

## CHAPTER NINE

# Generic Operations

## 9.1 Introduction

We described data abstraction in Chapter 6 as a barrier between the way a data type is used and the way it is represented. There are a number of reasons to use data abstraction, but perhaps its greatest advantage is that the programmer can rely on an *abstract mental model* of the data rather than worrying about such mundane details as how the data is represented. For example, we can view the game-state ADT from Chapter 6 as a snapshot picture of an evolving Nim game and can view lists as finite sequences of objects. The simplification resulting from using abstract models is essential for many of the complicated problems programmers confront. In this chapter we will exploit and extend data abstraction by introducing *generic operations*, which are procedures that can operate on several different data types.

We rely on our mental model of an ADT when pondering how it might be used in a program. To actually work with the data, however, we need procedures that can manipulate it; these procedures are sometimes called the ADT's *interface*. For example, all of the procedures Scheme provides for manipulating lists comprise the list type's interface. The barrier between an ADT's use and its implementation results directly from the programming discipline of using the interface procedures instead of explicitly referring to the underlying representation. The interface must give us adequate power to manipulate the data as we would expect, given our mental model of the data, but we still have some flexibility in how the interface is specified. On the other hand, once we have specified the interface, we can easily imagine that some of the interface procedures would be appropriate for other data types. For example, most ADTs could benefit from a type-specific display procedure, if

only for debugging purposes; such a procedure should “do the right thing” for its data, regardless of how the data is represented. Generic operators allow us to share common operators among several different data types.

Another advantage of generic operators is that they can be used to maintain a uniform interface over similar data types that have entirely different representations. One example of this occurs when a data type can be represented in significantly different ways. For instance, suppose we wish to implement an ADT *date* to represent the date (i.e., the day, month, and year) when something occurs. One way to do this would be by using a three-part data structure containing integers representing the day, month, and year in the obvious manner (e.g., May 17, 1905, would be represented by the triple (17, 5, 1905)). An altogether different way would be to represent a date by the integer that equals the number of days that date fell after January 1, 1900 (January 1, 1901, would be represented by 365 and May 17, 1905, by 1962). Which representation we use can have a significant impact on performance. For example, if we want to determine whether a given date occurs in the month of May, that would be easier to do using the first representation than the second. On the other hand, if we want to find the number of days between two dates, the relative difficulty would be reversed. Of course we can convert between the two representations, but the formula would be quite messy in this case and in general might entail significant computational complexity.

When forced to decide between significantly different representations, the programmer must make a judgment based on how the ADT is likely to be used. However, in this chapter we’ll discover another option open to the programmer: allow multiple representations for the same abstract data type to coexist simultaneously. There are cases where this proves advantageous. We work through the details of such an example in Section 9.2.

More generally, it is not hard to imagine distinct data types that nonetheless share some commonality of form or purpose. For example, a library catalog will contain records of various kinds of items, say, books, movies, journals, and CDs. To a greater or lesser extent, all of these types of catalog items share some common attributes such as title, year of publication, and author (respectively, director, editor, and artist). Each kind of catalog item might have a separate interface (such as the interface we described for the movie ADT in Chapter 7). When combining the various types of catalog items into a common database, however, it would be greatly advantageous if they shared a common interface. We work through the details of this example in Section 9.3.

## 9.2 Multiple Representations

Scheme represents lists of values by explicitly consing together the elements in the list. Therefore there will be one `cons` pair per element in the list, which potentially requires a considerable amount of computer memory. Other lists can be represented

more efficiently, however, especially if the lists have some regularity in them. For example, if we know that a list consists of increasing consecutive integers in a given range (for example, 3 through 100), rather than storing the list (3 4 5 . . . 100), we could instead store the first and last elements of the range (3 and 100). Note that the standard list procedures can be easily computed in terms of the first and last elements. For example, the length of the example list is  $100 - 3 + 1 = 98$  and its “`cdr`” is the increasing consecutive list with first element 4 and last element 100. In this section, we’ll show how we can allow this new representation to coexist seamlessly with regular Scheme lists. To avoid confusion, we’ll think of both representations as implementations of *sequences* and use the term *list* to mean Scheme lists. Similarly, we’ll reserve the normal list notation, such as (1 2 3), for genuine Scheme lists, and when we want to write down the elements of a sequence, we will do so as  $\langle 1, 2, 3 \rangle$ , for example.

Let’s step back a moment to consider how Scheme deals with lists. We remarked in Chapter 7 that the choice of `cons`, `car`, `cdr`, and `null?` as the basic constructor and selectors for lists ran counter to good data-abstraction practice because they don’t sufficiently separate the use of lists from their representation. Even if we used more descriptive names like `make-list`, `head`, `tail`, and `empty-list?`, the underlying representation would be obscured but not fully hidden—`head` and `tail` are after all the two components of the underlying representation (the “two-part list viewpoint”). We will use more descriptive names like `head` and `tail` in our implementation of sequences, but these two procedures will not have the same “core” status as they do with Scheme lists.

In general, at least some of an ADT’s interface procedures must have direct access to the underlying representation, whereas others might well be implemented in terms of this basic set without direct reference to the underlying representation. For example, we indicated in Chapter 7 how `cons`, `car`, `cdr`, and `null?` formed such an essential set by showing how other list procedures such as `length`, `list-ref`, and `map` could be written in terms of them. However, Scheme itself may have a more complex representation that allows the other interface procedures to be more efficiently implemented. For example, the representation might keep track of the list’s length so that the length doesn’t have to be recalculated each time by `cdring` down the list. To allow our implementation of sequences to provide all operations as efficiently as possible, there will be no designated minimal set of core procedures. Instead, we will view the ADT *sequence* as being specified by its entire interface. That interface is implemented separately in terms of each underlying representation.

So how do we implement sequences in a manner that allows these two (and perhaps other) representations to coexist in a transparent manner? To start out, let’s suppose that we will have a variety of constructors (at least one for each representation) but will limit ourselves to the following selectors, which are modeled after the corresponding list procedures:

```

(head sequence)
  ; returns the first element of sequence, provided sequence is
  ; nonempty

(tail sequence)
  ; returns all but the first element of sequence as a
  ; sequence, provided sequence is nonempty

(empty-sequence? sequence)
  ; returns true if and only if sequence is empty

(sequence-length sequence)
  ; returns the length of sequence

```

We will implement sequences using a style of programming called *message-passing*, which exploits the fact that procedures are first-class data objects in Scheme. The data objects representing our sequences will not be passive: They will instead be procedures that respond appropriately to “messages,” which are symbols representing the various interface operations.

