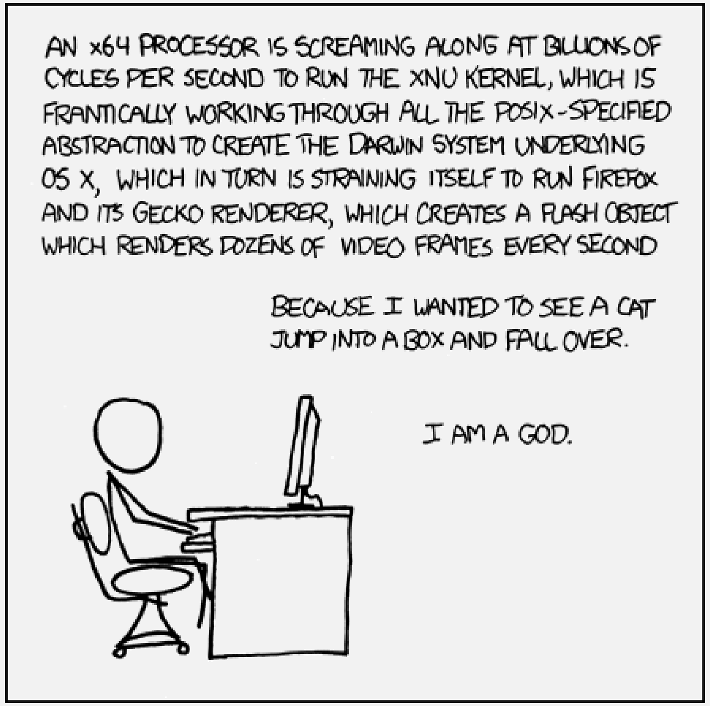
UMass Boston Computer Science

CS450 High Level Languages

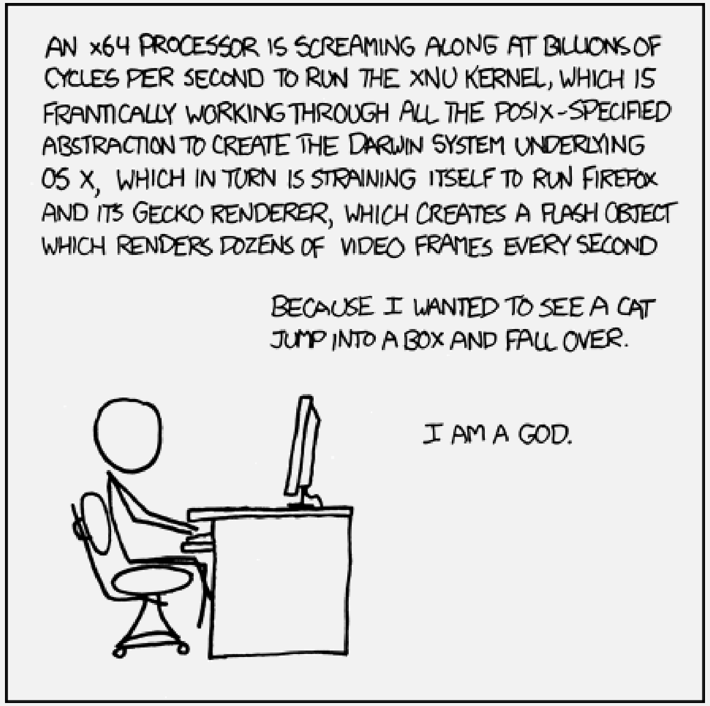
Abstraction

Thursday, February 27, 2025



Logistics

- HW 4 out
  - due: Tue 3/4 11am EST



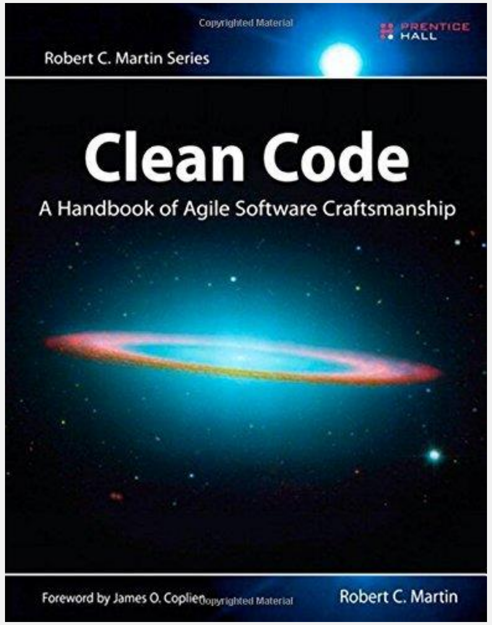
HW Advice #1

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

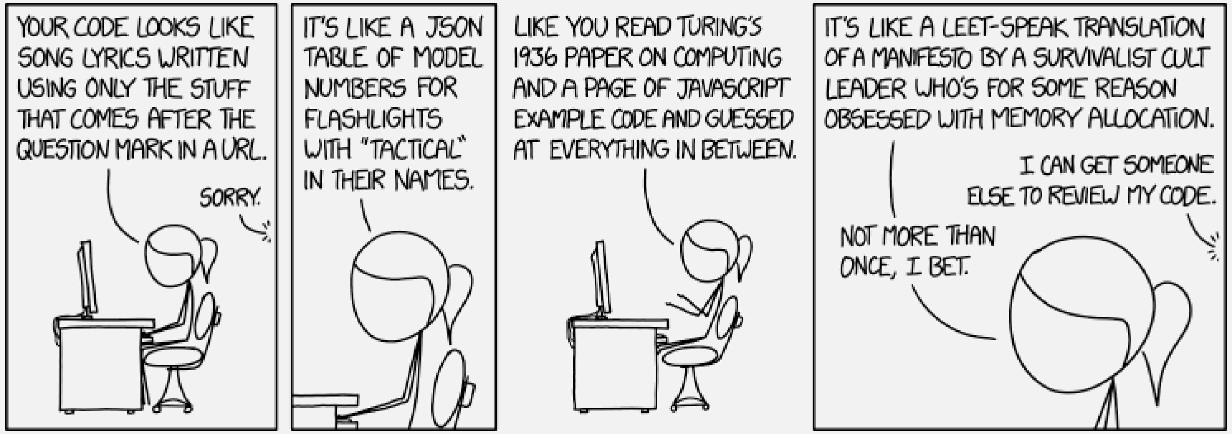
I hope by now, however, that this book has disabused you of that idea.

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



HW Advice #1



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

Many submissions ignored:

- Other steps of **Program Design Recipe**
- **Tests!**
- **Style Guide**
- **Other HW Instructions**

This hw will be graded accordingly:

