PART IV

Definition by Self-reference

Chapter 22

Lists and functions on them

22.1 Limitations of structs

When we define a struct, among the first questions we ask is "how many parts does the struct have?" This question assumes that the struct *has* a fixed number of parts: a `posn` has two numbers, a `color` has three numbers, a `person` has two strings and a number, a `moving-x` has a number and a string, ... In general, a struct is a way to collect a fixed number of related pieces of information into one "package" that we can pass around as a single object, and store in a single variable.

But what if we wanted to collect an *unknown number* of related pieces of information into one "package", pass it around as a single object, and store it in a single variable? For example, consider the collection of students registered for a particular course: if a student adds (or drops) the course, does that require redefining the struct with one more (or fewer) field, then rewriting and retesting every function that operated on the struct, since there are now one more (or one fewer) getter functions, and every constructor call must now have one more (or one fewer) argument than before? That seems ridiculous: one should be able to write software once and for all to work on a collection of students, allowing the number of students to change while the program is running.

There are many other ways to collect related pieces of information into a package, of which the simplest is a "list".

22.2 What is a list?

You're all familiar with lists in daily life: lists of friends, lists of schools or jobs to apply to, lists of groceries to buy, lists of things to pack before going on a trip. In each case, a list is not changed in any fundamental way by changing the number of items in it. A list can be reduced to 1 or 0 items, and then have more items added to it later; reducing it to 1 or even 0 items didn't make it stop being a list.

This fact — that a list can have as few as 0 elements — underlies the way we'll define lists in Racket. Here are three apparently-obvious facts about lists:

1. **A list is either empty or non-empty.**

   Not terribly exciting, although it suggests that we'll do some kind of definition by choices with two cases: empty and non-empty.

2. **An empty list has no parts.**

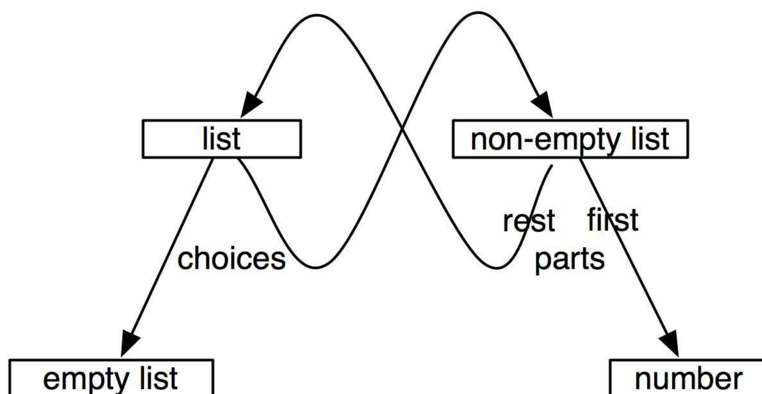
   Again, not terribly exciting. The next one is a little more interesting:

3. **A non-empty list has a first element. It also has "the rest" of the list, *i.e.* everything after the first element.**

   This looks like a definition by parts: there are two parts, the "first" element and the "rest". What types are these parts? If we were defining a list of numbers, obviously, the "first" element would be a number; in a list of strings, the "first" element would be a string. But what about the "rest"? If the list consists of only one element, the "rest" should be empty; if the list consists of two elements, the "rest" contains one element; and so on. In other words, the "rest" of a non-empty list is a list (either empty or non-empty).

When we introduced "definition by choices" and "definition by parts", we said they were ways to define a new data type *from previously-defined data types.* We've loosened that requirement a bit now: the *list* data type is defined by choices from the *non-empty-list* data type, which is defined by parts from (among other things) the *list* data type. Neither data type can be "previous" to the other, because each depends on the other. However, if we're willing to define both at once, we can get a tremendous amount of programming power.

The following diagram shows the relationships among types in a list of numbers: a "list" is defined by choices as either "empty list" or "non-empty list", while "non-empty list" is defined by parts as a number and a "list".



22.3 Defining lists in Racket

In this section we'll develop a definition of lists, and learn to write functions on them, using only what you already know about definition by parts and by choices. The resulting definition is a little awkward to work with, so in section 22.4 we'll discuss the more practical version of lists that's built into Racket. If you prefer to "cut to the chase," you can skip this section.

For concreteness, we'll define a list of strings; you can also define a list of numbers, or booleans, or even lists similarly. We'll use "los" as shorthand for "list of strings".

22.3.1 Data definitions

Recall our first fact about lists: "a list [of strings] is either empty or non-empty." This is a definition by choices with two choices. It tells us that for any function that takes in a list of strings, we'll need an empty test case and at least one non-empty test case. (In fact, we'll usually want at least three test cases: an empty example, an example of length 1, and at least one longer list.) It also tells us that the body of a function on lists of strings will probably involve a two-way `cond`, deciding between the empty and non-empty cases:

```
#|
(define (function-on-los L)
  ; L                    a list of strings
  (cond [...    ...]
        [...    ...]
  ))
|#
```

The second fact about lists, "An empty list has no parts," can be represented in Racket by defining a struct with no parts:

```
; An empty list has no parts.
(define-struct empty-list ())
; make-empty-list :  nothing -> empty-list
; empty-list?  :  anything -> boolean
#|
(define (function-on-empty-list L)
  ; L     an empty-list
  ...)
|#
```

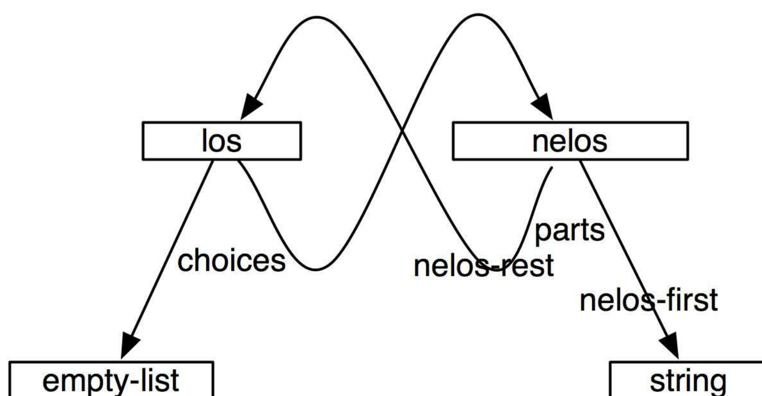
This looks a little weird, admittedly: we've never before defined a struct with no parts, but there's no rule against it. Since there are no parts, there are no getters; there's only a constructor (which takes no parameters) and a discriminator. In other words, we can create an empty list, and recognize it when we see it.

Note that I haven't specified that an empty-list is an empty-list *of strings*: since it doesn't contain anything anyway, an empty-list of strings can be exactly the same as an empty-list of numbers or anything else.

In practice, there's seldom much point in writing a function whose only input is an empty list. All empty lists contain exactly the same information, so such a function would have to return the same answer in all cases, so why bother writing a function? So we'll skip the *function-on-empty-list* template from now on.

Now for the third fact about lists: "A non-empty list has two parts: the first element and the rest." More specifically: "A non-empty list of strings has two parts: the first element (a string) and the rest (a list of strings)." This seems to call for definition by parts. I'll use "nelos" as shorthand for "non-empty list of strings".

```
; A nelos has two parts:  first (a string) and rest (a los)
(define-struct nelos (first rest))
; make-nelos :  string los -> nelos
; nelos-first :  nelos -> string
; nelos-rest :  nelos -> los
; nelos?  :  anything -> boolean
#|
(define (function-on-nelos L)
  ; L                                  a nelos
  ; (nelos-first L)                    a string
  ; (nelos-rest L)                     a los
  ; (function-on-los (nelos-rest L)) whatever this returns
  ...)
|#
```



Since `(nelos-rest L)` is a list of strings, the obvious thing to do to it is call some function that works on a list of strings, like `function-on-los`. It's quite useful to include this in the inventory, as we'll see shortly.

