How to Program Racket: a Style Guide

Version 8.7

Matthias Felleisen, Matthew Flatt, Robby Findler, Jay McCarthy

November 11, 2022

Since 1995 the number of “repository contributors” has grown from a small handful to three dozen and more. This growth implies a lot of learning and the introduction of inconsistencies of programming styles. This document is an attempt leverage the former and to start reducing the latter. Doing so will help us, the developers, and our users, who use the open source code in our repository as an implicit guide to Racket programming.

To help manage the growth of our code and showcase good Racket style, we need guidelines that shape the contributions to the code base. These guidelines should achieve some level of consistency across the different portions of the code base so that everyone who opens files can easily find their way around.

This document spells out the guidelines and best practices. They cover a range of topics, from basic work (commit) habits to small syntactic ideas like indentation and naming.

Many pieces of the code base don’t live up to the guidelines yet. Here is how we get started. When you start a new file, stick to the guidelines. If you need to edit a file, you will need to spend some time understanding its workings. If doing so takes quite a while due to inconsistencies with the guidelines, please take the time to fix (portions of) the file. After all, if the inconsistencies throw you off for that much time, others are likely to have the same problems. If you help fix it, you reduce future maintenance time. Whoever touches the file next will be grateful to you. *Do* run the test suites, and do *not* change the behavior of the file.

Also, look over the commit messages. If you see problems with the code deltas, let the contributor know. If you see a bug fix without docs and tests, let the contributor know. Code should be viewed by more than one person because a second person is likely to catch logical mistakes, performance problems, and unintended effects.

Request This document isn’t complete and it isn’t perfect. Consider it a call for improvements and suggestions. If you have ideas, contact the first author via email. If your request gets ignored, appeal to all four authors.

Note The recommendations in this style guide may not jibe with what you grew up with. (They conflict with some of the ideas that the primary author had about style.) But if you do write code that ends up in the Racket code base, please follow the recommendations here. If/when someone else works on your code, this person may “fix” your code if it isn’t in compliance with the style guide.

1 Basic Facts of Life

Favor readers over writers. — Yaron Minsky, JaneStreet, 2010 at NEU/CCS

This ordering is occasionally wrong. For example, we could avoid IEEE floating point numbers nearly all of the time. To make this precise, the Racket `sqrt` function could return a rational number close to the IEEE float result. We don't do such silly things, however, because we have decided to value speed over precision in this context.

Strive to write code that is correct; maintainable; and fast. The ordering of these adjectives is critical: correct is more important than maintainable; maintainable is more important than fast; and fast is important to include, because nobody wants to live with slow programs.

This section explains these three points as far as the Racket code base is concerned. The rest of this guide is to spell out suggestions that should help you make correct, maintainable, and fast contributions to the Racket code base.

1.1 Correctness and Testing

I have bug reports, therefore I exist. – Matthias, watching Matthew, Robby, Shriram and others create the original code base

It is the way we choose to fight our bugs that determines our character, not their presence or absence. – Robby, in response

PLT aims to release good code and to eliminate mistakes as quickly as possible. All software has mistakes; complete correctness is a perfectionist goal. If mistakes are unknown, the software isn't being used. The goal is, however, to ensure some basic level of correctness before a feature is released and to ensure that the same mistake isn't introduced again.

We ensure this basic level of correctness with large test suites. Our test suites contain tests at all levels. In addition to unit tests, you will find test suites that use a "random testing" strategy and tools, others use fuzz testing, yet others are end-to-end "systems level" tests, and DrRacket comes with an automatic GUI player that explores its functionality.

For details on testing in the context of the Racket code base, see §2 "Testing".

1.2 Maintenance

If we wish to create maintainable code, we must ensure that our code is comprehensible. Code is comprehensible when you can understand its external purpose; when you can guess from its external purpose at its organization; when the organization and the code live up to consistent criteria of style; and when the occasional complex part comes with internal documentation.

Released code must have documentation. Conversely a change to the external behavior of code must induce a simultaneous change to its documentation. Here "simultaneous" means that the two changes are in the same 'push' to the code base, not necessarily in the same 'commit'. Also see §8 "Retiquette: Branch and Commit" for more on Git actions.

For style rules on documenting code, refer to the style guide in the Scribble manual. Ideally documentation comes in two parts, possibly located in the same document: a "Guide" section, which explains the purpose and suggests use cases, and a traditional "Reference" section, which presents the minutiae. The documentation for HtDP/2e teachpacks is an example where the two parts are collocated. Also consider adding examples for each function and construct in your "Reference" section. Finally, ensure you have all the correct `for-label` `require`s and make use of other useful cross-references.

Having said that, the production of a system like Racket occasionally requires experimentation. Once we understand these new pieces of functionality, though, it is imperative to discard the "failure branches" of an experiment and to turn the successful part into a maintainable package. You may even consider converting your code to Typed Racket eventually.

Without adherence to basic elements of style, code comprehension becomes impossible. The rest of this document is mostly about these elements of style, including some suggestions on minimal internal documentation.

1.3 Speed

Making code fast is an endless task. Making code *reasonably fast* is the goal.

As with correctness, performance demands some "testing." At a minimum, exercise your code on some reasonably realistic inputs and some larger ones. Add a file to the test suite that runs large inputs regularly. For example, a regular test suite for a Universe display deals with a 50 x 50 display window; one of its stress tests checks whether Universe event handlers and drawing routines can cope with laptop size displays or even a 30in display. Or, if you were to write a library for a queue data structure, a regular test suite ensures that it deals correctly with enqueue and dequeue for small queues, including empty ones; a stress test suite for the same library would run the queue operations on a variety of queue sizes, including very large queues of say tens of thousands elements.

Stress tests don't normally have an expected output, so they never pass. The practice of writing stress tests exposes implementation flaws or provides comparative data to be used when choosing between two APIs. Just writing them and keeping them around reminds us that things can go bad and we can detect when performance degrades through some other door. Most importantly, a stress test may reveal that your code isn't implementing an algorithm with the expected $O(.)$ running time. Finding out that much alone is useful. If you can't think of an improvement, just document the weakness in the external library and move on.

And as you read on, keep in mind that we are not perfectionists. We produce reasonable software.

2 Testing

2.1 Test Suites

Most of our collections come with test suites. These tests suites tend to live in `collects/tests/` in the PLT repository, though due to historical reasons, a few collections come with their own local test suites. If you add a new collection, create a new test suite in the `tests` collection.

Run the test suites before you commit. To facilitate testing, we urge you to add a `TESTME.txt` file to your collections. Ideally, you may also wish to have a file in this directory that runs the basic tests. See the 2htdp, which is one of the collections with its own testing style. The file should describe where the tests are located, how to run these tests, and what to look for in terms of successes and failures. These files are necessary because different collections have different needs for testing, and testing evolved in many different ways in our history.

After you commit, watch for and read(!) DrDr's emails. Do *not* ignore them. If you have tests that are known to fail and fixing this requires a lot of work, consider splitting your test directory into two parts: `success` and `failure`. The former is for tests that should succeed now, and the latter is for tests that are currently expected to fail. See the Typed Racket testing arrangement for an example. When you create such `failure` tests, you may wish to disable DrDr's checking like this:

```
  git prop set drdr:command-line "" <file> ...
```

This is a Racket-specific `git` command.

2.2 Always Test!

When you debug an existing piece of code, formulate a test case first. Put it into the test suite for the component so the mistake will never be accidentally re-introduced and add a note that points to the problem report. Second, modify the code to fix the mistake. Do this second to be sure you didn't introduce a mistake in your tests; it is all too easy to think you have fixed a mistake when in reality your new test just doesn't properly reveal the old mistake. Third, re-run the test suite to ensure that the mistake is fixed and no existing tests fail.

If there is no test suite and you aren't sure how to build one, then ask on the developer mailing list. Perhaps people will explain why there isn't one or they will sketch how to create one. Please don't ignore the problem. If you cannot build a test suite, you have a few options:

1. Add functionality to the library to enable testing. Of course, adding functionality

means adding external documentation. Robby and Matthew have done so for the GUI library, and there is now a large automated test suite for DrRacket. So even GUI programs can come with extended test suites.

2. Add an end-to-end test that may have to be verified by a human. For example, it might be hard to test Slideshow, so you could create a slide set and describe what it should look like so future maintainers to verify when *they* make changes. Consider this the *last and least desirable* option, however.

