5 Data

From a bit to a few hundred megabytes, from a microsecond to half an hour of computing confronts us with the completely baffling ratio of 10^9! .... By evoking the need for deep conceptual hierarchies, the automatic computer confronts us with a radically new intellectual challenge that has no precedent in our history.
Edsger Dijkstra

For all the programs so far, we have been limited to simple data such as numbers and Booleans. We call this *scalar* data since it has no structure. As we saw in Chapter 1, we can represent all discrete data using just (enormously large) whole numbers. For example, we could represent the text of a book using only one (very large!) number, and manipulate the characters in the book by changing the value of that number. But, it would be very difficult to design and understand computations if we only use numbers to represent data.

Instead, we need more complex data structures that allow us to model the structured data our problem concerns with structures our programs can more directly manipulate. We want to represent data in ways that allow us to think about the problem we are trying to solve, rather than the low-level details of how data is represented and manipulated.

In this chapter, we develop techniques for building complex data structures and for defining procedures that manipulate structured data, and introduce data abstraction as a tool for managing program complexity.

5.1 Types

All data in a program has an associated type. Internally, all data is stored just as a sequence of bits, so the type of the data is important to understand what it means. We have seen several different types of data already: Numbers, Booleans, and Procedures (we use initial capital letters to signify a datatype).

A datatype defines a set (often infinite) of possible values. The Boolean

datatype contains the two Boolean values, *true* and *false*. The Number type includes the infinite set of all whole numbers (it also includes negative numbers and rational numbers). We think of the set of possible Numbers as infinite, even though on any particular computer there is some limit to the amount of memory available, and hence, some largest number that can be represented. On any real computer, the number of possible values of any data type is always finite. But, we can imagine a computer large enough to represent any given number.

The type of data determines what can be done with it. For example, a Number can be used as one of the inputs to the primitive procedures $+$, $*$, and $=$. A Boolean can be used as the first sub-expression of an if-expression and as the input to the *not* procedure (note that the way *not* is defined, it can also take a Number as its input, but for all Number value inputs the output is false), but cannot be used as the input to $+$, $*$, or $=$.[1]

A Procedure can be the first sub-expression in an application expression, and can be passed as an input to the *fcompose* procedure (defined in the previous chapter). There are infinitely many different types of Procedures, since the type of a Procedure depends on its input and output types. For example, recall *bigger* procedure from Chapter 3:

(**define** (*bigger a b*) (**if** ($>$ *a b*) *a b*))

It takes two Numbers as input and produces a Number as output. We denote this type as:

$$\text{Number} \times \text{Number} \rightarrow \text{Number}$$

The inputs to the procedure are shown on the left side of the arrow. The type of each input is shown in order, separated by the $\times$ symbol.[2] The output type is given on the right side of the arrow.

From its definition, we know the *bigger* procedure takes two inputs. But how do we know the inputs must be Numbers and the output is a Number?

The body of the *bigger* procedure is an if-expression with the predicate expression ($>$ *a b*). This applies the $>$ primitive procedure to the two inputs.

[1] The primitive procedure *equal?* is a more general comparison procedure that can take as inputs any two values, so could be used to compare Boolean values. For example, (*equal? false false*) evaluates to true and (*equal? true* 3) is a valid expression that evaluates to *false*.

[2] The notation using $\times$ to separate input types makes sense if you think about the number of different inputs to a procedure. For example, consider a procedure that takes two Boolean values as inputs, so its type is Boolean $\times$ Boolean $\rightarrow$ Value. Each Boolean input can be one of two possible values. If we combined both inputs into one input, there would be 2×2 different values needed to represent all possible inputs.

The type of the $>$ procedure is Number $\times$ Number $\rightarrow$ Boolean. So, for the predicate expression to be valid, its inputs must both be Numbers. This means the input values to *bigger* must both be Numbers. We know the output of the *bigger* procedure will be a Number by analyzing the consequence and alternate sub-expressions: each evaluates to one of the input values, which must be a Number.

Starting with the primitive Boolean, Number, and Procedure types, we can build arbitrarily complex datatypes. In this chapter, we will introduce mechanisms for building complex datatypes by combining the primitive datatypes.

Exercise 5.1. Describe the type of each of these expressions.

a. 17

b. (**lambda** (*a*) ($>$ *a* 0))

c. ((**lambda** (*a*) ($>$ *a* 0)) 3)

d. (**lambda** (*a*) (**lambda** (*b*) ($>$ *a* *b*)))

e. (**lambda** (*a*) *a*)

Exercise 5.2. For each part, define or identify a procedure that has the given type.

a. Number $\times$ Number $\rightarrow$ Boolean

b. Number $\rightarrow$ Number

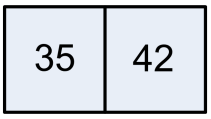
c. (Number $\rightarrow$ Number) $\times$ (Number $\rightarrow$ Number) $\rightarrow$ (Number $\rightarrow$ Number)

d. Number $\rightarrow$ (Number $\rightarrow$ (Number $\rightarrow$ Number))

5.2 Pairs

The simplest structured data construct is a *Pair*. A Pair packages two values together. We draw a Pair as two boxes, each containing a value. We call each box of a Pair a *cell*. Here is a Pair where the first cell has the value 35 and the second cell has the value 42:

Pair



Scheme provides built-in procedures for constructing a Pair, and for extracting each cell from a Pair:

- *cons*: Value × Value → Pair — evaluates to a Pair where the first cell is the first input value and the second cell is the second input value. The Value inputs can be of any type.
- *car*: Pair → Value — evaluates to the first cell of the input, which must be a Pair.
- *cdr*: Pair → Value — evaluates to the second cell of input, which must be a Pair.

These rather unfortunate names come from the original LISP implementation on the IBM 704. The name *cons* is short for "construct". The name *car* is short for "*C*ontents of the *A*ddress part of the *R*egister" and the name *cdr* (pronounced "could-er") is short for "Contents of the *D*ecrement part of the *R*egister". The designers of the original LISP implementation picked the names because of how pairs could be implemented on the IBM 704 using a single register to store both parts of a pair, but it is a mistake to name things after details of their implementation (see Section 5.6). Unfortunately, the names stuck and continue to be used in many LISP-derived languages, including Scheme.

We can construct the Pair shown in the previous diagram by evaluating (*cons* 35 42). DrScheme will display a Pair by printing the value of each cell separated by a dot: (35 . 42). The interactions below show some example uses of *cons*, *car*, and *cdr*.

```
> (define mypair (cons 35 42))
> mypair
(35 . 42)
> (car mypair)
35
> (cdr mypair)
42
```

The values in the cells of a Pair can be any values, including other Pairs!

For example,

```
(define doublepair (cons (cons 1 2) (cons 3 4)))
```

defines a Pair where each cell of the Pair is itself a Pair.

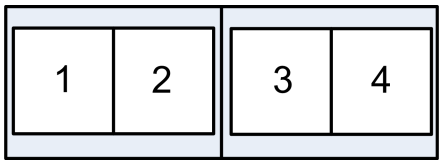
We can use the *car* and *cdr* procedures to access components of our nested *doublepair* structure: (*car doublepair*) evaluates to the Pair (1 . 2), and (*cdr doublepair*) evaluates to the Pair (3 . 4).

We can compose multiple *car* and *cdr* applications to extract components from the nested pairs:

```
> (cdr (car doublepair))
2
> (car (cdr doublepair))
3
> ((fcompose cdr cdr) doublepair)
4
> (car (car (car doublepair)))
car: expects argument of type <pair>; given 1
```

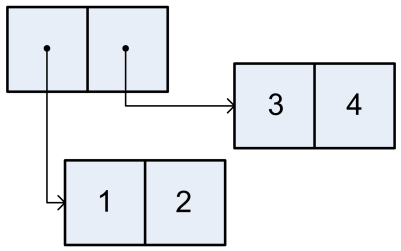
The last expression produces an error when it is evaluated since *car* is applied to the scalar value 1. The *car* and *cdr* procedures can only be applied to an input that is a Pair. Hence, an error results when we attempt to apply *car* to a scalar value. This is an important property of data: the type of data (e.g., a Pair) defines how it can be used (e.g., passed as the input to *car* and *cdr*). Every procedure expects a certain type of inputs, and typically produces an error when it is applied to values of the wrong type.

