Racket Style Guide

Fall 2017

Contents

1 Introduction

All the Racket code you write in CS 17 must follow all the guidelines in this document.

2 Naming

The designers of Racket snubbed the styles of both the East and West. In Racket, the preferred style is to use dashes, or hyphens, to name identifiers, as in `my-procedure`.

3 Formatting

Closing Parentheses A line of Racket code should never start with a right parenthesis. You should only ever use right parentheses in the middle or at the end of a line of Racket code. This convention makes Racket code more concise without sacrificing readability.

These are examples of good Racket style:

```
(define (f x) (+ 17 (* 18 (/ x 19))))
```

```
(define (f x)
  (+ 17 (* 18 (/ x 19))))
```

These are examples of *bad* Racket style:

```
(define (f x) (+ 17 (* 18 (/ x 19
))))

(define (f x) (+ 17 (* 18 (/ x 19
   )
  )
 )
)
```

Note that this convention is different from the stylistic conventions of many other programming languages, including those you will use in CS 18.

Square Brackets To make `cond` expressions easier to read, use square brackets around each clause. Square brackets more clearly delineate the clauses. Square brackets are particularly useful when debugging code with many parentheses.

This is okay:

```
(cond
  ((> x 0) (sqrt x))
  ((= x 0) 0)
  ((< x 0) (make-error "BLURGH!")))
```

But this is better:

```
(cond
  [(> x 0) (sqrt x)]
  [(= x 0) 0]
  [(< x 0) (make-error "BLURGH!")])
```

Line Breaks It can be difficult to read a single line of code with lots of parentheses and nested clauses. For example, try reading this convoluted line of code:

```
(cond [(> x 0) (sqrt x)] [(= x 0) 0] [(< x 0) (make-error "BLURGH!")])
```

To fix this, you should add line breaks like this:

```
(cond
  [(> x 0) (sqrt x)]
  [(= x 0) 0]
  [(< x 0) (make-error "BLURGH!")])
```

Indentation DrRacket indents your code for you, so most of the time you should be all set. But if your code ever gets mangled, you can always use the `Racket | Reindent All` menu entry. Indeed, your code's indentation should always reflect the running of this command.

Alignment It can happen that procedure names are too long, or that they take too many arguments, for an entire procedure application to fit on one line. In such cases, there are some acceptable and some unacceptable alignments.

This is acceptable:

```
(procedure-that-takes-four-arguments (if (zero? num) 15 16)
                                     "cats"
                                     "lichen"
                                     true)
```

Observe that the first argument is on the same line as the procedure name, and all subsequent arguments are on a new line, aligned with the first argument.

These are *not* acceptable:

```
(procedure-that-takes-four-arguments
 (if (zero? num) 15 16) "cats" "lichen" true)

(procedure-that-takes-four-arguments (if (zero? num) 15 16)
 "cats"
 "lichen"
 true)
```

But you don't really have to worry too much about the rules of alignment, because DrRacket will align your arguments for you automatically.

Spacing Although excess white space does not change the functionality of a program, it can impact its readability.

Here is an example good Racket style:

```
(baz (foo bar) qux)
```

Here are two examples of *bad* Racket style:

```
(baz(foo bar)qux)
(baz ( foo bar ) qux)
```

This code is less readable, particularly to programmers experienced with LiSP-like programming languages. By abiding by the community's convention, you ultimately make things easier for everyone, yourself included (even if the preferred style goes against your personal aesthetic).

Predicates By convention, predicate names in Racket end with a question mark; after all, they are essentially questions. For example, these two predicates are built in to Racket:

1. `zero?`, which returns **`true`** if the input number is zero
2. `empty?`, which returns **`true`** if the input list is empty

When you define your own predicates, you should follow this same convention. That is, you should append to the name of your predicate a question mark.

In fact, we have a little secret. The predicate `succ?` is not built in to Racket. It is only part of CS 17 Racket. But you wouldn't know that, since we followed the proper naming convention.

4 Equality

Racket has a variety of built-in procedures used to check whether or not two data are equal.

- (= 17 18) or (= **`true`** **`false`**)
  Works on numbers and booleans.
- (`string=?` `"foo""bar"`)
  Works on strings.
- (`equal?` (`list` 1 2 3)15)
  Works on everything.

Although `equal?` can be used in all of the above examples, it's best to use = or `string=?`. First of all, these procedures more precisely specify to the human reader what the code is checking. But more importantly, these procedures can detect errors in your code. For example, (`string=?` 17 `"seventeen"`) will fail, thereby alerting you to the presence of an error in your code. But (`equal?` 17 `"seventeen"`) will return false and carry on evaluating your code as if all were well with the world.

You should use `equal?` only when writing a polymorphic procedure, such as `member?`.

