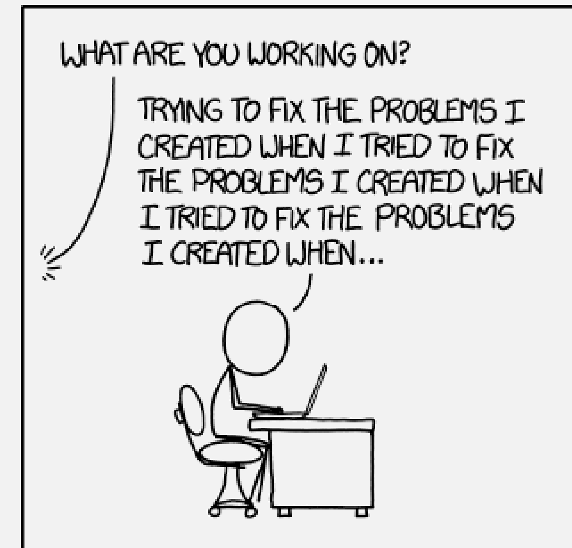
UMass Boston Computer Science

CS450 High Level Languages (section 2)

Recursive Data Definitions

Monday, September 30, 2024



Logistics

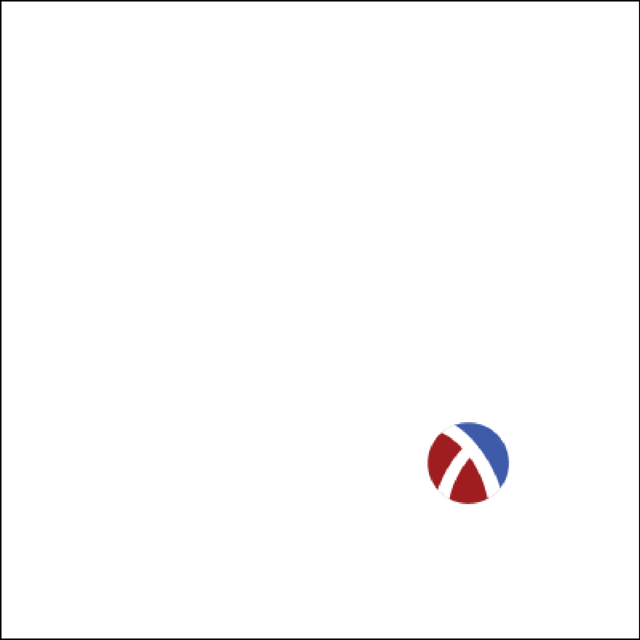
- HW 3 in
  - ~~due: Mon 9/30 12pm (noon) EST~~
- HW 4 out
  - due: Mon 10/7 12pm (noon) EST
- No class: Monday 10/14
  - Indigenous Peoples Day

(What's wrong with this recursion?)

No base case!

Last Time

Bouncing Ball



Make it bounce?

```
;; A WorldState is a
(struct world [x y xvel yvel])
;; where:
;; x: Coordinate – represents x coordinate of ball center
;; y: Coordinate – represents y coordinate of ball center
;; xvel: Velocity - in x direction
;; yvel: Velocity – in y direction
```

```
;; next-world : WorldState -> WorldState
;; Computes the next ball pos
```

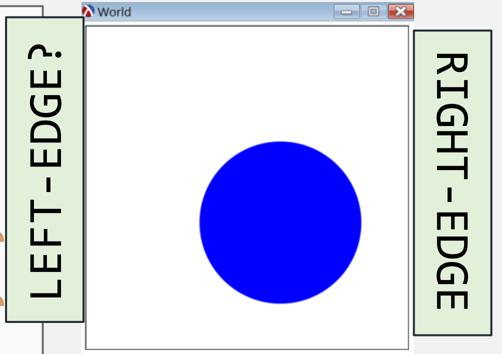
```
(define (next-world w)
  (match-define (world x y xvel yvel) w)



  (world (+ x xvel) (+ y yvel) xvel yvel)))
```

Make it bounce?

```
;; A WorldState is a
(struct world [x y xvel yvel])
;; where:
;; x: Coordinate - represents x coordinate of ball center
;; y: Coordinate - represents y coordinate of ball center
;; xvel: Velocity - in x direction
;; yvel: Velocity - in y direction
```



