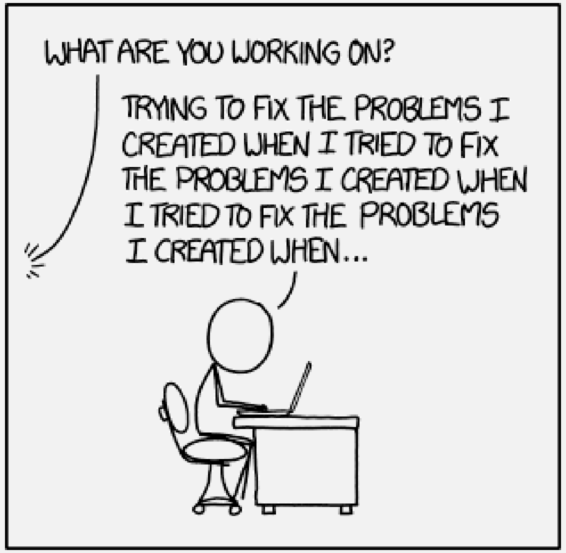
UMass Boston Computer Science

CS450 High Level Languages (section 2)

Recursive Data Definitions

Monday, October 2, 2023


WHAT ARE YOU WORKING ON?
TRYING TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN...

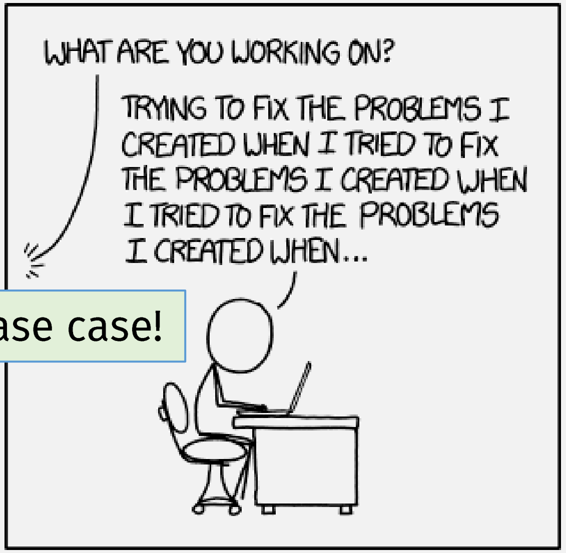
Logistics

- HW 2 in
  - ~~due: Sun 10/1 11:59 pm EST~~

- HW 3 delayed
  - out: **tomorrow**
  - due: **Sun 10/15 11:59 pm EST**
  - (2 weeks)

- No class: next Monday 10/9
  - Indigenous Peoples Day

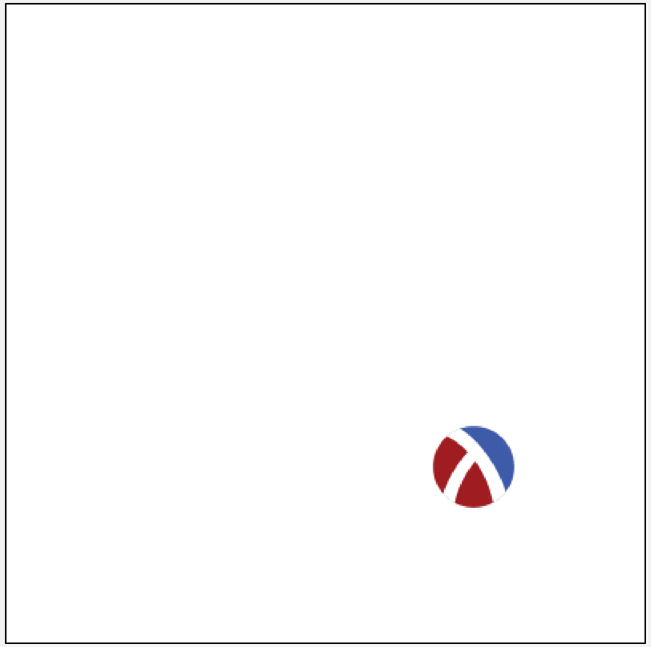
(What's wrong with this recursion?)


WHAT ARE YOU WORKING ON?
TRYING TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN...

No base case!

Last Time

Bouncing Ball



Last Time

Multi-ball Animation

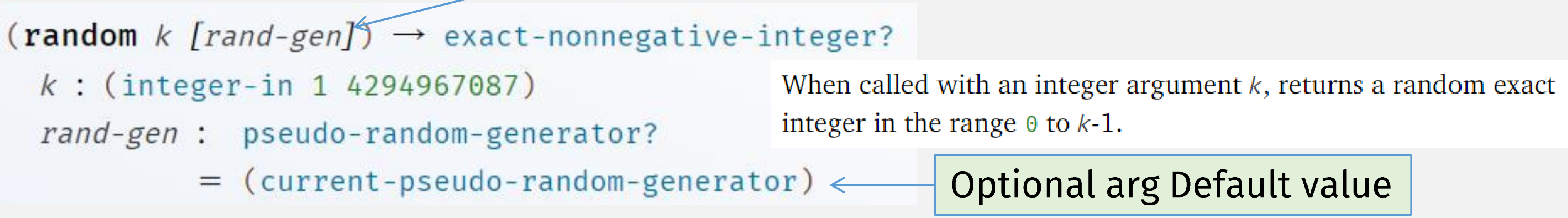
Design a **big-bang** animation that:

- Start: a single ball, moving with random *x* and *y* velocity

- If a ball "hits" an edge:
  - for **vertical** edge, **flip** *x* **velocity direction**
  - for **horizontal** edge, **flip** *y* **velocity direction**

Randomness

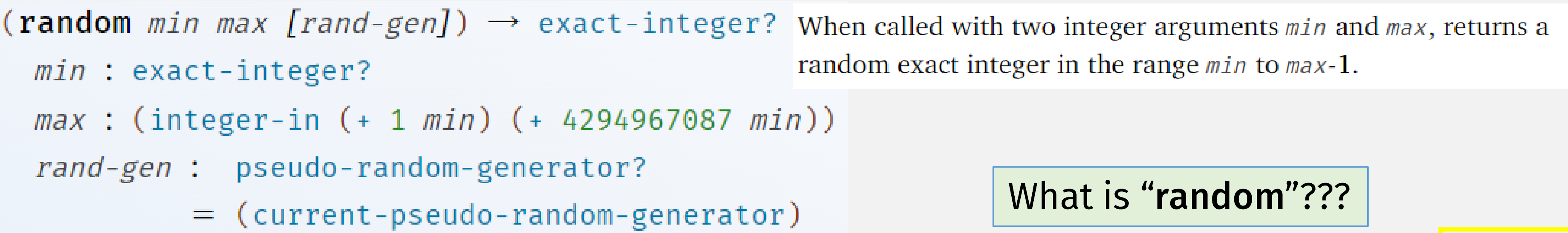
[bracketed args] = optional



```
(random k [rand-gen]) → exact-nonnegative-integer?
  k : (integer-in 1 4294967087)
  rand-gen : pseudo-random-generator?
           = (current-pseudo-random-generator)
```

When called with an integer argument *k*, returns a random exact integer in the range `0` to *k*-1.

