SICP sections 3.1.2 - 3.1.3

September 27, 2007 at 09:55 **Tags** SICP

The code for this section is in Scheme.

Section 3.1.2

Exercise 3.5

Here's the code for integral estimation using the monte-carlo method. Note that I changed the definition of `random-in-range` a little, since PLT Scheme's `random` doesn't accept inexact numbers.

```
(define (random-in-range low high)
  (let ((range (- high low)))
    (+ low (* (random) range))))

(define (monte-carlo trials experiment)
  (define (iter trials-remaining trials-passed)
    (cond ((= trials-remaining 0)
           (/ trials-passed trials))
          ((experiment)
           (iter (- trials-remaining 1) (+ trials-passed 1)))
          (else
           (iter (- trials-remaining 1) trials-passed))))
  (iter trials 0))

(define (rect-area x1 x2 y1 y2)
  (abs (* (- x2 x1) (- y2 y1))))

(define (estimate-integral p x1 x2 y1 y2 n-trials)
  (let ((integral-test
         (lambda ()
           (p (random-in-range x1 x2)
              (random-in-range y1 y2)))))
    (*  (rect-area x1 x2 y1 y2)
        (monte-carlo n-trials integral-test))))
```

To estimate `pi` :

```
(define (unit-pred x y)
  (<=
    (+ (square x) (square y))
    1))

(do ((i 0 (+ i 1)))
    ((= i 10) '())
  (printf "Pi estimated: ~a~%"
    (estimate-integral unit-pred 1.0 -1.0 1.0 -1.0 100000)))
```

The results I got in one run:

```
Pi estimated: 3.14004
Pi estimated: 3.1354
Pi estimated: 3.14428
Pi estimated: 3.13584
Pi estimated: 3.14064
Pi estimated: 3.13584
Pi estimated: 3.13852
Pi estimated: 3.1366
Pi estimated: 3.14224
Pi estimated: 3.14808
```

These aren't as stable as I'd like after 100,000 iterations. The reason for the relative poorness of the results is either because of the exactness limit of Scheme numbers, or the poorness of the pseudorandom number generator, or both.

Exercise 3.6

There are two ways to go about this exercise – either use the imaginary `rand-update` function the authors refer to in the text, or write something that will surely work in a concrete Scheme implemenentation. I'll take the second way, using the random-number facilities of PLT Scheme.

```
(define (rand command)
  (case command
    ('generate (random))
    ('reset
      (lambda (new)
        (random-seed new)))
    (else
      (error "Bad command -- " command))))
```

`random-seed` is a built in function of the MzScheme language[1].

Section 3.1.3

Exercise 3.7

I think this is quite possible to achieve without actually modifying the solution to exercise 3.3:

```
(define (make-joint acc acc-pass new-pass)
  (define (proxy-dispatch password m)
    (if (eq? password new-pass)
      (acc acc-pass m)
      (error "Bad joint password -- " password)))
  proxy-dispatch)
```

We have to carefully understand where the state is stored here… `make-joint` accepts `acc` as the original account, and `proxy-dispatch` "closes over" it. The same with `acc-pass` which is the original password to the account. Then, when the proxy[2] function is called, it just checks if its password is correct and forwards the call to the original account (which, in turn, checks that *its* password is correct).

Exercise 3.8

Here's one such function. It is very contrived and file-tailored to the exercise, of course:

```
(define f
  (let ((state 1))
    (lambda (n)
      (set! state (* state n))
      state)))
```

PLT Scheme evaluates arguments left-to-right. So:

```
(+ (f 0) (f 1))
=> 0
```

To "simulate" right-to-left evaluation, I'll just change the order of the arguments to `+`:

```
(+ (f 1) (f 0))
=> 1
```

[1] MzScheme is the formal name of the Scheme dialect used inPLT Scheme. It's a superset of R5RS Scheme.

[2] This functions doesn't do anything by itself, but rather forwards all calls to another function. Such a pattern is often called *proxy*.

Back to top