Chapter 6

Animations in DrRacket

6.1 Preliminaries

Up to this point we've been working with static pictures. But it's much more fun and interesting to deal with pictures that change over time and interact with the user.

We think of an animation as showing, at all times, a simulated "world" or "model".[1] The world changes over time, and we watch it through the animation window. Our animations will use a built-in function (defined in the teachpack) named `big-bang` (because it "starts the world"). Here's its contract:

```
; big-bang :  image(start) event-handler ...  -> image
```

What does all this mean? The `big-bang` function takes in one or more parameters: the first parameter is the first image the animation should show, and any remaining parameters are "event handlers". An event handler is a kind of function, and since there are several different kinds of events to be handled, we need to specify which one goes with which kind of event. When the animation ends, the function returns the last image it showed. (Again, this is slightly simplified; we'll see more details in Chapters 8 and 10.)

An event handler is a function for the animation to call whenever a certain "event" happens. There are several kinds of "events": draw the screen, clock tick, mouse activity, keyboard activity, *etc.* but for our first example, the only one we need is called a "draw handler", whose job is to decide what to show on the screen at each step. For now, we'll use a built-in function named `show-it`, which takes in an image and returns it unchanged, as our draw handler. To tell DrRacket to use `show-it` as a draw handler, we put `(on-draw show-it)` as one of the "event handler" arguments to `big-bang`. This means that whenever DrRacket needs to redraw the screen, it will show the current image.

Worked Exercise 6.1.1 ***Develop an animation*** *that displays an unchanging green circle of radius 20.*

Solution: The starting image needs to be a green circle of radius 20; we know how to create such a thing with `(circle 20 "solid" "green")`. And since what we want

[1] There's also something called a "universe", which is several worlds running at the same time, either on the same computer or on different computers communicating over a network. We won't get to that in this book, but the `picturing-programs` library provides everything you need. If you want to learn to write such programs, first get through Chapter 10, then open DrRacket's Help Desk and search for "multiple worlds".

to show in the animation window is exactly this picture, we'll use `show-it` as our draw handler. The result is

```
(big-bang (circle 20 "solid" "green") (on-draw show-it))
```

Try this, either by typing it into the Interactions pane and hitting ENTER, or by typing it into the Definitions pane and clicking "Run". It should bring up a separate window with a green dot in it; when you click the "close" box of this window, the window goes away and `big-bang` returns the picture of the green dot. ■

Practice Exercise 6.1.2 ***Try*** *some different pictures.*

Exercise 6.1.3 ***Develop an animation*** *that displays an unchanging green circle of radius 20 in the center of a square white window 100 pixels on a side.*

Hint: Use either `place-image` or `overlay` to put the circle onto a background built using `rectangle`.

Specifying window size
In several of our animations so far, we've used `overlay` or `place-image` to place a picture onto a background, which is usually built by `rectangle`. If all you want is to have a larger animation window, there's a simpler way: specify the width and height of the window as additional arguments to `on-draw`.

```
(big-bang pic:calendar (on-draw show-it 300 200))
```

and see what happens. It should create an animation window, 300 pixels wide by 200 high, with a calendar in the top-left corner.

Of course, as you've already seen, these extra arguments to `on-draw` aren't required; if you leave them out, the animation window will be just big enough to hold your starting picture.

Practice Exercise 6.1.4 *Make up several examples like the above with pictures of various sizes and animation windows of various sizes. See how they behave.*

6.2 Tick handlers

Of course, these "animations" aren't very interesting. To get one to change over time, we need another *event handler*: specifically a *tick handler* , *i.e.* a handler which will be called "every time the clock ticks". You tell `big-bang` that you're giving it a tick-handler by saying `(on-tick` ***`function-name interval`*** `)` as one of the arguments to `big-bang`. The *interval* parameter is how many seconds there should be between one "tick" and the next; if you leave it out, the animation will run as fast as it can. The *function-name* parameter is the name of any function that takes in an old image and returns a new one. Ordinarily, you'll write a function yourself to do this job, but for this example, we'll use one that's already built in.

Worked Exercise 6.2.1 ***Develop an animation*** *of a calendar that rotates* 90° *clockwise every half second in the center of a 100x100 window.*

Solution: The starting picture is our calendar picture, `overlay`-ed into the middle of an empty 100x100 box. The interval is half a second. For the tick handler, we need a function that takes in an image and returns that image rotated clockwise 90°. Conveniently enough, `rotate-cw` does exactly this, so our handler will be `(on-tick rotate-cw 1/2)` and the whole animation becomes

```
(big-bang (overlay pic:calendar (rectangle 100 100 "solid" "white"))
          (on-draw show-it)
          (on-tick rotate-cw 1/2))
```

Try this.

