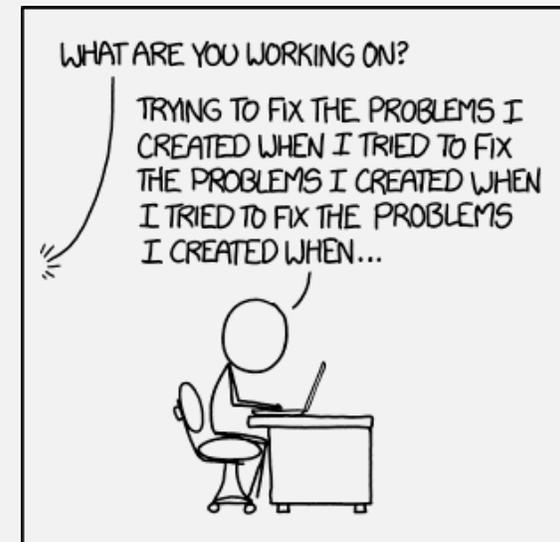UMass Boston Computer Science
CS450 **High Level Languages**
Recursive Data Definitions

Tuesday, March 3, 2026

Logistics

- HW 4 in
  - ~~due: Tue 3/3 11am EST~~

- HW 5 out
  - due: Tue 3/10 11am EST

No base case!

HW Advice

"Perhaps you thought that "**getting it working**" was the first order of business for a professional developer.

THIS COURSE

I hope by now, however, that ~~this book~~ has disabused you of that idea.

The functionality that you create today has a good chance of changing in the next release, but the **readability of your code** will have a profound effect on all the changes that will ever be made."

— **Robert C. Martin,**
Clean Code: A Handbook of Agile Software Craftsmanship

(won't work in this course, or real life)

Most students submit here
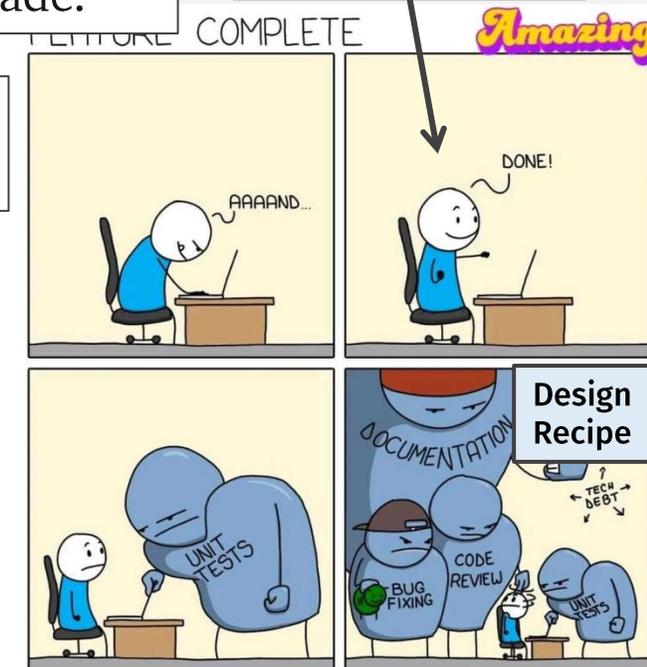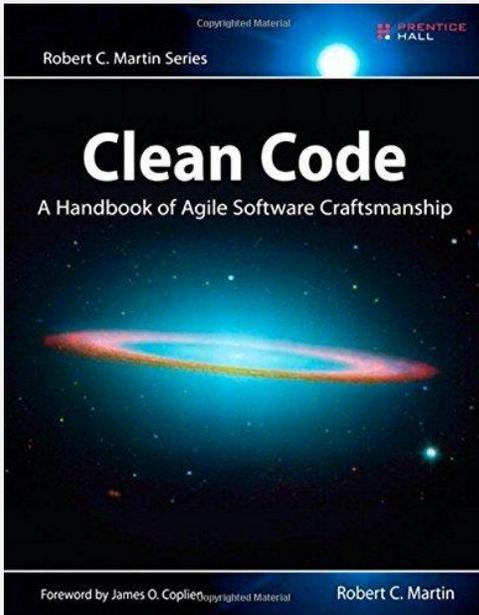
Design Recipe

