```
          (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.

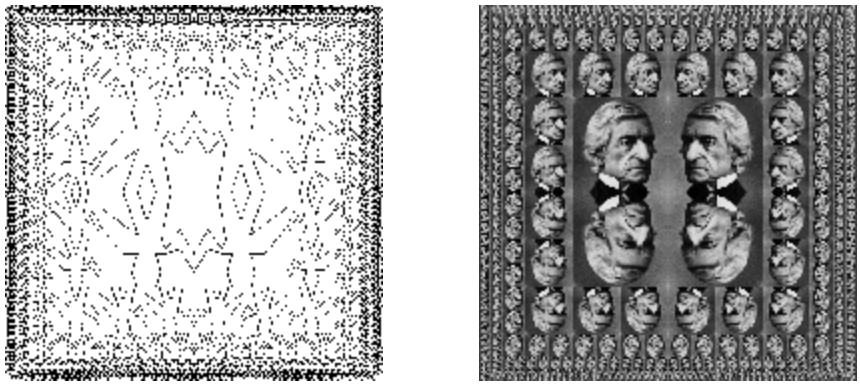


Figure 2.9: Designs generated with the picture language.

The picture language

When we began our study of programming in section 1.1, we emphasized the importance of describing a language by focusing on the language's primitives, its means of combination, and its means of abstraction. We'll follow that framework here.

Part of the elegance of this picture language is that there is only one kind of element, called a *painter*. A painter draws an image that is shifted and scaled to fit within a designated parallelogram-shaped frame. For example, there's a primitive painter we'll call `wave` that makes a crude line drawing, as shown in figure 2.10. The actual shape of the drawing depends on the frame -- all four images in figure 2.10 are produced by the same `wave` painter, but with respect to four different frames. Painters can be more elaborate than this: The primitive painter called `rogers` paints a picture of MIT's founder, William Barton Rogers, as shown in figure 2.11.[23] The four images in figure 2.11 are drawn with respect to the same four frames as the `wave` images in figure 2.10.