With this information, we can write the complete data definition, with input templates for both *los* and *nelos*, fairly concisely:

```
; A los is either (make-empty-list) or a nelos
#|
(define (function-on-los L)
  ; L   a list of strings
  (cond [ (empty-list?  L) ...]
        [ (nelos?  L)      (function-on-nelos L)]
  ))
|#
```

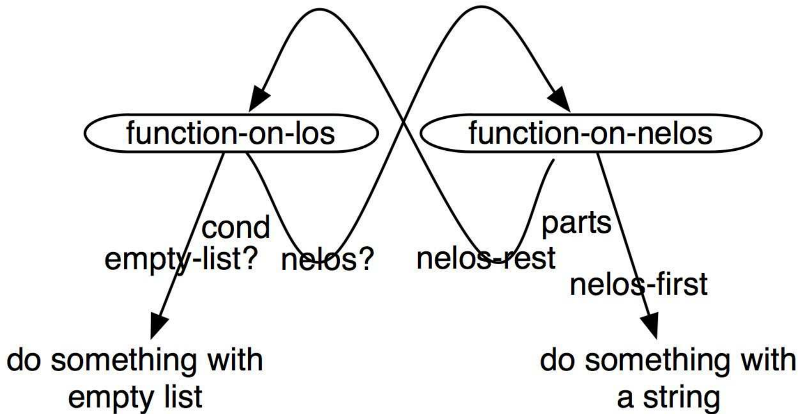
```
; A nelos looks like
; (make-nelos string los)

#|
(define (function-on-nelos L)
  ; L                                  a cons
  ; (nelos-first L)                    a string
  ; (nelos-rest L)                     a los
  ; (function-on-los (nelos-rest L))   whatever this returns
  ...)
|#
```

(We'll come back to the output template in Chapter 23.)

Note that once a *los* has been determined to be non-empty, the obvious thing to do to it is call some function that works on non-empty lists, like `function-on-nelos`.

Note also that because the *los* and *nelos* data types each refer to one another, the `function-on-los` and `function-on-nelos` templates refer to one another in a corresponding way.



22.3.2 Examples of the los data type

As with any new data type, we should make up some examples to make things feel more real and concrete. We'll need at least one empty example, which we can build with `make-empty-list`:

```
(define nothing (make-empty-list))
```

and at least one non-empty example, which we can build with `make-nelos`. The `make-nelos` function expects two arguments: a string and a *los*. The only *los* we already have is `nothing`, so

```
(define english (make-nelos "hello" nothing))
```

This represents a list of strings whose first element is `"hello"` and whose rest is an empty list, so there is no second element.

Suppose we wanted a list with `"bonjour"` as its first element and `"hello"` as the second and last. This is easy by calling `make-nelos`:

```
(define fr-eng (make-nelos "bonjour" english))
```

We can go on to build even longer lists, as shown in Figure 22.1.

Figure 22.1: Defining and using list-of-strings

```
 ; An empty list has no parts.
(define-struct empty-list ())
; make-empty-list :  nothing -> empty-list
; empty-list?  :  anything -> boolean

; A nelos has two parts:  first (a string) and rest (a los).
(define-struct nelos (first rest))
; make-nelos :  string los -> nelos
; nelos-first :  nelos -> string
; nelos-rest :  nelos -> los
; nelos?  :  anything -> boolean
#|
(define (function-on-nelos L)
  ; L                               a nelos
  ; (nelos-first L)                 a string
  ; (nelos-rest L)                  a los
  ; (function-on-los (nelos-rest L)) whatever this returns
  ...)
|#

; A los is either an empty-list or a nelos.
#|
(define (function-on-los L)
  ; L                               a los
  (cond [(empty-list?  L) ...]
        [(nelos?  L)      (function-on-nelos L)]
  ))
|#
(define nothing (make-empty-list))
(define english (make-nelos "hello" nothing))
(define fr-eng (make-nelos "bonjour" english))
(define heb-fr-eng (make-nelos "shalom" fr-eng))
(define shfe (make-nelos "buenos dias" heb-fr-eng))
(define ashfe (make-nelos "salaam" shfe))
(define dwarfs (make-nelos "sleepy" (make-nelos "sneezy"
  (make-nelos "dopey" (make-nelos "doc" (make-nelos "happy"
  (make-nelos "bashful" (make-nelos "grumpy" nothing)))))))
```

Practice Exercise 22.3.1 *Copy Figure 22.1 into your Definitions pane (it should be available as a download from the textbook Web site), and try the following expressions. For each one,* predict *what it will return before hitting ENTER, and see whether you were*

right. If not, figure out why it came out as it did.

```
(empty-list? nothing)
(nelos? nothing)
(nelos-first nothing)
(nelos-rest nothing)
(empty-list? english)
(nelos? english)
(nelos-first english)
(nelos-rest english)
(empty-list? (nelos-rest english))
(nelos? fr-eng)
(nelos-first fr-eng)
(nelos-rest fr-eng)
(nelos? (nelos-rest fr-eng))
(nelos-first (nelos-rest fr-eng))
(nelos-rest (nelos-rest fr-eng))
(nelos? ashfe)
(nelos-first ashfe)
(nelos-rest ashfe)
(nelos-first (nelos-rest (nelos-rest ashfe))))
(nelos-first (nelos-rest (nelos-rest (nelos-rest dwarfs))))
```

22.3.3 Writing a function on los

How would we write a function on the *los* data type? In a way, this is the wrong question: our templates above show *two* functions, *function-on-los* and *function-on-nelos*, which depend on one another, so when we write a specific function, it too will probably consist of two functions that depend on one another.

Worked Exercise 22.3.2 *Define a function* `count-strings` *that takes in a* los *and returns how many strings are in it: 0 for an empty list, 1 for a list of one element, and so on.*

Solution: We'll write two functions: one that works on a *los* and one that works on a *nelos*.

Contracts:

```
; count-strings : los -> number
; count-strings-on-nelos : nelos -> number
```

The data analysis is already done.

Test cases:

```
(check-expect (count-strings nothing) 0)
(check-expect (count-strings english) 1)
(check-expect (count-strings fr-eng) 2)
(check-expect (count-strings ashfe) 5)
(check-expect (count-strings dwarfs) 7)
```

```
; can't call (count-strings-on-nelos nothing)
; because nothing isn't a nelos
(check-expect (count-strings-on-nelos english) 1)
(check-expect (count-strings-on-nelos fr-eng) 2)
(check-expect (count-strings-on-nelos ashfe) 5)
(check-expect (count-strings-on-nelos dwarfs) 7)
```

Skeletons and Inventories:
Conveniently, we already have templates that do most of the work for us:

```
(define ( count-strings L)
  ; L   a los
  (cond [(empty-list?  L) ...]
        [(nelos?  L)      ( count-strings-on-nelos L)]
  ))

(define ( count-strings-on-nelos L)
  ; L                               a nelos
  ; (nelos-first L)                 a string
  ; (nelos-rest L)                  a los
  ; ( count-strings (nelos-rest L))a number
  ...)
```

Note that `count-strings` and `count-strings-on-nelos` refer to one another in the same way `function-on-los` and `function-on-nelos` refer to one another, which in turn corresponds to the way *los* and *nelos* refer to one another.

Now we just need to fill in everywhere that there's a "...". The first one, the answer in the `(empty-list?  L)` case, is easy: an empty list has a length of 0. (We *could* write a `count-strings-on-empty-list` function to do this, but that seems like too much work just to get the answer 0.)

```
(define (count-strings L)
  ; L   a los
  (cond [(empty-list?  L) 0]
        [(nelos?  L)      (count-strings-on-nelos L)]
  ))
```

The other "..." is the body of `count-strings-on-nelos`. From the inventory, we have an expression for the number of strings in the rest of the list. So how many strings are there in the *whole* list? If you see immediately how to do this, great; if not, let's try an **inventory with values**. We'll pick a moderately complicated example:

```
(define (count-strings-on-nelos L)
  ; L                   a nelos  (cons "a" (cons "b" (cons "c" empty)))
  ; (nelos-first L)     a string "a"
  ; (nelos-rest L)      a los    (cons "b" (cons "c" empty))
  ; (count-strings (nelos-rest L))    a number 2
  ; right answer                      a number 3
```

How could you get the right answer, 3, from the things above it? The one that most closely resembles 3 is 2; you can get 3 from 2 by adding 1. This suggests the body

```
(define (count-strings-on-nelos L)
  ; L                    a nelos  (cons "a" (cons "b" (cons "c" empty)))
  ; (nelos-first L)      a string "a"
  ; (nelos-rest L)       a los    (cons "b" (cons "c" empty))
  ; (count-strings (nelos-rest L))   a number 2
  ; right answer                     a number 3
  (+ 1 (count-strings (nelos-rest L))) )
```

Does this make sense? It says the number of strings in the whole list is one more than the number of strings in the rest of the list, which is certainly true.