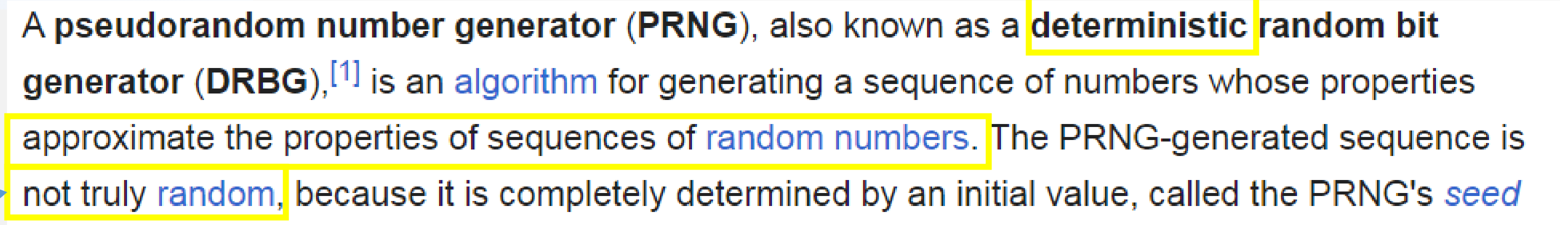
Optional arg Default value



```
(random min max [rand-gen]) → exact-integer?
  min : exact-integer?
  max : (integer-in (+ 1 min) (+ 4294967087 min))
  rand-gen : pseudo-random-generator?
           = (current-pseudo-random-generator)
```

When called with two integer arguments *min* and *max*, returns a random exact integer in the range *min* to *max*-1.

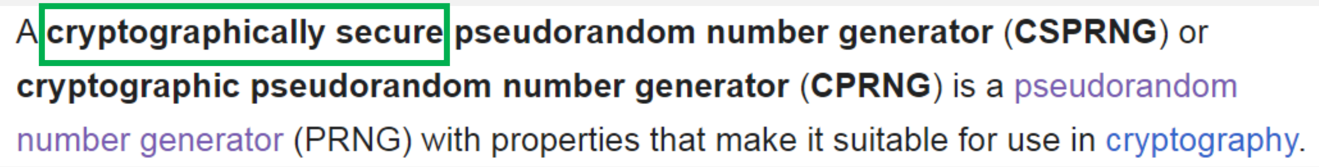
What is “**random**”???



A **pseudorandom number generator** (**PRNG**), also known as a **deterministic random bit generator** (**DRBG**),[1] is an algorithm for generating a sequence of numbers whose properties approximate the properties of sequences of random numbers. The PRNG-generated sequence is not truly random, because it is completely determined by an initial value, called the PRNG's *seed*

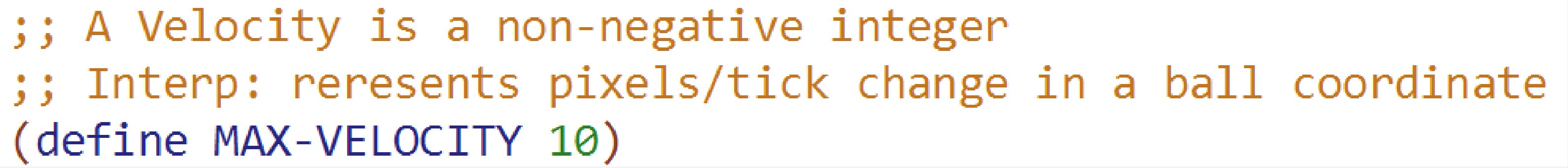
Not secure!
e.g., for generating passwords

vs

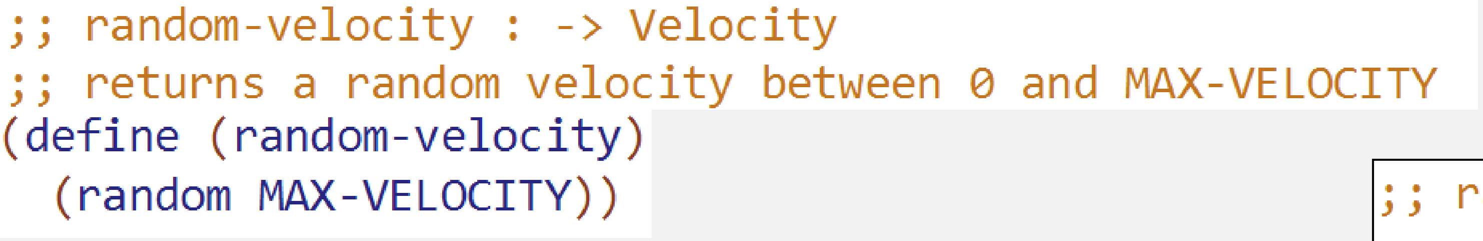


A **cryptographically secure pseudorandom number generator** (**CSPRNG**) or **cryptographic pseudorandom number generator** (**CPRNG**) is a pseudorandom number generator (PRNG) with properties that make it suitable for use in cryptography.

Designing Random Functions: Same Recipe!



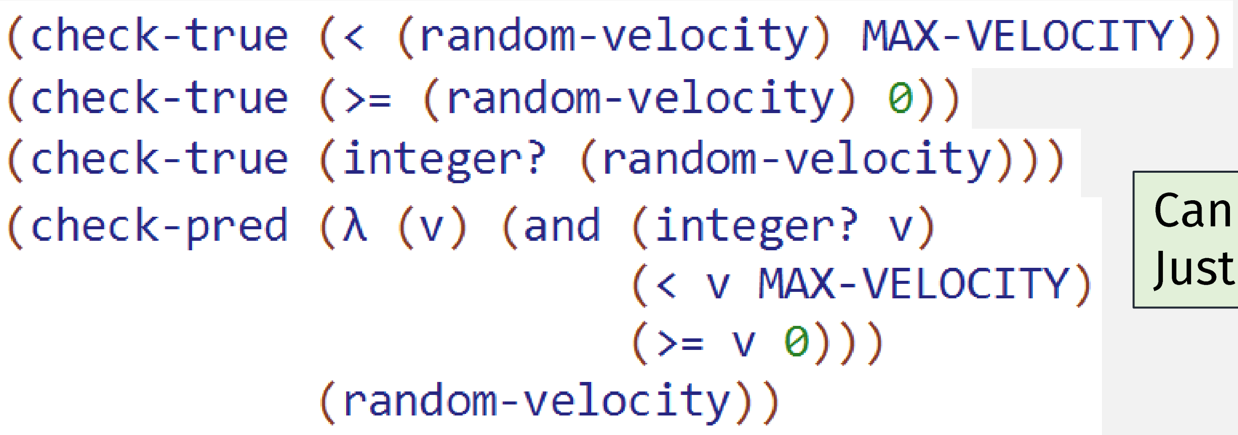
```
;; A Velocity is a non-negative integer
;; Interp: reresents pixels/tick change in a ball coordinate
(define MAX-VELOCITY 10)
```



```
;; random-velocity : -> Velocity
;; returns a random velocity between 0 and MAX-VELOCITY
(define (random-velocity)
  (random MAX-VELOCITY))
```

Functions can have zero args

```
;; random-x    : -> ???
;; random-y    : -> ???
;; random-ball : -> ???
```



```
(check-true (< (random-velocity) MAX-VELOCITY))
(check-true (>= (random-velocity) 0))
(check-true (integer? (random-velocity)))
(check-pred (λ (v) (and (integer? v)
                        (< v MAX-VELOCITY)
                        (>= v 0)))
            (random-velocity))
```

Can still **test!** Just less precise

Last Time

Multi-ball Animation

Design a **big-bang** animation that:

- Start: a single ball, moving with random *x* and *y* velocity
- On a click: add a ball at random location with random velocity
- If a ball "hits" an edge:
  - for **vertical** edge, **flip** *x* **velocity direction**
  - for **horizontal** edge, **flip** *y* **velocity direction**

