Lab 1

0. Goals

Relevant sections of text: *HtDP/2e Prologue and Chapters 1–3*

This week's lab will help you to practice:

- using DrRacket: interactions vs. definitions; stepper; error messages; indentation
- simple image functions; using the Help Desk
- function composition: write more complex functions by combining simpler functions.

Note: Exchange email addresses and phone numbers with your partner. Decide on your next meeting time this week in case you don't finish. Make sure **both** names are at the top of your lab submission!

1. Using DrRacket

Problem 1.

You may remember from math class that the distance from the origin point $(0, 0)$ to any point (x, y) in the cartesian plane is

$$\mathsf{distance}(x, y) = \sqrt{x^2 + y^2},$$

which is a function composed from other, more basic math functions (addition, squaring, and square root).

This function can be written in BSL as follows:

```
;; distance: Number Number -> Number [This is the SIGNATURE]
;; Consumes: [PURPOSE STATEMENT incl. parameter descriptions]
;;   Number x : x-coordinate of a point in 2d-plane
;;   Number y : y-coordinate of the point
;; Produces: the distance from origin to point (x,y).

(define (distance x y)
  (sqrt (+ (* x x)
           (* y y))))
```

Follow the step-by-step instructions below.

- Copy this definition to the DrRacket **definitions** window (the top window).
- Press Run. Notice which parts of the code are still in black/orange—those have not yet been **tested**.
- **Call** the function (i.e., "run the program") by typing `(distance 3 4)` at the prompt in the **interactions** window (lower window). Notice that the entire function definition is now tested. (Whether the answer is correct is a different issue :-)
- Now call the function to find the distance from the origin to the point $(5, 5)$. (**Note**: you can hit escape followed by `p` to have DrRacket scroll backwards through the previous expressions that you typed at the interactions prompt.) Do you recall that `#i` indicates that the result is an *inexact* number?

- **Testing**: to properly test a function, there should be at least one test per function. (We'll be even more specific about this next week.) The black/orange coloring when you press Run indicates that that part of your code has not been tested. Tests on individual functions like this are called *unit tests*. We will typically write the tests (as examples) **BEFORE** writing the function. (After all, if you don't know what the function is supposed to do, how can you possibly write the body?! Review Section 3.5 *On Testing*.) The most common unit testing function is `check-expect`. For example, in the definitions pane add:

```
(check-expect (distance 0 0) 0) ; the distance from (0,0) to (0,0)
                                ; should be 0
```

  However, `check-expect` can only be used for checking exact equality of things. It will not work for inexact numbers. Try it: to the definitions pane add the following and press Run:

```
(check-expect (distance 5 5) 7.071)
```

  Note the error, note what is highlighted. The Unit Test Results window may be closed, or "docked" to the definitions and interactions panes. (When "docked" there will be three panes, not two).

  To test functions that return inexact numbers, use `check-within`. This unit testing function lets you say how close the answer must be to pass the test. Is within 0.1 close enough? 0.000001? For example,

```
(check-within (distance 5 5) 7.071 0.0001)
     ; result must be between [7.0709 and 7.0711], inclusive
```

  Write two unit tests for the distance function, one with `check-expect` and one with `check-within`.
- Call the function in the interactions window on incorrect arguments: `(distance 4 "red")`. Read and understand the error message!! What is highlighted in the definitions window? What if you type `(distance true 10)` in the interactions pane?
- **Using the Stepper**: Put `(distance 3 4)` in the **definitions** window at the bottom. Press the Step button. This will bring up a separate window which lets you see how DrRacket is evaluating your program. Step through the program, making sure that you can correctly predict what the next step will be each time. Note that this is just simple algebraic substitution that you have used since middle school. Use the stepper anytime you are confused about what your function is doing.
- Save your program and tests as `lab1.rkt` using the **Save Definitions** option in the file menu. There is nothing else to turn in for problem 1.

Problem 2.

Reread HtDP/2e Section 2.3, *Composing Functions*. Included are two solutions to the following sample problem:

> **Sample Problem** The owner of a monopolistic movie theater in a small town has complete freedom in setting ticket prices. The more he charges, the fewer people can afford tickets. The less he charges, the more it costs to run a show because attendance goes up. In a recent experiment the owner determined a relationship between the price of a ticket and average attendance.
>
> At a price of $5.00 per ticket, 120 people attend a performance. For each 10-cent change in the ticket price, the average attendance changes by 15 people. That is, if the owner charges $5.10, some 105 people attend on the average; if the price goes down to $4.90, average attendance increases to 135. Let us translate this idea into

a mathematical formula:

$$\textit{avg. attendance} = 120\ \textit{people} - \frac{\$(\textit{change in price})}{\$0.10}\ \cdot\ 15\ \textit{people}$$

Stop! Explain the minus sign before you proceed.