For example, we could write a procedure `sequence-from-to` that returns an increasing sequence of consecutive integers in a given range as follows:

```

(define sequence-from-to
  (lambda (low high)
    (lambda (op)
      (cond ((equal? op 'empty-sequence?)
             (> low high))
            ((equal? op 'head)
             low)
            ((equal? op 'tail)
             (sequence-from-to (+ low 1) high))
            ((equal? op 'sequence-length)
             (if (> low high) 0 (+ (- high low) 1)))
            (else (error "illegal sequence operation" op))))))

```

In this code, `op` is a symbol (the “message”) that represents the desired operator. After evaluating the procedure above, we might then have the following interaction:

```

(define seq-1 (sequence-from-to 3 100))
(seq-1 'head)

```

```
(seq-1 'sequence-length)
98
```

```
(seq-1 'tail)
#<procedure>
```

```
((seq-1 'tail) 'head)
4
```

Although this style of programming will probably appear quite odd the first few times you see it, a number of programming languages (notably *Smalltalk* and other *object-oriented* languages) successfully exploit the message-passing approach. Nevertheless, we can layer the more traditional interface on top of message-passing by defining the interface procedures as follows:

```
(define head
  (lambda (sequence)
    (sequence 'head)))

(define tail
  (lambda (sequence)
    (sequence 'tail)))

(define empty-sequence?
  (lambda (sequence)
    (sequence 'empty-sequence?)))

(define sequence-length
  (lambda (sequence)
    (sequence 'sequence-length)))
```

Our earlier interaction would then contain the following more familiar code:

```
(head seq-1)
3
```

```
(sequence-length seq-1)
98
```

```
(head (tail seq-1))
4
```

 **Exercise 9.1**

As is evident from the the output given above, we would be better able to check our procedures if we could easily display the sequences we construct. Instead of writing an ADT display procedure for sequences, an easier approach is to write a procedure `sequence->list` that converts a sequence to the corresponding Scheme list, which can then be directly displayed. Write this procedure, accessing the sequence only through the interface procedures `head`, `tail`, and `empty-sequence?`.

 **Exercise 9.2**

The sequences we just described are restricted to consecutive increasing sequences of integers (more precisely, to increasing arithmetic sequences where consecutive elements differ by 1). We can easily imagine similar but more general sequences such as  $\langle 6, 5, 4, 3, 2 \rangle$  or  $\langle 5, 5.1, 5.2, 5.3, 5.4, 5.5 \rangle$ —in other words, general *arithmetic sequences* of a given length, starting value, and increment (with decreasing sequences having a negative increment value).

- a. Write a procedure `sequence-with-from-by` that takes as arguments a length, a starting value, and an increment and returns the corresponding arithmetic sequence. Thus, `(sequence-with-from-by 5 6 -1)` would return the first and `(sequence-with-from-by 6 5 .1)` would return the second of the two preceding sequences. Remember that sequences are represented as procedures, so your new sequence constructor will need to produce a procedural result.
- b. The procedure `sequence-from-to` can now be rewritten as a simple call to `sequence-with-from-by`. The original `sequence-from-to` procedure made an empty sequence if its first argument was greater than its second, but you should make the new version so that you can get both increasing and decreasing sequences of consecutive integers. Thus, `(sequence-from-to 3 8)` should return  $\langle 3, 4, 5, 6, 7, 8 \rangle$ , whereas `(sequence-from-to 5 1)` should return  $\langle 5, 4, 3, 2, 1 \rangle$ .
- c. Write a procedure `sequence-from-to-with` that takes a starting value, an ending value, and a length and returns the corresponding arithmetic sequence. For example, `(sequence-from-to-with 5 11 4)` should return  $\langle 5, 7, 9, 11 \rangle$ .

Having given constructors for arithmetic sequences, we can add sequences represented by traditional Scheme lists by writing a procedure `list->sequence` that returns the sequence corresponding to a given list:

```
(define list->sequence
  (lambda (lst)
    (lambda (op)
      (cond ((equal? op 'empty-sequence?)
             (null? lst))
            ((equal? op 'head)
             (car lst))
            ((equal? op 'tail)
             (list->sequence (cdr lst)))
            ((equal? op 'sequence-length)
             (length lst))
            (else (error "illegal sequence operation" op))))))
```

In essence, we are off-loading each of the sequence procedures to the corresponding list procedure. Note that to the user, the various representations of sequences work together seamlessly and transparently:

```
(define seq-2 (sequence-with-from-by 6 5 -1))
(define seq-3 (list->sequence '(4 3 7 9)))
(head seq-2)
5
(head seq-3)
4
```

In a sense, each of the interface procedures triggers a representation-specific behavior that knows how to “do the right thing” for its representation.

### Exercise 9.3

Use `list->sequence` to write a procedure `empty-sequence` that takes no arguments and returns an empty sequence.

### Exercise 9.4

One disadvantage with the preceding version of `list->sequence` is that the Scheme procedure `length` normally has linear complexity in the list’s length (unless the version of Scheme you use does something like the trick we will now describe that reduces `sequence-length` to constant complexity).

- a. Modify `list->sequence` so that it has a `let` expression that computes the list's length once at sequence construction time and then uses that value when asked for the sequence's length.
- b. The problem with the solution in part a is that the tail's length is computed each time you return the tail. Because the complexity of calculating a list's length is proportional to the length, if you do the equivalent of `cdring` down the sequence, the resulting complexity is quadratic in the list-sequence's length, certainly an undesirable consequence.

One solution to this problem is to write an auxiliary procedure `list-of-length->sequence` that is passed both a list and its length and returns the corresponding sequence. This procedure can efficiently compute its tail, and `list->sequence` can be reimplemented as a call to `list-of-length->sequence`. Carry out this strategy.

This solution seems to nicely accomplish our goal of seamlessly incorporating different underlying representations, but because we have only implemented the four selectors `head`, `tail`, `empty-sequence?`, and `sequence-length`, we have not really tested the limits of this approach. To do so, let's attempt to add the selector and constructors corresponding to `list-ref`, `cons`, `map`, and `append`.

The selector that corresponds to `list-ref` differs significantly from the other selectors we've seen so far. Each of those takes only one parameter (the sequence) and, as a result, always returns the same value for a given sequence. In contrast, `sequence-ref` will take two parameters, a sequence and an integer index, and the value returned will depend on both the sequence and the index. Consequently, the `cond` branch corresponding to sequence reference in `sequence-from-to` or `list->sequence` should be a *procedure* that takes an integer index  $n$  and returns the  $n$ th number in the sequence. To see how this works, here is the expanded version of `sequence-from-to` that includes a branch for sequence reference:

```
(define sequence-from-to
  (lambda (low high)
    (lambda (op)
      (cond ((equal? op 'empty-sequence?)
             (> low high))
            ((equal? op 'head)
             low)
            ((equal? op 'tail)
             (sequence-from-to (+ low 1) high))
            ((equal? op 'sequence-length)
             (if (> low high) 0 (+ (- high low) 1)))
            ;;(continued)
```

```

(equal? op 'sequence-ref)
(lambda (n)
  (if (and (<= 0 n) (<= n (- high low)))
      (+ low n)
      (error "sequence-from-to: index out of range"
             n))))))

```

We can then implement `sequence-ref` as follows:

```

(define sequence-ref
  (lambda (sequence n)
    ((sequence 'sequence-ref) n)))

```

### ▶ Exercise 9.5

Rewrite `list->sequence` so that it has a branch for sequence reference.

The remaining three operators we will add to sequences correspond to the list operators `cons`, `map`, and `append`; for simplicity, we will call these operators `sequence-cons`, `sequence-map`, and `sequence-append`. Note that all three of these operators are in fact constructors, because their agenda is to create a *new* sequence from the given data. Therefore, rather than being implemented as branches of the `conds` of the other sequence constructors, we should simply write a new sequence constructor for each of these operators.

