SICP Exercise Solutions for Section 2.2.3 and 2.2.4

Paul L. Snyder

October 17, 2014

Contents

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

```
#lang planet neil/sicp
```

2.2.3 Sequences as Conventional Interfaces

WRITEUP Exercise 2.33: List manipulations as accumulations

Problem

Fill in the missing expressions to complete the following definitions of some basic list-manipulation operations as accumulations:

```
(define (map p sequence)
  (accumulate (lambda (x y) <??>) nil sequence))

(define (append seq1 seq2)
  (accumulate cons <??> <??>))

(define (length sequence)
  (accumulate <??> 0 sequence))
```

Solution

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
```

```
        (accumulate op initial (cdr sequence)))))

(define (map-a p sequence)
  (accumulate (lambda (x y) (cons (p x) y)) nil sequence))

(define (append-a seq1 seq2)
  (accumulate cons seq2 seq1))

(define (length-a sequence)
  (accumulate (lambda (x y) (+ y 1)) 0 sequence))
```

Note that the accumulator is the *second* argument to the supplied procedure, not the first (as it is in Clojure).

WRITEUP Exercise 2.34: Horner's Rule

Problem

Evaluating a polynomial in x at a given value of x can be formulated as an accumulation. We evaluate the polynomial

$$a_n r^n | a_(n-1) r^(n-1) + \cdots + a_1 r + a_0$$

using a well-known algorithm called "Horner's rule", which structures the computation as

$$(\cdots (a_n r + a_(n-1))r + \cdots + a_1)r + a_0$$

In other words, we start with a_n, multiply by x, add $a_(n-1)$, multiply by x, and so on, until we reach a_0.(3)

Fill in the following template to produce a procedure that evaluates a polynomial using Horner's rule. Assume that the coefficients of the polynomial are arranged in a sequence, from a_0 through a_n.

```
(define (horner-eval x coefficient-sequence)
  (accumulate (lambda (this-coeff higher-terms) <??>)
              0
              coefficient-sequence))
```

For example, to compute $1 + 3x + 5x^3 + x^(5)$ at $x = 2$ you would evaluate

```
(horner-eval 2 (list 1 3 0 5 0 1))
```

Solution

It seems odd that the coefficient sequence is ordered in an order that is reversed from what one might expect. But, as can be seen from a close examination, `accumulate` builds up its results recursively.

First, we'll calculate the long way, to make sure the procedure we write produces the right result.

$$1 + 3 \cdot 2 + 5 \cdot 2^3 + 2^5 = 1 + 6 + 5 \times 8 + 32 = 79$$

```
(define (horner-eval x coefficient-sequence)
  (accumulate (lambda (this-coeff higher-terms)
                (+ this-coeff (* x higher-terms)))
              0
              coefficient-sequence))
```

This problem almost helps you too far along the path. It's quite clear how the arguments to the lambda should be manipulated in the body, given its naming. The reversal remains profoundly counter-intuitive, though (at least to me).

We can easily verify that this procedure produces the correct result:

```
(horner-eval 2 (list 1 3 0 5 0 1))
```

Exercise 2.35: `count-leaves` as an accumulation

Problem

Redefine `count-leaves` from section 2.2.2 as an accumulation:

```
(define (count-leaves t)
  (accumulate <??> <??> (map <??> <??>)))
```

Solution

We begin by recalling the definition of `count-leaves` from the text:

```
(define (count-leaves x)
  (cond ((null? x) 0)
        ((not (pair? x)) 1)
        (else (+ (count-leaves (car x))
                 (count-leaves (cdr x))))))
```

We can see that the exercises are beginning to give us less help... but there's still an important piece of information: the sequence is processed using `map` before the `accumulate` call is processed. The formulation of the original `count-leaves` further lets us that that, almost certain, recursion will be required to solve this problem.

The first thing we realize is that the initial value should be 0, since we're counting things:

```
(define (count-leaves t)
  (accumulate <??> 0 (map <??> <??>)))
```

We also know that the first argument is going to be a procedure, so we can plug in a lambda as a placeholder.

