more Recursive Data Definitions

Wednesday, October 4, 2023
Logistics

• HW 3 out today
  • due: Sun 10/15 11:59 pm EST
  • (2 weeks)

• No class: next Monday 10/9
  • Indigenous Peoples Day
HW 1 recap (part 2)

;;; A WorldState is a Non-negative Integer
;;; Interp: Represents the y Coordinate of a
;;; ball (center) in `big-bang` animation

Your task: Modify this

(and then the code)

So ball ...
HW 1 recap (part 2)

;; A WorldState is a Non-negative Integer
;; Interp: Represents the y Coordinate of a ball (center) in `big-bang` animation
HW 1 recap (part 2)

;; A **WorldState** is a Non-negative Integer
;; Interp: Represents the y Coordinate of a
;; ball (center) in `big-bang` animation

Wraps around
Program Design Recipe (initially)

1. Data Designs
2. Function Designs
Program Design Recipe (more accurately)
Function Design Recipe

1. Name
2. Signature – types of function input(s) and output
3. Description – explain (in English prose) function behavior
4. Examples – show (using rackunit) function behavior
5. Template – sketch out function structure (using input’s Data Definition)
6. Code – implement rest of function (with arithmetic)
7. Tests – check (using rackunit) function behavior
Function Design Recipe

... is iterative!

1. **Name**
2. **Signature** – types of function input(s) and output
3. **Description** – explain (in English prose) function behavior
4. **Examples** – show (using rackunit) function behavior
5. **Template** – sketch out function structure (using input’s Data Definition)
6. **Code** – implement rest of function (with arithmetic)
7. **Tests** – check (using rackunit) function behavior
Function Design Recipe

1. **Name**
2. **Signature** – types of function input(s) and output
3. **Description** – explain (in English prose) function behavior
4. **Examples** – show (using rackunit) function behavior
5. **Template** – sketch out function structure (using input’s Data Definition)
6. **Code** – implement rest of function (with arithmetic)
7. **Tests** – check (using rackunit) function behavior

Hw said **not** to jump directly to code step!

Most students jumped directly to code step!
Function Design Recipe

1. **Name**
2. **Signature** – *types* of function input(s) and output
3. **Description** – *explain* (in English prose) function behavior
4. **Examples** – *show* (using `rackunit`) function behavior
5. **Template** – sketch out function structure (using input’s Data Definition)
6. **Code** – *implement* rest of function (with arithmetic)
7. **Tests** – *check* (using `rackunit`) function behavior

---

* DON’T ONLY FOCUS ON CODE STEP!!!
* I won’t give credit for this ...

- Interviewers
- Employers
- Other programmers
• Focusing on code is bad because ...

• ... all code is wrong!
  • i.e., has bugs

• The only hope of fixing it is ...

• ... if it is readable by others!
Some Student Examples *(sorry)*

```scheme
;; Draws a WorldState as a 2htdp/image Image
(define/contract (render-world w)
  (-> WorldState? Img?)
  (cond
    [(>= 150 w)
      (place-image
       BALL-IMG
       BALL-X w
       EMPTY-SCENE)]
    [(and (< 150 w) (>= 250 w))
      (place-image
       BALL-IMG
       BALL-X (- w 200)
       (place-image
        BALL-IMG
        BALL-X w
        EMPTY-SCENE))]))
```
Some Student Examples (sorry)

;; render-world: WorldState -> Image
;; draws a WorldState as a 2htdp/image Image
(define/contract (render-world w)
  (-> WorldState? image?)
(place-images (list BALL-IMG
               BALL-IMG
               BALL-IMG)
  (list (make-posn BALL-X w)
        (make-posn BALL-X (- w WORLD-HEIGHT))
        (make-posn BALL-X (+ w WORLD-HEIGHT)))
  EMPTY-SCENE))
Some Student Examples (sorry)

```scheme
;; render-world: WorldState -> Image
;; Draws a WorldState as a 2htdp/image Image
(define (render-world w)
  (place-image
   BALL-IMG
   BALL-X
   (+ (modulo w (+ WORLD-HEIGHT BALL-RADIUS)) BALL-RADIUS)
   (place-image
    BALL-IMG
    BALL-X
    (- 0 (- WORLD-HEIGHT (modulo w (+ WORLD-HEIGHT BALL-RADIUS))))
    EMPTY-SCENE)))
```

