CS 2500, Lab 7—Natural number recursion

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

Part I: Recursion over natural numbers

A recursive data structure we use very often in programming is the collection of natural numbers:

```
;; A Nat (natural number) is one of:
;; - 0
;; - (add1 Nat)
;;
;; 0 predicate:  zero?
;;
;; (add1 n) predicate:  positive?
;; (add1 n) accessor:  sub1
```

Exercise 1: What is the template for Nat?

In the following exercises we will redefine some built-in arithmetic functions to get practice writing recursive functions over Nats, so don't simply reuse the built-in functions.

Exercise 2: Design a function `nat-even?` that returns true if the given `Nat` is even.
You may only use `sub1` (and possibly `not`). I.e., do not use `even?`, `odd?`, `modulo`, etc.

Exercise 3: Design a function `double` that doubles the given `Nat`. Again, you may only use `add1` and `sub1` (and `double` of course).

Exercise 4: Design a function `down-from` that takes a `Nat` n and returns the list of `Nat`s counting down from n. For example, `(down-from 3) = (list 3 2 1 0)`.

Exercise 5: Design a function `repeat` that takes a `Nat` n and a `String` s and returns a list that repeats s n times. For example, `(repeat "buffalo" 8) = (list "buffalo" "buffalo" "buffalo" "buffalo" "buffalo" "buffalo" "buffalo" "buffalo")`. Do not use make-list! (though it's good to know about).

Exercise 6: Design a function `nat+` that takes two `Nat`s and computes their sum. (Use recursion, not the built-in `+` function.)

Exercise 7: Design a function `nat*` that takes two `Nat`s and computes their product. (Again use recursion, not the built-in `*` function, though you may use your `nat+` now.)

Exercise 8: Design a function `sqware` that squares the given `Nat` (Note the intended name misspelling!) WITHOUT using `nat*`! Again, you may only use `add1`, `sub1`, `double`, and `nat+` (and `sqware` of course).

Part II: Concentric rings in the World

Some basic setup:

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

(define width 400)
(define height 400)
```

In this animation, a World is a collection of Rings, each of which has a size and a location.

```
; A World is a [listof Ring]

; A Ring is a (make-ring Nat Posn)
 (define-struct ring (size center))
```

Exercise 9: Design a `grow-ring` function that increases a `Ring`'s size by 1.

Exercise 10: Design a little `draw-ring` function that takes a `Nat` r as input and simply returns an image of a circle with radius r. (We'll make this more interesting later.)

Exercise 11: Design a `place-ring` function that draws a `Ring` into the given `Scene` at the `Ring`'s location. (Use `draw-ring` here so that we can modify it later to change the animation.)

Exercise 12: Design a `draw` function that renders a `World` as a `Scene` by drawing all the `Ring`s in their correct locations.

Exercise 13: Design a `mouse` function that, when the mouse is clicked, adds a 0-size `Ring` to the `World` at the location of the click.

Exercise 14: Design a `tick` function that grows all the `Ring`s in the `World` using `grow-ring`.

Put it all together and see what you get:

```
(big-bang empty
       (on-tick tick .25)
       (to-draw draw)
       (on-mouse mouse))
```

Exercise 15: Now let's redesign the `draw-ring : Nat -> Image` function. Instead of making an image of a solid circle, let's make concentric rings of many circles. We can achieve this by overlaying many circles of increasing sizes:

Natural number recursion should serve you well here...