By the way, you could also have said

```
(big-bang (overlay pic:calendar (rectangle 100 100 "solid" "white"))
          (on-tick rotate-cw 1/2)
          (on-draw show-it))
```

and this would work too: the various handlers can be specified in *any order*, as long as the starting picture comes first.

■

Practice Exercise 6.2.2 *What would happen in Exercise 6.2.1 if you skipped the* `overlay` *and the 100x100 rectangle, and simply used the additional arguments to* `on-draw` *to make the window 100x100?* ***Try it.*** *Do you understand why it did what it did?*

Exercise 6.2.3 ***Develop an animation*** *of a picture of your choice that flips upside down every 1.5 seconds in the center of the window.*

Practice Exercise 6.2.4 ***Make up*** *some other variations: different time intervals, different pictures, different functions in place of* `rotate-cw`*. (Note that the only functions that make sense here are functions that take in one image and return another — for example,* `rotate-cw` *and* `flip-vertical` *but not* `beside` *or* `overlay/xy`*.)*

What happens if you change the solution to Exercise 6.2.1 to use a colored background? What if the background isn't a square (say, it's wider than it is high)? What if the calendar isn't in the center? If you don't like the results of some of these, we'll see later how to fix them.

SIDEBAR:

Technically, `on-tick` and `on-draw` aren't functions, but rather something called *special forms.* What this means in practice is that you can't use them anywhere except as an argument to `big-bang`.

6.3 Common beginner mistakes

Leaving out the draw handler

If you write an animation with no `on-draw` clause, like

```
(big-bang pic:calendar (on-tick rotate-cw 1/2))
```

then DrRacket doesn't know how you want to show the animation window, so it doesn't. Depending on what version of DrRacket you have, this may produce an error message, or the animation may run but with no animation window (you'll have to hit the "Stop" button in the top-right corner of the DrRacket window).

There actually are times when you want to run an animation without displaying it — *e.g.* when it's communicating over the network with other computers, and *they're* displaying it — but we won't get to that in this book.

Testing `big-bang` with `check-expect`
A student wrote the following:

```
; rotating :  image -> image
(check-expect (rotating pic:calendar)
  (big-bang (overlay pic:calendar
                     (rectangle 200 200 "solid" "white"))
            (on-draw show-it)
            (on-tick rotate-cw 1/2)))
(define (rotating pic)
  (big-bang (overlay pic
                     (rectangle 200 200 "solid" "white"))
            (on-draw show-it)
            (on-tick rotate-cw 1/2)))
```

This is all legal. It passes a syntax check, it runs, and it shows a rotating calendar on the screen. In fact, after I close the animation window, it shows *another* rotating calendar on the screen. After I close the *second* animation window, it tells me that the program failed its one test case. What's going on here?

The above code calls `(rotating pic:calendar)`, which calls `big-bang`, which starts an animation. When a user closes this animation window, it returns from `rotating`, and `check-expect` calls `big-bang` again to construct the "right answer"; this starts *another animation*. When the user closes *this* animation window, `check-expect` compares the result of the first `big-bang` with the result of the second.

Recall from Section 6.1 that the result of `big-bang` is the last image it showed in the animation window. So unless both animations happen to end with the calendar pointing the same direction, the results won't match and `check-expect` will say the test failed. In general, `check-expect` **is not useful on the results of** `big-bang`.

This student has made things much more complicated than they need to be. I seldom call `big-bang` inside a function definition at all; instead, I **call `big-bang` directly in the Definitions or Interactions window, *not* inside a function definition**. So instead of the eleven lines the student wrote above, I would write the three lines

```
(big-bang (overlay pic:calendar (rectangle 200 200 "solid" "white"))
          (on-draw show-it)
          (on-tick rotate-cw 1/2))
```

If you *really* want to define a function that runs a particular animation, so you have a shorter name for it, the above definition is a good one: it takes in an image, and puts that image rotating every half second in the middle of a 200x200 animation window. But don't bother writing test cases for it, since we're interested in the function for the animation it runs, not for the result it returns.

6.4 Writing tick handlers

In most cases, you want to do something more complicated when the clock ticks, something for which there isn't already a built-in function like `rotate-cw`. So we *write one*.

