`ormap`, `andmap`, and `filter`

CS 5010 Program Design Paradigms
Lesson 5.3

Introduction

- In this lesson, we will see more common patterns of function definitions that differ only by what functions they call.

Learning Objectives

- At the end of this lesson you should be able to:
  - recognize the **`ormap`**, **`andmap`**, and **`filter`** patterns
  - state the contracts for **`ormap`**, **`andmap`**, and **`filter`**, and use them appropriately.
  - combine these functions using higher-order function combination

Let's look at **`find-dog`** again

```
;; find-dog : ListOfString -> Boolean
;; RETURNS: true if "dog" is in the given list.
;; STRATEGY: Use template for ListOfString 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)
```

Here's another function with a similar structure

```
;; has-even? : ListOfInteger -> Boolean
;; RETURNS: true iff the given list contains
;;  an even number
;; STRATEGY: Use ListOfInteger on los

(define (has-even? los)
  (cond
    [(empty? los) false]
    [else (or
           (even? (first los))
           (has-even? (rest los)))]))
```

Let's compare

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

```
(define (has-even? los)
 (cond
  [(empty? los) false]
  [else
   (or
    (even? (first los) )

    (has-even?
     (rest los)))]))
```

Generalize by adding an argument

```
;; STRATEGY: Use template for ListOfX on lst
(define (ormap fn lst)
  (cond
     [(empty? lst) false]
    [else
      (or
        (fn (first lst))
        (ormap fn (rest lst)))]))
```

As before, we can generalize by adding an argument for the difference.

And re-create the originals

```
;; STRATEGY: Use HOF ormap on lst
(define (find-dog lst)
  (ormap
    ;; String -> Boolean
    (lambda (str) (string=? "dog" str))
    lst)))

;; STRATEGY: Use HOF ormap on lst
(define (has-even? lst)
  (ormap even? lst))
```

Again as before, we re-create the originals using our generalized function.

If you're afraid of lambda, you can define **`is-dog?`** or use a local.
But it's good to get comfortable with lambda– it's so useful that it was added to Java as of Java 8.

What's the contract for **`ormap`**?

- Let's see what kind of values each of the pieces of **ormap** returns.
- Step through the animation on the next slide to watch this work.

What's the contract?

```
ormap : (X -> Bool) ListOfX -> Bool
(define (ormap fn lst)
  (cond
    [(empty? lst) false]
    [else
     (or
      (fn (first lst))
      (ormap fn (rest lst)))]))
```

fn must take an X, because its argument is an X, and it must return a boolean, because its return value is an argument to **or**.

must return a Boolean

X

ListOfX

Boolean

X -> Bool

So fn must be a function from X's to Booleans, and lst must be a ListOfX. We write all this down in the contract.

What's the purpose statement?

We've written the function definition and the contract, but we won't be done until we have a purpose statement. Having a purpose statement allows another programmer to use this function without having to look at the code.

```
;; ormap : (X -> Boolean) ListOfX -> Boolean
;; GIVEN: A predicate p on X's and a list of X's, lox
;; RETURNS: true iff p holds for at least one value in lox
;; that is, (ormap p (list x_1 ... x_n))
;;            = (or (p x_1) ... (p x_n))
(define (ormap p lox) ...)
```

And of course we can do the same thing for **and**.

```
(define (andmap fn lst)
  (cond
    [(empty? lst) true]
    [else
      (and
        (fn (first lst))
        (andmap fn (rest lst)))]))
```

Contract and Purpose Statement

```
;; andmap : (X -> Bool) ListOfX -> Bool
;; GIVEN: A predicate p on X's
;; and a list of X's, lox
;; RETURNS: true iff p holds for every value
;; in lox
;; that is, (andmap p (list x_1 ... x_n))
;;            = (and (p x_1) ... (p x_n))
```

The contract and purpose statement look very much like the ones for **ormap**.

Another common pattern

- Another common list-manipulation problem is to take a list and return a list of those values in the list that pass a certain test.
- For example, here's a function that returns only the even values in a list of integers.

only-evens

```
;; only-evens
;;   : ListOfInteger -> ListOfInteger
;; returns the list of all the even values
;; in the list
;; STRATEGY: Use template for ListOfInteger on lst
(define (only-evens lst)
  (cond
    [(empty? lst) empty]
    [else (if (even? (first lst))
              (cons (first lst)
                    (only-evens (rest lst)))
              (only-evens (rest lst)))]))
```

Generalize: **filter**

```
;; filter : (X -> Boolean) ListOfX
;;                -> ListOfX
;; RETURNS: the list of all the elements
;; in the list that satisfy the test
;; STRATEGY: Use template for ListOfX on lst
(define (filter fn lst)
  (cond
    [(empty? lst) empty]
    [else (if (fn (first lst))
              (cons (first lst)
                    (filter fn (rest lst)))
              (filter fn (rest lst)))]))
```

The obvious thing to do here is to replace **even?** with an extra argument.

These can be strung together

```
;; ListOfInteger -> ListOfInteger
;; RETURNS: the squares of the
;; evens in the given list
;; STRATEGY: Use HOF filter on lon,
;;   followed by HOF map
(define (squares-of-evens lon)
 (map sqr
  (filter even? lon)))
```

One of the nice things about these functions is that they can be combined to create multi-pass functions.

Go crazy with these!

```
;; STRATEGY: Use HOF filter on lon,
;;   followed by HOF map twice
(define (squares-of-evens+1 lon)
 (map add1
  (map sqr
   (filter even? lon))))
```

But always make sure that your definitions are CLEAR AND UNDERSTANDABLE!

Summary

- You should now be able to:
  - recognize the **`ormap`**, **`andmap`**, and **`filter`** patterns
  - state the contracts for **`ormap`**, **`andmap`**, and **`filter`** , and use them appropriately.
  - combine these functions to form more complicated operations on lists.

Next Steps

- Study 05-3-map.rkt in the examples folder.
- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practice 5.3
- Go on to the next lesson