PART III

Definition by Parts

Chapter 20

Using Structures

20.1 The `posn` data type

Recall Exercise 18.2.1, 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 Racket, 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, DrRacket 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 DrRacket Interactions pane:

```
(define where (make-posn 7 12))
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:

```
(posn-x where)        ; should be 7
(posn-y where)        ; should be 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 draw handler).

Practice Exercise 20.1.1 ***Create*** *(in the Interactions pane) several variables containing different* `posns`*. Extract their x and y coordinates and make sure they are what you expected.*

Try the `posn?` *function on these variables, and on some expressions that use* `make-posn` *directly (*e.g. `(make-posn 13 5)`*), and on some things that* aren't *posns.*

Common beginner mistakes

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))
```

20.2 Definition by parts

In Chapter 16 we learned about "defining a new data type by choices," and in Section 15.8 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.

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 21, we'll see more examples of definition by parts.

20.3 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-posn 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-posn 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.

20.4 Writing functions on posns

So now let's write some actual functions involving posns.

Worked Exercise 20.4.1 ***Develop a function*** *named* `right-of-100?` *that 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 uses 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 actually uses 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 102 123)) true)
(check-expect (right-of-100?  (make-posn 100 123)) false)
; borderline case
```

The template gives us most of the skeleton and inventory. And since it's hard to imagine solving this problem without using the number 100, we'll put that into the inventory too:

```
(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 function should return true, and if this expression is false, it should return false, so we can just use this expression itself as the body:

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

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

Common beginner mistakes

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 first of these passes a syntax check in Racket. 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, and see what happens.

Exercises: writing functions on posns

Exercise 20.4.2 ***Develop a function*** *named* ***`above-diagonal?`*** *that 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 20.4.3 ***Write a function*** *named* ***`distance-to-top-left`*** *that 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 subcategories 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. Note that since there are square roots involved, the answers may be inexact, so we use `check-within` rather than `check-expect`.

```
(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)
  ; (sqr (posn-x the-point))     a number (x^2)
  ; (sqr (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)
  ; (sqr (posn-x the-point))     a number (x^2)
  ; (sqr (posn-y the-point))     a number (y^2)
  ; (+ (sqr (posn-x the-point))
  ;     (sqr (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)
  ; (sqr (posn-x the-point))         a number (x^2)
  ; (sqr (posn-y the-point))         a number (y^2)
  ; (+   (sqr (posn-x the-point))
  ;       (sqr (posn-y the-point)))   a number (x^2 + y^2)
  (sqrt (+ (sqr (posn-x the-point))
           (sqr (posn-y the-point))))
  )
```

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

Of course, as you get more comfortable with writing functions on posns, you won't need to write down all these intermediate steps, and can simply write

```
(define (distance-to-top-left the-point)
  (sqrt (+ (sqr (posn-x the-point))
           (sqr (posn-y the-point))))
  )
```

instead. But for now, I'd like you to use the inventory; discuss with your instructor when you can get away with skipping it. ∎

Exercise 20.4.4 ***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 20.4.5 ***Develop a function*** *named* `distance` *that takes in* two `posn`s *(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 20.4.6 ***Develop a function*** *named* `posn=?` *that takes in two* `posn`s *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.

You may *not* use the built-in `equal?` function to solve this problem.

Exercise 20.4.7 ***Develop a function*** *named* ***`distance-to-origin`*** *that 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)`****.*

20.5 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 20.5.1 ***Develop a function*** *named* ***`diagonal-point`*** *that 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
  ; right answer    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
  ; right answer    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 20.5.2 ***Develop a function*** *named* `swap-x-y` *that 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 20.5.3 ***Develop a function*** *named* `scale-posn` *that 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 20.5.4 ***Develop a function*** *named* `add-posns` *that takes in two* `posn`*s 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 20.5.5 ***Develop a function*** *named* `sub-posns` *that takes in two* `posn`*s 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 20.5.6 ***Redefine*** *the* `distance` *function from Exercise 20.4.5 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 Racket code.

Exercise 20.5.7 ***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* `posn`*s, 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.

20.6 Writing animations involving posns

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

Worked Exercise 20.6.1 ***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 is a `posn`, we'll need a draw handler with contract

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

and a key handler with contract

```
; handle-key :  posn key -> posn
```

Draw handler

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"))
...
(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)
  ; right answer   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)
  ; right answer   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.

Key handler

Now for the key handler. Recall that the contract is

```
; handle-key :  posn key -> posn
```

where "key" is really a string, but limited to certain specific strings. In this problem we're interested in four specific keys — `"left"`, `"right"`, `"up"`, and `"down"` — plus "any other key," which we'll ignore.

