

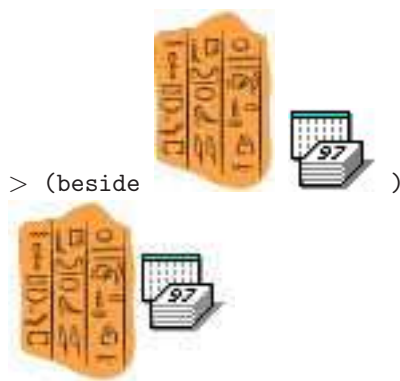
Chapter 3

Building more interesting pictures

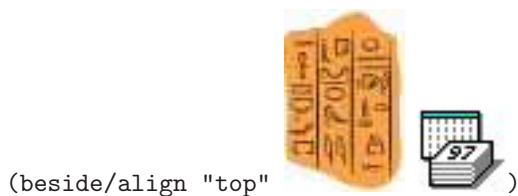
3.1 Other kinds of arguments

3.1.1 Strings as arguments



You may have noticed that if you put two images of different height **beside** one another, or two images of different width **above** one another, their centers are usually lined up:





Now let's try something slightly different. Type each of the following three expressions and observe the results:







(beside/align "bottom"  )

(beside/align "middle"  )



Try using `above` with two or more pictures of different widths; then try each of the following three expressions.

(above/align "right"  )

(above/align "left"  )

(above/align "middle"  )

You've just seen two new functions, `beside/align` and `above/align`, each of which expects an *extra argument* to indicate how you want things lined up. This extra argument isn't an image, it's a different kind of information called a *string* — a sequence of letters enclosed in double-quote marks. (We'll go into more detail on this in Section 3.4.) If you leave out the quotation marks, *e.g.*

(above/align **right**  )

then DrRacket will think `right` is an undefined variable, and will give you an error message.

There's also an `overlay/align` function with *two* string arguments: it expects the first argument to be either "left", "right", or "middle", and the second to be either "top", "bottom", or "middle". The third, fourth, *etc.* arguments should be images. **Play with these functions.**


3.1.2 Numbers as arguments

Next, try


(rotate 15 )

which rotates the image by 15 degrees instead of 90. **Play with this.**

Now, try


(scale 2 )

(scale 2/3 )

(scale 1.41 )

which makes an image larger or smaller. **Play with this.**

Note that the `rotate` and `scale` functions expect an extra argument that is neither an image nor a string — it's a *number*. Numbers can be written in several ways, as you saw above, but they cannot contain spaces (so, for example,

(scale 2 / 3 )

wouldn't work). We'll learn more about numbers in Racket in Chapter 7.

(There's also a `scale/xy` function that allows you to stretch or squash a picture by one factor vertically and a different factor horizontally. Look it up in the Help Desk.)

3.2 More mistakes

Now that we've seen functions that expect different types of arguments, there's a whole new world of things that can go wrong, with a whole new set of error messages to tell you about them. As before, let's make some of these mistakes *on purpose* so that when we make them by accident later, we'll recognize what's going on.

Try typing



```
(beside/align   "top" )
```

You would get the error message

beside/align: expected <y-place> as first argument, given: ...

because `beside/align` expects its *first* argument to be the string (specifically, a “y-place”, *i.e.* one of the strings “top”, “middle”, or “bottom”).

Likewise,



```
(overlay/align "top" "right"   )
```

produces an error message because it expects an *x-place* (*i.e.* either “left”, “right”, or “middle”) as its first argument, and “top” is none of those.

3.3 Creating simple shapes

So far, all your pictures have either been copied-and-pasted from other programs, or constructed from copied-and-pasted pictures using `beside`, `rotate-cw`, *etc.* In this section we'll learn to build simple geometric shapes from scratch. (As in the rest of the book, we've put the pictures in black and white to save on printing. If some of them don't make sense, try looking at the textbook Web site, where a lot of this stuff appears in color.)

One of the simplest geometric shapes is a rectangle. But there are lots of different possible rectangles, of different sizes, shapes, and colors. In addition, DrRacket allows

you to draw both *solid* rectangles  and *outline* rectangles . When we create a rectangle, we need to make all these decisions and tell DrRacket exactly what kind of rectangle we want.

