```
        (iter (cdr things)
              (cons (square (car things))
                    answer))))
  (iter items nil))
```

Unfortunately, defining `square-list` this way produces the answer list in the reverse order of the one desired. Why?

Louis then tries to fix his bug by interchanging the arguments to `cons`:

```
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons answer
                    (square (car things))))))
  (iter items nil))
```

This doesn't work either. Explain.

Exercise 2.23. The procedure `for-each` is similar to `map`. It takes as arguments a procedure and a list of elements. However, rather than forming a list of the results, `for-each` just applies the procedure to each of the elements in turn, from left to right. The values returned by applying the procedure to the elements are not used at all -- `for-each` is used with procedures that perform an action, such as printing. For example,

```
(for-each (lambda (x) (newline) (display x))
          (list 57 321 88))
57
321
88
```

The value returned by the call to `for-each` (not illustrated above) can be something arbitrary, such as true. Give an implementation of `for-each`.

2.2.2 Hierarchical Structures

The representation of sequences in terms of lists generalizes naturally to represent sequences whose elements may themselves be sequences. For example, we can regard the object `((1 2) 3 4)` constructed by

```
(cons (list 1 2) (list 3 4))
```

as a list of three items, the first of which is itself a list, `(1 2)`. Indeed, this is suggested by the form in which the result is printed by the interpreter. Figure 2.5 shows the representation of this structure in terms of pairs.

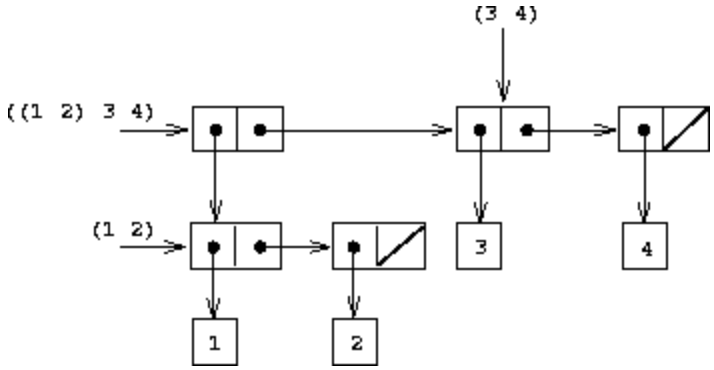

Figure 2.5: Structure formed by `(cons (list 1 2) (list 3 4))`.

Another way to think of sequences whose elements are sequences is as *trees*. The elements of the sequence are the branches of the tree, and elements that are themselves sequences are subtrees. Figure 2.6 shows the structure in figure 2.5 viewed as a tree.

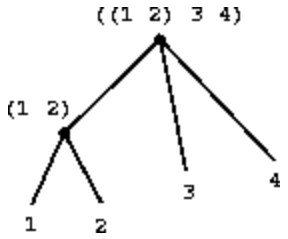

Figure 2.6: The list structure in figure 2.5 viewed as a tree.

Recursion is a natural tool for dealing with tree structures, since we can often reduce operations on trees to operations on their branches, which reduce in turn to operations on the branches of the branches, and so on, until we reach the leaves of the tree. As an example, compare the `length` procedure of section 2.2.1 with the `count-leaves` procedure, which returns the total number of leaves of a tree:

```
(define x (cons (list 1 2) (list 3 4)))

(length x)
3
(count-leaves x)
4

(list x x)
(((1 2) 3 4) ((1 2) 3 4))

(length (list x x))
2

(count-leaves (list x x))
8
```

To implement `count-leaves`, recall the recursive plan for computing `length`:

- `Length` of a list `x` is 1 plus `length` of the `cdr` of `x`.
- `Length` of the empty list is 0.

`Count-leaves` is similar. The value for the empty list is the same:

- `Count-leaves` of the empty list is 0.

But in the reduction step, where we strip off the `car` of the list, we must take into account that the `car` may itself be a tree whose leaves we need to count. Thus, the appropriate reduction step is

- `Count-leaves` of a tree `x` is `count-leaves` of the `car` of `x` plus `count-leaves` of the `cdr` of `x`.

