Class 37: Higher-Order Procedures, Summarized

Held: Friday, 3 November 2006

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

Related Pages:
- EBoard.
- Code examples from class

Due
- Exam 2.

Notes:
- EC: Drag Extravaganza Friday, Football Saturday.
- No reading for the weekend; exam to be returned Monday.
- HW for Tuesday!
- Since many of you did not understand the “Trogdor!” references, I’m going to try to show [the background video]. You can also read some [background information].

Overview:
- Background: Guiding Principles.
- Procedures as Parameters.
- Anonymous Procedures.
- Procedures as Return Values.
- Encapsulating Control.

Background: Guiding Principles

- **Write less, not more**
- **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))))

- **Refactor**
Procedures as Parameters

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

- Consider problem from exam. Some wrote
  
  \[
  \text{(define count-smoking} \quad \lambda (s_1 \ s_2) \ (\text{if} \ (\text{equal?} \ (\text{smokes?} \ s_1) \ (\text{smokes} \ s_2)) \ 1 \ 0))
  \]
  
  \[
  \text{(define count-bedtime} \quad \lambda (s_1 \ s_2) \ (\text{if} \ (\text{equal?} \ (\text{get-bedtime} \ s_1) \ (\text{get-bedtime} \ s_2)) \ 1 \ 0))
  \]

- Incredibly similar. Factor out the test
  
  \[
  \text{(define count-attribute} \quad \lambda (\text{get-attribute}) \ \lambda (s_1 \ s_2) \ (\text{if} \ (\text{equal?} \ (\text{get-attribute} \ s_1) \ (\text{get-attribute} \ s_2)) \ 1 \ 0))
  \]

- Now we can define the previous two in terms of this
  
  \[
  \text{(define count-smoking} \quad \text{(count-attribute smokes?)})
  \]
  
  \[
  \text{(define count-bedtime} \quad \text{(count-attribute get-bedtime)})
  \]

- In fact, we can even do without them
  
  \[
  \text{(if} \ (\geq \ (+ \ ((\text{count-attribute} \ \text{smokes?))} \ s_1 \ s_2) \ ((\text{count-attribute} \ \text{get-bedtime}) \ s_1 \ s_2) ... \ 3) \ ...)
  \]

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 \(\lambda (\text{params}) \ \text{body}\)
- 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:

\[
\text{(define redder} \quad \lambda (\text{amt}) \ \lambda (\text{color}) \ \text{(rgb ...))})
\]

- How to think about this:
  
  - a procedure that takes \text{amt} as a parameter,
  - returns a new procedure that takes \text{color} as a parameter

- Can also take procedures as parameters
- One favorite: \text{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))))

- 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))))

- 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))))))
● A 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.
   ○ And yes, you learn something

● Applied above:

```
(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 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
  ○ (define scale-grades (l-s map (l-s * 4/3)))
  ○ (define extra-credit (l-s map (l-s + 2)))

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

• Common code

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

• Useful on the exam:

(defun valid-form?
  (lambda (val) (and (pair? val) (string? (car val)) (integer? (cdr val))))))
(defun all-valid?
  (lambda (lst) (list-of? valid-form? lst)))

• Or

(defun all-valid? (l-s list-of? valid-form?))

Or

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

• 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.

Copyright © 2006 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.