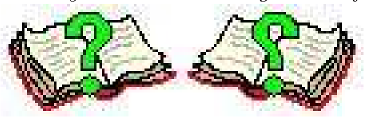
Chapter 4

Shorthand for operations: writing your own functions

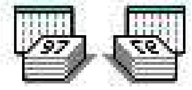
As a general rule in computer science, *if you write almost the exact same thing over and over, you're doing something wrong.* For example, we've already seen how to *define a variable* so we don't have to keep finding or building the exact same image over and over. In this chapter, we'll learn to write *programs* as a way to avoid writing similar expressions over and over.

4.1 Defining your own functions

Worked Exercise 4.1.1 *Write a Racket expression that produces a picture of a textbook, side by side with its right-to-left reflection:*

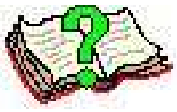


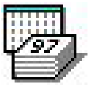
And then a picture of a calendar, side by side with its right-to-left reflection:



And then a picture of me, side by side with my right-to-left reflection:

Solution: Each of these should be routine by now: you can get the results by typing

(beside (flip-horizontal ))

(beside (flip-horizontal ))

```
(beside bloch (flip-horizontal bloch))
```

(the last one assumes that you've defined a variable `bloch` to hold a picture of me). ∎

Obviously, these are all very similar expressions: they all match the pattern

(beside *something* (flip-horizontal *something*))

with various different pictures plugged in as *something.* Now imagine you were working on a project in which you needed a *lot* of different pictures, *each* side by side with its right-to-left reflection. You could type basically this same expression over and over again, but it would be tedious, repetitive, and error-prone. Frankly, human beings don't do tedious, repetitive jobs very well without making mistakes. *But computers do!* It makes sense, therefore, to use the computer to *automate* this tedious, repetitive process, so we can concentrate our human abilities on more interesting tasks like picking the right pictures for our project.

The task can be broken down into two parts: the part that's the same every time, *i.e.*

(beside *something* (flip-horizontal *something*))

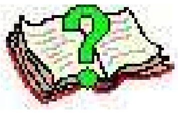
and the part that's different every time, *i.e.* what specific picture is plugged in as *something.* We'll teach the computer that repeated pattern, and whenever we want to use it, we'll simply provide the right picture to plug in for *something.*

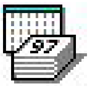
We need to provide a *name* for our new pattern, so we can refer to it by name and the computer will know which pattern we're talking about. Let's use the name `mirror-image`. Type the following in the Definitions pane (after the usual `(require picturing-programs)`):

```
(define (mirror-image something)
  (beside something (flip-horizontal something)))
```

and click the "Run" button.

Now to get a mirror-image effect on various pictures, we can just type

(mirror-image )

(mirror-image )

```
(mirror-image bloch)
(mirror-image (flip-vertical bloch))
```

etc.

We've *defined a new function* named `mirror-image`, and taught the computer that whenever you call that function on an image, it should match up the image you provided with the "place-holder"' word *something*, and act as though you had typed `(beside`, the image, `(flip-horizontal`, the same image again, and two right-parentheses. From now until you quit DrRacket, it will remember this pattern and use it whenever you call the `mirror-image` function. Now that the computer "knows" how to do this, we never have to worry about it again ourselves. We've taught the computer a new trick by combining things it already "knows" how to do; in other words, we've *written a program*.

By the way, there's nothing magical about the name `something`; we could equally well have said `x`, or `picture`, or `horse-chestnut`, as long as we spell the name the exact same way in all three places it appears. The name is just a "place-holder" (the technical word is *parameter*), to be matched up with whatever specific picture is provided when a user calls the function. The most sensible of these names would be *picture*, as that's what it's supposed to represent, so a better definition would be

```
(define (mirror-image picture)
  (beside picture (flip-horizontal picture)))
```

Practice Exercise 4.1.2 *Type the above definition (the version with the parameter name* `picture`*) into the Definitions pane of DrRacket, along with definitions of the variables* `book`*,* `calendar`*, and* `bloch`*. Click "Run", then type the examples*

```
(mirror-image book)
(mirror-image calendar)
(mirror-image bloch)
(mirror-image (flip-vertical bloch))
```

into the Interactions pane. After each one, hit ENTER/RETURN and you should see the correct result.

4.2 What's in a definition?

4.2.1 Terminology

The definition above is actually made up of several parts. The

```
(define (mirror-image picture)
```

part, which I've put on the first line, is called the *function header*, while the

```
  (beside picture (flip-horizontal picture)))
```

part, which I've put on the second line, is called the *function body.*

The function header is always made up of a left-parenthesis, the word `define`, a left-parenthesis, a function name (which must be an identifier that's not already defined, just like when you define a variable), one or more parameter names (identifiers), and a right-parenthesis.