Finally, by taking `car`s we reach actual leaves, so we need another base case:

- `Count-leaves` of a leaf is 1.

To aid in writing recursive procedures on trees, Scheme provides the primitive predicate `pair?`, which tests whether its argument is a pair. Here is the complete procedure:[13]

```
(define (count-leaves x)
  (cond ((null? x) 0)
        ((not (pair? x)) 1)
        (else (+ (count-leaves (car x))
                 (count-leaves (cdr x))))))
```

Exercise 2.24. Suppose we evaluate the expression `(list 1 (list 2 (list 3 4)))`. Give the result printed by the interpreter, the corresponding box-and-pointer structure, and the interpretation of this as a tree (as in figure 2.6).

Exercise 2.25. Give combinations of `car`s and `cdr`s that will pick 7 from each of the following lists:

```
(1 3 (5 7) 9)

((7))

(1 (2 (3 (4 (5 (6 7))))))
```

Exercise 2.26. Suppose we define `x` and `y` to be two lists:

```
(define x (list 1 2 3))
(define y (list 4 5 6))
```

What result is printed by the interpreter in response to evaluating each of the following expressions:

```
(append x y)

(cons x y)

(list x y)
```

Exercise 2.27. Modify your `reverse` procedure of exercise 2.18 to produce a `deep-reverse` procedure that takes a list as argument and returns as its value the list with its elements reversed and with all sublists deep-reversed as well. For example,

```
(define x (list (list 1 2) (list 3 4)))

x
((1 2) (3 4))

(reverse x)
((3 4) (1 2))

(deep-reverse x)
((4 3) (2 1))
```

Exercise 2.28. Write a procedure `fringe` that takes as argument a tree (represented as a list) and returns a list whose elements are all the leaves of the tree arranged in left-to-right order. For example,

```
(define x (list (list 1 2) (list 3 4)))

(fringe x)
(1 2 3 4)

(fringe (list x x))
(1 2 3 4 1 2 3 4)
```

Exercise 2.29. A binary mobile consists of two branches, a left branch and a right branch. Each branch is a rod of a certain length, from which hangs either a weight or another binary mobile. We can represent a binary mobile using compound data by constructing it from two branches (for example, using `list`):

```
(define (make-mobile left right)
  (list left right))
```

A branch is constructed from a `length` (which must be a number) together with a `structure`, which may be either a number (representing a simple weight) or another mobile:

```
(define (make-branch length structure)
  (list length structure))
```

a. Write the corresponding selectors `left-branch` and `right-branch`, which return the branches of a mobile, and `branch-length` and `branch-structure`, which return the components of a branch.

b. Using your selectors, define a procedure `total-weight` that returns the total weight of a mobile.

c. A mobile is said to be *balanced* if the torque applied by its top-left branch is equal to that applied by its top-right branch (that is, if the length of the left rod multiplied by the weight hanging from that rod is equal to the corresponding product for the right side) and if each of the submobiles hanging off its branches is balanced. Design a predicate that tests whether a binary mobile is balanced.

d. Suppose we change the representation of mobiles so that the constructors are

```
(define (make-mobile left right)
  (cons left right))
(define (make-branch length structure)
  (cons length structure))
```

How much do you need to change your programs to convert to the new representation?

Mapping over trees

Just as `map` is a powerful abstraction for dealing with sequences, `map` together with recursion is a powerful abstraction for dealing with trees. For instance, the `scale-tree` procedure, analogous to `scale-list` of section 2.2.1, takes as arguments a numeric factor and a tree whose leaves are numbers. It returns a tree of the same shape, where each number is multiplied by the factor. The recursive plan for `scale-tree` is similar to the one for `count-leaves`:

```
(define (scale-tree tree factor)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (* tree factor))
```

```
        (else (cons (scale-tree (car tree) factor)
                    (scale-tree (cdr tree) factor)))))
(scale-tree (list 1 (list 2 (list 3 4) 5) (list 6 7))
            10)
(10 (20 (30 40) 50) (60 70))
```

Another way to implement `scale-tree` is to regard the tree as a sequence of sub-trees and use `map`. We map over the sequence, scaling each sub-tree in turn, and return the list of results. In the base case, where the tree is a leaf, we simply multiply by the factor:

```
(define (scale-tree tree factor)
  (map (lambda (sub-tree)
         (if (pair? sub-tree)
             (scale-tree sub-tree factor)
             (* sub-tree factor)))
       tree))
```

Many tree operations can be implemented by similar combinations of sequence operations and recursion.

