Chapter 16

Lists and List-Based Recursion

Previous chapters have highlighted the many important roles that non-empty lists play in Scheme's computational model. For example, the Default Rule for evaluating non-empty lists can be used to apply functions to inputs, the `define` special form can be used to assign values to variables, the `quote` special form can be used to shield a datum from evaluation, and so on. In contrast, this chapter focuses on lists as *containers of data.* When viewing lists as containers of data, we typically don't want them to be evaluated. In addition, to do any meaningful computations involving lists (e.g., to sort a list of numbers or recursively walk through a list of data), we need to be able to *access* the individual elements. Finally, we will often want to be able to construct lists *incrementally*, for example, by attaching a new element to the front of a list.

Scheme provides the following built-in functions to facilitate the use of lists as containers of data:

`first`	to access the *first* element of a list
`rest`	to access the *rest* of a list
`cons`	to construct a new list by attaching a new element to the front of an existing list

These few functions, together with the `null?` type-checker predicate from Chapter 8, will enable us to design functions that can recursively process the elements in a list.

We shall see that list-based recursion is quite similar to numerical recursion. Whereas numerical recursion is driven by the size of a numerical input, list-based recursion is driven by some feature of a list—usually whether that list is empty or not. In list-based recursion, there is a base case—usually signaled by the empty list (analogous to $n = 0$); and there is a recursive case—usually signaled by a non-empty list (analogous to $n > 0$). And, just as a numerical-recursive function can typically process numerical inputs of any size, a list-based recursive function can typically process lists containing any number of elements.

16.1 The Built-in Functions: `first`, `rest` and `cons`

This section describes the built-in functions, `first`, `rest` and `cons`, that Scheme provides to enable us to access parts of lists, and to attach new elements to pre-existing lists.

The `first` and `rest` accessor functions. The `first` and `rest` functions are called *accessor functions* because they enable us to access certain parts of a non-empty list. The contracts for these built-in functions are given below.

```
;;  FIRST -- built-in function
;; ------------------------------------------------------
;;  INPUT:   LISTY, a non-empty list
;;  OUTPUT:  The FIRST element of LISTY
```

```
;;  REST -- built-in function
;; -----------------------------------------------------
;;  INPUT:   LISTY, a non-empty list
;;  OUTPUT:  The REST of LISTY (i.e., the portion of LISTY
;;           that contains all but its first element)
```

⋆ Note that the *rest* of a non-empty list is necessarily a list.

Example 16.1.1

The following Interactions Window session demonstrates the use of the `first` *and* `rest` *accessor functions to access the parts of a non-empty list.*

```
> (first '(a b c d e))
a
> (rest '(a b c d e))
(b c d e)
> (first '(64))
64
> (rest '(64))
()                              ⟵the rest is a list, even if it is empty
```

Example 16.1.2: Accessing other elements of a non-empty list

We can combine the `first` *and* `rest` *functions to access any individual element of a list, as follows:*

```
> (first (rest '(a b c d e)))                ⟵ access second element
b
> (first (rest (rest '(a b c d e))))         ⟵ access third element
c
> (first (rest (rest (rest '(a b c d e))))) ⟵ access fourth element
d
```

Rather than re-typing these sorts of cumbersome expressions to access various elements of a list, we can define functions to simplify the process, as illustrated below:

```
;;  SEKUND/THURD/FORTH
;; -------------------------------
;;  INPUT:   LISTY, a list containing at least two elements
;;  OUTPUT:  The second/third/fourth element of LISTY

(define sekund
  (lambda (listy)
    (first (rest listy))))

(define thurd
  (lambda (listy)
    (first (rest (rest listy)))))

(define forth
  (lambda (listy)
    (first (rest (rest (rest listy))))))
```

The following interactions demonstrate the use of these functions:

```
> (sekund '(a b c d e))
b
> (thurd '(yes #t 383 () why))
383
> (forth '(my bonnie lies over the ocean))
over
```

Although we could continue in this fashion, defining additional accessor functions called `fiffth`, `sicksth`, *and so on, we shall soon discover that there is a much easier way to access any desired element of a list: using recursion! In the meantime, you should know that Scheme provides a slew of built-in functions for accessing individual elements of a list in the manner seen above. They are called* `second`, `third`, `fourth`, *etc. As you may have guessed, the existence of these built-in functions is the reason that I gave names such as* `sekund`, `thurd` *and* `forth` *to the functions defined above.*

In-Class Problem 16.1.1: Checking for a one-element list

Define a function, called `one-elt-list?`, *that satisfies the following contract:*

```
;;  ONE-ELT-LIST?
;; ------------------------------------------------------
;;  INPUTS:  LISTY, any list
;;  OUTPUT:  #t if LISTY contains *exactly* one element;
;;           #f otherwise.
```

Here are some examples of the desired behavior:

```
> (one-elt-list? ())
#f
> (one-elt-list? '(xyz))
#t
> (one-elt-list? '(a b c d))
#f
```

Hint: Use some *of these:* `null?`, `first`, `rest`.

Using `cons` **to construct a new list.** The built-in `cons` function constructs a new list by attaching a new element onto the front of an existing list. Here is its contract:

```
;;  CONS -- built-in function
;; ------------------------------------------------------------
;;  INPUTS:  FST, any Scheme datum
;;           RST, a list (either empty or non-empty)
;;  OUTPUT:  A new list whose FIRST element is FST, and
;;           the REST of whose elements are RST.
```

⋆ When using the `cons` function to construct a new list, the second input must be a list!

Example 16.1.3

The following Interactions Window session demonstrates the use of the `cons` *function.*

```
> (cons 8 '(a b c))
(8 a b c)
> (cons 'john '(paul george ringo))
(john paul george ringo)
> (cons 64 ())              ⟵ the second input must be a list, even if it is empty
(64)
> (define my-list '(a b c))
> (define new-list (cons 'x my-list))
> new-list
(x a b c)
> my-list
(a b c)
```

The last example shows that the `cons` *function is non-destructive. The new list* `(x a b c)` *formed by attaching* `x` *to the front of* `my-list` *does not change* `my-list`.

In-Class Problem 16.1.2: Using `cons` to create short lists

Define functions, called `list-one` *and* `list-two`*, that satisfy the following contracts:*

```
;;  LIST-ONE
;; -------------------------------------------------------------
;;  INPUT:   DATUM, anything
;;  OUTPUT:  A list that contains DATUM as its only element

;;  LIST-TWO
;; -------------------------------------------------------------
;;  INPUTS:  ONE, TWO, anything
;;  OUTPUT:  A list whose first element is ONE, and whose
;;           second element is TWO
```

Here are examples of the desired behavior:

```
> (list-one 'a)
(a)
> (define listy '(a b c))
> (define symby 'xyz)
> (list-one listy)
((a b c))
> 'listy                    ⟵ quote produces different results!
listy
> (list-one symby)
(xyz)
> 'symby                    ⟵ quote produces different results!
symby
> (list-two 'a 'b)
(a b)
> (list-two listy symby)
```

```
((a b c) xyz)
> '(listy symby)            ⟵ quote produces different results!
(listy symby)
```

Hint: Use the built-in `cons` *function.*

There is a built-in function, called `list`, that takes any number of inputs. It returns as its output a list containing those inputs, as illustrated below:

```
> (list 'a (+ 2 3) #f)
(a 5 #f)
```

Notice the difference between result obtained from the above example and that obtained by evaluating the following `quote` special form.

```
> '(a (+ 2 3) #f)
(a (+ 2 3) #f)
```

16.2 List-based Recursion

Chapter 12 introduced recursive functions for which the recursion was driven by the size of a number. For example, in the factorial function (cf. Example 12.1.1), $f(4)$ was computed by multiplying 4 by $f(3)$, where $f(3)$ was computed by multiplying 3 by $f(2)$, where $f(2)$ was computed by multiplying 2 by $f(1)$, and where $f(1) = 1$ terminated the recursion. The relevant sequence of computations is shown below:

$f(4) = 4 \cdot f(3)$	Recursive call: $f(4) = 4 \cdot f(3)$
$= 4 \cdot (3 \cdot f(2))$	Recursive call: $f(3) = 3 \cdot f(2)$
$= 4 \cdot (3 \cdot (2 \cdot f(1)))$	Recursive call: $f(2) = 2 \cdot f(1)$
$= 4 \cdot (3 \cdot (2 \cdot 1))$	Base case: $f(1) = 1$
$= 4 \cdot (3 \cdot 2)$	$2 \cdot 1 = 2$
$= 4 \cdot 6$	$3 \cdot 2 = 6$
$= 24$	$4 \cdot 6 = 24$

More generally, for any $n > 1$, the factorial of n can be computed by making a sequence of $n - 1$ recursive function calls, terminating in the base case, where $f(1) = 1$. Of course, numerical recursion can take many forms. For example, the input n might start out at 0 and increase by 3 on each recursive function call until some stopping value (e.g., 90) is reached. Or the value of n might be multiplied by some value at each recursive function call. But the common feature is that deciding between the base case and the recursive case is based on the size of some number.

This section introduces *list-based recursion.* In list-based recursion the recursion is driven not by the size of a number, but by some feature of a list. In many cases, the relevant feature is simply whether a certain list is empty or not: if the list is empty, we're in the base case; otherwise, we're in the recursive case. For example, if a typical recursive function is applied to a list containing, say, five elements, then, because that list is non-empty, a recursive function call will be made on the *rest* of that list (i.e., a list containing four elements). And because *that* list is non-empty, another recursive function call will be made, this time on the *rest* of *that* list (i.e., a list containing three elements). The sequence of recursive function calls will eventually lead to the function being applied to the empty list, at which point the base case will terminate the recursion. This common kind of list-based recursion is explored in the following example.

Example 16.2.1

Suppose we are given the following contract for a function called `mult-all`*:*

```
;;  MULT-ALL
;; ------------------------------------------------------
;;  INPUT:   LISTY, a list of numbers
;;  OUTPUT:  The product of all the elements of LISTY
```

Here are some examples of the desired behavior:

```
> (mult-all '(2 3 4 10))
240
> (mult-all '(10 2 4))
80
```

This function can be defined recursively since:

(the product of all *of the elements of a non-empty list)*

$$= \begin{cases} \textit{(the } \text{first} \textit{ element of the list)} \\ \times \\ \textit{(the product of the } \text{rest} \textit{ of the elements of the list)} \end{cases}$$

For example:

(the product of all of the elements of `(2 3 4 10)`*)*

$$= \begin{cases} 2 \\ \times \\ \textit{(the product of all of the elements of } \texttt{(3 4 10)}\textit{)} \end{cases}$$

Stated in terms of the `mult-all` *function, where* `listy` *is a variable whose value is* `(2 3 4 10)`*:*

```
(mult-all listy) ⇒ (* (first listy) (mult-all (rest listy)))
```

Note that if this relationship is going to hold for all non-empty lists, then `(mult-all ())` *must evaluate to* `1` *(i.e., the multiplicative identity), as illustrated below:*

```
(mult-all '(4)) ⇒ (* 4 (mult-all ())) ⇒ (* 4 1) ⇒ 4
```

In view of all of the above, we might imagine the evaluation of `(mult-all '(2 3 4 10))` *proceeding as follows, where, for example, the recursive function call on the* rest *of the list* `(2 3 4 10)` *is represented by* `(mult-all '(3 4 10))`*:*

```
(mult-all '(2 3 4 10))                              ⟵ Recursive Case
    ⇒ (* 2 (mult-all '(3 4 10)))                    ⟵ Recursive Case
    ⇒ (* 2 (* 3 (mult-all '(4 10))))                ⟵ Recursive Case
    ⇒ (* 2 (* 3 (* 4 (mult-all '(10)))))            ⟵ Recursive Case
    ⇒ (* 2 (* 3 (* 4 (* 10 (mult-all ())))))        ⟵ Base Case
    ⇒ (* 2 (* 3 (* 4 (* 10 1))))
    ⇒ (* 2 (* 3 (* 4 10)))
    ⇒ (* 2 (* 3 40))
    ⇒ (* 2 120)
    ⇒ 240
```

As long as the list in question is non-empty, the recursive case evaluates an expression of the form `(* (first some-list) (mult-all (rest some-list)))`. *However, when the list in question is empty, the base case is reached, terminating the recursion. These sorts of considerations lead to the following solution:*

```
(define mult-all
  (lambda (listy)
    (cond
      ;; Base Case:  LISTY is empty
      ((null? listy)
       ;; The product of all the elements of the empty list is
       ;; taken to be 1, the multiplicative identity.
       1)
      ;; Recursive Case:  LISTY is non-empty (and so we can use
      ;;   the FIRST and REST accessor functions on LISTY)
      (else
       ;; The product of all of the elements of LISTY is obtained
       ;; by multiplying the FIRST element of LISTY by the
       ;; product of all of the REST of the elements of LISTY.
       ;; The latter job is handled by the recursive func. call.
       (* (first listy)
          (mult-all (rest listy)))))))
```