We can draw the value of *doublepair* by nesting Pairs within cells:

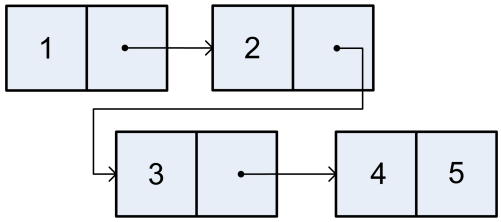


Drawing Pairs within Pairs within Pairs can get quite difficult, however. For example, try drawing (*cons* 1 (*cons* 2 (*cons* 3 (*cons* 4 5)))) this way.

Instead, we us arrows to point to the contents of cells that are not simple values. This is the structure of *doublepair* shown using arrows:



Using arrows to point to cell contents allows us to draw arbitrarily complicated data structures, keeping the cells reasonable sizes. For example, here is one way to draw (*cons* 1 (*cons* 2 (*cons* 3 (*cons* 4 5)))):



Exercise 5.3. Suppose the following definition has been executed:

```
(define tpair
  (cons (cons (cons 1 2) (cons 3 4))
        5))
```

Draw the structure defined by *tpair*, and give the value of each of the following expressions.

a. (*cdr tpair*)

b. (*car* (*car* (*car tpair*)))

c. (*cdr* (*cdr* (*car tpair*)))

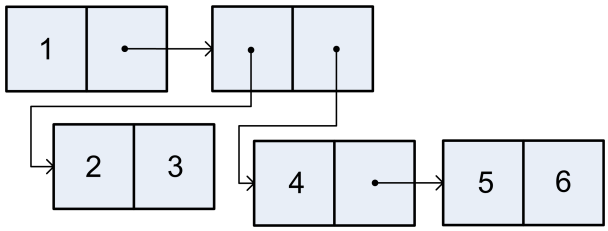
d. (*car* (*cdr* (*cdr tpair*)))

Exercise 5.4. Suppose the following definition has been executed:

(**define** *fstruct* (*cons* 1 (*cons* 2 (*cons* 3 4))))

Write expressions that extract each of the four elements from *fstruct*.

Exercise 5.5. What expression procedures the structure shown below:



5.2.1 Making Pairs

Although Scheme provides the built-in procedures *cons*, *car*, and *cdr* for creating Pairs and accessing their cells, there is nothing magic about these procedures. We could define procedures with the same behavior ourselves using only the subset of Scheme introduced in Chapter 3. For example, here is one way to define the pair procedures (we prepend an *s* to the names to avoid confusion with the built-in procedures):

(**define** (*scons a b*) (**lambda** (*w*) (**if** *w a b*)))
(**define** (*scar pair*) (*pair true*))
(**define** (*scdr pair*) (*pair false*))

The *scons* procedure takes the two parts of the Pair as inputs, and produces as output a procedure. The output procedure takes one input, a selector that determines which of the two cells of the Pair to output. If the selector is true, the value of the if-expression is the value of the first cell; if false, it is the value of the second cell. The *scar* and *scdr* procedures apply a procedure constructed by *scons* to either *true* (to select the first cell in *scar*) or *false* (to select the second cell in *scdr*).

Exercise 5.6. Convince yourself the definitions of *scons*, *scar*, and *scdr* above work as expected by following the evaluation rules to evaluate

(*scar* (*scons* 1 2))

Exercise 5.7. Suppose we define *tcons* as:

(**define** (*tcons a b*) (**lambda** (*w*) (**if** *w b a*)))

Show the corresponding definitions of *tcar* and *tcdr* needed to provide the correct pair selection behavior for a pair created using *tcons*.

5.2.2 Triples to Octuples

Pairs are useful for representing data that is composed of two parts such as a calendar date (composed of a number and month), or a playing card (composed of a rank and suit). But, what if we want to represent data composed of more than two parts such as a date (composed of a number, month, and year) or a poker hand consisting of five playing cards? For more complex data structures, we want to construct data structures that have more than two components.

A *triple* has three components. Here is one way to define a triple datatype and its accessors:

```
(define (make-triple a b c)
  (lambda (w) (if (= w 0) a (if (= w 1) b c))))


(define (triple-first t)  (t 0))
(define (triple-second t) (t 1))
(define (triple-third t)  (t 2))
```

Since a triple has three components we need three different selector values; we use 0, 1, and 2.

A simpler way to make a triple would be to combine two Pairs. We do this by making a Pair whose second cell is itself a Pair:

```
(define (make-triple a b c) (cons a (cons b c)))
```

Then, we can use the *car* and *cdr* procedures to define procedures for selecting elements of the triple:

```
(define (triple-first t)  (car t))
(define (triple-second t) (car (cdr t)))
(define (triple-third t)  (cdr (cdr t)))
```

Similarly, we can define a *quadruple* as a Pair whose second cell is a triple:

```
(define (make-quad a b c d) (cons a (make-triple b c d)))
(define (quad-first q)  (car q))
(define (quad-second q) (triple-first (cdr q)))
(define (quad-third q)  (triple-second (cdr q)))
(define (quad-fourth q) (triple-third (cdr q)))
```

We could continue in this manner defining increasingly large tuples:

- A *triple* is a Pair whose second cell is a Pair.
- A *quadruple* is a Pair whose second cell is a *triple*.
- A *quintuple* is a Pair whose second cell is a *quadruple*.
- A *sextuple* is a Pair whose second cell is a *quintuple*.
- A *septuple* is a Pair whose second cell is a *sextuple*.
- ...
- A $(+\ n\ 1)$-*uple* is a Pair whose second cell is an *n-uple*.

Hence, building from the simple Pair, we can construct tuples containing any number of components.

Exercise 5.8. Define a procedure that constructs a quintuple and procedures for selecting the five elements of a quintuple.

Exercise 5.9. Another way of thinking of a triple is as a Pair where the first cell is a Pair and the second cell is a scalar. Provide definitions of *make-triple*, *triple-first*, *triple-second*, and *triple-third* for this alternate triple definition.

5.3 Lists

In the previous section, we saw how to construct arbitrarily large tuples building from Pairs. This way of managing data is not very satisfying, however. It requires defining different procedures for constructing and accessing elements of every length tuple. For many applications, we want to be able to manage data of any length such as all the items in a web store, or all the bids on a given item. Since the number of components in these objects can change, it would be very painful to need to define a new tuple type every time an item is added. Hence, we need a data type that can hold any number of items, and a way to define procedures that work on any length tuple.

What we want is a way to construct and manipulate tuples that works for tuples containing *any* number of elements. This definition of an *any-uple* almost provides what we need:

> An *any-uple* is a Pair whose second cell is an *any-uple*.

This seems to allow an *any-uple* to contain any number of elements. The problem is we have no stopping point. With only the definition above, there is no way to construct an *any-uple* without already having one.

The situation is similar to defining *MoreDigits* as zero or more digits in Chapter 2, defining *MoreExpressions* in the Scheme grammar in Chapter 3 as zero or more *Expressions*, and recursive composition in Chapter 4.

Recall the grammar rules for *MoreExpressions*:

$$\begin{aligned} \textit{MoreExpressions} &::\Rightarrow \textit{Expression MoreExpressions} \\ \textit{MoreExpressions} &::\Rightarrow \epsilon \end{aligned}$$

The rule for constructing an *any-uple* above corresponds to the first *MoreExpression* replacement rule. To allow an *any-uple* to be constructed, we also need a construction rule similar to the second rule, where *MoreExpression* can be replaced with nothing. Since it is hard to type and read nothing in a program, Scheme has a name for this value: *null*.

null

DrScheme will print out the value of *null* as (). It is also known as the *empty list*, since it represents the List containing no elements. The built-in procedure *null?* takes one input parameter and evaluates to true if and only if the value of that parameter is null.

List

Using null, we can now define a *List*:

> A *List* is either (1) null or (2) a Pair whose second cell is a *List*.

Symbolically, we can define a List as:

$$\begin{aligned} \textit{List} &::\Rightarrow \mathsf{null} \\ \textit{List} &::\Rightarrow (\textit{cons Value List}) \end{aligned}$$

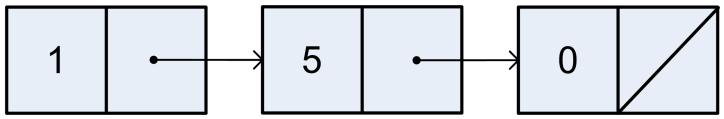
These two rules define a List as a data structure that can contain any number of elements. Starting from null, we can create Lists of any length. For example:

- *null* evaluates to a List containing no elements.