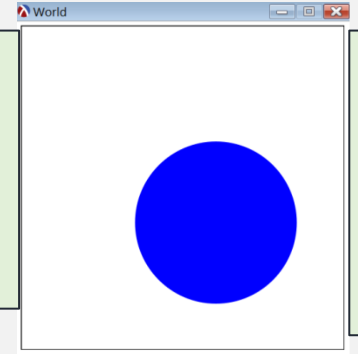

```
;; next-world : WorldState -> WorldState
;; Computes the next ball pos
```

```
(define (next-world w)
  (match-define (world x y xvel yvel) w)
  (define new-xvel
    (if (>= x RIGHT-EDGE) (- xvel) xvel))

  (world (+ x xvel) (+ y yvel) new-xvel yvel)))
```

Make it bounce?

```
;; A WorldState is a
(struct world [x y xvel yvel])
;; where:
;; x: Coordinate - represents x coordinate of ball center
;; y: Coordinate - represents y coordinate of ball center
;; xvel: Velocity - in x direction
;; yvel: Velocity - in y direction
```




```
;; next-world : WorldState -> WorldState
;; Computes the next ball pos
```

```
(define (next-world w)
  (match-define (world x y xvel yvel) w)
  (define new-xvel
    (if (or (>= x RIGHT-EDGE)
            (<= x LEFT-EDGE)) (- xvel) xvel)
  (world (+ x xvel) (+ y yvel) new-xvel yvel)))
```

Make it bounce?

```
;; A WorldState is a
(struct world [x y xvel yvel])
;; where:
;; x: Coordinate – represents x coordinate of ball center
;; y: Coordinate – represents y coordinate of ball center
;; xvel: Velocity - in x direction
;; yvel: Velocity – in y direction
```

```
;; next-world : WorldState -> WorldState
;; Computes the next ball pos
(define (next-world w)
  (match-define (world x y xvel yvel) w)
  (define new-xvel
    (if (or (>= x RIGHT-EDGE)
            (<= x LEFT-EDGE)) (- xvel) xvel)
  (world (+ x new-xvel) (+ y yvel) new-xvel yvel)))
```

Should this be **xvel** or **new-xvel**???

Make it bounce?

```
;; A WorldState is a
(struct world [x y xvel yvel])
;; where:
;; x: Coordinate - represents x coordinate of ball center
;; y: Coordinate - represents y coordinate of ball
;; xvel: Velocity - in x direction
;; yvel: Velocity - in y direction
```

If you're **no longer following the template**, then the **Data Definitions** need updating!

```
;; next-world : WorldState -> WorldState
;; Computes the next ball
(define (next-world w)
  (match-define (world x
  (define new-xvel
    (if (or (>= x RIGHT-EDGE)
            (<= x LEFT-EDGE
  (define new-yvel???
    (if (or (>= y BOTTOM-EDGE)
```

Keep hacking and hope that it works???

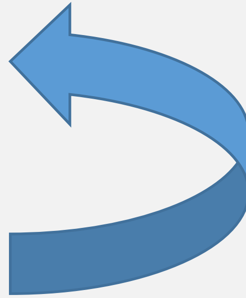
DON'T PROGRAM LIKE THIS!!!

This is **undisciplined** programming and is much **slower and error-prone** than thinking first!

Program Design Recipe

... is iterative!

