

Reviewing Templates

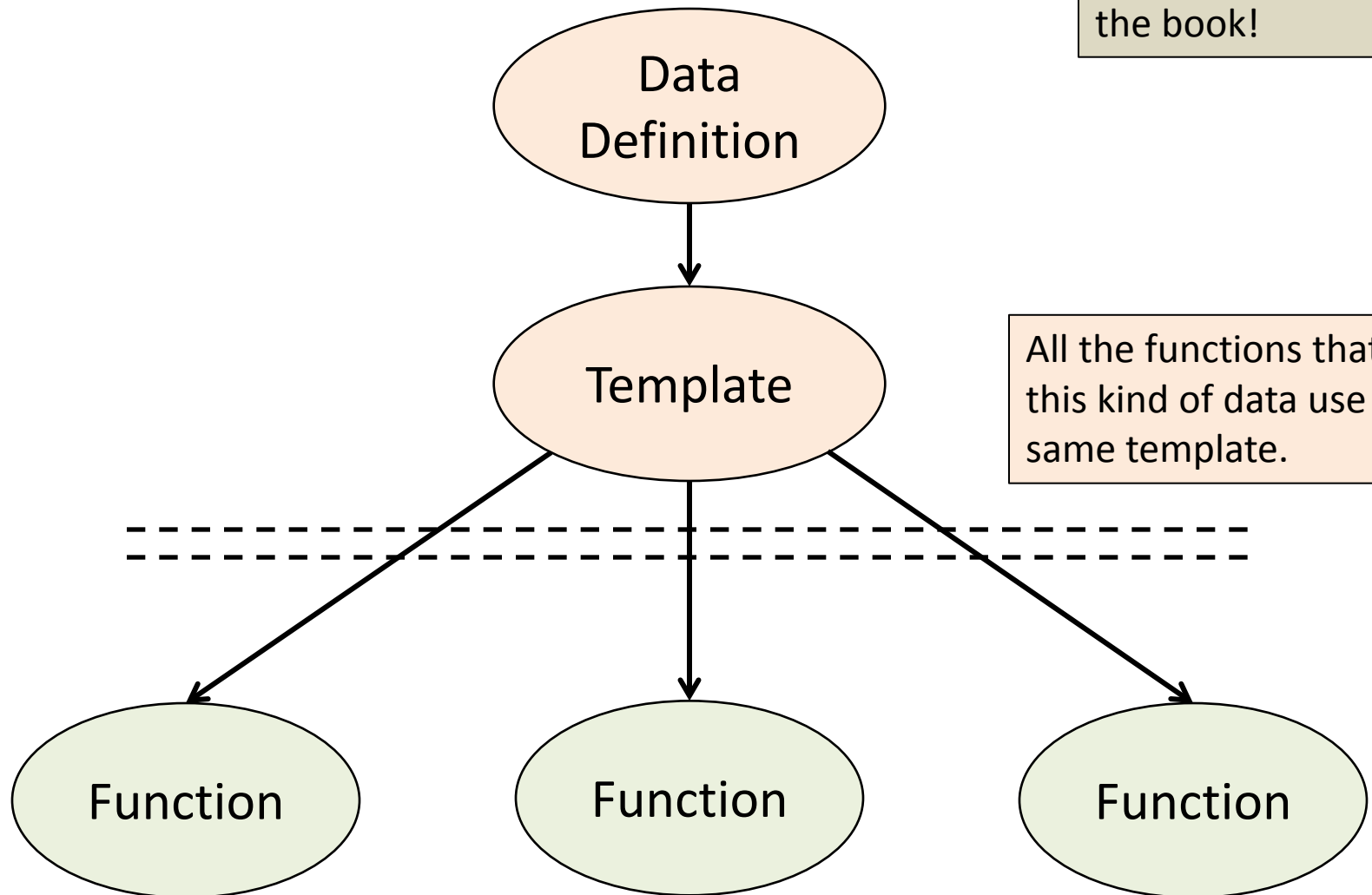
CS 5010 Program Design Paradigms
“Bootcamp”
Lesson 3.1

The Template goes with the Data Definition

- Add this to your "slogans" list!
- It is a deliverable for step 1 of the recipe!

Templates go with Data Definitions

This is different from the book!



Recipe for Templates

Question	Answer
Does the data definition distinguish among different subclasses of data?	Your template needs as many <u>cond</u> clauses as subclasses that the data definition distinguishes.
How do the subclasses differ from each other?	Use the differences to formulate a condition per clause.
Do any of the clauses deal with structured values?	If so, add appropriate selector expressions to the clause.
Does the data definition use self-references?	Formulate "natural recursions" for the template to represent the self-references of the data definition.

Example

;;; A NonEmptyList<X> is either

;;; (cons X empty)

;;; (cons X NonEmptyList<X>)

;;; f : NonEmptyList<X> -> ??

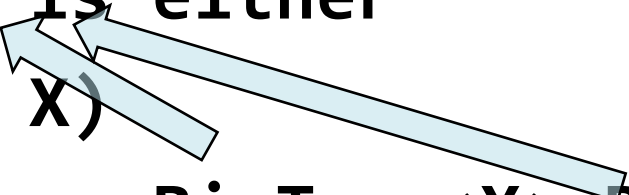
;;; livecoding!

Example

We'll study multiply recursive data definitions next week.

```
(define-struct bintree (left right))  
(define-struct leaf (data))
```

```
;;; A BinTree<X> is either  
;; -- (make-leaf X)  
;; -- (make-bintree BinTree<X> BinTree<X>)
```



```
;;; f : BinTree<X> -> ???  
;;; livecoding!
```

A Bigger Example

```
;;; A CatAnimationKeyEvent is a KeyEvent that is one of
;;; " "      interp: pause the animation
;;; "n"      interp: create a new cat
;;; "k"      interp: kill the selected cat if there is one
;;; any other one-character key event  interp: display a "?" on the screen
;;; "left"   interp: move the selected cat left if there is one
;;; "right"  interp: move the selected cat right if there is one
;;; "drop"   interp: drop the selected cat to the bottom
;;; any other KeyEvent
```

```
;;; cake-fn : CatAnimationKeyEvent -> ???
(define (cake-fn kev)
  (cond
    [(key=? kev " ") ...]
    [(key=? kev "n") ...]
    [(key=? kev "k") ...]
    [(= (string-length kev) 1) ...]
    [(key=? kev "left") ...]
    [(key=? kev "right") ...]
    [(key=? kev "drop") ...]
    [else ...]))
```

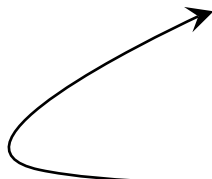
Exhaustive: covers all possibilities

1-1 correspondence!
Same order!

What's wrong with this data definition?

From Template to Code

1. Do the first four steps of the design recipe first!!!!!!
2. Copy the template and uncomment it.
3. Add additional arguments if necessary
4. Plug in the name of the function you are defining.
5. Fill in the blanks in the template.



