Recommended Design Process for Functions

COMP 105

Spring 2019

Recommended Process

This lightweight process is based on test-driven development methods used in industry. For details, see the handout on programming with proof systems and algebraic laws.

1. Using the descriptions given to you, understand the forms of the data that will be input to the function. Forms might be described by proof rules, by list of cases, or even by a type definition. However they are described, the forms of data must be *distinguishable* by writing code.

2. For each form of data, write an example input which has that form. Every form of data needs an example.

3. If it's not already given to you, choose a name for the function. Use a noun, verb, or property, as described in the general coding rubric, under "Naming."

4. If it's not already given to you, write the function's contract: in a simple, clear sentence, what should the function return, as determined by its argument(s)?

5. Look the example inputs from part 2. On each example, what should the function return? Write your answer as a `check-expect` or `check-assert` unit test.

6. Generalize the example unit tests to algebraic laws. Some values in the examples will turn into variables. This step may be hard.

7. Looking at only the *left-hand sides* of the algebraic laws, start coding the body of the function. The function should begin by asking the input data, "How were you formed?," which tells the code which algebraic law to follow. Code one case per algebraic law. Distinguish cases using `if`-expressions.

8. Finish the function. For each case, ask the input, "From what parts, if any, were you formed?" Then complete the case using the *right-hand side* of the corresponding algebraic law.

9. Revisit the unit tests. First, look at them. Do they test every form of input? If the function's result is Boolean, add new tests so that you have both a "true" and a "false" test for each form of input.

   Next, run them. If there are any test failures, look at the algebraic laws first.

Rationale

Here are reasons for each step.

1. The shape of the data determines the shape of the code. This idea, popularized by Fred Brooks in *The Mythical Man-Month*, applies to many programming languages and paradigms. It has been known since the 1960s.

2. Examples are the easiest place to start, and they are what people learn from.

3. A meaningful name is critical to code review. By writing it early, you clarify what you are aiming for.

4. Contracts aren't just useful in code review: writing the contract first is a form of "design first, code later," which you may have practiced in COMP 40. And the contract can help alert you to a design that is too complex; if your contract isn't simple, your code may be hard to get right.

   If you've been taught to think of a contract only as documentation you write after the fact, you may be surprised at how much you get out of a "contract first" approach.

5. Writing down results on example inputs ensures that we know where we're going. If something is going to be wrong, misunderstood, or confusing, we want to identify it early—for example, before we start coding the wrong function.

   Writing examples as unit tests gives the interpreter the job of checking that everything works as expected—every time. If anything goes wrong with your code, you want the bug to manifest as a failed unit test.

6. Algebraic laws are the single most powerful tool you will learn in COMP 105. They occupy a middle ground between vague English and executable code, where they simplify both coding and review.

7. Case analysis is always the enemy. This step shows you where you *must* have it.

8. This step reduces the coding task to a bite-sized piece involving one case at a time.

9. Adding test cases for both "true" and "false" results finds many bugs, as does actually running the unit tests.