HW Advice #1

Many submissions only focused on: **"getting it working"**
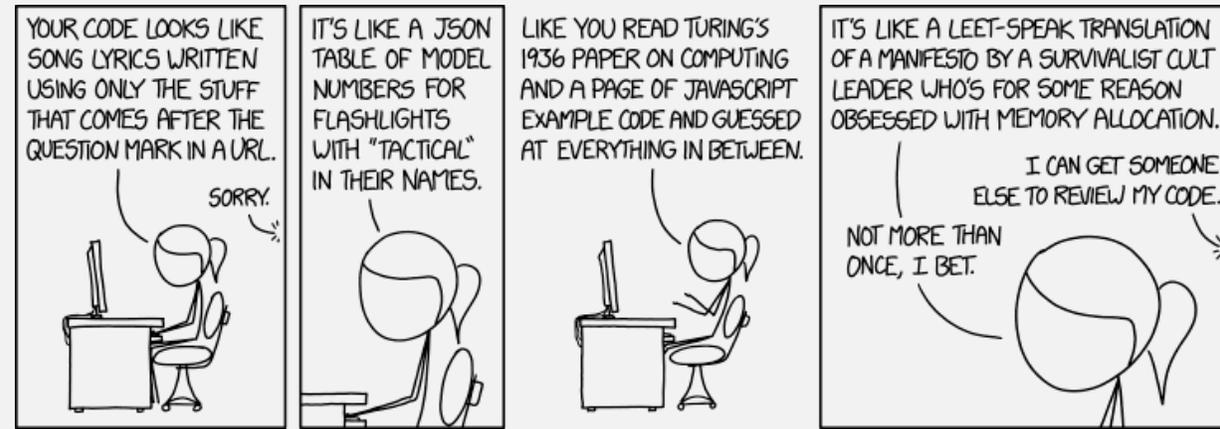
Many submissions ignored:

- Other steps of **Program Design Recipe**
- Tests!
  - (should actually do testing)
  - Not sufficient test: `image?`
- Style Guide
- Other HW Instructions

See Incremental Pledge, minimum requirements

This hw will be graded accordingly:

This may soon become <u>minimum</u> requirements

- correctness (Autograded) (10 pts)
- design recipe (15 pts)
- testing (15 pts)

Won't be able to get other points **unless submission is fully "correct"**

- style (7 pts)
- README (1 pt)

Total: 48 points

Minimum requirements

- Reminder: **Not ok to submit code** that … <small>(will not receive credit)</small>
  - doesn't (or hasn't been) **run**
  - **Crashes or errors when runs**
  - has **no tests**
  - has **failing / erroring tests**
  - doesn't **match Github** (???) or **no Github**
  - has **large blocks of commented code**
  - has **undescriptive commit msgs**
  - is uploaded with **GitHub "file upload" feature**

- See: **Incremental Programming Pledge!**

(minimum requirements)

The Incremental Programming Pledge

"slow down to speed up"

At all times, all of the following should be **true** of your code:

1. Comments (data defs, signatures, etc) match code

2. Code has no syntax errors
   1. E.g., missing / extra parens

3. Runs without runtime errors / exceptions
   1. E.g., use undefined variables, `div by zero`, call a "non function"

4. All tests pass

When you make a code edit that renders one of the above `false`, **STOP** …

… and don't do anything else until all the statements are true again.

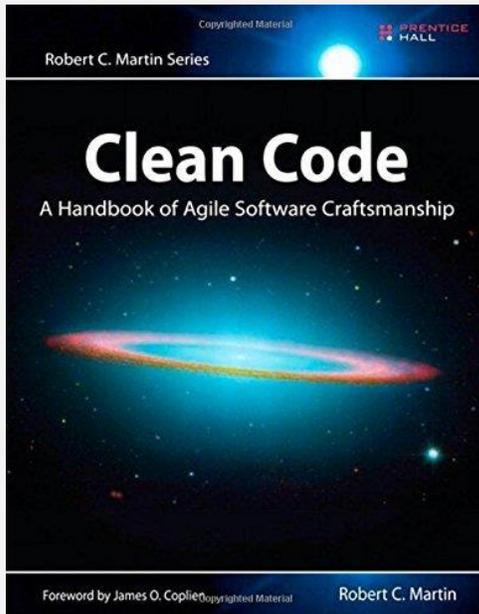(this way, it's easy to revert back to a "working" program)

HW Advice #2

"The first rule of functions is that they should be small.

The second rule of functions is that they should be smaller than that."

— **Robert C. Martin,**
Clean Code: A Handbook of Agile Software Craftsmanship

In this class:

1 function does
1 task which processes
1 kind of data

HW Observations / Reminders

i ain't reading all that

i'm happy for u tho

or sorry that happened

- **1 function,** does **1 task,** that processes **1 kind of data**
  - e.g., **handle-key**
- **Define helper function**(s)!

Follows template for:

```
;; handle-key:  GameState KeyEvent -> GameState
;; Update GameState on key
(define (handle-key gs key)
  (cond
    [(key=? key " ") (handle-space gs)]
    [else gs])
```

KeyEvent — Enum data

VS

```
(define/contract (handle-key g k)
  (-> gamestate? key-event? gamestate?)

  (cond

    ;; get red
    [(string=? k "r")
     (if (> (unowned-red-count g) 0)
         (switch-active
          (if (string=? (gamestate-active g) PLAYER1)
              (gamestate
               (player (+ 1 (player-red (gamestate-p1 g)))
                       (player-blue (gamestate-p1 g)))
               (gamestate-p2 g)
               (gamestate-active g))
              (gamestate
               (gamestate-p1 g)
               (player          ...ed (gamestate-p2 g)))
                       ...gamestate-p2 g)))
```

???

