FREE eBook

LEARNING racket

Free unaffiliated eBook created from
Stack Overflow contributors.

#racket

Table of Contents

About 1

Chapter 1: Getting started with racket 2

Remarks 2

Versions 2

Examples 3

Hello, World! 3

Racket 4

Installation 4

Simple Recursive Function Definition 4

Find Racket sources in all subdirs 4

Installation or Setup 5

Chapter 2: Closures 6

Introduction 6

Remarks 6

Examples 6

Closure with static environment 6

Chapter 3: Comments 8

Remarks 8

Examples 8

Single line comments 8

Block comments 8

S-expression comments 8

Comments in at-exps 8

Chapter 4: Functions 10

Syntax 10

Examples 10

Simple Function Calls 10

Keyword arguments 10

The `apply` function 11

Function Definitions 11

Chapter 5: Higher Order Functions 13
Examples 13
Map 13
Fold 13
Filter 13
Compose 14
Curry 14
Chapter 6: Installation steps (Linux) 16
Examples 16
Installation or Setup 16
Installation steps for Linux: 16
Downloading 16
Starting the installer 16
Installing 17
Starting DrRacket 17
Running your first program 17
Chapter 7: Installation steps (macOS) 19
Examples 19
Installation or Setup 19
Installation steps for macOS: 19
Downloading 19
Starting the installer 19
Setting up command-line tools 19
Running your first program 19
Chapter 8: Installation steps (Windows) 21
Examples 21
Installation or Setup 21
Installation steps for Windows: 21
Downloading 21
Starting the installer 21
Setting up command-line tools 21

Running your first program 21
Chapter 9: Recursion 23
Examples 23
Using define 23
Using let-rec 23
Using a named let 23
Using rec 24
Using higher-order functions instead of recursion 24
Chapter 10: Scribble 25
Introduction 25
Examples 25
Paragraphs and sections 25
Documenting a binding provided by a package 25
Credits 27

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: racket

It is an unofficial and free racket ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official racket.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with racket

Remarks

This section provides an overview of what racket is, and why a developer might want to use it.

It should also mention any large subjects within racket, and link out to the related topics. Since the Documentation for racket is new, you may need to create initial versions of those related topics.

Versions

Version	Documentation	Release date
Nightly builds	Latest Documentation	2999-12-31
Version 6.9	Documentation	2017-04-01
Version 6.8	Documentation	2017-01-01
Version 6.7	Documentation	2016-10-01
Version 6.6	Documentation	2016-07-01
Version 6.5	Documentation	2016-04-01
Version 6.4	Documentation	2016-02-01
Version 6.3	Documentation	2015-11-01
Version 6.2.1	Documentation	2015-08-01
Version 6.2	Documentation	2015-06-01
Version 6.1.1	Documentation	2014-11-01
Version 6.1	Documentation	2014-08-01
Version 6.0.1	Documentation	2014-05-01
Version 6.0	Documentation	2014-02-01
Version 5.93	Documentation	2014-01-01
Version 5.92	Documentation	2014-01-01
Version 5.3.6	Documentation	2013-08-01
Version 5.3.5	Documentation	2013-06-01

Version	Documentation	Release date
Version 5.3.4	Documentation	2013-05-01
Version 5.3.3	Documentation	2013-02-01
Version 5.3.2	Documentation	2013-02-01
Version 5.3.1	Documentation	2012-11-01
Version 5.3	Documentation	2012-08-01
Version 5.2.1	Documentation	2012-03-01
Version 5.2	Documentation	2011-11-01
Version 5.1.3	Documentation	2011-08-01
Version 5.1.2	Documentation	2011-08-01
Version 5.1.1	Documentation	2011-04-01
Version 5.1	Documentation	2011-02-01
Version 5.0.2	Documentation	2010-11-01
Version 5.0.1	Documentation	2010-08-01
Version 5.0	Documentation	2010-06-01

Examples

Hello, World!