- (*cons* 1 *null*) evaluates to containing one element.
- (*cons* 1 (*cons* 5 *null*)) evaluates to a List containing two elements.
- (*cons* 1 (*cons* 5 (*cons* 0 *null*))) evaluates to a List containing three elements.

Scheme provides a convenient procedure, *list*, for constructing a List. The *list* procedure takes zero or more inputs, and evaluates to a List containing those inputs in order. The following expressions are equivalent to the corresponding expressions above: (*list*), (*list* 1), (*list* 1 5), and (*list* 1 5 0).

Lists are just a collection of Pairs, so we can draw a List using the same box and arrow notation we used to draw structures created with Pairs. Here is the structure resulting from (*list* 1 5 0):



There are three Pairs in the List, the second cell of each Pair is a List. For the third Pair, the second cell is the List *null*, which we draw as a slash through the final cell in the diagram.

Table 5.1 summarizes some of the built-in procedures for manipulating Pairs and Lists.

Procedure	Type	Output
cons	Value × Value → Pair	a Pair consisting of the two inputs
car	Pair → Value	the first cell of the input Pair
cdr	Pair → Value	the second cell of the input Pair
list	zero or more Values → List	a List containing the inputs
null?	Value → Boolean	true if the input is null, otherwise false
pair?	Value → Boolean	true if the input is a Pair, otherwise false
list?	Value → Boolean	true if the input is a List, otherwise false

Table 5.1. Selected Built-In Scheme Procedures for Lists and Pairs.

Exercise 5.10. For each of the following expressions, explain whether or not the expression evaluates to a List. You can check your answers with a Scheme interpreter by using the *list?* procedure that takes one input and evaluates to true if the value of that input is a List, and false otherwise.

a. *null*

b. (*cons* 1 2)

c. (*cons* 1 *null*)

d. (*cons null null*)

e. (*cons* (*cons* (*cons* 1 2) 3) *null*)

f. (*cons* 1 (*cons* 2 (*cons* 3 (*cons* 4 *null*))))

g. (*cdr* (*cons* 1 (*cons* 2 (*cons null null*))))

h. (*cons* (*list* 1 2 3) 4)

5.4 List Procedures

Since the List data structure is defined recursively, it is natural to define recursive procedures to examine and manipulate lists. Whereas most recursive procedures on inputs that are Numbers usually used zero as the base case, for lists the most common base case is the empty list. With numbers, we make progress by subtracting one to produce the input for the recursive application; with lists, we make progress by using *cdr* to reduce the length of the input List by one element for each recursive application. This means we often break problems involving Lists into figuring out what to do with the first element of the List and the result of applying the recursive procedure to the rest of the List.

Next we consider procedures that examine lists by walking through their elements and producing a scalar value. Section 5.4.2 generalizes these procedures. In Section 5.4.3, we explore procedures that produce lists as their output.

5.4.1 Procedures that Examine Lists

Here we explore procedures that take a single List as input and produce a scalar value that depends on the elements of the List as output. All of the examples in this section involve base cases where the List is empty, and recursive cases that apply the recursive procedure to the *cdr* of the input

List.

Example 5.1: Length. How many elements are in a given List?[3]

Our standard recursive problem solving technique is to "Think of the simplest version of the problem, something you can already solve." For this procedure, the simplest version of the problem is when the input is the empty list (*null*). We already know the length of the empty list is 0. So, the base case test is (*null? p*) and the output for the base case is 0.

For the recursive case, we need to consider the structure of all lists other than *null*. Recall our definition of a List: either *null* or (*cons Value List*). The base case handles the *null* list; the recursive case must handle a List that is a Pair of an element and a List. The length of this List is one more than the length of the List that is the *cdr* of the Pair.

```
(define (list-length p)
  (if (null? p)
      0
      (+ 1 (list-length (cdr p)))))
```

Here are a few example applications of our *list-length* procedure:

```
> (list-length null)
0
> (list-length (cons 0 null))
1
> (list-length (list 1 2 3 4))
4
> (list-length (cons 1 2))
cdr: expects argument of type <pair>; given 2
```

The last evaluation produces and error, since we are applying *list-length* to an input that is not a List.

Example 5.2: List Sums and Products. First, we define a procedure that takes a List of numbers as input and produces as output the sum of the numbers in the input List. As usual, the base case is when the input is *null*: the sum of an empty list is 0. For the recursive case, we need to add the value of the first number in the List, to the sum of the rest of the numbers in the List.

[3] Scheme provides a built-in procedure *length* that takes a List as its input and outputs the number of elements in the List. Here, we will define our own *list-length* procedure that does this (without using the built-in *length* procedure). As with many other examples and exercises in this chapter, it is instructive to define our own versions of some of the built-in list procedures.

```
(define (list-sum p)
    (if (null? p)
        0
        (+ (car p) (list-sum (cdr p)))))
```

We can define *list-product* similarly, except here we multiply the numbers in the List. The base case result cannot be 0, though, since then the final result would always be 0 since any number multiplied by zero is zero. We follow the mathematical convention that the product of the empty list is 1.

```
(define (list-product p)
    (if (null? p)
        1
        (* (car p) (list-product (cdr p)))))
```

Exercise 5.11. [⋆] Define a procedure *is-list?* that takes one input and produces true if the input is a List, and false otherwise. (Scheme provides a built-in *list?* procedure with this behavior, but you should not use it in your definition.) Hint: recall the definition of a List in Section 5.3.

Exercise 5.12. [⋆] Define a procedure *list-max* that takes a List of non-negative numbers as its input and produces as its result the value of the greatest element in the List (or 0 if there are no elements in the input List). For example, (*list-max* (*list* 1 5 0)) should evaluate to 5.

5.4.2 Generic Accumulators

The *list-length*, *list-sum*, and *list-product* procedures all have very similar structures. They all have a structure where the base case is when the input is the empty list, and the recursive case involves doing something with the first element of the List and recursively calling the procedure with the rest of the List:

```
(define (Recursive-Procedure p)
    (if (null? p)
        Base-Case-Result
        (Accumulator-Function (car p) (Recursive-Procedure (cdr p)))))
```

We can define a generic accumulator procedure for lists by making the base case result and accumulator function inputs:

```
(define (list-accumulate f base p)
  (if (null? p)
      base
      (f (car p) (list-accumulate f base (cdr p)))))
```

Then, we can define the *list-sum* and *list-product* procedures using *list-accumulate*:

```
(define (list-sum p) (list-accumulate + 0 p))
(define (list-product p) (list-accumulate * 1 p))
```

Defining the *list-length* procedure is a bit more complicated.

In our previous definition, the recursive case in the *list-length* procedure is (+ 1 (*list-length* (*cdr p*))). Unlike the *list-sum* example, the recursive case for *list-length* does not use the value of the first element of the List. The way *list-accumulate* is defined, we need a procedure that takes two inputs—the first input is the first element of the List; the second input is the result of applying *list-accumulate* to the rest of the List.

We should follow our usual strategy: be optimistic! Being optimistic as in recursive definitions, the value of the second input should be the length of the rest of the List. Hence, we need to pass in a procedure that takes two inputs, ignores the first input, and outputs one more than the value of the second input. We can make this procedure using (**lambda** (*a b*) (+ 1 *b*)). In the definition, we give the parameters more meaningful names:

```
(define (list-length p)
  (list-accumulate (lambda (el length-rest) (+ 1 length-rest)) 0 p))
```

Exercise 5.13. Define the *list-max* procedure (from Exercise 12) using *list-accumulate*.

Exercise 5.14. [⋆] Define the *list?* procedure (from Exercise 11) using *list-accumulate*.

Example 5.3: Accessing List Elements. The built-in *car* procedure provides a way to get the first element of a list, but what if we want to get the third element? We can do this by taking the *cdr* twice to eliminate the first two elements, and then using *car* to get the third: (*car* (*cdr* (*cdr p*))) evaluates to the third element of the List *p*.

We want a more general procedure that can access any selected list element. Hence, the procedure needs two inputs: a List, and an index Number that identifies the element. If we start counting from 1, then the base case is when the index is 1 and the output should be the first element of the List:

(**if** (= *n* 1) (*car p*) ...).