```
(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))
```

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 string
  ; (posn-x where)    a number(x)
  ; (posn-y where)    a number(y)
  ...)
```

We could also add the "outventory" line

```
; (make-posn some-number-x some-number-y)
```

because we know that `handle-key` is supposed to return a `posn`.

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".

```
(define (handle-key where key)
  ; where              a posn
  ; key                a key
  ; (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 key
  ; (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                string
  ; (posn-x where)     a number(x)     12
  ; (posn-y where)     a number(y)     19
  (cond [ (key=?  key "up")    ; right answer (make-posn 12 18)
          ]
        [ (key=?  key "down")  ; right answer (make-posn 12 20)
          ]
        [ (key=?  key "left")  ; right answer (make-posn 11 19)
          ]
        [ (key=?  key "right") ; right answer (make-posn 13 19)
          ]
        [ else                 where ]
        ))
```

From these "right answers", it's pretty easy to write the formulæ using `make-posn`:

```
(cond [ (key=?  key "up")    ; right answer (make-posn 12 18)
        (make-posn (posn-x where) (- (posn-y where) 1))]
      [ (key=?  key "down")  ; right answer (make-posn 12 20)
        (make-posn (posn-x where) (+ (posn-y where) 1))]
      [ (key=?  key "left")  ; right answer (make-posn 11 19)
        (make-posn (- (posn-x where) 1) (posn-y where))]
      [ (key=?  key "right") ; right answer (make-posn 13 19)
        (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 20.5.4. So assuming you've done that exercise, we can solve the problem as follows:

```
(cond [ (key=?  key "up")    ; right answer (make-posn 12 18)
        (add-posns where (make-posn 0 -1))]
      [ (key=?  key "down")  ; right answer (make-posn 12 20)
        (add-posns where (make-posn 0 1))]
      [ (key=?  key "left")  ; right answer (make-posn 11 19)
        (add-posns where (make-posn -1 0))]
      [ (key=?  key "right") ; right answer (make-posn 13 19)
        (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:

```
(big-bang (make-posn (/ WIDTH 2) (/ HEIGHT 2))
          (check-with posn?)
          (on-draw show-picture)
          (on-key handle-key))
```

■

Exercise 20.6.2 *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. Compare the length of the resulting function with the length of the function definition in Exercise 20.6.1 above.*

Exercise 20.6.3 ***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 20.6.1, since you don't need to worry about what key was pressed.

Use a `posn` as the model. You *can* get this to work with an image as the model, but Exercise 20.6.4 builds on this one, and it's *much* easier if you use a `posn` as the model.

Exercise 20.6.4 ***Modify Exercise 20.6.3*** *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 20.6.5 ***Modify Exercise 20.6.1 or 20.6.3*** *so that if the user types the letter "q", the animation ends.*

Exercise 20.6.6 ***Modify Exercise 20.6.1 or 20.6.3*** *so that whenever the user clicks the mouse, the dot jumps immediately to the mouse location .*

Exercise 20.6.7 ***Modify Exercise 20.6.1 or 20.6.3*** *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 20.6.8 ***Modify Exercise 20.6.1*** *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 20.6.9 ***Modify Exercise 20.6.1*** *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`.

20.7 Colors

20.7.1 The `color` data type

The `posn` type is just one example of a *structure*, about which we'll learn more in Chapter 21. The `posn` type wraps up two numbers that need to travel together and must be kept in the right order (the position $(3, 5)$ is very different from $(5, 3)$).

Recall from Section 7.8 that the color of each pixel in an image is represented by three numbers: the red, green, and blue components. This is another classic situation that calls for a structure: three pieces of information that always travel together and need to be kept in the correct order. And in fact, DrRacket has a predefined structure named `color`, for which the following functions are predefined:

```
; make-color :  number(r) number(g) number(b) [number(alpha)] -> color
; color-red :  color -> number
; color-green :  color -> number
; color-blue :  color -> number
; color-alpha :  color -> number
; color?  :  anything -> boolean
```

You've been using `make-color` since Chapter 3 to *put together* colors, but now you can also use `color-red`, `color-green`, `color-blue`, and `color-alpha` to *take them apart*.

As before, each of the color components should be an integer from 0 through 255, and (as you know), the alpha component defaults to 255 if you leave it out. For example, `(make-color 0 0 0)` is black, `(make-color 255 255 255)` is white, `(make-color 0 200 250)` is a light greenish blue, *etc.*

Most of the functions in Chapter 3 that accept a color name also accept a `color`, so for example one could write

```
(overlay (text "hello" 20 (make-color 200 200 50))
         (ellipse 100 60 "solid" (make-color 50 50 200)))
```

Practice Exercise 20.7.1 *Play with this.*

Exercise 20.7.2 ***Write an animation*** *that displays a large disk whose color changes with the clock and the mouse position: the x coordinate of the mouse should control the amount of red, the y coordinate should control the amount of blue, and the clock should control the amount of green.*

Hint: As in Section 7.8.3, use `min` to make sure you never pass numbers larger than 255 to `make-color`.

