Due Monday, February 12, at the **start** of class. Not after class!

Take in a single, 5-hour period, self-timed. Time begins when you look at the next page.

You may refer to any of your notes, handouts, lecture slides, course readings, or sample solutions to this year's homework. You may **not** discuss these questions with anyone else (other than asking the course instructors for clarification), nor may you try to find solutions to these problems elsewhere, e.g., on the web or from previous years' exams or homeworks.

For short-answer questions, you shouldn't need to write more than 100 words or so, and for most questions a few dozen words in telegraph-ese should be sufficient.

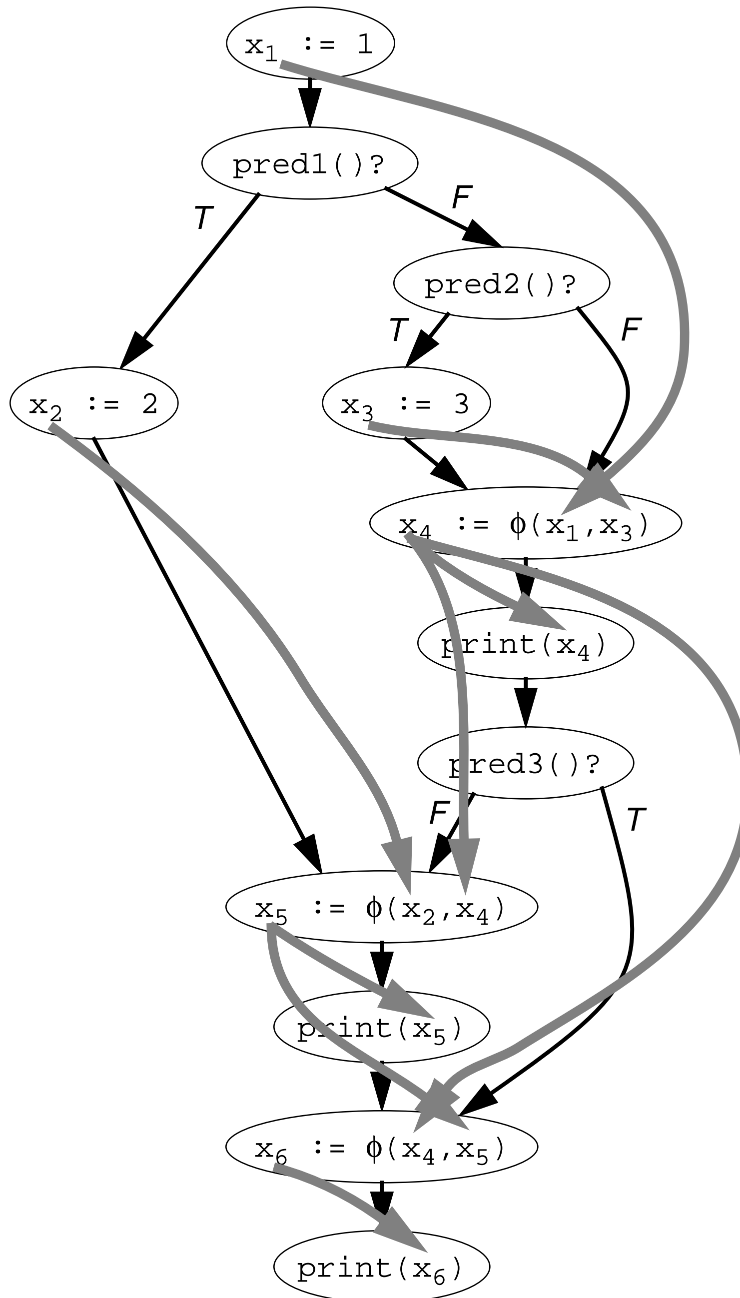
You should turn in a hardcopy of your solutions. You may develop all or part of your solutions on-line, as long as you print out your electronic solutions and turn them in. (Make sure you do not develop your solutions using software that does not reasonably word-wrap long lines!)

100 points total.

