

## Chapter 5

# A recipe for defining functions

### 5.1 Step-by-step recipes

I will now give you a simple recipe to accomplish *anything* in the world. Ready?

Design recipe for anything, version 0

1. Decide what you want to do.
2. Do it.
3. Check that you did it right.
4. Keep doing it.

Okay, I admit that's a little vague; you'd need to fill in a lot of details. But it's basically right. In fact, many of the bad inventions, bad laws, bad wars, bad teaching, and bad computer programs in the world can be blamed on somebody skipping either step 1 or step 3 (or both).

Skipping step 3 is understandable, since after you've done your great creative work, *checking* it is boring and unrewarding. But, you may ask, how could anybody skip step 1? In practice, people often charge into "doing it" before they've decided *clearly, unambiguously, and in detail* what they want to do and how they'll know when they've done it. Since they have only a vague mental picture of what they want to accomplish, what actually gets done is a hodgepodge of different goals. (This is especially bad when the task is undertaken by a *group* of people, each of whom has a slightly different idea of what they're trying to do!)

### 5.2 A more detailed recipe

Here's a version that's more useful for actual programming.

## Design recipe for functions, version 1

1. Write a *function contract* (and possibly a *purpose statement*) for the function you want to write.
2. Write several *examples* of how the function will be used, with their correct answers.
3. Write a *skeleton* of the function you want to write.
4. Add to the skeleton an *inventory* of what you have available to work with.
5. Add the *body* of the function definition.
6. *Test* your program on the examples you chose in step 2.
7. *Use* your program to solve other problems.

Steps 1 and 2 correspond to “decide what you want to do”; steps 3–5 to “do it”; and step 6, obviously, to “check that you did it right.” Step 7 corresponds roughly to “keep doing it”: in the real world, programs aren’t a goal in themselves, but are written in order to get answers to questions. We’ll look at all these steps in more detail in a moment.

But first, why do we *need* a recipe like this? For very simple functions, frankly, we don’t. But we won’t always be writing “very simple functions”. I want you to get into the habit of following these steps now, so that by the time you really need them, they’ll be second nature. So how does it help?

- Each step is small and manageable. Even if you have no idea how to solve the whole problem, you can do one step of it and have a small feeling of accomplishment.
- You always know what to work on next, and what questions to ask yourself.
- The contract, purpose, and examples help you understand the question before you start trying to solve it.
- In my class, you get partial credit for each step you solve correctly.
- In my class, the teacher stubbornly refuses to give you any help with a later step until you’ve finished all the previous ones.

Now let’s look at the individual steps and see how to do them.

### 5.3 Function contracts and purpose statements

Before you can solve any programming problem, you have to understand the problem. Sounds obvious, but I’ve had a lot of students over the years charge into writing the program before they had finished reading the assignment. (They ended up writing the wrong program and getting lousy grades.) In particular, you need to be able to write a *function contract* (remember Section 4.2?) for the function you’re about to define.

What’s in a function contract? Three essential pieces of information: the *name* of the function, the *type(s)* of the input(s), and the *type* of the result. (So far, the type of the result has always been “image”, but that will change soon.) Once you’ve written this down (in a comment, in the usual notation from Section 4.2), both you and any other programmer who reads the contract will know how to use your function.

Sometimes, if the program's purpose isn't obvious from its name, it's useful to also write a "purpose statement": a brief sentence or two, also in Racket comments, explaining what the program does. This does *not substitute* for a contract, but *expands* on the contract.

**Worked Exercise 5.3.1** *Write a contract and purpose for the counterchange function of Exercise 4.2.3.*

To remind you what Exercise 4.2.3 was, I'll repeat it here:

**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. The result should look like

**Solution:** The function name is obviously `counterchange`. The assignment says it is to be "given two images", which tells us that it takes two parameters of type "image". It "produces a two-by-two square", which must be an image. So we could write (in the Definitions pane) something like

```
; counterchange : image image -> image
```

This technically answers all the questions, but it doesn't really give a user all the necessary information to use the function. Which of the two images is which? So a better contract would be

```
; counterchange : image (top-left) image (top-right) -> image
```

If we think it's necessary to add a purpose statement, we might write

```
; counterchange : image (top-left) image (top-right) -> image
; Produces a square arrangement with the top-left image also
; in the bottom right, and the top-right image also in the
; bottom left.
```

**Exercise 5.3.2** *Write a contract and purpose statement for a function named `copies-beside` that takes in a number and an image, and produces that many copies of the image side by side.*