Worked Exercise 6.4.1 ***Develop an animation*** *of a calendar that starts in the top-left corner of a window and moves 10 pixels to the right every second.*

Solution: Obviously, we'll need a tick handler with an interval of 1 second. It'll be convenient to specify the size of the window with something like

```
(on-draw show-it 500 100)
```

(note that we've left lots of width so there's room for the calendar to move right). The starting picture can be just `pic:calendar`. What's not so obvious is how to move a picture to the right by 10 pixels; there isn't already a built-in function that does that. So we'll write one.

Following the usual design recipe, we'll start with the **contract**. Let's name our function `move-right-10`. In order to work as a tick handler, it must take in an image and return an image, so the contract is

```
; move-right-10 :  image -> image
```

The purpose is obvious from the function name, so we'll skip the purpose statement.

Next we need some **examples**. Any image should work, e.g.

```
(move-right-10 pic:calendar)
```

But what should the right answer be? We could move the calendar right 10 pixels by putting it to the right of something 10 pixels wide with `beside`. What should we put it beside? A rectangle 10 pixels wide should do the job...but how high? We don't actually want to *see* a rectangle there, so let's make it 0 pixels tall (which makes it invisible; it really doesn't matter whether it's outlined or solid, or what color it is.)

```
(check-expect (move-right-10 pic:calendar)
              (beside (rectangle 10 0 "solid" "white") pic:calendar))
(check-expect (move-right-10 (circle 3 "solid" "green"))
              (beside (rectangle 10 0 "solid" "white")
                      (circle 3 "solid" "green")))
```

The next step in the design recipe is a **function skeleton**. We know that the function will be named `move-right-10`, and it'll take one parameter that's a picture; in fact, let's name the parameter "picture". So our function skeleton looks like

```
(define (move-right-10 picture)
  ...)
```

Next, we need to add an **inventory**. We only have one parameter, so we'll write down its name and datatype:

```
(define (move-right-10 picture)
  ; picture     image
  ...)
```

The next step in the design recipe is filling in the **function body**. We know that the function body will use the parameter name `picture`, and from our examples we see that the right answer tends to look like

```
(beside (rectangle 10 0 "solid" "white") something)
```

where *something* is whatever picture you want to move to the right.

So our function definition becomes

```
(define (move-right-10 picture)
  ; picture    image
  (beside (rectangle 10 0 "solid" "white")
          picture)
  )
```

The complete definition window should now look like Figure 6.1.

Figure 6.1: Complete definition of move-right-10

```
 (require picturing-programs)
; move-right-10 :  image -> image

(check-expect (move-right-10 pic:calendar)
              (beside (rectangle 10 0 "solid" "white") pic:calendar))
(check-expect (move-right-10 (circle 3 "solid" "green"))
              (beside (rectangle 10 0 "solid" "white")
                      (circle 3 "solid" "green")))

(define (move-right-10 picture)
  ; picture    image
  (beside (rectangle 10 0 "solid" "white") picture)
  )
```

Now we can **test** the function by clicking the “Run” button.

Everything we just went through to define `move-right-10` was so that we could *use* the `move-right-10` function in an animation. The animation itself is now pretty simple:

```
(big-bang pic:calendar
          (on-draw show-it 500 100)
          (on-tick move-right-10 1))
```

Type this into the Interactions pane and see whether it works. ▮

SIDEBAR:

The idea of providing a function for the animation to call as necessary — *e.g.* whenever the clock ticks — is called “event-driven programming with callbacks.” That is, you’re providing a function, you call the animation, and it “calls back” your function whenever an interesting “event” happens. If you have friends or relatives who do computer programming, tell them this is what you’re studying.

Exercise 6.4.2 ***Develop an animation*** *which moves a picture of your choice* down *5 pixels every half second, starting at the top-left corner of the window.*

Exercise 6.4.3 ***Develop an animation*** *which moves a picture of your choice* to the left *by 3 pixels every half second, starting at the top-left corner (so the picture seems to fall off the left edge of the window).*

Hint: You can get the effect of moving to the left by cutting off the left-hand few pixels of the image. You'll want to start with either a fairly large picture, or one that's `place-image`d away from the left edge.

Exercise 6.4.4 ***Develop an animation*** *which starts with a small red dot at the top-left corner of the window, then replaces it with two red dots side by side, then with a row of four, then a row of eight, then a row of sixteen . . . doubling every three seconds.*

6.5 Writing draw handlers

As you recall, when you define a function you need to distinguish between *what's the same every time* (which tends to become the body of the function) and *what's different every time* (which tends to become the parameters of the function). Something similar happens in animations: we need to identify *what about the animation is always the same* and *what parts of the animation can change while it runs.* The latter is called the *model.*

In all the examples so far, the *model* was the complete contents of the animation window. Whenever that happens, we can use `show-it` as the draw handler.

But sometimes we want only *part* of the animation window to change: for example, exercise 6.5.2 has a stick-figure flipping upside down, somewhere on a background scene that doesn't change. In this case the *model* is only the stick-figure, and the draw handler has the job of embedding the model (the changing part) into the background image (the unchanging part).

A draw handler can be any function with contract `image -> image`; the image it takes in is the model, and the image it produces is the whole contents of the animation window. You can write such a function, then use it (rather than `show-it`) as an argument to `on-draw`.

Worked Exercise 6.5.1 *Develop an animation of a calendar that sits at coordinates (100, 40) of a 150x200 pink window and rotates clockwise every 1/2 second.*

Solution: Using what we've already seen, we could set the starting image to be just a calendar, which would make it easy to rotate using `(on-tick rotate-cw 1/2)` . . . but then it'll appear at the top-left corner of a white window rather than where we want it in a pink window. (Try it!)

Or we could set the starting image to be `(place-image pic:calendar 100 40 (rectangle 150 200 "solid" "pink"))` . . . but when we rotated it, it would rotate the *whole window* rather than rotating just the calendar in place. (Try it!)

We'll solve both of these problems by letting the model be *only the calendar*, rather than the whole window, and writing our own draw handler to place the calendar on the pink background. Let's name it `place-on-pink`. It's pretty straightforward:

Contract:

```
; place-on-pink :  image -> image
```

Examples:

```
(check-expect (place-on-pink pic:calendar)
              (place-image pic:calendar
                           100 40
                           (rectangle 150 200 "solid" "pink")))
(check-expect (place-on-pink (triangle 30 "solid" "blue"))
              (place-image (triangle 30 "solid" "blue")
                           100 40
                           (rectangle 150 200 "solid" "pink")))
```

Skeleton and Inventory:

```
(define (place-on-pink picture)
  ; picture      image
  ...
  )
```

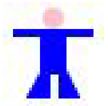
Body:

```
(define (place-on-pink picture)
  ; picture      image
  (place-image picture
               100 40
               (rectangle 150 200 "solid" "pink"))
  )
```

Once we've tested this, we can *use* it in an animation:

```
(big-bang pic:calendar
          (on-tick rotate-cw 1/2)
          (on-draw place-on-pink))
```

Note that we didn't need to specify the window size, because `place-on-pink` always returns a rectangle of the right size. ▮

Exercise 6.5.2 ***Find an outdoor scene*** *on the Web.* ***Develop*** *an animation in which a stick-figure*  *is positioned somewhere appropriate in the scene, and flips upside-down every second, staying in the same place; the background should* not *flip upside-down!*

Exercise 6.5.3 ***Modify*** *your solution to Exercise 6.4.4 so the row of dots is always centered in the window.*

Exercise 6.5.4 ***Develop an animation*** *which shows a picture of your choice at the center of the animation window,* rotating smoothly *(say, 5 degrees every 1/10 second).*

Hint: If you do this the obvious way, the picture may wander around a bit. This is because `overlay` lines up the center of the picture with the center of the background. But the "center" is defined as "halfway between the leftmost and rightmost points, halfway between the top and bottom points", and when the picture is rotated, this can refer to different parts of the picture than before. One way around this is to first overlay the picture on an invisible (*i.e.* the same color as the background) circle that completely contains it, so whatever point on your picture is at the center of the circle will stay put.

Exercise 6.5.5 ***Develop an animation*** *in which a stick-figure stands at the edge of a lake (part of the background scene) and does slow cartwheels (say, 5 degrees every 1/10 second). The figure's reflection should appear below him in the lake, upside down and doing slow cartwheels in the opposite direction (*i.e. *if the stick figure is rotating clockwise, the reflection should be rotating counter-clockwise).*

Hint: You don't have to do anything special to get the reflection to rotate in the opposite direction: if the stick-figure is rotating, then its upside-down reflection *will* be rotating the opposite direction.

Exercise 6.5.6 ***Develop an animation*** *in which a stick-figure appears in three different places, scattered around a background scene. One should be right-side-up, one rotated 180 degrees, and one flipped upside-down. The first and second should rotate slowly clockwise, while the third rotates slowly counter-clockwise.*

6.6 Other kinds of event handlers

As mentioned earlier, "tick handlers" are the simplest kind, but one can also provide handlers to respond whenever somebody types a key on the keyboard, or whenever somebody moves or clicks the mouse. The details are in Figure 6.2. (There's another kind of handler which we use to *stop* an animation; we'll discuss it in Chapter 14.)

Here's a simple example of a mouse handler:

Worked Exercise 6.6.1 ***Develop an animation*** *of a picture of your choice, moving right 10 pixels whenever the mouse is moved or clicked.*

Solution: Again, we'll use the calendar picture, with the same width and height as before. We need to install a handler using `on-mouse`, but (according to the contract in Figure 6.2) the function we give to `on-mouse` must take in a picture, two numbers, and a mouse-event, even though we're not interested in most of these things (and don't even know what a "mouse-event" is!). So we'll write a function similar to `move-right-10`, but taking in this extra information and ignoring it.

Contract:

```
; move-right-10-on-mouse :
;      image number number mouse-event -> image
```

Purpose statement:

```
; Just like move-right-10, but takes in three extra
; parameters and ignores them.
```

Examples: We've already come up with examples for `move-right-10`, so these should be similar, only with some extra arguments plugged in. Since we're writing the function and we know it will ignore the "mouse-event" parameter, it doesn't matter what we plug in there; we'll use strings.

```
(check-expect
  (move-right-10-on-mouse pic:calendar 318 27 "whatever")
  (beside (rectangle 10 0 "solid" "white") pic:calendar))
(check-expect
  (move-right-10-on-mouse (circle 3 "solid" "green") -3784 3.7 "blah")
  (beside (rectangle 10 0 "solid" "white")
          (circle 3 "solid" "green")))
```

Figure 6.2: big-bang and event handlers

The big-bang function has the contract

```
; big-bang :  image(start) handler ... -> image
```

tick handlers must have the contract

```
; function-name :  image (old) -> image (new)
```

They are installed by using `(on-tick` *`function-name interval`*`)` as an argument to `big-bang`. The *interval* is the length of time (in seconds) between clock ticks; if you leave it out, the animation will run as fast as it can.

key handlers must have the contract

```
; function-name :  image (old) key -> image (new)
```

The "key" parameter indicates what key was pressed; we'll see how to use it in Chapter 18.

They are installed with `(on-key` *`function-name`*`)`.

mouse handlers must have the contract

```
; function-name :  image (old)
;                  number (mouse-x) number (mouse-y)
;                  event
;                  -> image (new)
```

The first parameter is the old picture; the second represents the *x coordinate*, indicating how many pixels from the left the mouse is; the third represents the *y coordinate*, indicating how many pixels down from the top the mouse is; and the "event" tells what happened (the mouse was moved, the button was pressed or released, *etc.*); we'll see in Chapter 18 how to use this.

Mouse handlers are installed with `(on-mouse` *`function-name`*`)`.

draw handlers must have the contract

```
; function-name :  image (current) -> image
```

and are installed with `(on-draw` *`function-name width height`*`)`. If you leave out the *width* and *height*, the animation window will be the size and shape of the result the first time the draw handler is called.

An especially simple draw handler, `show-it`, is predefined: it simply returns the same image it was given. Use it when your model represents the *entire* animation window.

Function skeleton:

```
(define (move-right-10-on-mouse picture x y mouse-event)
  ...)
```

Fill in the inventory:

```
(define (move-right-10-on-mouse picture x y mouse-event)
  ; picture     image
  ; x           number
  ; y           number
  ; mouse-event whatever this is
  ...)
```

Fill in the body: We already know how to use `picture` to get the desired answer. The other three parameters are of no interest to us, so we just won't use them. Thus we can write

```
(define (move-right-10-on-mouse picture x y mouse-event)
  ; picture     image
  ; x           number
  ; y           number
  ; mouse-event whatever this is
  (beside (rectangle 10 0 "solid" "white") picture)
  )
```

Notice that the function body is the same as in `move-right-10`. Most computer scientists would consider this inelegant: if you already wrote it once, the computer knows it, so why should you have to write it again? A briefer and more elegant way to do it is to *re-use* the `move-right-10` function (assuming the definition of `move-right-10` is still in the definitions window, somewhere up above). The entire definitions window (including both definitions) should now look as in Figure 6.3.

