PART II

Data Abstraction

In the previous part, we looked at how procedures describe computational processes. In this part, we will turn our attention to the data that is manipulated by those processes and how it can be structured. There are many qualitatively dissimilar ways of structuring data. For example, the list of stops for a bus route bears little resemblance to someone's family tree. In this part, we'll focus on a few representative data structures and the collection of operations that is appropriate to each one.

In Chapter 6, we'll work through the fundamentals of using and representing compound data in the relatively simple context of data types with a fixed number of components. Our focus is on the collection of operations that forms the interface between the uses of the data type and its representation. In Chapter 7, we'll show how two-component data structures can actually be used to represent lists of any length, by treating each nonempty list as having a first element and a list of the remaining elements. We'll extend this recursive approach to structuring data to hierarchical, tree-like structures in Chapter 8. Next we'll examine how a diverse collection of different data representations can present a single, uniform collection of interface operations. Finally, we'll look at programs as themselves being hierarchically composed data (expressions made of subexpressions) and see how to provide a uniform "evaluate" operation across the diversity of different expression types. By doing so, we'll show that implementing a programming language is really a specific application of the techniques introduced in this part of the book.

CHAPTER SIX

Compound Data and Data Abstraction

6.1 Introduction

Up until now, each value passed to one of our procedures as an argument or returned as a result was a single thing: a number, an image, a truth value, or a procedure. If we wanted to pass a procedure two numbers, we needed two separate arguments, because each argument value could only be a single thing. This kind of data is called *atomic* data. On the other hand, you can easily think of programs that use more complex data. For example, a program that plays poker would use hands of cards. There might be a `compare-hands` procedure that takes two arguments, namely, the two hands to compare, and reports which is better. Each argument to this procedure is a single thing, namely, a hand. Yet we can also select an individual card from a hand. So, at the same time as the hand is a single thing, it is also a collection of component cards. How about the cards themselves? Each card is clearly a single thing, yet we can also treat it as a combination of a suit and a rank. Data such as this, which we can interpret as both a single entity and also as a collection of parts, is called *compound data*.

In order to see how we can get a computer to navigate these strange waters between singular and plural, consider the following scenario. Suppose you run a mail-order company and want to start selling custom-knit "socks" for cats' tails—great for those cold nights. (You figure you'll have the market all to yourself.) The problem is, your order form only has space for a single model number, but the customer needs to specify both the length (to the nearest centimeter) and also whether or not they want the deluxe (mohair) version. In other words, they need to send you a combination of a number (the length) and a truth value (whether or not deluxe) but can only send in one thing—the model number. What do you do? You hire a consulting firm.

The consulting firm designs three calculator-like gizmos with keypads and displays. One of them, the *constructor*, is to be mailed out to your customers along with their catalogs. It *constructs* the model number from the length and the deluxeness choice. The other two gizmos, the *selectors*, are for use at your company. The deluxeness selector displays "yes" or "no" on its display when you enter a model number. The length selector displays the length in centimeters when you enter a model number.

You and your customers can use these devices without needing to know anything about how they work. In particular, you don't need to know how the two pieces of information are encoded into the model number. In contrast, the consultants who designed the gizmos (and who are likely to be the only ones who get rich on this whole harebrained scheme) need to decide on this encoding. They need to choose a particular *representation* of the two pieces of information, and this same representation needs to be embodied in both the constructor and the selectors.

This idea, that the representation of data can be exclusively the concern of a constructor and selectors, rather than of the ultimate creators and users of the data, is known as *data abstraction*. In slightly more general terms, data abstraction refers to separating the way a new type of data is used from the way it is represented. This means that when we add a new data type, we first decide what operations (procedures) are necessary to create and manipulate the data values. Then we figure out a good way of representing the data values using the types that are already part of our Scheme system. Finally, we find algorithms necessary for implementing the essential operations. Whenever we use the new data type in another program, we create and manipulate the data values *only* by using the essential operations, and not by accessing the underlying representation of the data.

The self-discipline of data abstraction brings three rewards. First of all, by specifying the basic operations that manipulate the new data type, the programmer needn't worry about how the data values are actually represented. This means that she can work with the abstract model of the data that she has in her head rather than constantly switching from the model to the underlying representation and back. (A programmer working in this representation-independent way is said to be using an *abstract data type*, or *ADT* .) Secondly, if we separate the way the data is used from how it is implemented, the implementation can be developed independently from the programs using the data. Because of this, we can often break down a large programming project into pieces that different teams of programmers can work on simultaneously. Finally, because the application programs access the data only through the basic operations, we can easily change the way the data is represented by simply changing these procedures. In particular, if we want to move a large software program (such as the Scheme system) from one type of computer to another (this process is called *porting*), a lot of the work is restricted to modifying the ADT implementations.

In the remainder of this chapter, we illustrate the technique of data abstraction by writing a program that plays the game of Nim. This is our first example of an interactive program, thereby introducing the reader to some simple input and output procedures in Scheme. For most of this chapter, our compound data will have only two components; however, we show how to create data types with three components in Section 6.4. This technique can be extended to any (fixed) number of components. Our application section considers how to use higher-order programming together with data abstraction to add strategies to the game of Nim.

In later chapters, we consider more complex data structures, such as lists and hierarchically structured data, that do not restrict the number of components in the data. We also consider complex "generic" operators that can be used on multiple data types. Finally, we apply these ideas to one very special kind of compound data, programs, and a particularly interesting operation, the running of those programs.

6.2 Nim

In this section we will start to write the procedures for playing a variation on the game of Nim. The first appearance of "official" rules for Nim is in a 1901 paper by Charles Bouton, in which he analyzed the game and presented a winning strategy. However, like many informal folk games, there are many different ways of playing Nim. All the variations start with objects of some sort arranged in some way. Two players alternate removing objects according to certain rules, and the player who takes the last object is the winner (or the loser, in some variations). We present three ways to play Nim:

1. The version we'll call three-pile Nim is presented in Bouton's paper. It is played using three piles of objects, say coins. When the game starts, the three piles have different numbers of coins. Two players take turns removing coins. Each player must take at least one coin each turn and may take more as long as they are all from the same pile. The winner is the player who takes the last coin (or coins). Because finding a winning strategy for this version is relatively easy, Bouton suggested the variation in which the player who takes the last coin loses the game.
2. Instead of three piles, some other number of piles can be used, each starting with some arbitrary number of coins in it. The game proceeds exactly as above, with the winner (or loser) being the last person to remove one or more coins. In particular, we'll work with the two-pile version in most of this chapter.
3. The final version is the tastiest one. The game wouldn't be conceptually any different if something else were progressively reduced other than the number of coins in piles. How about the number of rows and the number of columns in a chocolate bar? You start with the kind of chocolate candy bar that is scored into rows and columns so that you could break it into small squares. Pretend that the square in the bottom left-hand corner is poisoned. The players take turns breaking

the bar into two pieces by breaking along one of the horizontal grooves or one of the vertical grooves in the bar and then eating the section not containing the poisoned square. Eventually, one player will be left with just one square, which is poisoned. That player loses.