DrRacket has a built-in function named `rectangle` that creates such pictures. It needs to be told the width and height of the rectangle (in pixels on the screen), whether it's solid or outline, and what color it should be, in that order:

```
> (rectangle 34 21 "solid" "green")
```



```
> (rectangle 15 36 "outline" "purple")
```



Practice Exercise 3.3.1 *Make up several examples of rectangles by plugging in different values for width, height, solid vs. outline, and color.*

If you try a color DrRacket doesn't recognize, you'll get an error message, but it won't hurt anything. Likewise, if you put anything other than "solid" or "outline" in the third argument position, you'll get an error message, but it won't hurt anything.

SIDEBAR:

The complete list of known color names is in the Help Desk; look up `color-database<?>`.

Practice Exercise 3.3.2 *What do you think would happen if you left out the color, e.g. `(rectangle 34 21 "solid")`? Type it in and find out whether you were right.*

What happens if you put the color first, e.g. `(rectangle "green" 34 21 "solid")`? Try various other mistakes, read the error messages, and make sure you understand them.

Practice Exercise 3.3.3 *Define a variable named `solid-green-box` whose value is a solid green rectangle, and another named `outline-blue-box` which is what it sounds like. Combine these in various ways using *above*, *beside*, *overlay*, etc.*

Another built-in function, `circle`, does exactly what you expect: it creates circles. Circles, like rectangles, can be either solid or outline, and of various colors, but rather than having a separate width and height, they have only a radius: for example, `(circle 10 "solid" "orange")` produces a solid orange circle of radius 10.

Practice Exercise 3.3.4 *Make up several examples of circles. Use *above*, *beside*, and *overlay* to compare a circle of radius 10 with a square whose width and height are both 10. How would you build a picture of a solid orange circle just fitting inside a solid blue*

square,  ?

Yet another built-in function, `ellipse`, has arguments similar to those of `rectangle`: width, height, solid or outline, and color. Try it.

Practice Exercise 3.3.5 *Make up several examples of ellipses. Show another way to*

construct a picture like .

The `triangle` built-in function has arguments similar to those of `circle`: a number representing the length of each edge of the triangle, the word "outline" or "solid", and a color name. It builds an equilateral triangle pointing up.


There are lots of other built-in functions like these. Look up the following in the Help Desk:

- `right-triangle`
- `isosceles-triangle`
- `rhombus`
- `regular-polygon`
- `star`
- `star-polygon`
- `line`
- `add-line`

Practice Exercise 3.3.6 *Make up several examples using these functions.*

Of course, nobody actually memorizes all these functions — I certainly haven’t! You should know that these functions exist, and how to look them up when you need them.



Exercise 3.3.7 *How would you construct a picture like ? (Note that the top edge of the triangle matches exactly the top edge of the square, and the bottom point of the triangle is exactly in the middle of the bottom edge of the square.)*

Hint: It *is* possible to do this, using what you’ve seen so far, with no math beyond elementary school.

3.4 Data types and contracts

In the previous chapter, you asked Racket to operate on images and produce images. In this chapter we’ve seen two additional kinds of information, or *data types*: numbers and strings.


You already have a pretty good idea of what numbers are, but strings may be new to you. In Racket (and C, and C++, and Java, and most other programming languages), a *string* is a sequence of letters, numbers, punctuation marks, spaces, etc. surrounded by double quote marks. The rules for what makes a string are similar to, but not quite the same as, those for what makes an identifier: an identifier can’t contain spaces, or certain punctuation marks, while *almost any* key you can type on the keyboard can go inside a string. (The main exception is the double-quote mark itself, as this indicates the *end* of the string. If you really want to put a double-quote mark inside a string, there is a way to do it, but we won’t go into that here.)

3.4.1 String literals and identifiers

When you type a string, enclosed in quotation marks, it’s a *literal* — that is, its value is just itself. By contrast, when you type a word that isn’t enclosed in quotation marks, DrRacket thinks of it as an *identifier*: if there’s a function or a variable by that name, that’s what it “stands for”, and if there isn’t, it doesn’t “stand for” anything.

A variable in Racket may “stand for” *any type* of information — a picture, a number, a string, or other types we’ll learn about later.