The lack of tests for some collection will not disappear overnight. But if we all contribute a little bit, we will eventually expand the test suites to cover the entire code base, and future generations of maintainers will be grateful.

3 Units of Code

3.1 Organization Matters

We often develop units of code in a bottom-up fashion with some top-down planning. There is nothing surprising about this strategy because we build code atop of existing libraries, which takes some experimentation, which in turn is done in the REPL. We also want testable code quickly, meaning we tend to write down those pieces of code first for which we can develop and run tests. Readers don't wish to follow our development, however; they wish to understand what the code computes without necessarily understanding all the details.

So, please take the time to present each unit of code in a top-down manner. This starts with the implementation part of a module. Put the important functions close to the top, right below any code and comments as to what kind of data you use. The rule also applies to classes, where you want to expose `public` methods before you tackle `private` methods. And the rule applies to units, too.

3.2 Size Matters

Keep units of code small. Keep modules, classes, functions and methods small.

A module of 10,000 lines of code is too large. A module of 1,000 lines is tolerable. A module of 500 lines of code has the right size.

One module should usually contain a class and its auxiliary functions, which in turn determines the length of a good-sized class.

And a function/method/syntax-case of roughly 66 lines is usually acceptable. The 66 is based on the length of a screen with small font. It really means "a screen length." Yes, there are exceptions where functions are more than 1,000 lines long and extremely readable. Nesting levels and nested loops may look fine to you when you write code, but readers will not appreciate it keeping implicit and tangled dependencies in their mind. It really helps the reader to separate functions (with what you may call manual lambda lifting) into a reasonably flat organization of units that fit on a (laptop) screen and explicit dependencies.

For many years we had a limited syntax transformation language that forced people to create *huge* functions. This is no longer the case, so we should try to stick to the rule whenever possible.

If a unit of code looks incomprehensible, it is probably too large. Break it up. To bring across what the pieces compute, implement or serve, use meaningful names; see §6.6 "Names". If you can't come up with a good name for such pieces, you are probably looking at the wrong kind of division; consider alternatives.

3.3 Modules and their Interfaces

The purpose of a module is to provide some services:

Equip a module with a short purpose statement.

Often "short" means one line; occasionally you may need several lines.

In order to understand a module's services, organize the module in three sections below the purpose statement: its exports, its imports, and its implementation:

good

```
#lang racket/base

;; the module implements a tv server

(provide
 ;; launch the tv server function
 tv-launch
 ;; set up a tv client to receive messages from the tv server
 tv-client)

;; ---------------------------------------------------------------------
;; import and implementation section

(require 2htdp/universe htdp/image)

(define (tv-launch)
  (universe ...))

(define (tv-client)
  (big-bang ...))
```

If you choose to use `provide` with `contract-out`, you may wish to have two `require` sections:

- the first one, placed with the `provide` section, imports the values needed to formulate the contracts and
- the second one, placed below the `provide` section, imports the values needed to implement the services.

If your contracts call for additional concepts, define those right below the `provide` specification:

good

```
#lang racket/base

;; the module implements a tv server

(require racket/contract)

(provide
 (contract-out
  ;; initialize the board for the given number of players
  [board-init        (-> player#/c plain-board/c)]
  ;; initialize a board and place the tiles
  [create-board      (-> player#/c (listof placement/c)
                         (or/c plain-board/c string?))]
  ;; create a board from an X-expression representation
  [board-deserialize (-> xexpr? plain-board/c)]))

(require xml)

(define player# 3)
(define plain-board/c
  (instanceof/c (and/c admin-board%/c board%-contracts/c)))

(define placement/c
  (flat-named-contract "placement" ...))

;; ------------------------------------------------------------------
;; import and implementation section

(require 2htdp/universe htdp/image)

;; implementation:
(define (board-init n)
  (new board% ...))

(define (create-board n lop)
  (define board (board-init n))
  ...)

(define board%
  (class ... some 900 lines ...))
```

In the preceding code snippet, `xml` imports the `xexpr?` predicate. Since the latter is needed to articulate the contract for `board-deserialize`, the `require` line for `xml` is a part of the `provide` section. In contrast, the `require` line below the lines imports an event-handling mechanism plus a simple image manipulation library, and these tools are needed only for the implementation of the provided services.

Prefer specific export specifications over `(provide (all-defined-out))`.

A test suite section—if located within the module—should come at the very end, including its specific dependencies, i.e., `require` specifications.

3.3.1 Require

With `require` specifications at the top of the implementation section, you let every reader know what is needed to understand the module.

3.3.2 Provide

A module's interface describes the services it provides; its body implements these services. Others have to read the interface if the external documentation doesn't suffice:

Place the interface at the top of the module.

This helps people find the relevant information quickly.

good

```
#lang racket

;; This module implements
;; several strategies.

(provide
 ;; Stgy = State -> Action

 ;; Stgy
 ;; people's strategy
 human-strategy

 ;; Stgy
 ;; tree traversal
 ai-strategy)

;; - - - - - - - - - - - -
;; implementation

(require "basics.rkt")

(define (general p)
  ...)

... some 100 lines ...
(define human-strategy
  (general create-gui))

... some 100 lines ...
(define ai-strategy
  (general traversal))
```

bad

```
#lang racket

;; This module implements
;; several strategies.

;; - - - - - - - - - - - -
;; implementation

(require "basics.rkt")

;; Stgy = State -> Action

(define (general p)
  ...)
... some 100 lines ...

(provide
 ;; Stgy
 ;; a person's strategy
 human-strategy)

(define human-strategy
  (general create-gui))
... some 100 lines ...

(provide
 ;; Stgy
 ;; a tree traversal
 ai-strategy)

(define ai-strategy
  (general traversal))
... some 100 lines ...
```

As you can see from this comparison, an interface shouldn't just `provide` a list of names. Each identifier should come with a purpose statement. Type-like explanations of data may also show up in a `provide` specification so that readers understand what kind of data your public functions work on.

While a one-line purpose statement for a function is usually enough, syntax should come with a description of the grammar clause it introduces *and* its meaning.

good

```
#lang racket
```

```
(provide
 #; (define-strategy (s:id a:id b:id c:id d:id) action:definition-or-
expression)


 ;; (define-strategy (s board tiles available score) ...)
 ;; defines a function from an instance of player to a
 ;; placement. The four identifiers denote the state of
 ;; the board, the player's hand, the places where a
 ;; tile can be placed, and the player's current score.
 define-strategy)
```

Use `provide` with `contract-out` for module interfaces. Contracts often provide the right level of specification for first-time readers.

At a minimum, you should use type-like contracts, i.e., predicates that check for the constructor of data. They cost almost nothing, especially because exported functions tend to check such constraints internally anyway and contracts tend to render such checks superfluous.

If you discover that contracts create a performance bottleneck, please report the problem to the Racket developer mailing list.

3.3.3 Uniformity of Interface

Pick a rule for consistently naming your functions, classes, and methods. Stick to it. For example, you may wish to prefix all exported names with the name of the data type that they deal with, say `syntax-local`.

Pick a rule for consistently naming and ordering the parameters of your functions and methods. Stick to it. For example, if your module implements an abstract data type (ADT), all functions on the ADT should consume the ADT-argument first or last.

Finally pick the same name for all function/method arguments in a module that refer to the same kind of data—regardless of whether the module implements a common data structure. For example, in `"pkgs/racket-index/setup/scribble.rkt"`, all functions use `latex-dest` to refer to the same kind of data, even those that are not exported.

3.3.4 Sections and Sub-modules

Finally, a module consists of sections. It is good practice to separate the sections with comment lines. You may want to write down purpose statements for sections so that readers can easily understand which part of a module implements which service. Alternatively, consider using the large letter chapter headings in DrRacket to label the sections of a module.

With `rackunit`, test suites can be defined within the module using `define/provide-test-suite`. If you do so, locate the test section at the end of the module and `require` the necessary pieces for testing specifically for the test suites.

As of version 5.3, Racket supports sub-modules. Use sub-modules to formulate sections, especially test sections. With sub-modules it is now possible to break up sections into distinct parts (labeled with the same name) and leave it to the language to stitch pieces together.

