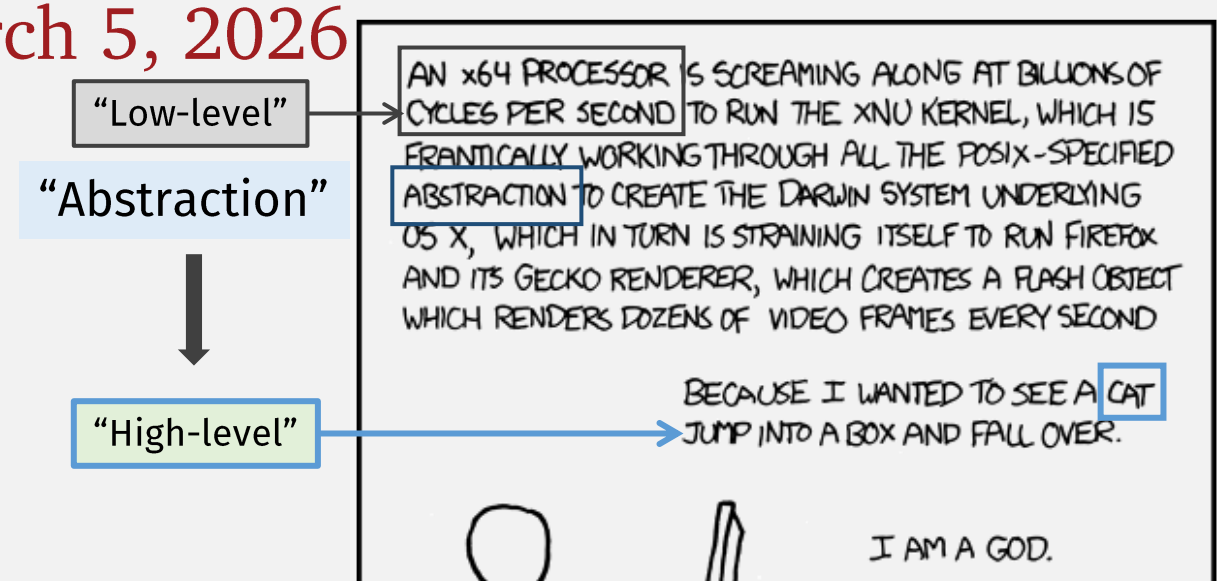
UMass Boston Computer Science

CS450 High Level Languages

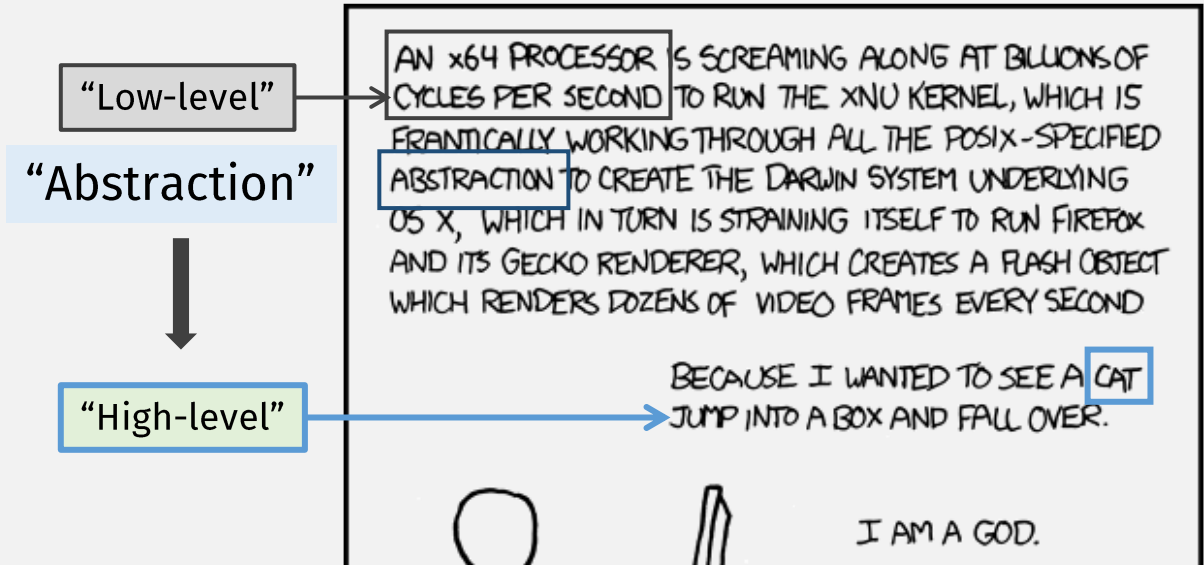
List Functions, Abstraction

Thursday, March 5, 2026



Logistics

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



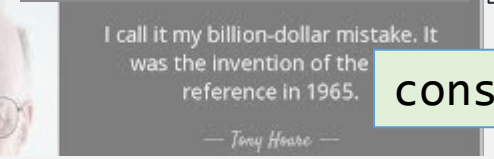
Previously

CS450 List Data Definition Example

(compare to C)

terrible Data Definition
☹ (no base case)

