Lists

29 September 2020
Beyond fixed-size data
Aquarium

Our zoo of rabbits, armadillos, and ants was so successful, we’ve decided to start an aquarium.

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

```
(define-struct aq [first second])
;; An Aquarium is
;;   (make-aq Number Number)
```
(define-struct aq [first second])
;; An Aquarium is
;;   (make-aq Number Number)
#
(define (aquarium-temp a)
  (... (aq-first a) ... (aq-second a) ...))

;; aq-weight : Aquarium -> Number
;; Compute the total weight of the fish in an aquarium

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

(define (aq-weight a)
  (+ (aq-first a) (aq-second a)))

And so on, for many other simple Aquarium functions…
Tragedy strikes the aquarium

Poor blue fish…now we have only one.
Tragedy strikes the aquarium

Poor blue fish…now we have only one.

Worse, we have to re-write all our functions…

(define-struct aq [first])
;; An Aquarium is
;; (make-aq Number)
(define-struct aq [first second])
;; An Aquarium is
;; (make-aq Number Number)
#
(define (aquarium-temp a)
  (... (aq-first a)....))

;; aq-weight : Aquarium -> Number
;; Compute the total weight of the fish in an aquarium

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

(define (aq-weight a)
  (aq-first a))

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

Hooray, we have two new fish!
The aquarium expands

Hooray, we have two new fish!

I’m different!
The aquarium expands

Hooray, we have two new fish!

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

(define-struct aq [first second third])

;; An Aquarium is

;; (make-aq Number Number Number Number Number)
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.
Structures as boxes

Pictorially,

`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.

```
(define-struct rabbit
  [name weight food])
```

```
(define-struct ant
  [weight loc])
```
Boxes stretch

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

| "Fiver"  | 1.6  | "spinach" |

Even other boxes:

| 0.002 | 2 | 3 |

But 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…
Packing boxes

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

But this works!
Packing boxes

And here’s eight fish:
Packing boxes

And here’s eight fish:

And here’s 16 fish!
Packing boxes

And here’s eight fish:

And here’s 16 fish!

But what if we just add 1 fish, instead of doubling the fish? What if we have 0 fish?
General strategy for packing boxes

Here’s a general strategy:

For 0 fish, use '()'

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

To combine many fish, start with '()' and add fish one at a time.
General strategy for packing boxes

Here’s a general strategy:

For 0 fish, use '()'

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

To combine many fish, start with '()' and add fish one at a time.

'()'
General strategy for packing boxes

Here’s a general strategy:

For 0 fish, use '()'
If you have a package and a new fish, put them together

To combine many fish, start with '()' and add fish one at a time.
General strategy for packing boxes

Here’s a general strategy:

For 0 fish, use ‘()’

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

To combine many fish, start with ‘()’ and add fish one at a time.
General strategy for packing boxes

Here’s a general strategy:

For 0 fish, use `()`

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

To combine many fish, start with `()` and add fish one at a time.
To represent the aquarium as a list of numbers, use the same idea:

For 0 fish, use `()`

If you have a bigger list and a number, put them together with `make-bigger-list`
To represent the aquarium as a list of numbers, use the same idea:

For 0 fish, use '()'

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

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

For 0 fish, use '()

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

'(())

(make-bigger-list 10 '(())
To represent the aquarium as a list of numbers, use the same idea:

For 0 fish, use '()

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

'(())

(make-bigger-list 10 '())

(make-bigger-list 5 (make-bigger-list 10 '())
To represent the aquarium as a list of numbers, use the same idea:

For 0 fish, use '()

If you have a bigger list and a number, put them together with \texttt{make-bigger-list}

\texttt{'}()

\texttt{(make-bigger-list 10 '())}

\texttt{(make-bigger-list 5 (make-bigger-list 10 '()))}

\texttt{(make-bigger-list 7 (make-bigger-list 5 (make-bigger-list 10 '())))}
Defining self-referential data
List of Numbers: Data definition

(define-struct bigger-list [first rest])

;; A ListOfNumbers is either
;; - ()
;; - (make-bigger-list Number ListOfNumbers)
List of Numbers: Template

(define-struct bigger-list [first rest])
;; A ListOfNumbers is either
;; - '()
;; - (make-bigger-list Number ListOfNumbers)

;; Template: Remind ourselves how to use the data
(define (lon-template lon)
  (... lon))
Writing a template based on a data definition

Do we see “is either” or “is one of”? 

Then we need a cond.
List of Numbers: Template

(define-struct bigger-list [first rest])
;;; A ListOfNumbers is either
;;; - '()
;;; - (make-bigger-list Number ListOfNumbers)

;;; Template: Remind ourselves how to use the data
(define (lon-temp lon)
  (cond [...]
        [...])))
Writing a template based on a data definition

Do we see “is either” or “is one of”?

