SICP Exercise Solutions for Section 2.1

Paul L. Snyder

July 1, 2014

Contents

1 2.1.1 Example: Arithmetic Operations for Rational Numbers

1.1 WRITEUP Exercise 2.1: Improving `make-rat`

1.1.1 Problem

Define a better version of `make-rat` that handles both positive and negative arguments. `make-rat` should normalize the sign so that if the rational number is positive, both the numerator and denominator are positive, and if the rational number is negative, only the numerator is negative.

1.1.2 Solution

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

(define (make-rat n d)
  (let ((nf (if (< d 0) (- n) n))
        (df (abs d))
        (g (abs (gcd n d))))
    (cons (/ nf g) (/ df g))))
```

2 2.1.2 Abstraction Barriers

2.1 WRITEUP Exercise 2.2: Representations for line segments

2.1.1 Problem

Consider the problem of representing line segments in a plane. Each segment is represented as a pair of points: a starting point and an ending point. Define a constructor `make-segment` and selectors `start-segment` and `end-segment` that define the representation of segments in terms of points. Furthermore, a point can be represented as a pair of numbers: the x coordinate and the y coordinate. Accordingly, specify a constructor `make-point` and selectors `x-point` and `y-point` that define this representation. Finally, using your selectors and constructors, define a procedure `midpoint-segment` that takes a line segment as argument and returns its midpoint (the point whose coordinates are the average of the coordinates of the endpoints). To try your procedures, you'll need a way to print points:

```
(define (print-point p)
  (newline)
  (display "(")
  (display (x-point p))
  (display ",")
  (display (y-point p))
  (display ")"))
```

2.1.2 Solution

```
(define (make-point x y)
  (cons x y))

(define (x-point p)
  (car p))

(define (y-point p)
  (cdr p))

(define (make-segment a b)
  (cons a b))

(define (start-segment s)
  (car s))

(define (end-segment s)
  (cdr s))

(define (midpoint-segment s)
  (let ((x1 (x-point (start-segment s)))
        (y1 (y-point (start-segment s)))
        (x2 (x-point (end-segment s)))
        (y2 (y-point (end-segment s))))
    (make-point (/ (+ x1 x2) 2) (/ (+ y1 y2) 2))))
```

2.2 Exercise 2.3: Rectangular representations

2.2.1 Problem

Implement a representation for rectangles in a plane. (Hint: You may want to make use of Exercise 2-2.) In terms of your constructors and selectors, create procedures that compute the perimeter and the area of a given rectangle. Now implement a different representation for rectangles. Can you design your system with suitable abstraction barriers, so that the same perimeter and area procedures will work using either representation?

2.2.2 Solution

One possibility is to represent a rectangle using any two points in the plane. For tidiness, we can normalize the internal representation of these much as we did for exercise 2.1.

```
(define (make-rectangle point1 point2)
  ;; normalize to top-left and bottom-right points
  (let ((x1 (min (x-point point1) (x-point point2)))
        (x2 (max (x-point point1) (x-point point2)))
        (y1 (max (y-point point1) (y-point point2)))
        (y2 (min (y-point point1) (y-point point2))))
    (cons (make-point x1 y1) (make-point x2 y2))))

(define (top-left-rectangle r)
  (car r))

(define (bottom-right-rectangle r)
  (cdr r))

(define (width-rectangle r)
  (let ((x1 (x-point (top-left-rectangle r)))
        (x2 (x-point (bottom-right-rectangle r))))
    (abs (- x2 x1))))

(define (height-rectangle r)
  (let ((y1 (y-point (top-left-rectangle r)))
        (y2 (y-point (bottom-right-rectangle r))))
    (abs (- y1 y2))))
```

```
(define (area-rectangle r)
  (* (width-rectangle r) (height-rectangle r)))

(define (perimeter-rectangle r)
  (* 2 (+ (width-rectangle r) (height-rectangle r))))
```

For another representation we could construct a representation from a point and offsets from this point in terms of width and height.

If we were constructing a real representation (rather than just working with an exercise in a book), we might like to do some additional normalization for this representation as well. In this case, though, we'll just go with a simple constructor.

Since we haven't been introduced to robust error-checking mechanisms, yet (such as contracts in Racket), we'll just assume that the width and height provided are positive, and providing negative values will result in undefined behavior.