Example 16.2.2: Summing the numbers in a list

The following defines a `sum-all` *function that sums the numbers in the input list. Its structure is similar to that of the* `mult-all` *function.*

```
;;  SUM-ALL
;; ------------------------------------------------
;;  INPUT:   LISTY, a list of numbers
;;  OUTPUT:  The sum of all the elements of LISTY

(define sum-all
  (lambda (listy)
    (cond
      ;; Base Case:  LISTY is empty
      ((null? listy)
       ;; The sum of all the elements of the empty list
       0)
      ;; Recursive Case:  LISTY is non-empty
      (else
       ;; The recursive function call computes the sum of all
       ;; the numbers in the rest of LISTY; we just add on the
       ;; first element.
       (+ (first listy) (sum-all (rest listy)))))))

> (sum-all '(1 2 3 4))
10
> (sum-all '(1 10 100 1000))
1111
```

```
> (sum-all '(2 5 3 8 1))
19
```

In-Class Problem 16.2.1

Define a function, called `add-squares`*, that satisfies the following contract:*

```
;;  ADD-SQUARES
;; ---------------------------------------
;;  INPUT:  LISTY, a list of numbers
;;  OUTPUT: The sum of the squares of the numbers in LISTY
```

Here are some examples of the desired behavior:

```
> (add-squares '(2 3 10))      ⟵  2^2 + 3^2 + 10^2 = 4 + 9 + 100 = 113
113
> (add-squares '(1 0 5 2))     ⟵  1^2 + 0^2 + 5^2 + 2^2 = 1 + 0 + 25 + 4 = 30
30
```

In-Class Problem 16.2.2: Computing the length of a list

Define a function, called `lengthy`*, that computes the number of elements of the input list. Here is its contract:*

```
;;  LENGTHY
;; ------------------------------------------------------------
;;  INPUT:  LISTY, any list
;;  OUTPUT: The number of elements of LISTY (i.e., its length)
```

Here are some examples of the desired behavior:

```
> (lengthy '(a b c d e))
5
> (lengthy '(#t () 22 xyz))
4
```

Hints: Use list-based recursion. What's the relationship between the length of `listy` *and the length of* `(rest listy)`*? And how many elements are in the empty list?*

Incidentally, now that you know how to define a function to compute the length of a list, it's time to tell you that there is a built-in function, called `length`, that does just that!

In-Class Problem 16.2.3: Accessing the N^{th} element of a list

Define a function, called `fetch-nth-element`*, that satisfies the following contract:*

```
;;  FETCH-NTH-ELEMENT
;; ------------------------------------------------------------
```

```
;;  INPUTS:  LISTY, a list
;;           N, a non-negative integer treated as an "index"
;;  OUTPUT:  Returns the Nth element of LISTY
;;           (or #f if LISTY doesn't have an Nth element)
;;  NOTE:    The elements of LISTY are indexed starting at 0.
```

Thus, for example, `a` *is considered to be the zeroeth element of the list* `(a b c d e)`*, while* `c` *is considered to be the element with index* `2`*. Thus, the elements in a list containing five elements will have indices ranging from 0 to 4, inclusive. Here are some examples of the behavior of the* `fetch-nth-element` *function:*

```
> (fetch-nth-element '(a b c d e) 0)
a
> (fetch-nth-element '(a b c d e) 2)
c
> (fetch-nth-element '(a b c d e) 8)
#f
```

Incidentally, now that you know how to implement the `fetch-nth-element` function, I can tell you that there is a built-in function, called `list-ref`, that does the same thing. Like `fetch-nth-element`, the `list-ref` function treats the first element of a list as having index `0`.

Example 16.2.3

Suppose we want to define a function called `is-elt-of?` *that satisfies the following contract:*

```
;;  IS-ELT-OF?
;; ------------------------------------------------------------
;;  INPUTS:  ITEM, anything
;;           LISTY, a list of stuff
;;  OUTPUT:  #t (or something that counts as true) if ITEM
;;             appears as an element of LISTY -- as judged by EQ?
;;           #f otherwise.
```

Here are examples of the desired behavior:

```
> (is-elt-of? 3 '(3 4 5))
#t
> (is-elt-of? 3 '(1 2 3 4 5))
#t
> (is-elt-of? 'x '(a b a b a))
#f
```

Consider the first example, where `ITEM` *is* `3`*, and* `LISTY` *is* `(3 4 5)`*. In this case, it is clear that* `ITEM` *appears in* `LISTY` *because it appears as the* first *element. (Notice that this is a kind of base case since, once we find an occurrence of* `ITEM` *in* `LISTY`*, there is no need to continue looking any further.) On the other hand, in the second example, where* `ITEM` *is* `3`*, and* `LISTY` *is* `(1 2 3 4 5)`*, it is true that* `ITEM` *appears in* `LISTY` *because, as a sequence of recursive functions call might discover,* `ITEM` *appears somewhere in the* rest *of* `LISTY`*. Finally, in the third example, where* `ITEM` *is* `x`*, and* `LISTY` *is* `(a b a b a)`*, we could imagine a sequence of recursive function calls that never discover an occurrence of* `x`*,*

eventually leading to the base case: `(is-elt-of? 'x ())`*, which must evaluate to* `#f`*, since nothing can appear as an element of the empty list.*

In view of these considerations, we are led to the following solution:

```
(define is-elt-of?
  (lambda (item listy)
    (cond
     ;; Base Case 1:  LISTY is EMPTY
     ((null? listy)
      ;; No occurrence of ITEM in the empty list
      #f)
     ;; Base Case 2:  ITEM appears as first element of LISTY
     ((eq? item (first listy))
      ;; We found ITEM in LISTY!
      #t)
     ;; Recursive Case:  Haven't found ITEM in LISTY yet
     (else
      ;; Keep looking
      (is-elt-of? item (rest listy))))))
```

Notice that we must check whether `LISTY` *is empty* before *trying to use* `first` *or* `rest`*, since those accessor functions can only be used on* non-empty *lists.*

Example 16.2.4: The built-in `member` function

Now that you know how to define the `is-elt-of?` *function, I can tell you that there is a built-in function, called* `member`*, that does the same thing! The only difference is that the value returned by* `member`*, in cases where it finds* `ITEM` *in* `LISTY`*, is the portion of* `LISTY` *that starts from the first occurrence of* `ITEM`*, as illustrated below:*

```
> (member 3 '(1 2 3 4 5))
(3 4 5)
> (member 'x '(a b c d e f x y z))
(x y z)
```

Recall that anything other than than `#f` *counts as true. So, expressions such as the following are handled appropriately:*

```
> (if (member 3 '(1 2 3 4 5)) 'say_yes 'say_no)
say_yes
```

In this case, the condition evaluated to the list `(3 4 5)`*, which counts as true, so the* `if` *special form evaluated the expression '*`say_yes`*, generating the output value* `say_yes`*. For this reason, it does no harm for* `member` *to return something that* counts *as true. Furthermore, in some cases, you might be glad to have access to the list returned by* `member` *as its output.*

Example 16.2.5: An alternative implementation of `is-elt-of?`

Recall from Section 13.3 that, when defining a predicate (i.e., a function that returns a boolean value), one can often write the body of the function using the boolean operators, `and`, `or` *and* `not`, *instead of the conditional expressions,* `if` *or* `cond`. *Recall further that:*

- ⋆ *When defining a predicate using only the boolean operators, the body of the predicate should specify the conditions under which the predicate should output the value* `#t` *(or something that counts as true).*

Regarding `(is-elt-of? item listy)`, *we know that it will evaluate to* `#f` *if* `listy` *is empty; therefore, it can only evaluate to* `#t` *if* `listy` *is non-empty. However, that is not enough. In addition, we need to* find `item` *somewhere in* `listy`. *What are the possibilities? Well,* `item` *can appear either as the* first *element of* `listy`, *or somewhere in the* rest *of* `listy`. *These considerations lead to the following alternative definition of the* `is-elt-of?` *function. To distinguish the two versions, we call this one* `is-elt-of-alt?`.

```
(define is-elt-of-alt?
  (lambda (item listy)
    ;; The following expression specifies the conditions under
    ;; which this function should output #t (or something that
    ;; counts as true):
    ;;   (1) LISTY must NOT be empty;
    ;;   AND
    ;;   (2) ITEM must appear as the FIRST element of LISTY
    ;;       OR
    ;;       ITEM must appear somewhere in the REST of LISTY
    (and (not (null? listy))
         (or (eq? item (first listy))
             (is-elt-of-alt? item (rest listy))))))
```

Try using this function in the Interactions Window to confirm that it works as advertised.

In-Class Problem 16.2.4: Is a list of numbers in increasing order?

Define a function, called `incr?`, *that satisfies the following contract:*

```
;;  INCR?
;; -------------------------------------------------------
;;  INPUT:   LISTY, a non-empty list of numbers
;;  OUTPUT:  #t if the numbers in LISTY are in strictly
;;             *increasing* order; #f otherwise
```

Here are some examples illustrating its behavior:

```
> (incr? '(1 3 8 9 15))
#t
> (incr? '(1 3 4 4 6 9))          ⟵ Not strictly increasing
#f
> (incr? '(2 5 8 5 2))
#f
```

- ⋆ *What's the best way of checking whether the input list contains exactly one element?*

Write one version of `incr?` *that uses* `if` *or* `cond`*, and another that uses some combination of* `and`*,* `or` *and* `not`*.*

Example 16.2.6: Printing a histogram

The goal for this example is to define a function, called `print-histy`*, that satisfies the following contract:*

```
;;  PRINT-HISTY
;; ------------------------------------------------------------
;;  INPUT:   LISTY, a list of non-negative integers
;;  OUTPUT:  None
;;  SIDE EFFECT: Displays a histogram in the Interactions Window
;;    based on the numbers in LISTY.  In particular, for each
;;    number in LISTY, prints one row of that many asterisks.
```

Here are some examples of the desired behavior:

```
> (print-histy '(3 2 8 4 6))
***
**
********
****
******
> (print-histy '(1 2 3 4))
*
**
***
****
```

Consider the first example: `(print-histy '(3 2 8 4 6))`*. The beauty of recursive programming is that we can write a function that explicitly does only a small part of the job, while leaving most of the work to the recursive function call. For example, to print out the desired histogram, we can just print out the first row of* `3` *asterisks, and then let the recursive function call take care of printing the rest of the histogram, based on the rest of the list (i.e.,* `(2 8 4 6)`*). Of course, in the base case, when the list is empty, we're all done!*

```
(define print-histy
  (lambda (listy)
    (cond
     ;; Base Case:  LISTY is empty
     ((null? listy)
      ;; Use the built-in VOID function to do ... nothing!
      (void))
     ;; Recursive Case:  LISTY is non-empty
     (else
      ;; Use a helper function to print one row of the histogram
      (print-n-stars (first listy))
      ;; Then print out the rest of the histogram
      (print-histy (rest listy))))))
```

Notice that since there's nothing to do in the base case, we just use the built-in `void` *function to do . . . nothing! (Recall from Section 2.1.4, the* `void` *function acutally outputs the special* void *value which DrScheme interprets as "no output".) Here's the helper function, which is a slight re-write of the* `print-n-dashes` *function from Example 14.2.1:*

```
(define print-n-stars
  (lambda (n)
    (cond
     ((<= n 0)
      (newline))
     (else
      (printf "*")
      (print-n-stars (- n 1))))))
```

Finally, note that because the `print-histy` *function does nothing in the base case, returning* void *as its output, the* `print-histy` *function can be re-written as follows, using the* `when` *special form.*

```
(define print-histy
  (lambda (listy)
    ;; Base Case:  LISTY is empty, do nothing.
    ;; Recursive Case:  LISTY is non-empty
    (when (not (null? listy))
      ;; Use a helper function to print one row of the histogram
      (print-n-stars (first listy))
      ;; Then print out the rest of the histogram
      (print-histy (rest listy))))))
```

Because this simplification effectively hides the base case, a comment has been inserted to remind the reader that the base case is implicitly handled by `when` *returning* void.

16.3 Recursively Generating Lists as Output Values

So far, we have seen examples of recursive functions where the recursion is driven by a list, and the output has been a number, a boolean, or *void*—along with some side-effect printing. This section addresses list-based recursion where the output value is a list that has been incrementally generated by the recursive function calls. The incremental generation of lists is accomplished using the built-in `cons` function, introduced in Section 16.1.