Run the test cases, and they should all work. Use the Stepper to watch the computation for some non-trivial examples, like `(count-strings shfe)`. ■

22.3.4 Collapsing two functions into one

We've written functions before that depended on auxiliary functions; the only new thing here is that the auxiliary function depends on the original function in turn. And it's perfectly natural that when we're working with two different data types, we have to write two different functions. However, the only place `count-strings-on-nelos` is used is inside `count-strings`, so if we prefer, we can replace the call to `count-strings-on-nelos` with its body:

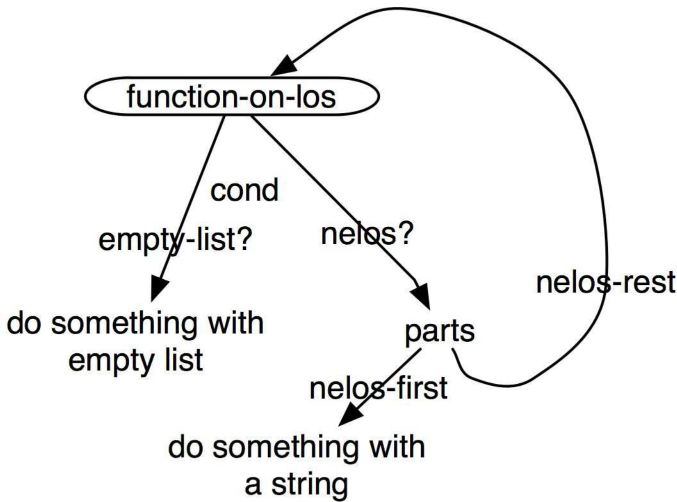
```
(define (count-strings L)
  ; L   a los
  (cond [(empty-list?  L)                  0]
        [(nelos?  L)
        ; L                                a nelos
        ; (nelos-first L)                  a string
        ; (nelos-rest L)                   a los
        ; (count-strings (nelos-rest L))   a number
        (+ 1 (count-strings (nelos-rest L))) ]
  ))
```

Note that now the `count-strings` function *calls itself*. Some of you may have written functions in the past that called themselves, and the most likely result was something called an "infinite loop": the function called itself to answer the same question, then called itself to answer the same question, then called itself to answer the same question, and never accomplished anything. What we've done here is different: rather than calling the function to answer the *same* question, we're calling it to answer a *smaller* question, then using the result to figure out the answer to the original question.

The single-function solution is usually shorter and simpler, but later on we'll encounter situations in which we *have* to use the two-function solution, so you should know both approaches.

In order to write functions on lists as a single function rather than two, we must likewise collapse the two templates into one:

```
#|
(define (function-on-los L)
  ; L   a los
  (cond [(empty-list?  L)                    ...]
        [(nelos?  L)
        ; L                                  a nelos
        ; (nelos-first L)                    a string
        ; (nelos-rest L)                     a los
        ; (function-on-los (nelos-rest L))  whatever this returns
        ...]))
|#
```



22.4 The way we really do lists

The approach taken in Section 22.3 works, but it's rather awkward to work with. Lists are so common and useful that they're built into Racket (and some other languages). In reality, most Racket programmers use the built-in list functions rather than defining a list data type themselves.

As in section 22.3, we'll define a list of strings for concreteness. You can also define a list of numbers, or booleans, or even lists similarly. We'll use "los" as shorthand for "list of strings".

22.4.1 Data definitions

Recall our first fact about lists: "a list [of strings] is either empty or non-empty." This is a definition by choices, with two choices. It tells us that for any function that takes in a list of strings, we'll need an empty test case and at least one non-empty test case. (In fact, we'll usually want at least three test cases: an empty example, an example of length 1, and at least one longer list.) It also tells us that the body of a function on lists of strings will probably involve a two-way `cond`, deciding between the empty and non-empty cases:

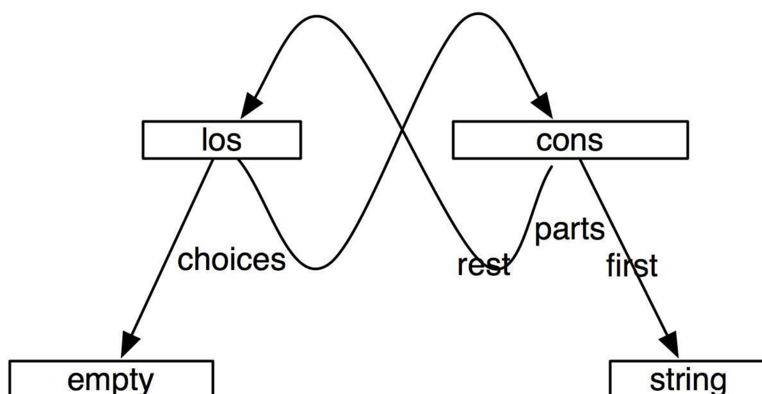
```
#|
(define (function-on-los L)
  ; L                      a list of strings
  (cond [...    ...]
        [...    ...]
  ))
|#
```

The second fact about lists is “An empty list has no parts.” Racket provides a built-in constant `empty` and a built-in function `empty?` to represent and recognize empty lists, respectively.

```
; empty :  a constant that stands for an empty list
; empty?  :  anything -> boolean
```

Now for the third fact about lists: “A non-empty list has two parts: the first element and the rest.” Let’s make it more specific: “A non-empty list of strings has two parts: the first element (a string) and the rest (a list of strings).” This seems to call for definition by parts. For convenience, Racket has a built-in data type to represent a non-empty list. Since putting one non-empty list inside another is the usual way to “construct” a large list, we use the word `cons` (short for “construct”). Racket provides the following built-in functions:

```
; A non-empty list, or cons, has two parts:
;   first (whatever type the elements are) and
;   rest (a list)
; cons :  element list -> non-empty-list
; first :  non-empty-list -> element
; rest :  non-empty-list -> list
; cons?  :  anything -> boolean
```



With this information, we can write the complete data definition, with input templates for both *los* and *nelos*, fairly concisely:

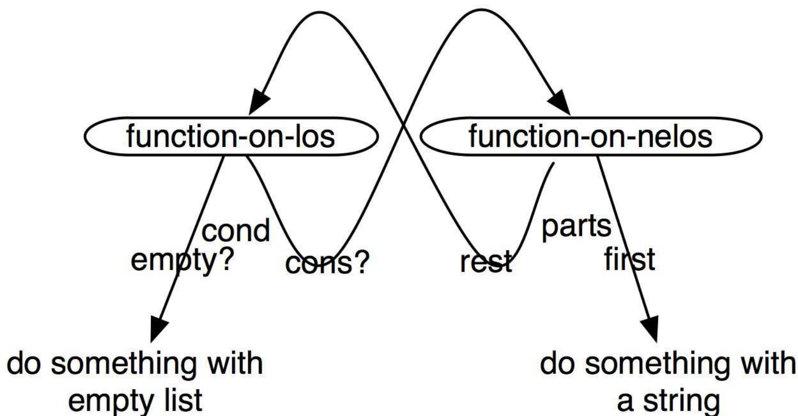
```
; A los is either empty or a nelos
#|
(define (function-on-los L)
  ; L   a list of strings
  (cond [(empty?  L)      ...]
        [(cons?  L)       (function-on-nelos L)]
  ))
|#

; A nelos looks like
; (cons string los)

#|
(define (function-on-nelos L)
  ; L                               a cons
  ; (first L)                       a string
  ; (rest L)                        a los
  ; (function-on-los (rest L))      whatever this returns
  ...)
|#
```

(We'll come back to the output template in Chapter 23.)

Note that because the *los* and *nelos* types refer to one another, the `function-on-los` and `function-on-nelos` templates refer to one another in a corresponding way.



22.4.2 Examples of the los data type

As with any new data type, we should make up some examples to make things feel more real and concrete. We have `empty` to provide an empty example. We'll need to build non-empty examples using `cons`, which (in our list-of-strings example) expects a string and a list of strings. The only list of strings we already have is `empty`, so we'll use that:

```
(define english (cons "hello" empty))
```

This represents a list of strings whose first element is `"hello"` and whose rest is an empty list, so there is no second element.

Suppose we wanted a list with `"bonjour"` as its first element and `"hello"` as the second and last. This is easy by calling `cons`:

```
(define fr-eng (cons "bonjour" english))
```

We can go on to build even longer lists, as shown in Figure 22.2.