Follows template for:

```
;; handle-space : GameState  ->  GameState
;; Update GameState on space press
(define (handle-space gs)
  (mk-GameState
    (update-A (GameState-A gs))
    (update-B (GameState-B gs) (GameState-A gs)))))
```

GameState — compound

template:

Itemization

```
;; update-B : B-items -> ???
;; change "A" depending on what "B" is
(define (update-B b a)
  (cond
    [(X? b) (do-A-X a)]
    [(Y? b) (do-A-Y a)]
    [(Z? b) a])
```

"A" functions

Falling "Ball" Example
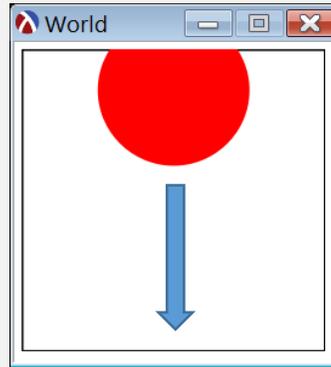
World

What if: the **ball can also move side-to-side**?

WorldState <u>would need</u>:
<u>**two**</u> **pieces of data** - the **x** *and* **y coordinates**

Falling "Ball" Example
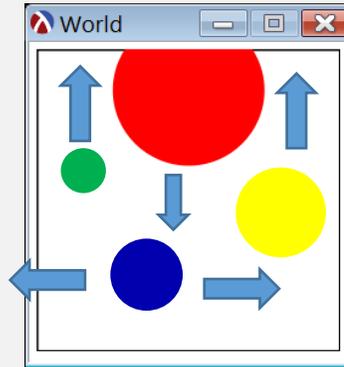
What if: the **ball can also move side-to-side**?

What if: **there are multiple balls?**

`WorldState` would need:
two pieces of data - the **x** *and* **y coordinates**

`WorldState` would need:
multiple (compound) **x** *and* **y coordinates**

DEMO

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., <u>either</u> a string or number (Generalizes enumerations)

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., <u>either</u> a string or number (Generalizes enumerations)
- **Compound Data** ???
  - Data that is a **combination of values from other data definitions**

Multi-ball Animation

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

- <u>Start</u>: a **single ball, moving** with **random x** and **y velocity**

- <u>On-click</u>: **add** a **ball** at click location, with **random velocity**

- If any ball <u>"hits" an edge</u>:
  - 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!
```

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., <u>either</u> a string or number (Generalizes enumerations)
- **Compound Data – Combines values from other data definitions**
  - **Fixed size** (e.g., `struct`)
  - today ➡️ **Arbitrary size**

Arbitrary Size Data - Lists

Where's **base case??**

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

In C
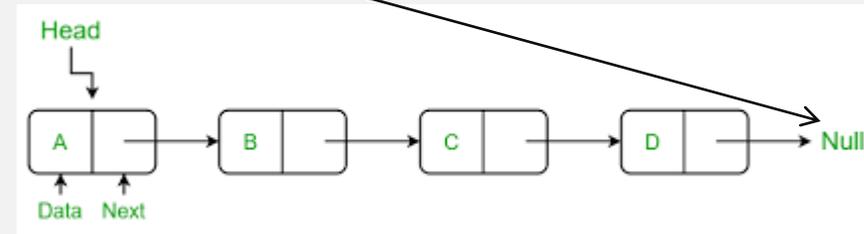
This is a
**terrible Data
Definition** ☹

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

Recursion is only **valid if** there is <u>both</u>
- A **base case**
- A **recursive case** (that is "smaller")

CS450 List Data Definition Example

(compare to C)

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

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

Empty (**base**) case

Non-empty (**recursive**) case

cons = "node" constructor

Recursive!
(using a definition to define itself)

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

(shallow) **predicate?**
Use **list?** (for now)

Recursion is only **valid if** there is both
- A **base** case
- A **recursive** case (that is "smaller")

List Constructor and Accessors

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

cons = "node" constructor

```
(first (cons 99 empty))            ; => 99
(rest (cons 99 (cons 88 empty)))   ; => (cons 88 empty)
```

Alternate List Constructor

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

```
(list 1 2 3) = (cons 1 (cons 2 (cons 3 empty)))
```

```
(first (list 1 2 3))   ; => 1
(rest (list 1 2 3))    ; => (list 2 3)
```

Also:
```
(second (list 1 2 3))  ; => 2
(third (list 1 2 3))   ; => 3
```

Racket List Data Definition Example

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

TEMPLATE??

(what kind of data definition is this?)

Racket List Data Definition Example

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

Empty (**base**) case

Non-empty (**recursive**) case

This is an **itemization**, so template has cond

The **shape** of the function <u>matches</u> The **shape** of the data definition!

TEMPLATE??

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

Empty (**base**) case

Non-empty (**recursive**) case

Racket List Data Definition Example

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

"first"

"rest"

This is **both**
itemization,
    so template has **cond**

and

compound data,
    so template has **"getters"**

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

Wait, where is the
recursion???

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

Racket List Data Definition Example

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

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

TEMPLATE??

... is also recursive!

The **shape** of **the function** <u>matches</u> The **shape** of **the data definition**!

So <u>recursion</u> in the **data definition** ... means <u>recursion</u> in the (template) **function**!

Racket Recursive List Fn Example: `sum-lst`

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

Head

A | | → B | | → C | | → D | | → NULL

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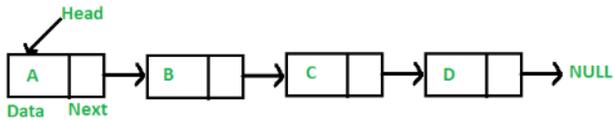
Task is to do A + B + C + D.

Examples:

geeksforgeeks.com

(follows Design Recipe!)