The function body is simply an expression (plus a right-parenthesis at the end to match the one at the beginning of the header), in which the parameter names may appear as though they were defined variables. Indeed, if you have a parameter name in the header that doesn't appear inside the function body, you're probably doing something wrong.

4.2.2 Lines and white space

In each of the above examples, I've written the definition over two lines. You could write it all on one line:

```
(define(mirror-image picture)(beside picture(flip-horizontal picture)))
```

Or you could break it up over many lines, with as many extra blanks as you wish:

```
(
 define                           (
       mirror-image



    picture
                    )          ( beside
       picture       (
           flip-horizontal
                picture    )
   )
       )
```

Racket doesn't care: both of these will work equally well. However, neither of them is particularly easy for a human being to read. For the sake of human readers, most Racket programmers put the function header on one line, then hit ENTER/RETURN and put the function body on one more subsequent lines, depending on how long and complicated it is:

```
(define (mirror-image picture)
  (beside picture (flip-horizontal picture)))
```

or

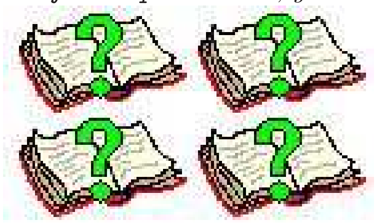
```
(define (mirror-image picture)
  (beside picture
          (flip-horizontal picture)))
```

Also notice that if you write a definition over more than one line, DrRacket will *automatically indent* each line depending on how many parentheses it's inside; this makes it easier for the human reader to understand what's going on. If you have a badly-indented definition (like the one above for mirror-image spread over eleven lines!), you can select it all with your mouse, then choose "Reindent All" from the Racket menu in DrRacket, and DrRacket will clean up the indentation (but not the line breaks or the extra spaces in the middles of lines).

Exercise 4.2.1 ***Define*** *a function named* `vert-mirror-image` *which, given an image, produces that image above its top-to-bottom reflection. As before, type your definition into the Definitions pane, after the definitions already there. Put the function header on one line, and the function body on the next line or two.*

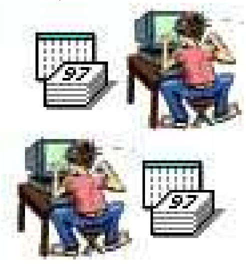
Click "Run", then ***test*** *your function by trying it on various images, such as* `book`, `calendar`, *and* `bloch`, *and checking that the results are what you expected.*

Exercise 4.2.2 ***Define*** *a function named* `four-square` *which, given an image, produces a two-by-two square of copies of it, e.g.*



. Put the function header on one line, and the function body on the next line or two. ***Test*** *your function by trying it on various images and checking that the results are what you expected.*

Worked Exercise 4.2.3 *Define a function named* `counterchange` *which, given two images, produces a two-by-two square with the first image in top-left and bottom-right positions, and the second image in top-right and bottom-left positions,* e.g.

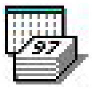
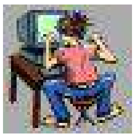


Solution: The only new feature of this problem is that the function takes in *two* images rather than one. To keep them straight, we need to have two different parameter names, *e.g.* `image1` and `image2`. The definition then looks like

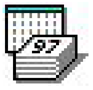
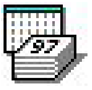
```
(define (counterchange image1 image2)
  (above (beside image1 image2)
         (beside image2 image1)))
```

Note that as usual, we've put the function header on one line and the function body on another — except that this function body is long enough and complicated enough that we split it across two lines. DrRacket's automatic indentation lines up the two calls to `beside` so the reader can easily see that they are both used as arguments to `above`.