**Exercise 5.3.3** *Write a contract and purpose statement for a function named `pinwheel` that takes in a picture and produces four copies of it in a square, differently rotated: the original picture in the top left, rotated 90° clockwise in the top right, rotated 180° in the bottom right, and rotated 90° counterclockwise in the bottom left. The result*



should look like

**Worked Exercise 5.3.4** *Write a contract and purpose statement for a function named `checkerboard2` that produces a 2x2 square checkerboard in specified colors. Each square should be 20 pixels on a side.*

**Solution:** The function name is obviously `checkerboard2`. The result is an image. As for the input, the assignment refers to “specified colors”, but doesn’t say what they are, which implies that they can vary from one function call to the next — in other words, they are inputs to the function. The only way we know to specify colors so far is by their names, which are strings in Racket. There are two colors, and the assignment doesn’t say which is which, so let’s decide arbitrarily that the first one will be in the top-left corner of the checkerboard, and the second in the top-right corner. The “2x2” and “20 pixels on a side” don’t concern us yet. Furthermore, it doesn’t make sense to call the function with strings that aren’t color-names, *e.g.* “screwdriver”. So we might write (in the Definitions pane)

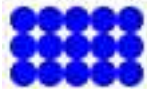
```
; checkerboard2 : string (top-left-color)
;                 string (top-right-color) -> image
; Assumes that both strings are color names.
; Produces a 2x2 checkerboard, with each small square 20
; pixels on a side, with the top-left color in the top-left
; and bottom-right positions, and the top-right color in
; the other two positions.
```

■

**Exercise 5.3.5** *Write a contract and purpose statement for the following problem: Design a program named `bullseye` which produces a “bull’s eye” style target with two rings. It takes in two numbers indicating the radii of the outer ring and the inner disk, and two strings representing the colors of the outer ring and the color of the inner disk.*

**Exercise 5.3.6** *Write a contract and purpose statement for the following problem: Design a program named `dot-grid` which expects two numbers (the width and height of the grid, respectively) and produces a rectangular grid of radius-5 circular blue dots .*

```
> (dot-grid 5 3)
```



**Exercise 5.3.7** *Write a contract and purpose statement for the following problem: Design a program named `lollipop` which produces a picture of a lollipop. It takes in two numbers — the radius of the lollipop “head” and the length of the “stick” — and a*

*string, indicating the color of the lollipop. For the stick, use a rectangle of width 1.*



## 5.4 Examples (also known as Test Cases)

A function contract and purpose statement are big steps towards understanding what the program needs to do, but they're not specific enough. So for the next step, we'll write down several *examples* of how we would use the program if it were already written. Next to each example, we'll write down, as precisely as we can, the *right answer* to that example (as in Chapter 4).

Why? Several reasons. Most obviously, it provides you with test cases that you can use later in testing. Since testing can be the most frustrating part of programming, your mind will take any excuse it can find to avoid testing. (After all, how often do you do something for which the whole point is to get bad news? Do you look forward to doing such things?) And since testing is also the *last* thing you do before turning in the program, it tends to get skipped when you're running behind schedule. Writing the test cases in advance — before you write the program itself — means you have one less excuse for not testing.

Your mind will try to trick you in another way, too: you'll look at the results of a test run and say to yourself “yes, that's right; that's what I expected,” when you really had only a vague idea what to expect. This is why you *must write down right answers* next to each test case: it's much harder to fool yourself into thinking everything is fine when a picture of a blue box is next to the words “should be a green circle”. If you use `check-expect`, it becomes even harder to fool yourself.

Another benefit: it makes you specify *very precisely* what the program is supposed to produce. No room for vagueness here. Of course, we'll have to be precise when we write the program; this gives us a chance to “warm up” our precision muscles, and come up with suitably nasty “special cases” that might possibly throw the program off, without having to think about Racket syntax at the same time.

By the way, if you have friends who are professional programmers, you can tell them you're learning “test-driven development”. (There's more to test-driven development than this, but its most important feature is writing test cases before you write the program.)