```
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
```
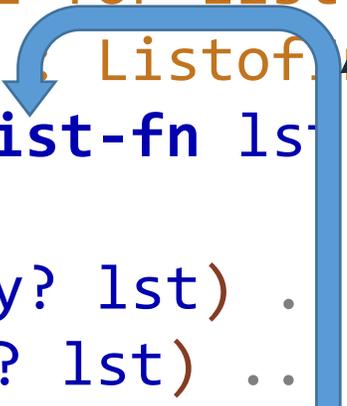
Examples!

```
;; 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-lst`

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-lst`

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

```
;; 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)) ....]))
```

NEXT: How to "combine" pieces?

Think about data types!

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)))]))
```

List contracts

shallow check,
but also (**delayed**)
"deeper check"
(only when needed)

```
;; Returns sum of list of ints
(define/contract (sum-lst lst)
  (-> (listof integer?) integer?)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst)
             (sum-lst (rest lst)))]))
```

Multi-ball Animation

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

- <u>Start</u>: a **single ball, moving** with **random x** and **y velocity**

- <u>On-click</u>: **add** a **ball,** with **random velocity**

- If any ball <u>"hits" an edge</u>:
  - 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])
;; wher  ball
;; 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
```

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

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

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

;; A WorldState is a ListofBall

"next World"

List **template!**

```
;; list-fn: WorldState -> ???
;; list of Ball template
(define (list-fn w)
  (cond
    [(empty? w) ....]
    [else .... (first w) ....
          .... (list-fn (rest w)) ....]))
```

"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)) ....]))
```

"next World" example

1 function does
1 task which processes
1 kind of data

```
;; 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

```
(check-equal?
  (next-World (list (mk-Ball 0 0 1 1)))
  (list (next-Ball (mk-Ball 0 0 1 1)))
)
```

"next Ball"

This was the <u>previous</u> "**next-World**" function!

```
;; computes  next  ition and vel of ball after 1 tick
(define (next-World w)
  (mat    ine (world x y xv yv) w)
  (mk-WorldState       ball
    (next-x x xv)
    (next-y y ...)
    (next-xv xv ...)
    (next-yv yv ...)))
```

Ball

Ball

ball

"next World"

Fill in template …

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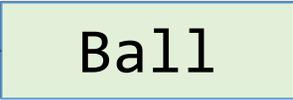
```
;; next-World : WorldState -> WorldState
;; Computes the next world state on a tick
(define (next-World w)
  (cond
    [(empty? w) ....]
    [else  .... (first w) ....
           .... (next-World (rest w)) ....]))
```

"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)) ....]))
```

Ball

1 function does
1 task which processes
1 kind of data

"next World"

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

NEXT: How to
"combine" pieces?

Think about
data types!

Want:
Ball + **ListofBall**
-> ListofBall

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

"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)))]))
```

NEXT: How to "combine" pieces?

Think about data types!

"Render World"

List **template!**

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

"Render World"

```
;; world-img : World -> Image
;; Draws Ball images into a scene
(define (world-img w)
  (cond
    [(empty? w) .... ]
    [else
        .... (first w) ....
        .... (world-img (rest w)) .... ]))
```

"Render World"

```
;; world-img : World -> Image
;; Draws Ball images into a scene
(define (world-img w)
  (cond
    [(empty? w)  EMPTY-SCENE ]
    [else
          .... (first w) ....
          .... (world-img (rest w)) .... ]))
```

Ball

"Combine" the pieces

"Render World"

```
;; world-img : World -> Image
;; Draws Ball images into a scene
(define (world-img w)
  (cond
    [(empty? w) EMPTY-SCENE]
    [else
  (ball-img (first w) ???
       .... (world-img (rest w)) .... ]))
```

Want:
Ball -> Image and:
Image + Image -> Image

??

"Combine" the pieces

"Render World"

```
;; world-img : World -> Image
;; Draws Ball images into a scene
(define (world-img w)
  (cond
    [(empty? w) EMPTY-SCENE]
    [else (add-Ball-to-scene
            (first w)
            (world-img (rest w)))]))
```

Or:
Ball + Image -> Image

??