Practice Exercise 3.4.1 *Type each of the following expressions into a newly-opened interactions pane, one by one. For each one, predict what you think it will do, then hit ENTER and see whether you were right. **Explain** the results. If you get an error message (as you should for some of them), read and understand the error message.*

- `"hello"`
- `hello`
- `(define "author" "Bloch")`
- `(define author Bloch)`
- `(define author "Bloch")`
- `"author"`
- `author`
- `(define author "Bloch")`
- `(define calendar )`
- `(define age 19)`
- `"calendar"`
- `calendar`
- `"age"`
- `age`
- `"Bloch"`
- `Bloch`
- `(beside calendar calendar)`
- `(beside "calendar" "calendar")`
- `(define age 20)`
- `(define 20 age)`

3.4.2 Function contracts

As you've already seen if you tried exercise 3.3.2, each built-in function “knows” how many pieces of information, of what kinds, in what order, it should be given, and will reject other kinds of information. For example, `flip-vertical`, `rotate-cw`, *etc.* all expect to be given a single image, and will produce an error message if you give them *no* image, or *more than one* image, or a *non-image* such as a number or a string. (**Try** each of these mistakes and see what message you get.) This kind of pickiness may sometimes feel as though it's intended to annoy you, but really, what would it *mean* to “rotate” two pictures, or a number? The designers of DrRacket didn't see any obvious answer to these questions, so they simply made it illegal to do those things.

Similarly, `beside`, `above`, and `overlay` each expect to be given two or more images; they produce an error message if you give them too few images, or anything other than an image. The `beside/align` and `above/align` functions each expect a string and two or more images, while `overlay/align` expects two strings and two or more images.

Practice Exercise 3.4.2 *See what happens if you break these rules.*

The `rectangle` and `ellipse` functions expect to be given two numbers and two strings, in that order; furthermore, the first string must be either `"solid"` or `"outline"`, and the second must be a color name. If you give either of these functions the wrong number of things, or the wrong types of things, or the right types in the wrong order, it'll produce an error message.

Practice Exercise 3.4.3 *Try these various mistakes and see what different messages you can get.*

Obviously, you can't properly use any function unless you know how many arguments, of what types, in what order, it expects to be given. You also need to know what type of *result* it produces (so far, all our functions produce images, but that will change soon!). All of this information together is called a *function contract*: think of it as the function making a “promise” that “if you give me two numbers and two strings, in that order, I'll give you back an image.” A function contract can be described in words, as in the previous three paragraphs, but we'll have a *lot* of functions to deal with in this course, and that gets tiresome. Instead, we'll adopt a shorter convention for writing function contracts:

```
flip-vertical : image -> image
beside : image image ... -> image
above/align : string image image ... -> image
rotate : number image -> image
```

In this convention, we write the *name* of the function, then a *colon* (:), then the *type(s)* of the arguments, then an *arrow* (I usually use a minus sign and a greater-than sign, which together look sorta like an arrow), then the *type* of the result. We use ellipses (...) to indicate that there may be an indefinite number of additional arguments.

When a function takes several arguments of the same type, it often helps to say something about what each one means, so you remember to use them in the right order. I do this with parentheses:

```
rectangle: number(width) number(height)
           string("outline" or "solid") string(color) -> image
```

By reading this one contract, you can immediately tell that to create, say, an outlined blue rectangle 30 wide by 17 high, you should type `(rectangle 30 17 "outline" "blue")`

Practice Exercise 3.4.4 Write the function contracts for *ellipse*, *circle*, *triangle*, and *star-polygon*, using the standard convention.

There's nothing magical about this convention for writing function contracts, but following a common convention makes it easier for programmers to understand one another.


3.4.3 Comments

If you try to type function contracts into the DrRacket window, you'll get an error message because they're not legal expressions (according to rules 1-4). However, people frequently want to include function contracts in a Racket program, so they use *comments* to indicate something intended for the human reader, not for Racket.

Different programming languages specify comments in different ways, but every programming language I know of has *some* way to write comments. Here are three common ways it's done in Racket:

End-of-line comments

The most common way of making something a comment in Racket is to put a semicolon at the beginning of the line; everything else on that line will be *ignored completely*. You can write anything in a comment, including a letter to your grandmother:



```
( define calendar )
; Dear Grandma,
; I am learning to program, using the Racket language. So far
; I've learned to rotate and scale pictures, put pictures
; together in various ways, and make rectangles, circles,
; ellipses, triangles, and stars, and I can keep these pictures
; in variables to use later. However, this letter is in
; English, not in Racket, so I've "commented it out" to keep
; DrRacket from complaining about it. That's all for now!
; Love,
;                               Joe Student
(beside calendar calendar)
```