**Worked Exercise 5.4.1** *Write several test cases for the counterchange function of Exercise 5.3.1.*

**Solution:** We've already identified the contract for this function, so we should be able to call it with any two images. By Syntax Rules 1 and 2, an example would be







(counterchange  )

What should this produce? Since the contract tells us the first parameter will go into the top-left position and the second into the top-right, we could write

```
"should be a picture with a calendar in the top-left
and bottom-right corners, and a hacker in the top-right
and bottom-left"
```

If we wanted to give a more precise “right answer”, we could come up with an expression that actually builds the right picture, *e.g.*

```







(check-expect (counterchange   )
              (above (beside   )
                    (beside   )))

```

This is, of course, more work, but it also brings two extra benefits: when we get to writing the body, this example will help us do it, and “check-expect” style test cases are much easier to check.

So far we’ve given only one test case, and it’s possible that the function might get that case right even though it’s wrong in general. So let’s try another:

```

(check-expect (counterchange   )
              (above (beside   )
                    (beside   )))

```

**Exercise 5.4.2** Write several test cases for the *copies-beside* function described in Exercise 5.3.2.

**Hint:** the function takes in a “number”, but in fact it wouldn’t make sense to call this function on a fraction, a negative number, or an irrational number. The only numbers that make sense are *whole numbers*, also known as *counting numbers*: 0, 1, 2, 3, .... In trying to break the program, think about whole numbers that might be considered “special cases”. We’ll revisit whole numbers in Chapter 24.

**Exercise 5.4.3** Write several test cases for the *pinwheel* function of Exercise 5.3.3.

**Exercise 5.4.4** Write several test cases for the *checkerboard2* function of Exercise 5.3.4.

**Hint:** Technically, any two strings would satisfy the contract, but since the purpose statement adds “Assumes that both strings are color names”, you don’t need to test it on strings that aren’t color names. Ideally, a program to be used by human beings should be able to handle a wide variety of incorrect inputs, and we’ll learn techniques for handling such situations in Chapter 19.

**Exercise 5.4.5** Write several test cases for the *bullseye* function of Exercise 5.3.5.

**Exercise 5.4.6** Write several test cases for the *dot-grid* function of Exercise 5.3.6.

**Exercise 5.4.7** Write several test cases for the *lollipop* function of Exercise 5.3.7.

## 5.5 The function skeleton

A *function skeleton* is a “first draft” of the function definition, based only on Syntax Rule 5 and the information in the contract (not what specific problem the function is supposed to solve). As you recall from Chapter 4, every function definition follows a simple pattern:

```
(define (function-name param-name param-name ...)
  ...)
```

Once you’ve written the contract, you already know the function name as well as the number, types, order, and meanings of all the parameters. So it’s easy to write the header; the only faintly creative part is choosing good, meaningful names for the parameters. We’ll come back to this in a moment.

**Worked Exercise 5.5.1** Write a *function skeleton* for the *counterchange* function of Exercise 5.3.1.

**Solution:** Recall that the function takes in two images, which we referred to in the contract as *top-left* and *top-right*. These are probably good choices for parameter names: they make clear what each parameter is supposed to represent. Following Syntax Rule 5, the function definition must look like

```
(define (counterchange top-left top-right)
  ...)
```



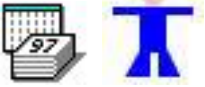


## 5.6 Common beginner mistakes

Here are some of the things I’ve seen a lot of students do wrong in writing examples and skeletons.

### Not calling the function by name

A student working on *counterchange* writes the example

```
(check-expect
  
  (above (beside
    
  )
  (beside
    
  )))
```

This “example” never actually mentions the `counterchange` function, so DrRacket doesn’t know what to do with the two pictures. See Section 4.5.2.

**The contract, examples, and skeleton must agree on the number, type, and order of parameters.**

For example, suppose I were writing a function that took in a string and two numbers, and returned an image:

```
; do-it : string number number -> image
```

A student writes the example `(check-expect (do-it "boojum") "fnord")`. This example violates the contract in at least two ways: it gives the function only a string rather than a string and two numbers, and it expects the answer to be a string rather than an image. A *correct* test case would have to call `do-it` on a string and two numbers, and have an image as the “right answer”:

```
(check-expect (do-it "boojum" 0 3.14)
              (circle 10 "outline" "green"))
```



```
(check-expect (do-it "blah" 7 32)
              (triangle 10 "solid" "blue"))
(check-expect (do-it "fnord" 5/3 -6)
              (triangle 10 "solid" "blue"))
```

Another student writes the example

`(check-expect (do-it 3 4 "boojum") ...)`. This has the right number of parameters, and the right return type, but the parameters are in a different order from what the contract said: the string is third rather than first.

