SICP Exercise Solutions for Section 1.1

Paul L. Snyder

March 19, 2014

Contents

1 Section 1.1.6

1.1 Exercise 1.1: Interpreting expressions

1.1.1 Problem

Below is a sequence of expressions. What is the result printed by the interpreter in response to each expression? Assume that the sequence is to be evaluated in the order in which it is presented.

```
10
(+ 5 3 4)
(- 9 1)
(/ 6 2)
(+ (* 2 4) (- 4 6))
(define a 3)
(define b (+ a 1))
(+ a b (* a b))
(= a b)
(if (and (> b a) (< b (* a b)))
    b
    a)
(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25))
(+ 2 (if (> b a) b a))
(* (cond ((> a b) a)
         ((< a b) b)
         (else -1))
   (+ a 1))
```

1.1.2 Answer

```
> 10
10
> (+ 5 3 4)
12
> (- 9 1)
8
> (/ 6 2)
3
```

```
> (+ (* 2 4) (- 4 6))
-16
> (define a 3)
> (define b (+ a 1))
> (+ a b (* a b))
19
> (= a b)
#f
> (if (and (> b a) (< b (* a b)))
      b
      a)
4
> (cond ((= a 4) 5)
        ((= b 4) (+ 6 7 a)))
16
```

1.2 Exercise 1.2: Translating to prefix notation

Translate the following expression into prefix form.

$$\frac{5+4+(2-(3=(6+\frac{4}{5})))}{3(6-2)(2-7)}$$

```
(/ (+ 5 4 (- 2 (- 3 (+ 6 (/ 4 5)))))
   (* 3 (- 6 2) (- 2 7)))
```

1.3 Exercise 1.3: Summing the squares of two larger numbers

1.3.1 Problem

Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers.

1.3.2 Answer

```
(define (square a) (* a a))

(define (sum-of-squares a b) (+ (square a) (square b)))

(define (sum-of-squares-of-larger-two a b c)
  (cond
```

```
   ((and (<= a b) (<= a c))  (sum-of-squares b c))
   ((and (<= b a) (<= b c))  (sum-of-squares a c))
   (else                     (sum-of-squares a b))))

(define (sum-of-squares-of-larger-two-alt a b c)
  (if (>= a b)
      (if (>= b c)    ; we know a is in, as it's at least as large
                      ; as one other number
          (sum-of-squares a b)     ; c is out
          (sum-of-squares a c))    ; b is out
      (if (>= a c)    ; we know b is in for this branch, since
                      ;it's strictly larger than a
          (sum-of-squares b a)     ; c is out
          (sum-of-squares b c))))  ; a is out

;; This can be made even shorter by exploiting the return value of if
(define (sum-of-squares-of-larger-two-altalt a b c)
  (if (>= a b)
      (sum-of-squares a (if (>= b c) b c))
      (sum-of-squares b (if (>= a c) a c))))
```

1.4 Exercise 1.4: Evaluating combinations

1.4.1 Problem

Observe that our model of evaluation allows for combinations whose operators are compound expressions. Use this observation to describe the behavior of the following procedure:

```
(define (a-plus-abs-b a b)
  ((if (> b 0) + -) a b))
```

1.4.2 Answer

When the expression containing the `if` operator is evaluated, if `b` is strictly larger than 0, the expression returns `+`, which is used as the operator for the containing expression, adding `a` and `b`. Otherwise, the `if` returns `-`, subtracting `b` from `a`; since `b` is negative in this case (or 0) this increases the value by the same amount as it would have increased were `b` to be positive.

1.5 Exercise 1.5: Applicative order vs normal order

1.5.1 Problem

Ben Bitdiddle has invented a test to determine whether the interpreter he is faced with is using applicative-order evaluation or normal-order evaluation. He defines the following two procedures:

```
(define (p) (p))

(define (test x y)
  (if (= x 0)
      0
      y))
```

Then he evaluates the expression

```
(test 0 (p))
```

What behavior will Ben observe with an interpreter that uses applicative-order evaluation? What behavior will he observe with an interpreter that uses normal-order evaluation? Explain your answer. (Assume that the evaluation rule for the special form 'if' is the same whether the interpreter is using normal or applicative order: The predicate expression is evaluated first, and the result determines whether to evaluate the consequent or the alternative expression.)

1.5.2 Answer

The procedure `p` will cause an infinitely recursive loop if executed. If Ben is using an applicative-order interpreter, the operands will be evaluated first, and the command will enter the loop. If the interpreter is normal order, the value of the operands will not be resolved immediately. Instead, test will be executed. `x` will be resolved in the predicate of the `if`, which succeeds. As the consequent branch is followed, `y` will never be evaluated, and thus `p` will never be executed.

To clarify the difference between normal and applicative order, consider nested applications of the `sq` procedure:

```
(define (sq n) (* n n)
(sq (sq (sq (+ 1 1))))
```

Suppose this is evaluated in normal order (combining some of the expansion steps to save space):

```
1> (sq (sq (sq (+ 1 1))))
2> (* (sq (sq (+ 1 1)))
      (sq (sq (+ 1 1))))
3> (* (* (sq (+ 1 1))
         (sq (+ 1 1))
      (* (sq (+ 1 1))
         (sq (+ 1 1)))
4> (* (* (* (+ 1 1) (+ 1 1))
         (* (+ 1 1) (+ 1 1)))
      (* (* (+ 1 1) (+ 1 1))
         (* (+ 1 1) (+ 1 1))))
5> (* (* (* 2 2) (* 2 2))
      (* (* 2 2) (* 2 2)))
6> (* (* 4 4) (* 4 4)
7> (* 16 16)
8> 256
```

As can be seen, none of the "leaf" additions are evaluated until everything is expanded, so there are eight individual addition and seven multiplication operations performed (as seen in steps 4 through 7). Compare this to an applicative-order evaluation:

```
1> (sq (sq (sq (+ 1 1))))
2> (sq (sq (sq 2)))
3> (sq (sq (* 2 2)))
4> (sq (sq 4))
5> (sq (* 4 4))
6> (sq 16)
7> (* 16 16)
8> 256
```

Only a single addition and three multiplications need to be performed!