How is the last version related to the previous two? In fact it is equivalent to two-pile Nim. To see this, you need to realize that the candy bar is completely specified by the number of horizontal and vertical grooves (or scores). If we represent the bar by two piles, one with a coin for each horizontal score and the other with a coin for each vertical score, breaking the candy bar along a vertical or horizontal score corresponds to removing coins from one of the piles, and ending with the poisoned square corresponds to both piles being empty (because when the bar is down to a single square, there are no scores left). Breaking the chocolate at the last place it can be broken is like taking the last coin.

Before we write a program for the computer to play Nim with a human user, we must first decide which version we want to play. We've chosen to concentrate on the two-pile version, with the winner being the player who takes the last coin. As the above discussion suggests, it doesn't matter whether we actually play with two piles of coins or, instead, use a chocolate bar. When we play with the computer, we presumably won't use either physical piles or a chocolate bar, but rather some third option better suited to the computer's capabilities. To get a feel for how to play Nim, find a partner and play a few games. (We disclaim all responsibility if you choose the chocolate bar version.) As you play, think about how you might write a program that could play Nim with you.

What type of data will such a program need? If you think about how you played, you will see that you and your partner started with a particular configuration of coins and took turns transforming the current configuration into a new configuration by making legal moves. The configuration of the coins in the two piles described the state (or condition) of the game at a given time. For this reason, we will call the configurations *game states*; they are our new data type. That is, we will arrange things so that we can pass a game state into a procedure as an argument or return a game state as the result of a procedure, just as we can with any other type of value. That way, the transformation you do in making a move can be a procedure. Game states can be physically represented by two piles of coins, which we call the first pile and the second pile.

Next, find a third person to be a gamekeeper and play another game with your partner. This time, instead of physically removing coins from piles, have the gamekeeper do all the work. The gamekeeper should keep track of the individual game states; you and your partner will give him directions and ask him questions. As you play, concentrate on what directions you give the gamekeeper and what questions you ask. You should discover that you repeatedly ask how many coins there are in a particular pile of the current game state and that you tell the gamekeeper to change to a new game state by removing some number of coins from a particular pile. This tells us that there are at least two operations we need for our data structure. One tells us how many coins are in either one of the piles and the other allows us to "remove" a specified number of coins from a pile, by making a new state with fewer coins in that pile. In Scheme, we could specify these operations as follows:

```
(size-of-pile game-state p)
  ;returns an integer equal to the number of coins in the p-th
  ;pile of the game-state

(remove-coins-from-pile game-state n p)
  ;given a Nim game-state, returns a new game state with n
  ;fewer coins in pile p
```

We will also need an operation that creates a new game state with a specified number of coins in each pile. This operation is what is used to set up an initial game state:

```
(make-game-state n m)
  ;returns a game state with n coins in the first pile
  ;and m coins in the second pile
```

Exercise 6.1

A fourth version of Nim uses two piles of coins but adds the restriction that a player can remove at most three coins in any one turn. To implement this version, we could use the three operations, `make-game-state`, `size-of-pile`, and `remove-coins-from-pile`, as before. In this case, whenever we use `remove-coins-from-pile`, the parameter indicating how many coins to remove should have a value of 1, 2, or 3.

Alternatively, we could replace `remove-coins-from-pile` with the three operations, `remove-one`, `remove-two`, and `remove-three`, where each of these operations would have just two parameters: the current game state and the pile to remove from. With this implementation, we would have five operations instead of three. Compare these two approaches to implementing this new version of Nim. What are the advantages and disadvantages of each one? Is one implementation better than the other?

Are these three operations (`make-game-state`, `size-of-pile`, and `remove-coins-from-pile`) enough? In other words, if someone else implemented the abstract data type of game states for us, could we then write the procedures we need to have the computer play Nim? Let's try to do this.

Recall that when you and your partner played Nim, you progressed through a succession of game states by alternately making moves and that you continued to do so until all the coins were gone. The game would eventually end, because every move reduces the total number of coins by at least one. In other words, each move transforms the game into a smaller game, if we measure the size of the game in terms of the total number of coins. Therefore, the main game-playing procedure, which we will call `play-with-turns`, should repeatedly reduce the game state by alternately having the computer and the human make a move. We will have two procedures, `human-move` and `computer-move`, to do these separate reductions. When the game is over (which will be determined by an as yet unwritten predicate `over?`,) the computer should announce the winner; we'll do that using the procedure `announce-winner`.

The procedure `play-with-turns` will use two new aspects of Scheme: quoted *symbols* and the equality predicate `equal?`. A symbol is a basic Scheme data type that is simply a name used as itself, rather than as the name of something. You specify a symbol by putting a single *quote mark* before it. To illustrate, here is some Scheme dialog:

```
'human
human

'x
x

(define x 'y)
x
y

(define z x)
z
y

(define w 'x)
w
x
```

The quote mark isn't part of the symbol itself; instead, the combination of the quote mark and the symbol is an expression, the value of which is the symbol. As this example shows, the value of the expression `'x` is the symbol `x`, whereas the value of the expression `x` is whatever `x` has been defined as a name for (here, the symbol `y`).

Because the equality predicate `=` only works for numbers, we need to use the more general equality predicate `equal?` for symbols. Putting this all together, we come up with the following version of `play-with-turns`:

```
(define play-with-turns    ;warning: this is not the final version
  (lambda (game-state player)
    (cond ((over? game-state)
           (announce-winner player))
          ((equal? player 'human)
           (play-with-turns (human-move game-state) 'computer))
          ((equal? player 'computer)
           (play-with-turns (computer-move game-state) 'human)))))
```

We would play a game by evaluating an expression such as

```
(play-with-turns (make-game-state 5 8) 'human)
```

There is one major problem with `play-with-turns`. Consider what would happen if we attempted to play a game by evaluating the expression `(play-with-turns (make-game-state 5 8) 'humman)`. Because the symbol `humman` is neither `human` nor `computer`, and because the game is clearly not over, none of the conditions in the `cond` would be met. Thus, the game would not be played and an undefined value would be returned. Furthermore, the user would have no idea why nothing happened. One way to fix this is to use an `else` instead of `(equal? player 'computer)`; however, this is still unsatisfactory because the game gets played with the computer having the first turn and the user has no idea why he didn't get to go first.

A better strategy is to use the procedure `error`, which, although not a part of R^{4}RS Scheme, is predefined in the versions of Scheme that are recommended for this book. The procedure `error` stops the normal execution of the program and notifies the user that an error occurred. In order to tell the user the nature of the error, we give `error` a description of the error as an argument, as in

```
(error "player wasn't human or computer")
```

This descriptive argument is a *character string*, which is written as a sequence of characters enclosed in double quotes. In this book, we'll generally use character strings in cases like this, for output. We can also tell the user more about the error by passing additional arguments to `error`, which it will display. For example, we can use

```
(error "player wasn't human or computer:" player)
```

to let the user know what specific unexpected player argument was provided.