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




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

"smaller" than full list

Empty (**base**) case

Non-empty (**recursive**) case

cons = "node" constructor

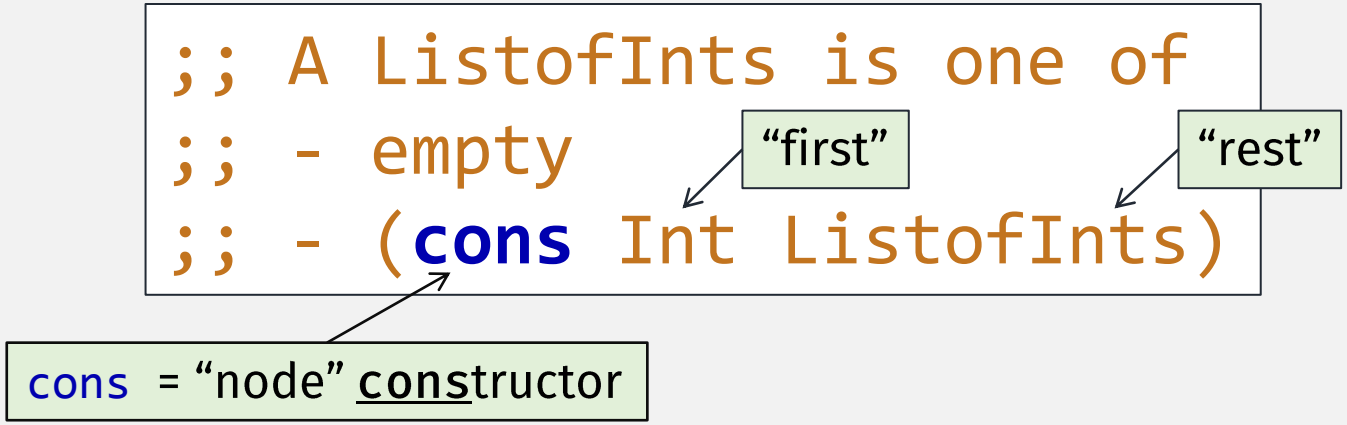
Recursive!
(using a definition to define itself)

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

(shallow) **predicate?**
`list?` (is one possibility)

Recursion is **valid** (from math), but only if there is:
- A **base** case
- A **recursive case** (that is "smaller")

List Constructor and Accessors




```
(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
```

List Data Definition Template

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

```
TEMPLATE??
```

(what kind of data definition is this?)

List Data Definition Template

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

Empty (**base**) case

Non-empty (**recursive**) case

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

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

List Data Definition Template

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

"first" "rest"