Now we need to test the function. With the above definition in the Definitions pane, we hit the "Run" button, then type (in the Interactions pane)

`(counterchange`   `)`

and get the result above. For another example, try

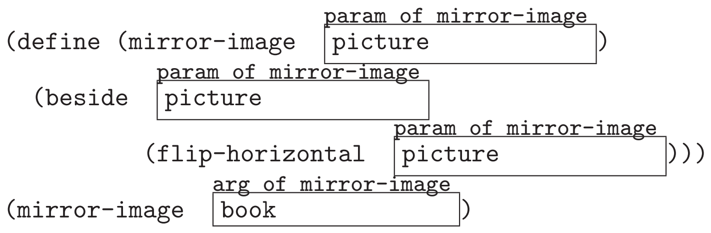
`(counterchange`  `(flip-horizontal`  `))`

Is the result what you expected? ■

Exercise 4.2.4 *Define a function named* `surround` *which, given two images, produces a picture with the second, first, and second side by side, i.e. the first image is "surrounded" on left and right by the second image. Follow the usual convention for code layout, and test your function on at least two different examples.*

4.3 Parameters and arguments

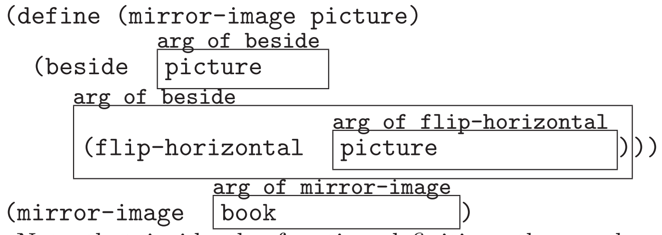
Some people use the words *argument* and *parameter* interchangeably. But there is a subtle, yet important, difference. An *argument* is a specific value in a function call, while a *parameter* is a "place-holder" introduced in the header part of a function definition. An argument may look like a number, string, image, variable, or more complex expression, while a parameter always looks like just a variable. For example,



```
                          param of mirror-image
(define (mirror-image     picture              )
                param of mirror-image
  (beside       picture
                                 param of mirror-image
          (flip-horizontal       picture              )))
                  arg of mirror-image
(mirror-image     book                )
```

The name `picture` is introduced in the "header" part of the definition of `mirror-image`, so it is a *parameter* of `mirror-image`. That same parameter is used two other times in the definition of `mirror-image`. The name `book` is used as an *argument* in a call of `mirror-image`.

For that matter, `mirror-image` is defined by calling two other functions, which have arguments of their own:



```
(define (mirror-image picture)
                arg of beside
  (beside       picture
          arg of beside
                              arg of flip-horizontal
          (flip-horizontal    picture                )))
                  arg of mirror-image
(mirror-image     book                )
```

Note that inside the function definition, the word `picture` can be thought of both as a parameter to `mirror-image` (because it appeared in the header of `mirror-image`'s definition), and also as an argument to `beside` and `flip-horizontal` (because it appears in calls to those functions).

To put it another way, an "argument" and a "parameter" both represent information passed into a function from its caller, but the "argument" is that information as seen from the caller's point of view, and the "parameter" is the information as seen by the function being called.

Exercise 4.3.1 *Consider the following Racket code:*

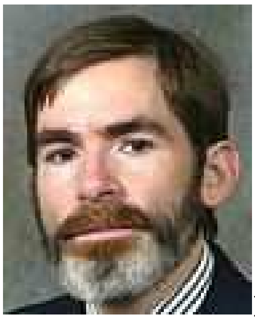
```
( define ( mystery x y )

  ( above ( flip-horizontal x ) y ) )

( mystery calendar book )

( mystery book calendar )
```

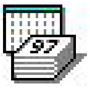
What words are used as parameters to which functions? What words are used as arguments to which functions?

Exercise 4.3.2 *Consider your solution and test cases for Exercise 4.2.1, 4.2.2, or 4.2.4. What words are used as parameters to which functions? What words (and expressions) are used as arguments to which functions?*

4.4 Parameters, arguments, and the Stepper

To get a clearer picture of what's going on, type the definitions of `bloch` and `mirror-image`, along with several examples of the latter, into the Definitions pane:

```
(define bloch
```

```
)
(define (mirror-image picture)
  (beside picture (flip-horizontal picture)))
(mirror-image
```

```
)
(mirror-image
```

```
)
(mirror-image bloch)
(mirror-image (flip-vertical bloch))
```

Now, instead of clicking “Run”, click the “Step” button. It’ll skip through the definitions, and start stepping at the expression

```
(mirror-image
```

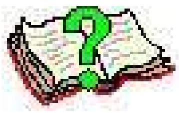
```
)
```

which expands to

```
(beside
```

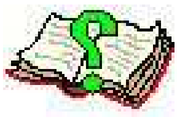
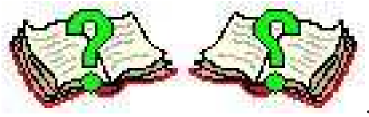
```
(flip-horizontal
```

```
))
```

This is because Racket matched up the argument  with the parameter `picture`, and everywhere inside the function definition that the parameter `picture` appeared, replaced it with that specific picture. The next few steps in the Stepper will behave exactly as you expect them to: the innermost expression

```
(flip-horizontal
```

```
)
```

is replaced by , and then the `beside` function combines this with the original picture to produce the result .