The following example declares a piece of code to be written in Racket, and then prints the string `Hello, world`.

```
#lang racket
"Hello, world!"
```

Racket code can either be run directly from the command line or on the DrRacket IDE. Typing `racket` on the command line will start a REPL, and typing `racket` followed by a file name will evaluate the contents of the file. For example, suppose the file `hello.rkt` contains the above code. Here is an example of running Racket on the command line.

```
$ racket
Welcome to Racket v6.5.
> "Hello, world!"
"Hello, world!"
> (exit)
```

```
$ racket hello.rkt
"Hello, world!"
```

Racket

Racket is a full-spectrum programming language. It goes beyond Lisp and Scheme with dialects that support objects, types, laziness, and more. Racket enables programmers to link components written in different dialects, and it empowers programmers to create new, project-specific dialects. Racket's libraries support applications from web servers and databases to GUIs and charts.

The offical, comprehensive and very well-written documentation can be found at [http://docs.racket-lang.org/][1]. On this site (Stack Overflow Documentation) you can find user-contributed examples.

Installation

Go to http://racket-lang.org and click the download button.

Simple Recursive Function Definition

In Racket, we use recursion very frequently. Here is an example of a function that sums all of the numbers from zero to the parameter, `n`.

```
(define (sum n)
    (if (zero? n)
        0
        (+ n (sum (sub1 n)))))
```

Note that there are many helpful convenience based functions used here, such as `zero?` and `sub1`. Each respectively does just what you might expect: `zero?` returns a boolean which says whether the given number was equal to zero, and `sub1` subtracts one from its argument.

Find Racket sources in all subdirs

```
#lang racket
(for ([path (in-directory)]
  #:when (regexp-match? #rx"[.]rkt$" path))
  (printf "source file: ~a\n" path))
```

The `#lang` line specifies the programming language of this file. `#lang racket` we are using the baseline, battery-included Racket programming language. Other languages ranen from Racket flavors such as Type Racket (`#lang typed/racket`) or the documentation language Scribble (`#lang scribble`), to small convenience languages such as the language for defining packages (`#lang info`).

The `in-directory` function constructs a sequence that walks a directory tree (starting with the current directory, by default) and generates paths in the tree. The `for` form binds `path` to each path in the sequence, and `regexp-match?` applies a pattern to the path.

To run the example, install Racket, start DrRacket, paste the example program into the top area in DrRacket, and click the Run button. Alternatively, save the program to a file and run `racket` from the command line on the file.

Installation or Setup

The installation is very simple. If you are used to this kind of thing, just go to https://download.racket-lang.org. If you prefer, there are more detailed step-by-step installation instructions for the following systems:

- Installation steps (Windows)
- Installation steps (Linux)
- Installation steps (macOS)

Read Getting started with racket online: https://riptutorial.com/racket/topic/1134/getting-started-with-racket

Chapter 2: Closures

Introduction

From the StackOverflow closure tag:

A closure is a first-class function that refers to (closes over) variables from the scope in which it was defined. If the closure still exists after its defining scope ends, the variables it closes over will continue to exist as well.

Remarks

It is sometimes useful to consider closures and objects as similar.

The venerable master Qc Na was walking with his student, Anton. Hoping to prompt the master into a discussion, Anton said "Master, I have heard that objects are a very good thing - is this true?" Qc Na looked pityingly at his student and replied, "Foolish pupil - objects are merely a poor man's closures." Chastised, Anton took his leave from his master and returned to his cell, intent on studying closures. He carefully read the entire "Lambda: The Ultimate..." series of papers and its cousins, and implemented a small Scheme interpreter with a closure-based object system. He learned much, and looked forward to informing his master of his progress.

On his next walk with Qc Na, Anton attempted to impress his master by saying "Master, I have diligently studied the matter, and now understand that objects are truly a poor man's closures." Qc Na responded by hitting Anton with his stick, saying "When will you learn? Closures are a poor man's object." At that moment, Anton became enlightened.