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

and

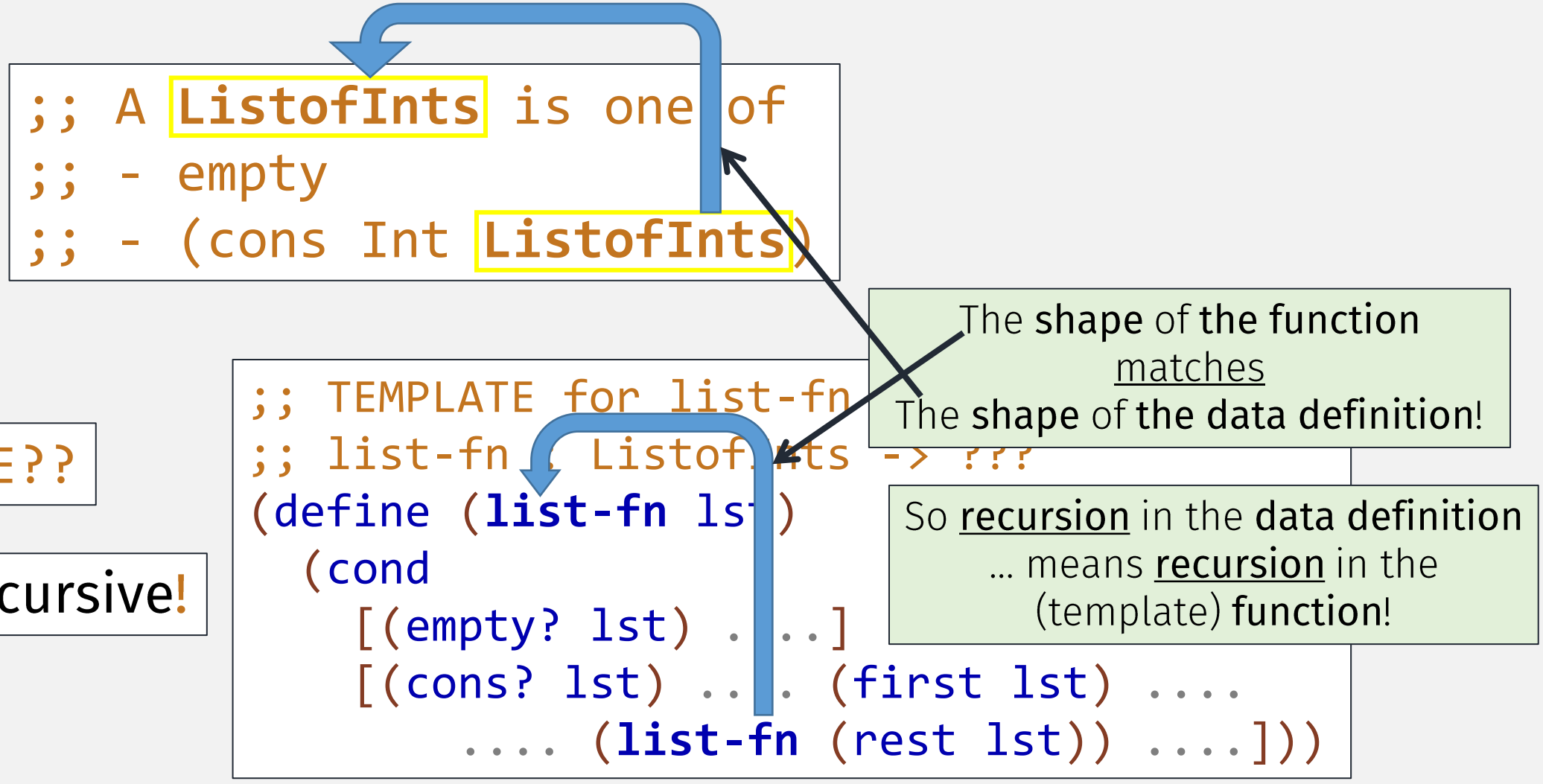
compound data, so **template** has "getters"

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

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

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

List Data Definition Template is Recursive!



In-class Tasks

Follow the design recipe!

Start with TEMPLATE!

Write the following functions:

```
;; smaller-than: ListofInt Int -> ListofInt
;; Computes a list containing elements of given list
;; that are less than the given int
```

```
(check-equal?
  (smaller-than (list 1 3 4 5 9) 4)
                (list 1 3))
```

```
;; larger-than: ListofInt Int -> ListofInt
;; Computes a list containing elements of given list
;; that are greater than the given int
```

```
(check-equal?
  (greater-than (list 1 3 4 5 9) 4)
                (list 5 9))
```

```
;; quicksort: ListofInt -> ListofInt
;; sorts a given list (with no dups) in ascending order
(define (quicksort lst)
  (define pivot (random lst))
  (append (quicksort (smaller-than lst pivot)) pivot (quicksort (greater-than lst pivot))))
```

Start with TEMPLATE!

```
;; smaller-than: ListofInt Int -> ListofInt
;; Computes a list containing elements of given list
;; that are less than the given int
```

```
(define (smaller-than lst x)
  (cond
   [(empty? lst) ...]
   [else ...    (first lst)    ...

         ... (smaller-than (rest lst) x)) ...]))
```

Recursive call

Don't forget extra arg

```
;; smaller-than: ListofInt Int -> ListofInt
;; Computes a list containing elements of given list
;; that are less than the given int
```

```
(define (smaller-than lst x)
  (cond
   [(empty? lst) empty]
   [else ...     (first lst)    ...

         ... (smaller-than (rest lst) x)) ...]))
```

What type of data?