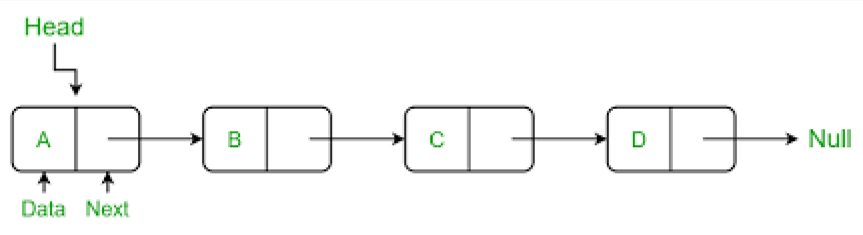
```
;; A WorldState is … an unknown number of balls!
```

Last Time

Arbitrary Size Data - Lists

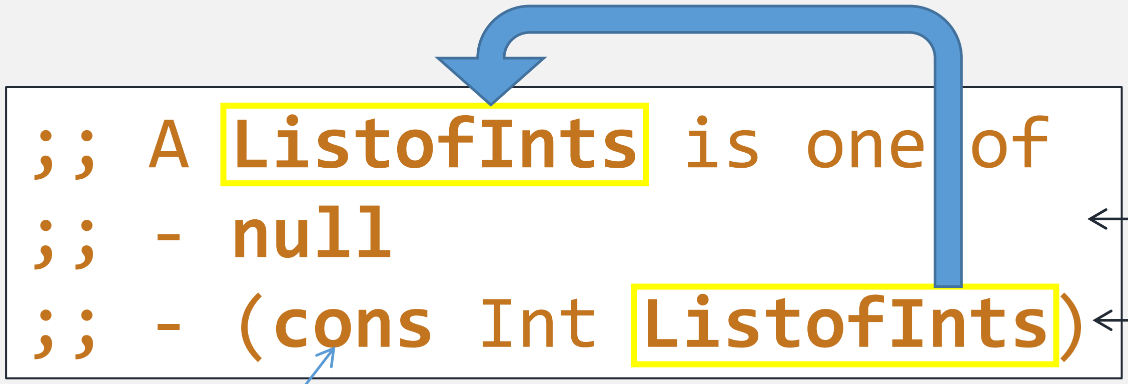
In C

```
struct node
{ int data;
  struct node *next; } *head;
```



This is a **self-referential** (i.e., **recursive**!) definition!

Racket List Data Definition Example



```
;; A ListofInts is one of
;; - null
;; - (cons Int ListofInts)
```

Empty (**base**) case

Non-empty (**recursive**) case

cons = “node”

Recursive!
(using a definition to define itself)

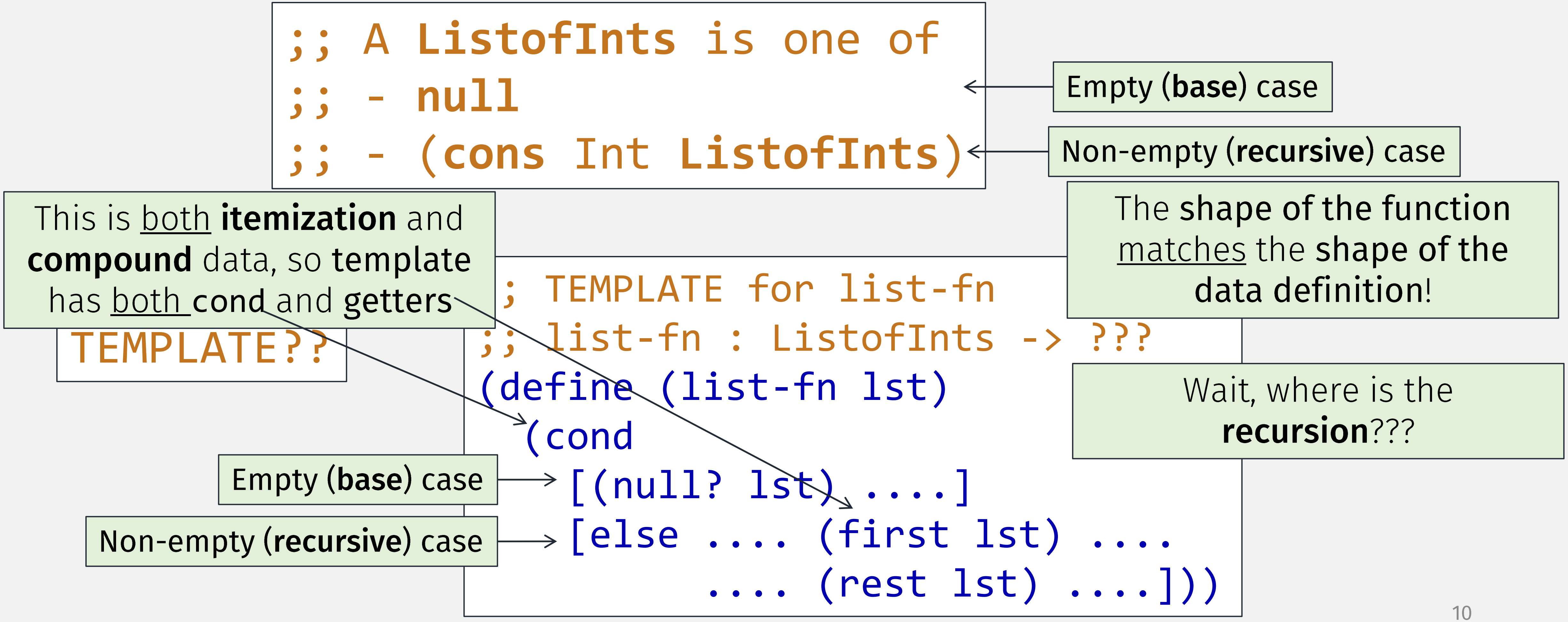
TEMPLATE??

(how can we **use a list of ints to define a list of ints**?!?)

Recursion is only **valid if** there is both

- A **base** case
- A **recursive** case

Racket List Data Definition Example



```
;; A ListofInts is one of
;; - null
;; - (cons Int ListofInts)
```

Empty (**base**) case

Non-empty (**recursive**) case

This is both **itemization** and **compound** data, so **template** has both **cond** and **getters**

TEMPLATE??

```
; TEMPLATE for list-fn
;; list-fn : ListofInts -> ???
(define (list-fn lst)
  (cond
    [(null? lst) ....]
    [else .... (first lst) ....
          .... (rest lst) ....]))
```

The **shape of the function** matches the **shape of the data definition!**

Wait, where is the **recursion**???