Of course, that's not a realistic use of DrRacket: there are much better programs around for writing letters to your grandmother! More likely, if you were using several built-in functions, you would write down their contracts in comments for easy reference:

```
; beside : image image ...-> image
; rectangle : number (width) number (height)
;           string ("outline" or "solid") string (color) -> image
; rotate-cw : image -> image
```

Multi-line comments

If you want to write several lines of comments in a row, it may be more convenient to use another kind of comment: type `#|` (the number sign and the vertical bar), and everything after that (even onto later lines) will be ignored, until you type `|#`.



```
( define calendar
  #|
  Here are several lines of commented-out contracts.
  beside : image image ...-> image
  rectangle : number (width) number (height)
             string ("outline" or "solid") string (color) -> image
  rotate-cw : image -> image
 |#
  (beside calendar calendar)
```

Practice Exercise 3.4.5 Write six different expressions on separate lines of the Definitions pane. “Comment out” the second one with a semicolon, and the fourth and fifth with `#|...|#`. Hit “Check Syntax”, and the commented parts should turn brown. Hit “Run”, and you should see the results of the first, third, and sixth expressions.

Expression comments

Yet a third kind of comment allows you to “comment out” exactly one expression, regardless of whether it’s a single line, part of a line, or multiple lines.

Practice Exercise 3.4.6 Type the following lines into the Definitions pane (assuming you’ve already got definitions of the variables `calendar`, `hacker`, and `solid-green-box`):

```
#: calendar hacker
#: (beside calendar hacker)
#: (beside
  hacker
  calendar
  ) solid-green-box
(beside calendar #: hacker solid-green-box)
```

On the first line, `calendar` is ignored, but `hacker` isn’t, so you get a picture of a hacker. The entire next line is ignored. Of the next four lines, `(beside hacker calendar)` is ignored, but the `solid-green-box` is not. And on the last line, the `hacker` is ignored and you get a picture of a calendar next to a solid green box.

Regardless of which kind of comment you use, DrRacket will automatically color it brown to show what parts it is ignoring.

3.4.4 Comments in Practice

There are two especially common reasons that we’ll use comments: to write down function contracts, and to temporarily “hide” part of a program while working on another part. For example,

```

; beside : image image ... -> image
; flip-vertical : image -> image
; image-width : image -> number
; rotate-180 : image -> image

```



```

(define calendar )
; (define two-cals (beside calendar calendar))
; (above two-cals (rotate-180 two-cals))
(above calendar (flip-vertical (scale/xy 1 1/2 calendar)))

```

The first four lines specify the contracts of functions we may be using. The next defines a variable. The next two are “commented out”: presumably either they already work, and we don’t want to be bothered with them while working on something else, or they *don’t* work as desired yet, and we’ll come back to them later.

SIDEBAR:

If you have a large section of program that you want to comment out temporarily, select all the relevant lines and use the “Comment Out with Semicolons” command on the Racket menu, and it’ll put a semicolon in front of each line. Likewise, the “Uncomment” menu command allows you to remove the semicolons from a whole bunch of lines at once.

3.5 More functions on pictures

3.5.1 Cutting up pictures

So you know how to get a picture of a circle:

```
(circle 30 "solid" "green")
```



How would you get a picture of the *upper half* of this circle? The `picturing-programs` library includes a function named `crop-bottom` which helps with this kind of problem. Its contract is

```
crop-bottom : image number -> image
```

It cuts off (or “crops”) the specified number of pixels from the bottom of a picture.

Worked Exercise 3.5.1 *Write an expression which produces the upper half of a solid green circle of radius 30.*

Solution: We already know that `(circle 30 "solid" "green")` produces the whole circle. How many pixels do we want to cut off the bottom? Well, the *radius* of a circle is the distance from the center to any edge of the circle, in particular the bottom edge. So if the radius is 30, then we want to cut off 30 pixels:

```
(crop-bottom (circle 30 "solid" "green") 30)
```





Exercise 3.5.2 *Here's a picture of me.*

Write an expression that chops the bottom 25 pixels off this picture.

The `picturing-programs` library also contains functions named `crop-top`, `crop-left`, and `crop-right`, which behave the same way but crop the top, left, or right edges respectively. There's also a `crop` function which allows you to pick *any* rectangular region from a picture. As usual, in exchange for more power, it's a bit harder to use. Look it up in the Help Desk.