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

```
;; smaller-than: ListofInt Int -> ListofInt
;; Computes a list containing elements of given list
;; that are less than the given int
```

```
(define (smaller-than lst x)
  (cond
   [(empty? lst) empty]
   [else ... (f (first lst))   ...

         ... (smaller-than (rest lst) x)) ...]))
```

"int" fn

```
;; smaller-than: ListofInt Int -> ListofInt
;; Computes a list containing elements of given list
;; that are less than the given int
```

```
(define (smaller-than lst x)
  (cond
   [(empty? lst) empty]
   [else ... (f (first lst) x)   ...

         ... (smaller-than (rest lst) x)) ...]))
```

"int" fn will need **x**

```
;; smaller-than: ListofInt Int -> ListofInt
;; Computes a list containing elements of given list
;; that are less than the given int
(define (smaller-than lst x)
  (cond
   [(empty? lst) empty]
   [else ... (< (first lst) x)   ...

         ... (smaller-than (rest lst) x)) ...]))
```

Now we need to **combine** these **pieces** using "**arithmetic**"

In this case:
- **Bool** …
- **ListofInt** …

(As usual), look at **types** for a hint

Interlude

Another Valid "**if**" Use-case

Allowed in HW5!

(only if it's clear that **fn** deals with boolean computation!)

Function name and **description** clearly indicate a **boolean operation** (comparison)

```
;; smaller-than: ListofInt Int -> ListofInt
;; Computes a list containing elements of given list
;; that are less than the given int
(define (smaller-than lst x)
  (cond
   [(empty? lst) empty]
   [else ... (< (first lst) x)   ...

         ... (smaller-than (rest lst) x)) ...]))
```

Now we need to **combine** these **pieces** using "**arithmetic**"

In this case:
- **Bool** …
- **ListofInt** …

boolean arithmetic fns …
and, **or**, **not**, **if**

Rule of thumb:
- one **if** allowed per function …

```
;; smaller-than: ListofInt Int -> ListofInt
;; Computes a list containing elements of given list
;; that are less than the given int
```

```
(define (smaller-than lst x)
  (cond
   [(empty? lst) empty]
   [else (if (< (first lst) x)
             ...
        ... (smaller-than (rest lst) x)) ...]))
```

Type of data?

```
;; smaller-than: ListofInt Int -> ListofInt
;; Computes a list containing elements of given list
;; that are less than the given int
```

```
(define (smaller-than lst x)
  (cond
   [(empty? lst) empty]
   [else (if (< (first lst) x)
             ...
             (smaller-than (rest lst) x))]))
```

```
;; smaller-than: ListofInt Int -> ListofInt
;; Computes a list containing elements of given list
;; that are less than the given int
(define (smaller-than lst x)
  (cond
   [(empty? lst) empty]
   [else (if (< (first lst) x)
             (cons (first lst) (smaller-than (rest lst) x))
             (smaller-than (rest lst) x))]))
```

(Repetition here is ok, because it will only get run once)

Don't prematurely optimize!

Solutions to in-class Tasks

Functions are mostly the same!

```
(define (smaller-than lst x)
  (cond
   [(empty? lst) empty]
   [else (if (< (first lst) x)
             (cons (first lst) (smaller-than (rest lst) x))
             (smaller-than (rest lst) x))]))
```

Is this a "common" list operation?

Hold that thought! …

```
(define (larger-than lst x)
  (cond
   [(empty? lst) empty]
   [else (if (> (first lst) x)
             (cons (first lst) (larger-than (rest lst) x))
             (larger-than (rest lst) x))]))
```

List Function Template

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

Need “int” fn

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 to combine:
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** 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”

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 to combine:
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)))]))
```

Not allowed in HW5!

Comparison

Functions are mostly the same!

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

Is this a "common" list operation?

Not allowed in HW5!

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

Input / Output are same type?

