

UMass Boston Computer Science  
**CS450 High Level Languages**  
**List Fns2, More Abstraction**

Tuesday, March 4, 2025

map, filter, and reduce  
explained with emoji 🤔

```
map([🐮, 🍌, 🐔, 🌽], cook)  
=> [🍔, 🍟, 🍗, 🍿]
```

```
filter([🍔, 🍟, 🍗, 🍿], isVegetarian)  
=> [🍟, 🍿]
```

```
reduce([🍔, 🍟, 🍗, 🍿], eat)  
=> 🤩
```

# Logistics

- HW 4 in
  - ~~due: Tues 3/4 11am EST~~
- HW 5 out
  - due: Tues 3/11 11am EST

map, filter, and reduce  
explained with emoji 🤔

```
map([🐮, 🍌, 🐔, 🌽], cook)  
=> [🍔, 🍟, 🍗, 🍿]
```

```
filter([🍔, 🍟, 🍗, 🍿], isVegetarian)  
=> [🍟, 🍿]
```

```
reduce([🍔, 🍟, 🍗, 🍿], eat)  
=> 🤮
```

# big-bang stop-when

```
(stop-when last-world?)
```

syntax

```
last-world? : (-> WorldState boolean?)
```

tells DrRacket to call the *last-world?* function at the start of the world program and after any other world-producing callback. If this call produces `#true`, the world program is shut down. Specifically, the clock is stopped; no more tick events, `KeyEvents`, or `MouseEvent`s are forwarded to the respective handlers. The `big-bang` expression returns this last world.

```
(stop-when last-world? last-picture)
```

```
last-world? : (-> WorldState boolean?)
```

```
last-picture : (-> WorldState scene?)
```

game-over? predicate

render-last function

tells DrRacket to call the *last-world?* function at the start of the world program and after any other world-producing callback. If this call produces `#true`, the world program is shut down after displaying the world one last time, this time image rendered with *last-picture*. Specifically, the clock is stopped; no tick events, `KeyEvents`, or `MouseEvent`s are forwarded to the respective handlers. The `big-bang` expression returns this last world.

```
(define (main)
  (big-bang INIT-WORLDSTATE
    [on-tick next-WorldState 1]
    [stop-when game-over? render-last]
    [on-key handle-key]
    [to-draw WorldState->Image]))
```

*Last  
Time*

# List (Recursive) Data Definition 1

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

Last  
Time

# List (Recursive) Data Definition 1: Fn Template

Recursive call matches  
recursion in data definition

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

cond clause for each  
itemization item

Extract pieces of  
compound data

Last  
Time

# Recursive List Fn Example 1: `inc-list`

## Function design recipe:

1. Name
2. Signature
3. Description
4. Examples
5. Template
- ...

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

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

Last  
Time

# Recursive List Fn Example 1: **inc-list**

```
;; inc-list : ListofInt -> ListofInt  
;; increments each list element by 1
```

```
(define (inc-lst lst)  
  (cond
```

Empty input produces empty output  
(look at signature for help if needed)

```
    [(empty? lst) empty]
```

```
    [(cons? lst) .... (first lst) ....  
                      .... (inc-lst (rest lst)) ....]))
```

Last  
Time

# Recursive List Fn Example 1: **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)) .... ])))
```

Call another function to process  
(first) (Int) list element



Last  
Time

# Recursive List Fn Example 1: **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)))]))
```

Figure out how to “combine” with  
recursive call result  
(look at signature for help if needed)

*Last  
Time*

## List (Recursive) Data Definition 2

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

Last  
Time

# List (Recursive) Data Definition 2: Fn Template

Recursive call matches  
recursion in data definition?

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

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

cond clause for each  
itemization item?

Extract pieces of  
compound data?

