1 Welcome to Racket

Depending on how you look at it, **Racket** is

- a *programming language*—a dialect of Lisp and a descendant of Scheme;

> See Dialects of Racket and Scheme for more information on other dialects of Lisp and how they relate to Racket.

- a *family* of programming languages—variants of Racket, and more; or
- a set of *tools*—for using a family of programming languages.

Where there is no room for confusion, we use simply *Racket*.

Racket's main tools are

- **racket**, the core compiler, interpreter, and run-time system;
- **DrRacket**, the programming environment; and
- **raco**, a command-line tool for executing **Ra**cket **co**mmands that install packages, build libraries, and more.

Most likely, you'll want to explore the Racket language using DrRacket, especially at the beginning. If you prefer, you can also work with the command-line `racket` interpreter and your favorite text editor; see also Command-Line Tools and Your Editor of Choice. The rest of this guide presents the language mostly independent of your choice of editor.

If you're using DrRacket, you'll need to choose the proper language, because DrRacket accommodates many different variants of Racket, as well as other languages. Assuming that you've never used DrRacket before, start it up, type the line

```
#lang racket
```

in DrRacket's top text area, and then click the **Run** button that's above the text area. DrRacket then understands that you mean to work in the normal variant of Racket (as opposed to the smaller `racket/base` or many other possibilities).

> More Rackets describes some of the other possibilities.

If you've used DrRacket before with something other than a program that starts `#lang`, DrRacket will remember the last language that you used, instead of inferring the language from the `#lang` line. In that case, use the **Language|Choose Language...** menu item. In the dialog that appears, select the first item, which tells DrRacket to use the language that is declared in a source program via `#lang`. Put the `#lang` line above in the top text area, still.

1.1 Interacting with Racket

DrRacket's bottom text area and the `racket` command-line program (when started with no options) both act as a kind of calculator. You type a Racket expression, hit the Return key, and the answer is printed. In the terminology of Racket, this kind of calculator is called a *read-eval-print loop* or *REPL*.

A number by itself is an expression, and the answer is just the number:

```
> 5
5
```

A string is also an expression that evaluates to itself. A string is written with double quotes at the start and end of the string:

```
> "Hello, world!"
"Hello, world!"
```

Racket uses parentheses to wrap larger expressions—almost any kind of expression, other than simple constants. For example, a function call is written: open parenthesis, function name, argument expression, and closing parenthesis. The following expression calls the built-in function `substring` with the arguments `"the boy out of the country"`, `4`, and `7`:

```
> (substring "the boy out of the country" 4 7)
"boy"
```

1.2 Definitions and Interactions

You can define your own functions that work like `substring` by using the `define` form, like this:

```
(define (extract str)
  (substring str 4 7))

> (extract "the boy out of the country")
"boy"
> (extract "the country out of the boy")
"cou"
```

Although you can evaluate the `define` form in the REPL, definitions are normally a part of a program that you want to keep and use later. So, in DrRacket, you'd normally put the definition in the top text area—called the *definitions area*—along with the `#lang` prefix:

```
#lang racket

(define (extract str)
  (substring str 4 7))
```

If calling `(extract "the boy")` is part of the main action of your program, that would go in the definitions area, too. But if it was just an example expression that you were using to explore `extract`, then you'd more likely leave the definitions area as above, click **Run**, and then evaluate `(extract "the boy")` in the REPL.

When using command-line `racket` instead of DrRacket, you'd save the above text in a file using your favorite editor. If you save it as `"extract.rkt"`, then after starting `racket` in the same directory, you'd evaluate the following sequence:

> If you use `xrepl`, you can use `,enter extract.rkt`.

```
> (enter! "extract.rkt")
> (extract "the gal out of the city")
"gal"
```

The `enter!` form both loads the code and switches the evaluation context to the inside of the module, just like DrRacket's **Run** button.

1.3 Creating Executables

If your file (or definitions area in DrRacket) contains

```
#lang racket

(define (extract str)
  (substring str 4 7))

(extract "the cat out of the bag")
```

then it is a complete program that prints “cat” when run. You can run the program within DrRacket or using `enter!` in `racket`, but if the program is saved in ‹*src-filename*›, you can also run it from a command line with

```
racket ‹src-filename›
```

To package the program as an executable, you have a few options:

- In DrRacket, you can select the **Racket|Create Executable...** menu item.
- From a command-line prompt, run `raco exe` ‹*src-filename*›, where ‹*src-filename*› contains the program. See `raco exe`: Creating Stand-Alone Executables for more information.
- With Unix or Mac OS, you can turn the program file into an executable script by inserting the line

  > See Scripts for more information on script files.

  ```
  #! /usr/bin/env racket
  ```

  at the very beginning of the file. Also, change the file permissions to executable using `chmod +x` ‹*filename*› on the command line.

  The script works as long as `racket` is in the user’s executable search path. Alternately, use a full path to `racket` after `#!` (with a space between `#!` and the path), in which case the user’s executable search path does not matter.

1.4 A Note to Readers with Lisp/Scheme Experience

If you already know something about Scheme or Lisp, you might be tempted to put just

```
(define (extract str)
  (substring str 4 7))
```

into `"extract.rktl"` and run `racket` with

```
> (load "extract.rktl")
> (extract "the dog out")
"dog"
```

That will work, because `racket` is willing to imitate a traditional Lisp environment, but we strongly recommend against using `load` or writing programs outside of a module.

Writing definitions outside of a module leads to bad error messages, bad performance, and awkward scripting to combine and run programs. The problems are not specific to `racket`; they’re fundamental limitations of the traditional top-level environment, which Scheme and Lisp implementations have historically fought with ad hoc command-line flags, compiler directives, and build tools. The module system is designed to avoid these problems, so start with `#lang`, and you’ll be happier with Racket in the long run.