For the recursive case, we make progress by eliminating the first element of the list. But, we also need to adjust the index. Since we have removed the first element of the list, the index should be reduced by one. For example, instead of wanting the third element of the original list, we now want the second element of the *cdr* of the original list.

```
(define (list-get-element p n)
  (if (= n 1)
      (car p)
      (list-get-element (cdr p) (- n 1))))
```

What happens if we apply *list-get-element* with an index that is larger than the size of the input List (for example, (*list-get-element* (*list* 1 2) 3))?

The first recursive call will be (*list-get-element* (*list* 2) 2). The second recursive call will be (*list-get-element* (*list*) 1). At this point, *n* is 1, so the base case is reached and (*car p*) is evaluated. But, *p* is the empty list (which is not a Pair), so an error results.

A better version of *list-get-element* would provide a meaningful error message when the requested element is out of range. We can do this by adding an if-expression that tests if the input List is *null*:

```
(define (list-get-element p n)
  (if (null? p)
      (error "Index out of range")
      (if (= n 1)
          (car p)
          (list-get-element (cdr p) (- n 1)))))
```

The built-in procedure *error* takes a String as input. The String datatype is a sequence of characters; we can create a String by surrounding characters with double quotes, as in the example. The *error* procedure terminates program execution with a message that displays the input value.

defensive programming

Checking explicitly for invalid inputs is known as *defensive programming*. As your programs get more complex, programming defensively will help you avoid tricky to debug errors and will make it easier to understand and remember what your code is doing.

Exercise 5.15. [⋆] Define a procedure *list-last-element* that takes as input a List and outputs the last element of the input List. It is an error to apply *list-last-element* to the empty list.

Exercise 5.16. [⋆] The error message produced by our *list-get-element* is not very helpful. A more helpful message would print out the original input list and index. This is not possible to add with the provided implementation, since by the time the error is reached several elements of the list may have been eliminated and the index reduced through the recursive calls. Define a version of *list-get-element* that behaves identically for normal inputs, but produces more helpful error messages when the input index is out of range. A built-in procedure that will be useful for this is the *format* procedure. It is similar to the *printf* procedure described in Section 4.5.1, but instead of printing it returns a String. For example, (*format* "The list ~a is different from the number ~a." (*list* 1 5 0) 150) evaluates to the String "The list (1 5 0) is different from the number 150.".

Exercise 5.17. [⋆] Define a procedure *list-ordered?* that takes two inputs, a test procedure and a List. It evaluates to true if all the elements of the List are ordered according to the test procedure. For example, (*list-ordered?* $<$ (*list* 1 2 3)) should evaluate to true, and (*list-ordered?* $>$ (*list* 4 3 3 2)) should evaluate to false.

5.4.3 Procedures that Construct Lists

The procedures in this section take values (including lists) as input, and produce a new List as output. As before, the empty list is typically the base case. Since we are producing a List as output, the result for the base case is also usually null. The recursive case will use *cons* to construct a List combining the first element with the result of the recursive application on the rest of the List.

Example 5.4: Mapping. One common task for manipulating a List is to produce a new List that is the result of applying the same procedure to every element in the input List.

For the base case, applying a procedure to every element of the empty list produces the empty list. For the recursive case, we use *cons* to construct a List consisting of the procedure applied to the first element of the List and the result of applying the procedure to every element in the rest of the List.

For example, here is a procedure that constructs a List that contains the square of every element of the input List:

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

We can generalize this by making the procedure which is mapping each List element an input. The procedure *list-map* takes a procedure as its first input and a List as its second input. It outputs a List whose elements are the results of applying the input procedure to each element of the input List.[4]

```
(define (list-map f p)
  (if (null? p)
      null
      (cons (f (car p))
            (list-map f (cdr p)))))
```

Then, *square-all* could be defined as:

```
(define (square-all p) (list-map square p))
```

Exercise 5.18. Define a procedure *list-increment* that takes as input a List of numbers, and produces as output the input List with each value incremented by one. For example, (*list-increment* 1 5 0) should evaluate to the List (2 6 1).

Exercise 5.19. Use *list-map* and *list-sum* to define *list-length*:

```
(define (list-length p) (list-sum (list-map _____ p)))
```

Example 5.5: Filtering. Consider defining a procedure that takes as input a List of numbers, and evaluates to a List of all the non-negative numbers in the input. For example, (*list-filter-negative* (*list* 1 −3 −4 5 −2 0)) should evaluate to the List (1 5 0).

First, consider the base case when the input List is empty. If we filter the negative numbers from the empty list, the result is an empty list. So, for the base case, the result should be null.

[4]Scheme provides a built-in *map* procedure. It behaves like this one when passed a procedure and a single List as inputs, but can also work on more than one List input at a time.

In the recursive case, we need to determine if the first element should be included in the output List or not. If it should be included, we construct a new List consisting of the first element followed by the result of filtering the remaining elements in the List. If it should not be included, we do not include the first element, and the result is the result of filtering the remaining elements in the List.

```
(define (list-filter-negative p)
  (if (null? p)
      null
      (if (>= (car p) 0)
          (cons (car p) (list-filter-negative (cdr p)))
          (list-filter-negative (cdr p)))))
```

Similarly to *list-map*, we can generalize our filter by making the test procedure as an input, so we can use any predicate to determine which elements to include in the output List.[5]

```
(define (list-filter test p)
  (if (null? p)
      null
      (if (test (car p))
          (cons (car p) (list-filter test (cdr p)))
          (list-filter test (cdr p)))))
```

Using the *list-filter* procedure, we can define *list-filter-negative* as:

```
(define (list-filter-negative p) (list-filter (lambda (x) (>= x 0)) p))
```

We could also define the *list-filter* procedure using the *list-accumulate* procedure from Section 5.4.1:

```
(define (list-filter test p)
  (list-accumulate
    (lambda (el rest) (if (test el) (cons el rest) rest))
    null
    p))
```

Exercise 5.20. Define a procedure *list-filter-even* that takes as input a List of numbers and produces as output a List consisting of all the even elements of the input List.

[5] Scheme provides a built-in function *filter* that behaves like our *list-filter* procedure.

Exercise 5.21. [⋆] Define a procedure *list-remove* that takes two inputs: a test procedure and a List. As output, it produces a List that is a copy of the input List with all of the elements for which the test procedure evaluates to true removed. For example, (*list-remove* (**lambda** (*x*) (= *x* 0)) (*list* 0 1 2 3)) should evaluates to the List (1 2 3).

Exercise 5.22. [⋆⋆] Define a procedure *list-unique-elements* that takes as input a List and produces as output a List containing the unique elements of the input List. The output List should contain the elements in the same order as the input List, keeping the first appearance of each duplicate value.

Example 5.6: Append. The *list-append* procedure takes as input two lists and produces as output a List consisting of the elements of the first List followed by the elements of the second List.[6] For the base case, when the first List is empty, the result of appending the lists should just be the second List. When the first List is non-empty, we can produce the result by *cons*-ing the first element of the first List with the result of appending the rest of the first List and the second List.

```
(define (list-append p q)
  (if (null? p)
      q
      (cons (car p) (list-append (cdr p) q))))
```

Example 5.7: Reverse. The *list-reverse* procedure takes a List as input and produces as output a List containing the elements of the input List in reverse order.[7] For example, (*list-reverse* (*list* 1 2 3)) should evaluate to the List (3 2 1). As usual, we consider the base case where the input List is null first. The reverse of the empty list is the empty list. To reverse a non-empty List, we should put the first element of the List at the end of the result of reversing the rest of the List. The tricky part is putting the first element at the end, since *cons* only puts elements at the beginning of a List. We can use the *list-append* procedure defined in the previous example to put a List at the end of another List. To make this work, we need to turn the element at the front of the List into a List containing just that element. We do this using (*list* (*car p*)).

[6] There is a built-in procedure *append* that does this. The built-in *append* takes any number of Lists as inputs, and appends them all into one List.

[7] The built-in procedure *reverse* does this.

```
(define (list-reverse p)
  (if (null? p)
      null
      (list-append (list-reverse (cdr p)) (list (car p)))))
```

Exercise 5.23. [⋆] Define the *list-reverse* procedure using *list-accumulate*.