1) Consider the following program fragment:

```
    x := 1;  // statement 1

    if pred1() then
       x := 2;  // statement 2
    else
       if pred2() then
          x := 3;  // statement 3
       end

       print(x);  // statement 4

       if pred3() then
          goto done;
       end
    end

    print(x);  // statement 5

done:  // target of the goto

    print(x);  // statement 6
```

a) [6 pts] Draw the control flow graph for this program frament, in proper SSA form. Draw the def/use chains (connecting defs to uses, not statements or blocks) for the x variable of this CFG.



b) [2 pts] What are the reaching definitions of the x arguments of statements 4, 5, and 6?

4: x_4; 5: x_5; 6: x_6

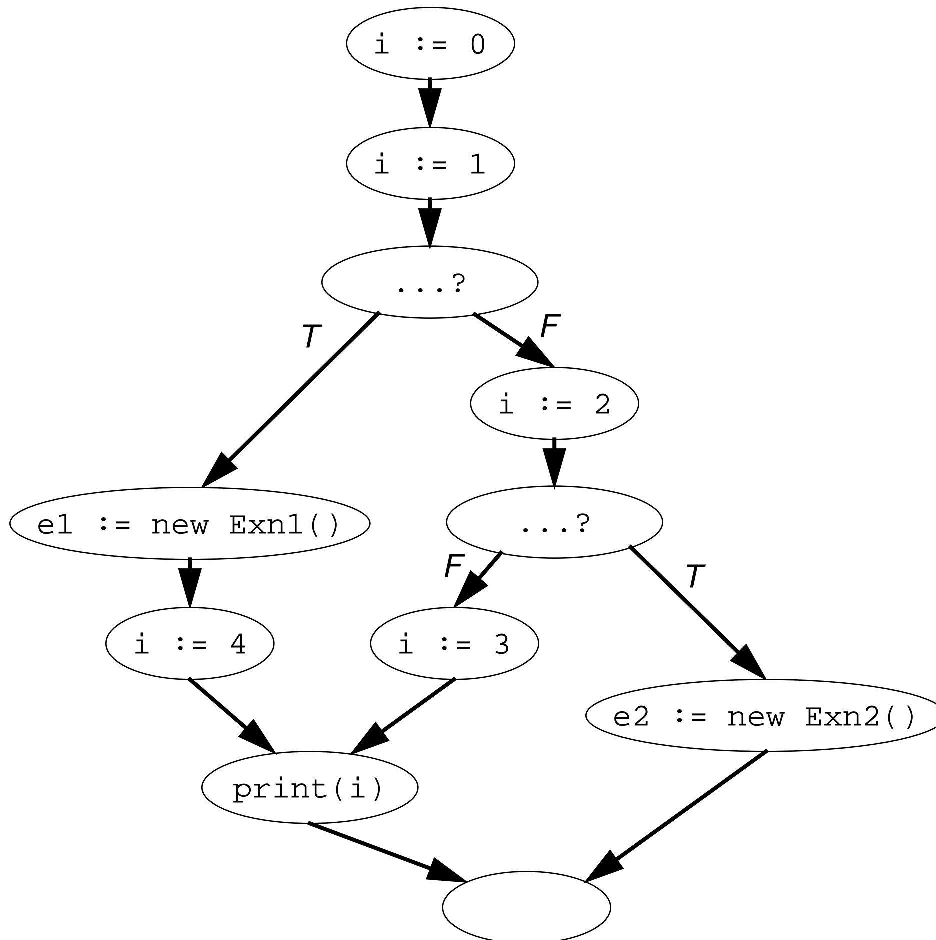
2) Consider the following Java program fragment containing exception throws and catches (where `Exn1` and `Exn2` are unrelated classes, so that the `Exn1` catch handler only catches the `Exn1` throw, and vice versa):

```
try {
   i := 0; // statement 1
   try {
      i := 1;  // statement 2
      if (...) throw new Exn1();
      i := 2;  // statement 3
      if (...) throw new Exn2();
      i := 3;  // statement 4
   } catch (Exn1 e1) {
      i := 4;  // statement 5
   }
   print(i);
} catch (Exn2 e2) {}
```

(FYI, an on-line reference manual for exceptions in Java is available at http://java.sun.com/docs/books/jls/html/11.doc.html#44121.)