```
(define (make-rectangle point width height)
  (cons point (cons width height)))

(define (bottom-right-rectangle r)
  (let ((x1 (x-point (top-left-rectangle r)))
        (y1 (y-point (top-left-rectangle r))))
    (make-point (+ x1 (car (cdr r))) (+ y1 (cdr (cdr r))))))
```

This isn't the most satisfying reimplementation, not least because we're missing a most important language feature: polymorphism: there's no way for a procedure to detect which internal representation is being used.

We could get around this by have this alternate constructor produce an internal representation that's the same as the first version, but that's not the description of the problem. For the moment, we'll just note that none of the other procedures need to be changed for this to work, but that based on the facilities we have available to us right now we can't easily use rectangles that use differing internal representations.

3 2.1.3 What Is Meant by Data?

3.1 Exercise 2.4: Lambdas as data structures

3.1.1 Problem

Here is an alternative procedural representation of pairs. For this representation, verify that `(car (cons x y))` yields `x` for any objects `x` and `y`.

```
(define (cons x y)
  (lambda (m) (m x y)))

(define (car z)
  (z (lambda (p q) p)))
```

What is the corresponding definition of `cdr`? (Hint: To verify that this works, make use of the substitution model of section 1.1.5.)

3.1.2 Solution

First, a definition for `cdr` using this approach:

```
(define (cdr z)
  (z (lambda (p q) q)))
```

Now, we verify these alternate versions of both `car` and `cdr`. First, to verify `car` we follow the suggestion in the exercise and use the substitution method.

```
1> (car (cons x y))
2> (car (lambda (m) (m x y)))
3> ((lambda (m) (m x y)) (lambda (p q) p))
4> ((lambda (p q) p) x y)
5> ((lambda (x y) x))
6> x
```

Verifying the alternative version of `cdr` follows exactly the same pattern.

```
1> (cdr (cons x y))
2> (cdr (lambda (m) (m x y)))
3> ((lambda (m) (m x y)) (lambda (p q) q))
4> ((lambda (p q) q) x y)
5> ((lambda (x y) y))
6> y
```

3.2 WRITEUP Exercise 2.5: Creative data representation

3.2.1 Problem

Show that we can represent pairs of non-negative integers using only numbers and arithmetic operations if we represent the pair a and b as the integer that is the product 2^a3^b. Give the corresponding definitions of the procedures `cons`, `car`, and `cdr`.

3.2.2 Solution

This one is fun. We can reuse the `expt` procedure for our encoding, but need a specific answer to decode a pair: the number of times 2 occurs as a factor of the pair is the value of a, and the number of time 3 occurs as a factor is the value of b. Rather than implement it separately for our `icar` and `icdr` procedures, it's best to capture this in another descriptive procedure: `factor-count`.

```
(define (icons a b)
  (* (expt 2 a) (expt 3 b)))

(define (factor-count i f)
  (define (iter i c)
    (if (= (remainder i f) 0)
        (iter (/ i f) (+ c 1))
        c))
  (iter i 0))

(define (icar p)
  (factor-count p 2))

(define (icdr p)
  (factor-count p 3))
```

3.3 WRITEUP Exercise 2.6: Church numerals

3.3.1 Problem

In case representing pairs as procedures wasn't mind-boggling enough, consider that, in a language that can manipulate procedures, we can get by without numbers (at least insofar as non-negative integers are concerned) by implementing 0 and the operation of adding 1 as

```
(define zero (lambda (f) (lambda (x) x)))

(define (add-1 n)
  (lambda (f) (lambda (x) (f ((n f) x)))))
```

This representation is known as "Church numerals", after its inventor, Alonzo Church, the logician who invented the λ calculus.

Define `one` and `two` directly (not in terms of `zero` and `add-1`). (Hint: Use substitution to evaluate `(add-1 zero)`). Give a direct definition of the addition procedure '+' (not in terms of repeated application of 'add-1').

3.3.2 Solution

There's another sneaky thing going on here. Note that `add-1` is defined as a *procedure*...this means that there's really an implicit lambda here that the syntactic sugar obscures. I wish that the authors hadn't done this, as it obscures the clarity of the substitution process. To make things a bit clearly, I'll use a slightly different (but effectively identical) desugared definition of `add-1`:

```
(define add-1 (lambda (n)
 (lambda (f) (lambda (x) (f ((n f) x))))))
```

Now, we can use this to shuffling all of these symbols around to try to come up with a direct definition of `one`:

```
1> (add-1 zero)
2> (add-1 (lambda (f) (lambda (x) x)))
3> ((lambda (n)
     (lambda (f)
      (lambda (x)
       (f ((n f) x))))
     (lambda (f) (lambda (x) x)))) ;; this line holds 'zero'
```

In step 3, note how `add-1`'s lambda is substituted into the execution position. Thus, `zero` becomes the argument `n` and is substituted into the body of `add-1`:

```
4> (lambda (f)
    (lambda (x)
     (f (( (lambda (f) (lambda (x) x)) ) f) x)))
```

Whew! Keeping all of the parentheses straight is pretty painful. The really important thing to note here is that `zero` lands not only in execution position, but in a *nested* execution position: it's going to be called itself, and then its resulting lambda will be called as well.

```
5> (lambda (f)
    (lambda (x)
     (f ((lambda (x) x)) x)))
```

After the first of these substitutions, the inner `f` disappears. Look again at the definition of `zero` to see why this must be so.

```
5> (lambda (f)
    (lambda (x)
     (f x)))
```

And here's the tasty surprise: we end up with a lambda that performs a single function application of its argument `f` on the argument to its inner lambda! Note that neither of these lambdas land in an execution position, so they can't be substituted out. This gives us our new, direct definition for `one`:

```
(define one (lambda (f) (lambda (x) (f x))))
```

With this in hand, we can take it to the next step: a definition for `two`. (You can probably guess what it is, if you've followed the logic of the substitutions to this point.)

(define add-1 (lambda (n) (lambda (f) (lambda (x) (f ((n f) x))))))

```
1> (add-1 one)
2> (add-1 (lambda (f) (lambda (x) (f x))))
3> ((lambda (n)
     (lambda (f)
      (lambda (x)
       (f ((n f) x)))))
    (lambda (f) (lambda (x) (f x))))
```

Having fought through `one`, teasing this out isn't too bad. The hardest part is keeping track of the scoping (that is, which nested `f` is associated with which lambda.

```
4> (lambda (f)
    (lambda (x)
     (f (( (lambda (f) (lambda (x) (f x))) f) x))))
5> (lambda (f)
    (lambda (x)
     (f ( (lambda (x) (f x)) x))))
```

Look carefully at the inner `lambda (f)` and the outer `lambda (f)`; the outer `f` is substituted into the inner for step 5. This is a likely source of confusion.

```
6> (lambda (f)
    (lambda (x)
     (f (f x))))
```

The same thing happens to inner and outer `x` in step 6. Yikes! But the result is reassuringly concise. (There are no prizes for guessing what the definition for `three` is going to be!)

So: we can see that a Church numeral is represented by a number of repeated applications of the same function to some abstract `x` at the bottom of the chain.

The jump from this to addition is actually extremely straightforward. Let's say we have two Church numerals is this representation, `p` and `q`: each is represented by a stack of nested calls to `f`. The goal is to merge those two stacks together.

Remember that each of those stacks is applied to some base case... which could simply be the result of the other stack! Thus, crafting a definition is actually quite straightforward (again, as long as we keep our nested function definitions straight). `q` needs to be applied to `x`, and then `p` needs to be applied to be result of that operation. Each of these stacks must also be passed the `f` that is being used, as well. This gives us our result:

```
(define (add p q)
 (lambda (f)
  (lambda (x)
   ((p f) ((q f) x)))))
```

4 2.1.4 Extended Exercise: Interval Arithmetic

4.1 WRITEUP Exercise 2.7: Selectors for interval arithmetic

4.1.1 Problem

Alyssa's program is incomplete because she has not specified the implementation of the interval abstraction. Here is a definition of the interval constructor:

```
(define (make-interval a b)
  (cons a b))
```

Define selectors `upper-bound` and `lower-bound` to complete the implementation.

4.1.2 Solution

First, we reproduce the definitions of Alyssa's procedures from the text:

```
(define (add-interval x y)
  (make-interval (+ (lower-bound x) (lower-bound y))
                 (+ (upper-bound x) (upper-bound y))))

(define (mul-interval x y)
  (let ((p1 (* (lower-bound x) (lower-bound y)))
        (p2 (* (lower-bound x) (upper-bound y)))
        (p3 (* (upper-bound x) (lower-bound y)))
        (p4 (* (upper-bound x) (upper-bound y))))
    (make-interval (min p1 p2 p3 p4)
                   (max p1 p2 p3 p4))))

(define (div-interval x y)
  (mul-interval x
                (make-interval (/ 1.0 (upper-bound y))
                               (/ 1.0 (lower-bound y)))))
```

The definitions of `upper-bound` and `lower-bound` are extremely straightforward, as long as we remember that the pair is not ordered. It might be desirable to have the `make-interval` constructor function check to make sure the `car` is always the lower bound, but here we're working with the definition as specified in the problem.

