Picturing Programs
An introduction to computer programming

Stephen Bloch

September 17, 2009

Part III

Definition by parts

Chapter 22

Animations and posns

22.1 The `posn` data type

Recall Exercise 19.2.1, in which a picture moved left or right in response to the left and right arrow keys, respectively. An obvious modification would be to have it move *up or down* in response to *those* arrow keys; this could be easily done by deciding that the model represented the y coordinate rather than the x coordinate. So how would we *combine* these two, allowing the picture to move up, down, left, and right in response to the appropriate arrow keys?

This is harder than it seems at first. For the left/right animation, our model was the x coordinate of the picture; for the up/down animation, it would be the y coordinate. But if the picture is to move in *both* dimensions, the model needs to "remember" *both* the x and y coordinates; it needs to hold *two numbers at once*.

Before explaining how to do this in Scheme, let me give an analogy. Last week I went to the grocery store. I like grapefruit, so I picked up a grapefruit in my hand. Then another grapefruit in my other hand. Then another, which I sorta cradled in my elbow... and another, and another, and a quart of milk, and a pound of butter. I made my way to the checkout counter, dumped them all on the conveyor belt, paid for them, picked them up, cradling them one by one between my arms, and carried them precariously out to the car.

What's wrong with this picture? Any sensible person would say "don't carry them all individually; *put them in a bag!*" It's easier to carry one bag (which in turn holds five grapefruit, a quart of milk, and a pound of butter) than to carry all those individual items loose.

The same thing happens in computer programming: it's frequently more convenient to *combine several pieces of information in a package* than to deal with them all individually. In particular, if we want an animation to "remember" both an x and a y coordinate (or, as we'll see in the next chapter, *any* two or more pieces of information), we need to package them up into a single object that can be "the model".

Since (x, y) coordinate pairs are so commonly used in computer programming, DrScheme provides a built-in data type named `posn` (short for "position") to represent them. A `posn` can be thought of as a box with two compartments labelled x and y, each of which can hold a number. There are four predefined functions involving `posn`s:

```
; make-posn :  number(x) number(y) -> posn
; posn-x :  posn -> number(x)
; posn-y :  posn -> number(y)
; posn?  :  anything -> boolean
```

To create a `posn`, we call the `make-posn` function, telling it what numbers to put in the x compartment and the y compartment: `(make-posn 7 12)`, for example, creates and returns a `posn` whose x coordinate is 7 and whose y coordinate is 12. For convenience in playing with it, however, we'll store it in a variable. Type the following into the DrScheme Interactions pane:

```
(define where (make-posn 7 12))
(check-expect where (make-posn 7 12))
```

Now we can use the `posn-x` function to retrieve the x coordinate, and `posn-y` to retrieve the y coordinate:

```
(check-expect (posn-x where) 7)
(check-expect (posn-y where) 12)
```

This may not look very exciting — after all, we just *put* 7 and 12 into the x and y compartments, so it's not surprising that we can get 7 and 12 out of them. But in a realistic program, the numbers would come from one place (perhaps the user providing arguments to a function, or clicking a mouse) and be used in a completely different place (such as a redraw handler).

Practice Exercise 22.1.1 ***Create*** *(in the Interactions pane) several variables containing different* `posn`*s. Extract their x and y coordinates and make sure they are what you expected.*

Common pitfalls

I've seen a lot of students write things like

```
(make-posn here)
(posn-x 7)
(posn-y 12)
(do-something-with here)
```

I know exactly what the student was thinking: "First I create a `posn` named `here`, then I say that its x coordinate is 7, and its y coordinate is 12, and then I can use it." Unfortunately, this isn't the way the functions actually work: the `make-posn` function does *not* define a new variable, and the `posn-x` and `posn-y` functions *don't change* the x and y coordinates of "the" `posn`.

To put it another way, the above example doesn't obey the contracts. The `make-posn` function does *not* take in a `posn`, much less a new variable name; it takes in *two numbers*, and *returns* a `posn`. The `posn-x` and `posn-y` functions do *not* take in a number; they take in a `posn` and *return* a number. A correct way to do what this student meant is

```
(define here (make-posn 7 12))
(do-something-with here)
```

or, more simply,

```
(do-something-with (make-posn 7 12))
```

22.2 Definition by parts

In Chapter 16 we learned about "defining a new data type by choices," and in Chapter 20 we saw more examples of definition by choices, of the form "a W is either an X, a Y, or a Z," where X, Y, and Z are previously-defined types. For example, a *key* can be defined as "either a char or a symbol."

Another way to define a new data type from previously-defined types is "definition by parts," and `posns` are our first example. A posn has two *parts*, both of which are numbers (a previously-defined type). In Chapter 23, we'll see more examples of definition by parts.

22.3 Design recipe for functions involving `posns`

Suppose the contract for a function specifies that it takes in a `posn`.

The data analysis (at least for the `posn` parameter) is already done: a `posn` consists of two numbers, x and y. (Although we may have more to say about the numbers themselves, or about other parameters, or about the output type.)

The examples will require creating some `posn`s on which to call the function. There are two common ways to do this: either store the `posn` in a variable, as above, and use the variable name as the function argument, or use a call to `make-posn` as the function argument. Both are useful: the former if you're going to use the same `posn` in several different test cases, and the latter if you're just making up one-shot examples.

```
(define where (make-posn 7 12))
(check-expect (function-on-posn where) ...)
(check-expect (function-on-posn (make-posn 19 5)) ...)
```

The skeleton and inventory will look familiar, with the addition of a few expressions you're likely to need in the body:

```
(define (function-on-posns the-posn)
  ; the-posn            a posn
  ; (posn-x the-posn)    a number (the x coordinate)
  ; (posn-y the-posn)    another number (the y coordinate)
  ...)
```

So here's a complete template for functions taking in a `posn`

```
#| (define where (make-posn 7 12))
(check-expect (function-on-posn where) ...)
(check-expect (function-on-posn (make-posn 19 5)) ...)

(define (function-on-posns the-posn)
  ; the-posn             a posn
  ; (posn-x the-posn)    a number (the x coordinate)
  ; (posn-y the-posn)    another number (the y coordinate)
  ...)
|#
```

In writing the body, you can now use `the-posn` directly, and (more commonly) you can use the expressions `(posn-x the-posn)` and `(posn-y the-posn)` to refer to its individual coordinates.

22.4 Writing functions on posns

So now let's write some actual functions involving `posn`s.

Worked Exercise 22.4.1 ***Develop a function*** *named* `right-of-100?` *which takes in a posn representing a point on the screen, and tells whether it is to the right of the vertical line* $x = 100$*. (For example, we might have a 200-pixel-wide window, and want to do one thing for positions in the right half and something else for positions in the left half.)*

(One might reasonably ask "This function only actually depends on the x coordinate; why does it take in a `posn`?" There are at least two answers. First, sometimes the program *has* a `posn` handy, and doesn't want to take the extra step of extracting the x coordinate from it to pass to `right-of-100?`. Second and more important, what the function depends on is the function's business, not the business of whoever is calling it. I shouldn't have to think about how to solve a problem myself in order to *call* a function whose job is to solve that problem. I should instead give the function whatever information it *might* need, and it will pick out the parts that it *does* need.)

Solution: The contract is

```
; right-of-100?  :  posn -> boolean
```

Data analysis: there's not much to say about the output type, `boolean`, except that it has two values, so we'll need at least two examples. The input type is `posn`, which consists of two numbers x and y. Of these, we're only interested in the x coordinate for this problem; in particular, we're interested in how the x coordinate compares with 100. It could be smaller, greater, or equal, so we'll actually need *three* examples: one with $x < 100$, one with $x = 100$, and one with $x > 100$. Note that although this function doesn't actually *use* the y coordinate, it still has to be there.

```
(check-expect (right-of-100?  (make-posn 75 123)) false)
(check-expect (right-of-100?  (make-posn 100 123)) false)
; borderline case
(check-expect (right-of-100?  (make-posn 102 123)) true)
```

The template gives us most of the skeleton and inventory, to which we add the number 100 because it's hard to imagine solving this problem without it.

```
(define (right-of-100?  the-posn)
  ; the-posn              a posn
  ; (posn-x the-posn)     a number (the x coordinate)
  ; (posn-y the-posn)     another number (the y coordinate)
  ; 100                   a fixed number we'll need
  ...)
```

Body: We don't actually need `(posn-y where)` in this problem, so we can drop it from the inventory. Of the remaining available expressions, there's a `posn` and two numbers. The obvious question to ask is whether one of those numbers (the x coordinate) is larger than the other (100):

```
(> (posn-x where) 100)
```

This expression returns a Boolean, so we could use it in a `cond` to make a decision. . . but this function is supposed to return a Boolean, so a `cond` is probably overkill. In fact, if this expression is true, the right answer from the function is true, and if this expression is false, the right answer is false, so we can just use this expression itself as the body:

```
(define (right-of-100?  where)
  ; where              a posn
  ; (posn-x where)     a number(x)
  ; (pos~~n-y where)    a number(y)~~
  ; 100                a fixed number we know we'll need
  (> (posn-x where) 100)
  )
```

When we test this function on the three examples we wrote earlier, it works.

■

Common pitfalls

Many students think of a `posn` as the same thing as two numbers, so if I had written the `right-of-100?` function above, they would call it in either of the following ways:

```
(right-of-100?  (make-posn 75 112))
(right-of-100?  75 112)
```

In fact, only the former passes a syntax check in Scheme. The `right-of-100?` function defined above expects one parameter of type `posn`, *not* two parameters of type `number`. Try each of the function calls above; before hitting ENTER, try to predict what will happen.

Exercise 22.4.2 ***Develop a function*** *named* `above-diagonal?` *which takes in a* `posn` *representing a point on the screen, and tells whether it's above the diagonal line* $x = y$*.*

Hint: Remember that in computer graphics, positive y-values are usually *down*, so this diagonal line is from the top-left to bottom-right of the window. Pick some specific positions, described in (x, y) coordinates, and decide whether they're above the diagonal or not; then generalize this to a test that tells whether *any* `posn` is above the diagonal (by looking at its x and y coordinates).

Worked Exercise 22.4.3 ***Write a function*** *named* `distance-to-top-left` *which takes in a* `posn` *representing a point on the screen, and computes the straight-line distance from this point to the top-left corner (*i.e. *coordinates* $(0, 0)$*) of the screen, in pixels.*

Hint: The formula for the distance is $\sqrt{x^2 + y^2}$.

Solution: Contract:

```
; distance-to-top-left:  posn -> number
```

Data analysis: we already know what `posn` and `number` mean, and there are no sub-categories of either one to worry about, only arithmetic.

For the examples, we'll start with really easy ones we can do in our heads, then work up to gradually more complicated ones that require a calculator.

```
"Examples of distance-to-top-left:"
(check-within (distance-to-top-left (make-posn 0 0)) 0 .1)
(check-within (distance-to-top-left (make-posn 6 0)) 6 .1)
(check-within (distance-to-top-left (make-posn 0 4.3)) 4.3 .1)
(check-within (distance-to-top-left (make-posn 3 4)) 5 .1)
```
; $3^2 + 4^2 = 9 + 16 = 25 = 5^2$
```
(check-within (distance-to-top-left (make-posn 4 7)) 8.1 .1)
```
; $4^2 + 7^2 = 16 + 49 = 65 > 8^2$

Skeleton and inventory (from the template):

```
(define (distance-to-top-left the-point)
  ; the-point               a posn
  ; (posn-x the-point)      a number (x)
  ; (posn-y the-point)      a number (y)
  ...)
```

Body: We have two numeric expressions, `(posn-x the-point)` and `(posn-y the-point)`, which represent the x and y coordinates respectively. We need to square each of them:

```
(define (distance-to-top-left the-point)
  ; the-point              a posn
  ; (posn-x the-point)     a number (x)
  ; (posn-y the-point)     a number (y)
  ; (* (posn-x the-point) (posn-x the-point))    a number (x^2)
  ; (* (posn-y the-point) (posn-y the-point))    a number (y^2)
  ...)
```

Note that there's getting to be a fuzzy line between inventory and body: we've added these expressions in comments, because they're not the final body but we know they're a step along the way.

Then we need to add those two squares:

```
(define (distance-to-top-left the-point)
  ; the-point              a posn
  ; (posn-x the-point)     a number (x)
  ; (posn-y the-point)     a number (y)
  ; (* (posn-x the-point) (posn-x the-point))    a number (x^2)
  ; (* (posn-y the-point) (posn-y the-point))    a number (y^2)
  ; (+ (* (posn-x the-point) (posn-x the-point))
  ;    (* (posn-y the-point) (posn-y the-point)))
  ;           a number (x^2 + y^2)
  ...)
```

and finally square-root that, using `sqrt`:

```
(define (distance-to-top-left the-point)
  ; the-point              a posn
  ; (posn-x the-point)     a number (x)
  ; (posn-y the-point)     a number (y)
  ; (* (posn-x the-point) (posn-x the-point))    a number (x^2)
  ; (* (posn-y the-point) (posn-y the-point))    a number (y^2)
  ; (+ (* (posn-y the-point) (posn-y the-point))
  ;    (* (posn-y the-point) (posn-y the-point)))
  ;           a number (x^2 + y^2)
  (sqrt (+ (* (posn-x the-point) (posn-x the-point))
           (* (posn-y the-point) (posn-y the-point))))
  )
```

We can now test this on the examples we wrote earlier, and it should work.