We wish to extend our compiler machinery to account for the control- and dataflow effect of exceptions, so that we can continue to do dataflow analysis and optimizations as we have come to know and love, even in the presence of exceptions.

a) [6 pts] Draw the control flow graph for this fragment that correctly accounts for the possible paths of control flow in this program (but no more paths than necessary), and that accounts for the flow of data values through the program. You can treat `new` expressions as basic operations for which there are corresponding three-address instructions in your CFG. Throws and catches should be turned into appropriate control flow edges, plus whatever instructions you need to pass data. You can assume that your compiler is smart enough to be able to track the classes of the exception objects and resolve the class-testing operations that are implicit parts of the catch clauses, e.g. that the `throw new Exn1()` exception is the one and only exception handled by the `catch (Exn1 e1)` clause.



b) [2 pts] What is the minimum set of reaching definitions for the `i` use in the `print(i)` statement?

s4 and s5

c) [3 pts] Compiler writers worried that Java's language design, where many operations could implicitly throw exceptions (such as null pointer exceptions on all object and array dereferences, and divide-by-zero exceptions on divide and mod operations), could lead to poor optimization quality. Aside from the run-time cost of testing for the excepting conditions (which often can be implemented by hardware checks, essentially for free), why might compiler writers be worried?

Because there are a lot of additional branches in what would otherwise be straight-line code. These branches break up basic blocks, limiting local analysis, and making intraprocedural analyses more costly (although probably not less precise). Catch clauses introduce merges where none used to be, which can dilute information.

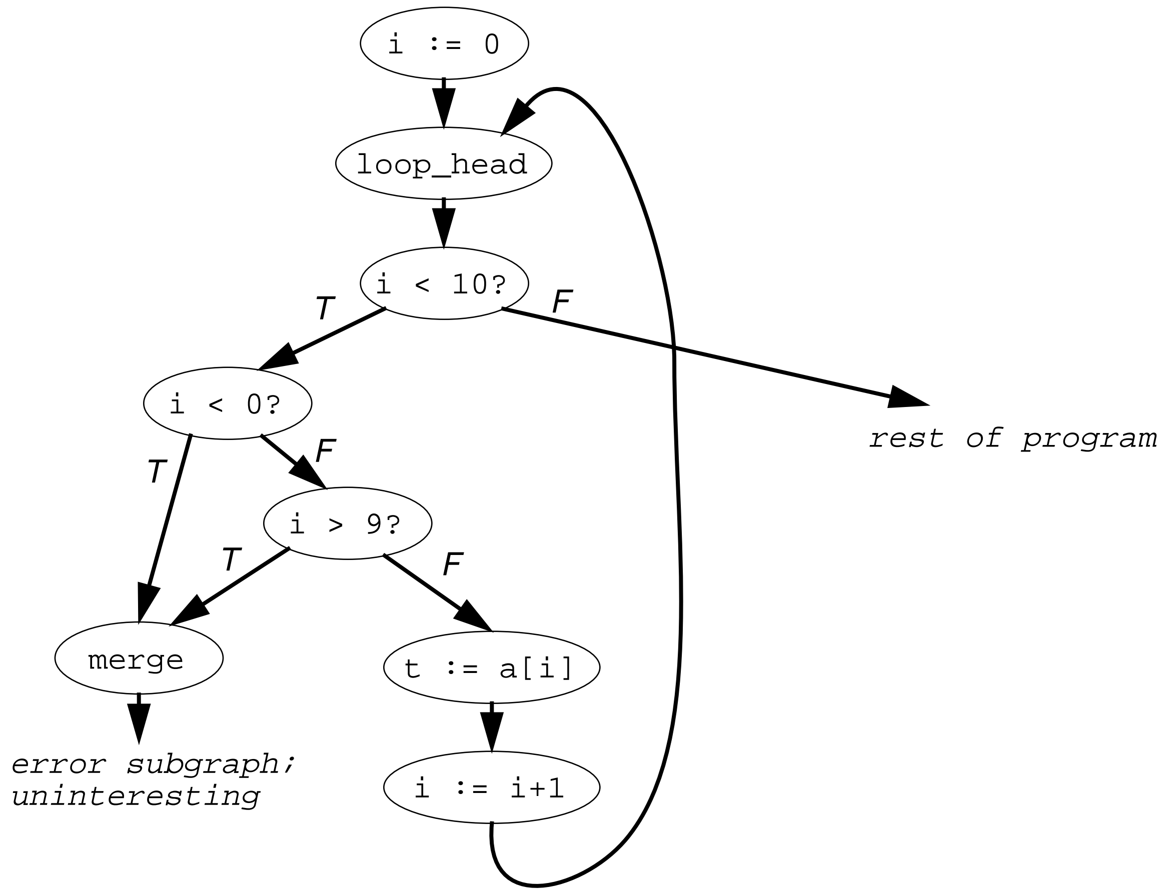
3) Consider the following program fragment:

```
int[] a := new int[10]; // an array of ints, indexed 0..9
... // a filled in with interesting values here

// fragment starts here
i := 0;
while i < 10 then
   t := a[i];
   print(t);
   i := i + 1;
end
```

