

Lab 2

Goals

Relevant sections of HtDP/2e: Chapters 3 and 4.

This week's lab will help you to practice:

- conditionals, intervals, enumerations, simple itemizations
- function composition: writing more complex functions by combining simpler functions
- using the design recipe!
- simple interactive programming using the universe library

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!

Batch Program Problems

Note: Follow the Design Recipe! For each problem, consider if you need a *Data Definition* more precise than just Number, String, Boolean, or Image. What is the function *Signature*? The *Purpose*? Write at least one *Example / Unit Test* for every different **kind** of data consumed and produced. Create the function *Header* with a body *stub* that is an atomic value of the correct output type. Only now do you need to consider how to actually write the function body (and if the inputs are enumerations, then the body should say, case-by-case, what to do for each possible value using `cond`). Don't be scared of a blank Definitions Window, and be aware that we grade each of these steps separately.

Body Mass Index

Problem 1: Body Mass Index (BMI) is a number calculated from a person's weight and height that is a fairly reliable indicator of body fatness for most people. [http://www.cdc.gov/healthyweight/assessing/bmi/adult_bmi/index.html]

Design a function `bmi` that consumes a person's weight in pounds and height in inches, and computes that person's BMI. *Make sure to carefully document your intended interpretation of the function parameters.*

Follow the Design Recipe! We expect EVERY function you write, from now on, with few exceptions, to include the signature, purpose statement, and examples turned into unit tests, in addition to the necessary function header and body.

Problem 2: For adults 20 years old and older, BMI is interpreted using standard weight status categories that are the same for all ages and for both men and women. For children and teens, on the other hand, the interpretation of BMI is both age- and sex-specific. The standard weight status categories are underweight, normal, overweight, and obese. Write a **data definition** for `WeightStatusCat` as an **enumeration**.

Problem 3: The standard weight status categories associated with BMI ranges for adults are shown in the following table:

| BMI | Weight Status |
|---------------------------------|---------------|
| Below 18.5 | Underweight |
| 18.5 up to but not including 25 | Normal |
| 25 up to but not including 30 | Overweight |
| 30 and above | Obese |

Design a function `bmi->wsc` to convert a BMI value to a standard weight status category.

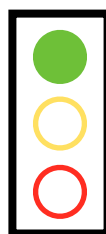
Traffic Light: Enumeration

Problem 4: A basic 3-bulb traffic light has three states, “red”, “yellow” and “green”. Write a **data definition** for a `TrafficLightState` (an **enumeration**).

Problem 5: Design a function `render-traffic-light` that consumes a `TrafficLightState` and produces an `Image` of the corresponding light. For example,

`(render-traffic-light "green")`

might display something similar to




WARNING: you must write your unit test(s) to use actual drawing functions, NOT an inserted image when unit testing functions that produce images. For example:

;; Good

```
(check-expect (my-hat "blue")
               (above (circle 10 "solid" "blue")
                       (rectangle 30 5 "solid" "blue")))
```

;; Bad

```
(check-expect (my-hat "blue")  );don't paste images in unit tests!
```

Traffic Light: Intervals

Problem 6: A basic traffic light timer takes on values in the *natural numbers* (the “counting numbers” starting at 0) from 0 to 100, inclusive. The interpretation is as follows: times 0 to 50 indicate a green light; times 51 to 65 a yellow light, and 66 to 100 a red light. Write a **data definition** for a `TrafficTimerValue` (an **itemization** of *intervals*).

Problem 7: Design a function `increment-timer` that consumes a `TrafficTimerValue` and produces the next `TrafficTimerValue`. Recall that the timer resets to 0 after reaching 100. There are two possible solutions to this problem (one using `cond`, and one using `modulo` or `remainder`); use either solution.

Problem 8: Design a function `render-traffic-timer` that consumes a `TrafficTimerValue` and produces an image of the attached traffic light with the correct bulb lit. *Hint: you may want to use some of the functions that you already wrote and tested earlier in the lab.*

Traffic Light: Simple Itemization

A traffic light can run on a timer, but many also have a manual override. A basic traffic light world state can be **either** a timer value (a natural number from 0 to 100 inclusive) OR a constant manual traffic light state indicating steady green, yellow, or red (a String enumeration). **The complete data definition is given as follows** [this should already be in your Solution File]:

```
;; A TrafficLightWorld is either
;; - a TrafficTimerValue [a Number interval], or
;; - a TrafficLightState [a String enumeration]
```

Now we are ready to build an **interactive** program: a simple traffic light simulator.

