UMass Boston Computer Science

CS450 High Level Languages (section 2)

Abstraction

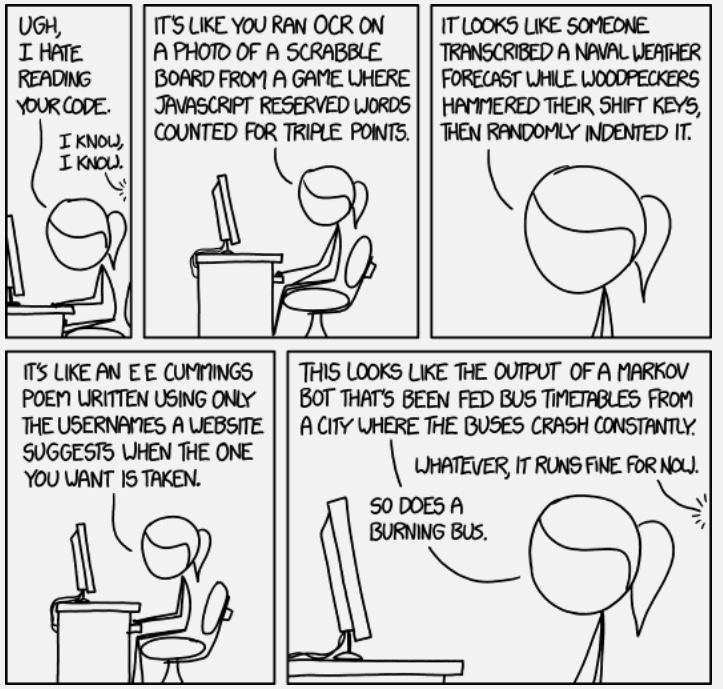
Wednesday, October 11, 2023

Logistics

- HW 3 out
  - due: Sun 10/15 11:59 pm EST

- HW 2 (and other) grades returned
  - Use GradeScope re-grade request for complaints or questions

CS 450 so far …

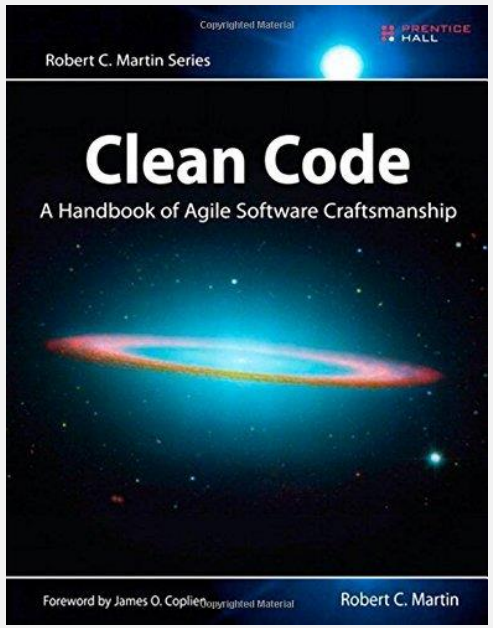


This class teaches:

- a **high-level programming "process"**
- i.e., a **design recipe** for creating clean, readable programs

- How to do well: **learn and follow the process**

- How to not do well: **focus only on "getting the code working"**

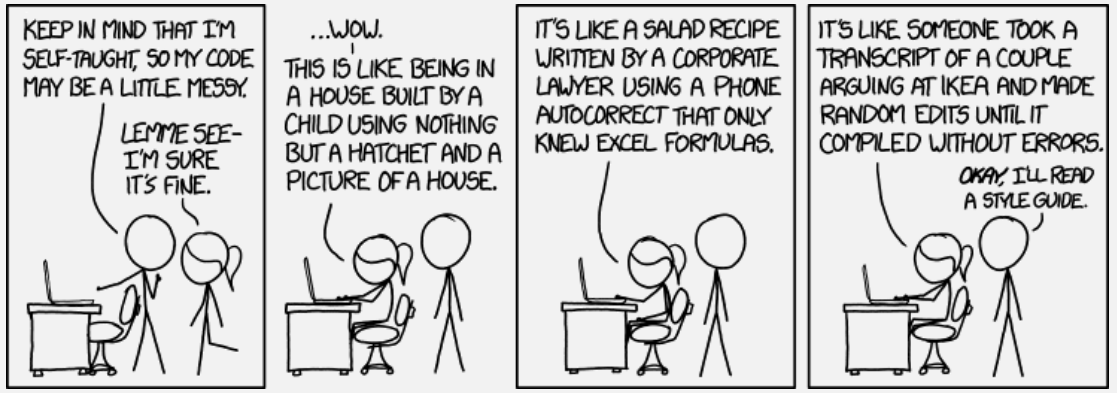


> “Perhaps you thought that “getting it working” was the first order of business for a professional developer.
>
> I hope by now, however, that this book has disabused you of that idea.

> 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 2 recap



Many submissions only focused on: **"getting the code working"**

Many submissions ignored:

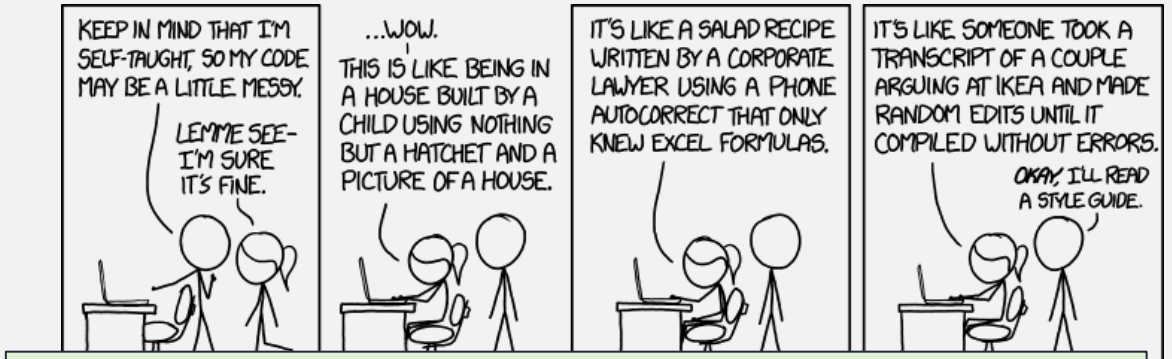
- all other steps of **programming design recipe**
- style guide
- Other instructions in hw

This hw will be graded accordingly:

- correctness (9 pts)
- design recipe (20 pts)
- style (5 pts)
- README (1 pt)

Total: 35 points

HW 2 solution (part 2)



"world state" now needs two "posn"s!

- To practice reading code, save the `big-bang` program from lecture 6 to a file named `hw2-bigbang.rkt`, and change it so when the user clicks the left-mouse button, a copy of the image is pinned to the canvas. (The original image should continue to move with the cursor.) Subsequent clicks should move the pinned image so that only one image is pinned at any time.

HW 2 recap – key design points

```
;; A WorldState is a:
(struct world [cursor pinned])
;; where
;; cursor : Posn - IMG location that moves with mouse cursor
;; pinned : MaybePosn - IMG location that is pinned (if there is one)
```

"world state" now needs two "posn"s!

one of them "maybe" will not have a value

```
;; A Posn is a
(struct posn [x y])
;; where
;; x: Integer - represents x coordinate in big-bang animation
;; y: Integer - represents y coordinate in big-bang animation
```

```
;; A MaybePosn is a
;; - Posn
;; - NO-POSN
;; represents a position, if there is one
```

```
;; constants and predicates for MaybePosn
(define NO-POSN #false)
(define (no-posn? x) (equal? x NO-POSN))
(define (maybe-posn? x) (or (posn? x) (no-posn? x)))
```

NOTE: skipped **Examples** Step

Mouse handler

```
;; mouse-handler : WorldState Coordinate Coordinate MouseEvent -> WorldState
;; Returns a WorldState where:
;; - "cursor" = current mouse loc
;; - "pinned" = current mouse loc (if mevt = "button-down")
```

```
;; TEMPLATE fn for MouseEvent : MouseEvent -> ???
(define (mevt-fn mevt)
  (cond
    [(string=? mevt "button-down") ....]
    .... ]))
```

Function splitting rule:

One (data definition processing) task, one function

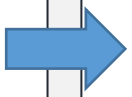
When a function has more than one argument, you choose which template to use

Mouse handler

```
;; mouse-handler : WorldState Coordinate Coordinate MouseEvent -> WorldState
;; Returns a WorldState where:
;; - "cursor" = current mouse loc
;; - "pinned" = current mouse loc (if mevt = "button-down")
```

```
(define (mouse-handler w x y mevt)

  (cond
    [(string=? mevt "button-down")
     (world (posn x y) (posn x y))]
    [else
     (world (posn x y) (world-pinned w))]))
```

Current mouse loc



```
(define (mouse-handler w x y mevt)
  (define current-mouse-pos (posn x y))
  (cond
    [(string=? mevt "button-down")
     (world current-mouse-pos
            current-mouse-pos)]
    [else
     (world current-mouse-pos
            (world-pinned w))]))
```

This field always the same

Mouse handler

```
;; mouse-handler : WorldS
;; Returns a WorldState w
;; - "cursor" = current m
;; - "pinned" = current m
```

```
(define (mouse-handler w x y mevt)
  (define current-mouse-pos (posn x y))
  (world
    current-mouse-pos
    (cond
     [(string=? mevt "button-down") current-mouse-pos)]
     [else (world-pinned w)])))
```

WAIT! Function split rule is:

One (data definition processing) task,

one function

Do we need a separate (world-processing) function?

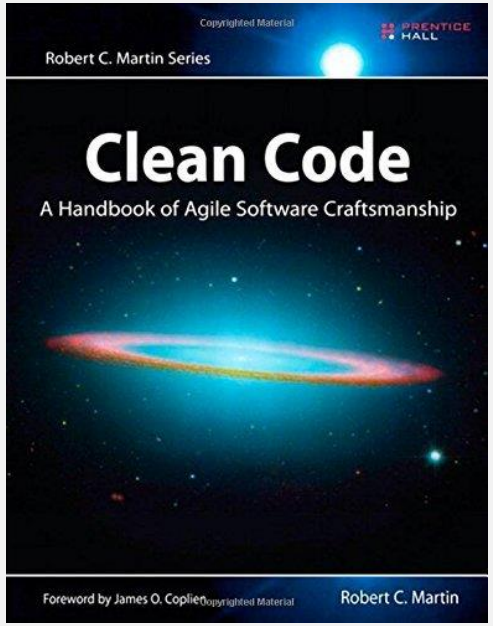
```
(define (mouse-handler w x y mevt)
  (define current-mouse-pos (posn x y))
  (cond
    [(string=? mevt "button-down")
     (world current-mouse-pos
            current-mouse-pos)]
    [else
     (world current-mouse-pos
            (world-pinned w))]))
```

On when to create a new function …

"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:

create one function per
(data definition processing) task

NOTE: skipped Contracts, and **Tests** Step

Mouse handler

```
;; mouse-handler : WorldS
;; Returns a WorldState w
;; - "cursor" = current m
;; - "pinned" = current m
```

```
(define (mouse-handler w x y mevt)
  (define current-mouse-pos (posn x y))
  (world
    current-mouse-pos
    (cond
     [(string=? mevt "button-down") current-mouse-pos)]
     [else (world-pinned w))])))
```

WAIT! Function split rule is:

One (data definition processing) task,

one function

Do we need a separate (world-processing) function?

YES. (But in this case, it would just **get** the **`world-pinned`** value)

Render world

```
;; render-world : WorldState -> Image
;; draws IMG at the "cursor" and "pinned" posn (if there is one)
```

```
;; TEMPLATE fn for WorldState: WorldState -> ???
(define (world-fn w)
  .... (world-cursor w) .... (world-pinned w) ....)
```

These are posns. Should we also extract posn fields here?

NO. They should be handled by a “posn” function (which follows “posn” template)

create one function per (data definition processing) task

Render world

```
;; render-world : WorldState -> Image
;; draws IMG at the "cursor" and "pinned" posn (if there is one)
```

```
;; TEMPLATE fn for WorldState: WorldState -> ???
(define (world-fn w)
  .... (world-cursor w) .... (world-pinned w) ....)
```

```
(define (render-world w)
  (render-posn
   (world-pinned w)
   (render-posn
    (world-cursor w)
    EMPTY-SCENE)))
```

But this is a “maybe” posn

Wish list:
`maybe-render-posn`

Render posn

```
;; maybe-render-posn : MaybePosn Image -> Image
;; draws IMG into given image at given posn;
;; unless posn is NO-POSN, then return img unchanged
```

```
;; TEMPLATE fn for MaybePosn fn: MaybePosn-> ???
(define (maybeposn-fn p)
  (cond
    [(posn? p) .... (posn-x p) .... (posn-y p) ....)]
    [(no-posn? p) ....]))
```

```
(define (maybe-render-posn p img)
  (cond
    [(posn? p) (place-image IMG (posn-x p) (posn-y p) img)]
    [else img]))
```

Render world

```
;; render-world : WorldState -> Image
;; draws IMG at the "cursor" and "pinned" posn (if there is one)
```

```
(define (render-world w)
  (maybe-render-posn
   (world-pinned w)
   (maybe-render-posn
    (world-cursor w)
    EMPTY-SCENE)))
```

Works for both!

Wish list:
~~maybe-render-posn~~

HW 2 recap - other points

- GitHub repos must be added to `cs450f23` organization (not your own account)
  - Otherwise I cant see it
  - (see directions in hw)

- Git commit messages must be meaningful

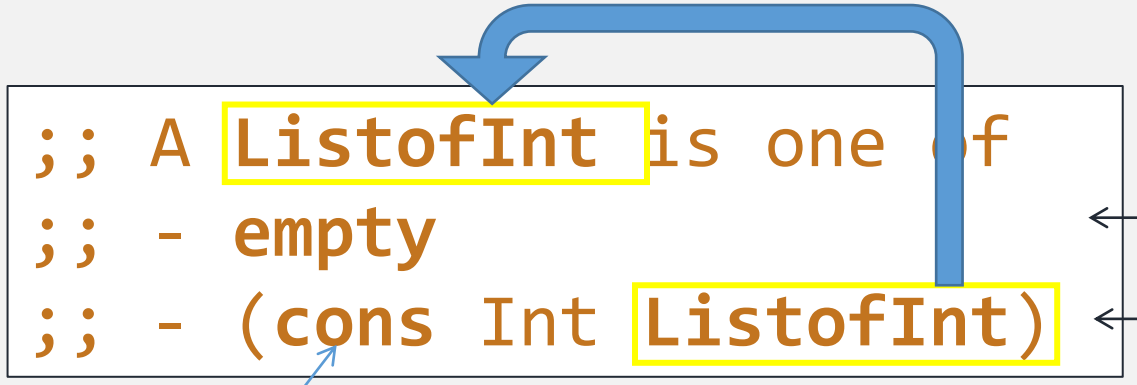
Do not …

- Add to temp files to repo
- Leave commented out code
- Leave “TODO”s in code

Lists and List Functions Review

Previously

Racket List Data Definition Example



Empty (**base**) case

Non-empty (**recursive**) case

cons = "node"

Recursive!
(using a definition to define itself)

TEMPLATE??

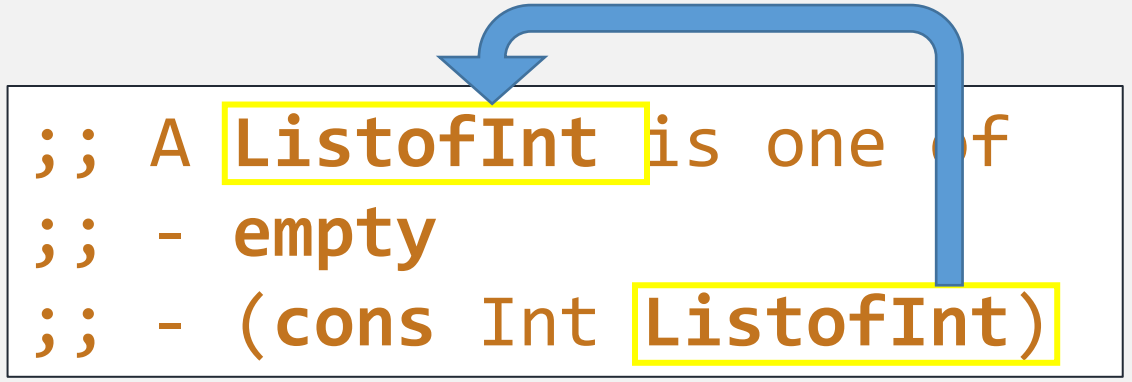
(how can we **use a list of ints to define a list of ints**?!?)

Recursion is a valid concept (from math), but **only if there is <u>both</u>**

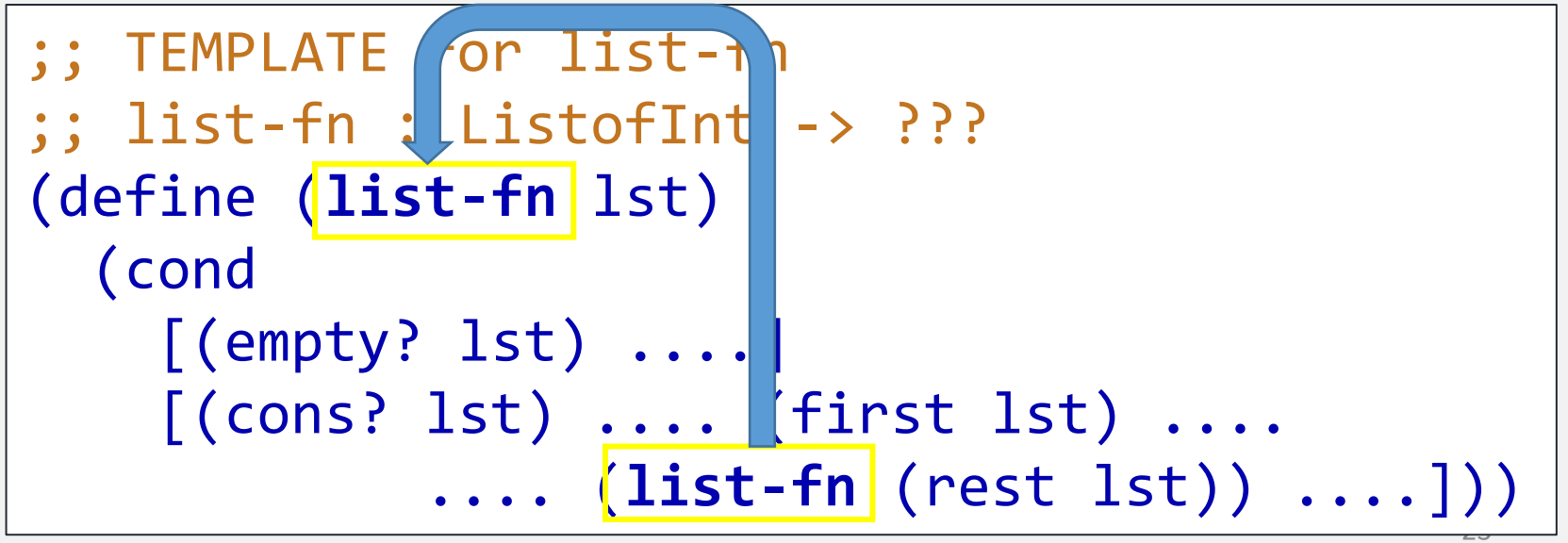
- A **base** case
- A **recursive** case

Previously

Racket Recursive List Fn Template



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



```
;; 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: `inc-list`

```
;; 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: `inc-list`

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

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

Racket Recursive List Fn: `inc-list`

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

Racket Recursive List Fn: `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))
                (list-fn (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
;; - null
;; - (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)) ....]))
```

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

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)))]))
```

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

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

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

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

```
(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 Compound data predicates should be **"shallow" checks,** i.e., `list?`

Makes abstraction easier

```
;; 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>)
```

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

`Listof<Int>`

`Listof<Ball>`

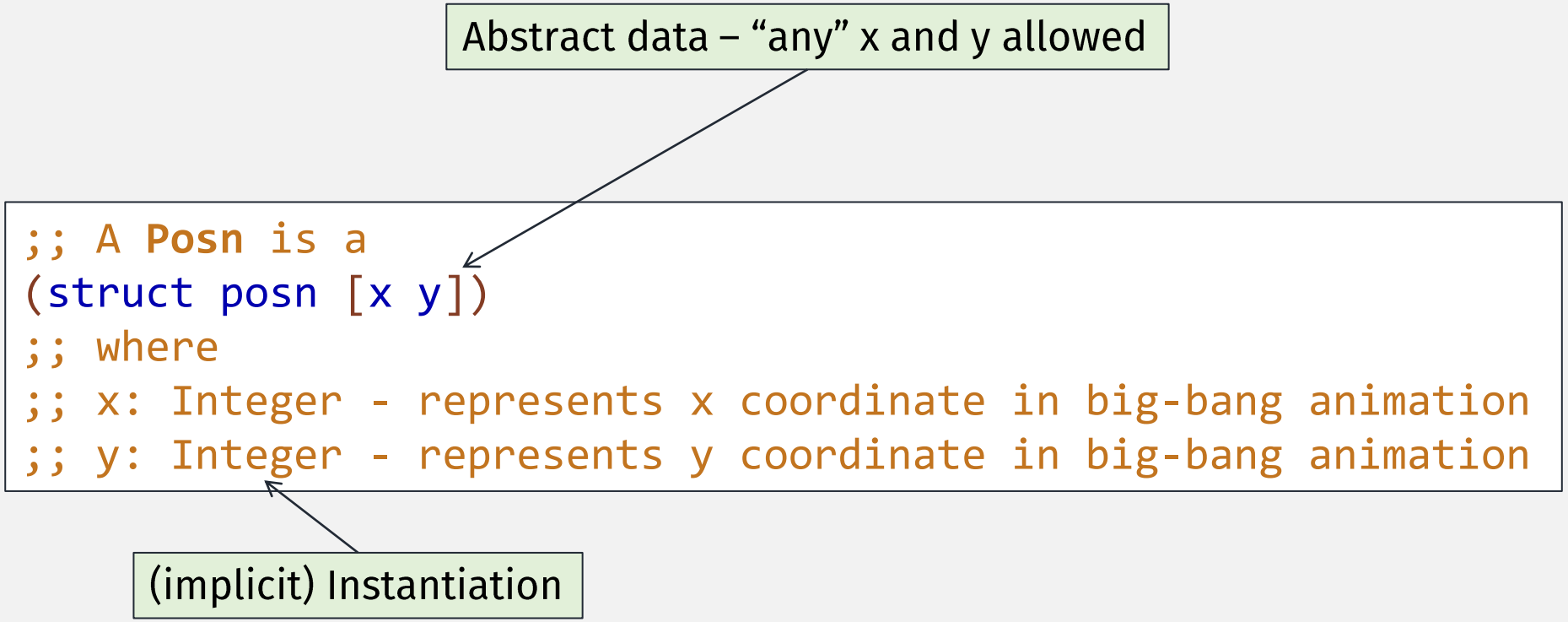
(**concrete** = opposite of **abstract**)

Abstract Data Defs common in every PL

```cpp
#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



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

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

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

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

Common List Function #1: **map**

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

```
map: (A B C … -> Z) Listof<A> Listof<B> Listof<C> … -> Listof<Z>
;; Applies the given fn to elements (at same index) of given lsts
```

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

Common List Function #2: ???

Previously

Racket Recursive List Fn Example: sum

```
;; 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

```
;; 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”

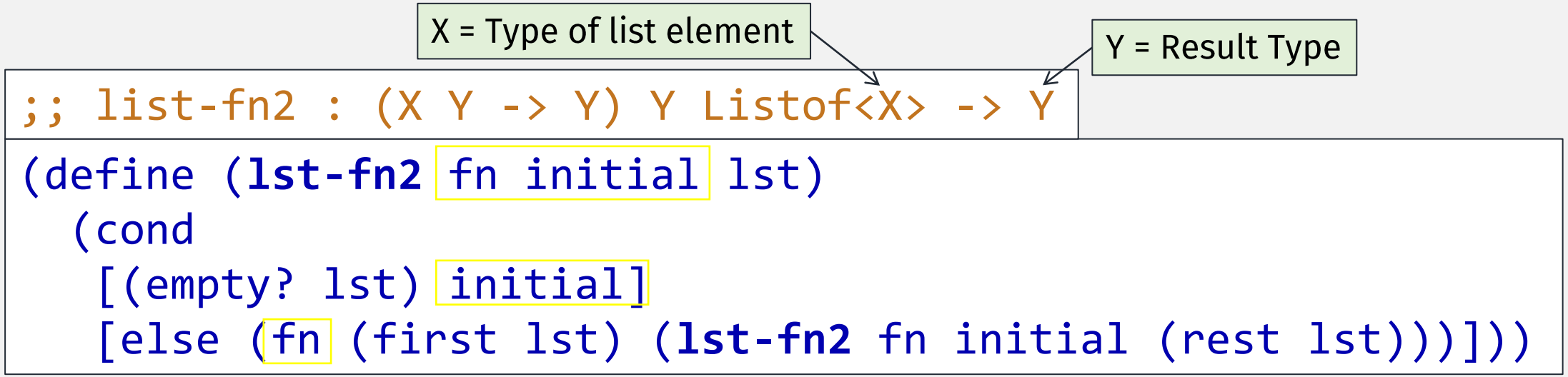
```
;; place-ball : Ball Image -> Image
;; Draws a ball, using its pos as the offset, into the given image
(define (place-ball b scene)
  (place-image BALLIMG (ball-x b) (ball-y b) scene))
```

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




```
;; 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

`foldr` (start at right): order matters?

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

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

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

```
(1 - (2 - 3)) = (1 - 2) - 3 ???
```

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

Challenge:

- Change `foldr` to `foldl`
- (so that the function is applied to the 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))) ....]))
```

Next time: Other common list functions

- Filter
- Find
- Reverse
- append

Check-In Quiz 10/11

on gradescope

(due 1 minute before midnight)