```
(define (count-leaves t)
  (accumulate (lambda (subtotal running-total) <??>) 0 (map <??> <??>)))
```

We can also begin to fill in the `map`'s arguments, since the first also has to be a lambda, while the second one is a list: and since the overall tree `t` has to be supplied at some point, the second position is the most sensible location

```
(define (count-leaves t)
  (accumulate (lambda (subtotal running-total) <??>)
              0
              (map (lambda (x) <??>) t)))
```

What could that second lambda be? Well, what we *wish* we could supply to the accumulate is a list with subtotal of the leaves in that particular branch of the tree. And, in fact, we have a procedure that can do that for us: `count-leaves` itself! Here's the recursive call.

```
(define (count-leaves t)
  (accumulate (lambda (subtotal running-total) <??>)
              0
              (map count-leaves t)))
```

This lets us realize what the first procedure should be as well: addition! All we need to do is sum the list of subtotals.

```
(define (count-leaves t)
  (accumulate + 0 (map count-leaves t)))
```

Sadly, our wishful thinking has taken us a bit too far: the `map` will fail if supplied a second argument that is not a list. So, we have to go back to a lambda, to wrap the call to `count-leaves` to handle the termination cases.

```
(define (count-leaves t)
  (accumulate + 0 (map
                   (lambda (x)
                     (cond ((null? x)       0)
                           ((not (pair? x)) 1)
                           (true            (count-leaves x))))
                   t)))
```

And, this does indeed work the way that we would hope and expect:

```
(count-leaves (list (list 1 2) 3 4))
```

Whether this formulation is an actual improvement is left as an exercise for the reader.

WRITEUP Exercise 2.36: Accumulating over multiple sequences

Problem

The procedure `accumulate-n` is similar to `accumulate` except that it takes as its third argument a sequence of sequences, which are all assumed to have the same number of elements. It applies the designated accumulation procedure to combine all the first elements of the sequences, all the second elements of the sequences, and so on, and returns a sequence of the results. For instance, if `s` is a sequence containing four sequences, `((1 2 3) (4 5 6) (7 8 9) (10 11 12))`, then the value of `(accumulate-n + 0 s)` should be the sequence `(22 26 30)`. Fill in the missing expressions in the following definition of `accumulate-n`:

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      nil
      (cons (accumulate op init <??>)
            (accumulate-n op init <??>))))
```

Solution

The approach to this problem builds quite naturally on everything that comes before, and a bit of thought should reveal that you can make a sequence of the first elements of each list by simple =map=ping over the =car=s. Similarly, =map=ping over the =cdr=s results will strip off the already-processed first elements.

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      nil
      (cons (accumulate op init (map car seqs))
            (accumulate-n op init (map cdr seqs)))))

(define s (list (list 1 2 3) (list 4 5 6) (list 7 8 9) (list 10 11 12)))

(display (accumulate-n + 0 s))
```

TODO Exercise 2.37: Implementing vectors and matrices with sequences

Problem

Suppose we represent vectors $\mathbf{v} = (v_i)$ as sequences of numbers, and matrices $\mathbf{M} = (m_{ij})$ as sequences of vectors (the rows of the matrix). For example, the matrix

$$\begin{matrix} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 6 \\ 6 & 7 & 8 & 9 \end{matrix}$$

is represented as the sequence `((1 2 3 4) (4 5 6 6) (6 7 8 9))`. With this representation, we can use sequence operations to concisely express the basic matrix and vector operations. These operations (which are described in any book on matrix algebra) are the following:

`(dot-product v w)`	returns the sum $\sum_i v_i w_i$
`(matrix-*-vector m v)`	returns the vector $\mathbf{t}$, where $t_i = \sum_j m_{ij} v_j$
`(matrix-*-matrix m n)`	returns the matrix $\mathbf{P}$, where $p_{ij} = \sum_k m_{ik} n_{kj}$
`(transpose m)`	returns the matrix $\mathbf{N}$, where $n_{ij} = m_{ji}$

We can define the dot product as

```
(define (dot-product v w)
  (accumulate + 0 (map * v w)))
```

Fill in the missing expressions in the following procedures for computing the other matrix operations. (The procedure `accumulate-n` is defined in Exercise 2.36.)