Consider `sequence-cons`, which is passed an element (to become the `head`) and an already-constructed sequence (to become the `tail`). Following is an implementation of `sequence-cons` that uses a `let` in order to calculate its length once and for all:

```

(define sequence-cons
  (lambda (head tail)
    (let ((new-length (+ 1 (sequence-length tail))))
      (lambda (op)
        (cond ((equal? op 'empty-sequence?)
              #f)
              ((equal? op 'head)
               head)
              ((equal? op 'tail)
               tail)
              ((equal? op 'sequence-length)
               new-length)
              ;;(continued)

```

```

(equal? op 'sequence-ref)
  (lambda (n)
    (if (= n 0)
        head
        (sequence-ref tail (- n 1))))
  (else (error "illegal sequence operation" op))))))

```

Note that we need not worry what “kind” of sequence `tail` is, because we are assured that whichever representation `tail` uses, it knows how to appropriately calculate `sequence-length` and `sequence-ref`. In particular, we can be sure that computational efficiencies constructed into `tail` (for example, in `sequence-length`) are maintained in `sequence-cons`.

### ▶ Exercise 9.6

Write the sequence constructor `sequence-map`, that outwardly acts like the list procedure `map`. However unlike `map`, which applies the procedural argument to all the list elements, `sequence-map` should not apply the procedural argument at all yet. Instead, when an element of the resulting sequence (such as its head) is accessed, that is when the procedural argument should be applied.

We finally arrive at the sequence constructor `sequence-append`. Just as we’ve shown how `append` can be used to append together two lists, we’ll write `sequence-append` such that it can append together any two sequences:

```

(define sequence-append
  (lambda (seq-1 seq-2)
    (cond ((empty-sequence? seq-1) seq-2)
          ((empty-sequence? seq-2) seq-1)
          (else
           (let ((seq-1-length (sequence-length seq-1))
                 (seq-2-length (sequence-length seq-2)))
             (lambda (op)
              (cond ((equal? op 'empty-sequence?)
                     #f)
                    ((equal? op 'head)
                     (head seq-1))
                    ((equal? op 'tail)
                     (sequence-append (tail seq-1) seq-2))
                    ((equal? op 'sequence-length)
                     (+ seq-1-length seq-2-length))
                    ;;(continued)

```

```

(equal? op 'sequence-ref)
(lambda (n)
  (cond ((< n seq-1-length)
        (sequence-ref seq-1 n))
        (else
         (sequence-ref seq-2
                        (- n seq-1-length))))))
(else (error "illegal sequence operation"
            op)))))))))

```

As with `sequence-cons`, the `let` expression is used primarily for efficiency. Note, however, that the length of the first subsequence is also used in `sequence-ref` to determine which subsequence to reference at what index.

## 9.3 Exploiting Commonality

Imagine that you have parlayed the movie query system from Chapter 7 into such an enormous success that it is currently being used by three major video store chains (not to mention the pirated versions being used by less honorable dealers). You have extended the natural language interface of your program (which you nicknamed Roger) to such an extent that many people think of Roger as a friend, and a few even consult him about their love life and other such befuddling aspects of their existence.

You were recently contacted by the owner of the Twilight Coffeehouse/Bookstore, who, in addition to selling books, also sells compact discs and rents obscure but interesting videos. She is very interested in extending Roger so that he could be consulted about books and CDs as well as videos. She already has two database programs, one for her books and one for her CDs (surprisingly, done in Scheme as well), but she prefers the Roger interface. What she would like to do is to combine these two databases and your video database into one large database. A tantalizing idea indeed, but how could it be done?

Combining the three databases involves more than just gluing the lists of records together; you also need to provide procedures that operate on the individual records in this new database. But there's a catch here: Each of those individual records could represent a book, a video, or a CD. Books, videos, and CDs have many properties in common; for example, each has a title and a year released. Also, the book's author more or less corresponds to the movie's director and the CD's recording artist. On the other hand, some attributes do not have obvious correlates from one data type to another (e.g., the actors in a movie have no obvious analogue in books or CDs). Ideally, we would like to have an interface that is as uniform as possible across the three underlying data types.

But speaking of an interface seems premature. After all, we defined an interface as the set of procedures that operate on an ADT, and we do not yet have a single ADT. From the point of view of generic operations, we can get around this problem by hypothesizing an ADT *catalog-item* that captures the *commonality* among the three underlying ADTs. Any given catalog-item will in fact be a movie, book, or CD (or any other catalog-item-like type we might later add), so movies, for example, might be called a “class” of catalog items. (We are essentially introducing here a form of *class hierarchy* that is a core concept in *object-oriented programming*, which we will describe in detail in Chapter 14.)

In order to fully specify the interface for catalog items, let’s write down the operations for each of the databases, grouping together those that are similar. Thus, we have three different operations for finding a title, three different operations for finding the year in which an object was made, and so on. The result is the table in Figure 9.1. The individual entries of the table are the procedures that operate on data of the given type. The columns of the table organize the operations by type, and the rows of the table group the analogous operations among the three types under a generic name indicating what is being done. In some cases (`title`, `year-made`, and `display-item`) there are obvious correlates for each of the three types. In other cases, such as `actors`, there are no obvious correlates, which leads to blanks in the table. In some cases, however, there are close analogues that we wrote in the table under more generic names (`creator` and `company`). Note that `company` applies to only two of the three data types because our movie database didn’t contain information about movies’ distributors. The names of the rows are precisely the generic operators we want to implement as the interface for catalog items.

We are left with the question of how we actually implement these generic operators. (Note that we are assuming that the three underlying types are already fully implemented, and our goal is to combine them in as transparent a manner as possible.) One method of accomplishing this would be to use the message-passing style of Section 9.2; we choose not to do it that way, mainly because we will take this opportunity to introduce another approach to genericity that involves tagging the

	movie	book	cd
title	movie-title	book-title	cd-title
year-made	movie-year-made	book-year-made	cd-year-made
display-item	display-movie	display-book	display-cd
creator	movie-director	book-author	cd-artist
company		book-publisher	cd-record-company
actors	movie-actors		

Figure 9.1 Operation table for Movies, Books, and CDs

data with its type. In addition to allowing us to explore a new technique, this latter approach is slightly more suited to the integration of already-existing types than message-passing. We will work through two variants of this type-tagging approach.

## Generic Operations through Symbolic Type Tags

Our first variant of the type-tagging approach involves attaching a symbolic tag, or label, to each piece of data that indicates whether the datum is a book, video, or CD. Then, when we write a generic procedure, we take a tagged datum, look at its tag, and use that to determine which operation to apply.

Tagging data is fairly simple. We need to create an ADT that binds together each datum with its tag. For example, we can do the following:

```
(define tagged-datum
  (lambda (type value)
    (cons type value)))

(define type car)

(define contents cdr)
```

The type argument could in general be various things; for now it will be a symbol that “names” the given type.