Step through the remaining examples, and make sure you understand what is being replaced with what, and why, at each step.

Practice Exercise 4.4.1 *Consider the Racket code from Exercise 4.3.1:*

```
(define (mystery x y)
  (above (flip-horizontal x) y))
(mystery calendar book)
(mystery book calendar)
```

In each of the two calls to mystery, ***tell*** *which argument is matched with which parameter, and* ***write down*** *(*without *using DrRacket) the sequence of steps of expansion that the Stepper would do. Type this code into DrRacket's Definitions pane, hit "Step" several times, and see whether you were right.*

Practice Exercise 4.4.2 *Repeat the previous exercise, replacing the mystery function with your solution to Exercise 4.2.1, 4.2.2, or 4.2.4.*

4.5 Testing a Function Definition

As this course goes on, you'll need to define hundreds of functions; defining a new function is one of the most common tasks for a programmer. But as you may have already found out, it's easy to make a mistake in defining a function. Before we can consider our task finished, we need to *test* the function for correctness.

Program testing can be an unpleasant job, as it often gives you the bad news that your program doesn't work. But it's much better for *you* to discover that your program doesn't work, than for your teacher or customer to make the discovery and penalize you for it. The worst thing that can happen is that we *think* our program is correct, but fail to spot an error that leads to somebody else getting wrong answers, which (in the real world) can cause airplanes or spaceships to crash, bridges to fall down, medical equipment to kill patients, *etc.*

So we owe it to ourselves to *try to break our own programs.* The harder we've tried without managing to break a program, the more confidence we have in turning it over to a teacher or customer. The more devious and malicious we are towards our own programs before releasing them, the less likely they are to be released with errors.

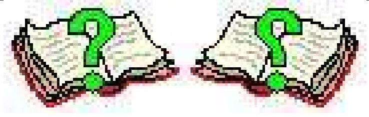
Recall from Chapter 1 that a function is considered correct only if it produces correct answers *for all possible inputs.* However, for a function like `mirror-image`, there are infinitely many possible pictures we could call it on; we can't possibly test it on every one. Fortunately, the function is simple enough that picking two or three different pictures is probably enough: it's unlikely to get those right without also getting everything else right.

4.5.1 Testing with string descriptions

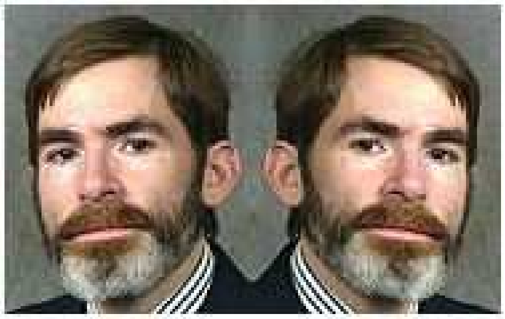
One way I often test a program is by writing several calls to the function in the Definitions pane, after the function definition itself. Each call is followed by a description, in quotation marks, of what the answer should have been. This way, when I hit "Run", DrRacket not only learns the function definition but shows the results of each call, followed by what it should have been. For example, the Definitions pane might look like (in part)

```
(define (mirror-image picture)
  (beside picture (flip-horizontal picture)))
(mirror-image book)
"should be two mirror-image books side by side, the left one with
a question mark, the right one with a backwards question mark"
(mirror-image bloch)
"should be two mirror-image pictures of me side by side,
faces turned slightly away from one another"
```

Hit "Run", and you'll get an Interactions pane looking like



```
"should be two mirror-image books side by side, the left one with
a question mark, the right one with a backwards question mark"
```



```
"should be two mirror-image pictures of me side by side,
faces turned slightly away from one another"
```

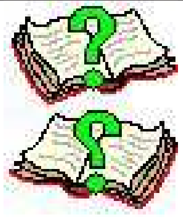
Since both actual answers match the descriptions of what they should be, we would tentatively conclude that the program works correctly.

Note that we've used strings to describe the "right answers". This works because when you type a quoted string by itself (not as an argument to a function), the value of that expression is simply the string itself.