Don't do anything else!

Following the template means filling in the blanks

```
;;; cake-fn : CatAnimationKeyEvent -> ???  
(define (cake-fn kev)  
  (cond  
    [(key=? kev " ") ...]  
    [(key=? kev "n") ...]  
    [(key=? kev "k") ...]  
    [(= (string-length kev) 1) ...]  
    [(key=? kev "left") ...]  
    [(key=? kev "right") ...]  
    [(key=? kev "down") ...]  
    [else ...]))
```

Fill in the function name, contract, arguments, and strategy

```
;; cat-after-key-event
;; : Cat CatAnimationKeyEvent -> Cat
;; strategy: Struct. Decomp. on kev
;;           [CatAnimationKeyEvent]
(define (cat-after-key-event cat kev)
  (cond
    [(key=? kev " ") ...]
    [(key=? kev "n") ...]
    [(key=? kev "k") ...]
    [(= (string-length kev) 1) ...]
    [(key=? kev "left") ...]
    [(key=? kev "right") ...]
    [(key=? kev "down") ...]
    [else ...]))
```

Now fill in the other blanks with
functional compositions of the
arguments

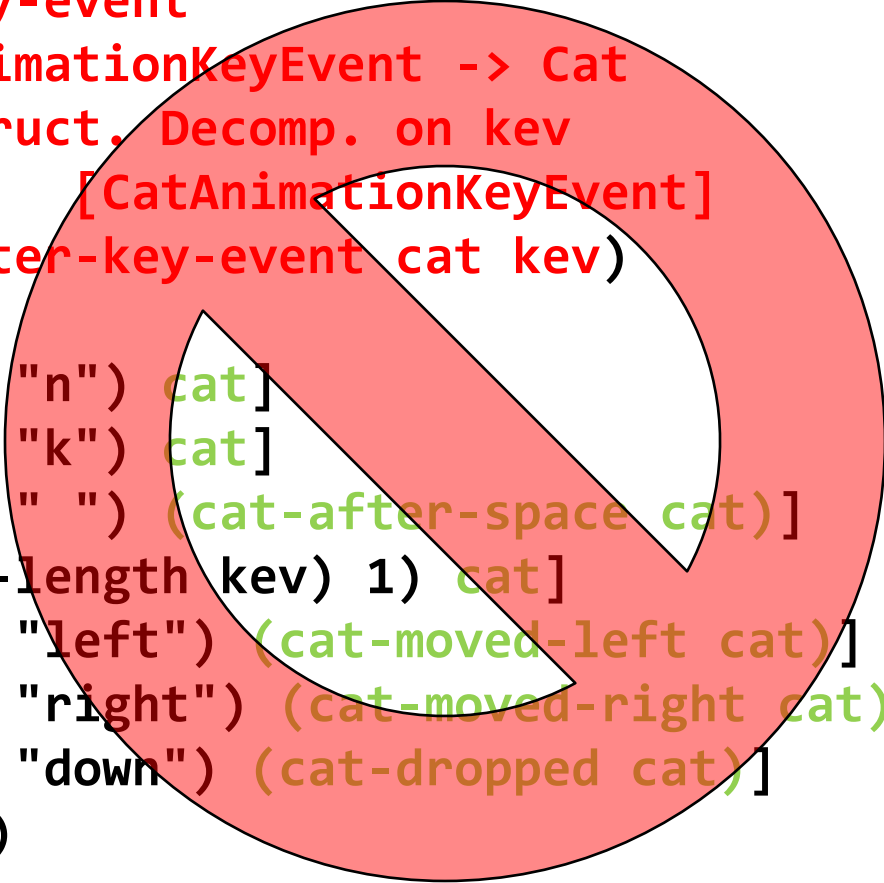
```
;; cat-after-key-event
;; : Cat CatAnimationKeyEvent -> Cat
;; strategy: Struct. Decomp. on kev
;; [CatAnimationKeyEvent]
(define (cat-after-key-event cat kev)
  (cond
    [(key=? kev " ") (cat-after-space cat)]
    [(key=? kev "n") cat]
    [(key=? kev "k") cat]
    [(= (string-length kev) 1) cat]
    [(key=? kev "left") (cat-moved-left cat)]
    [(key=? kev "right") (cat-moved-right cat)]
    [(key=? kev "down") (cat-dropped cat)]
    [else cat])))
```

Now fill in the other blanks with functional compositions of the arguments

```
;; cat-after-key-event
;; : Cat CatAnimationKeyEvent -> Cat
;; strategy: Struct. Decomp. on kev
;; [CatAnimationKeyEvent]
(define (cat-after-key-event cat kev)
  (cond
    [(key=? kev " ") (cat-after-space cat)]
    [(key=? kev "n") cat]
    [(key=? kev "k") cat]
    [(= (string-length kev) 1) cat]
    [(key=? kev "left") (cat-moved-left cat)]
    [(key=? kev "right") (cat-moved-right cat)]
    [(key=? kev "down") (cat-dropped cat)]
    [else cat]))
```

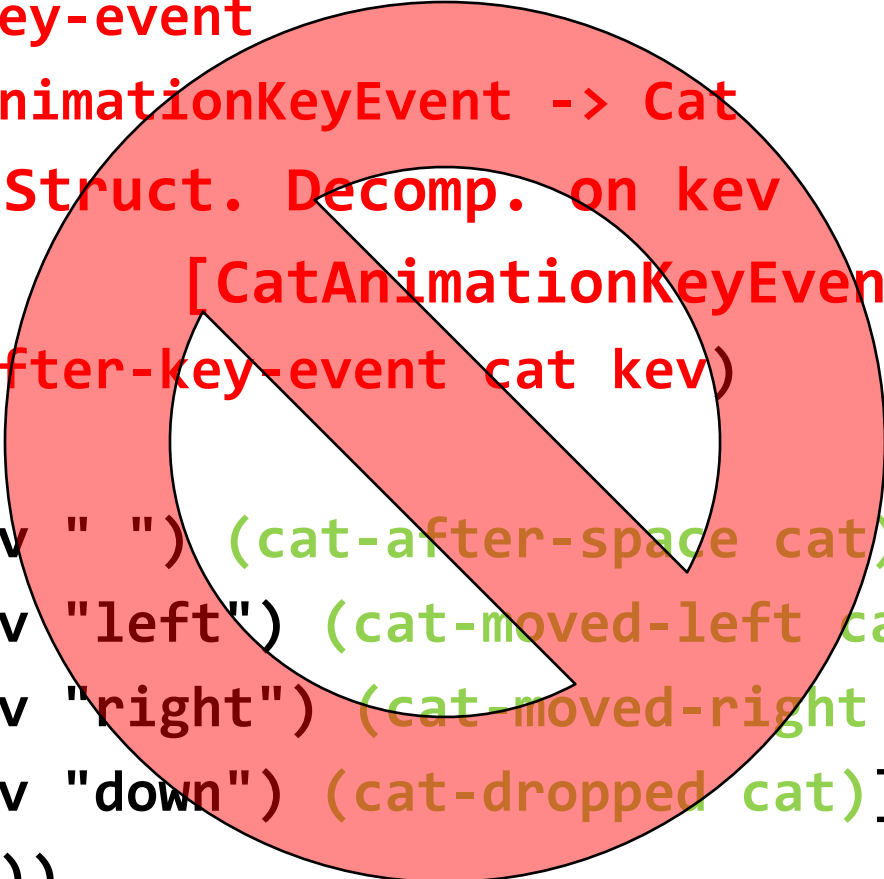
This code doesn't follow the template

```
;; cat-after-key-event
;; : Cat CatAnimationKeyEvent -> Cat
;; strategy: Struct. Decomp. on kev
;; [CatAnimationKeyEvent]
(define (cat-after-key-event cat kev)
  (cond
    [(key=? kev "n") cat]
    [(key=? kev "k") cat]
    [(key=? kev " ") (cat-after-space cat)]
    [(= (string-length kev) 1) cat]
    [(key=? kev "left") (cat-moved-left cat)]
    [(key=? kev "right") (cat-moved-right cat)]
    [(key=? kev "down") (cat-dropped cat)]
    [else cat]))
```