█

Exercise 22.4.4 ***Develop a function*** *named* `coordinate-difference` *which takes in a* `posn` *and gives back the* difference *between the coordinates (which tells you, in a sense, how far the point is from the diagonal line $x = y$).*

Hint: The answer should never be negative, so use the built-in `abs` (absolute-value) function to ensure this.

Exercise 22.4.5 ***Develop a function*** *named* `distance` *which takes in* two `posns` *(call them* `here` *and* `there`*), and computes the straight-line distance between them. The formula is*

$$\sqrt{(x_{here} - x_{there})^2 + (y_{here} - y_{there})^2)}$$

Hint: Since your function will have two parameters `here` and `there`, both of which are `posn`s, the skeleton will include

```
; here              a posn
; there             a posn
; (posn-x here)     a number(x coordinate of here)
; (posn-y here)     a number(y coordinate of here)
; (posn-x there)    a number(x coordinate of there)
; (posn-y there)    a number(y coordinate of there)
```

Exercise 22.4.6 ***Develop a function*** *named* `posn=?` *which takes in two* `posns` *and tells whether they're the same (*i.e. *they have the same x coordinate and the same y coordinate).*

Hint: Be sure your examples include two posns that are the same, two that differ only in x, two that differ only in y, and two that differ in both x and y coordinates.

Exercise 22.4.7 ***Develop a function*** *named* `distance-to-origin` *which takes in* either *a number* or *a* `posn` *and tells how far it is from the appropriate "origin". For numbers, that's 0; for* `posns`*, that's* `(make-posn 0 0)`*.*

22.5 Functions that return posns

Since `posn` is a data type, like `number`, `image`, *etc.*, you can write functions that *return* a `posn` too. Such functions will almost always use `make-posn` somewhere in the body. In other words, the output template for `posn` looks like this:

```
#|
(check-expect (function-returning-posn ...)  (make-posn 3 8))
...

(define (function-returning-posn ...)
  (make-posn ......)
  )
|#
```

Worked Exercise 22.5.1 ***Develop a function*** *named* `diagonal-point` *which takes in a number and returns a* `posn` *whose x and y coordinate are* both *that number.*

Solution: Contract:

```
; diagonal-point :  number -> posn
```

Data analysis: the input is a number, about which there's not much to say. The output is a `posn`, which has two numeric parts x and y.

Examples:

```
(check-expect (diagonal-point 0) (make-posn 0 0))
(check-expect (diagonal-point 3.7) (make-posn 3.7 3.7))
```

Skeleton/inventory (from the output template for `posn`):

```
(define (diagonal-point coord)
  ; coord    a number
  (make-posn ......)
  )
```

At this point we'll apply the "inventory with values" technique.

```
(define (diagonal-point coord)
  ; coord        a number    3.7
  ; should be    a posn      (make-posn 3.7 3.7)
  (make-posn ......)
  )
```

Body: The "inventory with values" makes this really easy: the only reasonable way we can get `(make-posn 3.7 3.7)` from a parameter `coord` with the value 3.7 is `(make-posn coord coord)`, so that becomes the body:

```
(define (diagonal-point coord)
  ; coord        a number    3.7
  ; should be    a posn      (make-posn 3.7 3.7)
  (make-posn coord coord)
  )
```

We run the test cases on this definition, and it works. ∎

The "inventory with values" technique tends to be more useful the more complicated the function's *result type* is. It doesn't really help when the result type is Boolean, it helps a little when the result type is a number, even more when the result type is a string or an image, and it's *extremely* helpful for functions that return a `posn` or the other complex data types we'll see in the next few chapters.

Exercise 22.5.2 ***Develop a function*** *named* `swap-x-y` *which takes in a* `posn` *and returns a new* `posn` *with the coordinates swapped: the x coordinate of the output should be the y coordinate of the input, and* vice versa*.*

Hint: This function both takes in and returns a `posn`, but they're not the same `posn`, so you'll need to use both the input and output templates for `posn`.

Exercise 22.5.3 ***Develop a function*** *named* `scale-posn` *which takes in a number and a* `posn`*, and returns a* `posn` *formed by multiplying the number by each of the coordinates of the input* `posn`*.*

For example,

```
(check-expect (scale-posn 3 (make-posn 2 5)) (make-posn 6 15))
```

Exercise 22.5.4 ***Develop a function*** *named* `add-posns` *which takes in two* `posns` *and returns a new* `posn` *whose x coordinate is the sum of the x coordinates of the two inputs, and whose y coordinate is the sum of the y coordinates of the two inputs.*

Exercise 22.5.5 ***Develop a function*** *named* `sub-posns` *which takes in two* `posns` *and returns a new* `posn` *whose x coordinate is the difference of the x coordinates of the two inputs, and whose y coordinate is the difference of the y coordinates of the two inputs.*

Exercise 22.5.6 ***Redefine*** *the* `distance` *function from Exercise 22.4.5 to be much shorter and simpler, by re-using functions you've already seen or written in this chapter.*

Hint: You should be able to do this in two fairly short lines of Scheme code.

Exercise 22.5.7 ***Develop a function*** *named* `choose-posn` *that takes in a string and two posns. The string should be either* `"first"` *or* `"second"`*. The* `choose-posn` *function should return either the first or the second of its two* `posns`*, as directed by the string.*

Hint: Although this function returns a `posn`, it can be written *without* using `make-posn` (except for the examples); indeed, it's much shorter, simpler, and easier without using `make-posn`. This situation doesn't happen often, but it does happen, so don't use `make-posn` blindly.

22.6 Writing animations involving `posns`

Now we can finally solve the problem that started this chapter.

Worked Exercise 22.6.1 ***Write an animation*** *of a picture that moves up, down, left, and right in response to the* `'up`*,* `'down`*,* `'left`*, and* `'right` *arrow keys. It should ignore all other keys.*

Solution: The model has to represent *both* the x and y coordinates of the object, so we'll use a `posn`. Since the model isn't an image, we'll need a redraw handler with contract

```
; show-picture :  posn -> image
```

and we'll obviously need a key handler with contract

```
; handle-key :  posn symbol-or-char -> posn
```

Let's do the `show-picture` function first. We have its contract already, and there's not much to say about the data types.

```
(define WIDTH 300)
(define HEIGHT 300)
(define BACKGROUND (empty-scene WIDTH HEIGHT))
(define DOT (circle 3 "solid" "blue"))
...
"Examples of show-picture:"
(check-expect (show-picture (make-posn 15 12))
  (place-image DOT 15 12 BACKGROUND))
(check-expect (show-picture (make-posn 27 149))
  (place-image DOT 27 149 BACKGROUND))
```

The skeleton and inventory are similar to those we've seen before involving `posn`s:

```
(define (show-picture where)
  ; where a posn
  ; (posn-x where)    a number(x)
  ; (posn-y where)    a number(y)
  ; DOT               a fixed image (to be placed)
  ; BACKGROUND        a fixed image (to use as background)
  ...)
```

Now let's try the "inventory with values" technique, using the "moderately complicated" example of `(make-posn 27 149)`.

```
(define (show-picture where)
  ; where          a posn        (make-posn 27 149)
  ; (posn-x where) a number(x)   27
  ; (posn-y where) a number(y)   149
  ; DOT            a fixed image (to be placed)
  ; BACKGROUND     a fixed image (to use as background)
  ; should be      an image    (place-image DOT 27 149 BACKGROUND)
  ...)
```

This makes the body pretty easy:

```
(define (show-picture where)
  ; where          a posn        (make-posn 27 149)
  ; (posn-x where) a number(x)   27
  ; (posn-y where) a number(y)   149
  ; DOT            a fixed image (to be placed)
  ; BACKGROUND     a fixed image (to use as background)
  ; should be      an image    (place-image DOT 27 149 BACKGROUND)
  (place-image DOT
               (posn-x where) (posn-y where)
               BACKGROUND)
  )
```

We can test this on the known examples, and it works.

Now for the key handler. Recall that the contract is

```
; handle-key :  posn symbol-or-char -> posn
```

Since the second parameter is "either a symbol or a char", we'll need at least two examples: a symbol and a char. In fact, if it's a symbol, there are four specific symbols we want to recognize — `'left`, `'right`, `'up`, and `'down` — plus "any other symbol," which we'll ignore.

```
"Examples of handle-key:"
(check-expect (handle-key (make-posn 12 19) #\e)
              (make-posn 12 19))
; ignore #\e by returning the same model we were given
(check-expect (handle-key (make-posn 12 19) 'left)
              (make-posn 11 19))
; move left by decreasing the x coordinate
(check-expect (handle-key (make-posn 12 19) 'right)
              (make-posn 13 19))
(check-expect (handle-key (make-posn 12 19) 'up)
              (make-posn 12 18))
; remember that positive y-values are down
(check-expect (handle-key (make-posn 12 19) 'down)
              (make-posn 12 20))
(check-expect (handle-key (make-posn 12 19) 'home)
              (make-posn 12 19))
; ignore special keys other than the four arrows
```

The skeleton is easy. The inventory will show the expressions we have available (based on the data type `posn`):

```
  (define (handle-key where key)
    ; where             a posn
    ; key               a symbol or char
    ; (posn-x where)    a number(x)
    ; (posn-y where)    a number(y)
    ...)
```

There are four specific values of `key` that we care about: `'up`, `'down`, `'left`, and `'right`. So we'll need a conditional with five cases: one for each of these, and one for "anything else" (which includes ordinary characters as well as other special keys).

```
(define (handle-key where key)
  ; where             a posn
  ; key               a symbol or char
  ; (posn-x where)    a number(x)
  ; (posn-y where)    a number(y)
  (cond [(key=?  key 'up)    ...]
        [(key=?  key 'down)  ...]
        [(key=?  key 'left)  ...]
        [(key=?  key 'right) ...]
        [else                ...]
        )
  ...)
```

We still need to fill in the answers. In the "ignore" case, we can simply return `where` unchanged:

```
(define (handle-key where key)
  ; where             a posn
  ; key               a symbol or char
  ; (posn-x where)    a number(x)
  ; (posn-y where)    a number(y)
  (cond [(key=?  key 'up)    ...]
        [(key=?  key 'down)  ...]
        [(key=?  key 'left)  ...]
        [(key=?  key 'right) ...]
        [else                where ]
  ))
```

The other four cases all require producing a `posn` that's similar to `where`, but moved slightly in either the x or the y dimension. The formulæ for these may be obvious to you, but in case they're not, let's try an "inventory with values" for each case.

```
(define (handle-key where key)
  ; where              a posn         (make-posn 12 19)
  ; key                a symbol or char
  ; (posn-x where)     a number(x)    12
  ; (posn-y where)     a number(y)    19
  (cond [ (key=?  key 'up)
          ; should be         (make-posn 12 18)
          ]
        [ (key=?  key 'down)
          ; should be         (make-posn 12 20)
          ]
        [ (key=?  key 'left)
          ; should be         (make-posn 11 19)
          ]
        [ (key=?  key 'right)
          ; should be         (make-posn 11 21)
          ]
        [ else                where ]
        ))
```

From these “right answers”, it’s pretty easy to write the formulæ using `make-posn`:

```
  (cond [ (key=?  key 'up)
          ; should be         (make-posn 12 18)
          (make-posn (posn-x where) (- (posn-y where) 1))]
        [ (key=?  key 'down)
          ; should be         (make-posn 12 20)
          (make-posn (posn-x where) (+ (posn-y where) 1))]
        [ (key=?  key 'left)
          ; should be         (make-posn 11 19)
          (make-posn (- (posn-x where) 1) (posn-y where))]
        [ (key=?  key 'right)
          ; should be         (make-posn 11 21)
          (make-posn (+ (posn-x where) 1) (posn-y where))]
        [ else                where ]
        ))
```

Alternatively, we could realize that moving up, moving down, moving left, and moving right can *all* be thought of as the same problem: *adding* something to *both* dimensions of the `posn`, and we’ve already written a function to do that, in Exercise 22.5.4. So assuming you’ve done that exercise, we can solve the problem as follows:

```
(cond [ (key=?  key 'up)
        ; should be         (make-posn 12 18)
        (add-posns where (make-posn 0 -1))]
      [ (key=?  key 'down)
        ; should be         (make-posn 12 20)
        (add-posns where (make-posn 0 1))]
      [ (key=?  key 'left)
        ; should be         (make-posn 11 19)
        (add-posns where (make-posn -1 0))]
      [ (key=?  key 'right)
        ; should be         (make-posn 11 21)
        (add-posns where (make-posn 1 0))]
      [ else                where ]
      ))
```

which is shorter and clearer.

