

Higher-Order Procedures, Revisited

Summary: Over the past few class sessions, we have visited a variety of important concepts in the context of our exploration of the GIMP and Script-Fu. Here we revisit a group of those concepts in brief. The concepts fall under the rubric of *higher-order programming*, a term we will explore further in the reading.

Contents:

- Introduction
- Guiding Principles
- Procedures as Parameters
- Anonymous Procedures
- Procedures as Return Values

Introduction

In the past few class sessions, we have explored a variety of techniques for drawing interesting images in the GIMP, including (a) randomness (b) color grids; (c) image transformations, and (d) representing drawings as sequences of points. In considering each of these topics, we also looked at some ways of representing the algorithmic concepts we wanted to express. Many of those ideas are important programming techniques, particularly in the *functional programming paradigm* that this course emphasizes.

From my perspective, there are three central concepts: procedures can take other procedures as parameters, procedures can be anonymous, and procedures can return other procedures as values. These three concepts, taken together, are generally referred to as *higher-order programming*, because procedures (which normally operate on data) now act as the data for other procedures.

We also covered a few key higher-order procedures that should be in every Scheme programmer's toolkit: `compose` (also called `o`), `left-section` and, implicitly, `right-section` (also called `l-s` and `r-s`), `map`, and `list-of?`.

In this reading, we revisit the key ideas of higher order programming and these toolkit procedures.

Guiding Principles

Over the years, programmers, particularly Scheme programmers have looked for ways to make their programs more concise and more elegant. Each programmer probably has his her or her own set of guiding principles, but there are three core guiding principles that most programmers abide by.

Write less, not more. In particular, find way to express algorithms and concepts as concisely as possible. Sometimes this will mean using features of the language that support concision, sometimes it will mean thinking about the problem (or the solution) in different ways, sometimes it requires learning a new way of programming (which is what we're starting to do).

Refactor. Programmers use the term “refactor” in a number of different ways. One of the most common is to follow the principles that if two chunks of code look the same (or nearly the same), then write a separate procedure that contains the common code. Doing so can reduce the size of your code (after all, you’ve it now appears once, rather than twice), ease updates (when you realize you got something a bit wrong, there’s only one procedure to change, rather than two or more), and makes it easier to write the next similar chunk of code.

Name appropriately: Choose good names for values, and don’t name values for which there is no real benefit in naming (e.g., values that only get used once and are intermediate to another computation). For example, in computing $(a^2+5)*3$, we don’t need to name the intermediate a^2 in our computation.

It turns out that higher-order programming can help with each of the three principles.

Procedures as Parameters

One interesting aspect of Scheme (as compared to many other languages) is that you can write procedures that take other procedures as parameters. We first explored this concept in the `color-grid` exercise and the corresponding homework assignment. As you may recall from both, we can think of this in terms of the procedure `draw-one-point`, which we might write as follows.

```
(define draw-one-point
  (lambda (image x y redfunc greenfunc bluefunc)
    (set-fgcolor (list (mod (redfunc x y) 256)
                      (mod (greenfunc x y) 256)
                      (mod (bluefunc x y) 256))))
  (blot image x y)))
```

As you may recall, `redfunc`, `greenfunc`, and `bluefunc` can be any procedures that take two integers as parameters and return a number as a result. For example,

```
(define func1
  (lambda (x y)
    (+ x y)))
(define func2
  (lambda (x y)
    (* x 3)))
(draw-one-point sample 10 10 func1 func2 func1)
```

Note here that because `redfunc` and its peers are procedures, we can use them in the “procedure position”, immediately after an open paren. In this case, when `func1` serves as `redfunc`, we add the `x` and `y` coordinates; when `func2` serves as `greenfunc`, we multiply the `x` coordinate by 3.

We returned most clearly to the idea of procedures as parameters in the reading on drawings as series of points, when we visited the `map` procedure, which applies another procedure to each element of a list, and can be defined as follows.

```
(define map
  (lambda (proc lst)
    (if (null? lst)
        null
        (cons (proc (car lst)) (map proc (cdr lst))))))
```

Why does this make our programs better? The advantage here is primarily in terms of refactoring. Particularly with `map`, we've written a lot of procedures that step through a list, doing something to each value in the list. In this case, we've written more or less the same structure. Now, we can write much less in each case. For example, if we have a list of grades and decide to give everyone five extra points, we can now write

```
(define add-five (lambda (x) (+ 5 x)))
(define grades (map add-five grades))
```

rather than

```
(define add-five-to-each
  (lambda (lst)
    (if (null? lst)
        lst
        (cons (* 5 (car lst)) (add-five-to-each (cdr lst)))))))
```

This is certainly more concise. If you think about later writing a procedure that multiplies each grade by 110% (or by 90% for a particularly unpleasant class), we save more and more each time.