# Recursive List Fn Example 2: **next-world**

## Function design recipe:

1. Name
2. Signature
3. Description
4. Examples
5. Template
- ...

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

Last  
Time

## Recursive List Fn Example 2: **next-world**

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

Empty input produces empty output  
(look at signature for help if needed)

Last  
Time

# Recursive List Fn Example 2: **next-world**

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

Call another function to process (first) list element?

Ball

Last  
Time

## Recursive List Fn Example 2: **next-world**

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

Call another (Ball) function to  
process (first) list element

Last  
Time

# Recursive List Fn Example 2: **next-world**

```
;; next-world: ListofBall -> ListofBall  
;; Updates position each ball by one tick
```

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

Figure out how to “combine” with recursive call result  
(look at signature for help if needed)



# Comparison 1

Differences?

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

*Last  
Time*

# Abstraction: Common List Function #1

Make the difference a  
parameter of a  
(function) abstraction

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

# Abstraction Recipe

1. Find similar patterns in a program
  - Minimum: 2
  - Ideally: 3+
2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
  - E.g., a function(al) abstraction

# 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)))]))
```

# Abstraction of Data Definitions

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

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

# Abstraction Recipe

1. Find similar patterns in a program
  - Minimum: 2
  - Ideally: 3+
- ➔ 2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
  - E.g., a function(al) abstraction

# Abstraction of Data Definitions

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

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

# Abstraction Recipe

1. Find similar patterns in a program
  - Minimum: 2
  - Ideally: 3+
2. Identify differences and make them parameters
- ➔ 3. Create a reusable abstraction with the discovered parameters
  - E.g., a function(al) abstraction
  - ➔ • E.g., a data abstraction



# Abstraction of Data Definitions

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

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



parameter

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

# Abstraction: Common List Function #1

NOTE: textbook writes it like this  
(both are ok, just follow data definition)

```
;; lst-fn1: [X -> Y] [Listof X] -> [Listof Y]  
;; Applies the given fn to each element of given lst
```

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

# Abstraction Recipe

1. Find similar patterns in a program
  - Minimum: 2
  - Ideally: 3+
2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
  - E.g., a function(al) abstraction
  - E.g., a data abstraction
- ➡ 4. Use the abstraction, by giving concrete arguments for parameters

# Abstraction: 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))
```

Q: Do these functions follow the design recipe (template)?

A: They do. Because “arithmetic” is always allowed.

```
(define (inc-1st lst) (lst-fn1 add1 lst))  
(define (next-world lst) (lst-fn1 next-ball lst))
```