In either case, after testing this, we can put together the animation:

```
(run-animation WIDTH HEIGHT
               (make-posn (/ WIDTH 2) (/ HEIGHT 2)) 1
               (on-redraw show-picture)
               (on-key handle-key))
```

█

Exercise 22.6.2 *You may notice that four of the five cases in the final version of the definition share the pattern*

```
(add-posns where some-posn)
```

Even the remaining example could be fit into this pattern by adding `(make-posn 0 0)`. This common pattern suggests that the function definition could be simplified by "factoring out" the `add-posns`, moving it outside the `cond` so the `cond` decides only what to use as the second argument to `add-posns`. Try this.

Exercise 22.6.3 ***Develop an animation*** *of a dot that jumps randomly around the window: every half second, it disappears from where it was and appears at a completely random location with* $0 \leq x \leq$ *WIDTH and* $0 \leq y \leq$ *HEIGHT.*

Hint: This is easier than Exercise 22.6.1, since you don't need to worry about what key was pressed.

Hint: Use a `posn` as the model. You *can* get this to work with an image as the model, but that'll make it difficult to modify for Exercise 22.6.4.

Exercise 22.6.4 ***Modify Exercise 22.6.3*** *so that if the user clicks the mouse on the dot (*i.e. *within a distance of 3 from its current center), the animation ends with the message "Congratulations!" This forms a sort of video-game, which will get harder if you shorten the time between ticks.*

The following five exercises list several fun features to add to these animations. They're independent of one another; you can do any or all of them, in whatever order you wish.

Exercise 22.6.5 ***Modify Exercise 22.6.1 or 22.6.3*** *so that if the user types the letter "q", the animation ends.*

Exercise 22.6.6 ***Modify Exercise 22.6.1 or 22.6.3*** *so that whenever the user clicks the mouse, the dot jumps immediately to the mouse location .*

Exercise 22.6.7 ***Modify Exercise 22.6.1 or 22.6.3*** *so that the display is a green dot if it's within 50 pixels from the center of the window (*i.e. *`(make-posn (/ WIDTH 2) (/ HEIGHT 2))`, and a red dot if it's farther away.*

Hint: re-use a function we've seen earlier in this chapter.

Exercise 22.6.8 ***Modify Exercise 22.6.1*** *so that in addition to responding to arrow keys, the dot moves slowly and randomly around the screen every half second: with equal probability, it moves up one pixel, down one pixel, left one pixel, or right one pixel.*

Hint: You'll obviously need to use `random`. Since all four random choices result in adding something to the current posn, you could write a helper function `choose-offset` that takes in a number (either 0, 1, 2, or 3) and returns the appropriate posn to add. Alternatively, you could write a function `random-offset` that takes in a dummy parameter, ignores it, picks a random number (either 0, 1, 2, or 3), and returns the appropriate posn to add. The latter approach is easier to use, but harder to test.

Exercise 22.6.9 ***Modify Exercise 22.6.1*** *so that if the dot reaches an edge of the window, it "wraps around". That is, if it's at x coordinate 0, and tries to move left, its x coordinate becomes `WIDTH`; if it's at x coordinate `WIDTH` and tries to move right, its x coordinate becomes 0. Likewise, if the y coordinate is 0 and it tries to move up, the y coordinate becomes `HEIGHT`, while if the y coordinate is `HEIGHT` and the dot tries to move down, it jumps to y coordinate 0.*

Hint: It may be easiest to just move the posn, without worrying about whether it's outside the window, and then call a helper function that takes in the "attempted" position of the dot and returns a "corrected" position with $0 \leq x \leq$ `WIDTH` and $0 \leq y \leq$ `HEIGHT`.

22.7 Review

Sometimes an animation (or other kind of program) needs to store *several* pieces of data together in a "package". DrScheme has a predefined data type `posn` to represent (x,y) coordinate pairs, perhaps the most common example of this situation. There are several predefined functions — `make-posn`, `posn-x`, `posn-y`, `posn?` — that work with `posn`s. When writing a function that takes in a `posn`, the inventory should list not only the parameter itself but the *x* and *y parts* of the parameter.

The "inventory with values" technique is especially helpful for functions with a complicated return type like `posn`, the other structures in the next chapter, lists, *etc.*)

One can also write functions that *return* a `posn`, typically (though not always) using `make-posn` inside the body of the function.

An animation can use a `posn` as its model; this gives you a great deal more power to write fun animations that move around the screen.

Chapter 23

Inventing new structures

23.1 Why and how

Chapter 22 showed how to store two numbers — an x coordinate and a y coordinate — in a single variable. This enabled us to write animations that "remember" a two-dimensional position, and can change either or both of the coordinates.

But what if you have *more* than two pieces of information to remember? Or what if one of them isn't a number? The `posn` data type won't help you much in those situations.

Let's review what a `posn` is, then see how to generalize the idea.

- A `posn` is a package containing two "parts" (also known as *fields* or *instance variables*) named `x` and `y`, each of which is a number.
- `posn` itself is a data type (like `number` or `image`), but there may be many *instances* of this data type: 2/3, 5, and -72541 are all instances of `number`, while `(make-posn 3 4)` and `(make-posn 92 -3/4)` are both instances of `posn`.
- There's a built-in function named `make-posn` that takes in two numbers and puts them together into a `posn` package. (Computer scientists call this a *constructor*.)
- There are two built-in functions named `posn-x` and `posn-y` that pull out the individual numbers from such a package. (Computer scientists call these *getters*.)
- There's a built-in function named `posn?` that takes in *any* Scheme object and tells whether or not it is a `posn`. (Computer scientists call this a *discriminator*.)

If we were trying to represent something other than a two-dimensional coordinate pair, we might need more fields, and they might have different names and types. We would still need a "constructor" function that takes in the values of the parts and puts them together into a package. We would still need several "getter" functions (one for each "part") that retrieve the individual parts from a package. And we would still need a "discriminator" function which tells us whether a given object is this kind of package at all.

Scheme provides a way to define other data types analogous to `posn`, with fields, constructor, getters, and discriminator. Here's the syntax rule:

Syntax Rule 7 *Anything matching the pattern*

```
(define-struct struct-name (field-name-1 ... field-name-n))
```

is a legal expression, as long as struct-name *is a previously undefined name. (The* field-name*s may or may not already be defined elsewhere; it doesn't matter.)*

The expression has no value, but the side effect of defining a new data type struct-name *and several functions with contracts*

```
; make-struct-name :  n objects -> struct-name
; struct-name-field-name-1 :  struct-name -> object
; ...
; struct-name-field-name-n :  struct-name -> object
; struct-name? :  object -> boolean
```

There's a lot going on in there, so let's see how it applies to the one `struct` we've already seen — a `posn`. This type happens to be predefined in the HtDP languages of DrScheme, but if it weren't, we could define it ourselves as follows:

```
(define-struct posn (x y))
```

The *struct-name* is `posn`. There are two fields, named `x` and `y`. So we've defined a new data type named `posn`, as well as the following functions:

```
; make-posn :  object(x) object(y) -> posn
; posn-x :  posn -> object
; posn-y :  posn -> object
; posn?  :  object -> boolean
```

which (mostly) agrees with what we learned in the previous chapter.

There's one difference between these contracts and those you learned in Chapter 22: the "parts" of a `posn` here are just "objects", rather than specifically numbers. In fact, you *can* build a posn whose "*x* coordinate" is a string and whose "*y* coordinate" is an image, and you won't get any error messages — but as soon as you try to *use* that `posn` in a function that expects the coordinates to be numbers, it'll crash. To avoid this, we agree to follow the *convention* that the coordinates in a `posn` are always numbers, so in practice the contracts really are

```
; make-posn :  number(x) number(y) -> posn
; posn-x :  posn -> number
; posn-y :  posn -> number
; posn?  :  object -> boolean
```

exactly as we learned in the previous chapter.

Worked Exercise 23.1.1 *Define a structure to represent a person, with first and last names and age.*

Solution: The structure has three parts, which can naturally be called *first*, *last*, and *age*. We'll agree to the convention that *first* and *last* are both strings, while *age* is a number. So the struct definition looks like

```
(define-struct person (first last age))
```

This has the effect of defining a new data type `person`, along with the functions

```
; make-person :  string(first) string(last) number(age) -> person
; person-first :  person -> string
; person-last :  person -> string
; person-age :  person -> number
; person?  :  object -> boolean
```

To see that this definition actually works, we put the `define-struct` line (and, ideally, the comments about function contracts) in the definitions pane, hit "Run", and we can now use the `person` type as follows:

```
>(make-person "Joe "Schmoe" 19)
(make-person "Joe" "Schmoe" 19)
>(define author (make-person "Stephen" "Bloch" 45))
>(define lambda-guy (make-person "Alonzo" "Church" 106))
>(person-first author)
"Stephen"
>(person-last author)
"Bloch"
>(person-last lambda-guy)
"Church"
>(person-first lambda-guy)
"Alonzo"
>(person-first (make-person "Joe" "Schmoe" 19))
"Joe"
>(person-age lambda-guy)
106
>(person?  author)
true
>(person?  "Bloch")
false
>(person?  (make-person "Joe" "Schmoe" 19))
true
```

Figure 23.1: Design recipe for defining a struct

1. **Identify the parts** of the desired data types: how many parts should it have, and what are their names and their types?
2. **Write a `define-struct`** according to Syntax Rule 7.
3. Write down (in comments) the **contracts** for the functions that "come for free":
   - a constructor, whose name is `make-` followed by the name of the struct;
   - several getters (one for each field) whose names are the name of the struct, a hyphen, and the name of one of the fields;
   - a discriminator whose name is the name of the struct, followed by a question mark.
4. Write some **examples** of objects of the new data type.
5. Write input and output templates for functions that work on the new type.

SIDEBAR:

Alonzo Church (1903-1995) invented a model of computation called the "lambda calculus" (no relation to the "calculus" that's about derivatives and integrals) which later became the inspiration for the Lisp and Scheme languages. This is why there's a Greek letter lambda (λ) in the DrScheme logo; we'll learn more about lambda in Chapter 33. Alonzo Church was also my Ph.D. advisor's Ph.D. advisor's Ph.D. advisor. So there.

Note that you don't need to *define* `make-person`, `person-first`, `person-last`, `person-age`, or `person?`; they "come for free" with `define-struct`. We wrote down their contracts only so we would know how to *use* them. ∎

23.2 A Recipe for Defining a Struct

Back in Chapter 5, we learned a step-by-step recipe for defining a function, and in Chapter 10 we learned a step-by-step recipe for writing an animation. A step-by-step recipe for defining a struct is in Figure 23.1.

Worked Exercise 23.2.1 *Define a data type to represent an employee of a business, including the employee's name (we won't bother with first and last names), ID number, and salary.*

Solution:

Identify the parts

```
; An employee has three parts:  name, id, and salary.
; The name is a string, while id and salary are numbers.
```

Write a `define-struct`

```
(define-struct employee (name id salary))
```

Write contracts for the functions that “come for free”

```
; make-employee:
  string(name) number(id) number(salary) -> employee
; employee-name:  employee -> string
; employee-id:  employee -> number
; employee-salary:  employee -> number
; employee?:  object -> boolean
```

Write examples of the new data type

```
(make-employee "Joe" 348 42995)
(make-employee "Mary" 214 49500)
(define emp1 (make-employee "Bob" 470 36000))
(define emp2 (make-employee "Chris" 471 41000))
(check-expect (employee-name emp1) "Bob")
(check-expect (employee-id emp2) 471)
(check-expect (employee-salary emp2) 41000)
(check-expect (employee-salary (make-employee "Mary" 214 49500))
              49500)
(check-expect (employee?  emp1) true)
(check-expect (employee?  "Mary") false)
```

Write templates
The input template is

```
#|
(check-expect (function-on-employee emp1) ...)
(check-expect (function-on-employee
                   (make-employee "Joe" 348 42995))
              ...)

(define (function-on-employee emp)
  ; emp                   an employee
  ; (employee-name emp)   a string
  ; (employee-id emp)     a number
  ; (employee-salary emp) a number
  ...)
|#
```

and the output template

```
#|
(check-expect (function-returning-employee ...)  emp1)
(check-expect (function-returning-employee ...)
              (make-employee "Joe" 348 42995))

(define (function-returning-employee ...)
  (make-employee ... ... ...)
  )
|#
```

23.3 Exercises and Common Pitfalls

Students often get confused between `define-struct` and `make-person` (and other constructors like `make-cd` and `make-employee`).

By way of analogy, imagine an inventor who has invented a new kind of cell phone. The inventor probably doesn't actually build cell phones herself; instead, she produces *blueprints*, *diagrams*, *etc.* for how the new kind of cell phone is supposed to go together. Based on these blueprints and diagrams, somebody builds a *factory* which then builds millions of individual cell phones.

In our setting, `define-struct` is like the inventor. The `make-person` and `make-cd` functions are like factories: they don't even exist until the inventor has done her work, but then they can be used to build as many instances of `person` or `cd` respectively as you wish.