Exercise 2.30. Define a procedure `square-tree` analogous to the `square-list` procedure of exercise 2.21. That is, `square-list` should behave as follows:

```
(square-tree
 (list 1
       (list 2 (list 3 4) 5)
       (list 6 7)))
(1 (4 (9 16) 25) (36 49))
```

Define `square-tree` both directly (i.e., without using any higher-order procedures) and also by using `map` and recursion.

Exercise 2.31. Abstract your answer to exercise 2.30 to produce a procedure `tree-map` with the property that `square-tree` could be defined as

```
(define (square-tree tree) (tree-map square tree))
```

Exercise 2.32. We can represent a set as a list of distinct elements, and we can represent the set of all subsets of the set as a list of lists. For example, if the set is `(1 2 3)`, then the set of all subsets is `(() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))`. Complete the following definition of a procedure that generates the set of subsets of a set and give a clear explanation of why it works:

```
(define (subsets s)
  (if (null? s)
      (list nil)
      (let ((rest (subsets (cdr s))))
        (append rest (map <??> rest)))))
```

2.2.3 Sequences as Conventional Interfaces

In working with compound data, we've stressed how data abstraction permits us to design programs without becoming enmeshed in the details of data representations, and how abstraction preserves for us the flexibility to experiment with alternative representations. In this section, we introduce another powerful design principle for working with data structures -- the use of *conventional interfaces*.

In section 1.3 we saw how program abstractions, implemented as higher-order procedures, can capture common patterns in programs that deal with numerical data. Our ability to formulate analogous operations for working with compound data depends crucially on the style in which

we manipulate our data structures. Consider, for example, the following procedure, analogous to the `count-leaves` procedure of section 2.2.2, which takes a tree as argument and computes the sum of the squares of the leaves that are odd:

```
(define (sum-odd-squares tree)
  (cond ((null? tree) 0)
        ((not (pair? tree))
         (if (odd? tree) (square tree) 0))
        (else (+ (sum-odd-squares (car tree))
                 (sum-odd-squares (cdr tree))))))
```

On the surface, this procedure is very different from the following one, which constructs a list of all the even Fibonacci numbers $Fib(k)$, where k is less than or equal to a given integer n:

```
(define (even-fibs n)
  (define (next k)
    (if (> k n)
        nil
        (let ((f (fib k)))
          (if (even? f)
              (cons f (next (+ k 1)))
              (next (+ k 1))))))
  (next 0))
```

Despite the fact that these two procedures are structurally very different, a more abstract description of the two computations reveals a great deal of similarity. The first program

- enumerates the leaves of a tree;
- filters them, selecting the odd ones;
- squares each of the selected ones; and
- accumulates the results using +, starting with 0.

The second program

- enumerates the integers from 0 to n;
- computes the Fibonacci number for each integer;
- filters them, selecting the even ones; and
- accumulates the results using `cons`, starting with the empty list.

A signal-processing engineer would find it natural to conceptualize these processes in terms of signals flowing through a cascade of stages, each of which implements part of the program plan, as shown in figure 2.7. In `sum-odd-squares`, we begin with an *enumerator*, which generates a ``signal'' consisting of the leaves of a given tree. This signal is passed through a *filter*, which eliminates all but the odd elements. The resulting signal is in turn passed through a *map*, which is a ``transducer'' that applies the `square` procedure to each element. The output of the map is then fed to an *accumulator*, which combines the elements using +, starting from an initial 0. The plan for `even-fibs` is analogous.

