PART I

Running and writing programs

Chapter 1

Picture this! Drawing pictures in DrRacket

As you probably know, computers are very good at doing arithmetic. But frankly, arithmetic is pretty boring. So to get our first taste of computer programming, we'll work with pictures instead. (Behind the scenes, the computer is *really* using arithmetic to control these pictures, but we don't need to worry about that for now.)

Before trying anything in this chapter, make sure you've installed DrRacket and the `picturing-programs` teachpack, as described in section 0.3.

1.1 Working with pictures

1.1.1 Importing pictures into DrRacket

The easiest ways to get a picture to work with is to copy it from somewhere: a Web page, or a file that's already on your computer. Here's how.

Without quitting DrRacket, open a Web browser and find a Web page that has pictures on it. For example, many of the pictures used in this textbook are on the book Web site at `http://www.picturingprograms.com/pictures/`. And you can find lots of good examples on Google Image Search (`http://images.google.com`); for purposes of this chapter I recommend restricting your search to "small" images.

Right-click (or control-click) on a picture, and choose "Copy image". Now switch back to DrRacket, click in the Interactions pane (the lower half of the window) to the right of the "> " prompt, and paste. You should see the same image in the DrRacket window.

That's fine for pictures on Web pages. If you have picture files (GIF, JPEG, TIFF, etc.) already on the computer you're using, there's another way to get them into DrRacket. Click in the Interactions pane (to the right of the "> " prompt), then pull down the "Insert" menu and select "Insert image...." Find your way to the image file you want and select it; the image will appear in the DrRacket window.

1.1.2 The Interactions and Definitions panes

When you type anything into the Interactions pane and hit RETURN/ENTER, DrRacket shows you the "value" of what you typed. In many cases, that'll be exactly the same thing as you typed in. For example, if you import an image into DrRacket in either of the above

ways, and then hit the RETURN or ENTER key on the keyboard, you'll see it again.
Try this.

When you start *manipulating* pictures in section 1.2, things will get more interesting.

The upper half of the window is called the "Definitions pane". We'll get to it shortly, but for now, especially if you're using large pictures, you may want to hide it. Pull down the "View" menu and select "Hide Definitions"; now the Interactions pane takes up the whole window, and you can see more of your pictures.

1.1.3   Choosing libraries

Once you've installed a library such as `picturing-programs`, you still have to decide whether you need it for a particular problem. For everything in the rest of this chapter, and most of this book, you'll need `picturing-programs` . To tell DrRacket that you want to use that library, type

```
(require picturing-programs)
```

in the Interactions Pane and hit RETURN/ENTER.

(If your DrRacket is older than version 5.1, use

```
(require installed-teachpacks/picturing-programs)
```

instead.)

Any time you re-start DrRacket, or hit the "Run" button at the top of the window, DrRacket will erase everything that was in the Interactions pane, so you'll need to type this `require` line again before you can do anything else with pictures. We'll see a way to avoid repeating this in section 1.6.

1.2   Manipulating pictures

Now we'll learn to do some more interesting things with pictures: move them around, combine them into larger pictures, and so on.

For the examples in this section, I suggest copying a reasonably small, but interesting, picture from the web, such as this "calendar" picture from
`http://www.picturingprograms.com/pictures` .

Click to the right of the "> " prompt and type

```
( flip-vertical
```

then paste or insert an image as above. Then type a right-parenthesis to match the left-parenthesis at the beginning of what you typed, and hit ENTER/RETURN. You should see the image upside-down:

> (flip-vertical)

Practice Exercise 1.2.1 *Try the same thing, with* `flip-horizontal` *in place of* `flip-vertical`, *and the image will be reflected left-to-right.*

Practice Exercise 1.2.2 *Try* `rotate-cw`*, which rotates clockwise;* `rotate-ccw`*, which rotates counterclockwise; and* `rotate-180`*, which rotates by 180 degrees. See if you can predict (*e.g. *by drawing a rough sketch on paper) what each result will look like before you hit ENTER/RETURN.*

By the way, at the end of this chapter is a list of the picture-manipulating functions covered in the chapter.

1.2.1   Terminology

All the stuff you've typed (from the left parenthesis through the matching right parenthesis) is called an *expression*.