Unfortunately, the increased attendance also comes at an increased cost. Every performance comes at a fixed costs of $180 to the owner plus a variable cost of $0.04 per attendee.

The owner would like to know the exact relationship between profit and ticket price so that he can maximize his profit.

The first solution follows the correct design philosophy of defining one function per task:

```
;; attendees: Number -> Number
;; Consumes:
;;   Number ticket-price : the price of one ticket in dollars
;; Produces:  the number of people who will attend at that price
(define (attendees ticket-price)
  (+ 120 (* (/ 15 0.1) (- 5.0 ticket-price))))

;; revenue: Number -> Number
;; Consumes:
;;   Number ticket-price : the price of one ticket in dollars
;; Produces: the revenue generated, i.e. ticket price times
;;   the number of attendees.
(define (revenue ticket-price)
  (* (attendees ticket-price) ticket-price))

;; cost: Number -> Number
;; Consumes:
;;   Number ticket-price : the price of one ticket in dollars
;; Produces: the costs incurred; costs have both a fixed [$180] and
;;   variable [$.04 per attendee] part.
(define (cost ticket-price)
  (+ 180 (* 0.04 (attendees ticket-price))))

;; profit: Number -> Number
;; Consumes:
;;   Number ticket-price : the price of one ticket in dollars
;; Produces: the profit: revenues minus costs.
(define (profit ticket-price)
  (- (revenue ticket-price)
     (cost ticket-price)))
```

Problem 2.1: Determine the potential profit for the following ticket prices: $1, $2, $3, $4, and $5. Which price should the owner of the movie theater choose to maximize his profits? Determine the best ticket price down to a dime. Place the answers as a comment in your `lab1.rkt` file.

The alternative, poorly-thought-out definition is as follows:

```
;; No signature and purpose statement?
;; Lose most points for this function!
(define (profit.bad price)
```

```
(- (* (+ 120
         (* (/ 15 0.1)
            (- 5.0 price)))
      price)
   (+ 180
      (* 0.04
         (+ 120
            (* (/ 15 0.1)
               (- 5.0 price)))))))
```

Problem 2.2: After studying the cost structure of a show, the owner discovered several ways of lowering the cost. As a result of his improvements, he no longer has a fixed cost. He now simply pays $1.50 per attendee. Modify both programs to reflect this change. (You do not have to keep the originals). When the programs are modified, test them again with ticket prices of $3, $4, and $5 and compare the results. Add at least one unit test example for each function. Notice that DrRacket does not register your definition window edits in the interactions window until you press Run. (You may wish to try running the program before and after you press Run to see what happens.)

2. String Arithmetic and the Help Desk

You may want to re-read HtDP/2e sections 1.2 and 1.3. Just as we can compose `+`, `-`, `sqrt`, `sin`, etc. for numbers, computation as the "algebra of things" allows us to compose functions on strings, images, boolean truth values, and any combination. Although you (I hope) have spent 12 years learning the common arithmetic number operations, computer languages tend to define their own operations on strings and images. (Even if your high school didn't teach them to you, mathematicians also named all the boolean operations, e.g., `and`, `or`, and `not`; these will come into play more next week). The book describes several string operations:

- `string-append` concatenates two given strings into one. Think of `string-append` as an operation that is just like `+`. While `+` consumes two (or more) numbers and produces a new number, `string-append` consumes two (or more) strings and produces a new string;
- `string-length` consumes a string and produces a (natural) number;
- `string-ith` consumes a string s and extracts the one-character substring located at the i^{th} position (counting from 0);
- `number->string` consumes a number and produces a string;
- `substring` ... hmmm, I wonder what this does? Look this up in the **Help Desk** and find out what it does. If the documentation appears confusing, experiment with the function in the interaction area.

The Help Desk: Access the help desk from the menu Help→Help Desk. This will bring up your web browser. Look under Teaching→How to Design Programs Languages. This will bring up all the teaching languages, and we are currently using BSL (Beginning Student Language). Click on the appropriate topic, e.g., "Strings", and scroll down until you find an entry for the function you are interested in (`substring`). Alternatively, you can type `substring` into the search box *[...search manuals...]*; make sure to only click on the result that is provided from `lang/htdp-beginner` for now!

