Structure and Interpretation of Computer Programs

Exercise solutions

Konstantinos Chousos

July 6, 2023

Introduction

This document contains my solutions to the exercises of the *Abelson, Sussman & Sussman (2002) Structure and Interpretation of Computer Programs, MIT Press* book. It will be updated to reflect my current progress. Each subsection consists of an exercise. The exercise is copied over from the book and presented in italics, followed by my (hopefully correct) solution.

Instead of reading from the PDF, I study SICP through its TeXInfo format[1]. This way, I read the material, interact with the REPL[2] and take notes all from my editor of choice, Emacs. For more details about my workflow, you can read my accompanying blog post[3].

This document is written in Org-Mode. The code blocks are evaluated using Org-Babel, which is also used to *tangle* the resulting sicp.rkt file and *weave* this very PDF, as in literate programming parlance.

Racket is used for implementing the exercises, using the `sicp` package[4]. Other possible options are guile and GNU/MIT-scheme.

[1]`https://www.neilvandyke.org/sicp-texi/`
[2]see "Read-Eval-Print-Loop"
[3]`https://kchousos.github.io/posts/sicp-in-emacs/`
[4]`https://docs.racket-lang.org/sicp-manual/index.html`

Chapter 1

1.1

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.

The interpreter's expected results are in the comments under each expression.

```
10
;; 10
(+ 5 3 4)
;; 12
(- 9 1)
;; 8
(/ 6 2)
;; 3
(+ (* 2 4) (- 4 6))
;; 6
(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) 6)
      ((= b 4) (+ 6 7 a))
      (else 25))
;; 16
(+ 2 (if (> b a) b a))
;; 6
(* (cond ((> a b) a)
         ((< a b) b)
         (else -1))
   (+ a 1))
;; 16
```

1.2

Translate the following expression into prefix form.

$$\frac{5 + 4 + (2 - (3 - (6 + 4/5)))}{3(6 - 2)(2 - 7)} \tag{1}$$

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

Results

```
-37/150
```

1.3

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

Instead of searching for the two larger numbers, we just find the smallest. This way, we don't need to think about duplicate cases etc.

```
(define (proc a b c)
  (cond ((and (> a c) (> b c))
         (+ (* a a) (* b b)))
        ((and (> a b) (> c b))
         (+ (* a a) (* c c)))
        ((and (> b a) (> c a))
         (+ (* b b) (* c c)))))
```

Listing 1: Sum of the two-out-of-three larger numbers

```
<<larger-squares>>
(proc 4 2 3)
```

Listing 2: Testing the previous procedure

Results

```
25
```

1.4

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))
```

The operator to be used depends on the evaluation of the `if` statement.

1.5

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:

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

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

Then he evaluates the expression

```scheme
(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.)

The two cases are examined below:

- Applicative-Order evaluation

  The interpreter will *"evaluate the arguments and then apply"*, so it will first evaluate 0, then it will try to evaluate `(p)` which will result in an infinite loop. That is because in `(define (p) (p))` `(p)`'s definition is itself.

- Normal-Order evaluation

  The interpreter will transform `(test 0 (p))` to

```scheme
(if (= 0 0)
      0
      (p))
```

and then will evaluate the expression. Since `(= 0 0)` evaluates to `#t`, `(p)` will never be evaluated because it is not needed.

1.6

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)
  (cond (predicate then-clause)
        (else 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)))
```

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

Since `new-if` is user-defined, all parameters will be evaluated before it is applied. Therefore, `(sqrt-iter (improve guess x) x)` will always be evaluated, and since it calls itself it will run indefinitely.

TODO 1.7

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?

1.8

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

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

Use this formula to implement a cube-root procedure analogous to the square-root procedure. (In section 1-3-4 we will see how to implement Newton's method in general as an abstraction of these square-root and cube-root procedures).

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

(define (good-enough? guess x)
  (< (abs (- (cube guess) x)) 0.0000001))

(define (improve guess x)
  (/ (+ (/ x (* guess guess)) (* 2 guess)) 3))

(define (cube-root-iter guess x)
  (if (good-enough? guess x)
      guess
      (cube-root-iter (improve guess x) x)))

(define (cube-root x)
  (cube-root-iter 1.0 x))
```

```scheme
<<cube-root>>
(cube-root 27)
```

Results

```
3.0000000000000977
```