Examining Two Pieces of Data

CS 5010 Program Design Paradigms "Bootcamp"

Lesson 2.3

Don’t take apart more than one value at a time

- Almost always this will leave you with a program that is a mess.
- Better: If you need to do examine more than one value, examine one argument first, using its observer template, and pass the results on to a suitable help function or functions.

Let's continue the traffic light problem from the last lesson

```
;; A Traffic Light changes its color every 20
seconds, controlled by a
;; countdown timer.

;; A TrafficLight is represented as a struct
;;  (make-light color time-left)
;; with the fields
;; color : Color  represents the current
;;                color of the traffic light
;; time-left : TimerState   represents the
;;              current state of the timer

;; For the purposes of this example, we leave
;; Color and TimerState undefined.  For a
;; working example, we would have to define
;; these.

;; IMPLEMENTATION
(define-struct list (color time-left))

;; CONSTRUCTOR TEMPLATE
;; (make-light Color TimerState)

;; OBSERVER TEMPLATE (omitted)
```

```
;; light-after-tick :
;;    TrafficLight -> TrafficLight
;; GIVEN: the state of a traffic light
;; RETURNS: the state of a traffic
;;    light after 1 second
;; EXAMPLES: (omitted)

;; DESIGN STRATEGY: Use constructor
;;   template for TrafficLight

(define (light-after-tick l)
  (make-light
   (color-after-tick l)
   (timer-after-tick l)))
```

First, let's fill in some of the details

```
;; CONSTANTS

;; the interval between color changes
(define COLOR-CHANGE-INTERVAL 20)

;; DATA DEFINITIONS:

;; Color

;; a Color is represented by one of the strings
;; -- "red"
;; -- "yellow"
;; -- "green"
;; INTERP: self-evident
;; EXAMPLES:
(define red-color "red")
(define yellow-color "yellow")
(define green-color "green")


;; countdown timer

;; A TimerState is represented a PositiveInteger
;; WHERE: 0 < t <= COLOR-CHANGE-INTERVAL
;; INTERP: number of seconds until the next color change.
;; If t = 1, then the color should change at the next
;; second.
```

```
;; timer-at-next-second : TimerState -> TimerState
;; GIVEN: A TimerState
;; RETURNS: the TimerState at the next second
;; EXAMPLES:
;; (timer-at-next-second 17) = 16
;; (timer-at-next-second 1) = COLOR-CHANGE-INTERVAL
;; STRATEGY: if t = 1 then recycle, otherwise decrement
(define (timer-at-next-second t)
  (if (= t 1)
    COLOR-CHANGE-INTERVAL
    (- t 1)))

(check-equal? (timer-at-next-second 17) 16)
(check-equal? (timer-at-next-second 1)
              COLOR-CHANGE-INTERVAL)
```

timer-at-next-second is easy. Note how I've written a design strategy that is more informative than "cases on t". But "cases on t" would have been an acceptable thing to write for the strategy.

What about **color-at-next-second**?

```
;; color-at-next-second : TLColor TimerState -> TLColor
;; GIVEN: a TLColor c and a TimerState t
;; RETURNS: the color of the traffic light at the next
second.
;; EXAMPLES:
;; (color-at-next-second red-color 7) = red-color
;; (color-at-next-second red-color 1) = green-color
;; (color-at-next-second green-color 1) = yellow-color
;; (color-at-next-second yellow-color 1) = red-color
```

color-at-next-second needs to inspect both the current color and the current timer state!

Which shall we inspect first? Let's try each one and see how each of them works out.

Version #1: Look at the color first

```
;; STRATEGY: Cases on c : Color

(define (color-at-next-second c t)
  (cond
    [(string=? c "red")
     (if (= t 1) "green" "red")]
    [(string=? c "yellow")
     (if (= t 1) "red" "yellow")]
    [(string=? c "green")
     (if (= t 1) "yellow" "green")]))
```

That's pretty ugly! Look at all the repeated (if (= t 1) ...)'s.

It also violates "one function, one task", since it has to decide WHEN to change color AND also what color to change to.

Version #2: Look at the timer first

```
;; STRATEGY: Cases on t
(define (color-at-next-second c t)
  (if (= t 1) (next-color c) c))
```

```
;; next-color : TLColor -> TLColor
;; GIVEN: a TLColor
;; RETURNS: the TLColor that follows
the given TLColor
;; (next-color "red") = "green"
;; (next-color "yellow") = "red"
;; (next-color "green") = "yellow"
;; STRATEGY: cases on c : TLColor

(define (next-color c)
  (cond
    [(string=? c "red")
     "green"]
    [(string=? c "yellow")
     "red"]
    [(string=? c "green")
     "yellow"]))
```

Ahh!! That's much better: No repeated code. Each function has its own task: **next-color** knows about colors, and **color-at-next-second** knows about the timer.