In a Java-like language, the array dereference `a[i]` will implicitly contain a bounds check. The internal form might be like:

```
if i < 0 || i > 9 then
   throw new ArrayIndexOutOfBoundsException();
end
t := a[i];
```

The control flow graph for the above fragment, in internal form, could be



We would like to develop a compiler optimization to remove redundant bounds checks. Some form of integer range analysis seems like a good basis.

a) [10 pts] Define an integer range analysis over the control flow graph that will deduce that the two bounds check conditional branches within the loop can be folded away. Define your lattice domain (the set of elements and the ordering operation over elements), and also identify the top and bottom lattice elements and the lattice meet operation implied by this definition. Define your flow functions. Treat regular merges, loop-head merges, and conditional branches as separate and distinct kinds of nodes (as in the CFG above) for which you can write flow functions; flow functions for the two kinds of merge nodes take a sequence of input values and produce a single output value, while the flow function for branch nodes takes a single input value and produces a pair of output values. For the purposes of this question, you can assume that there exists an appropriate *widening_meet* operation defined on your lattice domain; you don't have to define it explicitly. Show the information computed at the interesting program points by your analysis, and explain why it proves that the bounds check conditional branches are unnecessary.

Domain = $\langle Var \rightarrow (((Int+\{-\infty\}) \times (Int+\{\infty\})) + T), \leq_{map} \rangle$

where $map_1 \leq_{map} map_2$ iff

$\forall v \in dom(map_1) \cup dom(map_2)$.

$map_2(v) = T$ or

$(lo_1,hi_1) = map_1(v)$ and $(lo_2,hi_2) = map_2(v)$ and $lo_1 \leq lo_2$ and $hi_1 \geq hi_2$

(I.e., a map from each variable to a pair of integers giving that variable's lower and upper bounds, which can be positive or negative infinity, or to top (if the variable has an empty range).)

[map(v) is defined to be T if v isn't in dom(map)]

Top = {}

Bottom = $\{* \rightarrow (-\infty,\infty)\}$

map_1 meet map_2 = $\{ v \rightarrow r \mid (v \rightarrow r_1) \in map_1, (v \rightarrow r_2) \in map_2, r = r_1 \text{ meetElem } r_2 \}$

T meetElem r = r

r meetElem T = r

(lo_1,hi_1) meetElem $(lo_2,hi_2) = (\min(lo_1,lo_2), \max(hi_1,hi_2))$

Flow functions for straight-line code:

$RA_{\texttt{x := y op z}}$: succ = pred – $\{x \rightarrow *\} \cup \{x \rightarrow op(pred(y),pred(z)\}$

where e.g.

+(T,r) = T

+(r,T) = T

$+((lo_1,hi_1), (lo_2,hi_2)) = (lo_1 + lo_2, hi_1 + hi_2)$

[If some operand y is actually a constant k, then pred(y) is defined to be (k,k) in these flow functions.]

$RA_{\texttt{x := y}}$: succ = pred – $\{x \rightarrow *\} \cup \{x \rightarrow pred(y)\}$

$RA_{\texttt{x :=}\ \textit{some other unanalyzable expr}}$: succ = pred – $\{x \rightarrow *\} \cup \{x \rightarrow (-\infty,\infty)\}$

Flow functions for conditional branches (we will represent a branch successor that is proven unreachable by a top map, i.e., the branch operation is folded):

$RA_{\texttt{x op y?}}$: $succ_{true}$ = pred join op((x→pred(x)), (y→pred(y))}
$succ_{false}$ = pred join not-op((x→pred(x)), (y→pred(y))}

where e.g.