I often also see students write things like

```
(define-struct employee (name id salary))
(define emp1 (make-employee "Bob" 470 36000))
(check-expect emp1-salary 36000)
(check-expect (emp1-salary employee) 36000)
```

There is no variable or function named `emp1-salary`, nor is there a variable named `employee`, so the last two lines both produce error messages. But there *is* a function named `employee-salary`, which *takes in* an `employee` object; the student probably meant

```
(check-expect (employee-salary emp1) 36000)
```

Another pitfall: the same student writes

```
(check-expect (employee-salary "Bob") 36000)
```

What's wrong with this? Well, there *is* a function named `employee-salary`, but its contract specifies that it takes in an `employee`, not a string. What this student is trying to do is *look up* a previously-defined employee by one of its field values; we'll learn how to do this in Chapter **??**.

Exercise 23.3.1 *Define a data type to represent a CD in your audio library, including such information as the title, performer, what year it was recorded, and how many tracks it has.*

Exercise 23.3.2 *Define a data type to represent a candidate in an election. There should be two fields: the candidate's name and how many votes (s)he got.*

Exercise 23.3.3 *Define a data type to represent a course at your school, including the name of the course, the name of the instructor, what room it meets in, and what time it meets. (For now, assume all courses start on the hour, so you only need to know what hour the course starts.)*

Hint: You'll need to decide whether a "room" is best represented as a number or a string.

Exercise 23.3.4 *Define a data type to represent a basketball player, including the player's name, what team (s)he plays for, and his/her jersey number.*

Exercise 23.3.5 *Define a data type to represent a dog (or a cat if you prefer), with a name, age, weight, and color.*

Exercise 23.3.6 *Define a data type to represent a mathematical rectangle, whose properties are length and width.*

Hint: There's already a function named `rectangle`, so if you try to write `(define-struct rectangle ...)`, you'll probably get an error message. Name your struct `rect` instead.

Hint: This data type has *nothing to do with images.* A `rect` has no color, it is not outlined or solid, it has no position, *etc.*; it has *only* a length and a width.

Exercise 23.3.7 *Define a data type to represent a time of day, in hours, minutes, and seconds. (Assume a 24-hour clock, so 3:52:14 PM would have hours=15, minutes=52, seconds=14.)*

23.4 Writing functions on user-defined structs

Writing functions using a struct you've defined yourself is no more difficult than writing functions using `posn`s.

Worked Exercise 23.4.1 *Define a function that takes in an* `employee` *(from Exercise 23.2.1) and tells whether or not the employee earns over $100,000 per year.*

Solution: Before you type any of this stuff, make sure you've got the definition of the `employee` data type, and perhaps its examples, in the definitions pane. The following stuff should all appear *after* that definition.

Contract:

```
  ; earns-over-100k?  :  employee -> boolean
```

Examples:

```
(check-expect
  (earns-over-100k?  (make-employee "Phil" 27 119999)) true)
(check-expect
  (earns-over-100k?  (make-employee "Anne" 51 100000))
  false ; (borderline case)
(check-expect (earns-over-100k?  emp1) false)
  ; assuming the definition of emp1 from before
```

Skeleton:

```
  (define (earns-over-100k?  emp)
    ...)
```

Inventory:

```
  (define (earns-over-100k?  emp)
    ; emp                    employee
    ; (employee-name emp)    string
    ; (employee-id emp)      number
    ; (employee-salary emp)  number
    ; 100000                 fixed number
    ...)
```

Body:
We don't actually need the employee name or id, only the salary.

```
(define (earns-over-100k?  emp)
  ; emp                    employee
  ; (employee-name emp)    string
  ; (employee-id emp)      number
  ; (employee-salary emp)  number
  ; 100000                 fixed number
  (> (employee-salary emp) 100000)
  )
```

Testing:
Hit "Run" and see whether the actual answers match what you said they "should be". ∎

Exercise 23.4.2 *Develop a function `rec-before-1980?` which takes in a CD and returns `true` or `false` depending on whether it was recorded before 1980.*

Exercise 23.4.3 *Develop a function `older?` which takes in two `person` structs and tells whether the first is older than the second.*

Exercise 23.4.4 *Develop a function `same-team?` which takes in two basketball-player structs and tells whether they play for the same team.*

Exercise 23.4.5 *Develop a function `full-name` which takes in a `person` struct and returns a single string containing the person's first and last names, separated by a space.*

Exercise 23.4.6 *Develop a function `rect-area` which takes in a `rect` struct and returns the area of the rectangle (*i.e.* length times width).*

Exercise 23.4.7 *Develop a function `larger-rect?` which takes in two `rect` structs and tells whether the first has a larger area than the second.*

Hint: Copying the input template for the `rect` structure will take care of *one* of the two parameters; for the other, you'll need to copy the inventory again and change the parameter name.

Exercise 23.4.8 *Develop a function `secs-since-midnight` which takes in a time-of-day struct and returns how many seconds it has been since midnight.*

Exercise 23.4.9 *Develop a function `secs-between` which takes in two time-of-day structs and returns the difference between them, in seconds.*

Hint: For example, the time 11:01:14 is 124 seconds after the time 10:59:10.

Exercise 23.4.10 *Develop a function named* `who-won` *which takes in three candidate structures (from Exercise 23.3.2) and returns the name of the one with the most votes, or the word "tie" if two or more of them tied for first place.*

Note: Obviously, this resembles Exercise 15.5.4, but it doesn't assume that the candidates' names are always "Anne", "Bob", and "Charlie"; it'll work with any *names.*

23.5 Functions returning user-defined structs

Just as you can write a function to return a `posn`, you can also write a function that returns a `name`, `cd`, `employee`, or any other type you've defined. As in Section 22.5, you'll usually (but not always!) need a `make-`whatever in the body of your function. Use the output template.

Worked Exercise 23.5.1 *Define a function* `change-salary` *which takes in an employee (from Exercise 23.2.1) and a number, and produces a new employee just like the old one but with the salary changed to the specified number.*

Solution:

Contract:

```
; change-salary :  employee number -> employee
```

Examples:

```
(check-expect
  (change-salary (make-employee "Joe" 352 65000) 66000)
  (make-employee "Joe" 352 66000))
(check-expect
  (change-salary (make-employee "Croesus" 2 197000) 1.49)
  (make-employee "Croesus" 2 1.49))
```

Skeleton and Inventory
Since this function both takes in *and* returns an `employee`, we can use both the input and output templates to help us write it.

```
(define (change-salary emp new-salary)
  ; emp                      employee
  ; (employee-name emp)      string
  ; (employee-id emp)        number
  ; (employee-salary emp)    number
  ; new-salary               number
  (make-employee ... ... ...))
```

Since this function returns something of a complex data type, we'll use an **inventory with values**:

```
(define (change-salary emp new-salary)
  ; emp                    employee (make-employee "Joe" 352 65000)
  ; (employee-name emp)    string   "Joe"
  ; (employee-id emp)      number   352
  ; (employee-salary emp) number   65000
  ; new-salary             number   66000
  ; right answer           employee (make-employee "Joe" 352 66000)
  (make-employee ... ... ...))
```

This makes the **Body** fairly obvious:

```
(define (change-salary emp new-salary)
  ; emp                    employee (make-employee "Joe" 352 65000)
  ; (employee-name emp)    string   "Joe"
  ; (employee-id emp)      number   352
  ; (employee-salary emp) number   65000
  ; new-salary             number   66000
  ; right answer           employee (make-employee "Joe" 352 66000)
  (make-employee (employee-name emp)
                 (employee-id emp)
                 new-salary)
  )
```

Now test the function and see whether it works correctly on both examples.

■

Exercise 23.5.2 *Develop a function* `change-jersey` *which takes in a basketball player struct and a number and produces a basketball player with the same name and team as before, but the specified jersey number.*

Exercise 23.5.3 *Develop a function* `birthday` *which takes in a person struct and returns a person with the same first and last name, but one year older.*

Exercise 23.5.4 *Develop a function* `change-name-to-match` *which takes in two person structs and returns a person just like the first one, but with the last name changed to match the second one.*

Exercise 23.5.5 *Develop a function* `raise-salary-percent` *which takes in an employee structure and a number, and produces a copy of the employee with the specified percentage increase in salary.*

Exercise 23.5.6 *Develop a function* `add-a-vote` *which takes in a candidate structure and adds one to his/her vote count.*

Exercise 23.5.7 *Develop a function* `swap-length-width` *which takes in a* `rect` *structure and produces a new* `rect` *whose length is the width of the given* `rect`*, and* vice versa*.*

23.6 Animations using user-defined structs

Worked Exercise 23.6.1 *Write an animation of a picture that moves steadily to the right or left, say 3 pixels per second; if the user presses the right-arrow key, the picture starts moving to the right, and if the user presses the left-arrow key, the picture starts moving to the left.*

Solution:

Model
Since the picture only needs to move left and right, we need only the x coordinate of its location (we'll probably want to define a named constant for its y coordinate). However, we also need to keep track of which *direction* it's moving — left or right — so that a tick handler can move it in the appropriate direction every second. One way to do that is with a symbol which will always be either `'left` or `'right`. So our model needs to have two fields, which we can call `x` (a number) and `dir` (a symbol). We'll name such a data structure a `moving-x`.

Combining this English-language description with a `define-struct`, we get

```
; A moving-x consists of x (a number) and
;   dir (a symbol, either 'left or 'right)

(define-struct moving-x (x dir))
```

which gives us the following functions "for free":

```
; make-moving-x :  number symbol -> moving-x
; moving-x-x :  moving-x -> number
; moving-x-dir :  moving-x -> symbol
; moving-x?  :  object -> boolean
```

Some examples of the new data type:

```
(define state1 (make-moving-x 10 'right))
(define state2 (make-moving-x 29 'left))
(check-expect (moving-x-x state1) 10)
(check-expect (moving-x-dir state2) 'left)
```

An input template:

```
#|
  ; (moving-x-dir current) symbol
(define (function-on-moving-x current)
  ; current                moving-x
  ; (moving-x-x current)   number
  ; (moving-x-dir current) symbol
  ...)
|#
```

And an output template:

```
#|
(define (function-returning-moving-x current)
  (make-moving-x ... ...))
|#
```

Handlers and their contracts

Since the model isn't an image, we'll need a redraw handler with contract

```
; handle-redraw :  moving-x -> scene
```

Since we're doing something every second, we'll need a tick handler with contract

```
; handle-tick :  moving-x -> moving-x
```

And since we need to respond to key presses, we'll need a key handler with contract

```
; handle-key :  moving-x key -> moving-x
```

Writing the redraw handler

We already have a contract. To make the examples easy, we can revive the `calendar-at-x` function from Chapter 8 and say

```
(check-expect (handle-redraw state1) (calendar-at-x 10))
(check-expect (handle-redraw state2) (calendar-at-x 29))
```

The skeleton and inventory are easy from the input template:

```
(define (handle-redraw current)
  ; current                   moving-x
  ; (moving-x-x current)      number
  ; (moving-x-dir current)    symbol
  ...)
```

If you already see what to do, great. If not, we'll add an "inventory with values":

```
(define (handle-redraw current)
  ; current                  moving-x   (make-moving-x 10 'right)
  ; (moving-x-x current)     number     10
  ; (moving-x-dir current)   symbol     'right
  ; right answer             scene      (calendar-at-x 10)
  ...)
```

This makes the body easy:

```
(define (handle-redraw current)
  ; current                  moving-x   (make-moving-x 10 'right)
  ; (moving-x-x current)     number     10
  ; (moving-x-dir current)   symbol     'right
  ; right answer             scene      (calendar-at-x 10)
  (calendar-at-x (moving-x-x current))
  )
```

Test this function on the above test cases before going on. Once it works, and if it's OK with your instructor, you *might* want to take out the "scratch work", leaving only the real code, which is quite short:

```
  (define (handle-redraw current)
    (calendar-at-x (moving-x-x current))
    )
```

Writing the tick handler
We already have a contract. Since the speed of motion is a fixed number, let's define a constant for it:

```
  (define SPEED 3)
```

And since part of the input data type has two cases (`'left` and `'right`), we'll need at least two examples, one for each.

```
  (check-expect (handle-tick (make-moving-x 10 'right))
                (make-moving-x (+ 10 SPEED) 'right))
  (check-expect (handle-tick (make-moving-x 29 'left))
                (make-moving-x (- 29 SPEED) 'left))
```

For the skeleton and inventory, we copy the template, change the name, and add some special values:

```
(define (handle-tick current)
  ; current                   moving-x
  ; (moving-x-x current)      number
  ; (moving-x-dir current)    symbol
  ; SPEED                     fixed number
  ; 'left, 'right             fixed symbols
  ...)
```

Clearly, we'll need to do something different depending on whether the current direction is `'left` or `'right`, so we'll need a conditional with those two cases (plus an error-handling case):

```
(define (handle-tick current)
  ; current                   moving-x
  ; (moving-x-x current)      number
  ; (moving-x-dir current)    symbol
  ; SPEED                     fixed number
  ; 'left, 'right             fixed symbols
  (cond [ (symbol=?  (moving-x-dir current) 'left)
          ...
          ]
        [ (symbol=?  (moving-x-dir current) 'right)
          ...
          ]
        [ else (error 'handle-tick
        "Direction is neither left nor right!")]
        )
  )
```

To figure out what to do in each case, let's copy the relevant parts of the inventory into each case and do an "inventory with values" for each:

```
(define (handle-tick current)
  ; ...
  (cond [ (symbol=?  (moving-x-dir current) 'left)
          ; (moving-x-x current)      number    29
          ; (moving-x-dir current)    symbol    'left
          ; right answer              moving-x
          ;    (make-moving-x (- 29 SPEED) 'left)
          ]
        [ (symbol=?  (moving-x-dir current) 'right)
          ; (moving-x-x current)      number    10
          ; (moving-x-dir current)    symbol    'right
          ; right answer              moving-x
          ;    (make-moving-x (+ 10 SPEED) 'right)
          ]
        [ else (error 'handle-tick
                      "Direction is neither left nor right!")]
        )
  )
```

Which makes the "answer" part of each cond-clause pretty easy:

```
(define (handle-tick current)
  ; ...
  (cond [(symbol=?  (moving-x-dir current) 'left)
         ; (moving-x-x current)      number    29
         ; (moving-x-dir current)    symbol    'left
         ; right answer              moving-x
         ;    (make-moving-x (- 29 SPEED) 'left)
         (make-moving-x (- (moving-x-x current) SPEED) 'left)
         ]
        [ (symbol=?  (moving-x-dir current) 'right)
         ; (moving-x-x current)      number    10
         ; (moving-x-dir current)    symbol    'right
         ; right answer              moving-x
         ;    (make-moving-x (+ 10 SPEED) 'right)
         (make-moving-x (+ (moving-x-x current) SPEED) 'right)
         ]
        [else (error 'handle-tick
                     "Direction is neither left nor right!")]
        )
  )
```

Test this function on the above test cases before going on.