Example #2: **ball-after-mouse**

- Let's consider **ball-after-mouse**:
- We are modelling the behavior of a ball in a simulation.
- The ball responds to mouse events. To model this response, we will clearly have to look both at the ball and the mouse event.
- Let's look at the data definition and the functions.

ball-after-mouse (2)

```
;; ball-after-mouse :
;;    Ball Integer Integer MouseEvent -> Ball
;; GIVEN: a ball, a location and a mouse event
;; RETURNS: the ball after the given mouse event at
;; the given location.
```

- Remember, when we say "a ball", we mean “the state of the ball”: this function takes a ball state and returns another ball state.
- This is sometimes called “the successor-value pattern.”

Data Definition: Ball

```
;; REPRESENTATION:
;; A Ball is represented as a struct
;;   (ball x y radius selected?)
;; with the following fields:
;; x, y : Integer        the coordinates of the center of the ball, in pixels,
;;                       relative to the origin of the scene.
;; radius : NonNegReal   the radius of the ball, in pixels
;; selected? : Boolean   true iff the ball has been selected for dragging.

;; IMPLEMENTATION:
(define-struct ball (x y radius selected?))

;; CONSTRUCTOR TEMPLATE
;; (make-ball Integer Integer NonNegReal Boolean)

;; OBSERVER TEMPLATE
;; ball-fn : Ball -> ??
(define (ball-fn b)
  (... (ball-x b)
       (ball-y b)
       (ball-radius b)
       (ball-selected? b)))
```

Mouse events

- MouseEvent is defined in the 2htdp/universe module. Every MouseEvent is represented as a string, but not every string is the representation of a mouse event.
- Two mouse events can be compared with mouse=?
- Mouse events are reported with a location, consisting of two integers representing the x and y position of the event on the canvas.

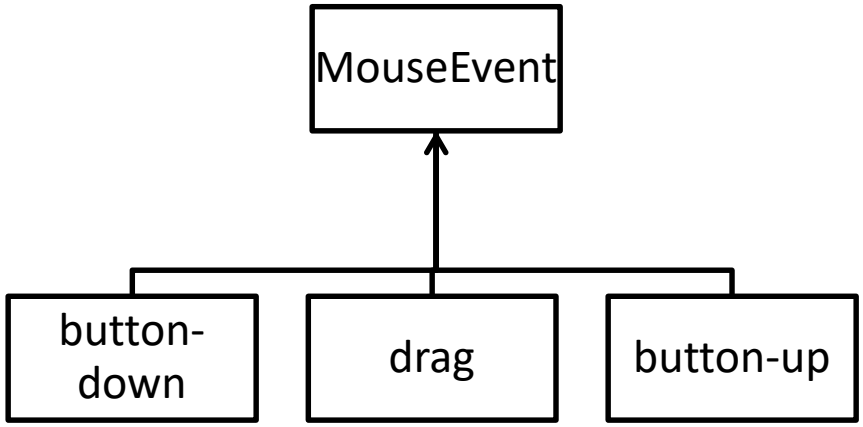
ball-after-mouse

```
;; ball-after-mouse :
;;    Ball Integer Integer MouseEvent -> Ball
;; GIVEN: a ball, a location and a mouse event
;; RETURNS: the ball after the given mouse event at
;; the given location.
;; STRATEGY: Cases on mev
(define (ball-after-mouse b mx my mev)
  (cond
    [(mouse=? mev "button-down")
     (ball-after-button-down b mx my)]
    [(mouse=? mev "drag")
     (ball-after-drag b mx my)]
    [(mouse=? mev "button-up")
     (ball-after-button-up b mx my)]
    [else b]))
```