Figure 22.2: Lists of strings, using built-in Racket features

```
 ; An empty list has no parts.
; empty :  a constant
; empty?  :  anything -> boolean

; A cons has two parts:  first (a string) and rest (a los).
; cons :  string los -> nelos
; first :  nelos -> string
; rest :  nelos -> los
; cons?  :  anything -> boolean
#|
(define (function-on-nelos L)
  ; L                                  a nelos
  ; (first L)                          a string
  ; (rest L)                           a los
  ; (function-on-los (rest L))         whatever this returns
  ...)
|#

; A los is either an empty-list or a nelos.
#|
(define (function-on-los L)
  ; L                                  a los
  (cond [(empty-list?  L)    ...]
        [(nelos?  L)         (function-on-nelos L)]
  ))
|#
(define english (cons "hello" empty))
(define fr-eng (cons "bonjour" english))
(define heb-fr-eng (cons "shalom" fr-eng))
(define shfe (cons "buenos dias" heb-fr-eng))
(define ashfe (cons "salaam" shfe))
(define dwarfs (cons "sleepy" (cons "sneezy" (cons "dopey" (cons "doc"
  (cons "happy" (cons "bashful" (cons "grumpy" empty))))))))
```

Practice Exercise 22.4.1 *Copy Figure 22.2 into your Definitions pane (it should be available as a download from the textbook Web site), and try the following expressions. For each one,* predict *what it will return before hitting ENTER, and see whether you were right. If not, figure out why it came out as it did.*

```
(empty? empty)
(cons? empty)
(first empty)
(rest empty)
(empty? english)
(cons? english)
(first english)
(rest english)
(empty? (rest english))
(cons? fr-eng)
(first fr-eng)
(rest fr-eng)
(cons? (rest fr-eng))
(first (rest fr-eng))
(rest (rest fr-eng))
(cons? ashfe)
(first ashfe)
(rest ashfe)
(first (rest (rest ashfe))))
(first (rest (rest (rest dwarfs))))
```

22.4.3 Writing a function on los

How would we write a function on the *los* data type? In a way, this is the wrong question: our templates above show *two* functions, *function-on-los* and *function-on-nelos*, which depend on one another, so when we write a specific function, it too will probably consist of two functions that depend on one another.

Worked Exercise 22.4.2 *Define a function* `count-strings` *that takes in a* los *and returns how many strings are in it: 0 for an empty list, 1 for a list of one element, and so on.*

Solution: We'll write two functions: one that works on a *los*, and one that works on a *nelos*.

Contracts:

```
; count-strings : los -> number
; count-strings-on-nelos : nelos -> number
```

The data analysis is already done.

Test cases:

```
(check-expect (count-strings empty) 0)
(check-expect (count-strings english) 1)
(check-expect (count-strings fr-eng) 2)
(check-expect (count-strings ashfe) 5)
(check-expect (count-strings dwarfs) 7)

; can't call (count-strings-on-nelos empty)
; because empty isn't a nelos
(check-expect (count-strings-on-nelos english) 1)
(check-expect (count-strings-on-nelos fr-eng) 2)
(check-expect (count-strings-on-nelos ashfe) 5)
(check-expect (count-strings-on-nelos dwarfs) 7)
```

Skeletons and Inventories:
Conveniently, we already have templates that do most of the work for us:

```
(define ( count-strings L)
  ; L   a los
  (cond [(empty?  L)    ...]
        [(cons?  L)     ( count-strings-on-nelos L)]
  ))

(define ( count-strings-on-nelos L)
  ; L                             a nelos
  ; (first L)                     a string
  ; (rest L)                      a los
  ; ( count-strings (rest L))     a number
  ...)
```

Note that `count-strings` and `count-strings-on-nelos` refer to one another in the same way `function-on-los` and `function-on-nelos` refer to one another, which in turn corresponds to the way *los* and *nelos* refer to one another. The "main" function, the one we're really interested in, is `count-strings`, but we need a helper function `count-strings-on-nelos`.

Now we just need to fill in everywhere that there's a ". . .". The first one, the answer in the `(empty?  L)` case, is easy: an empty list has a length of 0. (We *could* write a `count-strings-on-empty-list` function to do this, but that seems like too much work just to get the answer 0.)

```
(define (count-strings L)
  ; L   a los
  (cond [(empty?  L)    0]
        [(cons?  L)     (count-strings-on-nelos L)]
  ))
```

The other ". . ." is the body of `count-strings-on-nelos`. From the inventory, we have an expression for the number of strings in the rest of the list. So how many strings are there in the *whole* list? If you see immediately how to do this, great; if not, let's try an **inventory with values**. We'll pick a moderately complicated example:

```
(define (count-strings-on-nelos L)
  ; L             a nelos  (cons "a" (cons "b" (cons "c" empty)))
  ; (first L)     a string "a"
  ; (rest L)      a los    (cons "b" (cons "c" empty))
  ; (count-strings (rest L))    a number 2
  ; right answer                a number 3
```

How could you get the right answer, 3, from the things above it? The one that most closely resembles 3 is 2; you can get 3 from 2 by adding 1. This suggests the body

```
(define (count-strings-on-nelos L)
  ; L             a nelos  (cons "a" (cons "b" (cons "c" empty)))
  ; (first L)     a string "a"
  ; (rest L)      a los    (cons "b" (cons "c" empty))
  ; (count-strings (rest L))    a number 2
  ; right answer                a number 3
  (+ 1 (count-strings (rest L))) )
```

Does this make sense? It says the number of strings in the whole list is one more than the number of strings in the rest of the list, which is certainly true.

Run the test cases, and they should all work. Use the Stepper to watch the computation for some non-trivial examples, like `(count-strings shfe)`. ∎

22.4.4 Collapsing two functions into one

We've written functions before that depended on auxiliary functions; the only new thing here is that the auxiliary function depends on the original function in turn. And it's perfectly natural that when we're working with two different data types, we have to write two different functions. However, the only place `count-strings-on-nelos` is used is inside `count-strings`, so if we prefer, we can replace the call to `count-strings-on-nelos` with its body:

```
(define (count-strings L)
  ; L   a los
  (cond [(empty?  L)                     0]
        [(cons?  L)
        ; L                              a nelos
        ; (first L)                      a string
        ; (rest L)                       a los
        ; (count-strings (rest L))       a number
        (+ 1 (count-strings (rest L))) ]
  ))
```

Note that now the `count-strings` function *calls itself*. Some of you may have written functions in the past that called themselves, and the most likely result was something called an "infinite loop": the function called itself to answer the same question, then called itself to answer the same question, then called itself to answer the same question, and never accomplished anything. What we've done here is different: rather than calling the function to answer the *same* question, we're calling it to answer a *smaller* question, then using the result to figure out the answer to the original question.

SIDEBAR:

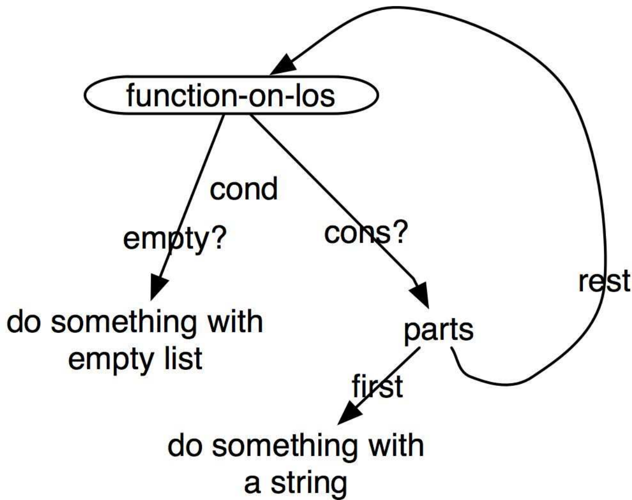
Computer scientists use the word "recursion" for the notion that a function can call itself, or two or more functions can call one another. Such functions are described as "recursive". Generations of computer science students have been mystified by recursion, often because they try to think through the entire computation at once. It seems to work better to think about only one level at a time. Concentrate on making sure that

- the answer to the empty case is right, and
- if you have a correct answer for the rest of the list, you can construct a correct answer for the whole list.

If this bothers you, here's a way to justify it. Suppose we wrote a function this way and it *didn't* work correctly, *i.e.* there were some legal inputs on which it produced a wrong answer; call these "bad" inputs. In particular, there must be a *shortest* bad input. We know the function works correctly on the empty list, so the shortest bad input must be non-empty, and thus have a "first" and a "rest". The "rest" of this list is shorter than the shortest bad input, so it's not bad, *i.e.* the function works correctly on it. We know the function produces a correct answer for the whole list from a correct answer for the rest of the list. Since the answer on the rest of the list *is* correct, that means the answer to the shortest bad input is correct too, which means this isn't a "bad" input after all. So the shortest "bad" input can't be empty, and it can't be one item longer than a "good" input, so it can't exist at all, *i.e.* the function is always correct.

