SICP Exercise Solutions for Section 2.2.1 and 2.2.2

Paul L. Snyder

August 29, 2014

Contents

1 2.2.1 Representing Sequences

1.1 WRITEUP Exercise 2.17:

1.1.1 Problem

Define a procedure `last-pair` that returns the list that contains only the last element of a given (nonempty) list:

```
(last-pair (list 23 72 149 34))
(34)
```

1.1.2 Solution

```
(define (last-pair lst)
  (if (null? (cdr lst))
      lst
      (last-pair (cdr lst))))

(last-pair (list 23 72 149 34))
```

1.2 WRITEUP Exercise 2.18:

1.2.1 Problem

Define a procedure 'reverse' that takes a list as argument and returns a list of the same elements in reverse order:

```
(reverse (list 1 4 9 16 25))
(25 16 9 4 1)
```

1.2.2 Solution

```
(define (reverse lst)
  (define (iter lst ret)
    (if (null? lst)
        ret
        (iter (cdr lst) (cons (car lst) ret))))
  (iter lst null))

 (reverse (list 1 4 9 16 25))
```

1.3 TODO Exercise 2.19:

1.3.1 Problem

Consider the change-counting program of section *Note 1-2-2::. It would be nice to be able to easily change the currency used by the program, so that we could compute the number of ways to change a British pound, for example. As the program is written, the knowledge of the currency is distributed partly into the procedure 'first-denomination' and partly into the procedure 'count-change' (which knows that there are five kinds of U.S. coins). It would be nicer to be able to supply a list of coins to be used for making change.

We want to rewrite the procedure ‘cc’ so that its second argument is a list of the values of the coins to use rather than an integer specifying which coins to use. We could then have lists that defined each kind of currency:

```
(define us-coins (list 50 25 10 5 1))

(define uk-coins (list 100 50 20 10 5 2 1 0.5))
```

We could then call ‘cc’ as follows:

```
(cc 100 us-coins)
292
```

To do this will require changing the program ‘cc’ somewhat. It will still have the same form, but it will access its second argument differently, as follows:

(define (cc amount coin-values) (cond ((= amount 0) 1) ((or (< amount 0) (no-more? coin-values)) 0) (else (+ (cc amount (except-first-denomination coin-values)) (cc (- amount (first-denomination coin-values)) coin-values)))))

Define the procedures ‘first-denomination’, ‘except-first-denomination’, and ‘no-more?’ in terms of primitive operations on list structures. Does the order of the list ‘coin-values’ affect the answer produced by ‘cc’? Why or why not?

1.3.2 Solution

1.4 WRITEUP Exercise 2.20:

1.4.1 Problem

The procedures `+`, `*`, and `list` take arbitrary numbers of arguments. One way to define such procedures is to use `define` with notation "dotted-tail notation". In a procedure definition, a parameter list that has a dot before the last parameter name indicates that, when the procedure is called, the initial parameters (if any) will have as values the initial arguments, as usual, but the final parameter’s value will be a "list" of any remaining arguments. For instance, given the definition

```
(define (f x y . z) <BODY>)
```

the procedure `f` can be called with two or more arguments. If we evaluate

```
(f 1 2 3 4 5 6)
```

then in the body of `f`, `x` will be 1, `y` will be 2, and `z` will be the list `(3 4 5 6)`. Given the definition

```
(define (g . w) <BODY>)
```

the procedure `g` can be called with zero or more arguments. If we evaluate

```
(g 1 2 3 4 5 6)
```

then in the body of `g`, `w` will be the list `(1 2 3 4 5 6)`.(4)

Use this notation to write a procedure `same-parity` that takes one or more integers and returns a list of all the arguments that have the same even-odd parity as the first argument. For example,

```
(same-parity 1 2 3 4 5 6 7)
(1 3 5 7)

(same-parity 2 3 4 5 6 7)
(2 4 6)
```

1.4.2 Solution

```
(define (same-parity key . vals)
  (define (match val) (= (remainder key 2) (remainder val 2)))
  (define (iter ret vals)
    (cond ((null? vals)       (reverse ret))
          ((match (car vals)) (iter (cons (car vals) ret) (cdr vals)))
          (true               (iter ret (cdr vals)))))
  (iter (list key) vals))
```