We can use `error` in `play-with-turns` by adding an `else` clause to the `cond`:

```
(define play-with-turns
  (lambda (game-state player)
    (cond ((over? game-state)
           (announce-winner player))
          ((equal? player 'human)
           (play-with-turns (human-move game-state) 'computer))
          ((equal? player 'computer)
           (play-with-turns (computer-move game-state) 'human))
          (else
           (error "player wasn't human or computer:" player)))))
```

Evaluating `(play-with-turns (make-game-state 5 8) 'humman)` still won't start the game. However, we'll see the message "player wasn't human or computer: humman," which will give us some explanation of what went wrong, unlike before.

We will write the procedure `computer-move` so that it uses the very simple strategy of removing one coin from the first pile. If the first pile is empty, the computer will remove one coin from the second pile. Note that if both piles are empty, the computer will still try to remove one coin from the second pile. However, this can never happen, because if both piles are empty, the game is over:

```
(define computer-move
  (lambda (game-state)
    (if (> (size-of-pile game-state 1) 0)
        (remove-coins-from-pile game-state 1 1)
        (remove-coins-from-pile game-state 1 2))))
```

Unfortunately, the human player has no way of knowing what strategy the computer is using and so has no idea what the state of the game is after the computer has moved. We must therefore add some output statements to tell the human player what the computer is doing. We will use two built-in Scheme procedures, `display` and `newline`. These procedures are best described by example, but roughly speaking, `display` takes a single argument that it immediately prints out as output, and `newline` takes no arguments and causes the output to go to the next line.

For example, the following procedure takes a game state and displays it in a reasonable manner:

```
(define display-game-state
  (lambda (game-state)
    (newline)
    (newline)
    (display "    Pile 1: ")
    (display (size-of-pile game-state 1))
    (newline)
    (display "    Pile 2: ")
    (display (size-of-pile game-state 2))
    (newline)
    (newline)))
```

Note that in two cases, we passed `display` a character string, because we wanted to control exactly what was output, as previously with `error`. Also, the definition of the Scheme programming language doesn't make any guarantees about the value (if any) returned by `display` and `newline`—they are only used for their effect.

How do we incorporate output into our program? Well, the game state should be displayed before each player makes a move, and the computer should inform the human player of its move. The first of these goals is most easily accomplished by adding one line to `play-with-turns`:

```
(define play-with-turns
  (lambda (game-state player)
    (display-game-state game-state)                  ;<-- output
    (cond ((over? game-state)
           (announce-winner player))
          ((equal? player 'human)
           (play-with-turns (human-move game-state) 'computer))
          ((equal? player 'computer)
           (play-with-turns (computer-move game-state) 'human))
          (else
           (error "player wasn't human or computer:" player)))))
```

In order to tell the human player what the computer is doing, we can add similar output statements. However, the output depends on which pile the computer uses. Because we also need to know which pile to use in the call to `remove-coins-from-pile`, we create a local variable by using a `let`:

```
(define computer-move
  (lambda (game-state)
    (let ((pile (if (> (size-of-pile game-state 1) 0)
                    1
                    2)))
      (display "I take 1 coin from pile ")
      (display pile)
      (newline)
      (remove-coins-from-pile game-state 1 pile))))
```

To write the procedure human-move, we need to get some input from the (human) player. We do this by first writing a procedure, prompt, which takes a "prompting string" and returns the number entered by the user:

```
(define prompt
  (lambda (prompt-string)
    (newline)
    (display prompt-string)
    (newline)
    (read)))
```

Here we used the basic Scheme input procedure read, which takes no arguments and returns whatever value the user enters. (The user's input must have the right form to be a Scheme value.) We are assuming that the user will type in an appropriate value, for example, when asked for the pile number, either 1 or 2.

We can use prompt in human-move to prompt for the pile number and the number of coins. Note that we use enclosing let statements in order to ensure that the user is always prompted for the pile first and the number of coins second. Depending on those answers, we remove the appropriate number of coins from the specified pile:

```
(define human-move
  (lambda (game-state)
    (let ((p (prompt "Which pile will you remove from?")))
      (let ((n (prompt "How many coins do you want to remove?")))
        (remove-coins-from-pile game-state n p)))))
```

Exercise 6.2

When you played Nim with another person and a gamekeeper, what did the gamekeeper do if you asked her to remove more coins from a pile than she could possibly remove? (For example, what if you asked her to remove six coins from a pile with only five coins in it?) If we want our computer version to work in a similar way, we need to build in some sort of error checking on the part of the input procedures. What would you modify in order to continue asking the user for another number if the number selected was illegal (both for coins and pile)? Is it better to do the error checking in human-move or prompt? Could you use a procedural parameter to good effect?

Finally, in order to finish the game, we need to be able to determine when the game is over and have the computer announce who the winner is. Note that we're assuming that the players play until the bitter end, so the game is over when both piles are empty:

```
(define total-size
  (lambda (game-state)
    (+ (size-of-pile game-state 1)
       (size-of-pile game-state 2))))

(define over?
  (lambda (game-state)
    (= (total-size game-state) 0)))

(define announce-winner
  (lambda (player)
    (if (equal? player 'human)
        (display "You lose. Better luck next time.")
        (display "You win. Congratulations."))))
```

There is the program. (For your convenience, we include the entire program in a sidebar. We don't include `display-game-state` and `total-size`, because we consider them to be part of the abstract data type game state.) Without having any idea of how we are going to represent our game states or implement the three operations `make-game-state`, `remove-coins-from-pile`, and `size-of-pile`, we've written all the procedures needed to play Nim. This illustrates one of the main advantages of data abstraction. We can develop the application of our data type independently of developing the implementation.

Exercise 6.3

The version of Nim we have just written designates the winner as the one taking the last coin. What needs to be changed in order to reverse this, that is, to designate the one taking the last coin as the loser?