### Exercise 9.7

Write a procedure `list->tagged-list` that takes a list of untagged elements and a type and returns the corresponding list where each element has been tagged by the given type tag. Thus, if `movies` is a list of movie records, you can define a (symbolically) tagged list of movie records by evaluating:

```
(define tagged-movies
  (list->tagged-list movies 'movie))
```

If our three databases are lists called `movies`, `books`, and `cds`, then we could create the combined database as follows:

```
(define database
  (append (list->tagged-list movies 'movie)
          (list->tagged-list books 'book)
          (list->tagged-list cds 'cd)))
```

How can we implement the generic operations? Probably the most obvious way is to do so one operation at a time. Viewed in terms of the table in Figure 9.1, we are filling out the table row by row. Assume that the data has been tagged as in Exercise 9.7, with each element tagged with one of the symbols `movie`, `book`, or `cd`. We can then easily test the type of a given item by using the following predicates:

```
(define movie?
  (lambda (x)
    (equal? (type x) 'movie)))
```

```
(define book?
  (lambda (x)
    (equal? (type x) 'book)))
```

```
(define cd?
  (lambda (x)
    (equal? (type x) 'cd)))
```

Using these predicates, the generic operations become easy to write. For example, here is `title`:

```
(define title
  (lambda (x)
    (cond ((movie? x) (movie-title (contents x)))
          ((book? x) (book-title (contents x)))
          ((cd? x) (cd-title (contents x)))
          (else (error "unknown object in title" x)))))
```

### Exercise 9.8

In the course of integrating databases, some of the operations that seem analogous between types might have some annoying differences. For example, suppose that the movie directors and actors have their names stored as lists with the family names last, whereas for books and CDs the authors' and artists' names are stored with family names first. Suppose that you decide for consistency's sake and ease of display to have all of the generic procedures return the names with the family name last.

- a. Write a procedure `family-name-last` that takes a name (as a list of symbols) with family name first and returns the corresponding list with family name last.
- b. Use `family-name-last` to write a generic operation `creator` that returns the name with the family name last in all cases.

Implementing generic operations becomes somewhat more delicate when an operation doesn't apply across the three types, say, for example, `actors`. One possibility would be to signal an error when this occurs. For example, we could handle the operation `actors` by using `error`:

```
(define actors
  (lambda (x)
    (cond ((movie? x) (movie-actors (contents x)))
          (else (error "Cannot find the actors of the given type"
                       (type x))))))
```

If we choose this approach, we must modify the query system appropriately. For example, suppose Roger were asked the following question:

```
(what films was randy quaid in)
```

Then the action matching this pattern must not apply the operation `actors` to all items in the database because otherwise it will signal an error. This means that in this case the database must first be filtered by the predicate `movie?`. In general, patterns that clearly indicate the type of the desired records should first filter the database by the type's predicate.

### ▶ Exercise 9.9

An alternative approach to this problem is to return a testable value, for example, the empty list, if there is no procedure for the given operation and type. Discuss this approach, especially as it pertains to Roger.

### ▶ Exercise 9.10

Because we have posed our problem in terms of integrating databases, we should not assume that the result will be the last word in entertainment databases. After all, we might add a new type (say magazines), a new piece of information (perhaps the cost for rental or purchase), or some other increased functionality. Let's consider how difficult these tasks would be using the current approach to generic operations.

- a. Discuss what you would need to do to integrate a magazine database into the current one consisting of movies, books, and CDs. What changes would you have to make to the generic operations you have already written?
- b. Discuss what you would need to do to add a new generic operation, for example, the cost for rental or purchase of the given item.

- c. Discuss what you would need to do to add a single entry to the operation table, for example, adding a way of finding a movie’s distributor to the “company” row of the table.

## Operation Tables as Type Tags

In the variant of the type-tagging approach described above, we symbolically tagged the data as being of a given type and wrote the generic operations using a `cond` that branched on the allowable types for the operation. Viewed in terms of the operation table of Figure 9.1, this approach fills the table out row by row, which is to say operation by operation. One could argue that there would be advantages to an approach that more directly mirrors how the individual databases were originally constructed, namely, type by type. If we had used message-passing as suggested at the beginning of this section, we would have had a separate constructor for each underlying type, precisely this desired approach. But there would also be a great deal of redundancy in the message-passing implementation: After all, each of the movies contains a variant of the same general method for responding to the operations; individual movies only differ in their “content” data. You might well argue that this is precisely the point, and you would be correct. But somehow or other, these general methods of responding seem more appropriately associated with the type than with each of the separate data objects.

Is there some way to combine the two approaches? One way to do this would be to tag the data as in the preceding subsection but let the type tags provide the general procedures for performing the operations instead of merely being symbolic type names. In other words, the tags would correspond to the columns of the operation table. In effect, each type would be a one-dimensional table that stores the particular procedure to be used for each of the various generic operations.

Let’s implement a *type* ADT, which contains the name of the type as well as the operation table, because it will prove useful to know the name of the type when reporting errors:

```
(define make-type
  (lambda (name operation-table)
    (cons name operation-table)))

(define type-name car)

(define type-operation-table cdr)
```

We implement one-dimensional tables (the columns of the full operation table) as an abstract data type with a constructor `make-table` that will make a table from a list of the symbols denoting the operations and a list of the corresponding procedures

to be used for those operations on the given type. For example, we would define the type `movie` as

```
(define movie
  (make-type 'movie
    (make-table
      '(title year-made director actors creator display-item)
      (list movie-title movie-year-made movie-director
            movie-actors movie-director display-movie))))
```

Having defined the types `book` and `cd` as well, we could then define our tagged database as follows:

```
(define database
  (append (list->tagged-list movies movie)
          (list->tagged-list books book)
          (list->tagged-list cds cd)))
```

Notice that the tags are no longer simply symbols but are instead type objects that also contain the column of the operation table corresponding to the type of the tagged data item.

At this point each data object includes not only its particular component values but also has access to the column of the operation table that tells how to do the various operations. What we need now is a procedure, which we will call `operate`, that when given the name of an operation and a tagged data value, looks up the appropriate procedure in the data value's operation table and applies that procedure to the contents of the (tagged) data value. Thus we could use `operate` to define the generic operation `title` as follows:

```
(define title
  (lambda (tagged-datum)
    (operate 'title tagged-datum)))
```

How do we define `operate`? Clearly we must look up the operation name in the operation table and apply the corresponding procedure (if it exists) to the contents of the given data object. If no such procedure is found for the given operation, an error should be reported. This process is complicated. It would probably be best to have `operate` spin the table-searching tasks off onto another more general table-lookup procedure, to which it passes the necessary arguments. We will define a procedure `table-find`, which is passed the operation table, the name of the operation, a procedure that describes what to do if the operation is found in the given table,

and a procedure that describes what to do if it is not found. Thus, we would call `table-find` as follows:

```
(define operate
  (lambda (operation-name value)
    (table-find (type-operation-table (type value))
                operation-name
                (lambda (procedure) ; use this if found
                  (procedure (contents value)))
                (lambda ()          ; use this if not found
                  (error "No way of doing operation on type"
                         operation-name
                         (type-name (type value)))))))
```

Note that the procedure that `operate` supplies to `table-find` for use if the table lookup is successful takes one argument, namely, the procedure that was found in the table. In contrast, the procedure that `operate` supplies to `table-find` for the not-found case takes no arguments; it simply reports the error that occurred.

At this point, we need to define the table ADT, with its `make-table` and `table-find` operations. There are many plausible representations for tables; here, we'll opt for simplicity and just cons together into a pair the list of keys and the list of values:

```
(define make-table
  (lambda (keys values)
    (cons keys values)))
```

The procedure `table-find` simply `cdrs` down the two lists, looking in the list of keys for the desired key, (i.e., the operation name):

```
(define table-find
  (lambda (table key what-if-found what-if-not)
    (define loop
      (lambda (keys values)
        (cond ((null? keys) (what-if-not))
              ((equal? key (car keys))
               (what-if-found (car values)))
              (else
               (loop (cdr keys) (cdr values))))))
    (loop (car table) (cdr table))))
```

 **Exercise 9.11**

How would you implement the type predicates such as `movie?` using this representation with type tags containing operation tables?

 **Exercise 9.12**

Discuss the questions from Exercise 9.10 in terms of the operation-table type-tag representation.