Now that we have the `move-right-10-on-mouse` function, we can *use* it in an animation:

```
(big-bang pic:calendar
          (on-draw show-it 500 100)
          (on-mouse move-right-10-on-mouse))
```

█

SIDEBAR:

Notice that the animation only pays attention to mouse motion when the mouse is *inside the animation window*; in the above example, you can move around the rest of the screen without the calendar moving.

Exercise 6.6.2 ***Develop an animation*** *of a picture of your choice that moves right 10 pixels whenever a key is pressed on the keyboard.*

Hint: Obviously, you need a key handler. Just ignore the "key" parameter for now; we'll see how to use it in Chapter 18.

A more realistic example of using a mouse handler is the following:

Figure 6.3: move-right-10 and move-right-10-on-mouse

```
 (require picturing-programs)

; move-right-10 :  image -> image
(check-expect (move-right-10 pic:calendar)
              (beside (rectangle 10 0 "solid" "white") pic:calendar))
(check-expect (move-right-10 (circle 3 "solid" "green"))
              (beside (rectangle 10 0 "solid" "white")
                      (circle 3 "solid" "green")))

(define (move-right-10 picture)
  ; picture image
  (beside (rectangle 10 0 "solid" "white") picture)
  )

; move-right-10-on-mouse :
;      image number number mouse-event -> image
; Just like move-right-10, but takes in three
; extra parameters and ignores them.

(check-expect
  (move-right-10-on-mouse pic:calendar 318 27 "whatever")
  (beside (rectangle 10 0 "solid" "white") pic:calendar))
(check-expect
  (move-right-10-on-mouse (circle 3 "solid" "green") -3784 3.7 "blah")
  (beside (rectangle 10 0 "solid" "white")
(define (move-right-10-on-mouse picture x y mouse-event)
  ; picture     image
  ; x           number
  ; y           number
  ; mouse-event whatever this is
  (move-right-10 picture)
  )
```

Worked Exercise 6.6.3 ***Develop an animation*** *of a picture of your choice that moves with the mouse on a 500x300 background.*

Solution: This animation doesn't do anything on a regular schedule, so it doesn't need a tick handler. It obviously needs a mouse handler, whose job is to place the picture at the right place. Do we need a draw handler? The main reason we've written draw handlers in the past is to put a picture at a specific location, and the mouse handler is taking care of that, so let's try doing without a draw handler.

The trick will be writing a mouse-handler function that puts the picture in the specified location. Let's use the calendar picture again, and name our function `calendar-at-mouse`.

Contract: We've already chosen the name, and the fact that it's a mouse handler forces the rest of the contract to be

```
; calendar-at-mouse :  image(old-picture)
                       num(x) num(y) mouse-event -> image
```

Purpose statement: Recall from above that mouse coordinates are measured in pixels from the left, and pixels from the top. In practice, the coordinates of the mouse will always be positive integers, so let's make that assumption explicit. The function's purpose can be stated as

```
; Produces a picture of a calendar, with its top-left corner
; x pixels from the left and y pixels down from the top of a
; 500x300 white rectangle.
; Assumes x and y are both positive integers.
```

Examples: The function will always draw a calendar on a blank background, regardless of the old picture, so we can ignore the old picture. And we don't even know what a mouse-event is yet, so we'll ignore that too. Thus for our examples, it shouldn't matter much what arguments are plugged in for these. As for `x` and `y`, we should be able to plug in any positive integers and get something reasonable.

```
(check-expect (calendar-at-mouse pic:stick-figure 34 26 "huh?") ...)
```

But what "should" the answer "be"? Ignoring the "old picture" and the "mouse-event", the result should clearly be a calendar that's 34 pixels over from the left and 26 pixels down. The easiest way to put it there is with `place-image`:

```
(check-expect
  (calendar-at-mouse pic:stick-figure 34 26 "huh?")
  (place-image pic:calendar
               34 26
               (rectangle 500 300 "solid" "white")))
```

The reader is encouraged to come up with another example.