In order to write functions on lists as a single function rather than two, we must likewise collapse the two templates into one:

```
#|
(define (function-on-los L)
  ; L   a los
  (cond [(empty?  L)                      ...]
        [(cons?  L)
        ; L                               a nelos
        ; (first L)                       a string
        ; (rest L)                        a los
        ; (function-on-los (rest L))      whatever this returns
        ...]))
|#
```



Again, it's a matter of personal preference whether you solve a problem like this with two functions that call one another, or one that calls itself; do whichever makes more sense to you. They both work.

Whether you use two functions that call one another or one function that calls itself, there will be somewhere that checks whether the list is empty, and returns an answer *without* calling itself. This is called the *base case* of the recursion.

22.5 Lots of functions to write on lists

So far we've seen how to solve only one problem on lists, *i.e.* counting how many strings are in a list of strings. We've seen slightly different definitions depending on whether we define our own structs or use Racket's built-in list features, and on whether we write it as two functions that call one another or one function that calls itself, but it's still only one problem. To really get the hang of writing functions on lists, you'll need to practice on a number of examples.

I've described these examples using Racket's built-in list features; they could of course be written to use the `empty-list` and `nelos` structures defined in section 22.3, but the definitions would be longer and harder to understand.

Worked Exercise 22.5.1 ***Define*** *a data type* list-of-numbers *(or* lon *for short), including a template for functions operating on lists of numbers.* ***Develop*** *a function* `count-numbers` *that takes in a list of numbers and returns a number.*

Solution: The data definition is similar to that for *list-of-strings*:

```
; A list-of-numbers is either
;   empty or
;   a nelon (non-empty list of numbers).
#|
(define (function-on-lon L)
  ; L   a list of numbers
  (cond [ (empty?  L)    ...]
        [ (cons?  L)     (function-on-nelon L)]
  ))
|#

; A nelon looks like
; (cons number lon)

#|
(define (function-on-nelon L)
  ; L                              a cons
  ; (first L)                      a number
  ; (rest L)                       a lon
  ; (function-on-lon (rest L))     whatever this returns
  ...)
|#
```

And not surprisingly, the function definition is quite similar to that of `count-strings`:

```
; count-numbers :  lon -> number
(check-expect (count-numbers empty) 0)
(check-expect (count-numbers (cons -4 empty)) 1)
(check-expect
  (count-numbers (cons 5 (cons 2 (cons 8 (cons 6 empty)))))
  4)
(check-expect (count-numbers-on-nelon (cons -4 empty)) 1)
(check-expect
  (count-numbers-on-nelon (cons 5 (cons 2 (cons 8 (cons 6 empty)))))
  4)

(define (count-numbers L)
  ; L   a lon
  (cond [(empty?  L)    0]
        [(cons?  L)     (count-numbers-on-nelon L)]
  ))

(define (count-numbers-on-nelon L)
  ; L                         a nelon
  ; (first L)                 a string
  ; (rest L)                  a lon
  ; (count-numbers (rest L)) a number
  (+ 1 (count-numbers (rest L))) )
```

or, writing it more simply as a single function,

```
(define (count-numbers L)
  ; L                                 a lon
  (cond   [(empty?  L)                0]
          [(cons?  L)
           ; L                        a nelon
           ; (first L)                a number
           ; (rest L)                 a lon
           ; (count-numbers (rest L))   a number
           (+ 1 (count-numbers (rest L)))]
  ))
```

In fact, aside from its name, this function is *identical* to `count-strings`: neither one actually makes any use of the type of the elements, so either one would work on a list of any type. There's a built-in Racket function `length` that does the same job, and now that you've seen how you could have written it yourself, you should feel free to use the built-in `length` function. ∎

The next example is more interesting, and depends on the type of the elements.

Worked Exercise 22.5.2 ***Develop*** *a function* `add-up` *that takes in a list of numbers and returns the result of adding them all together. For example,*

```
(check-expect (add-up (cons 4 (cons 8 (cons -3 empty)))) 9)
```

Solution: We already have the data definition for *list-of-numbers*, so we'll go on to the function. The contract, examples, skeleton, and inventory are easy:

```
; add-up :  list-of-numbers -> number
(check-expect (add-up empty) 0)
(check-expect (add-up (cons 14 empty)) 14)
(check-expect (add-up (cons 3 (cons 4 empty))) 7)
(check-expect (add-up (cons 4 (cons 8 (cons -3 empty)))) 9)
```

```
(define (add-up L)
  ; L                           a lon
  (cond [(empty?  L)            ...]
        [(cons?  L)             (add-up-nelon L)]
  ))

(define (add-up-nelon L)
  ; L                           a nelon
  ; (first L)                   a number
  ; (rest L)                    a lon
  ; (add-up (rest L))           a number
  ... )
```

We need to fill in the two "..." gaps. The answer to the empty case is obviously 0. For the other "...", let's try an inventory with values:

```
(define (add-up-nelon L)
  ; L                   nelon   (cons 4 (cons 8 (cons -3 empty)))
  ; (first L)           number  4
  ; (rest L)            lon     (cons 8 (cons -3 empty))
  ; (add-up (rest L))   number  5
  ; right answer        number  9
  ... )
```

So how can you get the right answer, 9, from the things above it? The two lists don't look promising, but the numbers 4 and 5 do: we can get 9 by adding the 4 (*i.e.* `(first L)`) and the 5 (*i.e.* `(add-up (rest L))`). This suggests the definition

```
(define (add-up-nelon L)
  ; L                   nelon   (cons 4 (cons 8 (cons -3 empty)))
  ; (first L)           number  4
  ; (rest L)            lon     (cons 8 (cons -3 empty))
  ; (add-up (rest L))   number  5
  ; right answer        number  9
  (+ (first L) (add-up (rest L))))
```

Does this make sense? Should the sum of a list of numbers be the same as the first number plus the sum of the rest of the numbers? Of course. Test the function, and it should work on all legal inputs.

Here's a shorter single-function version, developed the same way.

```
(define (add-up L)
  ; L                           lon
  (cond [(empty?  L)            0]
        [(cons?  L)
         ; L                    nelon
         ; (first L)            number
         ; (rest L)             lon
         ; (add-up (rest L))    number
         (+ (first L) (add-up (rest L)))
  ))
```

∎

Exercise 22.5.3 *Suppose you work for a toy company that maintains its inventory as a list of strings, and somebody has come into the store looking for a doll. You want to know whether there are any in stock.* ***Develop*** *a function* `contains-doll?` *that takes in a list of strings and tells whether any of the strings in the list is* `"doll"`.

Exercise 22.5.4 ***Develop*** *a function* `any-matches?` *that takes in a string and a list of strings and tells whether any of the strings in the list is the same as the given string. For example,*

```
(check-expect
  (any-matches?  "fnord" (cons "snark" (cons "boojum" empty)))
  false)
(check-expect
  (any-matches?  "fnord" (cons "snark" (cons "fnord" empty)))
  true)
```

Hint: The templates for operating on lists use a conditional to decide whether you've got an empty or a non-empty list. This function needs to make another decision: does the current string match the target or not? You can do this with another conditional, or (since this function returns a boolean), you can do it more simply without the extra conditional.

Exercise 22.5.5 ***Develop*** *a function* `count-matches` *that takes in an object and a list of objects and tells* how many *(possibly zero) of the objects in the list are the same as the given object. For example,*

```
(check-expect
  (count-matches "cat" (cons "dog" (cons "cat" (cons "fish"
    (cons "cat" (cons "cat" (cons "wombat" empty)))))))
  3)
(check-expect
  (count-matches 1 (cons 3 (cons 1 (cons 4
                         (cons 1 (cons 5 (cons 9 empty)))))))
  2)
```

Hint: For this one, you probably *will* need a nested conditional.

There's an additional difference: this function is supposed to work on *any kind of object*, not just strings. So instead of `string=?`, you'll need to use the built-in function `equal?`.

Exercise 22.5.6 ***Develop*** *a function* `count-votes-for-name` *that takes in a string (the name of a candidate) and a list of strings (the votes cast by a bunch of voters) and tells how many of the voters voted for this particular candidate.*

Hint: This is really easy if you re-use previously-written functions.