1.5 WRITEUP Exercise 2.21:

1.5.1 Problem

The procedure `square-list` takes a list of numbers as argument and returns a list of the squares of those numbers.

```
(square-list (list 1 2 3 4))
(1 4 9 16)
```

Here are two different definitions of `square-list`. Complete both of them by filling in the missing expressions:

```
(define (square-list items)
  (if (null? items)
      nil
      (cons <??> <??>)))

(define (square-list items)
  (map <??> <??>))
```

1.5.2 Solution

```
(define (square-list-1 items)
  (if (null? items)
      null
      (cons (* (car items) (car items)) (square-list-1 (cdr items)))))

(define (square-list-2 items)
  (map (lambda (x) (* x x)) items))
```

1.6 WRITEUP Exercise 2.22:

1.6.1 Problem

Louis Reasoner tries to rewrite the first `square-list` procedure of *Note Exercise 2-21:: so that it evolves an iterative process:

```
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons (square (car things))
                    answer))))
  (iter items nil))
```

Unfortunately, defining `square-list` this way produces the answer list in the reverse order of the one desired. Why?

Louis then tries to fix his bug by interchanging the arguments to `cons`:

```
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons answer
                    (square (car things))))))
  (iter items nil))
```

This doesn't work either. Explain.

1.6.2 Solution

For the first example, with each recursive call to `iter`, the `cons` adds successive items to the left side of the list: when using `cons`, an item added to a list is appended to the beginning (since the only $O(1)$ access using the pointer to the cell at the start of the list.

For the second, the procedure builds an ill-formed data structure. Scheme only treats the pattern of cons cells as a list if, for each cons structure, the left cell holds a data item and the right cell holds either a cons cell representing a properly structured list or `nil`.

1.7 WRITEUP Exercise 2.23:

1.7.1 Problem

The procedure `for-each` is similar to `map`. It takes as arguments a procedure and a list of elements. However, rather than forming a list of the results, `for-each` just applies the procedure to each of the elements in turn, from left to right. The values returned by applying the procedure to the elements are not used at all–=for-each= is used with procedures that perform an action, such as printing. For example,

```
(for-each (lambda (x) (newline) (display x))
          (list 57 321 88))
57
321
88
```

The value returned by the call to `for-each` (not illustrated above) can be something arbitrary, such as true. Give an implementation of `for-each`.

1.7.2 Solution

```
(define (for-each f items)
  (if (null? items)
      null
      (begin
        (f (car items))
        (for-each f (cdr items)))))

(for-each (lambda (x) (print "woo:") (print x) (newline)) '(1 2 5 6 7))
```

2 2.2.2 Hierarchical Structures

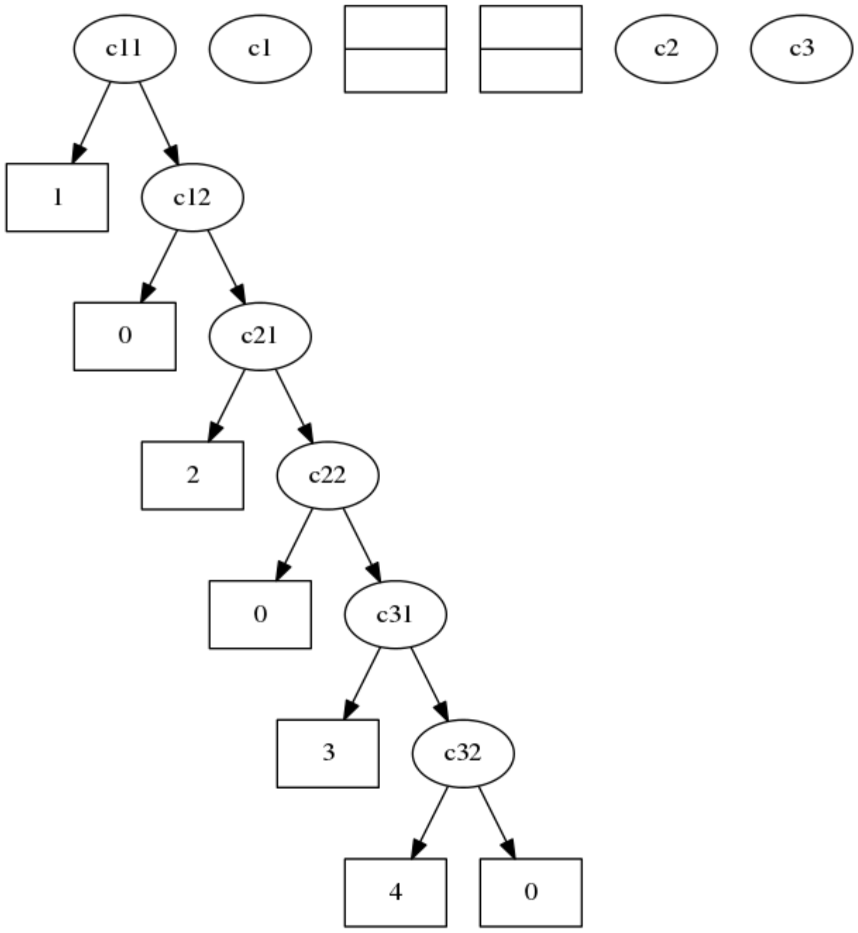
2.1 TODO Exercise 2.24:

2.1.1 Problem

Suppose we evaluate the expression `(list 1 (list 2 (list 3 4)))`. Give the result printed by the interpreter, the corresponding box-and-pointer structure, and the interpretation of this as a tree (as in *Note Figure 2-6::).

2.1.2 Solution

```
(print (list 1 (list 2 (list 3 4))))
```



2.2 WRITEUP Exercise 2.25:

2.2.1 Problem

Give combinations of 'car's and 'cdr's that will pick 7 from each of the following lists:

```
(1 3 (5 7) 9)
```

```
((7))
```

```
(1 (2 (3 (4 (5 (6 7))))))
```

2.2.2 Solution

```
(define l1 (list 1 3 (list 5 7) 9))
(define l2 (list (list 7)))
(define l3 (list 1 (list 2 (list 3 (list 4 (list 5 (list 6 7)))))))

(print (car (cdr (car (cdr (cdr l1))))))
(newline)
(print (car (car l2)))
(newline)
(print
 (car (cdr (car (cdr (car (cdr (car (cdr (car (cdr (car (cdr l3)))))))))))))
(newline)
```

2.3 WRITEUP Exercise 2.26:

2.3.1 Problem

Suppose we define `x` and `y` to be two lists:

```
(define x (list 1 2))
(define y (list 4 5 6))
```

What result is printed by the interpreter in response to evaluating each of the following expressions:

```
(append x y)
(cons x y)
(list x y)
```

2.3.2 Solution

```
(define x (list 1 2))
(define y (list 4 5 6))

(print (append x y))
(newline)
(print (cons x y))
(newline)
(print (list x y))
```

2.4 WRITEUP Exercise 2.27:

2.4.1 Problem

Modify your `reverse` procedure of *Note Exercise 2-18:: to produce a `deep-reverse` procedure that takes a list as argument and returns as its value the list with its elements reversed and with all sublists deep-reversed as well. For example,

```
(define x (list (list 1 2) (list 3 4)))


x
((1 2) (3 4))

(reverse x)
((3 4) (1 2))

(deep-reverse x)
((4 3) (2 1))
```

2.4.2 Solution

```
(define (deep-reverse lst)
  (define (recurse lst ret)
    (if (null? lst) ret
        (if (pair? lst)
            (recurse (cdr lst) (cons (deep-reverse (car lst)) ret))
            lst)))
  (recurse lst null))

(define z (list (list 1 2) (list 3 4)))

(reverse z)

(deep-reverse z)
```

2.5 WRITEUP Exercise 2.28:

2.5.1 Problem

Write a procedure `fringe` that takes as argument a tree (represented as a list) and returns a list whose elements are all the leaves of the tree arranged in left-to-right order. For example,

```
(define x (list (list 1 2) (list 3 4)))

(fringe x)
(1 2 3 4)

(fringe (list x x))
(1 2 3 4 1 2 3 4)
```

2.5.2 Solution

```
(define (fringe tr)
  (if (not (pair? tr))
      tr
      (let ((head (car tr))
            (tail (cdr tr)))
        (if (pair? head)
            (append (fringe head) (fringe tail))
            (cons head (fringe tail))))))

(define w (list (list 1 2) (list 3 4)))

(fringe w)
(fringe (list w w))
(fringe '(1 2))
```