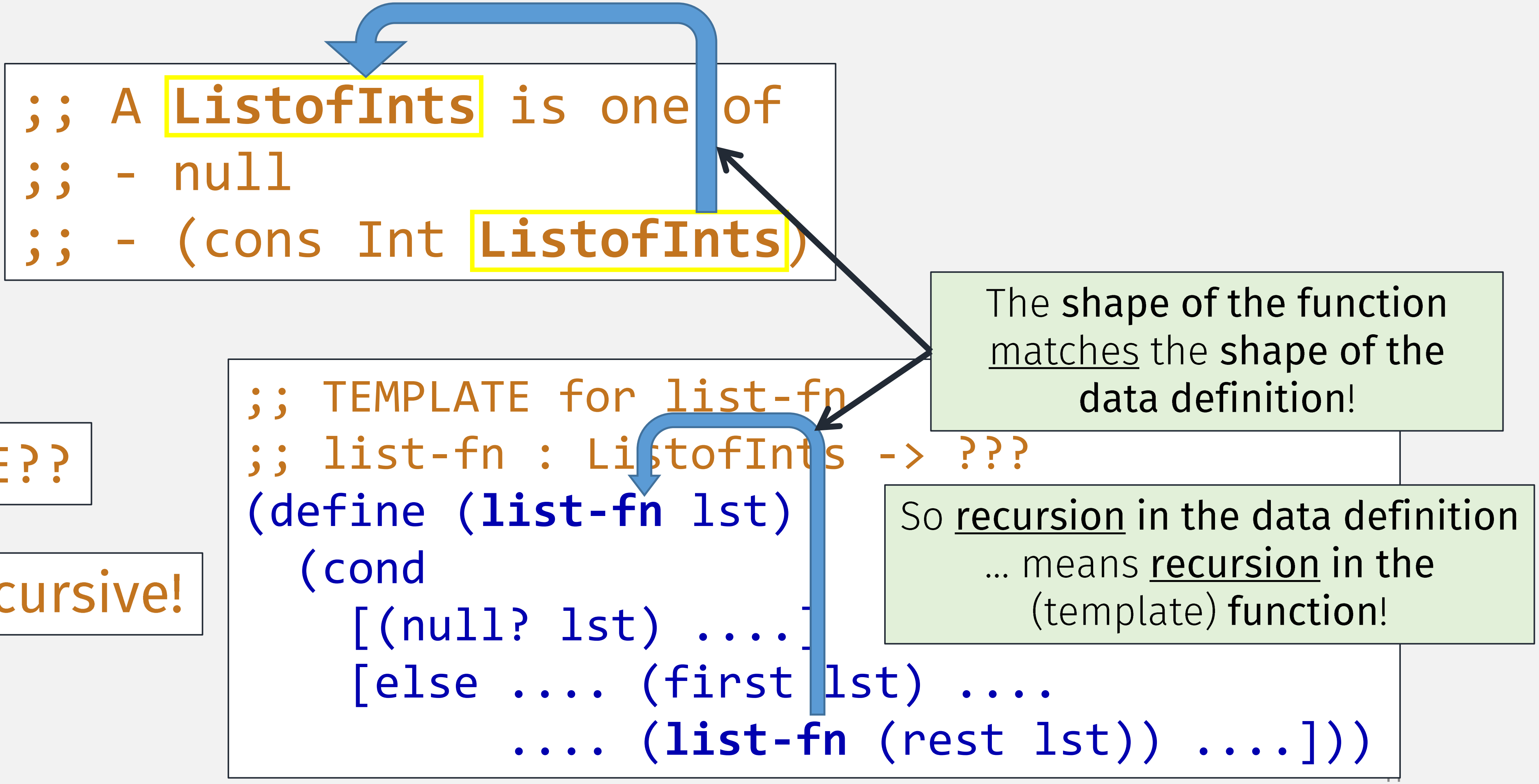
Empty (**base**) case

Non-empty (**recursive**) case

Racket List Data Definition Example

TEMPLATE??

... is also recursive!



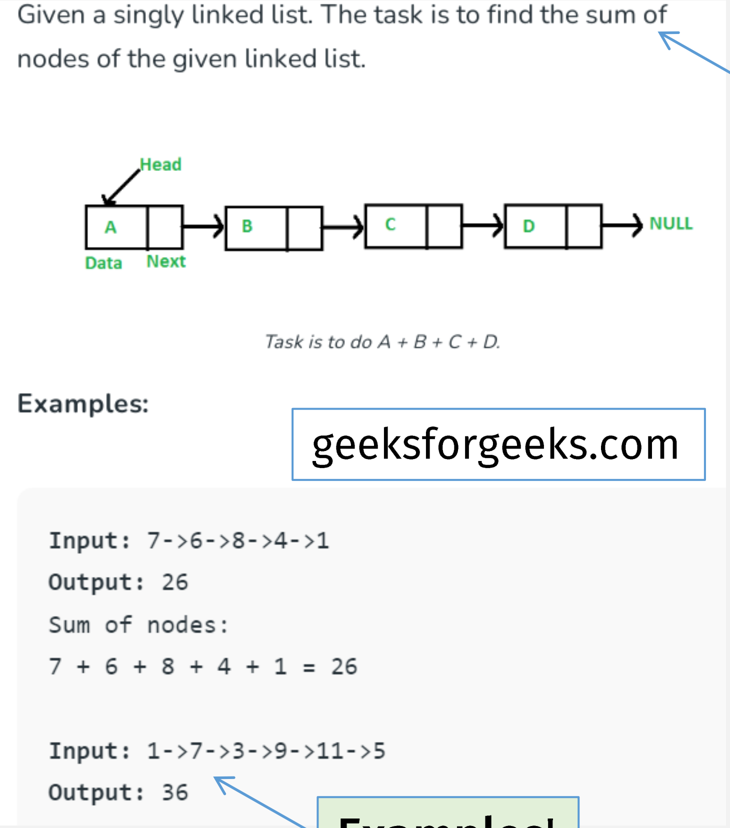
```
;; A ListofInts is one of
;; - null
;; - (cons Int ListofInts)
```

```
;; TEMPLATE for list-fn
;; list-fn : ListofInts -> ???
(define (list-fn lst)
  (cond
    [(null? lst) ....]
    [else .... (first lst) ....
          .... (list-fn (rest lst)) ....]))
```

The **shape of the function** matches the **shape of the data definition**!

So **recursion** in the data definition ... means **recursion** in the (template) **function**!

Racket Recursive List Fn Example: sum



Given a singly linked list. The task is to find the sum of nodes of the given linked list.

Description!

Task is to do A + B + C + D.

Examples:

geeksforgeeks.com

```
Input: 7->6->8->4->1
Output: 26
Sum of nodes:
7 + 6 + 8 + 4 + 1 = 26

Input: 1->7->3->9->11->5
Output: 36
```

Examples!

```
;; TEMPLATE for list-fn
;; list-fn : ListofInts -> ???
(define (list-fn lst)
  (cond
    [(null? lst) ....]
    [else .... (first lst) ....
          .... (list-fn (rest lst)) ....]))
```

Racket Recursive List Fn Example: sum

Design Recipe:
Now fill in template!
(with arithmetic)

```
;; Returns sum of list of ints
;; sum-lst: ListofInts -> Int
(define (sum-lst lst)
  (cond
    [(null? lst) ....]
    [else .... (first lst) ....
          .... (sum-lst (rest lst)) ....]))
```

Racket Recursive List Fn Example: sum

```
;; Returns sum of list of ints
;; sum-lst: ListofInts -> Int
(define (sum-lst lst)
  (cond
    [(null? lst) 0]
    [else .... (first lst) ....
          .... (sum-lst (rest lst)) ....]))
```

Racket Recursive List Fn Example: sum

```
;; Returns sum of list of ints
;; sum-lst: ListofInts -> Int
(define (sum-lst lst)
  (cond
    [(null? lst) 0]
    [else (+ (first lst)
             (sum-lst (rest lst)))]))
```

Racket Recursive List Fn Example: rev

```
;; TEMPLATE for list-fn
;; list-fn : ListofInts -> ???
(define (list-fn lst)
  (cond
    [(null? lst) ....]
    [else .... (first lst) ....
          .... (list-fn (rest lst)) ....]))
```

