SICP Exercise Solutions for Section 1.3

Paul L. Snyder

March 19, 2014

Contents

1 Section 1.3.1

1.1 Exercise 1.29: Implementing Simpson's Rule

1.1.1 Problem

Simpson's Rule is a more accurate method of numerical integration than the method illustrated above. Using Simpson's Rule, the integral of a function f between a and b is approximated as

$$\frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y + 3 + 2y_4 + \ldots + 2y_{n-2} + 4y_{n-1} + y_n$$

where $h = (b - a)/n$, for some even integer n, and\$ $\mathrm{y_k = f(a + kh)}$\$. (Increasing n increases the accuracy of the approximation.) Define a procedure that takes as arguments f, a, b, and n and returns the value of the integral, computed using Simpson's Rule. Use your procedure to integrate `cube` between 0 and 1 (with $n = 100$ and $n = 1000$), and compare the results to those of the `integral` procedure shown above.

1.1.2 Answer

```
(require (planet neil/sicp:1:17))

(define (square x) (* x x))
(define (cube x) (* x x x))

(define (inc n) (+ n 1))

(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))

(define (integral f a b dx)
  (define (add-dx x) (+ x dx))
  (* (sum f (+ a (/ dx 2.0)) add-dx b)
     dx))

(define (simpsons-integral f a b n)
  (define h (/ (- b a) n))
  (define (y k) (f (+ a (* k h))))
  (define (term-mult k)
    (cond ((or (= k 0) (= k n)) 1)
          ((odd? k) 4)
          ((even? k) 2)))
  (define (simpson-term k)
    (* (term-mult k) (y k)))
  (if (and (> n 0) (> b a) (even? n))
      (* (/ h 3) (sum simpson-term 0 inc n))
      (printf "n must be even and greater than 0, and a must be greater than b")))
```

1.2 Exercise 1.30: Iterative `sum`

1.2.1 Problem

The `sum` procedure above generates a linear recursion. The procedure can be rewritten so that the sum is performed iteratively. Show how to do this by filling in the missing expressions in the following definition:

```
(define (sum term a next b)
  (define (iter a result)
    (if <??>
        <??>
        (iter <??> <??>)))
  (iter <??> <??>))
```

1.2.2 Answer

```
(define (sum-iter term a next b)
  (define (loop a result)
    (if (> a b)
        result
        (loop (next a) (+ result (term a)))))
  (loop a 0))
```

1.3 Exercise 1.31: A `product` procedure

1.3.1 Problem

a. The `sum` procedure is only the simplest of a vast number of similar abstractions that can be captured as higher-order procedures. Write an analogous procedure called `product` that returns the product of the values of a function at points over a given range. Show how to define `factorial` in terms of `product`. Also use `product` to compute approximations to π using the formula

$$\frac{pi}{4} = \frac{2*4*4*6*6*9\ldots}{3*3*5*5*7*7\ldots}$$

b. If your `product` procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

1.3.2 Answer

```
(define (product-recur term a next b)
  (if (> a b)
      1
      (* (term a)
         (product-recur term (next a) next b))))

(define (product-iter term a next b)
  (define (loop a result)
    (if (> a b)
        result
        (loop (next a) (* result (term a)))))
  (loop a 1))

(define product product-iter)

(define (identity n) n)

(define (factorial n) (product identity 1 inc n))

(define (approx-pi n)
  (define (round-up-to-even i)
    (if (even? i) i (inc i)))
  (define (round-up-to-odd i)
    (if (odd? i) i (inc i)))
  (define (num k)
    (round-up-to-even (inc k)))
  (define (den k)
    (round-up-to-odd (inc k)))
  (define (term k)
    (/ (num k) (den k)))
  ;; Use a real number here to switch Racket's exact rationals to inexact
  ;; floating point
  (* 4.0 (product term 1 inc n)))
```

1.4 Exercise 1.32: Abstracting `accumulate`

1.4.1 Problem

a. Show that `sum` and `product` (*Note Exercise 1.31) are both special cases of a still more general notion called `accumulate` that combines a collection of terms, using some general accumulation function:

```
(accumulate combiner null-value term a next b)
```

`accumulate` takes as arguments the same term and range specifications as `sum` and `product`, together with a `combiner` procedure (of two arguments) that specifies how the current term is to be combined with the accumulation of the preceding terms and a `null-value` that specifies what base value to use when the terms run out. Write `accumulate` and show how `sum` and `product` can both be defined as simple calls to `accumulate`.

b. If your `accumulate` procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