Exercise 22.5.7 ***Develop a function*** `count-over` *that takes in a number and a list of numbers, and tells how many of the numbers in the list are larger than the specified number.*

Exercise 22.5.8 ***Develop*** *a function* `average` *that takes in a list of numbers and returns their average,* i.e. *their sum divided by how many there are. For this problem, you may* assume there is at least one number in the list*.*

Hint: Not *every* function on lists can best be written by following the templates ...

Exercise 22.5.9 ***Develop a function*** `safe-average` *that takes in a list of numbers and returns their average; if the list is empty, it should signal an error with an appropriate and user-friendly message.*

Exercise 22.5.10 ***Develop*** *a function* `convert-reversed` *that takes in a list of numbers. You may* assume *that all the numbers are integers in the range 0-9,* i.e. *decimal digits. The function should interpret them as digits in a decimal number,* ones place first *(trust me, this actually makes the problem easier!), and returns the number they represent. For example,*

```
(check-expect
  (convert-reversed (cons 3 (cons 0 (cons 2 (cons 5 empty)))))
  5203)
```

Do not *use the built-in* `string->number` *function for this exercise.*

Exercise 22.5.11 ***Develop*** *a function* `multiply-all` *that takes in a list of numbers and returns the result of multiplying them all together. For example,*

```
(check-expect (multiply-all (cons 3 (cons 5 (cons 4 empty)))) 60)
```

Hint: What is the "right answer" for the empty list? It may not be what you think at first!

Exercise 22.5.12 *A "dollar store" used to mean a store where everything cost less than a dollar.* ***Develop*** *a function* `dollar-store?` *that takes in a list of numbers (the prices of various items), and tells whether the store qualifies as a "dollar store".*

Exercise 22.5.13 ***Develop*** *a function* `all-match?` *that takes in a string and a list of strings, and tells whether* all *the strings in the list match the given string. For example,*

```
(check-expect
  (all-match?  "cat" (cons "cat" (cons "dog" (cons "cat" empty))))
  false)
(check-expect
  (all-match?  "cat" (cons "cat" (cons "cat" empty)))
  true)
```

Exercise 22.5.14 ***Develop*** *a function* `general-bullseye` *that takes in a list of numbers and produces a white image with black concentric rings at those radii.*

Hint: I recommend using an empty image like `(circle 0 "solid" "white")` as the answer for the empty case.

Exercise 22.5.15 ***Develop an animation*** *that displays a bull's-eye pattern of black rings on a white background. Each second, an additional ring will be added, three pixels outside the previous outer ring.*

Hint: Use a list of numbers as the model. For your tick handler, write a function that takes in a list of numbers and sticks one more number onto the front of the list, equal to three times the length of the existing list.

Exercise 22.5.16 ***Develop an animation*** *that displays a bull's-eye pattern, as in Exercise 22.5.15, but each second, an additional ring will be added at a* random radius*.*

In section 22.4.4, why did we move the body of `function-on-nelos` inside the body of `function-on-los`, rather than the other way around? Because in most cases, we want the resulting function to work on *all* lists, including empty. But sometimes it works better to move `function-on-los` inside `function-on-nelos` instead.

Exercise 22.5.17 ***Develop a function*** `largest` *that takes in a list of numbers and returns the largest one.*

Hint: This function doesn't really make sense on an empty list, so the input data type is really "non-empty list of numbers," and the simplest test case should be a one-element list. Since `largest` doesn't make sense on an empty list, you should be careful never to call it on an empty list.

If you use the two-function approach, the "main" function here is the one for non-empty lists; the one for possibly-empty lists is the helper function. And if you use a

one-function approach, you'll need to move the function for possibly-empty lists inside the function for non-empty lists, *e.g.*

```
(define (function-on-nelos L)
  ; (first L) a string
  ; (rest L)  a list
  (cond [(empty?  (rest L)) ...]
        [(cons?  (rest L))
         ; (function-on-nelos (rest L))    whatever this returns
         ...]))
```

Exercise 22.5.18 ***Develop a function*** `count-blocks` *that takes in a list of numbers, which may contain some repetitions, and tells how many* blocks *of repeated numbers there are. A* block *is one or more copies of the same number appearing in the list, with no other numbers in between. For example,*

```
(check-expect (count-blocks
  (cons 3 (cons 3 (cons 2 (cons 7 (cons 7 (cons 7
    (cons 2 (cons 2 empty))))))))
  4)
```

because this list has a block of 3's, then a block of 2's, then a block of 7's, then another block of 2's: four blocks in all.

Hint: I know of at least two ways to solve this problem. Both involve useful techniques that you should know; try both.

First, try writing a version of this function that only works on non-empty lists; as in Exercise 22.5.17, the base case becomes "is the list one element long?". For one-element lists, the answer is easy; for longer lists, you know that the list has both a first and a second element, and can reasonably ask whether they're the same. Once this works on all non-empty lists, add an "empty" case.

The other approach is to write a helper function that takes in not only a list of numbers but also the number we're "already looking at:" if the list is non-empty, you can reasonably check whether its first number is the same as the one you're already looking at.

Exercise 22.5.19 ***Develop a function*** `count-even-odd-blocks` *that takes in a list of integers and tells how many blocks of consecutive even or consecutive odd numbers there are. For example,*

```
(check-expect (count-even-odd-blocks
  (cons 3 (cons 9 (cons 2 (cons 7 (cons 1 (cons 1
    (cons 2 (cons 4 empty))))))))
  4)
```

because the numbers 3 and 9 form a block of odd numbers; 2 is a block of even numbers; 7, 1, and 1 form a block of odd numbers; and 2 and 4 are a block of even numbers, for four blocks in all.

Hint: Obviously, this is similar to Exercise 22.5.18, but if you use the "helper function" approach, it doesn't need to take in a specific "already seen" number, but only whether the previous number was even or odd. Instead of passing in the previous number, therefore, try writing two separate (but similar) helper functions `even-helper` and `odd-helper`.

This approach is a little longer, but it's a powerful technique that you can use for many problems in the future. Try it.

> SIDEBAR:
>
> There are many problems that call for scanning through a list from left to right, looking for particular patterns. The above approach is one that computer scientists call a *finite-state machine* or *finite-state automaton*.

Exercise 22.5.20 ***Develop a function*** `random-element` *that takes in a non-empty list and returns a randomly chosen element of it. Ideally, each element should be equally likely to be chosen.*

Hint: You'll probably need the built-in `list-ref` function, which takes in a non-empty list and a number, and returns the element that far away from the beginning of the list. For example,

```
(check-expect
  (list-ref (cons "a" (cons "b" (cons "c" empty))) 0)
  "a")
(check-expect
  (list-ref (cons "a" (cons "b" (cons "c" empty))) 1)
  "b")
(check-expect
  (list-ref (cons "a" (cons "b" (cons "c" empty))) 2)
  "c")
```

`list-ref` produces an error message if you give it too large a number, so make sure you don't do that.

Since `random-element` is unpredictable, you won't be able to test it with `check-expect`, but you can call it a bunch of times with the same list and see whether each element is chosen roughly the same number of times.

Warning: The `list-ref` function is useful when you need to get the element of a list at an *arbitrary numeric position*. That's actually not a common thing to need; 95% of the time, you'll be better off using `first` and `rest`.

Exercise 22.5.21 ***Write a data definition****, including templates, for a* list of lists of strings*. Write several examples of this data type.*

Develop a function `total-length` *that takes in a list of lists of strings and returns the total number of strings appearing in all the lists put together.*

Develop a function `longest` *that takes in a non-empty list of lists of strings and returns the longest of the lists. If there are two or more of the same maximum length, it may return either one at your choice.*

22.6 Lists of structs

As we've seen, writing a function to work on a list of numbers is almost exactly like writing a function to work on a list of strings. Not surprisingly, writing a function to work on a list of posns, or employees, or other types like that isn't much harder.

