Problem 1. Monotonicity

Which of the following functions are monotonic over the set of positive integers, as ordered by a less-than relation? For those that aren’t monotonic, please provide a counter example.

1. \( f(x) = \frac{x}{2} \)
2. \( f(x) = \log x \)
3. \( f(x) = x + \sin^2(x) \)

Now suppose we have a partial order defined over subsets of some universal set \( U = \{a, b, c, d, e\} \). Let the ordering be defined by a subset relation. Which of these functions is monotonic? For those functions that aren’t monotonic, please provide a counter example.

1. \( f(A) = A^c \), where \( A^c \) is the complement of \( A \).
2. \( f(A) = \mathcal{P}(A) \), where \( \mathcal{P} \) is the power set operation.
3. \( f(A) = A \cap \{a, b\} \)
**Problem 2.** True or False? Briefly justify your answer.

1. The iterative algorithm discussed in class will always terminate if the semi-lattice of a dataflow is of finite height.

2. There exists only one possible MOP solution for acyclic control flow graphs.

3. Setting all values to the bottom of the semi-lattice for a given data flow problem is the most conservative solution to the iteration equations.

4. A forward dataflow algorithm with a monotone framework and finite descending chains can never terminate if the semi-lattice is infinitely large.

5. You have defined a forward dataflow algorithm that is distributive, has a monotone framework, and has finite descending chains. If you were to initialize $\text{OUT}[b]$ for all basic blocks to a value between $\perp$ and $\top$ instead of $\top$,
   
   (a) your algorithm will still give a safe answer for all flow graphs.
   (b) your algorithm will give the MOP solution for some flow graphs.

**Problem 3.** Compute the available expressions (Chapter 9.2.6 in ALSU) on entry and exit for each basic block in the following flow graph:
**Problem 4. Live Range Analysis**

A path is *definition free* with respect to a variable $y$ if there does not exist a definition of variable $y$ along that path. The live range of a definition $d : y = x + z$ that defines variable $y$ includes all the program points $p$ such that (1) There is a path from $d$ to $p$ that is definition free with respect to $y$ and (2) There is a path from $p$ to $q$, a statement that uses the variable $y$, that is definition free with respect to $y$.

Intuitively, the live range of a definition consists of points along all subsequent paths until either the variable is no longer used along that path or a new definition overwrites it. This concept is applicable to register allocation: two definitions can be assigned to the same register if their live ranges do not intersect.

\[ \begin{array}{c}
b_0 & a = x + y; \\
b_1 & z = 2 \cdot a; \\
b_2 & a = 0; \\
b_3 & a = a + 1; \\
b_4 & b = a + z; \\
\end{array} \]

In the above example, the live range of definition $a = x + y$ is \( \text{exit}(b_0), \text{entry}(b_1), \text{exit}(b_1) \) and \( \text{entry}(b_3) \). Similarly, the live range of the definition $b = a + z$ is \( \text{exit}(b_4), \text{entry} \). The two live ranges do not intersect, so $b$ can reuse $a$'s register.

Describe an analysis that computes the live range for each definition in a program. You may use algorithms discussed in class.

**Problem 5. Interrupt Warnings**

Interrupts are signals used by hardware to communicate events to code running on the processor. A common type of interrupt is timer interrupts, which are triggered on a fixed clock interval. Kernel code will typically register a handler for timer interrupts - this handler can be used to perform context switching between kernel threads. This lets you implement time-sharing on a single-processor system.

However, we need a way to prevent kernel threads from being switched out while manipulating shared data structures - otherwise we could leave the data structure in an inconsistent state. This can be achieved by simply disabling interrupts before the critical section and re-enabling them after the critical section\(^1\). Unfortunately, it’s easy to forget to re-enable

---

\(^1\)Kernels typically use this pattern to implement higher level concurrency constructs like locks and semaphores.
interrupts after disabling them, especially if a function has multiple exit points. This will result in starvation of other threads that are waiting in the work queue.

Describe an analysis that detects if a function may exit while interrupts are still disabled - the analysis must also identify the `disable_interruption()` statement that is responsible. You may assume that the function is called with interrupts enabled, and that each basic block contains only one instruction and consists of one of the following operations:

- `disable_interruption()`
- `enable_interruption()`
- Other local code that doesn’t call any functions or modify the interrupt state.

First describe an analysis that fits within the data-flow framework. Then, describe how the data-flow results can be used to issue warnings to the programmer for any `disable_interruption()` statements where interrupts may not be re-enabled before exiting. Choose the direction of your analysis carefully. When describing your dataflow analysis, fill in a table like the following:

<table>
<thead>
<tr>
<th>Direction of your analysis (forward/backward)</th>
<th>Lattice elements and meaning</th>
<th>Lattice diagram</th>
<th>Is there a top element? If yes, what is it?</th>
<th>Is there a bottom element? If yes, what is it?</th>
<th>Meet operator</th>
<th>Transfer function of a basic block</th>
<th>Boundary condition initialization</th>
<th>Interior points initialization</th>
</tr>
</thead>
</table>

**Problem 6. Variable Sign Tracking**

Suppose that you are writing a program that performs some operations on real-valued variables, which are initialized to 0 at the start of the program. Basic blocks in your program can contain one of the following operations, where $x$ and $y$ are variables.

- $x = c$, where $c$ is a constant.
- $x = -y$, which negates the value of $y$.
- $x = \text{sqr}(y)$, which squares $y$.
- $x = \sqrt{y}$, which takes the square root of $y$.  

For every square root operation, we want to show a warning if it could potentially error due to the input being negative. Describe an analysis that can surface such warnings. First describe an analysis that fits within the data-flow framework. Then, describe how the data-flow results can be used to show warnings to the programmer. When describing your dataflow analysis, fill in a table as you did in Problem 5.

**Problem 7.** Constant Propagation

For the following program, apply the constant propagation algorithm as seen in class to replace operations and variables with constants when possible.