"Combine" the pieces

"handle Mouse"

Enumeration

```
;; handle-mouse : WorldState Xcoord Ycoord MouseEvent
;; Inserts new random ball onclick
(define (handle-mouse w x y mevt)
  (cond
    [(click? mevt) (add-random-ball w)]
    [else w]))
```

```
;; handle-mouse : WorldState Xcoord Ycoord MouseEvent
;; Inserts new random ball onclick
(define (handle-mouse w x y mevt)
  (cond
    [(click? mevt) (add-random-ball w)]
    [else w])
```

`-> WorldState`

Falling "Ball" Example


```
;; A ListofBalls is one of
;; - empty
;; - (cons Ball ListofBalls)
```

```
;; A WorldState is a ListofBalls
```

```
(define INITIAL-WORLD
  (list (random-ball)))
```
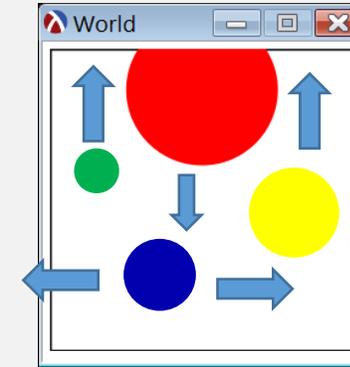
Not empty!

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)
```

predicate?

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


                              )
```

Just **cons?!**
shallow
(constant time)
check

Non-empty lists - template

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

Extract pieces of compound data (in both cases now)

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

template?

need to check <u>a little "deeper"</u> to distinguish cases (still a "**shallow**" check because not inspecting contents)

And recursive call

shape of the **function** <u>matches</u> **shape** of the **data definition**!

Next: Some Famous List Functions

- **Map**
- **Filter**
- **Fold** (reduce)

(2004)

Research at Google

Lecture: The Google MapReduce

Google Cloud Platform

"BIG"
data
processing

APACHE hadoop
(2006)

APACHE Spark™
(2010)

NVIDIA
(2026)
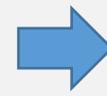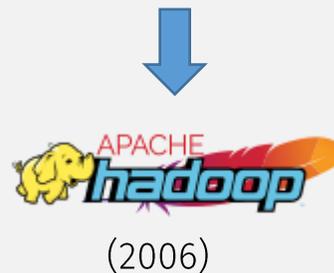
List Function Example

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

List Function Example: `inc-list`

```
;; inc-list : ListofInt -> ListofInt
;; increments each list element by 1
(define (inc-lst lst)
  (cond
    [(empty? lst) ....]
    [(cons? lst) .... (first lst)  ....
                 .... (inc-lst (rest lst)) ....]))
```

```
(check-equal?
  (inc-list (list 1 2 3))
            (list 2 3 4))
```

List Function Example: `inc-list`

```
;; inc-list : ListofInt -> ListofInt
;; increments each list element by 1
(define (inc-lst lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst) .... (first lst)  ....
              .... (inc-lst (rest lst)) ....]))
```

List Function Example: `inc-list`

```
;; inc-list : ListofInt -> ListofInt
;; increments each list element by 1
(define (inc-lst lst)
  (cond
    [(empty? lst) empty]
    [else ....  (add1 (first lst)) ....
          ....  (inc-lst (rest lst)) ....]))
```

Want:
Int + **ListofInt** -> **ListofInt**

List Function Example: `inc-list`

```
;; inc-list : ListofInt -> ListofInt
;; increments each list element by 1
(define (inc-lst lst)
  (cond
    [(empty? lst) empty]
    [else (cons (add1 (first lst))
                (inc-lst (rest lst)))]))
```

Multi-ball Animation

Design a **big-bang** animation that:
- <u>Start</u>: a **single ball, moving** with **random x** and **y velocity**
- <u>On a click</u>: **add** a **ball** with **random velocity**

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

next-world

List **template!**

```
;; next-world : WorldState -> WorldState
;; Updates position of all balls by one tick
(define (next-world w)
  (cond
    [(empty? w) ....]
    [else ....  (first w) ....
          ....  (next-world (rest w)) ....]))
```

next-world

```
;; next-world : WorldState -> WorldState
;; Updates position of all balls by one tick
(define (next-world w)
  (cond
    [(empty? w) empty]
    [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
;; Updates position of all balls by one tick
(define (next-world w)
  (cond
    [(empty? w) empty]
    [else ....  (next-ball (first w)) ....
          .... (next-world (rest w)) ....]))
```

Want:
Ball + ListofBall ->
ListofBall

next-world

```
;; next-world : WorldState -> WorldState
;; Updates position of all balls by one tick
(define (next-world w)
  (cond
    [(empty? w) empty]
    [else (cons (next-ball (first w))
                (next-world (rest w)))]))
```

next-world

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

```
;; next-world : ListofBall -> ListofBall
;; Updates position of all balls by one tick
(define (next-world lst)
  (cond
    [(empty? lst) empty]
    [else (cons (next-ball (first lst))
                (next-world (rest lst)))]))
```

Comparison

```
;; inc-lst: ListofInt -> ListofInt
;; Returns list with each element incremented
(define (inc-lst lst)
  (cond
    [(empty? lst) empty]
    [else (cons (add1 (first lst))
                (inc-lst (rest lst)))]))

(define (lst-fn1 fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (lst-fn1 (rest lst)))]))

    [else (cons (next-ball (first lst))
                (next-world (rest lst)))]))
```

Abstraction: Common List Function #1

```
;; lst-fn1: (?? -> ??) Listof?? -> Listof??
;; Applies the given fn to each element of given lst
```

```
(define (lst-fn1 fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (lst-fn1 (rest lst)))]))
```

```
(define (inc-lst lst) (lst-fn1 add1 lst)
(define (next-world lst) (lst-fn1 next-ball lst)
```