# 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))
```

# Abstraction Recipe

1. Find similar patterns in a program

- Minimum: 2
- Ideally: 3+

Abstractions should have a “clear and concisely defined task”

2. Identify differences and make them parameters

3. Create a reusable abstraction with the discovered parameters

- E.g., a function(al) abstraction
- E.g., a data abstraction

→ • The **abstraction must** have a short, clear name and “be logical”

4. Use the abstraction by giving concrete “arguments” parameters



# Abstraction Recipe



1. Find similar patterns in a program

- Minimum: 2
- Ideally: 3+

Not all “repeated code” should be abstracted

2. Identify differences and make them parameters

3. Create a reusable **Creating Bad Abstractions is Dangerous**

- E.g., a function(al) abstraction
- E.g., a data abstraction

**Creating Good Abstractions is Hard**

- The abstraction must have a short, clear name and “be logical”

4. Use the abstraction by giving concrete “arguments” parameters

# Abstraction Warning Story

I came to see the following pattern:

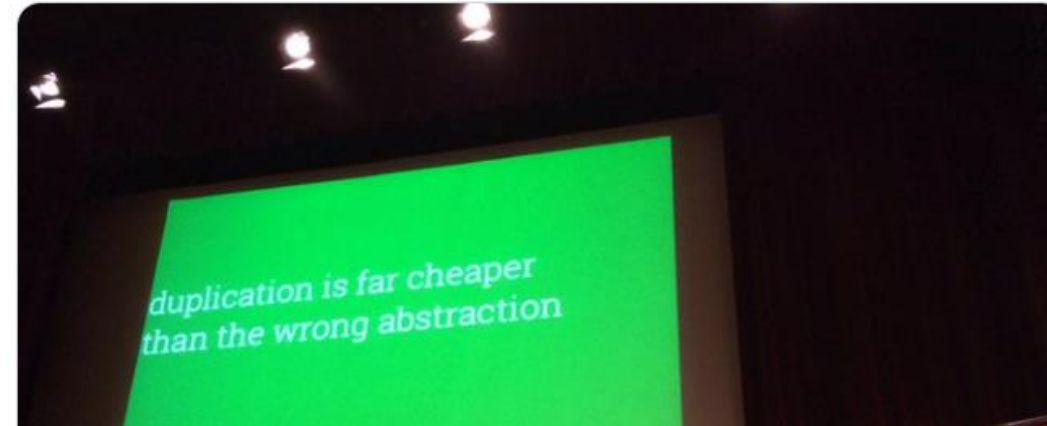
1. Programmer A sees duplication ...
2. Programmer A extracts duplication and gives it a name.  
This *creates a new abstraction*.
3. Programmer A replaces duplication with the new abstraction.  
*Ah, the code is perfect. Programmer A trots happily away.*

4. **Time passes ...**

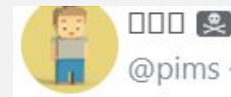


@pims · Follow

This, a million times this! “@BonzoESC: “Duplication is far cheaper than the wrong abstraction” @sandimetz @rbonales ”

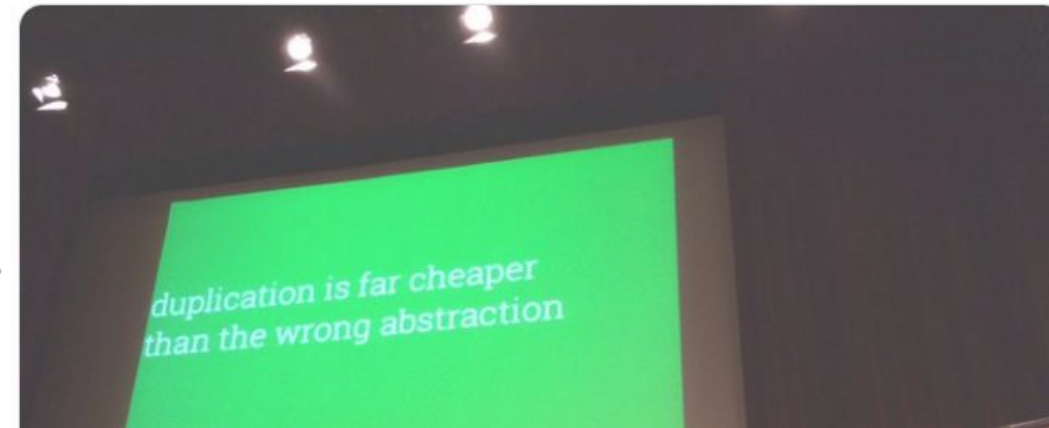


# Abstraction Warning Story



@pims · Follow

This, a million times this! "@BonzoESC: "Duplication is far cheaper than the wrong abstraction" @sandimetz @rbonales "



I came to see the following pattern:

1. Programmer A sees duplication.
2. Programmer A extracts duplication and gives it a name.  
*This creates a new abstraction.*
3. Programmer A replaces the duplication with the new abstraction.  
*Ah, the code is perfect. Programmer A trots happily away.*

4. Time passes ...

5. A new requirement appears ... for which the current abstraction is almost perfect.

