1 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 expression? 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)
a

(define b (+ a 1))
b

(+ 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:

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

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

1.3

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

```
(define (f x y z)
  (define (sqr a) (* a a))
  (cond ((and (>= x y) (>= y z)) (+ (sqr x) (sqr y)))
        ((and (> x y) (>= z y)) (+ (sqr x) (sqr z)))
        (else (+ (sqr y) (sqr z)))))
```

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 procedure sums a with the absolute value of b.

1.5

Comment on the differences between applicative and normal-order evaluation of the following:

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

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

(test 0 (p))
```

Applicative-order will recurse into infinity as the condition's alternative, `(p)`, evaluated. Normal order will defer calculation of `(p)` and instead return the condition's consequent `0` as `x == 0` is evaluated first.

1.6

Given:

```
(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))
```

Comment on the implementation with `new-if`:

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

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

The special form for `if` does not evaluate the `else-clause`, whereas in `new-if` the `else-clause` is evaluated in applicative-order, so the program recurses infinitely with closer and closer `x` values. (This resembles **1.5**)

1.7

Explain why the `good-enough?` criterium from **1.6** is not good for computing roots for small and large numbers. Design an alternative criterium that instead checks how much `guess` changes from iteration to iteration.

Hard-coding the value `0.001` will not be appropriate for different degrees of magnitude of the root we are trying to estimate. For example, given a radicand of 0.0001, the program will return 0 as an acceptable guess. With a very large radicand, it is possible that floating point operations necessarily "overshoot" the acceptable bound, even if our guess is the true root - that is, after squaring both, the loss of precision may make it impossible to have a difference whose absolute magnitude falls bellow the arbitrary `0.001`. Assuming that the root-finding method converges after infinite iterations:

```
(define (better-good-enough? previous-guess guess)
  (< (abs (/ (- previous-guess guess) guess)) 0.001))

(define (sqrt-iter guess x)
  (define (-sqrt-iter previous-guess guess x)
    (if (better-good-enough? previous-guess guess)
        guess
        (-sqrt-iter guess (improve guess x) x)))
  ; arbitrarily set to 2x on first iteration to not
  ; divide by 0 or be considered a "close" guess
  (-sqrt-iter (* 2 guess) guess x))
```

1.8

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

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

Implement a cube-root procedure analogous to the square root one.

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

(define (qbrt-iter guess x)
  (define (-qbrt-iter previous-guess guess x)
    (if (better-good-enough? previous-guess guess)
        guess
        (-qbrt-iter guess (improve guess x) x)))
  (-qbrt-iter (* 2 guess) guess x))
```

1.9

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 decreases 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))))
```

Illustrate the process generated by each procedure in evaluating `(+ 4 5)`. Are these procedures iterative or recursive?

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

First is recursive, second iterative.

1.10

The following procedure computes 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))))))
```

Consider the following procedures:

```
(define (f n) (A 0 n))
(define (g n) (A 1 n))
(define (h n) (A 2 n))
```

Give concise mathematical definitions for each.

$$f(n) = 2n \tag{1}$$
$$g(n) = 2^n \tag{2}$$
$$h(n) = 2^{h(n-1)} \tag{3}$$

1.11

A function f is defined by the rule that $f(n) = n$ if $n < 3$ and $f(n) = f(n-1)+2f(n-2)+3f(n-3)$ if $n \geq 3$. Write a procedure that computes f with by means of a recursive process. Write a procedure that computes f with by means of an iterative process.

```
(define (recur-f n)
  (if (< n 3)
      n
      (+ (recur-f (- n 1))
         (* 2 (recur-f (- n 2)))
         (* 3 (recur-f (- n 3)))
      )))

(define (iter-f n)
  (define (-iter-f a b c k)
    (if (= k 0)
        a
        (-iter-f (+ a (* 2 b) (* 3 c)) a b (- k 1))))
  (if (< n 3)
      n
      (-iter-f 2 1 0 (- n 2))))
```

1.12

Write a procedure that computes elements of Pascal's triangle by means of a recursive process.

```
(define (p-tri i j)
  (cond ((or (< j 0) (< i 0) (> j i)) 0)
        ((= i 0) 1)
        (else (+ (p-tri (- i 1) (- j 1))
                 (p-tri (- i 1) j)))))
```

1.13

Prove that $Fib(n)$ is the closest integer to $\phi^n/\sqrt{5}$, where $\phi = (1+\sqrt{5})/2$.