Writing the key handler
We already have a contract. One of the inputs is a *key*, which for our purposes

can be broken down into `'left`, `'right`, and anything else. To stay on the safe side, we should probably have four test cases: one `'left`, one `'right`, one other symbol, and one character.

```
(check-expect (handle-key state1 #\e) state1)
(check-expect (handle-key state1 'up) state1)
(check-expect (handle-key state1 'right) state1)
              ; since state1 is already going right
(check-expect (handle-key state1 'left)
              (make-moving-x 10 'left))
(check-expect (handle-key state2 'right)
              (make-moving-x 29 'right))
```

For the skeleton and inventory, we have a choice: since the function takes in *both* a `moving-x` and a *key*, we could use the template for either one. In fact, we'll probably need elements of both:

```
(define (handle-key current key)
  ; current                   moving-x
  ; (moving-x-x current)      number
  ; (moving-x-dir current)    symbol
  ; key                       char or symbol
  ; 'left, 'right             fixed symbols
  (cond [ (key=?  key 'left)  ...]
        [ (key=?  key 'right) ...]
        [ else                ...]
        )
  )
```

The "else" case is easy: return `current` without modification. For the other two, we can use an "inventory with values":

```
(cond [(key=?  key 'left)
        ; (moving-x-x current)   number   10
        ; (moving-x-dir current) symbol   'right
        ; right answer           moving-x (make-moving-x 10 'left)
        ...]
      [ (key=?  key 'right)
        ; (moving-x-x current)   number   10
        ; (moving-x-dir current) symbol   'right
        ; right answer           moving-x (make-moving-x 10 'right)
        ...]
      [ else                     current ]
      )
)
```

To fill in the first of the "..." gaps, we clearly need `(make-moving-x (moving-x-x current) key)`. For the second, there are two places we could get a `'right` from: `(moving-x-dir current)` and `key`. Which one should we use? One way to decide would be to do another "inventory with values", using an example that was traveling to the left ... but since we've already said `(make-moving-x (moving-x-x current) key)` in the `'left` case, it seems simpler to do the same thing in the `'right` case:

```
(cond [(key=?  key 'left)
        ; (moving-x-x current)   number   10
        ; (moving-x-dir current) symbol   'right
        ; right answer           moving-x (make-moving-x 10 'left)
        (make-moving-x (moving-x-x current) key)]
       [ (key=?  key 'right)
        ; (moving-x-x current)   number   10
        ; (moving-x-dir current) symbol   'right
        ; right answer           moving-x (make-moving-x 10 'right)
        (make-moving-x (moving-x-x current) key)]
       [ else                    current ]
       )
)
```

Notice that we're returning the exact same expression in the `'left` and `'right` cases. Recognizing this, we can simplify the program by combining them into one:

```
(define (handle-key current key)
  ; ...
  (cond [ (or (key=?  key 'left) (key=?  key 'right))
          (make-moving-x (moving-x-x current) key)]
        [ else                   current ]
        )
  )
```

Test this before going on.

Running the animation

Now that we know each of the handlers works by itself, we can put them together:

```
  (run-animation WIDTH HEIGHT
    (make-moving-x (/ WIDTH 2) 'right)
            ; start at middle, moving right
    1       ; tick every second
    (on-redraw handle-redraw)
    (on-tick handle-tick)
    (on-key handle-key)
    )
```

which, when I test it, works as it's supposed to. ■

Exercise 23.6.2 ***Modify*** *the animation of Exercise 23.6.1 so that if the x coordinate becomes less than 0, the direction switches to* `'right`*, and if the x* *coordinate becomes more than `WIDTH`, the direction switches to `'left` — in other words, the picture "bounces" off the walls.*

Exercise 23.6.3 ***Modify*** *the animation of Exercise 22.6.4 so that it keeps track of how many clicks you've done before successfully clicking on a dot. Once you do, the end-of-time message should read something like `"Congratulations! It took you 13 clicks to hit a dot."`*

Hint: Your model needs to "remember" the current x and y coordinates of the dot, as well as how many clicks there have been so far (initially zero). The tick handler will generate a new set of random coordinates but keep the click count unchanged. The mouse handler will add one to the click count, but leave the coordinates unchanged (unless the click was close enough, in which case it builds an appropriate end-of-time message using `number->string` and `string-append`).

Hint: This is easier to do using `end-of-time` rather than `stop-when`.

23.7 Structs containing other structs

In Exercise 23.6.3, you probably defined a struct with three fields: `x`, `y`, and `clicks`. Two of the three happen to be the exact same fields as in a `posn`, so an alternative way to define this struct would be as *two* fields, one of which is a `posn`. (Fields of a struct can be *any* type, even another struct.) This has some advantages — any function you've previously written to work on `posn`s can be re-used without change — and some disadvantages — building an example is more tedious, *e.g.* `(make-click-posn (make-posn 3 4) 5)` rather than `(make-click-posn 3 4 5)`.

Exercise 23.7.1 ***Modify*** *the animation of Exercise 23.6.3 to use this sort of a model. It should behave exactly as before. Is the code shorter or longer? Easier or harder to understand?*

Exercise 23.7.2 *Define a data type `placed-circ` to represent a mathematical circle with its two-dimensional location. It should have a `posn` for its center, and a number for its radius.*

Exercise 23.7.3 *Define a data type `placed-rect` to represent a mathematical rectangle with its two-dimensional location. It should have a `posn` for the "top-left corner" (a common way of representing rectangles in computer graphics), and two numbers for the width and height.*

Exercise 23.7.4 *Define a function `circs-overlap?` which takes in two of these `placed-circ` structures and tells whether they overlap.*

Hint: Use the distance between their centers, together with their radii.

Exercise 23.7.5 *Write an animation of a dot that moves around the screen at a constant speed until it hits the top, left, right, or bottom edge of the window, at which time it "bounces off".*

Hint: You'll need a `posn` to represent the current location, plus two numbers (or a `posn`, if you prefer) to represent the current *velocity* — how fast is it moving to the right, and how fast is it moving down? When you hit a wall, one component of the velocity should be reversed, and the other should stay as it was. You may find it easier to break your tick handler into *three* functions: one to move the dot, one to decide whether it should bounce in the x dimension, and one to decide whether it should bounce in the y dimension.

Exercise 23.7.6 *Modify the animation of Exercise 23.7.5 so that if you press any of the arrow keys, it* accelerates *the dot in that direction (that is, it changes the velocity, not the location). You now have a rocket-ship simulation.*

Exercise 23.7.7 *Modify the animation of Exercise 23.7.5 so that every second, the dot slows down a little bit (call it friction) — say, 5% per second. You now have a billiards simulation.*

Exercise 23.7.8 *Modify Exercise 20.4.2 (typing into the animation window) so there's a vertical-bar cursor showing where you're currently typing. The right-arrow key will move the cursor one character to the right (unless it's already at the end of the text), left-arrow one character to the left (unless it's already at the beginning), any ordinary character you type will be inserted into the text where the cursor is (and the cursor will move to the right), and the key #\backspace will delete the character just before the cursor.*

Hint: You'll need to define a structure to represent both the string that appears in the window and the location of the cursor. One good way to do this is to store two strings: the text before the cursor and the text after the cursor.

23.8 Decisions on types, revisited

In chapter 20, we learned to define a new data type "by choices", *e.g.* "an X is either a Y or a Z". But in that chapter, Y and Z were always predefined types like string, number, image, *etc.*. The technique of "definition by choices" becomes more useful when Y and Z are themselves defined "by parts", *i.e.* structs.

Recall that to write a function on a type defined by choices, we needed *discriminator functions* (*e.g.* `number?`, `string?`, `image?`) to tell which type something was. Conveniently enough, `define-struct` gives you a discriminator function for the newly-defined type, with the obvious name (`posn?`, `person?`, `employee?`, `candidate?`, ...).

Worked Exercise 23.8.1 ***Define a data type*** *placed-shape which is either a* `placed-circ` *(from Exercise 23.7.2) or a* `placed-rect` *(from Exercise 23.7.3).*

Develop a function `perimeter` *which works on a* placed-shape *and returns the length of the boundary of the shape.*

Solution: The data definition is simply "A placed-shape is either a placed-circ or a placed-rect." However, for this definition to be useful, we need some examples of the data type, and we need templates. Examples are easy: any `placed-circ` or any `placed-rect` will do (and to test a function on *placed-shape*, we should have at least one of each). Depending on exactly how you did Exercises 23.7.2 and 23.7.3, this could look like

```
(define shape-1 (make-placed-circ (make-posn 3 8) 5))
(define shape-2 (make-placed-rect (make-posn 15 21) 12 8))
```

The input template looks like

```
#|
(define (function-on-placed-shape s)
  (cond [(placed-circ?  s) (function-on-placed-circ s)]
        [(placed-rect?  s) (function-on-placed-rect s)]
  ))
|#
```

where `function-on-placed-circ` and `function-on-placed-rect` indicate functions written based on the input templates for those data types. If these functions are fairly short and simple, it may be more practical to combine all three into one, following a combined template like

```
#|
(define (function-on-placed-shape s)
  (cond [(placed-circ?  s)
         ; s                         placed-circ
         ; (placed-circ-center s)    posn
         ; (placed-circ-radius s)    number
         ...]
        [(placed-rect?  s)
         ; (placed-rect-top-left s) posn
         ; (placed-rect-width s)    number
         ; (placed-rect-height s)   number
         ...]
  ))
|#
```

Again, some of the details may vary depending on how you did Exercises 23.7.2 and 23.7.3.