To combine images, we use various operations that construct new painters from given painters. For example, the `beside` operation takes two painters and produces a new, compound painter that draws the first painter's image in the left half of the frame and the second painter's image in the right half of the frame. Similarly, `below` takes two painters and produces a compound painter that draws the first painter's image below the second painter's image. Some operations transform a single painter to produce a new painter. For example, `flip-vert` takes a painter and produces a painter that draws its image upside-down, and `flip-horiz` produces a painter that draws the original painter's image left-to-right reversed.

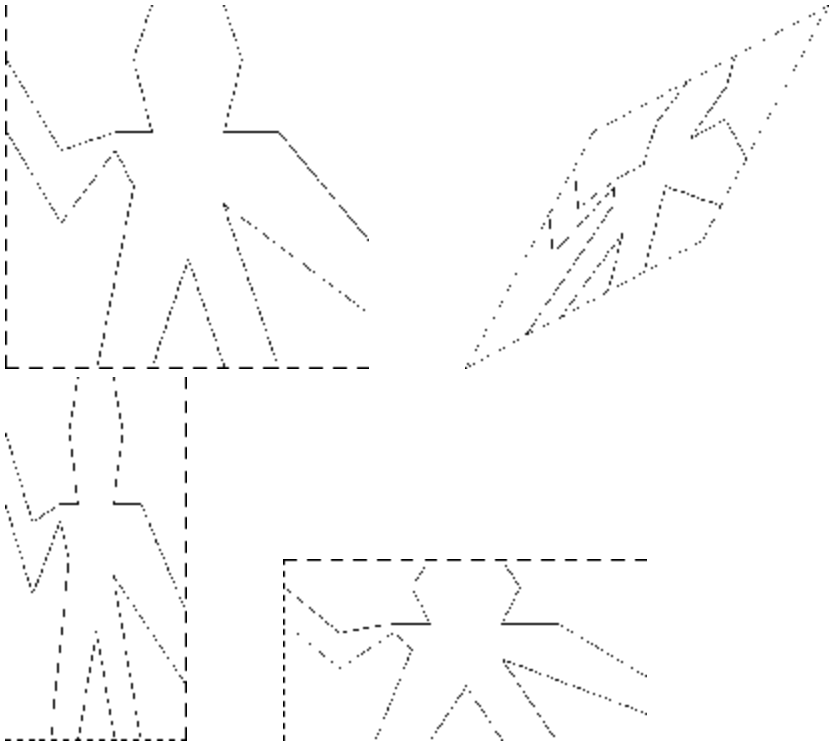


Figure 2.10: Images produced by the `wave` painter, with respect to four different frames. The frames, shown with dotted lines, are not part of the images.

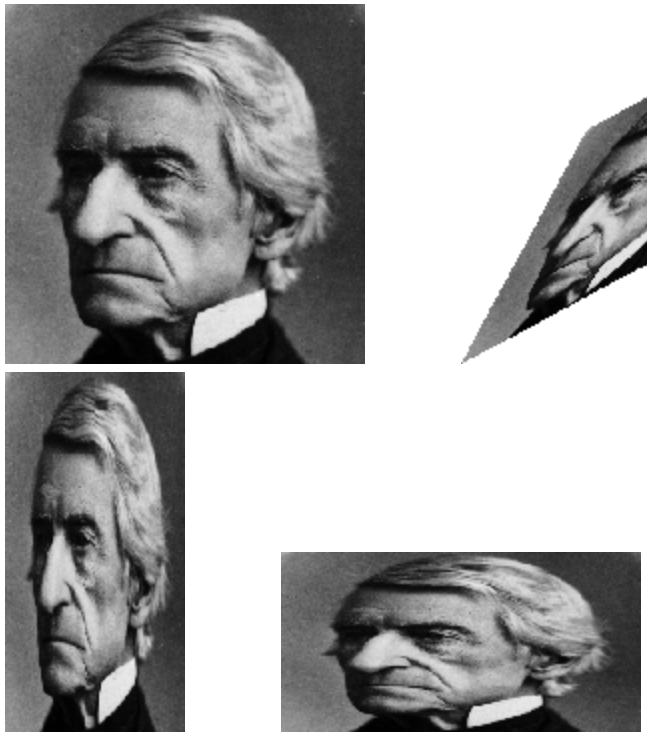
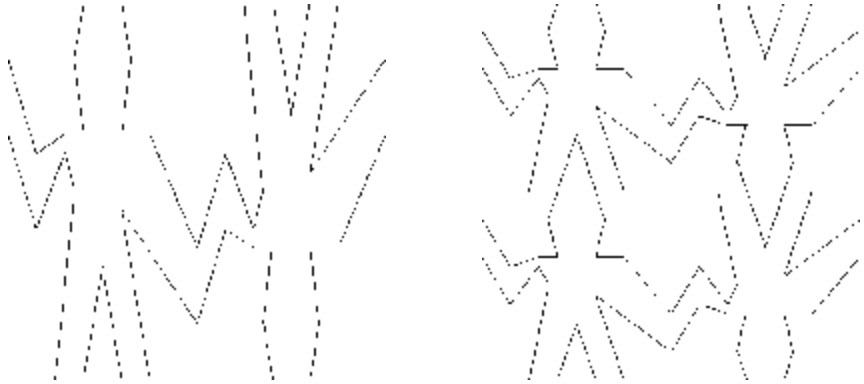


Figure 2.11: Images of William Barton Rogers, founder and first president of MIT, painted with respect to the same four frames as in figure 2.10 (original image reprinted with the permission of the MIT Museum).

Figure 2.12 shows the drawing of a painter called `wave4` that is built up in two stages starting from `wave`:

```
(define wave2 (beside wave (flip-vert wave)))
(define wave4 (below wave2 wave2))
```



```
(define wave2
  (beside wave (flip-vert wave)))
```

```
(define wave4
  (below wave2 wave2))
```

Figure 2.12: Creating a complex figure, starting from the `wave` painter of figure 2.10.

In building up a complex image in this manner we are exploiting the fact that painters are closed under the language's means of combination. The `beside` or `below` of two painters is itself a painter; therefore, we can use it as an element in making more complex painters. As with building up list structure using `cons`, the closure of our data under the means of combination is crucial to the ability to create complex structures while using only a few operations.

Once we can combine painters, we would like to be able to abstract typical patterns of combining painters. We will implement the painter operations as Scheme procedures. This

means that we don't need a special abstraction mechanism in the picture language: Since the means of combination are ordinary Scheme procedures, we automatically have the capability to do anything with painter operations that we can do with procedures. For example, we can abstract the pattern in `wave4` as

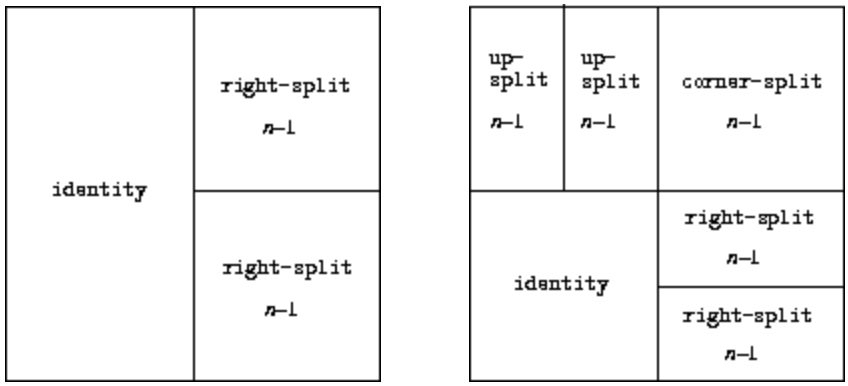
```
(define (flipped-pairs painter)
  (let ((painter2 (beside painter (flip-vert painter))))
    (below painter2 painter2)))
```

and define `wave4` as an instance of this pattern:

```
(define wave4 (flipped-pairs wave))
```

We can also define recursive operations. Here's one that makes painters split and branch towards the right as shown in figures 2.13 and 2.14:

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



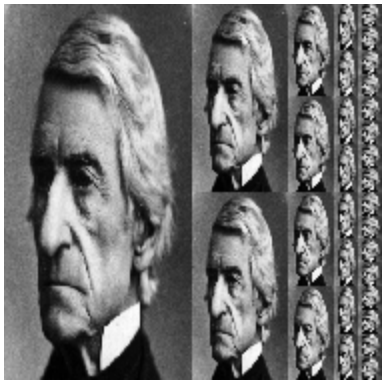
`right-split` *n* `corner-split` *n*

Figure 2.13: Recursive plans for `right-split` and `corner-split`.

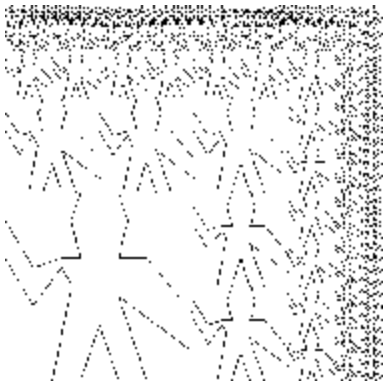
We can produce balanced patterns by branching upwards as well as towards the right (see exercise 2.44 and figures 2.13 and 2.14):

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

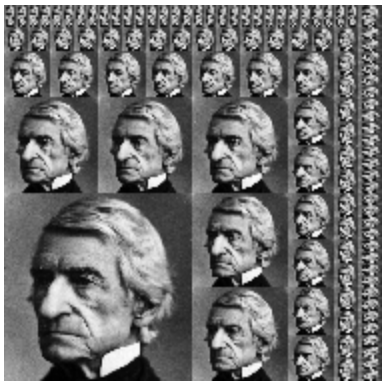
(right-split wave 4)



(right-split rogers 4)



(corner-split wave 4)



(corner-split rogers 4)

Figure 2.14: The recursive operations `right-split` and `corner-split` applied to the painters `wave` and `rogers`. Combining four `corner-split` figures produces symmetric `square-limit` designs as shown in figure 2.9.

By placing four copies of a `corner-split` appropriately, we obtain a pattern called `square-limit`, whose application to `wave` and `rogers` is shown in figure 2.9:

```
(define (square-limit painter n)
  (let ((quarter (corner-split painter n)))
    (let ((half (beside (flip-horiz quarter) quarter)))
      (below (flip-vert half) half))))
```

Exercise 2.44. Define the procedure `up-split` used by `corner-split`. It is similar to `right-split`, except that it switches the roles of `below` and `beside`.

Higher-order operations

In addition to abstracting patterns of combining painters, we can work at a higher level, abstracting patterns of combining painter operations. That is, we can view the painter operations as elements to manipulate and can write means of combination for these elements -- procedures that take painter operations as arguments and create new painter operations.

For example, `flipped-pairs` and `square-limit` each arrange four copies of a painter's image in a square pattern; they differ only in how they orient the copies. One way to abstract this pattern of painter combination is with the following procedure, which takes four one-argument painter operations and produces a painter operation that transforms a given painter with those four operations and arranges the results in a square. `Tl`, `tr`, `bl`, and `br` are the transformations to apply to the top left copy, the top right copy, the bottom left copy, and the bottom right copy, respectively.