2 Section 1.1.7

2.1 Exercise 1.6: `if` and the reason for special forms

2.1.1 Problem

Alyssa P. Hacker doesn't see why `if` needs to be provided as a special form. "Why can't I just define it as an ordinary procedure in terms of `cond?`," she asks. Alyssa's friend Eva Lu Ator claims this can indeed be done, and she defines a new version of `if`:

```
(define (new-if predicate then-clause else-clause)
  (if predicate
      then-clause
      else-clause))
```

Eva demonstrates the program for Alyssa:

```
(new-if (= 2 3) 0 5)
5

(new-if (= 1 1) 0 5)
0
```

Delighted, Alyssa uses `new-if` to rewrite the square-root program:

```
(define (sqrt-iter guess x)
  (new-if (good-enough? guess x)
          guess
          (sqrt-iter (improve guess x)
                     x)))

(sqrt-iter 1 4)

(if true
        guess
        (sqrt-iter (improve guess x)
                   x))


(new-if true
        guess
```

```
        (sqrt-iter (improve guess x)
                   x))
(new-if (good-enough? guess x)
        guess
        (new-if (good-enough? guess x)
                guess
                (sqrt-iter (improve guess x)
                           x)))
```

What happens when Alyssa attempts to use this to compute square roots? Explain.

2.1.2 Answer

In `sqrt-iter`, before `new-if` is called, the arguments are evaluated in applicative order. This includes the recursive call to itself. Thus, even when the good-enough? termination condition is satisfied, the recursion will continue, resulting in an infinite recursion.

2.2 Exercise 1.7: When `good-enough?` is not good enough

2.2.1 Problem

The `good-enough?` test used in computing square roots will not be very effective for finding the square roots of very small numbers. Also, in real computers, arithmetic operations are almost always performed with limited precision. This makes our test inadequate for very large numbers. Explain these statements, with examples showing how the test fails for small and large numbers. An alternative strategy for implementing `good-enough?` is to watch how `guess` changes from one iteration to the next and to stop when the change is a very small fraction of the guess. Design a square-root procedure that uses this kind of end test. Does this work better for small and large numbers?

2.2.2 Answer

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x)
                 x)))
```

```
(define (improve guess x)
  (average guess (/ x guess)))

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

(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))

(define (sqrt1 x)
  (sqrt-iter 1.0 x))

(define (percent-changed new old)
  (/ (- new old) old))

(define (small-change? new-guess old-guess)
  (< (abs (percent-changed new-guess old-guess)) 0.001))

(define (sqrt-scaled x)
  (define (sqrt-iter-scaled current-guess last-guess x)
    (if (smallest-divisor small-change? current-guess last-guess)
        current-guess
        (sqrt-iter-scaled (improve current-guess x)
                          current-guess x)))
  (sqrt-iter-scaled 1.0 100.0 x))

(define sqrt sqrt-scaled)
```

Checking the results:

```
(sqrt 10000000.0)        ; Racket's built-in
(sqrt1 10000000.0)       ; Version from text
(sqrt-scaled 10000000.0) ; Scale-sensitive version
```

```
> (sqrt 10000000.0)        ; Racket's built-in
3162.2776601683795
> (sqrt1 10000000.0)       ; Version from text
3162.277660168379
> (sqrt-scaled 10000000.0) ; Scale-sensitive version
3162.277666486375
```

As can be seen the scale sensitive version does not perform as well at large scales (as the iterative steps are larger, it is somewhat easier to satisfy the `small-change?` predicate.

```
> (sqrt 0.00000001)
0.0001
> (sqrt1 0.00000001)
0.03125010656242753
> (sqrt-scaled 0.00000001)
0.00010000000000082464
```

The value of this version becomes evident at very small scales. The original version from the text is more easily satisfied in this case, as the improvement steps are so tiny.

2.3 Exercise 1.8

2.3.1 Problem

Newton's method for cube roots is based on the fact that if y is an approximation to the cube root of x, then a better approximation is given by the value

$$\frac{x/y^2 + 2y}{3}$$

Use this formula to implement a cube-root procedure analogous to the square-root procedure. (In section 1.3.4we will see how to implement Newton's method in general as an abstraction of these square-root and cube-root procedures.)

2.3.2 Answer

Extending to cube roots is straightforward. Building on the code in the last exercise:

```
(define (cube x)
  (* x x x))

(define (improve-cubic-guess guess x)
  (average guess (/ (+ (/ x (square guess)) (* 2 guess)) 3)))
```

```
(define (cbrt-iter-scaled current-guess last-guess x)
  (if (small-change? current-guess last-guess)
      current-guess
      (cbrt-iter-scaled (improve-cubic-guess current-guess x) current-guess x)))

(define (cbrt x)
  (cbrt-iter-scaled 1.0 100.0 x))
```

This gives results that do indeed look like an approximation to the cubic root of a number. Comparing against the results of the Racket `expt` function:

```
> (cbrt 27)
3.0018696341051916
> (expt 27 1/3)
3.0

> (cbrt 1000)
10.009419213708078
> (expt 1000 1/3)
9.999999999999998

> (cbrt 100000000)
464.61792889946435
> (expt 100000000 1/3)
464.15888336127773

> (cbrt 0.0000001)
0.004645991760883334
> (expt 0.0000001 1/3)
0.00464158883361278
```