Example 16.3.1: Doubling all the elements of a list

Suppose we want to define a function, called `double-all`*, that satisfies the following contract:*

```
;;  DOUBLE-ALL
;; ---------------------------------------------
;;  INPUT:  LISTY, a list of numbers
;;  OUTPUT: A list of numbers, each of whose elements
;;   is twice the corresponding element in LISTY.
```

Here are some examples of the desired behavior:

```
> (double-all '(3 2 10 13))
(6 4 20 26)
```

```
> (double-all '(5 3 8))
(10 6 16)
```

Let's apply some recursive thinking to the first example: `(double-all '(3 2 10 13))`. *We can generate the desired output list* `(6 4 20 26)` *as follows.*

(1) Consider the following pieces of the desired output list, `(6 4 20 26)`*:*

- *Its* first *element:* `6`
- *The* rest *of its elements:* `(4 20 26)`

(2) Fetch the corresponding pieces of the input list, `(3 2 10 13)`*:*

- *Its* first *element:* `3`
- *The* rest *of the list:* `(2 10 13)`

(3) Do the following to the corresponding pieces of the input list:

- *Double the* first *element:* `(* 2 3)` ⇒ `6`
- *Use a recursive function call to double the* rest *of the elements:*
  `(double-all '(2 10 13))` ⇒ `(4 20 26)`

(4) Use the above pieces to construct the desired output list using `cons`*:*

- `(cons 6 '(4 20 26))` ⇒ `(6 4 20 26)`

We can more concisely describe the process outlined above, as follows. If `listy` *is a non-empty list, the element-wise doubling of* `listy` *can be obtained by the following expression:*

```
(double-all listy) ⇒ (cons (* 2 (first listy))
                           (double-all (rest listy)))
```

Before jumping to the completed function definition, we need to determine what should happen in the base case, where the input list is empty. There are two things to consider:

- *The list obtained by doubling each element of the empty list is . . . the empty list:*
  `(double-all ())` ⇒ `()`.
- *When the input list is a one-element list, the recursive rule described above looks like this:*

```
(double-all '(4)) ⇒ (cons (* 2 4) (double-all ()))
                  ⇒ (cons 8 ())
                  ⇒ (8)
```

Therefore, whether we consider the base case in isolation—what should `double-all` *do to the empty list based on the contract?—or we consider the base case as the terminating case of a sequence of recursive function calls, we conclude that* `(double-all ())` *should evaluate to* `()`.

Here's the finished product:

```
(define double-all
  (lambda (listy)
    (cond
     ;; Base Case:  LISTY is empty
     ((null? listy)
      ;; The double-all of () is ...
      ())
     ;; Recursive Case:  LISTY is non-empty
```

```
        (else
         ;; Double the first element and attach it to the
         ;; double-all of the rest of the list
         (cons (* 2 (first listy))
               (double-all (rest listy)))))))
```

Example 16.3.2: Applying a given function to each element of a list

Recall the `facty` *function seen in Example 12.1.1. It takes a single number as its input, and returns the factorial of that number as its output:*

```
> (facty 3)
6
> (facty 5)
120
```

For this exercise, we want to define a function called `mappy` *that takes two inputs: (1) a function* `func` *that, like* `facty`*, can be applied to a single input, and (2) a list* `listy`*, each of whose elements is a suitable input for* `func`*. The expression* `(mappy func listy)` *should generate as its output the list whose elements are obtained by applying* `func`*, in turn, to each of the elements of* `listy`*. Here are some examples:*

```
> (mappy facty '(3 4 5 6))
(6 24 120 720)
> (mappy even? '(1 2 3 4 5 6))
(#f #t #f #t #f #t)
> (mappy abs '(1 -1 2 -2 3 -3))
(1 1 2 2 3 3)
```

The last expression uses the built-in `abs` *function, which computes the absolute value of its input.*

As in Example 16.3.1, we analyze this problem by thinking recursively, using a concrete example:

`(mappy facty '(3 4 5 6))` ⇒ `(6 24 120 720)`

(1) The parts of the desired output list:

- *Its first element:* `6`
- *The rest of its elements:* `(24 120 720)`

(2) The corresponding parts of the input list:

- *Its first element:* `3`
- *The rest of its elements:* `(4 5 6)`

(3) Do the following to the pieces of the input list:

- *Apply* `facty` *to the first element:* `(facty 3)` ⇒ `6`
- *Let a recursive function call apply facty to the rest of the elements:* `(mappy facty '(4 5 6))` ⇒ `(24 120 720)`

(4) Use the `cons` *function to combine the above pieces:*

- `(cons 6 '(24 120 720))` ⇒ `(6 24 120 720)`

The above analysis suggests that for a non-empty list `listy`*, the following expression will evaluate to the desired result:*

```
(mappy func listy) ⇒ (cons (func (first listy))
                           (mappy func (rest listy)))
```

In addition, you should convince yourself that, as in Example 16.3.1, the base case, `(mappy func ())`*, should evaluate to* `()`*. Here is the completed solution.*

```
;;  MAPPY
;; ------------------------------------------------------
;;  INPUTS:  FUNC, a function that takes a single input
;;           LISTY, a list of suitable inputs for FUNC
;;  OUTPUT:  A list whose elements are obtained by applying
;;   FUNC to each of the elements of LISTY, in turn.

(define mappy
  (lambda (func listy)
    (cond
     ;; Base Case:  LISTY is empty
     ((null? listy)
      ;; Applying FUNC to each element of the empty list
      ;; yields ... the empty list
      ())
     ;; Recursive Case:  LISTY is non-empty
     (else
      ;; Apply FUNC to the FIRST element of LISTY, and then
      ;; use CONS to attach the result to the front of the
      ;; list obtained from the recursive function call on
      ;; the REST of LISTY.
      (cons (func (first listy))
            (mappy func (rest listy)))))))
```

Incidentally, now that you know how to implement the `mappy` function, I can tell you that there is a built-in function, called `map`, that does the same thing. The following example illustrates how the `map` function can be used to facilitate testing.

Example 16.3.3: Using `map` to facilitate testing

Suppose that you have defined a function, called `square`*, that squares its input. Instead of writing several* `tester` *expressions to test the performance of* `square` *on several inputs, you can write just one* `tester` *expression, using* `map` *to apply* `square` *to several inputs:*

```
> (tester '(map square '(1 2 3 4 10 25)))
(map (square '(1 2 3 4 10 25))) ==> (1 4 9 16 100 625)
```

In-Class Problem 16.3.1: Removing items from a list

Define a function, called `remover`*, that satisfies the following contract:*

```
;;  REMOVER
;; -----------------------------------------------
;;  INPUTS:  ITEM, anything
;;           LISTY, a list
;;  OUTPUT:  A list that contains all of the elements of
;;           LISTY, except any occurrences of ITEM.
```

Here are some examples:

```
> (remover 3 '(1 2 3 4 5 4 3 2 1))
(1 2 4 5 4 2 1)
> (remover 'a '(a b r a c a d a b r a))
(b r c d b r)
```

Incidentally, now that you know how to implement the `remover` function, I can tell you that there is a built-in function, called `remove`, that does the same thing, except that it only removes the *first* occurrence of `item` from `listy`.

The following example implements a function that takes two input lists, but uses only one to drive the recursion.

In-Class Problem 16.3.2: Concatenating two lists

Define a function, called `conc`*, that satisfies the following contract:*

```
;;  CONC
;; ------------------------------------------------------------
;;  INPUTS:  LISTY, LISTZ, two lists
;;  OUTPUT:  A list containing all of the elements of LISTY
;;    followed by all of the elements of LISTZ.
```

Here are some examples of the desired behavior:

```
> (conc '(1 2 3 4) '(a b c))
(1 2 3 4 a b c)
> (conc '(a b c) '(1 2 3 4))
(a b c 1 2 3 4)
```

Hints: Let `listy` *drive the recursion. What is the output when* `listy` *is empty?*

Now that you know how to implement the `conc` function, I can tell you that there is a built-in function called `append` that does the same thing!

The preceding examples showed how a recursive function can be used to incrementally generate a new list as its output. In each case, some input list was driving the recursion. However, as the following examples show, functions whose recursion is driven by the size of a number can also be used to incrementally generate output lists.

Example 16.3.4

The goal of this example is to define a function, called `list-down-to-zero`*, that satisfies the following contract:*

```
;;  LIST-DOWN-TO-ZERO
```

```
;; ---------------------------------------------------
;;  INPUT:   N, a non-negative integer
;;  OUTPUT:  A list of the form (N N-1 N-2 ... 2 1 0)
```

Here are some examples of the desired behavior:

```
> (list-down-to-zero 5)
(5 4 3 2 1 0)
> (list-down-to-zero 8)
(8 7 6 5 4 3 2 1 0)
```

Thinking recursively about the first example, we note that the list from 5 down to 0 can be constructed by attaching the number 5 to the front of the list from 4 down to 0. More generally, for any non-negative number `n`*:*

`(list-down-to-zero n)` ⇒ `(cons n (list-down-to-zero (- n 1))`

where, for the base case, we stipulate that: `(list-down-to-zero m)` ⇒ `()`*, for any* $m < 0$*. (Alternatively, we could use* `(list-down-to-0 0)` ⇒ `(0)` *as our base case.) Here is the completed solution:*

```
(define list-down-to-zero
  (lambda (n)
    (cond
     ;; Base Case:  N < 0
     ((< n 0)
      ())
     ;; Recursive Case: N >= 0
     (else
      (cons n (list-down-to-zero (- n 1)))))))
```

In-Class Problem 16.3.3

Define a function, called `list-up-to-n`*, that satisfies the following contract:*

```
;;  LIST-UP-TO-N
;; ------------------------------------------------------------
;;  INPUTS:  FROM, a non-negative integer (starting point)
;;           N, a non-negative integer (stopping point)
;;  OUTPUT:  A list of the form (FROM FROM+1 FROM+2 ... N)
```

Here are some examples of the desired behavior:

```
> (list-up-to-n 4 12)
(4 5 6 7 8 9 10 11 12)
> (list-up-to-n 3 7)
(3 4 5 6 7)
```

Hint: Fill in the blanks: The list of integers from 4 to 12 can be constructed by attaching _____ to the front of the list of integers from _____ to _____. *More generally:* The list of integers from `from` to `n` can be constructed by attaching _____ to the front of the list of integers from _____ to _____.

In-Class Problem 16.3.4

Define a function, called `random-flips`*, that satisfies the following contract:*

```
;;  RANDOM-FLIPS
;; -------------------------------------------------------
;;  INPUTS:  N, a non-negative integer
;;  OUTPUT:  A list containing N random flips of a coin,
;;           where each flip is either H or T
```

Here are some examples of the desired behavior:

```
> (random-flips 8)
(H H T H T T T H)
> (random-flips 5)
(T H H T H)
```

Hint: Use the `flip-coin` *function from Example 15.2.2 as a helper. Fill in the blanks:* A list of `n` random coin flips can be generated by attaching _____ to the front of a list of _____ random coin flips.

16.4 Tail Recursion, Accumulators, and Wrapper Functions Revisited

Sections 14.2 through 14.4 introduced the concepts of *tail recursion, accumulators,* and *wrapper functions,* respectively. As will be seen in this section, these concepts apply equally well to list-based recursion and the incremental generation of lists as output values.

Recall from Defn. 14.2 that a recursive function-call expression is *tail recursive* if, whenever its evaluation is needed as part of evaluating the parent function's body, its evaluation is the *last* step in that process. And a recursive function is tail-recursive if each of its recursive function-call expressions is tail recursive.

Checking the functions implemented in Examples 16.2.1 through 16.3.4 reveals that `mult-all`, `double-all`, `mappy` and `list-down-to-zero` are *not* tail recursive, while `is-elt-of?`, `is-elt-of?-alt` and `print-histy` *are* tail recursive. The following examples define tail-recursive versions of `mult-all`, `list-down-to-zero` and `double-all`, respectively called `mult-all-acc`, `list-down-to-zero-acc` and `double-all-acc`. As the names indicate, each of these tail-recursive functions will take an additional input that serves to *accumulate* the desired answer. For `mult-all-acc`, the extra input will incrementally accumulate the product of the numbers in the input list, much as the accumulator in `facty-acc` (cf. Example 14.3.3) accumulated the factorial of its input. For `list-down-to-zero-acc` and `double-all-acc`, the extra input will incrementally accumulate a *list:* in particular, each tail-recursive function call will include a call to the `cons` function to attach a new element to the front of some list. As in Section 14.4, for each accumulator-based, tail-recursive function we shall define an accompanying wrapper function that takes care of providing appropriate initial values for any additional inputs.

