Generalizing Over Functions

CS 5010 Program Design Paradigms
"Bootcamp"

Lesson 6.2

© Mitchell Wand, 2012-2016
This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Introduction

- In the previous lesson, we generalized over data items that were strings. In this lesson, we will see how to use the same idea to generalize over data items that are functions.
- We'll also learn about a new strategy, called *Use HOF* ("Use higher-order function")
- We'll learn how to write contracts for functions that take other functions as arguments.

Learning Objectives

- At the end of this lesson you should be able to:
  - recognize when two function definitions differ only in what functions are called at particular places in the definition
  - apply the generalization technique from Lesson 5.1 to such situations.
  - use the new strategy, called *Use HOF*
  - use **lambda** to define functions that don't need a name.
  - read and write contracts for functions that take other functions as arguments.

Example

```
;; NumberList -> NumberList
;; GIVEN: a list of numbers
;; RETURNS: a list with 1 added to each number
;; (add-1-to-each (list 11 22 33)) = (list 12 23 34)
;; STRATEGY: Use template for NumberList on lst
  (define (add-1-to-each lst)
    (cond
      [(empty? lst) empty]
      [(else (cons
               (add1 (first lst))
               (add1-to-each (rest lst))))]))
```

Here's another function definition with a similar structure

interp: salary in USD*100

```
(define-struct employee (name salary))
;; An Employee is a (make-employee String PosInt)

;; extract-names : ListOfEmployee -> ListOfString
;; GIVEN: a list of employees
;; RETURNS: the list of their names
;; STRATEGY: Use template for ListOfEmployee on loe
(define (extract-names loe)
  (cond
    [(empty? loe) empty]
    [else (cons
            (employee-name (first loe))
            (extract-names (rest loe)))]))
```

These functions only differ in one place

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

```
ListOfEmployee -> ListOfString
(define (extract-names loe)
  (cond
    [(empty? loe) empty]
    [else (cons
              (employee-name
                (first loe))
              (extract-names
                (rest loe)))]))
```

On one side, we use function **add1**, and on the other we use the function **employee-name**.

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

(define (extract-names loe)
  (apply-to-each employee-name loe))
```

So we can do the same thing we did before: we add an argument for the difference.

We recover the original functions by passing one or the other function as the value of the argument.

Let's watch this work

```
(apply-to-each add1 (cons 10 (cons 20 (cons 30 empty))))
= (cons (add1 10)
        (apply-to-each add1 (cons 20 (cons 30 empty))))
= (cons 11
        (apply-to-each add1 (cons 20 (cons 30 empty))))
= (cons 11
        (cons (add1 20)
              (apply-to-each add1 (cons 30 empty))))
= (cons 11 (cons 21 (apply-to-each add1 (cons 30 empty))))
= (cons 11 (cons 21 (cons (add1 30)
                          (apply-to-each add1 empty))))
= (cons 11 (cons 21 (cons 31 empty))))
```

Digression: Computing as algebra

- The calculation on the previous slide just used equational reasoning, like you did in Middle School algebra.
- The functional approach to programming, which we have been using now, allows us to reason about programs just using equations like these.
- This is much simpler than reasoning about programs with assignment statements.

What about the design strategy?

The definition for **apply-to-each** follows the template, so the strategy is "use template". This will work on lists of any kind of value, so we say it uses the template for **XList**.

```
;; STRATEGY: Use template for XList on lst
(define (apply-to-each fn lst)
   (cond
     [(empty? lst) empty]
     [else (cons
             (fn (first lst))
             (apply-to-each fn (rest lst)))]))
```

What about add-1-to-each and extract-names?

- Our new definitions for **`add-1-to-each`** and **`extract-names`** do not follow the template: they just *use* **`apply-to-each`**.
- We say that these functions use the strategy of *using a higher-order function (HOF)*.
- A higher-order function is simply a function where one or more of the arguments is a function, such as **add1** or **employee-name**.
- Note that this terminology is different from that in HtDP.

What about add-1-to-each and extract-names?

```
;; strategy: Use HOF apply-to-each on lst
(define (add-1-to-each lst)
   (apply-to-each add1 lst))

;; strategy: Use HOF apply-to-each on lst
 (define (extract-names lst)
   (apply-to-each employee-name lst))
```

Testing

- Testing for functions defined using higher-order function composition is just like testing we saw in the previous lesson.
- Original functions must be tested & working first
- Then write the generalized function and redefine your old functions in terms of the generalized one.
- Then comment out the old definitions, so your old tests will now see the new definitions.
- The original tests should still pass.

Doing something complicated?

- The function to be passed to **apply-to-each** is not always a built-in Racket function.
- Then just define your own:

```
(define (add5 n) (+ n 5))
(define (add-5-to-each lst)
  (apply-to-each add5 lst))
```

- Of course we'll need contracts, purpose statements, etc., for **add5**.

You can use ISL's **`local`** to do this

```
;; NumberList -> NumberList
;; GIVEN: a list of numbers
;; RETURNS: a list like the given one,
;; but with 5 added to each number.
;; STRATEGY: Use HOF apply-to-each
;;           on lst
(define (add-5-to-each lst)
    (local
      ;; add5 : Number -> Number
      ;; RETURNS: its argument + 5
      ((define (add5 n) (+ n 5)))
      (apply-to-each add5 lst)))
```

In ISL, **local** allows you to create local definitions. See HtDP2, sec 18.2.

Must provide contract and purpose statement for the local function.

Lambda can be used to define a function without giving it a name.