By the way, we're going to be using this 500x300 solid white rectangle a lot, so it makes sense to give it a name:

```
(define white-background (rectangle 500 300 "solid" "white"))
```

The example can then become

```
(check-expect
  (calendar-at-mouse pic:stick-figure 34 26 "huh?")
  (place-image pic:calendar 34 26 white-background))
```

Function skeleton: This is pretty straightforward from the contract:

```
(define (calendar-at-mouse old-picture x y mouse-event)
  ...)
```

Inventory: This too is pretty straightforward from the contract:

```
(define (calendar-at-mouse old-picture x y mouse-event)
  ; old-picture      image (ignored)
  ; x                positive integer
  ; y                positive integer
  ; mouse-event      whatever this is (ignored)
  ; pic:calendar     a picture we'll need
  ; white-background a picture we'll need
  ...)
```

Body: We've already decided to ignore `old-picture` and `mouse-event`, so we need only figure out how to use `x` and `y`. From the code in the "right answer", it seems that `x` and `y` should be the second and third arguments to `place-image`. The whole definition pane should now look like Figure 6.4.

Figure 6.4: Complete definition of calendar-at-mouse

```
 ; calendar-at-mouse :  image(old-picture)
;                       num(x) num(y) mouse-event -> image
; Produces a picture of a calendar, with its top-left corner
; x pixels from the left and y pixels down from the top.
; Assumes x and y are both positive integers.

(define white-background (rectangle 500 300 "solid" "white"))
(check-expect
  (calendar-at-mouse pic:stick-figure 34 26 "huh?")
  (place-image pic:calendar 34 26 white-background))
; whatever other examples you've come up with

(define (calendar-at-mouse old-picture x y mouse-event)
  ; old-picture image (ignored)
  ; x           positive integer
  ; y           positive integer
  ; mouse-event whatever this is (ignored)
  (place-image pic:calendar x y white-background)
  )
```

You can now **test** this definition by hitting "Run". Once we know that it works, we can **use** it in an animation.

To make sure there's enough room, we need to either provide a 500x300 starting picture (*e.g.* `white-background`) or specify a draw handler with dimensions of 500x300.

```
(big-bang white-background
          (on-draw show-it)
          (on-mouse calendar-at-mouse))
```

or

```
(big-bang (rectangle 0 0 "solid" "white")
          (on-draw show-it 500 300)
          (on-mouse calendar-at-mouse))
```

If everything works properly, whenever you move the mouse around within the animation window, a picture of a calendar will move with it. ■

6.7 Design recipe for an animation, first version

We've written a number of animations by now. What do they all have in common? I've listed the steps as a "design recipe for animations" in Figure 6.5.

Figure 6.5: Recipe for an animation, version 1

1. **Identify what handlers you'll need** (draw, tick, mouse, and/or key).
   - If your animation needs to change at regular intervals, you'll need a tick handler.
   - If your animation needs to respond to mouse movements and clicks, you'll need a mouse handler.
   - If your animation needs to respond to keyboard typing, you'll need a key handler.
   - You *always* need a draw handler, but in many cases you can get away with using `show-it`; see the next step.
2. **Identify the model**. What parts of the animation window can change while it runs, and what about it stays the same? If the model represents the *whole* animation window, you can use `show-it` as your draw handler; if it represents only *part* of the animation window, you'll need to write your own draw handler to build the whole picture from the part that changes.
3. **Write down the handlers' contracts** (using Figure 6.2). You can name these functions whatever you want but their contracts *must* be as in Figure 6.2.
4. **Develop each of these functions**, following the usual design recipe for each one. Don't go on to the next one until the previous one passes all of its test cases.
5. **Decide on the starting picture** the animation should start with.
6. **Decide on the width and height** (if they're not the same as those of the picture).
7. **Decide on the time interval between "ticks"** (if you have a tick handler).
8. **Call** `big-bang` with the starting picture and handlers (specified using `on-draw`, `on-tick`, `on-mouse`, and `on-key`). See whether it works.

Use this recipe to work the following exercises, each of which requires more than one handler.

Exercise 6.7.1 ***Modify*** *the animation of Exercise 6.6.3 so the picture slides left and right with the mouse, but stays at the same vertical position — say, halfway down the window — regardless of the mouse's vertical position.*

Exercise 6.7.2 ***Develop an animation*** *of a picture of your choice that moves to the right every second, and moves down whenever somebody types on the keyboard.*

Exercise 6.7.3 ***Develop an animation*** *of a picture of your choice that moves to the right every second, moves down whenever the mouse moves, and resets to its starting position whenever somebody types on the keyboard.*

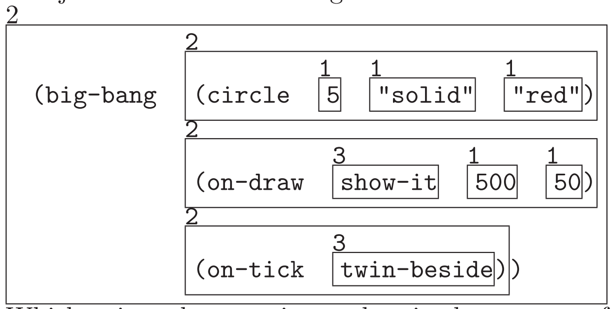
Hint: To "reset to its starting position", you'll need the model to become, again, the initial model that you gave as a first argument to `big-bang`.

