Solved Exercises for Structure and Interpretation of Computer Programs

Eduardo Acuña Yeomans

2014

1. Building Abstractions with Procedures

Exercise 1.01

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 ]=> 10

;Value: 10

1 ]=> (+ 5 3 4)

;Value: 12

1 ]=> (- 9 1)

;Value: 8

1 ]=> (/ 6 2)

;Value: 3

1 ]=> (+ (* 2 4) (- 4 6))

;Value: 6

1 ]=> (define a 3)

;Value: a

1 ]=> (define b (+ a 1))

;Value: b

1 ]=> (+ a b (* a b))

;Value: 19

1 ]=> (= a b)

;Value: #f
```

```
1 ]=> (if (and (> b a) (< b (* a b)))
          b
          a)

;Value: 4

1 ]=> (cond ((= a 4) 6)
            ((= b 4) (+ 6 7 a))
            (else 25))

;Value: 16

1 ]=> (+ 2 (if (> b a) b a))

;Value: 6

1 ]=> (* (cond ((> a b) a)
                ((< a b) b)
                (else -1))
          (+ a 1))

;Value: 16
```

Exercise 1.02

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)))
```

Exercise 1.03

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

```
;;; square : number -> number
;;; calculate the square of the number x
;;; use : (square -5) ]=> 25
(define (square x) (* x x))

;;; sum-of-squares-largest-two : number x number x number -> number
;;; sum the squares of the two largest numbers of the given three
;;; use : (sum-of-squares-largest-two 5 3 4) ]=> 41
(define (sum-of-squares-largest-two a b c)
  (if (> a b)
      (+ (square a) (square (if (> b c) b c)))
      (+ (square b) (square (if (> a c) a c)))))
```

Exercise 1.04

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 operation we want to compute is $a + |b|$. It doesnt alter the result if b is negative or not, for this reason $a + |b| = a + |-b|$. The case when b is positive we can just sum the two numbers, the other case can be thought like $a + |-b| = a + b = a - (-b)$, so we have to subtract $-b$ to a.

Exercise 1.05

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.)

In an applicative-order evaluator the arguments are evaluated first, therefor the evaluation of `(p)` in the procedure call `(test 0 (p))` will fall in an infinite loop of doom and the whole program will crash.

In a normal-order evaluator the arguments are evaluated when they are needed; in the given snippet of code, `x` will be evaluated to `0` in the conditional of the `test` procedure, then `(= 0 0)` will evaluate to `#t` and `0` will be the result. So `(p)` never gets evaluated.

Exercise 1.06

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:

```
1 ]=> (new-if (= 2 3) 0 5)

;Value: 5
```

```
1 ]=> (new-if (= 1 1) 0 5)

;Value: 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.

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

;;; Procedures for computing the square-root of a number

(define (average a b)
  (/ (+ a b) 2))

(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))

(define (improve guess x)
  (average guess (/ x guess)))

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

Scheme uses applicative-order evaluation, this means that the evaluator is going to evaluate all the arguments of the function.

In the `sqrt-iter` procedure, `new-if` takes three arguments:

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

There is no problem evaluating the first argument (that's what the evaluator does with the special form `if`) and the second argument (because is just a number). However, the third argument is a recursive call to the procedure that evaluated the `new-if` function; wich causes an infinite evaluation of the `new-if` function call inside the body of this procedure. That's why the `if` special form is neccessary for the `sqrt-iter` procedure to work as espected.

Note: If the interpreter had a normal-order evaluator, the `new-if` procedure would work just fine.

Exercise 1.07

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?

When we calculate the square of a very small number n ($0 < n < 1$) the result will be smaller than n. So it is very likely that the `(abs (- (square guess) x))` will be smaller than `0.001`. Take for example the `(sqrt 0)` it will be approximated to `0.03125` when it should be `0`. The `sqrt-mod` procedure get's it a little better, approximating $\sqrt{0}$ to `0.0009765625`.

The other case is when we calculate square roots of very big numbers. The problem in this case is that when the procedure squares the guess it might result in a very very big number, big enough so that the computer can't represent it.

The following code gives the definitions of the square root approximation with two different criteria for *what is good enough?*. Two more procedures are presented; one for measuring the error of a given square root procedure and a given number, the other for comparing how better or worst is one square root procedure to the other.

