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 construct.

5.3 Lists

In the previous section, we saw how to construct arbitrarily large tuples from Pairs. This way of managing data is not very satisfying since 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. We need a data type that can hold *any* number of items.

This definition 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 *Expression*s, and recursive composition in Chapter 4.

Recall the grammar rules for *MoreExpressions*:

> *MoreExpressions* ::⇒ *Expression MoreExpressions*
> *MoreExpressions* ::⇒ ϵ

The rule for constructing an *any-uple* is analogous 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*

DrRacket 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`.

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

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

Symbolically, we define a List as:

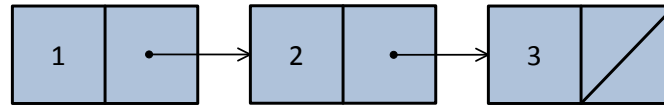
> *List* ::⇒ `null`
> *List* ::⇒ (*cons Value List*)

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:

- *null* evaluates to a List containing no elements.
- (*cons* 1 *null*) evaluates to a List containing one element.
- (*cons* 1 (*cons* 2 *null*)) evaluates to a List containing two elements.
- (*cons* 1 (*cons* 2 (*cons* 3 *null*))) evaluates to a 3-element List.

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 2), and (*list* 1 2 3).

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 2 3):



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.

Exercise 5.10. For each of the following expressions, explain whether or not the expression evaluates to a List. Check your answers with a Scheme interpreter by using the *list?* procedure.

a. *null*

b. (*cons* 1 2)

c. (*cons null null*)

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

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

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

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

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

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 `0` as the base case, for lists the most common base case is *null*. With numbers, we make progress by subtracting `1`; 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.

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 usually the rest (*cdr*) of the List.
4. Combine the base case and the recursive case to solve the problem.

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 output lists.

5.4.1 Procedures that Examine Lists

All of the example procedures in this section take a single List as input and produce a scalar value that depends on the elements of the List as output. These procedures have 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 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 from our definition that a List is 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.

[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-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
```

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

We can define *list-product* similarly, using $*$ in place of $+$. The base case result cannot be 0, though, since then the final result would always be 0 since any number multiplied by 0 is 0. 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 outputs true if the input is a List, and false otherwise. Your procedure should behave identically to the built-in *list?* procedure, but you should not use *list?* in your definition.

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 1 2 0)) should evaluate to 2.

5.4.2 Generic Accumulators

The *list-length*, *list-sum*, and *list-product* procedures all have very similar structures. 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)))))
```

We can use *list-accumulate* to define *list-sum* and *list-product*:

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

Defining the *list-length* procedure is a bit less natural. The recursive case in the original *list-length* procedure is (+ 1 (*list-length* (*cdr p*))); it does not use the value of the first element of the List. But, *list-accumulate* is defined to take 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:

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

Exercise 5.13. Use *list-accumulate* to define *list-max* (from Exercise 5.12).

Exercise 5.14. [⋆] Use *list-accumulate* to define *is-list?* (from Exercise 5.11).

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)))
```

We want a more general procedure that can access any selected list element. It takes two inputs: a List, and an index Number that identifies the element. If we start counting from 1 (it is often more natural to start from 0), 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. 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* to 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 is (*list-get-element* (*list* 2) 2). The second recursive call is (*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 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*. Programming defensively helps avoid tricky to debug errors and makes it easier to understand what went wrong if there is an error.

Exercise 5.15. Define a procedure *list-last-element* that takes as input a List and outputs the last element of the input List. If the input List is empty, *list-last-element* should produce an error.

Exercise 5.16. Define a procedure *list-ordered?* that takes two inputs, a test procedure and a List. It outputs true if all the elements of the List are ordered according to the test procedure. For example, (*list-ordered?* $<$ (*list* 1 2 3)) evaluates to true, and (*list-ordered?* $<$ (*list* 1 2 3 2)) evaluates to false. Hint: think about what the output should be for the empty list.

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 some procedure to every element in the input List.

For the base case, applying any procedure to every element of the empty list produces the empty list. For the recursive case, we use *cons* to construct a List. The first element is the result of applying the mapping procedure to the first element of the input List. The rest of the output List is the result of recursively mapping the rest of the input List.

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 generalize this by making the procedure which is applied to each 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)))))
```

We can use *list-map* to define *square-all*:

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

Exercise 5.17. Define a procedure *list-increment* that takes as input a List of numbers, and produces as output a List containing each element in the input List incremented by one. For example, (*list-increment* 1 2 3) evaluates to (2 3 4).

Exercise 5.18. 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)) evaluates to (1 5 0).

First, consider the base case when the input is the empty list. 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 whether or not the first element should be included in the output. 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 skip 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

[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 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 5.19. 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.

Exercise 5.20. 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 5.21. [⋆⋆] 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, but should only contain the first appearance of each value in the input List.

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))))
```

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

[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.

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)) evaluates 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*)).

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

Exercise 5.22. Define the *list-reverse* procedure using *list-accumulate*.

Example 5.8: Intsto

For our final example, we define the *intsto* procedure that constructs a List containing the whole numbers between 1 and the input parameter value. For example, (*intsto* 5) evaluates to the List (1 2 3 4 5).

This example combines 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 list 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 (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*.

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