1.4.2 Answer

```
(define (accumulate-recur combiner null-value term a next b)
  (if (> a b)
      null-value
      (combiner (term a)
                (accumulate-recur combiner null-value term (next a) next b))))

(define (accumulate-iter combiner null-value term a next b)
  (define (loop a result)
    (if (> a b)
        result
        (loop (next a) (combiner result (term a)))))
  (loop a null-value))
```

```
(define accumulate accumulate-iter)

(define (sum-acc term a next b)
  (accumulate-recur + 0 term a next b))

(define (product-acc term a next b)
  (accumulate-iter * 1 term a next b))
```

1.5 Exercise 1.33: Further generalization with `filtered-accumulate`

1.5.1 Problem

You can obtain an even more general version of `accumulate` (*Note Exercise 1.32) by introducing the notion of a "filter" on the terms to be combined. That is, combine only those terms derived from values in the range that satisfy a specified condition. The resulting `filtered-accumulate` abstraction takes the same arguments as accumulate, together with an additional predicate of one argument that specifies the filter. Write `filtered-accumulate` as a procedure. Show how to express the following using `filtered-accumulate`:

a. the sum of the squares of the prime numbers in the interval a to b (assuming that you have a `prime?` predicate already written)

b. the product of all the positive integers less than n that are relatively prime to n (i.e., all positive integers $i < n$ such that $\text{GCD}(i, n) = 1$).

1.5.2 Answer

```
(define (filtered-accumulate combiner null-value filter term a next b)
  (define (loop a result)
    (cond ((> a b) result)
          ((filter a) (loop (next a) (combiner result (term a))))
          (else (loop (next a) result))))
  (loop a null-value))

(define (square n) (* n n))

(define (smallest-divisor n)
  (find-divisor n 2))

(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))

(define (divides? a b)
  (= (remainder b a) 0))

;; Fixed this procedure to no longer classify 1 as prime
(define (prime? n)
  (and (= n (smallest-divisor n))
       (> n 1)))

(define (sum-of-squares-of-primes a b)
  (filtered-accumulate + 0 prime? square a inc b))

(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))

(define (product-of-relative-primes n)
  (define (relatively-prime? i)
    (= (gcd i n) 1))
  (filtered-accumulate * 1 relatively-prime? identity 1 inc n))
```

2 Section 1.3.2

2.1 Exercise 1.34: Perverse self-application

2.1.1 Problem

Suppose we define the procedure

```
(define (f g)
   (g 2))
```

Then we have

```
(f square)
4
```

```
(f (lambda (z) (* z (+ z 1))))
6
```

What happens if we (perversely) ask the interpreter to evaluate the combination `(f f)`? Explain.

2.1.2 Answer

It expands as:

```
(f f)
(f 2)
(2 2)
```

And it will terminate with an error, since `2` is not a procedure.

3 Section 1.3.3

3.1 TODO Exercise 1.35: The fixed point ϕ

3.1.1 Problem

Show that the golden ratio ϕ (section 1.22) is a fixed point of the transformation $x \mapsto 1 + 1/x$, and use this fact to compute ϕ by means of the `fixed-point` procedure.

3.1.2 Answer

3.2 Exercise 1.36: Observing `fixed-point` approximations

3.2.1 Problem

Modify `fixed-point` so that it prints the sequence of approximations it generates, using the `newline` and `display` primitives shown in *Note Exercise 1.22. Then find a solution to $x^x = 1000$ by finding a fixed point of $x \mapsto \log 1000/\log x$. (Use Scheme's primitive `log` procedure, which computes natural logarithms.) Compare the number of steps this takes with and without average damping. (Note that you cannot start `fixed-point` with a guess of 1, as this would cause division by $\log 1 = 0$.)

3.2.2 Answer

First, the definitions from the text:

```
(define (average x y)
  (/ (+ x y) 2))

(define (close-enough? x y)
  (< (abs (- x y)) 0.001))

(define tolerance 0.00001)

(define (fixed-point-orig f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

Now, we instrument `fixed-point` so we can observe its progress:

```
(define (fixed-point-verbose f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (display next)
      (newline)
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))

(fixed-point-verbose (lambda (x) (/ (log 1000) (log x))) 1.1)

(fixed-point-verbose (lambda (x) (average x (/ (log 1000) (log x)))) 1.1)