`rotate-cw`, `rotate-ccw`, and `rotate-180` are all *functions* (also called *operations* or *procedures*) which, given a picture, produce a different picture.

The picture you give them to work on is called an *argument* to the function.

The new picture you see as a result of applying the function to the argument is called the *value* of the expression.

By way of analogy, consider an English sentence like "Eat the banana." It contains a *verb*, "eat", which tells what to do, and an *object*, "the banana", which tells what to do it *to*. In computer programming, we use the words *function* and *argument* instead of *verb* and *object*, but the idea is similar.

A picture by itself, *without* parentheses or a function name, can also be thought of as an expression. It's an extremely simple expression in which there is nothing to "do"; the value of the expression is the expression itself. Such expressions (whose values are themselves) are called *literals*.

1.2.2   Combining pictures

Pick two different images of similar size and shape, both reasonably small. Click to the right of the "> " prompt and type (`above`, then an image, then another image, then a right-parenthesis. Hit ENTER/RETURN, and you should see one image stacked above the other. Try it again with the images in the opposite order. Note that whichever image you put in first ends up above the one you put in second.
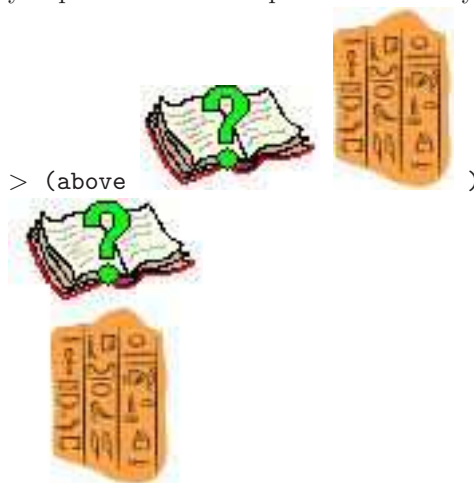
Practice Exercise 1.2.3 *Try the same experiment, but using the same image twice rather than two different images.*

Practice Exercise 1.2.4 *Try the same experiment with* `beside`, *which puts one image next to the other.*

Worked Exercise 1.2.5 *Try the same experiment with* `overlay`, *which draws two images in the same place, the first one overwriting part of the second. (If the first is larger than the second, you may not see any of the second at all.)*

   Be sure to try `overlay` *with two different images in both possible orders.*

Solution:

> (overlay   )

> (overlay   )

∎

Exercise 1.2.6
Now try the `above`, `beside`, *and* `overlay` *operations with* three *or more pictures. (For* `overlay`, *you'll want to pick a small picture as the first one, then larger and larger pictures, so you can see all of the results.)*

1.2.3   A Syntax Rule, Sorta

We can summarize what we've learned so far as follows:

Syntax Rule 0 *To do something to one or more images, type a left-parenthesis, the name of the operation you want to do, then the image(s) you want to do it to, then a right-parenthesis.*

Note that `beside`, `above`, and `overlay` are functions too, just like `flip-vertical`, `rotate-ccw`, *etc.*, but they work on *two or more* arguments rather than one; they wouldn't make sense applied to only one picture.

1.3 Making mistakes

In the course of typing the examples so far, you've probably made some mistakes. Perhaps you left out a left-parenthesis, or a right-parenthesis, or misspelled one of the operation names. This is nothing to be ashamed of: every programmer in the world makes mistakes like this every day. In fact, being a programmer is largely *about* mistakes: making them, recognizing them, figuring out how to fix them, figuring out how to avoid making the same mistake next time, making a different mistake instead.

In many math classes, you're given a large number of exercises to do, of which the odd-numbered ones have solutions given in the back of the book. What happens if you work out an exercise and your solution doesn't match the one in the back of the book? In many cases, all you can do is go on to the next problem and "hope and pray" that you get *that* one right.

Hope and prayer are not particularly effective in computer programming. Almost *no* computer program is exactly right on the first try. Rather than "hoping and praying" that the program will work, you need to develop the skills of *identifying* and *categorizing* mistakes, so that when you see a similar mistake in the future, you can recognize it as similar to this one, and fix it in the same way.

DrRacket provides a variety of useful *error messages*. Let's look at several of the most likely mistakes you might have made up to this point, make them *on purpose*, and see what message we get. That way, when you make similar mistakes by accident in the future, you'll recognize the messages.

1.3.1 Leaving out the beginning left-parenthesis

Ordinarily, when you type a right-parenthesis, DrRacket helpfully shades everything between it and the matching left-parenthesis.

> (`flip-vertical`)

Your first sign that you've left out a left-parenthesis is that when you type the right-parenthesis, it'll be highlighted in RED because DrRacket can't *find* "the matching left-parenthesis". To see this, try typing `flip-vertical`, then pasting a picture, and typing a right parenthesis.

> `flip-vertical`)

If you go ahead and hit RETURN/ENTER anyway, one of several things will happen. Some versions of DrScheme/DrRacket will treat `flip-vertical` and the picture as two

separate expressions: you'll see the word `flip-vertical`; then on the next line, the picture you pasted in; and on the next line, the error message

read: unexpected ')'.

In other versions, it just waits for you to type something reasonable. But nothing you can add after the right-parenthesis will make it reasonable. There are several things you can do: you can move (with the arrow keys or the mouse) to where the left parenthesis should have been, put it in, then move to the end and hit ENTER again; or you can hit BACKSPACE or DELETE until the right-parenthesis is gone (at which point you've simply typed two expressions on one line, and it'll give you the values of both).

1.3.2 Leaving out the ending right-parenthesis

Sometimes what you need to type between parentheses is longer than will fit on one typed line, e.g. several large pictures. So DrRacket allows you to hit ENTER/RETURN in the middle, and type or paste the next thing on the next line.

Note also that DrRacket will *automatically indent* the next line to line up nicely with the previous line. This is another clue that DrRacket thinks you're still inside an expression. If you don't want the line indented, you can hit DELETE/BACKSPACE a few times, but that doesn't change the fact that you're still inside an expression.

If you leave out the ending right-parenthesis, DrRacket thinks you've just gone to the next line and still want to type some more, so it'll quietly wait for you to finish. There is no error message, because DrRacket doesn't know that you've done anything wrong.
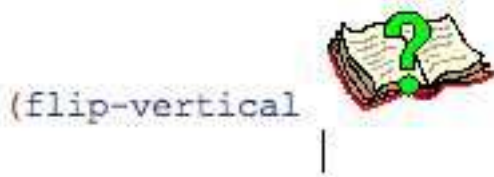
Fortunately, this is easy to fix, even if you've already hit ENTER/RETURN: just type the missing right-parenthesis, DrRacket will shade back to the left-parenthesis on the previous line, and you can hit ENTER/RETURN again to apply the operation.

1.3.3 Misspelling the operation name

Suppose you mistyped `flip-vertical` as `flip-verticle`. Any human would realize what was wrong, and guess that you actually meant `flip-vertical`. But computers aren't particularly good at "common sense" or guessing what you meant, so instead DrRacket produces the error message

> *reference to an identifier before its definition: flip-verticle*

What does this mean? "Identifier" simply means "name"; all the operations like `flip-vertical`, `above`, `overlay`, etc. are referred to by their names, but the name `flip-verticle` hasn't been defined. However, DrRacket leaves open the possibility that it *might* be defined in the future.

By the way, you might wonder why DrRacket isn't programmed to recognize that `flip-verticle` was probably supposed to be `flip-vertical`. This could be done, but if DrRacket had this "guessing" capability, it would eventually guess *wrong* without even telling you it was making a guess at all, and that kind of mistake is incredibly difficult to track down. The authors of DrRacket decided it was better to be picky than to try to guess what you meant. For the same reason, DrRacket is *case-sensitive*, that is, it doesn't recognize `FLIP-VERTICAL` or `Flip-Vertical`.

Likewise, DrRacket doesn't recognize names that have spaces in the middle, such as `flip - vertical`: it thinks you're calling a function named `flip` with `-` as its first argument and `vertical` as the second, which doesn't make sense.

1.3.4 Too few or too many arguments

Try typing `(flip-vertical)` and hitting ENTER/RETURN. You'll see the error message

> *procedure flip-vertical: expects 1 argument, given 0.*

This is a more helpful message, telling you precisely what went wrong: the `flip-vertical` operation (or "procedure") expects to work on an image, and you haven't given it one to work on.

Try typing `(flip-vertical`, then pasting in *two* images (or the same one twice), then typing a right-parenthesis. Again, the error message is fairly helpful:

> *procedure flip-vertical: expects 1 argument, given 2:...*

The rest of the error message tells what the arguments *were*, which isn't very helpful for images, but will be very helpful when we start working with numbers, names, etc.

1.3.5 Putting the operation and arguments in the wrong order

Suppose you wanted to put two pictures side by side, but had forgotten that the operation goes *before* the arguments; you might type something like

(beside)

You would get the error message

function call: expected a defined name or a primitive operation after an open parenthesis, but found something else

Again, this is a fairly specific and helpful message: the only things that can legally come after a left-parenthesis (for now) are function names, and a picture of a calendar is not a function name.

1.3.6   Doing something different from what you meant

All these error messages can get really annoying, but they're really your friends. Another kind of mistake is much harder to figure out and fix because there *is no error message.*

Suppose you wanted a left-to-right reflection of a particular picture, and you typed (`flip-vertical`, then pasted in the picture, and typed a right-parenthesis. You wouldn't get an error message, because what you've typed is perfectly legal. You would, however, get a *wrong answer* because what you've typed isn't what you meant. DrRacket can't read your mind, so it doesn't know what you *meant*; it can only do what you *said*. (This is one of the most frustrating things about computers, so much so that computer science students sometimes joke about a newly-defined function named `dwim`, for "Do What I Mean".) Of course, typing `flip-vertical` when you mean `flip-horizontal` is a fairly simple mistake, but in general these "wrong answer" errors are among the hardest ones to find and fix, because the computer can't give useful error messages to help you.

1.4   Getting Help

You've seen a number of builtin functions above, and you'll see many more in future chapters. Nobody can remember all of these, so (as mentioned in section 0.3.5) DrRacket has a "Help Desk" feature that allows you to look up a function by name. From the Help menu, choose "Help Desk"; it should open a Web browser window with a search box near the top. (By the way, this works even if you don't have a net connection at the moment.) Type the name of a function you want to know about, like `rotate-cw` or `above`, and it'll show you links to all the pages it knows about that function. (If there are more than one, look for one that's "provided from picturing-programs" or "provided from 2htdp/image".)

You can also type `picturing-programs` into the search box, and it'll show you a link to documentation about the whole teachpack.

1.5   More complex manipulations

Worked Exercise 1.5.1 *What would you do if you wanted to see a picture, beside its left-to-right reflection?*

Solution: You know how to get a reflection using `flip-horizontal`, and you know how to put one image next to another using `beside`, but how do you do *both*? You really want to put one image beside another, one of which is a reflection of the other.

Very simply, instead of pasting an image as one of the operands of the `beside` function, type in an expression involving `flip-horizontal`:
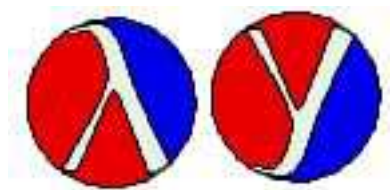
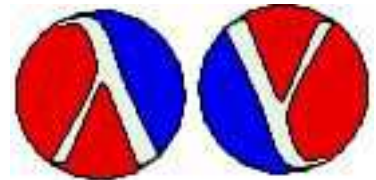> (beside (flip-horizontal))

∎

Since (flip-horizontal) would be a perfectly good expression in its own right, but it's also a *part* of a larger expression, we call it a *sub-expression*.

Exercise 1.5.2

Write an expression which displays a picture beside its top-to-bottom reflection.

Exercise 1.5.3

Write an expression which displays a picture beside its 180-degree rotation.

Exercise 1.5.4

Write an expression which displays four copies of a picture arranged in a two-by-two square.

Hint: There are at least two different ways to do this, using what you've seen so far. Either one is acceptable, as long as you type an expression that uses the smaller picture, and its value is the correct larger picture.

Exercise 1.5.5

Write an expression which displays four copies of a picture in a two-by-two square, each rotated differently: the top-right one should be rotated 90 degrees clockwise, the bottom-left one 90 degrees counter-clockwise, and the bottom-right one 180 degrees.

Hint: This expression will be fairly long and complicated; feel free to break it up over several lines. In particular, if you hit ENTER/RETURN after each right-parenthesis, DrRacket will automatically indent the next line in a way that indicates the structure of the expression: things inside more layers of parentheses are indented farther.

Hint: If you solve this problem the way I expect, it'll work well with square or nearly-square pictures, but won't look so good with long skinny pictures. We'll see how to improve it later.

1.6   Saving Your Work: the Definitions pane

When you type an expression in the Interactions pane and hit RETURN/ENTER, you immediately see the value of that expression. But as soon as you quit DrRacket, all your work is lost. Furthermore, even if you're not quitting DrRacket yet, sometimes you want to write expressions now and see the results later.

If you've hidden the Definitions pane earlier, show it again: pull down the "View" menu and choose "Show Definitions".

Click the mouse in the Definitions pane and type in the line

```
(require picturing-programs)
```

or, if you have an older version of DrRacket,

```
(require installed-teachpacks/picturing-programs)
```

as the first line of the Definitions pane. (Now that it's in the Definitions pane, you won't have to keep typing it again and again in the Interactions pane.) Hit RETURN/ENTER, and nothing will happen (except that the cursor will move to the next line). From now on, almost *every* Definitions Pane should start with that line.

On the next line of the Definitions pane, type in one of the expressions you've already worked with. Hit RETURN/ENTER. Type in another expression, and another. (These don't *have* to be on separate lines, but it's easier to keep track of what you're doing if they are. In fact, if they're long, complicated expressions, you might want to put a blank line or two in between them so you can easily see where one ends and the other begins.)

Now, to see how these expressions work, click the "Run" button just above the Definitions pane. Anything that was in the Interactions pane before will disappear and be replaced by the *values* of the expressions in the Definitions pane, in order. If any of them were illegal (e.g. mismatched parentheses, misspelled function names, etc.) it'll show an error message in the Interactions pane, and won't go on to the next expression.

If you've worked out a big, complicated expression (or several), and want to save it to use again tomorrow,

1. type the expression(s) into the Definitions window,

2. pull down the "File" menu,

3. choose "Save Definitions",

4. navigate to the appropriate folder on your computer,

5. type a suitable filename (I recommend a name ending with ".rkt"), and

6. click "Save" or "OK" or whatever it is on your computer.

Now you can quit DrRacket, double-click the new file, and it should start DrRacket again with those expressions in the Definitions window. Or you can double-click DrRacket, pull down the "File" menu, choose "Open...", and find the desired file to bring it into the Definitions window.

1.7 Working through nested expressions: the Stepper

When you develop a big, complicated expression and it doesn't work the way you expected it to, you need a way to see what it's doing along the way. The Stepper feature of DrRacket allows you to see the values of *sub-expressions*, one at a time, until you get to the whole expression.

For example, suppose you were working on exercise 1.5.2, and your (incorrect) attempt at the answer was

(beside (flip-horizontal))

If you type this into Interactions and hit RETURN/ENTER, or type it into Definitions and click the "Run" button, you'll get an answer, but not the *right* answer. To see what's going wrong, type the expression into the Definitions pane and, instead of clicking the "Run" button, click the "Step" button. You should see a new window, showing you the original expression on the left, and a slightly modified version of it on the right. In

particular, the *sub-expression* (flip-horizontal) on the left will be highlighted in green, while its *value*, another picture, will be highlighted in purple on the right. Everything else about the two expressions should be identical.

Worked Exercise 1.7.1 *Show the sequence of steps the Stepper would take in evaluating the expression*

(beside   (flip-horizontal   ))

At each step, underline the sub-expression that's about to be replaced.

Solution:

Step 1: (beside   (flip-horizontal))

Step 2: (beside   )

Step 3:

Exercise 1.7.2 ***Show the sequence of steps*** *the Stepper would take in evaluating the expression*

(beside (rotate-ccw) (rotate-cw))

1.8   Syntax and box diagrams

Recall rule 0:   *To do something to one or more images, type a left-parenthesis, the name of the operation you want to do, then the image(s) you want to do it to, then a right-parenthesis.*

In fact, as we've seen, things are a little more general and flexible than that: instead of putting images inside the parentheses, we can also put *sub-expressions* whose *values* are images. Indeed, these sub-expressions may in turn contain sub-expressions of their own, and so on.

At the same time, we've seen that certain attempts at expressions aren't grammatically legal. Computer scientists often explain both of these issues — how do you perform an operation, and what is or isn't a legal expression — at the same time, by means of *syntax rules*, and we now rephrase things in that style.

Syntax Rule 1 *Any picture is a legal expression; its value is itself.*

Syntax Rule 2 *A left-parenthesis followed by a function name, one or more legal expressions, and a right parenthesis, is a legal expression. Its value is what you get by applying the named function to the values of the smaller expressions inside it.*

Note that we can understand all the expressions we've seen so far by using a combination of these two rules, even the ones with several levels of nested parentheses, because rule 2 allows *any legal expressions* to appear as arguments to the function, even expressions constructed using rule 2 itself.

Let's illustrate this using "box diagrams". We'll start with an expression, then put a box around a sub-expression of it. Over the box we'll write a 1 or a 2 depending on which rule justifies saying that it is an expression.

**Worked Exercise 1.8.1** *Draw a box diagram to prove that the picture* *is a legal expression.*

Solution: Rule 1 tells us that any picture is a legal expression, so we put a box around it with the number 1 over it: ∎

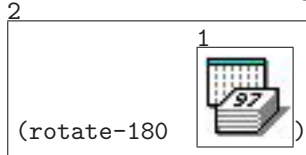**Worked Exercise 1.8.2** *Draw a box diagram to prove that*

(rotate-180)
is a legal expression.

**Solution:** We'll start from the inside out. The picture of the calendar is a legal expression
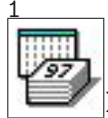
by rule 1, so we have (`rotate-180`)

Now that we know that the inner part is a legal expression, we can use Rule 2 (which requires a left-parenthesis, a function name, an expression, and a right-parenthesis) to show that the whole thing is a legal expression:

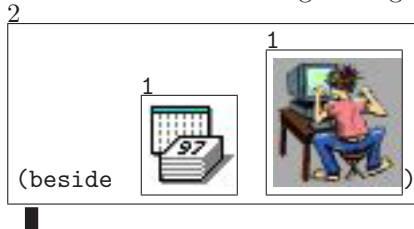**Exercise 1.8.3** *Draw a box diagram to prove that*

(rotate-cw *)*
is a legal expression.
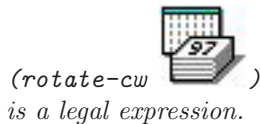
Worked Exercise 1.8.4 *Draw a box diagram to prove that*

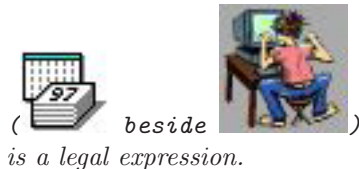*(beside* *)*
is a legal expression.
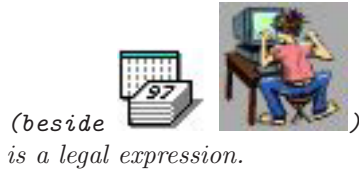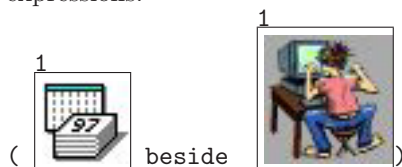
Solution: We need to use rule 1 twice:

Once we're convinced that both pictures are legal expressions, we need to use rule 2 to show that the whole thing is a legal expression:

Worked Exercise 1.8.5 *Draw a box diagram to prove that*

(*beside* *)*
is a legal expression.

Solution: We can use rule 1 twice to convince ourselves that the two pictures are legal expressions:


(beside   )

But now we're stuck: there is no rule in which an expression can appear between a left parenthesis and a function name. Since we are unable to prove that this is a legal expression, we conclude that it is *not* a legal expression. Indeed, if you typed it into DrRacket, you would get an error message:

> *function call: expected a defined name or a primitive operation name after an open parenthesis, but found something else.*

Whenever you type a left-parenthesis, Scheme expects the next things to be the name of an operation, and the calendar picture is not the name of an operation. ∎

Exercise 1.8.6 *Draw a box diagram to prove that*

(rotate-cw
is a legal expression.
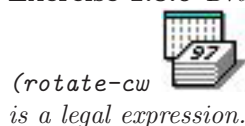
Hint: This should be impossible; it *isn't* a legal expression. But how far can you get? *Why* is it not a legal expression?

Exercise 1.8.7 *Draw a box diagram to prove that*

(rotate 5)
is a legal expression.
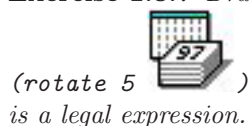
Hint: This too should be impossible. In fact, it *is* a legal expression, but not using the two rules you've seen so far; we'll add some more syntax rules later.

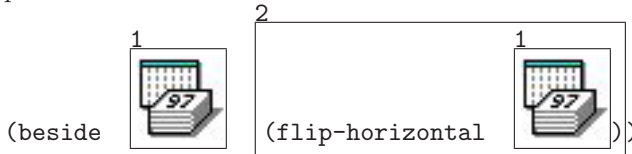**Worked Exercise 1.8.8** *Draw a box diagram to prove that*

(beside   (flip-horizontal))
is a legal expression.
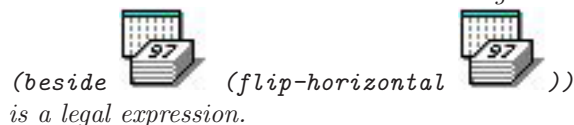
Solution: As usual, we'll work from the inside out. Each of the two pictures is obviously a legal expression by rule 1:
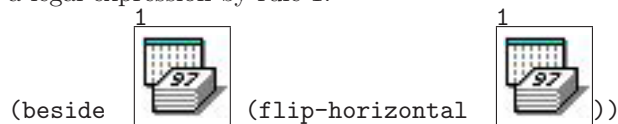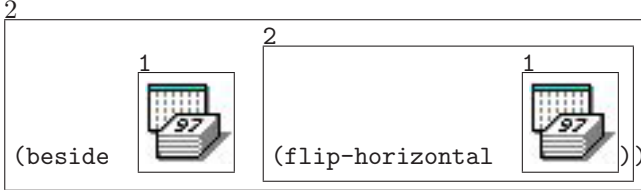
(beside   (flip-horizontal))
Next, we can apply rule 2 to the part of the expression starting with the inner left-

parenthesis:

Finally, we can apply rule 2 to the whole expression:

Exercise 1.8.9 *Draw a box diagram to prove that your solution to Exercise 1.5.2 or 1.5.4 is a legal expression.*

At this point you may be wondering how these "box diagrams" are supposed to help you write programs. The box diagram for a really simple expression (as in exercises 1.8.1 or 1.8.2), frankly, isn't very interesting or useful. But as the expressions become more complicated, the box diagrams become more and more valuable in understanding what's going on in your expression. Furthermore, every time you type an expression, DrRacket actually goes through a process (behind the scenes) very similar to these box diagrams, so by understanding them you can better understand DrRacket.

Ultimately, you should be able to avoid most syntax error messages by never typing in any expression that isn't grammatically legal; you'll know which ones are legal because you can draw box diagrams for them yourself.

1.9 Review of important words and concepts

Regardless of which pane you're typing into, you type **expressions** and (immediately or eventually) see their **values**.

A **literal** is an expression whose value is itself; the only examples you've seen so far are pictures copied-and-pasted into DrRacket, but there will be other kinds of literals in later shapters. More complicated expressions are built by applying a **function** or **operation** to one or more **arguments**, as in

In this example, `rotate-cw` is the name of a predefined function, and the literal picture is its argument. The parentheses around the whole expression let DrRacket know which function is being applied to which arguments. Note that different functions make sense for different numbers of arguments: `rotate-cw` only makes sense applied to one argument, while `beside` only makes sense for two or more. Other expressions can be even more complicated, containing smaller expressions in place of some of the pictures; these smaller expressions are called **sub-expressions**.

DrRacket has many built-in functions, and they each have to be called in a specific way with a specific number of arguments. Nobody memorizes all of them, so DrRacket's "Help Desk" feature allows you to look up a function by name.

1.10   Reference: functions that work on images

We've seen a number of built-in Scheme functions that work with images. These aren't really "important concepts", but here's a list of them that you can refer to later:

- flip-vertical

- flip-horizontal

- rotate-cw

- rotate-ccw

- rotate-180

- above

- beside

- overlay

We've also seen a special function named `require`, which is used to tell DrRacket that you need a particular library.