Compound Data

Lecture 6
23 September 2019
Humans make their lives easier by organizing the things that they have to deal with into various categories.

We’ve seen a variety of these categories in Racket.
Types of values in Racket

**Numbers**, e.g., 1, 3.14159

**Booleans**, e.g., #true, #false

**Strings**, e.g., "To be or not to be?"

**Procedures**, e.g., #<procedure: +>

**Images**, e.g., ☢️bbox(289,695,351,852)
There are more things in Heaven and Earth...

There are more categories than the ones we’ve seen and, importantly, those categories can have rich structure.

E.g., a personnel database might need entries for name, birth-date, income, etc.

E.g., the $x$ and $y$ positions of an object moving on the screen

These are types of compound data. Let’s start with $x$, $y$ positions.
Positions
Positions

A posn is

    (make-posn x y)

where x and y are numbers.
Positions

A posn is

(make-posn x y)

where x and y are numbers.

Examples:

(make-posn 1 2)

(make-posn 17 0)
Positions

A posn is a value, just like a number, string, or image, but it has distinct, named parts we can access.

The posn–x and posn–y operators extract numbers from a posn:

\[
\begin{align*}
\text{(posn–x (make–posn 1 2))} & \rightarrow 1 \\
\text{(posn–y (make–posn 1 2))} & \rightarrow 2
\end{align*}
\]
Positions

A **posn** is a value, just like a number, symbol, or image, but it has distinct, named parts we can access.

The **posn-x** and **posn-y** operators extract numbers from a **posn**:

\[
\begin{align*}
  (\text{posn-x } (\text{make-posn } x y)) & \rightarrow x \\
  (\text{posn-y } (\text{make-posn } x y)) & \rightarrow y
\end{align*}
\]

These are also called **selector** functions.
Positions and values

Is (make-posn 100 200) a value?
Positions and values

Is (make-posn 100 200) a value?

Yes.

A posn is

(make-posn x y)

where x and y are numbers.
Positions and values

Is `(make-posn (+ 1 2) 200)` a value?
Positions and values

Is `(make-posn (+ 1 2) 200)` a value?

No. `(+ 1 2)` is not a number yet.
Positions and values

Is `(make-posn (+ 1 2) 200)` a value?

**No.** `(+ 1 2)` is not a number yet.

Two more evaluation rules:

- \((\text{make-posn } x \ y) \rightarrow (\text{make-posn } z \ y)\)
  when \(x \rightarrow z\)

- \((\text{make-posn } x \ y) \rightarrow (\text{make-posn } x \ z)\)
  when \(y \rightarrow z\)
More examples

Try these in DrRacket’s stepper:

```
(define P1
  (make-posn (+ 1 2) (+ 3 4)))

(posn-x P1)

(posn? P1)
```
;; Posn -> NonnegativeNumber
;; Calculates the number of pixels from the upper left corner.
(define (pixels-from-corner p)
  (+ (posn-x p) (posn-y p)))

(check-expect (pixels-from-corner (make-posn 3 4)) 7)
(check-expect (pixels-from-corner (make-posn 10 6)) 16)
(check-expect (pixels-from-corner (make-posn 0 0)) 0)
;; Posn → Posn
;; Reverse the x and y parts of p
(define (flip-posn p)
  (make-posn (posn-y p) (posn-x p)))

(check-expect (flip-posn (make-posn 3 4))
  (make-posn 4 3))

(check-expect (flip-posn (make-posn 10 6))
  (make-posn 6 10))
Program design with compound data
Remember: The template is just the way of starting to write the body of the function by sketching it out.

We do that based on the input data (and only the input data).
If the input is compound data, start by selecting the parts.

;;; Posn -> Number
;;; Return the x part of p if it's bigger than the y part; otherwise, the y part.
(define (max-part p)
  (... p))

(check-expect (max-part (make-posn 10 11)) 11)
(check-expect (max-part (make-posn 7 5)) 7)
If the input is compound data, start by selecting the parts.

;; Posn -> Number
;; Return the x part of p if it's bigger than the y part; otherwise, the y part.
(define (max-part p)
  (... (posn-x p) (posn-y p)))

(check-expect (max-part (make-posn 10 11)) 11)
(check-expect (max-part (make-posn 7 5)) 7)
If the input is compound data, start by selecting the parts.

;;; Posn -> Number
;;; Return the x part of p if it's bigger than the y part; otherwise, the y part.
(define (max-part p)
  (if (> (posn-x p) (posn-y p))
      (posn-x p)
      (posn-y p))

(check-expect (max-part (make-posn 10 11)) 11)
(check-expect (max-part (make-posn 7 5)) 7)
Check that the number of parts in the template = the number of parts in the data definition named in the signature (the contract).
Other kinds of compound data
Beyond built-in data structures

Racket allows programmers to define their own structures that can represent all kinds of objects that have a fixed number of properties.
Suppose we want to represent rabbits (maybe virtual rabbits, maybe real rabbits available for adoption):
Suppose we want to represent rabbits (maybe virtual rabbits, maybe real rabbits available for adoption):

- name
- weight
- favorite food

What kind of data is appropriate for a 🐇?
Suppose we want to represent rabbits (maybe virtual rabbits, maybe real rabbits available for adoption):

- name
- weight
- favorite food

What kind of data is appropriate for a 🐰?

