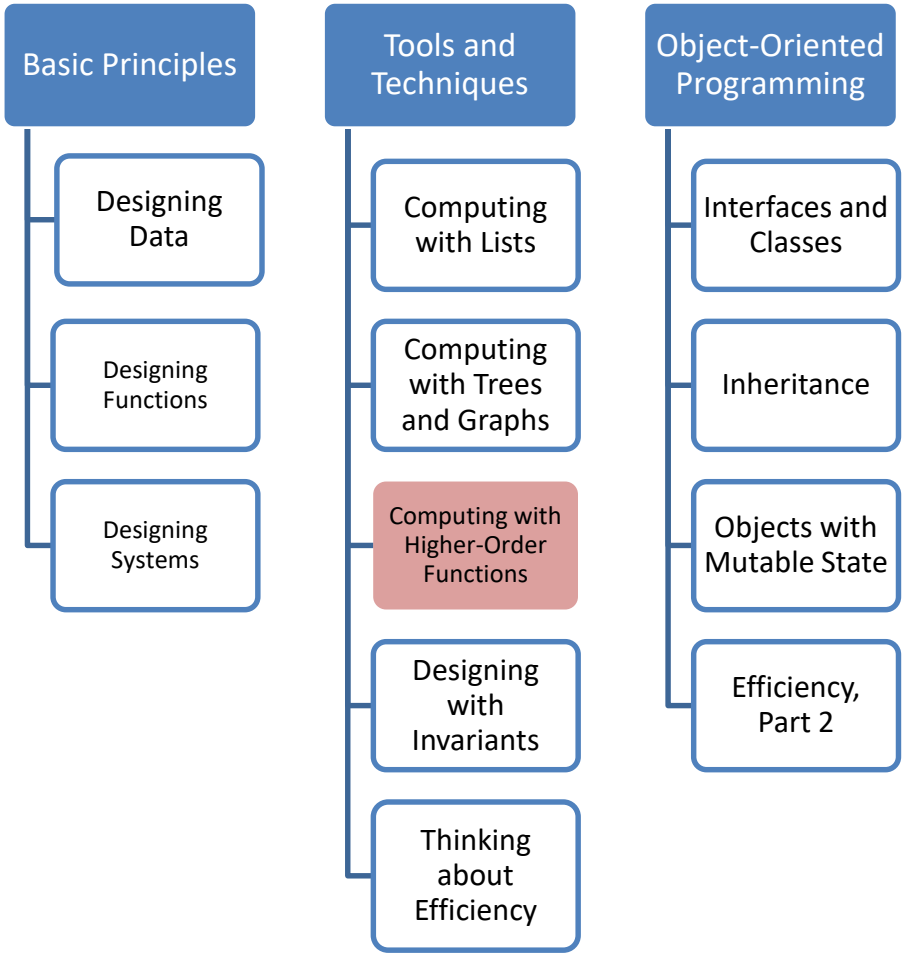
Generalizing Similar Functions

CS 5010 Program Design Paradigms "Bootcamp"

Lesson 6.1

© Mitchell Wand, 2012-2015
This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Module 06


Basic Principles
Designing Data
Designing Functions
Designing Systems
Tools and Techniques
Computing with Lists
Computing with Trees and Graphs
Computing with Higher-Order Functions
Designing with Invariants
Thinking about Efficiency
Object-Oriented Programming
Interfaces and Classes
Inheritance
Objects with Mutable State
Efficiency, Part 2

Generalization

- The goal of generalization is to avoid having to repeat code, whether the code is identical or slightly different.
- In this sequence of lessons, you will learn how to do this, starting with very simple situations, then covering more and more complex situations.

Slogans for Generalization

- Never write the same code twice
  - Don’t repeat yourself
  - Single Point of Control
    - fix each bug only once
    - easier maintenance, modification
- Copy and Paste is bad practice
- Also known as: Refactoring

Module Outline

- Generalizing a constant to a variable
- Generalizing over functions
- Using prepackaged generalizations: map, foldr, etc.

