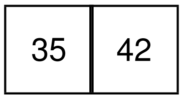
Chapter 5

Data

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. Although we could do arbitrarily complex computations using only scalar data, it would be very hard to understand those computations. 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. It would be very difficult to understand what is going on, though, and to write programs that correctly insert or replace characters in the text. Instead, we need more complex data structures that allow us to map the structured data we want to model more directly into structures our programs can manipulate. In this chapter, we develop those complex data structures and introduce techniques for defining procedures that manipulate structured data.

5.1 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. Here is a pair where the first component of the pair has the value 35 and the second component has the value 42:



We call a pair a *cons* cell, and all the first component the *car* of the cell, and the second component the *cdr* of the cell.[1] Scheme provides built-in procedures for constructing pairs and accessing their components:

- `(cons` *Expression Expression*`)` — evaluates to a pair where the first component is the value of the first *Expression* and the second component is the value of the second *Expression*
- `(car` *Expression*`)` — evaluates to the first component of the value of *Expression*, which must evaluate to a pair
- `(cdr` *Expression*`)` — evaluates to the second component of the value of *Expression*, which must evaluate to a pair

We can construct the pair shown in the picture by evaluating `(cons 35 42)`. DrScheme will display a pair by printing the components separated by a dot: (35 . 42). The interactions below show some uses of `cons`, `car`, and `cdr`.

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

Of course, the components of a pair can be any values, including other pairs! For example,

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

[1]The names come from the original LISP implementation on the IBM 704. The name `car` is short for "Contents of the Address part of the Register" and the name `cdr` (pronounced "could-er") is short for "Contents of the Decrement part of the Register".

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

The last expression results in an error, since it attempts to apply `car` to a value that is not a pair.

Exercise 5.1. Suppose the following definition has been executed:

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

Explain what each of the following expressions evaluates to, then try evaluating them in a Scheme interpreter to check your answers.

a. `(cdr tpair)`

b. `(car (car (car tpair)))`

c. `(cdr (cdr (car tpair)))`

d. `(car (cdr (cdr tpair)))`

■

Exercise 5.2. Suppose the following definition has been executed:

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

Write expressions that use `cars` and `cdrs` to extract each of the four elements from `fpair`. ■

5.1.1 Making Pairs

Although Scheme provides the built-in procedures `cons`, `car`, and `cdr` for creating pairs and accessing their parts, 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:

```
(define (cons a b)
   (lambda (w) (if w a b)))

(define (car pair) (pair #t))
(define (cdr pair) (pair #f))
```

The `cons` 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 parts of the pair to output. If the selector is true, the first part is the output; if false, the second part is the output. The `car` and `cdr` procedures apply a pair constructed by `cons` to either `#t` (to select the first part in `car`) or `#f` (to select the second part in `cdr`).

Exercise 5.3. Convince yourself the definitions of `cons`, `car`, and `cdr` above work as expected by following the evaluation rules to evaluate

```
(car (cons 1 2))
```

■

Exercise 5.4. Above, we illustrated how `cons`, `car`, and `cdr` could be defined. Suppose we instead defined `cons` using:

```
(define (cons a b)
   (lambda (w) (if w b a)))
```

Show the corresponding definitions of `car` and `cdr` needed to provide the correct `cons`, `car`, and `cdr` behavior. ■

5.1.2 Making Octuples

A pair is a construct that has two components. 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))
```

Here, we need three different selector values, so instead of using `#t` and `#f`, we use `0`, `1`, and `2`.

A simpler way of making a triple would be to make a pair, where the second part of the pair is 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 where the second part of the pair 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 tuples with more and more elements:

- A *quintuple* is a pair where the second part is a *quadruple*.
- A *sextuple* is a pair where the second part is a *quintuple*.
- A *septuple* is a pair where the second part is a *sextuple*.
- · · ·
- A *(+ n 1)-uple* is a pair where the second part of the pair is an *n-uple*.

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

Exercise 5.5. Define a procedure that constructs a quintuple and procedures for selecting the five elements of a quintuple. ■

Exercise 5.6. Another way of thinking of a triple is as a pair where the first part is a pair (and the second part is a scalar). Provide definitions of `make-triple`, `triple-first`, `triple-second`, and `triple-third` for this alternative. ■

5.2 Lists

In the previous section, we saw how to construct arbitrarily large tuples building from the `cons` pair. This way of managing data is not very satisfying, however. It requires defining different procedures for constructing and accessing elements of every length tuple. In many cases, we want to define procedures that work on any length tuple. But we could not do it with these tuples, since we need different procedures depending on whether we are accessing the second element of a pair, triple, quadruple, etc.

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

- An *any-uple* is a pair where the second part of the pair is an *any-uple*.

This would 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, and defining *MoreExpression* in the Scheme grammar in Chapter 3 as zero or more *Expression*s. Recall the grammar rules for *MoreExpressions*:

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

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. The primitive `null` evaluates to 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 procedure `null?` takes one input parameter and evaluates to #t if and only if the value of that parameter is null.

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

- The value null is a *list*.
- A pair where the second element is a *list*, is a *list*.

Symbolically, we can define a list as:

list	::⇒	`null`
list	::⇒	`(cons` *`element`* *list*`)`

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 using `cons`. For example:

- `null` evaluates to a list containing no elements.
- `(cons 1 null)` evaluates to containing one element, 1.
- `(cons 1 (cons 2 null))` evaluates to a list containing two elements, 1 and 2.

Scheme provides a more convenient procedure, `list`, for constructing lists. 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 expressions above: `(list)`, `(list 1)`, and `(list 1 2)`.

Exercise 5.7. 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 #t if the value of that input is a list, and #f 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)`

■

5.3 List Procedures

Since the list data structure is defined recursively, we can define recursive procedures to examine and manipulate lists. Whereas recursive procedures on inputs that are numbers usually used zero as the base case, for lists the base case is most naturally the case of the empty list. With numbers, we made progress by subtracting one to produce the input for the recursive application; with lists, we will use `cdr` to use the rest of the list as the input for the recursive application. Since the list was constructed using `cons`, this means we need to figure out what to do with the first element (`car`) of the list, and apply the recursive procedure to the rest (`cdr`) of the list.

5.3.1 Procedures that Examine Lists

These procedures 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. Scheme provides a procedure `length` that takes a list as its input and outputs the number of elements in the list. Here, we define the `length` procedure (without using the built-in procedure).

First, consider the base case when the list is null. The length of the empty list is 0. For the recursive case, we need to consider the case where the input list `p` is `(cons` *`element rest`*`)`. The length of this list is one more than the length of the *rest* list.

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

Here are a few example applications of our `length` procedure:

```
> (length null)
0
> (length (cons 0 null))
```

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

Note that the last evaluation produces and error, since we are applying `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.

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

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

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

Exercise 5.8. (◇) Define a procedure `list?` that takes one input and produces #t if the input is a list, and #f otherwise. (Scheme provides a built-in `list?` procedure with this behavior, but you should not use it in your definition.) Check that the results of your `list?` procedure match the results from the built-in `list?` procedure on the inputs from Exercise 5.2. (Hint: start from the definition of a list in Section 5.2.) ■

Exercise 5.9. (⋆) Define a procedure `max-list` 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). ■

5.3.2 Generic Accumulators

The `length`, `sum-list`, and `product-list` procedures all have very similar structures. They all match the pattern:

```
(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 parameters:

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

We can define the `sum-list` procedure using `accumulate`:

```
(define (sum-list p)
  (accumulate + 0 p))
```

Defining the `length` procedure is a bit more complicated. For the `f` procedure parameter, we need to pass in a procedure that takes two inputs—the first input is the first element of the list; the second input is the result of applying `accumulate` to the rest of the list. For the `length` procedure, the first parameter is not used since the values of list elements do not impact the length of the list.

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

Exercise 5.10.

a. Define the `product-list` procedure using `accumulate`.

b. Define the `list?` procedure (from Exercise 5.3.1) using `accumulate`.

c. Define the `max-list` procedure (from Exercise 5.3.1) using `accumulate`.

■

Exercise 5.11. (⋆) Define a procedure `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, `(ordered? < (list 1 2 3))` should evaluate to #t, and `(ordered? > (list 4 3 3 2))` should evaluate to #f. ■

5.3.3 Procedure 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, but since we are producing a list as output the result for the base case is usually null. Since the output is a list, 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 5.3: 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 (square-all p)
  (if (null? p) null
      (cons (square (car p))
            (square-all (cdr p)))))
```

We can generalize this by making the procedure to apply to all list elements a parameter. This is the `map` procedure[2]:

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

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

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

Example 5.4: 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, `(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 null. 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. 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 (filter-negative p)
   (if (null? p) null
       (if (>= (car p) 0)
           (cons (car p)
                 (filter-negative (cdr p)))
           (filter-negative (cdr p)))))
```

[2]Scheme provides a built-in `map` procedure. It behaves like this one when passed a procedure and a single list as parameters, but can also work on more than one list parameter.

Similarly to `map`, we can generalize our filter by taking a test procedure as a parameter, so we can use any predicate to determine which elements to include in the output list.

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

Using the `filter` procedure, we could define `filter-negative` as:

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

We could also define the `filter` procedure using the `accumulate` procedure from Section 5.3.1:

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

Exercise 5.12. Define a procedure `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. ■

Exercise 5.13. Define a procedure `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, `(remove (lambda (x) (= x 0)) (list 0 1 2 3))` should evaluates to the list (1 2 3). ■

Exercise 5.14. ($\star\star\star$) Define a procedure `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. Use the `equal?` procedure to determine if two elements have the same value. ■

Example 5.5: Append. The `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. 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 (append p q)
  (if (null? p) q
      (cons (car p) (append (cdr p) q))))
```

Example 5.6: Reverse. The `reverse` procedure takes a list as input and produces as output a list containing the elements of the input list in reverse order. For example, `(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 null. To reverse a non-empty list, we should put the first element of the list at the end of the reverse of 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 `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))`.

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

Exercise 5.15. ($\star$) Define the `reverse` procedure using `accumulate`. ■

Example 5.7: 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 have to 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 should be the empty list. For higher values, the output is the result of putting the input value inserted 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 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 desending order: `(revintsto 5)` evaluates to (5 4 3 2 1).

If you keep proving stuff that others have done, getting confidence, increasing the complexities of your solutions for the fun of it then one day you'll turn around and discover that nobody actually did that one! And that's the way to become a computer scientist.
Richard Feynman, *Lectures on Computation*

One solution is to reverse the result by composing `reverse` with `revintsto`:

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

Or, use the `compose` procedure from Section 4.2:

```
(define intsto
  (compose reverse revintsto))
```

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

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

Exercise 5.16. (◇) Define the `factorial` procedure (from Section 4.3) using `intsto`. ■

5.4 Summary

Building from the simple `cons` procedure that constructs a pair, we can create complex data structures including lists. A list is either null, or a pair whose second part 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 when the list is null.

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.

One limitation of lists is that the only structure they provide is a sequence of elements; for some problems, it will be convenient to have richer structures such as multi-dimensional lists (lists of lists), 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 `cons` pair 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 next chapter, we begin to consider the resources required to execute a procedure.