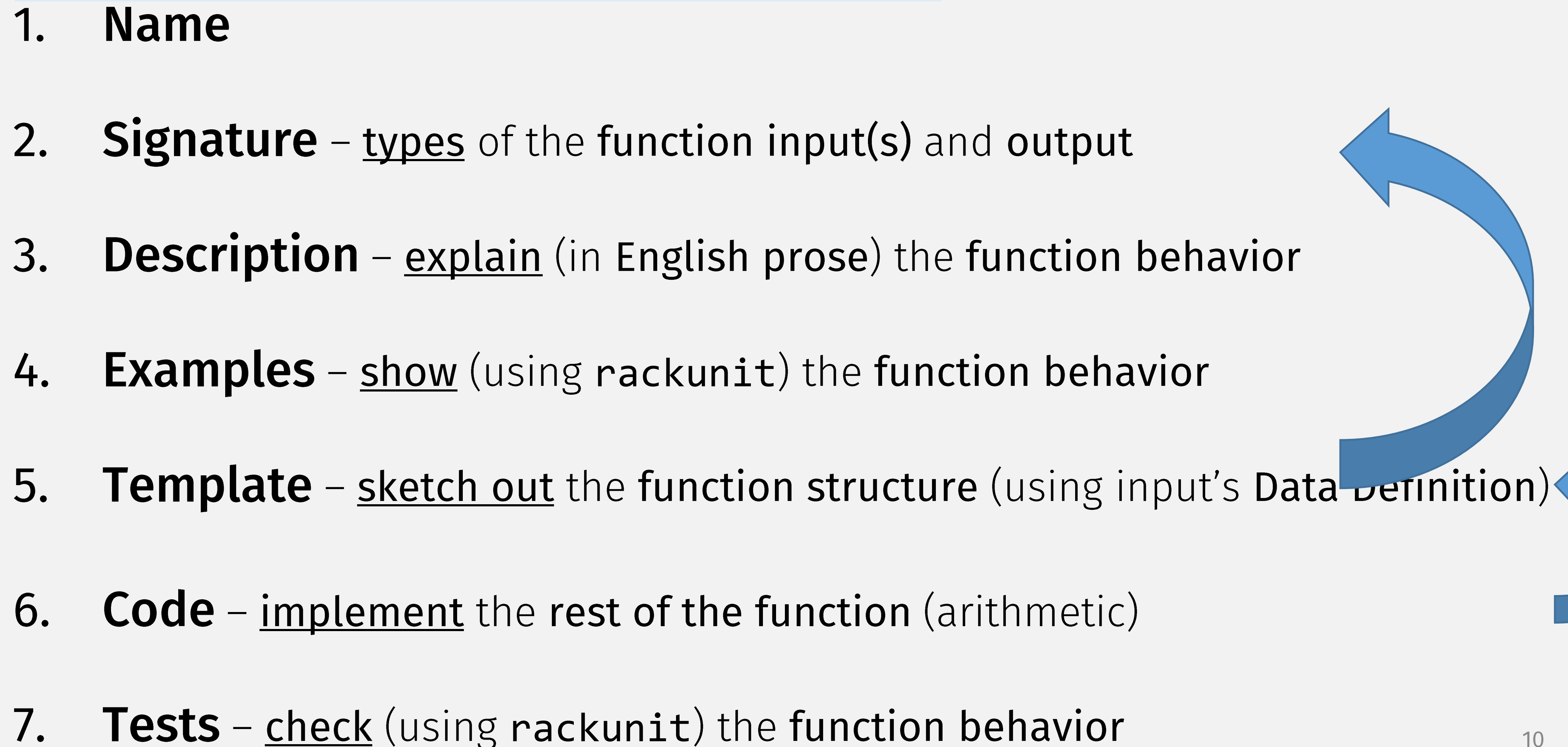
1. **Data Design**



2. **Function Design**

Function Design Recipe

... is iterative!



Make it bounce?

```
;; A WorldState is a
(struct world [x y xvel yvel])
;; where:
;; x: Coordinate - represents x coordinate of ball center
;; y: Coordinate - represents y coordinate of ball
;; xvel: Velocity - in x direction
;; yvel: Velocity - in y direction
```

If you're **no longer following the template**, then the **Data Definitions** need updating!

```
;; next-world : WorldState -> WorldState
;; Computes the next ball pos
(define (next-world w)
  (match-define (world x y xvel yvel) w)
  (define new-xvel
    (if (or (>= x RIGHT-EDGE)
            (<= x LEFT-EDGE)) (- xvel) xvel)
  (define new-yvel???
    (if (or (>= y BOTTOM-EDGE)
```

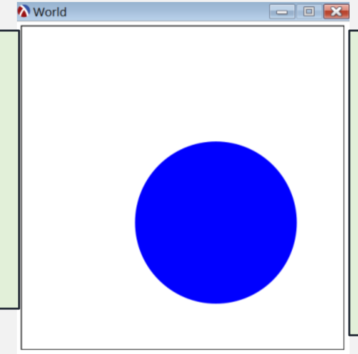
DON'T PROGRAM LIKE THIS!!!

