6.001 Recitation 8: Higher-Order Procedures (HOPs)

RI: Gerald Dalley, dalleyg@mit.edu, 2 Mar 2007
`http://people.csail.mit.edu/dalleyg/6.001/SP2007/`

Announcements / Notes

- Exam:
  - Thurs 8 Mar, 7:30-9:30pm
  - Conflict time: Wed 7 Mar @ 7:30pm. Contact Donna Kaufman *now* (dkauf@mit.edu) if you need to use the conflict time.
  - Room locations are posted on the course website. Attend the correct exam!
  - (Optional) LA Review Session: check the website, times to be posted soon.
- Guest lecturer next Thursday: spam fighting!
- Recitations
  - Today: Higher-Order Procedures – nuts and bolts
  - Next Wednesday: By default, we'll keep having fun with HOPs. Email me (dalleyg@mit.edu) if you'd like us to spend some time on another topic.

Rights & privileges

"First-class elements" in a language:

1. may be named by variables
2. may be passed as arguments to procedures
3. may be returned as the results of procedures
4. may be included in data structures

Numbers, cons cells, ..., and even procedures all have first class status in Scheme. This is where scheme really stands out from other languages you may know.

Side information: In C and C++, functions are actually first-class objects, but the lack of anonymous nested functions severely limits their utility. In Java, functions are not first-class objects at all. Matlab has recently been adding Scheme-like characteristics, but in a piecemeal and inconsistent manner (*e.g.* anonymous functions and named nested functions work very differently). We'll be able to write some really elegant code in 6.001 because of these key features in Scheme.

My top 3 favorite things in 6.001 and Scheme are:

1. Higher-order procedures!
2. ...
3. ...

Derivatives, more derivatives, and type reasoning

For the remainder of this recitation, we'll re-create a numeric derivative higher-order procedure, then compose a second derivative HOP, and finally generalize to n^{th} derivatives.