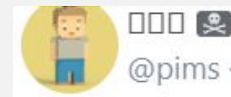
6. Programmer B gets tasked to implement this requirement ...

Programmer B tries to retain the existing abstraction ...

... but it's not perfect ... so *they alter the code to take a parameter,*

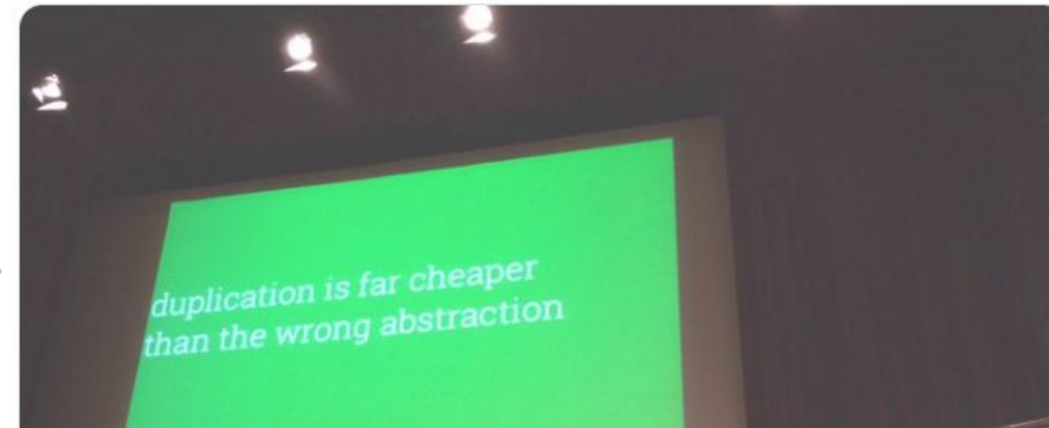
... and then add extra logic that is conditionally based on the value of that parameter.

# Abstraction Warning Story



@pims · Follow

This, a million times this! “@BonzoESC: “Duplication is far cheaper than the wrong abstraction” @sandimetz @rbonales ”



I came to see the following pattern:

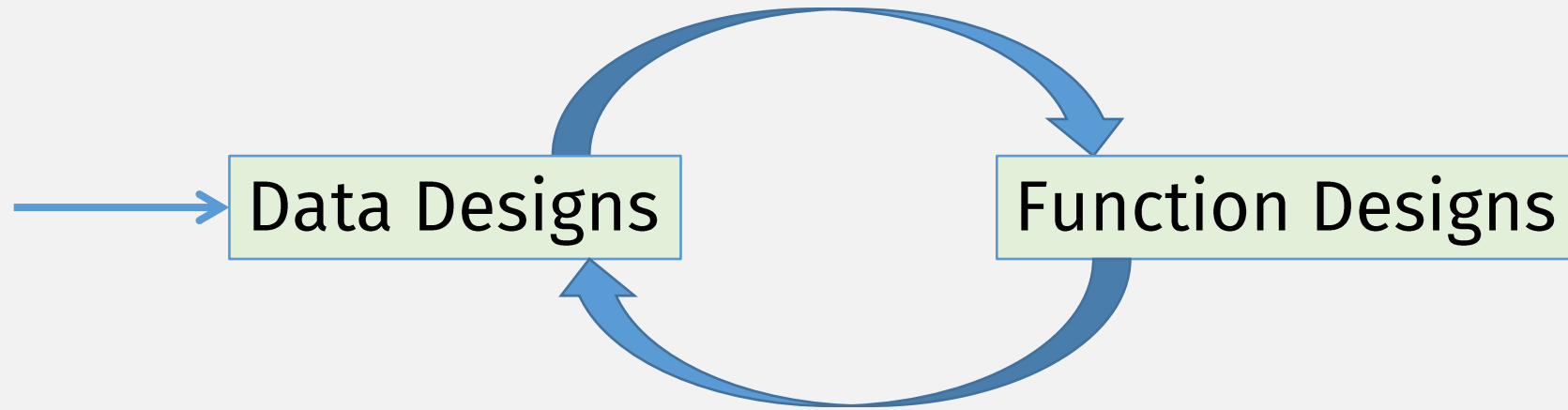
1. Programmer A sees duplication.
2. Programmer A extracts duplication and gives it a name.
3. Programmer A replaces duplication with the new abstraction.  
*Ah, the code is perfect. Programmer A trots happily away.*

**How to avoid?**

**Always be thinking about the data**

4. Time passes ...
5. A new requirement appears ... for which the current abstraction is almost perfect.
6. Programmer B gets tasked to implement this requirement ...  
Programmer B tries to retain the existing abstraction ...  
... but it's not perfect ... so they alter the code to take a parameter,  
... and then add extra logic that is conditionally based on the value of that parameter.
7. Another new requirement arrives ... and a new Programmer X, who adds an additional parameter ... and a new conditional ... Repeat until **code becomes incomprehensible**.
8. You appear in the story about here ... and your life takes a dramatic turn for the worse.

# Program Design Recipe

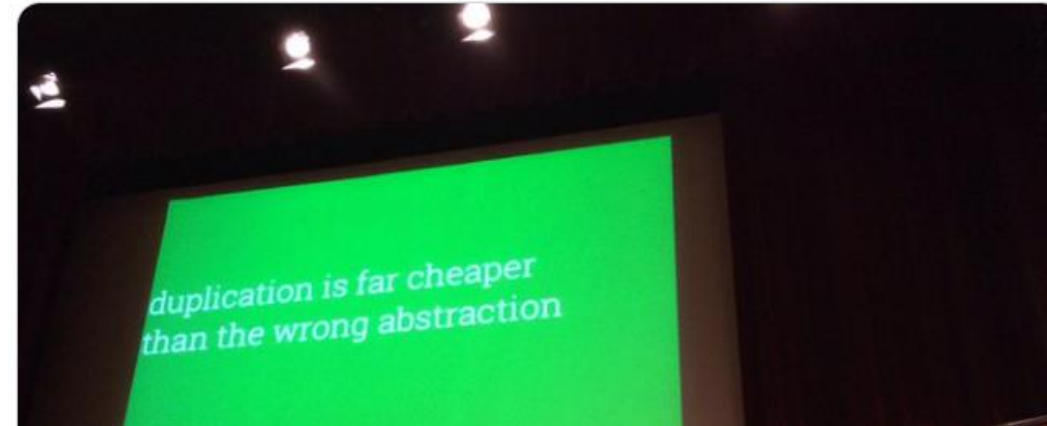


# Abstraction Warning Story



@pims · Follow

This, a million times this! “@BonzoESC: “Duplication is far cheaper than the wrong abstraction” @sandimetz @rbonales ”



**How to avoid?**

**Always be thinking about the data**

**Don't focus only on “getting the code working”**

**These programmers only cared about “getting the code working”**

*Programmer B*

*conditionally*

*new conditional. Loop until code becomes incomprehensible.*

**8. You appear in the story about here, and your life takes a dramatic turn for the worse.**

*Last  
Time*

# Common List Function #2: ???

Last  
Time

## Comparison #2

