Solutions to Structure and Interpretation of Computer Programs

Fabienne Ducroquet

February 11, 2018

Contents

Introduction

Solutions to most of the exercises of *Structure and Interpretation of Computer Programs*, second edition, by Harold Abelson and Gerald Jay Sussman with Julie Sussman.

The answers to these exercises have been tested with the Scheme interpreter from Gambit Scheme, at the exception of the exercises from section 2.2.4 about the picture language, which have been written with Racket, using the `graphics.ss` library.

1 Building Abstractions with Procedures

1.1 The Elements of Programming

1.1.1 Expressions

This subsection contains no exercises.

1.1.2 Naming and the Environment

This subsection contains no exercises.

1.1.3 Evaluationg Combinations

This subsection contains no exercises.

1.1.4 Compound Procedures

This subsection contains no exercises.

1.1.5 The Substitution Model for Procedure Application

This subsection contains no exercises.

1.1.6 Conditional Expressions and Predicates

Exercise 1.1

Check with a scheme interpreter.

Exercise 1.2

A minimally indented version:

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

A heavily indented version:

```
(/ (+ 5
      4
      (- 2
         (- 3
            (+ 6
```

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

Exercise 1.3

```
(define (sum-two-larger-squares a b c)
  (cond ((and (<= a b) (<= a c))
         (sum-of-squares b c))
        ((and (<= b a) (<= b c))
         (sum-of-squares a c))
        (else
          (sum-of-squares a b))))
```

Exercise 1.4

If $b > 0$, `(a-plus-abs-b a b)` returns $a + b$; otherwise it returns $a - b$. In other words, `(a-plus-abs-b a b)` returns $a + |b|$.

Exercise 1.5

With applicative-order evaluation, the interpreter tries to evaluate `(p)`, which results in an infinite loop, so the interpreter never returns (or returns an error).

With normal-order evaluation, the interpreter doesn't try to evaluate `(p)` until it's really needed, but that never happens since `(= x 0)` returns true, so the call returns 0.

1.1.7 Example: Square Roots by Newton's Method

Exercise 1.6

When Alyssa attempts to use this to compute square roots, the program never returns.

Explication: `new-if` is an ordinary procedure, so each time it is called, the evaluator tries to evaluate all of its arguments. In particular, each call to `sqrt-iter` will cause one more call to `sqrt-iter`, whether `(good-enough? guess x)` returns true or not, so the evaluator ends up in an infinite loop.

Exercise 1.7

A possible solution:

```
(define (sqrt-iter guess x)
  (define new-guess (improve guess x))
  ; let would be better than define here, but it has not been introduced in
  ; the book yet.
  (if (good-enough? guess new-guess)
      guess
      (sqrt-iter new-guess
                 x)))

(define (good-enough? guess new-guess)
```

```
(<= (abs (- guess new-guess))
    (* 1e-5 guess)))
```

Let's call x the number whose root we want to compute.
With the initial `good-enough?` test:

- If x is very small, the difference between the guess and x becomes smaller than 0.001 (or any number we would replace 0.001 with, for small enough numbers) while the guess is still several times larger than $\sqrt{x}$, or even orders of magnitude away from it.

  Example: `(sqrt 0.0001)` returns 0.03230844833048122 instead of 0.01 because `(abs (- (square 0.03230844833048122) 0.0001))` returns $9.438358335233747e-4$.

- If x is very large, the difference between the guess and x will always be found to be larger than 0.001 (or any number we would replace 0.001 with, for large enough numbers) because ($x-$ any number) can not be expressed to the precision required to compare it to 0.001, so the call never returns.

  Example: `(sqrt 1e+129)` does not return, while `(sqrt 1e+128)` returns a correct answer almost instantly[1].

With the modified versions of `good-enough?` and `sqrt-iter`, the above examples work.

Exercise 1.8

Here is a solution based on the solution of exercise 1.7.

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

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

(define (cuberoot x)
  ; Call cube-iter only with positive values because otherwise
  ; (improve guess x) can return 0, e.g. with (improve-guess 1.0 -2)
  (if (>= x 0)
    (cube-iter 1.0 x)
    (- (cube-iter 1.0 (- x)))))
```

[1] These values are implementation-dependent.

1.1.8 Procedures as Black-Box Abstractions

This subsection contains no exercises.

1.2 Procedures and the Processes They Generate

1.2.1 Linear Recursion and Iteration

Exercise 1.9

With the first procedure:

```
(+ 4 5)
(inc (+ 3 5))
(inc (inc (+ 2 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)
9
```

With the second procedure:

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

The first process is recursive, the second is iterative.

Exercise 1.10

Using the interpreter, we obtain $1024 = 2^{10}$ for `(A 1 10)`, and $65536 = 2^{16}$ for `(A 2 4)` and `(A 3 3)`.

By definition of the Ackermann function, `(A 0 n)`, i.e. `(f n)` computes $2n$.

If $n > 0$, `(g n)` computes 2^n.

Proof. By definition, `(g 1)` equals 2, and for $n > 1$, `(A 1 n)` equals `(A 0 (A 1 (- n 1)))`. Since `(A 0 n)` computes $2n$, the result follows by mathematical induction. □

If $n > 0$, `(h n)` computes $2 \uparrow\uparrow n$, that is, $2^{2^{2^{\cdots}}}$ with n copies of 2.

Proof. This is true for $n = 1$ by definition. For $n > 1$, `(A 2 n)` is equal to `(A 1 (A 2 (- n 1)))`. The result follows by mathematical induction using the previous result. □

Remark. `(A 3 3)` returns $2^{16} = 2^{2^{2^2}}$ as well. The recursion beginning with `(A 3 n)` with $n > 1$ gives `(A 2 (A 3 (- n 1)))`, so according to the previous result, `(A 3 n)` is obtained from `(A 3 (- n 1))` by computing $2^{2^{2^{\cdots}}}$ with a tower of `(A 3 (- n 1))` 2s, so since `(A 3 1)` is 2, `(A 3 2)` is $2^2 = 4$, and `(A 3 3)` is $2^{2^{2^2}}$. The general value of `(A 3 n)` can be noted $2 \uparrow\uparrow\uparrow n$, or $2 \uparrow^3 n$.

This notation can be extended for all ms, so `(A m n)` computes $2 \uparrow^m n$.

1.2.2 Tree Recursion

Example: Counting change

Exercise 1.11

Procedure computing f 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 computing f by means of a iterative process :

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

Exercise 1.12

Here is an example of a solution :

```
(define (c n k)
  (cond ((or (> k n) (< k 0) (< n 0))
         0)
        ((or (= k 0) (= n k))
         1)
        (else (+ (c (- n 1) k)
                 (c (- n 1) (- k 1))))))
```

The argument n is the line number from the top starting from 0, and k is the column number from the left starting from 0.

Exercise 1.13

Let's prove that for any $n \geq 0$, $\text{Fib}(n) = {(\phi^n - \psi^n)}/{\sqrt{5}}$, where $\phi = {\left(1+\sqrt{5}\right)}/{2}$ and $\psi = {\left(1-\sqrt{5}\right)}/{2}$.

It's true for $n = 0$ and $n = 1$.

Let's assume that it's true for any $k < n$. We have :

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

But ϕ and ψ are the roots of the equation $x^2 - x - 1 = 0$, in other words, $\phi^2 = \phi + 1$ and $\psi^2 = \psi + 1$, hence $\text{Fib}(n) = {(\phi^n - \psi^n)}/{\sqrt{5}}$.

Furthermore, $|1 - \sqrt{5}| < 2$, so for any $n \geq 0$, $|1 - \sqrt{5}|^n < 2^n$, so dividing by 2^n, we get $|\psi^n| < 1$, and by dividing by $\sqrt{5}$, $|{\psi^n}/{\sqrt{5}}| < {1}/{\sqrt{5}}$. Since ${1}/{\sqrt{5}} < {1}/{2}$, we have $|{\psi^n}/{\sqrt{5}}| < {1}/{2}$ for any $n \geq 0$, which means that $\text{Fib}(n)$ is the closest integer to ${\phi^n}/{5}$.

1.2.3 Orders of Growth

Exercise 1.14

The space required is proportional to the maximum depth of the tree, so it grows as $\Theta(n)$.

For the time complexity, let's use the mathematical notation $\text{cc}(n, k)$ rather than `(cc n k)`.

The time complexity for $\text{cc}(n, 1)$ grows as $\Theta(n)$.

If we note v the denomination of the k-th coin, we have:

$$\begin{aligned}
\text{cc}(n, k) &= \text{cc}(n - v, k) + \text{cc}(n, k-1) \\
&= \text{cc}(n - 2v, k) + 2\,\text{cc}(n, k-1) \\
&= \ldots \\
&= \text{cc}(n - \left\lceil \frac{n}{v} \right\rceil v, k) + \left\lceil \frac{n}{v} \right\rceil \text{cc}(n, k-1)
\end{aligned}$$

Since $n - \left\lceil \frac{n}{v} \right\rceil v \leq 0$, the time complexity of $\text{cc}(n, k)$ is proportional to n times the time complexity of $\text{cc}(n, k-1)$. As a consequence, the time complexity for 5 kinds of coins grows as $\Theta(n^5)$.

Exercise 1.15

a. If the argument is greater than 0.1, `p` is called once, and the argument is divided by three. So the number of steps required is the smallest integer n such that ${12.15}/{3^n} < 0.1$, or equivalently $121.5 < 3^n$. The smallest such n is 5, so `p` is called 5 times when `(sine 12.15)` is evaluated.

b. By the same calculation as above, if $a > 0.1$, the number of steps is the smallest n such that $10\,a < 3^n$. By taking the logarithm, we get $\log(10) + \log(a) < n \log(3)$, so $n = \lceil (\log(10) + \log(a)) / \log(3) \rceil$.

Therefore, the number of steps has order of growth $\Theta(\log(n))$. The space required is proportional to the number of steps, so its order of growth is the same.

1.2.4 Exponentiation

Exercise 1.16

A possible solution to compute exponentials in a logarithmic number of steps iteratively:

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

Exercise 1.17

A recursive process that multiplies two non-negative integers using a logarithmic number of steps.

```
(define (*-rec a b)
  (define (double n)
    (* 2 n))
  (define (halve n)
    (/ n 2))
  (cond ((= b 0) 0)
        ((= b 1) a)
        ((even? b) (*-rec (double a) (halve b)))
        (else (+ a
                 (*-rec a (- b 1))))))
```

Exercise 1.18

An iterative process that multiplies two non-negative integers using a logarithmic number of steps.

We keep a state variable c such that $ab + c$ is constant at each call of the inner function.

```
(define (*-iter a b)
  (define (double n)
    (* 2 n))
  (define (halve n)
    (/ n 2))
  (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

By calculation, we get $p' = p^2 + q^2$ and $q' = q^2 + 2pq$, so the procedure becomes:

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

1.2.5 Greatest Common Divisors

Exercise 1.20

With normal-order evaluation, `(gcd 206 40)` expands to `(gcd 40 (remainder 206 40))`, a `remainder` operation is performed to test whether the remainder is null, then the expression expands to `(gcd (remainder 206 40) (remainder 40 (remainder 206 40)))`. Two `remainder` operations are performed to test whether the second argument is null, and the expression expands to

```
(gcd (remainder 40 (remainder 206 40))
     (remainder (remainder 206 40) (remainder 40 (remainder 206 40))))
```

Four new executions of `remainder` are necessary to determine that the second argument is not null, and the expression becomes:

```
(gcd (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))
     (remainder (remainder 40 (remainder 206 40))
                (remainder (remainder 206 40)
                           (remainder 40 (remainder 206 40)))))
```

Seven executions of `remainder` are necessary to determine that the second argument is null, and the GCD is computed with four executions of `remainder`. In total, 18 `remainder` operations are performed in the normal-order evaluation.

With applicative-order evaluation, `(gcd 206 40)` expands to `(gcd 40 6)`, then to `(gcd 6 4)`, `(gcd 4 2)`, `(gcd 2 0)` and to 2. One `remainder` operation is performed each time `b` is not null, so four such operations are performed.

1.2.6 Example: Testing for Primality

Exercise 1.21

The smallest divisors of 199, 1999 and 19999 are 199, 1999 and 7 respectively.

Exercise 1.22

Example solution:

```
(define (search-for-primes start end)
  (define (iter start)
    (cond ((<= start end)
           (timed-prime-test start)
           (iter (+ 2 start)))
          (else
            (display "\n"))))
  (if (even? start)
    (iter (+ start 1))
    (iter start)))
```

Nowadays, it's necessary to use numbers much larger than those suggested in the book to test the prediction about the timing, but the data support the $\sqrt{n}$ prediction.

The result is compatible with the notion that programs run in time proportional to the number of steps required for the computation.

Exercise 1.23

The next procedure is:

```
(define (next n)
  (if (= n 2)
    3
    (+ n 2)))
```

and `smallest-divisor` becomes:

```
(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor)  n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (next test-divisor)))))
```

The modified version does not run twice as fast, but only about 1.7 times as fast as the original version. This is because time is necessary to apply the `next` procedure at each step.

Exercise 1.24

The only change needed is to replace `prime?` with `fast-prime?` using an arbitrary number of tests (100 here) in `start-prime-test`.

```
(define (start-prime-test n start-time)
  (if (fast-prime? n 100)
    (report-prime (- (runtime) start-time))))
```

When the number of digits is doubled, the time needed should be doubled as well since the Fermat test has logarithmic growth. This is what is found experimentally, though again, it's necessary to use numbers much larger than 1000 and 1,000,000 to get significant results.

Exercise 1.25

Alyssa's procedure computes the correct result but it is much slower because it deals with huge numbers, whereas by taking the remainder at each recursion step, the numbers remain smaller than the tested number.

Exercise 1.26

If we use an explicit multiplication rather than calling square, `(expmod base (/ exp 2) m)` is computed twice rather than once at each recursive call with `exp` even, so that the process becomes linear again.

Exercise 1.27

Here is an example of a procedure that tells whether a^n is congruent to a modulo n for every $a < n$:

```
(define (pass-fermat? n)
  (define (iter count)
    (cond ((= count 0) #t)
          ((= (expmod count n n) count)
           (iter (- count 1)))
          (else
           #f)))
  (iter (- n 1)))
```

It returns true for the given Carmichael numbers.

Exercise 1.28

A possible way to implement the Miller-Rabin test is:

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (non-trivial-root-test (expmod base (/ exp 2) m) m))
        (else
          (remainder (* base (expmod base (- exp 1) m))
                     m))))

(define (non-trivial-root-test a n)
  (let ((rm (remainder a n))
        (rm2 (remainder (square a) n)))
    (if (and (not (= rm 1))
             (not (= rm (- n 1)))
             (= rm2 1))
      0
```

```
      rm2)))

(define (miller-rabin-test n)
  (define (try-it a)
    (= (expmod a (- n 1) n) 1))
  (try-it (+ 1 (random (- n 1)))))

(define (fast-prime? n times)
  (cond ((= times 0) #t)
        ((miller-rabin-test n) (fast-prime? n (- times 1)))
        (else #f)))
```

It returns false on the Carmichael numbers listed in footnote 47.

1.3 Formulating Abstractions with Higher-Order Procedures

1.3.1 Procedures as Arguments

Exercise 1.29

Here is a solution:

```
(define (simpson f a b n)
  (define step (/ (- b a) n))
  (define (term-sim k)
    (cond ((= 0 k) (f a))
          ((= n k) (f b))
          ((even? k)
           (* 2 (f (+ a (* k step)))))
          (else
            (* 4 (f (+ a (* k step)))))))
  (* (/ step 3.0)
     (sum term-sim 0 inc n)))
```

Exercise 1.30

A sum procedure generating an iterative process:

```
(define (sum term a next b)
  (define (iter a result)
    (if (> a b)
      result
      (iter (next a) (+ result (term a)))))
  (iter a 0))
```

Exercise 1.31

a. Here is a procedure analogous to `sum` that computes a product, generating a recursive process, and examples of its use to define `factorial` and to compute approximations of π. In the latter case, we use $^{(i-1)(i+1)}/_{i^2} = {}^{i^2-1}/_{i^2}$ as the general term.

```
(define (product term a next b)
  (if (> a b)
    1
    (* (term a)
       (product term (next a) next b))))

(define (factorial n)
  (product identity 1 inc n))

(define (pi-approx n)
  (define (pi-term k)
    (/ (- (square k) 1)
       (square k)))
  (define (pi-next k)
    (+ k 2))
  (* 4.0
     (product pi-term 3 pi-next n)))
```

b. A `product` procedure generating an iterative process.

```
(define (product term a next b)
  (define (iter a result)
    (if (> a b)
      result
      (iter (next a) (* result (term a)))))
  (iter a 1))
```

Exercise 1.32

a. A recursive `accumulate` procedure and the definition of `sum` and `product` using that procedure:

```
(define (accumulate combiner null-value term a next b)
  (if (> a b)
    null-value
    (combiner (term a)
              (accumulate combiner null-value term (next a) next b))))

(define (sum term a next b)
  (accumulate + 0 term a next b))

(define (product term a next b)
  (accumulate * 1 term a next b))
```

b. An iterative version of `accumulate`:

```
(define (accumulate combiner null-value term a next b)
  (define (iter a result)
```

```
      (if (> a b)
        result
        (iter (next a)
              (combiner result (term a)))))
    (iter a null-value))
```

Exercise 1.33

A `filtered-accumulate` procedure generating a recursive process:

```
(define (filtered-accumulate filter combiner null-value term a next b)
  (if (> a b)
    null-value
    (combiner (if (filter a)
                (term a)
                null-value)
              (filtered-accumulate filter
                                   combiner
                                   null-value
                                   term
                                   (next a)
                                   next
                                   b))))
```

A `filtered-accumulate` procedure generating an iterative process:

```
(define (filtered-accumulate filter combiner null-value term a next b)
  (define (iter a result)
    (if (> a b)
      result
      (iter (next a)
            (combiner (if (filter a)
                        (term a)
                        null-value)
                      result))))
  (iter a null-value))
```

a. Assuming `prime?` is already written, the sum of the squares of the prime numbers in the interval a to b can be computed with:

```
(define (sum-squares-primes a b)
  (filtered-accumulate prime? + 0 square a inc b))
```

b. The product of all positive integers less than n that are relatively prime to n can be computed with:

```
(define (product-n-primes n)
  (define (n-prime? i)
    (= (gcd i n) 1))
  (filtered-accumulate n-prime? * 1 identity 1 inc n))
```

1.3.2 Construction Procedures Using `Lambda`

Exercise 1.34

If we try to evaluate `(f f)`, we get an error saying that the operator is not a procedure. The reason is that `(f f)` evaluates to `(f 2)`, which itself evaluates to `(2 2)`, and this operation is impossible since 2 is not a procedure.

1.3.3 Procedures as General Methods

Exercise 1.35

We already noticed in exercise 1.13 that $\phi^2 = \phi + 1$. By dividing this equation by ϕ, we get $\phi = 1 + 1/\phi$.

We can then compute ϕ with the command:

```
(fixed-point (lambda (x) (+ 1 (/ 1 x))) 1.0)
```

Exercise 1.36

Modified version of `fixed-points`:

```
(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (display "Current approximation : ")
      (display next)
      (newline)
      (if (close-enough? guess next)
        next
        (try next))))
  (try first-guess))
```

We can find a solution to $x^x = 1000$ with, for instance, without average damping:

```
(fixed-point (lambda (x) (/ (log 1000) (log x)))
             2.0)
```

And with average damping:

```
(fixed-point (lambda (x) (average x (/ (log 1000) (log x))))
             2.0)
```

The former takes 35 steps while the latter takes 9 steps, so average damping makes the search much faster here.

Exercise 1.37

a. A procedure `cont-frac` generating an iterative process, doing the computation starting from `k`:

```
(define (cont-frac n d k)
  (define (iter res k)
    (if (= k 0)
      res
      (iter (/ (n k)
               (+ (d k) res))
            (- k 1))))
  (iter 0 k))
```

11 steps are necessary to get an approximation that is accurate to 4 decimal places.

b. A procedure `cont-frac` generating a recursive process, doing the computation starting from 1:

```
(define (cont-frac n d k)
  (define (rec count)
    (if (> count k)
      0
      (/ (n count)
         (+ (d count)
            (rec (+ count 1))))))
  (rec 1))
```

Exercise 1.38

The following procedure computes an approximation of e using a k-term finite continued fraction.

```
(define (approx-e k)
  (+ 2
     (cont-frac (lambda (i) 1.0)
                (lambda (i)
                  (if (= (remainder i 3) 2)
                    (* 2 (+ 1 (quotient i 3)))
                    1))
                k)))
```

Exercise 1.39

A possible solution for `(tan-cf x k)`:

```
(define (tan-cf x k)
  (cont-frac (lambda (i)
               (if (= i 1)
                 x
                 (- (square x))))
             (lambda (i)
               (- (* 2 i) 1.0))
             k))
```

1.3.4 Procedures as Returned Values

Exercise 1.40

The procedure `cubic` is:

```
(define (cubic a b c)
  (lambda (x)
    (+ (cube x)
       (* a (square x))
       (* b x)
       c)))
```

Exercise 1.41

The procedure `double`:

```
(define (double f)
  (lambda (x)
    (f (f x))))
```

`(double double)` is a procedure that takes a procedure of one argument as argument and returns a procedure that applies the original procedure four times.

`((double (double double)) f)` evaluates to `((double double) ((double double) f))`, so it returns a procedures that applies `f` $4 \times 4 = 16$ times.

So the value returned by `(((double (double double)) inc) 5)` is 21.

Exercise 1.42

Here is a procedure `compose`:

```
(define (compose f g)
  (lambda (x)
    (f (g x))))
```

Exercise 1.43

A solution generationg a recursive process:

```
(define (repeated f n)
  (if (= n 1)
    f
    (compose f (repeated f (- n 1)))))
```

A solution generationg an iterative process:

```
(define (repeated f n)
  (define (iter count result)
    (if (= count n)
      result
      (iter (+ count 1)
            (compose f result))))
  (iter 1 f))
```

Exercise 1.44

A possible solution is:

```
(define (smooth f)
  (lambda (x)
    (let ((dx 0.00001))
      (/ (+ (f (- x dx))
            (f x)
            (f (+ x dx)))
         3))))
```

The n-fold smoothed function of a function `f` can be obtained with `((repeated smooth n) f)`.

Exercise 1.45

Experimentally, we find that the number of average dampings necessary to compute nth roots in this way is $\lfloor \log n \rfloor$, so the procedure to compute nth roots is:

```
(define (n-root n x)
  (let ((i (floor (/ (log n) (log 2)))))
    (fixed-point ((repeated average-damp i)
                  (lambda (y)
                    (/ x (expt y (- n 1)))))
                 2.0)))
```

Exercise 1.46

A solution for `iterative-improve`:

```
(define (iterative-improve good-enough? improve)
  (define (result guess)
    (if (good-enough? guess)
      guess
      (result (improve guess))))
  result)
```

The `sqrt` procedure of section 1.1.7 becomes:

```
(define (sqrt x)
  (define (improve guess)
    (average guess (/ x guess)))
  (define (good-enough? guess)
    (< (abs (- guess (improve guess)))
       (* 1e-10 guess)))
  (let ((guess 1.0))
    ((iterative-improve good-enough? improve) guess)))
```

The `fixed-point` procedure of section 1.3.3 becomes:

```
(define (fixed-point f first-guess)
  (define tolerance 0.00001)
  (define (close-enough? guess)
    (< (abs (- guess (f guess))) tolerance))
  ((iterative-improve close-enough? f) first-guess))
```

2 Building Abstractions with Data

2.1 Introduction to Data Abstraction

2.1.1 Example: Arithmetic Operations for Rational Numbers

Exercise 2.1

A possibility for a `make-rat` handling both positive and negative arguments:

```
(define (make-rat n d)
  (let ((g (gcd n d))
        (sign (* n d)))
    (if (> sign 0)
      (cons (/ (abs n) g)
            (/ (abs d) g))
      (cons (/ (- (abs n)) g)
            (/ (abs d) g)))))
```

2.1.2 Abstraction Barriers

Exercise 2.2

Exemple implementation for the representation of segments in a plane:

```
(define (make-segment p1 p2)
  (cons p1 p2))

(define (start-segment s)
  (car s))

(define (end-segment s)
  (cdr s))

(define (make-point x y)
  (cons x y))

(define (x-point p)
  (car p))

(define (y-point p)
  (cdr p))
```

```
(define (midpoint-segment s)
  (make-point (average (x-point (start-segment s))
                       (x-point (end-segment s)))
              (average (y-point (start-segment s))
                       (y-point (end-segment s)))))
```

Exercise 2.3

In the following implementation, a rectangle is represented by its two opposite sides, which must have the same orientation. The code makes use of auxiliary procedures defined below.

I added selectors to access each of the vertices of the rectangle to be able to print rectangles in a uniform format.

```
(define (make-rect s1 s2)
  (let ((s3 (make-segment (start-segment s1)
                          (start-segment s2)))
        (s4 (make-segment (end-segment s1)
                          (end-segment s2))))
    (if (or (not (parallel? s1 s2))
            (not (parallel? s3 s4))
            (not (perpendicular? s1 s3)))
      (error "The given segments must correspond to two opposite sides \
              of a rectangle and have the same orientation.")
      (cons s1 s2))))

(define (width-rect rect)
  (length s1))

(define (height-rect rect)
  (let ((s3 (make-segment (start-segment s1)
                          (start-segment s2))))
    (length s3)))

(define (first-point rect)
  (start-segment (car rect)))

(define (second-point rect)
  (end-segment (car rect)))

(define (third-point rect)
  (end-segment (cdr rect)))

(define (fourth-point rect)
  (start-segment (cdr rect)))
```

The procedures that compute the perimeter and area of a rectangle, and the procedure that prints a rectangle, are defined thus:

```
(define (perim-rect rect)
  (* 2 (+ (width-rect rect)
          (height-rect rect))))

(define (area-rect rect)
  (* (width-rect rect)
     (height-rect rect)))

(define (print-rect rect)
  (display "[")
  (print-point-i (first-point rect))
  (display ", ")
  (print-point-i (second-point rect))
  (display ", ")
  (print-point-i (third-point rect))
  (display ", ")
  (print-point-i (fourth-point rect))
  (display "]")
  (newline))
```

Another possibility is to represent a rectangle by its four vertices:

```
(define (make-rect a b c d)
  (let ((s1 (make-segment a b))
        (s2 (make-segment b c))
        (s3 (make-segment c d))
        (s4 (make-segment d a)))
  (if (or (not (perpendicular? s1 s2))
          (not (parallel? s1 s3))
          (not (parallel? s2 s4)))
    (error "The given points don't correspond to the vertices of a rectangle.")
    (cons (cons a b) (cons c d)))))

(define (width-rect rect)
  (length (make-segment (car (car rect))
                        (cdr (car rect)))))

(define (height-rect rect)
  (length (make-segment (cdr (car rect))
                        (car (cdr rect)))))

(define (first-point rect)
```

```
  (car (car rect)))

(define (second-point rect)
  (cdr (car rect)))

(define (third-point rect)
  (car (cdr rect)))

(define (fourth-point rect)
  (cdr (cdr rect)))
```

Yet another possibility is to represent a rectangle by two perpendicular segments with the same origin:

```
(define (make-rect s1 s2)
  (if (not (equal-point? (start-segment s1) (start-segment s2)))
    (error "The two segments must have the same origin.")
    (if (not (perpendicular? s1 s2))
      (error "The two segments must be perpendicular.")
      (cons s1 s2))))

(define (width-rect rect)
  (length (car rect)))

(define (height-rect rect)
  (length (cdr rect)))

(define (first-point rect)
  (start-segment (car rect)))

(define (second-point rect)
  (end-segment (car rect)))

(define (third-point rect)
  (find-fourth-point (car rect) (cdr rect)))

(define (fourth-point rect)
  (end-segment (cdr rect)))
```

The procedures `perim-rect`, `area-rect` and `print-rect` work in all three cases.

Complement:
The code above makes use of the following auxiliary procedures to check that the input is correct, compute the length of a segment, and print points inline for use in `print-rect`:

```
(define (equal-point? p1 p2)
  (and (= (x-point p1) (x-point p2))
```

```scheme
      (= (y-point p1) (y-point p2))))

; Version of print-point without newlines for printing of rectangles.
(define (print-point-i p)
  (display "(")
  (display (x-point p))
  (display ", ")
  (display (y-point p))
  (display ")"))

; Length of a segment.
(define (length s)
  (let ((p1 (start-segment s))
        (p2 (end-segment s)))
  (sqrt (+ (square (- (x-point p1) (x-point p2)))
           (square (- (y-point p1) (y-point p2)))))))

; x-vect and y-vect return the coordinates of the vector with the same starting
; point and ending point as the segment s.
(define (x-vect s)
  (- (x-point (end-segment s))
     (x-point (start-segment s))))

(define (y-vect s)
  (- (y-point (end-segment s))
     (y-point (start-segment s))))

; Mixed product of the vectors defined by s1 and s2.
(define (mixed-product s1 s2)
  (- (* (x-vect s1) (y-vect s2))
     (* (y-vect s1) (x-vect s2))))

; Scalar product of the vectors defined by s1 and s2.
(define (scalar-product s1 s2)
  (+ (* (x-vect s1) (x-vect s2))
     (* (y-vect s1) (y-vect s2))))

(define (parallel? s1 s2)
  (= (mixed-product s1 s2)
     0))

(define (perpendicular? s1 s2)
  (= (scalar-product s1 s2)
     0))

; Takes two segments with the same origin and returns the fourth point of the
; parallelogram they define.
(define (find-fourth-point s1 s2)
  (make-point (+ (x-point (end-segment s1))
```

```
              (x-vect s2))
           (+ (y-point (end-segment s1))
              (y-vect s2))))
```

2.1.3 What Is Meant by Data?

Exercise 2.4

With the representation of pairs given in the exercise, `(cons x y)` is a procedure that takes as its argument a procedure `m` with two arguments and returns the result of the application of `m` to `x` and `y`.

`(car z)` applies the procedure `(cons x y)` to the procedure that returns the first of its arguments, so `(car (cons x y))` yields `x`.

Using the substitution model, the successive steps are:

```
(car (cons x y))
(car (lambda (m) (m x y)))
((lambda (m) (m x y)) (lambda (p q) p))
((lambda (p q) p) x y)
x
```

The corresponding definition of `cdr` is:

```
(define (cdr z)
  (z (lambda (p q) q)))
```

The method to prove that `(cdr (cons x y))` yields `y` is the same as with `car`.

Exercise 2.5

If a and b are known, we can compute 2^a3^b, and since the decomposition of integers as a product of primes is unique, it's possible to find a and b from the value of 2^a3^b.

The procedures `cons`, `car`, and `cdr` corresponding to this representation can be defined as:

```
(define (cons a b)
  (* (expt 2 a)
     (expt 3 b)))

(define (iter power curr base)
  (if (not (= (remainder curr base) 0))
    power
    (iter (+ power 1)
          (/ curr base)
          base)))

(define (car z)
  (iter 0 z 2))

(define (cdr z)
  (iter 0 z 3))
```

Exercise 2.6

The successive substitution steps to evaluate `(add-1 zero)` are:

```
(add-1 zero)

(add-1 (lambda (f) (lambda (x) x)))

(lambda (f)
  (lambda (x)
    (f (((lambda (f)
           (lambda (x) x))
         f)
        x))))

(lambda (f)
  (lambda (x)
    (f ((lambda (x) x)
        x))))

(lambda (f)
  (lambda (x)
    (f x)))
```

In other words, `one` is a procedure that takes a one-argument procedure as its argument and returns it.

The substitution steps to evaluate `(add-1 one)` are:

```
(add-1 one)

(add-1 (lambda (f)
         (lambda (x)
           (f x))))

(lambda (f) (lambda (x)
              (f ((lambda (f)
                    (lambda (x)
                      (f x)))
                  x))))

(lambda (f)
  (lambda (x)
    (f (f x))))
```

In other words, `two` is a procedure that takes a one-argument procedure f as its argument and returns the procedure $f \circ f$ (f applied twice).

From these observations, and after remarking that `zero` is a procedure that takes one argument and always returns the identity procedure, we can make the hypothesis that the nth Church numeral is a procedure that takes a one-argument procedure f as its argument and returns the nth repeated application of f (see exercise 1.43). This can be proved by induction.

Proof. We've already shown that it's true for 0, 1 and 2. Let's assume that it's true for a positive integer n.

From the induction hypothesis, `(n f)` is the nth repeated application of f, so it's obvious that `(lambda (x) (f ((n f) x)))` is the $(n + 1)$th repeatead application of f, so the result is true for $n + 1$, hence it's true for any positive integer n. □

To apply a function $n + m$ times, we just need to apply it m times, and then n times more, so `+` can be defined directly as:

```
(define (+ n m)
  (lambda (f)
    (lambda (x)
      ((n f) ((m f) x)))))
```

2.1.4 Extended Exercise: Interval Arithmetic

Let's first define a function that prints intervals:

```
(define (print-interval x)
  (display "[")
  (display (lower-bound x))
  (display "; ")
  (display (upper-bound x))
  (display "]")
  (newline))
```

Exercise 2.7

Since `make-interval` has been defined as `cons`, `upper-bound` and `lower-bound` can be defined as `cdr` and `car` respectively.

```
(define (upper-bound x)
  (cdr x))

(define (lower-bound x)
  (car x))
```

Exercise 2.8

With the same reasoning as for division, the subtraction of two intervals is the addition of the first with the opposite of the second. The subtraction procedure can thus be defined:

```
(define (sub-interval x y)
  (add-interval x
```

```
                (make-interval (- (upper-bound y))
                               (- (lower-bound y)))))
```

Exercise 2.9

Let $[a; b]$ and $[c; d]$ be two intervals.

Their sum is $[a + c; b + d]$. Its width is ${}^{(b+d)-(a+c)}/_{2} = {}^{(b-a)}/_{2} + {}^{(d-c)}/_{2}$, in other words, the sum's width is the sum of the widths, so it depends only on the widths of the intervals being added.

The difference can be defined as the sum with the opposite, and taking the opposite doesn't change the width, so this is also true for differences.

For multiplication and division, the width of the result also depends on the values of the bounds. For instance, $[1; 2] \times [2; 3] = [2; 6]$, but $[0; 1] \times [2; 3] = [0; 3]$. In both cases, we multiply two intervals of width $1/2$, but the former product has a width of 2 while the latter has a width of $3/2$, so the width of the product is not a function of the widths of the intervals being multiplied.

Since division can be defined as a multiplication, this is also true for division.

Exercise 2.10

The new code of `div-interval` could be:

```
(define (div-interval x y)
  (let ((low-y (lower-bound y))
        (up-y (upper-bound y)))
    (if (and (<= low-y 0) (>= up-y 0))
      (error "Division by zero")
      (mul-interval x
                    (make-interval (/ 1.0 (upper-bound y))
                                   (/ 1.0 (lower-bound y)))))))
```

Exercise 2.11

There are three cases for each interval:

- the lower bound is positive or null;
- the upper bound is negative or null;
- the lower bound is negative and the upper bound is positive.

This results in a total of nine cases, and the only case where the smallest and greatest products can't be deduced from the signs is when both intervals span zero.

A procedure taking this suggestion into account is:

```
(define (mul-interval x y)
  (let ((a (lower-bound x))
        (b (upper-bound x))
        (c (lower-bound y))
        (d (upper-bound y)))
    (cond ((>= a 0)
```

```
          (cond ((<= d 0)
                 (make-interval (* b c) (* a c)))
                ((>= c 0)
                 (make-interval (* a c) (* b d)))
                (else
                 (make-interval (* b c) (* b d)))))
         ((<= b 0)
          (cond ((<= d 0)
                 (make-interval (* b d) (* a c)))
                ((>= c 0)
                 (make-interval (* a d) (* b c)))
                (else
                  (make-interval (* a d) (* a c)))))
         (else
           (cond ((<= d 0)
                  (make-interval (* b d) (* a d)))
                 ((>= c 0)
                  (make-interval (* a d) (* b d)))
                 (else
                   (make-interval
                     (min (* a d) (* b c))
                     (max (* a c) (* b d)))))))))
```

Exercise 2.12

The procedures `make-center-percent` and `percent` can be defined as:

```
(define (make-center-percent c p)
  (let ((width (* c (/ p 100))))
    (make-interval (- c width) (+ c width))))

(define (percent i)
  (* (/ (width i) (center i))
     100))
```

Exercise 2.13

Let c_1, c_2, w_1 and w_2 be the centers and widths of two intervals. We assume that all numbers are positive. The lower and upper bounds of the product are $(c_1 \pm w_1) \times (c_2 \pm w_2) = c_1c_2 \pm (c_1w_2 + c_2w_1) + w_1w_2$.

Since the percentages are small, w_1w_2 is negligible, and the product's width is $w \approx c_1w_2 + c_2w_1$.

Additionally, if we call the percentage tolerances p_1 and p_2 respectively, we have $w_i = c_i \times {}^{p_i}/100$ for $i = 1, 2$.

From there, $w \approx c_1c_2 \times {}^{p_1+p_2}/100$, and c_1c_2 is the product's center, so under the given conditions, the approximate percentage tolerance of the product is the sum of the tolerances of the factors.

Exercise 2.14
TODO

Exercise 2.15
TODO

Exercise 2.16
TODO

2.2 Hierarchical Data and the Closure Property

2.2.1 Representing Sequences

Exercise 2.17
The `last-pair` procedure can be defined as:

```
(define (last-pair items)
  (if (null? (cdr items))
    items
    (last-pair (cdr items))))
```

Exercise 2.18
The `reverse` procedure can be defined as:

```
(define (reverse items)
  (define (iter items result)
    (if (null? items)
      result
      (iter (cdr items) (cons (car items) result))))
  (iter items '()))
```

Exercise 2.19
The procedures can be defined respectively as `car`, `cdr` and `null?`.

```
(define (first-denomination coin-values)
  (car coin-values))

(define (except-first-denomination coin-values)
  (cdr coin-values))

(define (no-more? coin-values)
  (null? coin-values))
```

The order of the list `coin-values` does not affect the answer produced by `cc`, because `cc` gives the total number of combinations, and the relation used for the computation does not depend on a particular order.

Exercise 2.20

A possible solution is:

```
(define (same-parity . numbers)
  (define (iter rest)
    (cond ((null? rest)
           '())
          ((even? (- (car numbers) (car rest)))
           (cons (car rest) (iter (cdr rest))))
          (else
           (iter (cdr rest)))))
  (iter numbers))
```

Exercise 2.21

The completed procedures are:

```
(define (square-list items)
  (if (null? items)
    '()
    (cons (square (car items))
          (square-list (cdr items)))))

(define (square-list items)
  (map square items))
```

Exercise 2.22

With the first procedure, the answer list is in reverse order because the elements are added to it starting from the beginning of the initial list, and the first element added to a list is at its end.

With the second procedure, the result is not a list because `cons` is called with a list as its first argument and the element to add as its second argument. To add an element to a list, the order of the arguments should be the opposite.

Exercise 2.23

Here is a possible implementation of `for-each`:

```
(define (for-each proc items)
  (if (null? items)
    #t
    (begin
      (proc (car items))
      (for-each proc (cdr items)))))
```

2.2.2 Hierarchical Structures

Exercise 2.24

The result given by the interpreter is `(1 (2 (3 4)))`. To represent the corresponding box-and-pointer structure in terms of pairs, one must use the equality of `(list 1 (list 2 (list 3 4)))` and `(cons 1 (cons (list 2 (list 3 4)) nil))`, and similar equalities for the two other lists.

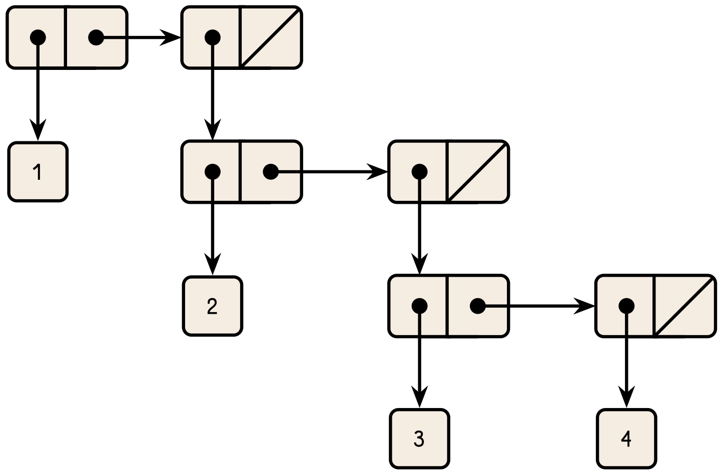

Figure 2.1: Box-and-pointer-structure of `(1 (2 (3 4)))`.

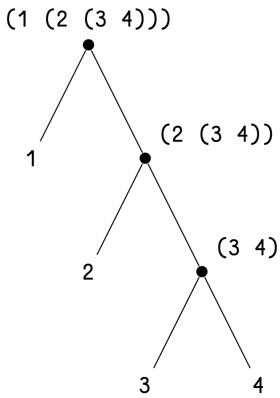

Figure 2.2: Tree representation of `(1 (2 (3 4)))`.

Exercise 2.25

If we call the three lists x, y and z respectively, the combinations `(car (cdr (car (cdr (cdr x)))))`, `(car (car y))` (which can be shortened to `(caar y)`), and `(car (cdr (car (cdr (car (cdr (car (cdr (car (cdr (car (cdr z))))))))))))` or, more simply, `(cadr (cadr (cadr (cadr (cadr (cadr z))))))`, all pick 7 from the lists.

Exercise 2.26

The results printed by the interpreter are `(1 2 3 4 5 6)` for `(append x y)`, `((1 2 3) 4 5 6)` for `(cons x y)` and `((1 2 3) (4 5 6))` for `(list x y)`.

Exercise 2.27

Here is a possible solution for `deep-reverse`:

```
(define (deep-reverse items)
  (define (iter items result)
    (cond ((null? items)
           result)
          ((not (pair? items))
           items)
          (else
            (iter (cdr items) (cons (reverse (car items)) result)))))
  (iter items '()))
```

Exercise 2.28

A possible solution for `fringe` is:

```
(define (fringe tree)
  (cond ((null? tree)
         '())
        ((not (pair? tree))
         (list tree))
        (else
          (append (fringe (car tree))
                  (fringe (cdr tree))))))
```

Exercise 2.29

a. The selectors can be defined as:

```
(define (left-branch mobile)
  (car mobile))

(define (right-branch mobile)
  (cadr mobile))

(define (branch-length branch)
  (car branch))

(define (branch-structure branch)
  (cadr branch))
```

b. The total weight can be computed with:

```
(define (total-weight mobile)
  (+ (branch-weight (left-branch mobile))
     (branch-weight (right-branch mobile))))

(define (branch-weight branch)
```

```
    (let ((struct (branch-structure branch)))
      (if (number? struct)
        struct
        (total-weight struct))))
```

c. Since we defined a `branch-weight` procedure in b., `balanced?` can be defined simply with:

```
(define (balanced? mobile)
  (let ((left (left-branch mobile))
        (right (right-branch mobile)))
    (= (* (branch-length left)
          (branch-weight left))
       (* (branch-length right)
          (branch-weight right)))))
```

d. To convert to the new representation, the only things that need to be changed are the `right-branch` and `branch-structure` selectors:

```
(define (right-branch mobile)
  (cdr mobile))

(define (branch-structure branch)
  (cdr branch))
```

Exercise 2.30

The two `square-list` procedures are identical to the `scale-tree` procedures defined in the text, except that there is only one argument and `(* tree factor)` is replaced with `(square tree)`.

Direct definiton:

```
(define (square-tree tree)
  (cond ((null? tree) '())
        ((not (pair? tree)) (square tree))
        (else
          (cons (square-tree (car tree))
                (square-tree (cdr tree))))))
```

Using `map` and recursion:

```
(define (square-tree tree)
  (map (lambda (sub-tree)
         (if (pair? sub-tree)
           (square-tree sub-tree)
           (square sub-tree)))
       tree))
```

Exercise 2.31

The procedure `tree-map` can be defined without using `map`:

```
(define (tree-map f tree)
  (cond ((null? tree) '())
        ((not (pair? tree)) (f tree))
        (else
          (cons (tree-map f (car tree))
                (tree-map f (cdr tree))))))
```

or using it:

```
(define (tree-map f tree)
  (map (lambda (sub-tree)
         (if (pair? sub-tree)
           (tree-map f sub-tree)
           (f sub-tree)))
       tree))
```

Exercise 2.32

The completed procedure is:

```
(define (subsets s)
  (if (null? s)
    (list '())
    (let ((rest (subsets (cdr s))))
      (append rest
              (map (lambda (x) (cons (car s) x))
                   rest)))))
```

The empty set has only one subset: the empty set.

For a non-empty (finite) set, with elements $\{a_1, ..., a_n\}$, the set of subsets is the reunion of the subsets not containing a_1 and the subsets containing a_1, and the application $S \mapsto S \cup \{a_1\}$ is a bijection between these two sets.

2.2.3 Sequences as Conventional Interfaces

Sequence Operations

Exercise 2.33

The operations can be redefined as:

```
(define (map p sequence)
  (accumulate (lambda (x y) (cons (p x) y)) '() sequence))

(define (append seq1 seq2)
  (accumulate cons seq2 seq1))

(define (length sequence)
  (accumulate (lambda (x y) (+ y 1)) 0 sequence))
```

Exercise 2.34

A polynomial can be evaluated using Horner's rule with the procedure:

```
(define (horner-eval x coefficient-sequence)
  (accumulate (lambda (this-coeff higher-terms)
                (+ (* x higher-terms)
                   this-coeff))
              0
              coefficient-sequence))
```

Exercise 2.35

This can be done with or without `enumate-tree`. Without that function:

```
(define (count-leaves tree)
  (accumulate +
              0
              (map (lambda (subtree)
                     (if (pair? subtree)
                       (count-leaves subtree)
                       1))
                   tree)))
```

The mapped function associates to each subtree its number of leaves: 1 if the subtree has no children, i.e. is a leaf, `(count-leaves subtree)` otherwise.

Exercise 2.36

The procedure `accumulate-n` can be defined as:

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
    '()
    (cons (accumulate op init (map car seqs))
          (accumulate-n op init (map cdr seqs)))))
```

Exercise 2.37

The matrix operation can be define as:

```
(define (dot-product v w)
  (accumulate + 0 (map * v w)))

(define (matrix-*-vector m v)
  (map (lambda (row)
         (dot-product row v))
       m))

(define (transpose mat)
  (accumulate-n cons '() mat))
```

```
(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map (lambda (row)
           (matrix-*-vector cols row))
         m)))
```

Exercise 2.38

The value of `(fold-right / 1 (list 1 2 3))` is $3/2$.

The value of `(fold-left / 1 (list 1 2 3))` is $1/6$.

The value of `(fold-right list nil (list 1 2 3))` is the list `(1 (2 (3 nil)))`.

The value of `(fold-left list nil (list 1 2 p))` is the list `(((nil 1) 2) 3)`.

`fold-right` and `fold-left` produce the same values for any sequence if (and only if) `op` is commutative.

Proof. Suppose `op` commutative. We'll show by induction that `fold-left` and `fold-right` always produce the same results.

`(fold-right op init nil)` and `(fold-left op init nil)` both produce `init`.

Let's assume that `fold-left` and `fold-right` produce the same values for any list of length $n \geq 0$. If `sequence` is a list of length $n + 1$, `(fold-right op init sequence)` equals `(op (car sequence) (fold-right op init (cdr sequence)))`, and `(fold-left op init sequence)` equals `(op (fold-left op init (cdr sequence)) (car sequence))`. It follows from the induction hypothesis and the commutativity of `op` that these two values are equal.

Hence, by induction, `fold-left` and `fold-right` always produce the same results.

Conversely, if `op` is not commutative, there exists elements `a` and `b` such that `(op a b)` is different from `(op b a)`, and these expressions are equal respectively to `(fold-left op a (list b))` and to `(fold-right op a (list b))`, so `fold-left` and `fold-right` don't always produce the same values. □

Exercise 2.39

The procedure `reverse` can be defined in terms of `fold-left` and `fold-right` as:

```
(define (reverse sequence)
  (fold-right (lambda (x y)
                (append y (list x)))
              '()
              sequence))

(define (reverse sequence)
  (fold-left (lambda (x y)
               (cons y x))
             '()
             sequence))
```

Nested Mappings

Exercise 2.40

The procedure `unique-pairs` and the simplified definition of `prime-sum-pairs` are:

```
(define (unique-pairs n)
  (flatmap (lambda (i)
             (map (lambda (j)
                    (list i j))
                  (enumerate-interval 1 (- i 1))))
           (enumerate-interval 1 n)))

(define (prime-sum-pairs n)
  (map make-pair-sum
       (filter prime-sum?
               (unique-pairs n))))
```

Exercise 2.41

A solution using `unique-pairs` from the previous exercise:

```
(define (triple-sum n s)
  (define (equal-sum? triple)
    (= (accumulate + 0 triple)
       s))
  (filter equal-sum?
          (flatmap
            (lambda (pair)
              (map (lambda (i)
                     (cons i pair))
                   (enumerate-interval (+ (car pair) 1) n)))
            (unique-pairs n))))
```

The triples (i, j, k) are in decreasing order because `unique-pairs` returns pairs (j, k) with $j > k$.

Exercise 2.42

A possible solution, with a position represented as the list of the numbers of the lines occupied by the queen in each column. The position of the queen in the first column is at the end of the list because the functions are easier to write this way. I removed the `k` parameter in `safe?` and `adjoin-position` because I didn't need it.

```
(define (queens board-size)
  (define empty-board '())

  (define (safe? positions)
    (define (iter delta-col rest-cols)
      (if (null? rest-cols)
```

```
        #t
        (let ((new-queen-pos (car positions))
              (col-pos (car rest-cols)))
          (and (not (= new-queen-pos col-pos))
               (not (= new-queen-pos (+ col-pos delta-col)))
               (not (= new-queen-pos (- col-pos delta-col)))
               (iter (+ delta-col 1) (cdr rest-cols))))))
  (iter 1 (cdr positions)))

(define (adjoin-position new-row rest-of-queens)
  (cons new-row rest-of-queens))

(define (queens-cols k)
  (if (= k 0)
    (list empty-board)
    (filter
      (lambda (positions) (safe? positions))
      (flatmap
        (lambda (rest-of-queens)
          (map (lambda (new-row)
                 (adjoin-position new-row rest-of-queens))
               (enumerate-interval 1 board-size)))
        (queens-cols (- k 1))))))

(queens-cols board-size))
```

Exercise 2.43

In Louis' program, each call to `(queen-cols k)` calls `(queen-cols (- k 1))` 8 times, instead of only one time with the program in exercise 2.42. Since there are 8 recursion levels, Louis' program will take about 8^8T time to solve the eight-queens puzzle.

2.2.4 Example: A Picture Language

The picture language

Complement:
To test the code in this section, it's necessary to load a module including graphical functions. The solution I chose here is to use DrRacket's `graphics.ss` library. It required me to adapt my solution to exercise 2.46 and to slightly modify the `segments->painter` to use the given viewport. The code from later exercises is sometimes needed to try out some previous code.

Code to put at the beginning of the file:

```
(require (lib "graphics.ss" "graphics"))
(open-graphics)
(define viewport (open-viewport "Window" 400 400))
```

Exercise 2.44

The definition of `up-split` is:

```
(define (up-split painter n)
  (if (= n 0)
    painter
    (let ((smaller (up-split painter (- n 1))))
      (below painter (beside smaller smaller)))))
```

Higher-order operations

Exercise 2.45

The `split` procedure can be defined as:

```
(define (split op-glob op-smaller)
  (define (rec painter n)
    (if (= n 0)
      painter
      (let ((smaller (rec painter (- n 1))))
        (op-glob painter (op-smaller smaller smaller)))))
  rec)
```

Frames

Exercise 2.46

A possible solution if we are not using `graphics.ss`:

```
(define (make-vect x y)
  (cons x y))

(define (xcor-vect v)
  (car v))

(define (ycor-vect v)
  (cdr v))

(define (add-vect v1 v2)
  (make-vect (+ (xcor-vect v1) (xcor-vect v2))
             (+ (ycor-vect v1) (ycor-vect v2))))

(define (sub-vect v1 v2)
  (make-vect (- (xcor-vect v1) (xcor-vect v2))
             (- (ycor-vect v1) (ycor-vect v2))))

(define (scale-vect s v)
```

```
  (make-vect (* s (xcor-vect v))
             (* s (ycor-vect v))))
```

With graphics.ss, we must redefine `make-vect`, `xcor-vect` and `ycor-vect`:

```
(define (make-vect x y)
  (make-posn (* 300 x)
             (* 300 (- 1 y))))
(define (xcor-vect vect)
  (/ (posn-x vect) 300) )
(define (ycor-vect vect)
  (+ 1 (- (/ (posn-y vect) 300))))
```

Exercise 2.47

The selectors `origin-frame` and `edge1-frame` are the same with both implementations:

```
(define (origin-frame frame)
  (car frame))

(define (edge1-frame frame)
  (cadr frame))
```

For the first implementation:

```
(define (make-frame origin edge1 edge2)
  (list origin edge1 edge2))

(define (edge2-frame frame)
  (caddr frame))
```

For the second implementation:

```
(define (make-frame origin edge1 edge2)
  (cons origin (cons edge1 edge2)))

(define (edge2-frame frame)
  (cddr frame))
```

Painters

Complement:
Sligthly modified version of `segments->painter` for use with graphics.ss:

```
(define (segments->painter segment-list)
  (lambda (frame)
    (for-each
      (lambda (segment)
        ((draw-line viewport)
         ((frame-coord-map frame) (start-segment segment))
```

```
        ((frame-coord-map frame) (end-segment segment))
        0))
     segment-list)))
```

Exercise 2.48

Representation of segments:

```
(define (make-segment start end)
  (cons start end))

(define (start-segment segment)
  (car segment))

(define (end-segment segment)
  (cdr segment))
```

Exercise 2.49

a. The painter drawing the outline can be defined as:

```
(define outline
  (let ((vbl (make-vect 0 0))
        (vbr (make-vect 1 0))
        (vtr (make-vect 1 1))
        (vtl (make-vect 0 1)))
    (segments->painter
      (list (make-segment vbl vbr)
            (make-segment vbr vtr)
            (make-segment vtr vtl)
            (make-segment vtl vbl)))))
```

b. The painter drawing an "X" can be defined as:

```
(define cross
  (let ((vbl (make-vect 0 0))
        (vbr (make-vect 1 0))
        (vtr (make-vect 1 1))
        (vtl (make-vect 0 1)))
    (segments->painter
      (list (make-segment vbl vtr)
            (make-segment vtl vbr)))))
```

c. The painter drawing a diamond can be defined as:

```
(define outline
  (let ((v1 (make-vect 0.5 0))
        (v2 (make-vect 1 0.5))
        (v3 (make-vect 0.5 1))
        (v4 (make-vect 0 0.5)))
```

```
    (segments->painter
      (list (make-segment v1 v2)
            (make-segment v2 v3)
            (make-segment v3 v4)
            (make-segment v4 v1)))))
```

d. The `wave` painter can be defined as:

```
(define wave
  (let ((p1 (make-vect 0.4 0.0))
        (p2 (make-vect 0.5 0.3))
        (p3 (make-vect 0.6 0.0))
        (p4 (make-vect 0.66 0.0))
        (p5 (make-vect 0.6 0.44))
        (p6 (make-vect 1.0 0.14))
        (p7 (make-vect 1.0 0.34))
        (p8 (make-vect 0.75 0.6))
        (p9 (make-vect 0.6 0.6))
        (p10 (make-vect 0.65 0.85))
        (p11 (make-vect 0.6 1.0))
        (p12 (make-vect 0.45 1.0))
        (p13 (make-vect 0.35 0.85))
        (p14 (make-vect 0.45 0.6))
        (p15 (make-vect 0.25 0.6))
        (p16 (make-vect 0.15 0.55))
        (p17 (make-vect 0.0 0.85))
        (p18 (make-vect 0.0 0.6))
        (p19 (make-vect 0.15 0.4))
        (p20 (make-vect 0.3 0.55))
        (p21 (make-vect 0.35 0.5))
        (p22 (make-vect 0.20 0.0)))
    (segments->painter
      (list
        (make-segment p1 p2)
        (make-segment p2 p3)
        (make-segment p4 p5)
        (make-segment p5 p6)
        (make-segment p7 p8)
        (make-segment p8 p9)
        (make-segment p9 p10)
        (make-segment p10 p11)
        (make-segment p12 p13)
        (make-segment p13 p14)
        (make-segment p14 p15)
        (make-segment p15 p16)
```

```
             (make-segment p16 p17)
             (make-segment p18 p19)
             (make-segment p19 p20)
             (make-segment p20 p21)
             (make-segment p21 p22)))))
```

Transforming and combining painters

Exercise 2.50

The given transformations can be defined as:

```
(define (flip-horiz painter)
  (transform-painter painter
                     (make-vect 1.0 0.0)
                     (make-vect 0.0 0.0)
                     (make-vect 1.0 1.0)))

(define (rotate180 painter)
  (transform-painter painter
                     (make-vect 1.0 1.0)
                     (make-vect 0.0 1.0)
                     (make-vect 1.0 0.0)))

(define (rotate270 painter)
  (transform-painter painter
                     (make-vect 0.0 1.0)
                     (make-vect 0.0 0.0)
                     (make-vect 1.0 1.0)))
```

Exercise 2.51

`below` defined as a procedure analogous to the `beside` procedure:

```
(define (below painter1 painter2)
  (let ((split-point (make-vect 0.0 0.5)))
    (let ((paint-bottom
            (transform-painter painter1
                               (make-vect 0.0 0.0)
                               (make-vect 1.0 0.0)
                               split-point))
          (paint-top
            (transform-painter paint2
                               split-point
                               (make-vect 1.0 0.5)
                               (make-vect 0.0 1.0))))
      (lambda (frame)
```

```
        (paint-bottom frame)
        (paint-top frame)))))
```

In terms of beside and rotation operations:

```
(define (below painter1 painter2)
  (rotate270 (beside (rotate90 painter2)
                     (rotate90 painter1))))
```

Levels of language for robust design

Exercise 2.52

a. Possible modified version of wave:

```
(define wave
  (let ((p1 (make-vect 0.4 0.0))
        (p2 (make-vect 0.5 0.3))
        (p3 (make-vect 0.6 0.0))
        (p4 (make-vect 0.66 0.0))
        (p5 (make-vect 0.6 0.44))
        (p6 (make-vect 1.0 0.14))
        (p7 (make-vect 1.0 0.34))
        (p8 (make-vect 0.75 0.6))
        (p9 (make-vect 0.6 0.6))
        (p10 (make-vect 0.65 0.85))
        (p11 (make-vect 0.6 1.0))
        (p12 (make-vect 0.45 1.0))
        (p13 (make-vect 0.35 0.85))
        (p14 (make-vect 0.45 0.6))
        (p15 (make-vect 0.25 0.6))
        (p16 (make-vect 0.15 0.55))
        (p17 (make-vect 0.0 0.85))
        (p18 (make-vect 0.0 0.6))
        (p19 (make-vect 0.15 0.4))
        (p20 (make-vect 0.3 0.55))
        (p21 (make-vect 0.35 0.5))
        (p22 (make-vect 0.20 0.0))
        (p23 (make-vect 0.42 0.8))
        (p24 (make-vect 0.46 0.7))
        (p25 (make-vect 0.58 0.7))
        (p26 (make-vect 0.62 0.8)))
    (segments->painter
      (list
        (make-segment p1 p2)
        (make-segment p2 p3)
```

```
(make-segment p4 p5)
(make-segment p5 p6)
(make-segment p7 p8)
(make-segment p8 p9)
(make-segment p9 p10)
(make-segment p10 p11)
(make-segment p12 p13)
(make-segment p13 p14)
(make-segment p14 p15)
(make-segment p15 p16)
(make-segment p16 p17)
(make-segment p18 p19)
(make-segment p19 p20)
(make-segment p20 p21)
(make-segment p21 p22)
(make-segment p23 p24)
(make-segment p24 p25)
(make-segment p25 p26)))))
```

b. Modified `corner-split`:

```
(define (corner-split painter n)
  (if (= n 0)
    painter
    (let ((up (up-split painter (- n 1)))
          (right (right-split painter (- n 1)))
          (corner (corner-split painter (- n 1))))
      (beside (below painter up)
              (below right corner)))))
```

c. Modified `square-limit`:

```
(define (square-limit painter n)
  (let ((combine4 (square-of-four flip-horiz identity
                                  rotate180 flip-vert)))
    (combine4 (corner-split (flip-horiz painter) n))))
```

2.3 Symbolic Data

2.3.1 Quotation

Exercise 2.53

Check with an interpreter.

Exercise 2.54

A possible implementation of `equal?`:

```
(define (equal? a b)
  (or (and (symbol? a)
           (symbol? b)
           (eq? a b))
      (and (null? a)
           (null? b))
      (and (pair? a)
           (pair? b)
           (equal? (car a) (car b))
           (equal? (cdr a) (cdr b)))))
```

Exercise 2.55

The expression `''abracadabra` is equivalent to `(quote (quote abracadabra))`, which evaluates to `(quote abracadabra)`, an expression whose `car` is `quote`.

2.3.2 Example: Symbolic Differentiation

Exercise 2.56

The modified version of `deriv` and the procedures `exponentiation?`, `base`, `exponent`, and `make-exponentiation` can be written as:

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
           (make-product (multiplier exp)
                         (deriv (multiplicand exp) var))
           (make-product (deriv (multiplier exp) var)
                         (multiplicand exp))))
        ((exponentiation? exp)
         (make-product (make-product (exponent exp)
                                     (deriv (base exp) var))
                       (make-exponentiation (base exp) (- (exponent exp) 1))))
        (else
          (error "Unknown expression type -- DERIV " exp))))

(define (exponentiation? exp)
  (and (pair? exp) (eq? (car exp) '**)))

(define (base e) (cadr e))
```

```
(define (exponent e) (caddr e))

(define (make-exponentiation base exponent)
  (cond ((=number? exponent 0) 1)
        ((=number? exponent 1) base)
        (else (list '** base exponent))))
```

Exercise 2.57

The deriv procedure always calls make-sum and make-product with only two arguments, so all that's necessary is to redefine augend and multiplicand as indicated in the text, for instance:

```
(define (augend s)
  (if (null? (cdddr s))
    (caddr s)
    (cons '+ (cddr s))))

(define (multiplicand p)
  (if (null? (cdddr p))
    (caddr p)
    (cons '* (cddr p))))
```

As a supplement, if we also want to be able to call make-sum and make-product with more than two arguments, we can redefine make-sum and make-product as well:

```
(define (augend s)
  (apply make-sum (cddr s)))

(define (multiplicand p)
  (apply make-product (cddr p)))

(define (make-sum . elts)
  (let ((nb (apply + (filter number? elts)))
        (exps (filter (lambda (x)
                         (not (number? x)))
                       elts)))
    (cond ((null? exps) nb)
          ((= nb 0)
           (if (null? (cdr exps))
             (car exps)
             (cons '+ exps)))
          (else (cons '+ (cons nb exps))))))

(define (make-product . elts)
  (let ((nb (apply * (filter number? elts)))
        (exps (filter (lambda (x)
```

```
                     (not (number? x)))
                   elts)))
  (cond ((null? exps) nb)
        ((= nb 0)
         0)
        ((= nb 1)
         (if (null? (cdr exps))
           (car exps)
           (cons '* exps)))
        (else (cons '* (cons nb exps))))))
```

These procedures also simplify the result very partially by grouping the numerical arguments together, however expressions such as `(make-sum 'x (make-sum 'y 'x))` or `(make-sum 'x 'y 2 'x)` are not simplified.

Exercise 2.58

a. It is sufficient to redefine the following procedures:

```
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2)) (+ a1 a2))
        (else (list a1 '+ a2))))

(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
        (else (list m1 '* m2))))

(define (sum? x)
  (and (pair? x) (eq? (cadr x) '+)))

(define (addend s) (car s))

(define (product? x)
  (and (pair? x) (eq? (cadr x) '*)))

(define (multiplier p) (car p))

(define (exponentiation? exp)
  (and (pair? exp) (eq? (cadr exp) '**)))

(define (base e) (car e))
```

```
(define (make-exponentiation base exponent)
  (cond ((=number? exponent 0) 1)
        ((=number? exponent 1) base)
        (else (list base '** exponent))))
```

b. The following implementations supports only multiplication and addition and can produce results with unnecessary parentheses.

It uses several helper procedures:

- `(elt-or-list elts)` returns the car of elts if elts is of length 1, and the list elts otherwise.
- `(take-until l elt)` returns a list containing all the elements of l until the first occurrence of `elt`, excluding it. If elt is not contained in the list, the full list is returned.
- `(intersperse l sep)` returns a list containing all the elements of l, with sep inserted between each pair of elements.

```
(define (sum? x)
  (memq '+ x))

(define (product? x)
  (and (not (sum? x)) (memq '* x)))

(define (elt-or-list l)
  (if (= 1 (length l))
    (car l)
    l))

(define (take-until l elt)
  (if (or (null? l) (eq? (car l) elt))
    '()
    (cons (car l) (take-until (cdr l) elt))))

(define (intersperse l sep)
  (if (<= (length l) 1)
    l
    (append (list (car l) sep) (intersperse (cdr l) sep))))

(define (addend s)
  (elt-or-list (take-until s '+)))

(define (multiplier p)
  (elt-or-list (take-until p '*)))
```

```scheme
(define (augend s)
  (elt-or-list (cdr (memq '+ s))))

(define (multiplicand p)
  (elt-or-list (cdr (memq '* p))))

(define (make-sum . elts)
  (let ((nb (apply + (filter number? elts)))
        (exps (filter (lambda (x)
                        (not (number? x)))
                      elts)))
    (cond ((null? exps) nb)
          ((= nb 0)
           (elt-or-list (intersperse exps '+)))
          (else (intersperse (cons nb exps) '+)))))

; TODO: Parenthèses
(define (make-product . elts)
  (let ((nb (apply * (filter number? elts)))
        (exps (filter (lambda (x)
                        (not (number? x)))
                      elts)))
    (cond ((null? exps) nb)
          ((= nb 0)
           0)
          ((= nb 1)
           (elt-or-list (intersperse exps '*)))
          (else (intersperse (cons nb exps) '*)))))
```

2.3.3 Example: Representing sets

Sets as unordered lists

Exercise 2.59

The `union-set` operation can be defined as:

```scheme
(define (union-set set1 set2)
  (cond ((null? set1) set2)
        ((element-of-set? (car set1) set2)
         (union-set (cdr set1) set2))
        (else (cons (car set1)
                    (union-set (cdr set1) set2)))))
```

Exercise 2.60

If duplicate elements are allowed, we can redefine `adjoin-set` and `union-set` in a more efficient way:

```
(define (adjoin-set x set)
  (cons x set))

(define (union-set set1 set2)
  (append set1 set2))
```

The operation `adjoin-set` now has $\Theta(1)$ complexity, while `union-set` has $\Theta(n)$ complexity. It's not possible to improve the performance of `element-of-set?` and `intersection-set`.

This representation would be preferable to the non-duplicate one when there is no need to worry about the size taken by the sets in memory, and when the operations `adjoin-set` and `union-set` are used a lot more than `element-of-set?` and `intersection-set`.

Sets as ordered lists

Exercise 2.61

A possible implementation of `adjoin-set` requiring on average half as many steps as with the unordered representation is:

```
(define (adjoin-set x set)
  (if (null? set)
    (list x)
    (let ((first (car set)))
          (cond ((= x first) set)
                ((< x first) (cons x set))
                ((< first x) (cons first (adjoin-set x (cdr set)))))))))
```

Exercise 2.62

By using the same method as for `intersection-set`, we get a $\Theta(n)$ `union-set` implementation:

```
(define (union-set set1 set2)
  (cond ((null? set1) set2)
        ((null? set2) set1)
        (else (let ((x1 (car set1))
                    (x2 (car set2)))
                (cond ((= x1 x2)
                       (cons x1 (union-set (cdr set1) (cdr set2))))
                      ((< x1 x2)
                       (cons x1 (union-set (cdr set1) set2)))
                      ((< x2 x1)
                       (cons x2 (union-set set1 (cdr set2))))))))))
```

Sets as binary trees

Exercise 2.63

a. Both procedures produce the same result for every tree. They produce the ordered list representation of the set represented by the tree. For the trees in figure 2.16, they produce the list `(1 3 5 7 9 11)`.

b. If $T(n)$ is the number of steps required to convert a balanced tree to a list, with the first procedure we have: $T(n) \approx 2T\left(n/2\right) + n/2$ since `append` has linear complexity. By applying this formula recursively, we can see that $T(n) \approx n + n \log n/2$, so that `tree->list-1` grows as $\Theta(n \log n)$.

   With the second procedure, we have $T(n) = 2T(n/2) + 1$, so `tree->list-2` grows as $\Theta(n)$.

Exercise 2.64

a. If $n = 0$, the constructed tree is empty.

   Otherwise, one element will be the tree's entry, so $n - 1$ elements will be in the subtrees. $l = \lfloor n-1/2 \rfloor$ elements are put in the left tree, and the remaining $r = n - 1 - l$ are put in the right tree.

   Since the list is ordered and the elements in the left tree must be smaller than the entry, the first elements of the list are used to build the left tree. The first remaining element is the entry, and the r following remaining elements are used to build the right tree. The remaining elements of this last operation are also the remaining elements for the current call to `partial-tree`, so all that remains to be done is to put all the results together.

b. For `partial-tree`, the number of steps $T(n)$ as a function of the size of the tree to build n verifies $T(n) \approx 2T\left(n/2\right)$, so its growth is in $\Theta(n)$, so `list->tree` has linear growth.

Exercise 2.65

If we call `union-set-ordered-list` and `intersection-set-ordered-list` the linear union and intersection procedures defined for ordered lists, we can define linear procedures `union-set` and `intersection-set` for binary trees as:

```scheme
(define (union-set set1 set2)
  (list->tree (union-set-ordered-lists (tree->list-2 set1)
                                       (tree->list-2 set2))))

(define (intersection-set set1 set2)
  (list->tree (intersection-set-ordered-list (tree->list-2 set1)
                                             (tree->list-2 set2))))
```

Sets and information retrieval

Exercise 2.66

The procedure is almost the same as `element-of-set?` for sets represented as binary trees:

```
(define (lookup given-key set-of-records)
  (cond ((null? set-of-records) false)
        ((equal? given-key (key (entry set-of-records)))
         (entry set-of-records))
        ((< given-key (key (entry set-of-records)))
         (lookup given-key (left-tree set-of-records)))
        (else
          (lookup given-key (right-tree set-of-records)))))
```

2.3.4 Example: Huffman Encoding Trees

Exercise 2.67

The encoded string is ADABBCA.

Exercise 2.68

A possible implementation of `encode-symbol` is:

```
(define (encode-symbol symbol tree)
  (cond ((leaf? tree)
         (if (eq? symbol (symbol-leaf tree))
           '()
           (error "Symbol not present in tree -- ENCODE-SYMBOL" symbol)))
        ((member symbol (symbols (left-branch tree)))
         (cons 0
               (encode-symbol symbol (left-branch tree))))
        (else
          (cons 1
                (encode-symbol symbol (right-branch tree))))))
```

Exercise 2.69

A possible implementation for `successive-merge` is:

```
(define (successive-merge leaf-set)
  (if (= 1 (length leaf-set))
    (car leaf-set)
    (successive-merge (adjoin-set (make-code-tree (car leaf-set)
                                                  (cadr leaf-set))
                                  (cddr leaf-set)))))
```

Exercise 2.70

With a Huffman encoding tree, 84 bits are required for encoding. With a fixed-length code, at least 3 bits per symbol are required for an alphabet of 8 symbols, and the message contains 36 symbols, so at least 108 bits would have been needed.

Exercise 2.71

Since $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$, every right (or every left) branch of the tree is a leaf, and the tree has a depth of $n - 1$. So the most frequent symbol is encoded with 1 bit, while the least frequent symbol is encoded with $n - 1$ bits.

Exercise 2.72

In the `encode-symbol` procedure given in exercise 2.68, only the symbol list of the left subtree is searched at each node encountered, so we can distinguish two cases for the distribution given in exercise 2.71:

The left subtree of each node corresponds to the most frequent symbol. Encoding the most frequent symbol takes $\Theta(1)$ steps since only one search in a list of size one is necessary before a leaf is reached.

Encoding the least frequent symbol takes $\Theta(n)$ steps, since all the levels of the tree are traversed, and at each level a search in a list of one element is performed.

The right subtree of each node corresponds to the most frequent symbol. Encoding the most frequent symbol takes $\Theta(n)$ steps, for the search in the symbol list.

For the least frequent symbol, at each of the $n-1$ tree levels the left subtree's symbol list is searched, which takes $n-1-i$ steps, where i is the level number, with 0 designing the tree root. The number of steps required grows as $\sum_{k=1}^{n-1} k = n(n-1)/2$, so the number of steps required to encode the least frequent symbol is in $\Theta(n^2)$.

2.4 Multiple Representations for Abstract Data

2.4.1 Representations for Complex Numbers

This subsection contains no exercises.

2.4.2 Tagged data

This subsection contains no exercises.

2.4.3 Data-Directed Programming and Additivity

Exercise 2.73

a. `same-variable?` and `number?` can't be integrated to the main dispatch because when they are true, `exp` has no operator or operands.

b. The procedures for derivation can be written in the following way, using `make-sum` and `make-product` from section 2.3.2.

```
(define (install-deriv)
  (define (addend operands) (car operands))
  (define (augend operands) (cadr operands))
  (define (deriv-sum operands var)
    (make-sum (deriv (addend operands) var)
              (deriv (augend operands) var)))
  (define (multiplier operands) (car operands))
  (define (multiplicand operands) (cadr operands))
```

```
(define (deriv-product operands var)
  (make-sum (make-product (deriv (multiplier operands) var)
                          (multiplicand operands))
            (make-product (multiplier operands)
                          (deriv (multiplicand operands) var))))
(put 'deriv '+ deriv-sum)
(put 'deriv '* deriv-product))
```

c. To allow differentiation of exponents, it's enough to call the following procedure, using `make-exporentiation` from exercise 2.56.

d. The only change required is to exchange the arguments when calling `put`, using e.g. `(put '+ 'deriv deriv-sum)` instead of `(put 'deriv '+ deriv-sum)`.

Exercise 2.74

a. The `get-record` procedure can be written using `apply-generic`, provided each division file is tagged with a division-specific tag, and each division has recorded a procedure get-record that takes a file's contents and returns a procedure returning a given employee's record.

```
(define (get-record employee file)
  ((apply-generic 'get-record file) employee))
```

b. `get-salary` can also be implemented with `apply-generic`, if each record is tagged with the division-specific tag, and each division records a procedure for the `get-salary` operation.

```
(define (get-salary record)
  (apply-generic 'get-salary record))
```

c. The `find-employee-record` procedure can be written as:

```
(define (find-employee-record employee files)
  (if (null? files)
    #f
    (or (get-record employee (car files))
        (find-employee-record employee (cdr files)))))
```

d. When Insatiable takes over a new company, the new company must tag its record file and each of its employee's record with a specific tag. It must install specific procedures for `get-record`, `get-salary`, etc. into the operation-and-type table.

Message passing

Exercise 2.75

The constructor `make-from-mag-ang` can be defined as follows:

```
(define (make-from-mag-ang r a)
  (define (dispatch op)
```

```
    (cond ((eq? op 'real-part) (* r (cos a)))
          ((eq? op 'imag-part) (* r (sin a)))
          ((eq? op 'angle) a)
          ((eq? op 'magnitude) r)
          (else
            (error "Unknown op -- MAKE-FROM-MAG-ANG" op))))
  dispatch)
```

Exercise 2.76

With generic operations with explicit dispatch, when a new type is added, all existing operations must be updated. When a new operation is added, no change to existing code is required.

With data-directed style, when a new type is added, all the operations for this type must be added to the table. When a new operation is added, it must be defined and inserted into the table for each type.

With message-passing, when a new type is added, no change is required to existing code. When a new operation is added, it must be added to all the existing types.

For a system in which new types must often be added, message-passing is most appropriate. If new operations must often be added, explicit dispatch is most appropriate.

2.5 Systems with Generic Operations

2.5.1 Generic Arithmetic Operations

Exercise 2.77

When one of the generic operations is called on a `complex` number, `apply-generic` is called and dispatches the call to the same generic operation on the `polar` or `rectangular` number contained in the `complex`.

For the object `z` given in figure 2.24, the procedures called are:

```
(magnitude z)
(apply-generic 'magnitude z)
(magnitude (contents z))
(apply-generic 'magnitude (contents z))
(magnitude-rectangular (3 . 4))
```

So `apply-generic` is called twice, it dispatches first to the same generic operation `magnitude`, then to the specific operation for rectangular numbers.

Exercise 2.78

The definitions of `type-tag`, `contents` and `attach-tag` can be modified in the following way to represent ordinary numbers simply as Scheme numbers.

```
(define (type-tag datum)
  (cond ((number? datum) 'scheme-number)
        ((pair? datum) (car datum))
```

```
        (else
          (error "Bad tagged datum -- TYPE-TAG" datum))))

(define (contents datum)
  (cond ((number? datum) datum)
        ((pair? datum) (cdr datum))
        (else
          (error "Bad tagged datum -- CONTENTS" datum))))

(define (attach-tag type-tag contents)
  (if (eq? 'scheme-number type-tag) contents
    (cons type-tag contents)))
```

Exercise 2.79

To define the `equ?` procedure, we first define it as a generic operation:

```
(define (equ? x y) (apply-generic 'equ? x y))
```

Then we add the following respectively to the Scheme number, rational and complex packages:

```
(put 'equ? '(scheme-number scheme-number) =)

(put 'equ? '(rational rational)
     (lambda (x y) (= (* (numer x) (denom y))
                      (* (numer y) (denom x)))))

(put 'equ? '(complex complex)
     (lambda (z1 z2)
       (and (equ? (real-part z1) (real-part z2))
            (equ? (imag-part z1) (imag-part z2)))))
```

Exercise 2.80

The definition of `=zero?` is very similar to that of `equ?`, first define it as a generic operation, then add the necessary code to the Scheme number, rational and complex packages.

```
(define (=zero? x) (apply-generic '=zero? x))

(put '=zero? '(scheme-number)
     (lambda (x) (= x 0)))

(put '=zero? '(rational)
     (lambda (r) (= (numer r) 0)))

(put '=zero? '(complex)
     (lambda (z) (= (magnitude z) 0)))
```

2.5.2 Combining Data of Different Types

Exercise 2.81

a. With Louis's coercion procedures installed, if `apply-generic` is called with two arguments of type `complex` for an operation that is not found in the table for those types, as in the example given for `exp`, `apply-generic` is called recursively with the same arguments, so there is an infinite loop.

b. The `apply-coercion` procedure works correctly, however, it's unnecessary to attempt coercion to the same type so it should not be attempted.

c. The new version of `apply-generic` could be:

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
        (apply proc (map contents args))
        (if (= (length args) 2)
          (let ((type1 (car type-tags))
                (type2 (cadr type-tags))
                (a1 (car args))
                (a2 (cadr args)))
            (if (eq? type1 type2)
              (error "No method for these types"
                     (list op type-tags))
              (let ((t1->t2 (get-coercion type1 type2))
                    (t2->t1 (get-coercion type2 type1)))
                (cond (t1->t2
                        (apply-generic op (t1->t2 a1) a2))
                      (t2->t1
                        (apply-generic op t1 (t2->t1 a2)))
                      (else
                        (error "No method for these types"
                               (list op type-tags)))))))
          (error "No method for these types"
                 (list op type-tags)))))))
```

Exercise 2.82

A version of `apply-generic` that handles coercion in the case of multiple arguments can look like:

```
(define (apply-generic op . args)
  ;; To avoid trying to coerce the arguments to a given type several times.
  (define (unique elts)
    (if (null? elts)
```

```
    '()
    (let ((rest (unique (cdr elts))))
      (if (memq (car elts) rest)
        rest
        (cons (car elts) rest)))))
;; Tries to coerce all elements of args to the given type. Returns false if
;; it's impossible.
(define (coerce-to-type type args)
  (if (null? args)
    '()
    (let* ((a (car args))
           (t (type-tag a))
           (others-coerced (coerce-to-type type (cdr args))))
      (cond ((not others-coerced) false)
            ((eq? t type)
             (cons a others-coerced))
            (else
              (let ((t->type (get-coercion t type)))
                (if t->type
                  (cons (t->type a) others-coerced)
                  false)))))))
(define (try-coercion type-tags)
  (define (try-types types-to-try)
    (if (null? types-to-try)
      (error "No method for these types"
             (list op type-tags))
      (let* ((type (car types-to-try))
             (coerced-args (coerce-to-type type args)))
        (if coerced-args
          (let ((proc (get op (map type-tag coerced-args))))
            (if proc
              (apply proc (map contents coerced-args))
              (try-types (cdr types-to-try))))
          (try-types (cdr types-to-try))))))
  (try-types (unique type-tags)))
(let* ((type-tags (map type-tag args))
       (proc (get op type-tags)))
  (if proc
    (apply proc (map contents args))
    (try-coercion type-tags))))
```

The strategy proposed is not sufficiently general: let's assume we define a procedure `exp` for arguments of type `complex` and `scheme-number`. If we call it with a rational and a Scheme number, the procedure above will try to coerce both arguments to rationals, and then to Scheme numbers, and will not find a suitable procedure in either case. However, it would have been enough to coerce the first argument to a complex to be able to apply the registered procedure.

Exercise 2.83

The `raise` operation can be installed with:

```
(define (raise x)
  (apply-generic 'raise x))

(define (install-raise)
  (put 'raise '(integer)
       (lambda (x)
         (make-rational x 1)))
  (put 'raise '(rational)
       (lambda (x)
         (make-real (exact->inexact (/ (car x)
                                       (cdr x))))))
  (put 'raise '(real)
       (lambda (x)
         (make-complex-from-real-imag x 0)) ))
```

Complement:
So far, we have only used the types `scheme-number`, `rational` and `complex`. Since a lot of Scheme implementations — including Gambit — provide the full numeric tower, including predicates `integer?` and `real?`, we can implement the full tower by loading the following code:

```
(define (type-tag datum)
  (cond ((and (integer? datum)
              (exact? datum)) 'integer)
        ((real? datum) 'real)
        ((pair? datum) (car datum))
        (else
          (error "Bad tagged datum -- TYPE-TAG" datum))))

(define (attach-tag type-tag contents)
  (if (or (eq? 'integer type-tag)
          (eq? 'real type-tag))
    contents
    (cons type-tag contents)))

(define (install-number-package type)
  (define (tag x)
    (attach-tag type x))
  (put 'add (list type type)
       (lambda (x y) (tag (+ x y))))
  (put 'sub (list type type)
       (lambda (x y) (tag (- x y))))
  (put 'mul (list type type)
       (lambda (x y) (tag (* x y))))
```

```
  (put 'div (list type type)
       (lambda (x y) (tag (/ x y))))
  (put 'equ? (list type type) =)
  (put '=zero? (list type)
       (lambda (x) (= x 0)))
  (put 'make type
       (lambda (x) (tag x)))
  'done)

(install-number-package 'integer)
(install-number-package 'real)

(define (make-integer n)
  ((get 'make 'integer) n))
(define (make-real n)
  ((get 'make 'real) n))
```

Exercise 2.84

The `apply-generic` procedure can be modified as shown below. To add a new level to the tower, in addition to defining the procedures for the new type, the only necessary change is to add the new level to the tower.

```
(define (raise-to-type type arg)
  (let ((t (type-tag arg)))
    (cond ((eq? t type) arg)
          ((higher? type t) (raise-to-type type (raise arg)))
          (else
            (error "Trying to raise an element to a type lower than its own type."
                   (list arg type))))))

(define (highest-type types)
  (define (iter current rest)
    (cond ((null? rest) current)
          ((higher? (car rest) current)
           (iter (car rest) (cdr rest)))
          (else
            (iter current (cdr rest)))))
  (iter (car types) (cdr types)))

(define tower '(integer rational real complex))

(define (higher? t1 t2)
  (let ((m1 (memq t1 tower))
        (m2 (memq t2 tower)))
    (cond ((not m1)
           (error "Type not found -- higher?" t1))
```

```
          ((not m2)
           (error "Type not found -- higher?" t2))
          (else
            (< (length m1) (length m2))))))

(define (apply-generic op . args)
  (let* ((type-tags (map type-tag args))
         (proc (get op type-tags)))
    (if proc
      (apply proc (map contents args))
      (let* ((type (highest-type type-tags))
             (raised-args (map (lambda (a)
                                 (raise-to-type type a))
                               args))
             (newproc (get op (map type-tag raised-args))))
        (if newproc
          (apply newproc (map contents raised-args))
          (error "No method for these types"
                 (list op type-tags)))))))
```

Exercise 2.85

Let's first define the `project` and `drop` procedures:

```
(define lowest-level
  (car tower))

(define (drop x)
  (if (eq? (type-tag x) lowest-level)
    x
    (let ((proj (project x)))
      (if (equ? (raise proj) x)
        (drop proj)
        x))))

(define (project x)
  (apply-generic 'project x))

(put 'project '(rational)
     (lambda (r)
       (make-integer (quotient (numer r)
                               (denom r)))))
(put 'project '(real)
     (lambda (x)
       (make-integer (inexact->exact (round x)))))
(put 'project '(complex)
```

```
     (lambda (x)
       (make-real (real-part x))))
```

Since raise is defined with apply-generic, simply calling drop after applying the procedure found causes an infinite loop. It's also unnecessary to call it after project, and it can be called only on results representing numbers. So we add entries in the table to indicate that drop should not be called if the operation applied is raise, project, equ? or =zero?. It is called by default.

```
(define (no-drop? op)
  (get 'no-drop? op))
(put 'no-drop? 'raise true)
(put 'no-drop? 'project true)
(put 'no-drop? 'equ? true)
(put 'no-drop? '=zero? true)

(define (apply-generic op . args)
  (define (apply-proc proc args-list)
    (if (no-drop? op)
      (apply proc args-list)
      (drop (apply proc args-list))))

  (let* ((type-tags (map type-tag args))
         (proc (get op type-tags)))
    (if proc
      (apply-proc proc (map contents args))
      (let* ((type (highest-type type-tags))
             (raised-args (map (lambda (a)
                                 (raise-to-type type a))
                               args))
             (newproc (get op (map type-tag raised-args))))
        (if newproc
          (apply-proc newproc (map contents raised-args))
          (error "No method for these types"
                 (list op type-tags)))))))
```

Exercise 2.86

The only change required is to use generic operations for all operations on numbers used in the complex, rectangular and polar packages. The generic operations add, sub, mul and div have already been defined, and it's necessary to define generic operations for cos, sin, atan, sqrt and square.

```
(define (install-rectangular-package)
  ;; internal procedures
  (define (real-part z) (car z))
  (define (imag-part z) (cdr z))
```

```scheme
  (define (magnitude z)
    (sqrt-generic (add (square-generic (real-part z)) (square-generic (imag-part z)))))
  (define (angle z)
    (atan-generic (imag-part z) (real-part z)))
  (define (make-from-real-imag x y) (cons x y))
  (define (make-from-mag-ang r a)
    (cons (mul r (cosine a)) (mul r (sine a))))
  ;; interface to the rest of the system
  (define (tag x) (attach-tag 'rectangular x))
  (put 'real-part '(rectangular) real-part)
  (put 'imag-part '(rectangular) imag-part)
  (put 'magnitude '(rectangular) magnitude)
  (put 'angle '(rectangular) angle)
  (put 'make-from-real-imag 'rectangular
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'rectangular
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)

(define (install-polar-package)
  ;; internal procedures
  (define (real-part z)
    (mul (magnitude z) (cosine (angle z))))
  (define (imag-part z)
    (mul (magnitude z) (sine (angle z))))
  (define (magnitude z) (car z))
  (define (angle z) (cdr z))
  (define (make-from-real-imag x y)
    (cons (sqrt-generic (add (square-generic x) (square-generic y)))))
  (define (make-from-mag-ang r a) (cons r a))
  ;; interface to the rest of the system
  (define (tag x) (attach-tag 'polar x))
  (put 'real-part '(polar) real-part)
  (put 'imag-part '(polar) imag-part)
  (put 'magnitude '(polar) magnitude)
  (put 'angle '(polar) angle)
  (put 'make-from-real-imag 'polar
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'polar
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)

(define (install-complex-package)
  ;; Imported procedures from rectangular and polar packages.
```

```
  (define (make-from-real-imag x y)
    ((get 'make-from-real-imag 'rectangular) x y))
  (define (make-from-mag-ang r a)
    ((get 'make-from-mag-ang 'polar) r a))

  ;; Internal procedures.
  (define (add-complex z1 z2)
    (make-from-real-imag (add (real-part z1) (real-part z2))
                         (add (imag-part z1) (imag-part z2))))
  (define (sub-complex z1 z2)
    (make-from-real-imag (sub (real-part z1) (real-part z2))
                         (sub (imag-part z1) (imag-part z2))))
  (define (mul-complex z1 z2)
    (make-from-mag-ang (mul (magnitude z1) (magnitude z2))
                       (add (angle z1) (angle z2))))
  (define (div-complex z1 z2)
    (make-from-mag-ang (div (magnitude z1) (magnitude z2))
                       (sub (angle z1) (angle z2))))

  ;; Interface to the rest of the system.
  (define (tag z) (attach-tag 'complex z))
  (put 'add '(complex complex)
       (lambda (z1 z2) (tag (add-complex z1 z2))))
  (put 'sub '(complex complex)
       (lambda (z1 z2) (tag (sub-complex z1 z2))))
  (put 'mul '(complex complex)
       (lambda (z1 z2) (tag (mul-complex z1 z2))))
  (put 'div '(complex complex)
       (lambda (z1 z2) (tag (div-complex z1 z2))))
  (put 'make-from-real-imag 'complex
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'complex
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)

;; Definition af the generic procedures.
(define (sqrt-generic x)
  (apply-generic 'sqrt x))

(define (square-generic x)
  (apply-generic 'square x))

(define (atan-generic y x)
  (apply-generic 'atan y x))
```

```
(define (cosine x)
  (apply-generic 'cos x))

(define (sine x)
  (apply-generic 'sin x))

(define (install-number-package-ext type)
  (define (tag x)
    (attach-tag type x))
  (put 'sqrt (list type)
       (lambda (x) (tag (sqrt x))))
  (put 'square (list type)
       (lambda (x) (tag (square x))))
  (put 'cos (list type)
       (lambda (x) (tag (cos x))))
  (put 'sin (list type)
       (lambda (x) (tag (sin x))))
  (put 'atan (list type type)
       (lambda (y x) (tag (atan y x))))
  'done)

(put 'sqrt '(rational)
     (lambda (r)
       (make-rational (sqrt (numer r))
                      (sqrt (denom r)))))

(put 'square '(rational)
     (lambda (r)
       (make-rational (square (numer r))
                      (square (denom r)))))

(put 'sin '(rational)
     (lambda (r)
       (make-real (sin (/ (numer r)
                          (denom r))))))

(put 'cos '(rational)
     (lambda (r)
       (make-real (cos (/ (numer r)
                          (denom r))))))

(put 'atan '(rational rational)
     (lambda (r1 r2)
```

```
        (make-real (atan (/ (numer r1) (denom r1))
                         (/ (numer r2) (denom r2))))))
```

2.5.3 Example: Symbolic Algebra

Arithmetic on polynomials

Complement:
I decided to add the polynomial type to the tower of types used since exercise 2.83. For this to work, I had to redefine the tower of types:

```
(define tower '(integer rational real complex polynomial))
```

and to define the procedures `project` and `equ?` for polynomials, as well as a `raise` procedure that transforms a complex number into a polynomial of degree 0. So I added the following code to the polynomial package given in the book. This still needs the `=zero?` procedure defined in the following exercise for `=equ?` to work on polynomials with polynomial coefficients.

```
(define (get-coeff-by-degree p degree)
  (get-coeff-by-degree-terms (term-list p) degree))

(define (get-coeff-by-degree-terms term-list degree)
  (cond ((empty-termlist? term-list) 0)
        ((= (order (first-term term-list)) degree)
         (coeff (first-term term-list)))
        (else
          (get-coeff-by-degree-terms (rest-terms term-list) degree))))

(define (degree p)
  (degree-terms (term-list p)))

(define (degree-terms term-list)
  (if (empty-termlist? term-list)
    0
    (order (first-term term-list))))

(define (equ-term-lists? L1 L2)
  (cond ((and (empty-termlist? L1) (empty-termlist? L2))
         true)
        ((or (empty-termlist? L1) (empty-termlist? L2))
         false)
        (else
          (let ((t1 (first-term L1))
                (t2 (first-term L2)))
            (cond ((> (order t1) (order t2))
                   (and (=zero? (coeff t1))
                        (equ-term-lists? (rest-terms L1) L2)))
                  ((< (order t1) (order t2))
                   (and (=zero? (coeff t2))
                        (equ-term-lists? L1 (rest-terms L2))))
                  (else
```

```
                    (and (equ? (coeff t1) (coeff t2))
                         (equ-term-lists? (rest-terms L1) (rest-terms L2)))))))))

(put 'project '(polynomial)
     (lambda (p)
       (let ((c0 (get-coeff-by-degree p 0)))
         (if (eq? (type-tag c0) 'polynomial)
           (project c0)
           c0))))

(put 'raise '(complex)
     (lambda (z)
       (make-polynomial 'x (adjoin-term (make-term 0 (drop (attach-tag 'complex z)))
                                        (make-empty-termlist 'dense)))))

(put 'equ? '(polynomial polynomial)
     (lambda (p1 p2)
       (if (and (not (eq? (variable p1) (variable p2)))
                (or (> (degree p1) 0) (> (degree p2) 0)))
         false
         (let ((L1 (term-list p1))
               (L2 (term-list p2)))
           (equ-term-lists? L1 L2)))))
```

Exercise 2.87

The =zero? procedure can be defined by adding the following code to the polynomial package:

```
(define (=zero-terms? L)
  (if (empty-termlist? L)
    true
    (and (=zero? (coeff (first-term L)))
         (=zero-terms? (rest-terms L)))))

(put '=zero? '(polynomial)
     (lambda (p) (=zero-terms? (term-list p))))
```

Exercise 2.88

We can define a generic negation operation neg for types other than polynomial with the following code:

```
(define (neg x)
  (apply-generic 'neg x))

(put 'neg '(integer)
     (lambda (x) (make-integer (- x))))
(put 'neg '(rational)
     (lambda (r)
       (make-rational (- (numer r))
```

```
                       (denom r))))
(put 'neg '(real)
     (lambda (x) (make-real (- x))))
(put 'neg '(complex)
     (lambda (z)
       (make-complex-from-real-imag (neg (real-part z))
                                    (neg (imag-part z)))))
```

Then, the neg and sub operations can be implemented for polynomials by adding the following to the polynomial package:

```
(define (neg-terms L)
  (if (empty-termlist? L)
    (make-empty-termlist (type-tag L))
    (let ((first-term (first-term L)))
      (adjoin-term (make-term (order first-term)
                              (neg (coeff first-term)))
                   (neg-terms (rest-terms L))))))
(define (neg-poly p)
  (make-polynomial (variable p)
                   (neg-terms (term-list p))))

(put 'neg '(polynomial)
     (lambda (p) (tag (neg-poly p))))
(put 'sub '(polynomial polynomial)
     (lambda (p1 p2) (tag (add-poly p1 (neg-poly p2)))))
```

Exercise 2.89

The only procedures that need to be redefined to implement the term-list representation appropriate for dense polynomials are `adjoin-term` and `first-term`:

```
(define (adjoin-term term term-list)
  (cond ((=zero? (coeff term))
         term-list)
        ((= (order term) (length term-list))
         (cons (coeff term) term-list))
        (else
          (adjoin-term term (cons 0 term-list)))))

(define (first-term term-list)
  (make-term (- (length term-list) 1)
             (car term-list)))
```

Exercise 2.90

I defined two packages to install the represention for the two kinds of term lists. Each package defines `adjoin-term`, `first-term`, `rest-terms`, `empty-termlist?`, as well as a constructor for the empty term list.

For the `adjoin-term` procedure, a generic procedure on the kind of term-list returns a lambda that takes a term as an argument. This avoids having to tag the term as well.

The procedures `coeff`, `order` and `make-term` have been put out of the polynomial package because they are needed in all three packages.

The only adaptation needed for the other procedures of the polynomial package is to replace calls to `(the-empty-termlist)` with calls to `make-empty-termlist` with the correct type. For simplicity, I used the type of the arguments rather than looking at the number of non-zero coefficients of the result to pick the best representation.

```
(define (adjoin-term term term-list)
  ((apply-generic 'adjoin-term term-list) term))
(define (first-term term-list)
  (apply-generic 'first-term term-list))
(define (rest-terms term-list)
  (apply-generic 'rest-terms term-list))
(define (empty-termlist? term-list)
  (apply-generic 'empty-termlist? term-list))
(define (make-empty-termlist type)
  ((get 'make type) '()))
(put 'no-drop? 'adjoin-term true)
(put 'no-drop? 'first-term true)
(put 'no-drop? 'rest-terms true)
(put 'no-drop? 'empty-termlist? true)

(define (make-term order coeff) (list order coeff))
(define (order term) (car term))
(define (coeff term) (cadr term))

(define (install-dense-termlist-package)
  (define (adjoin-term term term-list)
    (cond ((=zero? (coeff term))
           term-list)
          ((= (order term) (length term-list))
           (cons (coeff term) term-list))
          (else
            (adjoin-term term (cons 0 term-list)))))

  (define (first-term term-list)
    (make-term (- (length term-list) 1)
               (car term-list)))
  (define (rest-terms term-list) (cdr term-list))
  (define (empty-termlist? term-list) (null? term-list))
```

```
  (define (tag tl) (attach-tag 'dense tl))
  (put 'adjoin-term '(dense)
       (lambda (term-list)
         (lambda (term)
           (tag (adjoin-term term term-list)))))
  (put 'first-term '(dense)
       (lambda (term-list) (first-term term-list)))
  (put 'rest-terms '(dense)
       (lambda (term-list) (tag (rest-terms term-list))))
  (put 'empty-termlist? '(dense)
       (lambda (term-list) (empty-termlist? term-list)))
  (put 'make 'dense
       (lambda (termlist) (tag termlist))) )

(define (install-sparse-termlist-package)
  (define (adjoin-term term term-list)
    (if (=zero? (coeff term))
      term-list
      (cons term term-list)))
  (define (first-term term-list) (car term-list))
  (define (rest-terms term-list) (cdr term-list))
  (define (empty-termlist? term-list) (null? term-list))

  (define (tag tl) (attach-tag 'sparse tl))
  (put 'adjoin-term '(sparse)
       (lambda (term-list)
         (lambda (term)
           (tag (adjoin-term term term-list)))))
  (put 'first-term '(sparse)
       (lambda (term-list) (first-term term-list)))
  (put 'rest-terms '(sparse)
       (lambda (term-list) (tag (rest-terms term-list))))
  (put 'empty-termlist? '(sparse)
       (lambda (term-list) (empty-termlist? term-list)))
  (put 'make 'sparse
       (lambda (termlist) (tag termlist))))

(define (install-polynomial-package)
  ;; ... skipped ...
  (define (mul-terms L1 L2)
    (if (empty-termlist? L1)
      (make-empty-termlist (type-tag L1))
      (add-terms (mul-term-by-all-terms (first-term L1) L2)
```

```
                  (mul-terms (rest-terms L1) L2))))
  (define (mul-term-by-all-terms t1 L)
    (if (empty-termlist? L)
      (make-empty-termlist (type-tag L))
      (let ((t2 (first-term L)))
        (adjoin-term
          (make-term (+ (order t1) (order t2))
                     (mul (coeff t1) (coeff t2)))
          (mul-term-by-all-terms t1 (rest-terms L))))))
  ;; ... skipped ...
  'done)
```

Exercise 2.91

The `div-poly` and `div-terms` procedures can be defined in the following way. I used the polynomial package with two term-list representations from the previous exercise. For testing, I added `(put 'no-drop? 'div true)` to prevent `apply-generic` from attempting to simplify the list of two polynomials returned by the division, but it would be better to modify the way `apply-generic` decides whether to simplify its results to avoid disabling simplification entirely.

```
(define (sub-terms L1 L2)
  (add-terms L1 (neg-terms L2)))
(define (div-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
    (let ((div-result (div-terms (term-list p1)
                                 (term-list p2))))
      (list (make-polynomial (variable p1) (car div-result))
            (make-polynomial (variable p1) (cadr div-result))))
    (let ((same-var (make-same-var p1 p2)))
      (div (car same-var) (cdr same-var)))))
(define (div-terms L1 L2)
  (if (empty-termlist? L1)
    (list (make-empty-termlist (type-tag L1)) (make-empty-termlist (type-tag L1)))
    (let ((t1 (first-term L1))
          (t2 (first-term L2)))
      (if (> (order t2) (order t1))
        (list (make-empty-termlist (type-tag L1)) L1)
        (let ((new-c (div (coeff t1) (coeff t2)))
              (new-o (- (order t1) (order t2))))
          (let ((rest-of-result
                  (div-terms (sub-terms L1 (mul-term-by-all-terms (make-term new-o new-c) L2))
                             L2)))
            (list (adjoin-term (make-term new-o new-c) (car rest-of-result))
                  (cadr rest-of-result))))))))

(put 'div '(polynomial polynomial)
```

```
(lambda (p1 p2) (map tag (div-poly p1 p2))))
```

Hierarchies of types in symbolic algebra

Exercise 2.92

I tried to find a solution that was as general as possible: works for an arbitrary number of variables used in the polynomials, in any order. The variables are ordered alphabetically.

The `poly->monoms` procedure transforms a polynomial into an ordered list of monoms. A monom consists of a list of variable-power lists and a non-zero coefficient. The order on the variable-power lists is defined by: `(< (var1 n1) (var2 n2))` iff

```
(or (string<? var1 var2)
    (and (string=? var1 var2) (> n1 n2)))
```

(where we omitted the calls to `symbol->string` for simplicity. The monoms are ordered lexicographically on their lists of variables, with the empty list appearing last.

The `monoms->poly` procedure transforms the list of monoms built by `poly->monoms` back into a polynomial where each variable appears only at one level, and the variables appear in alphabetical order. Note that `monoms->poly` can actually return types other than polynomial, if the original polynomial was of degree 0.

The `make-same-var` procedure takes two polynomials and returns a pair of two polynomials in the same main variable. This is useful if the variables appearing in the two polynomials are different. If the same variables appear, but ordered differently, `monoms->poly` is enough.

Since `monoms->poly` can return types other than polynomial, it must return tagged data, and `add-poly`, `mul-poly`, `neg-poly` and `div-poly` have been transformed to return tagged data as well.

Complement:

I kept the original code in the case when the two polynomials are in the same variable, but in the case that, say, p1 is a polynomial in x with coefficients in y, some of them having coefficients in x as well, this won't work and the transformation to a canonical form should be used. We'll assume that this doesn't happen to avoid going through the transformation when it's not necessary.

It would also have been possible to rewrite `add-poly` and `mul-poly` so that the operations are made on the monoms rather than the terms, e.g. two polynomials can be added with:

```
(monoms->poly (merge-monoms (poly->monoms p1)
(poly->monoms p2)))
```

Additions to the polynomial package:

```
(define (make-monom vars coeff)
  (list vars coeff))
(define (vars monom)
  (if (null? monom) '()
    (car monom)))
(define (monom-coeff monom)
  (cadr monom))
(define (make-var-power var power)
  (list var power))
```

```
(define (get-var var-power)
  (car var-power))
(define (var-degree var-power)
  (cadr var-power))
(define (empty-var-list) '())

; Adds a list (var power) to the ordered list of variables defining a monom.
; e.g. (add-var (y 2) ((x 1) (y 1) (z 2))
; returns ((x 1) (y 3) (z 2))
(define (add-var var-power vars-list)
  (if (null? vars-list)
    (list var-power)
    (let* ((var (get-var var-power))
           (first (car vars-list))
           (first-var (get-var first))
           (var-name (symbol->string var))
           (first-var-name (symbol->string first-var)))
      (cond ((string<? first-var-name var-name)
              (cons first (add-var var-power (cdr vars-list))))
             ((string=? first-var-name var-name)
              (cons (make-var-power var
                                      (+ (var-degree first) (var-degree var-power)))
                    (cdr vars-list)))
             (else
               (cons var-power vars-list)) ))))

;; Merges two ordered lists of monoms.
(define (merge-monoms monoms1 monoms2)
  (cond ((null? monoms1) monoms2)
        ((null? monoms2) monoms1)
        (else
          (let* ((monom1 (car monoms1))
                 (monom2 (car monoms2))
                 (comp (compare-var-lists (vars monom1) (vars monom2))))
            (cond ((= 1 comp)
                   (cons (car monoms1)
                         (merge-monoms (cdr monoms1) monoms2)))
                  ((= -1 comp)
                   (cons (car monoms2)
                         (merge-monoms monoms1 (cdr monoms2))))
                  ((= 0 comp)
                   (let ((coeff (add (monom-coeff monom1) (monom-coeff monom2))))
                     (if (=zero? coeff)
                       (merge-monoms (cdr monoms1)
```

```
                                             (cdr monoms2))
                              (cons (make-monom (vars monom1)
                                                coeff)
                                    (merge-monoms (cdr monoms1)
                                                  (cdr monoms2)))))))))))

; 1 if var-list1 < var-list2
; 0 if var-list1 = var-list2
; -1 if var-list1 > var-list2
(define (compare-var-lists var-list1 var-list2)
  (cond ((and (null? var-list1) (null? var-list2))
         0)
        ((null? var-list1) -1)
        ((null? var-list2) 1)
        (else
          (let ((v1 (symbol->string (get-var (car var-list1))))
                (v2 (symbol->string (get-var (car var-list2)))))
            (cond ((string<? v1 v2) 1)
                  ((string>? v1 v2) -1)
                  (else
                    (let ((o1 (var-degree (car var-list1)))
                          (o2 (var-degree (car var-list2))))
                      (cond ((< o1 o2) -1)
                            ((> o1 o2) 1)
                            (else
                              (compare-var-lists (cdr var-list1) (cdr var-list2)))))))))) ))

; Transforms a polynomial into a list of monoms, e.g.
; (polynomial x dense (polynomial y dense 2 3) 0 2)
; becomes:
; ((((x 2) (y 1)) 2)
; (((x 2)) 3)
; (() 2))
(define (poly->monoms p)
  (let ((var (variable p))
        (terms (term-list p)))
    (if (empty-termlist? terms)
      '()
      (let* ((t1 (first-term terms))
             (o (order t1))
             (c (coeff t1))
             (var-power (make-var-power var o))
             (rest-monoms (poly->monoms (make-poly var (rest-terms terms)))))
        (cond ((eq? 'polynomial (type-tag c))
```

```
               (if (= o 0)
                 (merge-monoms (poly->monoms (contents c))
                               rest-monoms)
                 (merge-monoms (map (lambda (monom)
                                      (make-monom (add-var var-power (vars monom))
                                                  (monom-coeff monom)))
                                    (poly->monoms (contents c)))
                               rest-monoms)))
              ((=zero? c) rest-monoms)
              ((= o 0)
               (merge-monoms
                 (list (make-monom (empty-var-list) c))
                 rest-monoms))
              (else
                (merge-monoms
                  (list (make-monom (add-var var-power (empty-var-list)) c))
                  rest-monoms)))))))

; Assumes that monoms is a non-empty list of monoms and var is it's main
; variable. Returns the order of var in the first monom.
(define (order-var var monoms)
  (let ((var-list (vars (car monoms))))
    (if (or (null? var-list)
            (not (eq? (get-var (car var-list)) var)))
      0
      (var-degree (car var-list)))))

(define (build-poly var type monoms)
  ; Returns a pair with:
  ; - the coefficient of the term of degree o in the polynomial to build;
  ; - the remaining monoms.
  (define (first-coeff o monoms)
    (define (first-coeff* monoms)
      (cond ((null? monoms)
             (cons '() '()))
            ((= o 0)
             (cons monoms '()))
            (else
              (let* ((o1 (order-var var monoms)))
                (if (< o1 o)
                  (cons '() monoms)
                  (let ((first (car monoms))
                        (rest (first-coeff* (cdr monoms))))
                    (cons (cons (make-monom (cdr (vars first))
```

```
                                                 (monom-coeff first))
                                     (car rest))
                               (cdr rest))))))))
    (let ((monom-list (first-coeff* monoms)))
      (cons (monoms->poly type (car monom-list))
            (cdr monom-list))))

  (if (null? monoms)
    (make-polynomial var (make-empty-termlist type))
    (let* ((o (order-var var monoms))
           (coeff-and-rest (first-coeff o monoms)))
      (make-polynomial var
                        (adjoin-term (make-term o (car coeff-and-rest))
                                     (term-list (contents (build-poly var type (cdr coeff-and-rest))))))))

; Transforms a list of monoms into a polynomial or tagged data of a lower type.
(define (monoms->poly type monoms)
  (if (null? monoms)
    0
    (let* ((first (car monoms))
           (var-list (vars first)))
      (if (null? var-list)
        ; The list of monoms can have at most one monom with only a constant and
        ; no variables, so return it.
        (monom-coeff first)
        (let ((main-var (get-var (car var-list))))
          (build-poly main-var type monoms))))))

(define (reorder p)
  (monoms->poly (type-tag (term-list p))
                (poly->monoms p)))

(define (make-same-var p1 p2)
  (let* ((p1* (reorder p1))
         (p2* (reorder p2))
         (t1 (type-tag p1*))
         (t2 (type-tag p2*)))
    (cond ((and (not (eq? 'polynomial t1))
                (not (eq? 'polynomial t2)))
           (cons p1* p2*))
          ((not (eq? 'polynomial t1))
           (cons (make-polynomial (variable (contents p2*))
                                  (adjoin-term (make-term 0 p1*)
                                               (make-empty-termlist 'sparse)))
```

```
                  p2*))
          ((or (not (eq? 'polynomial t2))
               (not (same-variable? (variable (contents p1*)) (variable (contents p2*)))))
           (cons p1*
                 (make-polynomial (variable (contents p1*))
                                  (adjoin-term (make-term 0 p2*)
                                               (make-empty-termlist 'sparse)))))
          (else
            (cons p1* p2*)))))
```

Changes to existing code:

```
(define (add-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
    (make-polynomial (variable p1)
                     (add-terms (term-list p1)
                                (term-list p2)))
    (let ((same-var (make-same-var p1 p2)))
      (add (car same-var) (cdr same-var)))))

(define (mul-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
    (make-polynomial (variable p1)
                     (mul-terms (term-list p1)
                                (term-list p2)))
    (let ((same-var (make-same-var p1 p2)))
      (mul (car same-var) (cdr same-var)))))

(define (neg-poly p)
  (make-polynomial (variable p)
                   (neg-terms (term-list p))))

(define (div-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
    (let ((div-result (div-terms (term-list p1)
                                 (term-list p2))))
      (list (make-polynomial (variable p1) (car div-result))
            (make-polynomial (variable p1) (cadr div-result))))
    (let ((same-var (make-same-var p1 p2)))
      (div (car same-var) (cdr same-var)))))

(put 'add '(polynomial polynomial)
     (lambda (p1 p2) (add-poly p1 p2)))
(put 'mul '(polynomial polynomial)
     (lambda (p1 p2) (mul-poly p1 p2)))
```

```
(put 'neg '(polynomial)
     (lambda (p) (neg-poly p)))
(put 'sub '(polynomial polynomial)
     (lambda (p1 p2) (add-poly p1 (contents (neg-poly p2)))))
(put 'div '(polynomial polynomial)
     (lambda (p1 p2) (div-poly p1 p2)))
```

Extended exercise: Rational functions

Exercise 2.93

The changes to the rational packages are straightforward.

```
(define (install-rational-package)
  ;; Internal procedures.
  (define (numer x) (car x))
  (define (denom x) (cdr x))
  (define (make-rat n d)
    (cons n d))
  (define (add-rat x y)
    (make-rat (add (mul (numer x) (denom y))
                   (mul (numer y) (denom x)))
              (mul (denom x) (denom y))))
  (define (sub-rat x y)
    (make-rat (sub (mul (numer x) (denom y))
                   (mul (numer y) (denom x)))
              (mul (denom x) (denom y))))
  (define (mul-rat x y)
    (make-rat (mul (numer x) (numer y))
              (mul (denom x) (denom y))))
  (define (div-rat x y)
    (make-rat (mul (numer x) (denom y))
              (mul (denom x) (numer y))))

  ;; Interface to the rest of the system.
  (define (tag x) (attach-tag 'rational x))
  (put 'add '(rational rational)
       (lambda (x y) (tag (add-rat x y))))
  (put 'sub '(rational rational)
       (lambda (x y) (tag (sub-rat x y))))
  (put 'mul '(rational rational)
       (lambda (x y) (tag (mul-rat x y))))
  (put 'div '(rational rational)
       (lambda (x y) (tag (div-rat x y))))

  (put 'make 'rational
```

```
      (lambda (n d) (tag (make-rat n d))))
  'done)
```

Complement:
In order to continue to use all the fonctionalities defined in the exercises in the chapter, I had to modify the procedures defining `project` and `equ?` for rationals.

```
(put 'project '(rational)
     (lambda (r)
       (if (and (eq? (type-tag (numer r)) 'integer)
                (eq? (type-tag (denom r)) 'integer))
         (make-integer (quotient (numer r)
                                 (denom r)))
         0)))

(put 'equ? '(rational rational)
     (lambda (x y) (equ? (mul (numer x) (denom y))
                         (mul (numer y) (denom x)))))
```

Exercise 2.94

The procedures `remainder-terms` and `gcd-poly` can be implemented in the following way:

```
(define (remainder-terms a b)
  (cadr (div-terms a b)))

(define (gcd-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
    (make-poly (variable p1)
               (gcd-terms (term-list p1) (term-list p2)))
    (error "Polys not in same var -- GCD-POLY"
           (list p1 p2))))
```

Exercise 2.95

Dividing by hand, the remainder found after the first division is ${}^{1458}/{}_{169}x^2 - {}^{2916}/{}_{169}x + {}^{1458}/{}_{169}$, which divides Q_2.

With Gambit scheme, the first remainder is a polynomial with limited-precision decimal numbers as coefficients, and the second call does not return.

Exercise 2.96

a. The procedure `pseudoremainder-terms` can be defined as:

```
(define (pseudoremainder-terms a b)
  (let* ((o1 (degree-terms a))
         (o2 (degree-terms b))
         (c (get-coeff-by-degree-terms b o2)))
    (cadr (div-terms (mul-term-by-all-terms (make-term 0 (expt c (+ 1 (- o1 o2)))) a)
                     b))))
```

and the new version of `gcd-terms` is:

```
(define (gcd-terms a b)
  (if (empty-termlist? b)
    a
    (gcd-terms b (pseudoremainder-terms a b))))
```

On the example in exercise 2.95, `greatest-common-divisor` now produces the polynomial $1458x^2 - 2916x + 1458$.

b. The procedure can be rewritten in the following way to divide the coefficients of the answer by their greatest common divisor.

```
(define (gcd-terms a b)
  (if (empty-termlist? b)
    (let* ((coeffs-gcd (gcd-terms-coeffs a)))
      (divide-terms-by-int a coeffs-gcd))
    (gcd-terms b (pseudoremainder-terms a b))))

(define (divide-terms-by-int term-list n)
  (mul-term-by-all-terms (make-term 0 (make-rational 1 n)) term-list))

(define (gcd-terms-coeffs term-list)
  (cond ((empty-termlist? term-list)
         0)
        ((not (eq? (type-tag (coeff (first-term term-list)))
                   'integer))
         (error "Trying to compute GCD of a non-integer."
                term-list))
        (else
         (gcd (coeff (first-term term-list))
              (gcd-terms-coeffs (rest-terms term-list))))))
```

Exercise 2.97

a. The `reduce-terms` and `reduce-poly` procedures can be defined as:

```
(define (reduce-terms n d)
  (let* ((gcd-frac (gcd-terms n d))
         (o2 (degree-terms gcd-frac))
         (o1 (max (degree-terms n)
                  (degree-terms d)))
         (c (get-coeff-by-degree-terms gcd-frac o2))
         (mul-term (make-term 0 (expt c (+ 1 (- o1 o2)))))
         (n1 (quotient-terms (mul-term-by-all-terms mul-term n)
                             gcd-frac))
         (d1 (quotient-terms (mul-term-by-all-terms mul-term d)
                             gcd-frac)))
    (list (divide-terms-by-int n1 (gcd-terms-coeffs n1))
```

```
          (divide-terms-by-int d1 (gcd-terms-coeffs d1)))))

(define (quotient-terms n d)
  (car (div-terms n d)))

(define (reduce-poly p q)
  (if (same-variable? (variable p) (variable q))
    (map (lambda (term-list)
           (make-poly (variable p) term-list))
         (reduce-terms (term-list p) (term-list q)))
    (error "Polys not in same var -- REDUCE-POLY"
           (list p q))))
```

b. The only change needed besides adding the necessary entries to the associative table is to redefine `make-rat` as:

```
(define (make-rat n d)
  (let ((reduced (reduce n d)))
    (cons (car reduced) (cadr reduced))))
```

The result returned by `(add rf1 rf2)` corresponds to $\frac{-x^3-2x^2-3x-1}{-x^4-x^3+x+1}$, which is correctly reduced to lowest terms, though multiplying both the numerator and the denominator by -1 would seem more natural.

3 Modularity, Objects, and State

3.1 Assignment and Local State

3.1.1 Local State Variables

Exercise 3.1

The procedure `make-accumulator` can be written:

```
(define (make-accumulator sum)
  (lambda (x)
    (begin (set! sum (+ sum x))
           sum)))
```

Exercise 3.2

The `make-monitored` procedure can be written:

```
(define (make-monitored f)
  (let ((count 0))
    (define (mf x)
      (cond ((eq? x 'how-many-calls?)
             count)
            ((eq? x 'reset-count)
             (set! count 0))
            (else
              (set! count (+ count 1))
              (f x))))
    mf))
```

Exercise 3.3

The `make-account` procedure can be modified in the following way:

```
(define (make-account balance password)
  (define (withdraw amount)
    (if (>= balance amount)
      (begin (set! balance
                   (- balance amount))
             balance)
      "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
```

```
    balance)
  (define (incorrect-password amount)
    "Incorrect password")
  (define (dispatch given-pass m)
    (cond ((not (eq? given-pass password))
           incorrect-password)
          ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT"
                       m))))
  dispatch)
```

Exercise 3.4

The procedure can be rewritten as:

```
(define (make-account balance password)
  (define (withdraw amount)
    (if (>= balance amount)
      (begin (set! balance
                   (- balance amount))
             balance)
      "Insufficient funds"))

  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)

  (define (incorrect-password _)
    "Incorrect password")

  (define (call-the-cops _)
    "Cops called!")

  (let ((incorrect-count 0))
    (define (dispatch pass m)
      (if (not (eq? pass password))
        (begin
          (set! incorrect-count (+ incorrect-count 1))
          (if (> incorrect-count 7)
            call-the-cops
            incorrect-password))
        (begin
          (set! incorrect-count 0)
          (cond
            ((eq? m 'withdraw) withdraw)
```

```
            ((eq? m 'deposit) deposit)
            (else (error "Unknown request -- MAKE-ACCOUNT"
                         m))))))
    dispatch))
```

3.1.2 The Benefits of Introducing Assignment

Exercise 3.5

Using Gambit Scheme's `random-real` procedure, that generates a random real number between 0 and 1, `random-in-range` and the other procedures can be written:

```
(define (random-in-range low high)
  (let ((range (- high low)))
    (+ low (* range (random-real)))))

(define (estimate-integral P x1 x2 y1 y2 trials)
  (define (area-test)
    (let ((x (random-in-range x1 x2))
          (y (random-in-range y1 y2)))
      (P x y)))
  (* (- x2 x1)
     (- y2 y1)
     (monte-carlo trials area-test)))

(define (estimate-pi2 trials)
  (define (in-unit-circle? x y)
    (<= (+ (square x) (square y)) 1))
  (estimate-integral in-unit-circle? -1. 1. -1. 1. trials))
```

Exercise 3.6

The `rand` procedure can be rewritten as:

```
(define rand
  (let ((x random-init))
    (lambda (action)
      (cond ((eq? action 'reset)
             (lambda (new-value)
               (set! x new-value)))
            ((eq? action 'generate)
             (set! x (rand-update x))
             x)
            (else
              (error "Unknown request -- RAND" x))))))
```

3.1.3 The Costs of Introducing Assignment

Exercise 3.7

I simply added a `join` action to the account returned by `make-account` that creates an access with another password. I also make `incorrect-password` throw an error instead of simply returning a string, otherwise a call such as `(define new-acc (make-join account curr-pass new-pass))` with an incorrect current password will affect a string value to `new-acc` without reporting an error, and subsequent uses of the account will throw errors because `"Incorrect password"` is not a procedure.

```
(define (make-account balance password)
  (define (withdraw amount)
    (if (>= balance amount)
      (begin (set! balance
                   (- balance amount))
             balance)
      "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (incorrect-password amount)
    (error "Incorrect password"))
  (define (make-dispatch password)
    (lambda (given-pass m)
      (cond ((not (eq? given-pass password))
             incorrect-password)
            ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)
            ((eq? m 'join) make-dispatch)
            (else (error "Unknown request -- MAKE-ACCOUNT"
                         m)))))
  (make-dispatch password))

(define (make-join account curr-pass new-pass)
  ((account curr-pass 'join) new-pass))
```

Exercise 3.8

The procedure `f` returns:

- 0 if it is the first time it is called;
- the previous argument it was called with otherwise.

Thus, if we evaluate `(f 0)`, then `(f 1)`, we get 0 both times, but if we evaluate `(f 1)`, then `(f 0)`, we get 0 the first time and 1 the second.

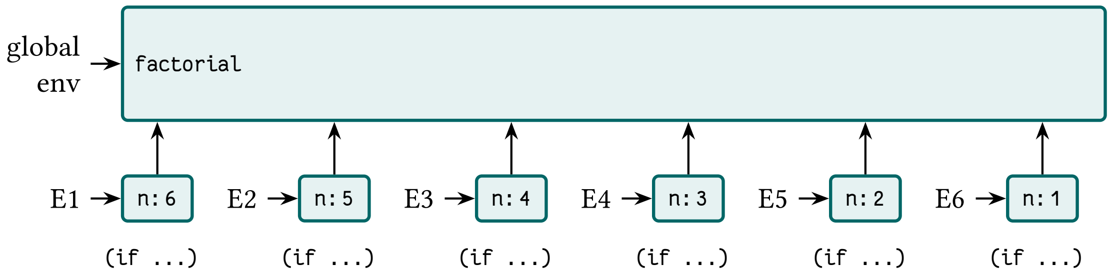

Figure 3.1: Environments created by evaluating `(factorial 6)` with the recursive procedure. In all the environments created, the code to evaluate corresponds to the body of the `factorial` procedure.

```
(define f
  (let ((arg 0))
    (lambda (x)
      (let ((prev-arg arg))
        (set! arg x)
        prev-arg))))
```

3.2 The Environment Model of Evaluation

3.2.1 The Rules for Evaluation

This subsection contains no exercises.

3.2.2 Applying Simple Procedures

Exercise 3.9

The environment structure created by evaluating `(factorial 6)` with both versions of the procedure are shown in figures 3.1 and 3.2.

3.2.3 Frames as the Repository of Local State

Exercise 3.10

The environments created after the execution of the three commands are shown in figures 3.3 and 3.4, see the captions for some details. As with the first version of `make-version`, each object created with a call to `make-version` uses a `balance` binding situated in an environment specific to the object.

In the second version, two environments are created instead of one, and the value of `initial-value` is unchanged.

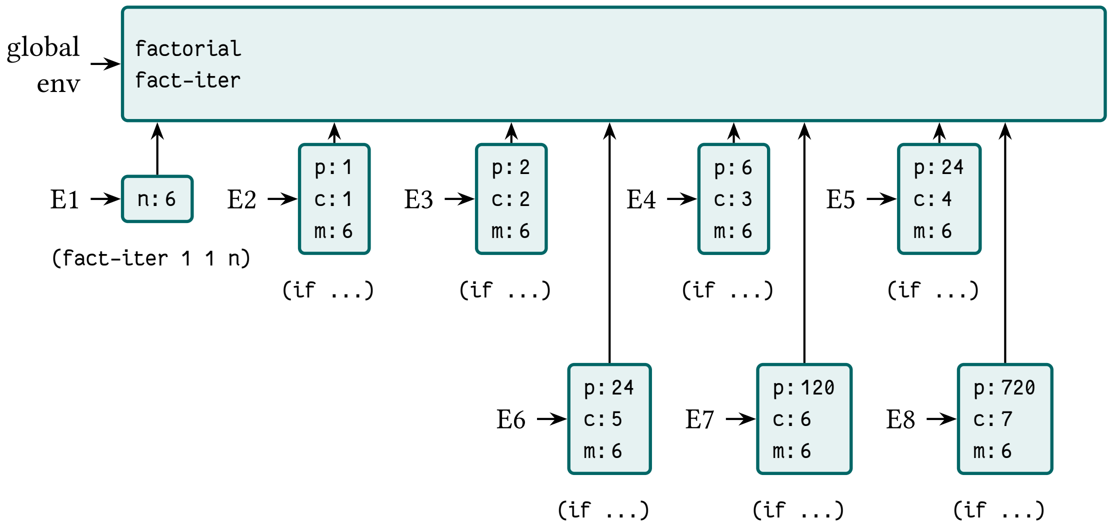

Figure 3.2: Environments created by evaluating `(factorial 6)` with the iterative procedure. In environments E2 to E8, the code to evaluate corresponds to the body of the `fact-iter` procedure.

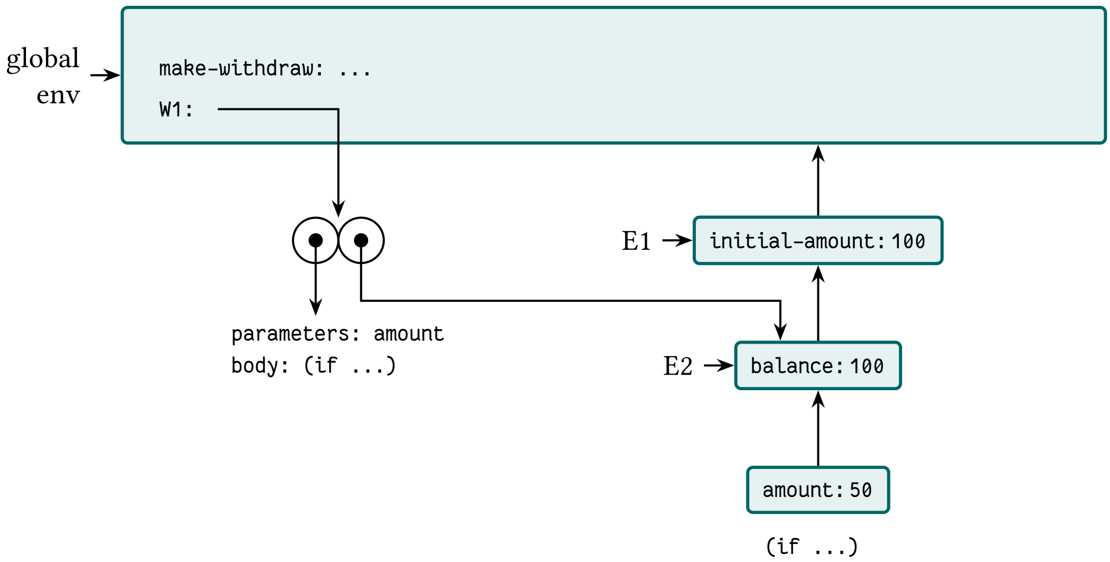

Figure 3.3: Environments created when executing `(W1 50)` after executing `(define (W1 (make-withdraw 100)))`. The environment E1 is created by the call to `make-withdraw`, E2 is created when the lambda procedure created by the `let` is executed. E2 is referenced by the procedure returned by `make-withdraw`. When `(W1 50)` is called, a new environment pointing to E2 is created, in which the body of `W1` is evaluated.

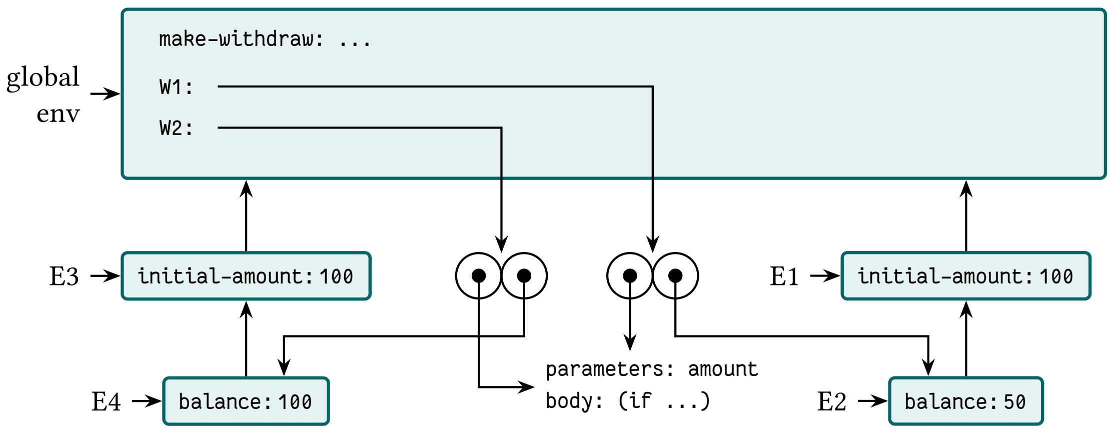

Figure 3.4: Environments created after the execution of `(define W1 (make-withdraw 100))`, followed by `(W1 50)`, then `(define W2 (make-withdraw 100))`. The call to `W1` modified the value of `balance` in E2, but `W2` uses the `balance` variable of environment E4.

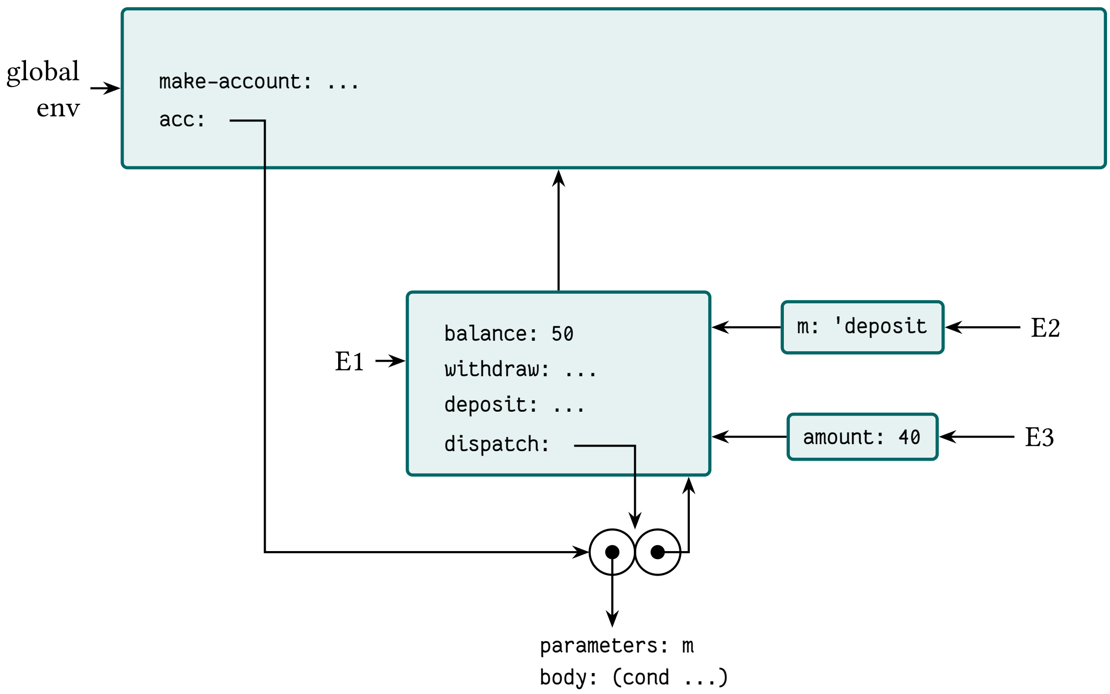

Figure 3.5: Environments when evaluating `((acc 'deposit) 40)` after evaluating `(define acc (make-account 50))`. E1 is created when defining `acc`, then the evaluation of `(acc 'deposit)` causes the creation of an environment referencing E1, and since the result of the evaluation is the procedure `deposit`, `(deposit 40)` is then evaluated in a new environment.

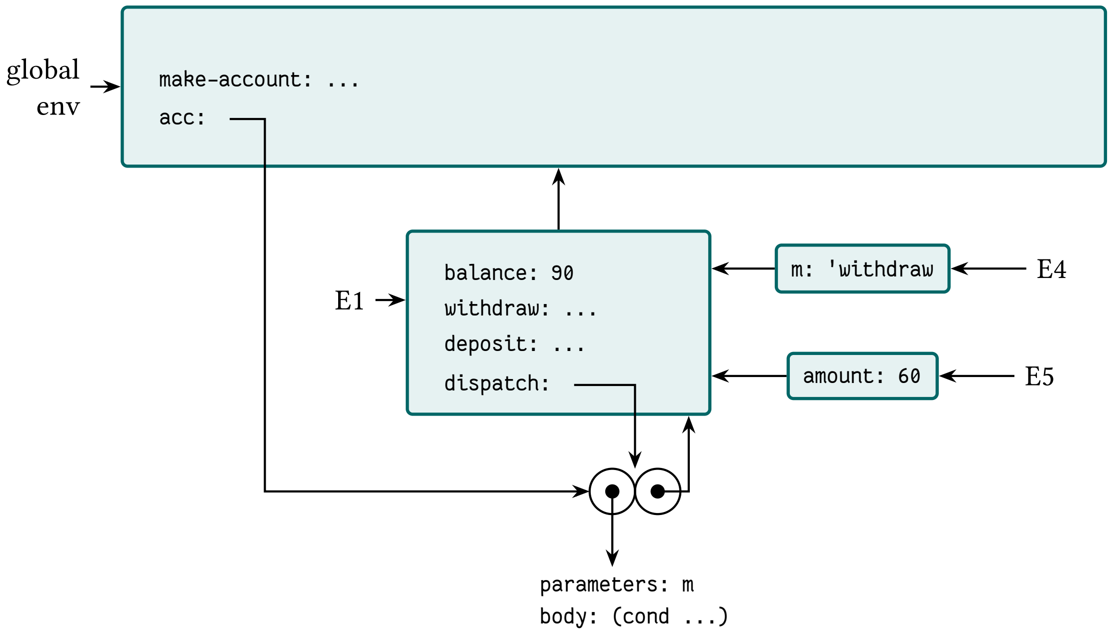

Figure 3.6: Environments during the evaluation of `((acc 'withdraw) 60)`.

3.2.4 Internal Definitions

Exercise 3.11

The environments generated by the evaluation of `(define acc (make-account 50))`, `((acc 'deposit) 40)` and `((acc 'withdraw) 60)` are shown in figures 3.5 and 3.6. The local state for `acc` is kept in the local environment referenced by the procedure object referenced by `acc`.

If a second environment is created, its local state is kept in a new environment created when evaluating the `make-account` procedure, so it does not interfere with `acc`'s local environment.

The environment structures of `acc` and `acc2` share the global environment.

3.3 Modeling with Mutable Data

3.3.1 Mutable List Structure

Exercise 3.12

The first response is `(b)`, the second response is `(b c d)`. The figure 3.7 shows the lists `x`, `y` and `z` right after the definition of `z`. The figure 3.8 shows the lists `x`, `y` and `w` after the definition of `w`.

Exercise 3.13

The structure `z` is shown in figure 3.9.

If we try to compute `(last-pair z)`, we get an infinite loop since `z` has a cycle.

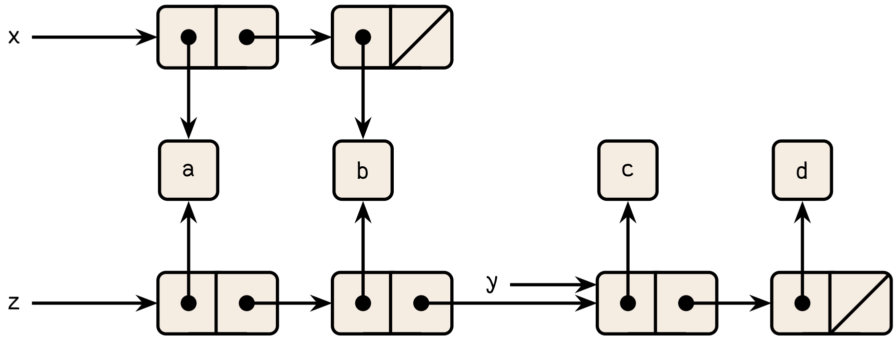

Figure 3.7: The lists `x`, `y` and `z` right after the definition of `z`.

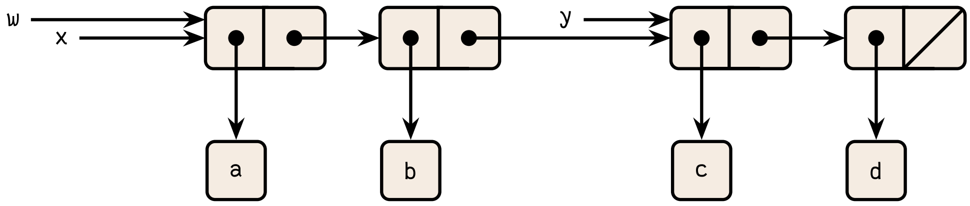

Figure 3.8: The lists `x`, `y` and `w` right after the definition of `w`.

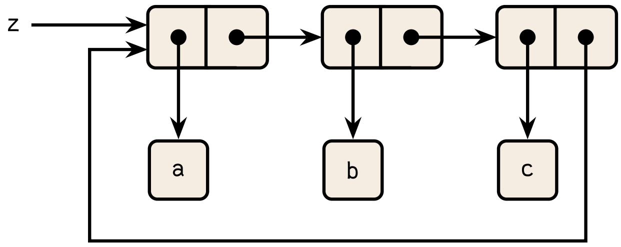

Figure 3.9: The structure created by `(define z (make-cycle (list 'a 'b 'c))`.

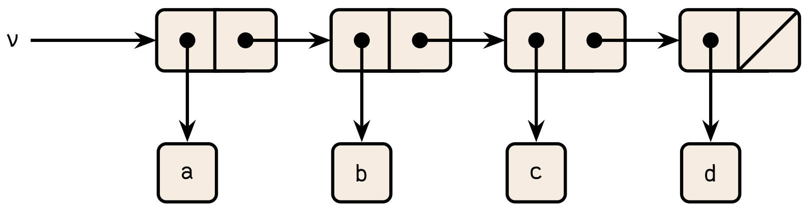

Figure 3.10: The list to which `v` is bound initially.

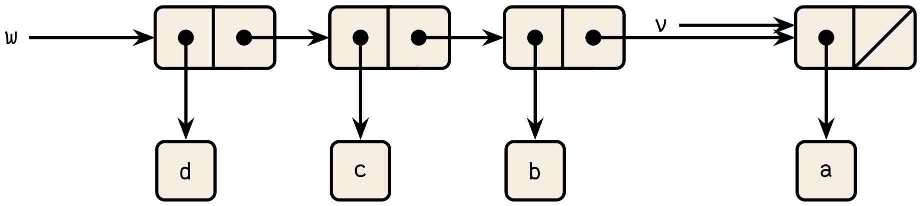

Figure 3.11: The lists `v` and `w` after calling `mystery`.

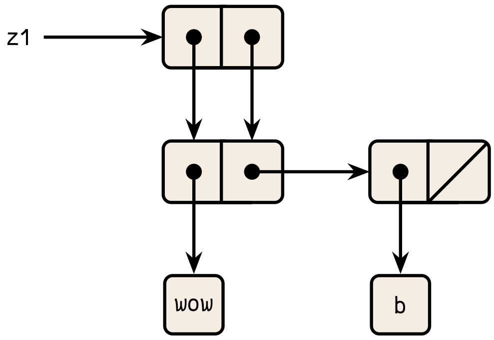

Figure 3.12: The list `z1` after applying `set-to-wow!` to it.

Exercise 3.14

The `mystery` procedure reverses the elements of the list. Figure 3.10 shows the list `v` as it is initially, and figure 3.11 shows the lists `v` and `w` after evaluating `(define w (mystery v))`. The values of `v` and `w` would be `(a)` and `(d c b a)`.

Sharing and identity

Exercise 3.15

The figures 3.12 and 3.13 show the effect of `set-to-wow!` on `z1` and `z2`.

Exercise 3.16

Figure 3.14 shows examples of list structures made up of exactly three pairs for which Ben's procedure returns 3, 4, 7, or never at all. These structures can be defined in the following way, using `make-cycle` from exercise 3.13 for the last one.

```
(define x (list 'a 'b 'c))

(define b-nil (cons 'b '()))
(define y (cons b-nil (cons 'a b-nil)))

(define a-nil (cons 'a '()))
(define z1 (cons a-nil a-nil))
(define z (cons z1 z1))
```

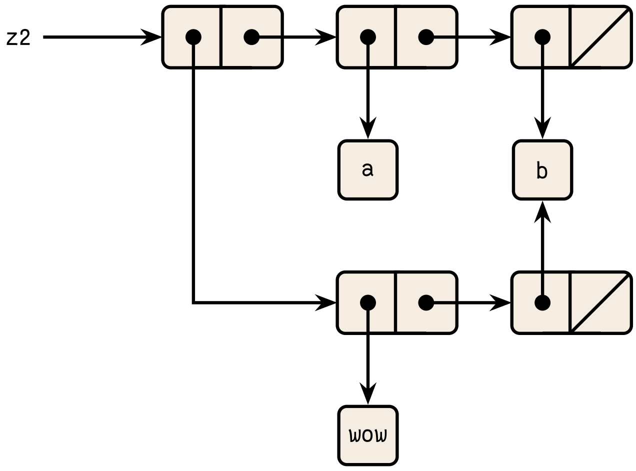

Figure 3.13: The list `z2` after applying `set-to-wow!` to it.

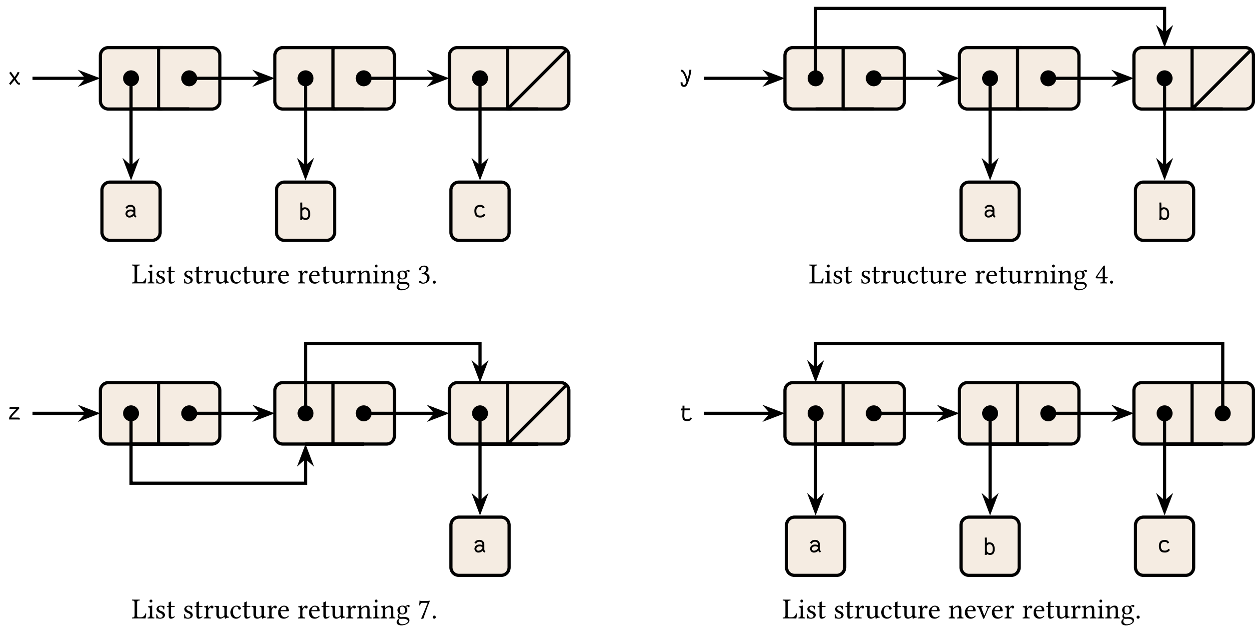

Figure 3.14: Structures made of exactly three pairs for which Ben's procedure returns different values.

```
(define t (make-cycle (list 'a 'b 'c)))
```

Exercise 3.17

A possible solution is:

```
(define (count-pairs x)
  (define (find-unique-pairs x visited)
    (if (or (not (pair? x))
            (memq x visited))
      visited
      (find-unique-pairs (car x)
                         (find-unique-pairs (cdr x) (cons x visited)))))
  (length (find-unique-pairs x '())))
```

Exercise 3.18

Here is a possible solution:

```
(define (contains-cycle? x)
  (define (helper x seen)
    (cond ((null? x) #f)
          ((memq x seen) #t)
          (else
            (helper (cdr x) (cons x seen)))))
  (helper x '()))
```

Exercise 3.19

We go through the list with two pointers: one advancing one step at a time, the other advancing two steps at a time. If the list contains a cycle, they'll end up pointing to the same pair after a while. Otherwise, the second one will reach the end of the list.

```
(define (contains-cycle? x)
  (define (helper x x2)
    (cond ((eq? x x2) #t)
          ((or (null? x2) (null? (cdr x2))) #f)
          (else (helper (cdr x) (cddr x2)))))
  (if (null? x)
    #f
    (helper x (cdr x))))
```

Mutation is just assignment

Exercise 3.20

Figure 3.15 shows the environments created by the definitions of `x` and `z`. When `(set-car! (cdr z) 17)` is evaluated, `(cdr z)` is evaluated first. This creates an environment E3 pointing to E2, where `(z 'cdr)` is evaluated, returning the `x` defined in the global environment. So the expression becomes `(set-car! x 17)`, evaluated in the global environment. The

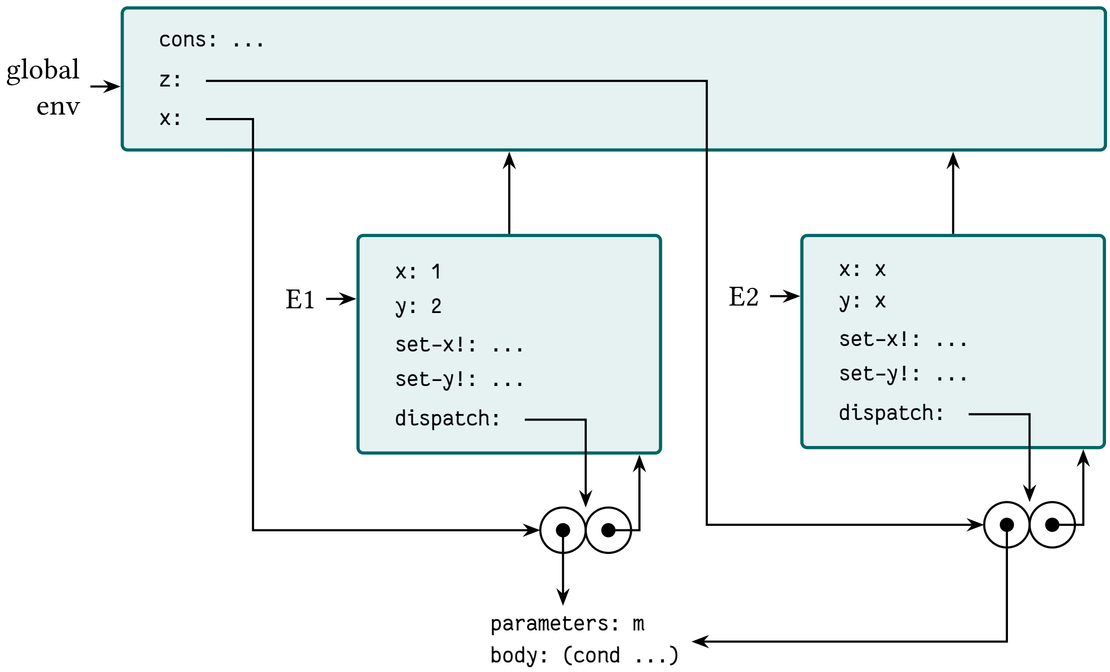

Figure 3.15: Environment structure after the definitions of x and z. In E2, the values of x and y correspond to the x defined in the global environment.

expression `((x 'set-car!) 17)` is then evaluated. The evaluation of `(x 'set-car!)` leads to the creation of an environment E4 pointing to E1, in which the evaluation returns the `set-x!` procedure from environment E1. The evaluation of the expression obtained `(set-x! 17)` leads to the modification of the value of `x` in environment E1. Lastly, `(car x)` is evaluated in an environment pointing to E1, so the value returned is 17.

3.3.2 Representing Queues

Exercise 3.21

The elements actually contained in the queue are only the contents of the queue's `car`. The queue's `cdr` points to the last element of the queue, so it is printed twice. The rear pointer is not updated when the last element from the queue is deleted, so the former last element is still printed although the queue is empty.

```
(define (print-queue queue)
  (write (front-ptr queue))
  (newline))
```

Exercise 3.22

The constructor, selectors and mutators can be defined in the following way. The implementation of the queue operations doesn't need to be modified.

```
(define (make-queue)
  (let ((front-ptr '())
        (rear-ptr '()))
    (define (set-front-ptr! value)
      (set! front-ptr value))
    (define (set-rear-ptr! value)
      (set! rear-ptr value))
    (define (dispatch m)
      (cond ((eq? m 'set-front-ptr!) set-front-ptr!)
            ((eq? m 'set-rear-ptr!) set-rear-ptr!)
            ((eq? m 'front-ptr) front-ptr)
            ((eq? m 'rear-ptr) rear-ptr)
            (else
              (error "Unknown request -- MAKE-QUEUE" m))))
    dispatch))

(define (front-ptr queue) (queue 'front-ptr))
(define (rear-ptr queue) (queue 'rear-ptr))
(define (set-front-ptr! queue item) ((queue 'set-front-ptr!) item))
(define (set-rear-ptr! queue item) ((queue 'set-rear-ptr!) item))
```

Exercise 3.23

To respect the requirement that all operations should be accomplished in $\Theta(1)$ steps, it's necessary to use a doubly-linked list instead of a singly-linked list. The deque is represented as a pair containing a pointer to the first element of a list and a pointer to the last element of this list just like the queue. Each element of the list is a pair containing the value and a pointer to the previous element of the list. Since such a structure can't be printed since it contains infinite loops as soon as the deque contains at least 2 elements, the insertion and deletion procedures return a list representation of the contents of the deque.

```
(define (make-deque) (cons '() '()))
(define (front-ptr deque) (car deque))
(define (rear-ptr deque) (cdr deque))
(define (set-front-ptr! deque item) (set-car! deque item))
(define (set-rear-ptr! deque item) (set-cdr! deque item))

(define (make-elt value prev next)
  (cons (cons value prev) next))
(define (value elt) (caar elt))
(define (prev elt) (cdar elt))
(define (next elt) (cdr elt))
(define (set-prev! elt item) (set-cdr! (car elt) item))
(define (set-next! elt item) (set-cdr! elt item))

(define (empty-deque? deque) (null? (front-ptr deque)))

(define (contents deque)
  (define (contents-helper front)
    (if (null? front)
      '()
      (cons (value front)
            (contents-helper (next front)))))
  (contents-helper (front-ptr deque)))

(define (front-deque deque)
  (if (empty-deque? deque)
    (error "FRONT called with an empty deque" (contents deque))
    (value (front-ptr deque))))

(define (rear-deque deque)
  (if (empty-deque? deque)
    (error "REAR called with an empty deque" (contents deque))
    (value (rear-ptr deque))))

(define (insert-deque! deque item pos)
  (cond ((empty-deque? deque)
         (let ((new-pair (make-elt item '() '())))
           (set-front-ptr! deque new-pair)
```

```
           (set-rear-ptr! deque new-pair)
           (contents deque)))
        ((eq? pos 'front)
         (let ((new-pair (make-elt item '() (front-ptr deque))))
           (set-prev! (front-ptr deque) new-pair)
           (set-front-ptr! deque new-pair)
           (contents deque)))
        ((eq? pos 'rear)
         (let ((new-pair (make-elt item (rear-ptr deque) '())))
           (set-next! (rear-ptr deque) new-pair)
           (set-rear-ptr! deque new-pair)
           (contents deque)))))

(define (front-insert-deque! deque item)
  (insert-deque! deque item 'front))

(define (rear-insert-deque! deque item)
  (insert-deque! deque item 'rear))

(define (delete-deque! deque pos)
  (cond ((empty-deque? deque)
         (error "DELETE! called with an empty deque" (contents deque)))
        ((eq? pos 'front)
         (set-front-ptr! deque (next (front-ptr deque)))
         (if (empty-deque? deque)
           (set-rear-ptr! deque '())
           (set-prev! (front-ptr deque) '()))
         (contents deque))
        ((eq? pos 'rear)
         (set-rear-ptr! deque (prev (rear-ptr deque)))
         (if (null? (rear-ptr deque))
           (set-front-ptr! deque '())
           (set-next! (rear-ptr deque) '()))
         (contents deque))))

(define (front-delete-deque! deque)
  (delete-deque! deque 'front))

(define (rear-delete-deque! deque)
  (delete-deque! deque 'rear))
```

3.3.3 Representing Tables

Exercise 3.24

The only necessary change is to define an `assoc` procedure that uses the provided `same-key?` instead of `equal?`. The code below is a possible solution for a one-dimensional table. For multi-dimensional tables, there is no reason to assume that the successive keys are of the same type or that the same equality test must be used at every level, so multiple comparison procedures should be provided, and the right procedure should be passed as an argument to `assoc` at each level of the table.

```
(define (make-table same-key?)
  (let ((local-table (list '*table*)))
    (define (assoc key records)
      (cond ((null? records) #f)
            ((same-key? key (caar records))
             (car records))
            (else (assoc key (cdr records)))))
    (define (lookup key)
      (let ((record (assoc key (cdr local-table))))
        (if record
          (cdr record)
          #f)))
    (define (insert! key value)
      (let ((record (assoc key (cdr local-table))))
        (if record
          (set-cdr! record value)
          (set-cdr! local-table
                    (cons (cons key value)
                          (cdr local-table))))))
    (define (dispatch m)
      (cond ((eq? m 'lookup-proc) lookup)
            ((eq? m 'insert-proc!) insert!)
            (else (error "Unknown operation -- TABLE" m))))
    dispatch))

(define table (make-table (lambda (k1 k2)
                            (< (abs (- k1 k2)) 0.01))))
(define get (table 'lookup-proc))
(define put (table 'insert-proc!))
```

Exercise 3.25

It would be possible to use the lists as keys directly but I don't think that's the point of the exercise. The solution I implemented allows different numbers of keys for different records, however it does not allow keys that are prefixes of each other: if a value is stored under the key `'(a b c)` and another value is then stored under `'(a b)`, the record for `'(a b c)` is silently deleted, and vice versa. It would also be inefficient for large tables since it checks whether the record found really contains a table by going through the whole record before looking for the following key in it.

```
(define (is-table? x)
  (define (is-records? x)
    (or (null? x)
        (and (pair? x) (pair? (car x)) (is-records? (cdr x)))))
  (and (pair? x)
       (is-records? (cdr x))))

(define (make-table)
  (let ((local-table (list '*table*)))
    (define (lookup-subtable keys subtable)
      (cond ((not subtable) #f)
            ((null? keys) (cdr subtable))
            ((not (is-table? subtable)) #f)
            (else
              (lookup-subtable (cdr keys) (assoc (car keys) (cdr subtable))))))

    (define (lookup keys)
      (lookup-subtable keys local-table))

    (define (insert-one! key value subtable)
      (let ((record (assoc key (cdr subtable))))
        (if record
          (set-cdr! record value)
          (set-cdr! subtable
                    (cons (cons key value)
                          (cdr subtable))))))

    (define (insert-subtable! keys value subtable)
      (let ((subsubtable (assoc (car keys) (cdr subtable))))
        (cond ((or (not subsubtable)
                   (not (is-table? subsubtable)))
               (set! subsubtable (list (car keys)))
               (set-cdr! subtable (cons subsubtable (cdr subtable)))))
        (if (null? (cdr keys))
          (set-cdr! subsubtable value)
          (insert-subtable! (cdr keys) value subsubtable))))

    (define (insert! keys value)
      (insert-subtable! keys value local-table))
```

```
    (define (dispatch m)
      (cond ((eq? m 'lookup-proc) lookup)
            ((eq? m 'insert-proc) insert!)
            (else (error "Unknown operation -- TABLE" m))))
    dispatch))

; For testing
(define table (make-table))
(define get (table 'lookup-proc))
(define put (table 'insert-proc))
```

Exercise 3.26

Here is an example of a one-dimensional table where the keys are ordered with the given comparison procedure `<?`. The local table is stored as a binary tree of records instead of a headed list. I used mutable trees instead of using the `adjoin-tree` procedure for binary trees of section 2.3.3 to avoid stacking recursive calls and creating multiple intermediate trees.

```
(define (entry tree) (car tree))
(define (left-branch tree) (cadr tree))
(define (right-branch tree) (caddr tree))
(define (make-tree entry left right)
  (list entry left right))
(define (make-leaf value)
  (make-tree value '() '()))
(define (empty-tree? tree) (null? tree))
(define (set-left-branch! tree branch)
  (set-car! (cdr tree) branch))
(define (set-right-branch! tree branch)
  (set-car! (cddr tree) branch))

(define (make-table <?)
  (let ((local-table '()))
    (define (assoc key records)
      (cond ((null? records) #f)
            ((equal? key (car (entry records)))
             (entry records))
            ((<? key (car (entry records)))
             (assoc key (left-branch records)))
            (else (assoc key (right-branch records)))))

    (define (lookup key)
      (let ((record (assoc key local-table)))
        (if record
          (cdr record)
          #f)))
```

```scheme
    (define (insert-binary-tree! key value tree)
      (cond ((equal? key (car (entry tree)))
             (set-cdr! (entry tree) value))
            ((<? key (car (entry tree)))
             (if (empty-tree? (left-branch tree))
               (set-left-branch! tree (make-leaf (cons key value)))
               (insert-binary-tree! key value (left-branch tree))))
            (else
              (if (empty-tree? (right-branch tree))
                (set-right-branch! tree (make-leaf (cons key value)))
                (insert-binary-tree! key value (right-branch tree))))))

    (define (insert! key value)
      (if (empty-tree? local-table)
        (set! local-table (make-leaf (cons key value)))
        (insert-binary-tree! key value local-table))
      'ok)

    (define (dispatch m)
      (cond ((eq? m 'lookup-proc) lookup)
            ((eq? m 'insert-proc!) insert!)
            (else (error "Unknown operation -- TABLE" m))))
    dispatch))

; For testing
(define table (make-table <))
(define get (table 'lookup-proc))
(define put (table 'insert-proc!))
```

Exercise 3.27

Figure 3.16 shows some of the environments created during the evaluation of `(memo-fib 3)`. The call to `memoize` creates a local environment E1 containing a table, and the procedure returned by `memoize` points to this environment. When `memo-fib` is called for the first time with a value, it puts the computed result into the table. When it is called again with the same value, it simply returns the value stored in the table instead of making recursive calls. So `memo-fib` computes the nth Fibonacci number in linear time because it never computes the value for the same number twice.

If we had defined `memo-fib` to be `(memoize fib)`, it would not have worked because `fib` calls itself rather than `memo-fib` recursively.

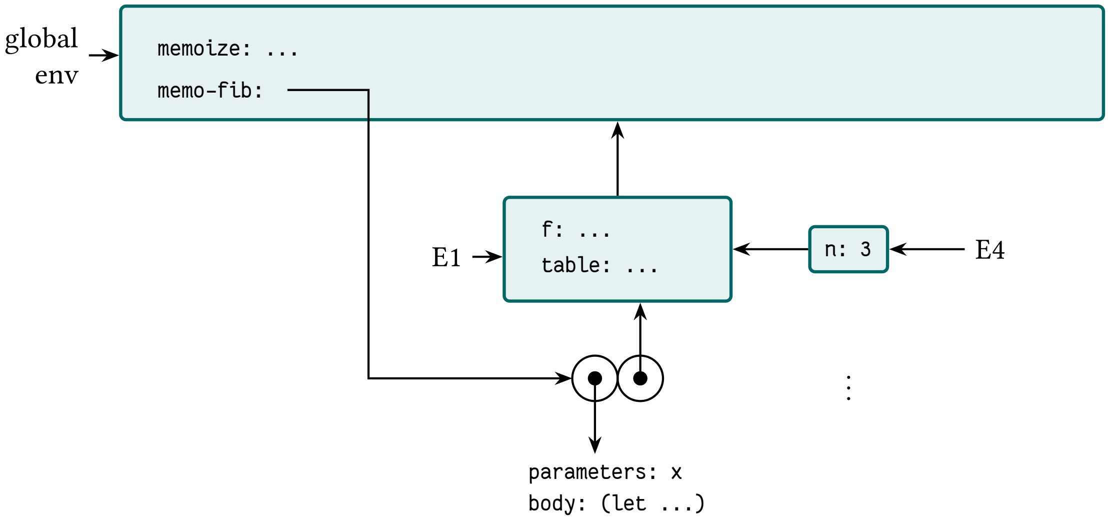

Figure 3.16: Environments created during the evaluation of `(memo-fib 3)`.

3.3.4 A Simulator for Digital Circuits

Primitive function boxes

Exercise 3.28

Here is a definition of an or-gate similar to the definiton of the and-gate:

```
(define (or-gate a1 a2 output)
  (define (or-action-procedure)
    (let ((new-value
            (logical-or (get-signal a1) (get-signal a2))))
      (after-delay or-gate-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! a1 or-action-procedure)
  (add-action! a2 or-action-procedure)
  'ok)

(define (valid-signal? s)
  (or (= s 0) (= s 1)))

(define (logical-or s1 s2)
  (cond ((or (not (valid-signal? s1))
             (not (valid-signal? s2)))
         (error "Invalid signal" (list s1 s2)))
        ((or (= 1 s1) (= 1 s2)) 1)
        (else 0)))
```

Exercise 3.29

Using the logical equivalency between $a \vee b$ and $\neg(\neg a \wedge \neg b)$, we can build an or-gate from and-gates and inverters. The delay for an or-gate built this way is the and-gate delay plus twice the inverter delay.

```
(define (or-gate a1 a2 output)
  (let ((n1 (make-wire))
        (n2 (make-wire))
        (n3 (make-wire)))
    (inverter a1 n1)
    (inverter a2 n2)
    (and-gate n1 n2 n3)
    (inverter n3 output)
    'ok))
```

Exercise 3.30

The ripple-carry-adder can be defined using the following procedure:

```
(define (ripple-carry-adder a b s c)
  (if (= (length a) 1)
    (half-adder (car a) (car b) (car s) c)
    (let ((c-in (make-wire)))
      (ripple-carry-adder (cdr a) (cdr b) (cdr s) c-in)
      (full-adder (car a) (car b) c-in (car s) c))))
```

Let's use the following notations: $o =$ or-delay, $a =$ and-delay, $i =$ inverter-delay. Let's call R_{Ci} the delay to obtain C_i in a ripple-carry adder, R_{Si} the delay to obtain S_i in a ripple-carry adder, F_s the delay to obtain the sum bit in a full-adder, F_c the delay to obtain the carry bit in a full-adder, H_s the delay to obtain the carry in a half-adder, H_c the delay to obtain the carry in a half-adder.

For the half-adder we have $H_c = a$ and $H_s = \max(2a + i, o + a)$.

For the full-adder, we have $F_s = 2H_s = 2\max(2a + i, o + a)$, and $F_c = H_s + H_c + o = a + o + \max(2a + i, o + a)$, from where $F_s \geq F_c$.

For the full-carry adder, we have: $R_{Ci} = (n - i) \times F_c$ and $R_{Si} = R_{Ci} + F_s$ because only the carry is transmitted to the following full-adders.

The delay to obtain the complete output from an n-bit ripple-carry adder is $R_{C1} + \max(F_s, F_c) = R_{C1} + F_s = (n - 1)F_c + F_s$. In terms of the delays for and-gates, or-gates and inverters, the delay for the ripple-carry adder is $(n - 1)(a + o) + (n + 1)\max(2a + i, o + a)$.

Representing wires

Exercise 3.31

The initialization is necessary to compute correctly all the signals with the initial values of the inputs. Since there are inverters, not all signals are 0 even if all inputs and outputs are 0, and if they are not initialized properly, the values computed after changing the inputs could be wrong as well.

In the case of the half-adder example, without the initialization, the output of the inverter would have a signal of 0 instead of 1 initially. Since setting the first input to 1 does not trigger a change to the inverter's input, the sum would remain 0 after the propagation. After setting the second input to 1, the sum would still remain 0 and the carry would become 1.

Implementing the agenda

Exercise 3.32

If the (last in, first out) order is used, the values set by the actions executed last in a segment could set incorrect values because they are not taking into account all the changes that occurred in that segment.

If the inputs a_1 and a_2 are 0 and 1 are both changed in the same segment, if the action triggered by the change of a_2 is executed first, an action setting the output value to 0 is scheduled, than when the change to a_1 is taken into account, the same action is scheduled again at the same time, so the order of execution of these two actions does not matter.

However, if the change to a_1 is treated first, an action setting the output signal to 1 is scheduled, and when a_2 switches to 0 an action setting the output signal to 0 is scheduled at the same time. If these actions are executed (last in, first out), the final output value will be 1 instead of 0 because the scheduled action executed last used stale values for some signals.

3.3.5 Propagation of Constraints

Exercise 3.33

The averager can be defined in the following way, since we want $a + b = 2c$:

```
(define (averager a b c)
  (let ((d (make-connector))
        (s (make-connector)))
    (constant 2 d)
    (multiplier c d s)
    (adder a b s)))
```

Exercise 3.34

The squarer defined in this way works only in one direction because the multiplier needs two of its three connectors to have a value to be able to set the third connector's value. If the value of `a` is set, the value of `b` will be set correctly, but if the value of `b` is set, the value of `a` won't be set because only one connector has a value.

Exercise 3.35

The squarer can be defined in the following way:

```
(define (squarer a b)
  (define (process-new-value)
    (if (has-value? b)
      (if (< (get-value b) 0)
        (error "square less than 0 -- SQUARER" (get-value b))
```

```
        (set-value! a
                    (sqrt (get-value b))
                    me))
      (if (has-value? a)
        (set-value! b
                    (square (get-value a))
                    me))))
  (define (process-forget-value)
    (forget-value! a me)
    (forget-value! b me)
    (process-new-value))
  (define (me request)
    (cond ((eq? request 'I-have-a-value)
           (process-new-value))
          ((eq? request 'I-lost-my-value)
           (process-forget-value))
          (else
            (error "Unknown request -- SQUARER" request))))
  (connect a me)
  (connect b me)
  me)
```

Exercise 3.36

Figure 3.17 shows the environment structure in which the expression is evaluated. The environments created when defining `a` and `b` are similar to those created in exercise 3.10: an environment E1 (resp. E3) is created for the evaluation of `(make-connector)`, then since `(make-connector)` contains a `let`, a new environment E2 (resp. E4) pointing to E1 (resp. E3) is created. The variables `value`, `informant`, `constraints` and the local procedures `set-my-value`, `forget-my-value`, `connect` and `me` are defined in E2 (resp. E4). The local procedure `me` pointing to E2 (resp. E4) is returned and bound to `a` (resp. `b`).

The following environments are created during the evaluation of `(set-value! a 10 'user)`:

- E5: for the evaluation of `(set-value! a 10 'user)`. It points to the global environment since `set-value!` is defined there.
- E6: for the evaluation of `(a 'set-value!)`. It points to E2 since `a` is a procedure pointing to E2. The evaluation returns the `set-my-value` procedure from E2.
- E7: for the evaluation of `(set-my-value 10 'user)`. It points to E2 since `set-my-value` is the procedure returned at the previous step.
- E8: After `(has-value? me)` is evaluated (environments omitted) the values of `value` and `informant` are changed in `set-my-value`, `for-each-except` is called, which leads to the creation of environment E8 pointing to the global environment since that's where `for-each-except` was defined. The `loop` procedure is defined in E8, so the evaluation of `(loop list)` leads to the creation of E9 pointing to E8.

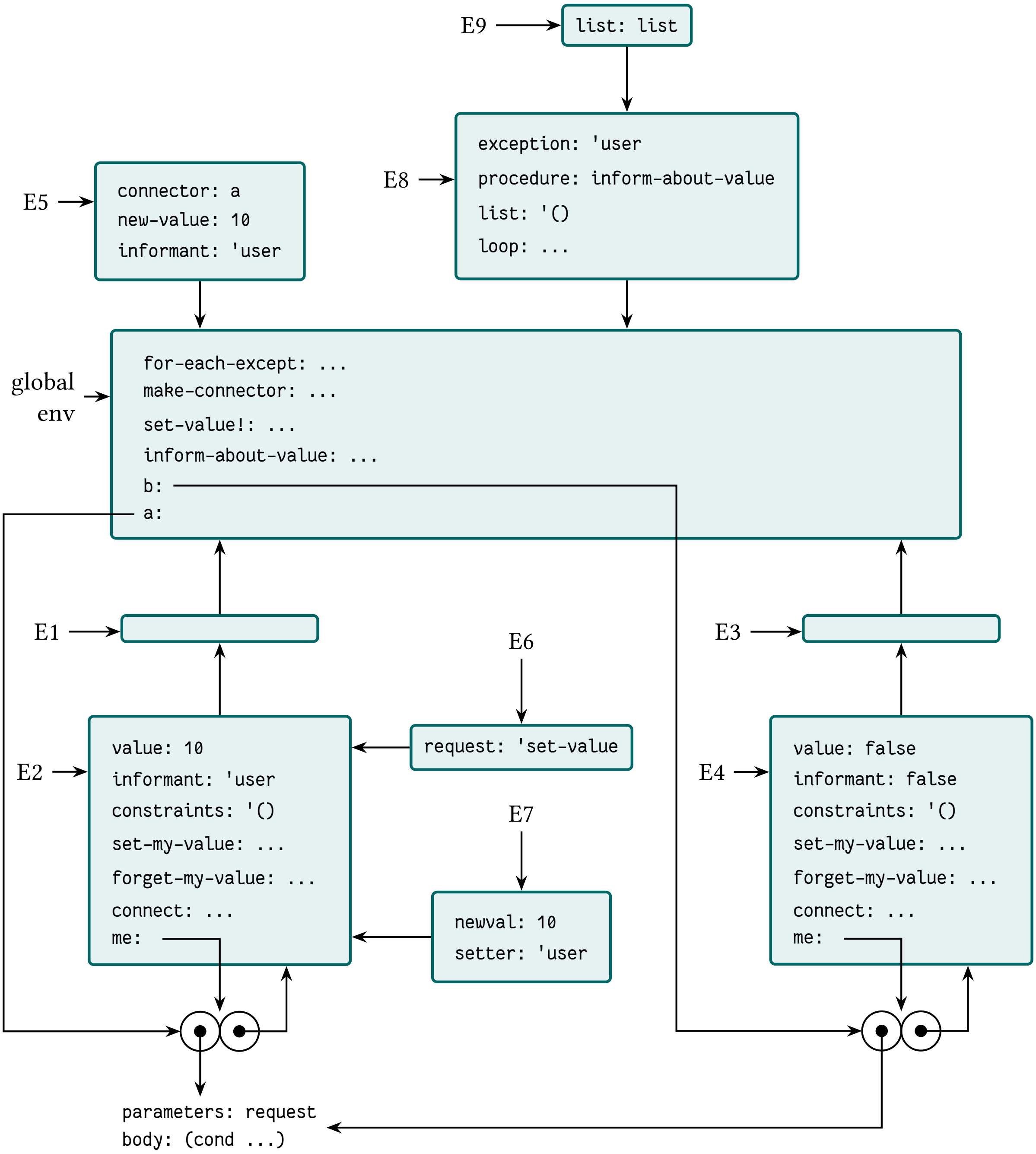

Figure 3.17: Environment structure in which the expresion `(for-each-except setter inform-about-value constraints)` is evaluated.

There are no constraints here so no further environments are created. Otherwise, for each constraint, an environment pointing to the global environment would be created, in which `(inform-about-value construint)` would be evaluated. Then `(constraint 'I-have-a-value)` would be evaluated in an environment pointing to the constraint's local environment.

Exercise 3.37

The procedures can be defined in the following way:

```
(define (c- x y)
  (let ((z (make-connector)))
    (adder y z x)
    z))

(define (c* x y)
  (let ((z (make-connector)))
    (multiplier x y z)
    z))

(define (c/ x y)
  (let ((z (make-connector)))
    (multiplier y z x)
    z))

(define (cv value)
  (let ((x (make-connector)))
    (constant value x)
    x))
```

3.4 Concurrency: Time Is of the Essence

3.4.1 The Nature of Time in Concurrent Systems

Exercise 3.38

a. If no interleaving is possible, the possible final values are \$35, \$40, \$45 and \$50:
   - Peter, Paul, Mary: \$45;
   - Paul, Peter, Mary: \$45;
   - Mary, Peter, Paul: \$40;
   - Mary, Paul, Peter: \$40;
   - Peter, Mary, Paul: \$35;
   - Paul, Mary, Peter: \$50.

b. Some of the possible other values are \$110, \$80, \$55, \$90. Figures 3.18 and 3.19 show timing diagrams explaining how the value \$110 and \$90 can occur.

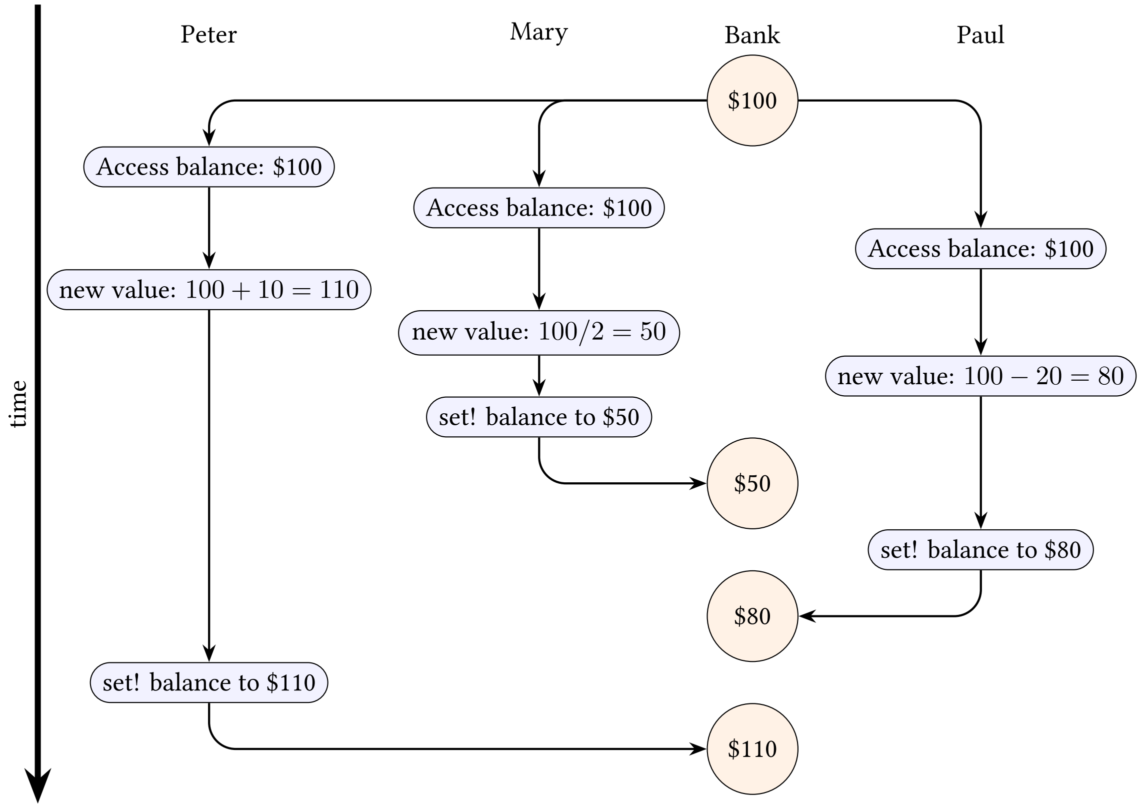

Figure 3.18: A timing diagram showing how the final value can be $110.

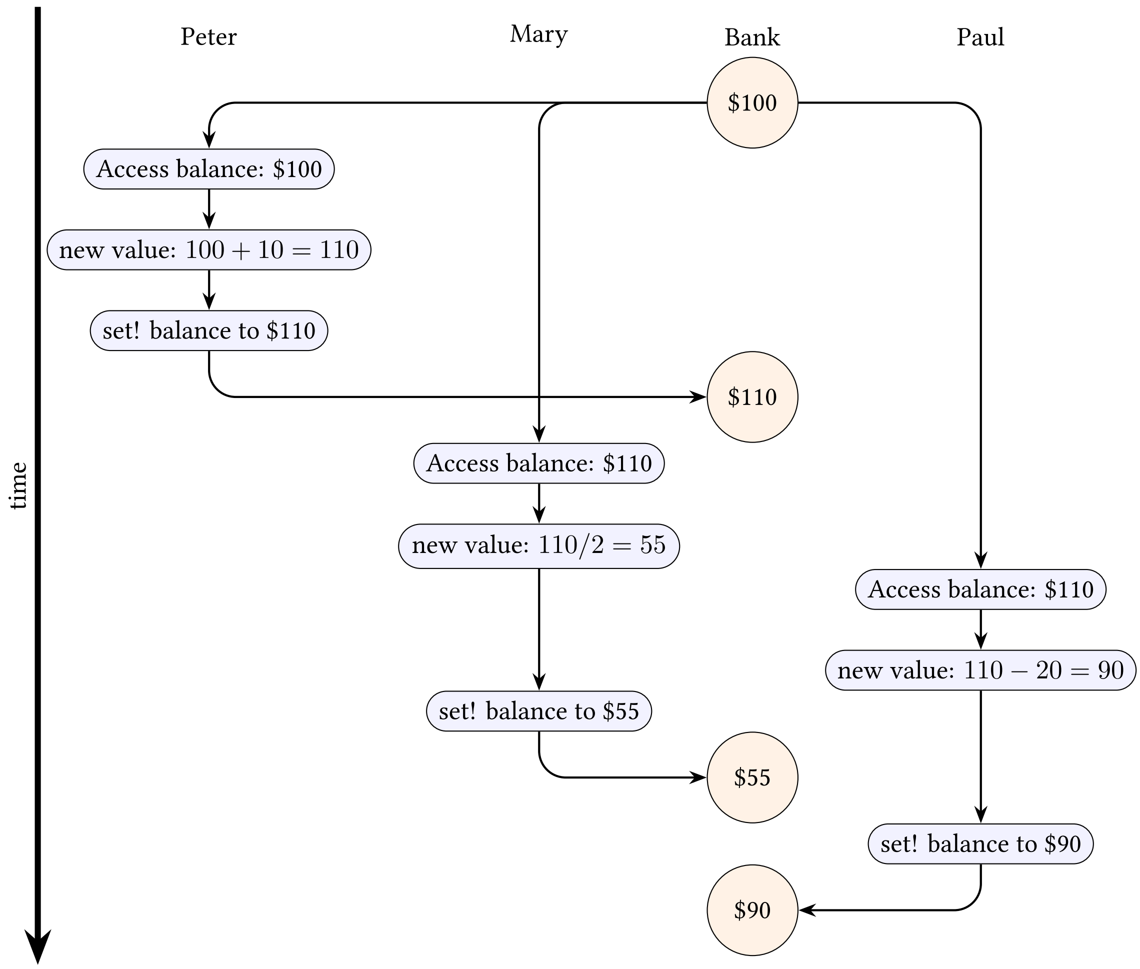

Figure 3.19: A timing diagram showing how the final value can be $90.

3.4.2 Mechanisms for Controlling Concurrency

Serializers in Scheme

Exercise 3.39

The remaining possibilities are 101, 121 and 100. The value of `x` can't change between the two times that P_1 accesses it to evaluate `(* x x)` so 110 is not possible anymore. The value of `x` can't change during the execution of P_2 so 11 is not possible anymore. The final value can be 100 if P_1 accesses `x` and computes the final value as 100 but P_2 accesses it and sets it to 11 before P_1 can set it to 100.

Exercise 3.40

The possible values are:

- 100: P_1 accesses `x` twice and P_2 accesses `x` thrice, then `x` P_2 sets `x` to 1000 before P_1 sets it to 100.
- 1000: P_1 accesses `x` twice and P_2 accesses `x` thrice, then `x` P_1 sets `x` to 100 before P_2 sets it to 1000.
- 10 000: either P_1 accesses `x` once before P_2 sets it to 1000, or P_2 accesses `x` twice before P_1 sets it to 100.
- 100 000: P_2 accesses `x` once before P_1 sets it to 100.
- 1 000 000: the two procedures execute sequentially in any order.

If the procedures are serialized the only possible value is 100 000.

Exercise 3.41

I don't think serializing access to balance is necessary because both `withdraw` and `deposit` make a single assignment to `balance`, so accessing `balance` concurrently with one of these procedures will reflect the state of the account either before or after the withdrawal or deposit, but it will correspond to a real state of the account.

Exercise 3.42

The change proposed by Ben Bitdiddle is safe to make. The two versions of `make-account` allow the same concurrency.

Complexity of using multiple shared resources

Exercise 3.43

Since each exchange run individually exchanges the balances of two of the accounts, if any number of exchanges happen sequentially the balances of the accounts will still be $10, $20 and $30 in some order.

Figure 3.20 shows how the account balances can be different than $10, $20 and $30 with the first version of the account-exchange program. The `withdraw` and `deposit` procedures from each account are serialized, so each of them is an atomic operation. Each exchange operation

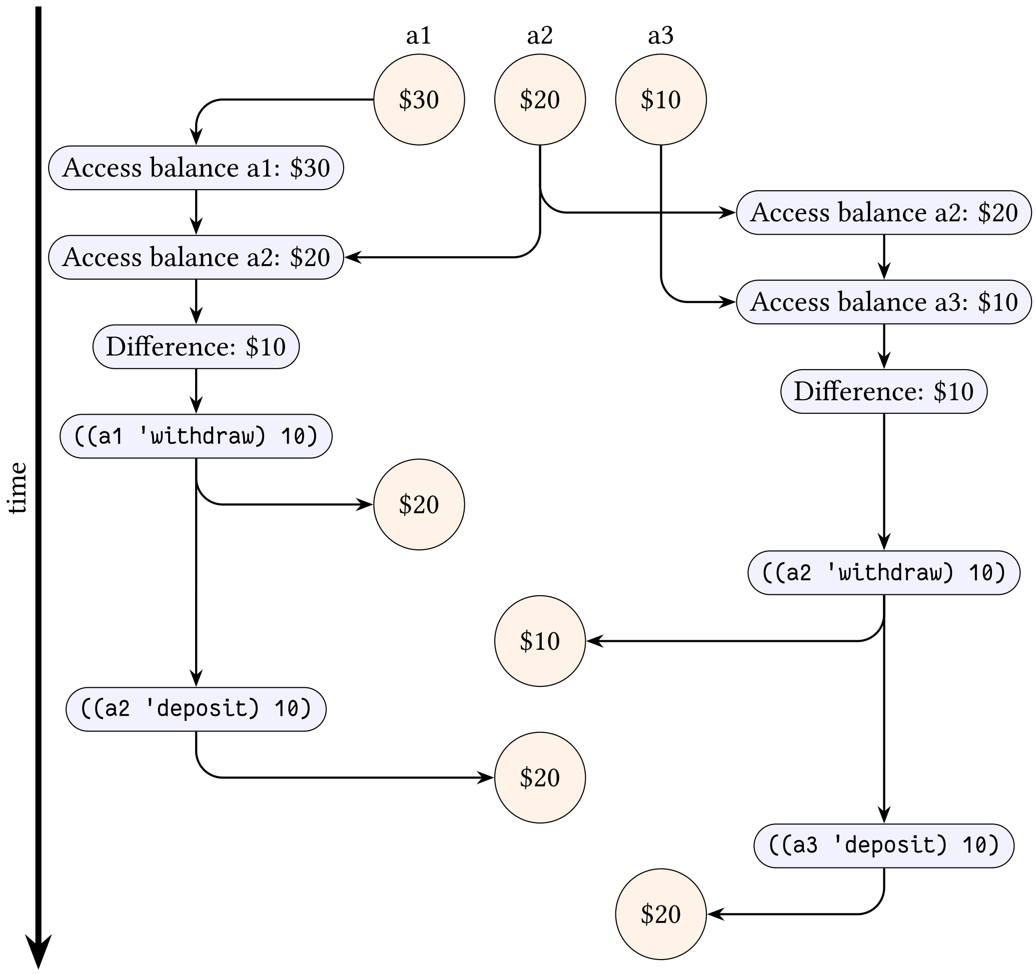

Figure 3.20: A timing diagram showing how the account balances can all three be $20 after exchanging concurrently `a1` and `a2` on one side, `a2` and `a3` on the other side.

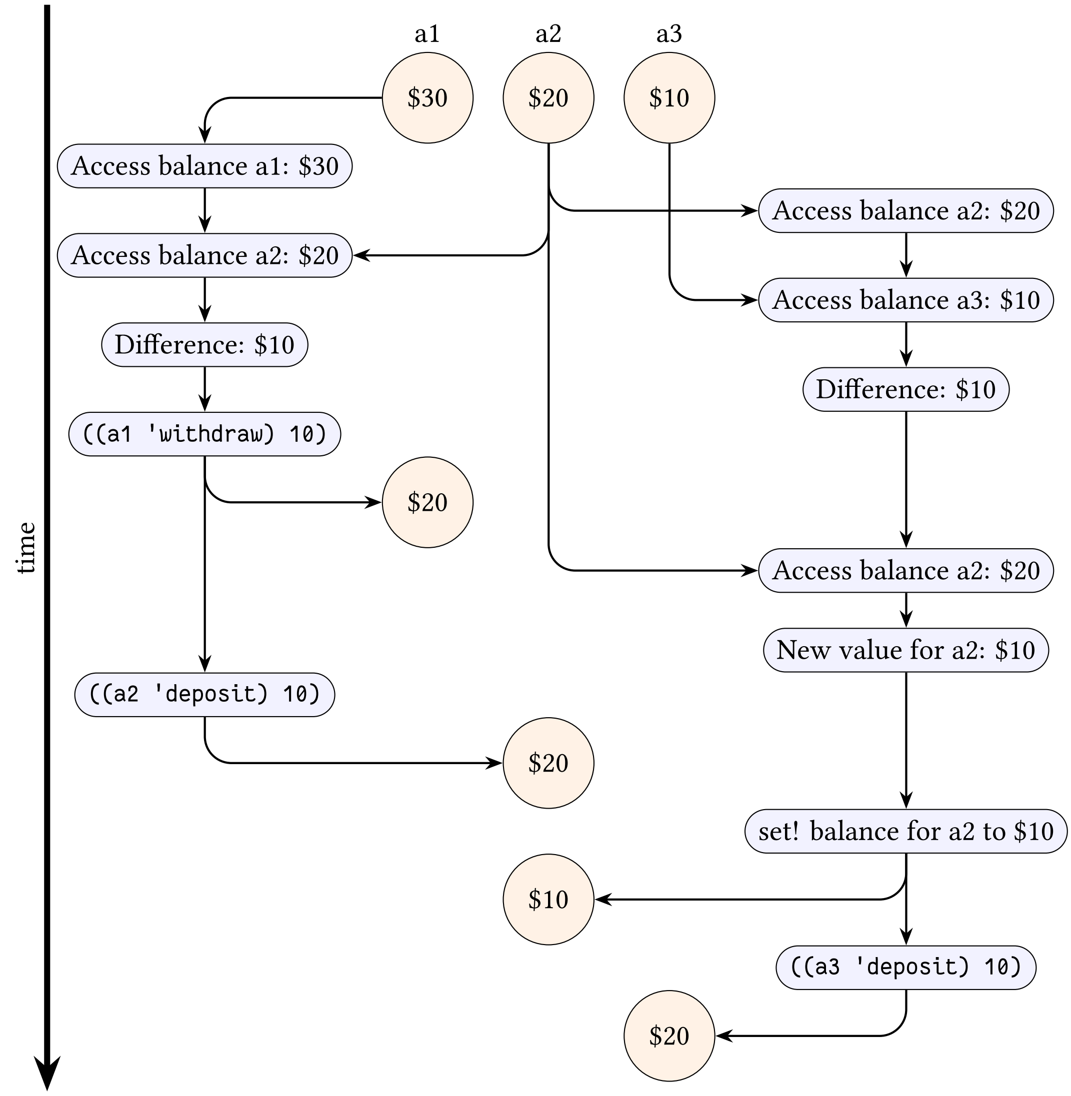

Figure 3.21: A timing diagram showing how the sum of the balances in the accounts is not preserved if the transactions on individual accounts are not serialized.

removes an amount from an account and adds the same amount to another account, and these operations are serialized, so the sum of the balances is preserved.

Figure 3.21 shows a case where the sum of the balances of the accounts is not preserved after exchanges between `a1` and `a2` on one side, `a2` and `a3` on the other side. Compared to the version from figure 3.20, the only transaction for a single account that does not happen as if it were serialized is `((a2 'withdraw) 10)`: the first exchange procedure sets the account's balance to $20 before the second procedure can set it to $10, and the final sum is $50 instead of $60.

Exercise 3.44

Louis is wrong, there is no problem with Ben Bitdiddle's procedure: at the end of the transfer, `amount` has been withdrawn from `from-account` and deposited on `to-account`, it does not matter whether other procedures access the accounts in-between. The fundamental difference with the exchange problem is that for the transfer the amount is an argument to the procedure, so there are only two atomic operations, one on each account. In the case of exchange there are two atomic operations for each account: an access to the balance and then either a deposit or a withdrawal.

Exercise 3.45

The problem with Louis' reasoning is that when a procedure serialized with an account's serializer attempts to withdraw or deposit to that account, it calls a procedure serialized with the same serializer and it gets stuck forever waiting for the mutex it holds to be available. In the case of `serialized-exchange`, it'll wait forever while trying to execute `((account1 'withdraw) difference)`.

Implementing serializers

Exercise 3.46

Figure 3.22 shows how the implementation of `test-and-set!` can fail if two procedures access the cell and find its contents at false before both set its contents to true.

Exercise 3.47

a. The following implementation of semaphores uses a mutex to guard a variable holding the number of available accesses to the semaphore:

```
(define (make-semaphore n)
  (let ((mutex (make-mutex))
        (left n))
    (define (acquire)
      (mutex 'acquire)
      (if (> left 0)
        (begin
          (set! left (- left 1))
          (mutex 'release))
        (begin
          (mutex 'release)
          (acquire))))
```

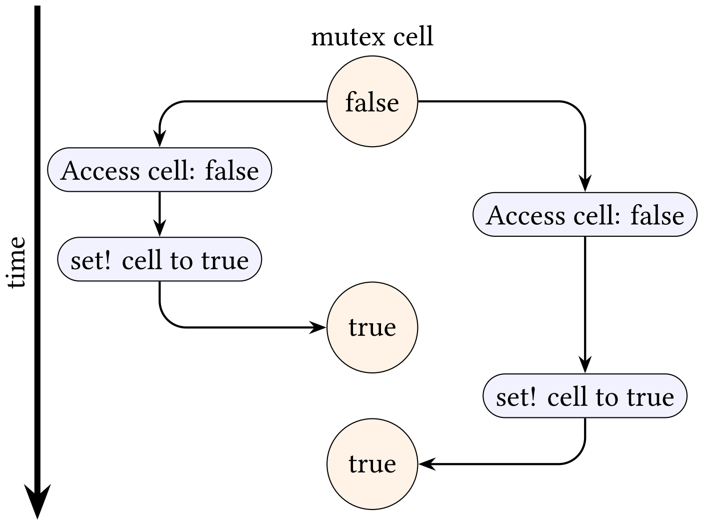

Figure 3.22: A timing diagram showing how the mutex implementation can allow to processes to acquire the mutex at the same time.

```
    (define (release)
      (mutex 'acquire)
      (if (< left n)
        (set! left (+ left 1)))
      (mutex 'release))
    (define (the-semaphor m)
      (cond ((eq? m 'acquire)
             (acquire))
            ((eq? m 'release)
             (release))))
    the-semaphor))
```

b. The following implementation uses a cell holding the number of available accesses, and an atomic `test-and-set!` operation to change the value of that cell:

```
(define (make-semaphore n)
  (let ((cell (list n)))
    (define (the-semaphore m)
      (cond ((eq? m 'acquire)
             (if (test-and-set! cell n -1)
               (the-semaphore 'acquire)))
            ((eq? m 'release)
             (test-and-set! cell n 1))))
    the-semaphore))

(define (test-and-set! cell max inc)
```

```
    (without-interrupts
      (lambda ()
        (let ((new-value (+ (car cell) inc)))
          (if (or (< new-value 0)
                  (> new-value max))
            true
            (begin (set-car! cell (+ (car cell) inc))
                   false))))))
```

Deadlock

Exercise 3.48

The `serialized-exchange` and `make-account-and-serializer` procedures can be modified in the following way to incorporate the numbering technique. It works because if two procedures need to access the same two accounts, they will both attempt to access the same account first and one of them will wait until the other one is done with no deadlock. More generally, if one procedure P1 already holds a lock L1 and waits to acquire a lock L2 held by another procedure P2, we know that P2 won't attempt to acquire L1, otherwise it would have done so before acquiring L2, so the lock held by P1 can't prevent P2 from completing.

```
(define (serialized-exchange account1 account2)
  (let ((n1 (account1 'number))
        (n2 (account2 'number))
        (serializer1 (account1 'serializer))
        (serializer2 (account2 'serializer)))
    (if (< n1 n2)
      ((serializer2 (serializer1 exchange)) account1 account2)
      ((serializer1 (serializer2 exchange)) account1 account2))))

(define (make-account-and-serializer balance account-number)
  (define (withdraw amount)
    (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Insufficient funds."))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((balance-serializer (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)
            ((eq? m 'balance) balance)
            ((eq? m 'serializer) balance-serializer)
            ((eq? m 'number) account-number)
```

```
            (else (error "Unknown request -- MAKE-ACCOUNT" m))))
    dispatch))
```

Exercise 3.49

If the resources to lock are not all known in advance, it's of course impossible to order the locks. For instance, let's assume that we must change a field in a row of a table in a database, and then find rows in other tables that contain the new value of the field to update these rows. (Not sure of a concrete example, but it's the idea...)

3.5 Streams

Complement:
In order to make the code in this section work with Gambit scheme, it's necessary to define `the-empty-stream` and `stream-null?`, as well as the macros `delay` and `cons-stream`. This can be done with the following:

```
(define-macro (delay p) `(memo-proc (lambda () ,p)))

(define-macro (cons-stream a b)
               `(cons ,a (delay ,b)))

(define the-empty-stream '())
(define (stream-null? stream) (null? stream))
```

3.5.1 Streams Are Delayed Lists

Exercise 3.50

The generalized version of `stream-map` is:

```
(define (stream-map proc . argstreams)
  (if (stream-null? (car argstreams))
    the-empty-stream
    (cons-stream
      (apply proc (map stream-car argstreams))
      (apply stream-map
             (cons proc (map stream-cdr argstreams))))))
```

Exercise 3.51

After the first expression, the interpreter prints `0`. This is the result of applying `show` to the first element of the stream in `stream-map`, in `(cons-stream (proc (stream-car s)) ...)`.

After the second expression, it prints `1 2 3 4 5 5` (one value per line). Each call to `stream-cdr` on the result of `stream-map` causes a call to `show` as the following value is evaluated. Then the value returned by `stream-ref` is returned.

After the third expression, it prints `6 7 7` (one value per line). Since `delay` uses `memo-proc`, `show` is only called for the elements of the stream not visited during the previous evaluation. Then the value `7` is returned.

Exercise 3.52

Since `delay` uses `memo-proc`, `accum` is called only once for each element of the enumerated interval, so the elements of `seq` are the sums of successive integers: $1, 3, 6, 10, 15, \ldots$. Furthermore, `sum` is equal to the sum of the integers from 1 to $n+1$, where n is the maximum of the indices of the elements already visited in the stream `seq`.

After the definition of `seq`, the value of `sum` is 1 because `accum` was applied to the first element of the enumeration by `stream-map` ($n = 0$).

The definition of `y` causes `stream-cdr` to be called on `seq` until the first even element is found. This happens for the third element of the stream interval, and then the value of `sum` is 6 ($n = 2$).

The definition of `y` causes `stream-cdr` to be called on `seq` until the first multiple of 5 is found. This is the fourth element of the stream, which is 10 ($n = 3$).

The eighth even element of `seq` is 136, so after evaluationg `(stream-ref y 7)`, `sum` will be equal to 136, which is also the printed response to the evaluation.

The evaluation of `display-stream` causes the stream `seq` to be fully evaluated, so at the end the value of `sum` is the last value of that stream: `210`. The evaluation of `(display-stream z)` displays the elements of `seq` that are multiples of 5, followed by `done`, one element per line: `10 15 45 55 105 120 190 210 done`.

The responses would differ if `delay` were implemented without using `memo-proc`. Since the result of `accum` depends on the value of a variable that changes with every call to `accum`, the stream `seq` would be completely different each time it is used. It becomes a lot harder to reason about what `seq`, `sum`, `y`, `z` represent since they don't correspond to something clearly definable.

3.5.2 Infinite Streams

Exercise 3.53

The elements of this stream are the powers of 2.

Exercise 3.54

The procedure `mul-streams` and the stream `factorials` can be defined as:

```
(define (mul-streams s1 s2)
  (stream-map * s1 s2))

(define factorials (cons-stream 1 (mul-streams factorials
                                               (stream-cdr integers))))
```

We use `(stream-cdr integers)` rather than `integers`, otherwise the nth element would be $n!$ and not $(n+1)!$.

Exercise 3.55

The procedure `partial-sums` can be defined as:

```
(define (partial-sums s)
  (add-streams s
               (cons-stream 0 (partial-sums s))))
```

Exercise 3.56

The required stream can be constructed as:

```
(define S (cons-stream 1 (merge (merge (scale-stream S 2)
                                       (scale-stream S 3))
                                (scale-stream S 5))))
```

Exercise 3.57

Only $n-1$ additions are performed to compute the nth Fibonacci number since the values already computed are not computed again.

If `memo-proc` is not used, the values are recomputed every time they are needed, so the number of additions is exponential.

Exercise 3.58

The given stream computes the expansion of the ratio of `num` and `den` in basis `radix`. More precisely, if we call n, d and b the values of `num`, `den` and `radix`, q_i the ith element of the stream, and n_i the value of `num` during the ith recursive call to `expand` (with $n = n_0$), we can prove by induction that for any $k \geq 0$:

$$n = \sum_{i=0}^{k} \frac{q_i d}{b^{i+1}} + \frac{n_{k+1}}{b^{k+1}}$$

so by taking the limit when k tends to $+\infty$:

$$\frac{n}{d} = \sum_{i \geq 0} \frac{q_i}{b^{i+1}}$$

The streams `(expand 1 7 10)` and `(expand 3 8 10)` give the decimal expansions of $1/7$ and $3/8$ respectively: in the first case the elements are 1, 4, 2, 8, 5 and 7 repeating indefinitely, in the second case the elements are 3, 7 and 5 followed by zeroes.

Exercise 3.59

a. The procedure `integrate-series` can be written:

```
(define (integrate-series s)
  (stream-map / s integers))
```

b. The series for sine and cosine can be defined with:

```
(define cosine-series
  (cons-stream 1 (scale-stream (integrate-series sine-series) -1)))

(define sine-series
  (cons-stream 0 (integrate-series cosine-series)))
```

Exercise 3.60

If we write the two power series to multiply as $S_1 = a_0 + xA_1$ and $S_2 = b_0 + xB_1$, where A_1 and B_1 are the power series represented by `(stream-cdr s1)` and `(stream-cdr s2)`, we have $S_1 S_2 = a_0 b_0 + x(b_0 A_1 + a_0 B_1 + x A_1 B_1) = a_0 b_0 + x(a_0 B_1 + A_1 S_2)$, so `mul-series` can be defined as:

```
(define (mul-series s1 s2)
  (cons-stream (* (stream-car s1) (stream-car s2))
               (add-streams (scale-stream (stream-cdr s2) (stream-car s1))
                            (mul-series (stream-cdr s1) s2))))
```

Exercise 3.61

The formula given in the text leads to the definition of `invert-unit-series` in the following way:

```
(define (invert-unit-series s)
  (define inverse (cons-stream 1
                               (stream-map - (mul-series inverse
                                                         (stream-cdr s)))))
  inverse)
```

I defined a local variable rather than calling recursively `invert-unit-series`, otherwise the computation does not benefit from the optimization provided by `memo-proc` and is a lot slower.

Exercise 3.62

If F and G are two power series with $G = c_0 + \sum_{i>0} c_i x^i$ and $c_0 \neq 0$, we can write $G = c_0 H$ where $H = \frac{1}{c_0} G$ is a power series with a constant term equal to 1. We have:

$$\frac{F}{G} = \frac{F}{c_0 H} = \frac{1}{c_0} \times F \times \frac{1}{H}$$

Since `invert-unit-series` can be used to compute the power series for $\frac{1}{H}$, `div-series` can be defined as shown belown, and the power series for tangent can be defined using $\tan x = \frac{\sin x}{\cos x}$.

```
(define (div-series num denom)
  (if (= 0 (stream-car denom))
    (error "Denominator has a zero constant term -- DIV-SERIES.")
    (let ((c (/ 1 (stream-car denom))))
      (scale-stream (mul-series num (invert-unit-series (scale-stream denom c)))
                    c))))

(define tan-series (div-series sine-series cosine-series))
```

3.5.3 Exploiting the Stream Paradigm

Formulating iterations as stream processes

Exercise 3.63

If we don't use a local variable, each recursive call to `sqrt-stream` produces a new stream, so if the n first elements of the result have been computed, to get the next element, the n first elements are computed again before the final mapping can be applied, so the time needed to compute the $(n+1)$-th element is at least twice the time needed to compute the nth element, the time complexity is in $\Theta(2^n)$.

With a local variable and `memo-proc`, the complexity is linear, but without `memo-proc` the complexity would still be exponential since all the previous elements of the stream must be recomputed to obtain the next element.

Exercise 3.64

The procedure `stream-limit` can be defined as shown below, assuming the stream is infinite or at least is guaranteed not to end before the number is found:

```
(define (stream-limit s tolerance)
  (let ((s0 (stream-ref s 0))
        (s1 (stream-ref s 1)))
    (if (< (abs (- s1 s0)) tolerance)
      s1
      (stream-limit (stream-cdr s) tolerance))))
```

Exercise 3.65

Three sequences analogous to those defined for π can be defined with:

```
(define (ln2-summands n)
  (cons-stream (/ 1.0 n)
               (stream-map - (ln2-summands (+ n 1)))))

(define ln2-stream (partial-sums (ln2-summands 1)))

(define ln2-stream-acc (euler-transform ln2-stream))

(define ln2-stream-acc2 (accelerated-sequence euler-transform ln2-stream))
```

As was the case for the approximations of π, the first sequence converges slowly: with 10 terms, the value of $\ln 2$ is bound between .63 and .75. The second sequence converges already more quickly: the first 10 terms give a value between .6930 and .6933. The third sequence converges even more quickly: the first 10 terms give a correct value to 14 decimal places.

Infinite streams of pairs

Exercise 3.66

Intuitively, since `interleave` produces a stream where half of the elements come from the first stream and half from the second stream, about one out of two elements of `int-pairs` comes from the first line, and more generally, the interval between two pairs starting with i is about 2^i.

To give more precise results, let's call S_i the stream produced by `pairs` with integers greater than or equal to i, so that `int-pairs` is equal to S_1. Because of the order of the arguments of `interleave` in `pairs`, the elements at index 0 and at odd indices of S_i start with i. The pairs with even indices ≥ 2 are elements of S_{i+1}.

Using this, we can show that the index of (i, i) in S_1 is equal to $2^i - 2$, and the index of (i, j) with $j > i$ is $(j - i)2^i + 2^{i-1} - 2$.

Proof. This is true for $i = 1$.

Let's assume that it's true for some $i \geq 1$. We can prove the formulas for $i + 1$ by using the two following facts: 1) the index of $(i + 1, j)$ in S_2 is the same as the index of $(i, j - 1)$ in S_1 since these streams are constructed in the same way, 2) the kth element of S_2 has index $2k + 2$ in S_1.

From the induction hypothesis, the index of $(i+1, i+1)$ in S_2 is $2^i - 2$. Since the kth element of S_2 has index $2k + 2$ in S_1, the index of $(i + 1, i + 1)$ is S_1 is $2(2^i - 2) + 2 = 2^{i+1} - 2$.

In the same way, the index of $(i + 1, j)$ with $j > i + 1$ in S_2 is equal to the index of $(i, j - 1)$ in S_1, that is, to $(j - i - 1)2^i + 2^{i-1} - 2$. So the index of $(i + 1, j)$ in S_1 is equal to $2((j - i - 1)2^i + 2^{i-1} - 2) + 2 = (j - i - 1)2^{i+1} + 2^i - 2$, which is the result wanted. □

By applying this result, we find that the pair $(1, 100)$ has index 197, so it is preceded by 197 pairs. The pair $(99, 100)$ has index $2^{99} + 2^{98} - 2$, and the pair $(100, 100)$ has index $2^{100} - 2$.

Exercise 3.67

We can divide the stream `(all-pairs S T)` containing all pairs of elements from the two streams into four parts: the first pair (S_0, T_0), the rest of the first line, the rest of the first column, and the stream `(all-pairs (stream-cdr S) (stream-cdr T))`. We only need to add the stream corresponding to the rest of the first column to the previous definition.

```
(define (all-pairs s t)
  (cons-stream
    (list (stream-car s) (stream-car t))
    (interleave
      (interleave
        (stream-map (lambda (x) (list (stream-car s) x))
                    (stream-cdr t))
        (stream-map (lambda (x) (list x (stream-car s)))
                    (stream-cdr t)))
      (all-pairs (stream-cdr s) (stream-cdr t)))))

(define all-int-pairs (all-pairs integers integers))
```

Exercise 3.68

Louis' definition of `pairs` causes an infinite loop when it is called with infinite streams because `interleave` is called directly, it is not put in an argument to `cons-stream`, which uses delayed evaluation. So the arguments of `interleave` are evaluated, which causes infinite recursive calls to `pairs`.

Exercise 3.69

The stream of triples can be generated similarly to the stream of pairs as shown below. Once `triples` is defined, applying a filter is enough to define a stream of all Pythagorean triples.

```
(define (triples s t u)
  (cons-stream
    (list (stream-car s) (stream-car t) (stream-car u))
```

```scheme
    (interleave (stream-map (lambda (pair) (cons (stream-car s) pair))
                            (stream-cdr (pairs t u)))
                (triples (stream-cdr s) (stream-cdr t) (stream-cdr u)))))

(define triples-stream (triples integers integers integers))

(define pythagoreans-stream
  (stream-filter (lambda (triple)
                   (= (+ (square (car triple)) (square (cadr triple)))
                      (square (caddr triple))))
                 triples-stream))
```

Exercise 3.70

The `merge-weighted` procedure is very similar to `merge`, except that if multiple elements have the same weight, they are all kept, whereas `merge` removed duplicates.

```scheme
(define (merge-weighted s1 s2 weight)
  (cond ((stream-null? s1) s2)
        ((stream-null? s2) s1)
        (else
          (let ((s1car (stream-car s1))
                (s2car (stream-car s2)))
            (if (<= (weight s1car) (weight s2car))
              (cons-stream s1car
                           (merge-weighted (stream-cdr s1) s2 weight))
              (cons-stream s2car
                           (merge-weighted s1 (stream-cdr s2) weight)))))))

(define (weighted-pairs s t weight)
  (cons-stream
    (list (stream-car s) (stream-car t))
    (merge-weighted
      (stream-map (lambda (x) (list (stream-car s) x))
                  (stream-cdr t))
      (weighted-pairs (stream-cdr s) (stream-cdr t) weight)
      weight)))
```

a. The ordered pairs can be defined as follows:

```scheme
(define pairs-sum-order
  (weighted-pairs integers
                  integers
                  (lambda (pair)
                    (+ (car pair) (cadr pair)))))
```

b. The required stream can be defined as:

```scheme
(define int-stream (stream-filter
                     (lambda (x)
                       (not (or (divides? 2 x)
                                (divides? 3 x)
                                (divides? 5 x))))
                     integers))

(define stream-235
  (weighted-pairs int-stream
                  int-stream
                  (lambda (pair)
                    (+ (* 2 (car pair))
                       (* 3 (cadr pair))
                       (* 5 (car pair) (cadr pair))))))
```

Exercise 3.71

I used a procedure `group-by` that takes as arguments an infinite stream and a test procedure `same-group?` that tests whether two successive elements of the stream belong to the same group, and returns a stream where the elements are grouped into lists according to the given procedure. To find all Ramanujan number, we first generate the stream of pairs of integers ordered by the sum of their cubes, then group the pairs with the same cube sum, and keep only the groups with more than one element.

```scheme
(define (group-by s same-group?)
  (define (helper s group)
    (if (same-group? (stream-car s) (car group))
      (helper (stream-cdr s)
              (cons (stream-car s) group))
      (cons-stream (reverse group)
                   (group-by s same-group?))))
  (helper (stream-cdr s) (list (stream-car s))))

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

(define (cube-sum pair)
  (+ (cube (car pair))
     (cube (cadr pair))))

(define pairs-by-cube-sum (weighted-pairs integers
                                          integers
                                          cube-sum))

(define ramanujan
  (stream-map (lambda (decompositions-list)
                (cons (cube-sum (car decompositions-list))
```

```
                      decompositions-list))
              (stream-filter (lambda (list)
                               (> (length list) 1))
                             (group-by pairs-by-cube-sum
                                       (lambda (p1 p2)
                                         (= (cube-sum p1) (cube-sum p2)))))))
```

The 6 first Ramanujan numbers are 1729, 4104, 13832, 20683, 32832 and 39312. The procedure given above returns them with their decompositions:

```
> (print-n ramanujan 6)
(1729 (1 12) (9 10))
(4104 (2 16) (9 15))
(13832 (2 24) (18 20))
(20683 (10 27) (19 24))
(32832 (4 32) (18 30))
(39312 (2 34) (15 33))
```

Exercise 3.72

Thanks to the definition of `group-by` in the previous exercise, the stream can be defined by applying a few stream operations in a way very similar to the definition of the stream of Ramanujan numbers.

```
(define (square-sum pair)
    (+ (square (car pair)) (square (cadr pair))))

(define three-square-sums
  (stream-map (lambda (dec-list)
                (cons (square-sum (car dec-list)) dec-list))
              (stream-filter
                (lambda (group)
                  (>= (length group) 3))
                (group-by (weighted-pairs integers
                                          integers
                                          square-sum)
                          (lambda (p1 p2)
                            (= (square-sum p1) (square-sum p2)))))))
```

The first 10 such numbers found are:

```
> (print-n three-square-sums 10)
(325 (1 18) (6 17) (10 15))
(425 (5 20) (8 19) (13 16))
(650 (5 25) (11 23) (17 19))
(725 (7 26) (10 25) (14 23))
(845 (2 29) (13 26) (19 22))
```

```
(850 (3 29) (11 27) (15 25))
(925 (5 30) (14 27) (21 22))
(1025 (1 32) (8 31) (20 25))
(1105 (4 33) (9 32) (12 31) (23 24))
(1250 (5 35) (17 31) (25 25))
```

Streams as signals

Exercise 3.73

The RC circuit can be modeled with the following procedure:

```
(define (RC R C dt)
  (lambda (i v0)
    (add-streams (scale-stream i R)
                 (integral (scale-stream i (/ 1 C)) v0 dt))))
```

Exercise 3.74

Complement:
To test the code from the book, let's first define a procedure to transform a list into a stream, as well as the `sign-change-detector` procedure from the book. Note that because of the way `make-zero-crossings` is defined, `sign-change-detector` works backwards: it returns 1 when its first argument is positive and the second is negative and vice-versa.

```
(define (sign-change-detector v1 v2)
  (cond ((and (< v1 0) (>= v2 0)) -1)
        ((and (>= v1 0) (< v2 0)) 1)
        (else 0)))

(define (list->stream elts)
  (fold-right (lambda (a b) (cons-stream a b)) '() elts))

(define sense-data
  (list->stream '(1 2 1.5 1 0.5 -0.1 -2 -3 -2 -0.5 0.2 3 4)))
```

With the suggestion from the book, the stream of zero crossing can be defined as:

```
(define zero-crossings
  (stream-map sign-change-detector
              sense-data
              (cons-stream 0 sense-data)))
```

However, it would be better to avoid adding an arbitrary initial value of 0 by defining `zero-crossings` instead in the following way:

```
(define zero-crossings
  (stream-map sign-change-detector
              (stream-cdr sense-data)
              sense-data))
```

Exercise 3.75

The "last-value" to use to compute the average should be the last value from the `sense-data` stream, and the value to use to detect the sign change should be the last average. Louis uses the last average for both. The solution is to pass both the last value and the last average as arguments to `zero-crossings`.

```
(define (make-zero-crossings input-stream last-value last-avg)
  (let ((avpt (/ (+ (stream-car input-stream) last-value) 2)))
    (cons-stream (sign-change-detector avpt last-avg)
                 (make-zero-crossings (stream-cdr input-stream)
                                      (stream-car input-stream)
                                      avpt))))
```

Exercise 3.76

The `smooth` procedure can be written as shown below, which allows us to redefine `make-zero-crossings` so that it takes it as an argument:

```
(define (smooth s)
  (scale-stream (add-streams s (stream-cdr s)) .5))

(define (make-zero-crossings input-stream smooth-func)
  (let ((smoothed (smooth-func input-stream)))
    (stream-map sign-change-detector
                (stream-cdr smoothed)
                smoothed)))

(define zero-crossings (make-zero-crossings sense-data smooth))
```

3.5.4 Streams and Delayed Evaluation

Complement:
The `solve` procedure does not work with Gambit Scheme, I used the variation:

```
(define (solve f y0 dt)
  (let ((y 'undefined)
        (dy 'undefined))
    (set! y (integral (delay dy) y0 dt))
    (set! dy (stream-map f y))
    y))
```

Exercise 3.77

The `integral` procedure given in the exercise with a delayed `integrand` argument becomes:

```
(define (integral delayed-integrand initial-value dt)
  (cons-stream initial-value
               (let ((integrand (force delayed-integrand)))
                 (if (stream-null? integrand)
                   the-empty-stream
```

```
                    (integral (delay (stream-cdr integrand))
                              (+ (* dt (stream-car integrand))
                                 initial-value)
                              dt)))))
```

Exercise 3.78

The `solve-2nd` procedure can be defined in the following way, using `let` and `set!` instead of `define` so it works with Gambit Scheme:

```
(define (solve-2nd a b y0 dy0 dt)
  (let ((y 'undefined)
        (dy 'undefined)
        (ddy 'undefined))
    (set! y (integral (delay dy) y0 dt))
    (set! dy (integral (delay ddy) dy0 dt))
    (set! ddy (add-streams (scale-stream dy a)
                           (scale-stream y b)))
    y))
```

Exercise 3.79

The generalization of `solve-2nd` is:

```
(define (solve-2nd f y0 dy0 dt)
  (let ((y 'undefined)
        (dy 'undefined)
        (ddy 'undefined))
    (set! y (integral (delay dy) y0 dt))
    (set! dy (integral (delay ddy) dy0 dt))
    (set! ddy (stream-map f dy y))
    y))
```

Exercise 3.80

The procedure to model the circuit and the example of streams can be defined as:

```
(define (RLC R L C dt)
  (lambda (vC0 iL0)
    (define vC (integral (delay (scale-stream iL (- (/ 1 C)))) iL0 dt))
    (define iL (integral (delay (add-streams (scale-stream vC (/ 1 L))
                                             (scale-stream iL (- (/ R L)))))
                         vC0
                         dt))
    (cons vC iL)))

(define RLC1 ((RLC 1 1 .2 .1) 10 0))
(define vC (car RLC1))
(define iL (cdr RLC1))
```

3.5.5 Modularity of Functional Programs and Modularity of Objects

Exercise 3.81

We assume that the elements of the stream of requests are either the string `generate` or a pair `(reset . new-value)` with the new value for the seed of the random numbers. The generator can thus be written as:

```
(define (rand requests random-init)
  (define randoms
    (cons-stream random-init
                 (stream-map
                   (lambda (request n)
                     (cond ((eq? 'generate request)
                            (rand-update n))
                           ((and (pair? request)
                                 (eq? 'reset (car request)))
                            (cdr request))
                           (else
                             (error "Unknown request -- RAND" m))))
                   requests
                   randoms)))
  randoms)

; For testing
(define requests
  (list->stream (list 'generate 'generate 'generate (cons 'reset 31) 'generate
                      'generate (cons 'reset 42) 'generate 'generate 'generate)))
```

Exercise 3.82

We can rewrite Monte Carlo integration in terms of streams as shown below.

```
(define (randoms-in-range low high)
  (let ((range (- high low)))
    (cons-stream (+ low (* range (random-real)))
                 (randoms-in-range low high))))

(define (estimate-integral P x1 x2 y1 y2)
  (define experiment-stream
    (stream-map P
                (randoms-in-range x1 x2)
                (randoms-in-range y1 y2)))
  (let ((area (* (- x2 x1) (- y2 y1))))
    (scale-stream (monte-carlo experiment-stream 0 0)
                  area)))

; Estimation of Pi by estimating the area of a unit circle.
```

```
(define estimate-pi
  (estimate-integral (lambda (x y)
                       (<= (+ (square x) (square y)) 1))
                     -1. 1. -1. 1.))
```

4 Metalinguistic Abstraction

4.1 The Metacircular Evaluator

4.1.1 The Core of the Evaluator

Exercise 4.1

We can put the value to evaluate first inside a `let` expression. Since the `let` is transformed into a `lambda` internally, we know that its argument will be evaluated before the body of the `let`.

```
(define (list-of-values-ltr exps env)
  (if (no-operands? exps)
    '()
    (let ((first-value (eval (first-operand exps) env)))
      (cons first-value
            (list-of-values-ltr (rest-operands exps) env)))))

(define (list-of-values-rtl exps env)
  (if (no-operands? exps)
    '()
    (let ((rest-values (list-of-values-rtl (rest-operands exps) env)))
      (cons first-value rest-values))))
```

4.1.2 Representing Expressions

Exercise 4.2

a. The test to determine whether an expression is a procedure just checks whether it is a pair, so it will return true for all list expressions: `if`, `cond`, `begin`, `define`, `set!`, etc., and the evaluator will try to apply the procedure `if`, `cond`, etc. to the rest of the arguments. It's not possible to transform the special forms into procedures because all the arguments of a procedure are evaluated before evaluation.

b. The only required change to `eval` is to put the `application?` test first. Then we only need changing the definition of the `application?` predicate and the associated selectors so they reflect the new syntax.

```
(define (application? exp) (tagged-list? exp 'call))

(define (operator exp) (cadr exp))

(define (operands exp) (cddr exp))
```

Exercise 4.3

As in exercise 2.73, a few cases can't be assimilated to the data-directed dispatch because they correspond to untagged data: here it's the case of `variable?`, `self-evaluating?` and `application?`. If we had kept Louis' idea from the previous exercise to start procedure applications with `call`, `application?` could have been assimilated into the dispatch. Since we don't keep this idea, we have to redefine the `application?` predicate: if we keep it as `pair?` we can't put it before the dispatch because all the special forms will be considered as procedure applications, and we can't put it after the dispatch either because the procedure won't be found in the table and it will cause an error. So I consider that every expression for which no procedure of the correct type is found in the table is a procedure application. I used a two-dimensional table as in chapter 2, though a one-dimensional table would be enough if we don't do any other dispatch on the expressions' type.

```
(define (type exp) (car exp))

(define (application? exp) (not (get 'eval (type exp))))

(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
          ((get 'eval (type exp)) exp env))))

(put 'eval 'quote
     (lambda (exp env) (text-of-quotation exp)))
(put 'eval 'set! eval-assignment)
(put 'eval 'define eval-definition)
(put 'eval 'if eval-if)
(put 'eval 'lambda
     (lambda (exp env)
       (make-procedure (lambda-parameters exp)
                       (lambda-body exp)
                       env)))
(put 'eval 'begin
     (lambda (exp env)
       (eval-sequence (begin-actions exp) env)))
(put 'eval 'cond
     (lambda (exp env) (eval (cond->if exp) env)))
```

Exercise 4.4

If we define specific evaluation procedures `eval-and` and `eval-or`, we have to add the following two lines to the definition of `eval`

```
((or? exp) (eval-or exp env))
((and? exp) (eval-and exp env))
```

Then we can implement the predicates, selectors and the evaluation procedures in the following way:

```
; And
(define (and? exp)
  (tagged-list? exp 'and))

(define (and-tests exp)
  (cdr exp))

(define (eval-and exp env)
  (define (eval-exps exps)
    (let ((first (eval (car exps) env)))
      (if (true? first)
        (if (last-exp? exps)
          first
          (eval-exps (cdr exps)))
        false)))
  (let ((exps (and-tests exp)))
    (if (null? exps)
      true
      (eval-exps exps))))

; Or
(define (or? exp)
  (tagged-list? exp 'or))

(define (or-tests exp)
  (cdr exp))

(define (eval-or exp env)
  (define (eval-exps exps)
    (if (null? exps)
      false
      (let ((first (eval (car exps) env)))
        (if (true? first)
          first
          (eval-exps (cdr exps))))))
  (let ((exps (or-tests exp)))
    (eval-exps exps)))
```

If `and` and or are implemented as derived expressions, the predicates and selectors don't change, but the lines regarding `and` and `or` in `eval` are replaced by:

```
((or? exp) (eval (or->if exp) env))
((and? exp) (eval (and->if exp) env))
```

And the procedures transforming `and` and `or` expressions into `if` expressions can be defined as:

```
(define (and->if exp)
  (expand-and (and-tests exp)))

(define (or->if exp)
  (expand-or (or-tests exp)))

(define (expand-or exps)
  (if (null? exps)
    'false
    (let ((first (car exps))
          (rest (cdr exps)))
      (make-if first
               first
               (expand-or rest)))))

(define (expand-and exps)
  (cond ((null? exps) 'true)
        ((last-exp? exps) (car exps))
        (else
          (let ((first (car exps))
                (rest (cdr exps)))
            (make-if first
                     (expand-and rest)
                     'false)))))
```

Exercise 4.5

We only need to add the appropriate predicate and selectors for this type of clause so that the appropriate action is taken if the predicate evaluates to a true value.

```
(define (cond-=>-clause? clause)
  (eq? (cadr clause) '=>))

(define (cond-recipient clause)
  (cddr clause))

(define (cond-actions clause)
  (if (cond-=>-clause? clause)
    (list (cond-recipient clause)
          (cond-predicate clause)))
  (cdr clause))
```

Exercise 4.6

The only necessary modification to `eval` is to add the line:

```
((let? exp) (eval (let->combination exp) env))
```

Then we can define the appropriate predicate and selectors and the transformation procedure `let->combination` in the following way:

```
(define (let? exp)
  (tagged-list? exp 'let))

(define (let-bindings exp) (cadr exp))
(define (let-vars exp) (map car (let-bindings exp)))
(define (let-args exp) (map cadr (let-bindings exp)))
(define (let-body exp) (cddr exp))

(define (let->combination exp)
  (let ((bindings (let-bindings exp)))
    (cons (make-lambda (let-vars exp)
                       (let-body exp))
          (let-args exp))))
```

Exercise 4.7

A `let*` expression can be rewritten as a set of nested `let` expressions defining only one binding at a time, so that the second binding is defined in the body of the first `let` etc. For instance, the example from the book becomes:

```
(let ((x 3))
  (let ((y (+ x 2)))
    (let ((z (+ x y 5)))
      (* x z))))
```

If we have already implemented `let`, it is sufficient to rewrite `let*` expressions in terms of `let` expressions to handle them. This can be done in the following way:

```
(define (let*? exp)
  (tagged-list? exp 'let*))

(define (let*-bindings exp) (cadr exp))
(define (let*-body exp) (cddr exp))

(define (make-let bindings body)
  (list 'let bindings body))

(define (let*->nested-lets exp)
  (define (make-nested-lets bindings body)
    (cond ((null? bindings)
```

```
            (sequence->exp body))
           (else
             (make-let (list (car bindings))
                       (make-nested-lets (cdr bindings) body)))))
  (make-nested-lets (let*-bindings exp) (let*-body exp)))
```

Exercise 4.8

To support named `let`, we modify the selectors for `let` so they handle both named `lets` and ordinary `lets`, and `let->combination` so the generated code creates a procedure with the given name instead of a lambda.

```
(define (named-let? exp) (variable? (cadr exp)))

(define (let-name exp) (cadr exp))

(define (let-bindings exp)
  (if (named-let? exp)
    (caddr exp)
    (cadr exp)))

(define (let-body exp)
  (if (named-let? exp)
    (cdddr exp)
    (cddr exp)))

(define (make-define name value)
  (list 'define name value))

(define (let->combination exp)
  (let ((bindings (let-bindings exp)))
    (if (named-let? exp)
      (make-begin
        (list (make-define (cons (let-name exp) (let-vars exp))
                           (sequence->exp (let-body exp)))
              (cons (let-name exp) (let-args exp))))
      (cons (make-lambda (let-vars exp)
                         (let-body exp))
            (let-args exp)))))
```

Exercise 4.9

I chose to implement three iterative control structures: a `while` loop, an `until` loop, and a `for` loop. The return value of all the expressions described is `false`, so they are useful only for their side effects. The `until` and the `for` constructs are derived from `while`.

while The syntax of the `while` loop is `(while <predicate> <body>)`. The predicate must consist of a single expression. The body can consist of several expressions. The body is executed while the predicate is true. If the predicate is false from the beginning, the body is never evaluated.

For instance, the evaluation of the following code prints the numbers from 0 to 5 inclusive, each on a new line:

```
(define x 0)
(while (<= x 5)
    (display x)
    (newline)
    (set! x (+ x 1)))
```

The transformation procedure for `while` expressions turns them into the definition of a recursive procedure that uses the predicate to decide whether to execude the body of the `while` or not, followed by a call to this procedure. The procedure name is generated by `gensym`, so it does not conflict with other names in the program, and if two `while` are evaluated in the same environment, the generated procedures will have different names.

```
(define (while? exp)
  (tagged-list? exp 'while))

(define (while-predicate exp)
  (cadr exp))

(define (while-body exp)
  (cddr exp))

(define (make-while test body)
  (list 'while test body))

(define (make-not exp)
  (list 'not exp))

(define (while->if exp)
  (let ((proc-name (gensym)))
    (sequence->exp
      (list
        (make-define
          (list proc-name)
          (make-if (while-predicate exp)
                   (sequence->exp
                     (append
                       (while-body exp)
                       (list (list proc-name))))
                   'false))
        (list proc-name)))))
```

until The syntax of the `until` loop is `(until <predicate> <body>)`, and such an expression is equivalent to `(while (not <predicate>) <body>)`. For instance, the following code prints the numbers from 0 to 4 inclusive, each on a new line:

```
(define x 0)
(until (= x 5)
    (display x)
    (newline)
    (set! x (+ x 1)))
```

`Until` expressions are easily derived from `while` expressions in the following way:

```
(define (until? exp)
  (tagged-list? exp 'until))

(define (until-predicate exp)
  (cadr exp))

(define (until-body exp)
  (cddr exp))

(define (until->while exp)
  (make-while (make-not (until-predicate exp))
              (sequence->exp (until-body exp))))
```

for The general syntax of the `for` loop is `(for (<var> <start> <end-test> [<inc-exp>]) <body>)`. The body can consist of several expressions. The other parts are:

- `<var>`: the name of the variable whose value is bound by the `for` loop.
- `<start>`: an expression giving the initial value of the variable.
- `<end-test>` can be either:
  - an expression where `<var>` appears as a free variable, for instance `(< <var> 3)`. The body of the `for` is evaluated as long as the evaluation of this expression with the current value of `<var>` returns `true`.
  - an expression not depending on the variable bound by the `for`, whose evaluation must return a number. This is equivalent to either `(<= <var> <end-test>)` or `(>= <var> <end-test>)`, depending on whether the returned value is greater or smaller than the initial value.
- `<inc-exp>` can be either:
  - an expression where `<var>` appears as a free variable, for instance `(+ <var> 2)`. It is evaluated to update the value of `<var>` before evaluating (or not) the body of the `for` again.
  - an expression not depending on the variable bound by the `for`, whose evaluation must return a number. This is equivalent to specifying `(+ <var> <inc-exp>)`.

  * empty, in which case it will be assumed to be 1.

For instance, the following expressions all print the numbers from 0 to 5:

```
(for (x 1 (<= x 5) (+ x 1)) (display x) (newline))

(for (x 1 (<= x 5) 1) (display x) (newline))

(for (x 1 (<= x 5)) (display x) (newline))

(for (x 1 5 (+ x 1)) (display x) (newline))

(for (x 1 5 1) (display x) (newline))

(for (x 1 5) (display x) (newline))
```

Using a step other than one, we can print 1, .8, .6, .4, .2 and 0 with:

```
(for (x 1 -.1 -.2)
    (display x) (newline))
```

With an update function that is not simply an addition or subtraction, the following prints 16, 8, 4, 2 and 1.

```
(for (x 16 1 (/ x 2))
    (display x) (newline))
```

It's possible to use nested `fors`, for instance:

```
(for (x 0 5)
    (for (y (+ x 1) (+ x 2))
        (display x)
        (display ", ")
        (display y)
        (newline)))
```

The output is:

```
0, 1
0, 2
1, 2
1, 3
2, 3
2, 4
3, 4
3, 5
4, 5
4, 6
5, 6
5, 7
```

The implementation is more complex than that of `while`, because several cases have to be considered. I had to write a procedure that checks whether a given variable appears as a free variable in a given expression. Then the different cases have to be considered when constructing the predicate that tests whether to interrupt the loop or not, and the procedure that updates the value of the variable. The `for` expression is then transformed in a `let` expression containing a `while`.

```
(define (for? exp)
  (tagged-list? exp 'for))

(define (for-body exp)
  (cddr exp))

(define (for-range-def exp)
  (cadr exp))

(define (for-variable exp)
  (car (for-range-def exp)))

(define (for-in-type? exp)
  (eq? (cadr (for-range-def exp)) 'in))

(define (for-start exp)
  (cadr (for-range-def exp)))

(define (make-set! name value)
  (list 'set! name value))

; Returns true if var appears as a free variable in the given expression.
(define (has-free-var? exp var)
  (cond ((variable? exp)
         (eq? exp var))
        ((lambda? exp)
         (and (not (memq var (lambda-parameters exp)))
              (has-free-var? (lambda-body exp) var)))
        ((definition? exp)
         (let ((def-var (definition-variable exp)))
           (and (not (if (variable? def-var)
                         (eq? var def-var)
                         (memq var def-var)))
                (has-free-var? (definition-value exp) var))))
        ((pair? exp)
         (or (has-free-var? (car exp) var)
             (has-free-var? (cdr exp) var)))
```

```
        (else false)))

; Returns a procedure taking as a parameter the variable of the for loop.
; The for loop is executed as long as this procedure returns true.
(define (for-end-test exp)
  (let ((var (for-variable exp))
        (end-exp (caddr (for-range-def exp))))
    (make-lambda
      (list var)
      (if (has-free-var? end-exp var)
        (list end-exp)
        (list
          (list (make-if (list '<= (for-start exp) end-exp) '<= '>=)
                var
                end-exp))))))

; Returns a procedure computing the new value of the variable from its current
; value.
(define (for-inc exp)
  (let* ((var (for-variable exp))
         (range-def (for-range-def exp))
         (inc (if (null? (cdddr range-def)) 1 (cadddr range-def))))
    (make-lambda (list var)
                 (if (has-free-var? inc var)
                   (list inc)
                   (list (list '+ var inc))))))

; Transforms a for expression into a let expression containing a while loop.
(define (for->let exp)
  (let ((var (for-variable exp)))
    (make-let (list (list var (for-start exp)))
              (make-while
                (list (for-end-test exp) var)
                (sequence->exp
                  (append
                    (for-body exp)
                    (list
                      (make-set! var
                                 (list (for-inc exp) var))))))))))
```

Exercise 4.10

It's easy to modify the symbol used as the car of an expression to define its type: we only need changing the predicate (and the associated constructor if any was defined) for this type of expression. For instance, we can replace `cond` with `case`, `and` with `&&`, and `or` with `||` with the following code:

```
(define (and? exp)
  (tagged-list? exp '&&))

(define (or? exp)
  (tagged-list? exp '||))

(define (cond? exp)
  (tagged-list? exp 'case))
```

We can replace `not` with `!` by replacing `(list 'not not)` with `(list '! not)` in the list of primitive procedures and redefining the `make-not` procedure defined in exercise 4.9:

```
(define (make-not exp)
  (list '! exp))
```

I also modified the selectors for procedure application so that `(proc args)` becomes `(proc (args))`. This change actually makes the syntax more complex since e.g. `(not (> x 3))` has to be written `(! ((> (x 3))))`, it's just for the sake of the example.

```
(define (operands exp) (cadr exp))
```

I think that more dramatic changes such as replacing the parentheses with brackets would require a lot of work since currently we rely on the underlying Scheme for list processing.

4.1.3 Evaluator Data Structures

Exercise 4.11

If each binding is represented as a name-value pair, each frame represents a table, so we can simplify the implementation by using `assoc`. We can change the implementation by rewriting the following procedures:

```
(define (make-frame variables values)
  (if (null? variables)
    (list '*frame*)
    (add-binding-to-frame! (car variables) (car values)
                           (make-frame (cdr variables) (cdr values)))))

(define (frame-bindings frame) (cdr frame))

(define (add-binding-to-frame! var val frame)
  (set-cdr! frame (cons (cons var val) (cdr frame)))
  frame)

(define (lookup-variable-value var env)
  (define (env-loop env)
```

```
    (define (scan bindings)
      (let ((binding (assoc var bindings)))
        (if binding
          (cdr binding)
          (env-loop (enclosing-environment env)))))
    (if (eq? env the-empty-environment)
      (error "Unbound variable" var)
      (let ((frame (first-frame env)))
        (scan (frame-bindings frame)))))
  (env-loop env))

(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan bindings)
      (let ((binding (assoc var bindings)))
        (if binding
          (set-cdr! binding val)
          (env-loop (enclosing-environment env)))))
    (if (eq? env the-empty-environment)
      (error "Unbound variable -- SET!" var)
      (let ((frame (first-frame env)))
        (scan (frame-bindings frame)))))
  (env-loop env))

(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan bindings)
      (let ((binding (assoc var bindings)))
        (if binding
          (set-cdr! binding val)
          (add-binding-to-frame! var val frame))))
    (scan (frame-bindings frame))))
```

Exercise 4.12

We can define a procedure that recursively traverses the environment structure and executes the given action when a binding is found for the given variable as shown below. By default, the procedure goes to the enclosing environment if the procedure is not found in the first frame, but a second procedure can be passed as an argument to specify another behavior.

Implementation for the representation of frames as a pair of lists:

```
(define (traverse-env env var var-found-action . null-action)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (if (null? null-action)
```

```
              (env-loop (enclosing-environment env))
              ((car null-action) env)))
           ((eq? var (car vars))
            (var-found-action vals))
           (else (scan (cdr vars)
                       (cdr vals)))))
    (if (eq? env the-empty-environment)
      (error "Unbound variable" var)
      (let ((frame (first-frame env)))
        (scan (frame-variables frame)
              (frame-values frame)))))
  (env-loop env))

(define (set-vals-car! val)
  (lambda (vals)
    (set-car! vals val)))

(define (lookup-variable-value var env)
  (traverse-env env var car))

(define (set-variable-value! var val env)
  (traverse-env env var (set-vals-car! val)))

(define (define-variable! var val env)
  (traverse-env env var (set-vals-car! val)
                (lambda (env)
                  (add-binding-to-frame! var val (first-frame env)))))
```

Implementation for the reperesentation of frames as a list of pairs:

```
(define (traverse-env env var var-found-action . null-action)
  (define (env-loop env)
    (define (scan bindings)
      (let ((binding (assoc var bindings)))
        (if binding
          (var-found-action binding)
          (if (null? null-action)
            (env-loop (enclosing-environment env))
            ((car null-action) env)))))
    (if (eq? env the-empty-environment)
      (error "Unbound variable" var)
      (let ((frame (first-frame env)))
        (scan (frame-bindings frame)))))
  (env-loop env))
```

```
(define (set-binding-value! val)
  (lambda (binding)
    (set-cdr! binding val)))

(define (lookup-variable-value var env)
  (traverse-env env var cdr))

(define (set-variable-value! var val env)
  (traverse-env env var (set-binding-value! val)))

(define (define-variable! var val env)
  (traverse-env env var (set-binding-value! val)
                (lambda (env)
                  (add-binding-to-frame! var val (first-frame env)))))
```

Exercise 4.13

The solutions I read on the internet unbind variables only from the first frame of the environment because modifying the enclosing environment seemed too risky or similar reasons. I chose to delete the first binding found even if it's not in the first frame: the interpreter implementation already allows us to change bindings in the enclosing environments, e.g. if I define a function `(define (f x) (set! + -) x)`, the sequence of interactions:

```
(+ 2 1)
(f 2)
(+ 2 1)
```

produces 3, 2 and 1. This choice is thus coherent with the rest of the implementation.

To add the `make-unbound!` operation, we add to `eval` the line:

```
((unbind? exp) (eval-unbind exp env))
```

The necessary procedures with the implementation of frames as a pair of lists are:

```
(define (unbind? exp)
  (tagged-list? exp 'make-unbound!))

(define (unbind-variable exp)
  (cadr exp))

(define (eval-unbind exp env)
  (unbind-var (unbind-variable exp) env))

(define (unbind-var var env)
  (if (not (delete-binding-from-frame var (first-frame env)))
    (unbind-var var (enclosing-environment env))))
```

```scheme
; Returns true if the variable was found in the given frame, false otherwise.
(define (delete-binding-from-frame var frame)
  (define (delete prev-vars curr-vars prev-vals curr-vals)
    (if (null? curr-vars)
      false
      (if (eq? (car curr-vars) var)
        (begin
          (set-cdr! prev-vars (cdr curr-vars))
          (set-cdr! prev-vals (cdr curr-vals))
          true)
        (delete curr-vars (cdr curr-vars) curr-vals (cdr curr-vals)))))
  (let ((vars (frame-variables frame))
        (vals (frame-values frame)))
    (if (null? vars)
      false
      (if (eq? var (car vars))
        (begin
          (set-car! frame (cdr vars))
          (set-cdr! frame (cdr vals))
          true)
        (delete vars (cdr vars) vals (cdr vals))))))
```

The implementation of `delete-binding-from-frame` is simpler with a list of pairs than with a pair of lists because there is only one list to modify, and the list is headed, so the case where the variable to unbind is the first in the frame need not be handled separately:

```scheme
(define (delete-binding-from-frame var frame)
  (define (delete prev-bindings curr-bindings)
    (if (null? curr-bindings)
      false
      (if (eq? (caar curr-bindings) var)
        (begin
          (set-cdr! prev-bindings (cdr curr-bindings))
          true)
        (delete curr-bindings (cdr curr-bindings)))))
  (delete frame (frame-bindings frame)))
```

4.1.4 Running the Evaluator as a Program

Exercise 4.14

Louis' `map` fails because the `map` procedure from the underlying Scheme is called with a procedure representation from the interpreter as the procedure to apply with the underlying Scheme's `apply`, and it considers that representation as a list and not as a procedure.

4.1.5 Data as Programs

Exercise 4.15

Let's assume that `(halts? try try)` returns true. Then `(run-forever)` is executed so `(try try)` does not halt.

Let's assume that on the contrary, `(halts? try try)` returns false. Then `(try try)` halts.

Both assumptions lead to a contradiction, so such a `halts?` procedure can't exist.

4.1.6 Internal Definitions

Exercise 4.16

a. Here is the updated version of `lookup-variable-value` for the frame representation as a list of pairs, with the abstractions defined in exercise 4.12:

```
(define (lookup-variable-value var env)
  (traverse-env env
                var
                (lambda (binding)
                  (let ((value (cdr binding)))
                    (if (eq? value '*unassigned*)
                      (error "Unassigned variable" var)
                      value))))))
```

b. The `scan-out-defines` procedure can be defined as:

```
(define (scan-out-defines body)
  (define (defines->let vars vals body)
    (if (null? vars)
      body
      (list (make-let (map (lambda (var)
                             (list var ''*unassigned*))
                           vars)
                      (sequence->exp
                        (append
                          (map (lambda (var val)
                                 (make-set! var val))
                               vars
                               vals)
                          body)))))))
  (define (scan body vars vals exps)
    (if (null? body)
      (defines->let (reverse vars) (reverse vals) (reverse exps))
      (let ((first (first-exp body)))
        (if (definition? first)
          (scan (cdr body)
                (cons (definition-variable first) vars)
```

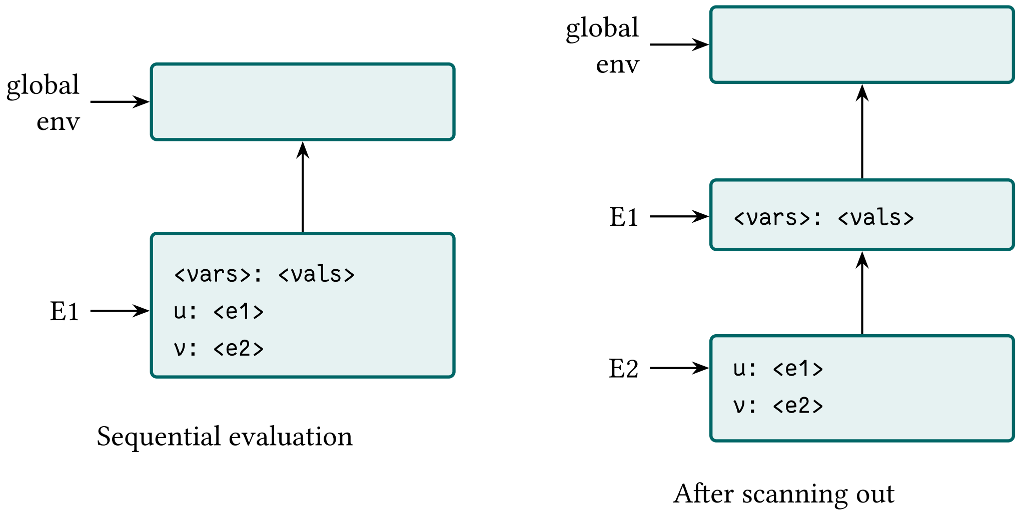

Figure 4.1: Environment structures in which <e3> is evaluated when the definitions are interpreted sequentially and after scanning out.

```
                  (cons (definition-value first) vals)
                  exps)
            (scan (cdr body) vars vals (cons first exps))))))
    (scan body '() '() '()))
```

No transformation must be done if no definition is found because `let` expressions are transformed into procedure applications, so this would lead to infinite loops. The `*unassigned*` symbol has to be quoted twice so that the returned code contains a quoted symbol.

c. It is better to install `scan-out-defines` in `make-procedure` than in `procedure-body`, because the latter is called every time the procedure is applied, while the former is called only when it is defined.

```
(define (make-procedure parameters body env)
  (list 'procedure parameters (scan-out-defines body) env))
```

Exercise 4.17

Figure 4.1 shows the environment structures in which <e3> is evaluated when the definitions are interpreted sequentially and when they are scanned out. In the later case, there is an extra frame because `let` corresponds to a `lambda` application.

This difference in environment structure makes no difference in a correct program's behavior because the same bindings are accessible when the body of the procedure is evaluated, and since the inner `let` contains the whole body of the outer lambda, both frames in the later case always go together: if the outer lambda returns a procedure, its environment will be E2, it can never be E1, so there can be no lost bindings.

A possible solution would be to move all the inner definitions to the top of the procedure body. This will work only if the values of the defined variables don't use the value of a variable defined later. This can be implemented in the following way:

```
(define (defines-first body)
  (define (scan body defines exps)
    (if (null? body)
      (append (reverse defines) (reverse exps))
      (let ((first (first-exp body)))
        (if (definition? first)
          (scan (cdr body) (cons first defines) exps)
          (scan (cdr body) defines (cons first exps))))))
  (scan body '() '()))

(define (make-procedure parameters body env)
  (list 'procedure parameters (defines-first body) env))
```

Exercise 4.18

This procedure won't work if internal definitions are scanned out as in this exercise because the value of `y` is needed to evaluate the value of `dy`, and in the exercise the value has been computed but not yet assigned to `y`.

It works if internal definitions are scanned out as shown in the text.

Exercise 4.19

Inner definitions should be simultaneous, so Eva is right, but if it's too difficult to implement internal definitions so they behave that way, it's better to signal an error than to use an incorrect value as Ben suggests.

To implement Eva's idea, we would have to sort the definitions so that if the value of a defined variable is needed to compute another's value, it should come first. However, it won't always be possible, definitions such as

```
(define (f x)
    (define a (+ b 5))
    (define b (+ a 1))
    <exps>)
```

should result in an error. But there are cases of mutual recursion that work, so it's not enough to scan the defined symbols' names in other defined symbols' values to order the definitions: mutually recursive procedure definitions are not problematic as long as neither procedure is called before both are defined. Mutual recursion is not problematic either in cases where evaluation is delayed, as in the `solve` example. A symbol could also appear at a place where it will never be evaluated, for instance:

```
(define (f x)
    (define a (if (> 0 1) (* b 2) 3))
    (define b 5)
    <exps>)
```

So it seems difficult to implement a general solution that does what Eva prefers.

Another possible solution is to make definitions and assignments lazy by automatically delaying their values' evaluation, and forcing them only when a variable's value is looked up. This is a big change to Scheme's evaluation order, but since we have already seen how to use `delay` and `force` in section 3.5 this is not difficult to implement. We need to redefine `eval-definition` and `eval-assignment` so they delay the values' evaluation:

```
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (delay (eval (definition-value exp) env))
                    env)
  'ok)

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (delay (eval (assignment-value exp) env))
                       env)
  'ok)
```

Then we must call `force` when we look up a variable's value, for instance by modifying the `variable?` case in `eval`:

```
((variable? exp) (force (lookup-variable-value exp env)))
```

Exercise 4.20

a. `Letrec` can be implemented as shown below. This is very similar to exercise 4.16, so the code could be mutualized:

```
(define (letrec? exp)
  (tagged-list? exp 'letrec))

(define (letrec-bindings exp)
  (cadr exp))

(define (letrec-body exp)
  (cddr exp))

(define (letrec->let exp)
  (let* ((bindings (letrec-bindings exp))
         (vars (map car bindings))
         (vals (map cadr bindings)))
    (if (null? vars)
      (letrec-body exp)
      (make-let (map (lambda (var)
                       (list var ''*unassigned*))
```

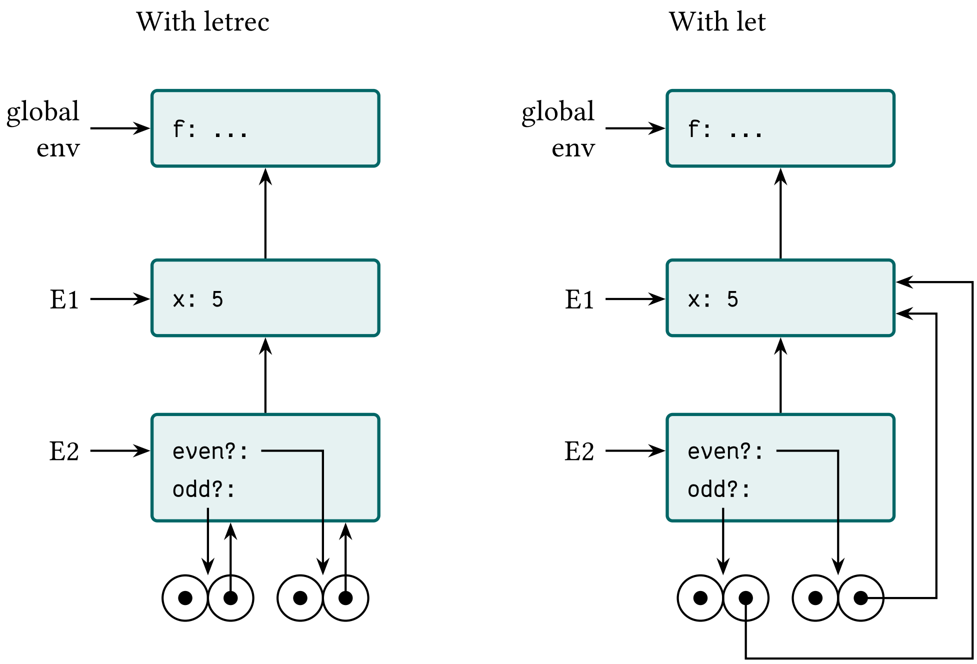

Figure 4.2: Environment structures in which `<rest of body of f>` is evaluated during evaluation of the expression `(f 5)`.

```
                vars)
           (sequence->exp
             (append
               (map (lambda (var val)
                      (make-set! var val))
                    vars
                    vals)
               (letrec-body exp)))))))
```

b. Figure 4.2 shows the environment structures in which the `<rest of body of f>` is evaluated during evaluation of the expression `(f 5)`, first with `letrec`, then with `let`. With `letrec`, the values of `odd?` and `even?` are evaluated in an environment where the bindings already exist, so the procedures created point to an environment where the other is defined. With `let`, the values of `even?` and `odd?` are evaluated before the bindings exist, so they point to an environment where they are undefined and the recursive calls won't work.

Exercise 4.21

a. The trick used here is to modify the recursive procedure (`(lambda (ft k) ...)` in the exercise) so it takes as an additional parameter the procedure to call where a recursive call would be used otherwise, and the argument used is the modified procedure itself.

The first inner `lambda` (`(lambda (fact) ...)`) is used to do the initial call to the modified (derecursified?) procedure.

The Fibonacci numbers can be computed similarly:

```
(define fib
  (lambda (n)
    ((lambda (fib)
       (fib fib n))
     (lambda (f k)
       (cond ((= k 0) 0)
             ((= k 1) 1)
             (else
               (+ (f f (- k 1))
                  (f f (- k 2)))))))))
```

b. This time, since there are two mutually recursive procedures, two procedure parameters are needed to pass around the procedures to call when we are not in the terminal case.

```
(define (f x)
  ((lambda (even? odd?)
     (even? even? odd? x))
   (lambda (ev? od? n)
     (if (= n 0) true (od? ev? od? (- n 1))))
   (lambda (ev? od? n)
     (if (= n 0) false (ev? ev? od? (- n 1))))))
```

4.1.7 Separating Syntactic Analysis from Execution

Exercise 4.22

Since `let` is a derived form, all that's needed to support it is to add the following line to `analyze`:

```
((let? exp) (analyze (let->combination exp)))
```

Complement:
We can just as easily add support for all the derived forms implemented in the previous exercises:

```
((or? exp) (analyze (or->if exp)))
((and? exp) (analyze (and->if exp)))
((let*? exp) (analyze (let*->nested-lets exp)))
((letrec? exp) (analyze (letrec->let exp)))
((while? exp) (analyze (while->if exp)))
((until? exp) (analyze (until->while exp)))
((for? exp) (analyze (for->let exp)))
```

We need to implement `analyze` procedures to support the versions of `or` and `and` defined with special evaluation functions, as well as `make-unbound!`:

```
(define (analyze-or exp)
  (lambda (env)
    (define (combine-procs procs)
```

```
      (if (null? procs)
        false
        (let ((first ((car procs) env)))
          (if (true? first)
            first
            (combine-procs (cdr procs))))))
    (combine-procs (map analyze (or-tests exp)))))

(define (analyze-and exp)
  (let ((exps (and-tests exp)))
    (if (null? exps)
      (lambda (env) true)
      (lambda (env)
        (define (combine-procs procs)
          (let ((first ((car procs) env)))
            (if (true? first)
              (if (null? (cdr procs))
                first
                (combine-procs (cdr procs)))
              false)))
        (combine-procs (map analyze exps))))))

(define (analyze-unbind exp)
  (lambda (env)
    (unbind-var (unbind-variable exp) env)))
```

Lastly, we can implement scanning out of internal definitions as in exercise 4.16 by calling `scan-out-defines` in `analyze-lambda`:

```
(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze-sequence (scan-out-defines (lambda-body exp)))))
    (lambda (env) (make-procedure vars bproc env))))
```

Exercise 4.23

If the sequence has just one expression, the procedure produced by Alyssa's program tests the `cdr` of `procs`, finds that it's null and then calls the first procedure. The program in the text does the test during analysis and returns the result of analyzing the expression directly, so the only work done during evaluation is to call it.

For a sequence with two expressions, Alyssa's procedure will loop through the procedures each time the sequence is evaluated, while the procedure from the text loops through them only once during analysis.

Exercise 4.24

I wrote a small procedure to interpret code with the interpreter without using the REPL (I should probably have done that some time ago...):

```
(define (run-in-interpreter . exps)
  (eval (sequence->exp exps) the-global-environment))
```

I then ran the following two tests: one with a recursive procedure and the other with a non-recursive procedure:

```
(time (run-in-interpreter
        '(define (fact n)
           (if (= n 0)
             1
             (* n (fact (- n 1)))))
        '(for (i 1 1000)
              (fact 500))))

(time (run-in-interpreter
        '(define (f x)
           (* 2 x))
        '(for (x 1 1000)
              (f x))))
```

With the evaluator in this section, the first test evaluates in 7877 ms, the second one in 36 ms.

With the original version, the first test evaluates in 16024 ms, the second one in 79 ms.

Separating analysis from execution speeds up execution by a factor of more than two, so we can estimate that about half the time was spent in analysis with the original version of the evaluator.

4.2 Variations on a Scheme—Lazy Evaluation

4.2.1 Normal Order and Applicative Order

Exercise 4.25

If we attempt to evaluate `(factorial 5)`, we will get an infinite loop since Scheme attempts to evaluate the second argument to `unless` recursively.

It would work in a normal-order language.

Exercise 4.26

Ben is right that it's possible to define `unless` as a derived expression:

```
(define (unless? exp)
  (tagged-list? exp 'unless))

(define (unless-condition exp)
  (cadr exp))

(define (unless-usual-value exp)
  (caddr exp))

(define (unless-exceptional-value exp)
```

```
  (cadddr exp))

(define (unless->if exp)
  (make-if (unless-condition exp)
           (unless-exceptional-value exp)
           (unless-usual-value exp)))
```

Then we just have to have the following line to `eval`:

```
((unless? exp) (eval (unless->if exp) env))
```

or the following line to `analyze` for the evaluator from section 4.1.7:

```
((unless? exp) (analyze (unless->if exp)))
```

It would be useful to have `unless` available as a procedure rather than a special form to use it as a parameter to higher-order procedures, for instance in `(map unless bools list1 list2)`, which returns a list where the element of index i is:

- the element of index i of `list2` if the element of index i in the list `bools` is true;
- the element of index i of `list1` otherwise.

4.2.2 An Interpreter with Lazy Evaluation

Exercise 4.27

The value of `count` the first time is 1 because the body of the outer `id` in the definition of `w` was evaluated. Then the value of `w` is 10 as expected, and once `w` has been forced the value of `count` is 2 because the inner `id` in the definition of `w` was evaluated as well.

Exercise 4.28

This forcing is needed if the operator has been passed as an argument to a higher-order procedure, for instance in `map`.

Exercise 4.29

Recursive procedures such as `factorial`, defined as usual as:

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

run much more slowly without memoization. Let's compare what happens when we evaluate, for intance `(factorial 10)` with and without memoization:

- With memoization, the argument is evaluated when the value of `(= n 0)` is needed in the `if` expression. Then `(factorial (- n 1))` is evaluated, and the argument passed to factorial is a thunk containing the expression `(- n 1)`, where `n` is an evaluated thunk with value 10. This new thunk is transformed into an evaluated thunk with value 9 when testing the predicate in the `if`, and this evaluated thunk's value is immediately accessible in the evaluation of the argument of the next call to `factorial`. And so on until 0 is reached. The time complexity of the computation is linear, the same as with applicative-order.

- Without memoization, the thunk with value 10 is evaluated in the `if` predicate, but it is not transformed into an evaluated thunk. So the recursive call to `factorial` has as its argument a thunk containing `(- n 1)`, where `n` is an unevaluated thunk. Two evaluations are needed to find that the value of this thunk is 9. Then, for the next recursive call, 3 evaluations are needed to find that the value of the argument is 8. During the terminal call, 11 thunk evaluations are needed each time the argument's value is needed. As a consequence, the time complexity of `factorial` with a lazy evaluator that does not memoize is quadratic instead of linear.

Of course, the same happens with any recursive procedure running in linear time, such as `length` to compute a list's length.

The value of `(square (id 10))` is 100 both when the evaluator memoizes and when it does not. When the evaluator memoizes, the value of `count` after evaluating `(square (id 10))` is 1 because `(id 10)` has been evaluated only once. When the evaluator does not memoize, the value of `count` is 2 because `(id 10)` has been evaluated twice.

Exercise 4.30

a. Ben is right about the behavior of `for-each` because each expression in the body of the `begin` expression is evaluated with `eval`, which causes each expression in the body of `proc` to be evaluated with each item in turn, so the side-effects they cause do take place.

   The only case where a side effect could not take place is when the expression defining it is delayed and then never forced, which can happen only if that expression was passed as an argument to a compound procedure, as in b. below.

b. With the original `eval-sequence`, the value of `(p1 1)` is `(1 2)` because the `set!` expression in the body of `p1` was evaluated. The value of `(p2 1)` is `1` because during the evaluation of `e` in the body of `p`, `e` is a variable whose value is a thunk containing an expression defining a side effect, but this expression is never actually evaluated because `e`'s value is not used.

   With Cy's proposed change to `eval-sequence`, the values would be `(1 2)` for both `(p1 1)` and `(p2 1)`.

c. The proposed change does not affect the behavior of the example in part a. because if the result of `eval` is not a thunk, applying `force-it` to it does nothing.

d. I prefer the approach in the text. I don't think the change proposed by Cy is necessary because, as noticed in part a., the only case when a side effect could not take place is when the expression causing it—not a procedure containing that expression, unless said procedure is not called of course—is passed as an argument to a compound procedure[1],

[1] It could also happen if an argument is the result of a procedure application that produces side effects and returns a value. If this argument's value is not needed in the body of the procedure, the side effects won't happen, but this is to be expected with normal-order evaluation and this is not directly related to the evaluation of sequences.

which I don't think should be done anyway: an expression passed as an argument should be there for its value, not for its side effects, and it does not make much sense to use an expression defining only side effects where a value should be used. For the example of `p2` in the exercise, if we want to use a parameter of `p` to define a side-effect before returning `x`, it is better to use a procedure, and it works without changing the evaluator's behavior:

```
(define (p3 x)
  (define (p e)
    (e)
    x)
  (p (lambda () (set! x (cons x '(2))))))
```

When reading the definition above, we can tell that the parameter `e` should be a procedure, and that it is expected to cause side-effects because its return value is not used. From reading the definition of `p2`, it is not clear at all what the evaluation of `e` is expected to do. It would be even less clear with applicative-order since `e` would already have been evaluated before applying `p`, so evaluating it again in the body of `p` would do absolutely nothing.

To sum it up, it can indeed happen that some side effects do not take place with the approach taken in the text, but I think that the programs where this happens can be modified in a straightforward way to fix the problem, and furthermore the modification is likely to make them clearer.

Exercise 4.31

In order to have both memoized and non memoized thunks, we define a new type of thunk with an associated delay procedure. I chose to define a new type for non memoized thunks. Then we redefine `force-it` so it handles the three types of thunks—unevaluated non memoized, unevaluated to memoize, and already evaluated—instead of two. And lastly we modify `apply` so it delays each argument or not as appropriate.

```
(define (delay-it-no-memo exp env)
  (list 'thunk-no-memo exp env))

(define (thunk-no-memo? exp)
  (tagged-list? exp 'thunk-no-memo))

(define (ext-force-it obj)
  (cond ((thunk? obj)
         (let ((result (actual-value
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)
           (set-cdr! (cdr obj) '())
           result))
        ((thunk-no-memo? obj)
```

```
         (actual-value (thunk-exp obj) (thunk-env obj)))
        ((evaluated-thunk? obj)
         (thunk-value obj))
        (else obj)))

(define (ext-apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure
           procedure
           (list-of-arg-values arguments env)))
        ((compound-procedure? procedure)
         (let* ((params (procedure-parameters procedure))
                (param-names (map (lambda (x) (if (list? x) (car x) x))
                                  params))
                (param-types (map (lambda (x) (if (list? x) (cadr x) 'default))
                                  params)))
           (eval-sequence
             (procedure-body procedure)
             (extend-environment
               param-names
               (ext-list-of-delayed-args arguments param-types env)
               (procedure-environment procedure)))))
        (else
          (error
            "Unknown procedure type -- APPLY" procedure))))

(define (ext-list-of-delayed-args exps types env)
  (define (value exp type)
    (cond ((eq? 'default type)
           (actual-value exp env))
          ((eq? 'lazy type)
           (delay-it-no-memo exp env))
          ((eq? 'lazy-memo type)
           (delay-it exp env))
          (else
            (error "Unknown argument type -- LIST-OF-ARGUMENTS" type))))
  (if (no-operands? exps)
    '()
    (cons (value (first-operand exps) (car types))
          (ext-list-of-delayed-args (rest-operands exps)
                                    (cdr types)
                                    env))))
```

4.2.3 Streams as Lazy Lists

Exercise 4.32

With the streams of section 3.5, there were places where we had to define the first element of a stream separately to avoid infinite loops. With the lazy evaluator, we can simplify some of the definitions. For instance, the following definition of `pairs` taken from exercise 3.68, where it caused an infinite loop, works for lazy lists:

```
(define (interleave l1 l2)
  (if (null? l1)
    l2
    (cons (car l1)
          (interleave l2 (cdr l1)))))

(define (pairs s t)
  (interleave
    (map (lambda (x) (list (car s) x))
         t)
    (pairs (cdr s) (cdr t))))

(define int-pairs (pairs integers integers))
```

Similar simplifications can be applied to the answers to exercises 3.69 and 3.70.

As noticed in the text, the values of the elements of the lazy list won't actually be computed until they are absolutely needed, so that for instance, if we define the lazy lists:

```
(define l1 (cons 1 (cons 2 (cons 3 '()))))

(define l2 (map (lambda (x) (/ x 0)) l1))
```

we get no error, while this would cause an error with streams.

And if we define a `length` procedure:

```
(define (length l)
  (if (null? l)
    0
    (+ 1 (length (cdr l)))))
```

we can compute `l2`'s length without trouble since the values of the elements are not used.

As noted in the book's footnote, lazy lists allow us to define arbitrary lazy data structures which can't be defined with the streams of section 3.5, such as trees where all branches could potentially be infinite.

Exercise 4.33

We need to convert the lists from the underlying Scheme to lazy lists using a series of `cons`. We now need to call `eval` when evaluating quotations, so the statement for quotations in `eval` is replaced with:

```
((quoted? exp) (eval-quotation exp env))
```

and the `eval-quotation` procedure can be defined as:

```
(define (eval-quotation exp env)
  (define (convert-to-lazy-list l)
    (if (null? l)
      '()
      (list 'cons (list 'quote (car l))
            (convert-to-lazy-list (cdr l)))))
  (let ((text (cadr exp)))
    (if (list? text)
      (if (null? text)
        '()
        (eval (convert-to-lazy-list text) env))
      text)))
```

Exercise 4.34

My initial idea was to implement `cons`, `car` and `cdr` as special forms. In the end, I decided to find a solution where they are ordinary procedures so that they can be used in higher-order procedures.

I redefined `cons` so that it produces a pair with the tag `lazy-pair`, the actual pair being represented by the same procedure as in the text. But the tagged pair must be produced with the `cons` procedure from the underlying Scheme, because it needs to be recognized by the implementation language. So we first save the values of `cons`, `car` and `cdr` as initial procedures before redefining them in the interpreter as shown below:

```
(define underlying-cons cons)
(define underlying-car car)
(define underlying-cdr cdr)

(define (cons x y)
  (underlying-cons 'lazy-pair (lambda (m) (m x y))))

(define (lazy-pair? x)
  (and (pair? x)
       (eq? (underlying-car x) 'lazy-pair)))

(define (car z)
  (if (lazy-pair? z)
    ((underlying-cdr z) (lambda (p q) p))
    (error "Not a pair -- CAR" z)))

(define (cdr z)
  (if (lazy-pair? z)
```

```
    ((underlying-cdr z) (lambda (p q) q))
    (error "Not a pair -- CDR" z)))
```

I also defined a `lazy-struct->pairs` procedure in the implemented language to transform a structure built from lazy pairs into a structure built from ordinary pairs. It uses the value of the global variables `*max-depth*` and `*max-breadth*` to limit the number of items included in the result. The advantage of defining the transformation in the implemented language rather than in the evaluator is that the values of `*max-depth*` and `*max-breadth*` can be changed from the driver loop.

```
(define *max-depth* 5)
(define *max-breadth* 20)

(define (lazy-struct->pairs l)
  (define (rec items depth breadth)
    (cond ((not (lazy-pair? items)) items)
          ((or (< depth 0) (= breadth 0))
           (underlying-cons '<...> '()))
          (else
            (underlying-cons (rec (car items) (- depth 1) *max-breadth*)
                             (rec (cdr items) depth (- breadth 1))))))
  (rec l *max-depth* *max-breadth*))
```

Then we need to define `lazy-pair?` in the underlying language so that the evaluator can identify them:

```
(define (lazy-pair? exp)
  (tagged-list? exp 'lazy-pair))
```

Then we can rewrite `user-print` so it applies `lazy-struct->pairs` to lazy pairs before printing the result:

```
(define (user-print object)
  (cond ((compound-procedure? object)
         (write (list 'compound-procedure
                      (procedure-parameters object)
                      (procedure-body object)
                      '<procedure-env>)))
        ((lazy-pair? object)
         (write (eval (list 'lazy-struct->pairs object) the-global-environment)))
        (else
          (write object))))
```

The last step is to modify the evaluator so it treats lazy pairs as self-evaluating values, for instance by adding:

```
((lazy-pair? exp) exp)
```

4.3 Variations on a Scheme—Nondeterministic Computing

4.3.1 Amb and Search

Exercise 4.35

The procedure `an-integer-between` can be defined as:

```
(define (an-integer-between low high)
  (require (<= low high))
  (amb low (an-integer-between (+ low 1) high)))
```

Exercise 4.36

Replacing `an-integer-between` by `an-integer-starting-from` in the procedure in exercise 4.35 wouldn't work because the interpreter would pick the lowest possible value for i and j and would then try all the possible values for k without ever finding a successful triple.

We can generate all Pythagorean triples by first picking j, and then picking $i \leq j$. We can then define k as $\sqrt{i^2 + j^2}$ and check if it's an integer. To define a procedure more similar to the one in exercise 4.35, we could choose k between finite bounds, for instance j and $2j$. Either way, once j has been picked, there are only a finite number of possibilities to test for i and k, and these possibilities contain all the Pythagorean triples for the given value of j, so all Pythagorean triples could in principle be generated by typing `try-again`.

```
(define (a-pythagorean-triple)
  (let ((j (an-integer-starting-from 1)))
    (let ((i (an-integer-between 1 j)))
      (let ((k (sqrt (+ (* i i) (* j j)))))
        (require (integer? k))
        (list i j k)))))
```

Exercise 4.37

Ben is correct. The procedure in exercise 4.35 systematically searches all possible triples (i, j, k) with $low \leq i \leq j \leq k \leq high$. Ben's version eliminates the pairs (i, j) for which $i^2 + j^2 > high^2$ since there is no possible value of k within the bounds in this case, and then instead of trying all possible values for k, it tests only whether $\sqrt{i^2 + j^2}$ is an integer, since this is necessarily the value of k if (i, j, k) is a Pythagorean triple.

4.3.2 Examples of Nondeterministic Programs

Logic Puzzles

Exercise 4.38

The only modification needed is to remove the line:

```
(require (not (= (abs (- smith fletcher)) 1)))
```

Without this requirement, there are five possible solutions:

```
((baker 1) (cooper 2) (fletcher 4) (miller 3) (smith 5))
((baker 1) (cooper 2) (fletcher 4) (miller 5) (smith 3))
((baker 1) (cooper 4) (fletcher 2) (miller 5) (smith 3))
((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))
((baker 3) (cooper 4) (fletcher 2) (miller 5) (smith 1))
```

Exercise 4.39

The order of the restrictions does not affect the answer. It does affect the time to find an answer but in a limited way: the original procedure takes about 370 ms on my computer, and I can't do better than about 270 ms with the following order:

```
(define (multiple-dwelling2)
  (let ((baker (amb 1 2 3 4 5))
        (cooper (amb 1 2 3 4 5))
        (fletcher (amb 1 2 3 4 5))
        (miller (amb 1 2 3 4 5))
        (smith (amb 1 2 3 4 5)))
    (require (> miller cooper))
    (require (distinct? (list baker cooper fletcher miller smith)))
    (require (not (= (abs (- smith fletcher)) 1)))
    (require (not (= (abs (- fletcher cooper)) 1)))
    (require (not (= baker 5)))
    (require (not (= cooper 1)))
    (require (not (= fletcher 1)))
    (require (not (= fletcher 5)))
    (list (list 'baker baker)
          (list 'cooper cooper)
          (list 'fletcher fletcher)
          (list 'miller miller)
          (list 'smith smith))))
```

The only influence of the order of restrictions is that a possibility that leads to a dead end can be rejected after less computation time depending on that order. However, the real reason why the procedure is slow is that a lot of possibilities are explored though they could be ruled out from the start (as shown in the following exercise). For instance, `cooper` must not be equal to 1, but for a given value of `baker`, the procedure will explore and eliminate all the $5^3 = 75$ possibilities where `cooper` is 1 before setting `cooper` to 2.

Exercise 4.40

Before the requirement that floor assignments be distinct, there are $5^5 = 3125$ sets of assignments of people to floors. After that requirement, there are $5! = 120$ such sets.

The following procedure finds the answer in about 40 ms:

```
(define (multiple-dwelling3)
  (let ((cooper (amb 1 2 3 4 5))
        (miller (amb 1 2 3 4 5)))
```

```
    (require (not (= cooper 1)))
    (require (> miller cooper))
    (let ((fletcher (amb 1 2 3 4 5)))
      (require (not (= fletcher 1)))
      (require (not (= fletcher 5)))
      (require (not (= (abs (- fletcher cooper)) 1)))
      (let ((smith (amb 1 2 3 4 5)))
        (require (not (= (abs (- smith fletcher)) 1)))
        (let ((baker (amb 1 2 3 4 5)))
          (require (not (= baker 5)))
          (require (distinct? (list baker cooper fletcher miller smith)))
          (list (list 'baker baker)
                (list 'cooper cooper)
                (list 'fletcher fletcher)
                (list 'miller miller)
                (list 'smith smith)))))))
```

We can make it even faster by breaking up the `(require (distinct? ...))` requirement into a series of `(require (not (= ...)))` so that any case where two values are not distinct is ruled out as soon as possible. The following procedure solves the problem in about 20 ms:

```
(define (multiple-dwelling4)
  (let ((cooper (amb 1 2 3 4 5))
        (miller (amb 1 2 3 4 5)))
    (require (not (= cooper 1)))
    (require (> miller cooper))
    (let ((fletcher (amb 1 2 3 4 5)))
      (require (not (= fletcher 1)))
      (require (not (= fletcher 5)))
      (require (not (= fletcher cooper)))
      (require (not (= fletcher miller)))
      (require (not (= (abs (- fletcher cooper)) 1)))
      (let ((smith (amb 1 2 3 4 5)))
        (require (not (= smith cooper)))
        (require (not (= smith miller)))
        (require (not (= smith fletcher)))
        (require (not (= (abs (- smith fletcher)) 1)))
        (let ((baker (amb 1 2 3 4 5)))
          (require (not (= baker 5)))
          (require (not (= baker cooper)))
          (require (not (= baker fletcher)))
          (require (not (= baker miller)))
          (require (not (= baker smith)))
          (list (list 'baker baker)
                (list 'cooper cooper)
```

```
                (list 'fletcher fletcher)
                (list 'miller miller)
                (list 'smith smith)))))))
```

It would also be more efficient to modify the list of possibilities in `amb` for the people with floor restrictions, for instance using `(amb 2 3 4)` for `fletcher`. But this could be considered as solving part of the problem for the evaluator instead of stating the solution's requirements.

Exercise 4.41

Each assignment of people to floors where each floor has a unique person corresponds to a permutation of the set `(1 2 3 4 5)`, where we can consider that the first number is Baker's floor, the second number is Cooper's floor and so on. We can solve the puzzle by generating all the permutations and then filtering the set to keep only those that verify the puzzle's requirements.

We already defined a `permutations` procedure in section 2.2.3, but I rewrote a different procedure to generate them before remembering that, and then I noticed that my version was faster so I kept it. To generate the permutations of a set S, I generate all the permutations of the set minus the first element, and then for each permutation I insert the removed element at each possible position in the returned list. The version in section 2.2.3 generated all permutations of $S - x$ for *each* element x of S and adjoined x at the front of each permutation, whereas I generate all permutations of $S - x$ only for one x and the adjoin x at all possible positions.

```
(define (multiple-dwelling)
  (filter (lambda (l)
            (let ((baker (list-ref l 0))
                  (cooper (list-ref l 1))
                  (fletcher (list-ref l 2))
                  (miller (list-ref l 3))
                  (smith (list-ref l 4)))
              (and (not (= baker 5))
                   (not (= cooper 1))
                   (not (= fletcher 5))
                   (not (= fletcher 1))
                   (> miller cooper)
                   (not (= (abs (- smith fletcher)) 1))
                   (not (= (abs (- fletcher cooper)) 1)))))
          (permutations '(1 2 3 4 5))))

(define (permutations s)
  (define (insert-elt-all-pos ps elt)
    (if (null? ps)
      (list (list elt))
      (cons (cons elt ps)
            (map (lambda (l) (cons (car ps) l))
                 (insert-elt-all-pos (cdr ps) elt)))))
  (if (null? s)
```

```
    (list '())
    (flatmap (lambda (ps) (insert-elt-all-pos ps (car s)))
             (permutations (cdr s)))))
```

Exercise 4.42

We can solve the puzzle by defining a `liars` procedure in the amb evaluator as shown below. It uses the helper procedure `xor` to simplify the expression of the requirements.

```
(define (xor a b)
  (or (and a (not b))
      (and (not a) b)))

(define (liars)
  (let ((ethel (amb 1 2 3 4 5))
        (joan (amb 1 2 3 4 5)))
    (require (xor (= ethel 1) (= joan 2)))
    (require (xor (= joan 3) (= ethel 5)))
    (let ((betty (amb 1 2 3 4 5))
          (kitty (amb 1 2 3 4 5)))
      (require (xor (= kitty 2) (= betty 3)))
      (let ((mary (amb 1 2 3 4 5)))
        (require (xor (= kitty 2) (= mary 4)))
        (require (xor (= mary 4) (= betty 1)))
        (require (distinct? (list betty ethel joan kitty mary)))
        (list (list 'Betty betty)
              (list 'Ethel ethel)
              (list 'Joan joan)
              (list 'Kitty kitty)
              (list 'Mary mary))))))
```

The procedure's output is:

```
((Betty 3) (Ethel 5) (Joan 2) (Kitty 1) (Mary 4))
```

so the order in which the girls were placed was: Kitty, Joan, Betty, Mary, and Ethel.

Exercise 4.43

The puzzle can be solved by the following procedure, where we define lists containing the daughter's name and the boat's name for each father.

```
(define (names)
  (define (daughter l) (car l))
  (define (boat l) (cadr l))
  (define names (list 'Mary-Ann 'Gabrielle 'Lorna 'Rosalind 'Melissa))
  ; First element = daughter's name, second element = boat's name
  (let ((barnacle (list (an-element-of names) (an-element-of names))))
    (require (eq? (boat barnacle) 'Gabrielle))
```

```
    (require (eq? (daughter barnacle) 'Melissa))
    (require (distinct? barnacle))
    (let ((moore (list (an-element-of names) (an-element-of names))))
      (require (eq? (daughter moore) 'Mary-Ann))
      (require (eq? (boat moore) 'Lorna))
      (require (distinct? moore))
      (let ((hall (list (an-element-of names) (an-element-of names))))
        (require (eq? (boat hall) 'Rosalind))
        (require (distinct? hall))
        (let ((downing (list (an-element-of names) (an-element-of names))))
          (require (eq? (boat downing) 'Melissa))
          (require (distinct? downing))
          (let ((parker (list (an-element-of names) (an-element-of names))))
            (require (distinct? parker))
            (let ((fathers (list moore downing hall barnacle parker)))
              (require (distinct? (map daughter fathers)))
              (require (distinct? (map boat fathers)))
              (for-each (lambda (father)
                          (require (or (not (eq? (daughter father) 'Gabrielle))
                                       (eq? (daughter parker) (boat father)))))
                        fathers))
            (list (list 'Moore moore)
                  (list 'Downing downing)
                  (list 'Hall hall)
                  (list 'Barnacle barnacle)
                  (list 'Parker parker))))))))
```

Some of the requirements are redundant, for instance since we know the name of Barnacle's boat and of his daughter, it's not strictly necessary to check that they are distinct. Keeping these requirements makes it easier to modify the procedure if some constraints are removed.

The procedure returns a single possible solution, which is

```
((Moore (Mary-Ann Lorna))
 (Downing (Lorna Melissa))
 (Hall (Gabrielle Rosalind))
 (Barnacle (Melissa Gabrielle))
 (Parker (Rosalind Mary-Ann)))
```

so Lorna's father is Colonel Downing.

If we are not told that Mary Ann's father is Moore, the problem has two solutions. The first one is the same as above, the second one is:

```
((Moore (Gabrielle Lorna))
 (Downing (Rosalind Melissa))
 (Hall (Mary-Ann Rosalind))
```

```
(Barnacle (Melissa Gabrielle))
(Parker (Lorna Mary-Ann)))
```

Exercise 4.44

We use the same idea as in exercise 2.42: we place a queen in each column successively. Once we have placed $k-1$ queens in the first $k-1$ columns, we place a queen in the kth column and require that it is safe with respect to the others. The `safe?` procedure is the same as in exercise 2.42.

```
(define (a-queens-pos board-size)
  (define (safe? positions)
    (define (iter delta-col rest-cols)
      (if (null? rest-cols)
        true
        (let ((new-queen-pos (car positions))
              (col-pos (car rest-cols)))
          (and (not (= new-queen-pos col-pos))
               (not (= new-queen-pos (+ col-pos delta-col)))
               (not (= new-queen-pos (- col-pos delta-col)))
               (iter (+ delta-col 1) (cdr rest-cols))))))
    (iter 1 (cdr positions)))

  (define (a-queens-cols-pos k)
    (if (= k 0)
      '()
      (let ((prev-cols (a-queens-cols-pos (- k 1)))
            (new-col-pos (an-integer-between 1 board-size)))
        (let ((new-pos (cons new-col-pos prev-cols)))
          (require (safe? new-pos))
          new-pos))))

  (a-queens-cols-pos board-size))
```

Parsing natural language

Exercise 4.45

The five ways in which the sentence can be parsed are:

- The professor lectures with the cat, in the class, to the student.

```
(sentence
  (simple-noun-phrase (article the) (noun professor))
  (verb-phrase
    (verb-phrase
      (verb-phrase (verb lectures)
                   (prep-phrase (prep to)
```

```
                                (simple-noun-phrase
                                  (article the)
                                  (noun student))))
     (prep-phrase (prep in)
                  (simple-noun-phrase (article the)
                                      (noun class))))
    (prep-phrase (prep with)
                 (simple-noun-phrase (article the)
                                     (noun cat)))))
```

- The professor lectures to the student, in the class that has the cat.

```
(sentence
  (simple-noun-phrase (article the) (noun professor))
  (verb-phrase
    (verb-phrase (verb lectures)
                 (prep-phrase (prep to)
                              (simple-noun-phrase
                                (article the) (noun student))))
    (prep-phrase (prep in)
                 (noun-phrase (simple-noun-phrase
                                (article the) (noun class))
                              (prep-phrase (prep with)
                                           (simple-noun-phrase
                                             (article the) (noun cat))))))))
```

- The professor lectures with the cat, to the student who is in the class.

```
(sentence
  (simple-noun-phrase (article the) (noun professor))
  (verb-phrase
    (verb-phrase (verb lectures)
                 (prep-phrase (prep to)
                              (noun-phrase
                                (simple-noun-phrase (article the)
                                                    (noun student))
                                (prep-phrase (prep in)
                                             (simple-noun-phrase
                                               (article the) (noun class)))))))
    (prep-phrase (prep with)
                 (simple-noun-phrase (article the) (noun cat)))))
```

- The professor lectures to the student in the class, the student has the cat.

```
(sentence
  (simple-noun-phrase (article the) (noun professor))
  (verb-phrase
```

```
    (verb lectures)
    (prep-phrase (prep to)
                 (noun-phrase
                   (noun-phrase
                     (simple-noun-phrase (article the) (noun student))
                     (prep-phrase (prep in)
                                  (simple-noun-phrase
                                    (article the) (noun class))))
                   (prep-phrase (prep with)
                                (simple-noun-phrase
                                  (article the) (noun cat)))))))
```

- The professor lectures to the student, who is in the class that has the cat.

```
(sentence
  (simple-noun-phrase (article the) (noun professor))
  (verb-phrase
    (verb lectures)
    (prep-phrase
      (prep to)
      (noun-phrase
        (simple-noun-phrase (article the) (noun student))
        (prep-phrase
          (prep in)
          (noun-phrase
            (simple-noun-phrase (article the) (noun class))
            (prep-phrase (prep with)
                         (simple-noun-phrase (article the) (noun cat)))))))))
```

Exercise 4.46

The `parse-word` procedure looks for the required part of speech at the beginning of the contents of `*unparsed*`. So with `parse-sentence` defined as:

```
(define (parse-sentence)
  (list 'sentence
        (parse-noun-phrase)
        (parse-word verbs)))
```

when we try to parse `(the cat eats)`, if `(parse-word verbs)` is evaluated first, it will fail since `the` is not in the list of verbs, and there are no alternatives to try.

Exercise 4.47

The change suggested by Louis does not work because the second branch of the `amb` contains an infinite loop. For instance, if we parse the sentence "The cat eats.", we get the correct result, but if we then type `try-again`, there is an infinite loop as the evaluator tries the second branch of the `amb` in `parse-verb-phrase`: the recursive call to `parse-verb-phrase` succeeds as it finds the verb `eats`, then the call to `parse-prepositional-phrase` fails since there is nothing left to parse, which causes the second branch of `amb` to be explored in the recursive call, and so on to infinity.

If we interchange the order of the expressions in the `amb`, the infinite loop is immediately apparent.

Exercise 4.48

I decided to allow an arbitrary number of adjectives in front of a noun, and an adverb after a verb. This is done by running the following code in the interpreter:

```
(define adjectives '(adjective black big lazy beautiful clever))

(define (parse-adjectives)
  (define (maybe-extend adjectives-list)
    (amb adjectives-list
         (maybe-extend (append adjectives-list (list (parse-word adjectives))))))
  (maybe-extend (list (parse-word adjectives))))

(define (parse-simple-noun-phrase)
  (amb (list 'simple-noun-phrase
             (parse-word articles)
             (parse-word nouns))
       (list 'adjectival-noun-phrase
             (parse-word articles)
             (append (parse-adjectives)
                     (list (parse-word nouns))))))

(define adverbs '(adverb fast well))

(define (parse-verb-phrase)
  (define (maybe-extend verb-phrase)
    (amb verb-phrase
         (maybe-extend (list 'verb-phrase
                             verb-phrase
                             (parse-prepositional-phrase)))))
  (maybe-extend (parse-simple-verb-phrase)))

(define (parse-simple-verb-phrase)
  (amb (list 'simple-verb
             (parse-word verbs))
       (list 'verb-with-adverb
             (parse-word verbs)
             (parse-word adverbs))))
```

We can then parse sentences sentences such as “The big black cat eats fast.” with as result:

```
(sentence
  (adjectival-noun-phrase
    (article the)
    ((adjective big) (adjective black) (noun cat)))
  (verb-with-adverb (verb eats) (adverb fast)))
```

Exercise 4.49

We make `parse-word` return a random element of the given list by redefining it as:

```
(define (parse-word word-list)
  (let ((words (cdr word-list)))
    (list-ref words (random-integer (length words)))))
```

The first sentences I get using `(parse '())` (the input is ignored) are:

- The students eats.
- A class sleeps.
- A student eats.
- The cat studies.
- A class studies.
- A cat eats.

Every time, the evaluator selects the first choice in `amb` to generate the sentence, so all the sentences have the same structure.

The result is not much better if I type `(parse '())` and then type `try-again` several times:

- A professor studies.
- A professor studies by the student.
- A professor studies by the student by a student.
- A professor studies by the student by a student by the professor.
- A professor studies by the student by a student by the professor in the cat.
- A professor studies by the student by a student by the professor in the cat for the professor.

A sentence consists of a noun phrase followed by a verb phrase, and a verb phrase is a verb optionally followed by one or more prepositional phrases. Each use of `try-again` causes the evaluator to go back to the choice point in the `amb` in `parse-verb-phrase`, which adds a new prepositional phrase at the end of the sentence.

4.3.3 Implementing the `Amb` Evaluator

Exercise 4.50

We can define `ramb` by adding the following dispatch clause to `analyze`:

```
((ramb? exp) (analyze-ramb exp))
```

and defining the following procedures:

```
(define (ramb? exp) (tagged-list? exp 'ramb))

(define (ramb-choices exp) (cdr exp))

; Removes the item with the given index from the given list.
; Does nothing if the given index is higher than the list's length.
(define (remove-index items index)
  (cond ((null? items) '())
        ((= index 0) (cdr items))
        (else (cons (car items)
                    (remove-index (cdr items) (- index 1))))))

(define (analyze-ramb exp)
  (let ((cprocs (map analyze (ramb-choices exp))))
    (lambda (env succeed fail)
      (define (try-next choices)
        (if (null? choices)
          (fail)
          (let ((choice (random-integer (length choices))))
            ((list-ref choices choice)
             env
             succeed
             (lambda ()
               (try-next (remove-index choices choice)))))))
      (try-next cprocs))))
```

Alyssa can replace `amb` with `ramb` in `parse-verb-phrase` and `parse-noun-phrase` so that her generator will use a random sentence structure. The sentences generated by `(parse '())` exhibit various structures:

- A professor with the student for the class for a student in a student in a student to the student by the class to a class to the cat with a class for a student with the class in a professor with a student by a class to the cat by the class by a cat in a cat for the class with a class by a cat for a professor by the professor for a cat by the student for the professor in a student for the class with a class with a professor to the student in the class by the class for the professor in the cat in a professor to the cat by a class with a student for a class in a professor to a student to the professor for the professor for a student by a cat with the student by the student in the cat by a class by the class to a cat by a student for a class by a student to the professor with a student with the professor with the cat to the student eats.

- The cat to the class lectures.
- The cat sleeps in a class by the cat by a student for the student by a professor with a student for a class.
- The class to the student by the student eats.
- A student eats to a class in a student.
- A professor sleeps.

Some generated sentences if we also include the adjectives and adverbs added in exercise 4.48:

- A class studies.
- The student for the cat to a beautiful student by the student with a class for a class eats by the professor to the cat in the big student in the class for the student in a professor for the class in a lazy professor by the lazy student in a lazy black clever lazy clever class in a lazy class by the student to the cat in a class in the clever cat by the lazy cat in the lazy professor by the clever professor for the big beautiful big cat to the big clever lazy big cat with the student with a class by the class by the beautiful beautiful beautiful class with the student with the beautiful big black cat by the clever clever professor for a clever clever professor in a black big clever student for the cat for the professor with a big student to the student to the cat in the professor with a black student to the cat for a professor.
- A cat lectures fast to a professor to a professor.
- The black professor eats well by the beautiful beautiful professor.

Exercise 4.51

The `permanent-set!` assignement can be defined similarly to `set!`, except that it simply passes along the failure continuation instead of intercepting it to undo the change in case of failure:

```
(define (permanent-assignment? exp)
  (tagged-list? exp 'permanent-set!))

(define (permanent-assignment-variable exp) (cadr exp))

(define (permanent-assignment-value exp) (caddr exp))

(define (analyze-permanent-assignment exp)
  (let ((var (permanent-assignment-variable exp))
        (vproc (analyze (permanent-assignment-value exp))))
    (lambda (env succeed fail)
```

```
    (vproc env
           (lambda (val fail2)
             (set-variable-value! var val env)
             (succeed 'ok fail2))
           fail))))
```

If we had used `set!` rather than `permanent-set!`, the displayed values would have been `(a b 1)`, since the increment of `count` done during the first trial would have been undone before the second trial.

Exercise 4.52

The `if-fail` construct can be defined in the following way, after adding the appropriate clause to `analyze` as usual:

```
(define (if-fail? exp) (tagged-list? exp 'if-fail))

(define (if-fail-test exp) (cadr exp))

(define (if-fail-failure exp) (caddr exp))

(define (analyze-if-fail exp)
  (let ((pproc (analyze (if-fail-test exp)))
        (aproc (analyze (if-fail-failure exp))))
    (lambda (env succeed fail)
      (pproc env
             succeed
             (lambda ()
               (aproc env succeed fail))))))
```

Exercise 4.53

The result is `((8 35) (3 110) (3 20))`. When `(amb)` is evaluated, it causes the interpreter to go back to the previous choice point, in `prime-sum-pair`, and the `let` expression fails only when `prime-sum-pairs` has no more alternative to explore, after the three pairs whose sum is prime have been permanently added to `pairs`.

Exercise 4.54

The `analyze-require` procedure could have been defined as:

```
(define (analyze-require exp)
  (let ((pproc (analyze (require-predicate exp))))
    (lambda (env succeed fail)
      (pproc env
             (lambda (pred-value? fail2)
               (if (false? pred-value?)
                 (fail2)
                 (succeed 'ok fail2)))
             fail))))
```

4.4 Logic Programming

4.4.1 Deductive Information Retrieval

Simple queries

Exercise 4.55

The queries that retrieve the required information are:

a. `(supervisor ?x (Bitdiddle Ben))`

b. `(job ?x (accounting . ?type))`

c. `(address ?x (Slumerville . ?rest-address))`

Compound queries

Exercise 4.56

The queries that retrieve the required information are:

a.
```
(and (supervisor ?person (Bitdiddle Ben))
     (address ?person ?address))
```

b.
```
(and (salary (Bitdiddle Ben) ?ben-salary)
     (salary ?other ?other-salary)
     (lisp-value < ?other-salary ?ben-salary))
```

c.
```
(and (supervisor ?person ?supervisor)
     (job ?supervisor ?supervisor-job)
     (not (job ?supervisor (computer ?rest-job))))
```

Rules

Exercise 4.57

The rule `can-replace` can be defined as:

```
(rule (can-replace ?person-1 ?person-2)
      (and (job ?person-1 ?job-1)
           (job ?person-2 ?job-2)
           (not (same ?person-1 ?person-2))
           (or (same ?job-1 ?job-2)
               (can-do-job ?job-1 ?job-2))))
```

a. The people who can replace Cy D. Fect can be found with the query:

   `(can-replace (Fect Cy D) ?person)`

b. The people who can replace someone who is being paid more than they are can be found thanks to the query:

```
(and (can-replace ?person-1 ?person-2)
     (salary ?person-1 ?salary-1)
     (salary ?person-2 ?salary-2)
     (lisp-value < ?salary-1 ?salary-2))
```

Exercise 4.58

The rule can be defined as:

```
(rule (big-shot ?person ?division)
      (and (job ?person (?division . ?rest-job))
           (or (not (supervisor ?person ?boss))
               (and (supervisor ?person ?boss)
                    (not (job ?boss (?division . ?rest-job-boss)))))))
```

Exercise 4.59

a. Ben should use the query:

```
(meeting ?division (Friday ?time))
```

b. Alyssa's rule can be defined as:

```
(rule (meeting-time ?person ?day-and-time)
      (or (meeting whole-company ?day-and-time)
          (and (job ?person (?division . ?rest-job))
               (meeting ?division ?day-and-time))))
```

c. Alyssa should run the query:

```
(meeting-time (Hacker Alyssa P) (Wednesday ?time))
```

Exercise 4.60

Each pair appears twice because the rule's body is symmetric in the variables `?person-1` and `?person-2`, so that if none of them is bound by the query, if a frame where `?person-1` is bound to person A and `?person-2` is bound to person B appears in the result, the frame where `?person-1` is bound to person B and `?person-2` is bound to person A appears in the result too.

The obvious idea is to define an order on the variables' values and modify the rule so that, for instance, the value of `?person-1` is smaller than that of `?person-2`. The trouble is that it would not work with queries where one of the variables `?person-1` or `?person-2` is already bound: the expected behavior is that the two queries:

```
(lives-near (Hacker Alyssa P) ?x)
(lives-near ?x (Hacker Alyssa P))
```

lead to the same set of possible values for `?x`, which won't be the case if `lives-near` is not symmetric, so the idea of sorting the variables' values does not work. The solution would involve defining a rule that behaves differently depending on whether one of the variables `?person-1` and `?person-2` has a value imposed by the query, which does not seem possible, or at least not easily.

Logic as programs

Exercise 4.61

The response to the query `(?x next-to ?y in (1 (2 3) 4))` is:

```
((2 3) next-to 4 in (1 (2 3) 4))
(1 next-to (2 3) in (1 (2 3) 4))
```

The response to the query `(?x next-to 1 in (2 1 3 1)` is:

```
(3 next-to 1 in (2 1 3 1))
(2 next-to 1 in (2 1 3 1))
```

Exercise 4.62

The operation can be implemented with the following rules:

```
(rule (last-pair (?u . ?rest) (?last))
      (last-pair ?rest (?last)))
(rule (last-pair (?elt) (?elt)))
```

They give the expected results with queries such as `(last-pair (3) ?x)`, `(last-pair (1 2 3) ?x)` or `(last-pair (2 ?x) (3))`.

With queries such as `(last-pair ?x (3))`, the behavior depends on the rules'order because of the evaluator's implementation: if they are entered as written above, the rule corresponding to lists with one element is checked first (since new rules are added at the beginning of the stream of rules and the stream is checked sequentially), and the evaluator displays the elements of an infinite stream of results with unbound variables:

```
(last-pair (3) (3))
(last-pair (?u-2 3) (3))
(last-pair (?u-2 ?u-6 3) (3))
(last-pair (?u-2 ?u-6 ?u-10 3) (3))
(last-pair (?u-2 ?u-6 ?u-10 ?u-14 3) (3))
(last-pair (?u-2 ?u-6 ?u-10 ?u-14 ?u-18 3) (3))
(last-pair (?u-2 ?u-6 ?u-10 ?u-14 ?u-18 ?u-22 3) (3))
(last-pair (?u-2 ?u-6 ?u-10 ?u-14 ?u-18 ?u-22 ?u-26 3) (3))
(last-pair (?u-2 ?u-6 ?u-10 ?u-14 ?u-18 ?u-22 ?u-26 ?u-30 3) (3))
(last-pair (?u-2 ?u-6 ?u-10 ?u-14 ?u-18 ?u-22 ?u-26 ?u-30 ?u-34 3) (3))
(last-pair (?u-2 ?u-6 ?u-10 ?u-14 ?u-18 ?u-22 ?u-26 ?u-30 ?u-34 ?u-38 3)
        (3))
(last-pair (?u-2 ?u-6 ?u-10 ?u-14 ?u-18 ?u-22 ?u-26 ?u-30 ?u-34 ?u-38
        ?u-42 3) (3))
(last-pair (?u-2 ?u-6 ?u-10 ?u-14 ?u-18 ?u-22 ?u-26 ?u-30 ?u-34 ?u-38
        ?u-42 ?u-46 3) (3))
[...]
```

If the rules are entered in reverse order, the rule corresponding to lists of several elements is checked first and the evaluator enters an infinite loop as it tries to build an infinite list ending in 3, so no result is displayed.

Exercise 4.63

The rules can be defined as:

```
(rule (grandson ?grand-father ?grandson)
      (and (son ?grand-father ?father)
           (son ?father ?grandson)))

(rule (son ?father ?son)
      (and (wife ?father ?wife)
           (son ?wife ?son)))
```

4.4.2 How the Query System Works

This subsection contains no exercises.

4.4.3 Is Logic Programming Mathematical Logic?

Exercise 4.64

The evaluator unifies the query with the conclusion of the rule for `outranked-by` by binding the variable `?staff-person` to `(Bitdiddle Ben)` and `?boss` to `?who`. It then evaluates the rule's body and finds the assertion in the database corresponding to Ben's supervisor. This produces the first frame of the result stream, and an answer corresponding to this frame's binding is displayed. The interpreter then processes the first part of the `and` query: `(outranked-by ?middle-manager ?boss)`, with both variables unbound (more precisely, `?boss` is bound to an unbound variable). The recursive rule evaluation leads again to a recursive evaluation of an `outranked-by` rule, which leads to an infinite loop.

With the initial version of `outranked-by`, when evaluating the compound `and` query, the query `(supervisor ?staff-person ?boss)`, with `?staff-person` bound to a constant, would have been evaluated first, and it would have produced either an empty stream, which would have ended the proccessing for the given frame, or a stream containing a single frame with a binding for `?middle-manager`, which would have caused the processing of `(outranked-by ?staff-person ?boss)` with `?staff-person` bound to the name of someone higher in the hierarchy than in the previous processing of the rule. Since the tower of hierarchy is not infinite, the evaluation of the rule's body terminates eventually.

Exercise 4.65

Oliver Warbucks is listed four times because the interpreter found four sets of bindings for the variables appearing in the body of the rule for `wheel` with `?who` bound to `(Warbucks Oliver)`. We can understand the result better if we evaluate the rule's body at the driver loop:

```
(and (supervisor (Scrooge Eben) (Warbucks Oliver))
     (supervisor (Cratchet Robert) (Scrooge Eben)))
(and (supervisor (Hacker Alyssa P) (Bitdiddle Ben))
     (supervisor (Reasoner Louis) (Hacker Alyssa P)))
(and (supervisor (Bitdiddle Ben) (Warbucks Oliver))
     (supervisor (Tweakit Lem E) (Bitdiddle Ben)))
```

```
(and (supervisor (Bitdiddle Ben) (Warbucks Oliver))
     (supervisor (Fect Cy D) (Bitdiddle Ben)))
(and (supervisor (Bitdiddle Ben) (Warbucks Oliver))
     (supervisor (Hacker Alyssa P) (Bitdiddle Ben)))
```

Exercise 4.66

Ben has just realized that some values could appear multiple times in the result stream. For instance, if we try to sum all the wheels' salaries using:

```
(sum ?amount
     (and (wheel ?w)
          (salary ?w ?amount)))
```

the result will be the sum of Ben's salary and four times Oliver Warbucks' salary instead of being the sum of Ben's and Oliver Warbucks' salaries.

Ben could filter the frames so that each possible set of values for the variables appearing in the query appears only once in the filtered stream before extracting the value of the designated variable. With the example above, the unfiltered stream contains five frames. In one of them, `?w` is bound to `(Bitdiddle Ben)` and `?amount` is bound to `60000`, and in four of them `?w` is bound to `(Warbucks Oliver)` and `?amount` is bound to `150000`, but the values of some other variables bound in these frames differ. The filtered stream would contain only one frame for Ben Bitdiddle and one for Oliver Warbucks.

Keeping unique frames would not work because in the example above, the five frames in the result stream are distinct if we take all the bindings they define into account. Keeping unique values of the extracted variable would not work either because there can be legitimate duplicates: several employees can have the same salary for instance.

Exercise 4.67

A possible way to detect loops it to define a global history that is reset at the beginning of `query-driver-loop`. This history contains pairs with a query and a frame containing the bindings defined before the given query was processed. We modify `simple-query` to check whether the given query has already been processed before going forward. If a loop is detected, `simple-query` returns the empty stream. Otherwise it adds an entry to the history before computing the result.

To check whether a query has already been processed, we check whether the instantiation of the query in the current frame (with the numbers in unbound variables removed) is equal to the instantiation of the stored query in the associated stored frame (with the numbers in unbound variables removed as well). If that's the case, we check whether the stored frame is reachable from the current frame: if that's not the case, the stored history entry does not belong to the current chain of deductions so we would detect a loop where there is none and lose potential results.

The problem with this implementation is that, though it does prevent infinite loops, it also cuts off some computations that would have terminated. For instance, with the version of `outranked-by` from exercise 4.64, when evaluating `(outranked-by (Reasoner Louis) ?x)`, an answer with Alyssa P. Hacker is found from the first branch of the `or`. Then the recursive evaluation of `outranked-by` (which is not eliminated because both variables are unbound whereas one of them was bound in the query) allows the interpreter to find that Louis is outranked by Ben Bitdiddle. But the second recursion on `outranked-by` with both variables unbound is eliminated, so the third answer with Oliver Warbucks is not found. The evaluation of `(outranked-by ?x ?y)` is even worse since the second branch is completely eliminated so we only get the supervisors. With `last-pair` from exercise 4.62, where we got an infinite stream of results by evaluating `(last-pair ?x (3))`, we only get the first two results because the rest of the computation is eliminated.

I spent some time trying to find a better solution without much success: either the system cut too much or not enough. After a look at some research articles on the subject, it seems that it is really difficult to find a system that both eliminates all infinite loops and does not eliminate valid computations. From what I understood (which is not much), it is certainly not easily feasible, so I decided to stick with this version.

