Our zoo was so successful, let’s start an aquarium

For a fish, we only care about its weight, so for two fish:

```scheme
; An aquarium is
; (make-aq num num)
(define-struct aq (first second))
```
Aquarium Template

; An aquarium is
; (make-aq num num)

Generic template:
; func-for-aq : aquarium -> ...
; (define (func-for-aq a)
; ... (aq-first a) ... (aq-second a) ...)

; aq-weight : aquarium -> num
(define (aq-weight a)
  (+ (aq-first a) (aq-second a)))

(check-expect (aq-weight (make-aq 7 8)) 15)

And so on, for many other simple aquarium functions...
Tragedy Strikes the Aquarium

Poor blue fish... now we have only one

; An aquarium is
; (make-aq num)
(define-struct aq (first))
Aquarium Template, Revised

; An aquarium is
; (make-aq num)

; func-for-aq : aquarium -> ... 
; (define (func-for-aq a)
; ... (aq-first a) ...)

; aq-weight : aquarium -> num
(define (aq-weight a)
  (aq-first a))

(check-expect (aq-weight (make-aq 7)) 7)

And so on, for all of the aquarium functions...
The Aquarium Expands

Hooray, we have two new fish!

Unfortunately, we have to re-re-write all our functions...

; An aquarium is
; (make-aq num num num num)
(define-struct aq (first second third))
A Flexible Aquarium Representation

Our data choice isn’t working

• An aquarium isn’t just 1 fish, 2 fish, or 100 fish—it’s a collection containing an arbitrary number of fish

• No data definition with just 1, 2, or 100 numbers will work

To represent an aquarium, we need a list of numbers

We don’t need anything new in the language, just a new idea
Structs as Boxes

Pictorially,

- \textbf{define-struct} lets us define a new kind of box
- The box can have as many compartments as we want, but we have to pick how many, once and for all

\begin{verbatim}
(define-struct \textbf{snake} (name weight food))
⇒ \[
\begin{array}{c}
\hline
\text{name} \\
\hline
\text{weight} \\
\hline
\text{food} \\
\hline
\end{array}
\end{verbatim}

\begin{verbatim}
(define-struct \textbf{ant} (weight loc))
⇒ \[
\begin{array}{c}
\hline
\text{weight} \\
\hline
\text{loc} \\
\end{array}
\end{verbatim}
Boxes Stretch

The boxes stretch to fit any one thing in each slot:

'slinky 12 'rats

Even other boxes:

0.002 2 3

Still, the number of slots is fixed
Packing Boxes

Suppose that

• You have four things to pack as one
• You only have 2-slot boxes
• Every slot must contain exactly one thing

How can you create a single package?
Packing Boxes

This isn’t good enough because it’s still two boxes...

But this works!
Packing Boxes

And here’s 8 fish:

And here’s 16 fish!

But what if we just add 1 fish, instead of doubling the fish?

But what if we have 0 fish?
General Strategy for Packing Boxes

Here’s a general strategy:

• For 0 fish, use **empty**

• If you have a package and a new fish, put them together

To combine many fish, start with **empty** and add fish one at a time
General Strategy for a List of Numbers

To represent the aquarium as a list of numbers, use the same idea:

• For 0 fish, use `empty`

• If you have a list and a number, put them together with `make-bigger-list`

\[
\text{empty} \\
(\text{make-bigger-list} \ 10 \ \text{empty}) \\
(\text{make-bigger-list} \ 5 \ (\text{make-bigger-list} \ 10 \ \text{empty})) \\
(\text{make-bigger-list} \ 7 \ (\text{make-bigger-list} \ 5 \ (\text{make-bigger-list} \ 10 \ \text{empty})))
\]
List of Numbers

; A list-of-num is either
;   - empty
;   - (make-bigger-list num list-of-num)
(define-struct bigger-list (first rest))
List of Numbers

; A list-of-num is either
;   - empty
;   - (make-bigger-list num list-of-num)
(define-struct bigger-list (first rest))

Generic template:
; func-for-lon : list-of-num -> ...
(define (func-for-lon l)
  ...
)
List of Numbers

; A list-of-num is either
; - empty
; - (make-bigger-list num list-of-num)
(define-struct bigger-list (first rest))

Generic template:
; func-for-lon : list-of-num -> ...
(define (func-for-lon l)
  (cond
   [(empty? l) ...]
   [(bigger-list? l) ...])))
List of Numbers

; A list-of-num is either
;   - empty
;   - (make-bigger-list num list-of-num)
(define-struct bigger-list (first rest))

Generic template:
; func-for-lon : list-of-num -> ... 
(define (func-for-lon l)
  (cond
    [(empty? l) ...]
    [(bigger-list? l)
      ... (bigger-list-first l)
      ... (bigger-list-rest l)
      ...]))
List of Numbers