Neither does this

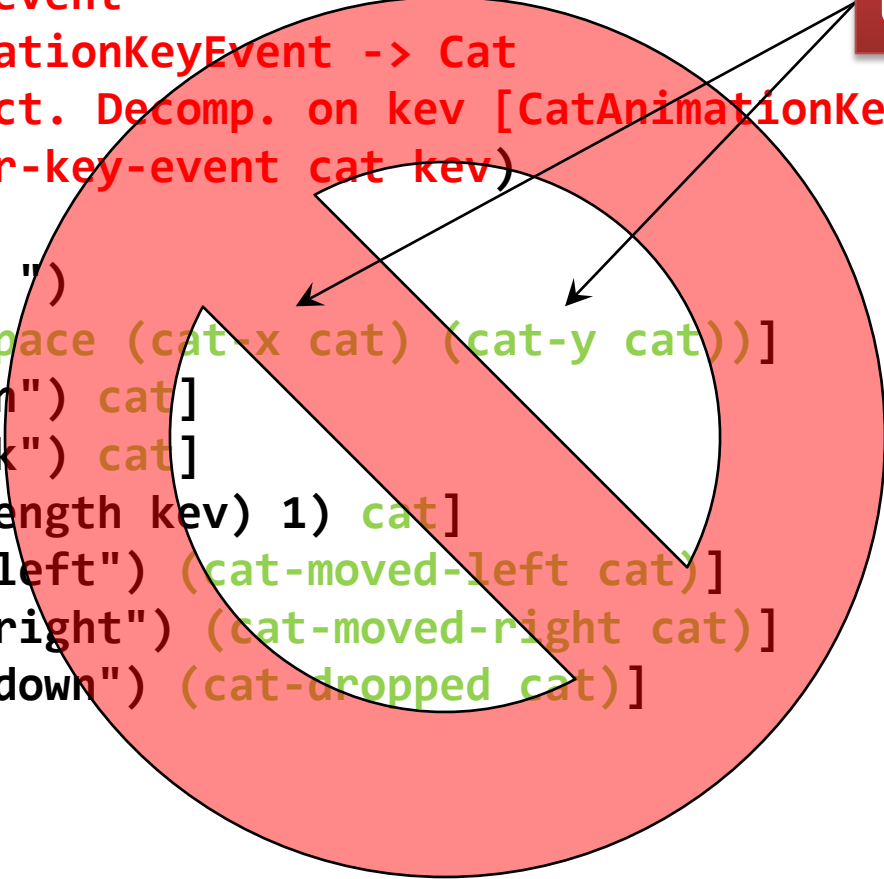
```
;; cat-after-key-event
;; : Cat CatAnimationKeyEvent -> Cat
;; strategy: Struct. Decomp. on kev
;; [CatAnimationKeyEvent]
(define (cat-after-key-event cat kev)
  (cond
    [(key=? kev " ") (cat-after-space cat)]
    [(key=? kev "left") (cat-moved-left cat)]
    [(key=? kev "right") (cat-moved-right cat)]
    [(key=? kev "down") (cat-dropped cat)]
    [else cat]))
```



And Neither Does This

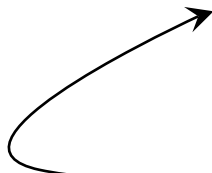
No opening up
the cat!

```
;; cat-after-key-event
;; : Cat CatAnimationKeyEvent -> Cat
;; strategy: Struct. Decomp. on kev [CatAnimationKeyEvent]
(define (cat-after-key-event cat kev)
  (cond
    [(key=? kev " ")
     (cat-after-space (cat-x cat) (cat-y cat))]
    [(key=? kev "n") cat]
    [(key=? kev "k") cat]
    [(= (string-length kev) 1) cat]
    [(key=? kev "left") (cat-moved-left cat)]
    [(key=? kev "right") (cat-moved-right cat)]
    [(key=? kev "down") (cat-dropped cat)]
    [else cat]))
```



From Template to Code

1. Do the first four steps of the design recipe first!!!!!!
2. Copy the template and uncomment it.
3. Add additional arguments if necessary
4. Plug in the name of the function you are defining.
5. Fill in the blanks in the template.



Don't do anything else!

Summary

- Write the template as part of your definition, before you code!
- If you get the template right, writing the code is just a matter of filling in the blanks.
- Getting the template right is 75% of the job.

Generalizing Similar Functions

CS 5010 Program Design Paradigms
“Bootcamp”
Lesson 3.1

Goal of Generalization

- Never write the same code twice
 - [Python: “Don’t repeat yourself”]
 - Single Point of Control
 - fix each bug only once!
 - for easier maintenance, modification
- Copy and Paste is Evil
- aka: Refactoring
- Book calls this "abstraction"

find-dog

```
;; find-dog : ListOf<String> -> Boolean
;; returns true if "dog" is in the given list.
;; strategy: structure (ListOf<String>) on los
(define (find-dog los)
  (cond
    [(empty? los) false]
    [else (or
            (string=? (first los) "dog")
            (find-dog (rest los)))])])

(check-equal? (find-dog (list "cat" "dog" "weasel")) true)
(check-equal? (find-dog (list "cat" "elephant" "weasel"))
  false)
```

find-cat

```
;; find-cat : ListOf<String> -> Boolean
;; returns true if "cat" is in the given list
;; strategy: structure (ListOf<String>) on los
(define (find-cat los)
  (cond
    [(empty? los) false]
    [else (or
            (string=? (first los) "cat")
            (find-cat (rest los)))])])

(check-equal? (find-cat (list "cat" "dog" "weasel")) true)
(check-equal? (find-cat (list "elephant" "weasel")) false)
```

Everything working?