Ma

```
;; A Coordinate is a Real
;; Represents x or y position on big-bang canvas
```

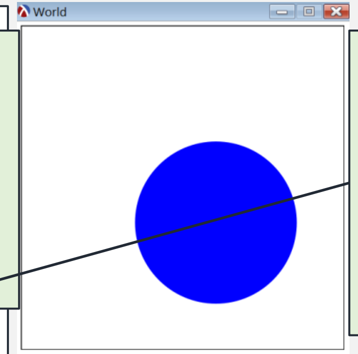
Seems like we want some **intervals**

```
;; A WorldState is a
(struct world [x y xvel yvel])
;; where:
;; x: Coordinate - represents x coordinate of ball center
;; y: Coordinate - represents y coordinate of ball center
;; xvel: Velocity - in x direction
;; yvel: Velocity - in y direction
```



Adding Intervals

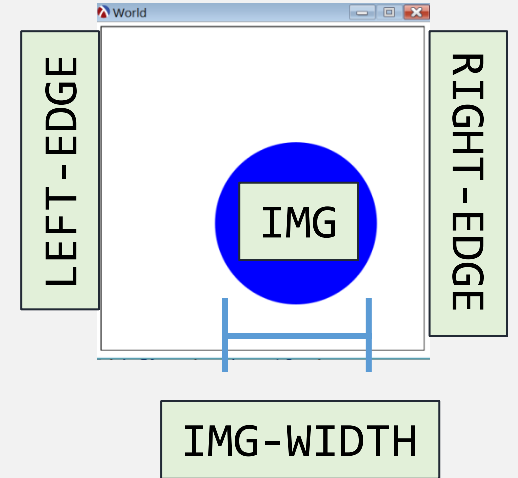
```
;; A WorldState is a
(struct world [x y xvel yvel])
;; where:
;; x: XCoordinate – represents x coordinate of ball center
;; y: Coordinate – represents y coordinate of ball center
;; xvel: Velocity - in x direction
;; yvel: Velocity – in y direction
```




```
;; An XCoordinate is a real number in one of these intervals:
;; (LEFT-EDGE, RIGHT-EDGE) : image fully within scene
;; (-infinity, LEFT-EDGE]  : (at least) part of image out of scene, to the left
;; [RIGHT-EDGE, +infinity) : (at least) part of imageout of scene, to the right
;; Interp: The coordinate is the x coordinate of image center;
;; the intervals represent whether the image is fully within t
```

WAIT! Is this correct?

Adding Intervals

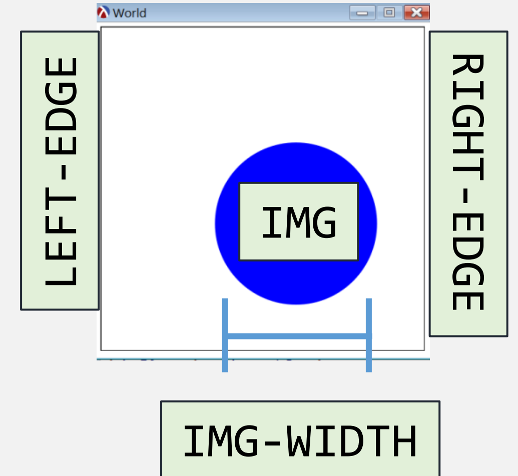



```
;; An XCoordinate is a real number in one of these intervals:
```

```
;; ( LEFT-EDGE + IMG-WIDTH/2, RIGHT-EDGE - IMG-WIDTH/2) : image fully within scene
;; (-inf, LEFT-EDGE + IMG-WIDTH/2]     : (part of) image out of scene, to the left
;; [RIGHT-EDGE - IMG-WIDTH/2, +inf)    : (part of) image out of scene, to the right
```

```
;; TEMPLATE???
```