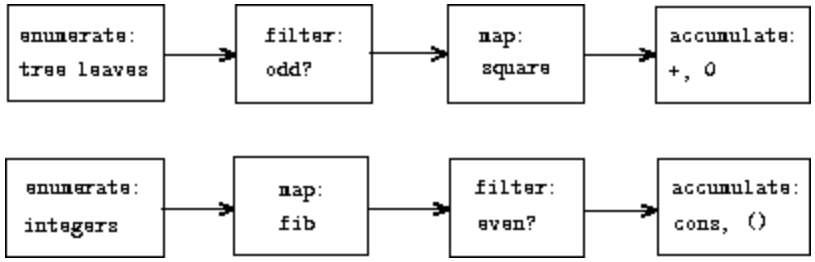

Figure 2.7: The signal-flow plans for the procedures `sum-odd-squares` (top) and `even-fibs` (bottom) reveal the commonality between the two programs.

Unfortunately, the two procedure definitions above fail to exhibit this signal-flow structure. For instance, if we examine the `sum-odd-squares` procedure, we find that the enumeration is implemented partly by the `null?` and `pair?` tests and partly by the tree-recursive structure of the procedure. Similarly, the accumulation is found partly in the tests and partly in the addition used in the recursion. In general, there are no distinct parts of either procedure that correspond to the elements in the signal-flow description. Our two procedures decompose the computations in a different way, spreading the enumeration over the program and mingling it with the map, the filter, and the accumulation. If we could organize our programs to make the signal-flow structure manifest in the procedures we write, this would increase the conceptual clarity of the resulting code.

Sequence Operations

The key to organizing programs so as to more clearly reflect the signal-flow structure is to concentrate on the ``signals'' that flow from one stage in the process to the next. If we represent these signals as lists, then we can use list operations to implement the processing at each of the stages. For instance, we can implement the mapping stages of the signal-flow diagrams using the `map` procedure from section 2.2.1:

```
(map square (list 1 2 3 4 5))
(1 4 9 16 25)
```

Filtering a sequence to select only those elements that satisfy a given predicate is accomplished by

```
(define (filter predicate sequence)
  (cond ((null? sequence) nil)
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate (cdr sequence))))
        (else (filter predicate (cdr sequence)))))
```

For example,

```
(filter odd? (list 1 2 3 4 5))
(1 3 5)
```

Accumulations can be implemented by

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))
(accumulate + 0 (list 1 2 3 4 5))
15
(accumulate * 1 (list 1 2 3 4 5))
120
(accumulate cons nil (list 1 2 3 4 5))
(1 2 3 4 5)
```

All that remains to implement signal-flow diagrams is to enumerate the sequence of elements to be processed. For `even-fibs`, we need to generate the sequence of integers in a given range, which we can do as follows:

```
(define (enumerate-interval low high)
  (if (> low high)
      nil
      (cons low (enumerate-interval (+ low 1) high))))
```

```
(enumerate-interval 2 7)
(2 3 4 5 6 7)
```

To enumerate the leaves of a tree, we can use[14]

```
(define (enumerate-tree tree)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (list tree))
        (else (append (enumerate-tree (car tree))
                      (enumerate-tree (cdr tree))))))
(enumerate-tree (list 1 (list 2 (list 3 4)) 5))
(1 2 3 4 5)
```

Now we can reformulate `sum-odd-squares` and `even-fibs` as in the signal-flow diagrams. For `sum-odd-squares`, we enumerate the sequence of leaves of the tree, filter this to keep only the odd numbers in the sequence, square each element, and sum the results:

```
(define (sum-odd-squares tree)
  (accumulate +
              0
              (map square
                   (filter odd?
                           (enumerate-tree tree)))))
```

For `even-fibs`, we enumerate the integers from 0 to n, generate the Fibonacci number for each of these integers, filter the resulting sequence to keep only the even elements, and accumulate the results into a list:

```
(define (even-fibs n)
  (accumulate cons
              nil
              (filter even?
                      (map fib
                           (enumerate-interval 0 n)))))
```

The value of expressing programs as sequence operations is that this helps us make program designs that are modular, that is, designs that are constructed by combining relatively independent pieces. We can encourage modular design by providing a library of standard components together with a conventional interface for connecting the components in flexible ways.

Modular construction is a powerful strategy for controlling complexity in engineering design. In real signal-processing applications, for example, designers regularly build systems by cascading elements selected from standardized families of filters and transducers. Similarly, sequence operations provide a library of standard program elements that we can mix and match. For instance, we can reuse pieces from the `sum-odd-squares` and `even-fibs` procedures in a program that constructs a list of the squares of the first $n + 1$ Fibonacci numbers:

```
(define (list-fib-squares n)
  (accumulate cons
              nil
              (map square
                   (map fib
                        (enumerate-interval 0 n)))))
(list-fib-squares 10)
(0 1 1 4 9 25 64 169 441 1156 3025)
```

We can rearrange the pieces and use them in computing the product of the odd integers in a sequence:

```
(define (product-of-squares-of-odd-elements sequence)
  (accumulate *
              1
              (map square
                   (filter odd? sequence))))
(product-of-squares-of-odd-elements (list 1 2 3 4 5))
225
```

We can also formulate conventional data-processing applications in terms of sequence operations. Suppose we have a sequence of personnel records and we want to find the salary of the highest-paid programmer. Assume that we have a selector `salary` that returns the salary of a record, and a predicate `programmer?` that tests if a record is for a programmer. Then we can write

```
(define (salary-of-highest-paid-programmer records)
  (accumulate max
              0
              (map salary
                   (filter programmer? records))))
```

These examples give just a hint of the vast range of operations that can be expressed as sequence operations.[15]

Sequences, implemented here as lists, serve as a conventional interface that permits us to combine processing modules. Additionally, when we uniformly represent structures as sequences, we have localized the data-structure dependencies in our programs to a small number of sequence operations. By changing these, we can experiment with alternative representations of sequences, while leaving the overall design of our programs intact. We will exploit this capability in section 3.5, when we generalize the sequence-processing paradigm to admit infinite sequences.

Exercise 2.33. Fill in the missing expressions to complete the following definitions of some basic list-manipulation operations as accumulations:

```
(define (map p sequence)
  (accumulate (lambda (x y) <??>) nil sequence))
(define (append seq1 seq2)
  (accumulate cons <??> <??>))
(define (length sequence)
  (accumulate <??> 0 sequence))
```

Exercise 2.34. Evaluating a polynomial in x at a given value of x can be formulated as an accumulation. We evaluate the polynomial

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

using a well-known algorithm called *Horner's rule*, which structures the computation as

$$\left(\cdots \left(a_n x + a_{n-1}\right) x + \cdots + a_1\right) x + a_0$$

In other words, we start with a_n, multiply by x, add a_{n-1}, multiply by x, and so on, until we reach a_0.[16] Fill in the following template to produce a procedure that evaluates a polynomial using Horner's rule. Assume that the coefficients of the polynomial are arranged in a sequence, from a_0 through a_n.

```
(define (horner-eval x coefficient-sequence)
  (accumulate (lambda (this-coeff higher-terms) <??>)
              0
              coefficient-sequence))
```

For example, to compute $1 + 3x + 5x^3 + x^5$ at $x = 2$ you would evaluate

```
(horner-eval 2 (list 1 3 0 5 0 1))
```

Exercise 2.35. Redefine `count-leaves` from section 2.2.2 as an accumulation:

```
(define (count-leaves t)
  (accumulate <??> <??> (map <??> <??>)))
```

Exercise 2.36. The procedure `accumulate-n` is similar to `accumulate` except that it takes as its third argument a sequence of sequences, which are all assumed to have the same number of elements. It applies the designated accumulation procedure to combine all the first elements of the sequences, all the second elements of the sequences, and so on, and returns a sequence of the results. For instance, if `s` is a sequence containing four sequences, `((1 2 3) (4 5 6) (7 8 9) (10 11 12))`, then the value of `(accumulate-n + 0 s)` should be the sequence `(22 26 30)`. Fill in the missing expressions in the following definition of `accumulate-n`:

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      nil
      (cons (accumulate op init <??>)
            (accumulate-n op init <??>))))
```