Example 16.4.1: Tail-recursive function: `mult-all-acc`

Recall that the `mult-all` *function computes the product of all of the numbers in a given list. The* `mult-all-acc` *function will work similarly, except that it will take an extra input, called* `acc`*, that will accumulate the desired product. In particular, as we walk through the given list of numbers, as each number is encountered, it will be multiplied into the accumulator. As with* `facty-acc` *from Example 14.3.3, the initial value of* `acc` *will be* `1` *(i.e., the multiplicative identity).*

It can often help to consider a concrete example. Therefore, suppose that we want to use `mult-all-acc` *to compute the product of the numbers in the list* `(3 7 2 4)`*. We start with* `acc` *equal to* `1`*. Imagine the computation proceeding as follows, where the first input to* `mult-all-acc` *is*

the list of numbers, and the second input is the accumulator:

```
(mult-all-acc '(3 7 2 4) 1)
  ⇒ (mult-all-acc '(7 2 4) 3)     ⟵ rec. case: "accumulate" a factor of 3
  ⇒ (mult-all-acc '(2 4) 21)      ⟵ rec. case: "accumulate" a factor of 7
  ⇒ (mult-all-acc '(4) 42)        ⟵ rec. case: "accumulate" a factor of 2
  ⇒ (mult-all-acc () 168)         ⟵ rec. case: "accumulate" a factor of 4
  ⇒ 168                           ⟵ base case: accumulator has the answer!
```

Notice that the inputs for each recursive function call are:

- *the* rest *of the current list, and*
- *the product of the* first *element of the current list and the current accumulator.*

Thus, by the time the base case (i.e., the empty list) is reached, the accumulator has the desired product: $3 \cdot 7 \cdot 2 \cdot 4 = 168$. *Here is the completed solution:*

```
;;  MULT-ALL-ACC
;; ------------------------------------------------------------------
;;  INPUTS:  LISTY, a list of numbers
;;           ACC, a number (accumulator of desired product)
;;  OUTPUT:  When called with ACC=1, the output is the product
;;    of all of the numbers in LISTY.  More generally, the output
;;    is the product of ACC and all of the numbers of LISTY

(define mult-all-acc
  (lambda (listy acc)
    (cond
     ;; Base Case:  LISTY is empty
     ((null? listy)
      ;; The accumulated product
      acc)
     ;; Recursive Case:  LISTY is non-empty
     (else
      ;; Tail-recursive function call on adjusted inputs:
      ;;  Note:  ACC "accumulates" (first listy)
      (mult-all-acc (rest listy)
                    (* (first listy) acc))))))
```

As is often the case, describing the output for accumulator-based functions can be challenging in the general case (e.g., above, when `ACC` *is something other than* `1`*). Here is the accompanying wrapper function:*

```
;;  MULT-ALL-WR
;; ---------------------------------------
;;  INPUT:   LISTY, a list of numbers
;;  OUTPUT:  The product of the numbers in LISTY

(define mult-all-wr
  (lambda (listy)
    ;; Call the tail-recursive helper with ACC=1:
    (mult-all-acc listy 1)))
```

Notice that the contract for `mult-all-wr` *is the same as that for* `mult-all`*—except for the name of the function. That is, the two functions are equivalent.*

Example 16.4.2: Tail-recursive function: `list-down-to-zero-acc`

Recall that the `list-down-to-zero` *function takes a non-negative integer* n *as its only input, and generates as its output a list of the form* `(`n $n-1$ $n-2$ `...` `2 1 0)`*. The* `list-down-to-zero-acc` *function will work similarly, except that it will incrementally accumulate the desired list in an extra input,* `acc`*. As in the* `double-all` *and* `mappy` *functions (cf. Examples 16.3.1 and 16.3.2, respectively) the list-accumulator will start out as the empty list.*

Consider the example where the numerical input n *is* `3`*, and we want to generate the list* `(3 2 1 0)`*. As in* `list-down-to-zero`*, the value of* `n` *will decrease by one on each recursive function call, but the accumulator will be adjusted by using the* `cons` *function to attach* `n` *to the front of the accumulator, as illustrated in the following sequence of evaluations:*

```
(list-down-to-zero-acc 3 ())
  ⇒ (list-down-to-zero-acc 2 '(3))          ⟵ attach 3 to front of acc
  ⇒ (list-down-to-zero-acc 1 '(2 3))        ⟵ attach 2 to front of acc
  ⇒ (list-down-to-zero-acc 0 '(1 2 3))      ⟵ attach 1 to front of acc
  ⇒ (list-down-to-zero-acc -1 '(0 1 2 3))   ⟵ attach 0 to front of acc
  ⇒ (0 1 2 3)                               ⟵ acc has the answer??!
```

Whoops! *While this would be fine for generating a list from* `0` *to* `n`*, that is not what we were aiming for! This example illustrates a common issue that arises when using list accumulators:*

- ⋆ *When using an accumulator to incrementally generate a list, the order of the elements in the accumulator ends up being the reverse of the order in which they were attached!*

There are two ways to fix this problem: (1) define a function to reverse the elements of a list; or (2) arrange to process the desired elements in the opposite order. Below, we take the second approach. Later on, we'll define a function for reversing the elements of a list.

For the `list-down-to-zero-acc` *function, we can arrange to visit the numbers in the order from* `0` *up to* `n` *by including yet another input, called* `curr` *(for* current *number), whose value shall start out at* `0` *and increment by one on each recursive function call. Since* `0` *will be the first number to be attached to the accumulator, it will end up being the last number in the generated list, as desired. So the inputs to* `list-down-to-zero-acc` *will be* `n`*,* `acc` *and* `curr`*. In this version, the value of* `n` *will be the same for each recursive function call. That is,* `n` *serves as an upper bound on the value of* `curr`*. When that upper bound is reached, the recursion will terminate, as illustrated below:*

```
(list-down-to-zero-acc 3 () 0)
  ⇒ (list-down-to-zero-acc 3 '(0) 1)          ⟵ attach 0 to front of acc
  ⇒ (list-down-to-zero-acc 3 '(1 0) 2)        ⟵ attach 1 to front of acc
  ⇒ (list-down-to-zero-acc 3 '(2 1 0) 3)      ⟵ attach 2 to front of acc
  ⇒ (list-down-to-zero-acc 3 '(3 2 1 0) 4)    ⟵ attach 3 to front of acc
  ⇒ (3 2 1 0)                                 ⟵ acc has the answer!
```

Notice that in this version of `list-down-to-zero-acc`*, the base case is signaled by* `curr` *being greater than* `n`*—in this example, when* $4 > 3$*. Here is the completed solution:*

```
;;  LIST-DOWN-TO-ZERO-ACC
;; -------------------------------------------------------
;;  INPUTS:  N, a non-negative integer
;;           ACC, a list accumulator
;;           CURR, a non-negative integer
;;  OUTPUT:  When called with ACC=() and CURR=0, the output
;;    is the list (N N-1 N-2 ... 2 1 0).  More generally,
;;    the output is the "concatenation" of the lists
;;    (N N-1 N-2 ... CURR) and ACC.

(define list-down-to-zero-acc
  (lambda (n acc curr)
    (cond
     ;; Base Case:  CURR > N
     ((> curr n)
      ;; The accumulator has the desired list
      acc)
     ;; Recursive Case:  CURR <= N
     (else
      ;; Tail-recursive function call with adjusted inputs:
      (list-down-to-zero-acc n (cons curr acc) (+ curr 1))))))
```

(You should convince yourself that the "more generally" part of the contract is correct.) Here is the associated wrapper function:

```
;;  LIST-DOWN-TO-ZERO-WR
;; -------------------------------------------------
;;  INPUT:   N, a non-negative integer
;;  OUTPUT:  The list (N N-1 N-2 ... 2 1 0)

(define list-down-to-zero-wr
  (lambda (n)
    ;; Call the tail-recursive helper with ACC=() and CURR=0:
    (list-down-to-zero-acc n () 0)))
```

Before introducing the `double-all-acc` function, which also uses a list accumulator and, so, suffers from the same problem seen earlier regarding the order of accumulated elements, we first introduce the `transfer-all` and `reversey` functions. The latter function can be used to reverse the elements in a list.

Example 16.4.3: The `transfer-all` and `reversey` functions

The goal for this example is to define a function, called `transfer-all`*, that satisfies the following contract:*

```
;;  TRANSFER-ALL
;; -----------------------------------------------------------
;;  INPUTS:  LISTY, LISTZ, two lists
;;  OUTPUT:  The list obtained by "popping" each element in
;;    turn off of the front of LISTY and "pushing" it onto
;;    the front of LISTZ.
```

Here are some examples of the desired behavior:

```
> (transfer-all '(a b c) '(1 2))
(c b a 1 2)
> (transfer-all '(1 2) '(a b c))
(2 1 a b c)
```

Notice that the elements from the first list appear in the reverse order in the output list. Here is a sample sequence of evaluations corresponding to the first example above:

```
(transfer-all '(a b c) '(1 2))
  ⇒ (transfer-all '(b c) '(a 1 2))    ⟵ attach a to front of second list
  ⇒ (transfer-all '(c) '(b a 1 2))    ⟵ attach b to front of second list
  ⇒ (transfer-all () '(c b a 1 2))    ⟵ attach c to front of second list
  ⇒ (c b a 1 2)                       ⟵ base case!
```

As the above example illustrates, the first list (i.e., `listy`*) is driving the recursion, and the second list (i.e.,* `listz`*) is acting like an accumulator. When* `listy` *is empty, the accumulator* `listz` *contains the desired answer. Here is the completed function definition:*

```
(define transfer-all
  (lambda (listy listz)
    (cond
     ;; Base Case:  LISTY is empty
     ((null? listy)
      ;; return the "accumulator"
      listz)
     ;; Recursive Case:  LISTY is non-empty
     (else
      ;; Tail-recursive function call with adjusted inputs
      (transfer-all (rest listy)
                    (cons (first listy) listz))))))
```

Next, we define a "wrapper" for `transfer-all` *which we shall call* `reversey`*, for reasons that will soon become apparent.*

```
;;  REVERSEY -- wrapper for TRANSFER-ALL
;; ------------------------------------------------
;;  INPUT:   LISTY, a list
;;  OUTPUT:  A list that contains the same elements as
;;           LISTY, but in the opposite order.

(define reversey
  (lambda (listy)
    ;; Call TRANSFER-ALL with LISTZ=():
    (transfer-all listy ())))
```

Here are some examples that illustrate that `reversey` *does indeed generate the reversal of its input:*

```
> (reversey '(a b c d))
(d c b a)
> (reversey '(1 2 3 4 5 6))
(6 5 4 3 2 1)
```

Incidentally, now that you know how to implement the `reversey` function, I can tell you that there is a

built-in function called `reverse` that does the same thing!

Example 16.4.4: Not all ways of reversing a list are equal!

This example considers an alternative approach to reversing a list, one based on repeated concatenation. Although this approach leads to a function that correctly reverses a list, it turns out to be very inefficient. First, since it is not tail recursive, it can use an awful lot of the computer's memory when reversing long lists. Second, by repeatedly concatenating long lists, it takes a lot longer to reverse a list than the `reversey` *function seen earlier. To illustrate the inefficiency of this approach, both functions,* `konk` *and* `bad-reverse`*, defined below, print out some information each time they are called. The* `konk` *function concatenates two lists;* `bad-reverse` *uses* `konk` *as a helper function.*

```
;;  KONK
;; ------------------------------------------------------------
;;  INPUTS:  LISTY, LISTZ, two lists
;;  OUTPUT:  A list containing all of the elements of LISTY,
;;           followed by all of the elements of LISTZ.

(define konk
  (lambda (listy listz)
    (printf "KONK:  LISTY: ~A, LISTZ: ~A~%" listy listz)
    (cond
      ;; Base Case:  LISTY is empty
      ((null? listy)
       ;; The concatenation of () and LISTZ is:
       listz)
      ;; Recursive Case:  LISTY is non-empty
      (else
       ;; Attach (FIRST LISTY) onto the concatenation
       ;; of (REST LISTY) and LISTZ
       (cons (first listy)
             (konk (rest listy) listz))))))

;;  BAD-REVERSE
;; -------------------------------------
;;  INPUT:   LISTY, any list
;;  OUTPUT:  A list containing the same elements as
;;           LISTY, but in the opposite order.

(define bad-reverse
  (lambda (listy)
    (printf "BAD-REVERSE:  LISTY: ~A~%" listy)
    (cond
      ;; Base Case:  LISTY is empty
      ((null? listy)
       ())
      ;; Recursive Case:  LISTY is non-empty
      (else
       ;; Recursive function call reverses the REST of LISTY.
       ;; So, we need to attach (first listy) at the end.
       ;; Unfortunately this involves walking through the
       ;; potentially long list returned by the recursive
       ;; function call.
```

```
          (konk (bad-reverse (rest listy))
                (cons (first listy) ()))))))
```

To get an idea of how inefficient `bad-reverse` *is, try evaluating the following expression in the Interactions Window:* `(bad-reverse '(a b c d e))`.

Example 16.4.5: The `double-all-acc` function

The goal of this example is to define a tail-recursive function that doubles all of the elements of a given list of numbers. Because we shall use a list accumulator, the doubled numbers in the accumulated list will come out in the wrong order. But we shall just use the built-in `reverse` *function to reverse the order of the accumulated list before returning it as the output. Here is the completed function definition:*

```
;;  DOUBLE-ALL-ACC
;; ------------------------------------------------
;;  INPUTS:  LISTY, a list of numbers
;;           ACC, a list accumulator
;;  OUTPUT:  When called with ACC=(), the output is
;;             a list just like LISTY, except that each
;;             element has been doubled.

(define double-all-acc
  (lambda (listy acc)
    (cond
     ;; Base Case:  LISTY is empty
     ((null? listy)
      ;; REVERSE the accumulator!
      (reverse acc))
     ;; Recursive Case:  LISTY is non-empty
     (else
      ;; Tail-recursive function call with adjusted inputs
      (double-all-acc (rest listy)
                      ;; "Accumulate" the first element doubled
                      (cons (* 2 (first listy))
                            acc))))))
```

As this example illustrates, the previously identified issue with list accumulators (i.e., that the accumulated elements come out in the opposite order) is easily resolved using the `reverse` *function at the very last instant!*

16.5 Sorting Algorithms

This section introduces two algorithms for sorting a list of numbers: the *insertion-sort* algorithm, and the *merge-sort* algorithm. After defining Scheme functions that implement these algorithms, they are compared by running them on long lists of randomly generated numbers. In what follows, we shall assume that the goal is to sort lists of numbers into *non-decreasing* order, as illustrated below:

Before sorting: `(3 2 1 4 3 2 3 3 6 1 0 5)`

After sorting: `(0 1 1 2 2 3 3 3 3 4 5 6)`

Notice that for any consecutive elements, x and y, in the sorted list, the following holds: $x \leq y$.