Now suppose we had written a program incorrectly, for example

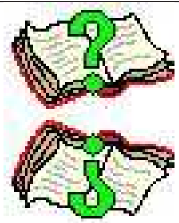
```
(define (vert-mirror-image picture)
  (above picture (flip-horizontal picture)))
(vert-mirror-image book)
"should be two books, one right side up with a question mark,
above another one upside down with an upside-down question mark"
```

Hit "Run", and the result will be



```
"should be two books, one right side up with a question mark,
above another one upside down with an upside-down question mark"
```

The answer the function *actually* produced doesn't match what we *said* it should produce, so something is wrong. Looking through the function definition (possibly with the aid of the Stepper), we would soon realize that the picture was being reflected horizontally, rather than vertically. We would correct this, hit "Run" again, and get



```
"should be two books, one right side up with a question mark,
above another one upside down with an upside-down question mark"
```

This time the actual answer *does* match the "right" answer, so things look good. One test case, however, is almost never enough to build real confidence in a function; **choose at least two or three test cases**.

Not all test cases are created equal. For example, if instead of using the picture of a book, we had written

```
(vert-mirror-image (rectangle 30 20 "solid" "blue"))
"should be a 30x20 solid blue rectangle, above another one upside down"
```

and then clicked "Run", the actual answer would have looked correct, even *without* finding and fixing the error, because a rectangle flipped upside down looks just like a rectangle flipped left-to-right (or not flipped at all). In other words, there would be an error in the program, but we wouldn't know about it because we weren't nasty enough to our program. **Choose test cases that are likely to reveal errors**.

Occasionally, you'll get an actual result that doesn't match what it "should be", and the function is actually right; your "should be" answer was wrong. For example, suppose you had defined `vert-mirror-image` correctly but written the test case

```
(vert-mirror-image calendar)
"should be a calendar, above another calendar flipped left-to-right"
```

When you hit "Run", you'll get an image that doesn't match the description of what it "should be", but this time it's because your "right answer" was wrong. That doesn't mean everything is OK: a correct program which looks wrong is almost as bad as an incorrect program which looks right. But this situation is at least easy to fix: once you're sure the program is right, just correct the "right answer". **This doesn't happen often**: when the actual answer doesn't match your "right answer", it's much more likely that the program is wrong. But keep this other possibility in mind.

4.5.2 Common beginner mistakes

I've frequently had students write "test cases" like the following (I've highlighted them for visibility):

```
(define (mirror-image picture)
  (beside picture (flip-horizontal picture)))
book
"should be two mirror-image books side by side,
the left one with a question mark, the right one
with a backwards question mark"
bloch
"should be two mirror-image pictures of me side by side,
faces turned slightly away from one another"
```

What's wrong with this? Well, when you type `book` or `bloch` by itself, DrRacket has no idea what you want to *do* with that picture. DrRacket certainly can't guess that you want to look at its mirror image (as opposed to rotating it clockwise, or putting it above itself, or thousands of other things DrRacket might be able to do with it). The fact that you've recently defined the `mirror-image` function does not mean that the `mirror-image` function will be called automatically; if you want to use a particular function, you have to call it by name. Just saying `book` will give you a picture of a book, no matter what functions you've defined.

Another common mistake in writing test cases:

```
(define (mirror-image picture)
  (beside picture (flip-horizontal picture)))
 (beside book (flip-horizontal book))
"should be two mirror-image books side by side,
the left one with a question mark, the right one
with a backwards question mark"
 (beside bloch (flip-horizontal bloch))
"should be two mirror-image pictures of me side by side,
faces turned slightly away from one another"
```

There are two things wrong with this. First, it misses the point of defining a new function: once you've defined a function, you shouldn't have to think about (or repeat) its body ever again. Second and more seriously, it *doesn't test the function* you defined. These examples test whether `beside` and `flip-horizontal` work correctly, but since they never actually use `mirror-image` itself, they don't tell whether or not *it* works correctly. If the `mirror-image` function had been written incorrectly (*e.g.* using `above` rather than `beside`), these test cases wouldn't show us that anything was wrong.

Many of my students in the past have balked at describing "what the right answer should be"; they would rather type in the test case, run it, see what the answer *is*, then (since the mean old professor insists on test cases) describe this answer and say that's what it "should be". **Don't do this!** These students will *never* discover any errors in their programs, because they're assuming the program is correct. They have completely missed the point of testing (and lost a lot of points on their homework grades, to boot!)

4.5.3 The `check-expect` function

If you've got a lot of test cases for a particular function, or if you have a lot of functions in the Definitions pane, it can be a lot of work to look through all the answers and compare them with their descriptions. DrRacket comes with a function named `check-expect` that automates this process: no matter how many test cases you have in the Definitions pane, it tells you instantly how many of the actual answers matched what you said they "should be", and which of them didn't.