```
(define (matrix-*-vector m v)
  (map <??> m))

(define (transpose mat)
  (accumulate-n <??> <??> mat))

(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map <??> m)))
```

Solution

The first definition, the matrix-vector product is fairly easy: from the definition in the table, we can see that the form of the operation is quite similar to dot product. In fact, the \$i\$th element of the resulting vector will be the dot production of the \$i\$th row of the matrix with the vector **v**.

```
(define (matrix-*-vector m v)
  (map (lambda (m_i) (dot-product m_i v)) m))
```

For a ready-figured example, `http://mathinsight.org/matrix_vector_multiplication`

```
(display (matrix-*-vector (list (list 1 -1 2)
                                (list 0 -3 1))
                          (list 2 1 0)))
```

The definition of `transpose` is brilliantly simple. From our construction of `accumulate-n`, we saw how easy it was to slice-and-dice the initial elements of multiple lists; we can use this exact same procedure to build a new list of each using `cons`.

```
(define (transpose mat)
  (accumulate-n cons nil mat))

(define m (list (list 1 -1 2) (list 0 -3 1)))

(display (transpose m))
```

The definition of matrix-matrix product in the text is a bit unhelpful in the context of this exercise, though it is exactly the procedure you would perform to calculate each resulting cell. It's easier to describe the matrix-matrix product in terms of other operations: each column of the resulting product matrix **P** is the matrix-vector product of **A** with the corresponding column of **B**.

The authors kindly transpose **N** for us in the provided template, as by turning columns into rows we can easily map over them.

This looks like a good candidate for `accumulate-n`, at first…but the entirety of **M** has to be used for every row of **N**.

```
(define A (list (list 0 4 -2) (list -4 -3 0)))
(define B (list (list 0 1) (list 1 -1) (list 2 3)))
;; AB = ((0 -10) (-3 -1))

(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (transpose
     (map (lambda (p) (matrix-*-vector m p)) cols))))

(display
 (matrix-*-matrix A B))
```

Figure out a way around the extra transpose!

TODO Exercise 2.38: Considering `fold-left`

Problem

The `accumulate` procedure is also known as `fold-right`, because it combines the first element of the sequence with the result of combining all the elements to the right. There is also a `fold-left`, which is similar to `fold-right`, except that it combines elements working in the opposite direction:

```scheme
(define (fold-left op initial sequence)
  (define (iter result rest)
    (if (null? rest)
        result
        (iter (op result (car rest))
              (cdr rest))))
  (iter initial sequence))
```

What are the values of

```scheme
(fold-right / 1 (list 1 2 3))
```

```scheme
(fold-left / 1 (list 1 2 3))
```

```scheme
(fold-right list nil (list 1 2 3))
```

```scheme
(fold-left list nil (list 1 2 3))
```

Give a property that `op` should satisfy to guarantee that `fold-right` and `fold-left` will produce the same values for any sequence.

Solution

First, an alias for fold-right,

```scheme
(define fold-right accumulate)
```

```scheme
(fold-right / 1 (list 1 2 3))
```

```scheme
(fold-left / 1 (list 1 2 3))
```

```scheme
(display (fold-right list nil (list 1 2 3)))
```

```scheme
(display (fold-left list nil (list 1 2 3)))
```

`op` should satisfy the property of *commutativity* in order to have the same result regardless of the direction in which the operations are performed. That is, for an operation $\odot$, it should be the case that, for any valid choices of a, b, and c, $(a \odot b) \odot c) = a \odot (b \odot c)$.

TODO Exercise 2.39: Defining `reverse` via `fold`

Problem

Complete the following definitions of `reverse` (*Note Exercise 2-18::) in terms of `fold-right` and `fold-left` from *Note Exercise 2-38:::

```
(define (reverse sequence)
  (fold-right (lambda (x y) <??>) nil sequence))