Adding Intervals

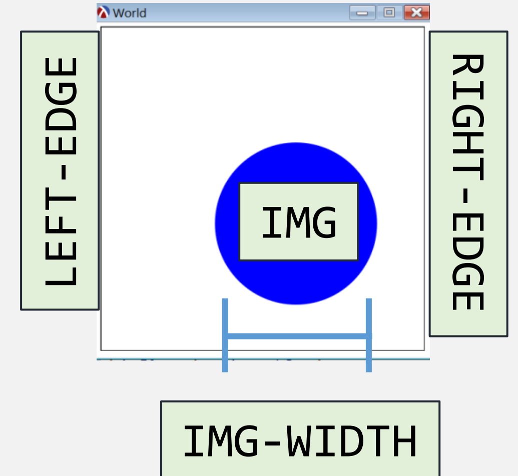


Now the **shape of the function** matches the **shape of the data definition**!

```
;; An XCoordinate is a real number in one of these intervals:
;; ( LEFT-EDGE + IMG-WIDTH/2, RIGHT-EDGE - IMG-WIDTH/2) : image fully within scene
;; (-inf, LEFT-EDGE + IMG-WIDTH/2]     : (part of) image out of scene, to the left
;; [RIGHT-EDGE - IMG-WIDTH/2, +inf)    : (part of) image out of scene, to the right
```

```
;; TEMPLATE
(define (x-fn x)
  (cond [(< (/ IMG-WIDTH 2) x (- RIGHT-EDGE (/ IMG-WIDTH 2))) ....]
        [(<= x (/ IMG-WIDTH 2)) ....]
        [(>= x (- RIGHT-EDGE (/ IMG-WIDTH 2))) ....]))
```

Adding Intervals



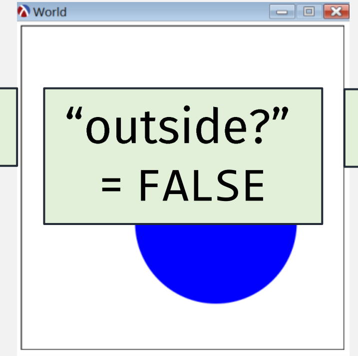

```
;; An XCoordinate is a real number in one of these intervals:
```

```
;; ( LEFT-EDGE + IMG-WIDTH/2, RIGHT-EDGE – IMG-WIDTH/2) : image fully within scene
;; (-inf, LEFT-EDGE + IMG-WIDTH/2]     : (part of) image out of scene, to the left
;; [RIGHT-EDGE – IMG-WIDTH/2, +inf)    : (part of) image out of scene, to the right
```

```
;; outside-L/R-edges? : XCoordinate -> Bool
(define (outside-L/R-edges? x)
  (cond [(< (/ IMG-WIDTH 2) x (- RIGHT-EDGE (/ IMG-WIDTH 2))) ....]
        [(<= x (/ IMG-WIDTH 2)) ....]
        [(>= x (- RIGHT-EDGE (/ IMG-WIDTH 2))) ....]))
```

Adding Intervals

“outside?” = TRUE



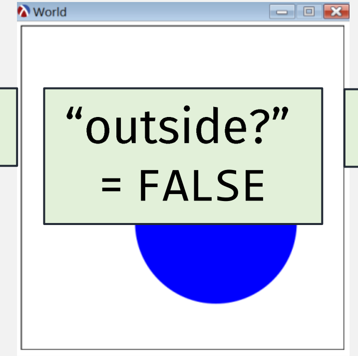
“outside?” = FALSE

TRUE

A **cond that evaluates to a boolean** is slightly <u>awkward</u> …
Because the **tests already compute the correct value**!

```
;; outside-L/R-edges? : XCoordinate -> Bool
(define (outside-L/R-edges? x)
  (cond [(< (/ IMG-WIDTH 2) x (- RIGHT-EDGE (/ IMG-WIDTH 2))) #false]
        [(<= x (/ IMG-WIDTH 2)) #true]
        [(>= x (- RIGHT-EDGE (/ IMG-WIDTH 2))) #true]))
```

Adding Intervals



A **cond that evaluates to a boolean** is slightly awkward …

Instead, use `or` and **just keep true cases**!

```
;; outside-L/R-edges? : XCoordinate -> Bool
(define (outside-L/R-edges? x)
  (or (<= x (/ IMG-WIDTH 2))
      (>= x (- RIGHT-EDGE (/ IMG-WIDTH 2)))))
```

Helper function?

