RACKET WEEKEND

© 2018, Jesse Alama

Contents

Foreword

Welcome to *Racket Weekend*!

My hope is that, thanks to this brief course, you'll whet your appetite for some of the basics of Racket and will be in a position to dive in and make awesome software with it, or maybe even your own language (that's Racket's specialty).

We're ignoring all sorts of cool features of Racket to concentrate on the basics, on topics that are likely to be part of your everyday Racket workflow as you get into the language. If you want to get into the really exciting, advanced aspects of Racket, you'll still need to know about these bits. (At the end of the coure, I'll give you a few pointers for where you can go next.)

The course is not inteded to be terribly long. I've deliberately tried to curtail certain discussions from going off the deep end with the aim of making the course compact. I've tried not to make 'a Racket course', which could be a very large undertaking indeed, since Racket is overflowing with features. Instead, I've tried to make sure that we stay on a more or less straight path that you can actually traverse from beginning to end. Of course, I'm not watching you; take as much time as you want, and meander or dive in when you find things that you want to know more about.

To prepare yourself for the course, make sure you have installed Racket. To get Racket, go to **its homepage** and click the Download button.

I assume you're somewhat comfortable at the command line. Racket comes with a graphical editor and runtime called DrRacket, which is, despite initial appearances, a serious power tool. Virtually everything we do in the course could be done with DrRacket. But we won't use DrRacket here. It's one of the (many!) things that got cut in favor of keeping the course more focused.

If you have any questions, I'm happy to answer them. I'm `alamajesse` on Twitter, and you can reach me at `jesse@lisp.sh`.

Have a nice weekend!

(lesson 1) Program structure. Definitions & basic data

In which we begin by encountering some basic data, defining functions, and start running programs. Those famous parentheses rear their head.

It all begins with a #lang

The first line of many Racket programs begins with `#lang`. Here's a well-formed Racket program:

```
#lang racket
```

As you might imagine, this program doesn't do anything. To add some meat to it, let's define something. Definitions assign values to variables (AKA identifiers).

Let's define a greeting:

```
(define welcome-msg "Welcome to Racket Weekend!")
```

There's nothing tricky going on here: we're introducing a new variable, `welcome-msg`, and assigning it a literal string value. Racket literal strings are double-quoted. Strings are UTF-8.

The program still doesn't do anything, but at least now it's no longer empty.

Let's define another variable:

```
(define cheers "Cheers!")
```

This definition is just like the previous one. Nothing fancy. Let's add a number:

```
(define course-num-days 2)
```

We're defining `course-num-days` to be the number 2. Racket's support for arithmetic is great. Out of the box, you've got integers, rationals (ratios of integers), floats, even complex numbers (which are handy when doing graphics, because a single complex number can be understood as a point on a 2-dimensional plane).

Our program *still* isn't doing anything yet; we're just defining things but never using them. And we're still using just 'simple' data (strings, integers).

Let's get the ball rolling by defining a function. Functions in Racket are like functions in other programming languages: they take arguments and have a body, which is where you write code that does something.

Here's a function that starts to combine data to produce a result:

```
(define (explanation num-days)
  (format "Racket Weekend is a ~a-day course on Racket."
          num-days))
```

Our intention here is to define a function that takes in an integer and produces a string. We use `format`, which is a string-building function analogous to `sprintf` in other languages. The first argument is a so-called format string, which is a template specifying how the resulting string should look, given certain arguments. The little `~a` is where we plug in a value.

This `define` looks a bit different from the other defines. There, we used the variable name directly. It's clear that when we write

```
(define cheers "Cheers!")
```

that `cheers` is to have the value `"Cheers!"`. But what are we defining when we write

```
(define (explanation num-days)
  (format "Racket Weekend is a ~a-day course on Racket."
          num-days))
```

?

The answer is that we're defining `explanation`. Here, though, `explanation` isn't a piece of "simple" data. It's a function, taking one argument.

Our program still does not really *do* anything, apart from defining some variables, one of which (`explanation`) is a function. Let's get the ball rolling by producing some output:

```
(define (welcome)
  (displayln welcome-msg))
```

We're defining a new function here, `welcome`. (The giveaway that this is a function definition is that the name, after `define`, is wrapped in parentheses.) It takes no arguments. `displayln` is a built-in function that prints its argument to standard output (and ensures that a newline is added after outputting the argument—the `ln` stands for 'line'). It uses `welcome-msg`, which we defined earlier. As you might imagine, calling this function causes our welcome message to be printed.

Here's another function:

```
(define (cheer)
  (displayln cheers))
```

This function does essentially the same thing as the function `welcome`, just with a different message. Finally, consider:

```
(define (explain num-days)
  (displayln (explanation num-days)))
```

Here we're defining a function that takes one argument. The argument, `num-days`, gets passed along to the `explanation` function that we defined earlier. That function gives us back a string, and `displayln` prints it.

Running the program

Let's save all this work in a file, say, `structure.rkt`. The program should look like this:

```
#lang racket

(define welcome-msg "Welcome to Racket Weekend!")

(define cheers "Cheers!")

(define course-num-days 2)

(define (explanation num-days)
  (format "Racket Weekend is a ~a-day course on Racket."
          num-days))

(define (welcome)
  (displayln welcome-msg))

(define (cheer)
  (displayln cheers))

(define (explain num-days)
  (displayln (explanation num-days)))
```

What happens if you go to the command line and do

```
racket structure.rkt
```

?

The answer, perhaps surprisingly, is: nothing.

That's not, strictly speaking, true. Something *is* happening when you execute the program: the definitions are executed. But we're still not *doing* anything.

The way to finally get the program to do something is to put what you want into the `main` submodule. Here's how that's done. After all those definitions, we wrap calls to `welcome`, `explain`, and `cheer` as follows:

```
(module+ main
  (welcome)
  (explain course-num-days)
  (cheer))
```

Save and run the program again at the command line. You should now see:

```
Welcome to Racket Weekend!
Racket Weekend is a 2-day course on Racket.
Cheers!
```

The `module+` bit allows to embed a so-called submodule into our program. `main` is the name of the module. Using `main` is a signal to the Racket runtime that we intend that the code that follows is to be executed. (It is not unlike the `main` function in C programs, which serves a similar purpose.) The whole submodule is—as with many things Lisp—wrapped in parentheses.

Notice that we're able to use the definitions we provided in the enclosing module (which, unlike `main`, is anonymous). The submodule can 'look outside' and see what `cheer`, `welcome`, `explain`, and `course-num-days` mean.

require & provide

A module might live in isolation, taking nothing from anyone and giving nothing to no one.

But most programs are divided into pieces, with each piece (module) requiring something to do its work, having some private bits, and sharing some with others. Racket is no exception to this.

It's not really fair to say that a module starts from nothing at all. We started our little program here with `#lang racket`, which in fact 'pulls in' all of Racket, or, more precisely, makes all of it available to us. Saying `#lang racket` is even deeper than that: using that line specifies the way our expressions will be read (or parsed).

With Racket, one can specify new languages. You don't always have to hack in `racket`. This course won't show how to do that (though we'll see some hints of that in the lessons on macros and on documentation, where a new language, Scribble, will be used).

Modules exchange information with one another using `require` and `provide`. The first is used to indicate what the module needs (or imports) from other modules; the second indicates what it makes public (or exports). Traditionally, to assist readability, one writes (after the `#lang` line) what the current module provides; the next line(s) indicate what one requires. (But that's just convention. You can write the `require` and `provide` anywhere in the toplevel of a module. You can even use more than one `require` and more than one `provide`.

Here's an example:

```
#lang racket
(provide cool-func
         super-awesome-thing)
(require sweet-package
         radical-library
         helpful-thingamajig)

(define (cool-func x)
  ; implementation here; you can use
  ; whatever sweet-package, radical-library, or
  ; helpful-thingamajig provide to
  ; help you
)
```

We will see several examples of this as we proceed in the course. There will be a lesson coming up where we talk about how to get packages from other Racket developers, so you have something to `require`. And, later, we'll briefly see a 'sublanguage' of `racket` called `racket/base` where one might have to reach up and grab some pieces from the 'superlanguage'.

Takeaways

The program we've written here is, clearly, very simple. But it illustrates a few ideas used throughout Racket and in everyday Racket programming. We've seen here:

- The `#lang` line. 'lang' is for language; we used `racket`. As you might imagine, there are other languages. (And you can define your own! That is one of the killer features of Racket.)
- We've seen a few different kinds of basic data: strings, numbers, and functions.

- Definitions take a couple of forms: one can simply define a new identifier using either

  ```
  (define identifier value)
  ```

  or

  ```
  (define (func arg1 arg2)
    ...)
  ```

  to define a new function.
- A hint of string-building in Racket using `format`.
- How to output a string with `displayln`.
- How to make your programs do something: use the `main` submodule. In this course we'll see two cases of submodules: `main` (for building up the 'executable part' of your program) and `test` (for building up your tests).

`(lesson 2)` The Read-Evaluate-Print Loop (REPL)

The read-evaluate-print loop provides a way to work with your programs directly. It is hands down one of the killer tools offered by Racket.

In the previous lesson, we wrote a very simple program that welcomes you to *Racket Weekend*. Our program was structured as a few definitions, which then got used in the `main` submodule. We interacted with our program by running it at the command line.

For simple programs, there's nothing really wrong with that. As programs grow in complexity, running the whole program this way becomes heavier. It takes more time and probably goes through computations that aren't terribly important to you. If you're debugging, you're likely interested in just a handful of functions.

There's another way to interact with your program: the read-eval-print loop (REPL).

The REPL offers a way to load your program, with its definitions, without running it. You can then call whatever functions you like and see the value.

The REPL is a great environment for manual testing, by which I mean writing out, by hand, which parts of your program you want to run. (I use the term 'manual' to distinguish this use of the REPL from *automated* testing, where the testing tool turns your tests into the 'executable' part of your program.)

To enter your program in the REPL, execute

```
$ racket
```

at the command line, with no arguments. You should see this:

```
Welcome to Racket v6.12.
>
```

(I'm using Racket 6.12 here. Your results may differ, but if they do, that's rather unlikely to be a problem.)

Hit Ctrl-D, or type `(exit)`—with the parentheses—to exit the REPL and return to your shell.

Think of the REPL as a shell for your program. Or another way to think about it: it's a kind of backdoor into your program.

Get into your program

Go to the directory where you've got the code that accompanies the previous lesson. Let's assume it's called `structure.rkt`. Now go back to the REPL (that is, invoke the `racket` command line program with no arguments). Let's enter your program. When you're in the REPL, type this (followed by return):

```
,enter "structure.rkt"
```

(Yes, with a comma at the beginning. And double quotes.) You should then see—after a bit of work—that the REPL hs 'entered' your program. Here's what I see:

```
"structure.rkt">
```

Just like a shell, the REPL is waiting for input. Type `course-num-days`, which we defined. You should see

```
"structure.rkt"> course-num-days
2
```

which looks right.

What's going on is that Racket is reading your expressions, evaluating them, and printing the results. In the case of `course-num-days`, that's quite straightforward: in our program, that's a variable (AKA identifier) with the value 2, so all Racket needs to do is look up the value of that variable and give it back to you (or, more precisely, print it out).

Let's evaluate something that takes a least a little bit more work than looking up the value of a variable. Tr y evaluating this:

```
(explanation 5)
```

You should see

```
"Racket Weekend is a 5-day course on Racket."
```

Fantastic.

What's going on is that Racket is taking your input—here, `(explanation 5)`—and evaluating it. We defined `explanation` as a (one-argument) function, whose value is a string.