Let $\psi = (1-\sqrt{5})/2$. Show that $Fib(n) = (\phi^n - \psi^n)/\sqrt{5}$.
For the base case of $n = 1$, $(\phi - \psi)/\sqrt{5} = (\frac{(1+\sqrt{5})}{2} - \frac{(1-\sqrt{5})}{2}))/\sqrt{5} = (\frac{2\sqrt{5}}{2})/\sqrt{5} = 1 = Fib(1)$.
Show that $(\phi^{n+1} - \psi^{n+1})/\sqrt{5} = (\phi^n - \psi^n)/\sqrt{5} + (\phi^{n-1} - \psi^{n-1})/\sqrt{5}$.

$$\phi^{n+1} - \psi^{n+1} = (\phi^n - \psi^n) + (\phi^{n-1} - \psi^{n-1}) \tag{4}$$
$$\phi^{n+1} - \psi^{n+1} = (1+1/\phi)\phi^n - (1+1/\psi)\psi^n \tag{5}$$
$$(\phi - 1 - 1/\phi)\phi^n = (\psi - 1 - 1/\psi)\psi^n \tag{6}$$
$$\phi - 1 - 1/\phi = \frac{-1+\sqrt{5}}{2} - \frac{2}{1+\sqrt{5}} \tag{7}$$
$$= \frac{5-1-4}{2(1+\sqrt{5})} = 0 \tag{8}$$
$$\psi - 1 - 1/\psi = \frac{-1-\sqrt{5}}{2} - \frac{2}{1-\sqrt{5}} \tag{9}$$
$$= \frac{5-1-4}{2(1+\sqrt{5})} = 0 \tag{10}$$
$$0(\phi^n) = 0(\psi^n) \tag{11}$$

To show that $Fib(n)$ is the closest integer to $\phi^n/\sqrt{5}$, rewrite $\phi^n/\sqrt{5} = Fib(n) + \psi^n/\sqrt{5}$. If $|\psi^n/\sqrt{5}| \leq |\psi/\sqrt{5}| < \frac{1}{2}$, $Fib(n)$ is the closest integer to $\phi^n/\sqrt{5}$. Square to show $\frac{3-\sqrt{5}}{10} < \frac{1}{4}$. Since $2 < \sqrt{5} < 3$, $\frac{3-\sqrt{5}}{10} < \frac{1}{10} < \frac{1}{4}$. ■

1.14

What are the orders of growth in space and the number of steps as the cent input grows in the `count-change` procedure?

Consider a call to `(cc n 5)` for some large `n`. In a tree diagram of evaluation, the call to `(cc n 5)` can be considered a root that branches out into a call to `(cc (- n (denom 5)) 5)` and `(cc n 4)`. On the `(cc (- n 5) 5)` branch, with $d(x)$ denoting the denomination of coin number x, we will have $\lfloor n/d(5) \rfloor$ branching nodes plus a terminal node where $n - \lceil n/d(5) \rceil * d(5) \leq 0$. Each of these branching nodes spawns a sequence where we evaluate the algorithm with a coin of denomination 4 (i.e. 25). The pattern repeats itself, with each root spaning a sequence of $\lfloor k/d(4) \rfloor$ branching nodes where a quarter is subtracted from each k. In turn, each of these will branch into a sequence of $\lfloor k/d(3) \rfloor$ branching nodes and so on until the smallest denomination of coin, $d(0)$ (i.e. 1). Ignoring the initial evaluation and terminal nodes, the number of steps required will be proportional to $\Pi_{c=1}^{c=5} \lfloor n/d(i) \rfloor \propto n^5$, so the process is $\Theta(n^5)$.
Space complexity is proportional to the depth of the tree, which is greatest along the branch where the amount is reduced by the smallest coin denomination, $d(1) = 1$. This will require 5 levels of depth to get down from $d(5)...d(1)$ and then n evaluations to terminate the branch. Ignoring the constant term, space complexity is thus $\Theta(n)$.

1.15

The sine of an angle (specified in radians) can be computed by making use of the approximation $\sin x \approx x$ if x is sufficiently small, and by the trigonometric identity

$$\sin x = 3 \sin \frac{x}{3} - 4 \sin^3 \frac{x}{3}$$

to reduce the argument of sin. Consider the following procedure for estimating $\sin x$:

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

(define (p x) (- (* 3 x) (* 4 (cube x))))

(define (sine angle)
    (if (not (> (abs angle) 0.1))
    angle
    (p (sine (/ angle 3.0)))))
```

a. How many times is the procedure `p` applied when `(sine 12.15)` is evaluated?

5 times ($12.15/3^4 > 0.1$, $12.15/3^5 < 0.1$).

b. What is the order of growth in space and number of steps as a function of a generated by the `sine` procedure when `(sine a)` evaluated?

Each time a triples, we need to invoke `sine` one more time, so the growth in number of steps of `sine` is $\Theta(\log_3 a) = \Theta(\log a)$. For each subsequent step, size increases constantly (another application of `p` in the call stack), so the growth in size is also $\Theta(\log a)$.

More specifically, $T(a)$ is the number of steps required if it is the smallest integer s.t. $1 = \lceil (\frac{a}{0.1})/3^{T(a)} \rceil$. So, $1 \geq (\frac{a}{0.1})/3^{T(a)} \rightarrow 3^{T(a)} \geq a/0.1 \rightarrow T(a) \geq \log_3(a/0.1) = \log_3(a) - \log_3(0.1) \geq k_2 \log(a)$ for some k_2 as a grows. Since $T(a)$ is the smallest integer that fulfils the conditions, $1 < (\frac{a}{0.1})/3^{T(a)-1} \rightarrow T(a) < \log_3(a/0.1) + 1 \leq k_1 \log(a)$ for some k_1. Finally, $k_1 \log(a) \geq T(a) \geq k_2 \log(a) \rightarrow T(a) = \Theta(log(a))$.

1.16

Design a procedure that evolves an iterative exponentiation process that uses successive squaring and uses a logarithmic number of steps.