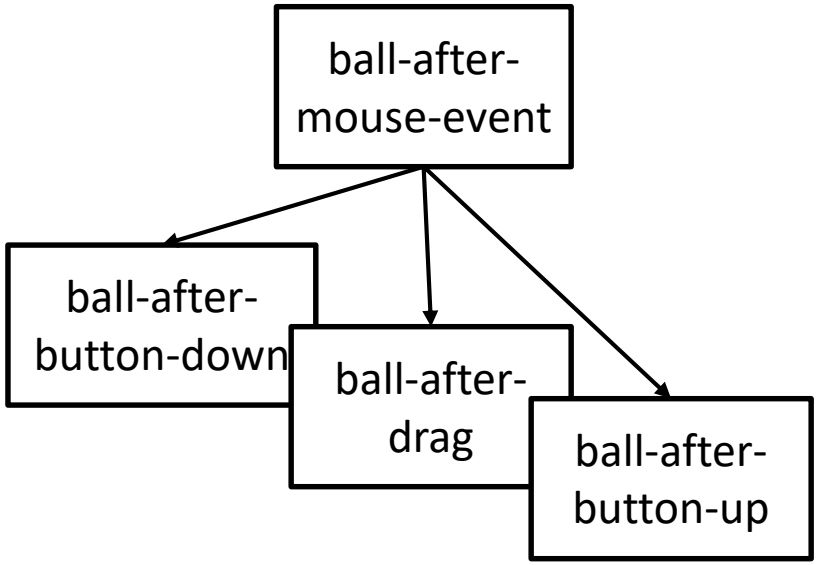
We first do cases on the mouse event. The data is handed off to one of several help functions. Each help function will decompose the compound data.

We now have a wishlist of functions to design:
ball-after-button-down
ball-after-drag
ball-after-button-up

Remember: The Shape of the Program Follows the Shape of the Data



Data Hierarchy



Call Tree (the arrow goes from caller to callee)

ball-after-drag

```
;; ball-after-drag
;;      : Ball Integer Integer -> Ball
;; GIVEN: a ball and a location
;; RETURNS: the state of the ball after a drag
;;    event at the given location.
;; STRATEGY: Use template for Ball on b.
(define (ball-after-drag b x y)
  (if (ball-selected? b)
    (ball-moved-to b x y)
    b))
```

or “cases on whether ball is selected”. Either is an OK description of the strategy.

This moves the ball so its center is at the mouse point. That’s probably not what you want in a real application. You probably want something that we call “smooth drag”, which we’ll learn about in a problem set coming up soon.

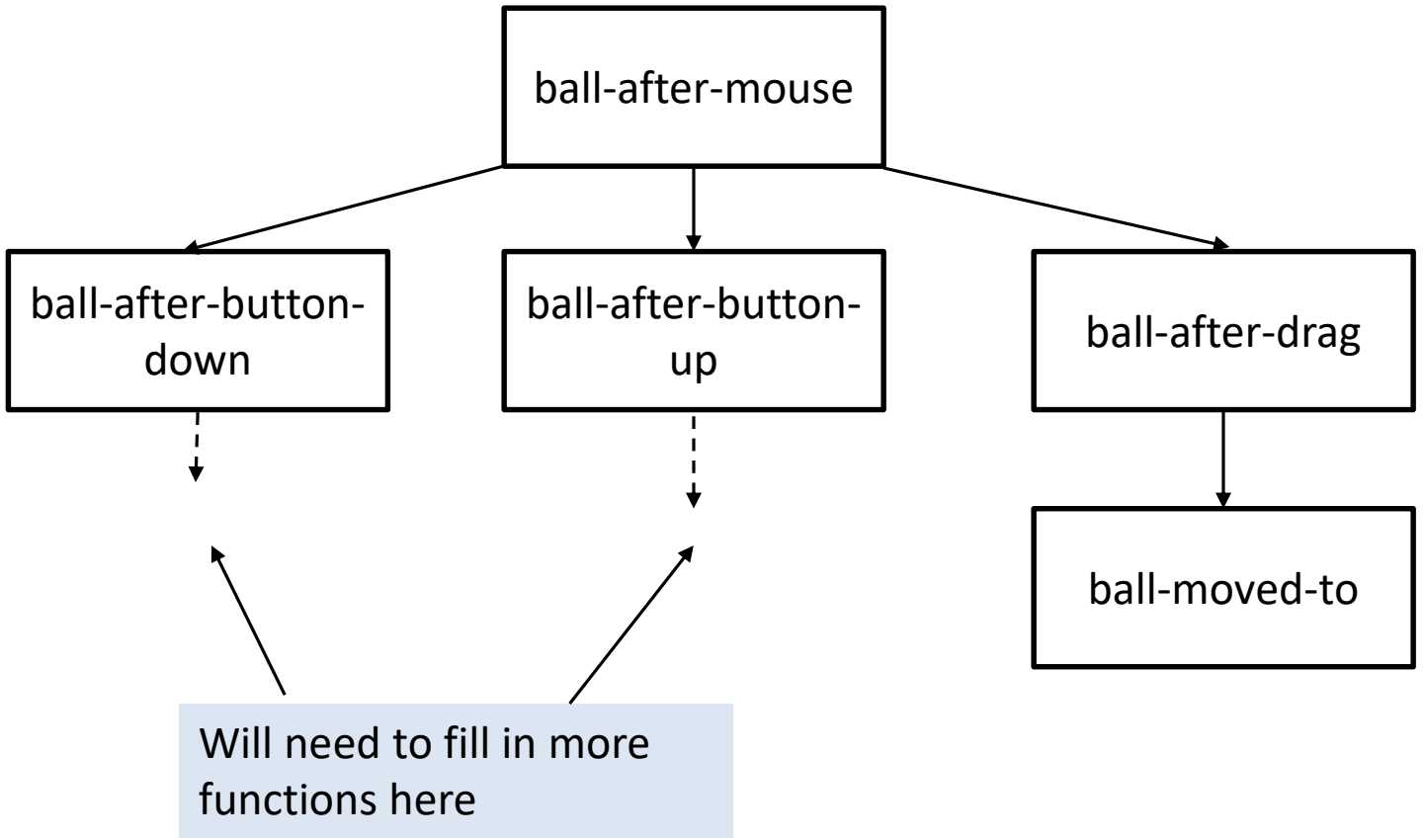
ball-moved-to

```
;; ball-moved-to : Ball Integer Integer -> Ball
;; GIVEN: a ball and a set of coordinates
;; RETURNS: a ball like the given one, except
;; that it has been moved to the given
;; coordinates.
;; STRATEGY: use template for Ball on b

(define (ball-moved-to b x y)
  (make-ball x y
    (ball-radius b)
    (ball-selected? b)))
```