<((x→T),(y→r)) = {} // can make branch unreachable if either operand is T
<((x→r),(y→T)) = {}
<((x→(lo_1,hi_1)), (y→(lo_2,hi_2))) =
if $lo_1 \geq hi_2$ then {} // make branch unreachable if always false
else {x→(lo_1,min(hi_1,hi_2-1)), y→(max(lo_2,lo_1+1),hi_2),*allothervars*→(-∞,∞)}
// try to narrow ranges of x & y

[if either x or y is a constant, then we don't actually store new bindings for them in the map.]

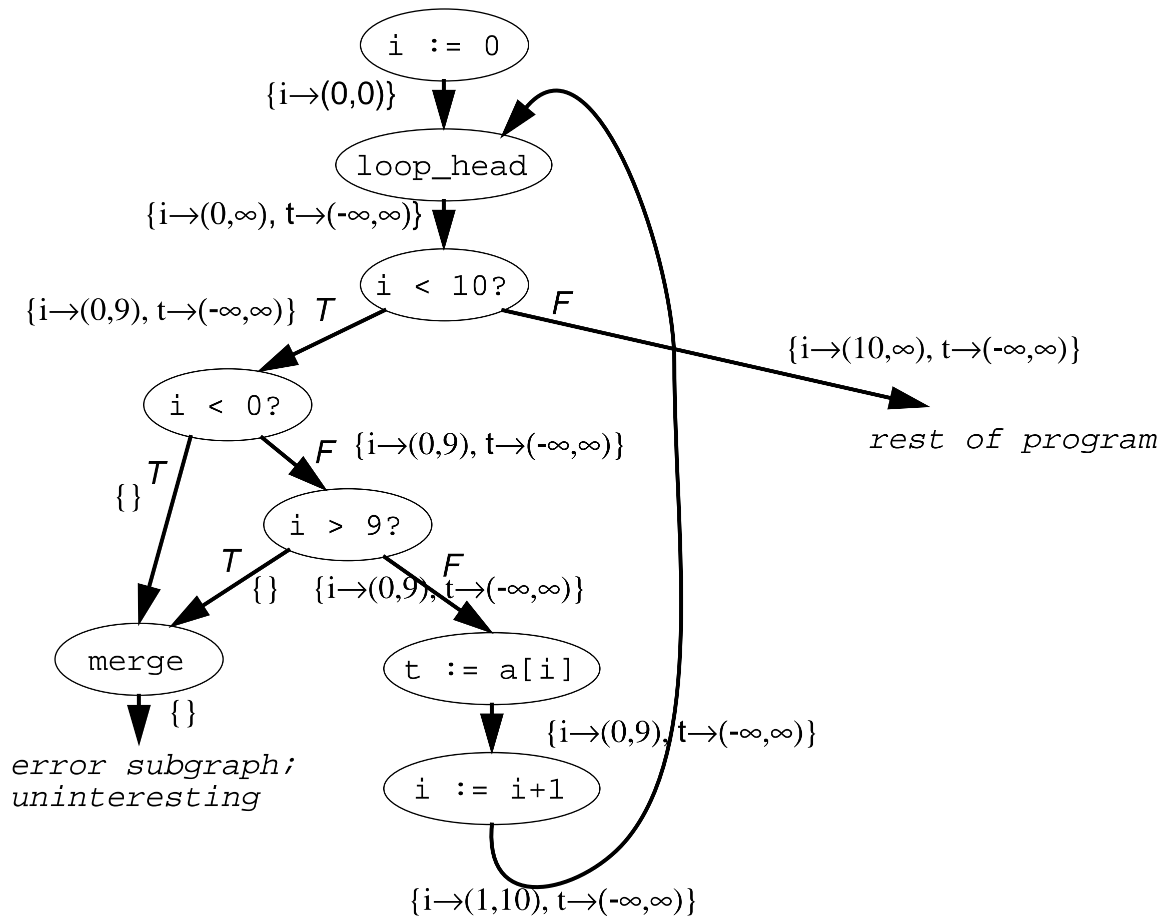
[join is the least-upper-bound function on the lattice, complementing the meet function. we use it so that we can produce a mapping that will selectively improve parts of the input mapping.]

[not-op is just the negation of op, e.g. not-< is ≥.]

