Class 41: Higher-Order Procedures, Summarized

Held: Monday, April 16, 2007

Summary: Today we revisit some of the important behind-the-scenes issues that last week’s readings covered.

Related Pages:

- EBoard.
- Reading: Higher-Order Procedures.

Overview:

- Background: Guiding Principles.
- Background: Writing Similar Code.
- Procedures as Parameters.
- Anonymous Procedures.
- Procedures as Return Values.
- Encapsulating Control.
- Final Thoughts.

Background: Guiding Principles

- Write less, not more
- Refactor
- Name appropriately
  - Good names for things that need names
  - No names for things that don’t
  - Example: Don’t name the components in

    (define hyp (lambda (a b) (sqrt (+ (* a a) (* b b)))))

Background: A Related Philosophy

- The first time you read a new procedure structure (such as recursion over a list), you learn something.
- The second time you read the same structure, you learn something else.
- The third time, you learn a bit more.
- After that, reading doesn’t give much benefit.
- The first time you write the same structure, you learn something more about that structure
- The second time, you learn even more.
- The third time, you learn a bit more.
- After that, there’s no benefit.
So ... extract the common code so you don’t have to write it again. d yes, you learn something

Two Motivating Examples

- all-real? and all-integer?
- add-5-to-each and multiply-each-by-5

Procedures as Parameters

- First explored in the color-grid exercise.
- Useful
- Concise
- Supports refactoring

Anonymous Procedures

- Sometimes we don’t even need to bother to define procedures (just like we don’t define the parts of a compound expression).
- Strategy: Just use (lambda (params) body)
- We call such procedures anonymous.

Procedures as Return Values

- Another way to create procedures (anonymous and named).
- Strategy: Write procedures that return new procedures.
- These procedures can take plain values as parameters:

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

- How to think about this:
  - a procedure that takes amt as a parameter,
  - returns a new procedure that takes color as a parameter
- Can also take procedures as parameters
- One favorite: compose

  (define compose
   (lambda (f g)
      (lambda (x)
         (f (g x)))))

- Examples
  - sine of square root of x: (compose sin sqrt)
  - last element of a list: (compose car reverse)
• Another: left-section

(define left-section
  (lambda (func left)
    (lambda (right)
      (func left right))))
(define l-s left-section)

• Examples:
  ○ add two: (l-s + 2)
  ○ double: (l-s * 2)

• Not mentioned in the reading, but there’s a corresponding right-section

(define right-section
  (lambda (func right)
    (lambda (left)
      (func left right))))
(define r-s right-section)

• If we were confident with this procedure, we could use it in the exam

(define smokes? (r-s vector-ref 3))

Encapsulating Control

• Possible for complex common code, too (particularly control).

• Sample: Whoops ... no one got problem 2 right. Perhaps I should just scale each grade by 4/3.

(define scale-grades
  (lambda (grades)
    (if (null? grades)
        null
        (cons (* 4/3 (car grades))
          (scale-grades (cdr grades))))))
(define fixed-grades (scale-grades original-grades))

• Another sample: Oh yeah, everyone gets seven points of extra credit

(define extra-credit
  (lambda (grades)
    (if (null? grades)
        null
        (cons (+ 7 (car grades))
          (extra-credit (cdr grades))))))

• The common code.

(define map
  (lambda (fun lst)
    (if (null? lst)
        null
        (cons (fun (car lst))
          (map fun (cdr lst))))))
• Rewriting ...

(define scale-grades
  (lambda (grades)
    (map (lambda (grade) (* 4/3 grade)) grades)))

(define extra-credit
  (lambda (grades)
    (map (lambda (grade) (+ 7 grade)) grades)))

• We can simplify the lambda with \texttt{l-s}

(define scale-grades
  (lambda (grades)
    (map (l-s * 4/3) grades)))

(define extra-credit
  (lambda (grades)
    (map (l-s + 7) grades)))

• Or even more concisely
  \begin{itemize}
    \item (define scale-grades (l-s map (l-s * 4/3)))
    \item (define extra-credit (l-s map (l-s + 2)))
  \end{itemize}

• Another issue: Checking the type of elements in a list

(define list-of-numbers?
  (lambda (lst)
    (or (null? lst)
      (and (pair? lst)
        (real? (car lst))
        (list-of-numbers? (cdr lst)))))

(define list-of-symbols?
  (lambda (lst)
    (or (null? lst)
      (and (pair? lst)
        (symbol? (car lst))
        (list-of-symbols? (cdr lst)))))

• Common code

(define list-of?
  (lambda (test? lst)
    (or (null? lst)
      (and (pair? lst)
        (test? (car lst))
        (list-of? test? (cdr lst)))))

• Useful on the exam:

(define valid-form?
  (lambda (val) (and (pair? val) (string? (car val)) (integer? (cdr val)))))

(define all-valid?
  (lambda (lst) (list-of? valid-form? lst)))

• Or
(define all-valid? (l-s list-of? valid-form?))

Or

(define all-valid? (l-s list-of? (lambda (val) (and (pair? val) (string? (car val)) (integer? (cdr val))))))

Concluding Comments

- Yes, skilled Scheme programmers write this way.
  - It’s quick.
  - It’s clear (at least to skilled Schemers).
  - It reduces mistakes.
- Such control The ability to encapsulate control in this way is fairly unique to Scheme,
- It’s one of the reasons we love it at Grinnell.