```
(define (square-of-four tl tr bl br)
  (lambda (painter)
    (let ((top (beside (tl painter) (tr painter)))
          (bottom (beside (bl painter) (br painter))))
      (below bottom top))))
```

Then `flipped-pairs` can be defined in terms of `square-of-four` as follows:[24]

```
(define (flipped-pairs painter)
  (let ((combine4 (square-of-four identity flip-vert
                                  identity flip-vert)))
    (combine4 painter)))
```

and `square-limit` can be expressed as[25]

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

Exercise 2.45. `Right-split` and `up-split` can be expressed as instances of a general splitting operation. Define a procedure `split` with the property that evaluating

```
(define right-split (split beside below))
(define up-split (split below beside))
```

produces procedures `right-split` and `up-split` with the same behaviors as the ones already defined.

Frames

Before we can show how to implement painters and their means of combination, we must first consider frames. A frame can be described by three vectors -- an origin vector and two edge vectors. The origin vector specifies the offset of the frame's origin from some absolute origin in the plane, and the edge vectors specify the offsets of the frame's corners from its origin. If the edges are perpendicular, the frame will be rectangular. Otherwise the frame will be a more general parallelogram.

Figure 2.15 shows a frame and its associated vectors. In accordance with data abstraction, we need not be specific yet about how frames are represented, other than to say that there is a constructor `make-frame`, which takes three vectors and produces a frame, and three corresponding selectors `origin-frame`, `edge1-frame`, and `edge2-frame` (see exercise 2.47).

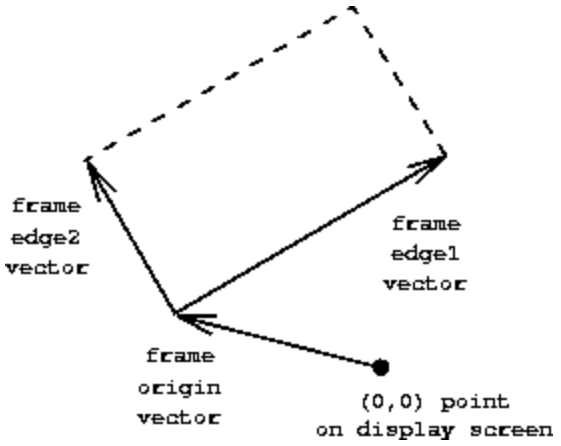


Figure 2.15: A frame is described by three vectors -- an origin and two edges.

We will use coordinates in the unit square ($0 \leq x,y \leq 1$) to specify images. With each frame, we associate a *frame coordinate map*, which will be used to shift and scale images to fit the frame. The map transforms the unit square into the frame by mapping the vector $\boldsymbol{v} = (x,y)$ to the vector sum

$$\mathrm{Origin}(\mathrm{Frame}) + x \cdot \mathrm{Edge}_1(\mathrm{Frame}) + y \cdot \mathrm{Edge}_2(\mathrm{Frame})$$

For example, (0,0) is mapped to the origin of the frame, (1,1) to the vertex diagonally opposite the origin, and (0.5,0.5) to the center of the frame. We can create a frame's coordinate map with the following procedure:[26]

```
(define (frame-coord-map frame)
  (lambda (v)
    (add-vect
     (origin-frame frame)
     (add-vect (scale-vect (xcor-vect v)
                           (edge1-frame frame))
               (scale-vect (ycor-vect v)
                           (edge2-frame frame))))))
```

Observe that applying `frame-coord-map` to a frame returns a procedure that, given a vector, returns a vector. If the argument vector is in the unit square, the result vector will be in the frame. For example,