Learning Objectives for this Lesson

- By the end of this lesson, you should be able to
  - recognize when two functions differ only by a constant
  - rewrite the two functions using a single more general function
  - test your new function definitions

Imagine the following:

- Your boss comes to you and asks you to write a function called **`find-dog`**.
- You follow the design recipe, write the code, and test it.
- Your boss and you are both happy.
- Here's what you wrote:

find-dog

```
;; find-dog : StringList -> Boolean
;; GIVEN: a list of strings
;; RETURNS: true iff "dog" is in the given list.
;; STRATEGY: Use template for StringList 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)
```

The story continues

- The next morning, your boss comes to you and asks you to write **`find-cat`**.
- You follow the design recipe, write the code, and test it.
- Here's what you wrote:

find-cat

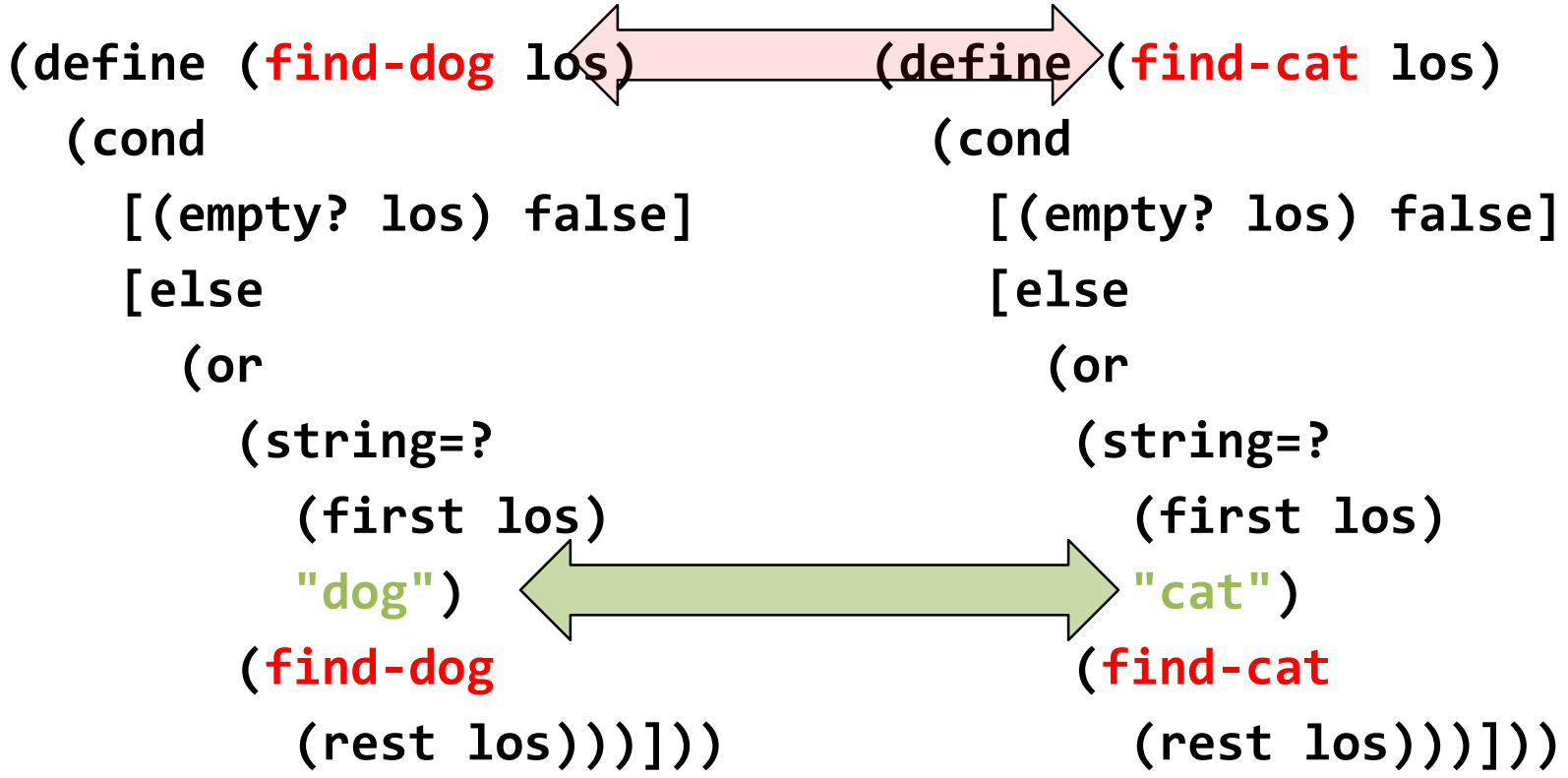
```
;; find-cat : StringList -> Boolean
;; GIVEN: a list of strings
;; RETURNS: true iff "cat" is in the given list.
;; STRATEGY: Use template for StringList on los
(define (find-cat los)
  (cond
    [(empty? los) false]
    [else (or
           (string=? (first los) "cat")
           (find-cat (rest los)))]))

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

A lot of repeated work there!

- Your boss is happy, but you are less happy; what if the next day, he asks you to write **`find-elephant`**?
- You feel like you are wasting a lot of time!
- Let's see just how alike these functions were.

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)))]))
```