Practice Exercise 3.5.3 *Play with these.*

SIDEBAR:

Actually, you don't need all four of these: any one would be enough, combined with things you already know. Try writing an expression that chops the *leftmost* 25 pixels off the picture of me, using `crop-bottom` but none of the other cropping functions.

The technique you probably used to do this is something mathematicians call *conjugation*. It's not difficult, and it's worth knowing for future purposes, but for the sake of convenience, we've given you all four cropping functions.

Exercise 3.5.4 *Write an expression which produces the bottom 45 pixels of an out-*



lined circle of radius 30.

Exercise 3.5.5 *Write an expression which produces the top-right quarter of a solid ellipse 50 wide by 30 high.*



Exercise 3.5.6 *Invent some other interesting pictures, using cropping together with the other functions you've already seen. Go wild.*

3.5.2 Measuring pictures

In order to know how many pixels to crop, it might be helpful to know how many pixels there *are*. The library provides two functions to help with this:

```
; image-width : image -> number
; image-height : image -> number
```

Notice that these are the first functions we've seen yet that return a *number* as their result, rather than an *image*. This will become more useful once we study arithmetic in Chapter 7.

Practice Exercise 3.5.7 *Find the widths and heights of some of your favorite pictures. Then write an expression to cut off the left one-third of a particular picture.*

Note that numeric results appear in the Interactions window just as image results have been appearing.

3.5.3 Placing images precisely



Here's a picture of me.

Suppose you wanted to take revenge on me for writing this book by blotting out my eyes:



How would you do this? Obviously, you need a black rectangle, and after some trial and error you might conclude that `(rectangle 45 15 "solid" "black")` is the right size and shape.



But when you try to overlay it on the picture, you get this picture instead: the blot is exactly centered in the picture, and my eyes aren't. You could make the blot bigger, but then it would blot out more of my nose and eyebrows. You could move the blot all the way to the top, or the bottom, or the left, or the right, using `overlay/align`, but none of those puts it exactly where you want it. To deal with this sort of situation, DrRacket provides a function named `place-image` that superimposes one picture at a *specified location* on another. Its contract is

```
place-image : image (foreground)
              number (horizontal offset) number (vertical offset)
              image (background) -> image
```

As with `overlay`, the second image is the “background”, and the first is superimposed “on top” of it. Unlike `overlay`, it only accepts *exactly two* images. It places the *center* of the foreground image *horizontal-offset* pixels from the left, and *vertical-offset* pixels down from the top, of the background image.

Experiment by plugging in various horizontal and vertical offsets, both positive and negative, to see what happens. What horizontal and vertical offsets do you need in order to blot out my eyes as in the picture above?

Exercise 3.5.8 Write a Racket expression, using `place-image`, to produce a solid blue rectangle 80×50 with a solid orange rectangle 30×20 whose bottom-left corner is at the



center of the blue box:

Recall that the `place-image` function places the *center* of the foreground image. Sometimes it’s more natural to place the top-left corner, or the bottom-right corner, *etc.* **Look up** the `place-image/align` function in the Help Desk.

In experimenting with `place-image`, you may have noticed that the result is always the same size and shape as the background image. Even if the foreground image laps over the edges of the background, the excess is cut off. This is often what you want, but not always. **Look up** the `overlay/xy` function in the Help Desk, and **play with it**.

3.5.4 Text

Suppose you wanted to draw a picture with words in it. In theory, you could build letters from tiny rectangles, circles, ellipses, lines, *etc.* and place them where you wanted them with `place-image`, but that would be a Royal Pain. Instead, DrRacket provides a function named `text` to do exactly this job. Its contract is

```
text : string (text to draw) number (font size)
      string (color) -> image
```

For example, try `(text "hello there" 12 "forest green")`. **Experiment** with different font sizes and colors.

Exercise 3.5.9 Write a Racket expression that produces my picture with the caption “Wanted!” in red near the bottom:



Exercise 3.5.10 Write a Racket expression placing a word (in blue) on a yellow back-



ground inside a purple border:

Exercise 3.5.11 Write a Racket expression for an interesting picture involving text, positioning, geometric shapes, rotation, scaling, etc. Go wild.

