*BaThis document does not need to be updated with wording changes for the questions.*

The purpose of this assignment as a whole is to get students to “see” the Level of Logic directly, and to understand that, while it is not normally written down, it is every bit as rule-based and mechanical as the code it guides. The questions of this assignment are each focused on drawing different lessons from the layer of logic which are applicable to everyday software engineering.

**Exercise 1** Conceptual question: In the latter example, how could we change the code without altering the postcondition? How does the “forgetting” of assertions correspond to a form of modularity?

**Intent:**

This is a direct demonstration of the interface/implementation distinction and the many-to-many relationship between them. It gives a very tiny but explicit example of how many implementations can have the same specification, and how a weaker specification allows more modularity.

A student who changes the assignments of x and y has only partially understood this. The best answer will say that we can eliminate x and y entirely, and set z to anything.

**Exercise 2** Look at the code in Figure 1. Fill in all assertions.

**Intent:** This is a basic intro to Hoare logic, a warmup for the future exercises. Most students get it perfectly. We accept if they simplify the assertions going bottom up, but they should not weaken or strengthen any, because the problem does not require that and doing so means they’ve failed to apply the rules mechanically. They should get either *true* or the empty precondition at the top.

**Exercise 3** We haven’t given you any rules other than the ones for straight-line code. But we can still write some interesting programs if we use integer division, which we introduce in this exercise. In this exercise, assume all variables are integers, and all division is integer division (rounds toward 0)

**Intent**: The purpose of this exercise is to help them make fewer mistakes in Exercise 4. It guides them away from incorrect simplifications. It is also easier to help them follow the rules instead of giving up and using their intuition if they can’t apply the rules. They should get it exactly.

**Exercise 4**: This exercise is much harder, but is meant to illustrate two important points in software design: that conditionals are not simply “if-statements” and can appear even in straight-line code, and that Hoare logic gives an objective measure of complexity which can motivate refactoring. We will do part of this example now, and revisit in again in the next section. In this exercise, as before, assume all variables are integers, and all division is integer division (rounds toward 0). Also assume that x/0 == 0 for all x. 1. Look at the code in Figure 3. Fill in the assertions between each line. We have done the last and first ones for you. 2. In what sense does this code contain a conditional?

**Intent:** This is the main exercise of this assignment. It drives home how “being a conditional is a logical, not a syntactic notion.” It also feeds into the next exercise, which uses the same code, also showing a lesson about complexity.  
  
There is alas some incidental complexity to this question which gets in the way of the lesson: in needing to encode a conditional using nothing but arithmetic and straight-line code, there is a funny statement that uses division in a weird way. This can be nice, as it’s a small cute puzzle in the middle of the exercise. What’s not nice is students needing to unlearn things they know about algebra to do it. But that has the added bonus of helping students think in a more rule-based way, moving towards the point of seeing intuitive systems fall out of rigorous rules, something which comes up again in the Algebraic Refactoring unit, and which lurks beneath the surface in other units.

**Exercise 5** Revisit the code in Figure 3. Notice how it required you to track a lot of information between some of the intermediate lines. How might you be able to reorder the statements to make the code simpler? The precondition and postcondition of the code as a whole should remain unchanged.

**Intent:** This exercise is meant to show how tracking the state needed to prove future things corresponds to intuitive complexity, and justifies how reordering code can simplify it. Even though it’s almost identical to something in the lecture, lots of students miss this, or they reorder the code in a way that breaks semantics (suggesting they’re compartmentalizing this lesson and not unifying the knowledge here with their everyday coding knowledge). Just goes to show you that someone thinking they understand a lecture is often an illusion, and actual practice is like truth serum.

**Exercise 6 Optional**: Consider the following code for a sequential search procedure:

1. Prove this sequential search procedure correct by choosing a proper loop invariant. 2. State the loop invariant you chose

**Intent**: This one actually is there to give people practice dealing with loops in Hoare logic, the hardest part of it. This question actually is more about learning verification than about learning software design, although knowledge of loop invariants is indeed useful for talking about loops / you will sometimes see comments on tricky loops stating what the invariant is. (Jimmy was taught about loop invariants in high school while studying for the AP Computer Science AB test [unsure if it was actually on the test]. He had to memorize “holds before the loop, after the loop, and at the end of each iteration,” though was not taught any mechanical rules for using or applying them. Meanwhile, Bertrand Meyer insists that all undergrads at ETH Zurich taking intro programming must understand loop invariants before they can write loops.)

We can give some leniency in how much detail they give in showing all the rule applications to annotate everything with assertions, but people doing this exercise are doing it for enrichment in learning Hoare logic. They can and should be held to a higher standard for accuracy. Further, they must get the loop invariant correct.