Worked Exercise 22.6.1 ***Develop a function*** `any-on-diag?` *that takes in a list of* `posn` *structures and tells whether any of them are "on the diagonal,"* i.e. *have* x *and* y *coordinates equal to one another.*

Solution: The data definition is straightforward:

```
; A list-of-posns is either
;   empty or
;;  a nelop (non-empty list of posns).
#|
(define (function-on-lop L)
  ; L   a list of posns
  (cond [ (empty?  L)    ...]
        [ (cons?  L)     (function-on-nelop L)]
  ))
|#

; A nelop looks like
; (cons posn lop)

#|
(define (function-on-nelop L)
  ; L                            a cons
  ; (first L)                    a posn
  ; (posn-x (first L))           a number
  ; (posn-y (first L))           a number
  ; (rest L)                     a lop
  ; (function-on-lop (rest L))   whatever this returns
  ...)
|#
```

For test cases, we need an empty list and at least two or three non-empty lists: at least one with right answer `true` and at least one with right answer `false`.

```
(check-expect (any-on-diag?  empty) false)
(check-expect (any-on-diag?  (cons (make-posn 5 6) empty)) false)
(check-expect (any-on-diag?  (cons (make-posn 5 5) empty)) true)
(check-expect (any-on-diag?  (cons (make-posn 5 6)
                                   (cons (make-posn 19 3) empty)))
              false)
(check-expect (any-on-diag?  (cons (make-posn 5 6)
                                   (cons (make-posn 19 19) empty)))
              true)
(check-expect (any-on-diag?  (cons (make-posn 5 5)
                                   (cons (make-posn 19 3) empty)))
              true)
```

The function templates give us a good start on writing the `any-on-diag?` function:

```
(define ( any-on-diag?  L)
  ; L   a list of posns
  (cond [ (empty?  L)    ...]
        [ (cons?  L)     (any-on-diag-nelop?  L)]
  ))

(define ( any-on-diag-nelop?  L)
  ; L                           a cons
  ; (first L)                   a posn
  ; (posn-x (first L))          a number
  ; (posn-y (first L))          a number
  ; (rest L)                    a lop
  ; ( any-on-diag?  (rest L))     a boolean
  ...)
```

The right answer for the empty list is `false` (that was one of our test cases), so we can fill that in immediately. And the obvious question to ask about the `posn` is "are the x and y coordinates equal?", *i.e.* `(= (posn-x (first L)) (posn-y (first L)))`, so we'll add that to the template too:

```
(define (any-on-diag?  L)
  ; L   a list of posns
  (cond [(empty?  L)    false]
        [(cons?  L)     (any-on-diag-nelop?  L)]
  ))
(define (any-on-diag-nelop?  L)
  ; L                                           a cons
  ; (first L)                                   a posn
  ; (posn-x (first L))                          a number
  ; (posn-y (first L))                          a number
  ; (= (posn-x (first L)) (posn-y (first L))) a boolean
  ; (rest L)                                    a lop
  ; (any-on-diag?  (rest L))                    a boolean
  ...)
```

Now let's try an inventory with values. In fact, since the function has to return a boolean, we'll do *two* inventories-with-values, one returning `true` and one returning `false`:

```
(define (any-on-diag-nelop?  L)
 ; L  a cons   (cons (make-posn 5 6) (cons (make-posn 19 3) empty))
 ; (first L)                 a posn    (make-posn 5 6)
 ; (posn-x (first L))        a number  5
 ; (posn-y (first L))        a number  6
 ; (= (posn-x (first L)) (posn-y (first L)))  a boolean   false
 ; (rest L)                  a lop     (cons (make-posn 19 3) empty)
 ; (any-on-diag?  (rest L)) a boolean false
  ; right answer             a boolean false
 ...)
```

```
(define (any-on-diag-nelop?  L)
 ; L  a cons   (cons (make-posn 5 5) (cons (make-posn 19 3) empty))
 ; (first L)                 a posn    (make-posn 5 5)
 ; (posn-x (first L))        a number  5
 ; (posn-y (first L))        a number  5
 ; (= (posn-x (first L)) (posn-y (first L)))  a boolean   true
 ; (rest L)                  a lop     (cons (make-posn 19 3) empty)
 ; (any-on-diag?  (rest L)) a boolean false
  ; right answer             a boolean true
 ...)
```

What expression could we fill in for the "..." that would produce the right answer in both cases? Well, the right answer is a boolean, and there are two booleans above it in the inventory. The most common ways to combine booleans to get another boolean are `and` and `or`. In this case `or` gives the right answer:

```
(define (any-on-diag-nelop?  L)
  ; L                                           cons
  ; (first L)                                   posn
  ; (posn-x (first L))                          number
  ; (posn-y (first L))                          number
  ; (= (posn-x (first L)) (posn-y (first L))) boolean
  ; (rest L)                                    lop
  ; (any-on-diag?  (rest L))                    boolean
  (or (= (posn-x (first L)) (posn-y (first L)))
      (any-on-diag?  (rest L))))
```

Test the function(s), and you should get correct answers.

If you prefer to solve this as a single function, the process is similar, but the end result is

```
(define (any-on-diag? L)
  ; L                                              list of posns
  (cond[(empty? L)      false]
       [(cons? L)
        ; L                                        a cons
        ; (first L)                                posn
        ; (posn-x (first L))                       number
        ; (posn-y (first L))                       number
        ; (= (posn-x (first L)) (posn-y (first L))) boolean
        ; (rest L)                                 lop
        ; (any-on-diag? (rest L))                  boolean
        (or (= (posn-x (first L)) (posn-y (first L)))
            (any-on-diag? (rest L)))
       ]))
```

■

Exercise 22.6.2 ***Develop a function*** `any-over-100K?` *that takes in a list of* `employee` *structures (from Exercise 21.2.1) and tells whether any of them earn over $100,000 per year.*

Exercise 22.6.3 ***Develop a function*** `lookup-by-name` *that takes in a string and a list of* `employee` *structures (from Exercise 21.2.1) and returns the first one whose name matches the string. If there is none, it should return* `false`*.*

Exercise 22.6.4 ***Develop a function*** `total-votes` *that takes in a list of* `candidate` *structures (from Exercise 21.3.3) and returns the total number of votes cast in the election.*

Exercise 22.6.5 ***Develop a function*** `avg-votes` *that takes in a list of* `candidate` *structures and returns the* average *number of votes for each candidate.*

Hint: This doesn't have a reasonable answer if there are no candidates. How do you want to handle this case?

Exercise 22.6.6 ***Develop a function*** `winner` *that takes in a list of* `candidate` *structures and returns the one with the most votes. If there are two or more tied for first place, you can return whichever one you wish.*

Hint: This doesn't have a reasonable answer if there are no candidates. How do you want to handle this case?

Exercise 22.6.7 ***Develop an animation*** *similar to Exercise 20.6.4, but every few seconds a dot is* added *to the screen (in addition to whatever dots are already there), and if you click inside* any *of the existing dots, the game ends. (The game will be easy to win, since pretty soon the screen fills with dots so it's hard* not *to hit one.)*

Hint: Use a list of posns as the model.

22.7 Strings as lists

Worked Exercise 22.7.1 *Develop a function* `count-e` *that takes in a string and returns the number of times the letter "e" appears in it. You may* assume *there are no capital letters (*i.e. *you don't need to count "E").*

Solution: The contract and examples are fairly straightforward:

```
; count-e :  string -> number
(check-expect (count-e "") 0)
(check-expect (count-e "a") 0)
(check-expect (count-e "e") 1)
(check-expect (count-e "ab") 0)
(check-expect (count-e "ae") 1)
(check-expect (count-e "ea") 1)
(check-expect (count-e "ee") 2)
(check-expect
  (count-e "Tweedledum and Tweedledee were going to a fair")
  10)
```

But how do we write the function?

In a way, this looks similar to the `count-matches` function. Intuitively, a string is a sequence of characters, which "feels" sort of like a list. But we don't have a template for writing functions on a string, looking at each letter one at a time.

There are two ways to handle this. One is to develop such a template so we can write functions that operate directly on strings. The other is to convert a string into a list and then use functions on lists to handle it. Both approaches are useful to know, so let's try both.

A template for strings

To develop a template for operating on strings, we'll proceed by analogy with lists. A string is either empty or non-empty; if it's non-empty, it has a first character and "the rest".

