CS 2500, Lab 8—Abstraction and Loop Functions

- Work in pairs
- Change roles often!
- Follow the design recipe and/or the abstraction recipe for every problem.

We're going to design some nice abstract functions and have some more fun with Posns, but before we do let's make sure we can use more abstract data definitions.

Part I:Polymorphic (A.K.A. Parametrized) Data Definitions

We start with a `List` warmup. Here's are usual data definitions for `List`s:

```
;; A(n) LoN is one of
;; - empty
;; - (cons Number LoN)

;; A(n) LoS is one of
;; - empty
;; - (cons Symbol LoS)
```

Exercise 1: What are the contracts of `cons`, `append`, and `length` for each of these definitions? Hint: the contracts of `list` would be something like this:

```
;; list :  Number ...  Number -> LoN

;; list :  Symbol ...  Symbol -> LoS
```

Exercise 2: Abstract these two data definitions into a single definition for `[Listof X]`.

Exercise 3: What would the more general contracts of `cons`, `append`, and `length` be now? Hint: the general contract of `list` would be something like this:

```
;; list :  X ...  X -> [Listof X]
```

Part II: Abstracting Functions

First, we abstract plain *values* within function definitons.

Exercise 4: Design a function that consumes a `[Listof Number]` and checks if 0 is in the list.

Exercise 5: Design a function that consumes a `[Listof Number]` and checks if 5 is in the list.

Exercise 6: Abstract the previous two functions and design a function that consumes a `[Listof Number]` and a `Number` and checks if the number is in the list.

**** Make sure your new function has a precise contract...**

Exercise 7: Comment out the bodies of your previous functions and rewrite them using your abstraction. Much shorter (and useful) right?

As we saw in lecture, functions are just a special type of value. *We'll say that again*: **functions are just a special type of value!**

Now we abstract over values again, but this time they will be *functions*.

Exercise 8: Design a function that consumes a `[Listof Any]` and removes all the `Symbols` from the list. How do you know when something is a `Symbol`?

Exercise 9: Design a function that consumes a `[Listof Any]` and removes all the `Numbers` from the list.

Exercise 10: Abstract these two into a single function, called `filter-out`, that takes a predicate function to determine which elements to remove from the list.

Hint: The contract of your abstract function should look something like this:

```
;; filter-out :  [Listof Any] [Any -> Boolean] -> [Listof Any]
```

Though you could also change the order of the parameters.

Part III: DrRacket's "*loop*" functions

DrRacket has built-in functions to help us write functions that deal with lists. (See the HTDP link here.)

For the functions below, remember that DrRacket has the functions `odd?` and `even?` built-in, both are `[Number -> Boolean]`

Exercise 11: Design a function `all-odd?` that takes a `[Listof Number]` and returns `true` if all the numbers in the list are odd, and `false` otherwise. *Hint*: use `andmap`.

Exercise 12: Design the same function, call it `all-odd-2?`, but use `ormap` this time. *Hint*: if *all* the numbers are odd, then *none* of them are even, right?

Exercise 13: Design the function `range` that takes two numbers (say n and m) and returns a list of all numbers from n to $m-1$ (inclusive). *Hint*: use `build-list`, and you'll need to create a helper.

Exercise 14: Using your function `range`, design the function `evens` that takes two numbers, and returns a list of all the even numbers in that range. Use `filter`.

Exercise 15: Using `foldr` or `foldl`, implement the function `sum` that computes the sum of all the elements in a list of numbers.

Study this function definition:

```
;; minus-all :  Number [listof Number] -> Number
;; Subtract all the numbers in the list from the given one
(define (minus-all n lon)
   (foldl - n lon))

(check-expect (minus-all 20 empty) 20)
(check-expect (minus-all 20 (list 5 2)) 13)
(check-expect (minus-all 20 (list 5 4 3 2 1)) 5)
```

Exercise 16: Why doesn't this function work? Fix the function so that it produces the correct results. *Hint*: subtraction is not *commutative*... i.e., it is order dependent. Use `local`.