`fahrenheit.rkt`

```
#lang racket

(provide
 (contract-out
  ;; convert a fahrenheit temperature to a celsius
  [fahrenheit->celsius (-> number? number?)]))

(define (fahrenheit->celsius f)
  (/ (* 5 (- f 32)) 9))

(module+ test
  (require rackunit)
  (check-equal? (fahrenheit->celsius -40) -40)
  (check-equal? (fahrenheit->celsius 32) 0)
  (check-equal? (fahrenheit->celsius 212) 100))
```

If you develop your code in DrRacket, it will run the test sub-module every time you click "run" unless you explicitly disable this functionality in the language selection menu. If you have a file and you just wish to run the tests, use `raco` to do so:

```
  $ raco test fahrenheit.rkt
```

Running this command in a shell will require and evaluate the test sub-module from the `fahrenheit.rkt`.

3.4 Classes & Units

(I will write something here sooner or later.)

3.5 Functions & Methods

If your function or method consumes more than two parameters, consider keyword arguments so that call sites can easily be understood. In addition, keyword arguments also "thin" out calls because function calls don't need to refer to default values of arguments that are considered optional.

Similarly, if your function or method consumes two (or more) *optional* parameters, keyword arguments are a must.

Write a purpose statement for your function. If you can, add an informal type and/or contract statement.

3.6 Contracts

A contract establishes a boundary between a service provider and a service consumer aka *server* and *client*. Due to historical reasons, we tend to refer to this boundary as a *module boundary*, but the use of "module" in this phrase does *not* only refer to file-based or physical Racket modules. Clearly, *contract boundary* is better than module boundary because it separates the two concepts.

When you use `provide` with `contract-out` at the module level, the boundary of the physical module and the contract boundary coincide.

When a module becomes too large to manage without contracts but you do not wish to distribute the source over several files, you may wish to use one of the following two constructs to erect contract boundaries internal to the physical module:

- `define/contract`
- `module`, as in submodule.

Using the first one, `define/contract`, is like using `define` except that it is also possible to add a contract between the header of the definition and its body. The following code display shows a file that erects three internal contract boundaries: two for plain constants and one for a function.

`celsius.rkt`

```
#lang racket

(define/contract AbsoluteC real? -273.15)
(define/contract AbsoluteF real? -459.67)

(define/contract (celsius->fahrenheit c)
  ;; convert a celsius temperature to a fahrenheit temperature
  (-> (and/c real? (>=/c AbsoluteC))
      (and/c real? (>=/c AbsoluteF)))
  ;; - IN -
  (+ (* 9/5 c) 32))
```

```
(module+ test
  (require rackunit)
  (check-equal? (celsius->fahrenheit -40) -40)
  (check-equal? (celsius->fahrenheit 0) 32)
  (check-equal? (celsius->fahrenheit 100) 212))
```

To find out how these contract boundaries work, you may wish to conduct some experiments:

1. Add the following line to the bottom of the file:

   ```
   (celsius->fahrenheit -300)
   ```

   Save to file and observe how the contract system blames this line and what the blame report tells you.

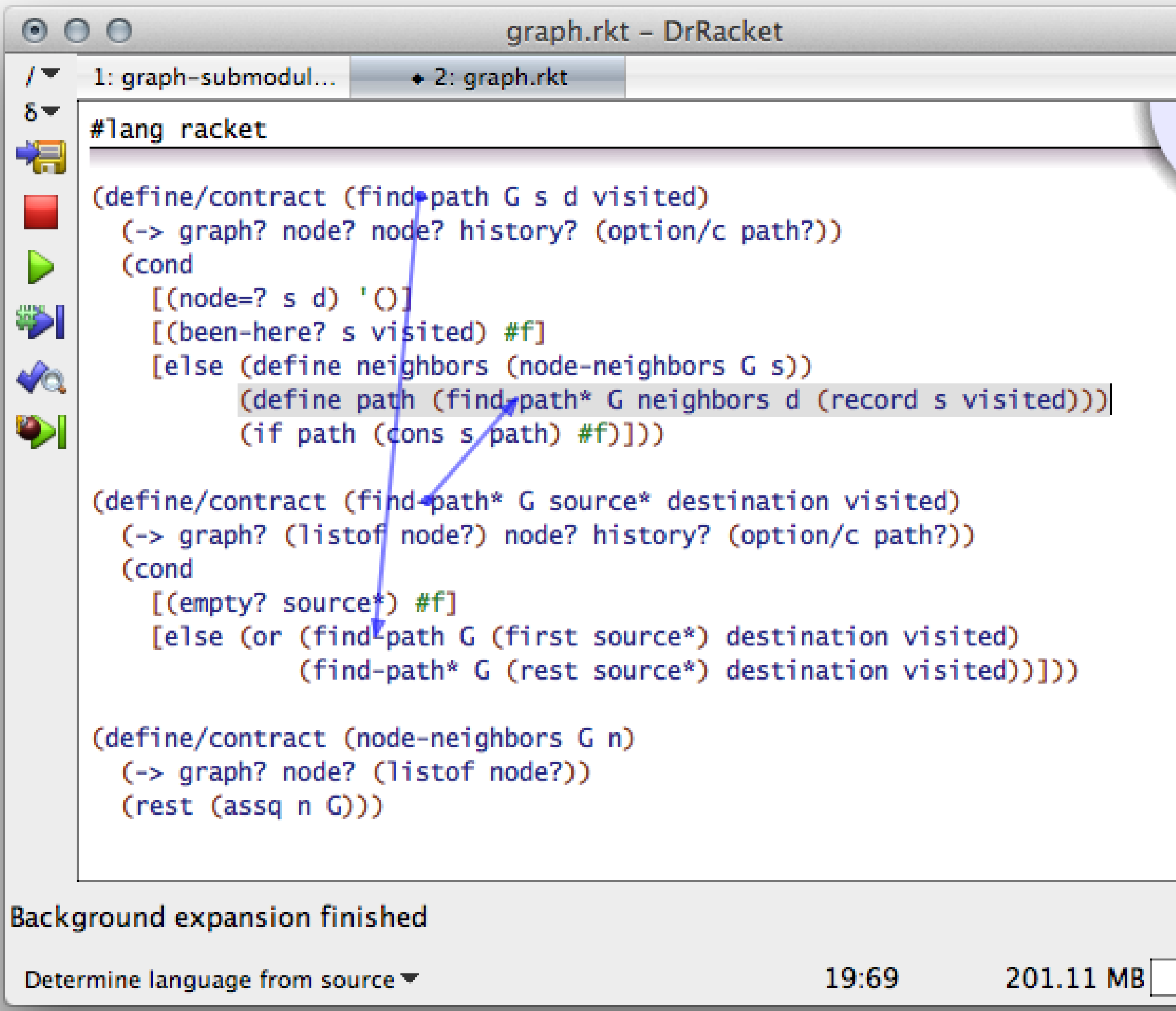
2. Replace the body of the `celsius->fahrenheit` function with

   ```
   (sqrt c)
   ```

   Once again, run the program and study the contract exceptions, in particular observe which party gets blamed.

3. Change the right-hand side of `AbsoluteC` to `0.0-273.15i`, i.e., a complex number. This time a different contract party gets blamed.