Flow functions for merges:

RA_{merge}: succ = $pred_1$ meet ... meet $pred_n$

$RA_{loop\text{-}merge}$: succ = $pred_1$ widening_meet ... widening_meet $pred_n$



[This information assumes that the widening_meet operation is smart enough to generalize the hi operand, but not the low operand, when doing a widening_meet of (0,0) and (1,1), and then (0, 1) with (1, 2). It would also be fine for it to figure out somehow that (0,10) is the range that i might take on at the loop head.]

Because the two bounds checks on i have unreachable true successors (info is {}, the top element), the branch operation is folded to be a no-op, passing control only to their false successors.

b) [4 pts] For faster analysis, one might like to perform integer range analysis along def/use chains. Why would this not be able to discover that the conditional branches are redundant?

Because there's no way to refine the information after the i < 10 test; the i assignments go directly to the i uses, without passing through the conditional branches.

c) [4 pts extra credit] Can you suggest a way of extending def/use chains that would allow analyses over them to identify these tests as redundant, without recourse to the control flow graph?

We need to treat conditional branches as pseudo-assignments, one for each successor edge, of their operand variables, so that the value associated with the variable can change after the branch. Such an assignment could have the form "$x_i := \chi(x_j)$", with the value of x_i by default just being x_j, but for analyses like this range analysis, being something more refined.

4) [4 pts] Why are program points, for which dataflow information is computed, defined as the edges in the graph rather than the nodes?

Because edges have a clear notion of the information at that program point. Information at nodes is not as clear-cut, because the information is in flux across a node. Really, the information has to be about the point before the node, or after the node (and there can be several of those, in the case of branches and merges). To avoid this ambiguity, we just put information at edges, where there is no ambiguity.

(Some compiler representation can still store information with nodes, but this is only a particular implementation; logically, the information is still stored with edges, either the edge(s) before or after the node. Some convention on how to interpret information stored with nodes is still required to turn the physical representation into the appropriate logical one.)

5) a) [5 pts] The general worklist algorithm for iterative dataflow analysis first initializes all edges to top, to ensure that it converges to the best fixed point solution (if it terminates). Despite this, explain why the safety, termination, and worst-case time complexity of the worklist algorithm are not hurt in any way by what lattice values the edges are initialized to (they could be initialized to any random lattice elements without hurting safety or worst-case convergence speed).

The algorithm runs until all flow functions reach a fixed point, which is (by definition) safe. This processing doesn't depend on what the initial lattice values are. Termination and complexity are based on the height of the lattice, the longest path from the top to the bottom of the lattice. Starting from top is already the worst case, so starting anywhere else can't hurt the previous worst-case situation.

b) [3 pts] Explain why the worklist algorithm might not converge to the best fixed point solution (assuming it terminates) if edges are not all initialized to top.

If the edges are initialized to some point below what would have been the original fixed point for the standard algorithm starting from top, it may be the case that the iteration won't be able to refine the information back up to the best fixed point. If the edges are all initialized to something that's >= the original fixed point, then we're going to get the same good result, but we have no such guarantees otherwise.

c) [5 pts] Explain why the safety, termination, and (if it terminates) worst-case time complexity of the worklist algorithm do not depend in any way on the order in which nodes are analyzed.

All these properties were computed in a “flow insensitive” manner, not caring in what order node flow functions were processed. Safety depended on running all flow functions, in any order, until no more edge changes happened. Termination and complexity were based on arguments about how often a given edge’s value could be lowered (not depending in what order the various edges were lowered), and monotonicity of each flow function in isolation.

6) Consider the following program fragment:

```
void P() {
   int a := A;
   int b := R(a);
   int c := b*b*b*b*b*b;
   print(c);
}
void Q() {
   int i := I;
   int j := R(i);
   int k := j*j*j*j*j*j;
   print(k);
}
int R(int x) {
   int y := x*x*x*x*x*x;
   return y;
}
```

We’d like to use interprocedural constant propagation (both of arguments and results) to fold the expensive computations of `c`, `k`, and `y`.

a) [4 pts] Give expressions to replace *A* and *I* that will allow a context-sensitive version but not a context-insensitive version of interprocedural constant propagation to fold the `c` and `k` computations. Explain why your choices have this effect.