Part IV: Fun with `local`, and Loop functions

The goal of this part of the Lab is to use the ISL loop functions (e.g., `map`, `foldr`...) to do cool stuff (for a Computer Science student's definition of *cool*).

Here are some definitions to get you going...

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

;; Scene Width and Height...
(define WIDTH 400)
(define HEIGHT 400)

;; A Planet is:
```

```
;; (make-planet Number Number Number Number String)
(define-struct planet (x y vx vy color))
;; x,y represent the planet's current location, and vx,vy represent
;; its velocity (speed/direction)

;; Number of colors
(define NUM-COLORS 3)

;; color-for :  Number -> String
;; Return a color for the given number
(define (color-for n)
   (cond [(= n 0) "red"]
         [(= n 1) "green"]
         [else "blue"]))
```

A Planet represents an object (presumably in space) that will act and react to other objects. Our *World State* will be a `[Listof Planet]`.

While we're here... add some more colors if you want.

Exercise 17: Design the function `move-all` that moves a `[Listof Planet]` in the speed/direction each is headed. This means creating a new `Planet` with $new\text{-}x = x + vx$ and $new\text{-}y = y + vy$. Don't change the velocities.

Create a `local` function that moves a single `Planet`, and use `map`.

Exercise 18: Design the function `draw-lop` that draws a `[Listof Planet]` in an `empty-scene`.

Create a `local` function that adds a `Planet` to a `Scene`, and use `foldr`.

Now for the brain-bender... Here's some more code that does the math for you. See if you can understand what it's doing:

```
;; distance :  Planet Planet -> Number
;; Calculate the distance between the Planets
(define (distance p1 p2)
   (sqrt (+ (sqr (- (planet-x p1) (planet-x p2)))
            (sqr (- (planet-y p1) (planet-y p2))))))

;; apply-gravity :  Planet Planet -> Planet
;; Apply the gravitational effects of the other Planet to the
;; second Planet (note the order of the arguments...)
(define (apply-gravity p-other p)
   (local [(define dist (distance p p-other))
           (define dx (- (planet-x p) (planet-x p-other)))
           (define dy (- (planet-y p) (planet-y p-other)))]
```

```
(cond [(< dist 1) p]
      [else (make-planet (planet-x p) (planet-y p)
                         (- (planet-vx p) (/ dx dist))
                         (- (planet-vy p) (/ dy dist))
                         (planet-color p))]))
```

Exercise 19: Design the function `gravity-one` that uses `apply-gravity` to apply the gravitational effects of the `[Listof Planet]` to a single `Planet`. Here's what your contract should be:

```
;; gravity-one :  Planet [Listof Planet] -> Planet
```

Use use `foldr`. Hint: No helper required!

Exercise 20: Design the function `gravity-all` that uses `gravity-one` to apply the effects of all `Planets` to all the `Planets`. Your contract should be:

```
;; gravity-all :  [Listof Planet] -> [Listof Planet]
```

Create a `local` function that calls `gravity-one`, then use `map`. Make sure your contract matches up with the general contract of `map` (just ask if you need help).

Yes! Now we're all set to roll. Here's the `Big-Bang` code to finish it all off.

```
;; tick :  [Listof Planet] -> [Listof Planet]
;; Apply gravity, then move all the Planets
(define (tick lop)
   (move-all (gravity-all lop)))

;; mouse :  [Listof Planet] Number Number String -> [Listof Planet]
;; Add a new planet where the mouse was clicked
(define (mouse lop x y me)
   (cond [(string=?  me "button-down")
          (cons (make-planet x y 0 0 (color-for (random NUM-COLORS)))
                lop)]
         [else lop]))

;; Start with no planets...
(define last (big-bang empty
             (on-mouse mouse)
             (to-draw draw-lop)
             (on-tick tick 1/20)))
```

Enjoy!