There's a built-in function `color?` which tells whether something is a color, and a built-in function `color=?` which tells whether two colors are the same.

Exercise 20.7.3 *Suppose the `color=?` function weren't already provided. How could you write it yourself?* ***Develop a function*** *`my-color=?` that takes in two `color` structs and tells whether they're the same.*

Exercise 20.7.4 ***Extend*** *the function `my-color=?` so that each argument can be either a `color` or a string (color name). You'll need at least eight test cases: the first argument can be a string or a `color`, the second argument can be a string or a `color`, and the right answer can be `true` or `false`. Your function should also not crash if it's given a string that isn't a recognized color name:*

```
(check-expect (color=?  "forest green" (make-color 34 139 34))
              true)
(check-expect (color=?  (make-color 58 72 14) (make-color 58 72 14))
              true)
(check-expect (color=?  "plaid" "orange")
              false)
```

20.7.2 Building images pixel by pixel

Recall the `build3-image` function of Section 7.8. There's a simpler version, `build-image`, which takes in *one* function rather than three:

```
; build-image :  number(width) number(height) function -> image
; The function argument to build-image must have the contract
; whatever :  number(x) number(y) -> color
```

For example, Exercise 7.8.1 can be re-done using `build-image` as follows:

```
; red-gradient-pixel :  number(x) number(y) -> color
(check-expect (red-gradient-pixel 0 53) (make-color 0 0 0))
(check-expect (red-gradient-pixel 7 45) (make-color 35 0 0))
(check-expect (red-gradient-pixel 50 17) (make-color 250 0 0))

(define (red-gradient-pixel x y)
  ; x      a number
  ; y      a number
  (make-color (* 5 x) 0 0))

(build-image 50 50 red-gradient-pixel)
```

Exercise 20.7.5 ***Re-do*** *some of the exercises from Section 7.8 using* `build-image` *instead of* `build3-image`*.*

20.7.3 Building images pixel by pixel from other images

Similarly, there's a `map-image` function that's like `map3-image`, but takes in only *one* function:

```
; map-image :  function image -> image
; The function argument must have contract
; whatever :  number(x) number(y) color -> color
```

Exercise 20.7.6 ***Re-do*** *some of the exercises from Section 7.8 using* `map-image` *instead of* `map3-image`*.*

Exercise 20.7.7 ***Develop a function*** `replace-green-white` *which replaces every pure-green pixel in an image with a pure-white pixel.*

Hint: This will be easier than it would be with `map3-image` because you can use `color=?`.

Exercise 20.7.8 ***Develop a function*** `replace-with-white` *which takes in a color and an image, and replaces every pixel in the image that is the specified color with white. In other words, if we had this function, we could write* `replace-green-white` *as*

```
(define (replace-green-white pic)
  (replace-with-white "green" pic))
```

Hint: The helper function in this case needs to know what color to replace, *i.e.* the color parameter of `replace-with-white`. One way to do this is to `build-image/extra` or `map-image/extra` (look them up).

SIDEBAR:

It would be nice to generalize the function still farther to `replace-colors`, which takes in two colors and an image and replaces every pixel of the image that is the first color with the second color. Unfortunately, that would mean the helper function depends on *two* pieces of information from `replace-color`. One way to do that is to define a `struct` that contains two colors, as in Chapter 21, and use this `struct` as the "extra" parameter to `map-image/extra`. We'll see another, more general, way in Chapters 27 and 28.

The `get-pixel-color` function allows you to get the color of *any* pixel in a given image, rather than only the one at the same location as the pixel you're currently computing.

Practice Exercise 20.7.9 ***Look it up*** *in the Help Desk. Play with it. Go wild.*

20.8 Review of important words and concepts

Sometimes an animation (or other kind of program) needs to store *several* pieces of data together in a "package"; we call this *definition by parts*. DrRacket 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.

Another built-in structure type is `color`. If you think of a `posn` as a box with two compartments labelled x and y, then a `color` is a box with three compartments labelled `red`, `green`, and `blue`. The `map-image`, `build-image`, *etc.* functions allow you to operate on the colors of an image.

20.9 Reference: Built-in functions on posns and colors

In this chapter we've introduced or mentioned the following built-in functions:

- `make-posn`

- posn-x
- posn-y
- posn?
- make-color
- color-red
- color-green
- color-blue
- color?
- color=?
- map-image
- build-image
- map-image/extra
- build-image/extra
- get-pixel-color