Source: http://c2.com/cgi/wiki?ClosuresAndObjectsAreEquivalent

Examples

Closure with static environment

A closure is a procedure that holds internal state:

Define a procedure that returns a closure

The procedure `make-an-adder` takes one argument `x` and returns a function that closes over the value. Or to put it another way, `x` is within the lexical scope of the returned function.

```
#lang racket
(define (make-an-adder x)
  (lambda (y)
    (+ y x)))
```

Usage

Calling the procedure `make-an-adder` returns a procedure that is a closure.

```
Welcome to DrRacket, version 6.6 [3m].
Language: racket, with debugging; memory limit: 128 MB.
> (define 3adder (make-an-adder 3))
> (3adder 4)
7
> (define 8adder (make-an-adder 8))
> (8adder 4)
12
```

Read Closures online: https://riptutorial.com/racket/topic/7176/closures

Chapter 3: Comments

Remarks

The most common comment types are line and s-expression comment (using `;` and `#;` respectively).

It is common to use from 1 to 3 semi colons depending on the type of comment made. Refer to ???

Examples

Single line comments

```
; We make single line comments by writing out text after a semicolon
```

Block comments

```
#| We make
block comments
like this |#
```

S-expression comments

```
#;(define (commented-out-function x)
    (print (string-append "This entire "
            "s-expression is commented out!")))
```

Comments in at-exps

When a module is using at expressions, such as:

```
#lang at-exp racket/base
```

or

```
#lang scribble/manual
```

You have access to the following types of comments:

```
@;{Block text that goes
   until the closing
   brace.}
```

As well as:

```
@; Single line text.
```

Note that if you are using a language that only uses at-exps (such as most scribble languages), you will need ot use one of these types of comments.

Read Comments online: https://riptutorial.com/racket/topic/3147/comments

Chapter 4: Functions

Syntax

- (define (name arguments ...) body)
- (function arguments ...)

Examples

Simple Function Calls

You can call a function in Racket by wrapping it in parentheses with the arguments after it. This looks like `(function argument ...)`.

```
> (define (f x) x)
> (f 1)
1
> (f "salmon")
"salmon"
> (define (g x y) (string-append x y))
> (g "large" "salmon")
"largesalmon"
> (g "large " "salmon")
"large salmon"
```

Operations like `+` and `*` are functions as well, and they use the same syntax as calling `f` or `g`.

```
> (+ 1 2)
3
> (* 3 4)
12
> (+ (* 3 3) (* 4 4))
25
```

For more information and examples, see Function Calls in the Racket Guide.

Keyword arguments

Racket functions can also have *keyword arguments*, which are specified with a keyword followed by the argument expression. A keyword begins with the characters `#:`, so a keyword argument looks like `#:keyword arg-expr`. Within a function call this looks like `(function #:keyword arg-expr)`.

```
> (define (hello #:name n)
    (string-append "Hello " n))
> (hello #:name "John")
"Hello John"
> (hello #:name "Sarah")
"Hello Sarah"
```

```
> (define (kinetic-energy #:mass m #:velocity v)
    (* 1/2 m (sqr v)))
> (kinetic-energy #:mass 2 #:velocity 1)
1
> (kinetic-energy #:mass 6 #:velocity 2)
12
```

For more information and examples, see Keyword Arguments in the Racket Guide.

The `apply` function

If you have a list, and you want to use the elements of that list as the arguments to a function, what you want is `apply`:

```
> (apply string-append (list "hello" " " "and hi" " " "are both words"))
"hello and hi are both words"
> (apply + (list 1 2 3 4))
10
> (apply append (list (list "a" "b" "c") (list 1 2 3) (list "do" "re" "mi")))
(list "a" "b" "c" 1 2 3 "do" "re" "me")
```

`apply` takes two arguments. The first argument is the function to apply, and the second argument is the list containing the arguments.

An `apply` call like

```
(apply + (list 1 2 3 4))
```

Is equivalent to

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

The major advantage of `apply` is that it works on arbitrary computed lists, including appended lists and lists that come from function arguments.

```
> (apply + (append (list 1 2 3 4) (list 2 3 4)))
19
> (define (sum lst)
    (apply + lst))
> (sum (list 1 2 3 4))
10
> (sum (append (list 1 2 3 4) (list 2 3 4)))
19
```

For more information and examples, see The `apply` function in the Racket Guide.