```
;;; Computing square roots (book procedures)
(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))

;;; Modified square-root procedure
(define (sqrt-iter-mod guess old-guess x)
  (if (good-enough-mod? guess old-guess)
      guess
      (sqrt-iter-mod (improve guess x)
                     guess
                     x)))

(define (good-enough-mod? guess old-guess)
  (<= (abs (- guess old-guess)) .001))
```

```
(define (sqrt-mod x)
  (sqrt-iter-mod 1.0 0.0 x))


;;; Verify how good is a square root aproximation
(define (sqrt-error proc x)
  (define result (proc x))
  (- x (square result)))

(define (sqrt-compare proc1 proc2 x)
  (define error1 (sqrt-error proc1 x))
  (define error2 (sqrt-error proc2 x))
  (exact->inexact (/ error1 error2)))
```

Exercise 1.08

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.)

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

(define (good-enough? guess n)
  (> .001 (abs (- n (* guess guess guess)))))

(define (cube-root n)
  (define (cube-root-iterative guess n)
    (if (good-enough? guess n)
        guess
        (cube-root-iterative (improve guess n) n)))
  (cube-root-iterative 1.0 n))
```

Exercise 1.09

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 (inc int)
  (1+ int))

(define (dec int)
  (1- int))

(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?

The first recursive procedure describes a recursive process:

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

The second recursive procedure describes an iterative process:

```
;(+ 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)
;Value: 9
```

Exercise 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?

```
1 ]=> (A 1 10)

;Value: 1024
```

```
1 ]=> (A 2 4)

;Value: 65536

1 ]=> (A 3 3)

;Value: 65536
```

Consider the following procedures, where `A` is the procedure defined above:

```
(define (f n) (A 0 n))
(define (g n) (A 1 n))
(define (i n) (A 2 n))
(define (k n) (* 5 n n))
```

Give concise mathematical definitions for the functions computed by the procedures `f`, `g`, and `h` for positive integer values of `n`. For example, `(k n)` computes $5n^2$.

$$f(n) = 2n$$

$$g(n) = 2^n$$

$$i(n) = 2_1^{2_2^{\cdot^{\cdot^{\cdot^{2n}}}}}$$

$$k(n) = 5n^2$$

Exercise 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 by means of a recursive process. Write a procedure that computes f by means of an iterative process.

```
;;; Procedure by means of a recursive process.
(define (f n)
  (if (< n 3)
      n
      (+ (f (- n 1))
         (* 2 (f (- n 2)))
         (* 3 (f (- n 3))))))

;;; Procedure by means of an iterative process.
(define (f n)
  (define (f-iter n i f1 f2 f3)
    (cond ((< n 3) n)
          ((= i n) (+ f1 (* 2 f2) (* 3 f3)))
          (else (f-iter n (+ i 1) f2 f3 (+ f1 (* 2 f2) (* 3 f3))))))
  (f-iter n 3 0 1 2))
```

Exercise 1.12

The following pattern of numbers is called Pascal's triangle.

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
      . . .
```

The numbers at the edge of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it. Write a procedure that computes elements of Pascal's triangle by means of a recursive process.