(prin1 "Without damping:")
```

```
(print (length (split-string normal "\n")))
(prin1 "With average damping:")
(print (length (split-string damped "\n")))
```

3.3 Exercise 1.37: Infinite continued fractions

3.3.1 Problem

a. An infinite "continued fraction" is an expression of the form

$$f = \frac{N_1}{D_1 + \frac{N_2}{D_2 + \frac{N_3}{D_3} + \cdots}}$$

As an example, one can show that the infinite continued fraction expansion with the N_i and the D_i all equal to 1 produces $1/\phi$, where ϕ is the golden ratio (described in section 1.2.2). One way to approximate an infinite continued fraction is to truncate the expansion after a given number of terms. Such a truncation—a so-called finite continued fraction "k-term finite continued fraction"—has the form

$$\frac{N_1}{D_1 + \frac{N_2}{\cdots + \frac{N_K}{D_K} + \cdots}}$$

Suppose that `n` and `d` are procedures of one argument (the term index i) that return the N_i and D_i of the terms of the continued fraction. Define a procedure `cont-frac` such that evaluating `(cont-frac n d k)` computes the value of the k-term finite continued fraction. Check your procedure by approximating $1/\phi$ using

```
(cont-frac (lambda (i) 1.0)
           (lambda (i) 1.0)
            k)
```

for successive values of `k`. How large must you make `k` in order to get an approximation that is accurate to 4 decimal places?

b. If your `cont-frac` procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

3.3.2 Answer

```
(define (cont-frac-recur n d k)
  (if (= k 1)
      (/ (n 1) (d 1))
      (/ (n k) (+ (d k) (cont-frac n d (- k 1))))))

(define (cont-frac-iter n d k)
  (define (loop i result)
    (if (= i 0)
        result
        (loop (- i 1) (/ (n i) (+ (d i) result)))))
  (loop k 0))

(define cont-frac cont-frac-iter)

(display (cont-frac (lambda (i) 1.0) (lambda (i) 1.0) 1))
(newline)
(display (cont-frac (lambda (i) 1.0) (lambda (i) 1.0) 2))
(newline)
(display (cont-frac (lambda (i) 1.0) (lambda (i) 1.0) 10))
(newline)
(display (cont-frac (lambda (i) 1.0) (lambda (i) 1.0) 11))
(newline)
(display (cont-frac-recur (lambda (i) 1.0) (lambda (i) 1.0) 11))
(newline)
```

3.4 Exercise 1.38: Euler's continued fraction for $e - 2$

3.4.1 Problem

In 1737, the Swiss mathematician Leonhard Euler published a memoir *De Fractionibus Continuis*, which included a continued fraction expansion for $e-2$, where e is the base of the natural logarithms. In this fraction, the N_i are all 1, and the D_i are successively $1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, \ldots$ Write a program that uses your `cont-frac` procedure from Exercise 1-37 to approximate e, based on Euler's expansion.

3.4.2 Answer

```
(define (euler-e k)
  (+ 2.0 (cont-frac (lambda (i) 1)
                    (lambda (i)
                      (if (= (remainder i 3) 2)
                          (* 2 (/ (+ i 1) 3))
                          1))
                   k)))
```

3.5 WRITEUP Exercise 1.39: Lambert's continued fraction for tangents

3.5.1 Problem

A continued fraction representation of the tangent function was published in 1770 by the German mathematician J.H. Lambert:

$$\tan x = \cfrac{x}{1 - \frac{x^2}{3 - \frac{x^2}{5 - \cdots}}}$$

where x is in radians. Define a procedure `(tan-cf x k)` that computes an approximation to the tangent function based on Lambert's formula. `k` specifies the number of terms to compute, as in *Note Exercise 1.37.

3.5.2 Answer

```
(define (tan-cf x k)
  (cont-frac (lambda (i)
               (if (= i 1)
                   x
                   (- (* x x))))
             (lambda (i)
               (+ 1.0 (* 2.0 (- i 1.0))))
             k))
```

The only tricky bits here are making sure that all of the N_i is negative for $i > 1$ and finding a tidy expression for the D_i.

4 Section 1.3.4

4.1 WRITEUP Exercise 1.40: Approximating cubics with Newton's method

4.1.1 Problem

Define a procedure `cubic` that can be used together with the `newtons-method` procedure in expressions of the form

```
(newtons-method (cubic a b c) 1)
```

to approximate zeros of the cubic $x^3 + ax^2 + bx + c$.

4.1.2 Answer

```
(define fixed-point fixed-point-orig)

(define (average-damp f)
  (lambda (x) (average x (f x))))

(define (sqrt x)
  (fixed-point (average-damp (lambda (y) (/ x y)))
               1.0))

(define dx 0.00001)

(define (deriv g)
  (lambda (x)
    (/ (- (g (+ x dx)) (g x))
       dx)))

(define (newton-transform g)
  (lambda (x)
    (- x (/ (g x) ((deriv g) x)))))

(define (newtons-method g guess)
  (fixed-point (newton-transform g) guess))

(define (cubic a b c)
  (lambda (x) (+ (* a (cube x)) (* b (square x)) (* c x) x)))
```

4.2 WRITEUP Exercise 1.41: Doubling `double`

4.2.1 Problem

Define a procedure `double` that takes a procedure of one argument as argument and returns a procedure that applies the original procedure twice. For example, if `inc` is a procedure that adds 1 to its argument, then `(double inc)` should be a procedure that adds 2. What value is returned by