Unfortunately, `check-expect` isn't smart enough to understand an English-language description of the right answer, so to take advantage of it, you have to build the *exact* right answer for each specific test case.

For example, consider the `vert-mirror-image` function from before. To test it using `check-expect`, we would replace the test case

```
(vert-mirror-image book)
"should be two books, one right side up with a question mark,
above one upside down with an upside-down question mark"
```

with

```
(check-expect (vert-mirror-image book)
              (above book (flip-vertical book)))
```

In addition, let's use the "bad test case" from section 4.5.1:

```
(check-expect
   (vert-mirror-image (rectangle 30 20 "solid" "blue"))
   (above (rectangle 30 20 "solid" "blue")
          (flip-vertical (rectangle 30 20 "solid" "blue")))
```

Practice Exercise 4.5.1 *Type a correct definition of `vert-mirror-image` into the DrRacket definitions pane, followed by the above `check-expect` lines. Click "Run" and see what happens.*

Now change the definition to be incorrect *— for example, use `flip-horizontal` instead of `flip-vertical` — but leave the `check-expect` the same. Click "Run" and see what happens.*

By the way, `check-expect` *is* "smart" in another way: you can put test cases using `check-expect` *ahead* of the definition and DrRacket won't complain that the function isn't defined yet. This doesn't work with "should be"-style test cases.

4.6 A new syntax rule

Why is something like

```
(define (mirror-image picture)
  (beside picture (flip-horizontal picture)))
```

legal Racket? Well, based on the syntax rules you've seen so far, it isn't: there is no way to draw a box diagram for it, justifying each box with one of the rules

Syntax Rule 1 *Any picture, number, or string 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.*

Syntax Rule 3 *Any identifier, if already defined, is a legal expression.*

Syntax Rule 4 *A left-parenthesis followed by the word `define`, a previously-undefined identifier, a legal expression, and a right-parenthesis is a legal expression. It has no "value", but the side effect of defining the variable to stand for the value of the expression.*

To define new functions, as we've just done, we need a new syntax rule:

Syntax Rule 5 *A left parenthesis followed by the word* `define`*, a left parenthesis, a previously-undefined identifier, one or more identifiers, a right parenthesis, a legal expression, and another right parenthesis is a legal expression. Think of it as anything matching the pattern*

```
(define ( new-identifier identifier ...)  expression )
```

This sort of expression has no "value", but the side effect of defining a new function whose name is the new-identifier*. Note that the parameter names from the function header can appear inside the function body as though they were defined variables.*

Notice the difference between Syntax Rules 4 and 5: a variable definition looks like

```
(define variable-name ...)
```

whereas a function definition looks like

```
(define (function-name parameter-names) ...)
```

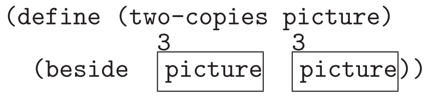
Racket can tell which one you mean by whether there's a left parenthesis after the `define`.

Worked Exercise 4.6.1 *Draw a box diagram to prove that*

```
( define ( two-copies picture )
  ( beside picture picture ))
```

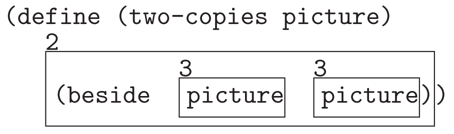
is a legal expression. Assume that `two-copies` *is not already defined.*

Solution: Since `picture` is one of the identifiers in the function header, it can appear inside the function body as if it were a defined variable. So the last two occurrences of the word picture are legal expressions, by Rule 3:



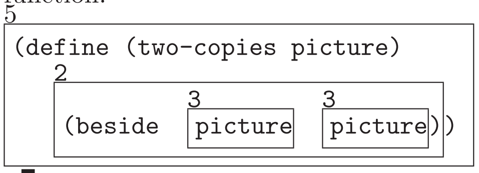
```
(define (two-copies picture)
  (beside picture picture))
```

Then the whole call to `beside` is a legal expression, by Rule 2:



```
(define (two-copies picture)
  (beside picture picture))
```

Finally, by Rule 5, we can recognize the whole thing as a legal expression defining a new function:



```
(define (two-copies picture)
  (beside picture picture))
```

■

Exercise 4.6.2 *Draw a box diagram to prove that*

```
(define (mirror-image picture)
  (beside picture (flip-horizontal picture)))
```

is a legal expression. Assume that `mirror-image` *is not already defined.*

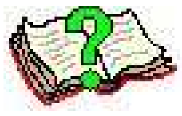
Exercise 4.6.3 *Draw a box diagram to prove that your solution to Exercise 4.2.1, 4.2.2, or 4.2.4 is a legal expression.*