Exercise 2.37. Suppose we represent vectors $v = (v_i)$ as sequences of numbers, and matrices $m = (m_{ij})$ as sequences of vectors (the rows of the matrix). For example, the matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 6 \\ 6 & 7 & 8 & 9 \end{bmatrix}$$

is represented as the sequence `((1 2 3 4) (4 5 6 6) (6 7 8 9))`. With this representation, we can use sequence operations to concisely express the basic matrix and vector operations. These operations (which are described in any book on matrix algebra) are the following:

`(dot-product v w)`	returns the sum $\sum_i v_i w_i$:
`(matrix-*-vector m v)`	returns the vector t, where $t_i = \sum_j m_{ij} v_j$:
`(matrix-*-matrix m n)`	returns the matrix p, where $p_{ij} = \sum_k m_{ik} n_{kj}$:
`(transpose m)`	returns the matrix n, where $n_{ij} = m_{ji}$.

We can define the dot product as[17]

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

Fill in the missing expressions in the following procedures for computing the other matrix operations. (The procedure `accumulate-n` is defined in exercise 2.36.)

```
(define (matrix-*-vector m v)
  (map <??> m))
(define (transpose mat)
  (accumulate-n <??> <??> mat))
(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map <??> m)))
```

Exercise 2.38. The `accumulate` procedure is also known as `fold-right`, because it combines the first element of the sequence with the result of combining all the elements to the right. There is also a `fold-left`, which is similar to `fold-right`, except that it combines elements working in the opposite direction:

```
(define (fold-left op initial sequence)
  (define (iter result rest)
    (if (null? rest)
        result
        (iter (op result (car rest))
              (cdr rest))))
  (iter initial sequence))
```

What are the values of

```
(fold-right / 1 (list 1 2 3))
(fold-left / 1 (list 1 2 3))
(fold-right list nil (list 1 2 3))
(fold-left list nil (list 1 2 3))
```

Give a property that `op` should satisfy to guarantee that `fold-right` and `fold-left` will produce the same values for any sequence.

Exercise 2.39. Complete the following definitions of `reverse` (exercise 2.18) in terms of `fold-right` and `fold-left` from exercise 2.38:

```
(define (reverse sequence)
  (fold-right (lambda (x y) <??>) nil sequence))
(define (reverse sequence)
  (fold-left (lambda (x y) <??>) nil sequence))
```

Nested Mappings

We can extend the sequence paradigm to include many computations that are commonly expressed using nested loops.[18] Consider this problem: Given a positive integer n, find all ordered pairs of distinct positive integers i and j, where $1 \le j < i \le n$, such that $i + j$ is prime. For example, if n is 6, then the pairs are the following:

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
$i+j$	3	5	5	7	7	7	11

A natural way to organize this computation is to generate the sequence of all ordered pairs of positive integers less than or equal to n, filter to select those pairs whose sum is prime, and then, for each pair (i, j) that passes through the filter, produce the triple $(i,j,i+j)$.

Here is a way to generate the sequence of pairs: For each integer $i \le n$, enumerate the integers $j<i$, and for each such i and j generate the pair (i,j). In terms of sequence operations, we map along the sequence `(enumerate-interval 1 n)`. For each i in this sequence, we map along the sequence `(enumerate-interval 1 (- i 1))`. For each j in this latter sequence, we generate the pair `(list i j)`. This gives us a sequence of pairs for each i. Combining all the sequences for all the i (by accumulating with `append`) produces the required sequence of pairs:[19]

```
(accumulate append
            nil
            (map (lambda (i)
                   (map (lambda (j) (list i j))
```

```
                (enumerate-interval 1 (- i 1))))
        (enumerate-interval 1 n)))
```

The combination of mapping and accumulating with `append` is so common in this sort of program that we will isolate it as a separate procedure:

```
(define (flatmap proc seq)
  (accumulate append nil (map proc seq)))
```

