My SICP exercises

Evgeny Markin

2023

Contents

Preface

Exercises are from ”Structure and Interpretation of Computer Programs” by Abelson and Sussmans. Most of those exercises are just the programs that need to be written, but some require you to write something down as a text.

Chapter 1

Building Abstractions with Procedures

1.1

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

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

Verified most of them in guile.

1.2

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))
```

1.4

Observe that our model for combinations whose operators are compound expres- sions. Use this observation to describe the behavior of the following procedure:

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

Inside if returns plus or minus depending on the value of b, thus this funtion returns

$$a + b$$

in case if $b > 0$ and

$$a - b$$

otherwise, making it effectively equivalent to

$$a + |b|$$

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:

```
(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: Thee predicate expression is evaluated first, and the result determines whether to evaluate the consequent or the alternative expression.)

If the interpreter uses the applicative-order evaluation, then it would firstly evaluate the expressions in the test, before applying test. In this case we go into an infinite loop, since (p) produces anoter (p).

If the interpretes uses the normal-order evaluateion, then it would apply the test procedure, then evaluated if and returned with an answer.

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)))
```

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

The problem, that arises here is that the standart if (and cond for that matter) evaluates the expression only in the case if the clause evauates to a true value. In the case of new-if we are going to evaluate the arguments of the expression, and get into an infinite loop.

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?

Suppose that we want to evaluate the square root of a number 0.00001. In the case of our square root we won't go through the first iteration and collect the first incorrect guess.

For example, guile returns

```
(msqrt 0.000001) = 0.031260655525445276 = (msqrt 0.0000001)
```

which is obviously incorrect.

Also, because the precision is liminted, we've got that

```
(msqrt 10000000000000)
```

enters an infinite loop. Thus it is reasonable to set new good-enough? to check the precision to be relative to the magnitute of the x. (maybe linj)

```
(define (improved-good-enough?  guess x)
  (< (abs (- (square guess) x)) (/ x 10000)))
```

This effectively solves the above-mentioned problems, but in order to have the precise best approximation of the square root, we've got to have implementation-specific epsilons, relative to the guess variables.

1.8

GOTO progs/ch01/newton.scm

1.9

Each of the following two procedures defines a method for adding two positive integers in terms of the procedures inc, which increments its argument by 1, and dec, which decrements its argument by 1.

```
(define (+ a b)
(if (= a 0) b (inc (+ (dec a) b))))
(define (+ a b)
(if (= a 0) b (+ (+ dec a) (inc b))))
```

Using the substitution model, illustrate the process generated by each procedure in evaluating (+ 4 5). Are these processes iterative or recursive?

```
(+ 4 5)
(inc (+ (dec 4) 5))
(inc (+ 3 5))
(inc (inc (+ (dec 3) 5)))
(inc (inc (+ 2 5)))
(inc (inc (inc (+ (dec 2) 5))))
(inc (inc (inc (+ 1 5))))
(inc (inc (inc (inc (+ (dec 1) 5)))))
(inc (inc (inc (inc (+ 0 5)))))
(inc (inc (inc (inc 5))))
(inc (inc (inc 6)))
(inc (inc 7))
(inc 8)
9
```

```
(+ 4 5)
(+ (dec 4) (inc 5))
(+ 3 6)
(+ (dec 3) (inc 6))
(+ 2 7)
(+ (dec 2) (inc 7))
(+ 1 8)
(+ (dec 1) (inc 8))
(+ 0 9)
9
```

Therefore we've got a recursive process in the first case, and iterative in the second

1.10

The following procedure computes a mathematical function called Ackermann's function.

```
(define (A x y)
(cond ((= y 0) 0)
((= x 0) (* 2 y))
((= y 1) 2)
(else (A (- x 1) (A x (- y 1))))))
```

What are the values of the following expressions?

```
(A 1 10) -> (A 0 (A 1 9))) -> ... ->  (A 0 (A ) -> 4
(A 2 4) -> (A 1 (A
```