Not Number, Boolean, String, Image, or Posn…
Data definitions and define-struct

Here’s what we’d like:

A *Rabbit* is

(make-rabbit *String* *Number* *String*)
Data definitions and define-struct

Here’s what we’d like:

A *Rabbit* is

  (make-rabbit *String* *Number* *String*)

...but *make-rabbit* isn’t built into DrRacket.
Data definitions and **define-struct**

We can tell DrRacket about rabbits:

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

which creates the following:

- **constructor**: `make-rabbit`
- **selectors**: `rabbit-name`, `rabbit-weight`, `rabbit-food`
- **predicate**: `rabbit?`
Data definitions and `define-struct`

We can tell DrRacket about rabbits:

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

which creates the following:

```
(rabbit-name (make-rabbit x y z))  →  x
(rabbit-weight (make-rabbit x y z))  →  y
(rabbit-food (make-rabbit x y z))  →  z
(rabbit? (make-rabbit x y z))  →  #true
```
(define-struct rabbit [name weight food])

(make-rabbit "Fiver" 1.6 "spinach")

(make-rabbit "Hazel" 2.5 "carrots")

(define-struct posn [x y])

(make-posn 3 4)

(make-posn 8 -2)

BunnyCo.
Est. 1972

Posn Inc.
Est. 1865
Deciding to define *Rabbit* is the first step of the design recipe.
Deciding to define *Rabbit* is the first step of the design recipe.

*Hand-in artifact*: A comment and `define-struct`.

```scheme
(define-struct rabbit [name weight food])
```
Deciding to define **Rabbit** is the first step of the design recipe.

**Hand-in artifact**: A comment and **define-struct**.

```
(define-struct rabbit [name weight food])
```
Deciding to define **Rabbit** is the first step of the design recipe.

**Hand-in artifact**: A comment and **define-struct**.

```
;; A Rabbit is a structure:
;;  (make-rabbit String Number String)
(define-struct rabbit [name weight food])
```
Deciding to define **Rabbit** is the first step of the design recipe.

**Hand-in artifact:** A comment and `define-struct`.

```scheme
;; A Rabbit is a structure:
;;   (make-rabbit String Number String)
;; interp. (make-rabbit name weight food) is a rabbit with
;;   name is the name
;;   weight is the latest weight in lbs
;;   food is the rabbit's favorite food
(define-struct rabbit [name weight food])
```
Deciding to define *Rabbit* is the first step of the design recipe.

**Hand-in artifact:** A comment and `define-struct`.

```scheme
;;; A Rabbit is a structure:
;;;   (make-rabbit String Number String)
;;; ...
(define-struct rabbit [name weight food])
```

Now that we’ve defined Rabbit, we can use it in function signatures.
Programming with Rabbits

Implement `rabbit-skinny?`, which takes a rabbit and returns `true` if the rabbit weighs less than 2 pounds and `false` otherwise.
;; Rabbit -> Boolean
;; Determine whether r is strictly less than 2 lbs.
(define (rabbit-skinny? r) #false)

(check-expect (rabbit-skinny? (make-rabbit "Fiver" 1.6 "spinach")) #true)
(check-expect (rabbit-skinny? (make-rabbit "Hazel" 2.5 "carrots")) #false)
(check-expect (rabbit-skinny? (make-rabbit "Peter" 4 "cabbage")) #false)
;; Rabbit -> Boolean
;; Determine whether r is strictly less than 2 lbs.
(define (rabbit-skinny? r)
  (... (rabbit-name r)
       (rabbit-weight r)
       (rabbit-food r)))

(check-expect
  (rabbit-skinny? (make-rabbit "Fiver" 1.6 "spinach")))
#true
(check-expect
  (rabbit-skinny? (make-rabbit "Hazel" 2.5 "carrots")))
#false
(check-expect
  (rabbit-skinny? (make-rabbit "Peter" 4 "cabbage")))
#false)
;; Rabbit -> Boolean
;; Determine whether r is strictly less than 2 lbs.
(define (rabbit-skinny? r)
  (< (rabbit-weight r) 2))

(check-expect
  (rabbit-skinny? (make-rabbit "Fiver" 1.6 "spinach"))
  #true)
(check-expect
  (rabbit-skinny? (make-rabbit "Hazel" 2.5 "carrots"))
  #false)
(check-expect
  (rabbit-skinny? (make-rabbit "Peter" 4 "cabbage"))
  #false)
Programming with Rabbits

Implement `rabbit-skinny?`, which takes a rabbit and returns `true` if the rabbit weighs less than 2 pounds and `false` otherwise.