The only differences between the functions are their names, and the fact that one refers to “dog” and the other refers to “cat”.

So generalize them by adding an argument

```
;; find-animal : StringList String -> Boolean
;; returns true iff the given string is in the given list of strings.

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

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

Nothing mysterious here!

What did we do here?

- If two functions differ only in a few places, add extra arguments for those places.
- **find-dog** and **find-cat** can be generalized to get **find-animal**. We replace a constant, like **"dog"** or **"cat"** with an argument, here **str**.
- Moving common code to a single function with some extra arguments is what is often called "refactoring".

Generalization

- Both functions were special cases of a more general function.
- The more general function takes extra arguments that express the differences.
- The arguments "specialize" the function.
- Must make sure that we can to specialize back to our original functions:

Confirm that the original functions can still be expressed.

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

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

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

find-elephant is now a one-liner. Yay!

What's the strategy?

```
;; STRATEGY: Use template for StringList on los
(define (find-animal los str)
  (cond
    [(empty? los) false]
    [else (or
           (string=? (first los) str)
           (find-animal (rest los) str))]))

;; STRATEGY: Call a more general function
(define (find-dog los)
  (find-animal los "dog"))
```

In this function we are still using the template

We could describe this as "call a simpler function", but it seems more accurate to describe this as calling a *more general* function

Don't get all anxious about the difference.

How to test the new definitions

- To test the new definitions, comment out the old definitions. This can be accomplished by using the Racket menu item for "comment out with semicolons".
- An entire parenthesized expression can also be commented out by prefixing it with **#;** (see the Help Desk for details).
- Do NOT use the Racket menu item "comment out in a box"—the result will be that your Racket file is converted to a form that is no longer plain text, and will not be viewable with ordinary tools (text editors, web browsers, etc.).

Your file should now look like this:

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

The old definitions are commented out

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

find-dog now refers to the new definition

Now your old tests should work WITHOUT CHANGE

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

The new definitions of **find-dog** and **find-cat** are the only ones visible, so these are now testing the new definitions.

Another Example: Pizza!

```
;; Data Definitions:

;; A Topping is a String.

;; A Pizza is represented as a list of Toppings
;; INTERP: a pizza is a list of toppings, listed from top to bottom

;; pizza-fn : Pizza -> ??
;; (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"))
```

The toppings are listed in a certain order, so we must explain the order in the interpretation.

replace-all-anchovies-with-onions

```
;; replace-all-anchovies-with-onions
;;   : Pizza -> Pizza
;; GIVEN: a 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))))]))
```

Opportunities for Generalization

We can generalize over onions to get **`replace-all-anchovies`**.

```
;; replace-all-anchovies
;;   : Pizza Topping -> Pizza
;; GIVEN: A pizza and a topping
;; RETURNS: a pizza like the given pizza, but
;; with all anchovies replaced by the given
;; topping.
```

Opportunities for Generalization

Generalize over anchovies to get **`replace-topping`**.

```
;; replace-topping
;;  : Pizza Topping Topping -> Pizza
;; GIVEN: a pizza and two toppings
;; RETURNS: a pizza like the given one, but
;; with all instances of the first topping
;; replaced by the second one.
```

Summary

- Functions will sometimes differ only in choice of data items.
- Functions can be generalized by adding new argument(s) for the differences.
- No magic here, but we will do the same thing in more interesting ways in the following lessons.
- Confirm the original functions work before generalizing.
- Test functions by renaming the originals and running the same tests.

Next Steps

- Study 06-1-1-find-dog.rkt and 06-1-2-pizza.rkt in the examples folder.
- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practice 6.1
- Go on to the next lesson.