6.3 Representations and Implementations

In order to actually use the program in the previous section, we need to implement the ADT of game states. This means that we need to find some way of representing game states and we need to figure out the algorithms and write the procedures for the three operations. To do this, let's think about physical (as opposed to electronic) ways of constructing piles. First of all, the fact that we used coins was totally irrelevant to how we played the game. We could have used packets of sugar, vertical hatch

Nim Program

```
(define play-with-turns
  (lambda (game-state player)
    (display-game-state game-state)
    (cond ((over? game-state)
           (announce-winner player))
          ((equal? player 'human)
           (play-with-turns (human-move game-state) 'computer))
          ((equal? player 'computer)
           (play-with-turns (computer-move game-state) 'human))
          (else
           (error "player wasn't human or computer:" player)))))

(define computer-move
  (lambda (game-state)
    (let ((pile (if (> (size-of-pile game-state 1) 0)
                    1
                    2)))
      (display "I take 1 coin from pile ")
      (display pile)
      (newline)
      (remove-coins-from-pile game-state 1 pile))))


(define prompt
  (lambda (prompt-string)
    (newline)
    (display prompt-string)
    (newline)
    (read)))

(define human-move
  (lambda (game-state)
    (let ((p (prompt "Which pile will you remove from?")))
      (let ((n (prompt "How many coins do you want to remove?")))
        (remove-coins-from-pile game-state n p)))))

(define over?
  (lambda (game-state)
    (= (total-size game-state) 0)))

(define announce-winner
  (lambda (player)
    (if (equal? player 'human)
        (display "You lose. Better luck next time.")
        (display "You win. Congratulations."))))
```

marks, or the horizontal and vertical score lines on a chocolate bar. Similarly, how we arranged our coins in two piles was also unimportant. We could have made two heaps, we could have made two neat lines, or we could have arranged the coins in two separate rectangular arrays. Some arrangements might make counting the number of coins in a pile easier, but as long as we can determine how many coins there were in each pile, how we arrange the piles doesn't matter. What really matters is the number of coins in each pile. Thus, in order to represent game states in Scheme, we can use two numbers that we glue together in some way. We consider four different ways to do so.

The first way of representing game states is based on the fact that, as humans, we can easily see the separate digits in a numeral. If we add the restriction that there can't be more than nine coins in each pile, we can use two-digit numbers to represent the two piles. The first digit will be the number of coins in the first pile and the second digit will be the number in the second pile. (For example, a game state of 58 would have five coins in the first pile and eight coins in the second pile.) To create a game state with n coins in the first pile and m coins in the second, we would just physically write those two digits together, nm. This number is $n \times 10 + m$. Therefore, we can implement the operations `make-game-state` and `size-of-pile` as follows:

```
(define make-game-state
  ;; assumes no more than 9 coins per pile
  (lambda (n m) (+ (* 10 n) m)))

(define size-of-pile
  (lambda (game-state pile-number)
    (if (= pile-number 1)
        (quotient game-state 10)
        (remainder game-state 10))))
```

Removing coins from a pile can be done in two different ways: either taking advantage of the particular representation we've chosen, or not. If we take advantage of our particular representation, in which pile 1 is represented by the tens place and pile 2 by the ones place, we can remove coins from a pile by subtracting the specified number of either tens or ones:

```
(define remove-coins-from-pile
  (lambda (game-state num-coins pile-number)
    (- game-state
       (if (= pile-number 1)
           (* 10 num-coins)
           num-coins))))
```

Alternatively, we can have `remove-coins-from-pile` first select the two pile numbers using `size-of-pile`, then subtract the number of coins from the appropriate one of them, and finally use `make-game-state` to glue them back together:

```
(define remove-coins-from-pile
  (lambda (game-state num-coins pile-number)
    (if (= pile-number 1)
        (make-game-state (- (size-of-pile game-state 1)
                            num-coins)
                         (size-of-pile game-state 2))
        (make-game-state (size-of-pile game-state 1)
                         (- (size-of-pile game-state 2)
                            num-coins)))))
```

This version of `remove-coins-from-pile` has the advantage that when we change representations, all we need to change are the algorithms for `make-game-state` and `size-of-pile`.

Exercise 6.4

The restriction that we can only use at most nine coins in each pile is somewhat unreasonable. A more reasonable one would be to limit the size of the piles to at most 99 coins. Change the implementation above so that it reflects this restriction.

Exercise 6.5

What happens if we try to remove more coins from a pile than are actually in the pile? For example, what would be the result of evaluating

```
(remove-coins-from-pile (make-game-state 3 2) 5 1)
```

Modify `remove-coins-from-pile` so that such a request would result in just removing all of the coins from the specified pile.

Exercise 6.6

What are some other ways of coping with errors?

The biggest problem with this way of gluing our two numbers together is that we must put some arbitrary restriction on the size of the numbers. This approach is fine when we can reasonably make this restriction, as in two-pile Nim. However, there are data types where we can't reasonably restrict the size of the components, for example, the budget of the U.S. government. Our second representation (theoretically) gets around this restriction.

In this representation we will use integers of the form $2^n \times 3^m$, where n is the number of coins in the first pile and m is the number of coins in the second. Constructing a game state is quite easy, using the built-in procedure `expt`:

```
(define make-game-state
  (lambda (n m) (* (expt 2 n) (expt 3 m))))
```

Getting at the component parts of a game state is not so bad either, using the procedure in the following exercise.

Exercise 6.7

Look back at the procedures for calculating the exponent of 2 in a number that you wrote in Exercise 2.12 on page 40 and Exercise 3.2 on page 54. Generalize one of these to a procedure `exponent-of-in` such that `(exponent-of-in` n m`)` returns the exponent of the highest power of n that divides evenly into m.

With this procedure, we can easily write `size-of-pile`:

```
(define size-of-pile
  (lambda (game-state pile-number)
    (if (= pile-number 1)
        (exponent-of-in 2 game-state)
        (exponent-of-in 3 game-state))))
```

This implementation has two drawbacks. The first is that accessing the number of items in a pile does not take constant time. Instead, the time depends linearly on the size of the pile. The second drawback is that the integers representing the game states get very big very rapidly. This is not so important when we're implementing game states; however, when we implement a data structure where the component parts are often big numbers, this method would result in representations too large to fit in the computer's memory.

The first two implementations use integers to represent the values that a game state could have. Note that some integers, such as 36, could be used in either representation. That is, a game state represented by 36 could be either one where the first pile has three coins and the second has six (in the first representation) or one where each pile has two coins (in the second representation). In fact, the only way we know what game state is represented by a specific integer is by knowing which representation we're using. The three operations are what interpret the values for us. The need for a consistent interpretation is one of the reasons that we use *only* the specified operations to manipulate values in an abstract data type.

Our third representation for game states uses procedures instead of integers to represent the game states. So when we apply `make-game-state`, the result should be a procedure, because `make-game-state` creates game states. Now, a game state has two observable properties, the number of coins in each of the two piles. Because the only property that we can observe about a procedure is its return value, the procedure generated by `make-game-state` should have two different return values. Therefore, this procedure should have at least one argument so that we can have some way of controlling which of the two values should be returned. Because there is no need for more than one argument, we want `(make-game-state n m)` to produce a procedure with one argument that sometimes returns `n` and sometimes returns `m`. What should this procedure be? We have complete freedom to choose. It could return `n` when its argument is odd and `m` when its argument is even; it could return `n` for positive values of its argument and `m` for negative values, or whatever. For now, let's arbitrarily decide to use the first option:

```
(define make-game-state
  (lambda (n m)
    (lambda (x)
      (if (odd? x)
          n
          m))))
```

Now we need to write the procedure `size-of-pile`. If we think about how `make-game-state` and `size-of-pile` should work together, we can write two equations:

(size-of-pile (make-game-state *n* *m*) 1) = *n*

(size-of-pile (make-game-state *n* *m*) 2) = *m*

Because `make-game-state` produces a procedure that returns *n* when it gets an odd argument and *m* when it gets an even one, and 1 happens to be odd and 2 even, one way to write `size-of-pile` is to have it simply apply its game state argument (which is a procedure) to the pile number argument:

```
(define size-of-pile
  (lambda (game-state pile-number)
    (game-state pile-number)))
```

Exercise 6.8

Verify that the two equations relating `make-game-state` and `size-of-pile` just given hold for the procedural representation.