Racket Recursive List Fn Example: rev

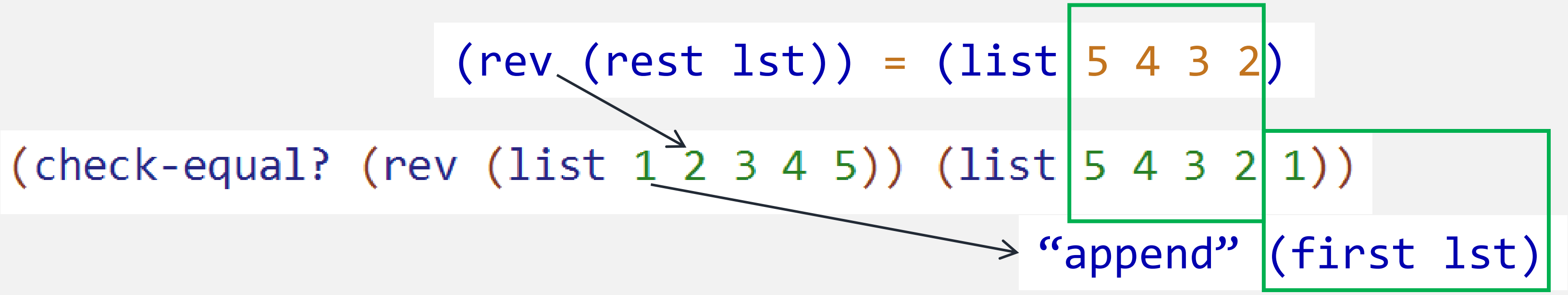
Design Recipe:
Now fill in
template!
(with arithmetic)

```
;; reverses a list of ints
;; rev: ListofInts -> ListofInts
(define (rev lst)
  (cond
    [(null? lst) ....]
    [else .... (first lst) ....
          .... (rev (rest lst)) ....]))
```

Racket Recursive List Fn Example: rev

```
;; reverses a list of ints
;; rev: ListofInts -> ListofInts
(define (rev lst)
  (cond
    [(null? lst) null]
    [else .... (first lst) ....
          .... (rev (rest lst)) ....]))
```

Racket Recursive List Fn Example: rev



```
(rev (rest lst)) = (list 5 4 3 2)
(check-equal? (rev (list 1 2 3 4 5)) (list 5 4 3 2 1))
"append" (first lst)
```

```
;; reverses a list of ints
;; rev: ListofInts -> ListofInts
(define (rev lst)
  (cond
    [(null? lst) null]
    [else (append (rev (rest lst))
                  (list (first lst)))]))
```

Recursive rev fn, with “temp” vars (preview)

```
;; TEMPLATE for list-fn
;; list-fn : ListofInts -> ???
(define (list-fn lst)
  (cond
    [(null? lst) ....]
    [else .... (first lst) ....
          .... (list-fn (rest lst)) ....]))
```

Recursive rev fn, with “temp” vars (later)

```
;; reverses a list of ints
;; rev : ListofInts -> ListofInts
(define (rev lst)
  (cond
    [(null? lst) ....]
    [else .... (first lst) ....
          .... (rev (rest lst)) ....]))
```

Recursive rev fn, with “temp” vars (later)

An internal “**helper**” function adds a “temp” variable
(main fn calls helper fn)

```
;; reverses a list of ints
;; rev : ListofInts -> ListofInts
(define (rev lst rev-lst-so-far)
  (define (rev/tmp lst rev-lst-so-far)
    (cond
      [(null? lst) ....]
      [else .... (first lst) ....
            .... (rev/tmp (rest lst) ....)
            .... rev-lst-so-far ....]))
  (rev/tmp lst null))
```

Still follows design recipe!

Tmp var = reversed list “so far” (initially `null`)

Recursive rev fn, with “temp” vars (later)

```
;; reverses a list of ints
;; rev : ListofInts -> ListofInts
(define (rev lst rev-lst-so-far)
  (define (rev/tmp lst rev-lst-so-far)
    (cond
      [(null? lst) rev-lst-so-far]
      [else .... (first lst) ....
            .... (rev/tmp (rest lst))
            .... rev-lst-so-far ....
  (rev/tmp lst null))
```

Now figure out how to “combine” these pieces (with “arithmetic”)

Recursive rev fn, with "temp" vars (later)

```
;; reverses a list of ints
;; rev : ListofInts -> ListofInts
(define (rev lst rev-lst-so-far)
  (define (rev/tmp lst rev-lst-so-far)
    (cond
      [(null? lst) rev-lst-so-far]
      [else (rev/tmp
              (rest lst)
              (cons (first lst) rev-lst-so-far))]))
  (rev/tmp lst null))
```

Add next list item to reversed list "so far"

Multi-ball Animation

Design a **big-bang** animation that:

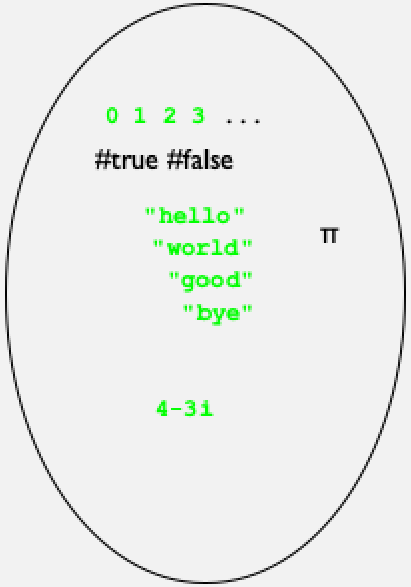
- Start: a single ball, moving with random x and y velocity
- On a click: add a ball at random location, with random velocity
- If any ball "hits" an edge:
  - if it's a **vertical** edge, **the x velocity should flip direction**
  - If it's a **horizontal** edge, **the y velocity should flip direction**

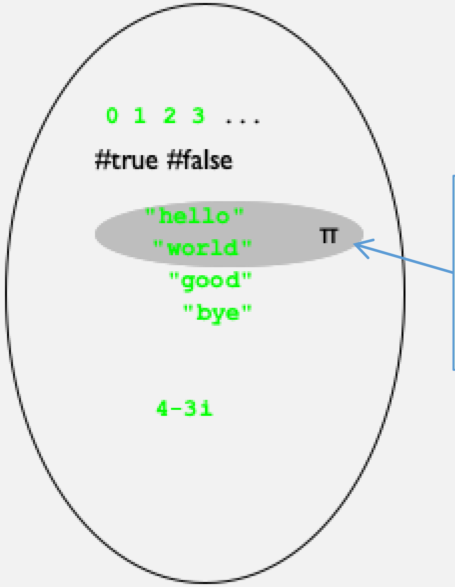
```
;; A WorldState is … an unknown number of balls!
```

```
;; A WorldState is … a list of balls!
```

Interlude: Data Definitions (ch 5.7)

All possible data values





A **data definition**
= (a named) subset of all possible values

We are defining which data values are valid for our program!