Now filter this sequence of pairs to find those whose sum is prime. The filter predicate is called for each element of the sequence; its argument is a pair and it must extract the integers from the pair. Thus, the predicate to apply to each element in the sequence is

```
(define (prime-sum? pair)
  (prime? (+ (car pair) (cadr pair))))
```

Finally, generate the sequence of results by mapping over the filtered pairs using the following procedure, which constructs a triple consisting of the two elements of the pair along with their sum:

```
(define (make-pair-sum pair)
  (list (car pair) (cadr pair) (+ (car pair) (cadr pair))))
```

Combining all these steps yields the complete procedure:

```
(define (prime-sum-pairs n)
  (map make-pair-sum
       (filter prime-sum?
               (flatmap
                (lambda (i)
                  (map (lambda (j) (list i j))
                       (enumerate-interval 1 (- i 1))))
                (enumerate-interval 1 n)))))
```

Nested mappings are also useful for sequences other than those that enumerate intervals. Suppose we wish to generate all the permutations of a set S; that is, all the ways of ordering the items in the set. For instance, the permutations of {1,2,3} are {1,2,3}, { 1,3,2}, {2,1,3}, { 2,3,1}, { 3,1,2}, and { 3,2,1}. Here is a plan for generating the permutations of S: For each item x in S, recursively generate the sequence of permutations of S - x,[20] and adjoin x to the front of each one. This yields, for each x in S, the sequence of permutations of S that begin with x. Combining these sequences for all x gives all the permutations of S:[21]

```
(define (permutations s)
  (if (null? s)                    ; empty set?
      (list nil)                   ; sequence containing empty set
      (flatmap (lambda (x)
                 (map (lambda (p) (cons x p))
                      (permutations (remove x s))))
               s)))
```

Notice how this strategy reduces the problem of generating permutations of S to the problem of generating the permutations of sets with fewer elements than S. In the terminal case, we work our way down to the empty list, which represents a set of no elements. For this, we generate `(list nil)`, which is a sequence with one item, namely the set with no elements. The `remove` procedure used in `permutations` returns all the items in a given sequence except for a given item. This can be expressed as a simple filter:

```
(define (remove item sequence)
  (filter (lambda (x) (not (= x item)))
```

```
          sequence))
```

Exercise 2.40. Define a procedure `unique-pairs` that, given an integer n, generates the sequence of pairs (i,j) with $1 \leq j < i \leq n$. Use `unique-pairs` to simplify the definition of `prime-sum-pairs` given above.

Exercise 2.41. Write a procedure to find all ordered triples of distinct positive integers i, j, and k less than or equal to a given integer n that sum to a given integer s.

Exercise 2.42.

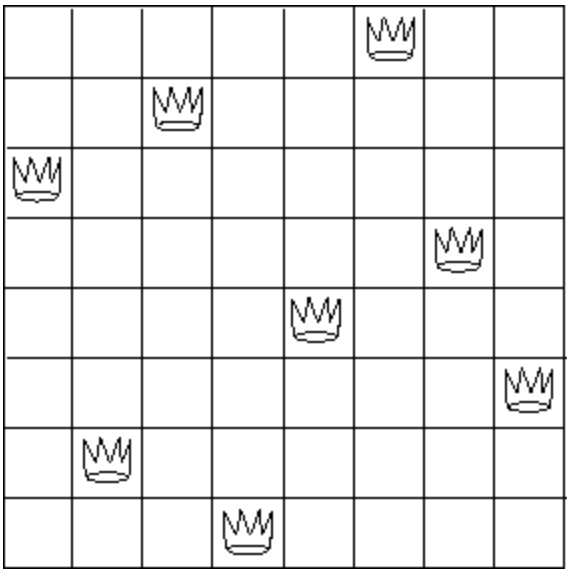

Figure 2.8: A solution to the eight-queens puzzle.