- Original functions are tested 😊
- OK, now we can try to eliminate duplication

These functions are very similar:

```
(define (find-dog los)
  (cond
    [(empty? los) false]
    [else
     (or
      (string=?
       (first los)
       "dog")
      (find-dog
       (rest los)))]))

(define (find-cat los)
  (cond
    [(empty? los) false]
    [else
     (or
      (string=?
       (first los)
       "cat")
      (find-cat
       (rest los)))]))
```

So generalize them by adding an argument

```
;; find-animal : ListOf<String> String -> Boolean  
;; returns true iff the given string is in the given los.
```

```
(define (find-animal los str)  
  (cond  
    [(empty? los) false]  
    [else (or  
            (string=? (first los) str)  
            (find-animal (rest los) str))]))
```

This is what they call
"refactoring"

```
(check-expect  
  (find-animal (list "cat" "elephant" "weasel") "elephant")  
  true)  
(check-expect  
  (find-animal (list "cat" "elephant" "weasel") "beaver")  
  false)
```

Nothing
mysterious here!

Abstraction = Generalization

- We saw that our functions were both special cases of a more general function.
- The more general function takes extra arguments that express the differences
- The arguments "specialize" the function.
- We must check to see that we can really specialize back to our original functions:

Make sure you can still express the
original functions!

```
(define (find-dog los)
  (find-animal los "dog"))
```

```
(define (find-cat los)
  (find-animal los "cat"))
```

```
(define (find-elephant los)
  (find-animal los "elephant"))
```

How to test the new definitions?

- Comment out the originals to get them out of the way:

```
#;(define (find-dog los) ...)
; (define (find-cat los)
; ...)
```

DON'T use
"comment out in a box"

- Now the old names refer to the new versions, and all the old tests should still work.

Time for pizza

```
;; Data Definitions:
```

```
;; A Topping is a String.
```

```
;; A Pizza is a ListOf<Topping>
```

```
;; interp: a pizza is a list of toppings, listed from top to bottom
```

```
;; pizza-fn : Pizza -> ??
```

```
; Given a Pizza, produce ....
```

```
;; (define (pizza-fn p)
```

```
;;   (cond
```

```
;;     [(empty? p) ...]
```

```
;;     [else (... (first p)
```

```
;;               (pizza-fn (rest p))]))))
```

```
;; Examples:
```

```
(define plain-pizza empty)
```

```
(define cheese-pizza (list "cheese"))
```

```
(define anchovies-cheese-pizza (list "anchovies" "cheese"))
```

replace-all-anchovies-with-onions

```
;; replace-all-anchovies-with-onions
;;   : Pizza -> Pizza
;; Returns a pizza like the given pizza, but with
;; anchovies in place of each layer of onions
(define (replace-all-anchovies-with-onions p)
  (cond
    [(empty? p) empty]
    [else (if (string=? (first p) "anchovies")
              (cons "onions"
                    (replace-all-anchovies-with-onions
                     (rest p)))
              (cons (first p)
                    (replace-all-anchovies-with-onions
                     (rest p)))))]))
```

More and more general...

```
;; replace-all-anchovies-with-onions
```

```
;;   : Pizza -> Pizza
```

```
;; Returns a pizza like the given pizza, but with
```

```
;; anchovies in place of each layer of onions
```

```
;; replace-all-anchovies : Pizza Topping -> Pizza
```

```
;; Returns a pizza like the given pizza, but with
```

```
;; all anchovies replaced by the given topping.
```

```
;; replace-topping : Pizza Topping Topping -> Pizza
```

```
;; Returns a pizza like the given one, but with
```

```
;; all instances of the first topping replaced by
```

```
;; the second one.
```

Summary

- Sometimes functions will differ only in choice of data items
- Generalize these by adding new argument(s) for the differences.
- Make sure originals work before generalizing
- Test by commenting out the originals and running the same tests.

Abstracting Over Functions

CS 5010 Program Design Paradigms
“Bootcamp”
Lesson 3.2

Sometimes the differences are themselves functions

Consider:

```
:: ListOf<Number> -> ListOf<Number>  
(define (add-1-to-each lon)  
  (cond  
    [(empty? lon) empty]  
    [(else (cons  
              (add1 (first lon))  
              (add1-to-each (rest lon))))]))
```

Versus...

```
(define-struct personnel (name salary))  
;; Personnel = (make-personnel String Number)  
  
;; ListOf<Personnel> -> ListOf<String>  
(define (extract-names lop)  
  (cond  
    [(empty? lop) empty]  
    [else (cons  
              (personnel-name (first lop))  
              (extract-names (rest lop))))]))
```

Differences

ListOf<Number> -> ListOf<Number>

```
(define (add-1-to-each lon)
  (cond
    [(empty? lon) empty]
    [(else (cons
              (add1
                (first lon))
              (add1-to-each
                (rest lon))))]))
```

ListOf<Personnel> -> ListOf<String>

```
(define (extract-names lop)
  (cond
    [(empty? lop) empty]
    [(else (cons
              (personnel-name
                (first lop))
              (extract-names
                (rest lop))))]))
```




So add an argument for the difference:

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

```
(define (add-1-to-each lon)
  (apply-to-each add1 lon))
```

```
(define (extract-names lop)
  (apply-to-each personnel-name lop))
```

Passing a function as an argument is not legal in BSL, so we are switching to ISL.



What strategy is this?

```
;; strategy: struct decomp on lst : ListOf<X>  
(define (apply-to-each fn lst)  
  (cond  
    [(empty? lop) empty]  
    [else (cons  
              (fn (first lst))  
              (apply-to-each fn (rest lst))))]))
```

```
;; strategy: higher-order function composition  
(define (add-1-to-each lon)  
  (apply-to-each add1 lon))
```

Different from book!



```
;; strategy: higher-order function composition  
(define (extract-names lop)  
  (apply-to-each personnel-name lop))
```

Testing: as in last lesson

- Get original functions tested & working first
- Then comment them out.
- The tests will now see your new versions
- The tests should still work

Doing something complicated?

Define your own function. Can use `local` for this:

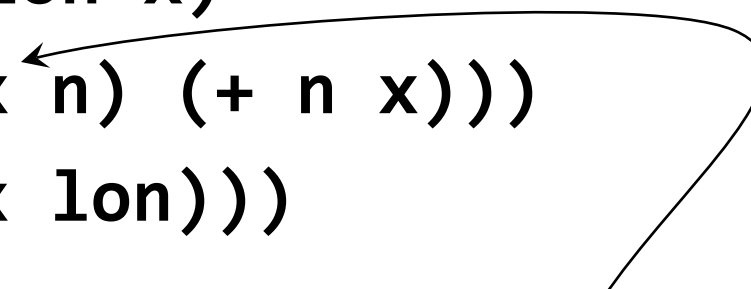
```
;; ListOf<Number> -> ListOf<Number>  
;; returns a list like the given one,  
;; but with 5 added to each number.  
(define (add-5-to-each lon)  
  (local ((define (add5 n) (+ n 5)))  
    (apply-to-each add5 lon)))
```

In ISL, `local` allows you to create *local* definitions. See HtDP2, sec 6.3.2.

