This is an open-book, open-notes, open-laptop, closed-network exam. Please do not post anything on Piazza until the solutions are put up on the class website.

You have 1 hour 20 minutes to work on this exam. The examination has 5 problems worth 70 points. Please budget your time accordingly. Write your answers in the space provided on the exam. If you use additional scratch paper, please turn that in as well.

Your Name: _________________________ SUNet ID: _________________________

The following is a statement of the Stanford University Honor Code:

1. The Honor Code is an undertaking of the students, individually and collectively:

   (a) that they will not give or receive aid in examinations; that they will not give or receive unpermitted aid in class work, in the preparation of reports, or in any other work that is to be used by the instructor as the basis of grading;

   (b) that they will do their share and take an active part in seeing to it that others as well as themselves uphold the spirit and letter of the Honor Code.

2. The faculty on its part manifests its confidence in the honor of its students by refraining from proctoring examinations and from taking unusual and unreasonable precautions to prevent the forms of dishonesty mentioned above. The faculty will also avoid, as far as practicable, academic procedures that create temptations to violate the Honor Code.

3. While the faculty alone has the right and obligation to set academic requirements, the students and faculty will work together to establish optimal conditions for honorable academic work.

Signature: _________________________

<table>
<thead>
<tr>
<th>Problem</th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max</td>
<td>15</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>25</td>
<td>70</td>
</tr>
</tbody>
</table>
Problem 1. True or False? Briefly justify your answer in 1 to 5 lines [15 Points]

1. A monotone forward data flow analysis must set OUT[Entry] to the “top” of the semilattice, and a backward flow must set IN[Exit] to the “bottom”.
   False. The boundary conditions are set according to the dataflow algorithm, and need not be either top or bottom. For example, if a program includes “return x”, the variable “x” is live at the exit so the liveness algorithm initializes its boundary to to { x }

2. Partial Redundancy Elimination can introduce speculative execution (execution of instructions that have no side effects but potentially would not be executed by the original program).
   False. PRE only introduces expressions at points where the expression is anticipated, thus guaranteed to execute.

3. Two natural loops (arising from two back edges) can intersect without being completely nested one into the other.
   True. They can share the header.

4. Given a machine with n registers, without the spilling heuristic, if register coloring gets stuck, then rerunning the algorithm and picking nodes in a different order may find a coloring successfully.
   False. Without spilling, the graph coloring algorithm only removes nodes with degree less than n, and removing a node only decreases the degrees of the neighbors. In other words, once a node is removable, it is removable for the rest of the algorithm, and thus will be removed at some point. Therefore, no matter the order of operations, the algorithm will remove the same set of nodes.

5. Given a control flow graph, the set of retreating edges is unique.
   False. It depends on the order in which nodes are visited by the DFS: some edges can be advancing or retreating based on the visit order.
   For example in this flow graph, the edge from A to B can be retreating or advancing depending on which one is visited first:
Problem 2. Register Allocation [5 Points]
Consider the following interference graph:

On a machine with 3 registers, will the heuristic coloring algorithm always, sometimes, or never find a coloring (with spilling)?

This graph is 3-colorable. A coloring is:

- x1 = 1
- x2 = 2
- x3 = 3
- x4 = 1
- x5 = 1
- x6 = 2

On the other hand, the heuristic algorithm removes x1 and then gets stuck, because all other nodes have degree 3. Depending on how it chooses the next step it may or may not color the graph successfully.
Problem 3. Dominators [10 points]
Consider the following control flow graph:

- Draw the immediate dominator tree for the flow graph above.

- What are the natural loops in the above flow graph? Indicate both the set of blocks and the corresponding back edge.
- Back edge $G \rightarrow D$, natural loop $\{D, E, G\}$
- Back edge $F \rightarrow A$, natural loop $\{A, B, C, D, E, F, G\}$

- Is the flow graph reducible? Explain your answer.
  No. Depending on how you construct your tree, the $B \rightarrow D$ or $E \rightarrow B$ edge becomes a retreating edge. The $G \rightarrow F$ edge however is a cross edge.

**Problem 4.** Partial Redundancy Elimination [15 Points].
Show the result of running partial redundancy elimination. What’s the final optimized flow graph? You don’t need to show the intermediate steps.
The solution adds a temporary at the beginning of the block of w and on the edge coming out of \texttt{a = read()} that leads to the center of the graph. It changes \(w\), \(x\), and \(y\) to use the temporary. \(z\) is not changed.
Problem 5. Taint Analysis [25 points]

Given a string variable \( x \), we say that \( x \) is *tainted* if any portion of \( x \) can potentially come from input from the user. Your goal is to design an analysis that can detect usage of variables that are potentially tainted. This is useful to detect common vulnerabilities such as SQL Injection, Cross Site Scripting and others.

Your language supports the following operations:

- \( x = \text{const} \): set a variable to a constant
- \( z = x + y \): concatenate the strings \( x \) and \( y \)
- \( x = \text{input}() \): read from the input
- \( \text{use}(x) \): use the variable \( x \)

You can treat uninitialized variables as you prefer, but state your assumptions.

1. Design a dataflow analysis that can detect potentially tainted variables. Is it a forward or backward pass? What is your semi-lattice? What is the transfer function?

   Forward pass. The semilattice is sets of variables; meet operator is union. A variable is in the set if tainted. Initially all variables are not tainted (empty set). Handle the operations as:
   - \( x = \text{const} \): clean the variable (remove it from taint set)
   - \( z = x + y \): taint the variable \( z \) (add it to the taint set) if either \( x \) or \( y \) is tainted.
   - \( x = \text{input}() \): taint the variable (add it to the taint set)
   - \( \text{use}(x) \): no changes

2. Is your framework monotone?

3. Is it distributive?

   Yes and yes (same argument as uninitialized variables from HW1, which in turn boils down to reaching definitions)

4. How do you warn for tainted variables?

   At any \( \text{use}(x) \), warn if \( x \) is in the taint set.

5. Suppose we add another operation, \( y = \text{sanitize}(x) \). This operation reads \( x \) and transforms it into a string that is not tainted. What is the transfer function for this new instruction?

   Remove the variable from the taint set when you encounter \( \text{sanitize} \); this is no different than setting \( y \) to a constant, and \( x \) does not matter.
6. We now extend the language with pure function calls. These calls take arguments and return values, but cannot have side effects, and cannot modify local variables (e.g. through pointers). What is the new transfer function? You are not allowed to analyze across functions.

The result is tainted if any of the argument is tainted (like string concatenation).

7. Finally, we generalize the language to arbitrary function calls, include those that can indirectly call input. How do you conservatively define the transfer function in this case? Again, assume that local variables cannot be modified (e.g. through pointers) by the function call, and you cannot see the code inside the called function.

In the general case, treat any function call as input, and taint the result.