Make it bounce?

```
;; A WorldState is a
(struct world [x y xvel yvel])
;; where:
;; x: Coordinate - represents x coordinate of ball center
;; y: Coordinate - represents y coordinate of ball center
;; xvel: Velocity - in x direction
;; yvel: Velocity - in y direction
```

```
;; next-world : WorldState -> WorldState
;; Computes the next ball pos
(define (next-world w)
  (match-define (world x y xvel yvel) w)
  (define new-xvel
    (if (or (>= x RIGHT-EDGE)
            (<= x LEFT-EDGE)) (- xvel) xvel)
  (define new-yvel???
    (if (or (>= y BOTTOM-EDGE)
```

DON'T PROGRAM LIKE THIS!!!

Computing new velocity

```
;; next-xvel : Xcoordinate Velocity -> Velocity
;; Computes a (possibly) new velocity, based on x position
```

```
(define (next-xvel x xvel)
  (if (outside-L/R-edges? x)
      (- xvel)
      xvel)))
```

flips

No flip

```
(check-equal? (next-xvel LEFT-EDGE -10) 10)
```

flips

```
(check-equal? (next-xvel RIGHT-EDGE 10) -10)
```

flips

```
(check-equal? (next-xvel (sub1 RIGHT-EDGE) 10) 10)
```

No flip

Make it bounce?

```
;; A WorldState is a
(struct world [x y xvel yvel])
;; where:
;; x: Coordinate – represents x coordinate of ball center
;; y: Coordinate – represents y coordinate of ball center
;; xvel: Velocity - in x direction
;; yvel: Velocity – in y direction
```

```
;; next-world : WorldState -> WorldState
;; Computes the next ball pos
(define (next-world w)
  (match-define (world x y xvel yvel) w)
  (define new-xvel
    (if (or (>= x RIGHT-EDGE)
            (<= x LEFT-EDGE)) (- xvel) xvel)
  (define new-yvel???
    (if (or (>= y BOTTOM-EDGE)
```

DON'T PROGRAM LIKE THIS!!!

Make it bounce?

```
;; A WorldState is a
(struct world [x y xvel yvel])
;; where:
;; x: Coordinate – represents x coordinate of ball center
;; y: Coordinate – represents y coordinate of ball center
;; xvel: Velocity - in x direction
;; yvel: Velocity – in y direction
```

1 **function** does
1 **task** which processes
1 **kind of data**

```
;; next-world : WorldState -> WorldState
;; Computes the next ball pos
```

```
(define (next-world w)
  (match-define (world x y xvel yvel) w)
  (define new-xvel (next-xvel x xvel))
  (define new-yvel (next-yvel y yvel))
  (world (+ x new-xvel) (+ y new-yvel) new-xvel new-yvel)))
```

Random

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

```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

[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.

Random Functions: Same Recipe (almost)!

```
;; 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 functions have <u>no examples</u>

```
(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

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

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!
```

Previously

Kinds of Data Definitions

- **Basic data**
  - E.g., **numbers, strings,** etc
- **Intervals**
  - Data that is from a **range of values,** e.g., [0, 100)
- **Enumerations**
  - Data that is **one of a list of possible values**, e.g., `"green"`, `"red"`, `"yellow"`
- **Itemizations**
  - Data value that can be from a **list of possible other data definitions**
  - E.g., **either** a **string** or **number** (Generalizes enumerations)

Previously

Kinds of Data Definitions

- Basic data
  - E.g., **numbers, strings,** etc
- Intervals
  - Data that is from a **range of values,** e.g., [0, 100)
- Enumerations
  - Data that is **one of a list of possible values**, e.g., "green", "red", "yellow"
- Itemizations
  - Data value that can be from a **list of possible other data definitions**
  - E.g., either a **string** or **number** (Generalizes enumerations)
- **Compound Data**
  - Data that is a **combination of values from other data definitions**

