Structures

22 September 2020
Data definitions and world programs
[Review Lab 3 solution]
mouseclick.rkt
Compound data
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., 🐱
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

A *Posn* is

\[(\text{make-posn } x \ y)\]

where *x* and *y* are numbers.
Positions

A \textit{Posn} is

\[(\text{make-posn } x \ y)\]

where \(x\) and \(y\) are numbers.

Examples:

\[(\text{make-posn } 1 \ 2)\]

\[(\text{make-posn } 17 \ 0)\]
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:

\[
\begin{align*}
(posn-x (make-posn 1 2)) & \rightarrow 1 \\
(posn-y (make-posn 1 2)) & \rightarrow 2
\end{align*}
\]
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:

\[
(posn-x (make-posn x y)) \rightarrow x \\
(posn-y (make-posn x y)) \rightarrow y
\]

These are also called selector functions.
Positions and values

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

Is \( \text{(make-posn 100 200)} \) a value?

Yes.

A \textit{Posn} is

\[
\text{(make-posn } x \text{ } y)\]

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 \((\text{make-posn } (+ 1 2) 200)\) a value?

\textbf{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)
```
[mouseclick-posn.rkt]
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

\[
\text{(make-rabbit } \text{String Number String})
\]

…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:

- `make-rabbit` (constructor)
- `rabbit-name`, `rabbit-weight`, `rabbit-food` (selectors)
- `rabbit?` (predicate)
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])

(bunnyco)

(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)
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`.

```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**.

```scheme
;; 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`.

```
;;; 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.
We know that when we’re writing functions that take a Rabbit as an argument we’ll need a template, so we write that as the last step in defining our data:

```
(define (rabbit-template r)
  (... (rabbit-name r)
       (rabbit-weight r)
       (rabbit-food r)))
```
Programming with Rabbits

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

This lecture incorporates material from:

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Shriram Krishnamurthi
Marc Smith