 **Exercise 9.13**

Through this entire section, we've been glossing over a minor difficulty, namely, that many books are coauthored. Thus, it would be more likely that the book database actually supported a `book-authors` operation, which returns a list of authors, rather than the `book-author` operation we've been presuming. The primary difficulty this would cause is that we'd wind up with a `creator` generic operation that returns a single director for a movie, but a list of authors for a book. If we were processing a query like (`what do you have by John Doe`), we would have to in some cases test for equality between (`John Doe`) and the creator and in other cases test for membership of (`John Doe`) in the creator list.

- a. How would you arrange, at database integration time, for there to be a `creators` generic operation that returned a list of creators for any type of object, even a movie? Assume that the movie database is unchanged, so there is still just a singular director, whereas the book database is now presumed to have the `book-authors` operation. (Which assumption seems more plausible for CDs?)
- b. An alternative would be to change the movie database to directly support a list of directors, rather than a single director, for each movie. What are the relative advantages and disadvantages of the two approaches?

In the next section you'll have an opportunity to apply the technology of generic operations; we also use it as a tool while covering other topics in the next two chapters. We'll return to our consideration of generic operations as a topic of study in its own right in Chapter 14, which discusses object-oriented programming. The implementation technique we use there is a variant of the "operation tables as type tags" theme, with techniques we'll encounter in the meantime used to improve the efficiency of the table lookup.

## 9.4 An Application: Computer Graphics

In this section, we'll look inside graphics operations like those used in Chapters 1 through 4. We'll show how to use the message-passing technique to implement those graphics operations in terms of lower-level ones. In fact, we'll serve up a double helping of generic operations because there are two different abstract data types we'll use:

- Drawing media, on which we can perform the basic graphics operations of drawing lines and filled triangles
- Images, which can be arbitrarily complex assemblies so long as they know how to draw themselves onto a medium

First, let's consider why we want to treat images as an abstract data type with generic operations that can be used across multiple implementations. We have lots of different kinds of images from simple ones such as lines and filled triangles to more complex images such as pinwheeled quilts and c-curve fractals. Nonetheless, there are certain operations we want to perform on any image, without needing to know what kind of image it is. For example, we should be able to find the width and height of any image. If nothing else, this information is needed for error checking when we perform stacking and overlaying operations. (Only images of the same width can be stacked, and only images of the same width and height can be overlaid.) We also want to be able to draw an image onto a drawing medium in a single simple operation, without concerning ourselves with what conglomeration of lines and triangles may need to be drawn.

The situation with drawing media is a bit more interesting. First, there can be multiple forms of graphics output. For example, we can draw onto an on-screen window, or we can “draw” by writing to a file stored in some graphics file format, for later use. Thus we can foresee having at least two kinds of drawing media: windows and files. We can perform the same basic operations of drawing lines and filled triangles in either case but with different results depending on the kind of medium we are drawing on. Because we'll use generic operations to uniformly access any medium, we'll be able to construct complex images that know how to “draw themselves” on any medium, without the images needing to be aware of the different kinds of media. Additionally, we will show how we can layer a new “virtual medium” on top of an existing medium. We do this layering to make it easy to perform a transformation, such as turning an image.

Before we begin looking closely at images and drawing media, we need to take care of two details. First, both images and drawing media use a two-dimensional coordinate system. For example, if we wanted to create an image with a line, we would specify the two coordinates for each of the line's two endpoints. Now that

we've learned how to make compound data, we can make a point ADT. We define the constructor and selectors for points as follows:

```
(define make-point cons)
```

```
(define x-coord car)
```

```
(define y-coord cdr)
```

We'll use the convention that  $x$  coordinates increase from left to right and  $y$  coordinates increase from bottom to top. This is mathematically conventional but not in agreement with all computer systems' conventions. On some computer systems, the  $y$  coordinates in a window start with 0 at the top and increase as you go down the screen. On such a system, the low-level type of drawing medium for drawing on windows will need to take care of reversing the  $y$  coordinates.

We will, however, use two different ranges of coordinate values. One range is for images and so will be used for the arguments the user provides to the constructors of fundamental images, `make-line` and `make-filled-triangle`. For convenience and consistency with earlier chapters, these two constructors expect points with coordinates in the range from  $-1$  to  $1$ .

Our other range of coordinates will be used for doing the actual drawing on drawing media. For this drawing, we'll use coordinates that range from 0 up to the width or height of the medium. What units do we use to measure the width and height? We do not specify the unit of measure, but one reasonable unit for images displayed on a computer screen would be the size of a *pixel*, that is, one of the little dots that the computer can individually light up. For example, a  $100 \times 200$  medium might be drawing to a window of those dimensions so that valid  $x$  coordinates for drawing on the medium range from 0 at the far left to 100 at the far right, whereas the valid  $y$  coordinates range from 0 at the bottom to 200 at the top. We chose this coordinate system, with the origin in the lower left-hand corner rather than in the center, because it will simplify the calculations needed to stack images.

The second detail we need to take care of is providing an interface that hides our decision to use the message-passing style. That is, each image or drawing medium will be represented as a procedure that can perform the various operations when passed an appropriate symbolic message indicating the desired operation. However, our users will be thinking that various operations are performed on the images. Thus, we'll define the following interface procedures:

```
;; Interface to image operations
```

```
(define width
  (lambda (image)
    (image 'width)))
```

```

(define height
  (lambda (image)
    (image 'height)))

(define draw-on
  (lambda (image medium)
    ((image 'draw-on) medium)))

;; Interface to drawing medium operations

(define draw-line-on
  (lambda (point0 point1 medium)
    ((medium 'draw-line) point0 point1)))

(define draw-filled-triangle-on
  (lambda (point0 point1 point2 medium)
    ((medium 'draw-filled-triangle) point0 point1 point2)))

```

At this point, we know what operations we can invoke on an image or a medium, even though we don't have any images or media on which to invoke those operations. Conversely, we know what operations any image or medium we construct will need to support. By putting these two kinds of information together, we can begin to write some constructors. We'll start with the constructors for two fundamental images, `make-line` and `make-filled-triangle`. (We've chosen to call these procedures `make-line` and `make-filled-triangle`, rather than `line` and `filled-triangle`, to help you distinguish the procedures we're writing in this section from the predefined ones we used in the earlier chapters. We'll similarly avoid reusing other names.) These images support the `draw-on` operation for drawing themselves on a medium by using the `draw-line-on` and `draw-filled-triangle-on` interface operations specified above for media.