What does it do?
Is it “correct”? (NO)
How would you fix it?
Some Student Examples (sorry)

```scheme
;; next-worldstate : WorldState -> WorldState
;; Each tick increments y pos by 1
(define (next-worldstate w)
  (if (= w WORLD-HEIGHT)
    0
    (add1 w)))
```

Wrong style
Wrong code!
Wrong description!
Some Student Examples (sorry)

```
;; next-worldstate : WorldState -> WorldState
;; Each tick increments y pos by 1
(define (next-worldstate w)
  (if (= w WORLD-HEIGHT)
      0
      (add1 w)))
```
Some Student Examples (sorry)

```scheme
;; next-worldstate : WorldState -> WorldState
;; Each tick increments y pos by 1
(define (next-worldstate w)
  (if (= w WORLD-HEIGHT)
      0
      (add1 w)))
```

No wraparound
Some Student Examples (sorry)

```scheme
;; next-worldstate : WorldState -> WorldState
;; Each tick increments y pos by 1
(define (next-worldstate w)
  (if (= w WORLD-HEIGHT)
      0
      (add1 w)))
```

When it gets here ...
Some Student Examples (sorry)

```scheme
;; next-worldstate : WorldState -> WorldState
;; Each tick increments y pos by 1
(define (next-worldstate w)
  (if (= w WORLD-HEIGHT)
      0
      (add1 w)))
```

(Guess how many students worked through examples before jumping to code???)
Start over ... examples first

... Need “second” ball here

To get proper wraparound
Start over ... examples first

When first ball is at top ...

... second ball is off screen

Distance apart?

WORLD-HEIGHT
Examples: Corner cases

When first ball is at bottom ...

... second ball?

is about to come in
Examples: Corner cases

When first ball goes off screen ...  

... second ball is fully in
Examples: Corner cases

... So we can reset

User sees the same thing here
Examples: Corner cases

;;; A WorldState is a Non-negative Integer
;;; Interp: Represents the y Coordinate of the center
;;; of a ball in a `big-bang` animation.

So we want to “reset” when top of first ball is just off screen
A **Coordinate** is a Non-Negative Integer

**Interp:** represents x or y coordinate in a big-bang animation

where (0,0) origin is to left

A **BallTop** is a Coordinate Integer between [0 WORLD-HEIGHT]

**Interp:** Represents y coordinate of (the **top** of) a ball image.

When any part of ball "falls off" bottom, it re-appears at the top.

A **WorldState** is a BallTop

(define INITIAL-WORLD 0)
;; next-world : WorldState -> WorldState
;; the next world state = prev worldstate + 1;
(define (next-world w)
  (add1 w))

;; next-world : WorldState -> WorldState
;; the next world state = prev worldstate + 1;
;; reset to zero when the state gets to WORLD-HEIGHT
(define (next-world w)
  (modulo (add1 w) WORLD-HEIGHT))

;; function description is still wrong
original
;; render-world: WorldState -> Image
;; Draws a WorldState as a 2htdp/image Image
;; - Ball starts flush at top; wraps back to top when it “falls off” bottom.
;; - Use two ball imgs (“lead” and “follow” ball) to simulate "wraparound"
;; - when not wrapping, only "lead" ball is visible

(define (render-world w)
  (define lead-y
    (world->ballpos w)) ; "lead" ball (bottom)
(define (render-world w)  
  (define lead-y  
    (world->ballpos w)) ; "lead" ball (bottom)