The content of our discussion will be posted with the solutions. Next recitation, we'll have some fun showing off the power of HOPs through usage of procedures like `map`, `filter`, and `fold-right`.

```
;; ----------------------------------------------------------------------

;; First, we'll look at a few simple procedures as a warmup for reasoning
;; about types...

(define (square x) (* x x)) ; square: number -> number
(define (negate x) (- x))   ; negate: number -> number

;; ----------------------------------------------------------------------

;; In yesterday's lecture, we saw a version of compose.  Here we'll make
;; a version that generates a new procedure instead of a value.
;; ...discussion of types...

(define compose             ; compose: (B -> C), (A -> B) -> (A -> C)
  (lambda (f g)
    (lambda (x) (f (g x)))))

;; Let's see an example of using compose.  What's the type of neg-sq?

(define neg-sq (compose negate square)) ; neg-sq: number -> number
(neg-sq 3) ; -> -9

;; ----------------------------------------------------------------------

;; In lecture we saw a derivative procedure that took a scalar function
;; and returned a new scalar function that numerically approximates the
;; derivative of the input.  Let's write it again, thinking about the
;; types.

(define epsilon 0.001)
(define deriv
  ; deriv: (number -> number) -> (number -> number)
  (lambda (f)
    (lambda (x)
      (/ (- (f (+ x epsilon))
            (f x))
         epsilon))))

;; Okay, now let's see what happens when we try to use it...

(deriv square) ; --> (number -> number)
;(deriv 1)     ; --> procedure that will always generate an error
((deriv square) 0)  ; --> ~0
((deriv square) 10) ; --> 2*10 = ~20
((deriv (lambda (x) (+ (* 3 x x x) (* -6 x x) 4))) 10) ; --> 3*3(10)^2 - 2*6(10) = 780

;; ----------------------------------------------------------------------

;; One derivative is useful, what about second derivatives?
;; An alternative to the solution below would be to use
;;    (define deriv2 (lambda (f) (deriv (deriv f))))
;; Tradeoffs?

(define deriv2
  ; deriv2: A -> C
  ;         (number -> number) -> (number -> number)
  (compose deriv deriv))

;; What's this do?

((deriv2 (lambda (x) (+ (* 3 x x x) (* -6 x x) 4))) 10) ; --> 2*3*3(10) - 2*6 = 168

;; ----------------------------------------------------------------------

;; Now, let's think about doing nth derivatives.  If we just sit down and
;; try to code this up, we're likely to get confused.  On the other hand,
;; if we first reason about the type signature of derivn, it becomes
;; much easier to write the procedure.

(define derivn
  ; derivn: (number -> number), integer -> (number -> number)
  (lambda (f n)
    (cond ((< n 0) (error "can't␣take␣negative␣derivatives"))
          ((= n 0) f)
          (else (deriv (derivn f (- n 1)))))))

;; Some tests...

(derivn square 0) ; --> number -> number
(derivn square 1000) ; --> number -> number
((derivn square 0) 10) ; --> (10)^2 = 100
((derivn square 1) 10) ; --> 2*(10) = 20
((derivn square 2) 10) ; --> 2
((derivn square 3) 10) ; --> 0
((derivn (lambda (x) (+ (* 3 x x x) (* -6 x x) 4)) 2) 10) ; --> 2*3*3(10) - 2*6 = 168

;; Here are some other implementations...

(define derivn-2
  ; derivn-2: (number -> number), integer -> (number -> number)
  (lambda (f n)
    (cond ((< n 0) (error "can't␣take␣negative␣derivatives"))
          ((= n 0) f)
          (else (derivn-2 (deriv f) (- n 1))))))

;; Q: what's the difference between derivn and derivn-2?

(define (check-similar-deriv f n x)
  (if (> (abs (- ((derivn   f n) x)
                 ((derivn-2 f n) x)))
         (* 5 epsilon))
      (error "derivn␣and␣derivn-2␣are␣giving␣different␣answers")))

(define (mypoly x)  (+ (* 3 x x x) (* -6 x x) 4))
(check-similar-deriv mypoly 0 0)
(check-similar-deriv mypoly 0 1000)
(check-similar-deriv mypoly 1 0)
(check-similar-deriv mypoly 1 1000)
(check-similar-deriv mypoly 2 0)
(check-similar-deriv mypoly 2 1000)

;; A: derivn-2 is iterative, but derivn is iterative

;; Here's an implementation suggested by one of the students in recitation...


(define derivn-3
  ; derivn: (number -> number), integer -> (number -> number)
  (lambda (f n)
    (cond ((< n 0) (error "can't␣take␣negative␣derivatives"))
          ((= n 0) f)
          (else (compose deriv (derivn-3 f (- n 1)))))))

(check-similar-deriv mypoly 0 0)
(check-similar-deriv mypoly 0 1000)
(check-similar-deriv mypoly 1 0)
(check-similar-deriv mypoly 1 1000)
(check-similar-deriv mypoly 2 0)
(check-similar-deriv mypoly 2 1000)


;; -----------------------------------------------------------------------
```

```scheme
;; some common HOP implementations, in case the issues arise... (otherwise
;; we'll wait for next recitation)

(define (map op lst)
  ; map: (A -> B), list<A> -> list<B>
  (if (null? lst)
      '()
      (cons (op (car lst))
            (map op (cdr lst)))))

(map square (list 1 2 3 4)) ; --> (1 4 9 16)

(define (filter pred lst)
  ; filter: (A -> boolean), list<A> -> list<A>
  (cond ((null? lst)      '())
        ((pred (car lst)) (cons (car lst) (filter pred (cdr lst))))
        (else             (filter pred (cdr lst)))))

(filter even? (list 1 2 3 4 5)) ; --> (2 4)

(define (fold-right op init lst)
  ; fold-right: (A -> A), A, list<A> -> A
  (if (null? lst)
      init
      (op (car lst)
          (fold-right op init (cdr lst)))))

(fold-right + 0 (list 1 2 3 4 5)) ; --> 15
```