Abstraction: Common List Function #1

Argument is a function

```
;; lst-fn1: (X -> X) ListofX -> ListofX
;; Applies the given fn to each element of given lst
```

```
(define (lst-fn1 fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (lst-fn1 (rest lst)))]))
```

```
(define (inc-lst lst) (lst-fn1 add1 lst)
(define (next-world lst) (lst-fn1 next-ball lst)
```

Abstraction: Common List Function #1

```
;; lst-fn1: (X -> Y) ListofX -> ListofY
;; Applies the given fn to each element of given lst
```
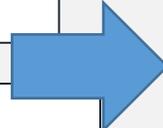
```
(define (lst-fn1 fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (lst-fn1 (rest lst)))]))
```

```
(define (inc-lst lst) (lst-fn1 add1 lst)
(define (next-world lst) (lst-fn1 next-ball lst)
```

Abstraction: Data Definitions

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

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

```
;; A Listof<X> is one of
;; - empty
;; - (cons X Listof<X>)
```

parameter

NOTE: this shows why our Compound data predicates should be "shallow" checks, i.e., `list?`

(Makes abstraction easier)

Not allowed in HW5!

To use this **abstract** data definition, must **instantiate** X with a **concrete** data definition

Listof<Int>

Listof<Ball>

(**concrete** = opposite of **abstract**)

Abstract Data Defs common in every PL

Generic / Abstract
data – "any" x and y
allowed

Instantiation

(C++ STL)

```
64  #include<iostream>
65  #include <vector>
66  using namespace std;
67
68  int main()
69  {
70      vector<int> v;
71
72      for (int i = 1; i <= 10; i++)
73      {
74          v.push_back(i);
75      }
76      cout << "Size : " << v.size();
77
78      v.resize(7);
79
80      cout << "\nAfter resizing it becomes : " << v.size();
```

Structs define abstract data

Instantiation

```
;; A Posn is a (mk-Posn [x : Int] [y : Int])
;; where
;; x: Int - represents x coordinate in big-bang animation
;; y: Int - represents y coordinate in big-bang animation
(struct posn [x y])
(define/contract (mk-Posn x y)
  (-> integer? integer? posn?)
  (posn x y))
```

Abstract data – "any" x and y allowed

Common List Function #1

```
;; lst-fn1: (X -> Y) Listof<X> -> Listof<Y>
;; Applies the given fn to each element of given lst
```

```
(define (lst-fn1 fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (lst-fn1 (rest lst)))]))
```

```
(define (inc-lst lst) (lst-fn1 add1 lst)
(define (next-world lst) (lst-fn1 next-ball lst)
```

Common List Function #1: map

```
;; map: (X -> Y) Listof<X> -> Listof<Y>
;; Applies the given fn to each element of given lst
```

```
(define (map fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (map (rest lst)))]))
```

```
(define (inc-lst lst) (map add1 lst)
(define (next-world lst) (map next-ball lst)
```

Common List Function #1: map

```
;; map: (X -> Y) Listof<X> -> Listof<Y>
;; Produces a list resulting from applying
;; a given fn to each element of a given lst
```

function "**application**"
(in high-level languges)
= function "call" (in
imperative languages)

```
(define (map fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (map (rest lst)))]))
```

```
(map proc lst ...+) → list?                          procedure
  proc : procedure?
  lst : list?
```

Applies *proc* to the elements of the *lst*s from the first elements to the last. The *proc* argument must accept the same number of arguments as the number of supplied *lst*s, and all *lst*s must have the same number of elements. The result is a list containing each result of *proc* in order.

Examples:

```
> (map (lambda (number1 number2)
         (+ number1 number2))
       '(1 2 3 4)
       '(10 100 1000 10000))
'(11 102 1003 10004)
```

```
(check-equal? (map + (list 1 2 3)
                     (list 4 5 6)
                     (list 5 7 9))
```

RACKET's map takes
multiple lists

`map` in other high-level languages

Array.prototype.map()

The `map()` method of `Array` instances creates a new array populated with the results of calling a provided function on every element in the calling array.

JavaScript Demo: Array.map()

```
1  const array1 = [1, 4, 9, 16];
2
3  // Pass a function to map
4  const map1 = array1.map((x) => x * 2);
5
6  console.log(map1);
7  // Expected output: Array [2, 8, 18, 32]
```

Lambda
("arrow function expression")

Python3

```
# Add two lists using map and lambda

numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]

result = map(lambda x, y: x + y, numbers1, numbers2)
print(list(result))
```

lambda

Common List Function #2: ???