Example 5.8: Intsto. For our final example, we consider the problem of constructing a List containing the whole numbers between 1 and the input parameter value. For example, (*intsto* 5) should evaluate to the List (1 2 3 4 5).

Here, we combine some of the ideas from the previous chapter on creating recursive definitions for problems involving numbers, and from this chapter on lists. Since the input parameter is not a List, the base case is not the usual base case when the input is *null*. Instead, we use the input value 0 as the base case. The result for input 0 is the empty list. For higher values, the output is the result of putting the input value at the end of the List of numbers up to the input value minus one.

A first attempt that doesn't quite work is:

```
(define (revintsto n)
  (if (= n 0)
      null
      (cons n (revintsto (- n 1)))))
```

The problem with this solution is that it is *cons*-ing the higher number to the front of the result, instead of at the end. Hence, it produces the List of numbers in descending order: (*revintsto* 5) evaluates to the List (5 4 3 2 1).

One solution is to reverse the result by composing *list-reverse* with *revintsto*:

```
(define (intsto n) (list-reverse (revintsto n)))
```

Equivalently, we can use the *fcompose* procedure from Section 4.2:

```
(define intsto (fcompose list-reverse revintsto))
```

Alternatively, we could use *list-append* to put the high number directly at the end of the List. Since the second operand to *list-append* must be a List, we use (*list n*) to make a singleton List containing the value as we did for *list-reverse*.

```
(define (intsto n)
  (if (= n 0)
      null
      (list-append (intsto (- n 1)) (list n))))
```

Although all of these procedures are functionally equivalent (for all valid inputs, each function produces exactly the same output), the amount of computing work (and hence the time they take to execute) varies substantially across the implementations. In Chapter 7, we introduce techniques for understanding the cost of evaluating a procedure and we consider the costs of the different *intsto* implementations.

Exercise 5.24. Define the *factorial* procedure (from Example 1) using *intsto*.

5.5 Lists of Lists

The elements of a List can be any datatype, including, of course, other Lists. In defining procedures that operate on Lists of Lists, we often use more than one recursive call when we need to go inside the inner Lists.

Example 5.9: Summing Nested Lists. Consider the problem of summing all the numbers in a List of Lists. For example, (*nested-list-sum* (*list* (*list* 1 2 3) (*list* 4 5 6))) should evaluate to 21. We can define *nested-list-sum* using *list-sum* on each List.

```
(define (nested-list-sum p)
  (if (null? p)
      0
      (+ (list-sum (car p))
         (nested-list-sum (cdr p)))))
```

This works when we know the input is a List of Lists. But, what if the input can contain arbitrarily deeply nested Lists?

To handle this, we need to recursively sum the inner Lists. Each element in our deep List is either a List or a Number. If it is a List, we should add the value of the sum of all elements in the List to the result for the rest of the List. If it is a Number, we should just add the value of the Number to the result for the rest of the List. So, our procedure involves two recursive calls: one for the first element in the List when it is a List, and the other for the rest of the List.

```
(define (deep-list-sum p)
  (if (null? p)
      0
      (+ (if (list? (car p))
             (deep-list-sum (car p))
             (car p))
         (deep-list-sum (cdr p)))))
```

Example 5.10: Flattening Lists. Another way to compute the deep list sum would be to first flatten the List, and then use the *list-sum* procedure.

Flattening a nested list takes a List of Lists and evaluates to a List containing the elements of the inner Lists. We can define *list-flatten* by using *list-append* to append all the inner Lists together.

```
(define (list-flatten p)
  (if (null? p)
      null
      (list-append (car p) (list-flatten (cdr p)))))
```

This flattens a List of Lists into a single List. If we want to completely flatten a deeply nested List, we need to use multiple recursive calls as we did with *deep-list-sum*.

```
(define (deep-list-flatten p)
  (if (null? p)
      null
      (list-append (if (list? (car p))
                       (deep-list-flatten (car p))
                       (list (car p)))
                   (deep-list-flatten (cdr p)))))
```

Now we can define *deep-list-sum* as

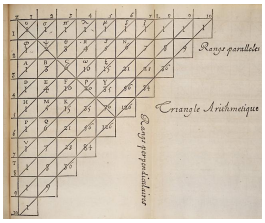
```
(define deep-list-sum (fcompose deep-list-flatten list-sum))
```

Exercise 5.25. Define a procedure *deep-list-map* that behaves similarly to *list-map* but on deeply nested lists. It should take two parameters, a mapping procedure, and a List (that may contain deeply nested Lists as elements), and output a List with the same structure as the input List with each value mapped using the mapping procedure.

Exercise 5.26. Define a procedure *deep-list-filter* that behaves similarly to *list-filter* but on deeply nested lists.

Exploration 5.1: Pascal's Triangle

This triangle is known as Pascal's Triangle (named for Blaise Pascal, although known to many others before him):



Pascal's Triangle

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
...
```

Each number in the triangle is the sum of the two numbers immediately above and to the left and right of it. The numbers in Pascal's Triangle are the coefficients in a binomial expansion. The numbers of the n^{th} row (where the rows are numbered starting from 0) are the coefficients of the binomial expansion of $(x+y)^n$. For example, $(x+y)^2 = x^2 + 2xy + y^2$, so the coefficients are `1 2 1`, matching the third row in the triangle; from the fifth row, $(x+y)^4 = x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4$. The values in the triangle also match the number of ways to choose k elements from a set of size n (see Exercise 5) — the k^{th} number on the n^{th} row of the triangle gives the number of ways to choose k elements from a set of size n. For example, the third number on the fifth (n = 4) row is 6, so there are 6 ways to choose 3 items from a set of size 4.

The goal of this exploration is to define a procedure, *pascals-triangle* to produce Pascal's Triangle. The input to your procedure should be the number of rows; the output should be a list, where each element of the list is a list of the numbers on that row of Pascal's Triangle. For example, (*pascals-triangle* `0`) should produce `()` (the empty list), (*pascals-triangle* `0`) should produce `((1))` (a list containing one element which is a list containing the number `1`), and (*pascals-triangle* `4`) should produce `((1) (1 1) (1 2 1) (1 3 3 1) (1 4 6 4 1))`.

Very ambitious readers will attempt to define *pascals-triangle* themselves; the sub-parts below provide some hints for one way to define it.

a. First, define a procedure *expand-row* that expands one row in the triangle. It takes a List of numbers as input, and as output produces a List with one more element than the input list. The first number in the output List should be the first number in the input List; the last number in the output List should be the last number in the input List. Every other number in the output List is the sum of two numbers in the input List. The n^{th} number in the output List is the sum of the $n-1^{th}$ and n^{th} num-

bers in the input List. For example, (*expand-row* (*list* 1)) should evaluate to the List (1 1); (*expand-row* (*list* 1 1)) should evaluate to the List (1 2 1); and (*expand-row* (*list* 1 4 6 4 1)) should evaluate to the List (1 5 10 10 5 1). This is trickier than the recursive list procedures we have seen so far since the base case is not the empty list (it is okay if your *expand-row* procedure produces an error when the input is an empty list). It also needs to deal with the first element specially. So, to define *expand-row*, it will be helpful to divide it into two procedures, one that deals with the first element of the list, and one that produces the rest of the list:

```
(define (expand-row p)
  (cons (car p) (expand-row-rest p)))
```

b. Next, define a procedure *pascals-triangle-row* that takes one input, n, and outputs the n^{th} row of Pascal's Triangle. For example, (*pascals-triangle-row* 0) should evaluate to the List (1) and (*pascals-triangle-row* 4) should produce (1 4 6 4 1).

c. Finally, define *pascals-triangle* with the behavior described above. (If you have a hard time getting the rows of the triangle to appear in the right order, look at the *intsto* example.)

5.6 Data Abstraction

The mechanisms we have for constructing and deconstructing complex data structures are valuable because they enable us to think about programs closer to the level of the problem we are solving than the low level of how data is stored and manipulated in the computer. Our goal is to hide unnecessary details about how data is represented so we can focus on the important aspects of what the data means and what we need to do with it to solve our problem. The technique of hiding how data is represented from how it is used is known as *data abstraction*. *data abstraction*

The datatypes we have seen so far are not very abstract. We have datatypes for representing Pairs, triples, and Lists, but we want datatypes for representing objects closer to the level of the problem we want to solve. A good data abstraction is abstract enough to be used without worrying about details like which cell of the Pair contains which data and how to access the different elements of a List. Instead, we want to define procedures with meaningful names that extract or manipulate the relevant parts of our data.

The rest of this section is an extended example that illustrates how we can solve problems by first identifying the objects we need to model to model

the problem, and then implementing data abstractions that represent those objects. Once the appropriate data abstractions are designed and implemented, the solution to the problem often follows readily. This example also uses many of the list procedures defined earlier in this chapter.

Example 5.11: Pegboard Puzzle. For this example, we develop a program to solve the infamous pegboard puzzle, often found tormenting unsuspecting diners at pancake restaurants. The standard puzzle is a one-player game played on a triangular-shaped board with fifteen holes, initially with a peg in each hole. The game begins by removing one of the pegs. Then, the goal is to remove all but one of the pegs by jumping pegs over one another. A peg may jump over an adjacent peg only when there is a free hole on the other side of the peg. The jumped peg is removed. The game ends when there are no possible moves. If there is only one peg remaining, the player wins (according to the Cracker Barrel version of the game, "Leave only one—you're genius"). If more than one peg remains, the player loses ("Leave four or more'n you're just plain 'eg-no-ra-moose'.").

Pegboard Puzzle

Our goal is to develop a program that provides a winning solution to the pegboard game, regardless of the starting position (which hole is initially empty). We will use a *brute force* approach: try all possible moves until we find one that works. Brute force solutions work well when problems are fairly small. Because they have to try all possibilities, however, they are often too slow for solving large problems (even on the most powerful computers).[8]

brute force

The first thing to think about to solve a complex problem is what datatypes we need. We want datatypes that represent the things we need to model in our problem solution. For the pegboard game, one thing we need to model is the board. The important thing about a datatype is what you can do with it. So, to design our board datatype we need to think about what we want to do with a board. In the physical pegboard game, the board holds the pegs. The important property we need to observe about the board is which holes on the board contain pegs. For this, we need a way of identifying board positions.

[8] As we will see in Chapter 16, the generalized pegboard puzzle is an example of a class of problems known as *NP-Complete*. This means it is not known whether or not any solution exists that is substantially better than the brute force solution, but it would be extraordinarily surprising (and have momentous impact) to find one.

We can identify the board positions using row and column numbers:

(1 1)
(2 1) (2 2)
(3 1) (3 2) (3 3)
(4 1) (4 2) (4 3) (4 4)
(5 1) (5 2) (5 3) (5 4) (5 5)

So, we will make a Position datatype to represent a position on the board. A position has a row and a column, so we could just use a Pair to represent a position. This would work, but we prefer to have a more abstract datatype so we can think about a position's row and column, rather than thinking that a position is a Pair and using the *car* and *cdr* procedures to extract the row and column from the position.

Our Position datatype should provide at least these operations:

- *make-position*: Number × Number → Position — create a Position representing the row and column given by the inputs
- *position-get-row*: Position → Number — gets the row number of the input Position
- *position-get-column*: Position → Number — gets the column number of the input Position

Since the Position needs to keep track of two numbers, a natural way to implement the Position datatype is to use a Pair:

```
(define (make-position row col) (cons row col))
(define (position-get-row posn) (car posn))
(define (position-get-column posn) (cdr posn))
```

A more defensive way to implement the Position datatype is to use a *tagged list*. We can use a symbol to identify the type of data, and the remaining elements in the list as the data. Symbols are a quote (') followed by a sequence of characters. The important operation we can do with a Symbol, is test whether it is an exact match for another symbol using the *eq?* procedure. So, we first define the Tagged-List datatype (we use the *list-get-element* procedure from Example 3, and the built-in *format* procedure introduced in Exercise 16):

tagged list

```
(define (make-taggedlist tag p) (cons tag p))
(define (taggedlist-get-tag p) (car p))
```

```
(define (taggedlist-get-element tag p n)
  (if (eq? (taggedlist-get-tag p) tag)
      (list-get-element (cdr p) n)
      (error (format "Bad list tag: ~a (expected ~a)"
                     (taggedlist-get-tag p) tag))))
```

This is an example of defensive programming. Using our tagged lists, if we accidentally attempt to use a value that is not a Position as a position, we will get an easy to understand error message instead of a hard-to-debug error (or worse, an unnoticed incorrect result).

Using the tagged list, we define the Position datatype as:

```
(define (make-position row col) (make-taggedlist 'Position (list row col)))
```

These procedures provide ways to extract the row and column from a Position, and to determine if two Positions are the same place:

```
(define (position-get-row posn)
  (taggedlist-get-element 'Position posn 1))

(define (position-get-column posn)
  (taggedlist-get-element 'Position posn 2))
```

Here are some example interactions with our Position datatype:

```
> (define pos (make-position 2 1))
> pos
(Position 2 1)
> (get-position-row pos)
2
> (get-position-row (list 1 2))
Bad list tag: 1 (expected Position)
```

The last evaluation produces an error, since we are attempting to apply a procedure that expects a Position as its input to a value that is not a Position.

Now we have a way of identifying positions, we can make progress on the board datatype. It should provide at least these two procedures:

- *make-board*: Number → Board — create a board full of pegs with the input number of rows. (In the physical board, the board is always the same size with 5 rows, but we define our Board datatype to support any number of rows.)

- *board-contains-peg?*: Position → Boolean — evaluates to *true* if and only if the hold at the input Position contains a peg.

We also need to be able to change the state of the board to reflect moves in the game. This means the board data type should have operations for removing and adding pegs to the board. Nothing we have seen so far, however, provides a means for changing the state of an existing object.[9] So, instead of thinking of these operations as changing the state of the board, what they really do is create a new board that is different from the input board by the one new peg. Hence, these procedures will take a Board and Position as inputs, and produce as output a Board.

- *board-add-peg*: Board × Position → Board — the output is a Board containing all the pegs of the input Board and one additional peg at the input Position. If the input Board already has a peg at the input Position, produces an error.

- *board-remove-peg*: Board × Position → Board — the output is a Board containing all the pegs of the input Board except for the peg at the input Position. If the input Board does not have a peg at the input Position, produces an error.

There are lots of different ways we could represent the Board. One possibility is to keep a List of the Positions of the pegs on the board. Another possibility is to keep a List of the Positions of the empty holes on the board. Yet another possibility is to keep a List of Lists, where each List corresponds to one row on the board. The elements in each of the Lists are Booleans representing whether or not there is a peg at that position. The good thing about data abstraction is we could pick any of these representations and change it to a different representation later (for example, if we needed a more efficient board implementation). As long as the procedures for implementing the Board are updated the work with the new representation, all of the other code should continue to work correctly without any changes.

We choose the third option, to represent a Board using a List of Lists, where each element of the inner lists is a Boolean indicating whether or not the corresponding position contains a peg. So, our *make-board* procedure should evaluate to a List of Lists, where each element of the List contains the row number of elements and all the inner elements are true (the initial board is completely full of pegs). First, we define a procedure *make-list-of-constants*

[9] We will introduce mechanisms for changing state in Chapter 10. Allowing state to change leads to lots of complexities including breaking our substitution model of evaluation.

that takes two inputs, a Number, *n*, and a Value, *val*. The output is a List of length *n* where each element has the value *val*.

```
(define (make-list-of-constants n val)
  (if (= n 0)
      null
      (cons val (make-list-of-constants (- n 1) val))))
```

To make the initial board, we can use *make-list-of-constants* to make each row of the board. As usual, a recursive problem solving strategy works well: the simplest board is a board with zero rows (which should be represented as the empty list); for each larger board, we need to add a row with the right number of elements.

The tricky part is getting the rows to be in order. This is similar to the problem we faced with *intsto*, and a similar solution using *append-list* works here.

```
(define (make-board rows)
  (if (= rows 0)
      null
      (list-append (make-board (- rows 1))
                   (list (make-list-of-constants rows true)))))
```

For example, (*make-board* 5) evaluates to the List of Lists:

```
((true)
 (true true)
 (true true true)
 (true true true true)
 (true true true true true))
```

The *board-rows* procedure takes a Board as input and outputs the number of rows on the board.