2.6 TODO Exercise 2.29:

2.6.1 Problem

A binary mobile consists of two branches, a left branch and a right branch. Each branch is a rod of a certain length, from which hangs either a weight or another binary mobile. We can represent a binary mobile using compound data by constructing it from two branches (for example, using `list`):

```
(define (make-mobile left right)
  (list left right))
```

A branch is constructed from a `length` (which must be a number) together with a `structure`, which may be either a number (representing a simple weight) or another mobile:

```
(define (make-branch length structure)
  (list length structure))
```

a. Write the corresponding selectors `left-branch` and `right-branch`, which return the branches of a mobile, and `branch-length` and `branch-structure`, which return the components of a branch.

b. Using your selectors, define a procedure `total-weight` that returns the total weight of a mobile.

c. A mobile is said to be "balanced" if the torque applied by its top-left branch is equal to that applied by its top-right branch (that is, if the length of the left rod multiplied by the weight hanging from that rod is equal to the corresponding product for the right side) and if each of the submobiles hanging off its branches is balanced. Design a predicate that tests whether a binary mobile is balanced.

d. Suppose we change the representation of mobiles so that the constructors are

```
(define (make-mobile left right)
  (cons left right))

(define (make-branch length structure)
  (cons length structure))
```

How much do you need to change your programs to convert to the new representation?

2.6.2 Solution

2.7 WRITEUP Exercise 2.30:

2.7.1 Problem

Define a procedure `square-tree` analogous to the `square-list` procedure of *Note Exercise 2-21::. That is, `square-list` should behave as follows:

```
(square-tree
 (list 1
       (list 2 (list 3 4) 5)
       (list 6 7)))
(1 (4 (9 16) 25) (36 49))
```

Define `square-tree` both directly (i.e., without using any higher-order procedures) and also by using `map` and recursion.

2.7.2 Solution

```
(define (square-tree-1 tr)
  (cond ((null? tr) null)
        ((not (pair? tr)) (* tr tr))
        (true (cons (square-tree-1 (car tr)) (square-tree-1 (cdr tr))))))

(define (square-tree-2 tr)
  (define (square x) (* x x))
  (cond ((null? tr) null)
        ((not   (pair? tr)) (square tr))
        (true   (map square-tree-2 tr))))

(define num-tree
  (list 1
        (list 2 (list 3 4) 5)
        (list 6 7)))

(print (square-tree-1 num-tree))
(newline)
(print (square-tree-2 num-tree))
(newline)
```

2.8 WRITEUP Exercise 2.31:

2.8.1 Problem

Abstract your answer to *Note Exercise 2-30:: to produce a procedure `tree-map` with the property that `square-tree` could be defined as

```
(define (square-tree tree) (tree-map square tree))
```

2.8.2 Solution

```
(define (square x) (* x x))

(define (tree-map f tr)
  (define (tree-map-h t) (tree-map f t))
  (cond ((null? tr) null)
        ((not   (pair? tr)) (f tr))
        (true   (map tree-map-h tr))))
```

```
(define (square-tree-3 tr)
  (tree-map square tr))

(define num-tree
  (list 1
        (list 2 (list 3 4) 5)
        (list 6 7)))

(print (square-tree-3 num-tree))
```

2.9 WRITEUP Exercise 2.32:

2.9.1 Problem

We can represent a set as a list of distinct elements, and we can represent the set of all subsets of the set as a list of lists. For example, if the set is `(1 2 3)`, then the set of all subsets is `(() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))`. Complete the following definition of a procedure that generates the set of subsets of a set and give a clear explanation of why it works:

```
(define (subsets s)
  (if (null? s)
      (list nil)
      (let ((rest (subsets (cdr s))))
        (append rest (map <??> rest)))))
```

2.9.2 Solution

```
(define (subsets s)
  (if (null? s)
      (list null)
      (let ((rest (subsets (cdr s))))
        (append rest (map (lambda (t) (cons (car s) t)) rest)))))

(print (subsets (list 1 2 3)))
```

For a given set S, the set of all its subsets are frequently called its *power set*. A natural way to generate a set's power set is recursively: pick an element x of the set S; the power set is then the set of all the subsets of S that do not contain e, combined with all subsets that do contain e.