(define (reverse sequence)
  (fold-left (lambda (x y) <??>) nil sequence))
```

Solution

TODO Exercise 2.40: Generating unique pairs

Problem

Define a procedure `unique-pairs` that, given an integer n, generates the sequence of pairs (i, j) with $1 <= j < i <= n$. Use `unique-pairs` to simplify the definition of `prime-sum-pairs` given above.

Solution

TODO Exercise 2.41: Finding ordered triples

Problem

Write a procedure to find all ordered triples of distinct positive integers i, j, and k less than or equal to a given integer n that sum to a given integer s.

Solution

TODO Exercise 2.42: The Eight-Queens Puzzle

Problem

The "eight-queens puzzle" asks how to place eight queens on a chessboard so that no queen is in check from any other (i.e., no two queens are in the same row, column, or diagonal). One possible solution is shown in *Note Figure 2-8::. One way to solve the puzzle is to work across the board, placing a queen in each column. Once we have placed $k-1$ queens, we must place the kth queen in a position where it does not check any of the queens already on the board. We can formulate this approach recursively: Assume that we have already generated the sequence of all possible ways to place $k-1$ queens in the first $k-1$ columns of the board. For each of these ways, generate an extended set of positions by placing a queen in each row of the kth column. Now filter these, keeping only the positions for which the queen in the kth column is safe with respect to the other queens. This produces the sequence of all ways to place k queens in the first k columns. By continuing this process, we will produce not only one solution, but all solutions to the puzzle.

Figure 2.8: A solution to the eight-queens puzzle.

					Q		
		Q					
Q							
						Q	
				Q			
							Q
	Q						
			Q				

We implement this solution as a procedure `queens`, which returns a sequence of all solutions to the problem of placing n queens on an $n * n$ chessboard. `queens` has an internal procedure `queen-cols` that returns the sequence of all ways to place queens in the first k columns of the board.

```
(define (queens board-size)
  (define (queen-cols k)
    (if (= k 0)
        (list empty-board)
        (filter
         (lambda (positions) (safe? k positions))
         (flatmap
          (lambda (rest-of-queens)
            (map (lambda (new-row)
                   (adjoin-position new-row k rest-of-queens))
                 (enumerate-interval 1 board-size)))
          (queen-cols (- k 1))))))
  (queen-cols board-size))
```

In this procedure `rest-of-queens` is a way to place $k-1$ queens in the first $k-1$ columns, and `new-row` is a proposed row in which to place the queen for the kth column. Complete the program by implementing the representation for sets of board positions, including the procedure `adjoin-position`, which adjoins a new row-column position to a set of positions, and `empty-board`, which represents an empty set of positions. You must also write the procedure `safe?`, which determines for a set of positions, whether the queen in the kth column is safe with respect to the others. (Note that we need only check whether the new queen is safe–the other queens are already guaranteed safe with respect to each other.)

Solution

TODO Exercise 2.43: Performance of nested mappings

Problem

Louis Reasoner is having a terrible time doing *Note Exercise 2-42::. His `queens` procedure seems to work, but it runs extremely slowly. (Louis never does manage to wait long enough for it to solve even the $6 * 6$ case.) When Louis asks Eva Lu Ator for help, she points out that he has interchanged the order of the nested mappings in the `flatmap`, writing it as

```
(flatmap
 (lambda (new-row)
   (map (lambda (rest-of-queens)
          (adjoin-position new-row k rest-of-queens))
        (queen-cols (- k 1))))
 (enumerate-interval 1 board-size))
```

Explain why this interchange makes the program run slowly. Estimate how long it will take Louis's program to solve the eight-queens puzzle, assuming that the program in *Note Exercise 2-42:: solves the puzzle in time T.

Solution

2.2.4 Example: A Picture Language

WRITEUP Exercise 2.44: A simple picture procedure

Problem

Define the procedure `up-split` used by `corner-split`. It is similar to `right-split`, except that it switches the roles of `below` and `beside`.

Solution

First, we refer to the procedures defined in the text:

```
(define (flipped-pairs painter)
  (let ((painter2 (beside painter (flip-vert painter))))
    (below painter2 painter2)))