Notice that the output from the REPL is a double-quoted string. The REPL is printing the result for you; there's no explicit mention of printing anything in the definition of `explanation`. It just builds a string and returns it. Let's try evaluating something that *does* print something:

```
(explain 10)
```

(Recall that the `explain` function passes its sole argument over to `explanation`. That function returns a string, which is, in turn, passed to Racket's `displayln` function, which prints a line to standard output.) You should see this:

```
Racket Weekend is a 10-day course on Racket.
```

Here, the REPL is showing you the results of printing something to standard output. `displayln` doesn't have a return value, so the REPL doesn't print anything beyond the string that gets pushed to standard output.

Try cheering:

```
(cheer)
```

You ought to see:

```
Cheers!
```

There are no double quotes, because, again, `(cheer)` itself has no value, but it does cause 'Cheers!' to be printed to standard output.

What else is going on here?

There's more here than simple greetings and cheering.

When working in the REPL, you can use all Racket functions. (That's not true, strictly speaking. The more correct answer is that it depends on what `#lang` we're using. But since we're using `racket` as our `#lang`, we've got a *lot* of functionality lying dormant around us.) We can play around here, going beyond the limits of our program, to do anything we like. Try this:

```
(+ 1 2 3)
```

You ought to see

```
6
```

What we saw in the first lesson, with `define`, `format`, and `displayln`, is true for all functions in Racket. The way it works is that if you want to have a function applied to some arguments, name the function first, then add the arguments, all of which is wrapped in parentheses. (In the extreme case, there are no arguments, so what you type is just the function name wrapped in parentheses. We saw that with `cheer` and `welcome`.)

Arguments themselves can be be parenthesized expressions: the expression

```
(+ 1 (+ 2 (* 3 4)))
```

should evaluate to 15. Try that out in the REPL.

Command history

The REPL is similar to most command shells in that it keeps a record of which expressions you've entered and it provides keyboard shortcuts for retrieving previous commands. In the REPL, hit the up key. If you followed the previous suggestion, you ought to see

```
(+ 1 (+ 2 (* 3 4)))
```

automatically plugged in. Hitting enter now will cause that expression to be evaluated. (If you didn't follow the previous suggestion, the up key will yield whatever was the previously submitted expression.)

Analogously, the down key will show you progressively newer expressions.

Even more commands

We've seen the `enter` command. Recall that it was prefixed by a comma. `enter` isn't really part of the Racket language; it's the name of a command used to control the REPL. There are lots of other commands. Try entering

```
,help
```

in the REPL.

You ought to see a couple dozen or so, of which `enter` is but one. The description for `enter` looks pretty complicated, but I hope it makes some intuitive sense.

Tracing functions

Let's wrap up this lesson by taking a look at a nice utility provided by the REPL: tracing functions. Assuming we're still working in `structure.rkt` (if you've steered away from it, give `,enter "structure.rkt"` to the REPL) and enter

```
,trace explanation
```

What we're asking the REPL to do is show us all the times that `explanation` gets called. Who calls explanation in `structure.rkt`? Only `explain`. And no one else calls `explain`. Try evaluating this:

```
(explain 25)
```

You ought to see a couple of lines added by the REPL's function tracing utility, together with the value of `(explain 25)`:

```
>(explanation 25)
<"Racket Weekend is a 25-day course on Racket."
Racket Weekend is a 25-day course on Racket.
```

We're seeing here that `explain` calls `explanation`, and the argument is the same. The REPL prints out, during the evaluation of `(explain 25)`, what looks like a prompt: the same > symbol is used, as though there were a 'sub-prompt' here. You don't have a chance to input anything, which makes it rather weird prompt.

Let's add more tracing. Enter this:

```
"structure.rkt"> ,trace explain
```

Let's evaluate the same expression as before, namely, `(explain 25)`. The output should be a bit more elaborated:

```
>(explain 25)
> (explanation 25)
< "Racket Weekend is a 25-day course on Racket."
Racket Weekend is a 25-day course on Racket.
<#<void>
```

The trace facility makes something explicit here that I previously claimed, but which had to be trusted: `explain` doesn't return anything. Or, rather, its return value is `void`, which we now see here in the trace.

To stop tracing `explain`, enter

```
"structure.rkt"> ,untrace explain
```

Takeaways

The read-eval-print loop (REPL) is a power tool for getting in to the Racket universe. You can evaluate arbitrary expressions and see their results, which aids debugging and testing.

(lesson 3) Structures

In which we begin to give our data more structure, creating new data types and seeing more kinds of Racket data. Additional glimpses into the world of functional programming are offered.

Make your own data types

Racket comes with a simple, yet powerful way to define your own data types. They're called structs (AKA structures). The idea to define data composed of various fields. Here's a simple example. Let's imagine that we're programming a warehouse, and we'd like to represent the concept of a storage bin, a place in the warehouse where you can store items. It's a complex piece of data: a storage bin has (by our stipulation here)

- a location (a string),
- a product (name) (a string)
- a quantity (an integer)
- the property of whether the bin can currently be picked from (a boolean value)

Here's how you might represent that in Racket

```
(struct bin
  (location
   product
   qty
   pickable?))
```

That's it. By introducing this, we've added a new basic data type to Racket. The way you create a new storage bin is by using the name of the structure type as if it were a function, then listing the arguments in order:

```
(bin "13-A-2"
     "Super Shoes"
     4
     #t)
```

(The #t bit is boolean true; it's a built-in data type in Racket (and, as you can see, has its own notation). Boolean false is written as #f.) What we mean by "13-A-2" is that the bin is in aisle 13, section A, shelf 2. Think of a warehouse as a 3-dimensional space. The first two pieces of information are coordinates that locate the bin on a plane, and the third piece of data indicates how high it is.

While working with locations-as-strings where, by convention, the coordinate are separated by dashes, we could also introduce a new datatype for locations. Let's see how that would work.

```
(struct location
  (aisle
   section
   shelf))
```

Nothing fancy. The way you create an instance of this new type is:

```
(location 13 "A" 2)
```

To create an instance of a bin where the location slot contains this location struct, try this:

```
(bin (location 13 "A" 2)
     "Super Shoes"
     20
     #t)
```

You can define the location first, and then use the newly defined value:

```
(define super-shoes-loc
  (location 13 "A" 2))

(bin super-shoes-loc
     "Super Shoes"
     20
     #t)
```

Accessing fields

The way you can access the fields of a struct are by using the struct name, a dash, and then the name of the field. Thus:

```
(location-aisle super-shoes-loc)
```

produces 13. Likewise,

```
(define super-shoes-bin
  (bin super-shoes-loc
       "Super Shoes"
       20
       #t))

(bin-qty super-shoes-bin)
```

produces 20.

Let's define a convenience function for accessing the aisle of a bin. We need to reach inside the `location` struct that's contained within the `location` field of a bin. Here's how that can be done:

```
(define (bin-aisle bin)
  (location-aisle (bin-location bin)))
```

Likewise, we could define functions that give us the section and shelf of a bin.

Let's define a function that makes a string representation of a location:

```
(define (location->string loc)
  (format "~a-~a-~a"
          (location-aisle loc)
          (location-section loc)
          (location-shelf loc)))
```

We use our old friend `format` here and all three accessor functions for `locations`.

Another approach to the same problem would be to define intermediate variables: variables that store the parts of the location. We can do that in Racket using `define`. `define` works not only at the top level (which is the only place we've used it up till now), but also inside the definitions of functions. It would look like this:

```
(define (location->string loc)
  (define aisle (location-aisle loc))
  (define section (location-section loc))
  (define shelf (location-shelf loc))
  (format "~a-~a-~a"
          aisle
          section
          shelf))
```

This is the first time we've seen a function definition that has multiple expressions in it. As you might suspect, the value of this function is the value of the last form. In this case, that's `format`, which yields a string.

If the value of a sequence of expressions is always the value of the final item, what's the point of the other items? The non-final expressions can be useful for a number of purposes:

- they can define variables, as we're doing here
- they can execute code that has side effects, such as modifying a value or sleeping for a few seconds
- the can check that something is true before proceeding, possibly raising an error.

We'll see an example of the third point later in this lesson.

Before moving on, a brief remark is in order about the scope of these intermediate definitions. Racket uses lexical scope, so `aisle`, `section`, and `shelf` aren't available outside this block encapsulated by the outer definition.

New structs from old

Given a struct, how can we modify it? There are a couple of ways of going about this question. One is by asking whether you really need to *modify* a structure, or whether you can go on by creating a *new* structure which is just like the old one, but differs in one respect or another. From a database point of view, think of this as the difference between updating and creating data. Both approaches are possible in Racket. The second (favoring creating new data) is more in line with the functional approach that is often found in Racket, but if you need to modify existing data, that's OK. Here's how you can do it.

Creating brand new structs

The way we've defined locations and bins above, where we just list the fields, is the approach that supports functional programming. The way this works is with `struct-copy`. Let's imagine that we want to transform a bin from pickable to unpickable (imagine that we've just scheduled a forklift to come over and move the palette that the bin is sitting on to a new location, and we don't want anyone to try to come over there and grab some Super Shoes).

```
(struct-copy bin
             super-shoes-bin
             [pickable? #f])
```

To use `struct-copy`, one supplies

- the name of the structure type (here, `bin`),
- the value you want to copy, and
- a list of (pairs of) fields with their new values.

To get our hands dirty, let's try this:

```
(define unpickable-super-shoes-bin
  (struct-copy bin
               super-shoes-bin
               [pickable? #f]))
```

If you were to evaluate

```
(bin-pickable? unpickable-super-shoes-bin)
```

you would get `#f`. That's clear: that was our intention, and it's met here. The quantity of shoes in the unpickable Super Shoes bin is unchanged from the original Super Shoes bin: `(bin-qty? unpickable-super-shoes-bin)` yields `20`. That's because `struct-copy` affects only those fields are given to it; everything else is copied over unchanged.

Let's see how struct copy can be used with multiple fields:

```
(define unpickable-&-out-of-stock-super-shoes-bin
  (struct-copy bin
               super-shoes-bin
               [pickable? #f]
               [qty 0]))
```

Here, we're taking the original Super Shoes bin and making a new Super Shoes bin, in the same location, that is both unpickable and has a quantity of 0 (or, in other words, is out-of-stock).

Here's a function that takes a bin and creates a new one that is unpickable and out-of-stock:

```
(define (make-out-of-stock b)
  (struct-copy bin
               b
               [pickable? #f]
               [qty 0]))
```

A function that picks an item from a bin decreases the qty by one. Is it an error if the qty is already zero (or, worse, negative)? I'd say so; let's see how that would work.

```
(define (pick b)
  (define qty (bin-qty b))
  (define loc (bin-location b))
  (define product (bin-product b))
  (unless (> qty 0)
    (error
      (format
         "Cannot pick from bin ~a (~a): qty is non-positive."
         (location->string loc)
         product)))
  (struct-copy bin
               b
               [qty (sub1 qty)]))
```

The first three expressions define new variables. The fourth expression is a check that the quantity of the bin from which we're trying to pick is positive. We're using a new function here, `error`, which I won't explain in detail. Suffice it to say that error causes and exception to be raised (that's the Racket way of putting it) and aborts the computation of `pick`.

Notice once again the prefix notation used throughout Racket: the check that the quantity of the bin is greater than 0 is written `(> qty 0)`. It may take a moment's though to accept that that expression has that meaning; perhaps it feels backwards. That's normal. Take a moment.

We're using the function > here. We could have just as well used <, which also exists:

```
(unless (< 0 qty)
  ; ...
  )
```

(By the way, the semicolon `;` is how one writes a comment in Racket. It goes until the end of the line.)