Function Definitions

Functions in Racket can be created with the `lambda` form. The form takes a list of arguments and a body.

```
(lambda (x y) (* x y))
```

In the example above, the function takes in two arguments and returns the result of multiplying them.

```
> ((lambda (x y) (* x y)) 4 4)
16
> ((lambda (x y) (* x y)) 3 2)
6
```

It's tedious to re-write the function and its body every time we want to multiply two numbers, so let's give it a name. To give it a name, use the `define` form. This will bind functions to a name.

```
(define multiply (lambda (x y) (* x y)))
```

Now we can refer to our function by calling `multiply`

```
> (multiply 5 2)
10
```

Since it is very common to bind procedures to names, Racket provides a shorthand to define functions using the define form.

```
(define (multiply x y) (* x y))
```

For more information and examples, see Functions: lambda in the Racket Guide.

Read Functions online: https://riptutorial.com/racket/topic/1935/functions

Chapter 5: Higher Order Functions

Examples

Map

Map applies a function to every element of a list:

```
map: (a -> b) (listof a) -> (listof b)

> (map (lambda (x) (* x 2)) (list 1 2 3 4 5)
(list 2 4 6 8 10)

> (map sqrt (list 1 4 9))
(list 1 2 3)

> (map (lambda (x) (if (even? x) "even" "odd")) (list 1 2 3))
(list "odd" "even" "odd")
```

Fold

Fold Right successively applies a two-argument function to every element in a list from left to right starting with a base value:

```
foldr: (a b -> b) b (listof a) -> b

> (foldr + 0 (list 1 2 3 4))
10

> (foldr string-append "" (list "h" "e" "l" "l" "o"))
"hello"

> (foldr cons empty (list 1 2 3 4))
(list 1 2 3 4)
```

Fold Left performs the same action in the opposite direction:

```
foldl: (a b -> b) b (listof a) -> b

> (foldl + 0 (list 1 2 3 4)
10

> (foldl string-append "" (list "h" "e" "l" "l" "o"))
"olleh"

> (foldl cons empty (list 1 2 3 4))
(list 4 3 2 1)
```

Filter

`filter` returns a list of each item in the given list for which the given predicate returns a non-`#f`

value.

```
;; Get only even numbers in a list
> (filter even? '(1 2 3 4))
'(2 4)

;; Get all square numbers from 1 to 100
> (filter (lambda (n) (integer? (sqrt n))) (range 1 100))
'(1 4 9 16 25 36 49 64 81)
```

Compose

Lets you compose several functions $f_0\ f_1\ \ldots\ f_n$. It returns a function that will successively apply f_n to its arguments, then f_{n-1} to the result of f_n and so on. Function are applied from right to left, like for mathematical function composition: $(f \circ g \circ h)(x) = f(g(h(x)))$.

```
> ((compose sqrt +) 16 9)
5
> ((compose - sqrt) 16)
-4
```

The arity of each function should include the the number of returned values of the function immediately to its right. The rightmost function determines the arity of the whole composition. The compose1 function imposes that the functions return 1 value and expect 1 argument. However, compose1 does not restrict the input arity of the last function, nor the output arity of the first function.

```
[n input]--> first-function -->[1 output]--> ... last function -->[m output].

((compose + values) 1 2 3 4)
10
> ((compose1 + values) 1 2 3 4)
XX result arity mismatch;
 expected number of values not received
  expected: 1
  received: 4
  values...:
```

Curry

Returns a partially applied function.

```
> ((curry + 10) 20)
30
```

`curryr` can be used when the arguments need to be inserted at the end. In other words, `(curryr list 1 2)` will produce a function expecting some `new-arguments ...`. When called, that new function will in turn call `(list new-arguments ... 1 2)`.

```
> (((curryr list) 1 2) 3 4)
'(3 4 1 2)
```

```
> ((curryr list 1 2) 3 4)
'(3 4 1 2)
> ((curryr - 30) 40)
10
> (((curryr -) 30 40))
10
```

Read Higher Order Functions online: https://riptutorial.com/racket/topic/4433/higher-order-functions

Chapter 6: Installation steps (Linux)

Examples

Installation or Setup

Visit https://download.racket-lang.org and choose between the two available distributions:

- `Racket` is the main distribution, it comes with several additional packages like math/number-theory as well as the DrRacket IDE.
- `Minimal Racket` is far smaller and comes only with the needed packages.