; A list-of-num is either
;   - empty
;   - (make-bigger-list num list-of-num)
(define-struct bigger-list (first rest))

Generic template:
; func-for-lon : list-of-num -> ...
(define (func-for-lon l)
  (cond
   [(empty? l) ...]
   [(bigger-list? l)
    ... (bigger-list-first l)
    ... (bigger-list-rest l)
    ...]]))
List of Numbers

; A list-of-num is either
;   - empty
;   - (make-bigger-list num list-of-num)
(define-struct bigger-list (first rest))

Generic template:
; func-for-lon : list-of-num -> ...
(define (func-for-lon l)
  (cond
   [(empty? l) ...]
   [(bigger-list? l)
     ... (bigger-list-first l)
     ... (func-for-lon (bigger-list-rest l))
     ...]])
Aquarium Weight

; aq-weight : list-of-num -> num
; Sums the fish weights in l
(define (aq-weight l)
  ...)

Aquarium Weight

; aq-weight : list-of-num -> num
; Sums the fish weights in l
(define (aq-weight l)
  ...)

(check-expect (aq-weight empty) 0)
Aquarium Weight

; aq-weight : list-of-num -> num
; Sums the fish weights in l
(define (aq-weight l)
  ...)

(check-expect (aq-weight empty) 0)
(check-expect (aq-weight (make-bigger-list 2 empty)) 2)
Aquarium Weight

; aq-weight : list-of-num -> num
; Sums the fish weights in l
(define (aq-weight l)
  ...)

(check-expect (aq-weight empty) 0)
(check-expect (aq-weight (make-bigger-list 2 empty)) 2)
(check-expect (aq-weight (make-bigger-list 5 (make-bigger-list 2 empty))) 7)
Aquarium Weight

; aq-weight : list-of-num -> num
; Sums the fish weights in l
(define (aq-weight l)
  (cond
   [(empty? l) ...]
   [(bigger-list? l)
    ... (bigger-list-first l)
    ... (aq-weight (bigger-list-rest l))
    ...]]))

(check-expect (aq-weight empty) 0)

(check-expect (aq-weight (make-bigger-list 2 empty)) 2)

(check-expect (aq-weight (make-bigger-list 5 (make-bigger-list 2 empty))) 7)
Aquarium Weight

; aq-weight : list-of-num -> num
; Sums the fish weights in l
(define (aq-weight l)
  (cond
   [(empty? l) 0]
   [(bigger-list? l)
     (+ (bigger-list-first l)
        (aq-weight (bigger-list-rest l)))])))

(check-expect (aq-weight empty) 0)

(check-expect (aq-weight (make-bigger-list 2 empty)) 2)

(check-expect (aq-weight (make-bigger-list 5 (make-bigger-list 2 empty))) 7)
Aquarium Weight

; aq-weight : list-of-num -> num
; Sums the fish weights in l
(define (aq-weight l)
  (cond
   [(empty? l) 0]
   [(bigger-list? l)
     (+ (bigger-list-first l)
        (aq-weight (bigger-list-rest l)))]))

Try examples in the stepper

(check-expect (aq-weight empty) 0)
(check-expect (aq-weight (make-bigger-list 2 empty)) 2)
(check-expect (aq-weight (make-bigger-list 5 (make-bigger-list 2 empty))) 7)
Shortcuts

The name `make-bigger-list` is awfully long

DrRacket has built-in shorter versions

```scheme
make-bigger-list    \Rightarrow\  cons
bigger-list-first  \Rightarrow\  first
bigger-list-rest   \Rightarrow\  rest
bigger-list?       \Rightarrow\  cons?
```

```scheme
(first (cons 1 empty)) \rightarrow 1
(rest (cons 1 empty))  \rightarrow  empty
(cons? empty)          \rightarrow  false
```
Lists using the Shortcuts

; A list-of-num is either
;  - empty
;  - (cons num list-of-num)

; aq-weight : list-of-num -> num
(define (aq-weight l)
  (cond
   [(empty? l) 0]
   [(cons? l) (+ (first l)
                 (aq-weight (rest l)))]))

(check-expect (aq-weight empty) 0)

(check-expect (aq-weight (cons 5 (cons 2 empty))) 7)
Design Recipe for Lists

Design recipe changes for today:

None

Granted, the self-reference was slightly novel...

; A list-of-num is either
;   - empty
;   - (cons num list-of-num)
Recursion

A self-reference in a data definition leads to a **recursive** function—one that calls itself

```
(define (aq-weight l)
  (cond
    [(empty? l) 0]
    [(cons? l) (+ (first l)
                   (aq-weight (rest l)))]))
```