3.5.5 For further reading...

If you want to learn about other ways to manipulate pictures, go to the “Help” menu in DrRacket, select “Help Desk”, search for “2htdp/image”, and read the documentation. It’ll tell you about a number of other functions:

- square
- add-curve
- text/font
- underlay
- underlay/align
- underlay/xy
- empty-scene
- scene+line
- scene+curve
- scale/xy
- crop
- frame

Some of these may involve concepts — `define-struct`, Booleans, posns, and lists — that you haven’t seen yet, so don’t worry about them. As for the rest, play with them and have fun. Don’t worry about memorizing them all; just get a general idea of what’s there, so you can look it up when you need it.

3.5.6 Playing with colors

We’ve built a lot of images using color names like “purple”, “yellow”, etc.. This is great as long as the color you want happens to be one of the ones whose name DrRacket recognizes. But what if you want just a *little* bit more blue in your purple, or a *slightly darker* turquoise than “turquoise”?

The `picturing-programs` library, like most computer graphics systems, represents colors as combinations of red, green, and blue, each of which (for historical reasons) can be any whole number from 0 to 255. For example, black is made from 0 red, 0 green, and 0 blue; white is 255 red, 255 green and 255 blue; pure blue is 0 red, 0 green, and 255 blue; yellow is 255 red, 255 green, and 0 blue; and so on. You can mix your own colors by using the `make-color` function, and passing the result into a function like `rectangle`,

triangle, etc.in place of the color name:

```
> (rectangle 50 30 "solid"
      (make-color 180 100 150))
```



```
> (circle 20 "solid" (make-color 20 250 200))
```



This means the contracts for those functions are actually something like
`; circle : number string("solid" or "outline") string-or-color -> image`

Practice Exercise 3.5.12 *Play with this. Make up a bunch of shapes in various colors that you’ve mixed yourself, and compare them with shapes in predefined colors.*

If you want to see the numeric value of one of the standard named colors, use

```
; name->color : string -> color or false
; returns false if color name isn’t recognized
```

For example,

```
> (name->color "white")
(make-color 255 255 255 255)
> (name->color "forest green")
(make-color 34 139 34 255)
```

Note that this returns a color with *four* parts, not three. The fourth is called “alpha”, and it controls transparency: a color with an alpha of 255 is completely opaque, covering whatever was behind it, while a color with an alpha of 0 is completely transparent. If you call `make-color` with three arguments, it assumes you want the color to be opaque, so it fills in 255 for the fourth argument automatically, but if you wish you can make shapes that are “semi-transparent”.

Practice Exercise 3.5.13 *Play with this. Make a shape whose color has an alpha component of 120, and see what happens when you overlay or underlay it on another shape.*

We’ll learn more about the “color” data type in Chapter 20.

3.6 Specifying results and checking your work

When you try to do one of these exercises, the odds are that your first attempt won’t exactly work. How do you know? Well, you type the expression into the Interactions pane, hit RETURN/ENTER, and you get either an error message or a picture that isn’t what you were trying for. It may be interesting, it may be pretty, and often that’s how creative work starts — as an unsuccessful attempt to do something else — but it doesn’t fulfill the assignment. So you figure out what you did wrong, come up with something that addresses the problem, and try that to see if it actually works.

What about the “near misses”, in which the picture is *almost* what you wanted? In these cases, your mind may play tricks on you and pretend that what you got really is

what you wanted all along. (The first attempt at blotting out my eyes, in section 4.4, *almost* does the job, and you could convince yourself that it was OK, but in your heart you would know it wasn't quite right.) To prevent these mind tricks and make sure you come up with an expression that produces what you *really* wanted, I recommend **describing precisely, in writing** what the result should look like, before you try anything to get that result. This way if the result isn't correct, it's harder for your mind to fool itself into accepting it as correct.

For example, consider exercise 3.5.9 above. It's a reasonably precise description of what the result should be, but it still has some ambiguity. I didn't say specifically that the word "Wanted!" had to be superimposed on the picture; it could have been below the bottom of the picture. And I didn't say exactly what font size the word should be. To make this more precise, one might have written:

The word "Wanted!", in black, should be superimposed on the picture so there is little or no space between the bottoms of the letters and the bottom of the picture. The word should not lap outside either side of the picture, nor cover any part of Dr. Bloch's mouth (not to mention nose, eyes, etc.); however, it should be large enough to read easily.

Or one could go even farther and specify the font size of the word, the exact number of pixels between it and the bottom of the picture, etc. This sort of precise description of desired outcomes is easier when the result is a number, a string, or some other type that we'll see later in the course, but you can start developing the habit now.