```
;; 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>)
```

parameter

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

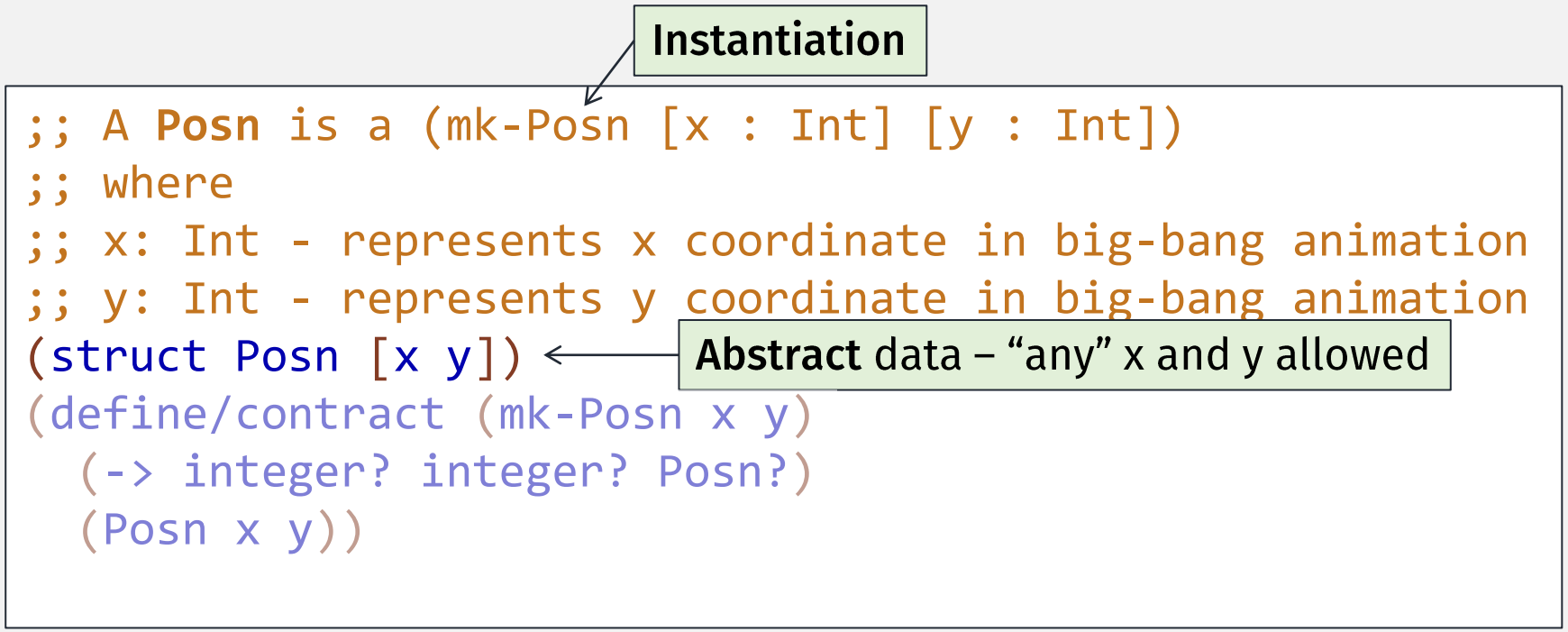
Generic / Abstract data – “any” x and y allowed

Instantiation

```
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();
```

(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 HW5!

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)))]))
```

```
(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()

```
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: ???

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

```
;; 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))
```

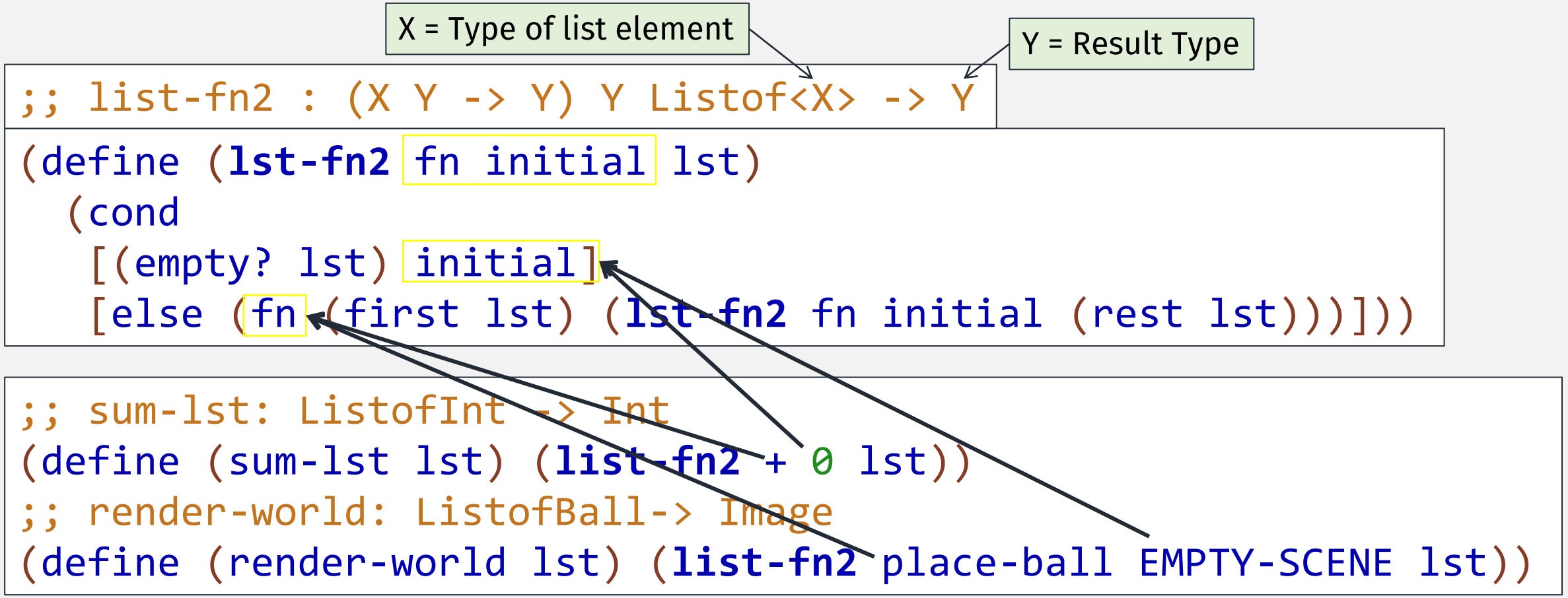
Create one function per “task”