16.5.1 The Insertion-Sort Algorithm

The insertion-sort algorithm uses a helper function, called `insert`, that inserts a number into an *already-sorted* list, such that the resulting list is still sorted. Here is its contract, followed by some examples of the desired behavior:

```
;;  INSERT
;; -------------------------------------------------------------
;;  INPUTS:  NUM, a number
;;           SORTED, a list of numbers that are already sorted
;;             into non-decreasing order
;;  OUTPUT:  The list obtained by inserting NUM into SORTED while
;;           preserving the non-decreasing ordering

> (insert 3 '(5 8 9 10 11))          ⟵ 3 goes at the front of the sorted list
(3 5 8 9 10 11)
> (insert 3 '(0 1 1 2))              ⟵ 3 goes at the end of the sorted list
(0 1 1 2 3)
> (insert 3 '(1 2 4 5 6))            ⟵ 3 goes somewhere in the middle
(1 2 3 4 5 6)
> (insert 3 '(1 2 2 3 4 4 4 9 12))   ⟵ Same as above, except that there's another 3
(1 2 2 3 3 4 4 4 9 12)
```

Intuitively, the `insert` function walks through the already-sorted list until it finds the proper place for the given number. (What distinguishes the "proper place" for the given number?) We'll have more to say about how the `insert` function might do this—in fact, we'll define the `insert` function from scratch—but, for now, we'll just take the `insert` function as given.

As indicated earlier, the *insertion-sort* algorithm takes a (usually unsorted) list of numbers as its only input. Its goal is to generate as its output a list containing the same elements, but sorted into non-decreasing order. Here is its contract:

```
;;  INSERTION-SORT
;; -------------------------------------------------------
;;  INPUTS:  LISTY, a list of numbers
;;  OUTPUT:  A list containing the same elements as LISTY,
;;           but sorted into non-decreasing order
```

It can be implemented using list-based recursion, as follows. First, as a base case, consider that the empty list is already sorted.[1] Next, for the recursive case (i.e., when its input is a non-empty list), the insertion-sort algorithm applies the following recursive rule:

```
(insertion-sort listy) ⇒ (insert (first listy)
                                 (insertion-sort (rest listy)))
```

According to its contract, the recursive call on the *rest* of `listy` should generate a sorted list containing all of the elements of `(rest listy)`.[2] Therefore, to generate the desired output (i.e., a sorted list that contains all of the elements of `listy`), it only remains to find out where `(first listy)` should be inserted into that sorted *rest* of `listy`. And that is precisely what the call to the `insert` helper function does. Here is the completed definition of the `insertion-sort` function:

[1] A one-element list is also already sorted, but we stick with the empty list as the base case to simplify the code slightly.

[2] In general, when defining recursive functions, we assume that the recursive function call will generate the right answer. After all, it will be evaluated using the same function that we are currently defining! This sort of assumption—which, at first, may seem crazy—is justified by *mathematical induction.*

```
(define insertion-sort
  (lambda (listy)
    (cond
      ;; Base Case:  LISTY is empty
      ((null? listy)
       ;; The empty list is already sorted
       ())
      ;; Recursive Case:  LISTY is non-empty
      (else
       (insert (first listy)
               (insertion-sort (rest listy)))))))
```

Example 16.5.1: Applying `insertion-sort` to a sample list

Suppose that `listy` *is the list* `(3 2 5 1 6)`*. Then the recursive function call on the rest of* `listy` *would be, in effect,*

```
(insertion-sort '(2 5 1 6))
```

Assuming that the recursive function call does the right thing, it should generate as its output the sorted *list* `(1 2 5 6)`*. Therefore, in this case, the above-mentioned recursive rule would, in effect, lead to the following sequence:*

```
(insertion-sort '(3 2 5 1 6))
  ⇒ (insert 3 (insertion-sort '(2 5 1 6)))
  ⇒ (insert 3 '(1 2 5 6))
  ⇒ '(1 2 3 5 6)
```

And if we were to consider the details of each recursive function call, we would, in effect, end up with the following sequence of evaluations, using the abbreviations, `i` *for* `insert`*, and* `isort` *for* `insertion-sort`*:*

```
(isort '(3 2 5 1 6))                          Recursive case
  ⇒ (i 3 (isort '(2 5 1 6)))                  Recursve case
  ⇒ (i 3 (i 2 (isort '(5 1 6))))              Recursive case
  ⇒ (i 3 (i 2 (i 5 (isort '(1 6)))))          Recursive case
  ⇒ (i 3 (i 2 (i 5 (i 1 (isort '(6))))))      Recursive case
  ⇒ (i 3 (i 2 (i 5 (i 1 (i 6 (isort ()))))))  Base case!
  ⇒ (i 3 (i 2 (i 5 (i 1 (i 6 ())))))          Insert 6 into ()
  ⇒ (i 3 (i 2 (i 5 (i 1 '(6)))))              Insert 1 into (6)
  ⇒ (i 3 (i 2 (i 5 '(1 6))))                  Insert 5 into (1 6)
  ⇒ (i 3 (i 2 '(1 5 6)))                      Insert 2 into (1 5 6)
  ⇒ (i 3 '(1 2 5 6))                          Insert 3 into (1 2 5 6)
  ⇒ '(1 2 3 5 6)                              Done!
```

In-Class Problem 16.5.1: The `insert` helper function

Define the `insert` *function to satisfy the contract given earlier.*

Hints: Use recursion to walk through `sorted` *until you find the proper place for* `num`*. How will you recognize the proper place for* `num`*? Consider* `(first listy)` *and* `num`*. Finally, what should you do if* `sorted` *is empty?*

In-Class Problem 16.5.2: Generating long lists of random numbers

Define a function, called `list-of-n-random-numbers`*, that satisfies the following contract:*

```
;;  LIST-OF-N-RANDOM-NUMBERS
;; ------------------------------------------
;;  INPUT:   N, a positive integer
;;  OUTPUT:  A list containing N numbers, each randomly generated
;;             from the set {0, 1, 2, ..., 99999}
```

Here are some examples of the desired behavior:

```
> (list-of-n-random-numbers 10)
(18980 44224 94176 57470 23568 47609 70753 77870 98756 11729)
> (list-of-n-random-numbers 5)
(68856 3578 85898 27820 87029)
```

Hint: In the recursive case, use the built-in `random` *function with an appropriate input.*

This function can be used to randomly generate lists of numbers for `insertion-sort` *to sort, as illustrated below:*

```
> (let* ((list-o-randies (list-of-n-random-numbers 5))
         (sorted (insertion-sort list-o-randies)))
    (printf "BEFORE: ~A~%" list-o-randies)
    (printf "AFTER:  ~A~%" sorted)
    sorted)
BEFORE: (68502 79284 50452 31764 48239)
AFTER:  (31764 48239 50452 68502 79284)
(31764 48239 50452 68502 79284)
> (let* ((list-o-randies (list-of-n-random-numbers 5))
         (sorted (insertion-sort list-o-randies)))
    (printf "BEFORE: ~A~%" list-o-randies)
    (printf "AFTER:  ~A~%" sorted)
    sorted)
BEFORE: (51897 96352 87874 82047 17760)
AFTER:  (17760 51897 82047 87874 96352)
(17760 51897 82047 87874 96352)
```

Of course, it will be more interesting to see how long it takes `insertion-sort` *to sort really* long *lists of numbers (e.g., lists having thousands of elements). In such cases, you wouldn't want to print out the before and after lists!*

To avoid excessive memory usage, it is better to implement accumulator-based tail-recursive versions of the `insert` and `insertion-sort` functions.

In-Class Problem 16.5.3: Accumulator-based tail-recursive version of the `insert` **function**

For this problem, the goal is to define an accumulator-based tail-recursive version of the `insert` *function, called* `insert-acc`*. Recall that the* `insert` *function aims to insert a given number* `num` *into its proper place in an already-sorted list,* `sorted`*. The main idea behind the accumulator-based tail-recursive*

approach is to walk through `sorted`*, accumulating all of its elements that are smaller than* `num`*, as illustrated below:*

(insert-acc	num	sorted	acc)
(insert-acc	5	'(1 2 4 6 12 15)	())
(insert-acc	5	'(2 4 6 12 15)	'(1))
(insert-acc	5	'(4 6 12 15)	'(2 1))
(insert-acc	5	'(6 12 15)	'(4 2 1))

Notice that when all of the numbers smaller than `num` *have been accumulated, the proper place for* `num` *has been found (i.e., the base case has been reached). The only thing that remains is to assemble the pieces into the final sorted list. In the above example, the desired list is* `(1 2 4 5 6 12 15)`*, which can be built as follows:*

(1) Use `cons` *to attach* `num` *to the front of* `sorted`*, yielding* `(5 6 12 15)`*.*

(2) Use `transfer-all` *(from Example 16.4.3) to transfer all of the elements of* `acc` *onto the result of Step 1, yielding* `(1 2 4 5 6 12 15)`*.*

Using the approach outlined above, define the `insert-acc` *to satisfy the following contract:*

```
;;   INSERT-ACC
;; ------------------------------------------------------------
;;  INPUT:  NUM, a number
;;          SORTED, a list of numbers that are already sorted
;;             into non-decreasing order
;;          ACC, a list of numbers in non-increasing order,
;;             where each number in ACC is less than NUM
;;  OUTPUT:  When called with ACC = (), the output is a list
;;            containing NUM and all the numbers in SORTED,
;;            all sorted into non-decreasing order.
```

Here are some examples of its use:

```
> (insert-acc 5 '(1 2 4 6 12 15) ())
(1 2 4 5 6 12 15)
> (insert-acc 3 '(1 1 2 2 3 3 4 4 5 5) ())
(1 1 2 2 3 3 3 4 4 5 5)
```

Finally, define a wrapper function, called `insert-wr`*, that satisfies the following contract, and exhibits the behavior shown below:*

```
;;  INSERT-WR  --  wrapper function for INSERT-ACC
;; ------------------------------------------------------------
;;  INPUT:  NUM, a number
;;          SORTED, a list of numbers that are already sorted
;;             into non-decreasing order
;;  OUTPUT: A list containing NUM and all the numbers in SORTED,
;;            all sorted into non-decreasing order.

> (insert-wr 5 '(1 2 4 6 12 15))
(1 2 4 5 6 12 15)
> (insert-wr 3 '(1 1 2 2 3 3 4 4 5 5))
(1 1 2 2 3 3 3 4 4 5 5)
```

In-Class Problem 16.5.4: Tail-recursive version of `insertion-sort`

For this problem, we seek a tail-recursive version of the insertion-sort *algorithm. For convenience, we call it* `isort-acc`*. The following sequence of recursive function calls illustrates the approach, which uses an extra accumulator argument to accumulate the sorted list. At each step the first element of the unsorted list is inserted into its proper place in the sorted list:*

```
⇒   (isort-acc '(4 9 2 6) ())                ⟵ recursive case
⇒   (isort-acc '(9 2 6) (insert-wr 4 ())
⇒   (isort-acc '(9 2 6) '(4))                ⟵ recursive case
⇒   (isort-acc '(2 6) (insert-wr 9 '(4))
⇒   (isort-acc '(2 6) '(4 9))                ⟵ recursive case
⇒   (isort-acc '(6) (insert-wr 2 '(4 9))
⇒   (isort-acc '(6) '(2 4 9))                ⟵ recursive case
⇒   (isort-acc () (insert-wr 6 '(2 4 9))
⇒   (isort-acc () '(2 4 6 9))                ⟵ base case
⇒   (2 4 6 9)
```

Once your `isort-acc` *function is working properly, define a wrapper function called* `isort-wr` *that calls* `isort-acc` *with an appropriate value for the accumulator.*

16.5.2 The Merge-Sort Algorithm

The *merge-sort* algorithm, like the *insertion-sort* algorithm, takes a (typically unsorted) list of numbers as its input, and generates a sorted version of that list as its output. Here is its contract:

```
;;  MERGE-SORT
;; ------------------------------------------------------
;;  INPUTS:  LISTY, a list of numbers
;;  OUTPUT:  A list containing the same elements as LISTY,
;;           but sorted into non-decreasing order
```

However, the merge-sort algorithm takes a very different approach to sorting lists, as follows. First, its base case handles the case where `listy` is a *one-element list* which, of course, must already be sorted. Second, when `listy` is non-empty, it uses recursion, as follows:

(1) *Split* `listy` into two lists, `lefty` and `righty`, of roughly the same size;

(2) Use the `merge-sort` function to sort `lefty`, yielding a sorted list, `sorted-lefty`; and use `merge-sort` to sort `righty`, yielding a sorted list, `sorted-righty`; and then

(3) *Merge* the two sorted lists, `sorted-lefty` and `sorted-righty`, into a single sorted list, which will be the desired output.