```
(define (pascal r c)
  (if (or (= r c) (= c 0))
      1
      (+ (pascal (- r 1) c) (pascal (- r 1) (- c 1)))))
```

Exercise 1.13

Prove that $Fib(n)$ is the closest integer to $\phi^n/\sqrt{5}$, where $\phi = (1+\sqrt{5})/2$. Hint: Let $\psi = (1-\sqrt{5})/2$. Use induction and the definition of the Fibonacci numbers (see section 1.2.2) to prove that $Fib(n) = (\phi^n - \psi^n)/\sqrt{5}$.

$$\begin{aligned}
Fib(1) =& ((1+\sqrt{5})/2 - (1-\sqrt{5})/2)/\sqrt{5} \\
=& ((1+\sqrt{5}-1+\sqrt{5})/2)/\sqrt{5} \\
=& (2\sqrt{5}/2)/\sqrt{5} \\
=& \sqrt{5}/\sqrt{5} \\
=& 1
\end{aligned}$$

We assume $Fib(n-1)$ and $Fib(n-2)$ satisfy the equation.

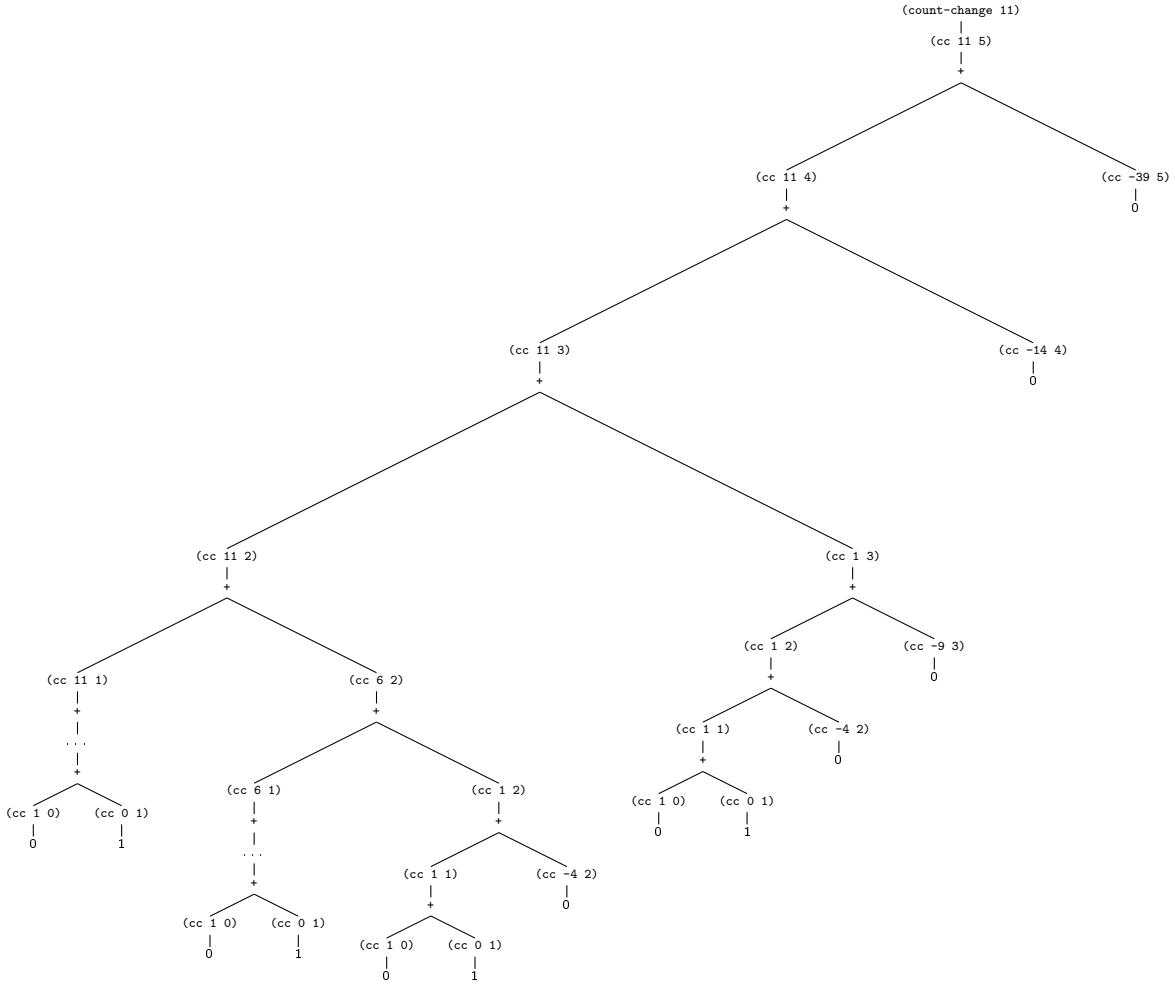
$$\begin{aligned}
Fib(n) =& Fib(n-1) + Fib(n-2) \\
=& \frac{\phi^{n-1} - \psi^{n-1}}{\sqrt{5}} + \frac{\phi^{n-2} - \psi^{n-2}}{\sqrt{5}} \\
=& \frac{1}{\sqrt{5}}(\phi^{n-1} - \psi^{n-1} + \phi^{n-2} - \psi^{n-2}) \\
=& \frac{1}{\sqrt{5}}(\phi^{n-1} + \phi^{n-2} - \psi^{n-1} - \psi^{n-2}) \\
=& \frac{1}{\sqrt{5}}((1+\phi)\phi^{n-2} - (1+\psi)\psi^{n-2}) \\
=& \frac{1}{\sqrt{5}}((\phi^2)\phi^{n-2} - (\psi^2)\psi^{n-2}) \\
=& \frac{1}{\sqrt{5}}(\phi^n - \psi^n) \\
=& \frac{\phi^n - \psi^n}{\sqrt{5}}
\end{aligned}$$

Exercise 1.14

Draw the tree illustrating the process generated by the count-change procedure of Section 1.2.2 in making change for 11 cents. What are the orders of growth of the space and number of steps used by this process as the amount to be changed increases?

```
(define (count-change amount) (cc amount 5))
(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (else (+ (cc amount
                     (- kinds-of-coins 1))
                 (cc (- amount
                        (first-denomination
                         kinds-of-coins))
                     kinds-of-coins)))))
(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))

;;; (count-change 11) produces:
(+ (+ (+ (+ (+ 0
                (+ 0
                   (+ 0
                      (+ 0
                         (+ 0
                            (+ 0
                               (+ 0
                                  (+ 0
                                     1)))))))))))
            (+ (+ 0
                  (+ 0
                     (+ 0
                        (+ 0
                           (+ 0
                              (+ 0
                                 1))))))
               (+ (+ 0
                     1)
                  0)))
         (+ (+ (+ 0
                  1)
               0)
            0))
      0)
   0)
```



Exercise 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 the trigonometric identity

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

to reduce the size of the argument of sin. (For purposes of this exercise an angle is considered "sufficiently small" if its magnitude is not greater than 0.1 radians.) These ideas are incorporated in the following procedures:

```
(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?

The procedure p is calles each time the angle is reduced by a factor of 3 in the sine call until the absolute value of the angle is less or equall than 0.1.

- 12,15/3 = 4,05 (first call)
- 4,05/3 = 1,35 (second call)
- 1,35/3 = 0,45 (third call)
- 0,45/3 = 0,15 (fourth call)
- 0,15/3 = 0,05 (fifth call)

The procedure is called 5 times.

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

Each recursive call reduces the problem size in one that is three times more simple. For that reason the number of procedure calls is $\log_3(angle)$, therefor the order of growth in *space* is $O(\log_3(n))$. The order of growth of the number of steps is the number of times the procedure p is called (The steps p computes are $O(1)$), so the *number of steps* are $O(\log_3(n))$.

Exercise 1.16

Design a procedure that evolves an iterative exponentiation process that uses successive squaring and uses a logarithmic number of steps, as does `fast-expt`. (Hint: Using the observation that $(b^{n/2})^2 = (b^2)^{n/2}$, keep, along with the exponent n and the base b, an additional state variable a, and define the state transformation in such a way that the product ab^n is unchanged from state to state. At the beginning of the process a is taken to be 1, and the answer is given by the value of a at the end of the process. In general, the technique of defining an *invariant* quantity that remains unchanged from state to state is a powerful way to think about the design of iterative algorithms.)

Using *successive squaring* means that rather than computing b^8 with 8 multiplications ($b^8 = b \times b \times b \times b \times b \times b \times b \times b$), we can compute it as three multiplications $b^2 = b \times b$, $b^4 = b^2 \times b^2$ and $b^8 = b^4 \times b^4$. The procedure `fast-expt` describes a recursive process that grows with $\Theta(\log_2(n))$ in steps and space:

```
(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))
```

Using the hint of the exercise, two procedures are written, the first one is my first attempt to solving the problem.

```
;;; Iterative exponentiation process using successive squaring
(define (fast-expt b n)
  (define (fast-expt-iter b n a)
    (cond ((= n 0)              a)
          ((and (even? n)
                (not (= n 2))) (fast-expt-iter b (/ n 2) (* a (square b))))
          (else                (fast-expt-iter b (- n 1) (* a b)))))
  (fast-expt-iter b n 1))

;;; Another way to write the procedure is:
(define (fast-expt b n)
  (define (fast-expt-iter b n a)
    (cond ((= n 0)   a)
          ((even? n) (fast-expt-iter (square b) (/ n 2) a))
          (else      (fast-expt-iter b (- n 1) (* a b)))))
  (fast-expt-iter b n 1))
```

Exercise 1.17

The exponentiation algorithms in this section are based on performing exponentiation by means of repeated multiplication. In a similar way, one can perform integer multiplication by means of repeated addition. The following multiplication procedure (in which it is assumed that our language can only add, not multiply) is analogous to the expt procedure:

```
(define (* a b)
  (if (= b 0)
      0
      (+ a (* a (- b 1)))))
```