```
((frame-coord-map a-frame) (make-vect 0 0))
```

returns the same vector as

```
(origin-frame a-frame)
```

Exercise 2.46. A two-dimensional vector **v** running from the origin to a point can be represented as a pair consisting of an *x*-coordinate and a *y*-coordinate. Implement a data abstraction for vectors by giving a constructor `make-vect` and corresponding selectors `xcor-vect` and `ycor-vect`. In terms of your selectors and constructor, implement procedures `add-vect`, `sub-vect`, and `scale-vect` that perform the operations vector addition, vector subtraction, and multiplying a vector by a scalar:

$$\begin{aligned}(x_1, y_1) + (x_2, y_2) &= (x_1 + x_2, y_1 + y_2)\\ (x_1, y_1) - (x_2, y_2) &= (x_1 - x_2, y_1 - y_2)\\ s \cdot (x, y) &= (sx, sy)\end{aligned}$$

Exercise 2.47. Here are two possible constructors for frames:

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

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

For each constructor supply the appropriate selectors to produce an implementation for frames.

Painters

A painter is represented as a procedure that, given a frame as argument, draws a particular image shifted and scaled to fit the frame. That is to say, if `p` is a painter and `f` is a frame, then we produce `p`'s image in `f` by calling `p` with `f` as argument.

The details of how primitive painters are implemented depend on the particular characteristics of the graphics system and the type of image to be drawn. For instance, suppose we have a procedure `draw-line` that draws a line on the screen between two specified points. Then we can create painters for line drawings, such as the `wave` painter in figure 2.10, from lists of line segments as follows:[27]

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

The segments are given using coordinates with respect to the unit square. For each segment in the list, the painter transforms the segment endpoints with the frame coordinate map and draws a line between the transformed points.

Representing painters as procedures erects a powerful abstraction barrier in the picture language. We can create and intermix all sorts of primitive painters, based on a variety of graphics capabilities. The details of their implementation do not matter. Any procedure can serve as a painter, provided that it takes a frame as argument and draws something scaled to fit the frame.[28]

Exercise 2.48. A directed line segment in the plane can be represented as a pair of vectors -- the vector running from the origin to the start-point of the segment, and the vector running from the origin to the end-point of the segment. Use your vector representation from exercise 2.46 to define a representation for segments with a constructor `make-segment` and selectors `start-segment` and `end-segment`.

Exercise 2.49. Use `segments->painter` to define the following primitive painters:

a. The painter that draws the outline of the designated frame.

b. The painter that draws an ``X'' by connecting opposite corners of the frame.

c. The painter that draws a diamond shape by connecting the midpoints of the sides of the frame.

d. The `wave` painter.

Transforming and combining painters

An operation on painters (such as `flip-vert` or `beside`) works by creating a painter that invokes the original painters with respect to frames derived from the argument frame. Thus, for example, `flip-vert` doesn't have to know how a painter works in order to flip it -- it just has to know how to turn a frame upside down: The flipped painter just uses the original painter, but in the inverted frame.

Painter operations are based on the procedure `transform-painter`, which takes as arguments a painter and information on how to transform a frame and produces a new painter. The transformed painter, when called on a frame, transforms the frame and calls the original painter on the transformed frame. The arguments to `transform-painter` are points (represented as vectors) that specify the corners of the new frame: When mapped into the frame, the first point specifies the new frame's origin and the other two specify the ends of its edge vectors. Thus, arguments within the unit square specify a frame contained within the original frame.

```
(define (transform-painter painter origin corner1 corner2)
  (lambda (frame)
    (let ((m (frame-coord-map frame)))
      (let ((new-origin (m origin)))
        (painter
         (make-frame new-origin
                     (sub-vect (m corner1) new-origin)
                     (sub-vect (m corner2) new-origin)))))))
```

Here's how to flip painter images vertically:

```
(define (flip-vert painter)
  (transform-painter painter
                     (make-vect 0.0 1.0)   ; new origin
                     (make-vect 1.0 1.0)   ; new end of edge1
                     (make-vect 0.0 0.0))) ; new end of edge2
```

Using `transform-painter`, we can easily define new transformations. For example, we can define a painter that shrinks its image to the upper-right quarter of the frame it is given:

```
(define (shrink-to-upper-right painter)
  (transform-painter painter
                     (make-vect 0.5 0.5)
                     (make-vect 1.0 0.5)
                     (make-vect 0.5 1.0)))
```

Other transformations rotate images counterclockwise by 90 degrees[29]

```
(define (rotate90 painter)
  (transform-painter painter
                     (make-vect 1.0 0.0)
                     (make-vect 1.0 1.0)
                     (make-vect 0.0 0.0)))
```

or squash images towards the center of the frame:[30]

```
(define (squash-inwards painter)
  (transform-painter painter
                     (make-vect 0.0 0.0)
                     (make-vect 0.65 0.35)
                     (make-vect 0.35 0.65)))
```

Frame transformation is also the key to defining means of combining two or more painters. The `beside` procedure, for example, takes two painters, transforms them to paint in the left and right halves of an argument frame respectively, and produces a new, compound painter. When the compound painter is given a frame, it calls the first transformed painter to paint in the left half of the frame and calls the second transformed painter to paint in the right half of the frame:

```
(define (beside painter1 painter2)
  (let ((split-point (make-vect 0.5 0.0)))
    (let ((paint-left
           (transform-painter painter1
                              (make-vect 0.0 0.0)
                              split-point
                              (make-vect 0.0 1.0)))
          (paint-right
           (transform-painter painter2
                              split-point
                              (make-vect 1.0 0.0)
                              (make-vect 0.5 1.0))))
      (lambda (frame)
```

```
        (paint-left frame)
        (paint-right frame)))))
```

Observe how the painter data abstraction, and in particular the representation of painters as procedures, makes `beside` easy to implement. The `beside` procedure need not know anything about the details of the component painters other than that each painter will draw something in its designated frame.

Exercise 2.50. Define the transformation `flip-horiz`, which flips painters horizontally, and transformations that rotate painters counterclockwise by 180 degrees and 270 degrees.

Exercise 2.51. Define the `below` operation for painters. `Below` takes two painters as arguments. The resulting painter, given a frame, draws with the first painter in the bottom of the frame and with the second painter in the top. Define `below` in two different ways -- first by writing a procedure that is analogous to the `beside` procedure given above, and again in terms of `beside` and suitable rotation operations (from exercise 2.50).

Levels of language for robust design

The picture language exercises some of the critical ideas we've introduced about abstraction with procedures and data. The fundamental data abstractions, painters, are implemented using procedural representations, which enables the language to handle different basic drawing capabilities in a uniform way. The means of combination satisfy the closure property, which permits us to easily build up complex designs. Finally, all the tools for abstracting procedures are available to us for abstracting means of combination for painters.

We have also obtained a glimpse of another crucial idea about languages and program design. This is the approach of *stratified design*, the notion that a complex system should be structured as a sequence of levels that are described using a sequence of languages. Each level is constructed by combining parts that are regarded as primitive at that level, and the parts constructed at each level are used as primitives at the next level. The language used at each level of a stratified design has primitives, means of combination, and means of abstraction appropriate to that level of detail.

Stratified design pervades the engineering of complex systems. For example, in computer engineering, resistors and transistors are combined (and described using a language of analog circuits) to produce parts such as and-gates and or-gates, which form the primitives of a language for digital-circuit design.[31] These parts are combined to build processors, bus structures, and memory systems, which are in turn combined to form computers, using languages appropriate to computer architecture. Computers are combined to form distributed systems, using languages appropriate for describing network interconnections, and so on.

As a tiny example of stratification, our picture language uses primitive elements (primitive painters) that are created using a language that specifies points and lines to provide the lists of line segments for `segments->painter`, or the shading details for a painter like `rogers`. The bulk of our description of the picture language focused on combining these primitives, using geometric combiners such as `beside` and `below`. We also worked at a higher level, regarding `beside` and `below` as primitives to be manipulated in a language whose operations, such as `square-of-four`, capture common patterns of combining geometric combiners.

Stratified design helps make programs *robust*, that is, it makes it likely that small changes in a specification will require correspondingly small changes in the program. For instance, suppose we wanted to change the image based on `wave` shown in figure 2.9. We could work at the lowest level to change the detailed appearance of the `wave` element; we could work at the middle level to change the way `corner-split` replicates the `wave`; we could work at the highest level to change

how `square-limit` arranges the four copies of the corner. In general, each level of a stratified design provides a different vocabulary for expressing the characteristics of the system, and a different kind of ability to change it.

Exercise 2.52. Make changes to the square limit of `wave` shown in figure 2.9 by working at each of the levels described above. In particular:

a. Add some segments to the primitive `wave` painter of exercise 2.49 (to add a smile, for example).

b. Change the pattern constructed by `corner-split` (for example, by using only one copy of the `up-split` and `right-split` images instead of two).

c. Modify the version of `square-limit` that uses `square-of-four` so as to assemble the corners in a different pattern. (For example, you might make the big Mr. Rogers look outward from each corner of the square.)

---

[6] The use of the word ``closure'' here comes from abstract algebra, where a set of elements is said to be closed under an operation if applying the operation to elements in the set produces an element that is again an element of the set. The Lisp community also (unfortunately) uses the word ``closure'' to describe a totally unrelated concept: A closure is an implementation technique for representing procedures with free variables. We do not use the word ``closure'' in this second sense in this book.

[7] The notion that a means of combination should satisfy closure is a straightforward idea. Unfortunately, the data combiners provided in many popular programming languages do not satisfy closure, or make closure cumbersome to exploit. In Fortran or Basic, one typically combines data elements by assembling them into arrays -- but one cannot form arrays whose elements are themselves arrays. Pascal and C admit structures whose elements are structures. However, this requires that the programmer manipulate pointers explicitly, and adhere to the restriction that each field of a structure can contain only elements of a prespecified form. Unlike Lisp with its pairs, these languages have no built-in general-purpose glue that makes it easy to manipulate compound data in a uniform way. This limitation lies behind Alan Perlis's comment in his foreword to this book: ``In Pascal the plethora of declarable data structures induces a specialization within functions that inhibits and penalizes casual cooperation. It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures."

[8] In this book, we use *list* to mean a chain of pairs terminated by the end-of-list marker. In contrast, the term *list structure* refers to any data structure made out of pairs, not just to lists.

[9] Since nested applications of `car` and `cdr` are cumbersome to write, Lisp dialects provide abbreviations for them -- for instance,

$$(\texttt{cadr}\ \langle arg\rangle) = (\texttt{car}\ (\texttt{cdr}\ \langle arg\rangle))$$

The names of all such procedures start with `c` and end with `r`. Each `a` between them stands for a `car` operation and each `d` for a `cdr` operation, to be applied in the same order in which they appear in the name. The names `car` and `cdr` persist because simple combinations like `cadr` are pronounceable.

[10] It's remarkable how much energy in the standardization of Lisp dialects has been dissipated in arguments that are literally over nothing: Should `nil` be an ordinary name? Should the value of `nil` be a symbol? Should it be a list? Should it be a pair? In Scheme, `nil` is an ordinary name, which we use in this section as a variable whose value is the end-of-list marker (just as `true` is an ordinary variable that has a true value). Other dialects of Lisp, including Common Lisp, treat `nil` as a special symbol. The authors of this book, who have endured too many language standardization brawls, would like to avoid the entire issue. Once we have introduced quotation in section 2.3, we will denote the empty list as `'()` and dispense with the variable `nil` entirely.

[11] To define `f` and `g` using `lambda` we would write