```
(define (upper-bound i)
  (if (> (car i) (cdr i))
      (car i)
      (cdr i)))

(define (lower-bound i)
  (if (< (car i) (cdr i))
      (car i)
      (cdr i)))
```

4.2 WRITEUP Exercise 2.8: Subtracting intervals

4.2.1 Problem

Using reasoning analogous to Alyssa's, describe how the difference of two intervals may be computed. Define a corresponding subtraction procedure, called `sub-interval`.

4.2.2 Solution

This one is a little bit tricky: the "analogous reasoning" that we need to do is to figure out the smallest and largest possible results of the operation. For the smallest, the lowest value we can achieve is when we subtract the highest possible value of the second interval for the lowest possible value of the first. Similarly, the largest results could occur if the first interval is at its highest value and the second interval is at its lowest.

```
(define (sub-interval x y)
  (make-interval (- (lower-bound x) (upper-bound y))
                 (- (upper-bound x) (lower-bound y))))
```

4.3 Exercise 2.9: Interval widths

4.3.1 Problem

The "width" of an interval is half of the difference between its upper and lower bounds. The width is a measure of the uncertainty of the number specified by the interval. For some arithmetic operations the width of the result of combining two intervals is a function only of the widths of the argument intervals, whereas for others the width of the combination is not a function of the widths of the argument intervals. Show that the width of the sum (or difference) of two intervals is a function only of the widths of the intervals being added (or subtracted). Give examples to show that this is not true for multiplication or division.

4.3.2 Solution

It may seem a bit surprising that the "width" of an interval is half of the distance it spans, but it may make more sense to think of it using a standard notation. For example, for the interval $[8, 12]$, the difference between 8 and 12 is 4, and half that (and hence the width) is 2. It would frequently be written as 10 ± 2.

```
(define width-interval x
  (/ (- (upper-bound x) (lower-bound x)) 2))
```

Suppose we have two intervals, x of width w_x and y of width w_y. The endpoints of x are, by definition $[x - w_x, x + w_x)]$, and y's endpoints are $[y - w_y, y + w_y]$.

Suppose we add these two intervals together. By the definitions given in the text, we get

$$[(x-w_x)+(y-w_y),(x+w_x)+(y+w_y)] = (x+y-w_x-w_y, x+y+w_x+w_y) = (x+y-(w_x+w_y), x+y+(w_x+w$$

Translating this new interval into the "center plus/minus width" notation, we get $(x + y) \pm (w_x + w_y)$. Thus, the width of the sum of the two intervals is $w_x + w_y$.

Similarly, suppose we subtract y from x. Using the definition from exercise 2.7, we get

$$[x-w_x, x+w_x]-[y-w_y, y+w_y] = [(x-w_x)-(y+w_y),(x+w_x)-(y-w_y)] = [x-w_x-y-w_y, x+w_x-y+w_y] =$$

Interestingly, we can see that the uncertainty (the width) grows when subtracting, just as it does during addition.

The conclusion, then, is that, when two intervals are either added together or subtracted, the width is the resulting interval will be the sum of their widths.

4.4 WRITEUP Exercise 2.10: Intervals spanning zero and error signalling

4.4.1 Problem

Ben Bitdiddle, an expert systems programmer, looks over Alyssa's shoulder and comments that it is not clear what it means to divide by an interval that spans zero. Modify Alyssa's code to check for this condition and to signal an error if it occurs.

4.4.2 Solution

```
(define (spans-zero? x)
  (and (< (lower-bound x) 0) (> (upper-bound x) 0)))