The screen shot below shows that `define/contract` works for mutually recursive functions with modules. This capability is unique to `define/contract`.



```
#lang racket

(define/contract (find-path G s d visited)
  (-> graph? node? node? history? (option/c path?))
  (cond
    [(node=? s d) '()]
    [(been-here? s visited) #f]
    [else (define neighbors (node-neighbors G s))
          (define path (find-path* G neighbors d (record s visited)))
          (if path (cons s path) #f)]))

(define/contract (find-path* G source* destination visited)
  (-> graph? (listof node?) node? history? (option/c path?))
  (cond
    [(empty? source*) #f]
    [else (or (find-path G (first source*) destination visited)
              (find-path* G (rest source*) destination visited))]))

(define/contract (node-neighbors G n)
  (-> graph? node? (listof node?))
  (rest (assq n G)))
```

Background expansion finished

Determine language from source 19:69 201.11 MB

In contrast, submodules act exactly like plain modules when it comes to contract boundaries. Like `define/contract`, a submodule establishes a contract boundary between itself and the rest of the module. Any value flow between a client module and the submodule is governed by contracts. Any value flow within the submodule is free of any constraints.

graph-traversal.rkt

```
#lang racket
...
(module traversal racket
  (provide
   (contract-out
    (find-path (-> graph? node? node? (option/c path?)))))

  (require (submod ".." graph) (submod ".." contract))

  (define (find-path G s d (visited history0))
    (cond
      [(node=? s d) '()]
      [(been-here? s visited) #f]
      [else (define neighbors (node-neighbors G s))
            (define there (record s visited))
            (define path (find-path* G neighbors d there))
            (if path (cons s path) #f)]))

  (define (find-path* G s* d visited)
    (cond
      [(empty? s*) #f]
      [else (or (find-path G (first s*) d visited)
                (find-path* G (rest s*) d visited))]))

  (define (node-neighbors G n)
    (rest (assq n G))))

(module+ test
  (require (submod ".." traversal) (submod ".." graph))
  (find-path G 'a 'd))
```

Since modules and submodules cannot refer to each other in a mutual recursive fashion, submodule contract boundaries cannot enforce constraints on mutually recursive functions. It would thus be impossible to distribute the `find-path` and `find-path*` functions from the preceding code display into two distinct submodules.

4 Choosing the Right Construct

Racket provides a range of constructs for the same or similar purposes. Although the Racket designers don't think that there is one right way for everything, we prefer certain constructs in certain situations for consistency and readability.

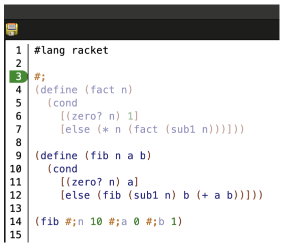
4.1 Comments

Following Lisp and Scheme tradition, we use a single semicolon for in-line comments (to the end of a line) and two semicolons for comments that start a line. Think of the second semicolon as making an emphatic point.

Seasoned Schemers, not necessarily Racketeers, also use triple and quadruple semicolons. This is considered a courtesy to distinguish file headers from section headers.

In addition to `;`, we have two other mechanisms for commenting code: `#|`...`|#` for blocks and `#;` to comment out an expression. *Block comments* are for those rare cases when an entire block of definitions and/or expressions must be commented out at once. *Expression comments*—`#;`—apply to the following S-expression. This makes them a useful tool for debugging. They can even be composed in interesting ways with other comments, for example, `#;#;` will comment two expressions, and a line with just `;#;` gives you a single-character "toggle" for the expression that starts on the next line.

The screenshots below illustrate the use of `#;` and how DrRacket and Emacs (Racket mode) color such comments by default.

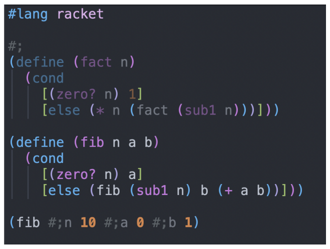


```
#lang racket

#;
(define (fact n)
  (cond
    [(zero? n) 1]
    [else (* n (fact (sub1 n)))]))

(define (fib n a b)
  (cond
    [(zero? n) a]
    [else (fib (sub1 n) b (+ a b))]))

(fib #;n 10 #;a 0 #;b 1)
```



```
#lang racket

#;
(define (fact n)
  (cond
    [(zero? n) 1]
    [else (* n (fact (sub1 n)))]))

(define (fib n a b)
  (cond
    [(zero? n) a]
    [else (fib (sub1 n) b (+ a b))]))

(fib #;n 10 #;a 0 #;b 1)
```

4.2 Definitions

Racket comes with quite a few definitional constructs, including `let`, `let*`, `letrec`, and `define`. Except for the last one, definitional constructs increase the indentation level. Therefore, favor `define` when feasible.

good

```
#lang racket

(define (swap x y)
  (define t (unbox x))
  (set-box! x (unbox y))
  (set-box! y t))
```

bad

```
#lang racket

(define (swap x y)
  (let ([t (unbox x)])
    (set-box! x (unbox y))
    (set-box! y t)))
```

Warning A `let*` binding block is not easily replaced with a series of `define`s because the former has *sequential* scope and the latter has *mutually recursive* scope.

works

```
#lang racket

(define (print-two f)
  (let* ([_ (print (first f))]
         [f (rest f)]
         [_ (print (first f))]
         [f (rest f)])
    ;; IN
    f))
```

does **not**

```
#lang racket

(define (print-two f)
  (print (first f))
  (define f (rest f))
  (print (first f))
  (define f (rest f))
  ;; IN
  f)
```

4.3 Conditionals

Like definitional constructs, conditionals come in many flavors, too. Because `cond` and its relatives (`case`, `match`, etc) now allow local uses of `define`, you should prefer them over `if`.

good

```
#lang racket

(cond
  [(empty? l) #false]
  [else
   (define f (first l))
   (define r (rest l))
   (if (discounted? f)
       (rate f)
       (curved (g r)))])
```

bad

```
#lang racket

(if (empty? l)
    #false
    (let ([f (first l)]
          [r (rest l)])
      (if (discounted? f)
          (rate f)
          (curved (g r)))))
```

Also, use `cond` instead of `if` to eliminate explicit `begin`.

The above "good" example would be even better with `match`. In general, use `match` to destructure complex pieces of data.

You should also favor `cond` (and its relatives) over `if` to match the shape of the data definition. In particular, the above examples could be formulated with `and` and `or` but doing so would not bring across the recursion as nicely.

4.4 Expressions

Don't nest expressions too deeply. Instead name intermediate results. With well-chosen names your expression becomes easy to read.

good

```
#lang racket
(define (next-month d)
  (define day (first d))
  (define month (second d))
  (if (= month 12)
      `(,(+ day 1) 1)
      `(,day ,(+ month 1))))
```

bad

```
#lang racket
(define (next-month d)
  (if (= (second d) 12)
      `(,(+ (first d) 1)
        1)
      `(,(first d)
        ,(+ (second d) 1))))
```

Clearly "too deeply" is subjective. On occasion it also isn't the nesting that makes the expression unreadable but the sheer number of subexpressions. Consider using local definitions for this case, too.

4.5 Structs vs Lists

Use `structs` when you represent a combination of a small and fixed number of values. For fixed length (long) lists, add a comment or even a contract that states the constraints.

If a function returns several results via `values`, consider using `structs` or lists when you are dealing with four or more values.

4.6 Lambda vs Define

While nobody denies that `lambda` is cute, `define`d functions have names that tell you what they compute and that help accelerate reading.

good

```
#lang racket

(define (process f)
  (define (complex-step x)
    ... 10 lines ...)
  (map complex-step
       (to-list f)))
```

bad

```
#lang racket

(define (process f)
  (map (lambda (x)
         ... 10 lines ...)
       (to-list f)))
```

Even a curried function does not need `lambda`.

good

```
#lang racket

(define ((cut fx-image) image2)
  ...)
```

acceptable

```
#lang racket

(define (cut fx-image)
  (lambda (image2)
    ...))
```

The left side signals currying in the very first line of the function, while the reader must read two lines for the version on the right side.

Of course, many constructs (e.g. `call-with-values`) or higher-order functions (e.g. `filter`) are made for short `lambda`; don't hesitate to use `lambda` for such cases.

4.7 Identity Functions

The identity function is `values`:

Examples:

```
> (map values '(a b c))
'(a b c)
> (values 1 2 3)
1
2
3
```

4.8 Traversals

With the availability of `for/fold`, `for/list`, `for/vector`, and friends, programming with `for` loops has become just as functional as programming with `map` and `foldr`. With `for*` loops, filter, and termination clauses in the iteration specification, these loops are also far more concise than explicit traversal combinators. And with `for` loops, you can decouple the traversal from lists.

See also `for/sum` and `for/product` in Racket.

good

```
#lang racket

;; [Sequence X] -> Number
(define (sum-up s)
  (for/fold ([sum 0]) ([x s])
    (+ sum x)))

;; examples:
(sum-up '(1 2 3))
(sum-up #(1 2 3))
(sum-up
  (open-input-string
    "1 2 3"))
```

bad

```
#lang racket

;; [Listof X] -> Number
(define (sum-up alist)
  (foldr (lambda (x sum)
           (+ sum x))
         0
         alist))

;; example:
(sum-up '(1 2 3))
```

In this example, the `for` loop on the left comes with two advantages. First, a reader doesn't need to absorb an intermediate `lambda`. Second, the `for` loop naturally generalizes to other kinds of sequences. Naturally, the trade-off here is a loss of efficiency; using `in-list` to restrict the `good` example to the same range of data as the `bad` one speeds up the former.