You could describe the strategy here as “use the observer template,” since we are using the fields of the ball, or as “use the constructor template”, since we are using make-ball. Either would be OK, or you could simply write “use template”, as we’ve done here.

A bigger portion of the call tree



An inferior version of **ball-after-drag**

```
;; ball-after-drag
;;   : Ball Integer Integer -> Ball
;; GIVEN: a ball and a location
;; RETURNS: the ball after a drag event at the
;; given location.
;; STRATEGY: Use template for Ball on b

(define (ball-after-drag b x y)
  (if (ball-selected? b)
    (make-ball x y
      (ball-radius b)
      (ball-selected? b)))
    b))
```

This version is not as good as the preceding one, because it does two tasks: it decides WHEN to move the ball, and it also figures out HOW to move the ball.

Exception

- Sometimes it’s really clearer to take apart two things at once.
- Almost always this is because you are taking apart two compounds.

Example: balls-collide.rkt

```
;; balls-intersect? : Ball Ball -> Boolean
;; GIVEN: two balls
;; ANSWERS: do the balls intersect?
;; STRATEGY: Use template for Ball on b1 and b2.

(define (balls-intersect? b1 b2)
  (circles-intersect?
    (ball-x b1) (ball-y b1) (ball-radius b1)
    (ball-x b2) (ball-y b2) (ball-radius b2)))
```

This is OK, because trying to take the balls apart in separate functions just leads to awkward code.

circles-intersect?

```
;; circles-intersect? : Real^3 Real^3 -> Boolean
;; GIVEN: two positions and radii
;; ANSWERS: Would two circles with the given
;;  positions and radii intersect?
;; STRATEGY: Transcribe formula
(define (circles-intersect? x1 y1 r1 x2 y2 r2)
  (<=
    (+
      (sqr (- x1 x2))
      (sqr (- y1 y2)))
    (sqr (+ r1 r2))))
```

circles-intersect? knows about geometry. It doesn't know about balls: eg it doesn't know the field names of Ball or about ball-selected? .

If we changed the representation of balls, to add color, text, or to change the names of the fields, **circles-intersect?** wouldn't need to change.

If you didn't break up **balls-intersect?** with a help function like this, you would very likely be penalized for "needs help function"

Writing good definitions

- If your code is ugly, try decomposing things in the other order
- Remember: Keep it short!
  - If you have complicated junk in your function, you must have put it there for a reason. Turn it into a separate function so you can explain it and test it.
  - If your function is long and unruly, it probably means you are trying to do too much in one function. Break up your function into separate pieces and use “Combine Simpler Functions.”

Next Steps

- Study the files
  - 02-3-1-traffic-light-with-timer.rkt
  - 02-3-2-ball-after-mouse.rkt
  - 02-5-balls-collide.rkt

  in the Examples folder.
- Run them. Observe how untested code appears in orange or black.
- If you have questions or comments about this lesson, post them on the discussion board.
- Go on to the next lesson.