Next, a student writes the skeleton `(define (do-it word) ...)`. Again, this violates the contract because `do-it` takes in *three* parameters, and this skeleton has only one. A *correct* skeleton for this function would have to have three parameter names, the first standing for a string and the second and third standing for numbers, for example

```
(define (do-it word num1 num2)
  ...)
```

Notice how the contract, examples, and skeleton *must* “match up”:

<pre>; do-it: string number number -&gt; image</pre>	
<pre>(check-expect ( do-it "blah" 7 32 )</pre>	
<pre>(define ( do-it word num1 num2 ) ...)</pre>	

The function name is the same in all three places. The number and order of parameters in the contract are the same as the number and order of arguments in each test case, which are the same as the number and order of parameters in the skeleton. The return type in the contract is the same as the type of the “right answer” in each test case, which is the same type as the “body” expression which will eventually replace the “...” in the skeleton.



### Misleading parameter names

In the above example skeleton, the parameter names `word`, `num1`, and `num2` were chosen to suggest to the reader that the first is a string and the other two are numbers. There's actually quite a bit of art to choosing good parameter names.

Remember that a parameter name is a *place-holder*; you can't assume that a particular parameter will always be a picture of a book, or will always be the string "yellow", or will always be the number 7. If the contract says that the first parameter is a picture, you can assume that it's a picture, but not *what* picture. I've seen students write a perfectly-good example like

(counterchange  )

but then write a function skeleton like

```
(define (counterchange calendar hacker) ...)
```

What's wrong with this? It's technically legal, but misleading: the parameter-name `calendar` will stand for whatever first argument is provided when the function is called (which may or may not be a calendar), and the parameter-name `hacker` will likewise stand for the second argument (which may not be a hacker). For example, when you try the second test case

(counterchange  )

the parameter name `calendar` will stand for a stick figure, and the name `hacker` will stand for a picture of a calendar! Remember that a function has to work on *all possible inputs*, not only the ones you had in mind.

### Duplicate parameter names



Another bad attempt at a skeleton for this function is

```
(define (counterchange picture picture) ...)
```

"picture" is a perfectly reasonable parameter name, but this student has used it *twice in the same function header*. The purpose of parameter names is to allow you to refer to them in the body of the function; if they both have the same name, how will Racket know which one you mean? This is illegal in Racket, and it'll produce an error message. (Try it.)

### Literals as parameter names

Yet another incorrect skeleton is 

```
(define (counterchange  ) ...)
```

This one is also illegal: the parameters in a function header must be variable names, not literals. DrRacket will give you an error message. (Try it.)







More generally, this kind of mistake points to a common beginner confusion between *calling* a function (on specific arguments, *e.g.* pictures or variables already defined to stand for pictures) and *defining* a function (in which you specify place-holder parameter names).

Remember, the parameter names in a function skeleton should indicate what the parameter “means” in general (*e.g.* `top-left` and `top-right`), and/or what *type* it is (*e.g.* `picture`), but should *not* assume anything about what *specific value* it is.






## 5.7 Checking syntax

Once you’ve written a contract, examples, and a function skeleton (with “...” where the function body will eventually be), DrRacket will help you check whether you’re on the right track. Obviously, since you haven’t written the function body yet, it won’t produce correct answers, but at least you can check whether you’re following all the syntax rules and calling the function with the right number of arguments, thus catching many common beginner mistakes. Click the “Check Syntax” button near the top of the screen. If your examples use the function with the wrong number of arguments, DrRacket will tell you so.

For example, if you made the first of the “common beginner mistakes” above,

```
(check-expect  
              (above (beside   )
              (beside   )))
```

and hit “Check Syntax”, DrRacket would notice that you were calling `check-expect` with three arguments rather than two, and tell you so. On the other hand, if you provided just a picture and an answer:

```
(check-expect 
              (above (beside   )
              (beside   )))
```

“Check Syntax” wouldn’t be able to tell that there was anything wrong, because all your function calls are with the right number of arguments.

The second of the “common beginner mistakes” above is calling a function with the wrong number of arguments, or arguments in the wrong order. “Check Syntax” will

complain if you define a function with two parameters but call it with one or three; it can't tell if you call it with two parameters in the wrong order.

The third of the “common beginner mistakes” above, misleading parameter names, is beyond “Check Syntax”'s ability to catch: all your function calls are legal.

However, it can easily catch the fourth and fifth:

```
(define (counterchange picture picture) ...)
```



```
(define (counterchange ) ...)
```

In both of these cases, the function definition doesn't fit Syntax Rule 5, and “Check Syntax” will tell you so.

