Assignment 1 Solutions

Nathan Hitchings

January 1, 2019

1 Exercise 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.

1.1 Solution

```
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
```

2 Exercise 1.2

Translate the following expression into prefix form

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

2.1 Solution

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

3 Exercise 1.3

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

3.1 Solution

```
(define (sum-of-squares-of-larger x y z)
  (cond ((and (>= x z) (>= y z))
         (sum-of-squares x y))
        ((and (>= x y) (>= z y))
         (sum-of-squares x z))
        (else (sum-of-squares y z))))
```

4 Exercise 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))
```

4.1 Solution

This procedure uses the sign of `b` to decide whether to add `b` to `a` or subtract `b` from `a`. If b is greater than zero then `b` is added if `b` is less than zero then `b` is subtracted essentially resulting in adding the absolute value of `b` to `a`.

5 Exercise 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: The predicate expression is evaluated first, and the result determines whether to evaluate the consequent or the alternative expression.)

5.1 Solution

Using applicative-order evaluation, the expression

```
(test 0 p)
```

will be evaluated by first evaluating the arguments. Thus, evaluating `p` leads to an infinite recursion and the evaluation will never complete. Using normal-order evaluation, we will first expand the expression then reduce. When this happens, the `if` form's condition will evaluate to true and the result of the expression will be zero with no attempt to evaluate p.

6 Exercise 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.

6.1 Solution

Since `new-if` is a procedure and must evaluate all of its arguments, `sqrt-iter` will loop forever. This happens because `sqrt-iter` must be evaluated before `new-if` can be applied thus calling `new-if` again which will try to evaluate `sqrt-iter` again and so on.

7 Exercise 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?

7.1 Solution

For very small numbers we have a very large relative error. For instance, in computing `(sqrt 0.001)` the difference between the `good-enough?` result is 70% different than the original argument. For very large numbers the machine precision is unable to represent small differences between large numbers.

```
(define (good-enough? previous-guess next-guess)
  (< (abs (- previous-guess next-guess)) (* 0.001 next-guess)))
(define (sqrt-iter first-guess second-guess x)
  (if (good-enough? first-guess second-guess)
      second-guess
      (sqrt-iter second-guess (improve second-guess x) x)))
(define (sqrt x)
  (sqrt-iter x 1.0 x))
```

8 Exercise 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.

8.1 Solution

```
(define (square x)
  (* x x))
(define (good-enough? previous-guess next-guess)
  (< (abs (- previous-guess next-guess)) (* 0.001 next-guess)))
```

```
(define (improve guess x)
  (/ (+ (/ x (square y)) (* y 2.0)) 3.0))
(define (cbrt-iter first-guess second-guess x)
  (if (good-enough? first-guess second-guess)
      second-guess
      (cbrt-iter second-guess (improve second-guess x) x)))
(define (cube-root x)
  (cbrt-iter x 1.0 x))
```