;; world->pos: WorldState -> Coordinate  
;; Computes y coordinate for ball center from the current WorldState (ball top)
  (check-equal? (world->ballpos 0) BALL-RADIUS)
  (define (world->ballpos w)  
    (+ w BALL-RADIUS))
;; render-world: WorldState -> Image
;; Draws a WorldState as a 2htdp/image Image
;; - Ball starts flush at top; wraps back to top when it “falls off” bottom.
;; - Use two ball imgs (“lead” and “follow” ball) to simulate "wraparound";
;; - when not wrapping, only "lead" ball is visible

(define (render-world w)
  (define lead-y
    (world->ballpos w)) ; "lead" ball (bottom)

  (define follow-y
    (lead->follow lead-y)) ; "follow" ball
(define follow-y
  (lead->follow lead-y)) ; "follow" ball

;; lead->follow : Coord -> Coord
;; Given a y coord of “lead” ball, compute y
;; of “follow” ball, in wraparound animation
(define (lead->follow y)
  (- y WORLD-HEIGHT))
;; render-world: WorldState -> Image
;; Draws a WorldState as a 2htdp/image Image
;; - Ball starts flush at top; wraps back to top when it “falls off” bottom.
;; - Use two ball imgs (“lead” and “follow” ball) to simulate "wraparound"
;; - when not wrapping, only "lead" ball is visible

(define (render-world w)
  (define lead-y
    (world->ballpos w)) ; "lead" ball (bottom)
  (define follow-y
    (lead->follow lead-y)); "follow" ball

  (place-ball-img
   lead-y
   (place-ball-img
    follow-y
    EMPTY-SCENE)))
(define BALL-RADIUS 48)
(define BALL-X (/ WORLD-WIDTH 2)); ball only moves up/down, so x is constant
(define BALL-IMG
  (circle BALL-RADIUS "solid" "red"))

;; place-ball-img: Coordinate Image -> Image
;; Produces new img with ball img added to given Image, ;; at the specified y coord for ball center
(define (place-ball-img y img)
  (place-image BALL-IMG BALL-X y img))
Moving on ...
Predicates for Compound Data

;;; A Ball is one of:
(struct ball [x y xvel yvel] #:transparent)
;;; x : XCoord – ball center horiz coord in animation
;;; y : YCoord – ball center vert coord in animation
;;; xvel : Velocity – ball horiz pixels/tick velocity
;;; yvel : Velocity – ball vert pixels/tick velocity

(define (Ball? arg)
  (and (ball? arg)
       (XCoord? (ball-x arg))
       (YCoord? (ball-y arg))
       (Velocity? (ball-xvel arg))
       (Velocity? (ball-yvel arg)))))

(define/contract (mk-ball x y xvel yvel)
  ([x XCoord? y YCoord? xvel Velocity? yvel Velocity?]
   (ball x y xvel yvel)))

This “deep” predicate checks too much...

... because it’s the job of “coordinate” and “velocity” processing functions to check those kinds of data
;; A ListofBalls is one of
;; - empty
;; - (cons Ball ListofBalls)

;; A WorldState is a ListofBalls

(define INITIAL-WORLD
  (list (random-ball)))
List Variations – Non-empty lists

;;; A NEListofBalls (non-empty) is one of:

???

;;; A WorldState is a NEListofBalls
List Variations – Non-empty lists

;; A NEListofBalls (non-empty) is one of:
;; - (cons Ball empty)
;; - (cons Ball NEListofBalls)

(define (non-empty-list? arg)
  (and (cons? arg)

  )

(predicate?)

Just cons?! (shallow check)
Non-empty lists - template

(define (non-empty-list-fn? lst)
  (cond
   [(empty? (rest lst)) .... (first lst) ....]
   [else .... (first lst) ....
    .... (non-empty-list-fn? (rest lst)) ....]])

Don't forget to extract pieces of compound data

shape of the function
matches
shape of the data definition!

template?

need to check a little “deeper” to distinguish cases
(still a “shallow” check because not inspecting contents)

And recursive call
Preview: Famous List Functions

• Map
• Filter
• Fold (reduce)
Check-In Quiz 10/4
on gradescope

(due 1 minute before midnight)