Racket Recursive List Fn Example: `sum-lst`

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

Racket Recursive List Fn Example: `sum-lst`

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

Render World: ListofBall edition

```
;; render-world : ListofBall -> Image
;; Draws the given world as an image by overlaying each ball,
;; at its position, into an initially empty scene
```

```
(define (render-world lst)
  (cond
    [(empty? lst) .... ]
    [else .... (first lst) .... (render-world (rest lst)) ....]))
```

Render World: ListofBall edition

```
;; render-world : ListofBall -> Image
;; Draws the given world as an image by overlaying each ball,
;; at its position, into an initially empty scene
```

```
(define (render-world lst)
  (cond
    [(empty? lst) EMPTY-SCENE]
    [else .... (first lst) .... (render-world (rest lst)) ....]))
```

Render World: ListofBall edition

```
;; render-world : ListofBall -> Image
;; Draws the given world as an image by overlaying each ball,
;; at its position, into an initially empty scene
```

```
(define (render-world lst)
  (cond
    [(empty? lst) EMPTY-SCENE]
    [else (place-ball-into-scene (first lst) (render-world (rest lst)))]))
```

Create one function per "task"

```
;; place-ball-into-scene : Ball Image -> Image
;; Draws a ball into given scene, using its pos as the offset
(define (place-ball-into-scene b scene)
  (place-image BALLIMG (ball-x b) (ball-y b) scene))
```

Comparison #2

```
;; sum-lst: ListofInt -> Int
(define (sum-lst lst)
  (cond
   [(empty? lst) 0]
   [else (+ (first lst)
            (sum-lst (rest lst)))]))
```

```
;; render-world : ListofBall -> Image
(define (render-world lst)
  (cond
   [(empty? lst) EMPTY-SCENE]
   [else (place-ball (first lst)
                     (render-world (rest lst)))]))
```

Common List Function #2

```
;; list-fn2 : (X Y -> Y) Y Listof<X> -> Y

(define (lst-fn2 fn initial lst)
  (cond
    [(empty? lst) initial]
    [else (fn (first lst) (lst-fn2 fn initial (rest lst)))]))
```

```
;; sum-lst: ListofInt -> Int
(define (sum-lst lst) (list-fn2 + 0 lst))
;; render-world: ListofBall-> Image
(define (render-world lst) (list-fn2 place-ball EMPTY-SCENE lst))
```

Common List Function #2: foldr (start at right)

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
    [(empty? lst) initial]
    [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

Function recurs and builds up fn calls until it gets to the end

Then they are evaluated, last one first

```
;; sum-lst: ListofInt -> Int
(define (sum-lst lst) (foldr + 0 lst))
;; render-world: ListofBall-> Image
(define (render-world lst) (foldr place-ball EMPTY-SCENE lst))
```

Common List Function #2: `foldr`

```
;; foldr: (X … Y -> Y) Y Listof<X> … -> Y
```

```
(foldr proc init lst ...+) → any/c
  proc : procedure?
  init : any/c
  lst : list?
```

Racket version can also take multiple lists

Also **called "reduce"**
Because a **list of values** is
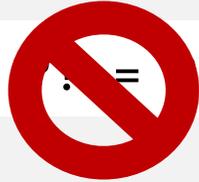**"reduced"** to **one value**

Do we always want to start at the right?

For some functions, order doesn't matter, but for others, it does?

```
(foldr + 0 (list 1 2 3)) = (1 + (2 + (3 + 0)))
```

```
(1 + (2 + (3 + 0))) = (((1 + 0) + 2) + 3)
```

(Addition is associative)

```
(1 - (2 - (3 - 0)))  ≠  ? (((1 - 0) - 2) - 3)
```

Need List Function #2b: `foldl` (start from left)

Challenge:

- **Change `foldr` to `foldl`**
- so that the **function is applied from the left** (first element first)

```
(define (foldr fn initial lst)
  (cond
    [(empty? lst) initial]
    [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

```
(1 + (2 + (3 + 0)))
```

```
(1 - (2 - (3 - 0)))
```

```
(define (foldl fn initial lst)
  (cond
    [(empty? lst) ....]
    [else .... (first lst) .... (foldl fn initial (rest lst))) ....]))
```

```
(((1 + 0) + 2) + 3)
```

```
(((1 - 0) - 2) - 3)
```

Need List Function #2b: `foldl` (start from left)

Y = Result Type

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y

(define (foldr fn initial lst)
  (cond
    [(empty? lst) initial]
    [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

Expressions with **needed "result"** type:
- `initial`
- **fn** call
- recursive call itself

(look at signature to help)

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y

(define (foldl fn initial lst)
  (cond
    [(empty? lst) ....]
    [else .... (first lst) .... (foldl fn initial (rest lst))) ....]))
```

Need List Function #2b: `foldl` (start from left)

Y = Result Type

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
    [(empty? lst) initial]
    [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

Expressions with needed "result" type:
- initial
- fn call
- recursive call itself

(look at signature to help)

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y
(define (foldl fn initial lst)
  (cond
    [(empty? lst) ....]
    [else (foldl .... (first lst) .... (rest lst))]))
```

Now fill in args to recursive call

Need List Function #2b: `foldl` (start from left)

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y

(define (foldr fn initial lst)
  (cond
    [(empty? lst) initial]
    [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y

(define (foldl fn initial lst)
  (cond
    [(empty? lst) .....]
    [else (foldl fn .... (first lst) .... (rest lst)))]))
```

only argument with type of first arg is first arg itself

Need List Function #2b: `foldl` (start from left)

