Lab 2: Racket Introduction
12:00 PM, Sep 17, 2017

Contents

1 Am I Blue? 1
2 Function equality 1
3 Test Cases 2
   3.1 Domain edge cases 3
   3.2 Description edge cases 3
   3.3 Generic cases 3
   3.4 Enumerated domains 3
   3.5 Distinguished elements in large domains 3
   3.6 Effective testing 4

Objectives

By the end of this lab, you will know:

- the difference between a function and a procedure
- what constitutes function equality

By the end of this lab, you’ll be able to:

- use Desmos
- write comprehensive test cases

1 Am I Blue?

One of the most important aspects of programming is understanding and being able to describe the inputs and outputs of the procedures you write.

Task: Write a procedure am-i-blue which can only take the strings “red” or “blue”. It should return the string “no” if it is given the string “red” and the string “yes” if it is given the string “blue”. Follow the design recipe guide—available on the course Piazza—when writing this procedure.

Hint: Be careful when describing your input and output.
2 Function equality

Let’s talk about the difference between the terms function and procedure. A function has a domain (or multiple domains, if the function takes in more than one argument) and pairs every element of the domain with an element in its codomain (or multiple codomains if, once again, the function takes in more than one argument). The method by which the function pairs the input and output is called a procedure.

Let’s say you have two functions, $f_1$ and $f_2$, that have the same domain, the reals, and both take in one argument, $x$.

$f_1$ maps its domain to its codomain thus $x \mapsto x + 5$

$f_2$ maps its domain to its codomain thus $x \mapsto (x + 2) + 3$

While these procedures look different, we can see that for any input $n \in \mathbb{R}$, both will produce the same output. For instance, $f_1(2) = 7$ and $f_2(2) = 7$, even though there are computational differences between how $f_1$ and $f_2$ arrive at their respective outputs. Therefore, we consider $f_1$ and $f_2$ to be equal to one another as functions, even though their procedures are different on the surface level.

**Task:** Determine whether the functions $f(x) = x^2$ and $g(x) = x^2 + \sin(\pi x)$ (both defined from $\mathbb{Z}$ to $\mathbb{Z}$) are the same. Explain in words.

**Task:** Are the functions $f(x) = x^2 + 1$ and $g(n) = n^2 + 1$, both defined from $\mathbb{Z}$ to $\mathbb{Z}$, equal? Explain in words.

**Task:** Use Desmos to determine the maximum value (i.e. the $x$ value where $f(x)$ is the greatest) for $f(x) = -(x^2 - 5x + 4)$ where $f : \mathbb{R} \to \mathbb{R}$.

**Task:** Determine the maximum value of $f(x) = -(x^2 - 5x + 4)$ where $f : \mathbb{Z} \to \mathbb{R}$. Does this maximum value differ from the previous task? Explain why or why not.

---

3 Test Cases

Testing your programs helps you find bugs, and (if you write tests before you write the program) can help you avoid introducing bugs in the first place. That’s why we ask you to write tests early in the Design Recipe.

In this course, we don’t aim for perfect testing, or even ideal testing — that’s another whole course in itself — but we do require that you test systematically based on the procedure description. You’re welcome, in general, to have more tests than our approach requires; indeed, this is often a good idea. But you need at least those that the procedure description suggests in ways that we’ll now describe.

What to do is not completely obvious. If you’re writing a program that consumes integers, does that mean you need a test case for every single integer input? To test thoroughly, yes. But that’s impractical. So what can you do? You can rely on the experience of others in testing to pick good examples. This problem is all about picking good examples, and about using examples to evaluate

---

1Is using Desmos going to help you here? Why or why not?
procedures that you (or others) write.

We’ve used the words “example” and “test” almost interchangeably here. An “example” is an input whose output you know from reading the problem description, and examples are part of how we make sure we didn’t make mistakes in writing a procedure. A “test” is a kind of example, an example designed to detect certain errors. So if we’re writing a procedure to turn a string containing a name into a string representing the first name, then turning “John Hughes” into “John” is an example. But the inputs “Prabhat” and “Jean-Paul Belmondo” and “Mary Jane Smith” might all be good tests, the first because the name Prabhat, in which there is only a single name rather than a first-name and last-name, and Jean-Paul, in which the first name contains a non-alphabetic character, and Mary Jane, where the first name contains a blank, all push the edges of the assumptions we may have made in writing the procedure. Indeed, if we think of these examples before we start writing the procedure, it may help us write it better. (To be honest: it’s basically impossible to write this procedure properly, as you can see from names like “Mary Jane Van Slyke”, where both the first and last names are multi-word combinations. Writing tests helps you realize this!)

Here are some guidelines for testing:

3.1 Domain edge cases

Look at your procedure’s domain, and try to identify edge cases. If your domain is the natural numbers, then 0 is an edge case, because it’s the only natural number that doesn’t have two neighbors; it’s at the “edge” of the set. If you’re looking at lists, the empty list is an edge case. If you’re looking at real numbers whose square is no more than 16, then ±4 are edge cases.