We're also using `unless` for the first time. As you might infer from its name, and from how we're using it here, this function has two parts: a condition to be checked, and an expression to be evaluated if the check fails. (If the check succeeds, the expression isn't evaluated.)

If the bin survives the check, we then proceed to the final expression, which is where we make a fresh bin whose qty is exactly one less than the qty of the given bin. The `sub1` subtracts one from its argument. (I'd say the `sub1` 'decrements' its argument, but that might misleadingly suggest that we'd be *changing* something in the given bin, or the argument.)

Modifying existing structs

In the previous section, the approach was predominantly functional: we defined functions that, given inputs (structs), produced fresh outputs. The inputs were never modified. Moreover, if we assign a value to a variable, we never changed that value or assigned some other value to the same variable.

If we want to modify our data, that's also possible in Racket. And for some problems, that makes sense and is easier to think through. And if you're just getting started with Racket, you may find the functional approach a bit odd at first. That's fine; I'm here to tell you that

We need to do a bit of work if we want to really modify parts of structs.

Transparent vs. opaque structs

Should you be able to look into your structs?

To see what I mean, go to the REPL and look at the value of, say,

```
(location 3 "A" 15)
```

You'll see that it's not terribly illuminating:

```
"main.rkt"> (location 3 "A" 15)
#<location>
```

All the REPL is telling us, it would appear, is that we've instantiated a structure type. We can't look inside: the `3`, `"A"`, and `15` are wrapped up inside the struct and are not accessible.

Structure types in Racket are, by default, *opaque*. That means that you can't look inside them except through the field accessors. It plays a role for the printing of structs (which is what our experiment with the REPL shows) as well as with equality (more on that in a minute).

To make a structure type transparent, supply a so-called *keyword argument* in the definition of the type, like so:

```
(struct location
  (aisle
   section
   shelf)
  #:transparent)
```

(The `#:transparent` bit is the keyword argument. Keyword arguments to functions are normally written with a `#:` prefix.)

After having redefined the `location` structure type to be transparent, we get a different result when we print out the struct:

```
"main.rkt"> (location "5" 3 "B")
(location "5" 3 "B")
```

Equality of structs

The opaque vs. transparent discussion also plays a role in the notion of equality of structures. To put it somewhat philosophically, when are two structures identical? Racket comes with a predicate, `equal?`, which checks whether two values are equal. If both arguments of `equal?` are structs (say, both are instances of `location`), what shall we do? Two approaches suggest themselves:

- If a structure type gets instantiated in two different places in a computation—even though the calls to the constructor were pairwise `equal?` values—the two structs are not `equal?`.
- Equality of structure types is to be determined recursively: if the arguments of the constructor are `equal?`, then the values are `equal?`, even if they are different memory locations.

There's no right or wrong here. The two approaches have their advantages and disadvantages, and make sense for some data types but probably not for others. I can't promulgate any hard rule; it's a design decision.

Takeaway

Structures are the first step to starting to give your data more form. You say what fields are needed, and Racket does the rest. Using `struct-copy`, you can create new instances of your structure type by copying parts of from an existing instance (also known as 'functional update', though that's a bit of a misnomer). Your structures may be opaque or transparent, which affects the notion of identity of structures as well as how they're printed; by default, structure types are opaque.

This lesson also started to reveal a bit more of real-world Racket programming not necessarily tied up with structures. We saw a couple of predicates—functions whose value is a boolean—having to do with numbers, and we started to check things using `unless`. We've also started to throw (or raise) exceptions, using `error`. We also saw how comments are written: one begins with a semicolon `;`. The comment goes to the end of the line.

A convention: predicates are usually written with a question mark at the end of their name. We saw that here with `pickable?` (a field name of a structure), `bin-pickable?` (a function we defined), and `equal?` (a built-in function that checks equality of its arguments).

(lesson 4) Object-oriented Programming

Structures are a straightforward way of adding structure to data. Organizing data into objects of various classes—object-oriented programming—is another common way of designing solutions to programming problems. Racket's got you covered there.

Lispy OOP

Racket is a member of the Lisp family of programming languages, which might make one think that there's no support for object-oriented programming, since, well, Lisp is super old and OOP is, although somewhat old, is not *that* old.

Racket comes with a delightful OOP layer out of the box. It's similar to programming with structs, as we saw in the previous lesson, but adds some richness to it.

Let's adapt the warehouse examples that we had last time to be in the OOP world.

Terminology. Mindset

Racket's take on OOP requires a bit of terminology.

It's a good thing to try to adapt the right mindset for defining classes in Racket. One of the main ideas to understand is that when we're defining classes, the rules of lexical scope are in effect. Class definitions will be a wrapped in a single block, like so:

```racket
(class object%
  ;; lots of stuff goes here
)
;; the internals of the class are inaccessible here
```

Such an approach is familiar from languages such as JavaScript (which has its own somewhat eclectic approach to object-orientation).

Multiple classes can be defined in a single Racket module (file). And, as we've seen previously, there's no need for any classes to be in a Racket file at all.

Moreover, if you're familiar with languages where everything in the language can be viewed as an object (that is, as an instance of some class), e.g., Smalltalk or Common Lisp, it is important to realize that that's not the case in Racket. Numbers, for instance, aren't objects.

Terminology: initialization argument. Field. Public method.

Getting started

Let's dive in and 'port' our structs from the previous lesson to classes. Locations are fairly straightforward; let's start with them. A skeletal version would look like this:

```
(define location%
  (class object%
    (init aisle
          section
          shelf)
    (define $aisle aisle)
    (define $section section)
    (define $shelf shelf)
    (super-new)
    (define/public (as-string)
      (format "~a-~a-~a"
              $aisle
              $section
              $shelf))))
```

As in other approaches to OOP, classes have a superclass. In Racket, the superclass from which all objects derive is called `object%`. It's a convention that names of classes end with the `%` symbol, and we're preserving it here when we say:

```
(define location%
  (class object%
    ; ...
  ))
```

`class` is what makes a class value. What's interesting in Racket is that classes are kinds of values. When we say

```
(define foo "bar")
```

we're saying that the identifier `foo` should have the value `"bar"`, a string; with

```
(define location%
  (class object%
    ; ...
  ))
```

we're likewise saying that `location%` should refer to a value, namely, a class of a certain kind.

When classes are instantiated, one gives certain values (if any) as arguments. The `(init ...)` bit specifies what the class constructor is like:

```
(define location%
  (class object%
    (init aisle
          section
          shelf)
    ; …
    ))
```

says that when instancing our class, three pieces of data are needed: an aisle, a section, and a shelf. These variables are available throughout the scope of the `class` form. One typically assigns the initialization arguments to internal fields (AKA class members), which is what we do next:

```
(define location%
  (class object%
    ; initialization part
    (define $aisle aisle)
    (define $section section)
    (define $shelf section)
    ; …
  ))
```

I'm using my own personal convention here of prefixing the names of identifiers with `$`; it's entirely optional, though the names should be different. The idea is that the values used to instantiate the class don't change (obviously, they're supplied once and consumed right away), but the internal state of a class may change during its lifetime. The `$aisle`, `$section`, and `$shelf` fields record the internal state, and, understandably, their initial values are the values used when instantiating the class.

There's one important piece of boilerplate:

```
(define location%
  (class object%
    ; init
    ; fields
    (super-new)
    ; …
  ))
```

The `(super-new)` bit is used for constructing the innards of the object. Make sure it's always there.

The final bit of our class definition is our first (public) method:

```
(define/public (as-string)
  (format "~a-~a-~a"
          $aisle
          $section
          $shelf))
```

What we're defining here is a function. Notice that the function takes no arguments. We use our old friend `format`. We refer to the `$aisle`, `$section`, and `$shelf` fields.

And that's it: our first class. We've specified how to construct it, what fields it has and how they receive their values at initialization time, and we've defined a public method.

Welcome to Racket OOP.

Instantiating classes

Classes are instantiated using the `new` function. One gives the class and any initialization arguments. To instantiate our `location%` class, try:

```
(new location%
     [aisle "13"]
     [section 5]
     [shelf 2])
```

One refers to the initialization arguments by name, and in brackets. One doesn't have to give the initialization arguments in the order that they appear in the classes's `init` specification; referring to them by name is sufficient.

Message passing

The mental model to have for method invocation in Racket is message passing. Accordingly, the function to use is `send`. One gives the class and any arguments. Thus, to make a string out of a location, try:

```
(define super-shoes-loc
  (new location%
       [aisle 13]
       [section "B"]
       [shelf 5]))
```

and then

```
(send super-shoes-loc as-string)
```

This yields the string `"13-B-5"`.

Defining bins

Let's continue with our discussion by defining bins using Racket OOP. This example will illustrate a bit more than locations, which, admittedly, are just straightforward containers for three values (aisle, section, and shelf). Bins, on the other hand, support a bit more functionality, e.g., picking, checking whether they're available, etc.

A direct translation of our `struct`-based approach is to say that a bin contains four pieces of data: a location (intended to be an instance of the `location%` class that we just defined), a product (a string), a qty (a non-negative integer), and a pickability attribute (indicating whether the bin should be among the bins that could be visited to pack a box & fulfill an order). By adapting our example above, we can get started this way:

```
(define bin%
  (class object%
    (init location
          product
          qty
          pickable?)
    (define $location location)
    (define $product product)
    (define $qty qty)
    (define $pickable? pickable?)
    (super-new)
    ; …
    ))
```

So far so good. Let's define a method that tells us whether a bin is available, which (recall from the previous lesson) is defined as (1) being pickable and (2) being non-empty (the bin contains at least one item). Inside the class definition, one could write this:

```
(define/public (is-available?)
  (and $pickable?
       (> $qty 0)))
```

How about picking (the act of taking one item from the bin)?

```
(define/public (pick!)
  (unless (> $qty 0)
    (error
     (format "Cannot pick non-positive qty ~a from bin ~a."
             $qty
             (send $location as-string))))
  (set! $qty (sub1 $qty)))
```

As before, we check whether we *can* pick from the bin. (One could perhaps define a way for picks to occur if the quantity of a product in a bin is zero, or even negative. This might make sense if one is dealing with 'virtual' picks. Here we consider bins to be physical objects, where a pick must be realized. With this understanding, the bin had better contain something!) Notice that we're using the fields `$qty`, and, when handling the error case, `$location` and `$product`, too. We `send` a message to our `$location`, asking for a nice string representation of it.

Getters and setters

One thing you might have expected to be available with our two classes, but in fact isn't, are getters and setters. As things stand with our classes, as we've defined them up to now, we have no way of accessing fields or setting them to new values. With `bin%`, we've defined `pick!`, which of course decrements the quantity of the bin. But there's no way to even access the current quantity of the bin, nor its location, nor its product, nor its pickability. Similarly, for the `location%` class, we're able to generate a string representation of an instance, but we can't access any of its fields.

The idea is that if there's no method, there's no way. Inside the class definition (that is, inside the `class` form), you can access whatever you want. Outside the `class`, you can communicate with class instances only through public methods. And the only public methods are the ones that you specify. There are no defaults that can be relied upon.