We can also write an output template:

```
#|
(define (function-returning-placed-shape ...)
  (cond [... (function-returning-placed-circ ...)]
        [... (function-returning-placed-rect ...)]
  ))
|#
```

As with the input template, if the relevant functions returning a *placed-circ* and a *placed-rect* are short and simple, it makes more sense to combine them all into one template:

```
#|
(define (function-returning-placed-shape ...)
  (cond [... (make-placed-circ ... ...)]
        [... (make-placed-rect ... ... ...)]
  ))
|#
```

To define the `perimeter` function, we have a choice: either we write three separate functions `circ-perimeter`, `rect-perimeter`, and `perimeter`, each of which is fairly short, or we combine them into one larger function. We'll do both here, so you can see the advantages and disadvantages of each approach.

```
; circ-perimeter :  placed-circ -> number
(define empty-circ (make-placed-circ (make-posn 0 0) 0))
(define circ-1 (make-placed-circ (make-posn 10 4) 1))
(check-within (circ-perimeter empty-circ) 0 .01)
(check-within (circ-perimeter circ-1) 6.28 .01)
(check-within (circ-perimeter shape-1) 31.4 .1)
(define (circ-perimeter c)
  ; c                      placed-circ
  ; (placed-circ-center c) posn
  ; (placed-circ-radius c) number
  (* pi 2 (placed-circ-radius c)))
```

Note that since the formula for the perimeter of a circle involves π, which can be represented only approximately in a computer, the answer is inherently approximate so we use `check-within` rather than `check-expect`.

```
; rect-perimeter :  placed-rect -> number
(define empty-rect (make-placed-rect (make-posn 0 0) 0 0))
(define horiz-line (make-placed-rect (make-posn -1 0) 2 0))
(define square-2 (make-placed-rect (make-posn 1 1) (sqrt 2) (sqrt 2)))
(check-expect (rect-perimeter empty-rect) 0)
(check-expect (rect-perimeter horiz-line) 4)
(check-within (rect-perimeter square-2) 5.66 .01)
(check-expect (rect-perimeter shape-2) 40)
(define (rect-perimeter r)
  ; r                      placed-rect
  ; (placed-rect-top-left r)posn
  ; (placed-rect-width r)  number
  ; (placed-rect-height r) number
  (* 2 (+ (placed-rect-width r) (placed-rect-height r))))
```

The function on *placed-shape*s is now fairly simple:

```
; perimeter :  placed-shape -> number
(check-within (perimeter empty-circ) 0 .01)
(check-within (perimeter empty-rect) 0 .01)
(check-within (perimeter circ-1) 6.28 .01)
(check-within (perimeter square-2) 5.66 .01)
(check-within (perimeter shape-1) 31.4 .1)
(check-within (perimeter shape-2) 40 .1)
(define (perimeter s)
  (cond [(placed-circ?  s) (circ-perimeter s)]
        [(placed-rect?  s) (rect-perimeter s)]
  ))
```

If we wanted to write the whole thing as one big function, it would look more like this (the contract and examples are unchanged):

```
(define (perimeter s)
  (cond [(placed-circ?  s)
         ; c                      placed-circ
         ; (placed-circ-center c) posn
         ; (placed-circ-radius c) number
         (* pi 2 (placed-circ-radius c))]
        [(placed-rect?  s)
         ; r                      placed-rect
         ; (placed-rect-top-left r)posn
         ; (placed-rect-width r)  number
         ; (placed-rect-height r) number
         (* 2 (+ (placed-rect-width r) (placed-rect-height r)))]
  ))
```

If you were sure you would only need the `perimeter` function, not the more specific versions of it for the `placed-circ` and `placed-rect` types, and if you were confident of your programming skills, the single-function solution would probably be quicker and easier to write. On the other hand, three little functions are generally easier to test and debug (one at a time!) than one big function, and they can be individually re-used. For example, if in some future problem you wanted the perimeter of something you *knew* was a `placed-circ`, not a `placed-rect`, you could use `circ-perimeter` rather than the more general, but slightly less efficient, `perimeter`. In the long run, you should know both approaches. ∎

Exercise 23.8.2 ***Develop a function*** *`area` which works on a* placed-shape *and returns the area of the shape.*

Exercise 23.8.3 ***Develop a function*** *`contains?` which takes in a* placed-shape *and a `posn` and tells whether the `posn` is inside the shape. Consider the shape to include its border, so a point exactly on the border* is *"contained" in the shape.*

Exercise 23.8.4 ***Develop a function*** *`shapes-overlap?` which takes in two* placed-shape*s and tells whether they overlap.*

Hint: This problem is a little harder. Since *each* of the two parameters can be either a circle or a rectangle, you have four cases to consider. The "both circles" case is handled by Exercise 23.7.4; the "both rectangles" case can be handled by using a previously-defined function on *placed-shape*s; and the "circle and rectangle" cases will require some geometrical thinking.

Exercise 23.8.5 ***Develop an animation*** *like Exercise 22.6.4 or 23.6.3, but with each shape being* either *a circle (with random location and radius) or a rectangle (with random location, width, and height), with a 50% probability of each shape. I recommend testing this with a slow clock tick,* e.g. *5 seconds, so you have time to try clicking in several places just outside various sides of the shape to make sure they don't count as hits.*

Exercise 23.8.6 ***Define a data type*** zoo-animal *which is either a monkey, a lion, a sloth, or a dolphin. All four kinds have a name and a weight. Lions have a numeric property indicating how much meat they need per day (in kilograms). Monkeys have a string property indicating their favorite food (*e.g. *"ants", "bananas", or "caviar"). Sloths have a Boolean property indicating whether they're awake.*

Exercise 23.8.7 ***Develop a function*** *`underweight?` that takes in a* zoo-animal *and returns whether the animal in question is underweight. For this particular kind of monkey, that means under 10 kg; for lions, 150 kg; for sloths, it's 30 kg; for dolphins, 50 kg.*

Exercise 23.8.8 ***Develop a function*** *`can-put-in-cage?` that takes in a* zoo-animal *and a number (the weight capacity of the cage) and tells whether the* *animal in question can be put into that cage. Obviously, if the weight of the animal is greater than the weight capacity of the cage, the answer is* `false`*. But sloths cannot be moved when they're asleep, and dolphins can't be put in a cage at all.*

Exercise 23.8.9 ***Define a data type*** vehicle *which is either a car, a bicycle, or a train. All three types of vehicle have a weight and a top speed; a bicycle has a number of gears; a train has a length; and a car has a horsepower (*e.g. *300) and a fuel-economy rating (*e.g. *28 miles/gallon).*

Exercise 23.8.10 ***Develop a function*** `what?` *on vehicles....*
TO DO: *FILL THIS IN*

23.9 Review

A *struct* is a data type made up of several "parts" or *fields*. An *instance* of a data type is an individual object of that type — for example, 2/3, 5, and -72541 are all instances of the type `number`, while `(make-posn 3 4)` is an instance of the type `posn`. The built-in function `define-struct` allows you to define a new struct type, and also provides several functions to allow you to manipulate the new type: a *constructor* which builds individual instances of the new data type; several *getters* (one for each field) which retrieve the value of that field from an instance of the new type; and a *discriminator* which tells whether something is of the new type at all.

There's a step-by-step recipe for defining a struct, just as for defining a function or writing an animation:

1. Identify the parts, their names and types
2. Write a `define-struct`
3. Write the contracts for the constructor, getters, and discriminator
4. Write some examples
5. If you expect to write several functions involving the data type, write input and output templates.

A function can not only take in but *return* instances of user-defined struct types, as with `posn`. If you need to write such a function, the "inventory with values" technique will be very useful.

Many animations need more than just two numbers in their models, so you often need to define a struct type for the purpose.

The fields of a struct can be of *any* type, even another struct. This sometimes allows you to better re-use previously-written functions (especially for `posn`s).

Definition by choices becomes much more interesting when the choices can themselves be user-defined types. Functions written to operate on such types may be written in one of two ways: either one function per type, with a short "main" function that simply decides which choice the input is and calls an appropriate helper function, or as one big function with the helper functions "collapsed" into the main function. The single-function approach may be more convenient if the helper functions are all very short and simple, and if they are unlikely to be needed on their own; otherwise, it's usually safer and more flexible to write one function per type.

Part IV

Definition by Self-reference

Chapter 24

Lists and functions on them

24.1 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 `person` has two strings and a number, a `moving-x` has a number and a symbol, ... 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 perhaps the simplest is a "list".

24.2 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 Scheme. 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.

24.3 Defining lists in Scheme

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 the next section we'll discuss the more practical version of lists that's built into Scheme. 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 symbols, or booleans, or even lists similarly. We'll use "los" as shorthand for "list of strings".

24.3.1 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 Scheme 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 at all. The purpose of this data type is simply that 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
  ...)
|#
```

Note that 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 25.)

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 refer to one another, the `function-on-los` and `function-on-nelos` templates refer to one another in a corresponding way.

24.3.2 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 `"buenos dias"` as its first element and `"hello"` as the second and last. This is easy by calling `make-nelos`:

```
(define span-eng (make-nelos "buenos dias" english))
```

We can go on to build even longer lists, as shown in Figure 24.1.

Practice Exercise 24.3.1 *Copy Figure 24.1 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 the answer was what it was before going on.*

- *(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? span-eng)*
- *(nelos-first span-eng)*
- *(nelos-rest span-eng)*
- *(nelos? (nelos-rest span-eng))*
- *(nelos-first (nelos-rest span-eng))*

Figure 24.1: Defining the list-of-strings data type

```
 ; 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 span-eng (make-nelos "buenos dias" english))
(define heb-span-eng (make-nelos "shalom" span-eng))
(define fhse (make-nelos "bonjour" heb-span-eng))
(define afhse (make-nelos "salaam" fhse))
(define dwarfs (make-nelos "sleepy" (make-nelos "sneezy"
  (make-nelos "dopey" (make-nelos "doc" (make-nelos "happy"
  (make-nelos "bashful" (make-nelos "grumpy" nothing))))))))
```

- *(nelos-rest (nelos-rest span-eng))*
- *(nelos? afhse)*
- *(nelos-first afhse)*
- *(nelos-rest afhse)*
- *(nelos-first (nelos-rest (nelos-rest afhse))))*
- *(nelos-first (nelos-rest (nelos-rest (nelos-rest dwarfs))))*

24.3.3 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 24.3.2 *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: `count-strings` that works on a *los*, and `count-strings-on-nelos` 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 span-eng) 2)
(check-expect (count-strings afhse) 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 span-eng) 2)
(check-expect (count-strings-on-nelos afhse) 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? Obviously, one more than in the *rest* of the list:

```
(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
  (+ 1 (count-strings (nelos-rest L))) )
```

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

24.3.4 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.

SIDEBAR:

Computer scientists use the word "recursion" for a function that calls itself, or (for that matter) for two or more functions that call one another. 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 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 justification (relying on the mathematical proof technique called "proof by contradiction"). Suppose we wrote a function this way and it *didn't* work correctly, *i.e.* there was at least one legal input on which it produced a wrong answer. Then there must be a *shortest* legal input on which it produces a wrong answer. This shortest "bad" input can't be the empty list, because we know the function produces the right answer on the empty list. So whatever the shortest "bad" input is, the function must produce correct answers on all shorter lists. Since the "rest" of this list is shorter than the whole list, the function must be correct on it, so we can legitimately assume that the call on the rest of the list produces a right answer. If we can convince ourselves that the function always produces a correct answer for the whole list from a correct answer for the rest of the list, then it must produce a correct answer to the shortest "bad" input, which contradicts the assumption that it produces a wrong answer on this input. Thus our assumption that the function sometimes produces wrong answers leads to an impossibility, so the assumption must be wrong, *i.e.* the function is always correct.

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
        ...]))
|#
```

24.4 The way we really do lists

The approach taken in Section 24.3 works, but it's rather awkward to work with. Lists are so common and useful that they're built into Scheme; in reality, most Scheme programmers use the built-in list functions rather than defining a list data type themselves.

As in section 24.3, we'll define a list of strings for concreteness. You can also define a list of numbers, or symbols, or booleans, or even lists similarly. We'll use "los" as shorthand for "list of strings".

24.4.1 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." Scheme 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, Scheme 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"). Scheme 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 25.)

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

24.4.2 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 `"buenos dias"` as its first element and `"hello"` as the second and last. This is easy by calling `cons`:

```
(define span-eng (cons "buenos dias" english))
```

We can go on to build even longer lists, as shown in Figure 24.2.

Practice Exercise 24.4.1 *Copy Figure 24.2 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 the answer was what it was before going on.*

- *(empty? empty)*
- *(cons? empty)*
- *(first empty)*
- *(rest empty)*
- *(empty? english)*
- *(cons? english)*
- *(first english)*
- *(rest english)*
- *(empty? (rest english))*
- *(cons? span-eng)*
- *(first span-eng)*
- *(rest span-eng)*
- *(cons? (rest span-eng))*
- *(first (rest span-eng))*
- *(rest (rest span-eng))*
- *(cons? afhse)*
- *(first afhse)*
- *(rest afhse)*
- *(first (rest (rest afhse))))*
- *(first (rest (rest (rest dwarfs))))*

Figure 24.2: Lists of strings, using built-in Scheme 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 span-eng (cons "buenos dias" english))
(define heb-span-eng (cons "shalom" span-eng))
(define fhse (cons "bonjour" heb-span-eng))
(define afhse (cons "salaam" fhse))
(define dwarfs (cons "sleepy" (cons "sneezy" (cons "dopey" (cons "doc"
  (cons "happy" (cons "bashful" (cons "grumpy" empty))))))))
```

24.4.3 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 oone another, so when we write a specific function, it too will probably consist of two functions that depend on one another.

Worked Exercise 24.4.2 *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: `count-strings` that works on a *los*, and `count-strings-on-nelos` 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 span-eng) 2)
(check-expect (count-strings afhse) 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 span-eng) 2)
(check-expect (count-strings-on-nelos afhse) 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.

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? Obviously, one more than in the *rest* of the list:

```
(define (count-strings-on-nelos L)
  ; L                             a nelos
  ; (first L)                     a string
  ; (rest L)                      a los
  ; (count-strings (rest L))      a number
  (+ 1 (count-strings (rest L))) )
```

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

24.4.4 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.

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.

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
        ...]))
|#
```

24.5 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 Scheme'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 Scheme's built-in list features; they could of course be written to use the `empty-list` and `nelos` structures defined in section 24.3, but the definitions would be longer and harder to understand.