This procedural representation of game states has the advantages of each of the previous ones, without the corresponding disadvantages. In other words, we don't need to restrict the size of the piles, and the procedure `size-of-pile` will still generate a constant-time process. We can do little to improve on this representation (but we still have a fourth representation to present anyway, for reasons that will become clear).

At this point, having seen two representations of game states as integers and one as procedures, you may be confused. You may be wondering just what *is* a game state? Surely there should be some definite thing that I can look at and say, *this* is a game state. There are two answers to this. The first answer is to say, Yes, at any time there is one kind of thing that is a game state, which depends on which matched set of constructor and selector definitions has been evaluated. If, for example, you've evaluated the most recent set, game states are procedures.

However, another, better answer to the question above is: Don't worry about what a game state is in that sense. Pretend a game state is a whole new kind of thing. This new kind of thing is produced by `make-game-state`, and you can find information out about it using `size-of-pile`.

In other words, instead of saying "a game state is an integer" or "a game state is a procedure," we'll say "a game state is what `make-game-state` creates (and `size-of-pile` expects)." If the procedures that operate on game states are happy with something, it is a game state. This is worth highlighting as a general principle:

The data-abstraction principle (or, the operational stance): If it acts like an X (i.e., is suitable for operations that operate on X's), it *is* an X.

One related question that you may have is what if you do a game-state operation to something that isn't a game state? Or what if you do some other operation to a game state, other than those that make sense for game states? For example, what if you evaluate any of the following?

```
(size-of-pile (* 6 6) 1)

(size-of-pile sqrt 1)

(sqrt (make-game-state 3 6))

((make-game-state 3 6) 7)
```

Again, there are two answers: (1) That depends on the details of which representation was chosen and how the operations were implemented and (2) Don't do that. As unsatisfactory as this second answer may sound, it generally is the more useful one. We can sum this all up as follows:

> **The strong data-abstraction principle:** If it acts like an X, it *is* an X. Conversely, if it is an X, only expect it to act in X-like ways.

When we discussed our procedural representation of game states above, we mentioned that we'd be hard pressed to improve upon it. So, why then do we still have a fourth representation up our sleeves? The answer is that although we'd be hard pressed to do better, someone else might not be. In particular, whoever designed your Scheme system might have some slightly more efficient way of gluing two values together, using behind-the-scenes magic.

So, what we'll do for our final implementation of game states is turn data abstraction to our advantage and use a built-in data type in Scheme called *pairs*. Whoever built your Scheme system represented pairs as efficiently as they possibly could. Exactly how they did this might vary from one Scheme system to another. However, thanks to data abstraction, all we have to know about pairs is what operations produce them and answer questions about them. The basic operations that Scheme provides to deal with pairs include a procedure for making a pair, a procedure for determining what the first element in a pair is, and a procedure for determining what the second element is. These have extremely weird names:

- `cons` takes any two objects and glues them together into a pair.
- `car` takes a pair of objects and returns the first object.
- `cdr` (pronounced "could-er") takes a pair of objects and returns the second object.

The name `cons` is easy to understand: It is short for constructor, and sure enough, the procedure called `cons` is the constructor for pairs. But what about the names `car` and `cdr`, which are the names of the selectors for the pair type? These two names are reminders that even smart people sometimes make dumb mistakes. The people who developed an early predecessor of Scheme (at MIT, in the late 1950s) chose to represent pairs on the IBM 704 computer they were using in such a way that the selectors could be implemented using low-level features of the IBM 704 hardware that had the acronyms CAR and CDR (for contents of address part of register and contents of decrement part of register). So, rather than call the selectors first and second, left and right, or one and the-other, they named them `car` and `cdr`, after how they were implemented on the 704. (One of the implementers later wrote that "because of an unfortunate temporary lapse of inspiration, we couldn't think of any other names.") In so doing, they were violating the spirit of the strong data-abstraction principle, by basing the abstract interface to the data type on their particular representation. (Of course, in a certain sense, "left" and "right" are just as representation-specific, because they are based on the way we westerners write things down on a page.) A few months after the original naming, the perpetrators of `car` and `cdr` tried renaming the operations, but to no avail: Other users were already accustomed to `car` and `cdr` and unwilling to change. At any rate, `car` and `cdr` have survived for over three decades as the names for the two selector operations on pairs, and so they are likely to survive forever as permanent reminders of how *not* to name operations.

As we said before, `cons` takes two objects and glues them together in a pair. How do we know which order it uses? In other words, if we use `cons` to glue a and b together in a pair, which will be the first object of that pair and which will be the second? What we're really asking here is how `cons`, `car`, and `cdr` work together. The answer is best described by two equations:

$$\texttt{(car (cons } a\ b\texttt{))} = a$$

$$\texttt{(cdr (cons } a\ b\texttt{))} = b$$

These say that if you cons two objects together into a pair, the first object becomes the car of the pair and the second object becomes the cdr of the pair. (We've used this paragraph to introduce you to the way Schemers talk about pairs. We use cons as a verb, as in "cons two objects together," and we talk about the car and cdr of a pair, instead of the first and second components of it.)

Pairs of this sort are a natural way to implement game states, because a game state is most easily thought of as a pair of numbers. Thus, our two operations would be

```
(define make-game-state
  (lambda (n m) (cons n m)))

(define size-of-pile
  (lambda (game-state pile-number)
    (if (= pile-number 1)
        (car game-state)
        (cdr game-state))))
```

Note that in the definition of `make-game-state`, we simply apply `cons` to the two arguments. In other words, `make-game-state` does exactly the same thing as `cons` and hence can simply be the same procedure as `cons`:

```
(define make-game-state cons)
```

The way Scheme displays pairs if left to its own devices is in general quite confusing. Therefore, when you are using pairs to represent something else, like

Game State ADT Implementation

```
(define make-game-state
  (lambda (n m) (cons n m)))

(define size-of-pile
  (lambda (game-state pile-number)
    (if (= pile-number 1)
        (car game-state)
        (cdr game-state))))

(define remove-coins-from-pile
  (lambda (game-state num-coins pile-number)
    (if (= pile-number 1)
        (make-game-state (- (size-of-pile game-state 1)
                            num-coins)
                         (size-of-pile game-state 2))
        (make-game-state (size-of-pile game-state 1)
                         (- (size-of-pile game-state 2)
                            num-coins)))))

(define display-game-state
  (lambda (game-state)
    (newline)
    (newline)
    (display "    Pile 1: ")
    (display (size-of-pile game-state 1))
    (newline)
    (display "    Pile 2: ")
    (display (size-of-pile game-state 2))
    (newline)
    (newline)))

(define total-size
  (lambda (game-state)
    (+ (size-of-pile game-state 1)
       (size-of-pile game-state 2))))
```

a game state, you should always look at them using an appropriate procedure like `display-game-state`.