```
;; sum-1st: ListofInt -> Int
(define (sum-1st lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst)
              (sum-1st (rest lst)))]))
```

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



# Abstraction Recipe

1. Find similar patterns in a program
  - Minimum: 2
  - Ideally: 3+
- ➔ 2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
  - E.g., a function(al) abstraction
  - E.g., a data abstraction
  - The abstraction must have a short, clear name and “be logical”
4. Use the abstraction by giving concrete “arguments” parameters

Last  
Time

## 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)))]))
```

# Abstraction Recipe

1. Find similar patterns in a program
  - Minimum: 2
  - Ideally: 3+
2. Identify differences and make them parameters
- ➔ 3. Create a reusable abstraction with the discovered parameters
  - E.g., a function(al) abstraction
  - E.g., a data abstraction
  - The abstraction must have a short, clear name and “be logical”
4. Use the abstraction by giving concrete “arguments” parameters

# Common List Function #2: **foldr**

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

Also called “reduce”

Because a list of values is  
“reduced” to one value

# Abstraction Recipe

1. Find similar patterns in a program
  - Minimum: 2
  - Ideally: 3+
2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
  - E.g., a function(al) abstraction
  - E.g., a data abstraction
  - The abstraction must have a short, clear name and “be logical”
- ➡ 4. Use the abstraction by giving concrete “arguments” parameters

# Common List Function #2: `foldr`

```
:: 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))))]))
```

```
:: 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))
```

# 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)

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

“result so far”

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

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

# Common list function #3

# Tasks

Follow the design recipe!