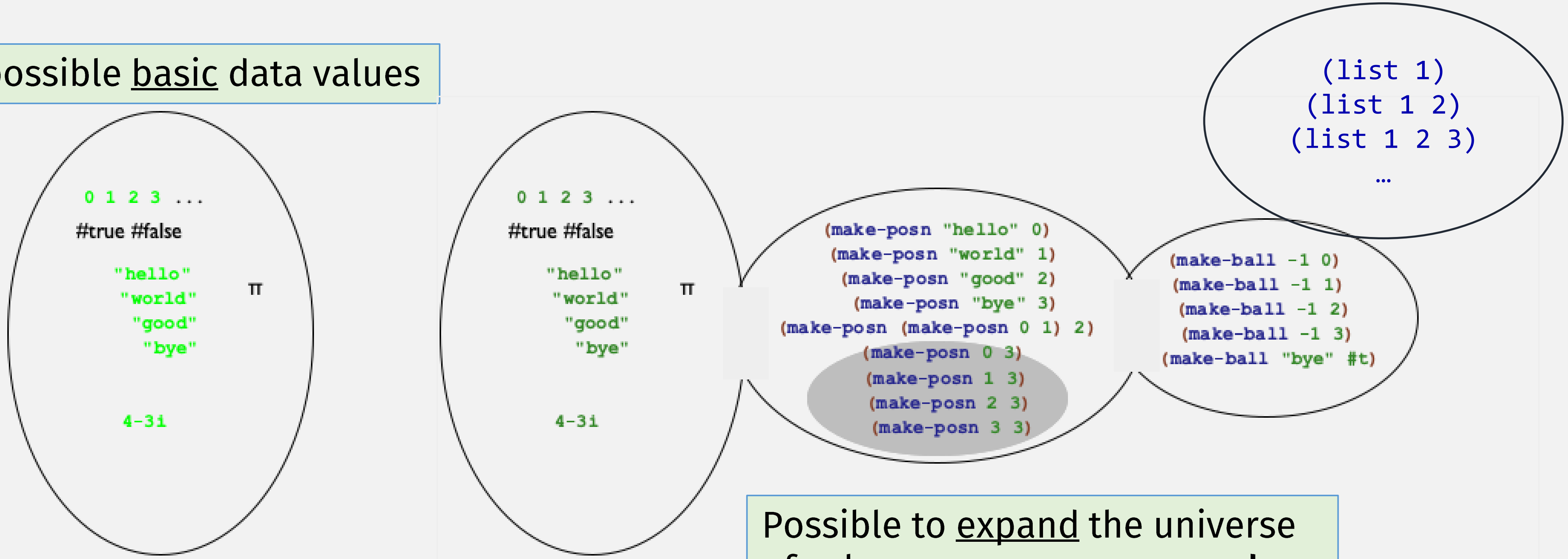
All programs are data manipulators ...

So this must be the first step of programming!

Also makes “error handling” easy

Interlude: Data Definitions (ch 5.7)

All possible basic data values



Possible to expand the universe of values, e.g., new **compound data definitions** (`struct`, or other data structure)

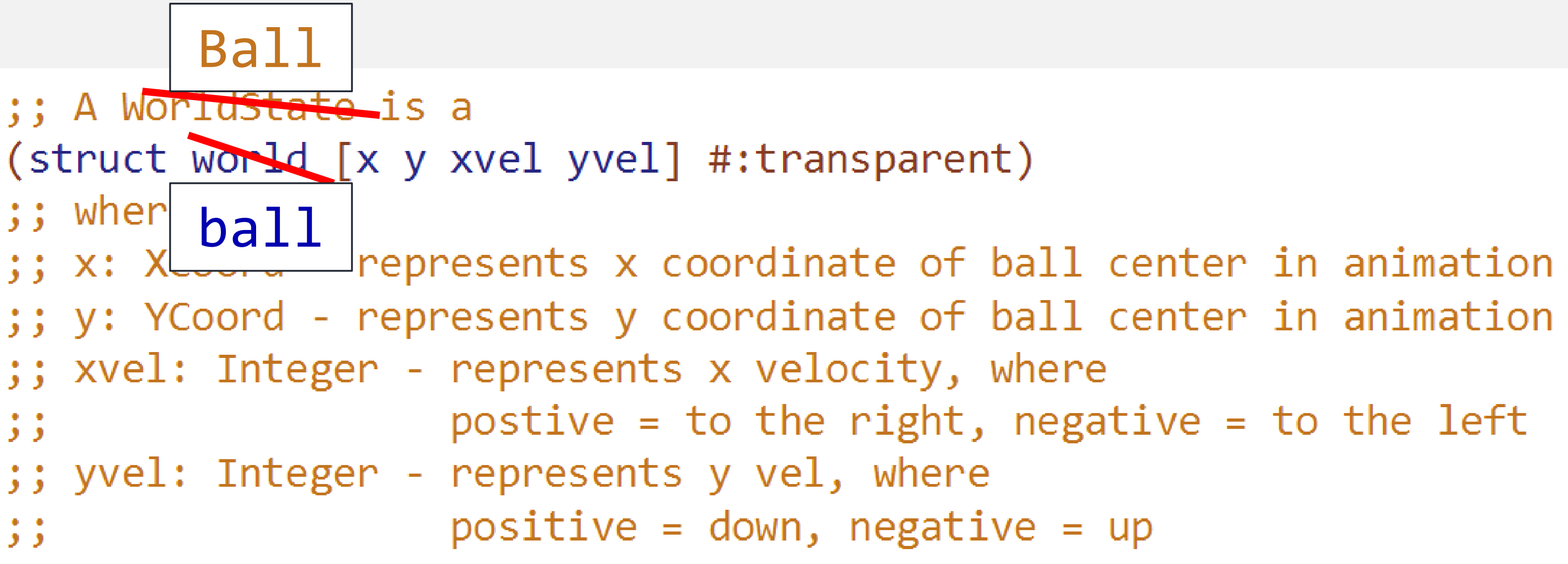
Multi-ball Animation

Design a **`big-bang`** animation that:

- Start: a single ball, moving with random x and y velocity
- On a click: add a ball at random location, with random velocity
- If any ball "hits" an edge:
  - if it's a **vertical** edge, **the x velocity should flip direction**
  - If it's a **horizontal** edge, **the y velocity should flip direction**

```
;; A WorldState is … an unknown number of balls!
```

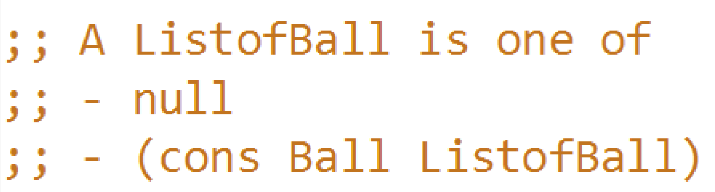
```
;; A WorldState is … a list of balls!
```



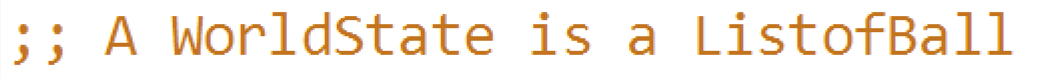
Ball

```
;; A ~~WorldState~~ is a
(struct ~~world~~ [x y xvel yvel] #:transparent)
;; where
;; x: XCoord - represents x coordinate of ball center in animation
;; y: YCoord - represents y coordinate of ball center in animation
;; xvel: Integer - represents x velocity, where
;;                 postive = to the right, negative = to the left
;; yvel: Integer - represents y vel, where
;;                 positive = down, negative = up
```

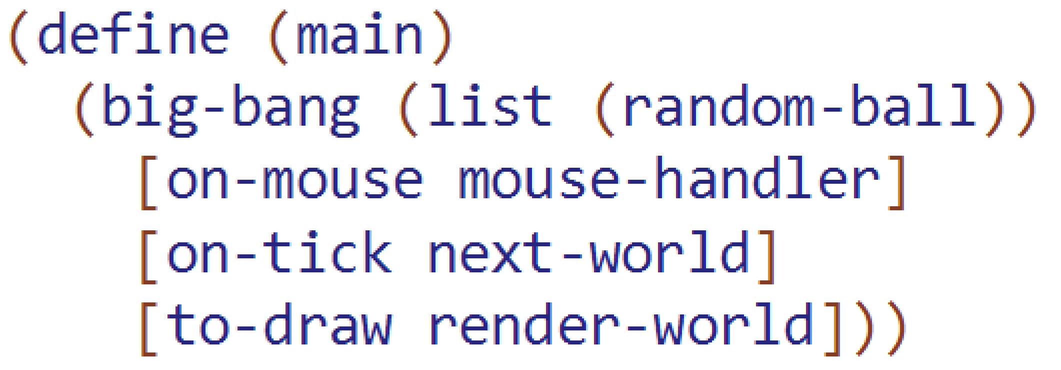
ball



```
;; A ListofBall is one of
;; - null
;; - (cons Ball ListofBall)
```