4.7 Scope and visibility

You may have already noticed (perhaps by accident) that two different functions can have parameters with the same name (which I've highlighted), *e.g.*

```
(define (horiz-mirror-image pic )
  (beside pic (flip-horizontal pic )))
(define (vert-mirror-image pic )
  (above pic (flip-vertical pic )))
```

In fact, there could even be a global variable by the same name:

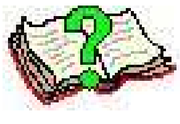
```
(define pic
```

```
)

(define (horiz-mirror-image pic )
  (beside pic (flip-horizontal pic )))
(define (vert-mirror-image pic )
  (above pic (flip-vertical pic )))

(rotate-cw pic )
(horiz-mirror-image pic )
(horiz-mirror-image (rotate-cw pic ))
```

There's nothing wrong with this: when a parameter name appears in a function header, it "hides" any other definition of that name that might already exist, until the end of the function definition.

By way of analogy, when I'm at home, the name "Deborah" always refers to my wife, even though there are lots of other Deborahs in the world. If I meant one of the other Deborahs, I'd have to specify a last name, *e.g.* the author Deborah Tannen; the name "Deborah" by itself still means my wife. Similarly, inside the body of `vert-mirror-image`, the word `pic` always refers to the parameter `pic` introduced in its header, regardless of whether there are other things named `pic` defined elsewhere.

Furthermore, there is *no way whatsoever* to refer to that parameter from outside the function; the parameter simply doesn't exist outside the function. (I suppose the best analogy to this would be if my wife never left the house, and I kept her existence a secret from the rest of the world....) The world outside the definition of `horiz-mirror-image` neither knows nor cares what parameter name it uses internally. Another result of this is that you can call `horiz-mirror-image` with an argument that happens to be named `pic`, or with some other argument; it makes no difference whatsoever.

Of course, if you *prefer* to use different parameter names in each function, there's nothing wrong with that either:



```
(define pic  )

(define (horiz-mirror-image pic2 )
  (beside pic2 (flip-horizontal pic2 )))
(define (vert-mirror-image pic3 )
  (above pic3 (flip-vertical pic3 )))

(rotate-cw pic )
(horiz-mirror-image pic )
(horiz-mirror-image (rotate-cw pic ))
```

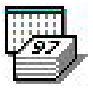
but it's an unnecessary complication. When choosing parameter names, I choose whatever makes the most sense inside this function, without worrying about whether there's something else with the same name somewhere else.

Computer scientists refer to the part of the program in which a variable is visible as that variable's *scope*. In the above example, the global variable `pic`'s scope is "everywhere, until you quit DrRacket"; the parameter `pic2`'s scope is "inside the definition of `horiz-mirror-image`", and the parameter `pic3`'s scope is "inside the definition of `vert-mirror-image`".

4.8 An analogy from English

We've encountered a number of new concepts in the past few chapters: *expression*, *function*, *argument*, *variable*, *data type*, *etc.* Some people may have an easier time understanding these terms by analogy to English.

4.8.1 Proper nouns and literals

In English (and most other natural languages), one can name an individual person, place, or thing with a *proper noun*, e.g. "Joe Smith", "New York", "Harvard University", "Rover", *etc.* The analogous concept in Racket (and most other programming languages) is a *literal*, *e.g.* a picture like  , a number like 7, or a string like `"hello there"`. All of these represent specific pieces of information, which you can tell simply by looking at them.

4.8.2 Pronouns and variables

English also has pronouns like "he", "she", "it", *etc.*: words that represent an individual person, place, or thing, but only in context. If I say "He is very tall," you know that I'm talking about a person, but you don't know *which* person unless you've heard the previous sentence or two.

Analogously, in Racket and other programming languages, we have *variables* that represent an individual piece of information, but just by looking at the variable in isolation you can't tell *which* piece of information; you need to see the variable's definition. For example, if I said `(beside image1 image2)`, you could tell that I was putting two images side by side, but without seeing the definitions of the variables `image1` and `image2`, you would have no idea *what* images I was putting beside one another. Parameters inside a function definition are just like variables in this sense.

4.8.3 Improper nouns and data types

English also has *improper nouns*: words describing a *category* of things rather than an *individual* thing, like "person", "state", "school", and "dog". The analogous concept in Racket and other programming languages is *data type*: the three data types we've seen so far are "image", "number", and "string".