(define (right-split painter n)
  (if (= n 0)
      painter
      (let ((smaller (right-split painter (- n 1))))
        (beside painter (below smaller smaller)))))
```

With this as a base, defining the `up-split` procedure is straightforward:

```
(define (up-split painter n)
  (if (= n 0)
      painter
      (let ((smaller (up-split painter (- n 1))))
        (below painter (beside smaller smaller)))))
```

With its dependencies defined, the `corner-split` procedure from the text can be evaluated:

```
(define (corner-split painter n)
  (if (= n 0)
      painter
      (let ((up (up-split painter (- n 1)))
            (right (right-split painter (- n 1))))
        (let ((top-left (beside up up))
              (bottom-right (below right right))
              (corner (corner-split painter (- n 1))))
          (beside (below painter top-left)
                  (below bottom-right corner))))))

(define (square-limit painter n)
  (let ((quarter (corner-split painter n)))
    (let ((half (beside (flip-horiz quarter) quarter)))
      (below (flip-vert half) half))))
```

WRITEUP Exercise 2.45: Generalizing splitting

Problem

`right-split` and `up-split` can be expressed as instances of a general splitting operation. Define a procedure `split` with the property that evaluating

```
(define right-split (split beside below))
(define up-split (split below beside))
```

produces procedures `right-split` and `up-split` with the same behaviors as the ones already defined.

Solution

This exercise is another good example of Abelson and Sussman's gentle approach to increasing abstractions. This is a short jump from the previous exercise, but it lands in much deeper functional waters.

Unfortunately, this exercise glosses over the significant challenges of creating a self-contained, recursive lambda. The natural way to do this in Scheme is via the `letrec` special form, but this doesn't get introduced until the last Chapter.

```
(define (split op1 op2)
  (letrec ((splitter (lambda (painter n)
                       (if (= n 0)
                           painter
                           (let ((smaller (splitter painter (- n 1))))
                             (op1 painter (op2 smaller smaller)))))))
    splitter))


(define (split-letrec op1 op2)
  (letrec ((splitter (lambda (painter n)
                        (if (= n 0)
                            painter
                            (let ((smaller (splitter painter (- n 1))))
                              (op1 painter (op2 smaller smaller)))))))
    splitter))

(define (split-ugly op1 op2)
```

```scheme
  (lambda (painter n)
    (if (= n 0)
        painter
        (let ((smaller ((split-ugly op1 op2) painter (- n 1))))
          (op1 painter (op2 smaller smaller))))))

(define (split-pure op1 op2)
  ((lambda (f) (f f))
   (lambda (split-wrap)
     (lambda (painter n)
       (if (= n 0)
           painter
           (let ((smaller ((split-wrap split-wrap) painter (- n 1))))
             (op1 painter (op2 smaller smaller))))))))
```

We can now define the splitting operations in terms of this more general procedure:

```scheme
(define rsplit (split beside below))
(define usplit (split below beside))
```

TODO Exercise 2.46: A vector data structure

Problem

A two-dimensional vector v running from the origin to a point can be represented as a pair consisting of an x-coordinate and a y-coordinate. Implement a data abstraction for vectors by giving a constructor `make-vect` and corresponding selectors `xcor-vect` and `ycor-vect`. In terms of your selectors and constructor, implement procedures `add-vect`, `sub-vect`, and `scale-vect` that perform the operations vector addition, vector subtraction, and multiplying a vector by a scalar:

$$\begin{aligned}(x_1, y_1) + (x_2, y_2) &= (x_1 + x_2, y_1 + y_2)\\ (x_1, y_1) - (x_2, y_2) &= (x_1 - x_2, y_1 - y_2)\\ s * (x, y) &= (sx, sy)\end{aligned}$$

Solution

```scheme
(define (make-vect x y)
  (cons x y))
```

```scheme
(define (xcor-vect v)
  (car v))

(define (ycor-vect v)
  (cdr v))

(define (add-vect v w)
  (make-vect (+ (xcor-vect v) (xcor-vect w))
             (+ (ycor-vect v) (ycor-vect w))))

(define (sub-vect v w)
  (make-vect (- (xcor-vect v) (xcor-vect w))
             (- (ycor-vect v) (ycor-vect w))))