As indicated above, the `merge-sort` function uses two helper functions: `split` and `merge`. These helpers will be defined shortly. For now, we will assume that they are available, and that they satisfy the following contracts:

```
;;  SPLIT
;; ------------------------------------------------------------
;;  INPUT:   LISTY, any list
;;  OUTPUT:  A list of the form (LEFTY RIGHTY) where LEFTY
;;    and RIGHTY are two subsidiary lists such that the
;;    elements of LISTY have been allocated as evenly as
;;    possible to LEFTY and RIGHTY, but with no regard to
;;    their order.
```

```
;;  MERGE
;; ------------------------------------------------------------
;;  INPUT:   SORTED-ONE, SORTED-TWO, two lists of numbers
;;             that are already sorted into non-decreasing order.
;;  OUTPUT:  A single list that contains all of the elements
;;             of SORTED-ONE and SORTED-TWO, sorted into
;;             non-decreasing order.
```

Example 16.5.2: The `split` and `merge` helper functions

Here are some examples of the behavior of the `split` *and* `merge` *helper functions:*

```
> (split '(5 3 1 2 8 4 9 4))          ⟵ Input has an even number of elements
((4 4 2 3) (9 8 1 5))
> (split '(5 3 1 2 7))                ⟵ Input has an odd number of elements
((7 1 5) (2 3))
> (merge '(1 3 5 7) '(2 4 6 8))
(1 2 3 4 5 6 7 8)
> (merge '(1 1 2 3 3 3 5 9) '(2 3 3 4 8 8 9)
> (1 1 2 2 3 3 3 3 3 4 5 8 8 9 9)
```

In the case of the `split` *function, notice that the order of the elements in the input list and the two subsidiary lists in the output do not matter at all. The reason is that* `split` *will typically be applied to unsorted lists—so the order of the elements doesn't matter. Also, if the input list has an even number of elements, then the two lists in the output will have the same number of elements; otherwise, one of the output lists will have the odd element. For the* `merge` *function, the two input lists must already be sorted, but they may have duplicate elements, and the two input lists need not have the same number of elements.*

Example 16.5.3: Applying `merge-sort` to a sample list

Here, we consider the application of the `merge-sort` *function to the input list* `(8 2 5 9 3 4 6 1)`. *As described previously, there are three steps to the recursive case:*

(1) Split `listy` *into two lists,* `lefty` *and* `righty`*, of roughly the same size. Here:*

```
lefty  = (6 3 5 8)
righty = (1 4 9 2)
```

(2) Use the `merge-sort` *function to sort* `lefty`*, yielding a sorted list,* `sorted-lefty`*; and use* `merge-sort` *to sort* `righty`*, yielding a sorted list,* `sorted-righty`*. Here:*

```
sorted-lefty  = (3 5 6 8)
sorted-righty = (1 2 4 9)
```

(3) Merge *the two sorted lists,* `sorted-lefty` *and* `sorted-righty`*, into a single sorted list, which will be the desired output. Here:*

`(merge '(3 5 6 8) '(1 2 4 9))` ⇒ `(1 2 3 4 5 6 8 9)`.

Here is the completed definition of the `merge-sort` function:

```
(define merge-sort
  (lambda (listy)
```

```
(cond
 ;; Base Case:  LISTY has exactly one element
 ((null? (rest listy))
  ;; A one-element list is already sorted
  listy)
 ;; Recursive Case:  LISTY has at least two elements
 (else
  (let* (;; LIST-O-LISTS has the form (LEFTY RIGHTY)
         (list-o-lists (split listy))
         ;; Access the two subsidiary lists in LIST-O-LISTS
         (lefty (first list-o-lists))
         (righty (second list-o-lists))
         ;; Recursively sort LEFTY and RIGHTY
         (sorted-lefty (merge-sort lefty))
         (sorted-righty (merge-sort righty)))
    ;; Body of the LET*:  MERGE the two sorted lists
    (merge sorted-lefty sorted-righty))))))
```

Notice that most of the work is done in the variable-declaration part of the `let*` special form. The body of the `let*` just applies the `merge` function to the two sorted lists.

Now it is time to define the `split` and `merge` helper functions needed by `merge-sort`.

In-Class Problem 16.5.5: The `split` helper function

Define the `split` *helper function to satisfy the contract seen earlier. Here are some hints:*

(1) Define an accumulator-based helper function, called `split-acc`*, that includes two extra inputs,* `lefty` *and* `righty`*. These will serve as accumulators for the two subsidiary lists.*

(2) In the base case, use the `list-two` *function defined in In-Class Problem 16.1.2 to create the desired list of lists. (Alternatively, use the built-in* `list` *function; or use a couple of calls to the* `cons` *function.)*

(3) Define `split` *as a wrapper function that simply calls* `split-acc` *with appropriate initial values for its accumulator inputs.*

In-Class Problem 16.5.6: The `merge` helper function

Define the `merge` *helper function to satisfy the contract seen earlier. Here are some hints:*

(1) When either list is empty, the answer is easy.

(2) When both lists are non-empty, compare their first elements to see which one comes first.

Define two versions of the `merge` *function: one that is not tail recursive (and perhaps easier to define), and one that is just a wrapper for a tail-recursive helper function called* `merge-acc`*. The contract for* `merge-acc` *is given below.*

```
;;  MERGE-ACC
;; ------------------------------------------------------------
;;  INPUTS:  SORTED-LEFTY, SORTED-RIGHTY, two lists of
;;             numbers, each sorted into non-decreasing order
;;           ACC, a list-accumulator
;;  OUTPUT:  When called with ACC=(), the output is a
```

```
;;              single list containing all of the elements
;;              of SORTED-LEFTY and SORTED-RIGHTY, sorted
;;              into non-decreasing order.
```

Hint: When both `sorted-lefty` *and* `sorted-righty` *are non-empty, "accumulate" the smaller of* `(first sorted-lefty)` *and* `(first sorted-righty)`*. In the base case, use the built-in* `reverse` *function to reverse the accumulated list.*

16.5.3 Comparing the Performance of Insertion Sort and Merge Sort

This section shows how we can write Scheme functions to automate a rigorous comparison of the *insertion-sort* and *merge-sort* algorithms. Some considerations include:

- We want to test these algorithms on really long lists of randomly generated numbers.
- For each randomly generated list, we want to test both algorithms on the *same* list.
- We'd like to know how long it takes each algorithm to sort the lists.

We already have the `list-of-n-random-numbers` function, from In-Class Problem 16.5.2. And since the two sorting algorithms are non-destructive, we can simply store the randomly generated list of numbers in a local variable, and then apply each sorting algorithm to the same list. As for timing their performance, Scheme provides a special form, called `time`, described below.

The `time` special form. The purpose of the `time` special form is to report how long it takes to evaluate a given expression. The syntax and semantics of the `time` special form are simple.

(Syntax) Any expression of the form `(time` *expr*`)` is a legal instance of the `time` special form.

(Semantics – Output Value) Any expression of the form `(time` *expr*`)` evaluates to whatever *expr* evaluates to.

(Semantics – Side Effect) The evaluation of an expression of the form `(time` *expr*`)` causes three pieces of timing information to be displayed in the Interactions Window:

`cpu time`	how many milliseconds DrScheme spent evaluating *expr*. (CPU is an acronym for the computer's *central processing unit.*)
`real time`	how many milliseconds elapsed while *expr* was evaluated.
`gc time`	how many milliseconds were spent in a memory-management process called *garbage collection.* (Garbage collection is an extremely interesting and important concept in the management of a computer's memory, but a discussion of it is beyond the scope of this book.)

The *cpu time* is typically a bit less than the *real time* because a computer's CPU typically does more than one thing during any given time interval; thus, the time the CPU devotes to DrScheme's evaluation of *expr* will typically be less than the elapsed time. For our purposes, the *cpu time* is the most relevant, because it most accurately reflects how much time DrScheme spent evaluating the given expression.

Example 16.5.4: Using the `time` special form

Here are some examples of the `time` *special form in action:*[a]

```
> (time (list-of-n-random-numbers 10000))
cpu time: 4 real time: 5 gc time: 0
(19207 53390 65067 65764 68321 75622 81451 38038 86109 ...)
```

```
> (time (insertion-sort (list-of-n-random-numbers 10000)))
cpu time: 7643 real time: 7849 gc time: 62
(10 12 26 30 50 65 70 77 80 83 94 104 108 113 114 150 ...)
> (let ((listy (list-of-n-random-numbers 10000)))
    (time (insertion-sort listy)))
cpu time: 7519 real time: 7674 gc time: 61
(2 9 14 16 26 31 32 37 38 40 84 85 113 114 115 119 171 ...)
```

The first example shows that it doesn't take DrScheme long to generate a list of 10,000 random numbers. The second example shows how long it takes to generate and sort *a list of numbers, using the* `insertion-sort` *function. The last example is the most important: it shows how long the sorting process takes; it ignores the time needed to generate the original list of random numbers.*

[a] To increase readability, the output lists have been cut off.

Example 16.5.5: Comparing the performance of the sorting algorithms

The following function can be used to compare the performance of the insertion-sort and merge-sort algorithms.

```
;;  COMPARE-SORTING-ALGS
;; ------------------------------------------------
;;  INPUT:   N, a positive integer
;;  OUTPUT:  None
;;  SIDE EFFECT:  Reports how long it took for the
;;    insertion-sort and merge-sort algorithms to sort
;;    the *same* randomly generated list of N numbers.

(define compare-sorting-algs
  (lambda (n)
    (let (;; Generate a list of n randomly generated numbers
          (listy (list-of-n-random-numbers n)))
      (printf "Running insertion-sort ...~%   ")
      (time (insertion-sort listy))
      (printf "~%Running merge-sort ...~%   ")
      (time (merge-sort listy))
      ;; Return VOID (so we don't see a long list of numbers)
      (void))))
```

Here is an example:

```
> (compare-sorting-algs 1000)
Running insertion-sort ...
   cpu time: 87 real time: 93 gc time: 0

Running merge-sort ...
   cpu time: 6 real time: 6 gc time: 0
```

In-Class Problem 16.5.7: A thorough comparison of `merge-sort` and `insertion-sort`

Use the `compare-sorting-algs` *function to compare the performance of the two sorting algorithms on lists of the following lengths: 1000, 2000, 4000, 8000, 16000, etc. Which algorithm would you recommend? Try running the faster of the two algorithms on* really *long lists (e.g., with 100,000 elements, or even a million elements).*

Example 16.5.6: The built-in `sort` function

Scheme provides a built-in function, called `sort`*, whose contract is given below, followed by some examples of its use.*

```
;;  SORT  --  built-in function
;; ------------------------------------------------------------
;;  INPUTS:  LISTY, a list of stuff
;;           COMPARER, a predicate that can be applied to
;;             any pair of elements in LISTY
;;  OUTPUT:  A list containing the same elements as LISTY,
;;             but sorted such that for any elements AAA and BBB
;;             in LISTY, if (COMPARER AAA BBB) ==> #t, then AAA
;;             comes before BBB in the output list.

> (sort '(5 2 1 3 3 2 5) <)       ⟵ sort into non-decreasing order
(1 2 2 3 3 5 5)
> (sort '(5 2 1 3 3 2 5) >)       ⟵ sort into non-increasing order
(5 5 3 3 2 2 1)
> (sort '(1 3 5 -2 -4 -6)
        (lambda (x y) (> (* x x) (* y y))))
(-6 5 -4 3 -2 1)
```

In the last case, the `COMPARER` *predicate is specified by a* `lambda` *special form. The sorting function uses this predicate to sort the numbers such that their* squares *are non-increasing.*

16.6 The Underlying Structure of Non-Empty Lists

Up to this point, we have seen that non-empty lists can often be effectively processed recursively using only the `first` and `rest` accessor functions. The reason for this is that the underlying structure of non-empty lists in Scheme is, in fact, based on decomposing them into their *first* and *rest* parts. The rest of this section explores that structure, revealing the central role of a data structure called a *cons cell*—also known as a *pair.*

16.6.1 Data Structures

In Computer Science, the term, *data structure,* refers to any organized (or structured) collection of data. Typically, each data structure has one or more slots for holding data. In some data structures, the slots for holding data are *indexed* so that any particular slot can be accessed by its corresponding (numerical) index. For example, the slots in *vectors*—to be discussed in Chapter 18—are indexed in this way. In other data structures, the slots for holding data are *named* so that any particular slot can be accessed by its name. Named slots are often called *fields*. For example, a `bank-account` data structure might have fields called `password` and `balance`. The rest of this section restricts attention to a very simple field-based data structure that, for historical reasons, is called a *cons cell.* Each cons cell has only two fields. For this reason, cons cells are also called *pairs.* General field-based data structures will be addressed thoroughly in Chapter 19.