You should always include edge cases in your tests.

3.2 Description edge cases

If you’re writing an “absolute value” procedure, the description might be “if the input, x, is negative, return the value −x; if it’s positive, return x; if it’s zero, return zero.” (It might instead break things into “negative” and “nonnegative”.) In this case, although there are items in the domain on either side of 0, it’s clear that 0 is a dividing line between the two kinds of cases of inputs, so it’s an edge-case.

If the domain was restricted to integers, the three classes you’d be considering would be the class containing 0, the class containing 1, 2, 3, . . . , and the class containing −1, −2, −3, . . . . In this case, 1 would be an edge case for the second class, and −1 would be an edge case for the third class, so you’d want to include those as well.

3.3 Generic cases

A generic case is one that’s not distinguished in any way in the domain or the procedure description. If the procedure is supposed to send odd numbers less than 10 to 1, and send any other integer n to n + 1, then a number like 42 is a generic case: it’s not near the edge case values of 10 and 11 .

Similarly, in the absolute value problem above, −12 and 28 are nice generic cases, one in each of the two classes of inputs.
You should include a test of a generic case in each class of input into which the specification divides the domain.

### 3.4 Enumerated domains

When the domain consists of a small fixed number of cases, say fewer than 20, like the A-T-C-G nucleotides in DNA, your test cases should include each one of the cases.

### 3.5 Distinguished elements in large domains

Imagine a procedure called `isLucky`, which consumes a natural number and returns `true` if the number is 7 or 11, and `false` otherwise. Also 7 and 11 are parts of a large group of numbers, they’re distinguished in the problem description, and therefore worth testing individually. (If you didn’t test either of them, you might report as “good” a procedure that simply always returned the value `false`, after all!) For this example, you should also test 0 (a domain edge case) and some other integer like 37 (a generic case).

### 3.6 Effective testing

Suppose that three people all have to write the `isLucky` procedure. You have to test them, and identify which procedures seem to be good and which seem to be bad. This is a classification problem, and testing is a way to approach it. We’re trying to separated the good programs from the bad, so we refer to them as “wheat” and “chaff”. A good test suite is one where each wheat program passes all the tests and fails none of them, but each chaff program fails at least one of the tests. If a chaff program passes all the tests, it’s a false positive: something that looks good but isn’t. If a wheat program fails one of the tests, it’s a false negative: something that looks bad, but isn’t.

Your test suites should never produce false negatives. If your test for absolute value, for instance, said that you expected the absolute value for the input $-5$ to be 6, then a correct absolute-value program would fail the test. That makes it a bad test.

Your test suites may, on the other hand, produce false positives, especially when the domain of application is infinite. For instance, an absolute value program might have the following logic:

```plaintext
if the input N is negative, return -N
otherwise, if N is 289183, return 6
otherwise, return N
```

Unless you check that this program works for the argument 289183, you’re not going to discover that the logic is incorrect. If you have the program in front of you, that’s easy to do. If you don’t, you just have to build a test suite using the rules above and hope that it catches almost all errors.

The rules suggested above are based on experience: many programming errors arise at edge cases of various sorts, and the non-edge-case errors tend to result in bad outputs for many generic cases.

---

2When wheat is harvested, there’s a good part that you want to keep, and a bad part — stems, hulls, etc., — that you want to throw away. This latter part is called “chaff”, hence the phrase “separating the wheat from the chaff.” The terms “spam” and “ham” are sometimes used with similar meanings, but they have the disadvantage of sounding alike, so it’s easy to confuse them.
rather than just one specific one, as in the contrived example I just wrote.

We’ve written two procedures, less1 and less2. The specification for the first is this:

```scheme
;; less1: int * int -> bool
;; Input: an integer x, an integer y
;; output: true if x is less than y; false otherwise

;; less2: int * int -> bool
;; Input: an integer x, an integer y
;; output: true if x is less than y; false otherwise
```

The first thing you need to do is copy over the source code. Navigate to your lab02 directory, and run the following command in your terminal:

```
cp -r /course/cs0170/src/racket/less-functions/compiled .
```

In order to be able to access these procedures, include the following line at the top of your Racket file:

```
(require "less-functions.rkt")
```

This will give you access to the procedures you need to test, but not their implementations.

You can now call the procedures by their names. For example, `(check-expect (less1 2 3) true)` will check if the output of less1 on inputs 2 and 3 will output true.

Note that these two procedures are defined to do the same thing! Your task is to tell the two apart, and discern which one is the faulty solution. The actual definitions of these procedures are hidden from you, but you can test them out in DrRacket by typing things like `(less1 41 1)` and seeing that the value produced is false, because 14 is not less than 1.

**Task:** Write a test-suite (i.e., a sequence of check-expect expressions), guided by the test-generating rules above, to try to tell whether the TA implementations of each of these two procedures are wheat or are chaff.

You’ll be doing something like this for almost every procedure you write in Racket; we’ll continue to use the wheat/chaff metaphor for testing throughout the course.

Once a lab TA signs off on your work, you’ve finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you’ve just written.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback)