UMass Boston Computer Science

CS450 High Level Languages (section 2)

Recursive Data Definitions

(part 2)

Wednesday, October 2, 2024

Logistics

- HW 4 out
  - due: Mon 10/7 12pm (noon) EST

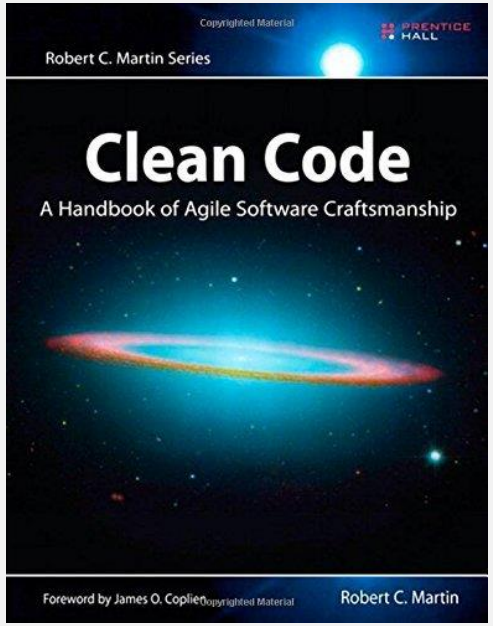
HW Advice

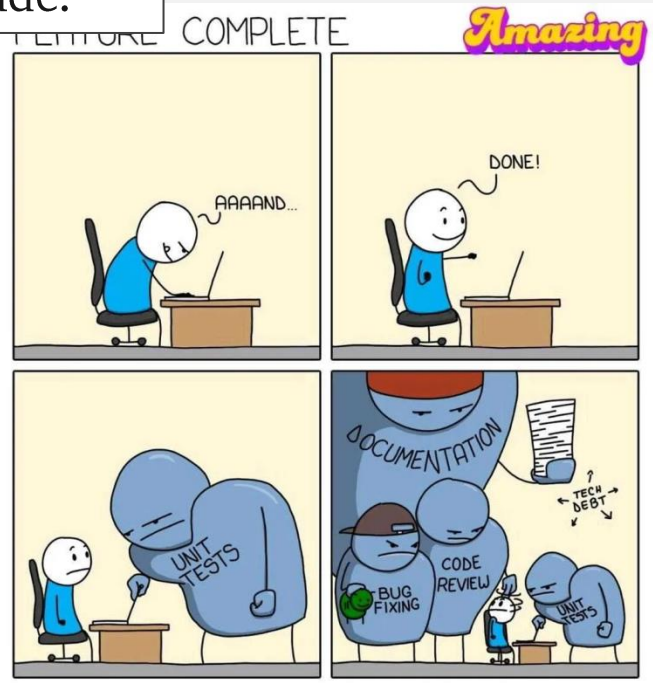
"Perhaps you thought that "**getting it working**" was the first order of business for a professional developer.

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

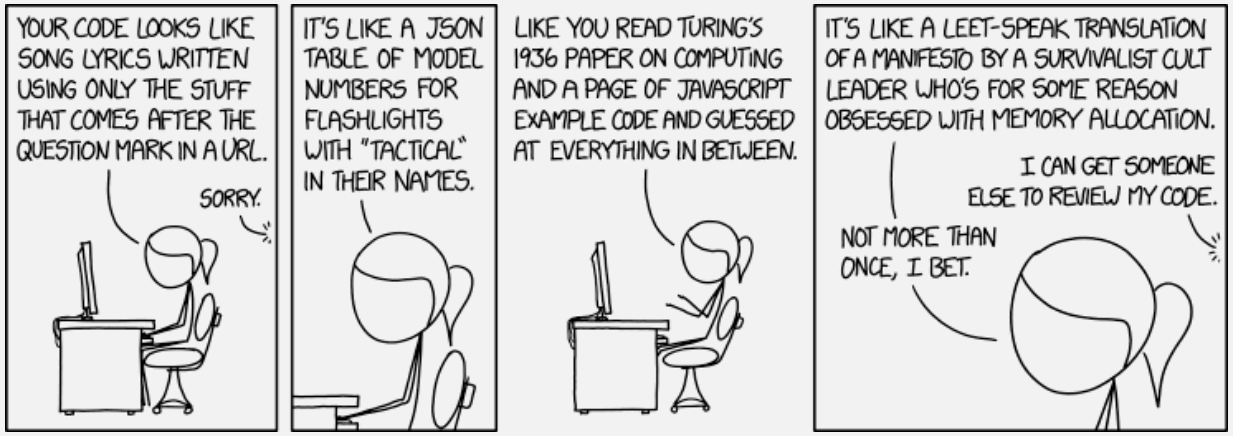
The functionality that you create today has a good chance of changing in the next release, but the **readability of your code** will have a profound effect on all the changes that will ever be made."

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





HW Advice



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

Many submissions ignored:

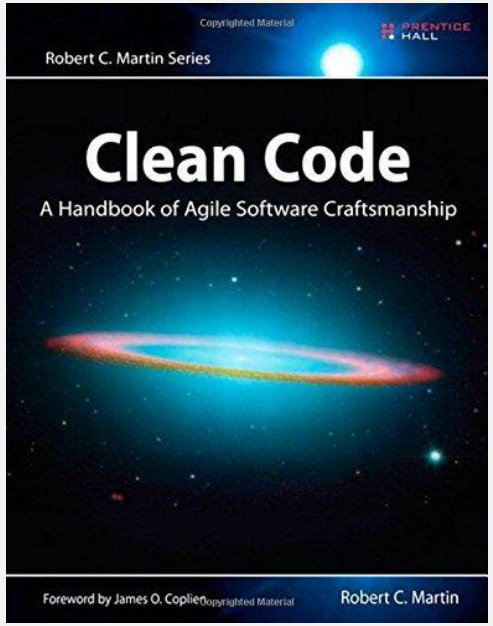
- all other steps of **programming design recipe**
- style guide
- Other instructions in hw

This hw will be graded accordingly:

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

Total: 35 points

HW Advice



> “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

Previously

Predicates for Compound Data

```
;; A Ball is one of:
(struct ball [x y xvel yvel] #:transparent)
;; x : XCoord – ball center horiz coord in animation
;; y : Ycoord – ball center vert coord in animation
;; xvel : Velocity – ball horiz pixels/tick velocity
;; yvel : Velocity – ball vert pixels/tick velocity
```

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

predicate?

`struct` already defines `ball?`, what about **fields?**

```
(define (Ball? arg)
  (and (ball? arg)
       (XCoord? (ball-x arg))
       (YCoord? (ball-y arg))
       (Velocity? (ball-xvel arg))
       (Velocity? (ball-yvel arg))))
```

???

This "deep" predicate checks too much …

… because it's the **job of** "coordinate" and "velocity" **processing functions** to **check** those kinds of data

Note:
Checked constructor ok

```
(define/contract (mk-ball x y xvel yvel)
  (-> XCoord? YCoord? Velocity? Velocity? ball?)
  (ball x y xvel yvel))
```

```
;; 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

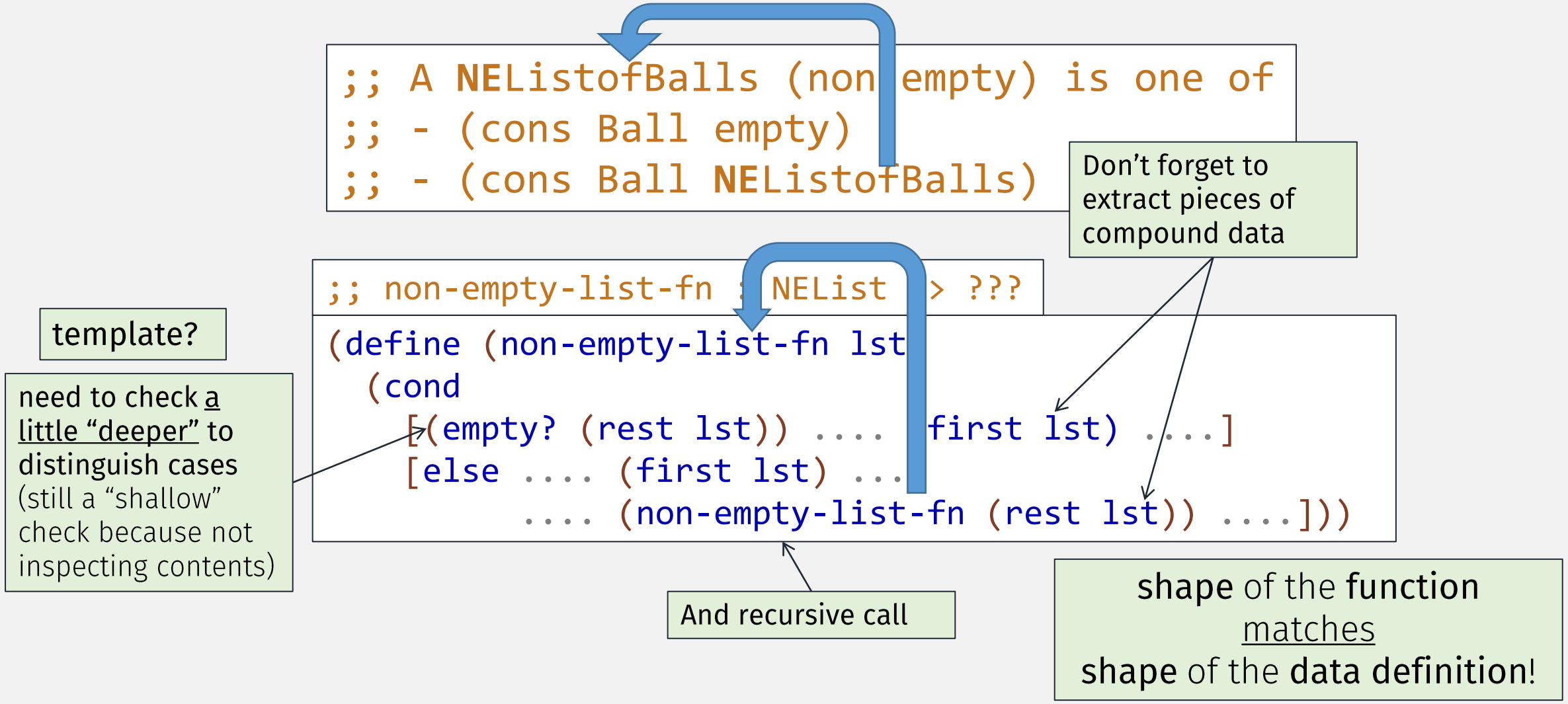
```
;; A NEListofBalls (non-empty) is one of:
;; - (cons Ball empty)
;; - (cons Ball NEListofBalls)
```

predicate?

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


                                  )
```

Just cons? !
(shallow check)

Non-empty lists - template



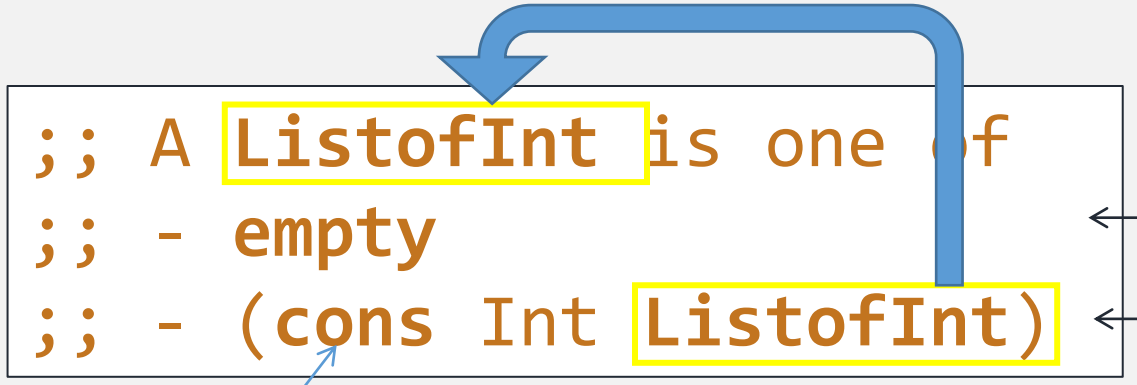
Next: Famous List Functions

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

"BIG" data processing

Previously

Racket List Data Definition Example



Empty (**base**) case

Non-empty (**recursive**) case

cons = “node”

Recursive!
(using a definition to define itself)

TEMPLATE??

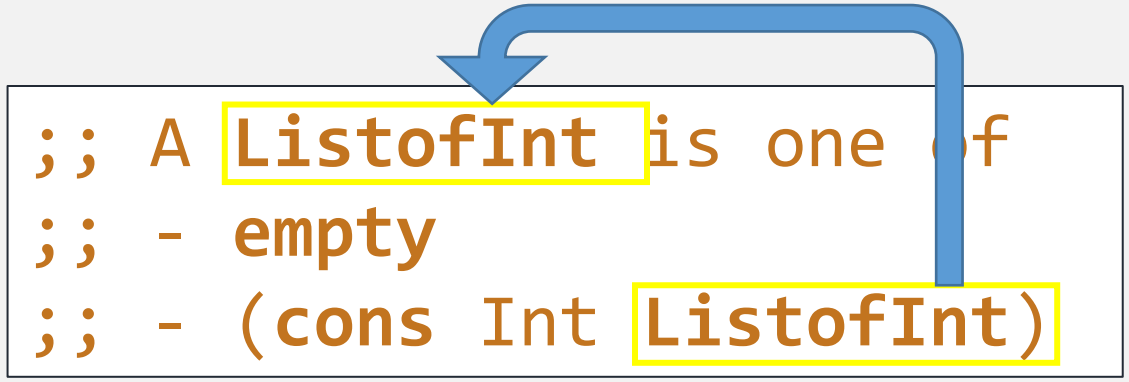
(how can we **use a list of ints to define a list of ints**?!?)

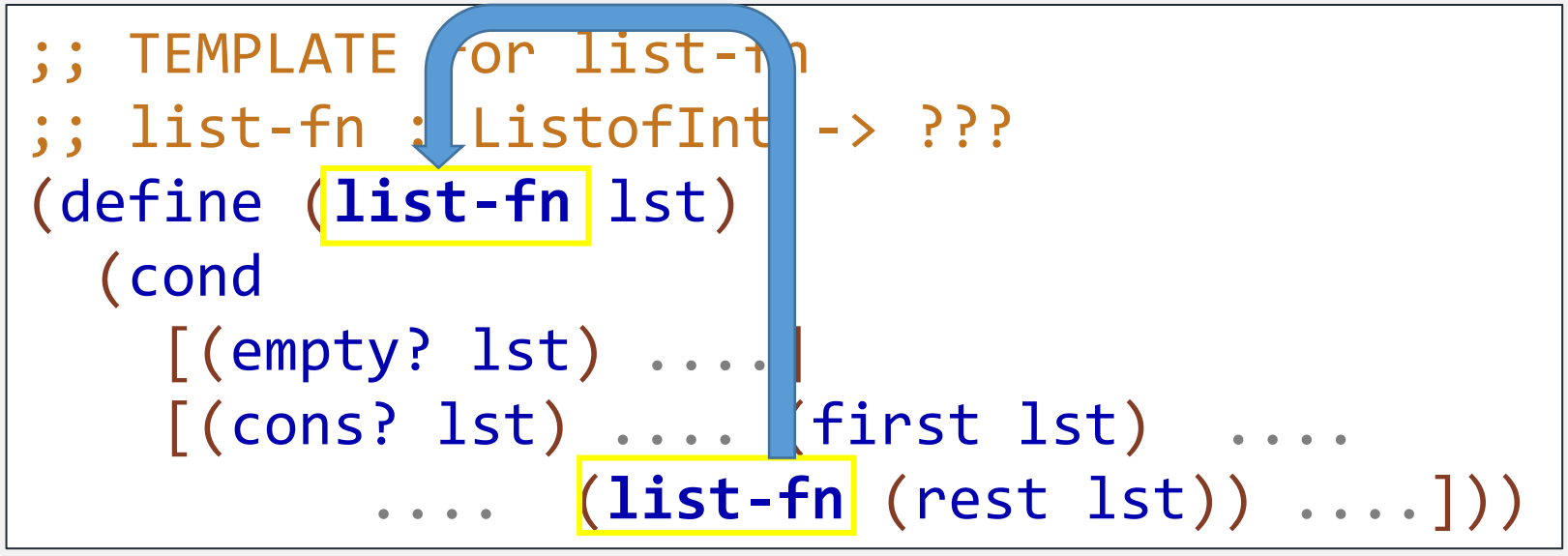
Recursion is a valid concept (from math), but **only if there is <u>both</u>**

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

Previously

Racket Recursive List Fn Template





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)  ....
                 ....  (inc-lst (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)  ....
                 ....  (inc-lst (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 ....  (add1 (first lst)) ....
          ....  (inc-lst (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))
                (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)) ....]))
```

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)))]))
```

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

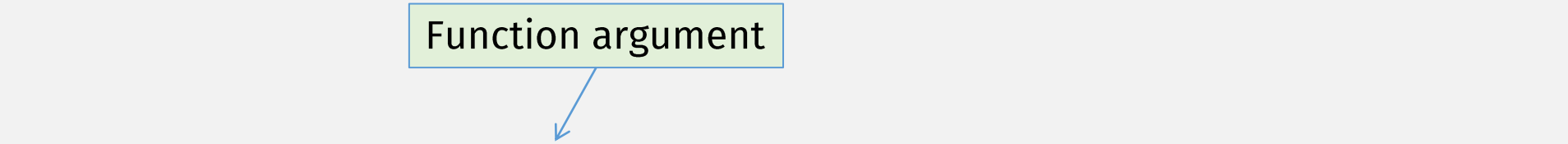
Abstraction: Common List Function #1

```
;; 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




```
;; 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 **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>`

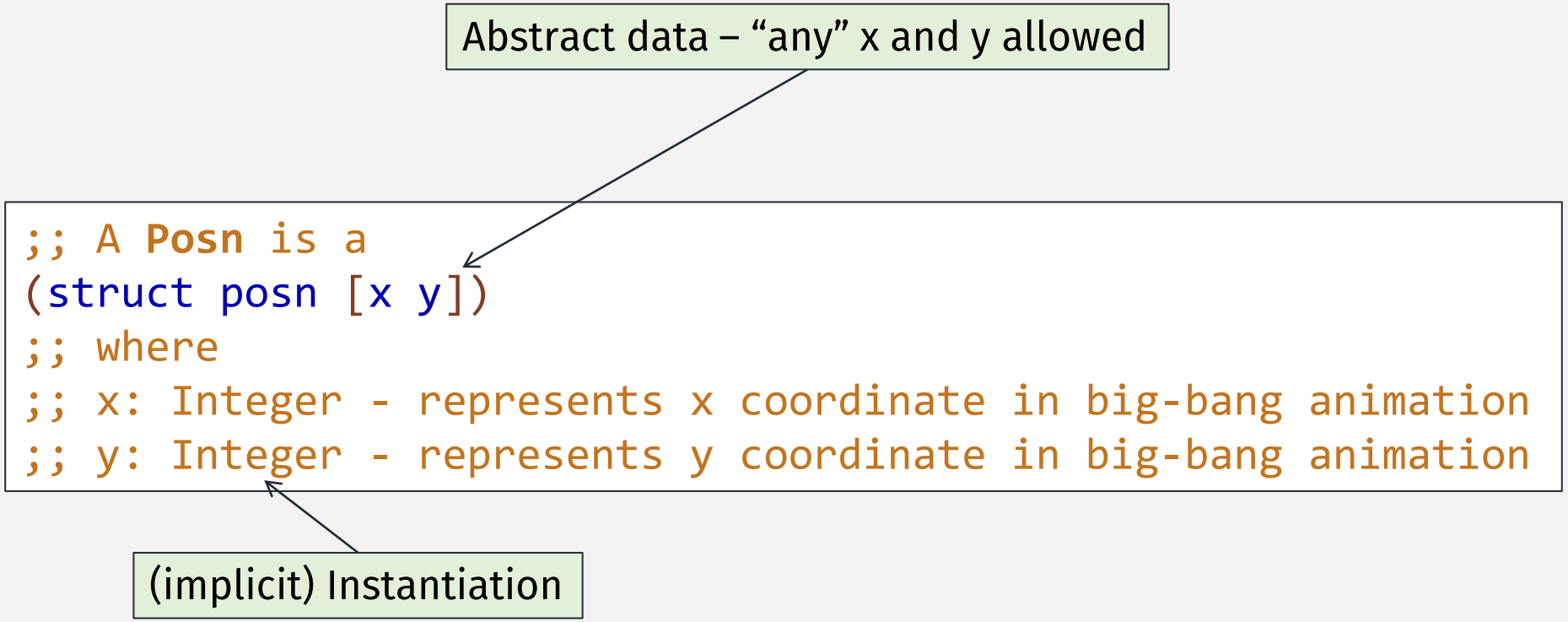
(**concrete** = opposite of **abstract**)

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) 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 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”

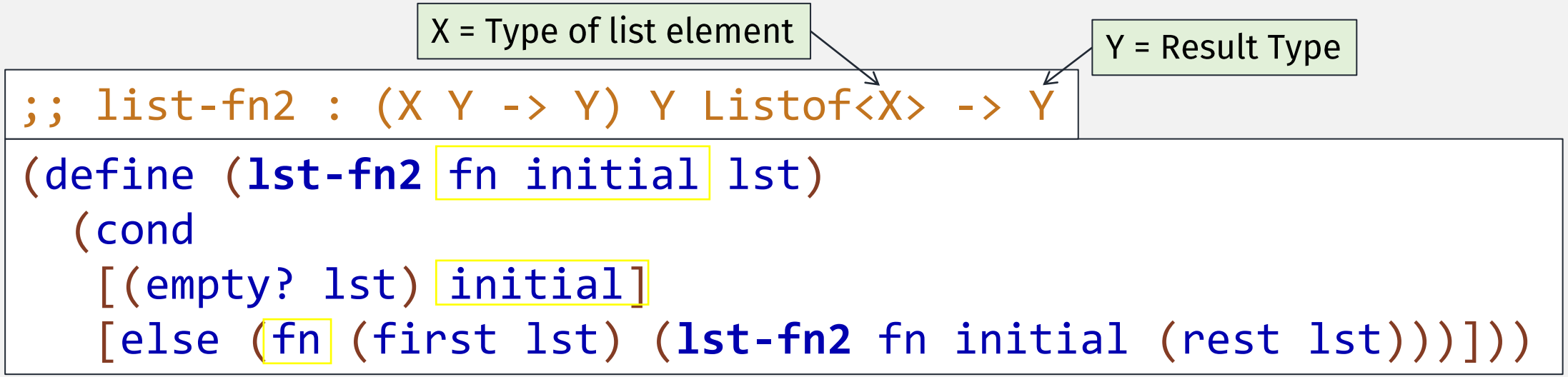
```
;; place-ball : Ball Image -> Image
;; Draws a ball, using its pos as the offset, into the given image
(define (place-ball b scene)
  (place-image BALLIMG (ball-x b) (ball-y b) scene))
```

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))
```

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

Is it ok to always 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)
```

```
(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)))]))
```

```
(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

In-class exercise: more hw4

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

- Work on other **`Notes?`** List functions
  - If you follow the template, this is super easy
  - (very similar to what you just saw!)

Submitting

1. File: `in-class-10-02-<Lastname>-<Firstname>.rkt`
2. Join the in-class team: `cs450f24/teams/in-class`
3. Commit to repo: `cs450f24/in-class-10-02`
   - (May need to **`merge/pull + rebase`** if someone **`push`**es before you)