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



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 HW5!

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:

(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 .... (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

Only way to **preserve "Y" type** is to "switch" **fn call** and **recursive call**

```
;; 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

Only way to **preserve "Y" type** is to "switch" **fn call** and **recursive call**

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

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

"initial"???

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"

Not allowed in HW5

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()

```
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)
```

Previously

Solutions to in-class Tasks

Functions are mostly the same!

```
(define (smaller-than lst x)
  (cond
   [(empty? lst) empty]
   [else (if (< (first lst) x)
             (cons (first lst) (smaller-than (rest lst) x))
             (smaller-than (rest lst) x))]))
```

Is this a "common" list operation?

Hold that thought! …

```
(define (larger-than lst x)
  (cond
   [(empty? lst) empty]
   [else (if (> (first lst) x)
             (cons (first lst) (larger-than (rest lst) x))
             (larger-than (rest lst) x))]))
```

Common list function #3?

Is this a “good” abstraction?

```
;; lst-fn3: (Int Int -> Boolean) ListofInt Int -> ListofInt
;; Computes a list containing elements of given list
;; that are ??? when compared to another given int
```

```
(define (lst-fn3 cmp? lst other-int)
  (cond
   [(empty? lst) empty]
   [else (if (cmp? (first lst) other-int )
             (cons (first lst) (lst-fn3 cmp? (rest lst) other-int))
             (lst-fn3 cmp? (rest lst) other-int))]))
```

Awkward to explain …

Common list function #3?

Is this a “good” abstraction?

What are possible use cases? i.e., Examples!!

Should be more than just the two examples we are abstracting!

```
;; lst-fn3: (Int Int -> Boolean) ListofInt Int -> ListofInt
;; Computes a list containing elements of given list
;; that are ??? when compared to another given int
```

```
(define (lst-fn3 cmp? lst other-int)
  (cond
   [(empty? lst) empty]
   [else (if (cmp? (first lst) other-int )
             (cons (first lst) (lst-fn3 cmp? (rest lst) other-int))
             (lst-fn3 cmp? (rest lst) other-int))]))
```

Awkward to explain …

More tasks

Write the following functions:

```
(check-equal?
  (shorter-than (list “a” “bc “abc”) 2)
                (list “a”))
```

```
;; shorter-than: ListofString Int -> ListofString
;; Computes a list containing elements of given list
;; that have length less than the given int
```

```
(check-equal?
  (shorter-than-str (list “a” “bc” “abc”) “xy”)
                    (list “a”))
```

```
;; shorter-than-str: ListofString String -> ListofString
;; Computes a list containing elements of given list
;; that have length less than the given string
```

Mor

```
;; lst-fn3: (Int Int -> Boolean) ListofInt Int -> ListofInt
;; Computes a list containing elements of given list
;; that are ??? When compared to another given int
```

Write the following functions:

```
;; shorter-than: ListofString Int -> ListofString
;; Computes a list containing elements of given list
;; that have length less than the given int
```

Could these be implemented with our new abstraction?

Should we be able to?

```
;; shorter-than-str: ListofString String -> ListofString
;; Computes a list containing elements of given list
;; that have length less than the given string
```