Installation steps for Linux:

The installation is very simple. If you are used to this kind of thing, just follow these four steps. A more detailed step-by-step walkthrough is detailed afterwards, if you prefer.

1. download it from https://download.racket-lang.org
2. `chmod +x racket-6.6-x86_64-linux.sh`
3. `./racket-6.6-x86_64-linux.sh`
4. Answer the questions, and possibly update your `$PATH`.

For a more detailed step-by-step guide, see below.

Downloading

1. Go to https://download.racket-lang.org .
2. Select `Platform: Linux i386` if you have a 32-bit system, or `Platform: Linux x86_64`.
3. Click the download button labeled `racket-6.9-x86_64-linux.sh (113M)` (the label may be slightly different depending on the version).

Starting the installer

4. Open a terminal.
5. If you downloaded the file to the `/home/YOUR_USER_NAME/Downloads`, type the following command:

```
cd /home/YOUR_USER_NAME/Downloads
```

Be sure to replace `YOUR_USER_NAME` by your actual user name and `/Downloads` by the actual path to the folder to which you downloaded Racket.

6. Type `chmod +x racket-6.6-x86_64-linux.sh` (change the version number and the `x86_64` to match the file you downloaded).
7. If you want to install Racket system-wide, type `sudo ./racket-6.6-x86_64-linux.sh` (change

the version number and the `x86_64` to match the file you downloaded).

Otherwise, if you are not an administrator on the computer, simply type `./racket-6.6-x86_64-linux.sh` to install it in your own home directory (change the version number and the `x86_64` to match the file you downloaded).

Installing

The installer will ask the following questions:

8. `Do you want a Unix-style distribution?`

Answer `no` (the default).

9. `Where do you want to install the "racket-6.6.0.4" directory tree?`

Select `/usr/racket` (type `1 Enter ⏎`) or `/usr/local/racket` (type `2 Enter ⏎`) if you are installing Racket system-wide. Otherwise, to install it in your own home directory (e.g. if you are not an administrator), select `~/racket (/home/YOUR_USER_NAME/racket)` (type `3 Enter ⏎`).

10. `If you want to install new system links within the "bin", "man" and "share/applications" subdirectories…`

If you are doing a system-wide installation it is a good idea to type `/usr/local` or `/usr` here (to know which, check which one is present in your `PATH`, by typing `echo $PATH` in another terminal window). If you are installing it in your own home directory, leave the answer empty and just press `Enter ⏎`.

Starting DrRacket

Depending on your answer to steps 9 and 10, you need to type one of the following commands in a terminal to start DrRacket:

- `drracket` (if step 10 was successful)
- `/usr/racket/bin/drracket`
- `/usr/local/racket/bin/drracket`
- `/home/YOUR_USER_NAME/racket/bin/drracket` (replace `YOUR_USER_NAME` by your actual username, or simply type `~/racket/bin/drracket`)

To avoid typing such a long command each time, you can add the following command to the file `~/.bashrc`, where `/path/to/the/containing/folder/` should be one of `/usr/racket/bin/`, `/usr/local/racket/bin/` or `/home/YOUR_USER_NAME/racket/bin/`:

```
export PATH="/path/to/the/containing/folder/:$PATH"
```

Running your first program

To run a program, open DrRacket as explained above, enter the program starting with `#lang racket`, and click the `Run` button near the top-right corner. Here is a first example program:

```
#lang racket
(displayln "Hello Racket!")
```

Read Installation steps (Linux) online: https://riptutorial.com/racket/topic/9870/installation-steps--linux-

Chapter 7: Installation steps (macOS)

Examples

Installation or Setup

Visit https://download.racket-lang.org and choose between the two available distributions:

- `Racket` is the main distribution, it comes with several additional packages like math/number-theory as well as the DrRacket IDE.
- `Minimal Racket` is far smaller and comes only with the needed packages.

