Data Unbounded

Lecture 7
25 September 2019
Fixed-size data
Expanding the zoo

Last class we used structures to define rabbits and armadillos.

Let’s add ants.

An ant has

  a weight

  a location in the zoo
;; An ant is
;;   (make-ant Number Posn)
(define-struct ant (weight loc))
;; An ant is
;; (make-ant Number Posn)
(define-struct ant (weight loc))

(make-ant 0.001 (make-posn 4 5))
(make-ant 0.007 (make-posn 3 17))
Ants

(make-ant 0.001 (make-posn 4 5))

(make-ant 0.007 (make-posn 3 17))
Programming with ants

Define `ant-at-home?`, which takes an ant and reports whether it’s at the origin.
Signature, purpose, and header
Signature, purpose, and header

;; Ant -> Boolean
Signature, purpose, and header

;; Ant -> Boolean
;; Check whether ant a is home
Signature, purpose, and header

;; Ant -> Boolean
;; Check whether ant a is home
(define (ant-at-home? a)
  ...)

Examples

;; Ant -> Boolean
;; Check whether ant a is home
(define (ant-at-home? a)
  #true)

(check-expect (ant-at-home? (make-ant 0.001 (make-posn 0 0)))
  #true)
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 1 1)))
  #false)
Template

;; Ant -> Boolean
;; Check whether ant a is home
(define (ant-at-home? a)
  (... (ant-weight a)
       ... (ant-loc a) ...))

(check-expect (ant-at-home? (make-ant 0.001 (make-posn 0 0))) #true)
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 1 1))) #false)
Template

;; Ant → Boolean
;; Check whether ant a is home
(define (ant-at-home? a)
  (... (ant-weight a)
       ... (posn-at-home? (ant-loc a)) ...))

(check-expect (ant-at-home? (make-ant 0.001 (make-posn 0 0))) #true)
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 1 1))) #false)
Template

;; Ant -> Boolean
;; Check whether ant a is home
(define (ant-at-home? a)
  (... (ant-weight a)
       ... (posn-at-home? (ant-loc a)) ...))

(define (posn-at-home? p)
  (... (posn-x p) ... (posn-y p) ...))

(check-expect (ant-at-home? (make-ant 0.001 (make-posn 0 0))) #true)
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 1 1))) #false)
;; Ant -> Boolean
;; Check whether ant a is home
(define (ant-at-home? a)
  (posn-at-home? (ant-loc a)))

(define (posn-at-home? p)
  (and (= (posn-x p) 0)
       (= (posn-y p) 0)))

(check-expect (ant-at-home? (make-ant 0.001 (make-posn 0 0))) #true)
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 1 1))) #false)
Shapes of data and templates

The shape of the template matches the shape of the data

;; An Ant is
;; (make-ant Number Posn)

;; A Posn is
;; (make-posn Number Number)

(define (ant-at-home? a)
  (... (ant-weight a)
       ... (posn-at-home? (ant-loc a)) ...))

(define (posn-at-home? p)
  (... (posn-x p) ... (posn-y p) ...))
Programming with ants

Define `feed-ant`, which feeds ant 0.001 lbs of food.

Define `move-ant`, which takes an ant, an amount to move \( X \), and an amount to move \( Y \), and returns a moved ant.
Animals

All animals need to eat...

Define `feed-animal`, which takes an animal (rabbit, dillo, or ant) and feeds it (0.25 lbs, 2 lbs, or 0.001 lbs, respectively).
Animals

All animals need to eat…

Define feed-animal, which takes an animal (rabbit, dillo, or ant) and feeds it (0.25 lbs, 2 lbs, or 0.001 lbs, respectively).

What’s an animal?
Animal data definition

;;; An Animal is either
;;; - Rabbit
;;; - Dillo
;;; - Ant

The “either” makes this definition data with varieties.

Examples:

(make-rabbit "Hopper" 2 "bananas")
(make-dillo 2 #true)
(make-ant 0.002 (make-posn 3 4))
Feeding animals

;; Animal -> Animal
;; Feed the animal a
(define (feed-animal a)
  ...)

