Topic 5.5
Higher Order Procedures
(This goes back and picks up section 1.3 and then sections in Chapter 2)

September 2008

Procedural Abstraction

- We have seen the use of procedures as abstractions.
- So far we have defined cases where the abstractions that are captured are essentially compound operations on numbers.
- What does that buy us?
  - Assign a name to a common pattern (e.g., cube) and then we can work with the abstraction instead of the individual operations.
- What more could we do?
  - What about the ability to capture higher-level "programming" patterns.
  - For this we need procedures are arguments/return values from procedures

The really big idea

- Procedures (function) should be treated as first-class objects
- In scheme procedures (functions) are *data*
  - can be passed to other procedures as arguments
  - can be created inside procedures
  - can be returned from procedures
- This notion provides big increase in abstractive power
- One thing that sets scheme apart from most other programming languages

Section 1.3 -Terminology

- Procedures that accept other procedures as input or return a procedure as output are **higher-order procedures.**
- **The other procedures are first-order procedures.**

- **Scheme treats functions/procedures as first-class objects. They can be manipulated like any other object.**

Book and Here…

- Book goes through showing several examples of the abstract pattern of summation, and then shows how you might want to abstract that into a procedure.
- CAUTION: I find some of the names that they use for their abstraction confusing – don't let that bother you! It just makes reading the book a little more difficult.
- I am going to borrow an introduction from some old slides from Cal-Tech. I think you should be able to put the two together very nicely.
- At least, that's my intention…

In mathematics…

- Not all operations take in (only) numbers

- +, -, *, /, expt, log, mod, …
  - take in numbers, return numbers

- but operations like Σ, d/dx, integration
  - take in functions
  - return functions (or numbers)

Math: Functions as Arguments

- You've seen:

$$a = \sum_{n=0}^{6} f(n)$$

a=f(0)+f(1)+f(2)+f(3)+f(4)+f(5)+f(6)

Math: Functions as Arguments

- Σ is a "function"
  - which takes in
    - a function
    - a lower bound (an integer)
    - an upper bound (also an integer)
  - and returns
    - a number

$$\sum_{x=0}^{6} f(x)$$

- We say that Σ is a "higher-order" function
- Can define higher-order fns in scheme

Transforming summation

$$\sum_{x=low}^{high} f(x)$$

is the same as…

$$f(low) + \sum_{x=low+1}^{high} f(x)$$

Summation in scheme

$$f(low) + \sum_{x=low+1}^{high} (f(x))$$

```
; takes a function a low value and a high value
; returns the sum of f(low)...f(high) by incrementing
; by 1 each time
(define (sum f low high)
  (if (> low high) 0
     (+ (f low)
        (sum f (+ low 1) high))))
```


Evaluating summation

- Evaluate: (sum square 2 4)
- ((lambda (f low high) …) square 2 4)
- substitute:
  - square for f
  - 2 for low, 4 for high

…continuing evaluation

- (if (> 2 4) 0
  (+ (square 2) (sum square 3 4)))
- (+ (square 2) (sum square 3 4)))
- (square 2) … 4
- (+ 4 (sum square 3 4)))

…continuing evaluation

- (+ 4 (sum square 3 4)))
- (+ 4 (if (> 3 4) 0
  (+ (square 3)
  (sum square 4 4))))
- (+ 4 (+ (square 3)
  (sum square 4 4))))
- (+ 4 (+ 9 (sum square 4 4))))

…continuing evaluation

- (+ 4 (+ 9 (sum square 4 4))))
- yadda yadda…
- (+ 4 (+ 9 (+ 16 (sum square 5 4))))
- (+ 4 (+ 9 (+ 16 (if (> 5 4) 0 …)
- (+ 4 (+ 9 (+ 16 0)))
- … 29 (whew!)
- pop quiz: what kind of process?
  - linear recursive

Also valid…

(sum (lambda (x) (* x x)) 2 4)

- this is also a valid call
- equivalent in this case
- no need to give the function a name

Iterative version

- sum generates a recursive process
- iterative process would use less space
  - no pending operations
- Can we re-write to get an iterative version?