Problem 3. Define the function `string-first`, which extracts the first character from a non-empty string. Don't worry about empty strings. The signature for this function is

```
;; string-first: String -> String
```

From now on we assume that without being reminded, you will have a contract, purpose statement, and at least one example/unit test for each function that you write. We will discuss testing in more detail next week.

Problem 4. Define the function `string-last`, which extracts the last character (as a string) from a non-empty string. Don't worry about empty strings.

Problem 5. Define the function `string-join`, which consumes two strings and appends them with "_" in the middle.

Problem 6. Define the function `string-insert`, which consumes a string and a number i and which inserts "_" at the i^{th} position of the string (counting from the left, starting at 0). Assume i is a number between 0 and the length of the given string (inclusive). Also ponder the question whether `string-insert` should deal with empty strings. If we insert "_" at position 5 in the string `helloworld`, we expect the result `hello_world`. We can write this example explicitly as a unit test:

```
(check-expect (string-insert "helloworld" 5) "hello_world")
```

Make sure that all of these functions and unit tests are included in your file `lab1.rkt`.

3. Images and More Function Composition

See HtDP/2e section 1.4, ...

In class we will play with some built-in image-processing functions. To use them, you will load a library of previously defined functions by starting your definitions file with the line

```
(require 2htdp/image)
```

From the Help menu, choose Help Desk. Under Teaching you will see How to Design Programs Teachpacks—click on it. Then click on HTDP/2e Teachpacks in the upper left menu. In Sections 2.2 (Image Guide) and 2.3 (Images: `image.rkt`) you will find a complete description of all the functions in the `image` library. Yes, some of the concepts there have not been covered yet in class, but at this point the examples should be mostly clear and you should feel free to experiment in the DrRacket Interactions pane. Make it a habit to try to answer your own questions using the Help Desk before asking the TA or LA.

Here is an expression that will display a solid red circle of radius 10:

```
(circle 10 "solid" "red")
```

Notice that this conforms to the definition of a compound expression. (The operator/function name is `circle`. The three arguments supplied to the `circle` operator are a number, 10, and two strings, `solid` and `red`.)

Problem 7. Use multiple image primitives, composed together, to create an image of a simple boat of your own design (have fun). Since the boat is always the same, define it as a constant:

```
;; a very unimaginative boat image using only one primitive
(define MY-BOAT (ellipse 60 20 "solid" "Chartreuse"))
```

Problem 8. Define the function `image-area`, which computes the area of a given image. **Note:** The area is also the number of pixels in the picture.

DrRacket can import images (e.g., from the web), either by cut-and-paste or the Insert→Image... menu item. Consider an image like this:

This can be inserted in DrRacket by copying and pasting into a definition:

```
(define PALM  )
```

Problem 9. Define the function `pinwheel` that consumes an image and produces a new image consisting of four copies of the image, each rotated around the center. For example, `(pinwheel PALM)` should produce something like:

Your function should work on `PALM`, `MY-BOAT`, or any image you pass in as an argument. For your unit tests, use any images you think appropriate.

Problem 10. Define an image `BACKGROUND`, starting with a 500×200 empty-scene, and adding some water (e.g., a `LightSeaGreen` rectangle) and an island with a palm tree.

Problem 11. Define a function `draw-boat` that consumes a natural number (interpreted as the number of clock ticks that have gone by) and produces an image of `MY-BOAT` on top of the `BACKGROUND`, where the boat moves 3 pixels every clock tick, and when the boat leaves the right side it simply reappears on the left and continues. (*Hint:* look up the definition of the arithmetic operation `modulo` in the Help Desk; consider `(modulo 501 500)`, `(modulo 502 500)`, etc.)

Make sure the line `(require 2htdp/universe)` occurs in your definitions area. Then add this as the final line in your definitions: `(animate draw-boat)`. When you click Run, an animation window should pop up with your background and boat, and the boat should begin moving across the background, wrapping around forever. Close the animation window and your unit tests will complete as usual.

Turn in your single file `lab1.rkt` file with all of your definitions, documentation, and tests from this lab using Sakai. **MAKE SURE THAT SAKAI PROVIDES YOU WITH A RECIEPT!!!** No receipt, no grade!