We'll need to make a rather arbitrary choice of size for these two fundamental images. (Other images, formed by stacking, turning, etc., will have sizes that derive from this basic image size.) The best choice depends on where the medium is doing its drawing; for example, if the medium is drawing on your computer screen, the best choice depends on such issues as the size of your computer's screen. However, the following value is probably in a plausible range:

```
(define basic-image-size 100)
```

Recall that `make-line` and `make-filled-triangle` need to convert from the user's coordinate range of  $-1$  to  $1$  into the drawing medium's coordinate range, which will

be from 0 to `basic-image-size`. We can convert a point from one range to the other using the following procedure:

```
(define transform-point ; from -1 to 1 into 0 to basic-image-size
  (lambda (point)
    (define transform-coord
      (lambda (coord)
        (* (/ (+ coord 1) 2) ; fraction of the way to top or right
           basic-image-size)))
      (make-point (transform-coord (x-coord point))
                  (transform-coord (y-coord point)))))
```

With these preliminaries in place, we can write our first image constructor:

```
(define make-line
  (lambda (point0 point1)
    (lambda (op)
      (cond
        ((equal? op 'width) basic-image-size)
        ((equal? op 'height) basic-image-size)
        ((equal? op 'draw-on)
         (lambda (medium)
            (draw-line-on (transform-point point0)
                          (transform-point point1)
                          medium)))
         (else (error "unknown operation on line" op))))))
```

As you can see, a line responds to queries about its width and height by reporting our `basic-image-size`, and it draws itself on a medium in the obvious way, by drawing a single line on that medium. So far, the image hasn't added any interesting functionality to that provided by the drawing medium itself. But remember, images can be more complex. For example, we could have an image that was a c-curve fractal of level 10. When we invoke its `draw-on` operation to draw it on a medium, 1024 `draw-line-on` operations will be performed on the medium for us.

So that you can test the preceding line constructor, we need to give you some way of making a drawing medium that actually displays an image on your screen. Later in this section we'll show how drawing media can be constructed, by working through the example of a type of drawing medium that writes a particular graphics file format. Meanwhile, you can use a procedure called `show` that's provided on the web site for this book. We provide specific versions of `show` for various different computer systems. You apply `show` to an image that needs to be shown, as in the call

```
(show (make-line (make-point 0 0) (make-point 1 1)))
```

The `show` procedure is more than just a drawing medium, however. First, `show` takes care of some system-dependent matters, such as opening a window that is the same size as the image. Then `show` constructs a drawing medium for drawing on that window (in some system-dependent way) and passes it to the image's `draw-on` procedure. When the drawing is all done, `show` takes care of any system-dependent wrap-up that needs to be done, such as notifying the system that the window is complete.

In the earlier chapters, we assumed that images constructed using the predefined image procedures were automatically shown, without needing to explicitly use a procedure such as `show`. The way this is implemented is that the Scheme system itself applies `show` when the value of a computation is an image, just as when the value is a number, it displays the digits representing the number. Thus `show` (or an analogue) was really at work behind the scenes. In this chapter we make it explicit.

We mentioned above that the images can be much more complex, like that 1024-line level-10 c-curve. However, before we move on to how these complex images are constructed, one other image type directly reflects the abilities of drawing media.

### ▶ Exercise 9.14

Write the `make-filled-triangle` image constructor. It should take three points as arguments, each with coordinates in the  $-1$  to  $1$  range. It should produce an image with the `basic-image-size` as its width and height, drawn on a medium as a filled triangle.

Now we are ready to consider how to build more complex images. We'll start with overlaying two images, because that is all we need to construct our c-curve example. Such an image should be able to report its height or width and should be able to draw itself. The size issue is fairly simple to deal with, but how do we get an overlaid image to draw itself? The answer is to use the fact that an overlaid image is a composite of two other images. When the overlaid image is asked to draw itself on a medium, it simply passes the buck to its two constituent images by asking them to draw themselves on that medium. This leads to the following constructor:

```
(define make-overlaid-image
  (lambda (image1 image2)
    (if (not (and (= (width image1) (width image2))
                 (= (height image1) (height image2))))
        (error "can't overlay images of different sizes")
        (lambda (op)
          (cond ((equal? op 'width) (width image1))
                ((equal? op 'height) (height image1))
                ;;(continued)
```

```

(equal? op 'draw-on)
  (lambda (medium)
    (draw-on image1 medium)
    (draw-on image2 medium)))
(else
 (error "unknown operation on overlaid image"
        op))))))

```

Notice that this is both a producer and a consumer of the interface we specified for images. Because what it produces is an image, the result needs to provide the `width`, `height`, and `draw-on` operations. Because that composite image is built from two preexisting images, it can count on `image1` and `image2` to be able to report their own width and height and to draw themselves appropriately. That way we don't need to care what sort of images are being overlaid.

You can try out the code thus far by using the `c-curve` procedure, rewritten in terms of our new constructors:

```

(define c-curve
  (lambda (x0 y0 x1 y1 level)
    (if (= level 0)
        (make-line (make-point x0 y0) (make-point x1 y1))
        (let ((xmid (/ (+ x0 x1) 2))
              (ymid (/ (+ y0 y1) 2))
              (dx (- x1 x0))
              (dy (- y1 y0)))
          (let ((xa (- xmid (/ dy 2)))
                (ya (+ ymid (/ dx 2))))
            (make-overlaid-image
             (c-curve x0 y0 xa ya (- level 1))
             (c-curve xa ya x1 y1 (- level 1))))))))))

```

With this definition in place, and using the `show` procedure we mentioned earlier to provide an appropriate on-screen drawing medium, you could do `(show (c-curve 0 -1/2 0 1/2 8))` to see a level-8 `c-curve`.

Let's now consider the example of turning an image a quarter turn to the right, as we did in designing quilt covers. We can use the ability to have different kinds of drawing media to great advantage here. When we want a turned image to draw itself on a particular medium, the turned image will create a new "virtual" medium layered on top of the given medium. This new medium takes care of doing the turning. In other words, when a turned image is asked to draw itself onto a base medium, it will pass the buck to the original image by asking it to draw itself on the virtual medium. The original image then asks the virtual medium to draw some lines

and/or triangles. The virtual medium responds to each of these requests by asking the base medium to draw a rotated version of the requested line or triangle.

How does the virtual medium turn the lines and triangles? The key to this turning is that we really only need to move the endpoints of the lines or the vertices of the triangle. A point that is near the left end of the medium's top edge will need to be transformed to a point near the top of the right-hand edge of the base medium, and a point at the center of the left edge will be transformed to a point at the center top. To turn a line connecting these two points, the virtual medium simply transforms each of the points and then asks the base medium to draw a line connecting the two transformed points.

When we write the constructor for this virtual medium, we'll assume that we have a `transform` procedure that can take care of transforming one point. That is, if we apply `transform` to the top center point, we get the center point of the right edge back. Given this point transformer, we can build the transformed medium using the following constructor:

```
(define make-transformed-medium
  (lambda (transform base-medium)
    (lambda (op)
      (cond
        ((equal? op 'draw-line)
         (lambda (point0 point1)
           (draw-line-on (transform point0) (transform point1)
                         base-medium)))
        ((equal? op 'draw-filled-triangle)
         (lambda (point0 point1 point2)
           (draw-filled-triangle-on (transform point0)
                                     (transform point1)
                                     (transform point2)
                                     base-medium)))
        (else
         (error "unknown operation on transformed medium"
                op))))))
```

Just as `make-overlaid-image` was both a producer and a consumer of the image interface, so too is `make-transformed-medium` both a producer and a consumer of the drawing medium interface. It constructs the new medium as a “wrapper” around the old medium—all operations on the new medium are translated into operations on the old medium.