Note `for` traversals of user-defined sequences tend to be slow. If performance matters in these cases, you may wish to fall back on your own traversal functions.

4.9 Functions vs Macros

Define functions when possible, Or, do not introduce macros when functions will do.

good

```
#lang racket
...
;; Message -> String
(define (name msg)
  (first (second msg)))
```

bad

```
#lang racket
...
;; Message -> String
(define-syntax-rule (name msg)
  (first (second msg)))
```

A function is immediately useful in a higher-order context. For a macro, achieving the same goal takes a lot more work.

4.10 Exceptions

When you handle exceptions, specify the exception as precisely as possible.

good

```
#lang racket
...
;; FN [X -> Y] FN -> Void
(define (convert in f out)
  (with-handlers
      ([exn:fail:read? X])
    (with-output-to out
      (writer f))))

;; may raise exn:fail:read
(define ((writer f))
  (with-input-from in
    (reader f)))

;; may raise exn:fail:read
(define ((reader f))
  ... f ...)
```

bad

```
#lang racket
...
;; FN [X -> Y] FN -> Void
(define (convert in f out)
  (with-handlers
      ([(lambda _ #t) X])
    (with-output-to out
      (writer f))))

;; may raise exn:fail:read
(define ((writer f))
  (with-input-from in
    (reader f)))

;; may raise exn:fail:read
(define ((reader f))
  ... f ...)
```

Using `(lambda _ #t)` as an exception predicate suggests to the reader that you wish to catch every possible exception, including failure and break exceptions. Worse, the reader may think that you didn't remotely consider what exceptions you *should* be catching.

It is equally bad to use `exn?` as the exception predicate even if you mean to catch all kinds of failures. Doing so catches break exceptions, too. To catch all failures, use `exn:fail?` as shown on the left:

good

```
#lang racket
...
;; FN [X -> Y] FN -> Void
(define (convert in f out)
  (with-handlers
      ([exn:fail? X])
    (with-output-to out
      (writer f))))

;; may raise exn:fail:read
(define ((writer f))
  (with-input-from in
    (reader f)))

;; may raise exn:fail:read
(define ((reader f))
 ... f ...)
```

bad

```
#lang racket
...
;; FN [X -> Y] FN -> Void
(define (convert in f out)
  (with-handlers
      ([exn? X])
    (with-output-to out
      (writer f))))

;; may raise exn:fail:read
(define ((writer f))
  (with-input-from in
    (reader f)))

;; may raise exn:fail:read
(define ((reader f))
 ... f ...)
```

Finally, a handler for a `exn:fail?` clause should never succeed for all possible failures because it silences all kinds of exceptions that you probably want to see:

bad

```
#lang racket
...
;; FN [X -> Y] FN -> Void
(define (convert in f out)
  (with-handlers ([exn:fail? handler])
    (with-output-to out
      (writer f))))

;; Exn -> Void
(define (handler e)
  (cond
    [(exn:fail:read? e)
     (displayln "drracket is special")]
    [else (void)]))

;; may raise exn:fail:read
(define ((writer f))
  (with-input-from in
    (reader f)))

;; may raise exn:fail:read
(define ((reader f))
  ... f ...)
```

If you wish to deal with several different kind of failures, say exn:fail:read? and exn:fail:network?, use distinct clauses in with-handlers to do so and distribute the branches of your conditional over these clauses.

4.11 Parameters

If you need to set a parameter, use parameterize:

good

```
#lang racket
...
(define cop
  current-output-port)

;; String OPort -> Void
(define (send msg op)
  (parameterize ([cop op])
    (display msg))
  (record msg))
```

bad

```
#lang racket
...
(define cop
  current-output-port)

;; String OPort -> Void
(define (send msg op)
  (define cp (cop))
  (cop op)
  (display msg)
  (cop cp)
  (record msg))
```

As the comparison demonstrates, parameterize clearly delimits the extent of the change, which is an important idea for the reader. In addition, parameterize ensures that your code is more likely to work with continuations and threads, an important idea for Racket programmers.

4.12 Plural

Avoid plural when naming collections and libraries. Use racket/contract and data/heap, not racket/contracts or data/heaps.

5 Scribbling Documentation

This section describes good style for Racket documentation writing.

5.1 Prose and Terminology

In the descriptive body of `defform`, `defproc`, etc., do not start with "This ..." Instead, start with a sentence whose implicit subject is the form or value being described (but only start the first sentence that way). Capitalize the first word. Thus, the description will often start with "Returns" or "Produces." Refer to arguments and sub-forms by name.

Do not use the word "argument" to describe a sub-form in a syntactic form; use the term "sub-form" instead, reserving "argument" for values or expressions in a function call. Refer to libraries and languages as such, rather than as "modules" (even though the form to typeset a library or language name is called `racketmodname`). Do not call an identifier (i.e., a syntactic element) a "variable" or a "symbol." Do not use the word "expression" for a form that is a definition or might be a definition; use the word "form," instead. Prefer "function" to "procedure."

Use the word "list" only when you mean a run-time value consisting of the empty list and cons cells; use the word "sequence" in other cases, if you must use any word. For example, do not write that `begin` has a "list of sub-forms;" instead, it has a "sequence of sub-forms." Similarly, do not refer to a "list of arguments" in a function call; just write "arguments" if possible, or write "sequence of argument expressions." (Unfortunately, "sequence" has acquired a specific run-time meaning, too, but the collision is less severe than the historical confusion between lists and other entities in Lisp.)

Avoid cut-and-paste for descriptive text. If two functions are similar, consider documenting them together with `deftogether`. To abstract a description, consider using explicit prose abstraction, such as "`x` is like `y`, except that ...," instead of abstracting the source and instantiating it multiple times; often, a prose abstraction is clearer to the reader than a hidden abstraction in the document implementation.

Hyphenate the words "sub-form" and "sub-expression."

Use "Windows," "Mac OS," and "Unix" for the three "platforms" (as opposed to "systems") on which Racket runs. Use "Unix" as a generic term for Unix-like operating systems—notably including Linux—other than Mac OS. Use "Unix" even when "Gtk" or "the X11 windowing system" would be more precisely correct, but use "X11" as adjective when necessary, such as "X11 display." Racket runs "on" a platform, as opposed to "under" a platform.

Avoid using a predicate as a noun that stands for a value satisfying the predicate. Instead, use `tech` and `deftech` to establish a connection between an English word or phrase that describes the class of values and the predicate (or contract). For example, avoid "supply a `path-string?`"; prefer "supply a path or string."

5.2 Typesetting Code

Use `id` or a name that ends `-id` in `defform` to mean an identifier, not `identifier`, `variable`, `name`, or `symbol`. Similarly, use `expr` or something that ends `-expr` for an expression position within a syntactic form. Use `body` for a form (definition or expression) in an internal-definition position—always followed by `...+` in a grammar description. Do not use `expr` for something that isn't exactly an expression, `id` for something that isn't exactly an identifier, etc.; instead, use `defform/subs` to define a new non-terminal.

Beware of using `deftogether` to define multiple variants of a syntactic form or procedure, because each `defform` or `defproc` creates a definition point, but each form or procedure should have a single definition point. (Scribble issues a warning when a binding has multiple definition points.) Instead, use `defproc*` or `defform*`.

For function arguments, use `v` as the meta-variable for "any value." Use `x` as a meta-variable only for numerical values. Other conventions include `lst` for a list and `proc` for a procedure.

Pay attention to the difference between identifiers and meta-variables when using `racket`, especially outside of `defproc` or `defform`. Prefix a meta-variable with `_`; for example,

```
@racket[(rator-expr rand-expr ...)]
```

would be the wrong way to refer to the grammar of a function call, because it produces `(rator-expr rand-expr ...)`, where `rator-expr` and `rand-expr` are typeset as variables. The correct description is

```
@racket[(_rator-expr _rand-expr ...)]
```

which produces `(`*`rator-expr rand-expr`*` ...)`, where `rator-expr` and `rand-expr` are typeset as meta-variables. The `defproc`, `defform`, etc. forms greatly reduce this burden in descriptions, since they automatically set up meta-variable typesetting for non-literal identifiers. In `defform`, be sure to include literal identifiers (i.e., those not meant as variables, other than the form name being defined) in a `#:literals` clause.