6.8 A note on syntax

According to the syntax rules we've seen so far, the only place a function name can appear is just after a left parenthesis, as in `(above pic:calendar (flip-vertical pic:calendar))`. But we've been using the `on-tick`, `on-key`, and `on-mouse` functions in a strange way: we've been giving them the *names* of functions *as arguments.* It turns out that this is legal, because function names are really *identifiers*, just like variable names, so an expression like

```
(big-bang (circle 5 "solid" "red")
          (on-draw show-it 500 50)
          (on-tick twin-beside))
```

can be justified with a box diagram as follows:



Which raises the question: what is the contract for a function like `on-tick`? Its argument is supposed to be not an image, or a number, or a string, but a *function*, and its result is an "event handler" (let's not worry about what this is for now). So to oversimplify slightly, the contracts of these functions can be written

```
; on-draw :  function -> handler
; on-tick :  function -> handler
; on-key :  function -> handler
; on-mouse :  function -> handler
```

In other words, "function" and "handler" are data types, like "image", "number", "string", *etc.*

However, `on-draw`, `on-tick`, *etc.* don't actually work on *any* function; they only work on functions with the right contract. In order for a function to be used as a draw or tick handler, it must take in an image and return an image; to be used as a key handler, it must take in an image and a key and return an image; and to be used as a mouse handler, it must take in an image, two numbers, and a mouse-event, and return an image. Or in short:

```
  ; on-draw :  (image -> image) -> handler
  ; on-tick :  (image -> image) -> handler
  ; on-key :  (image key -> image) -> handler
  ; on-mouse :  (image number number mouse-event -> image) -> handler
```

We'll investigate this in more detail, and learn to write functions that work on other functions, in Chapter 28.

Exercise 6.8.1 ***Draw a box diagram*** *for the* `big-bang` *call of one of the animations you've written in this chapter.*

Exercise 6.8.2 ***Draw a box diagram*** *for one of the function definitions you've written in this chapter.*

6.9 Recording an animation

If you'd like to show off your animations to your friends, you can record one and save it as an "animated GIF file", which you can then put on your Web page. Visitors to your Web page can't interact it with the program, but they can see what appeared on the screen when you ran it yourself. To do this, add the line `(record?  true)` to your `big-bang` call, *e.g.*

```
  (big-bang (overlay pic:calendar (rectangle 100 100 "solid" "white"))
            (on-draw show-it)
            (on-tick rotate-cw 1/2)
            (record?  true))
```

After the animation ends, you'll be asked to select a folder to store the result into. I recommend creating a new folder for this purpose, as DrRacket will put into it not only an animated GIF file, but dozens or hundreds of PNG files representing individual frames of the animation. (You can safely delete these once you've got the GIF file.)

6.10 Review of important words and concepts

You can create open-ended, animated pictures in DrRacket, providing *event handlers* or *callback functions* for the animation to "call back" whenever an interesting "event" (such as a clock tick, keyboard or mouse activity, *etc.*) occurs.

Each handler takes in (among other things) the current *model* — the part of the animation that can change while it runs — and returns either a new model or the image to show in the window.

To install a handler, you use a *function that takes in another function*; this is a powerful programming technique, about which which we'll learn more later.

Many of the details in this chapter are specific to DrRacket, and to this particular animation package, but the ideas of model/view separation, event-driven programming and callback functions are common to almost all graphics systems and almost all programming languages. For example, the Java language has `ActionListeners`, `MouseListeners`, `KeyListeners`, `paint` methods, *etc.* corresponding roughly to the various kinds of event handlers we've used in this chapter. So once you know how to design an animation in DrRacket, most of the ideas will work the same way in other languages; only the language syntax will be different.

6.11 Reference: Built-in functions for animation

In this chapter, we've introduced seven new built-in functions:

- `big-bang`
- `on-tick`
- `on-draw`
- `on-mouse`
- `on-key`
- `show-it`
- `record?`

(Technically, most of these aren't really "functions", because they only make sense inside `big-bang`, but they *look* like functions.)