Then we need a cond.

_How can we tell cases apart?

_These are the conditions in the cond._
List of Numbers: Template

(define-struct bigger-list [first rest])

;; A ListOfNumbers is either
;; - '()
;; - (make-bigger-list Number ListOfNumbers)

;; Template: Remind ourselves how to use the data
(define (lon-temp lon)
  (cond [(empty? lon) ...]
        [(bigger-list? lon) ...])))
Writing a template based on a data definition

Do we see “is either” or “is one of”?  

Then we need a **cond**.

How can we tell cases apart?  

These are the conditions in the **cond**.

*Can we deconstruct the data?*

*Use selectors.*
List of Numbers: Template

(define-struct bigger-list [first rest])
;; AListOfNumbers is either
;; - '()
;; - (make-bigger-list Number AListOfNumbers)

;; Template: Remind ourselves how to use the data
(define (lon-temp lon)
  (cond [(empty? lon) ...]
        [(bigger-list? lon)
         (... (bigger-list-first lon)
              ... (bigger-list-rest lon)
              ...)]))
Writing a template based on a data definition

Do we see “is either” or “is one of”? 

   Then we need a cond.

How can we tell cases apart?

   These are the conditions in the cond.

Can we deconstruct the data?

   Use selectors.

   Did we hit another data definition?

   Call its template.
List of Numbers: Template

(define-struct bigger-list [first rest])
;; A ListOfNumbers is either
;; - '()
;; - (make-bigger-list Number ListOfNumbers)

;; Template: Remind ourselves how to use the data
(define (lon-template lon)
  (cond
    [(empty? lon) ...]
    [(bigger-list? lon)
     (... (bigger-list-first lon)
          ... (lon-template (bigger-list-rest lon))
          ...)])
)
Functions that take self-referential data
Follow the template!
Example: Aquarium weight

;; aq-weight : ListOfNumbers -> Number
;;   Sums the fish weights in lon
Example: Aquarium weight

;; aq-weight : ListOfNumbers -> Number
;; Sums the fish weights in lon

(check-expect (aq-weight '()) 0)
(check-expect (aq-weight (make-bigger-list 2 '())) 2)
(check-expect (aq-weight (make-bigger-list 5 (make-bigger-list 2 '()))) 7)
Example: Aquarium weight

;; aq-weight : ListOfNumbers -> Number
;;   Sums the fish weights in lon

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

(define (aq-weight lon)
  (cond [(empty? lon) ...]
        [(bigger-list? lon)
         (... (bigger-list-first lon)
              ... (aq-weight (bigger-list-rest lon))
              ...)]))
Example: Aquarium weight

;;; aq-weight : ListOfNumbers -> Number
;;; Sums the fish weights in lon

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

(define (aq-weight lon)
 (cond [(empty? lon) 0]
      [(bigger-list? lon)
       [+ (bigger-list-first lon)
          (aq-weight (bigger-list-rest lon))]]))
Shortcuts

The name `make-bigger-list` is too long.

DrRacket has built-in shorter versions:

- `make-bigger-list` → `cons`
- `bigger-list-first` → `first`
- `bigger-list-rest` → `rest`
- `bigger-list?` → `cons?`
(first (cons 1 '())) → 1
(rest (cons 1 '())) → '()
(cons? '()) → #false
Lists using the shortcuts

;; A ListOfNumbers is either
;;  - '()
;;  - (cons Number ListOfNumbers)

;; aq-weight : ListOfNumbers -> Number
;; Sums the fish weights in lon

(check-expect (aq-weight '()) 0)
(check-expect (aq-weight (cons 5 (cons 2 '()))) 7)

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

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

;;; aq-weight : ListOfNumbers -> Number
;;; Sums the fish weights in lon
(define (aq-weight l)
  (cond [(empty? l) 0]
        [(cons? 1)
          (+ (first l)
             (aq-weight (rest l)))]))
Design a function `any-big?` that determines whether any fish in an aquarium weighs at least 10 lbs.
 ;; any-big? : ListOfNumbers -> Boolean
 ;; Determine whether lon has any fish >= 10 lbs

 (check-expect (any-big? '()) #false)
 (check-expect (any-big? (cons 5 '())) #false)
 (check-expect (any-big? (cons 15 '())) #true)
 (check-expect (any-big? (cons 15 (cons 7 '()))) #true)

 (define (any-big? lon)
   (cond [(empty? lon) #false]
         [(cons? lon)
          (or (>= (first lon) 10)
              (any-big? (rest lon)))]))
Acknowledgments

This lecture incorporates material from:


Marc Smith, Vassar College

Laney Strange, Northeastern University