Write the following functions:

```
;; smaller-than: ListofInt Int -> ListofInt
;; Returns 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
;; Returns 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))))
```

```
;; smaller-than: ListofInt Int -> ListofInt  
;; Returns 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) ...]))
```

```
;; smaller-than: ListofInt Int -> ListofInt  
;; Returns 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?

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

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

```
;; smaller-than: ListofInt Int -> ListofInt  
;; Returns 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) ...]))
```

What about x?



```
;; smaller-than: ListofInt Int -> ListofInt  
;; Returns 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) ...]))
```



Allowed in HW5!

# Two-Argument Templates

Sometimes ... two fn args are supposed to be processed together!

```
;; smaller-than: ListofInt Int -> ListofInt
;; Returns 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) ...]))
```

The function should combine the templates of the two kinds of data

In this case:

- **List** ...
- ... and **Int -> Bool** arithmetic

Template for **Bool** is ... **if**

Rule of thumb:

- one **if** allowed per function ...

```
;; smaller-than: ListofInt Int -> ListofInt
;; Returns 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) ...)]))
```

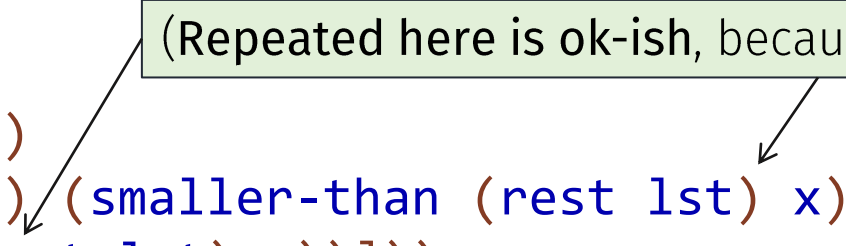
```
;; smaller-than: ListofInt Int -> ListofInt  
;; Returns a list containing elements of given list  
;; that are less than the given int
```

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

```
;; smaller-than: ListofInt Int -> ListofInt  
;; Returns 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))]))
```

(Repeated here is ok-ish, because it will only get run once)



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

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

# Abstraction Recipe

1. Find similar patterns in a program
  - Minimum: 2
  - Ideally: 3+
- ➔ 2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
  - E.g., a function(al) abstraction
  - E.g., a data abstraction
  - The abstraction must have a short, clear name and “be logical”
4. Use the abstraction by giving concrete “arguments” parameters

# Your tasks

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

```
(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  
;; Returns 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)))]))
```

# Abstraction Recipe

1. Find similar patterns in a program
  - Minimum: 2
  - Ideally: 3+
2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
  - E.g., a function(al) abstraction
  - E.g., a data abstraction
- ➡ • The **abstraction must** have a **short, clear name** and “**be logical**”
4. Use the abstraction by giving concrete “arguments” parameters

# Abstraction Recipe

1. Find similar patterns in a program
  - Minimum: 2
  - Ideally: 3+
2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
  - E.g., a function(al) abstraction
  - E.g., a data abstraction
  - The abstraction must have a short, clear name and “be logical”
- ➡ 4. Use the abstraction by giving concrete “arguments” parameters

# Common list function #3?

Is this a “good” abstraction?

What are possible use cases?

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