Iterative version

```
; takes a function a low value and a high value
; returns the sum of f(low)...f(high) by incrementing
; by 1 each time
(define (isum f low high)
   (sum-iter f low high 0))

(define (sum-iter f low high result)
   (if (> low high) result
      (sum-iter f (+ low 1) high (+ (f low) result))))
```


Evaluating iterative version

- (isum square 2 4)
- (sum-iter square 2 4 0)
- (if (> 2 4) 0
  (sum-iter square (+ 2 1) 4 (+ (square 2) 0)))
- (sum-iter square (+ 2 1) 4 (+ (square 2) 0))
- (sum-iter square 3 4 (+ 4 0))
- (sum-iter square 3 4 4)

eval iterative sum cont'd...

- (sum-iter square 3 4 4)
- (if (> 3 4) 4
  (sum-iter square (+ 3 1) 4 (+ (square 3) 4)))
- (sum-iter square (+ 3 1) 4 (+ (square 3) 4))
- (sum-iter square 4 4 (+ 9 4))
- (sum-iter square 4 4 13)

eval iterative sum cont'd...

- (sum-iter square 4 4 13)
- (if (> 4 4) 13
  (sum-iter square (+ 4 1) 4 (+ (square 4) 13)))
- (sum-iter square (+ 4 1) 4 (+ (square 4) 13))
- (sum-iter square 5 4 (+ 16 13))
- (sum-iter square 5 4 29)

eval iterative sum cont'd...

- (sum-iter square 5 4 29)
- (if (> 5 4) 29 (sum-iter ...))
- 29
- same result, no pending operations
- more space-efficient

recursive vs. iterative

```
(define (sum f low high)
  (if (> low high)
   0
   (+ (f low)
      (sum f
        (+ low 1) high))))
```

```
(define (isum f low high)
  (define (sum-iter f low high result)
    (if (> low high)
      result
      (sum-iter f
        (+ low 1) high
        (+ (f low) result))))
  (sum-iter f a b 0))
```


recursive vs. iterative

- recursive:
  - pending computations
  - when recursive calls return, still work to do
- iterative:
  - current state of computation stored in operands of internal procedure
  - when recursive calls return, no more work to do ("tail recursive")

Historical interlude

Reactions on first seeing "lambda":

- What the heck is this thing?
- What the heck is it good for?
- Where the heck does it come from?

This represents the essence of a function – no need to give it a name. It comes from mathematics. Where ever you might use the name of a procedure – you could use a lambda expression and not bother to give the procedure a name.

Generalizing summation

- What if we don't want to go up by 1?
- Supply *another* procedure
  - given current value, finds the next one

```
; takes a function, a low value, a function to generate the
   next
; value and the high value.  Returns f(low)...f(high) by
; incrementing according to next each time
(define (gsum f low next high)
   (if (> low high) 0
     (+ (f low)
        (gsum f (next low) next high))))
```


stepping by 1, 2, ...

```
; takes a number and increments it by 1
(define (step1 n) (+ n 1))

; new definition of sum...
(define (new-sum f low high) ; same as before
      (gsum f low step1 high))

; takes a number and increments it by 2
(define (step2 n) (+ n 2))

; new definition of a summation that goes up by 2 each time
(define (sum2 f low high)
      (gsum f low step2 high))
```


stepping by 2

- (sum square 2 4)
  $= 2^2 + 3^2 + 4^2$
- (sum2 square 2 4)
  $= 2^2 + 4^2$
- (sum2 (lambda (n) (* n n n)) 1 10)
  $= 1^3 + 3^3 + 5^3 + 7^3 + 9^3$

using lambda

- (define (step2 n) (+ n 2))
- (define (sum2 f low high)
     (gsum f low step2 high))
- Why not just write this as:
- (define (sum2 f low high)
     (gsum f low (lambda (n) (+ n 2)) high))
- don't need to name tiny one-shot functions

(ab)using lambda

- How about:
  - sum of n^4 for n = 1 to 100, stepping by 5?
- (gsum (lambda (n) (* n n n n))
       1
       (lambda (n) (+ n 5))
       100)
- NOTE: the n's in the lambdas are independent of each other

Big Ideas

- Procedures (functions) are data!
- We can abstract operations around functions as well as numbers
- Provides great power
  - expression, abstraction
  - high-level formulation of techniques
- We've only scratched the surface!