The ``eight-queens puzzle'' asks how to place eight queens on a chessboard so that no queen is in check from any other (i.e., no two queens are in the same row, column, or diagonal). One possible solution is shown in figure 2.8. One way to solve the puzzle is to work across the board, placing a queen in each column. Once we have placed k - 1 queens, we must place the kth queen in a position where it does not check any of the queens already on the board. We can formulate this approach recursively: Assume that we have already generated the sequence of all possible ways to place k - 1 queens in the first k - 1 columns of the board. For each of these ways, generate an extended set of positions by placing a queen in each row of the kth column. Now filter these, keeping only the positions for which the queen in the kth column is safe with respect to the other queens. This produces the sequence of all ways to place k queens in the first k columns. By continuing this process, we will produce not only one solution, but all solutions to the puzzle.

We implement this solution as a procedure `queens`, which returns a sequence of all solutions to the problem of placing n queens on an $n \times n$ chessboard. `Queens` has an internal procedure `queen-cols` that returns the sequence of all ways to place queens in the first k columns of the board.

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

```
          (queen-cols (- k 1))))))
  (queen-cols board-size))
```

In this procedure `rest-of-queens` is a way to place *k* - 1 queens in the first *k* - 1 columns, and `new-row` is a proposed row in which to place the queen for the *k*th column. Complete the program by implementing the representation for sets of board positions, including the procedure `adjoin-position`, which adjoins a new row-column position to a set of positions, and `empty-board`, which represents an empty set of positions. You must also write the procedure `safe?`, which determines for a set of positions, whether the queen in the *k*th column is safe with respect to the others. (Note that we need only check whether the new queen is safe -- the other queens are already guaranteed safe with respect to each other.)

Exercise 2.43. Louis Reasoner is having a terrible time doing exercise 2.42. His `queens` procedure seems to work, but it runs extremely slowly. (Louis never does manage to wait long enough for it to solve even the 6× 6 case.) When Louis asks Eva Lu Ator for help, she points out that he has interchanged the order of the nested mappings in the `flatmap`, writing it as

```
(flatmap
 (lambda (new-row)
   (map (lambda (rest-of-queens)
          (adjoin-position new-row k rest-of-queens))
        (queen-cols (- k 1))))
 (enumerate-interval 1 board-size))
```

Explain why this interchange makes the program run slowly. Estimate how long it will take Louis's program to solve the eight-queens puzzle, assuming that the program in exercise 2.42 solves the puzzle in time *T*.

2.2.4 Example: A Picture Language

This section presents a simple language for drawing pictures that illustrates the power of data abstraction and closure, and also exploits higher-order procedures in an essential way. The language is designed to make it easy to experiment with patterns such as the ones in figure 2.9, which are composed of repeated elements that are shifted and scaled.[22] In this language, the data objects being combined are represented as procedures rather than as list structure. Just as `cons`, which satisfies the closure property, allowed us to easily build arbitrarily complicated list structure, the operations in this language, which also satisfy the closure property, allow us to easily build arbitrarily complicated patterns.

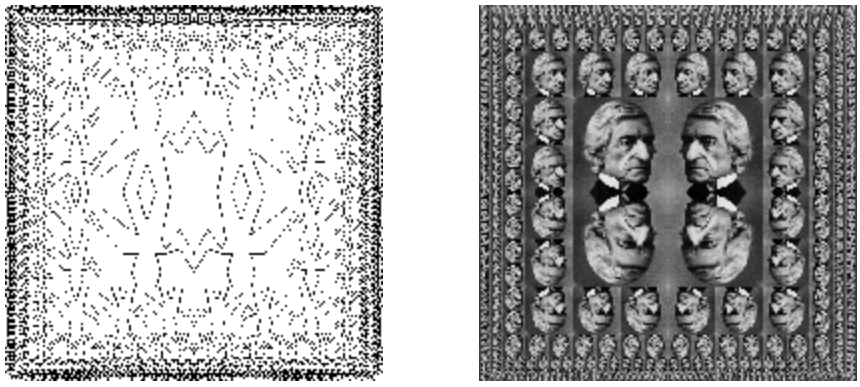

Figure 2.9: Designs generated with the picture language.

The picture language