Implement `feed-rabbit`, which takes a rabbit and returns a rabbit with the same name and favorite food, but 0.25 pounds heavier.
;; Rabbit -> Rabbit
;; Feed r 0.25 lbs of food
(define (feed-rabbit r) r)

(check-expect
 (feed-rabbit (make-rabbit "Fiver" 1.6 "spinach"))
 (make-rabbit "Fiver" 1.85 "spinach"))

(check-expect
 (feed-rabbit (make-rabbit "Peter" 1 "cabbage"))
 (make-rabbit "Peter" 1.25 "cabbage"))
;; Rabbit -> Rabbit
;; Feed r 0.25 lbs of food
(define (feed-rabbit r)
    (... (rabbit-name r)
         (rabbit-weight r)
         (rabbit-food r)))

(check-expect
    (feed-rabbit (make-rabbit "Fiver" 2.5 "spinach"))
    (make-rabbit "Fiver" 2.75 "spinach"))

(check-expect
    (feed-rabbit (make-rabbit "Peter" 4 "cabbage"))
    (make-rabbit "Peter" 4.25 "cabbage"))
;;; Rabbit -> Rabbit
;;; Feed r 0.25 lbs of food
(define (feed-rabbit r)
  (make-rabbit (rabbit-name r)
    (+ 0.25 (rabbit-weight r))
    (rabbit-food r)))

(check-expect
 (feed-rabbit (make-rabbit "Fiver" 2.5 "spinach"))
 (make-rabbit "Fiver" 2.75 "spinach"))
(check-expect
 (feed-rabbit (make-rabbit "Peter" 4 "cabbage"))
 (make-rabbit "Peter" 4.25 "cabbage"))
Programming with Armadillos

Pick a representation for armadillos (“dillo” for short), where a dillo has a weight and may or may not be alive.
;;;; A dillo is a structure:
;;;;   (make-dillo Number Boolean)
(define-struct dillo [weight alive?] )

;;;; Template:
#;
(define (dillo-fn d)
  (... (dillo-weight d) (dillo-alive? d)))
Programming with Armadillos

Pick a representation for armadillos (“dillo” for short), where a dillo has a weight and may or may not be alive.

Implement run-over-with-car, which takes a dillo and returns a dead dillo of equal weight.
;; Dillo -> Dillo
;; To (sadly) kill d
(define (run-over-with-car d) d)

(check-expect (run-over-with-car (make-dillo 2 #true)) (make-dillo 2 #false))
(check-expect (run-over-with-car (make-dillo 8 #false)) (make-dillo 8 #false))
;; Dillo -> Dillo
;; To (sadly) kill d
(define (run-over-with-car d)
  (... (dillo-weight d) (dillo-alive? d)))

(check-expect (run-over-with-car (make-dillo 2 #true))
  (make-dillo 2 #false))
(check-expect (run-over-with-car (make-dillo 8 #false))
  (make-dillo 8 #false))
;; Dillo -> Dillo
;; To (sadly) kill d
(define (run-over-with-car d)
  (make-dillo (dillo-weight d) #false))

(check-expect (run-over-with-car (make-dillo 2 #true))
  (make-dillo 2 #false))
(check-expect (run-over-with-car (make-dillo 8 #false))
  (make-dillo 8 #false))
Programming with Armadillos

Pick a representation for armadillos ("dillo" for short), where a dillo has a weight and may or may not be alive.

Implement `run-over-with-car`, which takes a dillo and returns a dead dillo of equal weight.

Implement `feed-dillo`, where a dillo eats two pounds of food at a time.
Programming with Armadillos

Pick a representation for armadillos ("dillo" for short), where a dillo has a weight and may or may not be alive.

Implement `run-over-with-car`, which takes a dillo and returns a dead dillo of equal weight.

Implement `feed-dillo`, where a dillo eats two pounds of food at a time.

*Except*: Dead armadillos eat no food.
;; Dillo -> Dillo
;; Feed d two lbs of food, if d is alive.
(define (feed-dillo d) d)

(check-expect (feed-dillo (make-dillo 2 #true))
               (make-dillo 4 #true))
(check-expect (feed-dillo (make-dillo 2 #false))
               (make-dillo 2 #false))
;; Dillo -> Dillo
;; Feed d two lbs of food, if d is alive.
(define (feed-dillo d)
  (... (dillo-weight d) (dillo-alive? d)))

(check-expect (feed-dillo (make-dillo 2 #true))
  (make-dillo 4 #true))
(check-expect (feed-dillo (make-dillo 2 #false))
  (make-dillo 2 #false))
;; Dillo -> Dillo
;; Feed d two lbs of food, if d is alive.
(define (feed-dillo d)
  (if (dillo-alive? d)
      (make-dillo (+ 2 (dillo-weight d)) #true)
      d))

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

This lecture incorporates material from:

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Shriram Krishnamurthi
Marc Smith