If only we had built-in functions analogous to `empty?`, `cons?`, `first`, and `rest`... Wait: Exercise 13.2.4 defines a function that tells whether a string is empty. We could easily write a `non-empty-string?` function from it using `not` (or we could just use `else`, and not define `non-empty-string?` at all). Exercise 19.3.3 defines a `first-char` function analogous to `first`, while exercises 9.2.3 and 19.3.2 define a `chop-first-char` function analogous to `rest`. So with these, the template is easy:

```
#|
(define (function-on-string str)
  ; str a string
  (cond [(empty-string?  str)        ...]
        [(non-empty-string?  str)    (function-on-nes str)]
  ))
|#
```

```
#|
(define (function-on-nes str)
  ; str                                     non-empty string
  ; (first-char str)                        length-1 string
  ; (chop-first-char str)                   string
  ; (function-on-string (chop-first-char str)) whatever
  ...)
|#
```

or, collapsing the two functions into one,

```
#|
(define (function-on-string str)
  (cond [(empty-string? str)               ...]
        [(non-empty-string? str)
         ; str                             non-empty string
         ; (first-char str)                length-1 string
         ; (chop-first-char str)           string
         ; (function-on-string (chop-first-char str)) whatever
         ...]))
|#
```

With this template, the solution is pretty easy:

```
(define (count-e str)
  (cond [(empty-string? str)               0 ]
        [(non-empty-string? str)
         ; str                             non-empty string
         ; (first-char str)                length-1 string
         ; (chop-first-char str)           string
         ; (count-e (chop-first-char str))
         (cond [(string=? (first-char str) "e")
                (+ 1 (count-e (chop-first-char str)))]
               [else (count-e (chop-first-char str))])]))
```

The `char` data type

Before we can discuss the other approach, we need to learn about another data type: *char*. Strings in Racket and most other languages are built from *characters*, which are actually another data type you can work with directly. There are built-in functions `char=?`, which compares two characters to see if they're the same, and `char?`, which checks whether something is a character at all.

Recall that to type in a string, regardless of length, you surround it in double-quotes. To type in a character, you put #\ in front of it. For example, the first (and only) character in the string `"e"` is #\e; the third character in the string `"5347"` is #\4; and the third character in the string `"Hi there"` is #\ , which can also be written more readably as #\space.)

Converting strings to lists

There's a built-in function `string->list` which converts a string into a list of chars. So using this approach, we could define `count-e` very simply as follows:

```
(define (count-e str)
  (count-matches #\e (string->list str)))
```

Note that `count-matches` works on a list of *any* type of object, including `char`. ■

I recommend trying some of the following problems using the template, and some using conversion to a list, so you're comfortable using both techniques.

Exercise 22.7.2 ***Develop a function*** `count-vowels` *that takes in a string and returns how many vowels (any of the letters "a", "e", "i", "o", or "u") it contains. You may* assume *there are no capital letters.*

Exercise 22.7.3 ***Develop a function*** `has-spaces?` *that takes in a string and tells whether it contains any blanks.*

Exercise 22.7.4 ***Develop a function*** `count-words` *that takes in a string and tells how many words are in it. A "word" is a sequence of letters; whether it's one letter or ten, it counts as a single word. Note also that there might be punctuation marks, spaces, multiple spaces, numbers,* etc. *in between words.*

Hint: This problem is similar to Exercise 22.5.19. In addition, you'll probably want the built-in function `char-alphabetic?`. Look it up in the Help Desk.

22.8 Arbitrarily nested lists

In Exercise 22.5.21 we saw that the elements of a list can themselves be lists (of strings, numbers, *etc.*). There is also no rule that all the elements of a list are the same type: one can have a list of which *some* elements are strings, others are lists of strings, and others even lists of lists of strings. For example, suppose we were working with English sentences, and had decided to represent a sentence as a list of words, *e.g.* the sentence "Bob likes Mary" could be stored in the computer as `(cons "Bob" (cons "likes" (cons "Mary" empty)))`. But how should we represent a sentence like "Jeff said "Bob likes Mary" yesterday"? The thing that Jeff said is in itself a whole sentence, so it would be nice to represent it the same way we represent sentences ... but it fits into the grammar of the whole sentence in exactly the same way as if he had said only one word.

```
(cons "Jeff" (cons "said" (cons
     (cons "Bob" (cons "likes" (cons "Mary" empty)))
    (cons "yesterday" empty))))
```

Exercise 22.8.1 ***Write a data definition****, including templates, for a* nested string list*, in which each element may be either a string or another nested string list.*

Exercise 22.8.2 ***Translate*** *the following English sentences into nested string lists, using a list to represent each quotation.*

- *"We watched "Rudolph the Red-Nosed Reindeer" and "Frosty the Snowman" on Christmas Eve."*
- *" "This is silly," said Mary."*
- *"Grandpa said "I'll read you a book called "The Princess Bride", one of my favorites. "Once upon a time, there was a beautiful princess named Buttercup. The stable-boy, Wesley, was in love with her, but never said anything but "As you wish." " " The boy was already asleep."*

Exercise 22.8.3 *Ingredient lists on food packaging sometimes get deeply nested. I found a package of ice cream with the following ingredients list (I am not making this up!) :*

> *Milk, skim milk, cream, hazelnut swirl (praline paste (hazelnuts, sugar, milk chocolate (sugar, cocoa butter, milk, chocolate, natural flavor, soy lecithin), bittersweet chocolate (sugar, chocolate, cocoa butter, butter oil, natural flavor, soy lecithin)), corn oil, powdered sugar (sugar, corn starch), dark chocolate (sugar, chocolate, cocoa butter, butter oil, natural flavor, soy lecithin), corn starch, cocoa processed with alkali, coconut oil, mono- and di-glycerides, salt, soy lecithin), sugar, chocolate truffle cake (semi-sweet chocolate (sugar, chocolate, cocoa butter, soy lecithin), cream, chocolate cookie crumbs (enriched flour (flour, niacin, reduced iron, thiamine mononitrate, riboflavin, folic acid), sugar, partially hydrogenated soybena, cottonseed, and canola oil, cocoa processed with alkali, high fructose corn syrup, yellow corn flour, chocolate, salt, dextrose, baking soda, soy lecithin), corn syrup, butter, chocolate, sugar, natural flavor), bittersweet chocolate (sugar, chocolate, cocoa butter, butter oil, natural flavor, soy lecithin), cocoa processed with alkali, egg yolks, natural flavor, guar gum, carob bean gum, carrageenan, dextrose*

We could represent this in Racket as follows:

```
(define milk-chocolate (cons "sugar" (cons "cocoa-butter"
  (cons "milk" (cons "chocolate" (cons "natural flavor"
  (cons "soy lecithin" empty)))))))
(define bittersweet-chocolate (cons "sugar" (cons "chocolate"
  (cons "cocoa butter" (cons "butter oil" (cons
  "natural flavor" (cons "soy lecithin" empty)))))))
(define praline-paste (cons "hazelnuts" (cons "sugar"
  (cons milk-chocolate (cons bittersweet-chocolate empty)))))
...
```

Note how I've defined Racket variables for the ingredients that have ingredient lists of their own.

Finish *translating this ingredient list to Racket.*

Exercise 22.8.4 ***Develop a function*** `count-strings-nested` *that takes in a nested string list and returns the total number of simple strings in it, no matter how many levels of nested lists they're inside.*

Exercise 22.8.5 ***Develop a function*** `max-nesting-depth` *that takes in a nested string list and returns its maximum nesting depth:* `empty` *has a nesting depth of 0, a list of strings has a nesting depth of 1, a list that contains some lists of strings has a nesting depth of 2,* etc.

It can be difficult to read and write such nested lists and the test cases for Exercises 22.5.21, 22.8.1, 22.8.3, 22.8.4, and 22.8.5. In Section 23.3 we'll learn a more compact notation for lists that makes this easier.

22.9 Review of important words and concepts

Whereas a structure collects a *fixed* number of related pieces of information into one object, a list allows you to collect a *variable* number of related pieces of information into

one object. The *list* data type is defined by combining techniques we've already seen: definition by choices (is it empty or not?) and definition by parts (if it's non-empty, it has a first and a rest, which is itself a list).

We already know how to write functions on data types defined by choices, and defined by parts; the new feature is that since a list really involves two interdependent data types, a function on lists is often written as two interdependent functions. However, since one of these is generally only used in one place in the other, we can often make the program shorter and simpler by combining the two functions into one that calls itself on the rest of the list.

A list can contain *any* kind of data: numbers, strings, structs, or even other lists. The template for functions that work on lists is almost the same for all of these; the only difference is what you can do with `(first the-list)`. In particular, if the first element of a list is itself a list, you may need to call the same function on it.

Operating on strings is much like operating on lists. A function that takes in a string can test whether it's the empty string or not, extract its first character and the remaining characters, and so on ... or it can use the built-in function `string->list` to convert the whole string into a list of characters, and then use the usual list template to work with this list of characters.

22.10 Reference: Built-in functions on lists

This chapter introduced the following built-in constants and functions:

- `empty`
- `empty?`
- `cons`
- `first`
- `rest`
- `cons?`
- `string->list`
- `char=?`
- `char?`
- `char-alphabetic?`
- `list-ref`