A=1, I = 2 (or any pair of different integers). Context sensitivity is necessary and sufficient to enable each call to R to be analyzed separately, computing a distinct result for each R call. Context insensitivity would smear the 1 and 2 values together, not allowing any folding within R, causing R to return bottom as its result, and therefore not allowing any folding in the callers based on the result of R.

b) [4 pts] Give expressions to replace *A* and *I* that will allow a context-insensitive version of interprocedural constant propagation but not intraprocedural constant propagation to fold the `c` and `k` computations. Explain why your choices have this effect.

A = I = 1 (or any common value). In this case, smearing doesn't hurt, so context insensitive interprocedural analysis is sufficient. But of course intraprocedural analysis of P and Q based on worst-case assumptions about R (i.e., what you do in the absence of interprocedural analysis of some form) won't let the caller learn anything about the results of the R calls.

c) [4 pts] Give expressions to replace *A* and *I* that will allow the `y` computation to be folded if using procedure specialization but not any kind of inter- or intraprocedural analysis. Explain why your choices have this effect.

A=1, I = 2 (or any pair of different integers). Here interprocedural analysis, whether context sensitive or not, will have to form a single lattice element to describe x for use when compiling the single copy of R, and the smearing will prevent the y computation from being folded. But procedure specialization can make two separate copies of R, one for calls where x is 1, and a separate one for calls where x is 2, allowing each copy to be optimized independently, and successfully.

7) You wish to compute a program's call graph. You decide to perform a summary-based interprocedural analysis. Your analysis computes, for each procedure, a summary listing the set of direct callee procedures. One can then assemble the whole program's call graph from all the summaries.

a) [4 pts] Assuming that there are no first-class function values (e.g. Scheme lambdas or C function pointers) or dynamically dispatched messages, can you compute each procedure's summary flow-insensitively, or is flow-sensitive analysis required? Explain your answer.

Flow insensitive analysis is sufficient, since all one needs to do is scan through each procedure to find its calls, in any order.

b) [4 pts] In what way(s) would the problem become hard in the face of calls of first-class, computed functions?

One would need to compute what the possible functions are that each function expression might evaluate to in a call statement. This probably requires some sort of dataflow analysis to be effective, which is flow-sensitive, and perhaps an interprocedural problem in its own right.

8) We studied algorithms for identifying loop-invariant assignment statements and then moving them outside the loop.

a) [6 pts] Explain how to extend the rules to determine whether or not a statement of the form

```
x := *p
```

is loop-invariant (ignore whether or not one could move such a statement). You can assume access to may-point-to information (from which you can derive must-point-to information) at each program point.

This stmt is loop-invariant if its rhs (*p) always computes the same value on each execution, and it has the same side-effects each time (i.e., *p is idempotent). This is true if p is loop-invariant (so that it always points to the same memory location each execution of the loop, even if statically there are multiple possible targets), and all the variables p may point to are loop-invariant (so that we get the same value back from the load each time). If p might be null or some other broken pointer value, then if *p is loop-invariant by this definition, dereferencing p will cause the same trap each time the loop is executed.

b) [6 pts] Explain how to extend the rules to determine whether or not a statement of the form

```
*p := x
```

is loop-invariant (ignore whether or not one could move such a statement). You can assume access to may-point-to information (from which you can derive must-point-to information) at each program point.

This stmt is loop-invariant if its rhs (x) is loop-invariant, and it has the same side-effects each time. This statement will update the same variable each time if p is loop-invariant. (We do not need to require p to be known statically to have a singleton may-point-to set; any ambiguity just means that we may not be able to infer that the target of the store is loop-invariant. We also do not need to worry about whether the target of p is loop-invariant, since (if unique) we're strongly updating it to be loop-invariant for downstream uses.)

c) [8 pts] Consider the situation where a conditional branch has been identified as having a loop-invariant predicate. Discuss how you might hoist such a branch out of the loop, describing sufficient conditions for safe motion, and identifying the opportunities that might arise to further optimize the loop body, e.g. identifying additional loop-invariant computations, after the conditional branch and any associated statements are hoisted out of the loop.