Again for your convenience, we include all of the ADT procedures in a sidebar, using just the pair implementation. Together with the Nim program on page 144, this is a full, working program. In the next section we examine the changes needed for three-pile Nim; in the final section we extend the two-pile program so that the computer can use other strategies for selecting its moves.

6.4 Three-Pile Nim

Now suppose we want to write a program that plays Nim with three piles instead of two. We'll need to extend the game state ADT so that it uses three piles instead of two. This means that the procedure `make-game-state` will get three parameters instead of two and needs to glue all three together somehow, depending on which representation we use. If we use one of the numerical representations, the main change would be to use three-digit numbers instead of two or to use numbers of the form $2^n \times 3^m \times 5^k$ instead of $2^n \times 3^m$. The procedural representation is equally easy to change: The procedures created by `make-game-state` must be able to return any of three values instead of just two. But, at first glance, using pairs seems impossible. After all, a pair has only two "slots," whereas we have three numbers, and we can't put three things into two slots.

Wait a minute—of course we can put three things into two slots, as long as we put two of them in one slot and the third in the other slot. How do we put two things into one slot, though? Each slot is allowed to contain only one thing. But there are no restrictions on what that one thing could be; for example, *it could be another pair*. Thus, in order to make a three-pile game state, we'll cons together two of the numbers and cons that together with the remaining one.

Does it matter which order we cons things together? The answer to that is no, sort of. We can cons the three numbers together in any order we like as long as whenever we ask for the number of coins in a particular pile, we get back the correct number. In other words, the procedures `make-game-state` and `size-of-pile` need to work together correctly—the constructors and selectors must agree on the representation, as usual.

Exercise 6.9

Write the equations for three-pile game states that correspond to those given earlier for two-pile game states.

For example, suppose we cons the third pile onto the cons of the first and the second:

```
(define make-game-state
  (lambda (n m k) (cons k (cons n m))))
```

Then how do we pull a game state apart? If we want the size of the third pile, we just need the first element of the game state (i.e., `(car game-state)`). But getting the size of the first or second pile is somewhat trickier, because those two numbers are bundled together. We can get this pair of first and second piles by taking `(cdr game-state)`. Then, to get the size of the first pile, say, we need to take the car of that pair, or `(car (cdr game-state))`. Similarly, to get the size of the second pile, we'll need the cdr of that pair, or `(cdr (cdr game-state))`. Putting this all together gives

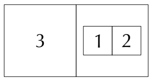
```
(define size-of-pile
  (lambda (game-state pile-number)
    (cond ((= pile-number 3)
           (car game-state))
          ((= pile-number 1)
           (car (cdr game-state)))
          (else ;pile-number must be 2
           (cdr (cdr game-state))))))
```

Exercise 6.10

Check that this implementation actually works (i.e., that the constructor and selector actually do agree on the representation).

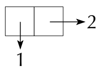
To help clarify how we get at the components of a three-pile game state, we can draw some pictures. Evaluating an expression such as `(cons 1 2)` results in a pair whose components are the numbers 1 and 2. If we think of that pair as having two slots that have been filled in with those numbers, the picture that comes to mind is $\boxed{1}\boxed{2}$.

Similarly, when we evaluate `(define gs (cons 3 (cons 1 2)))`, we know that `gs` is a pair whose first component is the number 3 and whose second component is the pair containing the numbers 1 and 2. Thus our picture would look like this:

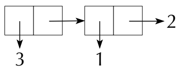


Now to get at that 2 in `gs`, we need to look at the second component of `gs`. This is itself the pair $\boxed{1}\boxed{2}$, and so we need to get the second component of this pair. Therefore we must evaluate `(cdr (cdr gs))`.

Although these pictures are quite helpful for understanding data entities that have three components, they quickly become unwieldy to draw if we start building any bigger structures. We can solve this problem by drawing the boxes a standard, small size, putting the contents outside, and using arrows to point to the contents. For a simple pair such as the value of `(cons 1 2)`, moving the contents out leads to



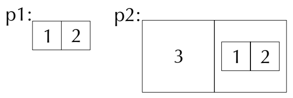
and a pair such as the value of `(cons 3 (cons 1 2))` would look like



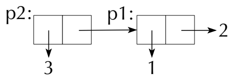
In addition to solving the problem of unwieldiness, moving the contents out of the boxes makes it easier to see what portion of a structure is reused or "shared." For example, if we evaluate the two definitions:

```
(define p1 (cons 1 2))
(define p2 (cons 3 p1))
```

and use our original style of drawing pairs, we get the picture



which seems to indicate that there are three pairs—at odds with the fact that `cons` was applied only twice. (We know that each time `cons` is applied, exactly one pair is created.) In contrast, with our new, improved style of diagram with the contents moved out of the boxes, we can draw



and now it is clear that only one new pair was produced by each of the two applications of `cons`.

Exercise 6.11

Now that we have the main constructor and selector for the three-pile game state ADT, we need to change the procedures `remove-coins-from-pile`, `total-size`, and `display-game-state` appropriately. Do so.

Exercise 6.12

What will go wrong if we use the existing `computer-move` with three-pile game states? Change `computer-move` so that it works correctly with three-pile game states.

6.5 An Application: Adding Strategies to Nim

In this section, we return to the two-pile version of Nim for simplicity's sake. (Also, we like playing the chocolate bar version.) Much of what we do here easily extends to three-pile Nim. However, finding a winning strategy for three piles is more challenging.

The computer's strategy of removing one coin from the first nonempty pile is not very intelligent. Although we might initially enjoy always winning, eventually it gets rather boring. Is there some way of having the computer use a better strategy? Or perhaps, could we program several different strategies? In that case, what would a strategy be?

If you think about it, a strategy is essentially a procedure that, when given a particular game state, determines how many coins to remove from which pile. In other words, a strategy should return two numbers, one for the number of coins and one for the pile number. Because procedures can return just one thing, we have a real problem here. One way to solve it is to think of these two numbers as an instruction describing the appropriate move to make. We can create a new data type, called *move instruction*, that glues together the number of coins to remove and the pile number to remove them from. We can then view a strategy as a procedure that takes a game state and returns the instruction for the next move.