Opportunity for more abstraction!

- Now abstract on the "5":

```
(define (add-x-to-each lon x)
  (local ((define (addx n) (+ n x))))
  (apply-to-each addx lon)))
```



This function has to be local.
Why?

What's the contract for apply-to-each?

```
(apply-to-each add1 lon)
```

```
(apply-to-each personnel-name lop)
```

apply-to-each :

(X->Y) List<X> -> List<Y>

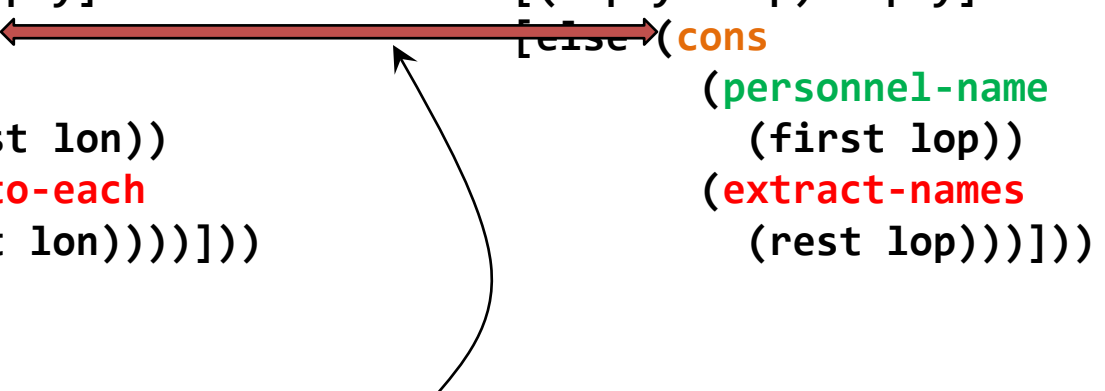
Standard name for apply-to-each: **map**.

That's what we will call it from now on.

What else could be different?

```
;ListOf<Number> -> ListOf<Number>
(define (add-1-to-each lon)
  (cond
    [(empty? lon) empty]
    [(else (cons
              (add1
                (first lon))
              (add1-to-each
                (rest lon))))))]))
```

```
;ListOf<Personnel> -> ListOf<String>
(define (extract-names lop)
  (cond
    [(empty? lop) empty]
    [(else (cons
              (personnel-name
                (first lop))
              (extract-names
                (rest lop))))))]))
```



These are both `cons`, but you could have different things here

Another example

```
;; ListOf<Number> -> Number
(define (sum lon)
  (cond
    [(empty? lon) 0]
    [else (+
            (first lon)
            (sum
             (rest lon))))]))
```

```
;; ListOf<Number> -> Number
(define (product lon)
  (cond
    [(empty? lon) 1]
    [else (*
            (first lon)
            (product
             (rest lon))))]))
```



Create 2 new arguments for the 2 differences.

We could call this "foldr" (don't ask why)

```
(define (foldr fcn val lon)
  (cond
    [(empty? lon) val]
    [else (fcn
              (first lon)
              (foldr fcn val (rest lon)))]))
```

This is predefined in ISL, so you don't need to write out this definition

```
;; strategy: higher-order function composition
(define (sum lon) (foldr + 0 lon))
(define (product lon) (foldr * 1 lon))
```

What's the contract for **foldr**?

foldr :

(Number Number -> Number) Number ListOf<Number>
-> Number

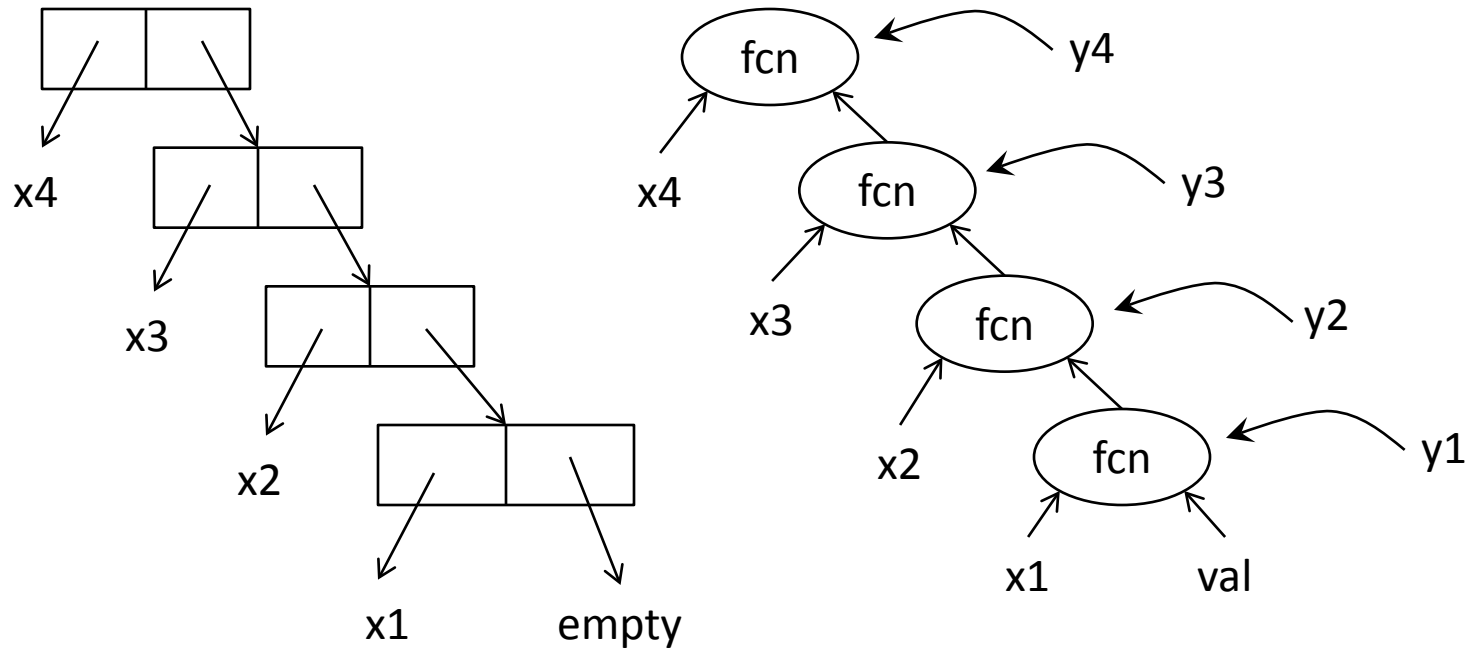
Does **foldr** really depend on the fact that it's working with numbers?