(check-expect
  (feed-animal (make-rabbit "Hopper" 2 "bananas"))
  (make-rabbit "Hopper" 2.25 "bananas"))

(check-expect
  (feed-animal (make-dillo 2 #true))
  (make-dillo 4 #true))

(check-expect
  (feed-animal (make-ant 0.002 (make-posn 3 4))))
  (make-ant 0.003 (make-posn 3 4)))
Template for Animals

For the template step...

(define (feed-animal a) ...)

Is a compound data?
Template for Animals

For the template step...

\[
\text{(define (feed-animal a) ...)}
\]

Is a compound data?

Technically yes, but the definition of Animal doesn’t have make-something so we don’t use the compound-data template rule.
Template for varieties

Choice in the data definition

;; An Animal is either
;; – Rabbit
;; – Dillo
;; – Ant

means a **cond** in the template:

```
(define (feed-animal a)
  (cond [(rabbit? a) ...]
        [(dillo? a) ...]
        [(ant? a) ...]))
```

Three data choices means three **cond** cases.
(define (feed-animal a)
  (cond
    [(rabbit? a) (feed-rabbit a)]
    [(dillo? a) (feed-dillo a)]
    [(ant? a) (feed-ant a)]))
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:

;;; An Aquarium is
;;;  (make-aq Number Number)
(define-struct aq [first second])
Aquarium template

;;; An Aquarium is
;;; (make-aq Number Number)

;;; Template:
;;; func-for-aq : Aquarium -> ...
;;; (define (func-for-aq a)
;;;  (... (aq-first a) ... (aq-second a) ...))

;;; aq-weight : Aquarium -> Number
(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.
Tragedy strikes the aquarium

Poor blue fish...now we have only one.

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

;; An Aquarium is
;; (make-aq Number)
(define-struct aq [first])
Aquarium template, revisited

;; An Aquarium is
;; (make-aq Number)

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

;; aq-weight : Aquarium -> Number
(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!
The aquarium expands

Hooray, we have two new fish!

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

```
;; An Aquarium is
;;   (make-aq Number Number Number)
(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.
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 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?
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.`
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.

'()'
Here’s a general strategy:

For 0 fish, use `0()

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

To combine many fish, start with `0() and add fish one at a time.
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.
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 make-bigger-list

'(())

(make-bigger-list 10 '())

(make-bigger-list 5 (make-bigger-list 10 '()))

(make-bigger-list 7 (make-bigger-list 5 (make-bigger-list 10 '())))
List of Numbers

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

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

Generic template:

;; func-for-lon : ListOfNumbers → ...
(define (func-for-lon lon)
  ...
List of Numbers

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

Generic template:

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

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

Generic template:

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

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

Generic template:

;;; func-for-lon : ListOfNumbers -> ...
(define (func-for-lon lon)
  (cond [(empty? lon) ...]
        [(bigger-list? lon)
         (... (bigger-list-first lon)
              ... (func-for-lon (bigger-list-rest lon))
              ...)]))
Aquarium weight

;; aq-weight : ListOfNumbers -> Number
;; Sums the fish weights in lon
(define (aq-weight lon)
  ...)


Aquarium weight

;; aq-weight : ListOfNumbers -> Number
;; Sums the fish weights in lon
(define (aq-weight 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)
Aquarium weight

;;; aq-weight : ListOfNumbers → Number
;;; Sums the fish weights in lon
(define (aq-weight lon)
  (cond [(empty? lon) ...]
        [(bigger-list? lon)
          (... (bigger-list-first lon)
               ... (aq-weight (bigger-list-rest 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)
Aquarium weight

;;; aq-weight : ListOfNumbers → Number
;;; Sums the fish weights in lon
(define (aq-weight lon)
  (cond [(empty? lon) 0]
        [(bigger-list? lon)
         (+ (bigger-list-first lon)
            (aq-weight (bigger-list-rest 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)
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
(define (aq-weight lon)
  (cond [(empty? lon) 0]
        [(cons? lon)
         (+ (first lon)
            (aq-weight (rest lon))))]))

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

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

\begin{verbatim}
;; 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)))]])
\end{verbatim}
Acknowledgments

This lecture incorporates material from:

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Shriram Krishnamurthi
Marc Smith