For the specific problem of turning an image a quarter turn to the right, consider the quarter turn illustrated in Figure 9.2. Clearly the width and height are interchanged in the turned image, and the  $x$  coordinate of a transformed point is the

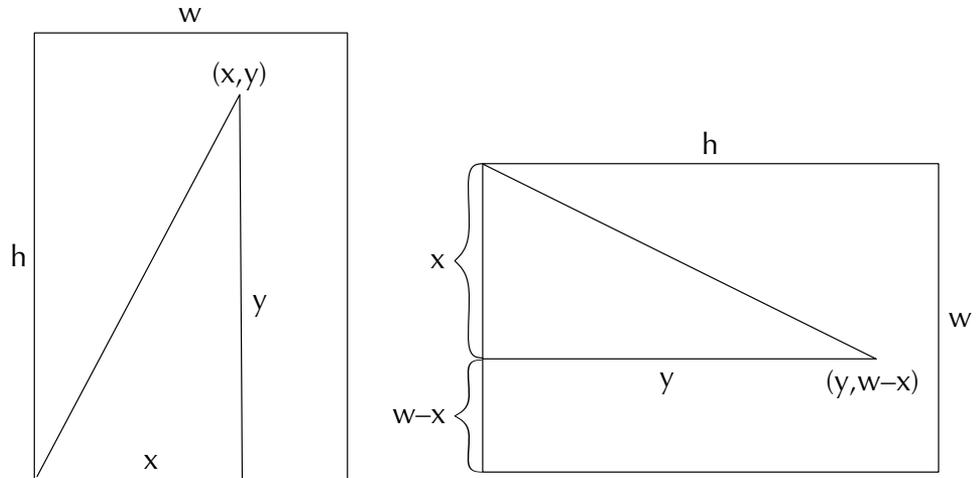


Figure 9.2 Illustration of what happens to the width, height, and a point  $(x, y)$  when an image is turned. The turned point has coordinates  $(y, w - x)$  where  $w$  is the width of the base image.

original point's  $y$  coordinate. (In our earlier example, this explains why a point on the top edge maps into a point on the right-hand edge.) Furthermore, we can obtain the transformed point's  $y$  coordinate by subtracting the original point's  $x$  coordinate from the new height, which is the old width. This leads to the following code:

```
(define make-turned-image ; quarter turn to the right
  (lambda (base-image)
    (define turn-point
      (lambda (point)
        ;; y becomes x, and the old width minus x becomes y
        (make-point (y-coord point)
                    (- (width base-image) (x-coord point))))))
    (lambda (op)
      (cond
        ((equal? op 'width) (height base-image))
        ((equal? op 'height) (width base-image))
        ((equal? op 'draw-on)
         (lambda (medium)
           (draw-on base-image
                    (make-transformed-medium turn-point medium))))
        (else (error "unknown operation on turned image" op))))))
```

You could test this out using lines, but if you've written `make-filled-triangle`, you can also try quarter turns out in their familiar context of quilting basic blocks, such as the two below:

```

(define test-bb
  (make-filled-triangle (make-point 0 1)
                        (make-point 0 -1)
                        (make-point 1 -1)))

(define nova-bb
  (make-overlaid-image
   (make-filled-triangle (make-point 0 1)
                         (make-point 0 0)
                         (make-point -1/2 0))
   (make-filled-triangle (make-point 0 0)
                         (make-point 0 1/2)
                         (make-point 1 0))))

```

Of course, we don't have to limit ourselves to just explicating the kind of image operations we used in earlier chapters. We can also add some new operations to our repertory.

### ▶ Exercise 9.15

Write a `make-mirrored-image` constructor. It should take a single image as an argument, like `make-turned-image` does. The image it produces should be the same size as the original but should be flipped around the vertical axis so that what was on the left of the original image is on the right of the mirrored image, and vice versa, as though the original image had been viewed in a mirror.

### ▶ Exercise 9.16

Write a `make-scaled-image` constructor. It should take a real number and an image as its arguments. The image it makes should be a magnified or shrunk version of the original image, under the control of the numeric scale argument. For example, `(make-scaled-image 2 test-bb)` should make an image twice as big as `test-bb`, whereas `(make-scaled-image 1/4 test-bb)` should make one-quarter as big as `test-bb`. (Of course, you can scale other images, like `c-curves`, as well.) Don't just scale the image's width and height; you also need to arrange for the scaling when the image is drawn.

To get full quilt covers, we also still need a way of stacking one image on top of another, making a new image with the combined heights. This is rather similar to `make-overlaid-image`, except that the top image will need to be fooled into drawing higher up on the drawing medium than it normally would so that its drawing goes above that of the bottom image. This can be achieved by giving it a transformed

medium to draw on. It will draw on that transformed medium using  $y$  coordinates that start at 0, but the transformed medium will translate that into drawing commands on the base medium that have larger  $y$  coordinate values.

### ▶ Exercise 9.17

Using this approach, write a `make-stacked-image` constructor that takes the top and bottom images as its arguments. You should initially test out your constructor by doing such simple evaluations as `(make-stacked-image test-bb nova-bb)`. Once it seems to work, you can make fancier quilt patterns as described below.

Using your `make-stacked-image` procedure along with our earlier `make-turned-image` procedure, the `pinwheel` procedure can be rewritten as follows:

```
(define pinwheel
  (lambda (image)
    (let ((turned (make-turned-image image)))
      (let ((half (make-turned-image (make-stacked-image turned
                                                                    image))))
        (make-stacked-image half
                              (make-turned-image
                                (make-turned-image half))))))))
```

With this in hand, you can make quilt covers by doing evaluations such as `(show (pinwheel (pinwheel nova-bb)))`. For large covers, you probably will want to `show` scaled-down versions made using `make-scaled-image`.

You may be feeling a bit ripped off because so far we haven't shown how a "real" drawing medium can be constructed, that is, one that doesn't just pass the buck in some transformed way to an underlying base medium. If you look on the web site for this book, you can find several system-dependent versions of the `show` procedure, each of which constructs some particular kind of on-screen drawing medium. At this point, we'll take a look at constructing a drawing medium that "draws" by writing to a file. This further highlights the benefits of generic operations. All of the image constructors we defined above are just as good for producing a graphical file as they are for drawing on the screen. That's because each of them draws on an arbitrary drawing medium, using the specified interface that all drawing media share.

The file format we'll write is one known as *Encapsulated PostScript*, or *EPS*. It is a popular format, which on many systems you'll be able to preview on-screen or include as illustrations in documents. (For example, you could write a *c-curve* or quilt pattern to an EPS file and then include that EPS file in a word-processed report. We used this technique to make illustrations for this book.) In addition to

the EPS format's popularity and versatility, it has the advantage of being a relatively straightforward textual format. For example, to draw a line from (0,0) to (100,100), we would put the following into the EPS file:

```
0 0 moveto 100 100 lineto stroke
```

Similarly, to draw a filled triangle with vertices (0,0), (100,0), and (50,100), we would put the following line into the file:

```
0 0 moveto 100 0 lineto 50 100 lineto closepath fill
```

Although this notation is likely to be unfamiliar to you, at least it is readable, unlike some other graphics file formats.

By using the `display` and `newline` procedures, we could write the EPS output to your computer's screen for you to see. That is, you'd wind up seeing a bunch of textual descriptions of graphical objects, like the examples given above. However, doing so would not be very useful. Instead, we'll write the EPS output into a file stored on your computer, ready for you to view using a previewer program or to incorporate into a word-processed document. To write this output to a file, we'll use a predefined Scheme procedure called `with-output-to-file`. It reroutes the output produced by procedures like `display` and `newline` so that they go into the file instead of to your screen. For example,

```
(with-output-to-file "foo"
  (lambda ()
    (display "hello, world")
    (newline)))
```

would create a one-line file called `foo` containing the message *hello, world*.