Going back to the `location%` class, let's define a getter for the aisle. It's rather straightforward:

```
(define/public (get-aisle)
  $aisle)
```

And here's a getter for the `bin%` class:

```
(define bin%
  (class object%
    ; …
    (define/public (get-aisle)
      $aisle)
    ; …
  ))
```

Bins contain locations, which themselves contain an aisle. The aisle of a bin could then be said to be the aisle of its location:

```
(define bin%
  (class object%
    ; …
    (define/public (get-aisle)
      (send $location get-aisle))
    ; …
  ))
```

What about setters? The job is accomplished using `set!`:

```
(define bin%
  (class object%
    ; …
    (define/public (make-unpickable!)
      (set! $pickable? #f))
    ; …
  ))
```

Takeaways

Using `class`, one defines new classes (starting with the root class `object%`). By convention, class names end with %. All variables are lexically scoped in a class and everything is, by default, private. One specifies initialization arguments for instantiating the class, possibly with default values. `new` is how to instantiate a class:

```
(new your-class%
     init-arg-1
     init-arg-2
     ...)
```

`define/public` is how to define (public) methods; invoke them using `send`, like this:

```
(send object
      method arg-1
      arg-2
      ...)
```

Using `set!` inside a class, you can modify the state of an object. (In fact, `set!` is used to assign a value to a variable; it's not specific to the OOP part of Racket.)

(lesson 5) Contracts

Usually the functions we write are intended to take arguments within a certain well-defined range of values, and return something that is likewise within some specifiable range. Racket's notion of contracts provides a way to ensure that your function fulfills its promises, provided its arguments, in turn, fulfill theirs.

The basics

Contracts provide a way to ensure that your function provides values that satisfy certain conditions, provided that the arguments also satisfy certain conditions.

Let's take a simple look at a contract.

```
(define (f x)
  (* x (add1 x)))
```

This function is supposed to take a number and multiply it by its successor. There's no check here that the argument `x` is a number. Racket is happy to pass, for instance, a string or a struct to this function. Naturally, it will error out if the argument is not a number, because `*` and `add1` functions are supposed to be given numbers as arguments. In a REPL, here's what happens:

```
"contracts.rkt"> (f 456)
912
```

If we give `f` a string, we get:

```
"contracts.rkt"> (f "ouch")
; *: contract violation
;   expected: number?
;   given: "ouch"
;   argument position: 1st
; [,bt for context]
```

Clearly something has gone wrong here. Indeed, looking at the error message, we can see that there's already something going here having to do with contracts. Racket is complaining about the first argument given to `*`. It's supposed to be a number, but we gave it `"ouch"`.

A simple contract here would be to specify that the argument is a number. And, under that condition, the value of the function will also be a number. Use `define/contract`:

```
(define/contract (f x)
  (number? . -> . number?)
  (* x (add1 x)))
```

The contract is the bit on the second line:

```
(number? . -> . number?)
```

This can be read as: I accept a single, non-optional argument. It is supposed to be a number. When I return, I will produce a value that is also a number.

(By the way, `number?` is a predicate in Racket (which is a fancy way of talking about functions that usually take a single argument and produce a boolean value). `number?` isn't a part of the little language of contracts that we're using here. You can use `number?` anywhere in your programs.)

If we give 456 to the redefined `f`, we get 912, as before. A couple invisible steps took place, though, that didn't take place with the 'uncontracted' `f`:

1. a check was made that 456 is a `number?` (that is, that `(number? 456)` produces a logically true value), and
2. once `f` finished executing, a check was made that its value is a number.

Let's see what happens when we give `f` something that breaks the contract:

```
(f "ouch")
; f: contract violation
; expected: number?
; given: "ouch"
; in: the 1st argument of
; (-> number? number?)
; contract from: (function f)
; blaming: /contracts.rkt
; (assuming the contract is correct)
; at: /contracts.rkt:10.18
; [,bt for context]
```

Ouch!

Notice that the error message here refers to the contract. What is perhaps not immediately obvious is that our contract,

```
(number? . -> . number?)
```

has been rewritten in the official syntax:

```
(-> number? number?)
```

The `. -> .` bit uses infix notation, where the arguments are to the left of the `. -> .` and the return value is to the right. I find this easier to read than the official notation, where one writes `->` first, then lists the arguments, and then, finally, the return value.

Notice also that there's a concept of 'blame': if a contract fails, there's a way to trace the spot where the contract was broken.

As expected, we don't even attempt to multiply `"ouch"` and `(add1 "ouch")`; computation didn't even reach that point owing to our contract checking.

Using other predicates

We used `number?` in our contract, but we could have used other predicates, too. Let's try this:

```
(define/contract (f x)
  (integer? . -> . integer?)
  (* x (add1 x)))
```

Here we're restricting attention to integers. Of course, `*` and `add1` make sense for numbers, too. But we can imagine that the only places where we'd really use `f` are places where the argument is an integer. Our contract then expresses our intention of how `f` should work in our overall program, rather than the widest possible inputs that `f` could accept and still make sense. To bring out that point, consider another function:

```
(define/contract (cool x y)
  (number? number? . -> . number?)
  (f (max (ceiling x)
          (ceiling y))))
```

Here we're considering a new function that uses `f` as well as a couple of other numeric functions `max` (which returns the maximum of its arguments) and `ceiling` (which returns the next integer bigger than its argument, a coarse-grained 'round up' function). Here, although we could certainly stick with the contract `(number? . -> . number?)` for `f`, using integers more closely captures the way we expect `f` to be used in our program.

Let's apply `cool` to some values:

```
"contracts.rkt"> (cool -4 3)
12
```

Interesting. Let's keep going:

```
"contracts.rkt"> (cool 4 99)
9900
```

Perhaps we can sharpen the contract on `cool` to ensure that it's return value is an integer. (Try it.)

We can sharpen things even further by insisting that `f` and `cool` return so-called *exact integers*. These are integers that have no decimal part at all. Things like 13.0 count, for Racket, as integers even though they also look like floating point numbers. Let's try this then:

```
(define/contract (f x)
  (exact-integer? . -> . exact-integer?)
  (* x (add1 x)))

(define/contract (cool x y)
  (number? number? . -> . exact-integer?)
  (f (max (ceiling x)
          (ceiling y))))
```

Now try playing around with some values for `cool`. Using the same arguments as before returns the same values.

So let's start stretching things:

```
"contracts.rkt"> (cool -4 3.56)
; f: contract violation
;   expected: exact-integer?
;   given: 4.0
;   in: the 1st argument of
;       (-> exact-integer? exact-integer?)
;   contract from: (function f)
;   blaming: /contracts.rkt
;    (assuming the contract is correct)
;   at: /contracts.rkt:18.18
```

Boom!

What happened here?

Something was given 4.0—an integer, but not an *exact* integer. It was *f*. Somehow, `cool` produced this value. How did that happen?

Playing around in the REPL, we see that if both of `max`'s arguments are exact integers, then its value is also an exact integer. So it must have been `ceiling`. Hunting around in the documentation (or just trusting me here), we need to find a function that takes an non-exact integer and turns it into an exact one. The `inexact->exact` function looks appropriate. Let's change the definition of `cool`:

```
(define/contract (cool x y)
  (number? number? . -> . exact-integer?)
  (f (inexact->exact
      (max (ceiling x)
           (ceiling y)))))
```

Now when we use the arguments -4 and 3.56 that previously got us into trouble, we get:

```
"contracts.rkt"> (cool -4 3.56)
20
```

Just what we wanted.

Trying to break contracts

The Racket contracts suite comes with a fantastic utility that tries to `break` contracts. Given a list of values, it looks up whether any of them have contracts and, if so, tries to cleverly find a way to break them. It doesn't always work, but as a debugging tool for your programs, it's worth its weight in gold.

We 'discovered' how `f` could generate a value that cannot (legally) be given to `f`. Here's how that might have been found automatically:

```
"contracts.rkt"> (contract-exercise cool f)
; f: contract violation
;   expected: exact-integer?
;   given: -0.0
;   in: the 1st argument of
;       (-> exact-integer? exact-integer?)
;   contract from: (function f)
;   blaming: /contracts.rkt
;    (assuming the contract is correct)
;   at: /contracts.rkt:24.18
```

We pass both functions because we're interested in the interaction between them. We find here a contract violation, again of `f`; it somehow received -0.0 (a [signed!] non-exact integer) as an argument. One has to sleuth around somewhat to try to reconstruct how it was possible that `cool` might have done that, but you're given a nice clue here. Somehow, we have to find a way to get -0.0. Keeping things very simple, we notice that if we give -0.0 and -0.0

to max, we get -0.0. So now the problem has been reduced to finding a way to get -0.0 out of ceiling. Again, taking the very simple work-backwards approach, we find, to our surprise, that ceiling -0.0 produces -0.0.

And there's the solution to the problem. We have to change our code that that either (1) we coerce the value of ceiling to an exact integer, or (2) we coerce the value of max to an exact integer.

But are we sure we're done? Imagine that we've fixed the code for cool so that it ensures that the argument to f is an exact integer.

Using contract-exercise *again*, we find something surprising:

```
"contracts.rkt"> (contract-exercise cool f)
; ceiling: contract violation
;   expected: real?
;   given: 623893768-39i
```

Now, the generator has produced a value that violates the contract for ceiling! It has produced a complex number (a number with both a real and imaginary part). ceiling accepts only real numbers as inputs. contract-exercise has produced a value that goes outside our intended range of application. We didn't mean to work with complex numbers; we want to work with real numbers only. So this gives us the change to chance to adjust the contract for cool:

```
(define/contract (cool x y)
  (real? real? . -> . exact-integer?)
  (f (inexact->exact
      (max (ceiling x)
           (ceiling y)))))
```

At this point I, even though I have experience with this tool, wanted to conclude this section by saying ‘We’re done, right?’ But then I checked my own work, using the tool I’ve been talking about here.

It’s *still* broken! I love edge cases, but `contract-exercise` loves edge cases even more. Watch this:

```
"contracts.rkt"> (contract-exercise cool f)
; inexact->exact: no exact representation
; number: +inf.0
; [,bt for context]
```

Here, `+inf.0` is a kind of real number that represents positive infinity. (One can quibble of whether that counts as a real number, but that’s the way it is.) We need to exclude positive infinity and other extremes of the Racket number system. We have no sensible way to handle them, except perhaps by simply detecting them and returning them. Let’s extend our contract to handle this case:

```
(define/contract (cool x y)
  (real? real? . -> . (or/c exact-integer?
                             (=/c +inf.0)))
  (define m (max (ceiling x)
                 (ceiling y)))
  (if (= m +inf.0)
      +inf.0
      (f (inexact->exact m))))
```

We’re using two new bits of the contracts language: `or/c` and `=/c`. The `or/c` bit builds a contract out of two contracts. The `(=/c +inf.0)` bit is a contract that holds exactly when the value supplied to it is equal (as a number—that’s what = is in Racket) to `+inf.0`.

Notice that we also handle this case. We store the result of the `max` computation in a new variable, `m`. We then check `m`: is is positive infinity? If so, return that. Otherwise, go to `f` (that's the normal case).

Takeaways

- The standard contract notation for functions is `(-> arg1 arg2 ...` A variant of this notation is `(arg1 arg2 ... . -> . value)`.
- `contract-exercise` tries to break contracts: it finds values that meet the conditions that the function's arguments are supposed to meet, but which break the promise associated with the return value. It is useful as a tool for checking your own functions. There may be edge cases that you didn't think of.
- We've seen a handful of useful contract makers: `=/c`, `or/c` (there's also `and/c`, and simple predicates, such as `integer?`, `real?`, `exact-integer?`, and so on.

(lesson 6) Functional Programming Bricolage

Racket encourages functional programming without being doctrinaire about it. In this lesson we'll learn about some of the idioms and basic ideas of FP, Racket-style.

Prelude: Lists

To help illustrate some ideas that feature in the coming discussion of functional programming, and to perhaps shore up some uncertainty you might be facing about some data showing up in previous lessons, let's do a mini-lesson in a *widely* used data structure in Racket: the humble list.