16.6.2 Cons Cells (a.k.a. Pairs)

A *cons cell* is a field-based data structure structure that has only two fields: one named *first*, and one named *rest*. (Yes, that's right! Stay tuned for the relationship between cons cells and non-empty lists.) Scheme provides the following built-in functions for computing with cons cells, one of which we have already seen:

`cons`	For constructing a new cons cell
`cons?`	Type-checker predicate for cons cells

Example 16.6.1: The `cons` function revisited

Here is a more accurate contract for the `cons` *function. Notice that the second input need not be a list.*

```
;;  CONS  --  built-in function
;; ------------------------------------------------
;;  INPUTS:  FST, RST, any Scheme data
;;  OUTPUT:  A cons cell whose FIRST field contains FST,
;;           and whose REST field contains RST.
```

The following Interactions Window session demonstrates that the output generated by the `cons` *function is indeed a cons cell, as confirmed by the built-in* `cons?` *type-checker predicate:*

```
> (cons 1 2)
(1 . 2)
> (cons? (cons 1 2))
#t
> (cons 'x "1232")
(x . "1232")
> (cons? (cons 'x "1232"))
#t
> (cons #t 'abc)
(#t . abc)
> (cons? (cons #t 'abc))
#t
```

Notice that if the output value is a cons cell, DrScheme displays the result using the dotted-pair *notation. For example, a cons cell whose* first *field contains* `1` *and whose* rest *field contains* `2` *is displayed as* `(1 . 2)` *by DrScheme.*

- ⋆ DrScheme uses the *dotted-pair* notation when the *rest* field of a cons cell is something other than a list.
- ⋆ The dotted-pair notation is not legal Scheme syntax; so we cannot use it in our Scheme programs or in the Interactions Window.

It must be stressed that:

- ⋆ Although the dotted-pair notation shown above utilizes parentheses, it does *not* represent a list!

However:

- ⋆ When the *rest* field of a cons cell contains a list, then that cons cell *is* a non-empty list!

In such cases, the Scheme datum is both a cons cell *and* a non-empty list. This does not contradict the statement made long ago—in Chapter 2—that a datum can only belong to one data type because:

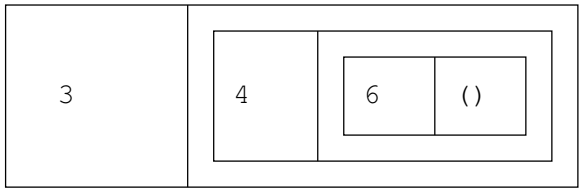

Figure 16.1: The non-empty list, `(3 4 6)`, as a single cons cell—with very particular contents

⋆ The set of non-empty lists is an example of a *compound data type.* Each non-empty list is, in fact, a cons cell that has special contents, in particular, one whose *rest* field contains a list.

Example 16.6.2: Cons cells vs. non-empty lists

The following interactions demonstrate that a non-empty list is a cons cell whose rest *field contains a list, whereas a cons cell whose* rest *field contains some other kind of data is not a list.*

```
> (cons? '(2 3 4))
#t
> (list? (rest '(2 3 4)))
#t
> (cons? (rest '(2 3 4)))
#t
> (cons 1 2)
(1 . 2)
> (list? (cons 1 2))        ⟵ A dotted pair is not a list
#f
```

Furthermore, as seen previously, when the rest *field of a cons cell contains a list, DrScheme displays that cons cell using the familiar list notation:*

```
> (cons 1 '(2 3 4))
(1 2 3 4)
> (cons 'x '(y z))
(x y z)
> (cons 1 ())
(1)
```

Fig. 16.1 shows one way of depicting the non-empty list, `(3 4 6)`—namely, as a single cons cell having very particular contents. In this case, the list is indeed represented as a single cons cell—the biggest one in the picture. The *first* field in this cons cell contains the datum `3`; the *rest* field of this cons cell contains another cons cell—one that represents the *rest* of the list (i.e., `(4 6)`). The *first* field of *that* cons cell contains the datum `4`; the *rest* field contains . . . yet another cons cell! The *first* field of the innermost cons cell contains the datum `6`; the *rest* field contains the empty list, which signals that we have reached the end of the list `(3 4 6)`. Notice that the list represented by these three nested cons cells has three elements: `3`, `4` and `6`. Notice further that the *first* field of each cons cell contains one of the elements of the list.

⋆ In general, if a list contains n elements, it can be represented by a nested structure of n cons cells.

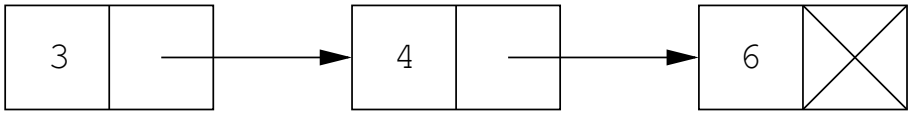

Figure 16.2: An alternative depiction of the non-empty list, `(3 4 6)`, as a chain of cons cells

Example 16.6.3: The structure of non-empty lists

The following interactions demonstrate that a list containing n elements can be represented by a nested structure of n cons cells.

```
> (cons 3 (cons 4 (cons 6 ())))
(3 4 6)
> (cons 1 (cons 2 (cons 3 (cons 4 ()))))
(1 2 3 4)
> (cons 'x (cons 'y (cons 'z ())))
(x y z)
```

Although Fig. 16.1 provides an accurate depiction of the nested structure of cons cells that can be used to represent a non-empty list, this kind of picture would get awfully difficult to draw for lists containing more than, say, five or ten elements. For this reason, we prefer to depict non-empty lists as *chains* of cons cells, using arrows, as illustrated in Fig. 16.2. It is important to realize that the non-empty list depicted by this figure is the same list as that depicted in Fig. 16.1 (i.e., we have two kinds of picture-syntax for one semantic list!). Instead of showing the rest of the list as a cons cell *nested inside* the *rest* field, this depiction uses an arrow from the *rest* field of one cons cell to the next cons cell in the chain. Similarly, the *rest* field of the second cons cell points to the third cons cell in the chain. Finally, the *rest* field of the last cons cell, which contains the empty list, is often depicted as a box with an X in it, signalling the end of the chain.

So... is a non-empty list a single cons cell? Or is it a chain of cons cells? The answer is: it depends on how you look at it! For example, according to the `cons?` type-checker predicate, a non-empty list is most definitely a single cons cell:

```
> (cons? '(2 3 4))
#t
```

On the other hand, if the *rest* field of a given cons cell C_1 contains a nested cons cell C_2, then the thing that actually gets written into the *rest* field of C_1 in the computer's memory is undoubtedly the *address* of C_2 (i.e., the location in the computer's memory where C_2 can be found). In other words, the *rest* field of C_1 contains a *pointer* to C_2—which can be represented by an arrow, as in Fig. 16.2! In short, you can look at it both ways. For our purposes, thinking of non-empty lists as chains of cons cells will be most convenient.

In-Class Problem 16.6.1: Defining our own type-checker predicate for lists

Define a predicate that satisfies the following contract:

```
;;  WELL-FORMED-LIST?
;; ------------------------------------------------------
```

```
;;  INPUT:   DATUM, anything
;;  OUTPUT:  #t if DATUM is an empty or non-empty list.
;;           If non-empty, DATUM should be a chain of cons
;;           cells, each of whose *rest* slot is filled by
;;           a well-formed list.
```

Here are some examples of its use:

```
> (well-formed-list? ())
#t
> (well-formed-list? '(a b c d))
#t
> (well-formed-list? (cons 1 (cons 2 3)))
#f
> (well-formed-list? 'xyz)
#f
```

Since this function is a predicate, you should be able to define it using `and`*,* `or` *and* `not`*, without using* `if` *or* `cond`*.*

⋆ Now that we have explored the underlying structure of non-empty lists in terms of cons cells, you should review all of the examples from earlier in this chapter to make sure that you understand the underlying structures of the lists involved.

Example 16.6.4: The `double-all` function revisited

Recall the definition of the `double-all` *function seen in Example 16.3.1 which takes a list of numbers as its input, and generates a list of the same length whose elements are obtained by doubling the corresponding elements from the input list.*

```
(define double-all
  (lambda (listy)
    (cond
     ;; Base Case:  LISTY is empty
     ((null? listy)
      ;; The double-all of () is ...
      ())
     ;; Recursive Case:  LISTY is non-empty
     (else
      ;; Double the first element and attach it to the
      ;; double-all of the rest of the list
      (cons (* 2 (first listy))
            (double-all (rest listy)))))))
```

Here's an example of its behavior:

```
> (double-all '(3 1 4 7))
(6 2 8 14)
```

In general, the `double-all` *function returns a list containing the same number of elements as its input. Equivalently, we may say that the* `double-all` *function is* length preserving. *This can be formally*

proved using the technique of mathematical induction; however, we shall content ourselves with a less formal analysis.

First, note that for any datum d *and any list* ℓ*, the list* `(cons d ℓ)` *has one more element than* ℓ*. Thus, for example, the list* `(3 1 4 7)`*, which is equivalent to* `(cons 3 '(1 4 7))`*, has one more element than* `(1 4 7)`*. But now consider* `(double-all '(3 1 4 7))`*. By the recursive case,* `(double-all '(3 1 4 7))` *effectively evaluates to* `(cons 6 (double-all '(1 4 7)))`*, which has one more element than* `(double-all '(1 4 7))`*. Therefore, if we want to show that* `(double-all '(3 1 4 7))` *and* `(3 1 4 7)` *have the same number of elements, we need only show that* `(cons 6 (double-all '(1 4 7)))` *and* `(cons 3 '(1 4 7))` *have the same number of elements, which is equivalent to showing that* `(double-all '(1 4 7))` *and* `(1 4 7)` *have the same number of elements. But then, by a similar line of reasoning, this will hold if and only if* `(double-all '(4 7))` *and* `(4 7)` *have the same number of elements. And that will hold if and only if* `(double-all '(7))` *and* `(7)` *have the same number of elements. And that will hold if and only if* `(double-all ())` *and* `()` *have the same number of elements. And that holds—since* `(double-all ())` *evaluates to* `()`*!*

The technique described in the preceding example can be used to show that the built-in `map` function is also length preserving. For example, `(1 2 3 4)` and the list generated by evaluating `(map facty '(1 2 3 4))` must have the same length.

In-Class Problem 16.6.2: Picturing the length preserving nature of `double-all` and `map`

Draw the chain of cons cells corresponding to the list `(3 1 4 7)`*. Draw a circle around the portion of that chain that corresponds to the* rest *of the list. Then draw the chain of cons cells corresponding to the list* `(6 2 8 14)` *generated by evaluating* `(double-all '(3 1 4 7))`*. Draw a circle around the portion of the chain corresponding to the rest of that list. Notice that the first cons cell in* `(3 1 4 7)` *is matched by the first cons cell in* `(6 2 8 14)`*; and that the rest of the cons cells in* `(3 1 4 7)` *are matched by the rest of the output list* `(6 2 8 14)` *generated by the recursive function call. In other words, each call to* `double-all` *effectively* consumes *one cons cell from the input list and* produces *one cons cell in the output list. For that reason, the input and output lists must have the same number of cons cells and, hence, the same number of elements.*

16.7 Hierarchical/Deep/Nested Lists

The syntax of Scheme expressions allows lists that contain other lists as elements. Indeed, lists may contain lists that contain other lists that contain other lists, and so on, to any desired depth.

⋆ A list that has at least one element that is itself a list is called a *hierarchical* (or *deep* or *nested*) list.

⋆ A list that does not contain any lists as elements is sometimes called a *flat* list.

For example, the expression `(x (2 (3) 2) #t)` denotes a hierarchical list whose *three* elements are: the symbol `x`, the subsidiary list `(2 (3) 2)`, and the boolean `#t`. This section demonstrates that recursively processing hierarchical lists is frequently only slightly more complicated than recursively processing flat lists. Indeed, when recursively processing the items in a deep list, it often happens that one need only insert one extra case to handle the possibility that the item currently under consideration is itself a list.

⇒ By convention, functions that recursively process hierarchical lists frequently have names ending in an asterisk (e.g., `sum-all⋆` instead of `sum-all`).