Interactive Program Problem

Let's actually build the traffic light simulator using the Universe package. An interactive program consists of several parts: an underlying computational **model** of the world, a mapping from that model to a **view** that people can see (here, a graphical view), and **controllers** that interact with people (or other things, like the computer's clock) to change the model.

At this point our model can only be a simple piece of atomic data or simple itemization (until next week!). **The model must represent what changes; what the user manipulates.** Here, our **model** is a `TrafficLightWorld`, which represents both a traffic light on a changing timer, **or** one that is controlled manually with the keyboard.

Interactive programs are created using the big-bang form, which in turn requires us to write several different functions with *pre-specified signatures*. We'll take these one at a time.

For this section, we use the following Data Definition (from above):

```
;; A TrafficLightWorld is either
;; - a TrafficTimerValue [a Number interval], or
;; - a TrafficLightState [a String enumeration]
```

Problem 9: Design a function `render-traffic-world` that consumes a `TrafficLightWorld` and produces an `Image` of the attached traffic light with the correct bulb lit. You may want to use some of the functions that you already wrote and tested. This function, `render-traffic-world`, maps the **model** to a **view**.

You may now define a function `main` that consumes an initial `TrafficLightWorld` and calls `big-bang` to create a new universe as follows:

```
;; main : TrafficLightWorld -> TrafficLightWorld
;; The main function consumes the initial TrafficLightWorld
;; and calls big-bang to create the universe.
(define (main init)
  (big-bang init ;the init parameter has type TrafficLightWorld
    (to-draw render-traffic-world))) ;TrafficLightWorld --> Image
```

If you load your definitions and call, for example, `(main 0)` or `(main "red")` you should get the correct traffic light image. Now to add some interaction using the keyboard!

Note: DO NOT write unit tests for main.

When you press a key, a key-event is generated by the keyboard **controller** that can be used to “change” the world (the **model**). For example, we could set it up that pressing “g” changes the state to “green”, pressing “r” changes it to “red”, “y” changes it to “yellow”, and “t” starts the timer at 0. Note that the key **controller** affects only the **model**, not the **view** (you already wrote a function that draws whatever the current **model** is on the screen: the **model** and the **view** are separate)!

Problem 10: Design a function `handle-key` that consumes a `TrafficLightWorld` and a `KeyEvent`, and produces a new `TrafficLightWorld` representing the new state. So for example

```
(check-expect (handle-key 55 "g") "green")
(check-expect (handle-key "red" "t") 0)
;; ignore any key that is not "g", "r", "y", or "t":
(check-expect (handle-key 77 "w") 77)
```

Note that you should compare keys using `(key=? akeyevent "5")`, and not `string=?` (see if you can figure out why).

Now modify `main`, adding a new clause for the key controller handler:

```
(define (main init)
  (big-bang init ;TrafficLightWorld
    (to-draw render-traffic-world) ;TrafficLightWorld --> Image
    (on-key handle-key))) ;TrafficLightWorld KeyEvent --> TrafficLightWorld
```

Now if you run `(main 0)` in the interactions window, you should be able to change the Light by pressing keys.

Finally, let's get the timer working.

Problem 11: Design a function `handle-tick` that consumes a `TrafficLightWorld` and produces a new `TrafficLightWorld`. If the `TrafficLightWorld` is a `TrafficTimerValue` (a Number), then increment it appropriately. If the `TrafficLightWorld` is a `TrafficLightState` (a String, indicating that the light is being manually controlled) then return the same value, changing nothing.

Now modify `main`, adding a new clause for the tick handler:

```
(define (main init)
  (big-bang init ;TrafficLightWorld
    (to-draw render-traffic-world) ;TrafficLightWorld --> Image
    (on-tick handle-tick) ;TrafficLightWorld --> TrafficLightWorld
    (on-key handle-key))) ;TrafficLightWorld KeyEvent --> TrafficLightWorld
```

Now if you run `(main 0)` in the interactions window, you should see the light change on its own timer AND be able to change the light by pressing keys.