CIS 341 Final Examination
30 April 2013

1	/40
2	/30
3	/30
4	/20
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

_____

Pennkey (i.e. user name)

1. **Short Answer** (40 points)

   a. Java's type system incorrectly treats arrays as *covariant*. Recall that that means Java uses the following (broken!) subtyping rule:

$$\frac{\vdash T <: S}{\vdash T\texttt{[]} <: S\texttt{[]}}$$

   (i) (4 points) Indicate which line of the program below would require the use of this (bogus) typechecking rule by the Java typechecker:

```
class S { }
class T extends S {
    public int x;
    public static void main(String[] args) {
        T[] ts = new T[3];
        S[] ss = ts;
        ss[0] = new S();
        int y = ts[0].x;
    }
}
```

   (ii) (8 points) Java "corrects" this problem with its type system by dynamically checking the type of an object when it is stored into an array. Briefly discuss the code performance implications of this decision.

b. (8 points) Consider optimizing the following program, where `a` is an `int array` and `b`, `i`, `j`, and `t` are `int`s:

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

Under what circumstances is it safe to optimize the above program to:

```
t = a[i];
a[j+1] = t << 1;       /* recall that << is arithmetic left shift */
b = t;
```

Assume that the programming language performs array-bounds checks and raises a Java-style catchable exception if the index is out of bounds.

Answer: This optimization is safe only when `i != j+1`. Moreover, the array bounds check must happen after the right-hand-side of the assignment has been evaluated to a value. (Otherwise rearranging the order of evaluation might cause a different exception to be thrown in the case that `a[i]` and `a[j+1]` both are out of bounds.)

c. (8 points) Briefly (very briefly!) explain what parts of the Oat compiler would need to be modified, and in what way, to implement Java-style `public` and `private` keywords that specify field and method access restrictions.

- Lexer: to add the new keywords
- Parser: to modify the grammar
- Typechecker: to modify the scoping rules for fields

d. (12 points) Consider the following legal Oat code:

```
class A <: Object {
  int a;
  new ()() { this.a = 0; }
  int f()  { return this.a; }
  int g()  { return 0; }
};
```

```
class B <: A {
 int b;
 new ()() { this.b = 1; }
 int f()  { return this.b; }
 int h()  { return 341; }
};
```

The Oat project compiler generates six LLVM structure types from the above classes: three structures that describe the object layouts (`%A`, `%B`, and `%Object`), and three structures that describe the vtables (`%_A_vtable`, `%_B_vtable`, and `%_Object_vtable`). When renamed (for the purposes of this question), these six structures' definitions are:

```
%U = type { %W*, i8*, i32, i32 }
%V = type { %Y*, i8*, i32 }
%W = type { %Y*, i8* (%X*)*, i32 (%U*)*, i32 (%V*)*, i32 (%U*)* }
%X = type { %Z*, i8* }
%Y = type { %Z*, i8* (%X*)*, i32 (%V*)*, i32 (%V*)* }
%Z = type { {  }*, i8* (%X*)* }
```

Match the anonymized types to their actual names by filling each of the following blanks with one of `%A`, `%B`, `%Object`, `%_A_vtable`, `%_B_vtable`, or `%_Object_vtable`:

`%U` = `%B`

`%V` = `%A`

`%W` = `%_B_vtable`

`%X` = `%Object`

`%Y` = `%_A_vtable`

`%Z` = `%_Object_vtable`

2. Data-flow Analysis (30 points)

Warning: read all of this problem before tackling the earlier parts—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 $0 \ldots 9$ (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, $\texttt{F}_\texttt{n}$ is the *flow function* for the node `n`, and $\sqcap$ is the *meet* combining operator.

```
for all nodes n: in[n] = ⊤, out[n] = ⊤;
repeat until no change {
  for all nodes n:
    in[n]  := ⊓_{n' ∈ pred[n]} out[n'];
    out[n] := F_n(in[n]);
}
```

a. (4 points) Recall that the flow functions $\texttt{F}_\texttt{n}$ 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:

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

$$[a, b] \sqcap [c, d] \ = \ [min(a, c), max(b, d)]$$

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; }
```

The lattice has infinite height, so the analysis might diverge.

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.

- $\mathtt{F}_{(\mathtt{x = } n)}(\ell) =$
  $\ell(\mathtt{x}) \mapsto [\lfloor n \rfloor, \lfloor n \rfloor]$

- $\mathtt{F}_{(\mathtt{x = y + z})}(\ell) =$
  $\ell(\mathtt{x}) \mapsto \mathtt{match}\ \ell(\mathtt{y})\ \mathtt{with}$
  $\quad \mid \top \to \top$
  $\quad \mid [a,b] \to \mathtt{match}\ \ell(\mathtt{z})\ \mathtt{with}$
  $\qquad \mid \top \to \top$
  $\qquad \mid [c,d] \to [\lfloor a + c \rfloor, \lfloor b + d \rfloor]$

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 $\ell(\texttt{x}) = [1, 5]$ and we execute the test `if (x < 3) then` $\texttt{lbl}_1$ `else` $\texttt{lbl}_2$. In the "then" branch (at $\texttt{lbl}_1$), we can refine the approximation to: $\ell(\texttt{x}) \mapsto [1, 2]$ and in the "else" branch (at $\texttt{lbl}_2$), we can refine it to: $\ell(\texttt{x}) \mapsto [3, 5]$ — the test narrows down the set of possible values for `x`.

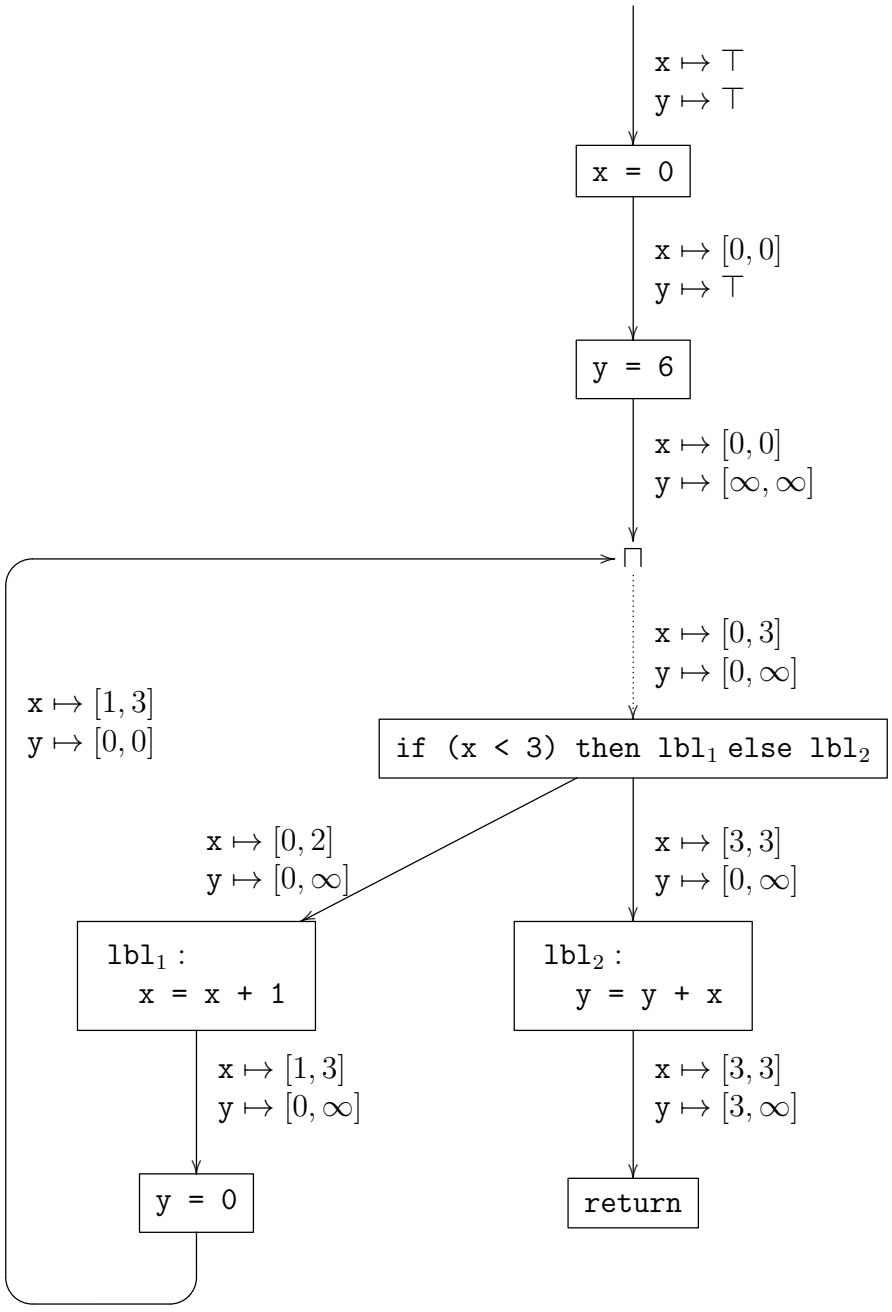
Complete the following transfer function for such conditional tests, where n is an integer constant:

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

$$\begin{array}{l}
\texttt{lbl}_1 : \ \ell(\texttt{x}) \mapsto \texttt{match}\ \ell(\texttt{x})\ \texttt{with} \\
\qquad\qquad \mid\ \top \to \top \\
\\
\qquad\qquad \mid\ [a, b] \to \texttt{if}\ n \leq a\ \texttt{then}\ \top \\