Example 16.7.1: Summing the items in a hierarchical list

Summing all of the items in a hiearchical list turns out to be only slightly more involved that summing the items in a flat list. (You may wish to review the `sum-all` *function defined in Example 16.2.2.) The contract for the hierachical version, called* `sum-all*`*, is given below, followed by some examples of its use.*

```
;;  SUM-ALL*
;; ------------------------------------------------------------
;;  INPUT:   HLISTY, a (possibly hierarchical) list of numbers
;;  OUTPUT:  The sum of all of the numbers appearing anywhere
;;           within HLISTY

> (sum-all* '(1 (2 (3 (4) 5) 6)))
21
> (sum-all* '(((((((10 100)))) 1))))
111
```

You may recall that the `sum-all` *function contained a* `cond` *expression with two cases: a base case and a recursive case. Below, the* `sum-all*` *function includes an extra recursive case that handles the possibility that the item currently under consideration (i.e.,* `(first hlisty)`*) is itself a list.*

```
(define sum-all*
  (lambda (hlisty)
    (cond
      ;; Base Case:  HLISTY is empty
      ((null? hlisty)
       0)
      ;; Recursive Case 1:  First element of HLISTY is a list
      ((list? (first hlisty))
       (+ (sum-all* (first hlisty))
          (sum-all* (rest hlisty))))
      ;; Recursive Case 2:  First element of HLISTY is not a list
      (else
       (+ (first hlisty)
          (sum-all* (rest hlisty)))))))
```

Notice that when `(first hlisty)` *is itself a list, it follows that both* `(first hlisty)` *and* `(rest hlisty)` *are lists. Therefore, the* `sum-all*` *function can be recursively applied to both of these lists, and the results added together to generate the desired sum. For example, if* `hlisty` *is the list* `((1 2 (3)) 4 (5 1))`*, then* `(first hlisty)` *is the list* `(1 2 (3))` *and* `(rest hlisty)` *is the list* `(4 (5 1))`*. Recursively applying* `sum-all*` *to these two lists yields the results,* `6` *and* `10`*, respectively. The sum of these two numbers (i.e.,* `16`*) is the sum of all of the numbers in* `hlisty`*.*

⇒ *Notice that, as usual, we let the recursive function calls do most of the work!*

Note. Using the `list?` predicate (e.g., in Recursive Case 1, above) to check whether `(first hlisty)` is a list can be terribly inefficient because, in cases where `(first hlisty)` happens to be a *long* list, the `list?` predicate will walk down its entire length, checking that it is a well formed chain of cons cells. Instead, if we assume that `hlisty` does *not* contain any malformed chains of cons cells, we can greatly increase the efficiency of Recursive Case 1 by using the `quick-list?` predicate, defined below.

```
;;  QUICK-LIST?
;; ------------------------------------------------
;;  INPUT:   DATUM, anything
;;  OUTPUT:  #t if DATUM is either () or a cons cell;
;;           #f otherwise.

(define quick-list?
  (lambda (datum)
    (or (null? datum) (cons? datum))))
```

Unlike `list?`, the `quick-list?` predicate does not walk down any chains of cons cells; instead, if `datum` is a cons cell, it simply assumes that it is the first cons cell in a well formed chain (i.e., that it is a non-empty list).

- ⋆ The rest of the examples in this section assume that all hierarchical lists are well formed (i.e., that they do not contain any malformed chains of cons cells).

Example 16.7.2: Top-level elements vs. leaf items in hierarchical lists

Recall In-Class Problem 16.2.2, whose goal was to define a function to compute the number of elements in a flat list. Here is one solution:

```
;;  LENGTHY
;; ---------------------------------------------------------------
;;  INPUT:  LISTY, any list
;;  OUTPUT: The number of elements of LISTY (i.e., its length)

(define lengthy
  (lambda (listy)
    (cond
     ;; Base Case:  LISTY is empty
     ((null? listy)
      ;; No elements in the empty list
      0)
     ;; Recursive Case:  LISTY is non-empty
     (else
      ;; (FIRST LISTY) is one element; the recursive
      ;; function call counts the REST of the elements
      (+ 1 (lengthy (rest listy)))))))
```

As demonstrated below, the `lengthy` *function does not care whether the individual elements of* `listy` *are symbols, numbers, booleans, or . . . even other lists! Thus, it counts what we sometimes call the* top-level elements *of* `listy`.

```
> (lengthy '(a b c d e))
5
> (lengthy '(x (1 1) (2 (3) 2) y))
4
> (lengthy '((((((3 3 3)))))))
1
```

For contrast, the function, `num-leaf-items⋆`*, counts the number of so-called* leaf *items in a possibly hierarchical list—that is, the items that appear at any level of the hierarchy.*

```
;;  NUM-LEAF-ITEMS*
;; -------------------------------------------------------
;;  INPUT:   HLISTY, a (possibly hierarchical) list
;;  OUTPUT:  The number of items that appear in HLISTY
;;             at any level of the hierarchy.
```

Here is how `num-leaf-items*` *treats the same lists encountered above:*

```
> (num-leaf-items* '(a b c d e))
5
> (num-leaf-items* '(x (1 1) (2 (3) 2) y))
7
> (lengthy '(((((((3 3 3)))))))
3
```

Notice that for flat lists such as `(a b c d e)`*, where each item occurs as a top-level element,* `num-leaf-items*` *outputs the same answer as* `lengthy`*. However,* `num-leaf-items*` *treats hierarchical lists much differently. Note that it does* not *count subsidiary lists, but only the primitive data that appear within them. Thus,* `(num-leaf-items* '(x (1 1) (2 (3) 2) y))` *outputs* `7`*, for the seven leaf items:* `x`*,* `1`*,* `1`*,* `2`*,* `3`*,* `2` *and* `y`*.*

Although `num-leaf-items*` *descends into the hierarchy of the input list, counting all the leaf items it finds along the way, defining this function is not difficult*—as long as we let recursive function calls do most of the work! *The following solution demonstrates that* `num-leaf-items*` *need only include one additional case, to handle the possibility that the element currently under consideration is itself a list:*

```
(define num-leaf-items*
  (lambda (hlisty)
    (cond
     ;; Base Case:  HLISTY is empty
     ((null? hlisty)
      0)
     ;; Recursive Case 1:  (FIRST HLISTY) is itself a list!
     ((quick-list? (first hlisty))
      ;; Recursive calls on (FIRST HLISTY) and (REST HLISTY)
      ;; compute the numbers of items in each part of HLISTY.
      (+ (num-leaf-items* (first hlisty))
         (num-leaf-items* (rest hlisty))))
     ;; Recursive Case 2:  (FIRST HLISTY) is NOT a list
     (else
      ;; Count 1 for (FIRST HLISTY); let the recursive
      ;; function call count the items in (REST HLISTY).
      (+ 1 (num-leaf-items* (rest hlisty)))))))
```

Notice that the Base Case and Recursive Case 2 are completely analogous to the Base Case and Recursive Case for `lengthy`*. The only difference is the insertion of Recursive Case 1, which handles the possibility that* `(first hlisty)` *is itself a list. And that case is easily handled because, in that case,* `(first hlisty)` *and* `(rest hlisty)` *are* both *lists. Recursively applying* `num-leaf-items*` *to both of those lists, and then summing the results, gives the desired answer.*

In-Class Problem 16.7.1: A hierarchical version of the `map` function

Define a function, called `map*`*, that satisfies the following contract:*

```
;;  MAP*
;; ------------------------------------------------------------
;;  INPUTS:  FUNC, a function that expects one input
;;           HLISTY, a (possibly hierarchical) list of
;;              suitable inputs for FUNC
;;  OUTPUT:  A list with the same structure as HLISTY, where
;;              each item is obtained by applying FUNC to the
;;              corresponding item in HLISTY.
```

Here are some examples of its behavior:

```
> (map* abs '(((-1) (2 -3) (-4 ((5)))))
(((1) (2 3) (4 ((5)))))
> (map* (lambda (x) (* x x)) '(1 (2 (3 (4) 5) 6) 7))
(1 (4 (9 (16) 25) 36) 49)
```

In-Class Problem 16.7.2: Flattening a hierarchical list

Define a function, called `flatten`*, that satisfies the following contract:*

```
;;  FLATTEN*
;; ------------------------------------------------------------
;;  INPUT:   HLISTY, a (possibly hierarchical) list
;;  OUTPUT:  A flat (i.e., non-hierarchical) list that contains
;;           all of the items from HLISTY "in the same order".
```

Here are some examples of its behavior:

```
> (flatten* '((4 2) 3 (x (y))))
(4 2 3 x y)
> (flatten* '(1 (2 (3) 4) 5))
(1 2 3 4 5)
```

Hint: In one case, use the built-in `append` *function; in another, use* `cons`.

16.8 Functions that can be Applied to Variable Numbers of Inputs

Recall that many of the built-in functions can be applied to variable numbers of inputs. For example, the built-in addition and multiplication functions can each be applied to zero or more inputs, as illustrated below.

```
> (+)                    ⟵ Adding no numbers together yields zero
0
> (+ 10 20)
30
> (+ 100 10 1)
111
> (+ 1000 200 30 4)
```

```
1234
> (*)                    ⟵ Multiplying no numbers together yields one
1
> (* 1 2 3 4 5)
120
> (* 10 10 10)
1000
```

Similarly, the built-in subtraction and division functions can each be applied to one or more inputs.

Given that function application in Scheme is provided through the evaluation of non-empty lists, it might not surprise you to learn that defining a function that can be applied to variable numbers of inputs can be handled by collecting the variable number of inputs into a list. As will be seen below, a slight extension to the syntax for the `lambda` special form enables this new capability.

Extending the Syntax of the `lambda` Special Form. In addition to the syntax shown in Chapter 7, the `lambda` special form also supports the following syntax.

```
(lambda args
  expr1
  expr2
  ...
  exprk)
```

where *`args`* can be any symbol expression. When such a function is applied to some number of inputs, those inputs are packaged together into a list, and that list of inputs becomes the value for the symbol *`args`* in the local environment inside the function-call box. Thus, inside the function-call box, this function behaves as though it received a list as its only input.

Example 16.8.1: Defining a function that can be applied to a variable number of inputs

For this example, we aim to define a function that can take any number of numerical inputs. To make things simple, this function will simply multiply those inputs together. We begin by defining a similar function that takes a single input *that contains a* list *of numbers.*

```
;;  MY-MULTY
;; ------------------------------------------------------
;;  INPUTS:  A list of numbers
;;  OUTPUT:  The product of the numbers in that list

(define my-multy
  (lambda (listy)
    (if (null? listy)
        1
        (* (first listy)
           (my-multy (rest listy))))))
```

With `my-multy` *in hand, the desired function,* `my-multy-multy`*, can be easily defined using the new syntax for the* `lambda` *special form, as follows.*

```
;;  MY-MULTY-MULTY
;; ------------------------------------------------
;;  INPUTS:  Any number of numbers
;;  OUTPUT:  The product of those numbers
```

```
(define my-multy-multy
  (lambda args
    ;; Since ARGS is a LIST of numbers...
    (my-multy args)))
```

The following interactions demonstrate the difference between `my-multy` *and* `my-multy-multy`*.*

```
> (my-multy '(1 2 3 4))
24
> (my-multy-multy 1 2 3 4)
24
> (my-multy '(10 10 10))
1000
> (my-multy-multy 10 10 10)
1000
```

Using the above example as a guide, we could convert any function that takes a single input that is a list into an equivalent function that can be applied to inputs that are drawn from such a list. However, we can also take a more direct approach to defining a function like `my-multy-multy` by using the built-in `apply` function.

Example 16.8.2: The built-in `apply` function

The built-in `apply` *function satisfies the following contract:*

```
;;  APPLY  --  built-in
;; ------------------------------------------------------------
;;  INPUTS:  FUNC, a function
;;           LISTY, a list of suitable inputs for FUNC
;;  OUTPUT:  The result of applying FUNC to the inputs in LISTY
```

The following interactions demonstrate the difference between applying a function (e.g., the built-in addition function) to a variable number of inputs versus using `apply` *to apply that same function to the elements of a given list.*

```
> (+ 100 10 1)
111
> (apply + '(100 10 1))
111
> (* 1 2 3 4 5)
120
> (apply * '(1 2 3 4 5))
```

There is little mystery behind the built-in `apply` function...

Example 16.8.3: Implementing our own version of `apply`

```
;;  MY-APPLY
;; -----------------------------------------------------
;;  INPUTS:  FUNC, a function
;;           LISTY, a list of suitable inputs for FUNC
;;  OUTPUT:  The result of applying FUNC to the elements
;;           of LISTY

(define my-apply
  (lambda (func listy)
    (eval (cons func listy))))

> (my-apply + '(1 2 3 4))
10
> (my-apply * '(10 10 10))
1000
```

Example 16.8.4: A more direct approach to defining a function that can be applied to a variable number of inputs

```
;;  MY-MULTY-MULTY-V2
;; -------------------------------------------------
;;  INPUTS:  Any number of numerical inputs
;;  OUTPUT:  The product of those numbers

(define my-multy-multy-v2
  (lambda args
    (cond
      ;; Base Case:  ARGS is empty
      ((null? args)
       1)
      ;; Recursive Case:  ARGS is non-empty
      (else
       (* (first args)
          ;; Since (REST ARGS) is a LIST of numbers...
          (apply my-multy-multy-v2 (rest args)))))))
```

Note the use of `apply` *in the last line. It is needed because* `my-multy-multy-v2` *is supposed to be applied to any number of numerical inputs, not a single input that is a list of numbers. Here are some examples of* `my-multy-multy-v2` *in action.*

```
> (my-multy-multy-v2 1 2 3 4)
24
> (my-multy-multy-v2 10 10 10)
1000
```

Special Forms Introduced in this Chapter

`time` Displays timing information

Built-in Functions Introduced in this Chapter

`abs`	Computes the absolute value of its input
`first`, `rest`	Accessor functions for lists
`cons`	Create a new list by attaching a new item to the front of a given list
`cons?`	Type-checker for cons cells
`second`, `third`, `fourth`, etc.	Additional accessor functions for lists
`list`	Create a list containing the specified items
`member`	Does an item appear in a list?
`map`	Apply given function to each element of a list, in turn
`length`	Compute the number of elements in a list
`list-ref`	Fetch the n^{th} element of a list—general purpose accessor function
`append`	Concatenate two lists
`reverse`	Reverse the elements of a list
`sort`	Sort a list according to a given comparison function
`apply`	Apply a function to the elements of a given list