To typeset an identifier with no particular interpretation—syntax, variable, meta-variable, etc.—use `racketidfont` (e.g., as in `rand-expr` above). Otherwise, use `litchar`, not merely `racketfont` or `verbatim`, to refer to a specific sequence of characters.

When a syntactic form synthesizes an identifier from a given identifier, use a combination of `racketidfont` and `racket` to describe the identifiers. For example, if *`id`* is combined with `is-` and `?` to form `is-`*`id`*`?`, then implement that identifier as `@racketidfont{is-}@racket[id]@racketidfont{?}`.

When using `defform` to describe a syntactic form, don't confuse the `#:contracts` clause with a grammar specification. Use `#:contracts` only for expressions within the syntactic form, and the contract is a run-time constraint—not a syntactic constraint, such as requiring a sub-form to be an identifier. Use `defform/subs` for syntactic constraints.

When showing example evaluations, use the REPL-snapshot style:

```
@examples[
 (+ 1 2)
]
```

See also the `scribble/example` library and §5.6 "Examples".

Use four dots, `....`, in place of omitted code, since `...` means repetition.

5.3 Typesetting Prose

Refrain from referring to documentation "above" or "below," and instead have a hyperlink point to the right place.

In prose, use ``` `` ``` and `''` quotation marks instead of `"`. Use `---` for an em dash, and do not include spaces on either side. Use American style for quotation marks and punctuation at the end of quotation marks (i.e., a sentence-terminating period goes inside the quotation marks). Of course, this rule does not apply for quotation marks that are part of code.

Do not use a citation reference (as created by `cite`) as a noun; use it as an annotation.

Do not start a sentence with a Racket variable name, since it is normally lowercase. For example, use "The *`thing`* argument is..." instead of "*`thing`* is..."

Use `etc` for "etc." when it does not end a sentence, and include a comma after "etc." unless it ends a sentence that is followed by other punctuation (such as a parenthesis).

5.4 Section Titles

Capitalize all words except articles ("the," "a," etc.), prepositions, and conjunctions that are not at the start of the title.

A manual title should normally start with a suitable keyword or key phrase (such as "Scribble" for this manual) that is in boldface. If the key word is primarily an executable name, use `exec` instead of `bold`. Optionally add further descriptive text in the title after a colon, where the text starting with the colon is not in boldface.

5.5 Indexing

Document and section titles, identifiers that are documented with `defproc`, `defform`, etc. are automatically indexed, as are terms defined with `deftech`.

Symbols are not indexed automatically. Use `indexed-racket` instead of `racket` for the instance of a symbol that roughly defines the use. For an example, try searching for "truncate" to find `'truncate` as used with `open-output-file`. Do not use something like `(index "'truncate")` to index a symbol, because it will not typeset correctly (i.e., in a fixed-width font with the color of a literal).

Use `index`, `as-index`, and `section-index` as a last resort. Create index entries for terms that are completely different from terms otherwise indexed. Do not try to index minor variations of a term or phrase in an attempt to improve search results; if search fails to find a word or phrase due to a minor variation, then the search algorithm should be fixed, not the index entry.

5.6 Examples

Strive to include examples (using `examples`) with the documentation of every function and syntactic form. When writing examples, refrain from using nonsense words like "foo" and "bar." For example, when documenting `member`, resist the temptation to write

```
> (member "foo" '("bar" "foo" "baz"))
'("foo" "baz")
```

and instead write something like

```
> (member "Groucho" '("Harpo" "Groucho" "Zeppo"))
'("Groucho" "Zeppo")
```

6 Textual Matters

Simple textual conventions help eyes find pieces of code quickly. Here are some of those that are easy to check—some automatically and some manually. If you find yourself editing a file that violates some of the constraints below, edit it into the proper shape.

Warning: On rare occasion a unit test may depend on the indentation of a file. This is extremely rare and must be noted at the top so that readers do not accidentally re-indent the file.

6.1 Where to Put Parentheses

Racket isn't C. Put all closing parentheses on one line, the last line of your code.

good

```
#lang racket

(define (conversion f)
  (* 5/9 (- f 32)))
```

really bad

```
#lang racket
(define (conversion f)
  (* 5/9 (- f 32)
    )
  )
```

You are allowed to place all closing parenthesis on a line by itself at the end of long sequences, be those definitions or pieces of data.

acceptable

```
#lang racket
(define modes
  '(edit
    help
    debug
    test
    trace
    step))
```

also acceptable

```
#lang racket
(define turn%
  (class object%
    (init-field state)

    (super-new)

    (define/public (place where tile)
      (send state where tile))

    (define/public (is-placable? place)
      (send state legal? place))
    ))
```

Doing so is most useful when you expect to add, delete, or swap items in such sequences.

6.2 Indentation

DrRacket indents code and it is the only tool that everyone in PLT agrees on. So use DrRacket's indentation style. Here is what this means.

For every file in the repository, DrRacket's "indent all" functions leaves the file alone.

If you prefer to use some other editor (emacs, vi/m, etc), program it so that it follows DrRacket's indentation style.

Examples:

good

```
#lang racket

;; drracket style
(if (positive? (rocket-x r))
    (launch r)
    (redirect (- x)))
```

bad

```
#lang racket

;; .el emacs-file if
(if (positive? (rocket-x r))
    (launch r)
  (redirect (- x)))
```

Caveat 1: Until language specifications come with fixed indentation rules, we need to use the *default* settings of DrRacket's indentation for this rule to make sense. If you add new constructs, say a for loop, please contact Robby for advice on how to add a default setting for the indentation functionality. If you add entire languages, say something on the order of Typed Racket, see DrRacket support for `#lang`-based Languages for how to implement tabbing.

Caveat 2: This rule does not apply to scribble code.

6.3 Tabs

Do not use tab characters in your code. Tabs make it hard to use textual tools like git or diff effectively. To disable tabs,

- in DrRacket: you are all set. It doesn't insert tabs.
- in Emacs: add `(setq indent-tabs-mode nil)` to your emacs initialization file.
- in vi: `:set expandtab`

6.4 Line Width

A line in a Racket file is at most 102 characters wide.

If you prefer a narrower width than 102, and if you stick to this width “religiously,” add a note to the top of the file—right below the purpose statement—that nobody should violate your file-local rule.

This number is a compromise. People used to recommend a line width of 80 or 72 column. The number is a historical artifact. It is also a good number for several different reasons: printing code in text mode, displaying code at reasonable font sizes, comparing several different pieces of code on a monitor, and possibly more. So age doesn't make it incorrect. We regularly read code on monitors that accommodate close to 250 columns, and on occasion, our monitors are even wider. It is time to allow for somewhat more width in exchange for meaningful identifiers.

So, when you create a file, add a line with `;;` followed by ctrl-U 99 and `-`. When you separate "sections" of code in a file, insert the same line. These lines help both writers and readers to orient themselves in a file. In scribble use `@;` as the prefix.

6.5 Line Breaks

Next to indentation, proper line breaks are critical.

For an `if` expression, put each alternative on a separate line.

good

```
#lang racket

(if (positive? x)
    (launch r)
    (redirect (- x)))
```

bad

```
#lang racket

(if (positive? x) (launch r)
    (redirect (- x)))
```

It is acceptable to have an entire `if` expressions on one line if it fits within the specified line width (102):

also good

```
#lang racket

(if (positive? x) x (- x))
```

Each definition and each local definition deserves at least one line.

good

```
#lang racket

(define (launch x)
  (define w 9)
  (define h 33)
  ...)
```

bad

```
#lang racket

(define (launch x)
  (define w 9) (define h 33)
  ...)
```

All of the arguments to a function belong on a single line unless the line becomes too long, in which case you want to put each argument expression on its own line

good

```
#lang racket

(place-image img 10 10 background)

;; and

(above img
       (- width  hdelta)
       (- height vdelta)
       bg)
```

bad

```
#lang racket

(above ufo
       10 v-delta bg)
```

Here is an exception:

good

```
#lang racket

(overlay/offset (rectangle 100 10 "solid" "blue")
                10 10
                (rectangle 10 100 "solid" "red"))
```

In this case, the two arguments on line 2 are both conceptually related and short.

