Contents

1.1 The Elements of Programming

Exercise 1.1

Question

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))
```

Answer

```
10

10

(+ 5 3 4)

12

(- 9 1)

8

(+ (* 2 4) (- 4 6))

6

(define a 3)

#<unspecified>
```

```
(define b (+ a 1))
```

```
#<unspecified>
```

```
(+ 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)
         ((> b a) b)
         (else -1))
   (+ a 1))
```

```
16
```

Exercise 1.2

Question

Translate the following expression into prefix form:

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

Answer

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

```
-37/150
```

Exercise 1.3

Question

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

Answer

```
(define (square x) (* x x))

(define (maxsq x y z)
  (square (cond ((> x y) x)
                ((> x z) x)
                (else 0))))

(define (summaxsq a b c)
  (+ (maxsq a b c)
     (maxsq b a c)
     (maxsq c a b)))

(summaxsq 1 2 3)
```

```
13
```

Exercise 1.4

Question

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))
```

Answer

If `b` defined in the parameter of `a-plus-abs-b` is greater than 0, the operator for calculating `a b` is $+$, else it is $-$.

Exercise 1.5

Question

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.)

Answer

1. Applicative-order Evaluation First, this expression:

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

   evaluates the arguments `0` and `(p)` first, which can be seen as `0`, which is not defined, and `(p)`, which is defined as `(p)`. This can be seen as:

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

   Then, the procedure `test` which can be seen as `(if (= x 0) 0 y)` is applied, which applies the evaluated arguments as the procedure arguments. This can be seen as:

```
(define (test 0 (p))
  (if (= 0 0) 0 (p)))
```

2. Normal-order Evaluation First, this expression:

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

   substitues the expressions inside until it only involves primitive expressions. This can be seen as:

   ```
   (define (test 0 (define (p) (p)))
     (if (= 0 0) 0 (p)))
   ```

   Then only after that the made expressions inside `test` are evaluated, which there is one i.e. `(define (p) (p))` then the operands itself. This can be seen as:

   ```
   (define (test 0 (p))
     (if (= 0 0) 0 (p)))
   ```

Exercise 1.6

Question

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.

Answer

Since Scheme uses the applicative-order evaluation as mentioned and new-if is a procedure, the operands will be evaluated first, so `(sqrt-iter guess x)` evaluates `guess` and `x` first. By doing this,

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

from `good-enough?` evaluates `guess x` first. Also an operand which is the parent procedure itself there calls recursively, which creates an infinite recursion because it calls itself every time `new-if` is called, which is its procedure.

Exercise 1.7

Question

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?

Answer

Based on the example in computing square roots:

```
(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 (sqrt x)
  (sqrt-iter 1.0 x))
```

For small numbers, the function works as intended as shown:

```
(sqrt 9)
```

```
3.00009155413138
```

However, large numbers like the one shown below:

```
(sqrt 12345678901234)
```

does not work properly, as the program appears to never end when evaluated. An alternative strategy for `good-enough` as mentioned would be to stop when the change is a very small fraction of `guess`.

```
(define (good-enough? guess x)
  (< (abs (- guess
             (improve guess x)))
     (/ guess 1000000)))
```

Using the new function, the program evaluates properly as shown:

```
(sqrt 12345678901234)
```

```
3513641.86446291
```

Exercise 1.8

Question

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 1.3.4 we will see how to implement Newton's method in general as an abstraction of these square-root and cube-root procedures.)

Answer

The formula is implemented into `cbrt-improve` where y becomes `guess` and x becomes the desired number to evaluate.

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

Then these are the correspondent functions for it:

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

(define (cbrt-good-enough? guess x)
  (< (abs (- guess
             (cbrt-improve guess x)))
     (/ guess 1000000)))

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

The result of the functions is shown below for the cube root of 64:

```
(cbrt 64)
```

```
4.000000000076121
```