## 5.8 Exercises on writing skeletons

In doing the following exercises, use “Check Syntax to see whether your examples and skeletons match up properly.

**Exercise 5.8.1** *Write a function skeleton for the `copies-beside` function of Exercise 5.3.2.*

**Exercise 5.8.2** *Write a function skeleton for the `pinwheel` function of Exercise 5.3.3.*

**Exercise 5.8.3** *Write a function skeleton for the `checkerboard2` function of Exercise 5.3.4.*

**Exercise 5.8.4** *Write a function skeleton for the `bullseye` function of Exercise 5.3.5.*

**Exercise 5.8.5** *Write a function skeleton for the `dot-grid` function of Exercise 5.3.6.*

**Exercise 5.8.6** *Write a function skeleton for the `lollipop` function of Exercise 5.3.7.*

## 5.9 The inventory

Imagine that you're trying to bake cookies. A smart cook will get all the ingredients (eggs, milk, butter, sugar, chocolate chips, etc.) out and put them on the counter before mixing anything together: that way you can see whether you have enough of everything. We'll do something similar: list everything that's available for you to use in defining the function, before starting to put things together. At this stage, that basically means the parameters (it will get more interesting later).

You should also recall from Chapter 4 that the parameter names that appear in the function header must exactly match those that appear in the function body — same spelling, same capitalization, etc. You may have no idea how the function body is going

to work, but you can be pretty sure that the parameter names you put in the header will be used in it. At this stage, I recommend writing the names of all the parameters, one on each line, commented out, in between the function header and the “...” where its body will eventually be. It’s often helpful to also write down, next to each parameter, what *data type* it is; this determines what you can reasonably do with it.

There may also be particular pieces of information that are always the same, regardless of the arguments passed into the function. For example, a function that is supposed to always draw in blue will presumably use the word “blue” at least once in its body.

I generally ask students to write a *skeleton*, as above, and then *insert* the inventory information before the “...”.

**Worked Exercise 5.9.1** *Add an inventory to the skeleton for the counterchange function of Exercise 5.5.1.*

**Solution:** We’ve already written the skeleton:

```
(define (counterchange top-left top-right)
  ...)
```

We don’t know yet how the function body will work, but we’re pretty sure it will involve the variable names `top-left` and `top-right`. So we list these, one on each line, commented out, along with their types. The complete function skeleton, with inventory then reads

```
(define (counterchange top-left top-right)
  ; top-left image
  ; top-right image
  ...)
```

Together with the contract and examples we wrote before, the Definitions pane should now look like Figure 5.1.



#### SIDEBAR:

Later in the book, we’ll talk about something analogous called an “outventory”. Where an inventory answers the question “what am I given, and what can I do with it?”, an outventory answers the question “what do I need to produce, and how can I produce it?”. If the inventory is like collecting the raw ingredients for cookies, the outventory is like observing that the *last* step in the recipe is baking, and concluding that you’d better preheat the oven and make sure you have a cookie sheet.

We’ll come back to this concept when we have problems to solve that need it. For now, inventories will do just fine.

**Exercise 5.9.2** *Add an inventory to the skeleton for the copies-beside function of Exercise 5.8.1.*

**Exercise 5.9.3** *Add an inventory to the pinwheel function of Exercise 5.8.2.*

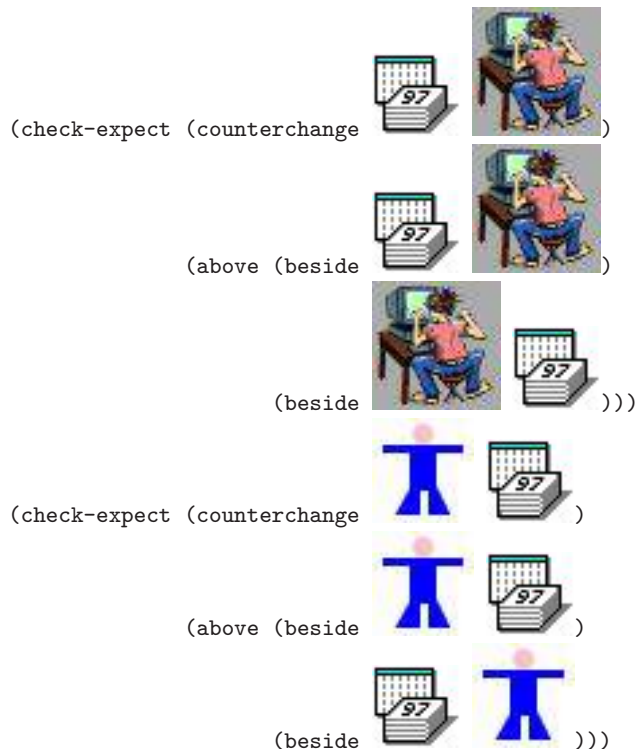
**Exercise 5.9.4** *Add an inventory to the checkerboard2 function of Exercise 5.8.3.*