Exercise 6.13

In this exercise, we will construct the move instruction data type and modify our program appropriately.

a. First, you need to decide what the basic operations for move instructions should be. There are several ways to do this. You can think about how move instructions are going to be used—in particular, what information other procedures need to know about a given move instruction. You can think how you would fully describe a move instruction to someone. You can model move instructions on game states. In any case, it should be clear that you will need one constructor and two selectors. Give a specification for move instructions similar to the one we gave the game state data type. That is, what is the name of each operation, what sort of parameters does it take, and what sort of result does it return? (We will call the move instruction constructor `make-move-instruction` in the following discussion and will assume it takes the number of coins as its first argument and the pile number as its second argument, so you might want to do the same.) Can you also write equations that describe how the operations are related?

b. Choose a representation and implement these procedures.

c. We have used the procedure `remove-coins-from-pile` to progress from one game state to the next, passing to it the current game state and the two integers that specify the move. But with our *move instruction* data type, it makes more sense to have a procedure that is passed the current game state and the move instruction and returns the next game state. We could call the procedure just `remove`; alternatively, we could call it `next-game-state`. The latter seems more descriptive.

Write the procedure `next-game-state`, which takes two parameters, a game state and a move instruction, and returns a game state. You will need to change `computer-move` and `human-move` so that they correctly call `next-game-state` instead of `remove-coins-from-pile`. Run your program to make sure everything works as before.

Type Checking

Both game states and move instructions are compound data types with exactly two components and integers as the values of these components. Let's suppose that we've decided to implement both of these types by using Scheme's pairs. In this case, the value of the expression `(cons 2 1)` could represent either a game state or a move instruction. This can create some havoc in our programs if we're not careful. For example, suppose you wanted to find the next game state after taking one coin from pile one, starting in a state with five and eight coins in the two piles. At first glance, the following looks reasonable:

```
(display-game-state
  (next-game-state (make-move-instruction 1 1)
                   (make-game-state 5 8)))
```

However, if you try this, you'll get output like the following:

```
Pile 1: 1
Pile 2: -4
```

What went wrong? We got the order of the parameters to `next-game-state` backward. Although the principle of data abstraction tells *us* to think of things

Continued

Type Checking (Continued)

as move instructions or game states, rather than as pairs of integers, the Scheme system regrettably thinks of both as just pairs of integers. Therefore, although we can see that we got the move instruction and game state backward, the program got exactly what it expected: two pairs of integers. This kind of error is particularly hard to catch. One way to find such errors is by doing a process called *type checking*. The basic idea is that every piece of data has a particular type, such as integer, game state, move instruction, etc. The type of a procedure is described by saying that it is a procedure that takes certain types of arguments and returns a certain type of result. For example, `make-move-instruction` is a procedure that takes two integers and returns a move instruction, wheras `next-game-state` takes a game state and a move instruction and returns a game state. We can check that a procedure application is probably correct by checking that the types of the arguments are consistent with those expected by the procedure. For example, we know that `(make-move-instruction 1 1)` has probably been called correctly, because its two arguments are integers. Notice that its return value will be a move instruction. On the other hand, the call

```
(next-game-state (make-move-instruction 1 1)
                 (make-game-state 5 8))
```

is definitely incorrect because `next-game-state` gets a move instruction and a game state for arguments when it is supposed to get a game state and a move instruction. Note that type checking only catches errors that are caused by using arguments that are the wrong types. It won't catch the error in `(make-move-instruction 5 6)`, where the pile number is too big, unless we use a more refined notion of type, where we can say that the second argument is "an integer in the range from 1 to 2" rather than just that it is an integer.

If we then view strategies as procedures that, when given a particular game state, return the instruction for the next move, we could write the simple strategy currently used by the computer as follows:

```
(define simple-strategy
  (lambda (game-state)
    (if (> (size-of-pile game-state 1) 0)
        (make-move-instruction 1 1)
        (make-move-instruction 1 2))))
```

But how do we need to change our program in order to incorporate various strategies into it? Certainly the procedure `computer-move` must be changed: In addition to the game state, it must be passed the strategy to be employed. But this means `play-with-turns` must also be changed, because it calls `computer-move`: It must have an additional argument that indicates the computer's strategy. If you do this correctly, an initial call of the form

```
(play-with-turns (make-game-state 5 8) 'human simple-strategy)
```

should play the game as before.

Exercise 6.14

In this exercise, you will change the procedures `computer-move` and `play-with-turns` as indicated previously. After making these changes, test the program by making the previous initial call.

a. Modify the procedure `computer-move` so that it takes an additional parameter called `strategy` and uses it appropriately to make the computer's move. Remember that when the strategy is applied to a game state, a move instruction is returned. This can then be passed on to `next-game-state`.

b. Modify `play-with-turns` so that it also has a new parameter (the computer's strategy), modifying in particular the call to `computer-move` so that the strategy is employed. Note that you must make additional changes to `play-with-turns` in order that the strategy gets "passed along" to the next iteration.

Now we can add a variety of different strategies to our program. This amounts to writing the various strategies and then calling `play-with-turns` with the strategies we want. We ask you in the next few exercises to program various strategies.

Exercise 6.15

Write a procedure `take-all-of-first-nonempty` that will return the instruction for taking all the coins from the first nonempty pile.

Exercise 6.16

Write a procedure `take-one-from-random-pile` that implements the following "random" strategy: randomly select a nonempty pile and then remove one coin from it. Randomness can be simulated using the `random` procedure, which should be pre-defined in any Scheme used with this book (although it isn't specified by the R^4RS standard for Scheme). If n is a positive integer, a call of the form `(random n)` will return a random integer between 0 and $n - 1$, inclusive. (Actually, it returns a so-called *pseudo-random integer*; pseudo-random integers are produced systematically and hence are not random, but sequences of consecutive pseudo-random integers have many of the same statistical properties that sequences of random integers do.)

Exercise 6.17

Take the previous exercise one step further by writing a procedure that, when given a particular game state, will return a move instruction where both components are chosen at random. Remember to ensure that the move instruction returned is a valid one. In particular, it should not suggest a move that takes coins from an empty pile.

Exercise 6.18

If we consider the chocolate bar version of Nim, we can describe a strategy that allows you to win whenever possible. Remember that in this version, the players alternate breaking off pieces of the bar along a horizontal or a vertical line, and the person who gets the last square of chocolate loses (so the person who makes the last possible break wins, just as the person who takes the last coin wins). If it's your turn and the chocolate bar is not square, you can always break off a piece that makes the bar into a square. If you do so, your opponent must make it into a nonsquare. If you always hand your opponent a square, he will get smaller and smaller squares, leading eventually to the minimal square (i.e., the poisoned square). Write a procedure which implements this strategy in two-pile Nim. What action should it take if presented with (the equivalent of) a square chocolate bar?

Exercise 6.19

Suppose you want to randomly intermingle two different strategies. How can this be done? The answer is with higher-order programming. Write a procedure `random-mix-of` that takes two strategies as arguments and returns the strategy that randomly chooses between these two procedures each turn. Thus, a call of the form

```
(play-with-turns (make-game-state 5 8)
                 'human
                 (random-mix-of simple-strategy
                                take-all-of-first-nonempty))
```

would randomly choose at each turn between taking one coin or all the coins from the first nonempty pile.

