the edge once a fixpoint is reached by a correct implementation of the interval analysis assuming the "clipping range" $k = 5$. To help you identify where the only non-trivial use of ⊓ is needed, it is marked as a node in the graph, and the resulting "in" edge is dotted (in all other cases, `in[n] = out[pred[n]]` because the nodes have only one predecessor).



4. **Control-flow Analysis** (28 points)

Consider the following control-flow graph $\mathbb{G}$ with nodes labeled A through G and edges labeled 1 through 9. Node A is the entry point.



a. (10 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. (8 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.

c. (5 points) Which nodes are in the dominance frontier of the node D?

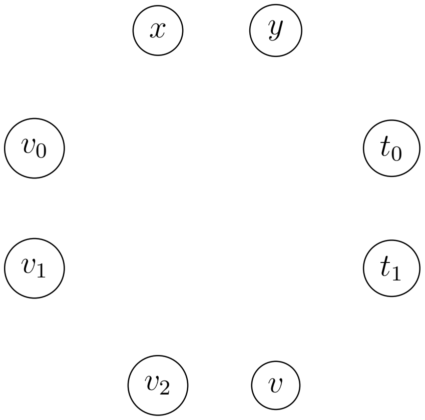
d. (5 points) Which nodes are in the dominance frontier of the node E?

5. Register Allocation (16 points)

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

```
lbl1:
  x = 3;
  y = 10;
  v0 = x + y;
  t0 = x + 1;
  jump lbl2;
lbl2:
  v1 = φ(v0, v2);
  t1 = x * v1;
  v2 = t1 + t0;
  if (v2 < 1000) lbl2 else lbl3;
lbl3:
  v = v2 + 341;
  return(v);
```

a. (8 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. You *do not* need to show move-related edges, only the interference edges.



b. (8 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, but use as few colors as possible. 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	
y	
t_0	
t_1	
v_0	
v_1	
v_2	
v	