6.6 Names

Use meaningful names. The Lisp convention is to use full English words separated by dashes. Racket code benefits from the same convention.

good

```
#lang racket

render-game-state

send-message-to-client

traverse-forest
```

bad

```
#lang racket

rndr-st

sendMessageToClient

traverse_forest
```

Note that _ (the underline character) is also classified as bad Racketeering within names. It is an acceptable placeholder in syntax patterns, match patterns, and parameters that don't matter.

Another widely used convention is to *prefix* a function name with the data type of the main argument. This convention generalizes the selector-style naming scheme of `struct`.

good

```
#lang racket

board-free-spaces      board-closed-spaces    board-serialize
```

In contrast, variables use a *suffix* that indicates their type:

good

```
#lang racket

(define (win-or-lose? game-state)
  (define position-nat-nat (game-state-position game-state))
  (define health-level-nat (game-state-health game-state))
  (define name-string      (game-state-name game-state))
  (define name-symbol      (string->symbol name-string))
  ...)
```

The convention is particularly helpful when the same piece of data shows up in different guises, say, symbols and strings.

Names are bad if they heavily depend on knowledge about the context of the code. It prevents readers from understanding a piece of functionality at an approximate level without also reading large chunks of the surrounding and code.

Finally, in addition to regular alphanumeric characters, Racketeers use a few special characters by convention, and these characters indicate something about the name:

Character	Kind	Example
`?`	predicates and boolean-valued functions	`boolean?`
`!`	setters and field mutators	`set!`
`%`	classes	`game-state%`
`<%>`	interfaces	`dc<%>`
`^`	unit signatures	`game-context^`
`@`	units	`testing-context@`
`#%`	kernel identifiers	`#%app`
`/`	"with" (a preposition)	`call/cc`

The use of `#%` to prefix names from the kernel language warns readers that these identifiers are extremely special and they need to watch out for subtleties. No other identifiers start with `#` and, in particular, all tokens starting with `#:` are keywords.

Identifiers with the `#%` prefix are mostly used in modules that define new languages.

6.7 Graphical Syntax

Do not use graphical syntax (comment boxes, XML boxes, etc).

The use of graphical syntax makes it impossible to read files in alternative editors. It also messes up some revision control systems. When we figure out how to save such files in an editor-compatible way, we may relax this constraint.

6.8 Spaces

Don't pollute your code with spaces at the end of lines.

If you find yourself breaking long blocks of code with blank lines to aid readability, consider refactoring your program to introduce auxiliary functions so that you can shorten these long blocks of code. If nothing else helps, consider using (potentially) empty comment lines.

In addition, every pair of expressions on a line should have at least one space between the two, even if they're separated by parentheses.

good

```
#lang racket

(define (f x g)
  (cond [(< x 3) (g (g 3))]))
```

bad

```
#lang racket

(define(f x g)
  (cond[(< x 3)(g(g 3))]))
```

6.9 End of File

End files with a newline.

7 Language and Performance

When you write a module, you first pick a language. In Racket you can choose a lot of languages. The most important choice concerns `racket/base` vs `racket`.

For scripts, use `racket/base`. The `racket/base` language loads significantly faster than the `racket` language because it is much smaller than the `racket`.

If your module is intended as a library, stick to `racket/base`. That way script writers can use it without incurring the overhead of loading all of `racket` unknowingly.

Conversely, you should use `racket` (or even `racket/gui`) when you just want a convenient language to write some program. The `racket` language comes with almost all the batteries, and `racket/gui` adds the rest of the GUI base.

7.1 Library Interfaces

Imagine you are working on a library. You start with one module, but before you know it the set of modules grows to a decent size. Client programs are unlikely to use all of your library's exports and modules. If, by default, your library includes all features, you may cause unnecessary mental stress and run-time cost that clients do not actually use.

In building the Racket language, we have found it useful to factor libraries into different layers so that client programs can selectively import from these bundles. The specific Racket practice is to use the most prominent name as the default for the module that includes everything. When it comes to languages, this is the role of `racket`. A programmer who wishes to depend on a small part of the language chooses to `racket/base` instead; this name refers to the basic foundation of the language. Finally, some of Racket's constructs are not even included in `racket`—consider `racket/require` for example—and must be required explicitly in programs.

Other Racket libraries choose to use the default name for the small core. Special names then refer to the complete library.

We encourage library developers to think critically about these decisions and decide on a practice that fits their taste and understanding of the users of their library. We encourage developers to use the following names for different places on the "size" hierarchy:

- `library/kernel`, the bare minimal conceivable for the library to be usable;
- `library/base`, a basic set of functionality.
- `library`, an appropriate "default" of functionality corresponding to either `library/base` or `library/full`.

- `library/full`, the full library functionality.

Keep two considerations in mind as you decide which parts of your library should be in which files: dependency and logical ordering. The smaller files should depend on fewer dependencies. Try to organize the levels so that, in principle, the larger libraries can be implemented in terms of the public interfaces of the smaller ones.

Finally, the advice of the previous section, to use `racket/base` when building a library, generalizes to other libraries: by being more specific in your dependencies, you are a responsible citizen and enable others to have a small (transitive) dependency set.

7.2 Macros: Space and Performance

Macros copy code. Also, Racket is really a tower of macro-implemented languages. Hence, a single line of source code may expand into a rather large core expression. As you and others keep adding macros, even the smallest functions generate huge expressions and consume a lot of space. This kind of space consumption may affect the performance of your project and is therefore to be avoided.

When you design your own macro with a large expansion, try to factor it into a function call that consumes small thunks or procedures.

good

```
#lang racket
...
(define-syntax (search s)
  (syntax-parse s
    [(_ x (e:expr ...)
        (~datum in)
        b:expr)
     #'(sar/λ (list e ...)
              (λ (x) b))]))

(define (sar/λ l p)
  (for ([a '()]) ([y l])
    (unless (bad? y)
      (cons (p y) a))))

(define (bad? x)
  ... many lines ...)
```

bad

```
#lang racket
...
(define-syntax (search s)
  (syntax-parse s
    [(_ x (e:expr ...)
        (~datum in)
        b:expr)
     #'(begin
         (define (bad? x)
           ... many lines ...)
         (define l
           (list e ...))
         (for ([a '()]) ([x l])
           (unless (bad? x)
             (cons b a))))]))
```

As you can see, the macro on the left calls a function with a list of the searchable values and a function that encapsulates the body. Every expansion is a single function call. In contrast, the macro on the right expands to many nested definitions and expressions every time it is used.

7.3 No Contracts

Adding contracts to a library is good.

On some occasions, contracts impose a significant performance penalty. For such cases, we recommend organizing the module into a main module as usual and a submodule called `no-contract` so that

- the `no-contract` submodule `provides` the functionality *without* contracts,
- the main module `provides` the functionality *with* contracts.

This section explains three strategies for three different situations and levels of implementation complexity.

We will soon supply a Reference section in the Evaluation Model chapter that explains the basics of our understanding of "safety" and link to it.

Warning Splitting contracted functionality into two modules in this way renders the code in the `no-contract` module **unsafe**. The creator of the original code might have assumed certain constraints on some functions' arguments, and the contracts checked these constraints. While the documentation of the `no-contract` submodule is likely to state these constraints, it is left to the client to check them. If the client code doesn't check the constraints and the arguments don't satisfy them, the code in the `no-contract` submodule may go wrong in various ways.

The *first* and simplest way to create a `no-contract` submodule is to use the `#:unprotected-submodule` functionality of `contract-out`.

good

```
#lang racket

(define state? zero?)
(define action? odd?)
(define strategy/c
  (-> state? action?))

(provide
 (contract-out
  [human strategy/c]
  [ai strategy/c]))


;; - - - - - - - - - - - -
;; implementation

(define (general p)
  (lambda (_) pi))

(define (human x)
  ((general 'gui) x))

(define (ai x)
  ((general 'tra) x))
```

fast

```
#lang racket

(define state? zero?)
(define action? odd?)
(define strategy/c
  (-> state? action?))

(provide
 (contract-out
  #:unprotected-submodule no-contract
  [human strategy/c]
  [ai strategy/c]))

;; - - - - - - - - - - - -
;; implementation

(define (general s)
  (lambda (_) pi))

(define (human x)
  ((general 'gui) x))

(define (ai x)
  ((general 'tra) x))
```

The module called good illustrates what the code might look like originally. Every exported functions come with contracts, and the definitions of these functions can be found below the `provide` specification in the module body. The `fast` module on the right requests the creation of a submodule named `no-contract`, which exports the same identifiers as the original module but without contracts.