We'll write a procedure called `image->eps` that writes an EPS version of a given image into a file with a given filename. Just like `show`, this procedure will take care of some start-up details before asking the image to draw itself. The procedure first writes a bit of header information to the file, including the information about how big the image is. Then, `image->eps` asks the image to draw itself on a specially constructed drawing medium called `eps-medium` that outputs the EPS commands for drawing the lines or filled triangles.

```
(define image->eps
  (lambda (image filename)
    (with-output-to-file filename
      (lambda ()
        (display "%!PS-Adobe-3.0 EPSF-3.0"))
```

```

(newline)
(display "%BoundingBox: 0 0 ")
;; We need to make sure the bounding box is expressed
;; using only exact integers, as required by PostScript.
;; Therefore we process the width and height of the
;; image using round and then inexact->exact. The
;; round procedure would convert 10.8 into the inexact
;; integer 11., which the inexact->exact then converts
;; to the exact integer 11
(display (inexact->exact (round (width image))))
(display " ")
(display (inexact->exact (round (height image))))
(newline)
;; Now do the drawing
(draw-on image eps-medium))))

```

How does `eps-medium` work? It simply needs to draw lines and filled triangles in the EPS manner illustrated earlier, which leads to the following definition:

```

(define eps-medium
  (lambda (op)
    (cond ((equal? op 'draw-line)
           (lambda (point0 point1)
             (display-eps-point point0)
             (display "moveto")
             (display-eps-point point1)
             (display "lineto stroke")
             (newline)))
          ((equal? op 'draw-filled-triangle)
           (lambda (point0 point1 point2)
             (display-eps-point point0)
             (display "moveto")
             (display-eps-point point1)
             (display "lineto")
             (display-eps-point point2)
             (display "lineto closepath fill")
             (newline)))
          (else
           (error "unknown operation on EPS medium"
                  op))))))

```

The `display-eps-point` procedure this uses simply writes out the  $x$  and  $y$  coordinates in a format suitable for PostScript. In particular, PostScript can't handle a fraction written with a slash, such as  $1/2$ . Therefore, we use the predefined procedure `exact->inexact` to convert numbers that aren't integers into their "inexact" form, which gets displayed as `.5` (for example) rather than  $1/2$ . (The procedure `exact->inexact` returns unchanged any number that is already inexact.)

```
(define display-eps-point
  (lambda (point)
    (define display-coord
      (lambda (coord)
        (if (integer? coord)
            (display coord)
            (display (exact->inexact coord))))))
    (display " ")
    (display-coord (x-coord point))
    (display " ")
    (display-coord (y-coord point))
    (display " ")))
```

### ▶ Exercise 9.18

Write a procedure `summarize-image` that takes an image as its argument and uses `display` to give you a summary of that image as follows. For each line segment the image contains, a letter `l` should be displayed, and for each filled triangle, a letter `t`. For example, if you evaluate `(summarize-image (c-curve 0 -1/2 0 1/2 3))`, you should see eight `l`'s because the level-3 `c-curve` is constituted out of eight line segments.

---

## Review Problems

### ▶ Exercise 9.19

You are hired to supervise a team of programmers working on a computerized geometry system. It is necessary to manipulate various geometric figures in standard ways. As project manager, you have to select an organizational strategy that will allow all different shapes of geometric figures to be accessed using generic selectors for such information as the  $x$  and  $y$  coordinates of the figure and the area.

State which strategy you have chosen and briefly justify your choice (e.g., use one to three sentences). For your programmers' benefit, illustrate how your choice would be used to implement constructors `make-square` and `make-circle` and generic selectors `center-x`, `center-y`, and `area`. The two constructors each take three

arguments; in both cases, the first two are the  $x$  and  $y$  coordinates of the center of the figure. The third argument specifies the length of the side for a square and the radius for a circle. The selectors should be able to take either a square or a circle as their argument and return the appropriate numerical value.

### ▷ Exercise 9.20

Global Amalgamations Corp. has just acquired yet another smaller company and is busily integrating the data processing operations of the acquired company with that of the parent corporation. Luckily, both companies are using Scheme, and both have set up their operations to tolerate multiple representations. Unfortunately, one company uses operation tables as type tags, and the other uses procedural representations (i.e., message passing). Thus, not only are multiple representations now co-existing, but some of them are type-tagged data and others are message-passing procedures. You have been called in as a consultant to untangle this situation.

What is the minimum that needs to be done to make the two kinds of representation happily coexist? Illustrate your suggestion concretely using Scheme as appropriate. You may want to know that there is a built-in predicate `pair?` that tests whether its argument is a pair, and a similar one, `procedure?`, that tests whether its argument is a procedure.

### ▷ Exercise 9.21

One way we can represent a set is as a predicate (i.e., a procedure that returns true or false). The idea is that to test whether a particular item is in the set, we pass it to the procedure, which provides the answer. For example, using this representation, the built-in procedure `number?` could be used to represent the (infinite) set of all numbers.

- a. Implement `element-of-set?` for this representation. It takes two arguments, an element and a set, and returns true or false depending on whether the element is in the set or not.
- b. Implement `add-to-set` for this representation. It takes two arguments, an element and a set, and returns a new set that contains the specified element as well as everything the specified set contained. *Hint: Remember that a set is represented as a procedure.*
- c. Implement `union-set` for this representation. It takes two arguments—two sets—and returns a new set that contains anything that either of the provided sets contains.
- d. Write a paragraph explaining why you think the authors included this exercise in this chapter rather than elsewhere in the book.

 **Exercise 9.22**

Assume that `infinity` has been defined as a special number that is greater than all normal (finite) numbers and that when added to any finite number or to itself, it yields itself. (In some Scheme systems you can define it as follows: `(define infinity (/ 1.0 0.0))`.) Now there is no reason why sequences need to be of finite length. Write a constructor for some interesting kind of infinite sequence.

 **Exercise 9.23**

Show how the movie, book, and CD databases could be combined using message-passing instead of type-tagging.

---

## Chapter Inventory

### Vocabulary

abstract mental model	commonality
generic operation	class hierarchy
interface	object-oriented programming
multiple representations	type tag
message-passing	operation table
Smalltalk	pixel
object-oriented language	Encapsulated PostScript (EPS)
arithmetic sequence	

### Abstract Data Types

date	table
sequence	drawing medium
catalog-item	image
tagged datum	point
type	

### New Predefined Scheme Names

with-output-to-file  
pair?  
procedure?

### Scheme Names Defined in This Chapter

head	sequence-length
tail	sequence-from-to
empty-sequence?	sequence->list

sequence-with-from-by  
sequence-from-to-with  
list->sequence  
empty-sequence  
list-of-length->sequence  
sequence-ref  
sequence-cons  
sequence-map  
sequence-append  
title  
year-made  
display-item  
creator  
company  
actors  
tagged-datum  
type  
contents  
list->tagged-list  
tagged-movies  
database  
movie?  
book?  
cd?  
family-name-last  
make-type  
type-name  
type-operation-table  
make-table  
movie  
operate  
table-find  
make-point  
x-coord  
y-coord  
make-line  
make-filled-triangle  
width  
height  
draw-on  
draw-line-on  
draw-filled-triangle-on  
basic-image-size  
transform-point  
show  
make-overlaid-image  
make-transformed-medium  
make-turned-image  
make-mirrored-image  
make-scaled-image  
make-stacked-image  
image->eps  
summarize-image  
make-square  
make-circle  
center-x  
center-y  
area  
element-of-set?  
add-to-set  
union-set