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
    [(empty? lst) initial]
    [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

Expressions with **needed "result" Y type:**
- **initial**
- **fn** call ⬅
- recursive call itself

Now just **need middle arg** (and need to use the "**first**" piece)

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y
(define (foldl fn initial lst)
  (cond
    [(empty? lst) ....]
    [else (foldl fn .... (first lst) .... (rest lst))]))
```

"rest" of list has **proper "list" type**

Need List Function #2b: `foldl` (start from left)

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y

(define (foldr fn initial lst)
  (cond
    [(empty? lst) initial]
    [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

Expressions with "result" Y type:
- `initial` ←
- `fn` call
- recursive call itself

Now just **need middle arg** (and need to use the "`first`" piece)

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y

(define (foldl fn initial lst)
  (cond
    [(empty? lst) ....]
    [else (foldl fn (fn (first lst) ....) (rest lst)))]))
```

`(((1 + 0) + 2) + 3)`

What goes here? (look at signature)

(and examples)

Need List Function #2b: `foldl` (start from left)

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y

(define (foldr fn initial lst)
  (cond
    [(empty? lst) initial]
    [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

Expressions with "result" Y type:
- `initial` ←
- `fn` call
- recursive call itself

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y

(define (foldl fn initial lst)
  (cond
    [(empty? lst) ....] ←
    [else (foldl fn (fn (first lst) initial) (rest lst)))]))
```

`(((1 + 0) + 2) + 3)`

Need List Function #2b: `foldl` (start from left)

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
    [(empty? lst) initial]
    [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

Expressions with "result" Y type:
- `initial` ⬅
- `fn` call
- recursive call itself

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y
(define (foldl fn initial lst)
  (cond
    [(empty? lst) initial]
    [else (foldl fn (fn (first lst) initial) (rest lst))]))
```

"initial"???

```
(((1 + 0) + 2) + 3)
```

Need List Function #2b: `foldl` (start from left)

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

Expressions with "result" Y type:
- ~~initial~~ **result-so-far**
- **fn** call
- recursive call itself

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y
(define (foldl fn result-so-far lst)
  (cond
   [(empty? lst) result-so-far]
   [else (foldl fn (fn (first lst) result-so-far) (rest lst))]))
```

"result so far"

$((((1 + 0) + 2) + 3)$

Need List Function #2b: `foldl` (start from left)

Challenge:

- **Change `foldr` to `foldl`**

- so that the **function is applied from the left** (first element first)

```
(define (foldr fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

```
(define (foldl fn initial lst)
  (cond
   [(empty? lst) ....]
   [else .... (first lst) .... (foldl fn initial (rest lst))) ....]))
```

Common List Function #2: `foldl` / `foldr`

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y
;; Computes a single value from given list, determined by given fn and initial val.
;; fn is applied to each list element, last-element-first
```

```
(1 + (2 + (3 + 0)))
```
```
(1 - (2 - (3 - 0)))
```

```
(define (foldr fn initial lst)
  (cond
    [(empty? lst) initial]
    [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y
;; Computes a single value from given list, determined by given fn and initial val.
;; fn is applied to each list element, first-element-first
```

```
(((1 + 0) + 2) + 3)
```
```
(((1 - 0) - 2) - 3)
```

```
(define (foldl fn result-so-far lst)
  (cond
    [(empty? lst) result-so-far]
    [else (foldl fn (fn (first lst) result-so-far) (rest lst))]))
```

fold (`reduce`) in other high-level languages

JavaScript Demo: Array.reduce()

```
1  const array1 = [1, 2, 3, 4];
2
3  // 0 + 1 + 2 + 3 + 4
4  const initialValue = 0;
5  const sumWithInitial = array1.reduce((resultSoFar, x) => resultSoFar + x, initial);
6
7  console.log(sumWithInitial);
8  // Expected output: 10
9
```

"list"

lambda

"initial"

JavaScript Demo: Array.reduceRight()

```
1  const array1 = [
2    [0, 1],
3    [2, 3],
4    [4, 5],
5  ];
6
7  const result = array1.reduceRight((resultSoFar, x) => resultSoFar.concat(x));
8
9  console.log(result);
10 // Expected output: Array [4, 5, 2, 3, 0, 1]
11 |
```

"initial" optional?

Fold "dual": `build-list`

```
(build-list n proc) → list?                                    procedure
  n : exact-nonnegative-integer?
  proc : (exact-nonnegative-integer? . -> . any)
```

Creates a list of *n* elements by applying *proc* to the integers from `0` to `(sub1 n)` in order. If *lst* is the resulting list, then `(list-ref lst i)` is the value produced by `(proc i)`.

Examples:

```
> (build-list 10 values)
'(0 1 2 3 4 5 6 7 8 9)
> (build-list 5 (lambda (x) (* x x)))
'(0 1 4 9 16)
```

```
(build-list 4 add1)

;; = (map add1 (list 0 1 2 3))

;; = (list 1 2 3 4)
```

More common list functions?

- Filter
- Find
- Reverse
- Append

Look at documentation for: `racket/list`

In-class exercise 3/3
on gradescope