Once the submodule exists, using the library with or without contracts is straightforward:

needs-goodness

```
#lang racket

(require "fast.rkt")

human
;; comes with contracts
;; as if we had required
;; "good.rkt" itself

(define state1 0)
(define state2
  (human state1))
```

needs-speed

```
#lang racket

(require (submod "fast.rkt" no-contract))

human
;; comes without
;; a contract

(define state*
  (build-list 0 1))
(define action*
  (map human state*))
```

Both modules require the fast module, but needs-goodness on the left goes through the contracted provide while needs-speed on the right uses the no-contract submodule. Tchnically, the left module imports human with contracts; the right one imports the same function without contract and thus doesn't have to pay the performance penalty.

Notice, however, that when you run these two client modules—assuming you saved them with the correct names in some folder—the left one raises a contract error while the right one binds action* to

```
'(3.141592653589793 3.141592653589793)
```

The no-contract submodule generated by this first, easy approach retains the dependency on racket/contract at both compile and run time. Here is a variant of the above module that demonstrates this point:

problems-with-unprotected-submodule

```
#lang racket

(define state? zero?)
(define action? odd?)
(define strategy/c (-> state? action?))

(provide
 (contract-out
  #:unprotected-submodule no-contract
  [human strategy/c]
  [ai strategy/c]))

(define (general p) pi)

(define human (general 'gui))
```

```
(define ai (general 'tra))
```

Even though the contract-out specification seems to remove the contracts, requiring the no-contract still raises a contract error:

```
  (require (submod "." server no-contract))
```

Explanation The no-contract submodule depends on the main module, so the require runs the body of the main module, and doing so checks the first-order properties of the exported values. Because human is not a function, this evaluation raises a contract error.

The *second* way to create a no-contract submodule requires systematic work from the developer and eliminates the run-time dependency on racket/contract. Here are the two modules from above, with the right one derived manually from the one on the left:

good2

```
#lang racket

(define state? zero?)
(define action? odd?)
(define strategy/c
  (-> state? action?))

(provide
 (contract-out
  [human strategy/c]
  [ai strategy/c]))

;; - - - - - - - - - - - -
;; implementation

(define (general p)
  (lambda (_) pi))

(define (human x)
  ((general 'gui) x))

(define (ai x)
  ((general 'tra) x))
```

fast2

```
#lang racket

(define state? zero?)
(define action? odd?)
(define strategy/c
  (-> state? action?))

(provide
 (contract-out
  [human strategy/c]
  [ai strategy/c]))

;; - - - - - - - - - - - -
;; implementation

(module no-contract racket
  (provide
   human
   ai)

  (define (general s)
    (lambda (_) pi))

  (define (human x)
    ((general 'gui) x))

  (define (ai x)
    ((general 'tra) x)))

(require 'no-contract)
```

The `fast2` module on the right encapsulates the definitions in a submodule called `no-contract`; the `provide` in this submodule exports the exact same identifiers as the `good2` module on the left. The main module `requires` the submodule immediately, making the identifiers available in the outer scope so that the contracted `provide` can re-export them.

While this second way of creating a `no-contract` submodule eliminates the run-time dependency on `racket/contract`, its compilation—as a part of the outer module—still depends on this library, which is problematic in a few remaining situations.

The *third* and last way to create a `no-contract` submodule is useful when contracts prevents a module from being used in a context where contracts aren't available at all—neither at compile nor at run time. One example is `racket/base`; another is the contracts library itself. Again, you may wish you had the same library without contracts. For these cases, we recommend a file-based strategy one. Assuming the library is located at `a/b/c`, we recommend

1. creating a `c/` sub-directory with the file `no-contract.rkt`,
2. placing the functionality into `no-contract.rkt`,
3. adding `(require "c/no-contract.rkt")` to `c.rkt`, and
4. exporting the functionality from there with contracts.

Once this arrangement is set up, a client module in a special context `racket/base` or for `racket/contract` can use `(require a/b/c/no-contract)`. In a regular module, though, it would suffice to write `(require a/b/c)` and doing so would import contracted identifiers.

7.4 Unsafe: Beware

Racket provides a number of unsafe operations that behave like their related, safe variants but only when given valid inputs. They differ in that they eschew checking for performance reasons and thus behave unpredictably on invalid inputs.

As one example, consider `fx+` and `unsafe-fx+`. When `fx+` is applied to a non-`fixnum?`, it raises an error. In contrast, when `unsafe-fx+` is applied to a non-`fixnum?`, it does not raise an error. Instead it either returns a strange result that may violate invariants of the run-time system and may cause later operations (such as printing out the value) to crash Racket itself.

Do not use unsafe operations in your programs unless you are writing software that builds proofs that the unsafe operations receive only valid inputs (e.g., a type system like Typed Racket) or you are building an abstraction that always inserts the right checks very close to the unsafe operation (e.g., a macro like `for`). And even in these situations, avoid unsafe operations unless you have done a careful performance analysis to be sure that the performance improvement outweighs the risk of using the unsafe operations.

8 Retiquette: Branch and Commit

This section is specifically for Racketeers who commit to the Racket code base.

Working with the bug database requires one critical work flow rule.

Working with the code base requires style rules for actions on the repository. Currently we are using Git and below are a few rules on how to act in this context.

8.1 Bugfix Workflow

Re-assign bug reports only after you can eliminate your own code as the source of a bug. The best way to accomplish this goal is to create a new example that re-creates the problem without involvement of your code. When you have such a code snippet, re-assign the code to the person responsible for the apparently buggy component and submit the code snippet as part of the justification.

8.2 Commit

New feature commit: Commit the new feature, its tests, and its documentations as you wish, but please push them together. However, do not commit states that don't run. (In Git, this means 'commit' and not just 'push'.)

Bug fix commit: When you fix a bug, make sure to commit (1) the code delta, (2) the new test case, and (3) the revised docs (if applicable) in one batch. If the creation of a single commit is too complex of if you wish to factor out one of the commits, please push all pieces at once. That way the code base is always in a state where code, tests, and documentation are in sync, and readers of commit messages can evaluate changes completely.

Style change commit: Submit changes to the style of a file separately from changes to its behavior (new features, bugs).

Write meaningful commit messages. The first line (say 72 chars) should provide a concise summary of the commit. If the message must be longer, edit the rest of the message in your text editor and leave a blank line between the summary line and the rest of the message, like this:

```
some quick description

more blah blah blah, with more
details about the actual change
```

The advantage of a blank line is that `git log` and other tools display the commit messages properly. If you prefer the `-m` command line flag over an editor, you can use several of them in a row.

The message for bug report fixes should contain “Close PR NNNNN” so that bug reports are automatically closed.

To avoid ‘merge commits’, update your repository with `git --rebase pull`.

8.3 No Commit “Bombs,” Please

On occasion, you will find that you are spending a significant amount of time working with someone else’s code. To avoid potentially painful merges, please (1) inform the author when you create the branch and (2) set the mail hook so that git sends a commit message to both you and the original author. Furthermore, you should test your changes on the actual code base. In some cases it is acceptable to delay such tests, e.g., when you will not know for a long time whether the performance implications allow a commit to the PLT repository.

9 Acknowledgment

The rules borrow from many sources, including numerous style guides that the first author re-read in preparation of writing this document. The first author also conducted focus sessions with the members of PLT at various sites, and these sessions produced many insights about coding style. Thank you to everyone for sitting through these sessions and for reading early drafts of the style guide. Eli Barzilay and Jacob Matthews took the time to write up their thoughts on various issues, and their help is much appreciated. Rouben Rostamian spotted some problems in an early draft of the report.

10 Todo List, Call for Contributions

1. Write a section on when macros, when functions.
2. Write a section on how to design test cases.
3. Write a section on how to check the stressability of your software.
4. Find and link to good/bad examples in the code base.