```
(define (board-rows board) (length board))
```

Now that we have defined our board datatype, we can implement the procedures for observing and manipulating the board. The *board-contains-peg?* procedure takes a Board and a Position as input, and should output a Boolean indicating whether or not that Position contains a peg. To implement *board-contains-peg?* we need to find the appropriate row in our board representation, and then find the element in its list corresponding to the Position's column. The *list-get-element* procedure (from Example 3) does exactly what we need. Since our board is represented as a List of Lists,

we need to use it twice: first to get the row, and then to select the column within that row:

```
(define (board-contains-peg? board pos)
  (list-get-element (list-get-element board (position-get-row pos))
                    (position-get-column pos)))
```

Defining the procedures for adding and removing pegs from the board is a bit more complicated. We need to make a board with every row identical to the input board, except the row where the peg is added or removed. For that row, we need to replace the corresponding value. First we implement the *board-add-peg* procedure. After implementing this, we will see that removing is very similar, so both adding and removing can be done with a more general procedure that takes an extra input.

To implement *board-add-peg*, we first consider the subproblem of adding a peg to a row. The procedure *row-add-peg* takes as input a List representing a row on the board and a Number indicating the column where the peg should be added. We can define *row-add-peg* recursively: the base case is when the new peg belongs at the beginning of the row (the column number is 1); in the recursive case, we copy the first element in the List, and add the peg to the rest of the list.

```
(define (row-add-peg pegs col)
  (if (= col 1)
      (cons true (cdr pegs))
      (cons (car pegs) (row-add-peg (cdr pegs) (- col 1)))))
```

To add the peg to the board, we can use *row-add-peg* to add the peg to the appropriate row, and keep all the other rows the same. We divide the problem into a procedure that checks the add is valid (the selected position does not already contain a peg), and a helper procedure that produces the new board.

```
(define (board-add-peg-helper board row col)
  (if (= row 1) ; this row
      (cons (row-add-peg (car board) col)
            (cdr board))
      (cons (car board)
            (board-add-peg-helper (cdr board) (- row 1) col))))
```

```
(define (board-add-peg board pos)
  (if (board-contains-peg? board pos)
      (error (format "Board already contains peg at position: ~a" pos))
      (board-add-peg-helper board (position-get-row pos)
                                  (position-get-column pos))))
```

To define *board-replace-peg* we would need to do essentially the same thing, but instead of making the value at the selected position *true* to represent a peg, we need to make it *false* to indicate an empty hole. One might be tempted to "cut-and-paste" to define *board-remove-peg* using the adding procedures already defined, but this makes the code nearly twice as long as it needs to be. It is much better to instead write a more general procedure that can be used to define both *board-add-peg* and *board-remove-peg*. We do this by defining the *board-replace-peg* procedure that takes an extra parameter (*true* for adding, *false* for replacing).

```
(define (row-replace-peg pegs col val)
  (if (= col 1)
      (cons val (cdr pegs))
      (cons (car pegs) (row-replace-peg (cdr pegs) (- col 1) val))))

(define (board-replace-peg board row col val)
  (if (= row 1)
      (cons (row-replace-peg (car board) col val)
            (cdr board))
      (cons (car board)
            (board-replace-peg (cdr board) (- row 1) col val))))
```

Then, both *board-add-peg* and *board-remove-peg* can be defined simply using *board-remove-peg*.

```
(define (board-add-peg board pos)
  (if (board-contains-peg? board pos)
      (error (format "Board already contains peg at position: ~a" pos))
      (board-replace-peg board (position-get-row pos)
                               (position-get-column pos) true)))

(define (board-remove-peg board pos)
  (if (not (board-contains-peg? board pos))
      (error (format "Board does not contain peg at position: ~a" pos))
      (board-replace-peg board (position-get-row pos)
                               (position-get-column pos) false)))
```

Now we have datatypes for representing a Position and the Board.

The next datatype we want is a way to represent a move. A move involves three positions: the position where the jumping peg starts, the position of the peg that is jumped and removed, and the landing position. One possibility would be to represent a move as a list of the three positions. A better option is to observe that once any two of the positions are known, the third position is determined. For example, if we know the starting position and the landing position, we know the jumped peg is at the position between them. So, we could represent a jump using a List containing the starting and landing positions and find the jumped position as needed from those.

Another possibility is to represent a jump by storing the starting Position and the direction. This is also enough to determine the jumpee and landing positions. This approach seems most natural, and avoids the difficulty of calculating jumpee positions. To do it, we first design a Direction datatype for representing the possible move directions. Directions have two components: the change in the row (we use 1 for down and -1 for up), and the change in the column (1 for right and -1 for left). We can move directly up, down, left, and right. Because of the triangular shape of the board, only two of the diagonal directions make sense: up-left and down-right (up-right does not make sense since moving up normally is actually moving up and right on the triangular board).

We implement the Direction datatype using a similar tagged list datatype to how we defined Position.

```
(define (make-direction down right)
  (make-taggedlist 'Direction (list down right)))


(define (direction-get-vertical dir)
  (taggedlist-get-element 'Direction dir 1))


(define (direction-get-horizontal dir)
  (taggedlist-get-element 'Direction dir 2))
```

We also define names representing the six possible move directions, and a list of all move directions. This will be useful when we consider all possible moves.

```
(define direction-up (make-direction -1 0))
(define direction-down (make-direction 1 0))
(define direction-left (make-direction 0 -1))
(define direction-right (make-direction 0 1))
(define direction-up-left (make-direction -1 -1))
(define direction-down-right (make-direction 1 1))
```

```
(define all-directions
        (list direction-up direction-down direction-left direction-right
        direction-up-left direction-down-right))
```

Now, we can define the Move datatype using the starting position and the jump direction.

```
(define (make-move start direction)
  (make-taggedlist 'Move (list start direction)))


(define (move-get-start move)
  (taggedlist-get-element 'Move move 1))


(define (move-get-direction move)
  (taggedlist-get-element 'Move move 2))
```

We also need to define procedures for getting the jumpee and landing positions of a move. The jumpee position is the result of moving one step in the move direction from the starting position. So, it will be useful to define a procedure that takes a Position and a Direction as input, and outputs a Position that is one step in the input Direction from the input Position.

```
(define (direction-step pos dir)
  (make-position
    (+ (position-get-row pos) (direction-get-vertical dir))
    (+ (position-get-column pos) (direction-get-horizontal dir))))
```

Using *direction-step* we can implement procedure to get the middle and landing positions.

```
(define (move-get-jumpee move)
  (direction-step (move-get-start move) (move-get-direction move)))


(define (move-get-landing move)
  (direction-step (move-get-jumpee move) (move-get-direction move)))
```

With our Board, Move, and Position datatypes, we can now implement a procedure that models making a move on a board. To make a move we remove the jumped peg, and move the peg at the move starting position to the landing position. We can consider moving the peg as removing the peg from the starting position and adding a peg to the landing position.

```
(define (execute-move board move)
  (board-add-peg
   (board-remove-peg (board-remove-peg board (move-get-start move))
                     (move-get-jumpee move))
   (move-get-landing move)))
```

Now that we have datatypes for modeling positions, moves, and the board, and a way to make moves on the board, we are ready to develop the solution. At each step, we need to try all valid moves on the board to see if any move leads to a winning position (that is, a position with only one peg remaining). So, we need a procedure that takes a Board as its input and outputs a List of all valid moves on the board. We break this down into the problem of producing a list of all conceivable moves (all moves in all directions from all starting positions on the board), filtering that list for moves that stay on the board, and then filtering the resulting list for moves that are legal (start at a position containing a peg, jump over a position containing a peg, and land in a position that is an empty hole).

First, we generate all conceivable moves by creating moves starting from each position on the board and moving in all possible move directions. We break this down further: the first problem is to produce a List of all positions on the board. We can generate a list of all row numbers using the *intsto* procedure (from Example 8): (*intsto* (*board-rows board*)). To get a list of all the positions, we need to produce a list of the positions for each row. We can do this by mapping each row to the corresponding list:

```
(list-map
  (lambda (row)
    (list-map
      (lambda (col) (make-position row col))
      (intsto row)))
  (intsto (board-rows board)))
```

This almost does what we need, except instead of producing one List containing all the positions, it produces a List of Lists for the positions in each row. The *list-flatten* procedure (from Example 10) produces a flat list containing all the positions.