Last time

Multi-ball Animation

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

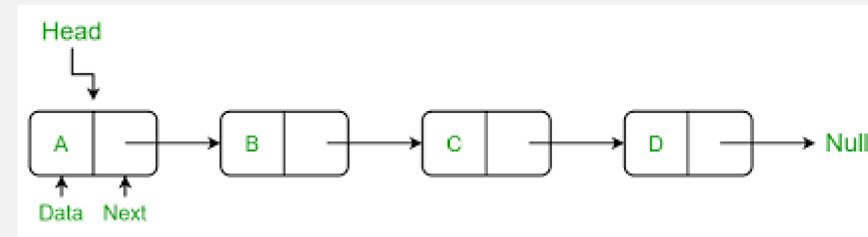
- Start: a single ball, moving with random x and y velocity
- On a click: add a ball at random location, with random velocity
- If any ball "hits" an edge:
  - if it's a **vertical** edge, the x velocity should flip direction
  - If it's a **horizontal** edge, the y velocity should flip direction

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

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
;; - empty
;; - (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
;; - empty
;; - (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??

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

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

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

Empty (**base**) case

Non-empty (**recursive**) case

Racket List Data Definition Example

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

TEMPLATE??

... is also recursive!

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

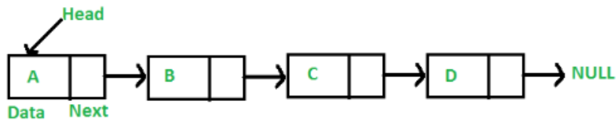
The **shape of the function** matches the **shape of the data definition**!

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
    [(empty? lst) ....]
    [(cons? lst) .... (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
    [(empty? 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
    [(empty? 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
    [(empty? 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
    [(empty? 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
    [(empty? 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
    [(empty? lst) empty]
    [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
    [(empty? lst) empty]
    [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
    [(empty? 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
    [(empty? 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)

Still follows design recipe!

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

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

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
      [(empty? lst) rev-lst-so-far]
      [else .... (first lst) ....
            .... (rev/tmp (rest lst))
            .... rev-lst-so-far ....
  (rev/tmp lst empty))
```

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
      [(empty? lst) rev-lst-so-far]
      [else (rev/tmp
              (rest lst)
              (cons (first lst) rev-lst-so-far))]))
  (rev/tmp lst empty))
```

Add next list item to reversed list "so far"

Multi-ball Animation

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

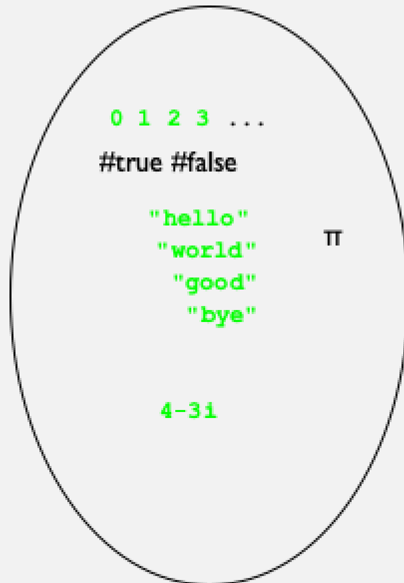
- Start: a single ball, moving with random x and y velocity
- On a click: add a ball at random location, with random velocity
- If any ball "hits" an edge:
  - if it's a **vertical** edge, **the x velocity should flip direction**
  - If it's a **horizontal** edge, **the y velocity should flip direction**