Procedures without names

- `(lambda (<param1> <param2> ...)`
  `<body>)`
- `(define (square x) (* x x))`
- `(define square`
  `(lambda (x) (* x x)))`
- `lambda = create-procedure`

Procedures are first-class objects

- Can be the value of variables
- Can be passed as parameters
- Can be return values of functions
- Can be included in data structures

Another Use for Lambda

- Providing "local" variables

```
(define (make-rat a b)
  (cons (/ a (gcd a b))
        (/ b (gcd a b))))

(define (make-rat a b)
  ((lambda (div)
     (cons (/ a div)
           (/ b div)))
   (gcd a b)))
```


More local variables

```
((lambda (x y) (+ (* x x)
                  (* y y)))
  5
  7)

((lambda (v1 v2 ...) <body>)
  val-for-v1
  val-for-v2
  ...)
```


The let special form ...

```
(let ((<var1> <expr1>)
      (<var2> <expr2>)
      ...)
     <body>)
```

Translates into…

```
((lambda (<var1> <var2> ...) <body>)
 <expr1>
 <expr2>
 ...)
```


Using let

```
(define (f x y)
  (let ((z (+ x y)))
       (+ z (* z z))))
```

Taking the Abstraction 1 Step Further…

- we can also construct and return *functions.*

Math: Operators as return values

- The derivative operator
  - Takes in…
    - A function
      - (from numbers to numbers)
  - Returns…
    - Another function
      - (from numbers to numbers)

$$f(x) = \frac{d}{dx}(F(x))$$

Math: Operators as return values

$$F(x) = \int f(x)dx$$

- The integration operator
  - Takes in…
    - A function
      - from numbers to numbers, and
    - A value of the function at some point
      - *E.g.* F(0) = 0
  - Returns
    - A function from numbers to numbers

Returning operators

- So operators can be return values, as well:

$$f(x) = \frac{d}{dx}(F(x))$$

$$F(x) = \int f(x)dx$$

Further motivation

- Besides mathematical operations that inherently return operators…
- …it's often nice, when designing programs, to have operations that help construct larger, more complex operations.

An example:

- Consider defining all these functions:

```
(define add1 (lambda (x) (+ x 1))
(define add2 (lambda (x) (+ x 2))
(define add3 (lambda (x) (+ x 3))
(define add4 (lambda (x) (+ x 4))
(define add5 (lambda (x) (+ x 5))
```

- …repetitive, tedious.

Avoid Needless Repetition

```
(define add1 (lambda (x) (+ x 1))
(define add2 (lambda (x) (+ x 2))
(define add3 (lambda (x) (+ x 3))
(define add4 (lambda (x) (+ x 4))
(define add5 (lambda (x) (+ x 5))
```

– Whenever we find ourselves doing something rote/repetitive… ask:
  - Is there a way to abstract this?

Abstract “Up”

- Generalize to a function that can *create* adders:

```
; function that takes a number and returns a function
; that takes a number and adds that number to the given number
(define (make-addn n)
     (lambda (x) (+ x n)))

;(define make-addn   ;; equivalent def
;       (lambda (n)
;            (lambda (x) (+ x n))))
```


How do I use it?

```
(define (make-addn n)
   (lambda (x) (+ x n)))

((make-addn 1) 3)
4

(define add3 (make-addn 3))
(define add2 (make-addn 2))
(add3 4)
7
```


Evaluating…

- (define add3 (make-addn 3))
  - Evaluate (make-addn 3)
    - Evaluate 3 -> 3.
    - Evaluate make-addn ->
      - (lambda (n) (lambda (x) (+ x n)))
    - Apply make-addn to 3…
      - Substitute 3 for n in (lambda (x) (+ x n))
      - Get (lambda (x) (+ x 3))
  - Make association:
    - add3 bound to (lambda (x) (+ x 3))

Evaluating (add3 4)

- (add3 4)
- Evaluate 4
- Evaluate add3
  - (lambda (x) (+ x 3))
- Apply (lambda (x) (+ x 3)) to 4
  - Substitute 4 for x in (+ x 3)
  - (+ 4 3)
  - 7

Big Ideas

- We can abstract operations around functions as well as numbers
- We can “compute” functions just as we can compute numbers and booleans
- Provides great power to
  - express
  - abstract
  - formulate high-level techniques