Figure 5.1: Skeleton and inventory of counterchange

```

; counterchange : image (top-left)
;                 image (top-right) -> image
; Produces a square arrangement with the top-left image also
; in the bottom right, and the top-right image also in the
; bottom left.

```



```

(define (counterchange top-left top-right)
  ; top-left image
  ; top-right image
  ...)

```

**Hint:** In addition to the parameters, this function will almost certainly need to use the number 20 (the size of each small square), so you can include another line with 20 on it. Its type, obviously, is *number*.

**Exercise 5.9.5** *Add an inventory to the bullseye function of Exercise 5.8.4.*

**Hint:** This function will need to make some solid circles, so it'll need the string "solid". Include this fixed value, on a line by itself, along with the parameters.

**Worked Exercise 5.9.6** *Add an inventory to the dot-grid function of Exercise 5.8.5.*

**Solution:** You should already have a skeleton, so we'll discuss only what to add to it. Suppose your parameter names are `width` and `height`. Obviously, you'll need them inside the body:

```
; width      a number
; height     a number
```

In addition, you know that the function will need radius-5 circular blue dots . To produce these, we can be fairly certain that we'll need the expression `(circle 5 "solid" "blue")`. This too can be added to the inventory. The skeleton with inventory now looks like

```
(define (dot-grid width height)
  ; width      a number
  ; height     a number
  ; (circle 5 "solid" "blue") an image
  ...)
```

**Exercise 5.9.7** *Add an inventory to the lollipop function of Exercise 5.8.6.*

## 5.10 Inventories with values

Sometimes listing the available expressions and their types is enough for you to figure out the body expression. But what if you do all that, and don't have any flashes of inspiration? A technique that has helped many of my students is writing an *inventory with values*.

Here's how it works. After you've written down all the "available expressions" and their types, *choose one of your test cases* (preferably one that's not too simple), and for each of the expressions in the inventory, *write down its value for that test case*. Then add another line to the inventory, labelled

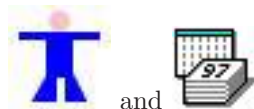
```
; right answer
```

and write down the right answer for that test case. Finally, look at the value of the right answer and the values of the available expressions, trying to find a way to get the right answer from the available expressions.

**Worked Exercise 5.10.1** *Add values to the inventory of the counterchange function of Exercise 5.5.1.*

**Solution:** We already have a skeleton and test cases in Figure 5.1.

We'll pick the second of our test cases, the one involving



and , and

```
(define (counterchange top-left top-right)
```

```
  ; top-left      image
```



```
  ; top-right     image
```



```
  ; right answer image
```

```
    (above (beside
```



```
  ;
  ...)
```

```
    (beside
```



```
    ))
```

■

Note that we're *not* saying that `top-left` will *always* be a stick-figure, or `top-right` will *always* be a calendar, only that we're using those values as an example to see more concretely what the inventory means. We still have to write the function in such a way that it works on *any* values we give it.

## 5.11 The function body

Now it's time to put some meat on the bones. This, frankly, is the hardest part of programming: coming up with an expression which, no matter what arguments the function is applied to, will *always* produce the right answer. We've done as much of the work ahead of time as possible: we know what types the function is taking in and returning, we've written a couple of specific examples, we have the basic syntax of a function definition, we know what parameters and values are available for us to use, and we have a specific example of them to compare with a specific right answer. Now we have to think about *how we would solve the problem if we were the computer*. In doing this, you will find function contracts (both for predefined functions and for your own functions) extremely useful: for example, if you have a string that you know is supposed to be the name of a color, you can be pretty sure it'll appear as the last parameter to the `circle`, `rectangle`, `ellipse`, or `triangle` function; which one depends on what the function is supposed to draw.