Lists of values are usually made by using the `list` function. Thus,

```
(list "a" 4 #t)
```

makes a 3-element list containing the string `"a"`, the integer `4`, and the boolean value `#t`. To refer to an element of a list, use `list-ref`:

```
(define a (list "a" 4 #t))
(list-ref a 0) ; "a"
(list-ref a 1) ; 4
(list-ref a 2) ; #t
```

(As you might imagine, using an out-of-bounds index leads to an error. Give it a try!) There are also some built-in shorthands for accessing certain elements in a list by their position:

```
(define a (list "a" 4 #t))
(first a) ; "a"
(second a) ; 4
(third a) ; #t
```

(As before, using `fourth`, `fifth`, and so on leads to an error.) The length of a list can be determined by using `length`:

```
(define a (list "a" 4 #t))
(length a) ; 3
```

The tail of a list—the list containing everything except the first element—can be got using `rest`:

```
(define a (list "a" 4 #t))
(rest a) ; '(4 #t)
```

To determine whether a list is empty, use `empty?`.

Ingredients of functional programming

Here are, in my view, the main ingredients of functional programming. I try to not get too worked about what is and what is not functional programming, and I likewise try to avoid going overboard with highly mathematical concepts that distract me from programming. (And I say this as a mathematician. I love mathematics, but I'm wary of mathematical discussions when talking about the nature of functional programming. I don't like the impression they leave: that one needs to know a ton of math to get a grip on FP.)

Functions are values

When writing a program, we often divide the work to be done by writing functions. We're already familiar, then, with the value of functions. One of the ideas found throughout Racket (and is certainly not unique to it) is that functions are values, too, just like numbers, strings, objects, and so on. One can pass functions as arguments to other functions. Realizing this may take a while, but it will open new doors for you when it comes to the way you attack a problem and design a program to solve it.

Another word used for this is 'higher-order functions'.

Preference for immutability

Immutability is the idea that data doesn't change. Given a string, one can't, say, change its first character. Given a list of numbers, you can't change one of them. Given a sequence of bytes, you can't XOR it with some other byte string. The immutability way of thinking about this would be:

- Given a string, produce a new string just like it, but with its first character uppercased. (The original string is unaltered.)
- Given a list of numbers, produce a new list where the numbers are doubled. (The original list is unchanged.)
- Given two byte strings, XOR them. (Neither byte string changes.)

I use the word 'preference' here deliberately. Many Racket programs deviate from this principle in some places. Racket allows you to modify data, and we've already done it: `set!` assigns a new value to a variable, `pick!` to modify a storage bin in a warehouse, and `set-unpickable!` and other functions for bins. Some problems become awkward or cumbersome if one insists on doing them in an immutable data.

That said, a number of Racket's data structures are built with immutability in mind.

Referential transparency

The flip side of immutability is referential transparency. That's a fancy word that means that one can reason with (or think about) a program by leaning on the how variables get passed around. This allows one to trace back the way variables get used all the way back to their source (that is, where they were defined). If you've done some programming, I bet you've had the experience of being puzzled by the ways in which data gets modified, and not really knowing why a variable carries a certain value. With functional programming, one of the byproducts is that it is easier to say how a variable got a certain value.

Functions as arguments

Let's take a look at a handful of the useful tools that Racket comes with for doing functional programming. More specifically, we'll talk here about ways in which functions are used as values (that is, how functions can be arguments of a function).

map: apply a function to elements of a list

```
(define/contract (cube x)
  (number? . -> . number?)
  (* x x x))
```

And let's imagine we have a list of numbers:

```
(define/contract l
  (listof number?)
  (list 4 -9 45.132 45-2i))
```

(By the way, we're seeing here yet another contract builder: `listof`.) (Did you see how we wrote a complex number?) The way we can apply `cube` to each element of `l` separately is:

```
(map cube l)
```

We then get the result

```
'(64 -729 91929.25453996798 90585-12142i)
```

Or consider Hamlet:

```
(define/contract hamlet
  string?
  "To be or not to be, that is the question.")
```

Let's make these words bigger:

```
(map uppercase (string-split hamlet))
```

where `uppercase` is defined as

```
(define/contract (uppercase str)
  (string? . -> . string?)
  (cond ((string=? str "")
         "")
        (else
         (define c (string-ref str 0))
         (format "~a~a"
                 (char-upcase c)
                 (substring str 1)))))
```

(Notice that we have to handle the empty string as a special case. `string-ref` gets the character in the string at a given index. `substring` extracts a substring of a given string; we want the whole string, minus its first character, i.e., starting at index 1 and going to the end.) We get:

```
'("To" "Be" "Or" "Not" "To" "Be,"
  "That" "Is" "The" "Question.")
```

Yes, It Is.

In the two examples so far, the functions we've used (`cube` and `upper-case`) take a single argument. `map` can also take a function that takes two arguments, provided one gives two lists (and the lists need to be of the same length). It goes like this:

```
(define/contract (square-sum x y)
  (number? number? . -> . number?)
  (expt (+ x y) 2))
```

(Here, `expt` is a function that raises its first argument to the power indicated by the second argument. Writing 2 here is just a fancy way of squaring.) Given two lists of numbers,

```
(define nums-1 (list -5 2 4.6))

(define nums-2 (list 4 01.3 1980))
```

one then uses `map` like so:

```
(map square-sum nums-1 nums-2)
```

which yields

```
'(1 10.889999999999999 3938637.1599999997)
```

foldl & foldr: reduce a list to another value

With `map`, we applying a function to a list (or lists) of values in such a way that there's no interaction between the computations. We just get a list back that's as long as the list that was given. That's fine and well for some applications, but often we need to accumulate a value as we go through a list. We need to have a way to refer to the `current` value of the computation. In the FP world, this is known as *reducing* a list to some value.

The general scheme is this: we are given a list and function that takes two arguments: (1) the 'current' value, and (2) the next element of the list. We get to specify what the initial value is. Thus, we're talking about a function that takes three arguments: (a) the initial value, (b) the two-argument 'accumulator' function, and (c) the list of values that we should work through.

An illustrative example (also found in the Racket documentation):

```
(foldl +
       0
       (list 1 2 3 4 5))
```

yields 15, which is the result of applying + to successive sums. The computation works like this:

1. + receives 0 (the initial argument) and 1 (the first element of the list). Result: 1.
2. + receives 1 (the result of the previous step) and 2 (the second element of the list). Result: 3.
3. + receives 3 (the result of the previous step) and 3 (the third element of the list). Result: 6.
4. + receives 6 (the result of the previous step) and 4 (the fourth element of the list). Result: 10.

5. + receives 10 (the result of the previous step) and 5 (the fifth element of the list). Result: 15.
6. the list is now empty, so we return what we've accumulated so far. Result: 15.

Notice that the first argument of `foldl` is a function, whereas the other two argument are 'data'. One of the core ideas is of functional programming is that functions (like +) are data, too, and you can refer to them just by using their name.

The name `foldl` contains 'fold', which is another word used in the FP world for what's going on here (taking a list and 'folding' it down to a single value).

Another function, `foldr`, does the same thing as `foldl`, but starts from the end of the list rather than the beginning. The '`r`' stands for *right* (and the 'l' stands for *left*).

Looping

One of the challenges programmer encounter early on when they're first learning about functional programming is the idea of loops. If one wants to go through the elements of a list (or array), repeatedly assigning one and the same variable a different value, and perhaps even incrementing a counter, it seems like we're throwing referential transparency out of the window.

Indeed, looping is often tackled in a couple ways:

- **recursion**: design a 'helper' function that takes a list as argument and possibly calls itself with a smaller sublist. The final return value is built up as one scans through the list. For complicated functions, recursion is a powerful tool to know.
- **looping functions**: these functions take a list (or lists) as arguments together with other functions. They're tremendously useful.

Recursion is a powerful problem-solving technique. I bet you've seen it before, and you know that learning recursion is not always easy. Instead of focusing on that, let's take a look at a couple of examples of functions that, internally, are written using recursion and which spare you the need to roll your own recursive solution.

andmap/ormap

You've got a list of items and you want to know whether they're all true (or, more commonly, you want to know whether, after you apply a function to each of them, whether they're all true). As soon as even one of them is found to be non-true, the computation can stop; there's no point going further, because we know what the final result will be.

Analogously, imagine having a list and needing to know whether any of them is true. Upon finding even one that's true, you can stop.

The `andmap` and `ormap` functions are designed for this purpose. They both have the same signature: they take a function and a list. Here's a typical case:

```
(andmap even?
        (list 313 55 94 4551 5+5i))
```

(`even?` is a predicate that determines whether its argument is an even integer.) We walk down the list till we get to 94, and we stop there, returning `#t`. If the 94 weren't in the list, we'd have walked down the whole list, failing to find any even number, and would have returned `#f`.

To convince yourself that computation did stop at 94, try evaluating the predicate on the last element of the list:

```
(even? 5+5i)
```

(You should get an error. There's a contract violation here: `even?` does not want to deal with complex numbers!)

We can define new functions to be passed in to `andmap/ormap`. (Such functions can be passed in *anywhere*, since they're just functions.) Here's a function that determines whether its argument is an imaginary number (a complex number that isn't a real number, or, equivalently, a complex number whose imaginary part is non-zero):

```
(define (imaginary? x)
  (and (number? x)
       (not (real? x))))
```

Now we can try again:

```
(ormap imaginary?
       (list 313 5.6 94 4.551 5+5i))
```

As you might imagine, we have to walk down the whole list before finding the first imaginary number, and `ormap` returns `#t`.

For loops, Racket-style

The simplest case of `for` takes a list, runs over it, and does something. It doesn't return anything; we just go through the list, doing something, and that's it. Here's an example where we print the cubes of some numbers:

```
(define/contract (talk-about-cubes nums)
  ((listof number?) . -> . void?)
  (for ([n nums])
    (displayln (format "The cube of ~a is ~a."
                       n
                       (cube n)))))
```

The idea is that we're given a list of numbers and print out their cubes. To do that, we walk down the list and simply print the line (using `displayln`). We start out with the contract—notice that the function doesn't really return anything, so we use `void?` as the return value 'type'—and then move on to the `for`. As we go through the `nums` list, we'll let `n` be the 'current' value.

When running function with the list

```
(list 4 1.9 -5 3+2i)
```

you'll see

```
The cube of 4 is 64.
The cube of 1.9 is 6.858999999999999.
The cube of -5 is -125.
The cube of 3+2i is -9+46i.
```

`for` can take multiple lists. It iterates through them independently. Try this:

```
(for ([a (list "a" "b" "c")]
      [b (list 1 2 3)])
  (displayln (format "a = ~a and b = ~a"
                     a
                     b)))
```

You'll see:

```
a = a and b = 1
a = b and b = 2
a = c and b = 3
```

Do you remember our storage bins, in the previous lesson on structures (and repeated again in the lesson on OOP)? Let's reuse that. We'll use `require` to import whatever we exported there:

```
(require (file "bin.rkt"))
```

Given a list of bins and a list of of new quantities, imagine that we want to update the qty of the bin. The following does the job:

```
(define (set-qtys-for-bins! bins qtys)
  (for ([b bins]
        [q qtys])
    (set-bin-qty! b q)))
```

Mix and match

In the immutable data world, we have some basic data that gets applied to various functions to make new ones. The idea is to view nearly all data as being built up from smaller pieces. Thus:

- a string is built up from a sequence of characters;
- a list is built from its members (which themselves might be lists),
- a structure was constructed by a function call where arguments were given to it

`match` is like `switch` on steroids.