```
;; lst-fn3: (Int Int -> Boolean) ListofInt Int -> ListofInt  
;; Returns 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))])])
```

# More tasks

Write the following functions:

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

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

More

```
;; 1st-fn3: (Int Int -> Boolean) ListofInt Int -> ListofInt  
;; Returns a list containing elements of given list  
;; that are ??? When compared to another given int
```

Write the following functions:

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

# Abstraction Recipe

1. Find similar patterns in a program
  - Minimum: 2
  - Ideally: 3+
2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
  - E.g., a function(al) abstraction
  - E.g., a data abstraction
  - The abstraction must have a short, clear name and “be logical”
- ➔ 4. Use the abstraction by giving concrete “arguments” parameters

# Abstraction Recipe

Remember:  
The Design Recipe (like good software development) is **iterative!**

1. Find similar patterns in a program
    - Minimum: 2
    - Ideally: 3+
  2. Identify differences and make them parameters
  3. Create a reusable abstraction with the discovered parameters
    - E.g., a function(al) abstraction
    - E.g., a data abstraction
    - The abstraction must have a short, clear name and “be logical”
  4. Use the abstraction by giving concrete “arguments” parameters
- 



# Common list function #3?

Is this a “good” abstraction?

```
;; lst-fn3: (Int Int -> Boolean) ListofInt Int -> ListofInt  
;; Returns 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))]))
```

# A Better common list function #3?

```
;; lst-fn3: (X -> Boolean) Listof<X> Int -> Listof<X>  
;; Returns a list containing elements of given list  
;; for which the given predicate returns true
```

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

# Common list function #3: **filter**

```
;; smaller-than: Listof<Int> Int -> Listof<Int>  
;; Returns a list containing elements of given list less than the given int
```

```
(define (smaller-than lst thresh)  
  (filter (lambda (x) (< x thresh)) lst)
```

↑  
lambda creates an anonymous “inline” function (expression)

```
;; filter: (X -> Boolean) Listof<X> -> Listof<X>  
;; Returns a list containing elements of given list  
;; for which the given predicate returns true
```

```
(define (filter pred? lst)  
  (cond  
    [(empty? lst) empty]  
    [else (if (pred? (first lst))  
              (cons (first lst) (filter pred? (rest lst)))  
              (filter pred? (rest lst))))])
```

# Common list function #3: `filter`

```
;; smaller-than: Listof<Int> Int -> Listof<Int>  
;; Returns a list containing elements of given list less than the given int
```

```
(define (smaller-than lst thresh)  
  (filter (lambda (x) (< x thresh)) lst)
```

↑  
lambda creates an anonymous “inline” function (expression)

```
;; filter: (X -> Boolean) Listof<X> -> Listof<X>  
;; Returns a list containing elements of given list  
;; for which the given predicate returns true
```

```
(define (filter pred? lst)  
  (cond  
    [(empty? lst) empty]  
    [else (if (pred? (first lst))  
              (cons (first lst) (filter pred? (rest lst)))  
              (filter pred? (rest lst)))])
```

## lambda rules:

- Can skip the **design recipe** steps, BUT
- **name**, **description**, and **signature** must be “obvious”
- **code** is arithmetic only
- otherwise, create standalone function define

# filter in other high-level languages

## JavaScript Demo: Array.filter()

```
1 const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];
2
3 const result = words.filter((word) => word.length > 6);
4
5 console.log(result);
6 // Expected output: Array ["exuberant", "destruction", "present"]
7
```

# Your Remaining tasks

## Implement with filter

```
:: smaller-than: ListofInt Int -> ListofInt  
:: Returns list containing elements of given list less than the given int
```

```
:: larger-than: ListofInt Int -> ListofInt  
:: Returns list containing elements of given list greater than the given int
```

```
:: shorter-than: ListofString Int -> ListofString  
:: Returns list containing elements of given list with length less than given int
```

```
:: shorter-than-str: ListofString String -> ListofString  
:: Returns list containing elements of given list with length less than given string
```