```
(define (add-5-to-each lst)
  (apply-to-each
    ;; Number -> Number
    ;; RETURNS: its argument + 5
    (lambda (n) (+ n 5))
    lst))
```

A function that adds 5 to its argument

If you write a function using **lambda**, you still need a contract and purpose statement.

Let's stop and talk about `lambda` for a minute

- 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 (n) (+ n 5))`** is a function that adds 5 to its argument.
- **`(apply-to-all (lambda (n) (+ n 5)) lst)`** returns a list like **`lst`**, but with 5 added to each element.

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

These two are the same:

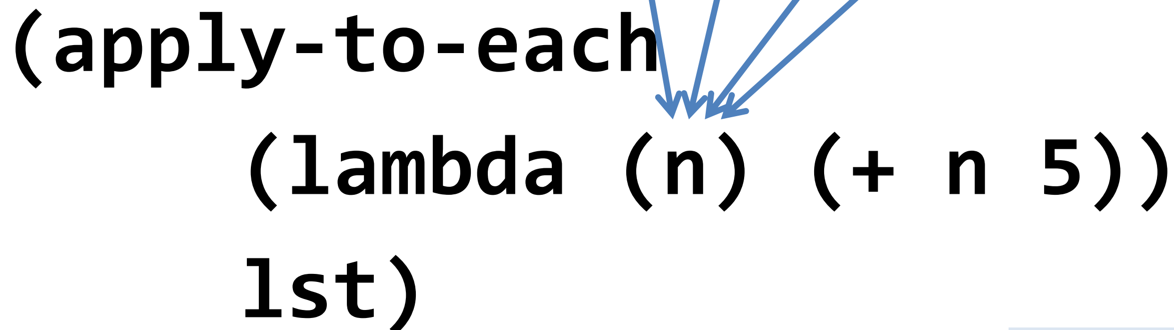
```
(local
  ((define (add5 n) (+ n 5)))
  (apply-to-all add5 lst))

(apply-to-all (lambda (n) (+ n 5)) lst)
```

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

Back to our example: where does the value of **n** come from?

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




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

apply-to-each applies the lambda-function to each element of the list in turn. Here, **n** takes on the value of each element of the list.

Opportunity for more generalization

- The 5 is a constant, so it can be generalized on by replacing it with a new argument **x**.
- Example:

```
(add-x-to-each (list 10 20 30) 7)
  = (list 17 27 37)
```

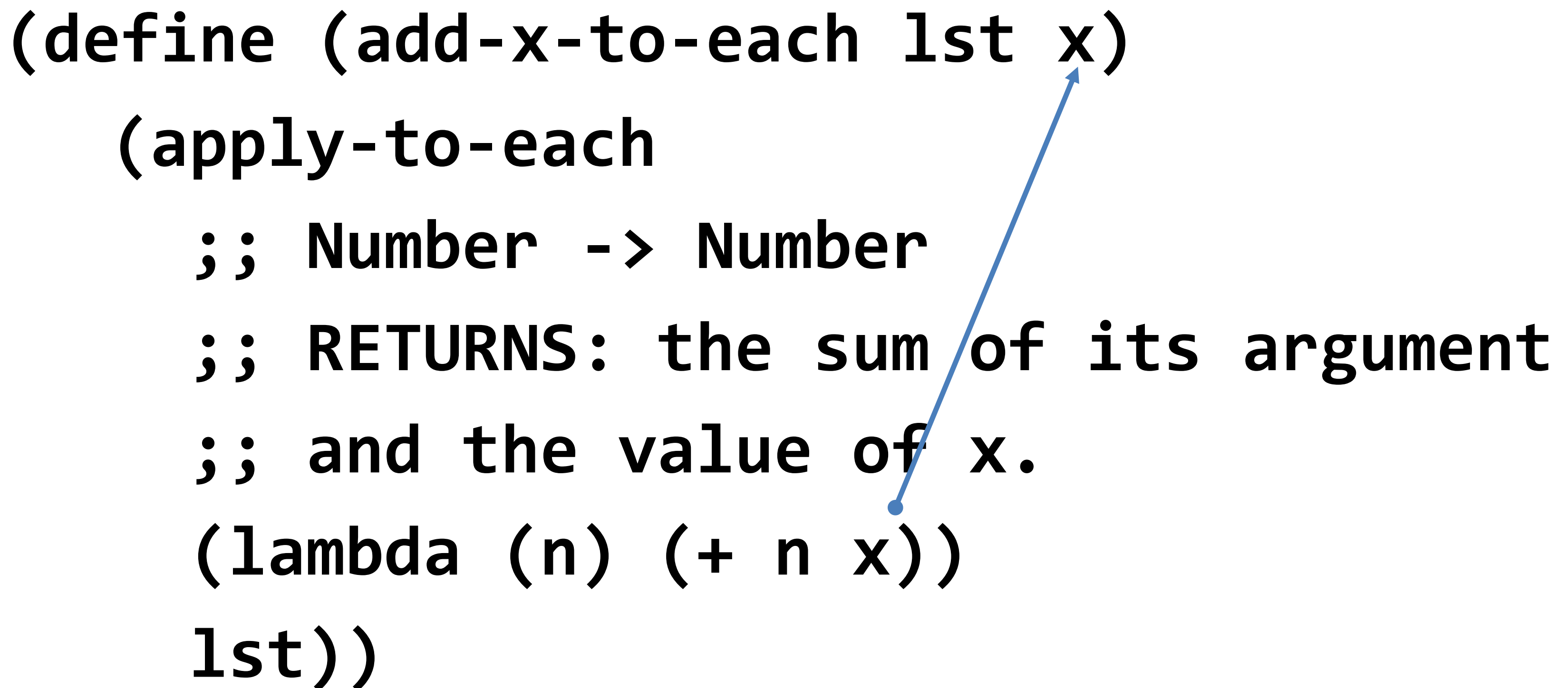
- We'll replace the local function **add5** by a new function called **addx**, which adds **x** to its argument to its argument **n**.

Here's the definition