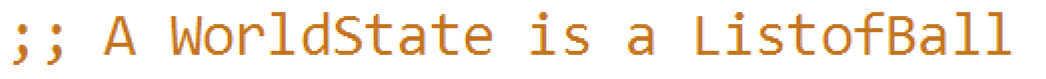


```
;; A WorldState is a ListofBall
```



```
(define (main)
  (big-bang (list (random-ball))
    [on-mouse mouse-handler]
    [on-tick next-world]
    [to-draw render-world]))
```

These need to be updated to handle new `WorldState` data def



```
;; A WorldState is a ListofBall
```

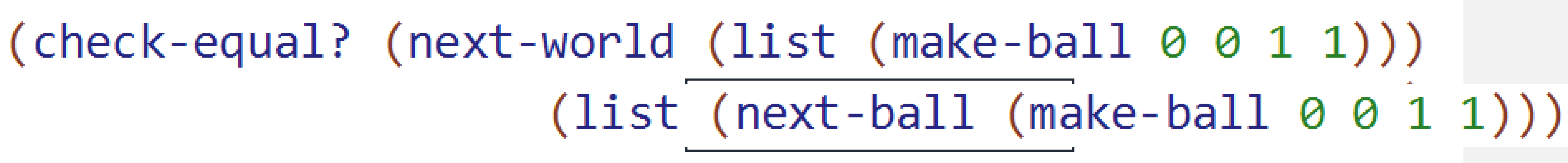
next-world

List **template!**

```
;; next-world : WorldState -> WorldState
;; Computes the next world state on a tick
(define (next-world w)
  (cond
    [(null? w) ....]
    [else .... (first w) ....
          .... (next-world (rest w)) ....]))
```

Ball

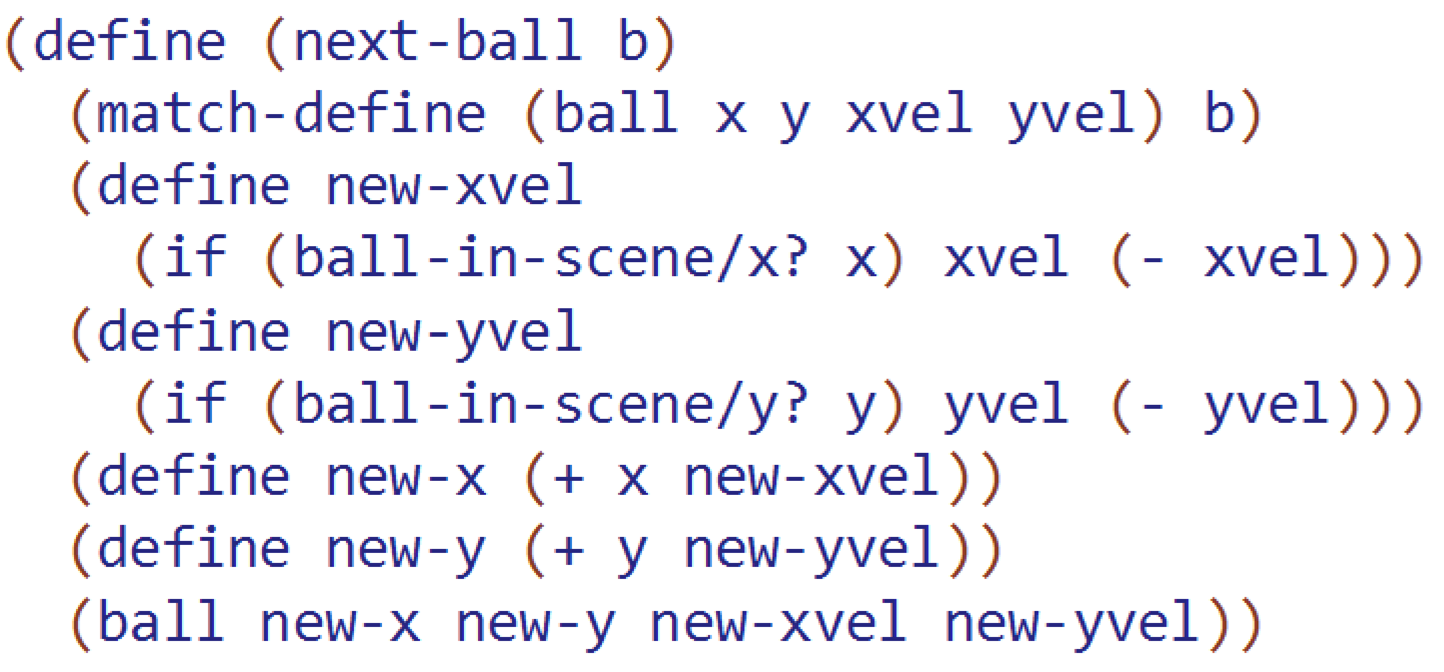
Create one function per “task”



```
(check-equal? (next-world (list (make-ball 0 0 1 1)))
                         (list (next-ball (make-ball 0 0 1 1))))
```

next-ball

This was the previous “next-world” function!



```
(define (next-ball b)
  (match-define (ball x y xvel yvel) b)
  (define new-xvel
    (if (ball-in-scene/x? x) xvel (- xvel)))
  (define new-yvel
    (if (ball-in-scene/y? y) yvel (- yvel)))
  (define new-x (+ x new-xvel))
  (define new-y (+ y new-yvel))
  (ball new-x new-y new-xvel new-yvel))
```

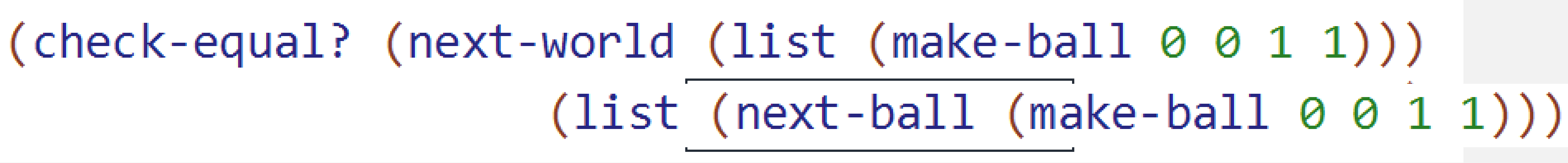
next-world

List **template!**

```
;; next-world : WorldState -> WorldState
;; Computes the next world state on a tick
(define (next-world w)
  (cond
    [(null? w) ....]
    [else .... (first w) ....
          .... (next-world (rest w)) ....]))
```