The modified implementation of `simple-query` and the other procedures needed can be defined as:

```scheme
(define (simple-query query-pattern frame-stream)
  (stream-flatmap
    (lambda (frame)
      (if (in-loop? query-pattern frame)
        the-empty-stream
        (begin
          (add-to-history! query-pattern frame)
          (stream-append-delayed
            (find-assertions query-pattern frame)
            (delay (apply-rules query-pattern frame))))))
    frame-stream))

(define (in-loop? query frame)
  (define (iter key history)
    (if (null? history)
      #f
      (let ((entry (car history)))
        (or (and (equal? key (entry-key entry))
                 (reachable-from? (entry-frame entry) frame))
            (iter key (cdr history))))))
  (iter (get-key query frame) *history*))

(define (reachable-from? target origin)
  (cond ((eq? origin target) #t)
        ((null? origin) #f)
        (else (reachable-from? target (cdr origin)))))

(define (get-key query frame)
  (instantiate query frame (lambda (var f)
```

```
                                   (if (number? (cadr var))
                                     (cons (car var) (cddr var))
                                     var))))

(define *history* '())
(define (reset-history!)
  (set! *history* '()))

(define (make-history-entry query frame)
  (cons (get-key query frame) frame))

(define (entry-frame history-entry)
  (cdr history-entry))

(define (entry-key history-entry)
  (car history-entry))

(define (add-to-history! query frame)
  (set! *history* (cons (make-history-entry query frame) *history*)))
```

It's also necessary to add `(reset-history!)` at the beginning of `query-driver-loop`.

Exercise 4.68

My initial rules were:

```
(rule (reverse () ()))
(rule (reverse (?a . ?rest) ?x)
      (and (reverse ?rest ?reverse-rest)
           (append-to-form ?reverse-rest (?a) ?x)))
```

These rules can answer `(reverse (1 2 3) ?x)` but not `(reverse ?x (1 2 3))` because in this case `(reverse ?rest ?reverse-rest)` is evaluated with both variables unbound and it leads to an infinite loop.

The second query can be solved by the rule:

```
(rule (reverse ?x (?a . ?rest))
      (and (reverse ?reverse-rest ?rest)
           (append-to-form ?reverse-rest (?a) ?x)))
```

but then it's the first one that leads to an infinite loop.

If we add the infinite loop detector from the previous exercise and use all three rules above, both queries are solved, and also queries like `(reverse (1 2 ?x) (3 . ?y))` and `(reverse (1 2  . ?x) (3 . ?y))`. However, `(reverse (1 2 . ?x) (4 3 . ?y))` returns no answer because the computation is interrupted by the loop detector.

Exercise 4.69

The rules can be defined in the following way:

```
(rule (ends-in-grandson (grandson)))

(rule (ends-in-grandson (?u . ?rest))
      (ends-in-grandson ?rest))

(rule ((grandson) ?x ?y)
      (grandson ?x ?y))

(rule ((great . ?rel) ?x ?y)
      (and (ends-in-grandson ?rel)
           (son ?x ?z)
           (?rel ?z ?y)))
```

Without the loop detector, queries such as `(?rel Adam Irad)` cause an infinite loop. With the loop detector, there are no infinite loops but queries such as `((great great grandson) ?x ?y)` return no result (though `((great grandson) ?x ?y)` works).

4.4.4 Implementing the Query System

Exercise 4.70

It's necessary to use `let` because the second argument of `cons-stream` is evaluated lazily, which implies that after evaluating `(set! THE-ASSERTIONS (cons-stream assertion THE-ASSERTIONS))`, `THE-ASSERTIONS` is an infinite stream and all its elements are `assertion`.

Exercise 4.71

The use of explicit delays postpones the apparition of some infinite loops caused by the rules' evaluation and allows the interpreter to display some answers. For instance, with explicit delays, the query `(married Minnie ?who)` displays an infinite stream of answers, but with the simpler version of `simple-query` no answer is displayed because the infinite loop appears during the evaluation of the second argument of `stream-append`. In the same way, with Louis' version of `outranked-by` from exercise 4.64, with explicit delays the query `(outranked-by (Bitdiddle Ben) ?x)` displays an answer before going into a loop. With the simple version of `disjoin` no answer is displayed because the loop happens during the evaluation of the second argument of `interleave`.

Exercise 4.72

If one of the streams resulting from the evaluation of a disjunct or from the application of the mapped function to the stream of frames is infinite, with append no element of any ulterior stream would appear in the result. With `interleave` we are guaranteed that any element from a partial result stream will appear in the global merged stream eventually.

Exercise 4.73

Without `delay`, `flatten-stream` is called on the input stream's `cdr` before any element of the flattened stream can be displayed, so if the value of `(flatten-stream (stream-cdr stream))` is infinite (either because `stream` is infinite or because one of its elements is an infinite stream), `flatten-stream` will never return.

Exercise 4.74

a. The program can be completed simply by filtering out the empty streams and taking the first and only element of non-empty streams:

```
(define (simple-flatten stream)
  (stream-map stream-car
              (stream-filter (lambda (stream)
                               (not (stream-null? stream)))
                             stream)))
```

b. The query system's behavior does not change.

Exercise 4.75

The `uniquely-asserted` procedure can be defined as:

```
(define (uniquely-asserted query frame-stream)
  (simple-stream-flatmap
    (lambda (frame)
      (let ((result-stream (qeval (unique-query query)
                                  (singleton-stream frame))))
        (if (singleton-stream? result-stream)
          result-stream
          the-empty-stream)))
    frame-stream))

(put 'unique 'qeval uniquely-asserted)

(define (singleton-stream? stream)
  (and (not (stream-null? stream))
       (stream-null? (stream-cdr stream))))

(define (unique-query exps) (car exps))
```

The query that lists all people who supervise precisely one person is similar to the query that lists all jobs that are filled by only one person:

```
(and (supervisor ?x ?s) (unique (supervisor ?anyone ?s)))
```

Exercise 4.76

We can define a `merge-frames` procedure that merges two frames as indicated in the text, a `merge-frame-streams` procedure that applies `merge-frames` to each possible pair of frames from two input streams, and rewrite `conjoin` using these two procedures.

```
(define (merge-frames frame1 frame2)
  (cond ((eq? frame2 'failed) 'failed)
        ((null? frame1) frame2)
        (else
```

```
          (let ((binding (car frame1)))
            (merge-frames (cdr frame1)
                          (extend-if-possible (binding-variable binding)
                                              (binding-value binding)
                                              frame2))))))

(define (merge-frame-streams stream1 stream2)
  (stream-flatmap
    (lambda (frame)
      (stream-filter (lambda (f) (not (eq? f 'failed)))
                     (stream-map (lambda (frame2)
                                   (merge-frames frame frame2))
                                 stream1)))
    stream2))

(define (conjoin conjuncts frame-stream)
  (if (empty-conjunction? conjuncts)
    frame-stream
    (merge-frame-streams
      (qeval (first-conjunct conjuncts)
             frame-stream)
      (conjoin (rest-conjuncts conjuncts) frame-stream))))

(put 'and 'qeval conjoin)
```

There are problems caused by the separate evaluation of the clauses of the `and` however: we have seen previously that in some cases, the evaluation of the second clause works correctly only because some bindings have been provided by the evaluation of the previous clause, such as in the `outranked-by` rule, or when using `not` or `lisp-value`. This new implementation won't give the expected results in such cases, it works only when the clauses can be evaluated in any order.

Exercise 4.77

First, let's define a procedure to check whether a query contains an unbound variable in a given frame:

```
(define (has-unbound-var? query frame)
  (define (tree-walk exp)
    (cond ((var? exp)
           (let ((binding (binding-in-frame exp frame)))
             (if binding
               (tree-walk (binding-value binding))
               #t)))
          ((pair? exp)
           (or (has-unbound-var? (car exp) frame)
```

```
              (has-unbound-var? (cdr exp) frame)))
          (else #f)))
  (tree-walk query))
```

Then I modified the procedures `negate` and `lisp-value` so that they check if the query contains unbound variables. If so, they add a promise to filter to the frame. If not, they do the filtering directly. A promise is a pair consisting of a query and a predicate applying to a frame that returns true if the frame must be kept and false if it must be filtered out. It is added to the frame as a special binding. Since `not` and `lisp-value` queries are necessarily part of an `and` expression, I then modified `conjoin` so that it filters out the stream of frames produced by the evaluation of the first subquery if any promises are found. The disadvantage of this method is that if the frame is kept, the same promise will be checked multiple times.

```
(define (negate operands frame-stream)
  (define (keep? frame)
    (stream-null? (qeval (negated-query operands)
                         (singleton-stream frame))))
  (simple-stream-flatmap
    (lambda (frame)
      (filter-or-add-promise frame (negated-query operands) keep?))
    frame-stream))

(put 'not 'qeval negate)

(define (lisp-value call frame-stream)
  (define (keep? frame)
    (execute (instantiate
              call
              frame
              (lambda (v f)
                (error "Unknown pat var -- LISP-VALUE" v)))))
  (simple-stream-flatmap
    (lambda (frame)
      (filter-or-add-promise frame call keep?))
    frame-stream))

(put 'lisp-value 'qeval lisp-value)

(define (filter-or-add-promise frame query keep?)
  (cond ((has-unbound-var? query frame)
         (singleton-stream (add-promise frame query keep?)))
        ((keep? frame)
         (singleton-stream frame))
        (else the-empty-stream)))
```

```
(define (conjoin conjuncts frame-stream)
  (if (empty-conjunction? conjuncts)
    frame-stream
    (conjoin (rest-conjuncts conjuncts)
             (stream-filter keep? (qeval (first-conjunct conjuncts)
                                        frame-stream)))))

(define (keep? frame)
  (define (iter bindings)
    (if (null? bindings)
      #t
      (let ((binding (car bindings)))
        (if (eq? (binding-variable binding) '*promise*)
          (let ((promise (binding-value binding)))
            (if (has-unbound-var? (promise-query promise) frame)
              (iter (cdr bindings))
              (and ((promise-proc promise) frame)
                   (iter (cdr bindings)))))
          (iter (cdr bindings))))))
  (iter frame))

(define (add-promise frame query promise)
  (extend '*promise* (make-promise query promise) frame))

(define (make-promise query promise)
  (cons query promise))

(define (promise-query promise) (car promise))
(define (promise-proc promise) (cdr promise))
```

Exercise 4.78

To implement the query language as a nondeterministic program, we first replace the driver loop with two procedures: `assert!` to add assertions and rules to the database and `request` to run queries.

```
(define (assert! assertion)
  (add-rule-or-assertion! (query-syntax-process assertion))
  (display "Assertion added to data base.")
  'ok)

(define (request query)
  (let ((q (query-syntax-process query)))
    (instantiate q
                 (qeval q '())
```

```
                  (lambda (v f)
                    (contract-question-mark v)))))
```

All the procedures on frame streams now operate on a single frame. `Qeval` is unchanged except that the `frame-stream` argument will now be a single frame, but `simple-query`, `conjoin`, `disjoin`, `negate` and `lisp-value` have to be modified:

```
(define (simple-query query-pattern frame)
  (amb (find-assertions query-pattern frame)
       (apply-rules query-pattern frame)))

(define (conjoin conjuncts frame)
  (if (empty-conjunction? conjuncts)
    frame
    (conjoin (rest-conjuncts conjuncts)
             (qeval (first-conjunct conjuncts)
                    frame))))

(define (disjoin disjuncts frame)
  (require (not (empty-disjunction? disjuncts)))
  (ramb (qeval (first-disjunct disjuncts) frame)
        (disjoin (rest-disjuncts disjuncts) frame)))

(define (negate operands frame)
  (require-fail (qeval (negated-query operands) frame))
  frame)

(define (lisp-value call frame)
  (require (execute
             (instantiate
               call
               frame
               (lambda (v f)
                 (error "Unknown pat var -- LISP-VALUE" v)))))
  frame)
```

For `negate`, we need to succeed if the evaluation of the negated query fails without producing any result, and I found no other way to do that than to define a new special form `require-fail` that succeeds (with the value `true`) if its argument fails:

```
(define (require-fail? exp) (tagged-list? exp 'require-fail))

(define (require-fail-test exp) (cadr exp))

(define (analyze-require-fail exp)
```

```
  (let ((pproc (analyze (require-fail-test exp))))
    (lambda (env succeed fail)
      (pproc env
             (lambda (val fail2)
               (fail))
             (lambda ()
               (succeed #t fail))))))
```

The procedures used for pattern matching and unification must be adapted as well. By using `amb` instead of returning the `failed` symbol in `pattern-match` and `unify-match`, we can simplify the procedures a little since it's not necessary to check for the `failed` symbol anymore.

```
(define (find-assertions pattern frame)
  (pattern-match pattern (an-element-of (fetch-assertions pattern)) frame))

(define (pattern-match pat dat frame)
  (cond ((equal? pat dat) frame)
        ((var? pat) (extend-if-consistent pat dat frame))
        ((and (pair? pat) (pair? dat))
         (pattern-match (cdr pat)
                        (cdr dat)
                        (pattern-match (car pat)
                                       (car dat)
                                       frame)))
        (else (amb))))

(define (apply-rules pattern frame)
  (apply-a-rule (an-element-of (fetch-rules pattern)) pattern frame))

(define (apply-a-rule rule query-pattern query-frame)
  (let ((clean-rule (rename-variables-in rule)))
    (qeval (rule-body clean-rule)
           (unify-match query-pattern
                        (conclusion clean-rule)
                        query-frame))))

(define (unify-match p1 p2 frame)
  (cond ((equal? p1 p2) frame)
        ((var? p1) (extend-if-possible p1 p2 frame))
        ((var? p2) (extend-if-possible p2 p1 frame))
        ((and (pair? p1) (pair? p2))
         (unify-match (cdr p1)
                      (cdr p2)
                      (unify-match (car p1)
                                   (car p2)
```

```
                                   frame)))
        (else (amb))))

(define (extend-if-possible var val frame)
  (let ((binding (binding-in-frame var frame)))
    (cond (binding
            (unify-match (binding-value binding) val frame))
          ((var? val)
           (let ((binding (binding-in-frame val frame)))
             (if binding
               (unify-match var (binding-value binding) frame)
               (extend var val frame))))
          (else
            (require (not (depends-on? val var frame)))
            (extend var val frame)))))
```

I also rewrote the code for maintaining the data base so that it uses lists instead of streams (not included here).

There are differences in the answers' order due to my use of `ramb` in `disjoin` where the stream-based version used `interleave-delayed`.

The main differences regard infinite loops. Because I used `ramb` in `disjoin`, elements from all the disjuncts will appear eventually but it's impossible to predict when. If the computation of one element causes an infinite loop, it may appear at the start or after some answers have been displayed. For instance, with Louis' version of `outranked-by` from exercise 4.64, sometimes the query `(outranked-by (Bitdiddle Ben) ?x)` loops without displaying anything, sometimes it answers first. But most of the time `(outranked-by ?x ?y)` displays several answers before going into a loop, whereas the stream-based version always displayed only the first answer.

With the query `(last-pair ?x (3))` from exercise 4.62, the successive results displayed show an increment of one only in the ids of the unbound variables, where the increment was of 4 in the stream version. The first results are:

```
(last-pair (3) (3))
(last-pair (?u-1 3) (3))
(last-pair (?u-1 ?u-2 3) (3))
(last-pair (?u-1 ?u-2 ?u-3 3) (3))
(last-pair (?u-1 ?u-2 ?u-3 ?u-4 3) (3))
```

The reason is that the rule-counter is decreased when the evaluator backtracks after a rule application fails (there are 4 rules in the test database).

5 Computing with Register Machines

5.1 Designing Register Machines

Exercise 5.1

The data-path and the controller diagrams for the iterative factorial machine are shown on figure 5.1.

5.1.1 A Language for Describing Register Machines

Exercise 5.2

Anticipating on the next section to use the register-machine simulator, we can define the iterative factorial machine of exercise 5.1 as:

```
(define fact-machine
  (make-machine
    '(n product counter)
    (list (list '> >) (list '* *) (list '+ +))
    '((assign product (const 1))
      (assign counter (const 1))
     test-counter
      (test (op >) (reg counter) (reg n))
      (branch (label fact-done))
      (assign product (op *) (reg product) (reg counter))
      (assign counter (op +) (reg counter) (const 1))
      (goto (label test-counter))
     fact-done)))
```

5.1.2 Abstraction in Machine Design

Exercise 5.3

Using the simulator again, the first version of the register machines can be defined as:

```
(define sqrt-simple
  (make-machine
    '(guess x)
    (list (list 'good-enough? good-enough?) (list 'improve improve))
    '((assign guess (const 1.0))
    test-good-enough
      (test (op good-enough?) (reg guess) (reg x))
```

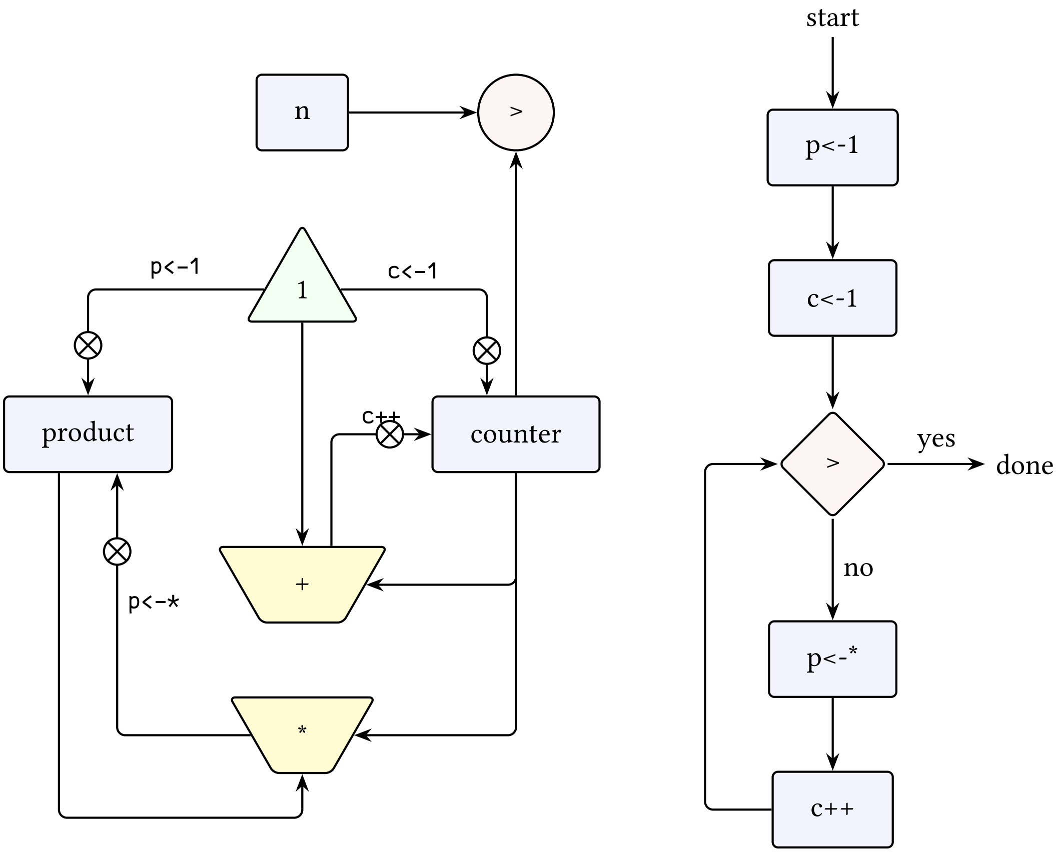

Figure 5.1: The data-path and controller diagrams for the iterative factorial machine.

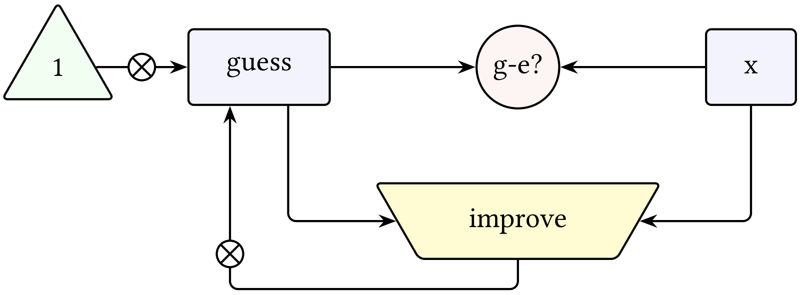

Figure 5.2: The data-path diagram for the square root machine using complex primitive operations.

```
    (branch (label sqrt-done))
    (assign guess (op improve) (reg guess) (reg x))
    (goto (label test-good-enough))
  sqrt-done)))
```

and the second version as:

```
(define sqrt-full
  (make-machine
    '(guess tmp x)
    (list (list '+ +) (list '- -) (list '* *) (list '/ /) (list '< <))
    '((assign guess (const 1.0))
    test-good-enough
      (assign tmp (op *) (reg guess) (reg guess))
      (assign tmp (op -) (reg tmp) (reg x))
      (test (op <) (const 0) (reg tmp))
      (branch (label after-abs))
      (assign tmp (op -) (reg tmp))
    after-abs
      (test (op <) (reg tmp) (const 0.001))
      (branch (label sqrt-done))
      (assign tmp (op /) (reg x) (reg guess))
      (assign guess (op +) (reg guess) (reg tmp))
      (assign guess (op /) (reg guess) (const 2))
      (goto (label test-good-enough))
    sqrt-done)))
```

The data-path diagrams are shown on figures 5.2 and 5.3 respectively.

5.1.3 Subroutines

This subsection contains no exercises.

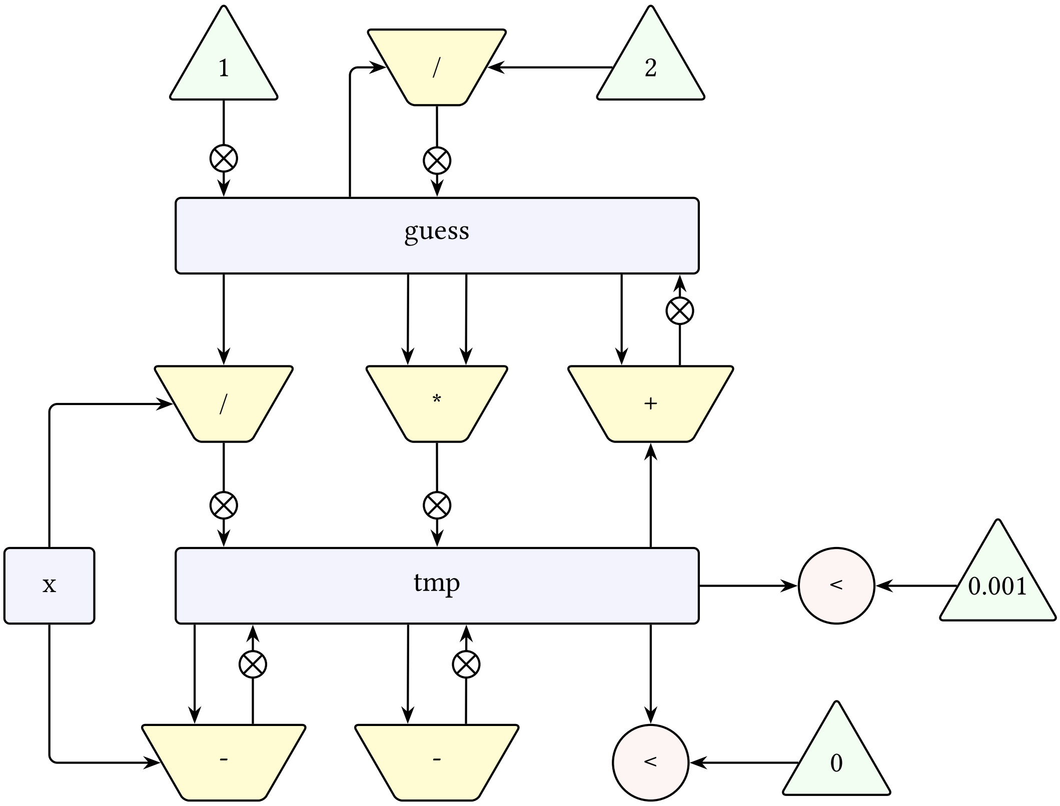

Figure 5.3: The data-path diagram for the square root machine using only basic primitive operations.

5.1.4 Using a Stack to Implement Recursion

Exercise 5.4

a. The recursive exponentiation machine can be defined as follows. The corresponding data-path diagram is shown on figure 5.4.

```
(define expt-rec
  (make-machine
    '(b n val continue)
    (list (list '= =) (list '- -) (list '* *))
    '((assign continue (label expt-done))
    expt-loop
      (test (op =) (reg n) (const 0))
      (branch (label base-case))
      (save continue)
      (assign continue (label after-expt))
      (save n)
      (assign n (op -) (reg n) (const 1))
      (goto (label expt-loop))
    after-expt
      (restore n)
      (restore continue)
      (assign val (op *) (reg b) (reg val))
      (goto (reg continue))
    base-case
      (assign val (const 1))
      (goto (reg continue))
    expt-done)))
```

b. The iterative exponentiation machine can be defined as follows. The corresponding data-path diagram is shown on figure 5.5.

```
(define expt-iter
  (make-machine
    '(b n counter product)
    (list (list '= =) (list '- -) (list '* *))
    '((assign counter (reg n))
      (assign product (const 1))
    expt-loop
      (test (op =) (reg counter) (const 0))
      (branch (label expt-done))
      (assign counter (op -) (reg counter) (const 1))
      (assign product (op *) (reg b) (reg product))
      (goto (label expt-loop))
    expt-done)))
```

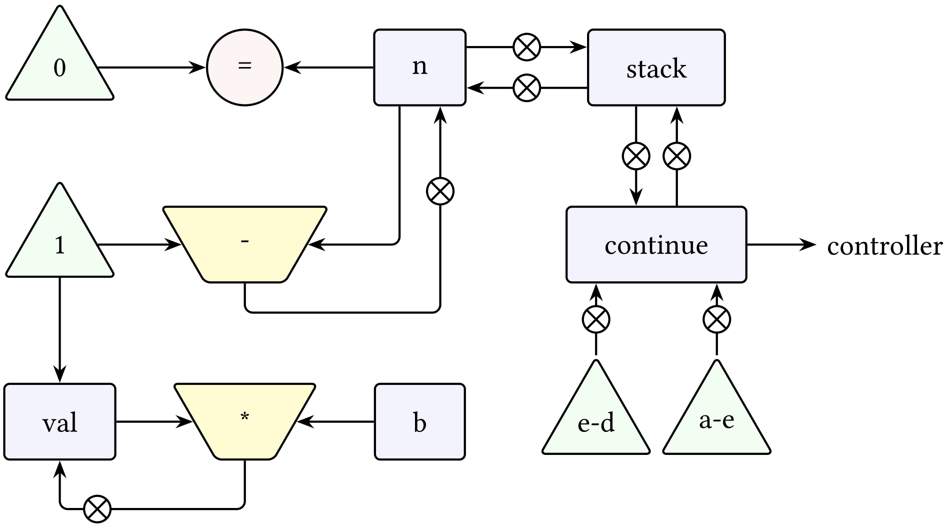

Figure 5.4: The data-path diagram for the recursive exponentiation machine.

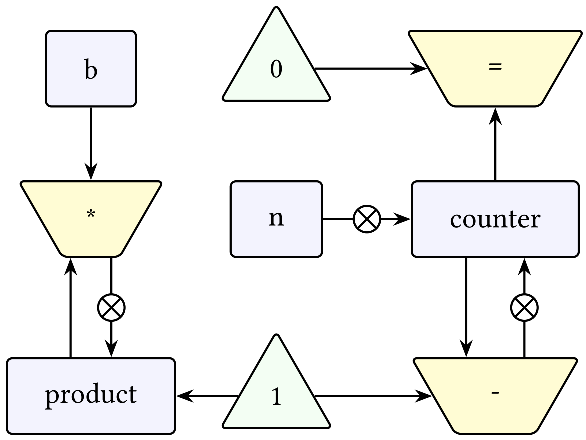

Figure 5.5: The data-path diagram for the iterative exponentiation machine.

Exercise 5.5

The following table lists the instructions evaluated during the simulation of factorial 3, with their effect on the values of the registers `n`, `val`, and `continue` and on the stack.

Instruction	**n**	**val**	**continue**	**stack**
(assign cont (label fact-done))	3	*unassigned*	fact-done	()
(test (op =) ...) -> false				
(branch (label base-case))				
(save continue)				(fact-done)
(save n)				(3 fact-done)
(assign n (op -) ...)	2			
(assign cont (label after-fact))			after-fact	
(goto (label fact-loop))				
(test (op =) ...) -> false				
(branch (label base-case))				
(save cont)				(after-fact 3 fact-done)
(save n)				(2 after-fact 3 fact-done)
(assign n (op -) ...)	1			
(assign cont (label after-fact))			after-fact	
(goto (label fact-loop))				
(test (op =) ...) -> true				
(branch (label base-case))				
(assign val (const 1))		1		
(goto (reg cont)) -> after-fact				
(restore n)	2			(after-fact 3 fact-done)
(restore cont)			after-fact	(3 fact-done)
(assign val (op *) ...)		2		
(goto (reg cont)) -> after-fact				
(restore n)	3			(fact-done)
(restore cont)			fact-done	()
(assign val (op *) ...)		6		
(goto (reg cont)) -> fact-done				

The following table lists the instructions evaluated during the simulation of fibonacci 3, with their effect on the values of the registers `n`, `val`, and `continue` and on the stack.

Instruction	**n**	**val**	**continue**	**stack**
(assign cont (label fib-done)	3	*unassigned*	fib-done	()
(test (op <) ...) -> false				
(branch (label imm-answer))				
(save cont)				(fib-done)
(assign cont (label after-fib-n-1))			after-fib-n-1	
(save n)				(3 fib-done)

Instruction	**n**	**val**	**continue**	**stack**
(assign n (op -) ...1)	2			
(goto (label fib-loop))				
(test (op <) ...) -> false				
(branch (label imm-answer))				
(save cont)				(after-fib-n-1 3 fib-done)
(assign cont (label after-fib-n-1))			after-fib-n-1	
(save n)				(2 after-fib-n-1 3 fib-done)
(assign n (op -) ...1)	1			
(goto (label fib-loop))				
(test (op <) ...) -> true				
(branch (label imm-answer))				
(assign val (reg n))		1		
(goto (reg cont)) -> after-fib-n-1				
(restore n)	2			(after-fib-n-1 3 fib-done)
(restore cont)			after-fib-n-1	(3 fib-done)
(assign n (op -) ...2)	0			
(save cont)				(after-fib-n-1 3 fib-done)
(assign cont (label after-fib-n-2))			after-fib-n-2	
(save val)				(1 after-fib-n-1 3 fib-done)
(goto (label fib-loop))				
(test (op <) ...) -> true				
(branch (label imm-answer))				
(assign val (reg n))		0		
(goto (reg cont)) -> after-fib-n-2				
(assign n (reg val))	0			
(restore val)		1		(after-fib-n-1 3 fib-done)
(restore cont)			after-fib-n-1	(3 fib-done)
(assign val (op +) ...)		1		
(goto (reg cont)) -> after-fib-n-1				
(restore n)	3			(fib-done)
(restore cont)			fib-done	()
(assign n (op -) ... 2)	1			
(save cont)				(fib-done)
(assign cont (label after-fib-n-2))			after-fib-n-2	
(save val)				(1 fib-done)
(goto (label fib-loop))				
(test (op <) ...) -> true				
(branch (label imm-answer))				
(assign val (reg n))		1		
(goto (reg cont)) -> after-fib-n-2				
(assign n (reg val))	1			

Instruction	n	val	continue	stack
(restore val)		1		(fib-done)
(restore cont)			fib-done	()
(assign val (op +) ...)		2		
(goto (reg cont)) -> fib-done				

Exercise 5.6

In `after-fib-n-1`, the instructions `(restore continue)` and `(save continue)` can be removed because no change is done to the `continue` register or to the stack between them.

5.1.5 Instruction Summary

This subsection contains no exercises.

5.2 A Register-Machine Simulator

Exercise 5.7

Already done while doing exercise 5.4.

5.2.1 The Machine Model

This subsection contains no exercises.

5.2.2 The Assembler

Exercise 5.8

With the simulator as written, the contents of register `a` will be 3: two labels with the name `here` are present in the list of labels, but the one corresponding to the first location comes first and is the one returned by `lookup-label`.

We can return an error if the same label name is used to indicate two different locations by modifying `extract-labels` as follows:

```
(define (extract-labels text receive)
  (if (null? text)
    (receive '() '())
    (extract-labels
      (cdr text)
      (lambda (insts labels)
        (let ((next-inst (car text)))
          (if (symbol? next-inst)
            (if (assoc next-inst labels)
              (error "Label defined multiple times:" next-inst)
              (receive insts
                       (cons (make-label-entry next-inst
                                               insts)
```

```
                     labels)))
         (receive (cons (make-instruction next-inst)
                        insts)
                  labels)))))))
```

5.2.3 Generating Execution Procedures for Instructions

Exercise 5.9

We can forbid labels as arguments to operations by adding a test to the `make-operation-exp` procedure:

```
(define (make-operation-exp exp machine labels operations)
  (let ((op (lookup-prim (operation-exp-op exp) operations))
        (aprocs (map (lambda (e)
                       (if (label-exp? e)
                         (error "Labels are not allowed as operation arguments" e)
                         (make-primitive-exp e machine labels)))
                     (operation-exp-operands exp))))
    (lambda ()
      (apply op (map (lambda (p) (p)) aprocs)))))
```

Exercise 5.10

I only modified the syntax of `op`, for instance `(assign <reg-name> (op <op-name>) <args>)` becomes `(assign <reg-name> op <op-name> <args>)`:

```
(define (operation-exp? exp)
  (tagged-list? exp 'op))

(define (operation-exp-op operation-exp)
  (cadr operation-exp))

(define (operation-exp-operands operation-exp)
  (cddr operation-exp))
```

Exercise 5.11

a. After the label `after-fib-n-2`, the last value placed on the stack is $\text{Fib}(n-1)$, and the `val` register contains $\text{Fib}(n-2)$. The Fibonacci machine places $\text{Fib}(n-2)$ in the `n` register before restoring the last saved value in the `val` register. We can instead restore the last saved value in the `n` register and remove an assignment operation. So the lines:

   ```
   (assign n (reg val))
   (restore val)
   ```

   become:

   ```
   (restore n)
   ```

b. We need to change `make-save` to put the register name on the stack, and `make-restore` to check that the original register corresponds to the target register:

```
(define (make-save inst machine stack pc)
  (let* ((reg-name (stack-inst-reg-name inst))
         (reg (get-register machine reg-name)))
    (lambda ()
      (push stack (make-stack-entry reg-name (get-contents reg)))
      (advance-pc pc))))

(define (make-restore inst machine stack pc)
  (let* ((reg-name (stack-inst-reg-name inst))
         (reg (get-register machine reg-name)))
    (lambda ()
      (let ((entry (pop stack)))
        (if (eq? (stack-entry-name entry) reg-name)
          (begin
            (set-contents! reg (pop stack))
            (advance-pc pc))
          (error "Last saved value does not come from register"
                 reg-name
                 'original 'register:
                 (stack-entry-name entry)))))))

(define (make-stack-entry name contents)
  (cons name contents))
(define (stack-entry-name entry)
  (car entry))
(define (stack-entry-contents entry)
  (cdr entry))
```

c. I chose to modify `make-register` to directly add a stack to each register:

```
(define (make-register name)
  (let ((contents '*unassigned*)
        (stack (make-stack)))
    (define (dispatch message)
      (cond ((eq? message 'get) contents)
            ((eq? message 'set)
             (lambda (value)
               (set! contents value)))
            ((eq? message 'stack) stack)
            ((eq? message 'initialize-stack)
             (stack 'initialize))
            (else
              (error "Unknown request: REGISTER" message))))
    dispatch))
```

```
(define (make-save inst machine pc)
  (let* ((reg-name (stack-inst-reg-name inst))
         (reg (get-register machine reg-name))
         (stack (reg 'stack)))
    (lambda ()
      (push stack (get-contents reg))
      (advance-pc pc))))

(define (make-restore inst machine pc)
  (let* ((reg-name (stack-inst-reg-name inst))
         (reg (get-register machine reg-name))
         (stack (reg 'stack)))
    (lambda ()
      (set-contents! reg (pop stack))
      (advance-pc pc))))
```

Some other procedures must be modified as well: no stack is needed anymore in `make-new-machine`, and the `stack` parameter can be removed from `make-execution-procedure`. As indicated in the text, the `initialize-stack` operation should initialize all the register stacks, which can be done by replacing the original definition of `the-ops` in `make-new-machine` with:

```
(the-ops
  (list (list 'initialize-stack
              (lambda ()
                (for-each
                  (lambda (reg) (reg 'initialize-stack))
                  (map cadr register-table))))))
```

Exercise 5.12

Since we need ordered sets, we first reuse a slightly modified version of the representation of sets as ordered lists defined in section 2.3.3:

```
(define (element-of-set? x set smaller?)
  (cond ((null? set) #f)
        ((equal? (car set) x) #t)
        ((smaller? x (car set)) #t)
        (else
          (element-of-set? x (cdr set) smaller?))))

(define (adjoin-set x set smaller?)
  (if (null? set)
    (list x)
    (let ((first (car set)))
         (cond ((equal? x first) set)
               ((smaller? x first) (cons x set))
```

```
                (else (cons first (adjoin-set x (cdr set) smaller?)))))))

(define (smaller? obj1 obj2)
  (string-ci<? (object->string obj1) (object->string obj2)))
```

To store the list of all instructions, we change `assemble` and `extract-label` to build the list gradually as the instructions are processed and then add it to the machine:

```
(define (assemble controller-text machine)
  (extract-labels controller-text
                  (lambda (insts labels insts-list)
                    (update-insts! insts labels machine)
                    ((machine 'install-instructions-list) insts-list)
                    insts)))

(define (extract-labels text receive)
  (if (null? text)
    (receive '() '() '())
    (extract-labels
      (cdr text)
      (lambda (insts labels insts-list)
        (let ((next-inst (car text)))
          (if (symbol? next-inst)
            (receive insts
                     (cons (make-label-entry next-inst
                                             insts)
                           labels)
                     insts-list)
            (receive (cons (make-instruction next-inst)
                           insts)
                     labels
                     (adjoin-set next-inst insts-list smaller?))))))))
```

We also modify `make-new-machine` to define `instructions-list` as the empty list at first and add new cases to the `dispatch` procedure:

```
((eq? message 'instructions-list) instructions-list)
((eq? message 'install-instructions-list)
 (lambda (insts)
   (set! instructions-list insts)))
```

To store the lists of the registers used to hold entry points and of the registers that are `saved` or `restored`, we change `make-new-machine` to define those lists, as well as procedures to add an element to them and new messages to access them:

```
(define (add-entry-point reg-name)
  (set! entry-points
```

```
    (adjoin-set reg-name entry-points smaller?)))

(define (add-saved-reg reg-name)
  (set! saved-regs
    (adjoin-set reg-name saved-regs smaller?)))

((eq? message 'add-entry-point) add-entry-point)
((eq? message 'entry-points) entry-points)
((eq? message 'add-saved-reg) add-saved-reg)
((eq? message 'saved-regs) saved-regs)
```

We then modify `make-goto`, `make-save` and `make-restore` to add elements to the lists:

```
(define (make-goto inst machine labels pc)
  (let ((dest (goto-dest inst)))
    (cond ((label-exp? dest)
           (let ((insts (lookup-label labels (label-exp-label dest))))
             (lambda ()
               (set-contents! pc insts))))
          ((register-exp? dest)
           (let ((reg (get-register machine (register-exp-reg dest))))
             ((machine 'add-entry-point) (register-exp-reg dest))
             (lambda ()
               (set-contents! pc (get-contents reg)))))
          (else
            (error "Bad GOTO instruction ASSEMBLE" inst)))))

(define (make-save inst machine stack pc)
  (let ((reg (get-register machine
                           (stack-inst-reg-name inst))))
    ((machine 'add-saved-reg) (stack-inst-reg-name inst))
    (lambda ()
      (push stack (get-contents reg))
      (advance-pc pc))))

(define (make-restore inst machine stack pc)
  (let ((reg (get-register machine
                           (stack-inst-reg-name inst))))
    ((machine 'add-saved-reg) (stack-inst-reg-name inst))
    (lambda ()
      (set-contents! reg (pop stack))
      (advance-pc pc))))
```

To store the sources from which each register is assigned, we change `make-register` to store this information in each register, and `make-assign` to add the sources to the target register. We also define a `get-sources` procedure to access this information more easily:

```
(define (make-register name)
  (let ((contents '*unassigned*)
        (sources '()))
    (define (add-source source)
      (set! sources
        (adjoin-set source sources smaller?)))
    (define (dispatch message)
      (cond ((eq? message 'get) contents)
            ((eq? message 'set)
             (lambda (value)
               (set! contents value)))
            ((eq? message 'sources) sources)
            ((eq? message 'add-source) add-source)
            (else
              (error "Unknown request: REGISTER" message))))
    dispatch))

(define (make-assign inst machine labels operations pc)
  (let ((target (get-register machine (assign-reg-name inst)))
        (value-exp (assign-value-exp inst)))
    ((target 'add-source) value-exp)
    (let ((value-proc
            (if (operation-exp? value-exp)
              (make-operation-exp value-exp machine labels operations)
              (make-primitive-exp (car value-exp) machine labels))))
      (lambda ()
        (set-contents! target (value-proc))
        (advance-pc pc)))))

(define (get-sources machine reg-name)
  ((get-register machine reg-name) 'sources))
```

Exercise 5.13

We change `make-machine` so it does not take a list of registers as an argument:

```
(define (make-machine ops controller-text)
  (let ((machine (make-new-machine)))
    ((machine 'install-operations) ops)
    ((machine 'install-instruction-sequence)
     (assemble controller-text machine))
    machine))
```

Then, we change the `lookup-register` procedure in `make-new-machine` so that it allocates a new register if no register with the given name exists. I also modified `allocate-register` to return the newly allocated register to simplify the procedures. The dispatch clause for `allocate-register` can be removed.

```
(define (allocate-register name)
  (if (assoc name register-table)
    (error "Multiply defined register: " name)
    (let ((new-reg (make-register name)))
      (set! register-table (cons (list name new-reg)
                                 register-table))
      new-reg)))

(define (lookup-register name)
  (let ((val (assoc name register-table)))
    (if val
      (cadr val)
      (allocate-register name))))
```

5.2.4 Monitoring Machine Performance

Exercise 5.14

The recursive factorial machine can be modified as shown below to initialize the stack and print the statistics.

```
(define fact-machine-rec
  (make-machine
    '(n val continue)
    (list (list '= =) (list '- -) (list '* *)
          (list 'read read) (list 'display display))
    '(start
      (perform (op initialize-stack))
      (assign n (op read))
      (assign continue (label fact-done))
    fact-loop
      (test (op =) (reg n) (const 1))
      (branch (label base-case))
      (save continue)
      (save n)
      (assign n (op -) (reg n) (const 1))
      (assign continue (label after-fact))
      (goto (label fact-loop))
    after-fact
      (restore n)
      (restore continue)
      (assign val (op *) (reg val) (reg n))
      (goto (reg continue))
    base-case
      (assign val (const 1))
      (goto (reg continue))
```

```
  fact-done
    (perform (op display) (reg val))
    (perform (op print-stack-statistics))
    (goto (label start)))))
```

The total number of push operations and the maximum stack depth used in computing $n!$ are both equal to $2(n-1)$: the `n` and `continue` registers are each saved $n-1$ times, then all the elements of the stack are popped.