(define (div-interval x y)
  (if (spans-zero? y)
      (error "can't divide by an interval spanning zero")
      (mul-interval x
                    (make-interval (/ 1.0 (upper-bound y))
                                   (/ 1.0 (lower-bound y))))))
```

4.5 TODO Exercise 2.11: Reducing multiplications for `mul-interval`

4.5.1 Problem

In passing, Ben also cryptically comments: "By testing the signs of the endpoints of the intervals, it is possible to break `mul-interval` into nine cases, only one of which requires more than two multiplications." Rewrite this procedure using Ben's suggestion.

4.5.2 Solution

4.6 WRITEUP Exercise 2.12: Center-percentage representations for intervals

4.6.1 Problem

Define a constructor `make-center-percent` that takes a center and a percentage tolerance and produces the desired interval. You must also define a selector `percent` that produces the percentage tolerance for a given interval. The `center` selector is the same as the one shown above.

4.6.2 Solution

```
(define (center i)
  (/ (+ (lower-bound i) (upper-bound i)) 2))

(define (width i)
  (- (upper-bound i) (lower-bound i)))

(define (percent i)
  (* 100.0 (/ (width i) 2 (center i))))

(define (make-center-percent c p)
  (let ((half-width (* c (/ p 100.0))))
    (make-interval (- c half-width) (+ c half-width))))
```

4.7 TODO Exercise 2.13: Approximating small tolerances

4.7.1 Problem

Show that under the assumption of small percentage tolerances there is a simple formula for the approximate percentage tolerance of the product of two intervals in terms of the tolerances of the factors. You may simplify the problem by assuming that all numbers are positive.

4.7.2 Solution

4.8 TODO Exercise 2.14: Problems with the interval representation

4.8.1 Problem

After considerable work, Alyssa P. Hacker delivers her finished system. Several years later, after she has forgotten all about it, she gets a frenzied call from an irate user, Lem E. Tweakit. It seems that Lem has noticed that the formula for parallel resistors can be written in two algebraically equivalent ways:

$$\frac{r_1 r_2}{r_1 + r_2}$$

and

$$\frac{1}{1/r_1 + 1/r_2}$$

He has written the following two programs, each of which computes the parallel-resistors formula differently:

```
(define (par1 r1 r2)
  (div-interval (mul-interval r1 r2)
                (add-interval r1 r2)))
```

```
(define (par2 r1 r2)
  (let ((one (make-interval 1 1)))
    (div-interval one
                  (add-interval (div-interval one r1)
                                (div-interval one r2)))))
```

Lem complains that Alyssa's program gives different answers for the two ways of computing. This is a serious complaint.

Demonstrate that Lem is right. Investigate the behavior of the system on a variety of arithmetic expressions. Make some intervals A and B, and use them in computing the expressions A/A and A/B. You will get the most insight by using intervals whose width is a small percentage of the center value. Examine the results of the computation in center-percent form (see Exercise 2.12).

4.8.2 Solution

Demonstrating that this is the case is straightforward:

```
(define aa (make-interval 9 11))
(define bb1 (make-interval 99 101))
(define bb2 (make-interval 90 110))
(define cc1 (make-interval 999 1001))
(define cc2 (make-interval 900 1100))

(define (examine i)
  (display i)
  (display ": ")
  (display (par1 i i))
  (display " ")
  (display (par2 i i))
  (newline))

(examine aa)
(examine bb1)
(examine bb2)
(examine cc1)
(examine cc2)

(define (exact-div-interval x y)
  (mul-interval x
               (make-interval (/ 1 (upper-bound y))
                              (/ 1 (lower-bound y)))))

(define (exact-par1 r1 r2)
  (exact-div-interval (mul-interval r1 r2)
               (add-interval r1 r2)))

(define (exact-par2 r1 r2)
  (let ((one (make-interval 1 1)))
    (div-interval one
                   (add-interval (exact-div-interval one r1)
                                 (exact-div-interval one r2)))))
```

4.9 TODO Exercise 2.15: Examining the growth of uncertainty

4.9.1 Problem

Eva Lu Ator, another user, has also noticed the different intervals computed by different but algebraically equivalent expressions. She says that a formula to compute with intervals using Alyssa's system will produce tighter error bounds if it can be written in such a form that no variable that represents an uncertain number is repeated. Thus, she says, `par2` is a "better" program for parallel resistances than `par1`. Is she right? Why?

4.9.2 Solution

4.10 TODO Exercise 2.16: On the non-equivalence of algebraic expressions

4.10.1 Problem

Explain, in general, why equivalent algebraic expressions may lead to different answers. Can you devise an interval-arithmetic package that does not have this shortcoming, or is this task impossible? (Warning: This problem is very difficult.)

4.10.2 Solution