Clearly not:

foldr : (X X -> X) X ListOf<X> -> X

Could it be even more general?

The general picture



fcn : X Y -> Y

val : Y

foldr : (X Y -> Y) Y ListOf<X> -> Y

Another example:

```
(define (add1-if-true b n)
  (if b (+ n 1) n))
```

```
(define (count-trues lob)
  (foldr add1-if-true 0 lob))
```

Or even better:

```
(define (count-trues lob)
  (local ((define (add1-if-true b n)
              (if b (+ n 1) n)))
    (foldr add1-if-true 0 lob)))
```

What are the contracts?

add1-if-true : Boolean Number -> Number
count-trues : ListOf<Boolean> -> Number

So

foldr :
 (Boolean Number -> Number)
 Number ListOf<Boolean> -> Number

In general:

foldr : (X Y -> Y) Y ListOf<X> -> Y

Local functions need contracts and purpose statements, too

```
(define (count-trues lob)
  (local (; Boolean Number -> Number
          ; add 1 to the number if boolean is true,
          ; otherwise leave it unchanged.
          (define (add1-if-true b n)
            (if b (+ n 1) n)))
    (foldr add1-if-true 0 lob)))
```

- They count as help functions, so they don't need separate tests.

The whole thing (less examples and tests)

```
;; count-trues : ListOf<Boolean> -> Number
;; returns the number of trues in the given list of booleans.
;; strategy: higher-order function composition
(define (count-trues lob)
  (local (; Boolean Number -> Number
          ; add 1 to the number if boolean is true,
          ; otherwise leave it unchanged.
          (define (add1-if-true b n)
            (if b (+ n 1) n)))
    (foldr add1-if-true 0 lob)))
```

Mapreduce, hadoop, and all that

`(mapreduce f v g lst) = (fold f v (map g lst))`

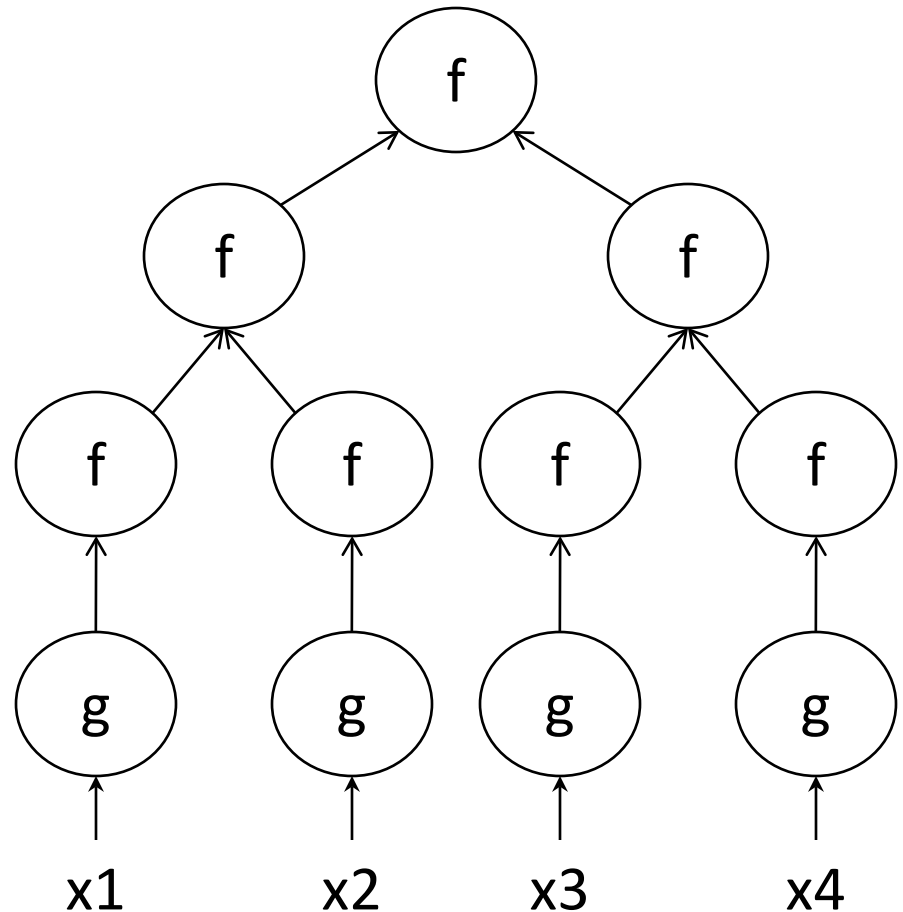
Therefore:

`(mapreduce f v g (list x1 ... xn)) =
 (f (g x1)
 (f (g x2)
 (f (g x3)
 ...
 v))))`

- If `f` is associative, and `v` is its identity, can turn the calls to "`f`" into a tree and do them in parallel on a server farm!

From linear time to logarithmic

$(f (g x1)$
 $(f (g x2)$
 $(f (g x3)$
 $(f (g x4)$
 $v)))) =$



<Gloat>

- The functional programming community has known about mapreduce for 30 years
- The rest of the world is just catching up!

Summary

- Sometimes the differences between functions are also functions
- Abstract in just the same way: add new arguments for each difference
- Can use **local** to define local functions.
- Local functions need contracts & purpose statements, too.
- New strategy: higher-order function composition

Using the Built-In List Abstractions

CS 5010 Program Design Paradigms
“Bootcamp”
Lesson 3.4

Goals of this lesson

- In the last lesson, we introduced a new strategy: “higher-order function composition”
- ISL comes with some other useful pre-built abstractions for processing lists.
- We'll review these and discuss how to choose the one that's appropriate for your function.
- Result: a recipe for converting your structural decomposition into a higher-order function composition

Pre-built list abstractions (1)

(Section 6.3.1, Page 300)

```
;; map : (X -> Y) ListOf<X> -> ListOf<Y>
;; construct a list by applying f to each item of the given
;; list.
;; that is, (map f (list x_1 ... x_n))
;;          = (list (f x_1) ... (f x_n))
(define (map f alox) ...)
```

```
;; foldr : (X Y -> Y) Y ListOf<X> -> Y
;; apply f on the elements of the given list from right to
;; left, starting with base.
;; (foldr f base (list x_1 ... x_n))
;;   = (f x_1 ... (f x_n base))
(define (foldr f base alox) ...)
```

Pre-built list abstractions (2)

(Section 6.3.1, Page 300)

```
;; build-list : N (N -> X) -> ListOf<X>  
;; construct (list (f 0) ... (f (- n 1)))  
(define (build-list n f) ...)
```

```
;; filter : (X -> Boolean) ListOf<X> -> ListOf<X>  
;; construct the list from all items on alox for which p  
;; holds  
(define (filter p alox) ...)
```

Pre-built list abstractions (3)