Worked Exercise 24.5.1 ***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 extremely similar to that for `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 Scheme 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 24.5.2 ***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 24.5.3 ***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 24.5.4 ***Develop*** *a function* `count-matches` *that takes in a string and a list of strings and tells* how many *(possibly zero) of the strings in the list are the same as the given string. For example,*

```
(check-expect
  (count-matches "cat" (cons "dog" (cons "cat" (cons "fish"
    (cons "cat" (cons "cat" (cons "wombat" empty)))))))
  3)
```

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

Exercise 24.5.5 ***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 24.5.6 ***Develop a function*** *`count-over` which 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 24.5.7 ***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 24.5.8 ***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 24.5.9 ***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)
```

Exercise 24.5.10 ***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 24.5.11 ***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 24.5.12 ***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 24.5.13 ***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 which 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 24.5.14 ***Develop an animation*** *that displays a bull's-eye pattern, as in Exercise 24.5.13, but each second, an additional ring will be added at a* random radius.

Exercise 24.5.15 ***Develop a function*** `largest` *which 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. One good way to ensure this is for your function to ask not whether the given list is empty or not, but rather whether the *rest* of the given list is empty or not, before doing anything with the rest of the list.

Exercise 24.5.16 ***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` *which 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` *which 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.*

Exercise 24.5.17 ***Write a data definition****, including templates, for a* general list*, in which each element may be either a simple object (like a string or a number) or another general list.*

Develop a function `count-simple-objects` *which takes in a general list and returns the total number of simple objects contained in it, no matter how many levels of nested lists they're inside.*

Develop a function `max-nesting-depth` *which takes in a general list and returns its maximum nesting depth:* `empty` *has a nesting depth of 0, a list of simple objects has a nesting depth of 1, a list that contains some lists of simple elements has a nesting depth of 2,* etc.

It can be difficult to read and write the test cases for Exercises 24.5.16 and 24.5.17. In Chapter 26 we'll learn a more compact notation for lists that makes this easier.

24.6 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 24.6.1 ***Develop a function*** `any-on-diag?` *which 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                                        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
  (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                                              a list of posns
  (cond[(empty?  L)      false]
       [(cons?  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
        (or (= (posn-x (first L)) (posn-y (first L)))
            (any-on-diag?  (rest L)))
       ]))
```

█

Exercise 24.6.2 ***Develop a function*** *`any-over-100K?` which takes in a list of `employee` structures (from Exercise 23.2.1) and tells whether any of them earn over $100,000 per year.*

Exercise 24.6.3 ***Develop a function*** *`total-votes` which takes in a list of `candidate` structures (from Exercise 23.3.2) and returns the total number of votes cast in the election.*

Exercise 24.6.4 ***Develop a function*** *`avg-votes` which 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 24.6.5 ***Develop a function*** *`winner` which 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 24.6.6 ***Develop an animation*** *similar to Exercise 22.6.4, 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.

24.7 Review

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.

Chapter 25

Functions that return lists

If you did exercises 24.5.13 or 24.5.14, you've already written some functions that return lists, but only in a very simple way: adding one new element to the front of an existing list. In this chapter we'll discuss functions that construct an entire list as their results.

25.1 Doing something to each element

Worked Exercise 25.1.1 ***Develop a function*** *`add1-each` which takes in a list of numbers and returns a list of numbers of the same length, with each element of the answer equal to 1 more than the corresponding element of the input. For example,*

```
(check-expect (add1-each (cons 3 (cons -12 (cons 7 empty))))
              (cons 4 (cons -11 (cons 8 empty))))
```

Solution: For brevity, I'll write this as a single function; the two-function version is quite similar. The contract, skeleton, and inventory are straightforward:

```
(define (add1-each nums)
  ; nums                        lon
  (cond [(empty?  nums)      ...]
        [(cons?  nums)
         ; nums                 nelon
         ; (first nums)         number
         ; (rest nums)          lon
         ; (add1-each (rest nums)) lon
         ...]
  ))
```

The answer to the empty case is obviously `empty` (since the result has to be the same length as the input). To fill in the non-empty case, let's do an inventory with values:

```
[(cons? nums)
 ; nums                      (cons 3 (cons -12 (cons 7 empty)))
 ; (first nums)              3
 ; (rest nums)               (cons -12 (cons 7 empty))
 ; (add1-each (rest nums)) (cons -11 (cons 8 empty))
 ; right answer              (cons 4 (cons -11 (cons 8 empty)))
 ...]
))
```

Notice that the recursive call `(add1-each (rest nums))` gives you most of the right answer, but it's missing a 4 at the front. Where could the 4 come from? Since `(first nums)` in this example is 3, an obvious choice is `(+ 1 (first nums))`.

```
[(cons? nums)
 ; nums                      (cons 3 (cons -12 (cons 7 empty)))
 ; (first nums)              3
 ; (rest nums)               (cons -12 (cons 7 empty))
 ; (add1-each (rest nums)) (cons -11 (cons 8 empty))
 ; right answer              (cons 4 (cons -11 (cons 8 empty)))
 (cons (+ 1 (first nums))
       (add1-each (rest nums))) ]
))
```

Test this, and it should work on all legal inputs. ∎

Exercise 25.1.2 ***Develop a function*** *`square-each` which takes in a list of numbers and returns a list of their squares, in the same order.*

Exercise 25.1.3 ***Develop a function*** *`string-lengths` which takes in a list of strings and returns a list of their (numeric) lengths, in the same order.*

Exercise 25.1.4 ***Develop a function*** *`stutter` which takes in a list of anything (it doesn't matter whether they're strings, numbers, or something else) and returns a list twice as long, with each element repeated twice in a row. For example,*

```
(check-expect (stutter (cons 5 (cons 2 (cons 9 empty))))
  (cons 5 (cons 5 (cons 2 (cons 2 (cons 9 (cons 9 empty)))))))
```

Exercise 25.1.5 ***Develop a function*** *`list-each` which takes in a list (of numbers, strings, symbols, it doesn't matter) and returns a* list of one-element lists*, each containing a different one of the elements in the original list. For example,*

```
(check-expect (list-each (cons "a" (cons "b" empty)))
  (cons (cons "a" empty) (cons (cons "b" empty) empty)))
```

25.2 Making decisions on each element

In some problems, you need to make a decision about each element of a list, using a conditional. As with Exercises 24.5.4, 24.5.6, *etc.*, this conditional is usually nested inside the one that decides whether the list is empty or not.

Exercise 25.2.1 ***Develop a function*** `substitute` *which takes in two strings and a list of strings, and returns a list the same length as the given list, but with all occurrences of the first string replaced by the second. For example,*

```
(check-expect
  (substitute "old" "new" (cons "this" (cons "that" (cons "old"
    (cons "new" (cons "borrowed" (cons "old" (cons "blue"
    empty))))))))
  (cons "this (cons "that" (cons "new" (cons "new"
    (cons "borrowed" (cons "new" (cons "blue" empty)))))))
```

Exercise 25.2.2 ***Develop a function*** `remove-all` *which takes in a string and a list of strings, and returns the same list but with all occurrences (if there are any) of the given string removed. For example,*

```
(check-expect
  (remove-first "old" (cons "this (cons "that" (cons "old"
    (cons "new" (cons "borrowed" (cons "old"
    (cons "blue" empty))))))))
  (cons "this" (cons "that" (cons "new" (cons "borrowed"
    (cons "blue" empty))))))
```

Exercise 25.2.3 ***Develop a function*** `remove-first` *which takes in a string and a list of strings, and returns the same list but with the first occurrence (if any) of the given string removed. For example,*

```
(check-expect
  (remove-first "old" (cons "this (cons "that" (cons "old"
    (cons "new" (cons "borrowed" (cons "old"
    (cons "blue" empty))))))))
  (cons "this" (cons "that" (cons "new" (cons "borrowed"
    (cons "old" (cons "blue" empty)))))))
```

Exercise 25.2.4 ***Develop a function*** `add-vote-for` *which takes in a string (representing a candidate's name) and a list of* `candidate` *structures, and returns a list of* `candidate` *structures in which that candidate has one more vote (and all the others are unchanged). You may assume that no name appears more than once in the list.*

Hint: What should you do if the name doesn't appear in the list at all?

Exercise 25.2.5 ***Develop a function*** `tally-votes` *which takes in a list of strings (Voter 1's favorite candidate, Voter 2's favorite candidate,* etc.*) and produces a list of* `candidate` *structures in which each candidate's name appears once, with how many votes were cast for that candidate.*

25.3 Animations with lists

Exercise 25.3.1 ***Write an animation*** *of a bunch of balls, each moving around the screen with constant velocity and bouncing off walls. Pressing the* `+` *key should create one more ball, with random initial location (inside the animation window) and random velocity (say, from -10 to +10 in each dimension). Pressing the* `-` *key should remove the most recently-added ball, unless there are no balls, in which case it should do nothing. Clicking with the mouse inside a ball should remove the ball you clicked on, leaving the rest of the balls unchanged.*

Hint: You'll need each ball to have a location and a velocity, as in exercise 23.7.5, and use a list of structs as your model, as in exercise 24.6.6.

Hint: What should happen if you click with the mouse in a place where two or more balls overlap? The assignment doesn't say exactly; you should decide in advance what you *want* to happen in this case, and make it work.

25.4 A shorter notation for lists

Writing `(cons "a" (cons "b" (cons "c" empty)))` for a three-element list is technically correct, but it's tedious. Fortunately, Scheme provides a shorter way to accomplish the same thing. There's a built-in function named `list` that takes zero or more parameters and constructs a list from them. For example,

```
>(list "a" "b" "c")
(cons "a" (cons "b" (cons "c" empty)))
```

Note that `list` is *just a shorthand*: it produces the exact same list as the `cons` form, and any function that works on one of them will still work on the other.

Note also that you can't just replace every `cons` in your code with `list`. `cons` can be thought of as *adding one element to the front* of a list, whereas `list` builds a list from scratch. For example,

```
>(define my-list (list "x" "y" "z"))
>(cons "w" my-list)
(cons "w" (cons "x" (cons "y" (cons "z" empty))))
>(list "w" my-list)
(cons "w" (cons (cons "x" (cons "y" (cons "z") empty) empty))
```

This makes it easier and more convenient to type in list, but it's still sort of a pain to read them. If you want lists to *print out* in `list` notation rather than nested-`cons` notation, simply go to the "Language" menu in DrScheme, select "Choose Language", and then (under the *How to Design Programs* heading) select "Beginning Student with List Abbreviations".

```
>(list "a" "b" "c")
(list "a" "b" "c")
>(cons "a" (cons "b" (cons "c" empty)))
(list "a" "b" "c")
>(define my-list (list "x" "y" "z"))
>(cons "w" my-list)
(list "w" "x" "y" "z")
>(list "w" my-list)
(list "w" (list "x" "y" "z"))
```

25.5 More complex functions involving lists

Lists allow us to solve much more interesting and complicated problems than we could solve before. Sometimes such problems require "helper" or "auxiliary" functions. Here are a few interesting examples.

Exercise 25.5.1 ***Develop a function*** *`sort` which takes in a list of numbers and returns a list of the same numbers in increasing order.*

Hint: Just follow the design recipe. When you reach the "inventory with values" point, you'll find that the "right answer" needs to be constructed by inserting one number into the right place in an already-sorted list. We don't already have a function to do that, so write one. I recommend distinguishing between `list-of-numbers` and `sorted-list-of-numbers`: when a function produces a sorted list, or assumes that it is given a sorted list, say so in the contract and inventory, and make sure your test cases satisfy the assumption.

Exercise 25.5.2 ***Develop a function*** *`sort-candidates` which takes in a list of `candidate` structures and returns a list of the same `candidate` structures in decreasing order by number of votes, so the winner is first in the list, the second-place winner is second,* etc.. *In case of ties, either order is acceptable.*

Exercise 25.5.3 ***Develop a function*** *`scramble` which takes in a list (of numbers, strings, symbols, it doesn't matter) and returns a list of lists representing* all possible orderings *of the elements in the original list. For example, `(scramble (list "a" "b" "c" )` should produce something like*

```
(list (list "a" "b" "c")
      (list "b" "a" "c")
      (list "a" "c" "b")
      (list "c" "a" "b")
      (list "b" "c" "a")
      (list "c" "b" "a"))
```

Chapter 0

To the Instructor

although students are welcome to read it too!

This chapter explains the philosophy of how I teach programming, and why the book is written the way it is. If you just want to get started programming, by all means skip this chapter, and perhaps come back and read it later.

0.1 What is this book about? Why Scheme?

This book may be described as "a Scheme textbook". That's not quite an accurate description: I'd prefer to call it "a programming textbook that happens to use the Scheme language." In other words, the Scheme language is not the goal, but only a means towards the goal of knowing how to program.

Here's why: Scheme, C++, Java, Python, or any other specific computer language will become obsolete in a few years. If you plan to get a job as a computer programmer next month, then by all means study the language(s) used in industry right now. But if you plan to get a job programming several years from now, you'll have to learn a new language then anyway. The current school term will be better spent learning more long-lasting skills, habits, and principles: how to structure a program, what steps to take in developing a program, how to manage your time so you finish the program on time, *etc.* And if you don't plan to be a professional programmer at all, then you don't need to learn this year's "hot" language at all; you need to learn the important principles of programming, in whatever language will "get out of the way" and let you learn them.