```
(define (exp-iter a b k)
  (cond ((= k 0) a)
        ((> (remainder k 2) 0) (exp-iter (* a b) b (- k 1)))
        (else (exp-iter a (* b b) (/ k 2)))))

(define (exp b k) (exp-iter 1 b k))
```

1.17

Using operations for addition, doubling, and halving an integer, design a recursive multiplication procedure that uses a logarithmic number of steps.

```
(define (* a b)
  (cond ((= b 0) 0)
        ((> (remainder b 2) 0) (+ a (* a (- b 1))))
        (else (dub (+ a (* a (- (halve b) 1)))))))
```

1.18

Write an iterative version of the procedure from **1.17**.

```
(define (* a b)
    (cond ((= b 0) 0)
          ((= b 1) a)
          ((> (remainder b 2) 0) (+ a (* a (- b 1))))
          (else (* (dub a) (halve b)))))
```

1.19

The Fibonacci sequence is a special case of the transformation T_{pq} s.t. $a \leftarrow qb + qa + pa$, $b \leftarrow pb + qa$ with $(p, q) = (0, 1)$. Show that applying T_{pq} twice is equivalent to applying a single transformation $T_{p'q'}$ of the same form, and compute p' and q' in terms of p and q. Use the formula for p' and q' to complete the snippet for a procedure that evaluates the nth Fibonacci number in log time.

$$a_2 = qb_1 + qa_1 + pa_1 = q(pb_0 + qa_0) + (p + q)(qb_0 + qa_0 + pa_0) \tag{12}$$

$$= 2q^2a_0 + 2pqa_0 + p^2a_0 + 2pqb_0 + q^2b_0 \tag{13}$$

$$= (q^2 + 2pq)b_0 + (q^2 + 2pq)a_0 + (p^2 + q^2)a_0 \tag{14}$$

$$b_2 = pb_1 + qa_1 = p(pb_0 + qa_0) + q(qb_0 + qa_0 + pa_0) \tag{15}$$

$$= p^2b_0 + q^2b_0 + q^2a_0 + 2pqa_0 = (p^2 + q^2)b_0 + (q^2 + 2pq)a_0 \tag{16}$$

$$p' = (p^2 + q^2) \tag{17}$$

$$q' = (q^2 + 2pq) \tag{18}$$

$$\tag{19}$$

```scheme
(define (fib n)
  (fib-iter 1 0 0 1 n))

(define (fib-iter a b p q count)
    (cond ((= count 0) b)
          ((even? count) (fib-iter a
                                   b
                                   (+ (* p p) (* q q)) ; p'
                                   (+ (* q q) (* 2 p q)) ; q'
                                   (/ count 2)))
          (else (fib-iter (+ (* b q) (* a q) (* a p))
                          (+ (* b p) (* a q))
                          p
                          q
                          (- count 1)))))
```

1.20

Using the substitution method with normal-order evaluation, show the process generated by `(gcd 206 40)`. How many `remainder` operations are performed? How many are performed in the applicative order?

This one seems a bit tedious TBH. Better off using applicative because the `remainder` will have to be calculated for each occurrence of `b`, which occurs in the predicate, consequent, and alternative. (TODO a better answer for n of calls without writing it all out?)

1.21

Use the `smallest-divisor` procedure to find the smallest divisor of each of the following numbers: 199, 1999, 19999.

199, 1999, 7.

1.22

Use some of the provided code to write a procedure `search-for-primes` that checks the primality of consecutive odd integers in a specified range.

```
(define (search-for-primes a b)
  (cond ((= (remainder a 2) 0) (search-for-primes (+ a 1) b))
        ((< a b)
          (timed-prime-test a)
          (search-for-primes (+ a 2) b))))
```

Does the evaluation time have a growth order of $\Theta(\sqrt{n})$?

Searching for primes around 10^{12} takes approx. 0.76 sec on my machine but with some variability. Around 10^{13} it grows to approx. 2.4 sec, so spot on $\sqrt{10} * 0.76 = 2.4$. In the 10^{14} range observing approx. 7.9 sec, with projected $\sqrt{10} * 2.4 = 7.6$. There is likely extra overhead for storing bigger numbers that is not accounted for in the by simply multiplying through by $\sqrt{10}$.