```
(define (all-positions board)
  (list-flatten
   (list-map
     (lambda (row)
       (list-map
         (lambda (col) (make-position row col))
         (intsto row)))
     (intsto (board-rows board)))))
```

Now, for each Position, we need to consider all possible moves. Again, *list-map* is useful. For each Position we create a list of moves where each move is that Position as the starting position and the Direction is one of the possible move directions. This produces a List of Lists, so we use *list-flatten* to flatten the output of the *list-map* application into a single List of Moves.

```
(define (all-conceivable-moves board)
  (list-flatten
    (list-map
      (lambda (pos)
        (list-map
          (lambda (dir) (make-move pos dir))
          all-directions))
      (all-positions board))))
```

The List of Moves produced by *all-conceivable-moves* includes moves that fly off the board. We use the *list-filter* procedure to remove those moves, to get the list of possible moves. For the filter procedure, we need a procedure that determines if a Position is on the board. A position is valid if its row number is between 1 and the number of rows on the board, and its column numbers is between 1 and the row number.

```
(define (valid-position? board pos)
  (and
   (>= (position-get-row pos) 1)
   (>= (position-get-column pos) 1)
   (<= (position-get-row pos) (board-rows board))
   (<= (position-get-column pos) (position-get-row pos))))
```

Hence, we can define *all-possible-moves* as:

```
(define (all-possible-moves board)
  (list-filter
    (lambda (move)
      (valid-position? board (move-get-landing move)))
    (all-conceivable-moves board)))
```

Finally, we need to filter out the moves that are not legal moves. A legal move must start at a position that contains a peg, jump over a position that contains a peg, and land in an empty hole. We can use *list-filter* to keep only the legal moves similarly to how we kept only the moves that stay on the board.

```
(define (all-legal-moves board)
  (list-filter
    (lambda (move)
      (and (board-contains-peg? board (move-get-start move))
           (board-contains-peg? board (move-get-jumpee move))
           (not (board-contains-peg? board
                                     (move-get-landing move)))))
   (all-possible-moves board)))
```

To find a solution, we will need a way to know when we have found a winning board. This is a board with only one peg. We implement a more general procedure to count the number of pegs on a board first. Our board representation used *true* to represent a peg. To count the pegs, we first map the Boolean values used to represent pegs to 1 if there is a peg and 0 if there is no peg. Then, we can use *sum-list* to count the number of pegs. Since the Board is a List of Lists, we first need to use *list-flatten* to put all the pegs in a single List.

```
(define (board-number-of-pegs board)
  (list-sum
   (list-map
    (lambda (peg) (if peg 1 0))
    (list-flatten board))))
```

Then, we can determine if a board represents a winning position by checking if it contains one peg.

```
(define (is-winning-board? board)
  (= (board-number-of-pegs board) 1))
```

Note that our *board-number-of-pegs* procedure depends on the way we choose to represent the Board data type (this is why we give it a name that begins with *board-*). Unlike the other procedures we have written for solving the game, this procedure would need to be reconsidered if we change the way the Board type is represented.

Our goal is to find a sequence of moves that leads to a winning position, starting from the current board. If there is a winning sequence of moves, we can find it by trying all possible moves on the current board. Each of

these moves leads to a new board. If the original board has a winning sequence of moves, at least one of the new boards has a winning sequence of moves. Hence, we can solve the puzzle by recursively trying all moves until finding a winning position.

```
(define (solve-pegboard board)
  (if (is-winning-board? board)
      null ; no moves needed to reach winning position
      (try-moves board (all-legal-moves board))))
```

The *solve-pegboard* procedure evaluates to a List of Moves, if there is a sequence of moves that wins the game starting from the input Board. This could be null, in the case where the input board is already a winning board. If there is no sequence of moves to win from the input board, it outputs false.

It remains to define the *try-moves* procedure. It takes a Board and a List of Moves as inputs, and outputs either false if there is no sequence of moves that wins from the input Board starting with one of the Moves in the input List, or a List of Moves that wins.

The base case is when there are no moves to try. When the input list is *null* it means there are no moves to try. We will use the output *false* to mean this attempt did not lead to a winning board. Otherwise, we should try the first move. If it leads to a winning position, we should evaluate to the List of Moves that starts with the first move, and is followed by the rest of the moves needed to solve the board resulting from taking the first move (that is, the result of *solve-pegboard* applied to the Board resulting from taking the first move). If the first move doesn't lead to a winning board, we should try the rest of the moves (by calling *try-moves* recursively).

```
(define (try-moves board moves)
  (if (null? moves)
      false ; didn't find a winner
      (if (solve-pegboard (execute-move board (car moves)))
          (cons (car moves)
                (solve-pegboard (execute-move board (car moves))))
          (try-moves board (cdr moves)))))
```

Evaluating (*solve-pegboard* (*make-board* 5)) produces false since there is no way to win starting from a completely full board. Evaluating (*solve-pegboard* (*board-remove-peg* (*make-board* 5) (*make-position* 1 1))) takes about three minutes to produce a result:

((*Move* (*Position* 3 1) (*Direction* −1 0))
(*Move* (*Position* 3 3) (*Direction* 0 −1))
(*Move* (*Position* 1 1) (*Direction* 1 1))
(*Move* (*Position* 4 1) (*Direction* −1 0))
(*Move* (*Position* 4 4) (*Direction* −1 −1))
(*Move* (*Position* 5 2) (*Direction* −1 0))
(*Move* (*Position* 5 3) (*Direction* −1 0))
(*Move* (*Position* 2 1) (*Direction* 1 1))
(*Move* (*Position* 2 2) (*Direction* 1 1))
(*Move* (*Position* 5 5) (*Direction* −1 −1))
(*Move* (*Position* 3 3) (*Direction* 1 0))
(*Move* (*Position* 5 4) (*Direction* 0 −1))
(*Move* (*Position* 5 1) (*Direction* 0 1)))

This is a sequence of moves for winning the game starting from a 5-row board with the top peg removed.

Exercise 5.27. [⋆] Change the implementation to use a different Board representation, such as keeping a list of the Positions of each hole on the board. Only the procedures with names starting with *board-* should need to change when the Board representation is changed. Compare your implementation to the one described here. Which representation is better?

Exercise 5.28. [⋆⋆] The described implementation is very inefficient. It does lots of redundant computation. For example, *all-possible-moves* evaluates to the same value every time it is applied to a board with the same number of rows. It is wasteful to recompute this over and over again to solve a given board. See how much faster you can make the pegboard solver. Can you make it fast enough to solve the 5-row board in less than half the original time? Can you make it fast enough to solve a 6-row board?

Exercise 5.29. [⋆] The standard pegboard puzzle uses a triangular board, but there is no reason the board has to be a triangle. Define a more general pegboard puzzle solver that works for a board of any shape.

5.7 Summary

Building from the simple pair, we can create complex data structures including lists. A List is either *null*, or a Pair whose second cell is a List. Since the List data structure is itself defined recursively, it is not surprising that many procedures that involve lists can be defined recursively.

We can specialize our general problem solving strategy from Chapter 3 for procedures involving lists:

1. Be *very* optimistic! Since lists themselves are recursive data structures, most problems involving lists can be solved with recursive procedures.
2. Think of the simplest version of the problem, something you can already solve. This is the base case. For lists, this is usually the empty list.
3. Consider how you would solve a big version of the problem by using the result for a slightly smaller version of the problem. This is the recursive case. For lists, the smaller version of the problem is the rest (*cdr*) of the List.
4. Combine the base case and the recursive case to solve the problem.

As we develop programs to solve more complex problems, it is increasingly important to find ways to manage complexity. We need to break problems down into smaller parts that can be solved separately. Data abstraction hides the details of how data is represented from what you can do with it. Solutions to complex problems can be developed by thinking about what objects need to be modeled, and designing data abstractions the implement those models. Most of the work in solving the problem is defining the right datatypes; once we have the datatypes we need to model the problem well, we are usually well along the path to a solution.

One limitation of lists is that the only structure they provide is a linear sequence of elements; for some problems, it will be convenient to have richer structures such as trees (structures where each element can have more than one successor) and graphs (structures where elements can be connected in arbitrary ways). All of these structures can be built using just the Pair structure introduced in this chapter. In later chapters, we will explore these more complex data structures. Most of the concepts needed to construct and manipulate them follow directly from our simple List structures. Another problem with lists is that because of their sequential structure, many tasks cannot be efficiently performed on lists that could be performed efficiently on other data structures. In the following chapters, we consider the resources required to execute a procedure.