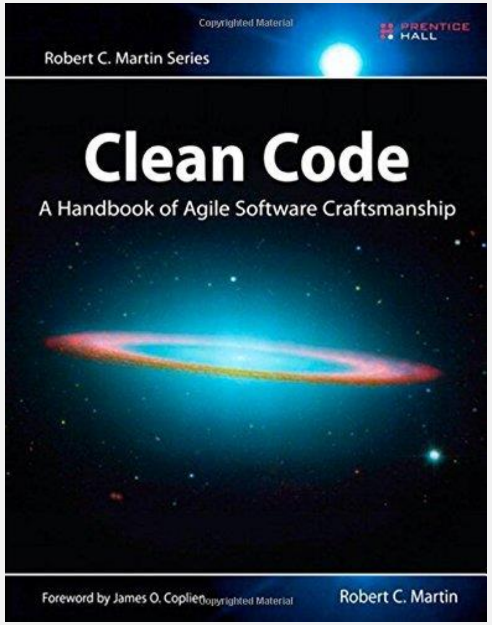
- correctness (autograder) (6 pts)
- design recipe (12 pts)
- testing (12 pts)
- style (5 pts)
- README (1 pt)

Total: 36 points

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

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

???

```
(define/contract (key-handler ws key)
  (-> WorldState? string? WorldState?)
  (if (and (string=? key " ")
           (<= (abs (- (+ (world-state-x ws) (/ REC-WIDTH 2))
                       (/ SCENE-WIDTH 2)))
               (/ REC-WIDTH 2)))
      (make-world-state (world-state-x ws)
                        (if (string=? (world-state-recfill ws) "solid")
                            "outline"
                            "solid"))
      ws))
```

Follows template for:

KeyEvent

Enum data

```
;; handle-key: WorldState KeyEvent -> WorldState
;; Update WorldState (rect                n key press
(define (handle-key ws key)
  (cond
    [(key=? key " ") (handle-space ws)]
    [else ws]))
```

vs

Follows template for:

WorldState

compound

```
;; handle-space : WorldState -> WorldState
;; Update WorldState (rect               n space press
(define (handle-space w)
  (mk-WorldState
    (update-x (world-x w))
    (update-rec (world-x w) (world-rec w))))
```

template:

XCoord

Itemization – of invervals

```
;; update-rec : XCoord RecType-> RecType
;; change rect c                        aps midline
(define (update-rec x rectype)
  (cond
    [(OverlapX? x) (toggle-rec-color rectype)]
    [else rectype]))
```

Last Time

A Recursive Data Definition

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

Empty (**base**) case

Non-empty (**recursive**) case

Recursive!
(using a definition to define itself)

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

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

Last Time

List Constructor and Accessors

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

"first" "rest"

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
```

Last Time

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

TEMPLATE??

(what kind of data definition is this?)

Template: Itemization

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

Empty (**base**) case

Non-empty (**recursive**) case

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

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

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

Template: Itemization + Compound Data

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

"first" "rest"

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

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

```
;; 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**???

Template: Itemization + Compound + Recursion

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

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

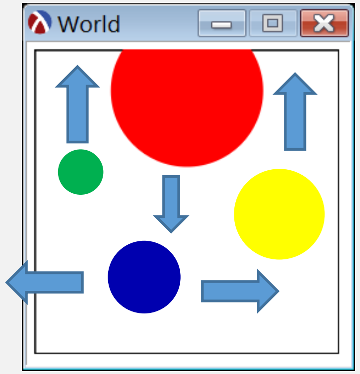
Recursion in the **data definition**
means …
Recursion in the (**template**) **function**!

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

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)

template?

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

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)

Research at Google (2004)

Lecture: **The Google MapReduce**

Google Cloud Platform

“BIG” data processing

APACHE hadoop (2006)

APACHE Spark (2010)

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)))]))
```

Previously

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

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

```
;; A    Ball    is a
(struct  ball  [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
```

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

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

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

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

```
ick
    [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

```
;; 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

Argument is a function

```
;; 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

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

<u>Makes abstraction easier</u>

```
;; 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

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

```
#include<iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> v;

    for (int i = 1; i <= 10; i++)
    {
        v.push_back(i);
    }
    cout << "Size : " << v.size();

    v.resize(7);

    cout << "\nAfter resizing it becomes : " << v.size();
```

(C++ STL)

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))
```

Not allowed in HW4!

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

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

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

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

Applies *proc* to the elements of the *lsts* from the first elements to the last. The *proc* argument must accept the same number of arguments as the number of supplied *lsts*, and all *lsts* 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)
```

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()

```
const array1 = [1, 4, 9, 16];

// Pass a function to map
const map1 = array1.map((x) => x * 2);

console.log(map1);
// 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: ???

Previously

Racket Recursive List Fn Example: **sum-lst**

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

Previously

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 (first lst) (render-world (rest lst)))]))
```

Create one function per “task”

```
;; place-ball : Ball Image -> Image
;; Draws a ball, using its pos as the offset, into the given image
(define (place-ball 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

X = Type of list element

Y = Result Type

```
;; 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))
```

Not allowed in HW4!

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
(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)))
```

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

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

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

JavaScript Demo: Array.reduce()

```javascript
const array1 = [1, 2, 3, 4];

// 0 + 1 + 2 + 3 + 4
const initialValue = 0;
const sumWithInitial = array1.reduce((resultSoFar, x) => resultSoFar + x, initial);

console.log(sumWithInitial);
// Expected output: 10

```

"list"

lambda

"initial"

JavaScript Demo: Array.reduceRight()

```javascript
const array1 = [
  [0, 1],
  [2, 3],
  [4, 5],
];

const result = array1.reduceRight((resultSoFar, x) => resultSoFar.concat(x));

console.log(result);
// Expected output: Array [4, 5, 2, 3, 0, 1]

```

"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)
```

Not allowed in HW4!

Next time: Other common list functions?

- Filter
- Find
- Reverse
- Append

Look at documentation for: `racket/list`

In-class exercise 2/27

on gradescope