```
(((double (double double)) inc) 5)
```

4.2.2 Answer

```
(define (double f)
  (lambda (x) (f (f x))))

(((double (double double)) inc) 5)
```

4.3 WRITEUP Exercise 1.42: Composing functions

4.3.1 Problem

Let f and g be two one-argument functions. The *composition* f after g is defined to be the function $x \mapsto f(g(x))$. Define a procedure `compose` that implements composition. For example, if `inc` is a procedure that adds 1 to its argument,

```
((compose square inc) 6)
49
```

4.3.2 Answer

- Note taken on *[2014-03-17 Mon 15:23]*

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

((compose square inc) 6)
```

4.4 WRITEUP Exercise 1.43: Repeated function application

4.4.1 Problem

If f is a numerical function and $n$ is a positive integer, then we can form the $n$th repeated application of f, which is defined to be the function whose value at x is $f(f(\ldots(f(x))\ldots))$. For example, if f is the function $x \mapsto x+1$, then the $n$th repeated application of f is the function $x \mapsto x+n$. If f is the operation of squaring a number, then the nth repeated application of f is the function that raises its argument to the 2^{n}th power. Write a procedure that takes as inputs a procedure that computes f and a positive integer $n$ and returns the procedure that computes the $n$th repeated application of f. Your procedure should be able to be used as follows:

```
((repeated square 2) 5)
625
```

Hint: You may find it convenient to use `compose` from Exercise 1-42.

4.4.2 Answer

```
(define (repeated-recur f n)
  (if (> n 1)
      (compose f (repeated-recur f (dec n)))
      f))

(define (repeated-it f n)
  (lambda (x)
    (define (repeated-loop f i x)
      (if (> i 0)
          (repeated-loop f (dec i) (f x))
          x))
    (repeated-loop f n x)))

(define repeated repeated-recur)

((repeated square 2) 5)
```

4.5 WRITEUP Exercise 1.44: Repeated smoothing

4.5.1 Problem

The idea of "smoothing" a function is an important concept in signal processing. If f is a function and dx is some small number, then the smoothed version of f is the function whose value at a point x is the average of $f(x-dx)$, $f(x)$, and $f(x+dx)$. Write a procedure `smooth` that takes as input a procedure that computes f and returns a procedure that computes the smoothed f. It is sometimes valuable to repeatedly smooth a function (that is, smooth the smoothed function, and so on) to obtained the "$n$-fold smoothed function". Show how to generate the $n$-fold smoothed function of any given function using `smooth` and `repeated` from Exercise 1.43.

4.5.2 Answer

```
(define (smooth f)
  (lambda (x)
    (average (f x) (f (- x dx)))))

(define (repeated-smooth f n)
  ((repeated smooth n) f))
```

4.6 TODO Exercise 1.45: $n$th roots with average damping

4.6.1 Problem

We saw in section 1.3.3 that attempting to compute square roots by naively finding a fixed point of $y \mapsto x/y$ does not converge, and that this can be fixed by average damping. The same method works for finding cube roots as fixed points of the average-damped $y \mapsto x/y^2$. Unfortunately, the process does not work for fourth roots—a single average damp is not enough to make a fixed-point search for $y \mapsto x/y^3$ converge. On the other hand, if we average damp twice (i.e., use the average damp of the average damp of $y \mapsto x/y^3$) the fixed-point search does converge. Do some experiments to determine how many average damps are required to compute $n$th roots as a fixed-point search based upon repeated average damping of $y \mapsto x/y^(n-1)$. Use this to implement a simple procedure for computing $n$th roots using `fixed-point`, `average-damp`, and the `repeated` procedure of Exercise 1-43. Assume that any arithmetic operations you need are available as primitives.

4.6.2 Answer

4.7 TODO Exercise 1.46: Generalizing iterative improvement

4.7.1 Problem

Several of the numerical methods described in this chapter are instances of an extremely general computational strategy known as "iterative improvement". Iterative improvement says that, to compute something, we start with an initial guess for the answer, test if the guess is good enough, and otherwise improve the guess and continue the process using the improved guess as the new guess. Write a procedure `iterative-improve` that takes two procedures as arguments: a method for telling whether a guess is good enough and a method for improving a guess. `iterative-improve` should return as its value a procedure that takes a guess as argument and keeps improving the guess until it is good enough. Rewrite the `sqrt` procedure of section 1.1.7 and the `fixed-point` procedure of section 1.3.3 in terms of `iterative-improve`.

4.7.2 Answer