Exercise 6.20

Those of us with a perverse sense of humor enjoy the idea of the computer playing games with itself. How would you modify `play-with-turns` so that instead of having the computer play against a human, it plays against itself, using any combination of strategies?

Exercise 6.21

By adding an "ask the human" strategy, the introverted version of `play-with-turns` from the preceding exercise can be made to be sociable again. In fact, it can even be turned into a gamekeeper for two human players. Demonstrate these possibilities.

Review Problems

Exercise 6.22

Suppose we decide to implement an ADT called *Interval* that has one constructor `make-interval` and two selectors `upper-endpoint` and `lower-endpoint`. For example,

```
(define my-interval (make-interval 3 7))

(upper-endpoint my-interval)
7
```

defines `my-interval` to be the interval $[3,7]$ and then returns the upper endpoint of `my-interval`.

Note that we are saying nothing about how *Interval* is implemented. Your work below should only use the constructor and selectors.

a. Write a procedure `mid-point` that gets an interval as an argument and returns the midpoint of that interval. For example, supposing that `my-interval` is as just defined:

```
(mid-point my-interval)
5
```

b. Write a procedure `right-half` that gets an interval as an argument and returns the right half of that interval. Again supposing that `my-interval` is as just defined:

```
(right-half my-interval)
```

returns the interval $[5, 7]$.

Exercise 6.23

A three-dimensional (3D) vector has x, y, and z coordinates, which are numbers. 3D vectors can be constructed and accessed using the following abstract interface:

```
(make-3D-vector x y z)
(x-coord vector)
(y-coord vector)
(z-coord vector)
```

a. Using this abstract interface, define procedures for adding two vectors, finding the dot-product of two vectors, and scaling a vector by a numerical scaling factor. (The sum of two vectors is computed by adding each coordinate independently. The dot-product of the vectors (x_1, y_1, z_1) and (x_2, y_2, z_2) is $x_1x_2 + y_1y_2 + z_1z_2$. To scale a vector, you multiply each coordinate by the scaling factor.)

b. Choose a representation for vectors and implement `make-3D-vector`, `x-coord`, `y-coord`, and `z-coord`.

Exercise 6.24

Suppose we wished to keep track of which classrooms are being used at which hours for which classes. We would want to have a compound data structure consisting of three parts:

- A classroom designation (e.g. OH321)
- A course designation (e.g. MC27)
- A time (e.g. 1230)

Assume that rooms and courses are to be represented by symbols and the times are to be represented as numbers. The interface is to look like this:

```
(make-schedule-item 'OH321 'MC27 1230)

(room (make-schedule-item 'OH321 'MC27 1230))
OH321

(course (make-schedule-item 'OH321 'MC27 1230))
MC27
```

```
(time (make-schedule-item 'OH321 'MC27 1230))
1230
```

Use a procedural representation to write a constructor and three selectors for this schedule-item data type.

Exercise 6.25

We previously said that each move in Nim "transforms the game into a smaller game, if we measure the size of the game in terms of the total number of coins." This raises the possibility that we could define a predicate `game-state-<` that would compare two game states and determine whether the first is smaller (in the sense of having a smaller total number of coins). Similarly, we could define `game-state->`, `game-state-=`, `game-state-<=`, etc. Define a general purpose procedure called `make-game-state-comparator` for making procedures like those just described, given the numerical comparison procedure (e.g., `<`) to use. Here are some examples of its use, together with examples using the comparators it makes:

```
(define game-state-< (make-game-state-comparator <))

(game-state-< (make-game-state 3 7) (make-game-state 1 12))
#t

(define game-state-> (make-game-state-comparator >))

(game-state-> (make-game-state 3 7) (make-game-state 1 12))
#f

(game-state-> (make-game-state 13 7) (make-game-state 1 12))
#t
```

Exercise 6.26

Recall that when you worked with fractals in Section 4.3, many of the procedures required parameters that represented the *x* and *y* coordinates of points in an image; for example, the built-in procedure `line` required coordinates for the starting point and the ending point, and the procedure `triangle` required coordinates for the triangle's three vertices.

a. Use `cons`-pairs to implement a *point* ADT. You should write a constructor `make-point`, that takes two arguments representing the *x* and *y* coordinates and returns the corresponding point and two selectors `x-coord` and `y-coord` that take a point and return the corresponding coordinate.

b. Write a procedure `distance` that takes two points and returns the distance between them. Use the selectors `x-coord` and `y-coord` rather than relying on the specific representation from part a. For example, you should see the following interaction:

```
(define pt-1 (make-point -1 -1))

(define pt-2 (make-point -1 1))

(distance pt-1 pt-2)
2
```

Remember: The distance between the points with coordinates (x_1, y_1) and (x_2, y_2) is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

Chapter Inventory

Vocabulary

atomic data
compound data
constructor
selector
representation
data abstraction
abstract data type (ADT)
porting
procedural representation
pseudo-random integer
type checking

Slogans

The data abstraction principle
(or, the operational stance)
The strong data-abstraction principle

Abstract Data Types

game states
move instructions
pairs
intervals
3D vectors
schedule items
points

New Predefined Scheme Names

The dagger symbol (†) indicates a name that is not part of the R^{4}RS standard for Scheme.

`equal?`	`cons`
`error`†	`car`
`display`	`cdr`
`newline`	`random`†
`read`	

New Scheme Syntax

symbols
quote mark
character strings

Scheme Names Defined in This Chapter

`size-of-pile`	`upper-endpoint`
`remove-coins-from-pile`	`lower-endpoint`
`make-game-state`	`mid-point`
`play-with-turns`	`right-half`
`computer-move`	`make-3D-vector`
`human-move`	`x-coord`
`over?`	`y-coord`
`announce-winner`	`z-coord`
`display-game-state`	`make-schedule-item`
`prompt`	`room`
`total-size`	`course`
`exponent-of-in`	`time`
`make-move-instruction`	`game-state-<`
`next-game-state`	`game-state->`
`simple-strategy`	`game-state-=`
`take-all-of-first-nonempty`	`game-state-<=`
`take-one-from-random-pile`	`make-game-state-comparator`
`random-mix-of`	`make-point`
`make-interval`	`distance`

Sidebars

Nim Program
Game State ADT Implementation
Type Checking

Notes

Bouton's seminal article on Nim is [8]. The chocolate-bar version of the game was presented as a puzzle by Dr. Ian Stewart on the Canadian Broadcasting Company's program "Quirks and Quarks" [10].

The history of the names `car` and `cdr` is told by Steve Russell (one of the originators of those names) in [45].

The idea of augmenting a game with strategy procedures and higher-order strategies comes from a lunar lander programming assignment developed by Abelson, Sussman, and friends at MIT [1].