I would want to hoist an entire if-then-else "subtree". In the form of a control flow graph, a subtree takes the form of a connected region of the CFG with a single entry (the conditional branch) and a single exit (the merge at the bottom). I would want the whole CFG to contain only loop-invariant statements, all of which would be legal to hoist out of the loop (imposing domination and dataflow restrictions as discussed in class). If so, I could just move the whole subgraph to the loop preheader. If I have moved such a subgraph, then instructions downstream of the "subtree" that used to have multiple definitions in the loop (e.g. one on each of the true and false branches) now may have all definitions outside of the loop, allowing them to become loop-invariant computations. In SSA form, the phi functions after the merge at the bottom of the loop-invariant subgraph should also be moved as part of the subgraph, leaving stmts after the phi nodes dependent on a single definition (the phi stmt) that is now outside the loop.

Another possibility would be to hoist just the if test outside the loop, with the whole loop replicated on both the true and false branches of the if. Within each copy of the loop, only one of the true and false branches of the original if would be retained; this could be implemented simply by folding the original if in each copy of the loop given the information implied by the truth or falsity of the if predicate guarding that copy of the loop.

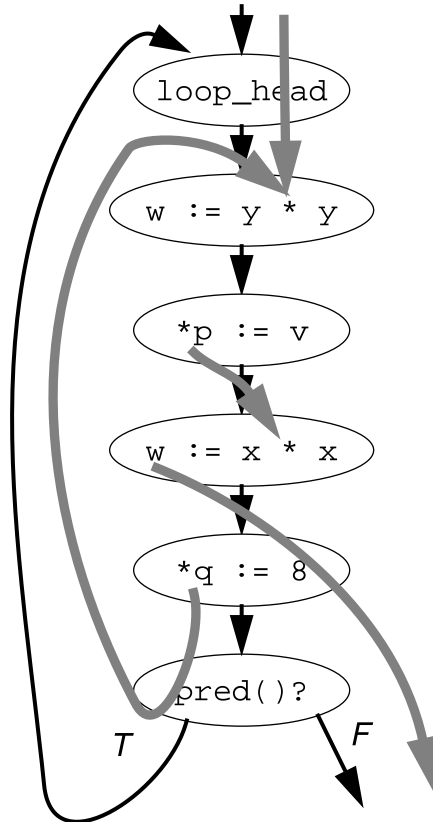
9) [6 pts] Consider the following program fragment, with may-point-to information (from which must-point-to information can be deduced) computed at the start of this fragment as given:

```
// may-point-to == {p->x, q->y, q->z}
v := 3;
w := 4;
x := 5;
y := 6;
z := 7;

do {
   w := y * y;   // statement 1
   *p := v;      // statement 2
   w := x * x;   // statement 3
   *q := 8;      // statement 4

} while pred();
```

```
print(w);
```

For each of statements 1-4, identify whether the statement is loop-invariant, whether it can be hoisted above the loop, and whether it can be sunk below the loop (moved right after the loop exit, before the print). (For the purposes of this question, do not perform any dead assignment elimination.) You might wish to construct the CFG, perhaps in SSA form, and then identify def/use chains, in order to help answer this question. Explain your answers.

First, here's the CFG + DFG (for w, x, y, and z; all other DFG edges go from the assignments outside the loop to inside) for the loop body:



stmt	loop-invariant?	hoisted above?	sunk below?
S1	no: multiple reaching defs for y not all outside loop	no, not loop-invariant	no, not loop-invariant

stmt	loop-invariant?	hoisted above?	sunk below?
S2	yes: p and v are loop-invariant	yes: dominates exit and doesn't change dataflow of x references	no: would change value read in following stmt
S3	yes: x is loop-invariant (its single definition is itself a loop-invariant statement)	no: would change value of w read below loop due to earlier non-loop-invariant definition of w	yes: dominates exit and doesn't change dataflow of w references
S4	yes: q is loop-invariant	no: could change value of y read in first loop statement on first iteration	no: could change value of y read in first loop statement on subsequent iterations

10) [4 pts] Dynamic profile information, such as the execution count of each program point on a previous run of the program, can be exploited by compilers to improve the quality of their optimizations. How might such information be used to make loop-invariant code motion more effective?

If profile data suggests that the loop is rarely executed, then hoisting loop-invariant computations that don't dominate the loop exit can be suppressed. Alternatively, if profile data suggests that the loop is frequently executed, then such hoisting can be encouraged.