This algorithm takes a number of steps that is linear in b. Now suppose we include, together with addition, operations double, which doubles an integer, and halve, which divides an (even) integer by 2. Using these, design a multiplication procedure analogous to fast-expt that uses a logarithmic number of steps.

```
(define (double x) (+ x x))

(define (halve x) (/ x 2))

(define (* a b)
  (cond ((= b 0) 0)
        ((even? b) (double (* a (halve b))))
        (else (+ (* a (- b 1)) a))))
```

Exercise 1.18

Using the results of Exercise 1.16 and Exercise 1.17, devise a procedure that generates an iterative process for multiplying two integers in terms of adding, doubling, and halving and uses a logarithmic number of steps.

```
(define (* a b)
  (define (*-iter a b c)
    (cond ((= b 0) c)
          ((even? b) (*-iter (double a) (halve b) c))
          (else (*-iter a (- b 1) (+ c a)))))
  (*-iter a b 0))
```

Exercise 1.19

There is a clever algorithm for computing the Fibonacci numbers in a logarithmic number of steps. Recall the transformation of the state variables a and b in the fib-iter process of Section 1.2.2: $a \leftarrow a + b$ and $b \leftarrow a$. Call this transformation T, and observe that applying T over and over again n times, starting with 1 and 0, produces the pair $Fib(n+1)$ and $Fib(n)$. In other words, the Fibonacci numbers are produced by applying T^n, the n^{th} power of the transformation T, starting with the pair $(1, 0)$. Now consider T to be the special case of $p = 0$ and $q = 1$ in a family of transformations T_{pq}, where T_{pq} transforms the pair (a, b) according to $a \leftarrow bq + aq + ap$ and $b \leftarrow bp + aq$. Show that if we apply such a transformation T_{pq} twice, the effect is the same as using a single transformation $T_{p'q'}$ of the same form, and compute p' and q' in terms of p and q. This gives us an explicit way to square these transformations, and thus we can compute T^n using successive squaring, as in the fast-expt procedure. Put this all together to complete the following procedure, which runs in a logarithmic number of steps:

```
(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
                   <??> ; compute p'
                   <??> ; compute q'
                   (/ count 2)))
        (else (fib-iter (+ (* b q) (* a q) (* a p))
                        (+ (* b p) (* a q))
                        p
                        q
                        (- count 1)))))
```

$T_{pq}^2 = T_{pq}(bq + aq + ap, bp + aq)$

$T_{pq}^2 = T_{p'q'} = (b(q^2+2pq)+a(q^2+2pq)+a(p^2+q^2), b(p^2+q^2)+a(q^2+2pq)) = (bq'+aq'+ap, bp'+aq')$

Where $p' = p^2 + q^2$ and $q' = q^2 + 2pq$

```
(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))
                   (+ (* q q) (* 2 p q))
                   (/ count 2)))
        (else (fib-iter (+ (* b q) (* a q) (* a p))
                        (+ (* b p) (* a q))
                        p
                        q
                        (- count 1)))))
```

Exercise 1.20

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

The process that a procedure generates is of course dependent on the rules used by the interpreter. As an example, consider the iterative gcd procedure given above. Suppose we were to interpret this procedure using normal-order evaluation, as discussed in Section 1.1.5. (The normal-order-evaluation rule for if is described in Exercise 1.5.) Using the substitution method (for normal order), illustrate the process generated in evaluating (gcd 206 40) and indicate the remainder operations that are actually performed. How many remainder operations are actually performed in the normal-order evaluation of (gcd 206 40) ? In the applicative-order evaluation?

Exercise 1.21

Exercise 1.22

Exercise 1.23

Exercise 1.24

Exercise 1.25

Exercise 1.26

Exercise 1.27

Exercise 1.28

Exercise 1.29

Exercise 1.30

Exercise 1.31

Exercise 1.32

Exercise 1.33

Exercise 1.34

Exercise 1.35

Exercise 1.36

Exercise 1.37

Exercise 1.38

Exercise 1.39

Exercise 1.40

Exercise 1.41

Exercise 1.42

Exercise 1.43

Exercise 1.44

Exercise 1.45

Exercise 1.46

Interesting explorations: Constructing the $Fib(n)$ closed formula