Let's see a simple example that illustrates the main idea:

```
(match "143"
  [(? integer?)
   (error "Will never happen.")]
  ["134"
   (error "Close but no cigar.")]
  [(pregexp "1[0-9]3")
   "yes"]
  [else
   "you shouldn't see me"])
```

We start with the string `"143"`. When then go down the list of `match` conditions, one at a time. We ask of `"143"`:

1. 'Are you an integer?' No, it's a string.
2. 'Are you *exactly* the string `"134"`?' No, they are different strings.
3. 'Do you match the pattern 'a `1`, followed by a digit, followed by a `3`'?' Yes.

Computation ends there, and we evaluate whatever is in the rest of the matching 'case'. In this case, it is the constant value `"yes"`. So if we evaluate this chunk of code, we get

```
"yes"
```

A fallback case—indicated by the `else`—was ignored. If the pattern hadn't matched, computation would have proceeded to that point.

Let's match a list and introduce another powerful idea provided by `match`. This time, we'll match not on a literal value (as we did in the previous example), but on a variable. Let's say:

```
(define louis
  (list "l" "ou" "i" "s"))
```

`louis` is a list of strings. Consider this:

```
(match louis
  [(? string?)
   "no way José"]
  [(list "l")
   "list contains more than that"]
  [(list-rest "l" "o" more)
   "close, but not quite"]
  [(list-rest "l" (pregexp "o[a-z]") more)
   (displayln
    (format "the second element of the list is: ~a"
            (second louis)))
   more]
  [else
   "ouch!"])
```

As before, we go down the list of match conditions and step into the one that matches. Notice that, at the end, there's a fallback condition, so we're guaranteed that we'll see `"ouch"` if none of the previous conditions matches. We ask the following questions about `louis`:

1. 'Are you a string?' No, it's a list.
2. 'Are you a list whose first element is (exactly) the string `"l"`, without any further elements?' No, because although `louis` starts with `"l"`, it has more elements.
3. 'Are you a list that starts with `"l"` and `"o"`? If so, let `more` be the remaining elements of the list.' No, because although `louis` starts with `"l"`, the second element is the string `"oi"`, not `"o"`. The assignment of `more` to the remaining elements is then moot.
4. 'Are you a list that starts with `"l"`, whose second element is a string that matches the pattern 'contains an o followed by another lowercase letter'? If so, let `more` be the remaining elements of the list.' Yes! In this case,

`more` is the list consisting of the elements that haven't been mentioned in the pattern. Which is to say, the third and fourth elements of `louis`.

Thus, evaluating this code yields

```
the second element of the list is: ou
'("i" "s")
```

Notice that in the body of the condition that matched, we called `displayln`. The body can contain any number of statements (at least one, of course). The final value returned from the match is the final value returned by the body of the matching condition.

What if nothing matches? Imagine that we trash the condition that we know matches, as well as the fallback `else` condition. Consider, in other words, this snippet:

```
(match louis
  [(? string?)
   "no way José"]
  [(list "l")
   "list contains more than that"]
  [(list-rest "l" "o" more)
   "close, but not quite"])
```

What happens?

```
match: no matching clause for '("l" "ou" "i" "s")
  location...:
   fp.rkt:112:0
  context...:
   /Racket v6.12/collects/racket/match/runtime.rkt:24:0:
match:error
   /fp.rkt: [running body]
```

Ouch! Walking into the `match` arena without a winning strategy can lead to an error. Having a fallback condition is usually a good idea.

Takeaways

We've taken in a lot here: lots of new built-in functions, and a glimpse of a new (or maybe not so new) way of approaching certain common programming tasks. We've talked about lists, matching, looping/iteration à la FP, and encountered some fancy terms like *referential transparency* and `immutability`.

As you immerse yourself more in Racket, these ideas will become, in time, more and more part of how you approach problems.

(lesson 7) Testing your programs

Making sure that your functions do what you think they do—writing tests—is bread & butter programming. Racket offers a delightfully straightforward approach to testing that will make you really want to test.

Using the REPL to run whatever code you like and get instant feedback can count as a kind of testing. But what we'll talk about here is testing outside the REPL; somewhat more robust sense of 'test' where the tests constitute a special kind of program, with their own structure. In short, we're talking about unit tests.

Testing with raco & RackUnit

Before talking about writing tests, let's talk about running them. Let's assume, then, that we've got some tests. The way to run them is with **raco**, the commandline utility for Racket. If we want to test the file `foo.rkt`, we'd run, at the command line:

```
$ raco foo.rkt
```

If `foo.rkt` doesn't have any tests, you'll see:

```
$ raco test foo.rkt
raco test: "foo.rkt"
```

And that's it. As we write more tests, we'll see more interesting output.

The test submodule

As we discussed in the first lesson, Racket programs use the concept of modules. When you write `#lang racket` at the top of a file, what's in fact happening is that you're telling Racket that you're starting a module, and the stuff contained in it is everything in the file. Thus, everything you write is implicitly included in a module.

A module can be included in another one. We've seen that, in lesson 1, with the `main` submodule. Putting code in that module will be considered the executable part of your program; it will be run when you execute your program on the command line (à la `racket foo.rkt`) or within DrRacket (when you click on the Run button).

Thus, `main` is a kind of special name for a submodule. There's another one: `test`. Things included in the `test` submodule will be considered tests by `raco test`.

Let's see how that goes. Let's return to some earlier code and start testing it.

Where to put the tests?

There are a couple ways to structure your tests:

- put them into a separate file (test for `foo.rkt` could go in, say, `foo-test.rkt`); or
- interweave them with your code.

RackUnit basics

When getting started with testing, one needs to import RackUnit. Do it like this:

```
(module+ test
  (require rackunit))
```

This way, only the `test` submodule contains the RackUnit bindings, keeping your main module clean. You can import code form elsewhere with:

```
(require (file "bin.rkt"))
```

Then, put your tests in the `test` submodule, like so:

```
(module+ test
  (check-true (location? super-shoes-loc)))
```

(The `module+` bit is fantastic: it gathers up all code wrapped in it, and produces a module whose contents are those forms. This spares us from having to define all our tests in a single place; we can scatter them about, and write tests next to code.)

To then run all your tests, invoke RackUnit from the command like like so:

```
$ raco test bin-test.rkt
```

This is not unlike running

```
$ racket bin-test.rkt
```

except that the `test` submodule will be 'executed' rather than the `main` submodule.

I have already used a bit of RackUnit without explanation: the `check-true` above comes from RackUnit. As you might imagine, it checks whether a certain form (not a *value*) evaluates to a true value. (What I'm hinting at is that `check-true` and all sorts of other RackUnit primitives aren't functions, but macros.) Our `super-shoes-loc` from before is an instance of the `location` structure type. And that's just what we're checking here with our test.

Let's write another test:

```
(check-= 5
         (location-aisle super-shoes-loc)
         0)
```

For brevity's sake, I'm omitting the fact that this form is wrapped inside the `test` submodule. Strictly speaking, the whole snippet of code that I'm talking about looks like this:

```
(module+ test
  ; tests
  (check-= 5
           (location-aisle super-shoes-loc)
           0)
  ; more tests
)
```

If you now run RackUnit, you'll find that this test doesn't work:

```
$ raco test bin-test.rkt
raco test: (submod "bin-test.rkt" test)
--------
FAILURE
name:       check-=
location:   bin-test.rkt:10:2
params:     '(5 13 0)
--------
1/2 test failures
```

By the way, the = part of `check-=` is for checking numeric equality. The `0` part is the tolerance that we allow for the values that are supposed to be equal. We're checking exact integers here, so our tolerance is very low; the values had better be exactly equal. (If we were testing code that involves fussy floating-point calculations, we may well allow some fuzziness.)

Let's test our `pick!` function. Let's test that `pick` a few assertions:

- `pick` does not modify its argument (specifically, it does not change the quantity contained in a bin);
- `pick` reduces the quantity in a bin by one;
- if `pick` is called on an unpickable bin, an exception is thrown;
- if the quantity in a bin is 0 and we try to `pick` it, an exception is thrown.

Let's take care of the first two tests:

```
(module+ test
  (define b (bin (location "A" 5 1)
                 "Fancy Pants"
                 1
                 #t))
  (define picked-b (pick b))
  (check-= 1
           (bin-qty b)
           0)
  (check-= 0
           (bin-qty picked-b)
           0))
```

Notice that we can use `define` in the submodule. The scope of the definition is the submodule; the `b` and `picked-b` that we're defining here aren't visible outside the submodule.

How can we test that an error gets thrown when executing some code? What we need to do is wrap the code we'd like to execute in a so-called thunk—a zero-argument function, a kind of 'just do it'. We also use a different bit of the RackUnit language. Here's how it looks:

```
(module+ test
  (check-exn exn:fail?
             (lambda ()
               (pick picked-b))))
```

A few things to notice here:

- We're using check-exn. The 'exn' is for *exception*.
- We've wrapped the (pick picked-b) in a zero-argument function, using lambda () …).
- We're using exn:fail? to check the 'kind' of exception being thrown. exn:fail? is a predicate.

Takeaways

Unit testing with Racket is done by writing tests in the test submodule. You can write the tests either in the same module as the code to be tested—possibly even adjacent to the function to be tested—or in a separate file. Use raco test file.rkt to run the tests in file.rkt. RackUnit comes with a bunch of built-in test tools, such as check-true, check-false (and check-not-false), and so on.

We've introduced lambda here and the idea of a *thunk*, which is a curious word for a zero-argument function that 'wraps' some code.

`(lesson 8)` Web programming basics

In which we begin to expose our Racket programs to the world via HTTP.

An HTTP server is responsible for accepting an HTTP request and returning an HTTP response, with some computation in between. That much you probably already knew.

Viewed more abstractly, an HTTP server can be considered a function that takes an HTTP request as argument and whose value is an HTTP response. (The underlying network connection between the client and server also figures in to this, but we will discount that here.)

A bit of terminology: a function from requests to responses is called, in the Racket world, a *servlet*.

Out of the box, Racket comes with two structure types, one for HTTP requests and another for HTTP responses. Using the conventional question mark, `request?` is a predicate that takes a Racket value and returns `#t` if it is an HTTP request. Similarly, `response?` is for HTTP responses. A servlet, then, if a function whose signature is

```
request? → response?
```

The Racket web server will handle a stream of bytes coming over the network and make sure that you, the programmer, get a `request?` value. Your task—should you choose to accept it—is to generate an HTTP response value.

Your job, then, is to define and combine servlets.

Servlets all around

A web application, from the server point of view, can be considered as a single servlet: a function that takes in every request whatsoever, and returns suitable responses. This suggests that servlets are ‘big’ functions. They carry a heavy load. As your web project grows, this one servlet gets bigger and bigger.

A more helpful perspective is to think of an HTTP server as being composed of servlets, each one devoted to handling a little part of your overall site. There’s the ‘main’ servlet, the one through which *every* request passes. But the main servlet can *dispatch* requests to other, smaller servlets. And these servlets, in turn, can themselves be composed of other servlets.

Think of servlets the way you think of the ‘main’ function in a program. The main function is, of course, a function. But I’ll bet that if your program has any interesting complexity to it at all, your main function will be divided into smaller functions. These smaller functions are written to help decompose our program, to make it more understandable and modular, and so on.

The same line of thinking applies to writing servlets.

Requests & responses

Requests (provided from `web-server/http/request-structs`) are structures with several components: the HTTP method, the headers, and so on. Likewise, responses are structs that contain the HTTP response code (200, 404, etc.).

One point to keep in mind is that these functions generally produce byte strings, or containers of bytes (e.g., headers). Byte strings, rather than plain UTF-8 strings, are used because that is in fact what is coming over the wire.