(Section 6.3.1, Page 302)

```
;; andmap : (X -> Boolean) ListOf<X> -> Boolean
;; determine whether p holds for every item on alox
;; that is, (andmap p (list x_1 ... x_n))
;;           = (and (p x_1) ... (p x_n))
(define (andmap p alox) ...)
```

```
;; ormap : (X -> Boolean) ListOf<X> -> Boolean
;; determine whether p holds for at least one item on alox
;; that is, (ormap p (list x_1 ... x_n))
;;           = (or (p x_1) ... (p x_n))
(define (ormap p alox) ...)
```

Recipe for Rewriting Your Function to Use the Pre-Built List Abstractions

Recipe for Rewriting your function to use the Pre-Built List Abstractions

0. Start with your function definition, written using structural decomposition
1. Determine whether the function is a candidate for using one of the built-in abstractions
2. Determine which built-in abstraction is appropriate
3. Rewrite your function using the built-in abstraction. The new strategy is Higher-Order Function Composition
4. Comment out the old definition. Do not change the contract, purpose statement, examples, or tests.

Example

```
;; strategy: structural decomposition
(define (add-1-to-each lon)
  (cond
    [(empty? lon) empty]
    [(else (cons
              (add1 (first lon))
              (add1-to-each (rest lon))))]))
```

```
;; strategy: higher-order function composition
(define (add-1-to-each lon)
  (apply-to-each add1 lon))
```

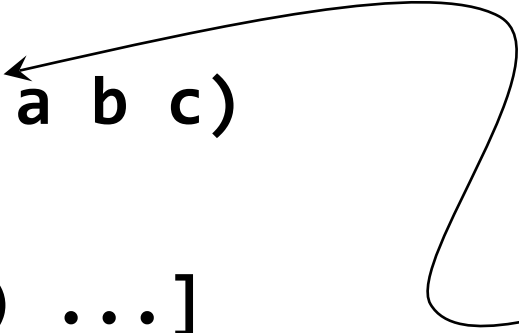
Another example

```
;; ListOf<Number> -> Number
;; strategy: struct decomp
(define (sum lon)
  (cond
    [(empty? lon) 0]
    [else (+
            (first lon)
            (sum
             (rest lon))))]))
```

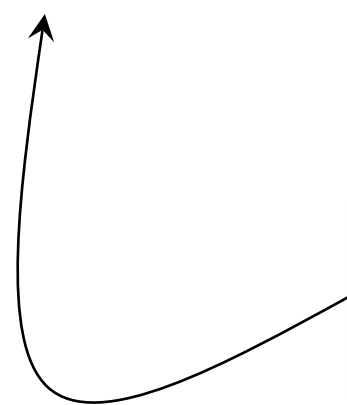
```
;; ListOf<Number> -> Number
;; strategy: struct decomp
(define (sum lon)
  (foldr + 0 lon))
```

A candidate for **map** or **foldr** looks like this:

```
(define (f lst a b c)
  (cond
    [(empty? lst) ...]
    [else (...
              (first lst)
              (f (rest lst) a b c))]))
```



takes a list and
some other
arguments



recurs on the rest of the
list; other arguments
don't change

A candidate for **map** looks like this:

```
(define (f lst a b c)
  (cond
    [(empty? lst) empty]
    [else (cons
              (... (first lst) a b c)
              (f (rest lst) a b c))]))
```

empty here



cons here

A candidate for **andmap** looks like this:

```
(define (f lst a b c)
  (cond
    [(empty? lst) true]
    [else (and
             (... (first lst) a b c)
             (f (rest lst) a b c))]))
```

true here



and here

A candidate for **ormap** looks like this:

```
(define (f lst a b c)
  (cond
    [(empty? lst) false]
    [else (or
              (... (first lst) a b c)
              (f (rest lst) a b c))]))
```

false here



or here

A candidate for **foldr** looks like this:

```
(define (f lst a b c)
  (cond
    [(empty? lst) ...]
    [else (...
              (first lst)
              (f (rest lst) a b c))]))
```

and none of the above patterns (**map**,
andmap, **ormap**) apply.

A candidate for **filter** looks like this:

```
(define (f lst a b c)
  (cond
    [(empty? lst) empty]
    [else (if (... (first lst) a b c)
                (cons
                  (first lst)
                  (f (rest lst) a b c))
                (f (rest lst) a b c))]))
```

takes a list and
some other
arguments

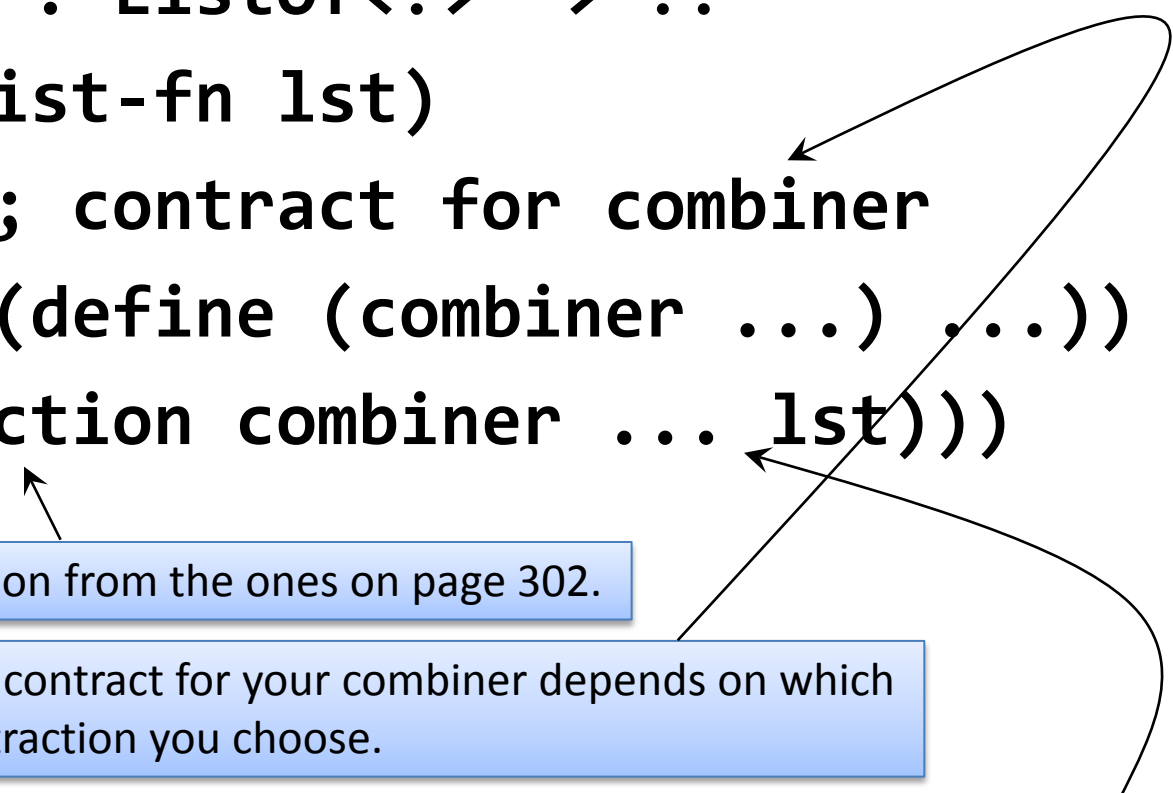
makes a decision
based only on the
first of the list

if test is true, includes
the first element in
the answer

recurs on the rest of the
list; other arguments
don't change

Pattern for higher-order function composition

```
;; list-fn : ListOf<?> -> ??  
(define (list-fn lst)  
  (local (; contract for combiner  
          (define (combiner ...) ...))  
    (abstraction combiner ... lst)))
```



Choose your abstraction from the ones on page 302.