4.8.4 Verbs and functions

English also has *verbs*: words that represent *actions* that take place, at a particular time, often to one or more particular objects. For example, "Sam kissed Angela last night" includes the verb "kissed", involving the two objects "Sam" and "Angela" (both of which are proper nouns, as discussed above), and the action took place at a particular time (last night).

The closest analogue to verbs in Racket and other programming languages is *functions*: when you call a function on particular arguments, it performs an action on those arguments. (By contrast, literals, variables, and data types aren't so bound to time: they just *are*, rather than doing anything at a particular time.) For example, when you type the expression

`(above` `)`

into the Interactions pane (or type it into Definitions and hit "Run"), the function `above` operates on the two specified pictures at that particular time and creates a new picture including both of them.

4.8.5 Noun phrases and expressions

In English, one can often string several words together into a *noun phrase*, like "my best friend's aunt's house," which represents a specific thing, like a proper noun or a pronoun but which requires a little more work to identify. In this case, it takes three steps: Who is my best friend? Who is that person's aunt? Where is that person's house?

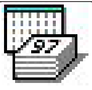
The analogous concept in Racket and other programming languages is the *expression*, which represents a specific piece of information that may be the result of calculation, *e.g.*

```
(beside (flip-vertical bloch) (circle 10 "solid" "green"))
```

whose evaluation requires four steps: What does the variable `bloch` refer to? What do I get when I reflect that picture vertically? What does a solid green circle of radius 10 look like? What do I get when I combine those two images side by side?

We summarize this section in table 4.1.

Table 4.1: Analogy between English parts of speech and Racket

English term	**English examples**	**Racket term**	**Racket examples**	**What it represents**
Proper noun	Joe, Harvard, Rover, Chicago	Literal	, 7, `"hello"`	A single object, which you can tell just by looking at it
Pronoun	Him, her, it	Variable, parameter	`calendar`, `picture`, `image1`	A single object, but needs *context* to tell which one
Improper noun	Person, school, dog, city	Data type	image, number, string	A *category* of objects
Verb	Eat, kiss, study	Function	`flip-vertical`, `mirror-image`, `counterchange`, `define`	An action applied to specific objects at a specific time
Noun phrase	"Jeff's house", "the tallest boy in the class"	Expression	`(flip-vertical book)`, `(above bloch (circle 10 "solid" "red"))`	A single object, maybe the result of computation

4.9 Review of important words and concepts

Instead of typing in a bunch of very similar expressions ourselves, we can use the computer to automate the process by *defining a function.* This requires identifying which parts of the expressions are always the same (these become the function body), and which parts are different (which are referred to inside the function body as *parameters*, and which are replaced with specific *arguments* when the function is called).

A function definition can be divided into a *function header* (specifying the name of the function and of its parameters) and a *function body* (an expression that uses the parameters as though they were defined variables). Ordinarily, the function header is written on one line and the function body on the next one (or more, if it's long and complicated). Racket doesn't actually care whether you define a function on one line or a hundred, nor how you indent the various lines, but human readers (including you!) will find it much easier to read and understand your programs if you follow standard conventions for line breaks and indentation. DrRacket helps with the indentation: every time you hit RETURN/ENTER, it automatically indents the next line as appropriate.

You can see what DrRacket is doing "behind the scenes" by using the *Stepper*: it shows how the arguments in a function call are matched up with the parameters in the function's definition, and how the function call is then replaced by the function body, with parameters replaced by the corresponding arguments.

Before a program can be turned over to a teacher or a customer, it must be carefully tested. If there are errors in a program, it's much better for *us* to discover them than for the teacher or customer to discover them, so try to come up with weird, malicious *test* *cases* that are likely to uncover errors.

One way to write test cases in DrRacket is to put them in the Definitions pane, after the definition itself, each test case followed by a description in quotation marks of what the right answer should be.

Another way is a little more work to write, but easier to use in the long run: the `check-expect` function, which compares the actual answer with what you said it should be, and gives you a report on how many and which of your test cases failed.

The act of defining a function requires a new syntax rule, Rule 5:

```
(define ( new-identifier identifier ...)  expression )
```

The parameter names introduced in the function header can be used inside the function body, but are not visible outside the definition.

Many of the new concepts introduced in the programming-language setting so far correspond to familiar notions from English grammar: literals are proper nouns, variables are pronouns, data types are improper nouns, functions are verbs, and expressions are noun phrases.

4.10 Reference: Built-in functions for defining and testing functions

The only new built-in functions introduced in this chapter are `define` (used to define a *function* rather than a *variable*) and `check-expect`.