Running servlets

Once you’ve got a servlet ready to roll, you can put it to use using **`serve/servlet`**. Here’s an invocation that you’ll see many times, with some variations, throughout the book:

```
(serve/servlet
   let-er-rip
   #:port 6995
   #:servlet-regexp #rx"")
```

If this function is run, you'll have an HTTP server listening for requests on port 6995 and which will call `let-er-rip` and serialize the response (that is, the value of `let-er-rip`) for you.

(The `#:servlet-regexp` bit is to ensure that *every* request received gets passed on to `let-er-rip`. The regular expression is a pattern that allows you to bypass certain patterns in the URLs. Using the empty string has the effect that nothing is filtered out.)

Routes

Routes are a powerful concept in web application design. Think of routes families of URL resources, where the families are distinguished from one another by straightforward URL patterns. Here's how you can build a route-based web applications in Racket.

(**NB**: The example in this chapter uses Arabic and Japanese text. For technical reasons, these characters are not shown here in the PDF. They are, however, included in the source code. I trust that you'll spot the missing text and be able to work around it.)

dispatch-rules and friends

If you like, you can manually set up dispatching rules for your requests. Go ahead and write a giant `cond` and check the value of `request-uri` for incoming requests, directing the request to other servlets.

But there's already a nicer mechanism built-in and ready for use: `dispatch-rules`.

`dispatch-rules` and friends take as input a description (one might even call it a mini-DSL) of what your URL rules should look like. The rules take the form of 'routes'—parameterized families of URLs. You specify what the URL looks like (possibly including placeholders), what request method is used, and, finally, a servlet that is responsible for handling precisely these kinds of requests.

Let's make that concrete. Here's a very simple web application that has, essentially, only one resource that says 'hello' in a few different ways. It will be available under `/hello`. We can use `GET` to access this resource; any other HTTP method should lead to an error. Accessing any other resource leads to an error, too.

Greetings, random and not

The raw greetings data takes the form of a hash table, mapping **ISO 639-1 language codes** to a short greeting.

```
(define greetings/hash
  (hash "en" "Hello!"
        "de" "Hallo!"
        "es" "¡Hola!"
        "pt" "Ola!"
        "jp" ""))
```

The list of all languages is then:

```
(define languages (hash-keys greetings/hash))
```

together with

```
;; the number of available language
(define num-languages (length languages))
```

The dispatcher

Our dispatcher for our little greeter service is defined as follows:

```
(define-values (dispatcher url-generator)
  (dispatch-rules
   [("hello") hello]
   [("hello" (string-arg)) hello+lang]
   [("hello") #:method (regexp ".*") not-allowed]))
```

Putting aside for the moment the fact that we haven't defined `hello` or `hello+lang` yet (we'll come to them in the next section), we can still make sense of what this dispatcher is offering us. With the dispatcher defined this way, we're saying that we accept HTTP requests whose path is either

- exactly `/hello` (and requested via GET, which is the default here when `#:method` is not supplied), or
- matches `/hello/X` for some string X (that does not contain a slash). (Again, only GET requests are accepted here.)

If the request URI is `/hello`, then we either use the `hello` or `hello+lang` servlet (both of which we will soon discuss), provided the request method is GET; if not, we bail out and return a 405 ('Method Not Allowed') response, if any other request method is used.

That's the behavior we're heading towards. What do the servlets look like?

Greetings as servlets

Let’s start with the servlet that has no randomness aspect: it takes a language code as an argument and returns a greeting in that language. It is defined like this:

```
;; request? string? -> response?
(define (hello+lang req lang)
  (define greeting (hash-ref greetings/hash
                             lang
                             #f))
  (cond ((string? greeting)
         (respond/text #:body greeting))
        (else
         (not-found req))))
```

(`respond/text` is a utility defined in `respond.rkt`. It is just like `respond`, except that it creates HTTP responses whose MIME type is (UTF-8) plain text.)

The resource will contain a `Location` header exposing a (relative) URL where precisely this greeting—as opposed to a random one—can be found.

The servlet for generating a random language uses the non-random greeter servlet, and sets a `Location` header, indicating where one can access the resource that is being returned:

```
;; request? -> response?
(define (hello req)
  (define lang (random-language))
  (define greeting (hash-ref greetings/hash lang))
  (set-location (respond/text #:body greeting)
                (url-generator hello+lang lang)))
```

where `random-language` is defined as:

```
;; -> string?
(define (random-language)
  (list-ref greetings (random num-greetings)))
```

The `set-location` bit comes from `respond.rkt`. It takes an HTTP response and a location as arguments, and returns a new response containing a `Location` header whose value is the supplied location.

Let's look more closely at this function. It contains a bit more magic than meets the eye.

`dispatch-rules`, which we used when specifying the dispatcher, returns two values. We've spent some time talking about the first—the dispatcher that handles the mini-DSL for specifying routes—but none so far about the second.

What's great about `dispatch-rules` is that its second return value is a kind of 'inverse' of the first.

The first value—the dispatcher—is a function that takes as input an HTTP request, chops up the URL (and uses the HTTP method), and decides where the request should go. Roughly speaking, then, the dispatcher take a URL and returns a 'route'.

The second value is a URL generator that takes a route, together with its 'arguments', and generates a URL. That's how we generate the value for the `Location` header above.

Running the server

With this setup, submitting GET requests for `/hello` will yield a random greeting, but with a suitable `Location` header. To see this information, use the ‘developer console’ in your browser, or do things at the command line where you can see the complete HTTP response, like so:

```
$ http GET http://localhost:6995/hello
```

with the response

```
HTTP/1.1 200 OK
Content-Length: 15
Content-Type: text/plain;charset=UTF-8
Date: Wed, 29 Nov 2017 05:26:44 GMT
Last-Modified: Wed, 29 Nov 2017 05:26:44 GMT
Location: /hello/pt
Server: Racket

Ola!
```

Ola! And again:

```
$ http GET http://localhost:6995/hello
```

```
HTTP/1.1 200 OK
Content-Length: 6
Content-Type: text/plain;charset=UTF-8
Date: Wed, 29 Nov 2017 05:27:26 GMT
Last-Modified: Wed, 29 Nov 2017 05:27:26 GMT
Location: /hello/de
Server: Racket

Hallo!
```

And if we submit a GET request for /hello/de, we will get a non-random result identical (ignoring the Date and Last-Modified headers) to the previous output. Similarly for /hello/jp, /hello/en, and so on.

Takeaways

Racket web programming, as discussed here, revolves around the idea of *servlets*, which are just functions that take a request and produce a response. The request is always given to you as a structure, and certain parts of it are themselves structures (e.g., the list of cookies in the request).

(lesson 9) Macros

Macros allow you to step between your program text and the compiler, expanding the base Racket language in a way that ordinary functions cannot. It's the gateway to one of the most powerful ideas supported by Racket: making your own language.

Copy-and-replace on steroids

Macros are one of the coolest features in Racket. Perhaps it was even one of the ideas that brought Racket (or, for that matter, the Lisp family in general) to your attention. The idea of macros is common to virtually all members of the Lisp family. The essential idea is that the code you're writing in the Racket language looks a lot like Racket data.

Take a look at one of our earliest examples of Racket code:

```
(define welcome-msg "Welcome to Racket Weekend!")
```

Looking at that as a program, we see that we're defining something as a fixed string.

Look at it again, though, as data (or, if you wish, a syntax tree). It sort of looks like this:

```
(list 'define 'welcome-msg "Welcome to Racket Weekend")
```

The quote before `define` and `welcome-msg` is a way of writing symbols, which are another of the basic data types supported by Racket. They're similar to, though different from, strings, in the sense that they are *atomic*, as opposed to strings, which are, ultimately, a sequence of characters. Perhaps a better analogue are, say, integers.

Try evaluating that in a REPL. You’ll see:

```
'(define welcome-msg "Welcome to Racket Weekend")
```

There’s the initial quote before the open parenthesis, which indicates that we’re quoting a whole list. Putting aside for the moment what that means, it is clear that this list that we built looks an awful lot like the program snippet from earlier.

If you’re familiar with the idea of abstract syntax trees, or of the idea of parsing program text, this may not be terribly surprising. Either way, there’s something a bit magical going on here. If the program is, ultimately, a kind of data, what’s stopping us from writing functions that generate this data? What if we wanted to have the the definition—the whole program snippet—be a function of the greeting? We could do that as follows:

```
(define (define-greeting greet)
  (list 'define 'welcome-msg greet))
```

Then, if we are in a REPL, we’ll see

```
> (define-greeting "What's up?")
'(define welcome-msg "What's up?")
```

Again, that’s ‘just’ normal Racket data. It might look like a bona fide definition, but it isn’t.

What if we could take that data, which looks so much like real Racket code that it hurts, and get Racket to *treat* it as code? There’s a way of doing that: `eval`.

```
> (eval (define-greeting "What's up?"))
```

It doesn't do anything. But something has happened. To convince yourself of that, try:

```
> welcome-msg
"What's up?"
```

We started with normal data and then got Racket to treat the data as code.

That is essentially the main idea of macros. We will show, briefly, how to write functions that take in code—as-data, manipulate it, and produce more code. You can do with the code—that is, the data—whatever you want.

All about syntax

Macros are functions that take syntax and produce syntax. 'Syntax' is a special term in the Racket world. Eventually, at compile-time, a program gets read in, then broken down to basic elements, which are then executed.

Up to now we've written functions that take normal Racket values (e.g., a number, a list, a string, and so on), do some stuff, and then return a value. To a first approximation, macros are also functions, but with one special condition: they take a piece of syntax as an argument and return a piece of syntax as value. Thus, functions transform code into other code. This transformed code is itself then transformed, and so on. The goal of the compiler, then, is to take your high-level program and turn it into the most basic bits. Macros are a way of steering the compiler.

Run time vs. compile time

In Racket there's an important distinction that is crucial to understand when thinking about macros. It is not so easy to get used to at first, but if you think about it, the idea will eventually stick with you. The idea is the separation between two aspects of 'executing' your Racket program.

1. the compiler reads your program and expands all the macros, breaking the complex things into simpler things. The result of this computation—which can be quite substantial—is a so-called fully-expanded program. You don't get to see the results of this computation.
2. the simple things get executed.

You may be asking: 'our programs so far are pretty simple—what 'complex' pieces need to get broken up? This is the first time I've dug into macros, so I'm pretty sure that our programs so far haven't used them.'

Au contraire. Many of the programs that we've written so far involve macros. That is, their input isn't Racket *data*, but a Racket program (or, if you prefer, a program snippet), which gets turned into another Racket program (snippet).

Some examples so far:

1. `define/contract`. Ultimately, this gets boiled down, as you might expect, into a `define`, But some stuff gets wrapped around it, as you might imagine.
2. `match`: this is a rather complex business, but `match` ultimately is a fancy `cond`, which is itself based on even more basic things like `if`.
3. `class`: the basic primitive of Racket's object-oriented layer isn't actually a primitive!

The list goes on.

'So what things *are* primitive?' Here's one: `if`. Another: `set!`. There are a few dozen, but they are all *really* low-level. Everything else is built on top of these.

As you see, even things that are built-in to Racket aren't necessarily *primitive*. Indeed, in the Scheme tradition that Racket comes from, there are indeed not a whole lot of primitives. But instead of offering you *only* the primitives, Racket comes out-of-the-box with lots of non-primitive, very helpful utilities.