```
;; add-x-to-each
;;  : NumberList Number -> NumberList
;; GIVEN: a list of numbers and a number
;; RETURNS: a list of numbers like the
;; given one, except that the given
;; number is added to each element of the
;; list.
;; STRATEGY: Use HOF apply-to-each on lst
(define (add-x-to-each lst x)
    (local ((define (addx n) (+ n x)))
      (apply-to-each addx lst)))
```

The “**x**” in **(+ n x)** refers the “**x**” in the argument.

As before, **lambda** can be used in order to avoid having to introduce a local name



What is the contract for apply-to-each?

- Here are two examples of the use of **apply-to-each**.

```
(apply-to-each add1 lst)
(apply-to-each employee-name loe)
```

- Each use can be described as follows: apply-to-each takes a function from **X**'s to **Y**'s, and a list of **X**'s, and it returns a list of **Y**'s
- In the first example **X** is Number and **Y** is also Number.
- In the second example, **X** is Employee and **Y** is String.

What is the contract for apply-to-each?

- We observed that apply-to-each takes a function from **X**'s to **Y**'s, and a list of **X**'s, and it returns a list of **Y**'s
- We write this down as a contract as follows:

```
apply-to-each :
   (X->Y) XList -> YList
```

Understanding this contract (1)

```
apply-to-each :
    (X->Y) XList -> YList
```

- Here there is something new: one of the arguments is a function, so the contract specifies the contract for that function: the first argument of **apply-to-each** must itself be a function that takes an **X** and returns a **Y**. We write this using the notation **(X->Y)**.
- Can't use any old function as the first argument– couldn't use +, for example.

Understanding this contract (2)

```
apply-to-each :
   (X->Y) XList -> YList
```

- The X and Y mean that this function works for any choice of X and Y.
- For example, we could use **`apply-to-each`** as

```
(Number -> Number) NumberList
  -> NumberList
```

or as

```
(Employee -> String) ListOfEmployee
  -> ListOfString
```

We say that a function with a contract like this "polymorphic"

Let's call this by its correct name

- The standard name of **`apply-to-each`** is **`map`**.
- That's what we'll call it from now on.

Higher-Order Functions FTW

- Now that we have higher-order functions, we can compose functions more easily. Example:

```
;; STRATEGY: Use HOF map on lst
;;   (twice)
(define (sqr-plus-one lst)
  (map add1 (map sqr lst)))

(sqr-plus-one (list 2 3 4))
  = (list 5 10 17)
```

One-Pass vs Multi-Pass functions

Here are two versions of **sqr-plus-one:**

```
(define (sqr-plus-one lst)
  (map add1 (map sqr lst)))

(define (sqr-plus-one lst)
  (map
    (lambda (n) (+ 1 (sqr n)))
    lst))

(sqr-plus-one (list 2 3 4))
  = (list 5 10 17)
```

Hand simulate each of these functions, like we did for (map add1 ...) back on slide 7.

The first version makes TWO passes through the argument. The second version goes through the argument only once.

Which of these is clearer? Which might be more efficient if the list is long?

Summary

- At the end of this lesson you should be able to:
  - recognize when two function definitions differ only in what functions are called at particular places in the definition
  - apply the generalization technique from Lesson 5.1 to such situations.
  - use the new strategy, called *Use HOF*
  - use **lambda** to define functions that don't need a name.
  - read and write contracts for functions that take other functions as arguments.

Next Steps

- Study 06-2-1-map.rkt in the examples folder
- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practice 6.2
- Go on to the next lesson