The contract for your combiner depends on which abstraction you choose.

The arguments for the different abstractions are different. If this were foldr, the base would go here.

Pattern for higher-order function composition: **map**

```
:: list-fn : ListOf<X> -> ListOf<Y>  
(define (list-fn lox)  
  (local (; X -> Y  
    ; purpose statement for operator  
    (define (operator x) ...))  
    (map operator lox)))
```

Pattern for higher-order function composition: **filter**

```
;; list-fn : ListOf<X> -> ListOf<X>
(define (list-fn lox)
  (local (; X -> Boolean
          ; purpose statement for test
          (define (test x) ...))
    (filter combiner lox)))
```

Pattern for higher-order function composition: andmap/ormap

```
;; list-fn : ListOf<X> -> Boolean
(define (list-fn lox)
  (local (; X -> Boolean
          ; purpose statement for test
          (define (test x) ...))
    (abstraction test lox)))
```



andmap or ormap

Pattern for higher-order function composition: foldr

```
;; list-fn : ListOf<X> -> Y
(define (list-fn lox)
  (local (; X Y -> Y
          ; purpose statement for combiner
          (define (combiner x y) ...))
    (foldr combiner base-val lox)))
```

Recognizing Opportunities for HOFC

- Once you get the idea, you can anticipate when you can use HOFC
- Key: look at the contracts
- Here's a recipe...

Recipe for Using Higher-Order Function Composition

1. Write the contract, purpose statement, and examples for your function.
2. Choose a function from page 302 whose contract matches yours. What choices for X , Y , etc. match your contract?
3. Create a copy of the pattern for the function. What is the contract for the combiner? What is its purpose?
4. Define the combiner function.
5. Test as usual.

Example

Step 1:

```
;; Nat -> ListOf<Posn>  
;; create the diagonal (0,0), ..., (n,n)  
;; (diagonal 1)  
;; = (list (make-posn 0 0) (make-posn 1 1))
```

Step 2: build-list is the only possible candidate, with X = Posn

Step 3:

```
(define (diagonal n)  
  (local (;; Nat -> Posn  
          ;; create a diagonal position from the given  
          ;; Nat.  
          (define (make-diagonal-posn ...) ...))  
    (build-list n make-diagonal-posn)))
```

Example, cont'd

Step 4:

```
(define (diagonal n)
  (local (;; Nat -> Posn
          ;; create a diagonal position from the given
          ;; Nat.
          (define (make-diagonal-posn i) (make-posn i i))
          (build-list n make-diagonal-posn)))
```

Step 5: test

Oops, it should have been

```
(build-list (+ n 1) make-diagonal-posn)
```

Summary

- ISL and Racket offer a bunch of pre-defined abstractions for working on lists.
- We'll see a few more of these later on
- Write your function first, then see if it can be rewritten using one of the abstractions
- OR: choose an abstraction by identifying the candidates with appropriate types.
- Then fill in the blanks for your operation, test, or combiner and base value.
- Remember, it's still just good old structural decomposition— it's just prepackaged for you.

Using HOF with lambda

CS 5010 Program Design Paradigms
“Bootcamp”
Lesson 3.4

Defining functions using **lambda**

- The value of a lambda expression is a function.
- You can use the lambda expression anywhere you would use the function
- The value of **(lambda (x) (+ x 5))** is a function that adds 5 to its argument.
- **(map (lambda (x) (+ x 5)) lon)** returns a list like **lon**, but with 5 added to each element.

Using **lambda** cuts down on the junk in your code

These two are the same:

```
(local  
  ((define (add5 x) (+ x 5))  
  (map add5 lon))
```

```
(map (lambda (x) (+ x 5)) lon)
```

Each returns a list like **lon**, but with 5 added to each element.

Defining combinators using lambda


- Don't even need to give a name to the combiner.
- Instead, can use **lambda** to create a function and pass it directly to the abstraction.
- Still need to give contract and purpose statement for the lambda function

Higher-order function composition using **lambda** : map

```
;; list-fn : ListOf<X> -> ListOf<Y>
(define (list-fn lox)
  (map
    ; X -> Y
    ; purpose statement for the operator
    (lambda (x) ...)
    lox))
```

Where does the value of **x** come from?

```
lon = (list 10 20 30 40)
```



```
(map (lambda (x) (+ x 5))  
lon)
```

```
= (list 15 25 35 45)
```

Higher-order function composition using **lambda** : filter

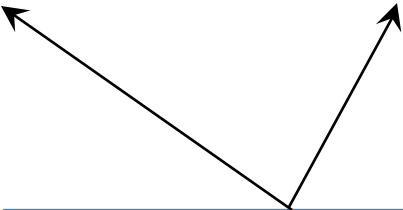
```
;; list-fn : ListOf<X> -> ListOf<X>
(define (list-fn lox)
  (filter
    ; X -> Boolean
    ; purpose statement for the test
    (lambda (x) ...)
    lox))
```

Higher-order function composition using **lambda** : andmap/ormap

```
;; list-fn : ListOf<X> -> Boolean
(define (list-fn lox)
  (andmap/ormap
    ; X -> Boolean
    ; purpose statement for the test
    (lambda (x) ...)
    lox))
```

Higher-order function composition using **lambda** : foldr

```
;; list-fn : ListOf<X> -> Y
(define (list-fn lox)
  (foldr
    ; X Y -> Y
    ; purpose statement for combiner
    (lambda (first-elt ans-for-rest) ...)
    base-val
    lox))
```



These variable names remind you where the values come from.

Combining function compositions

- You can compose these, just as you can with ordinary function composition.

Example

```
:: ListOf<Number> -> ListOf<Number>  
;; Return the squares of all the  
;; even numbers in the given list  
;; strategy: H0 Function  
Composition  
(define (squares-of-evens lon)  
  (map square (filter even? lon)))
```

Summary

- **lambda** makes it easier to write functions using the abstractions
- no need to make up a name or use **local**
- Remember, it's still just good old structural decomposition— it's just prepackaged for you.