Installation steps for macOS:

The installation is very simple. If you are used to this kind of thing, just go to https://download.racket-lang.org, then download and install the `.dmg` file. A more detailed step-by-step walkthrough is detailed afterwards, if you prefer.

Downloading

1. Go to https://download.racket-lang.org .
2. Select `Platform: Mac OS (Intel 32-bit)` if you have a 32-bit system, or `Platform: Mac OS (Intel 64-bit)` if you have a 64-bit system.
3. Click the download button labeled `racket-6.9-x86_64-macosx.dmg (106M)` (the label may be slightly different depending on the version).

Starting the installer

4. `FIXME: If you have macOS, please fill in this section`

Setting up command-line tools

On **Mac OS X**, you can visit the Help menu of DrRacket and use "Configure Command Line for Racket..." to set up racket tools for command line use. On **Windows** you will need to add the Racket installation folder to your `PATH` variable.

Running your first program

To run a program, open DrRacket, enter the program starting with `#lang racket`, and click the `Run` button near the top-right corner. Here is a first example program:

```
#lang racket
```

```
(displayln "Hello Racket!")
```

Read Installation steps (macOS) online: https://riptutorial.com/racket/topic/9872/installation-steps--macos-

Chapter 8: Installation steps (Windows)

Examples

Installation or Setup

Visit https://download.racket-lang.org and choose between the two available distributions:

- `Racket` is the main distribution, it comes with several additional packages like math/number-theory as well as the DrRacket IDE.
- `Minimal Racket` is far smaller and comes only with the needed packages.

To run a program, open DrRacket, enter the program starting with `#lang racket`, and click the `Run` button near the top-right corner.

Installation steps for Windows:

The installation is very simple. If you are used to this kind of thing, just go to https://download.racket-lang.org, then download and run the installer. A more detailed step-by-step walkthrough is detailed afterwards, if you prefer.

Downloading

1. Go to https://download.racket-lang.org .
2. Select `Platform: Windows (x86, 32-bit)` if you have a 32-bit system, or `Platform: Windows (x64, 64-bit)` if you have a 64-bit system. If in doubt, choose the 32-bit version.
3. Click the download button labeled `racket-6.9-i386-win32.exe (73M)` (the label may be slightly different depending on the version).

Starting the installer

4. Open the directory where the file was downloaded, and double-click on the `racket-….exe` file.
5. Follow the installer's instructions.

Setting up command-line tools

To set up the command-line tools, open DrRacket, click the Help menu, and click "Configure Command Line for Racket." This will install the `racket` and `raco` commands. (On Windows, the `racket` command is `Racket.exe`).

Running your first program

To run a program, open DrRacket, enter the program starting with `#lang racket`, and click the `Run` button near the top-right corner. Here is a first example program:

```
#lang racket
(displayln "Hello Racket!")
```

Read Installation steps (Windows) online: https://riptutorial.com/racket/topic/9871/installation-steps--windows-

Chapter 9: Recursion

Examples

Using define

```
#lang racket
(define (sum-of-list l)
  (if (null? l)
      0
      (+ (car l)
         (sum-of-list (cdr l)))))
(sum-of-list '(1 2 3 4 5)) ;; => 15
```

Using let-rec

```
#lang racket
(letrec ([sum-of-list (λ (l)
                        (if (null? l)
                            0
                            (+ (car l) (sum-of-list (cdr l)))))])
  (sum-of-list '(1 2 3 4 5)))
;; => 15
```

It is possible to write mutually recursive functions with `letrec`:

```
#lang racket
(letrec ([even? (λ (n) (if (= n 0) #t (odd?  (sub1 n))))]
         [odd?  (λ (n) (if (= n 0) #f (even? (sub1 n))))])
  (list (even? 3)
        (odd? 5)))
;; => '(#f #t)
```

Using a named let

A normal `let` form binds each value to its corresponding identifier, before executing the body. With a "named `let`", the body can then recursively be re-executed, passing a new value for each identifier.

```
#lang racket
(let sum-of-list ([l '(1 2 3)])
  (if (null? l)
      0
      (+ (car l) (sum-of-list (cdr l)))))
;; => 15
```