(define (scale-vect s v)
  (make-vect (* s (xcor-vect v)) (* s (ycor-vect v))))
```

TODO Exercise 2.47: Selectors for frame constructors

Problem

Here are two possible constructors for frames:

```scheme
(define (make-frame origin edge1 edge2)
  (list origin edge1 edge2))

(define (make-frame origin edge1 edge2)
  (cons origin (cons edge1 edge2)))
```

For each constructor supply the appropriate selectors to produce an implementation for frames.

Solution

```scheme
(define (make-frame origin edge1 edge2)
  (list origin edge1 edge2))

(define (make-frame-dotted origin edge1 edge2)
  (cons origin (cons edge1 edge2)))
```

```scheme
;; These selectors work with both internal representations
(define (frame-origin frame)
  (car frame))

(define (frame-edge1 frame)
  (cadr frame))

;; This works for the proper list representation
(define (frame-edge2 frame)
  (caddr frame))

;; Second version with dotted list representation
(define (frame-edge2-other frame)
  (cddr frame))
```

TODO Exercise 2.48: Representing segments

Problem

A directed line segment in the plane can be represented as a pair of vectors–the vector running from the origin to the start-point of the segment, and the vector running from the origin to the end-point of the segment. Use your vector representation from *Note Exercise 2-46:: to define a representation for segments with a constructor `make-segment` and selectors `start-segment` and `end-segment`.

Solution

```scheme
(define make-segment cons)

(define start-segment car)

(define end-segment cdr)
```

TODO Exercise 2.49: Defining some primitive painters

Problem

Use `segments->painter` to define the following primitive painters:

a. The painter that draws the outline of the designated frame.

b. The painter that draws an "X" by connecting opposite corners of the frame.

c. The painter that draws a diamond shape by connecting the midpoints of the sides of the frame.

d. The `wave` painter.

Solution

The SICP support package includes `segments->painter` (which is implemented in terms of a primitive `draw-lines-on-screen` rather than `draw-lines`), so we just rely on that rather than recreating the version from the text. It expects the segments to have selectors named `segment-start` and `segment-end`, rather than `start-segment` and `end-segment`.

```
(define segment-start start-segment)

(define segment-end end-segment)

(define (successive-pairs l)
  (define (recur lst ret)
    (if (or (null? lst) (null? (cdr lst)))
        (reverse ret)
        (recur (cdr lst)
               (cons (list (car lst) (cadr lst)) ret))))
  (recur l nil))

(define (make-path vects)
  (map (lambda (vector-pair)
         (make-segment (car vector-pair) (cadr vector-pair)))
       (successive-pairs vects)))

(define outline-path
  (make-path
   (make-vect 0.0 0.0)
   (make-vect 0.0 1.0)
   (make-vect 1.0 1.0)
   (make-vect 1.0 0.0)))

(define outline (segments->painter outline-path))
```

TODO Exercise 2.50: A horizontal flip transformer

Problem

Define the transformation `flip-horiz`, which flips painters horizontally, and transformations that rotate painters counterclockwise by 180 degrees and 270 degrees.

Solution

TODO Exercise 2.51: Defining `below`

Problem

Define the `below` operation for painters. `below` takes two painters as arguments. The resulting painter, given a frame, draws with the first painter in the bottom of the frame and with the second painter in the top. Define `below` in two different ways–first by writing a procedure that is analogous to the `beside` procedure given above, and again in terms of 'beside' and suitable rotation operations (from *Note Exercise 2-50::).

Solution

TODO Exercise 2.52: Changing patterns

Problem

Make changes to the square limit of `wave` shown in *Note Figure 2-9:: by working at each of the levels described above. In particular:

a. Add some segments to the primitive `wave` painter of *Note Exercise 2-49:: (to add a smile, for example).

b. Change the pattern constructed by `corner-split` (for example, by using only one copy of the `up-split` and `right-split` images instead of two).

c. Modify the version of `square-limit` that uses `square-of-four` so as to assemble the corners in a different pattern. (For example, you might make the big Mr. Rogers look outward from each corner of the square.)

Solution