Lab 2

Goals

Relevant sections of HtDP/2e: Chapters 1, 2, and 3 and Chapter 4 Sections 1.

This week's lab will help you to practice:

- conditionals, intervals, enumerations
- 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.

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 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->wvc` to convert a BMI value to a standard weight value category.

Problem 4: The Psychology department has purchased a sign to advertise how many students are needed for experiments today. The sign has a keypad to enter a number, but the actual text displayed on the sign must be programmed using a function that consumes a number and produces a string to display.

Unfortunately, the first version of the code has a problem. It works fine for any positive integer, or even 0 except for the number 1:

```
2  students are needed for experiments
22 students are needed for experiments
0  students are needed for experiments
1  students are needed for experiments
```

They would prefer it say “1 student is needed for experiments”. Design the function `psych-sign` to consume a non-negative integer and produce the correct string.

Problem 5: The local student chapter of the Association for Computing Machinery (one professional organization for computer scientists) really likes your sign, and asks for one customized for them. Design a function `acm-sign` that also consumes a non-negative integer, but produces either “n students join the ACM” or “1 student joins the ACM”.

Problem 6: Abstraction. You are now getting 10 sign requests every day. You decide to write **one** program to handle all of these signs, and even some you haven’t thought of yet. You look at your examples for patterns (we’ve added space just to make it more obvious):

2	students	are	needed for experiments
1	student	is	needed for experiments
22	students	join	the ACM
1	student	joins	the ACM
12	buses	are	leaving now
1	bus	is	leaving now
5	burgers	are	ready to eat
1	burger	is	ready to eat
16	fish	are	free with any purchase
1	fish	is	free with any purchase

The pattern you discover is:

<number> <noun> <verb> <phrase>

You initially consider designing a function that consumes a number, singular noun, plural noun, singular verb, plural verb, and the verb phrase, but realize that most of the time you can figure out the plural form of the noun and the verb given only the singular form, and the irregular cases you can handle as extra clauses in a conditional statement.

Because one of the most basic Rules of Program Design is

“ONE TASK, ONE FUNCTION, ONE PURPOSE”

and we have ***three*** tasks (pluralizing the nouns, pluralizing the verbs, and putting the whole sign together either with or without plurals), we will therefore design ***three*** functions.

- (6a) Design a function `any-sign` that consumes a non-negative integer, a singular noun string, a singular verb string, and a verb phrase string and produces the appropriate correct English output string depending on the value of the integer (1 means singular, and anything else is plural). This function should work for all the examples above, and call the following two helper functions.

- (6b) Design a function `pluralize-noun` that consumes a singular noun string and produces the plural. It should work for regular nouns (add an "s" at the end), regular nouns that end in "s" (add "es"), and the special cases "fish" and "deer" that are their own plurals. Note that new special cases can be added easily. Feel free to use functions you wrote for lab 1.
- (6c) Design a function `pluralize-verb` that consumes a singular verb string and produces the plural. It should work for regular verbs (remove the trailing "s") and the irregular verb "is" (plural "are").

When you have Problem 6 working, it is Best Practice to refactor [rewrite] your solutions for Problems 4 and 5 to use your new more abstract function: comment out your old definitions for Problems 4 and 5, and replace them with new definitions that call only our new, more general solution, `any-sign`.

Interactive Program Problem

Let's actually build the Psychology dept. sign 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 (wait until next week!) or simple itemization. The model must represent what changes; what the user manipulates. Here, that will be the number of students needed today. The number of students needed gives us the information to create a view, and the user can use the number keys to change the number displayed.

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 (copy into your solution file):

```
;; a PSWorld [Psych Student World] is a Non-Negative Integer,
;; representing the number of students we need today for experiments
```

Problem 7: Design the function `draw-psych-sign` that consumes a `PSWorld` and produces an Image of the `psych-sign` string (read that twice: consumes a `PSWorld` and produces an `Image` using the `text` function), at a large font size on top of an appropriate background. The font size, color, and background should all be defined a constants so that they are easy to change. This function maps the ***model*** to a ***view***.

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

```
;; The main function consumes the initial PSWorld
;; and calls big-bang to create the universe.
(define (main init)
  (big-bang init ;PSWorld
            (to-draw draw-psych-sign))) ;PSWorld --> Image
```

If you load your definitions and call, for example, `(main 5)`, you should get the correct sign. Now to add some interaction using the keyboard.

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 "1" changes the number of students to 1, pressing "7" changes it to 7, and so on for all 10 single digit keys. Note that the key ***controller*** effects 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 8: Design a function `handle-key` that consumes a `PSWorld` and a `KeyEvent`, and produces a new `PSWorld` representing the new number of students (forgetting whatever the number was beforehand). So for example

```
(check-expect (handle-key 5 “8”) 8)
(check-expect (handle-key 5 “g”) 5) ;ignore anything not a digit
```

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:

```
;; The main function consumes the initial PSWorld
;; and calls big-bang to create the universe.
(define (main init)
  (big-bang init ;PSWorld
            (to-draw draw-psych-sign) ;PSWorld --> Image
            (on-key  handle-key)))    ;PSWorld KeyEvent --> PSWorld
```

Now if you run `(main 5)` in the interactions window, you should be able to change the number in the display by typing keys.

Problem 9: Redesign `handle-key` so that the up arrow (key event "up") adds one to the current PSWorld, and the down arrow (key event "down") subtracts one.

Problem 10: Redesign handle-key so that if we type multiple numbers in a row we get larger numbers: that is, typing 435 would give a display "435 students are needed for experiments". Hint: if our world is 0 and we type 3, the new world is 3. If the world is 3 and we type 4, the new world is 34. If the world is 34 and we type 9, the new world is

349, etc. the new world is 10 times the old world plus the key typed. Add a clause that typing the space bar (key event " ") sets the world back to 0.

Problem 11: look up in the help desk how to add support for the mouse ***controller***. You will need to design a `handle-mouse` function, and add `(on-mouse handle-mouse)` to `big-bang` in your `main` function. Design the behavior that when you click the mouse, the `PSWorld` is set back to 0 (same as space bar).