\qquad\qquad\ \ \texttt{else}\ [a, min(b, \lfloor n - 1 \rfloor)]
\end{array}$$

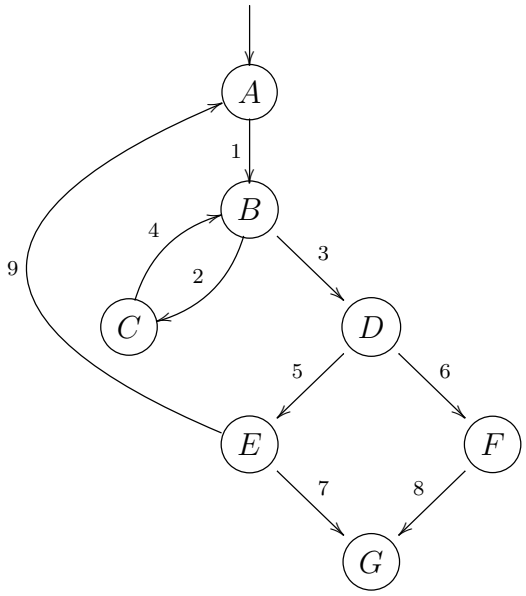
$$\begin{array}{l}
\texttt{lbl}_2 : \ \ell(\texttt{x}) \mapsto \texttt{match}\ \ell(\texttt{x})\ \texttt{with} \\
\qquad\qquad \mid\ \top \to \top \\
\\
\qquad\qquad \mid\ [a, b] \to \texttt{if}\ b < n\ \texttt{then}\ \top \\
\qquad\qquad\ \ \texttt{else}\ [max(a, \lfloor n \rfloor), b]
\end{array}$$

e. (8 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 $\sqcap$ 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).

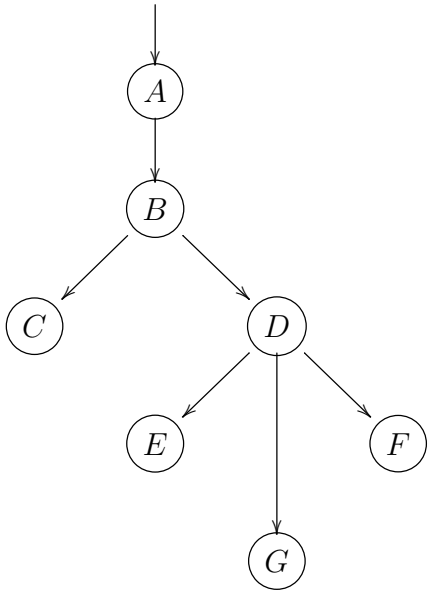


3. Control-flow Analysis (30 points)

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. (10 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.

4, {B, C} and 9, {A, B, C, D, E}

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

{A}

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

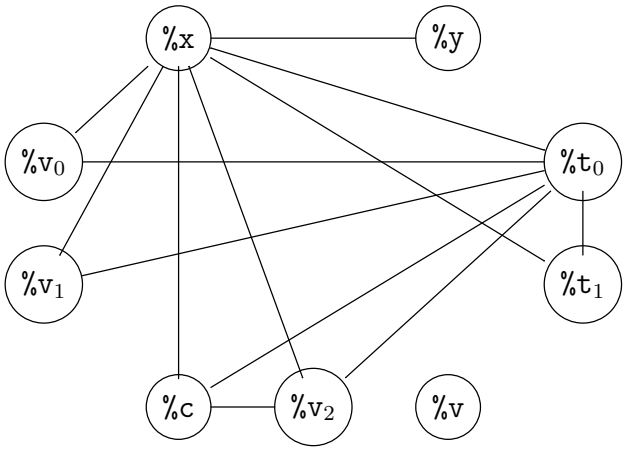
{A,G }

4. Register Allocation (20 points)

Consider the following LLVM IR program:

```
lbl1:
  %x = add i32 3, 4
  %y = add i32 %x, %x
  %v0 = add i32 %x, %y
  %t0 = add i32 %x, 1
  br label lbl2
lbl2:
  %v1 = phi i32 lbl1 %v0, lbl2 %v2
  %t1 = mult i32 %x, %v1
  %v2 = add i32 %t1, %t0
  %c = icmp slt i32 %v2, 1000
  br i1 %c, label lbl2, label lbl3
lbl3:
  %v = add i32 %v2, 341
  return %v
```

a. (11 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. (9 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	C_0
%y	C_1
$\%t_0$	C_2
$\%t_1$	C_1
$\%v_0$	C_1
$\%v_1$	C_1
%c	C_3
$\%v_2$	C_1
%v	C_1