Just as we added the inventory into the skeleton, we'll add the function body just after the commented-out "ingredients" from the inventory stage, still inside the parentheses, in place of the "...", so the whole thing becomes a legal function definition that happens to have a commented inventory in the middle.

**Worked Exercise 5.11.1** *Add a body to the skeleton and inventory for the counterchange function of Exercise 5.9.1.*

**Solution:** Of course, we’ve already done this in Exercise 4.2.3. But if we hadn’t, here is how we would figure it out. We already have the skeleton and inventory:

```
(define (counterchange top-left top-right)
```

```
  ; top-left      image
```



```
  ; top-right     image
```



```
  ; right answer image
```

```
    (above (beside
```



```
    )
```

```
  ;
  ...)
```

```
    (beside (above (beside (above (beside
```



```
    ) ) ) )
```

We know that the body will (almost certainly) use the names `top-left` and `top-right` (at least once each). If you immediately see how to put them together to get the right results, great. If not, look at the examples, both of which fit the pattern

```
(above (beside something something-else)
 (beside something-else something))
```

They differ only in what pictures are used as *something* and *something-else*, both of which must be images. Do we have any images available to work with? Yes: according to the inventory, `top-left` and `top-right` are images, so the obvious thing to try is to use those as *something* and *something-else*. But which one is which? In the “inventory with values”, the image that was the value of `top-left` was the first argument to the first `beside` and the second argument to the second, while the image that was the value of `top-right` was the second argument to the first `beside` and the first argument to the second. This suggests

```
(above (beside top-left top-right)
 (beside top-right top-left))
```

as the body, and we’ll type this in between the inventory and the closing parenthesis of the skeleton. We should now have a Definitions pane that looks like Figure 5.2. ■

Together with the contract and examples we wrote before, the Definitions pane should now look like Figure 5.2.

Unfortunately, I haven’t yet told you everything you need to write the bodies of `copies-beside` or `dot-grid`. (We’ll get to these in Chapter 24.) However, you should be able to do the following four, especially if you’ve done all the steps up until now for them.

**Exercise 5.11.2** *Add a body to the pinwheel function of Exercise 5.9.3.*



**Exercise 5.11.3** *Add a body to the checkerboard2 function of Exercise 5.9.4.*







Figure 5.2: Complete definition of counterchange



```



; counterchange : image (top-left)
;                image (top-right) -> image



(check-expect (counterchange   )

(above (beside   )


(beside   )))

(check-expect (counterchange   )


(above (beside   )

(beside   )))



(define (counterchange top-left top-right)





; top-left image



; top-right image

; right answer image (above (beside   )

; (beside   )))

(above (beside top-left top-right)
(beside top-right top-left))
)

```

**Hint:** You can write this function directly, using `beside` and `above`, or you can write it shorter and simpler by re-using another function we’ve already written. Shorter and simpler is good!

**Exercise 5.11.4** *Add a body to the `bullseye` function of Exercise 5.9.5.*

**Exercise 5.11.5** *Add a body to the `lollipop` function of Exercise 5.9.7.*

## 5.12 Testing

Now it’s the moment of truth: the time to find out whether your function definition *works*. If you’ve typed everything into the Definitions pane correctly, just click the “Run” button. If you’re using “should be” for test cases, you’ll see the results in the Interactions pane, each followed by the words “should be” and the right answer. Check that each of the actual answers matches what you said it “should be”: if any of them don’t match, figure out what’s wrong and fix it. If, on the other hand, you’re using `check-expect`, you should see a report telling you exactly how many and which of your test cases failed.

If you get an error message like *reference to an identifier before its definition*, it means that you tried to *use* the new function before *defining* it — for example, if you have “should be”-style examples appearing ahead of the definition. (Examples using `check-expect` are allowed to appear ahead of the definition.) A slightly different error message, *name is not defined, not an argument, and not a primitive name*, means that the name of the function in the examples doesn’t exactly match the name of the function in the definition (in spelling, or capitalization, or something like that).

If you get an error message like *this procedure expects 2 arguments, here it is provided 1 argument*, it means your contract, examples, and skeleton disagree on how many arguments the function is supposed to take in. An error message like *expects type <number> as 1st argument, given “hello”* likewise indicates a disagreement among contract, examples, and skeleton on either the types or the order of the arguments.

**Exercise 5.12.1** *Test each of the functions whose definitions you wrote in Section 5.11. If any of them produce wrong answers, fix them and test again until they produce correct answers for every test case.*

## 5.13 Using the function

Now that you’ve tested the function on problems for which you know the right answer, you have confidence that it works correctly. So you can use it on problems for which you *don’t* know the right answer, and be reasonably confident that the answers it produces are right.