3.7 Reading and writing images

If you've built a really cool picture, you may want to save it so you can use it outside DrRacket (*e.g.* on a Web page, or in a slide presentation or word-processing document). This is easy, using the `save-image` function, which takes in an image and a string, and stores the image in the file with that name. It uses the PNG image format, so it's a really good idea to choose a filename that ends in ".png". For example,

```
(save-image (triangle 40 "solid" "purple") "purple-triangle.png")
```

If you already have an image file, you can get it into DrRacket in any of several ways:

- As you already know, you can open it in a Web browser or other program, copy it, and paste into DrRacket.
- As you may also already know, you can use the "Insert Image" command (on the "Insert" menu of DrRacket).
- You can use the `bitmap` function, which takes in a string filename and returns the image. If the file doesn't exist, it'll return an image of 0 width and 0 height. For example,

```
(rotate-cw (bitmap "purple-triangle.png"))
```

By the way, although `save-image` always saves things in PNG format, `bitmap` can read things in a variety of formats, including PNG, GIF, JPG, *etc.*

3.8 Expanding the syntax rules

Based on the syntax rules you've seen in Section 2.4, most of the examples in this chapter aren't legal Racket expressions, because Syntax Rule 1 allows pictures, but not strings or numbers. A more accurate and inclusive version would say

Syntax Rule 1 *Any picture, number, or string literal is a legal expression; its value is itself.*

Worked Exercise 3.8.1 *Draw a box diagram to prove that*

```
(circle 10 "solid" "green")
```

is a legal expression.

Solution: With the new and improved Rule 1, we can recognize 10, "solid", and "green" each as legal expressions:

```
(circle 10 "solid" "green")
```

The whole thing then follows by Rule 2:

```
(circle 10 "solid" "green") ■
```

Exercise 3.8.2 *Draw a box diagram to prove that*

```
(define tri (triangle 15 "solid" "orange"))
```

is a legal expression (assuming tri isn't already defined).

Exercise 3.8.3 *Draw box diagrams to prove that*

```
(rectangle 40 26 "solid" "dark blue")
```

and

```
(rotate-180 (scale 2 (rectangle 20 13 "solid" "dark blue")))
```

are both legal expressions.

Are they the same expression? Type each of them into the Definitions pane and click "Run". Now click "Step" instead. What does "the same expression" mean?

3.9 Review of important words and concepts

In this chapter, we've seen how to use additional built-in functions to create pictures "from scratch", rather than copying and pasting them from the Web or other files. To tell these functions what we want them to do, we need to provide different kinds of information, *e.g. numbers* and *strings*. The categories "number", "string", "image", and many others that we'll see in later chapters are all called *data types*.

Each function "knows" how many arguments, of what types, in what order, it should be given, and what type of information it will produce. We typically write down this information, the *function contract*, in a standard format: the function name, a colon, the types of its arguments, an arrow, and the type of its result. Where there are multiple arguments of the same type, we may provide additional information about them in parentheses to keep track of which does what. However, this format is not Racket; it's only a convention among human programmers, so if you want to put function contracts into

a DrRacket window, you need to “*comment* them out” with one of the several kinds of Racket comments.

Since many of the functions in this chapter take in numbers or strings, we need to modify our syntax rules so that literal numbers and strings, as well as literal pictures, are treated as legal expressions, from which more complex expressions can be built. Henceforth we’ll use the expanded version of Syntax Rule 1, allowing not only images but numbers and strings.

3.10 Reference: Built-in functions for images

In this chapter, we discussed the following functions:

- above/align
- beside/align
- overlay/align
- rotate
- scale
- rectangle
- circle
- ellipse
- triangle
- star
- crop-bottom
- crop-top
- crop-left
- crop-right
- place-image
- text
- image-width
- image-height
- make-color
- name->color
- save-image
- bitmap

and mentioned the following, which you are invited to look up in the built-in DrRacket Help Desk:

- `place-image/align`
- `overlay/xy`
- `right-triangle`
- `isosceles-triangle`
- `square`
- `rhombus`
- `regular-polygon`
- `star-polygon`
- `radial-star`
- `line`
- `add-line`
- `add-curve`
- `text/font`
- `underlay`
- `underlay/align`
- `underlay/xy`
- `empty-scene`
- `scene+line`
- `scene+curve`
- `scale/xy`
- `crop`
- `frame`