Returning to the image example, we certainly could have written a procedure that used specific computations at each points, as in the following diagonal drawing procedure.

```
(define draw-diagonal
  (lambda (image width height)
    (letrec ((kernel (lambda (percent)
                      (if (<= percent 1.0)
                          (let ((x (* percent width))
                                (y (* percent height)))
                            (set-fgcolor (list (mod (+ x y) 256)
                                                (mod (* x 3) 256)
                                                (mod (+ x y) 256)))
                            (draw-one-point image
                                             (blot image x y)
                                             (kernel (+ percent .10)))))))
            (kernel 0))))))
```

However, in order to change the technique for generating the color, we have to change the body of `draw-diagonal` (or make a copy and change that copy), rather than calling it slightly differently.

Anonymous Procedures

Particularly with things like `color-grid`, we often wanted to define short procedures to test. Naming them was inconvenient, since we had to go through a write->save->load->try process. Naming them was also a bit silly, since they had no natural name, and naming them did not clarify anything.

As you may recall, we found that we could instead write the lambda expression without the `define`.

```
(color-grid 100 100 10 10 (lambda (x y) ...) (lambda (x y) ...) (lambda (x y) ...))
```

We call such procedures *anonymous procedures*. Anonymous procedures are particularly useful in conjunction with procedures like `map` (or, alternately, procedures like `map` benefit from anonymous procedures). For example, with our old problem of adding five to each grade, we can write even more concise instructions.

```
(define grades (map (lambda (x) (+ 5 x)) grades))
```

Procedures as Return Values

It turns out that `lambda` is not the only way to create anonymous procedures. We found that we could write procedures that created new procedures, and then use those. For example, consider the `redder` procedure from the reading and lab on image manipulation.

```
(define redder
  (lambda (amt)
    (lambda (color)
      (rgb ...))))
```

Think of this as “a procedure that takes *amt* as a parameter, and returns a new procedure that takes *color* as a parameter. That new procedure generates a color (although, for this discussion, the particulars of how don’t really matter)”

We also wrote some more general higher-order procedures, including `compose` (also called `o`), `left-section` (also called `l-s`), and `right-section` (also called `r-s`). These are defined as follows.

```
;;; Procedures:
;;; left-section
;;; l-s
;;; Parameters:
;;; binproc, a two-parameter procedure
;;; left, a value
;;; Purpose:
;;; Creates a one-parameter procedure by filling in the first parameter
;;; of binproc.
;;; Produces:
;;; unproc, a one-parameter procedure
;;; Preconditions:
;;; left is a valid first parameter for binproc.
;;; Postconditions:
;;; (unproc right) = (binproc left right)
(define left-section
  (lambda (binproc left)
    (lambda (right) (binproc left right))))
(define l-s left-section)

;;; Procedures:
;;; right-section
;;; r-s
;;; Parameters:
;;; binproc, a two-parameter procedure
;;; right, a value
;;; Purpose:
```

```

;;; Creates a one-parameter procedure by filling in the second parameter
;; of binproc.
;;; Produces:
;;; unproc, a one-parameter procedure
;;; Preconditions:
;;; left is a valid first parameter for binproc.
;;; Postconditions:
;;; (unproc left) = (binproc left right)
(define right-section
  (lambda (binproc right)
    (lambda (left) (binproc left right))))
(define r-s right-section)

;;; Procedures:
;;; compose
;;; o
;;; Parameters:
;;; f, a procedure
;;; g, a procedure
;;; Purpose:
;;; Compose f and g.
;;; Produces:
;;; fog, a procedure
;;; Preconditions:
;;; f can be applied to the results returned by g.
;;; Postconditions:
;;; (fog x) = (f (g x))
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))

```

Again, these make our code even more concise (and, some would say clearer). For example, consider once again the problem of adding 5 to every grade in a grades. We can now write

```
(define grades (map (l-s + 5) grades))
```

Consider also the problem of dividing every grade by .87 (perhaps to scale grades) and then adding 3 (perhaps to compensate for student stress). Before we learned about higher-order programming, we would have written something like the following.

```
(define fixgrade (lambda (grade) (+ 3 (/ grade .87))))
(define fixgrades
  (lambda (lst)
    (if (null? lst)
        null
        (cons (fixgrade (car lst)) (fixgrades (cdr lst))))))
(define grades (fixgrades grades))

```

If we incorporate the new idea of using map, we can do without the separate fixgrades procedure.

```
(define fixgrade (lambda (grade) (+ 3 (/ grade .87))))
(define grades (map fixgrade grades))

```

If we incorporate the new idea of anonymous procedures, we can do without naming `fixgrade`, which we only use once anyway.

```
(define grades (map (lambda (grade) (+ 3 (/ grade .87))) grades))
```

If we incorporate the three new procedures we've just learned, we can make this even more concise. (If it's less readable to you know, wait a bit and you'll start to learn to read this way.)

```
(define grades (map (o (1-s + 3) (r-s / .87)) grades))
```

An experienced Scheme programmer might read

- `(map ... grades)` as *for each element of grades*;
- `(o foo bar)` as *foo and then bar*;
- `(1-s + x)` as *add x*; and
- `(r-s / y)` as *divide by y*.

Putting this all together, we read it as *For each element of grades, divide by .87 and then add 3.*

Copyright © 2007 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.