The fact that many of them get transformed away before your programs get executed is immaterial.

define-syntax and friends

```
(module+ test
  (let-test ([a 5]
             [b 3])
    (check-true (exact-integer? a))
    (check-false (even? b))))
```

To accomplish this, we need:

```
(module+ test
  (require rackunit)
  (define-syntax (let-test stx)
    (syntax-case stx ()
        [(_ formals tests ...)
         #'(let formals
               (test-begin
                 tests ...))])))
```

There are a few things to notice here:

- We are defining new syntax (i.e., a macro) in a submodule. The new syntax isn't in the main module. We could have defined it in the enclosing module.
- We use the ellipsis `...` notation.
- The *whole* of the syntax being matched is available to us. This means that we're given syntax whose datum looks like `(let-test X Y)`. But we don't care about the `let-test`; we care about the X and Y bits. That's why we use _ here.

Working in a sublanguage

There's another 'gotcha' when working with macros: the code you write there is working in a 'sublanguage' of full Racket called `racket/base`.

`racket/base` is just like `racket`—which we've used in all our code so far—except that some functions (and macros) are missing. The idea, roughly speaking, is to have a lean base language.

The 'gotcha' is that, at compile-time, Racket is working in `racket/base` and not `racket`. This means that some things you're used to may not work, for the simple reason that they are missing.

But you're not nailed down. You can import code at compile time. To import, for example, `racket/match` (which gives us `match`, which we used in the lesson on functional programming), do this:

```
#lang racket

(require (for-syntax racket/match))

(define-syntax (foo stx)
  ; using match somewhere here is OK
  ; thanks to the (require (for-syntax ...))
  ; line above
)
```

The `for-syntax` bit is a signal to Racket that we want something not just for the code to run, but to even *generate* the code which is to be compiled.

Weapon of choice

Macros are a powerful tool to have in your Racket toolbox. But before diving in to them, it should be noted that macros are usually not the usual way to tackle problems. If you're just getting started with Racket, chances are good that you need to get used to making 'normal' programs, where 'normal' means that the focus is on defining functions and other basic values, and not getting distracted by syntax.

That being said, as you grow as a Racket developer, you'll sooner or later probably find yourself wanting to get into macros. Some developers use macros a *lot*, some relatively little. This lesson focuses less on giving you a guided tour of macros and more of a taste of the very idea.

Takeaway

Macros are a powerful tool for extending your programming powers into the stratosphere. Even modest little macros can give you new levels of productivity. As one develops ore macros, one gets the feeling that you're not just writing little utilities. You're actually making a new language!

Achieving such gain, though, takes some time & experience. Macros aren't terribly easy to get into because they require us to think about our programs in a different way. We need to adopt the standpoint of the Racket compiler and think about programs as data.

`(lesson 10)` Reading & writing documentation

The documentation in Racket is extensive, comprehensive, and high-quality. It encourages one to write good documentation. There's a little language—Scribble—for doing just that.

Read the docs

The main place for up-to-date documentation in Racket is **"https://docs.racket-lang.org"**. From there, you can search around and see some of the highlights of the Racket universe. There's a search bar in the upper left corner that allows you to search.

The Racket documentation is (in case you haven't taken a look yet) *really* extensive.

Thankfully, you don't need an Internet connection to read the documentation. Indeed, an installation of the core Racket documentation, as well as documentation for any packages that you've installed, is on your hard drive. In your terminal, try

```
$ raco docs
```

to open your browser to something quite similar, though entirely local (on your machine). (The next lesson will be about packages.)

Using `raco`, you can even search through the local documentation—useful if you're offline or, for network politeness, don't want to keep searching through the online Racket documentation. Try this:

```
$ raco docs match
```

to search for ‘match’. You ought to see a couple of results returned. The first is almost certainly going to be the one we intend.

Writing docs

Racket comes with a neat language for writing documentation called Scribble.

The Scribble language is a markup language, not unlike HTML, but with more flexibility—and power—owing to its foundation in Racket. In fact, Scribble is an example of a Racket language.

If you have a Scribble file, you can generate an HTML version of it by using the `scribble` program, which comes with a standard install of Racket:

```
$ scribble foo.scrbl
```

to generate an HTML version of `foo.scrbl`.

Scribble basics

Scribble commands are expressed using @-notation. Let’s dive in an take a look at a simple Scribble file, for one of my own Racket packages, Argo, for validating JSON documents against JSON Schema. It looks like this:

```
#lang scribble/manual

@require[@for-label[racket/base
                    json
                    argo]]
@title[#:style "toc"]{Argo: JSON Schema Adventures}
@author[(author+email "Jesse Alama" "jesse@lisp.sh")]
@defmodule[argo]

Argo is a JSON Schema validator. Work with your JSON data
knowing
that it adheres to some sensible constraints. If you have to
work
with JSON, even if only occasionally, you may want to
consider
validating it (that is, checking that is satisfies the
constraints
specified by the schema).

@include-section["installation.scrbl"]
@include-section["running.scrbl"]
@include-section["interface.scrbl"]
@include-section["technical.scrbl"]
@include-section["limitations.scrbl"]
@include-section["references.scrbl"]
```

You see plenty of `@` signs here. Notice that this example uses `scribble/manual` as its language. There's a large suite of Scribble functions available; the ones being used here are:

- `require`
- `for-label`
- `title`

- `author`
- `defmodule`
- `include-section`

Intuitively, sectioning is achieved through `include-section`. One refers to the name of another Scribble file. Let's take a look at something a bit more meaty. Here's one of those sections, `interface.scrbl`, trimmed down for lack of space here:

```
#lang scribble/manual

@require[@for-label[racket/base json argo]]
@title{Library interface}

@defproc[
(parse-json?
[jsonish (or path? bytes? string? input-port? url?)])
(values jsexpr? boolean?)]
Parse something that might be JSON (a path to a file, a byte
string, a string,
an input port, or a URL).

Returns two values. The first a @racket[jsexpr?] representing
the parsed content
of @tt{jsonish}. The second indicates whether the result
could be parsed as JSON
at all. @racket[#f] means that the argument represents
invalid JSON; in that case,
it doesn't matter what the first return value is.

@defproc[
(json-equal?
[data-1 jsexpr?]
[data-2 jsexpr?])
boolean]
Returns @racket[#t] if @tt{data-1} and @tt{data-2} represent
equal JSON objects.

Equality of two JSON documents @tt{doc-1} and @tt{doc-2} is
defined as follows:
@tt{doc-1} and @tt{doc-2} have the same type (both are JSON
objects, both are
JSON strings, etc.) and adhere to the type-specific rules for
equality, which are:
```

```
@itemlist[
  @item{two strings are equal is they are equal as Unicode
strings
(codepoint-for-codepoint equal sequences)}
```

One sees here a common patter for documenting functions: with `defproc` one names the function and annotates the formal arguments with the expected types, and one also mentions the type of the return value.

Notice that there's a list of items here; `itemlist` and `item` are the tools to use.

Takeaways

The Racket documentation is very big, so it's good to arm yourself with a bit of knowledge of how to navigate around to find what you're looking for. Thankfully, the documentation itself offers some tools for getting around.

Typically, one writes documentation in the Scribble language. Scribble comes with its own syntax, based on `@`, and there are a number of built-in functions for helping you to structure your documentation. In the end, the @-syntax turns out to be just a disguised form of Racket's familiar S-expressions.

(lesson 11) Packages

Where we connect to the wider Racket world: installing packages from other developers, and submitting our own work to others.

Getting packages from others

We'll focus here on using the official Racket package catalog, which you can see at **https://pkg.racket-lang.org**.

Use:

```
$ raco pkg install foo
```

To install the package called 'foo'. Here's a great library for date and time processing:

```
$ raco pkg install gregor
```

As with other package systems, there's the notion of a package depending on another one. When you ask `raco` to install a package like this, you'll be prompted whether you want to install the packages dependencies. To automatically answer 'yes' to that question, use

```
$ raco pkg install –auto gregor
```

To update a package, use:

```
$ raco pkg update gregor
```

To update *all* packages you've installed, use `–all`:

```
$ raco pkg update —all
```

To remove a package, use the `remove` subcommand:

```
$ raco pkg remove gregor
```

As a rule, all these commands come with helpful documentation. Try entering a command followed by `—help`, like so:

```
$ raco pkg remove —help
```

Making your own package

To make your own package, go to the directory/folder that contains your code, and then do

```
$ raco pkg create
```

That's it! You've just added an entry to the package catalog. By default, the name of your package is the basename of the directory you just added. Thus, if the directory you were in when you did `raco pkg create` was `/Users/foo/code/racket/blasters`, then you can use your blasters code in another project by saying:

```
#lang racket

(require blasters)
```

'But what code is actually available to me after doing `(require blasters)`?' Great question.

Basic package concepts

package is a set of modules organized into collections

collection

single-collection package (most straightforward approach)

The info.rkt file

In your package, create an `info.rkt` file. This file will contain the metadata for your package.

Submitting a new package

To escape from the loneliness of your laptop and submit your package to the (Racket) world for others to use, the right way to do this is to use the official Racket package server. Go to **https://pkg.racket-lang.org** and sign up for an account there. Once you've signed in, go to **https://pkgd.racket-lang.org/pkgn/create** (on the homepage you'll see an 'add your own [package]' button, and that's where the link takes you).

The package server checks your submission to make sure that you've listed the correct dependencies for your package.

The package server goes through new submissions—including updates of existing packages—only once per day. It is a kind of 'build bot' that does quite a lot of work.

It will also add your documentation to the big directory of all Racket docs.

You won't be informed by email if your build fails, but it will still be present in the package database. That means that if someone downloads your package

Testing your own package

To test your own work, try

```
raco test -package my-cool-package
```

Documentation workflow

Documentation for packages is customarily stored in the `scribblings` directory of your package.

You've written your documentation. You even see it when doing `raco doc your-cool-function`. That's great!

How to compile *all* your documentation and make everything available? The answer is to update your package using `raco`, as follows:

```
$ raco pkg update
```

This will rebuild your package, including the documentation.

Takeaways

Packages are how you get Racket code from others and how you submit Racket code for others to use. One specifies a package using `info.rkt`, which contains certain package metadata. There's a conventional way to include documentation for a package, written in Scribble.

What's Next

If you're still with me, congratulations! You've made it to the end. We've touched on a number of points. Understandably, there's a whole boatload of material that we only hinted at, and even more that received no attention at all. To learn more about Racket, consider these resources:

- *Beautiful Racket*, by Matthew Butterick. Focuses on the heart of Racket: making your own languages. Also offers many practical examples of macros and other Racket essentials.
- *Server: Racket—Practical Web Development with the Racket HTTP Server*, by myself.
  Lesson 8 in this course is a shortened version of chapter 1 of *Server: Racket*. (If you'd like to get a copy, use the discount code `serverweekend` at checkout to get 15% off.)
- *Fear of Macros*, by Greg Hendershott. An excellent tutorial on macros.
- *The Little Schemer*, by Daniel P. Friedman and Matthias Felleisen.
  Not about Racket, per se, and not exactly new, but still relevant since Racket historically has a close similarity with Scheme. (Racket used to be called 'PLT Scheme'.) The Racket founder, Matthias Felleisen, is one of the co-authors.
  A successor, *The Seasoned Schemer*, contains many examples of problem-solving by recursion, which is an essential skill to have when hacking Racket.
- *Realm of Racket*, by Forrest Bice, Rose DeMaio, Spencer Florence, Mimi Lin, Scott Lindeman, Nicole Nussbaum, Eric Peterson, Ryan Plessner, David Van Horn, Matthias Felleisen, and Conrad Barski.
  It is the Racket version of *Land of Lisp*, which focuses mainly on Common Lisp, probably the largest member of the Lisp family, to which Racket belongs.