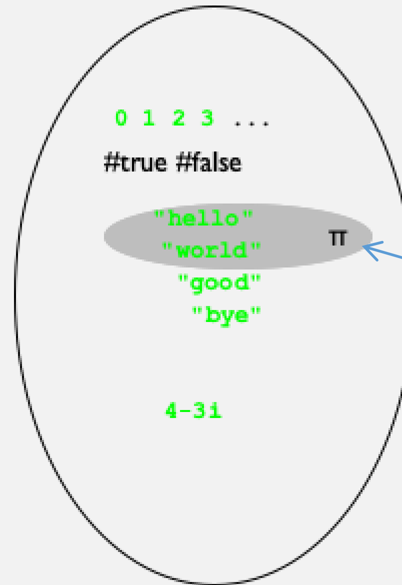
```
;; A WorldState is … an unknown number of balls!
```

```
;; 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!

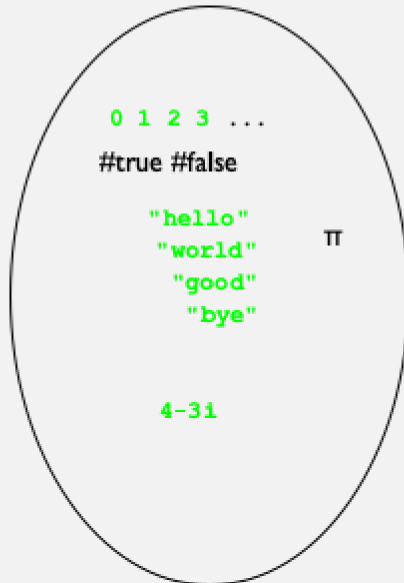
All programs are data manipulators …

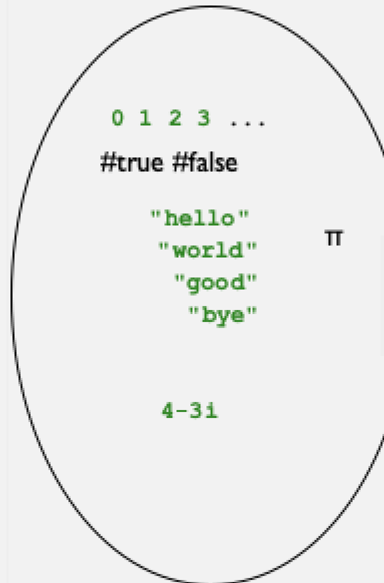
So this must be the first step of programming!

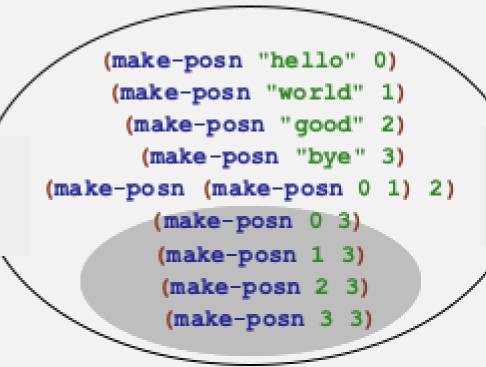
Also makes “error handling” easy

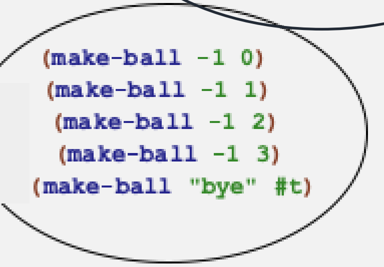
Interlude: Data Definitions (ch 5.7)

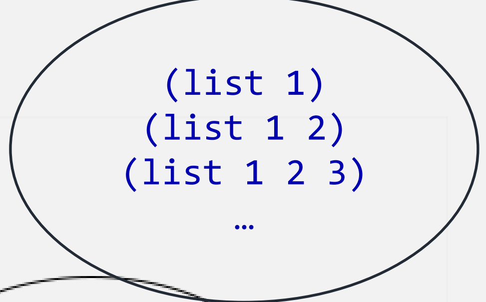
All possible basic data values











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)
;; wher
;; x: X          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
    [(empty? w) ....]
    [else .... (first w) ....
          .... (next-world (rest w)) ....]))
```

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
    [(empty? 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
    [(empty? w) empty]
    [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
    [(empty? w) empty]
    [else (cons (next-ball (first w))
                (next-world (rest w)))]))
```

1 function does
1 task which processes
1 kind of data

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

1 function does
1 task which processes
1 kind of data

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!
```

In-class exercise: start hw4

Write functions that process "**`Note`**"s and "**`ListofNote`**"s

- Add randomness: write a function **`insert-note?`** that takes no args and returns true approximately once every 100 calls
- Write a function **`Notes?`** that takes an arbitrary list and returns true if all are a **`Note?`**
  - If you follow the template, this is super easy

Submitting

1. File: `in-class-09-30-<Lastname>-<Firstname>.rkt`
2. Join the in-class team: `cs450f24/teams/in-class`
3. Commit to repo: `cs450f24/in-class-09-30`
   - (May need to `merge/pull + rebase` if someone `push`es before you)