It is common to use `rec` as the name for the let, which gives:

```
#lang racket
```

```
(let rec ([l '(1 2 3 4 5)])
  (if (null? l)
      0
      (+ (car l) (rec (cdr l)))))
;; => 15
```

Using rec

```
#lang racket
(require mzlib/etc)
((rec sum-of-list
   (λ (l)
     (if (null? l)
         0
         (+ (car l) (sum-of-list (cdr l))))))
 '(1 2 3 4 5))
;; => 15

;; Outside of the rec form, sum-of-list gives an error:
;; sum-of-list: undefined;
;;  cannot reference an identifier before its definition
```

This is similar to `define`, but the `sum-of-list` identifier is not visible outside of the `rec` form.

To avoid using an explicit `λ`, it is possible to replace `sum-of-list` with `(sum-of-list args ...)`:

```
#lang racket
(require mzlib/etc)
((rec (sum-of-list l)
   (if (null? l)
       0
       (+ (car l) (sum-of-list (cdr l)))))
 '(1 2 3 4 5))
;; => 15
```

Using higher-order functions instead of recursion

It is common practice to use higher order functions instead of recursion, if there is a higher order function which expresses the right recursion pattern. In our case, `sum-of-numbers` can be defined using `foldl`:

```
#lang racket
(define (sum-of-numbers l)
  (foldl + 0 l))
(sum-of-numbers '(1 2 3 4 5)) ;; => 15
```

It is possible to call `foldl` directly on the list:

```
#lang racket
(foldl + 0 '(1 2 3 4 5)) ;; => 15
```

Read Recursion online: https://riptutorial.com/racket/topic/6465/recursion

Chapter 10: Scribble

Introduction

Scribble is the tool used to create Racket's documentation, and you can document your own packages with it too! When published, their documentation will appear at https://docs.racket-lang.org/, alongside the main Racket documentation.

Scribble is implemented as a language for the Racket platform. Scribble documents will therefore usually start with `#lang scribble/manual`

Examples

Paragraphs and sections

```
#lang scribble/manual

@section{Introduction}

First paragraph. Some text, some text, some text,
some text, some text, some text.

@section{More stuff}

@subsection{This is a subsection}

Second paragraph. More text, more text, more text,
more text, more text, more text.
```

Documenting a binding provided by a package

```
#lang scribble/manual

@; Make sure that code highlighting recognises identifiers from my-package:
@require[@for-label[my-package]]

@; Indicate which module is exporting the identifiers documented here.
@defmodule[my-package]

@defproc[(my-procedure [arg1 number?] [arg2 string?]) symbol?]{
  The @racket[my-procedure] function repeats the @racket[arg2] string
  @racket[arg1] times, and transforms the result into a symbol.

  @history[#:added "1.0"
           #:changed "1.1" @elem{Improved performance,
                                 from @tt{O(n²)} to @tt{O(n)}}]
}
```

As a rule of thumb, a module (something that could appear on the right of a `(require foo/bar)`, i.e. `foo/bar`) should be documented by a single `.scribble` file. A `.scribble` file can document several

modules, as long as each one is documented in a separate `@section`.

Read Scribble online: https://riptutorial.com/racket/topic/9881/scribble

Credits

S. No	Chapters	Contributors
1	Getting started with racket	Alex Knauth, belph, bitrauser, Community, eyqs, Georges Dupéron, Guillaume Marceau, John Gallagher, Kronos, Leif Andersen, mnoronha, soegaard
2	Closures	ben rudgers
3	Comments	eyqs, Leif Andersen, pvdsp, soegaard
4	Functions	Alex Knauth, eyqs, Jason Yeo
5	Higher Order Functions	4444, Anjali Pal, Brendan, Georges Dupéron, Majora320, mathk
6	Installation steps (Linux)	Alex Knauth, bitrauser, eyqs, Georges Dupéron
7	Installation steps (macOS)	Georges Dupéron
8	Installation steps (Windows)	Georges Dupéron
9	Recursion	Georges Dupéron
10	Scribble	Georges Dupéron