Just as important, you can now use your new function in writing other functions, knowing that if something is wrong with the new function, it’s not the old function’s fault. When a new piece of commercial software is written, it may rely on hundreds or thousands of previously-written functions; if a programmer had to re-examine every one of them to fix a bug in the new program, nothing would ever get done.

## 5.14 Putting it all together

In reality, you would seldom want to write just the contract, or just the examples, of a function. Much more common is to go through all the steps for a single function, then go through all the steps for another function. In this section are several more function-definition exercises: for each one, go through all the steps of the design recipe.

**Exercise 5.14.1** *Develop a function named `diamond` that takes in a string (the color of the diamond) and a number (the length of one of its sides) and produces a diamond shape, i.e. two triangles connected along one side.*

```
> (diamond "blue" 20)
```



```
> (diamond "green" 30)
```



**Exercise 5.14.2** *Develop a function named `text-box` that takes in two strings, of which the second should be a color-name, and two numbers (width and height), and produces a picture of the first string, in 18-point black type, on a background rectangle of the specified color, width, and height.*

**Exercise 5.14.3** *(Thanks to Leon LaSpina for this problem)*

*Develop a function named `two-eyes` that, given a number and a color name, produces a picture of two circular “eyes”, 100 pixels apart horizontally. Each one should have a black “pupil” of radius 10, surrounded by an “iris” of the specified color and radius (which you may assume is more than 10). The 100-pixel separation is measured from edge to edge, not center to center.*

**Exercise 5.14.4** *(Thanks to Leon LaSpina for this problem)*

*Develop a function named `circle-in-square` that takes in a number (the length of a side of a square) and two strings (the colors of a square and a circle), and produces a picture of a square of one color, with a circle of the other color inscribed inside it. The diameter of the circle should be the same as the side of the square, so the circle just barely touches the edge of the square at the middle of each side.*

**Hint:** If you’ve already read Chapter 7, you’ll be tempted to do this using arithmetic. But it can be done without arithmetic, using only what you’ve seen so far.

**Exercise 5.14.5** *Develop a function named `caption-below` that takes in an image and a string, and produces a copy of the same image with a caption underneath it:*

```
> (caption-below pic:bloch "Dr. Bloch")
```



Dr. Bloch

## 5.15 Review of important words and concepts

Projects (such as writing a computer program) are more likely to be finished successfully if you follow a **recipe**. In particular, you need to have a very clear, precise idea of what a program is supposed to do before you start writing it. To design a Racket function, follow the following seven steps:

1. Write a *function contract* (and possibly a *purpose statement*) for the function you want to write. Include the *function name*, the *numbers*, *types*, and *order of inputs*, and the *type of the result* (which so far is always “image”).
2. Write several *examples* of how the function will be used, with their correct answers. Include special cases and anything else “weird” (but legal, within the contract) that might help you uncover errors in your program.
3. Write a *skeleton* of the function you want to write, by combining Syntax Rule 5 with the decisions you’ve already made about number, type, and order of parameters in step 1. At this point you can use “Check Syntax” to confirm that your examples and skeleton match one another.
4. Add an *inventory* to the skeleton, showing the names and types of all the parameters, and any “fixed data” that you know will be needed no matter what arguments are plugged in. If there are more complex expressions that you’re confident you’ll need, you can write those at this point too. It’s often a good idea to *choose a specific test case* and *write down the values* of all the inventory items for this test case, as well as what the right answer should be for this test case.
5. Fill in the *body* of the function definition. This will be based on the skeleton, with the help of any patterns you notice from the “right answers” of your examples. At this point you can use “Check Syntax” again to confirm that all the parentheses are matched correctly, you’re calling functions with the right number of arguments, and so on.
6. *Test* your program on the examples you chose in step 2. If the actual answers don’t match what you said they “should be”, figure out what’s wrong, fix it, and test again.

7. *Use* your program to answer questions for which you don't already know the right answer, or to build other, more complex functions, secure in the knowledge that *this* one works.

Although this chapter didn't introduce a lot of new Racket constructs, it is probably the most important chapter in the book. If, a year after completing this course, you've forgotten all your Racket but you remember the design recipe, I'll be happy, and you'll be well on your way to being a good programmer.

## 5.16 Reference

No new functions or syntax rules were introduced in this chapter.