```
(define f (lambda (x y . z) <body>))
(define g (lambda w <body>))
```

[12] Scheme standardly provides a `map` procedure that is more general than the one described here. This more general `map` takes a procedure of *n* arguments, together with *n* lists, and applies the procedure to all the first elements of the lists, all the second elements of the lists, and so on, returning a list of the results. For example:

```
(map + (list 1 2 3) (list 40 50 60) (list 700 800 900))
(741 852 963)

(map (lambda (x y) (+ x (* 2 y)))
```

```
    (list 1 2 3)
    (list 4 5 6))
(9 12 15)
```

[13] The order of the first two clauses in the `cond` matters, since the empty list satisfies `null?` and also is not a pair.

[14] This is, in fact, precisely the `fringe` procedure from exercise 2.28. Here we've renamed it to emphasize that it is part of a family of general sequence-manipulation procedures.

[15] Richard Waters (1979) developed a program that automatically analyzes traditional Fortran programs, viewing them in terms of maps, filters, and accumulations. He found that fully 90 percent of the code in the Fortran Scientific Subroutine Package fits neatly into this paradigm. One of the reasons for the success of Lisp as a programming language is that lists provide a standard medium for expressing ordered collections so that they can be manipulated using higher-order operations. The programming language APL owes much of its power and appeal to a similar choice. In APL all data are represented as arrays, and there is a universal and convenient set of generic operators for all sorts of array operations.

[16] According to Knuth (1981), this rule was formulated by W. G. Horner early in the nineteenth century, but the method was actually used by Newton over a hundred years earlier. Horner's rule evaluates the polynomial using fewer additions and multiplications than does the straightforward method of first computing $a_n x^n$, then adding $a_{n-1}x^{n-1}$, and so on. In fact, it is possible to prove that any algorithm for evaluating arbitrary polynomials must use at least as many additions and multiplications as does Horner's rule, and thus Horner's rule is an optimal algorithm for polynomial evaluation. This was proved (for the number of additions) by A. M. Ostrowski in a 1954 paper that essentially founded the modern study of optimal algorithms. The analogous statement for multiplications was proved by V. Y. Pan in 1966. The book by Borodin and Munro (1975) provides an overview of these and other results about optimal algorithms.

[17] This definition uses the extended version of `map` described in footnote 12.

[18] This approach to nested mappings was shown to us by David Turner, whose languages KRC and Miranda provide elegant formalisms for dealing with these constructs. The examples in this section (see also exercise 2.42) are adapted from Turner 1981. In section 3.5.3, we'll see how this approach generalizes to infinite sequences.

[19] We're representing a pair here as a list of two elements rather than as a Lisp pair. Thus, the ``pair'' (i,j) is represented as `(list i j)`, not `(cons i j)`.

[20] The set $S - x$ is the set of all elements of S, excluding x.

[21] Semicolons in Scheme code are used to introduce *comments*. Everything from the semicolon to the end of the line is ignored by the interpreter. In this book we don't use many comments; we try to make our programs self-documenting by using descriptive names.

[22] The picture language is based on the language Peter Henderson created to construct images like M.C. Escher's ``Square Limit'' woodcut (see Henderson 1982). The woodcut incorporates a repeated scaled pattern, similar to the arrangements drawn using the `square-limit` procedure in this section.

[23] William Barton Rogers (1804-1882) was the founder and first president of MIT. A geologist and talented teacher, he taught at William and Mary College and at the University of Virginia. In 1859 he moved to Boston, where he had more time for research, worked on a plan for establishing a ``polytechnic institute,'' and served as Massachusetts's first State Inspector of Gas Meters.

When MIT was established in 1861, Rogers was elected its first president. Rogers espoused an ideal of ``useful learning'' that was different from the university education of the time, with its overemphasis on the classics, which, as he wrote, ``stand in the way of the broader, higher and more practical instruction and discipline of the natural and social sciences.'' This education was likewise to be different from narrow trade-school education. In Rogers's words:

> The world-enforced distinction between the practical and the scientific worker is utterly futile, and the whole experience of modern times has demonstrated its utter worthlessness.

Rogers served as president of MIT until 1870, when he resigned due to ill health. In 1878 the second president of MIT, John Runkle, resigned under the pressure of a financial crisis brought on by the Panic of 1873 and strain of fighting off attempts by Harvard to take over MIT. Rogers returned to hold the office of president until 1881.

Rogers collapsed and died while addressing MIT's graduating class at the commencement exercises of 1882. Runkle quoted Rogers's last words in a memorial address delivered that same year:

> ``As I stand here today and see what the Institute is, ... I call to mind the beginnings of science. I remember one hundred and fifty years ago Stephen Hales published a pamphlet on the subject of illuminating gas, in which he stated that his researches had demonstrated that 128 grains of bituminous coal -- "

> ``Bituminous coal,'' these were his last words on earth. Here he bent forward, as if consulting some notes on the table before him, then slowly regaining an erect position, threw up his hands, and was translated from the scene of his earthly labors and triumphs to ``the tomorrow of death,'' where the mysteries of life are solved, and the disembodied spirit finds unending satisfaction in contemplating the new and still unfathomable mysteries of the infinite future.

In the words of Francis A. Walker (MIT's third president):

> All his life he had borne himself most faithfully and heroically, and he died as so good a knight would surely have wished, in harness, at his post, and in the very part and act of public duty.

[24] Equivalently, we could write

```
(define flipped-pairs
  (square-of-four identity flip-vert identity flip-vert))
```

[25] `Rotate180` rotates a painter by 180 degrees (see exercise 2.50). Instead of `rotate180` we could say `(compose flip-vert flip-horiz)`, using the `compose` procedure from exercise 1.42.

[26] `Frame-coord-map` uses the vector operations described in exercise 2.46 below, which we assume have been implemented using some representation for vectors. Because of data abstraction, it doesn't matter what this vector representation is, so long as the vector operations behave correctly.

[27] `Segments->painter` uses the representation for line segments described in exercise 2.48 below. It also uses the `for-each` procedure described in exercise 2.23.

[28] For example, the `rogers` painter of figure 2.11 was constructed from a gray-level image. For each point in a given frame, the `rogers` painter determines the point in the image that is mapped to it under the frame coordinate map, and shades it accordingly. By allowing different types of painters, we are capitalizing on the abstract data idea discussed in section 2.1.3, where we argued that a rational-number representation could be anything at all that satisfies an appropriate condition. Here we're using the fact that a painter can be implemented in any way at all, so long as it draws something in the designated frame. Section 2.1.3 also showed how pairs could be implemented as procedures. Painters are our second example of a procedural representation for data.

[29] `Rotate90` is a pure rotation only for square frames, because it also stretches and shrinks the image to fit into the rotated frame.

[30] The diamond-shaped images in figures 2.10 and 2.11 were created with `squash-inwards` applied to `wave` and `rogers`.

[31] Section 3.3.4 describes one such language.

[Go to first, previous, next page; contents; index]