Here's my guiding rule in writing this book:

> **Introduce each language feature only when it helps to teach an important programming principle.**

In this book, there will be no "black magic": nothing that you need to memorize on faith that you'll eventually understand it. On the first day, you will see just enough language to do what you need on the first day. By the end of the term, you will see just enough language to teach what you need in one term. Any language feature that doesn't help to teach an important programming principle doesn't belong in this book. Most programming languages, frankly, don't allow me to do that: in C++ or Java, for example, the very first program you write requires knowing dozens of language features that won't be fully explained for months. Scheme allows me to postpone irrelevant language features, and concentrate on the important stuff.

But again, Scheme is only a means to an end. If students, six months after taking this course, don't remember any Scheme syntax at all but can follow the design recipes (see below) in designing a correct program, the course has been a success.

0.2 Structure of the book

Each chapter is designed to progress from the concrete to the abstract, starting with either a specific problem to solve or a hands-on technique using the DrScheme programming environment, moving on to more general techniques and principles before wrapping up with a summary of what I think are the most important concepts and terms from the chapter.

In particular, many of the chapters have a section or two towards the end on formal language syntax. I put these sections in because I think they help: asking students to *justify* that they're writing legal code makes clear to them that the computer is not a mysterious black box, but that it operates according to *rules* that students too can learn and apply. The computer doesn't *substitute* for human thinking, but merely *accelerates* it. Ideally, once students internalize some of these rules, they won't fall prey to "my program doesn't work; make a random change to the code and see if it works now." However, if you feel that these sections don't help your students, you can omit the syntax sections.

Regardless of whether you use the syntax sections, I recommend periodically assigning students to show, *step by step*, the evaluation of expressions, in the same way the DrScheme Stepper does. Again, the idea is to demonstrate that everything DrScheme does, a human being can do and understand too, just a billion times slower. This technique also becomes a useful debugging tool when applied to complex programs, and it corresponds to the notion of *execution tracing* in imperative/sequential programming (*e.g.* in Java, C++, or Ada).

TO DO: Insert some worked and unworked exercises of this form into the book

For the most part, I recommend going through the chapters in the order they appear in the textbook, although

TO DO: EXCEPTIONS?

.

0.3 Problems, programs, and program testing

A computer program that answered only one specific question, like

> add 3 and 4

wouldn't be very useful. Most computer programs are written to be *general*, in that a *single* program can answer any one of *many similar questions*:

- add 3 and 4
- add 19 and -5
- add 102379 and -897250987

etc. Somebody writes the program to add two numbers once and for all; later on, when you *run* the program, you provide specific values like 3 and 4, and the program produces the right answer for those values. Run it again with different values, and it should produce the right answer for the new values instead.

To take a more realistic example, a word processor program is written to handle whatever words you choose to write. When you run the program, you provide specific words — a grocery list, a letter to your grandmother, the Great American Novel — and the program responds by doing things like formatting them to fit on a page. Likewise, when you run a Web browser, you provide a specific URL for a page you want to look at; the browser program uses the network to retrieve specific words and pictures from that Web page, and then arranges these words and pictures on the screen. If you've done a lot of Web surfing, you've probably found an occasional page that showed up on the screen as nonsense; this probably means the page had some weird information that the browser wasn't written to handle correctly.

Thus for a computer program to be considered "correct", it has to produce the right answer for *all possible* values it might be given to work on — even the weird ones. One of the important steps in writing a computer program is *testing* it to make sure it works correctly. However, since there are usually far too many possible values to test them all, we have to *choose test cases*, being careful to pick not only the easy cases but also the weird ones, so that if there's something our program doesn't handle correctly, we find out as soon as possible so we can fix it.

A program that hasn't been tested convincingly is worthless: nobody will (or should!) trust the answers it produces. Indeed, if you *tell* me you've tested the program, but don't provide me with what I need in order to test it myself, I may not trust *you or* the program's answers. In this book, I've tried to incorporate a philosophy of testing from the beginning; a program turned in without a good selection of test cases, together with what answers they are supposed to produce, is incomplete and should be graded accordingly.

0.4 Data types

The notion that data types are important in programming is nothing new, dating back at least to the 1960's. But in this book (and its predecessor, *How to Design Programs*), data types are used in very specific ways to help people program.

> **The shape of the data determines the shape of the code and the tests.**

For example, if a function deals with a data type that has three variants ("a W is either an X, a Y, or a Z"), then the function body will almost certainly involve a conditional with three cases, and the function's test cases must cover all three variants. Likewise, if a function deals with a data type that has three *parts* ("a W consists of an X, a Y, and a Z"), the function code will probably need to access those parts, and the test cases must set and/or test the values of those parts. These two approaches — "definition by choices" and "definition by parts" — can then be combined to produce powerful data structures such as linked lists, binary and n-ary trees, as well as correct, tested code that works on those data structures.

To take advantage of this principle, we shall *always* identify the input and output data types of our functions *before* trying to write test cases or code. This is one example of . . .

0.5 Design Recipes

This book places heavy emphasis on *design recipes*, step-by-step plans for getting from (say) an English-language description of a computational problem to a working, tested Scheme function. When I first encountered the recipe for writing a function, in a workshop led by Matthias Felleisen, I thought it was pretty dumb . . . but in the years since, as I've taught beginning programming courses using it, I've been more and more convinced of its value — particularly for beginning students, of course, but parts of it will benefit even professional programmers. For example, the "write test cases" step comes *before* the student has written any code to solve the problem; in the professional software field, this is (the beginnings of) *test-driven design*, a crucial component of Agile Programming.

For a design recipe to help your students, they have to *use* it. My sharpest students have often wanted to skip the recipe, and they usually manage to write working programs for a while by the seat of their pants, but about two thirds of the way through a semester I start seeing these "sharp" students at my office door with tangled messes of code. We throw away their code and start over, following the design recipe, and they walk out with a working program half as long as the buggy mess they brought in.

This teaches students a useful lesson, but it's better if they learn it earlier without spending half the semester on bad habits. To get students to use the design recipes consistently, I have three suggestions to teachers:

- Do not help a student with step N of the recipe until the student shows you satisfactory completion of step $N-1$. Be uncompromising and unwavering on this point. (There are a few situations in which the order of steps can be rearranged, *e.g.* sometimes it makes more sense to define data types *before* writing function contracts, and sometimes *after*; I'll try to draw attention to these exceptions as they come up.)

- Give students a grading rubric that explicitly gives partial credit for each step of the recipe successfully completed. In my class, a student can earn $30-50\%$ credit for a program without writing a line of code, simply by correctly analyzing what it's supposed to do and providing good test cases for it.

- And of course, *always* follow the design recipe yourself in class. No short-cuts; if you're allowed to skip steps, the students will too.

0.6 Acknowledgments

This book is based, heavily and intentionally, on *How to Design Programs*, by Felleisen, Flatt, Findler, and Krishnamurthi. The most important lessons of the present book were all introduced in that one: "the shape of the data determines the shape of the code and tests;" the design recipe for functions; the introduction of recursion as motivated by the "shape" of self-referential data; the taxonomy of recursion into structural, accumulative, and generative; and the broad order of topics.

I've diverged from *How to Design Programs* in several ways, informed by my experience teaching both a traditional college CS1 course and "programming for non-majors". First, the present book works extensively with graphics and animation before ever introducing arithmetic, as many of my non-CS-major students were intimidated by numbers and arithmetic; besides, graphics and animation are more *fun* than numbers and arithmetic. Second, I've written the prose at a slightly lower reading level, to make it accessible to a wider variety of students at the high school and college levels. Third, I've made a number of minor changes to terminology where I found that my students were consistently misunderstanding the terminology in *How to Design Programs*.

This material is based upon work supported by the National Science Foundation under Grant No. 0618543. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Chapter 1

Instructor's guide for chapter 5: A recipe for defining functions

As I point out at the end of the chapter, this is the single most important chapter in the book. If students internalize the steps of a design recipe like this, they'll be well on the way to being good programmers in *any* language, not only Scheme.

That said, the design recipe as stated in this chapter is specifically for *functional* programs — those that take in parameters and return values, but don't have side effects such as explicit input, output, or assignment. The design recipe can be adapted to handle those things as well, but it's more difficult: in particular, it's *much* more difficult to write a good test suite for a program that does I/O and assignment than for a function that returns a value determined entirely by its arguments. This is one of the main reasons we use a functional approach first, introducing I/O and assignment only later on.

To get across the lesson that *every step* of the design recipe matters, not only the coding, I encourage you to give partial credit for each step (and tell students this in advance!) For example, when grading individual functions, I often give 5 points for a correct contract, 5–10 for test cases, 5 for a correct function skeleton, 5–10 for a correct function inventory, and 15–20 for a correct function definition (as demonstrated by testing; I sometimes give another 5 points for turning in test runs). This means a student can get 60% or more credit on a problem without writing any working code (but with impeccable analysis and design). If this bothers you, remember Fred Brooks's rule of thumb that in a realistic software project, 1/3 of the time is spent on analysis and design, 1/6 on coding, and 1/2 on testing and debugging[**?**]. In my experience, students are more likely to overemphasize coding at the expense of analysis and design than *vice versa.*

Similarly, to emphasize doing the steps *in order*, I encourage you (and any tutors or teaching assistants you may have) to steadfastly refuse to help with a later stage of a program until the student has shown you the completed earlier stages. This doesn't necessarily mean the student has done them *correctly* — if a student shows you an incorrect contract, but doesn't know yet that it's incorrect, you'll have to use your judgment on whether to correct it immediately or go on and let the student discover the error at a later stage — but they must be completed to a basic, objectively-verifiable level before I'll give any help with a subsequent step.

Some students (particularly those who think they already know how to program better than you do) will either skip the design recipe entirely, or go ahead and write the program, then fill in the contract, examples, skeleton, and inventory steps after the fact. These students will eventually find themselves tangled up in an ill-understood and ill-specified program, will come to you for help, you'll throw out their program and start from scratch following the design recipe, and they'll finally be convinced that the recipe *does* help, at least in some cases; the only unknown is how long they'll be able to get by on native brilliance. Typically, my sharpest students have hit this wall when assigned programs that deal with lists of lists, *e.g.* power-set or permutations.

Chapter 2

Instructor's guide for chapter 22: Animations and posns

Here's a useful (and cheap!) classroom manipulable to help illustrate `posn`s. Take an ordinary letter-sized envelope, open along one of the long sides, and run a line of staples down the middle, perpendicular to the long sides. You have now subdivided the envelope into two compartments; label them "x" and "y" respectively. Now write two numbers on two small slips of paper, put one into the x compartment and the other into the y compartment. You've simulated the action of `make-posn`. Hand this to a student and ask the student for the x coordinate of the posn; the student will presumably reach into the x compartment and pull out the slip of paper, thus demonstrating the action of `posn-x`.

Chapter 3

Instructor's guide for chapter 23: Inventing new structures

One of my first book-testers, high school teacher Alvin Kroon, got this far in the book and then spent a few weeks teaching Microsoft Access before going on to struct definitions in Scheme. This way students were already familiar with the concept of a "record" having several "fields" before they had to do the same thing in Scheme.

If you did the "envelope with a line of staples" trick in Chapter 2, you can now take it a step farther by running *two* lines of staples across the envelope and labelling the compartments "first", "last", and "age" (for the `person` struct) *etc.*.

Chapter 4

Instructor's guide for chapter 24: Lists and functions on them

Section 24.3 presents a definition of a list data type "from first principles", but it's sorta inconvenient to work with, and section 24.4 shows the way we actually work with lists in practice. If your students are the sorts to be annoyed at having to read through a theoretical construction of lists that isn't the way they'll actually use them, feel free to skip section 24.3. If, on the other hand, they're the sorts who will better understand and appreciate the practical version of lists after building them from first principles, don't skip it.

Exercise 24.5.10 will challenge some students because they're convinced that the only possible answer to a question about the empty list is 0. The problem *can* be solved that way, but it's much easier if you decide that the answer to the empty list is 1 (*i.e.* the identity element for multiplication, as 0 is the identity for addition). Likewise, exercise 24.5.3 relies on the fact that `false` is the identity for `or`, and exercise 24.5.11 on the fact that `true` is the identity for `and`. If students don't believe that `(all-match empty)` should be true, you could try starting with a larger list, and observing that adding an element can change the answer from true to false (if the new element doesn't match) but never from false to true; on the other hand, removing an element can change the answer from false to true (if the removed element was the only one that didn't match) but never from true to false. Now try a one-element list whose one element matches; the answer is obviously true. Remove this element; since removing an element cannot change the answer from true to false, the answer must still be true.

Exercise 24.5.15 can be done in a number of ways. Students who follow the usual templates will probably decide that the right answer to `(largest empty)` is 0, which works until you try a test case which is non-empty and consists entirely of negative numbers, *e.g.* `(largest (cons -3 (cons -14 (cons -5` `empty))))`. The Hint suggests, rather, that the function *has* no right answer for an empty list, and therefore shouldn't be called on an empty list. But the usual template for functions on lists involves calling the function recursively on the rest of the list, which might be empty. We're really working with a different datatype here: non-empty list, which is either a single element or more than one element. It ALWAYS has a first and a rest, but if it's more than one element, the rest is itself a non-empty list. This data definition leads to a slightly different template that works well for `largest`.