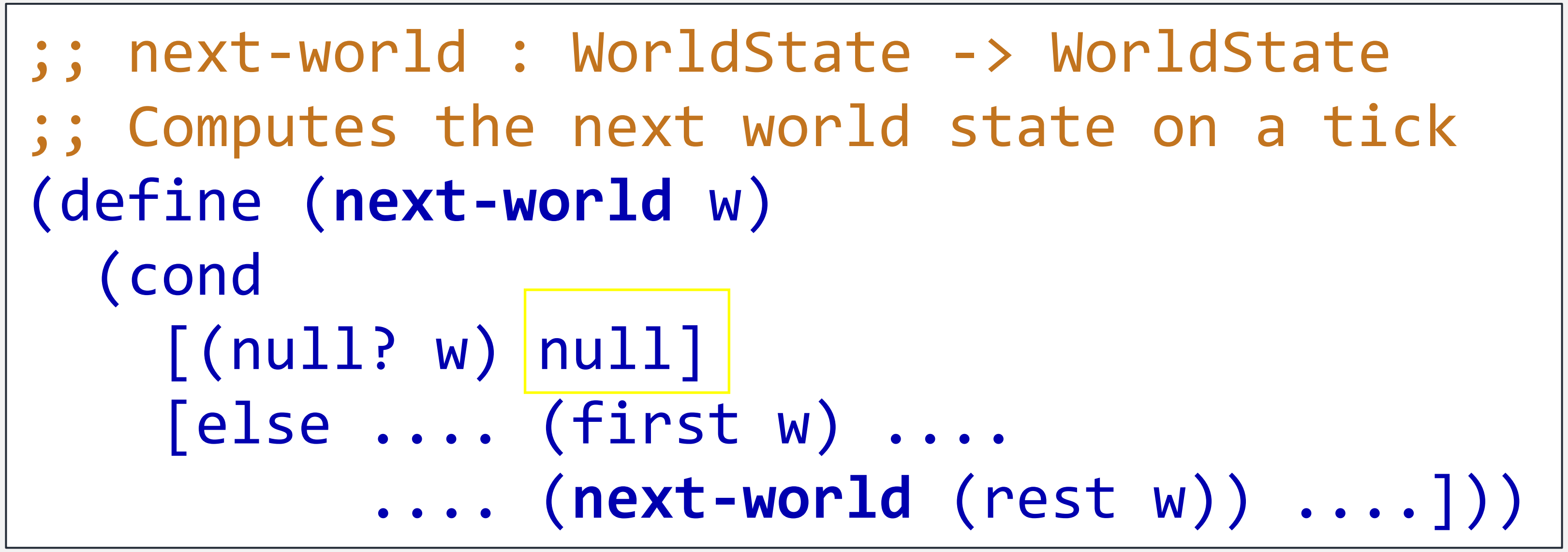
Ball

Create one function per “task”



```
(check-equal? (next-world (list (make-ball 0 0 1 1)))
                          (list (next-ball (make-ball 0 0 1 1))))
```

next-world



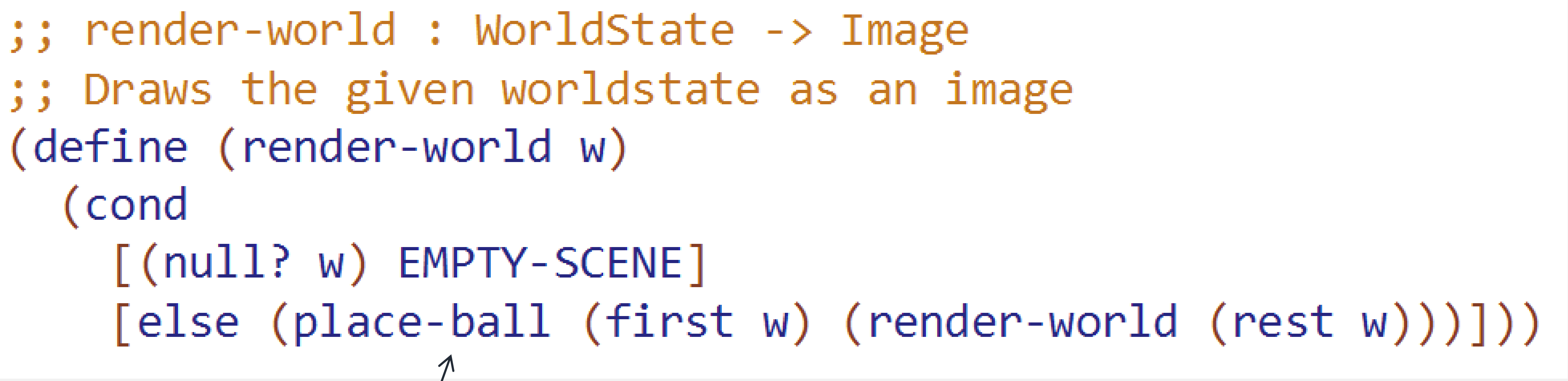
```
;; next-world : WorldState -> WorldState
;; Computes the next world state on a tick
(define (next-world w)
  (cond
    [(null? w) null]
    [else .... (first w) ....
          .... (next-world (rest w)) ....]))
```

next-world

```
;; next-world : WorldState -> WorldState
;; Computes the next world state on a tick
(define (next-world w)
  (cond
    [(null? w) null]
    [else (cons (next-ball (first w))
                (next-world (rest w)))]))
```

render-world

List **template!**

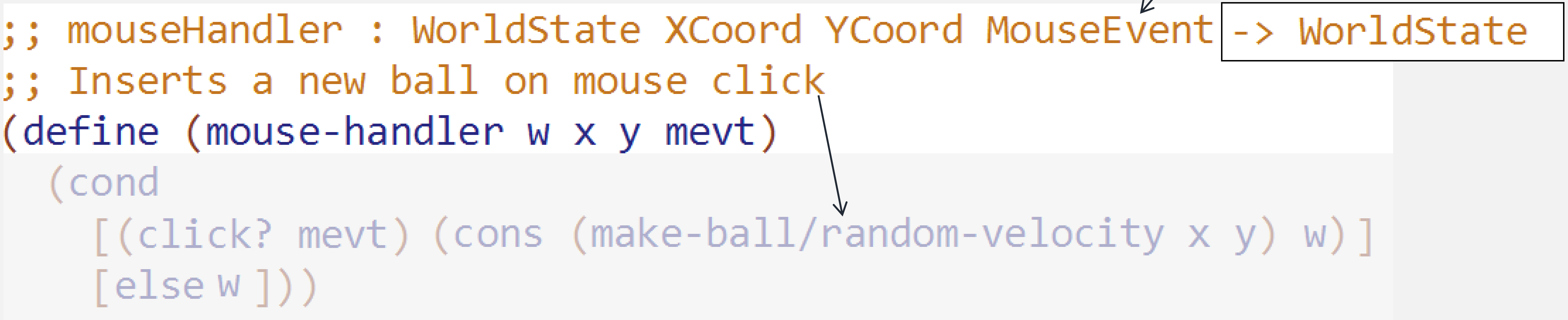


```
;; render-world : WorldState -> Image
;; Draws the given worldstate as an image
(define (render-world w)
  (cond
    [(null? w) EMPTY-SCENE]
    [else (place-ball (first w) (render-world (rest w)))]))
```

Separate “draw” function for the ball

For multi-arg function, you choose which (argument's) template to use

Enumeration



```
;; mouseHandler : WorldState XCoord YCoord MouseEvent -> WorldState
;; Inserts a new ball on mouse click
(define (mouse-handler w x y mevt)
  (cond
    [(click? mevt) (cons (make-ball/random-velocity x y) w)]
    [else w ]))
```

Enumeration template
(collapsed)

Multi-ball Animation: more?

Design a **big-bang** animation that:

- Start: a single ball, moving with random x and y velocity
- On a click: add a ball at random location, with random velocity
    - And **random size?**
    - And **random color?**
- If any ball “hits” an edge:
    - if it’s a **vertical** edge, **the x velocity should flip direction**
    - If it’s a **horizontal** edge, **the y velocity should flip direction**

```
;; A WorldState is … a list of balls!
```

Check-In Quiz 10/2

on gradescope

(due 1 minute before midnight)