When checking equality of compound data, you should write your own equality procedure. For example:

```
(define (posn=? posn1 posn2)
  (and (= (posn-x posn1) (posn-x posn2))
       (= (posn-y posn1) (posn-y posn2))))

(posn=? (make-posn 17 18) (make-posn (+ 15 2) (+ 16 2)))
=> true
```

(Why does `posn=?` end with a question mark?)

5 Conditionals

Many expressions can be expressed in a logically equivalent fashion using either **cond** or **if**. But **cond** is almost always preferred to **if**.

5.1 Prefer Cond to If

If you are working with mixed data, you must use **cond**, and you must follow the structure of the data definition. For example:

```
;; (datum list)
;; – empty
;; – (cons datum (datum list))
(cond
  [(empty? alon) "The input list is empty"]
  [(cons? alon) "The input list is not empty"])

;; shape
;; – circle
;; – triangle
;; – rectangle
(cond
  [(circle? alon) "The input is a circle"]
  [(triangle? alon) "The input is a triangle"]
  [(rectangle? alon) "The input is a rectangle"])
```

But even when you are not working with mixed data, **cond** is still preferred, for the simple reason that **cond** expressions warn you when your cases are not exhaustive:

```
(define (foo x)
  (cond
    [(> x 0) (sqrt x)]
    [(= x 0) 0]))

(foo -1)
=> cond: all question results were false
```

In contrast, **if** requires you to specify a value in both cases, so you run the risk of possibly supplying a bogus value, where you really would have liked to signal an error instead. This bogus value will let your program continue running as if nothing had gone wrong, potentially wreaking havoc somewhere farther along in its execution, which will only make it that much harder to detect the actual source of the error.

Furthermore, **cond** is visually more appealing. All questions are lined up nicely for simultaneous consideration. This is not the case for **if** expressions, which are nested for "readability":

```
(define (bar x)
  (if (> x 0)
    (sqrt x)
    (if (= x 0)
        0
```

```
          x)))

(bar -1)
=> -1 ;; bogus value
```

The only time that **if** might be preferred to **cond** is when the data are not mixed *and* the decision is binary. For example:

```
(if (= x 17)
    (+ x x)
    (* x x))
```

can be used in place of:

```
(cond
  [(= x 17) (+ x x)]
  [else (* x x)])
```

But even here, **cond** expressions are more easily extensible than **if** expressions. Suppose we want to augment our code with a third clause. The logic is immediate within the **cond** expression:

```
(cond
  [(= x 15) (- x x)]
  [(= x 17) (+ x x)]
  [else (* x x)])
```

Finally, only use an **else** clause within a **cond** expression when you are certain that the questions above **else** cover `all` other cases of interest. By adding an **else** clause, you are telling Racket that it is okay to turn off error-checking. You are assuming the burden of error-checking yourself!

5.2 Concise Conditionals and Results

You should never write something like (`equal?` `my-boolean` **true**). Instead, just use `my-boolean`.

Similarly, do not use a **cond** or an **if** expression to first evaluate a predicate, and then return true or false. For example, (= x 17) is equivalent to and should replace both

```
(if (= x 17)
    true
    false)
```

and

```
(cond
  [(= x 17) true]
  [else false])
```

In the next example, the two possible values of the **if** expression are nearly identical, so writing them out twice in full is redundant.

```
(if (zero? num)
    (procedure-that-takes-lots-of-arguments 15 "cats" "lichen" true)
    (procedure-that-takes-lots-of-arguments 16 "cats" "lichen" true))
```

This code can be rewritten more concisely like this:

```
(procedure-that-takes-lots-of-arguments
  (if (zero? num) 15 16)
  "cats"
  "lichen"
  true)
```

5.3 To Nest or Not to Nest?

Sometimes, you will need to test multiple conditions simultaneously. For example, you may want to do one thing when two lists are empty, something else when just one is empty, and something else entirely when neither is empty.

Here are two acceptable ways of structuring combinations of conditionals:

```
(define (check-empty list1 list2)
  (cond
    [(and (empty? list1) (empty? list2)) "both lists are empty"]
    [(and (empty? list1) (cons? list2)) "list one is empty"]
    [(and (cons? list1) (empty? list2)) "list two is empty"]
    [(and (cons? list1) (cons? list2)) "both lists are non-empty"]))
```

```
(define (check-empty list1 list2)
  (cond
    [(empty? list1)
      (cond
        [(empty? list2) "both lists are empty"]
        [(cons? list2) "list one is empty"])]
    [(cons? list1)
      (cond
        [(empty? list2) "list two is empty"]
        [(cons? list2) "both lists are non-empty"])]))
```

Each of these two styles emphasizes a slightly different way of organizing and thinking about the possible cases. Generally speaking, either is acceptable, but certain problems may lend themselves towards more readable code using one structure rather than the other.

Just as **cond** is preferred to **if** when coding up a single test, **cond** is again preferred to **if** when nesting tests. For example, the following is considered bad style:

```
(define (check-empty list1 list2)
  (cond
    [(empty? list1)
      (if (empty? list2) "both lists are empty" "list one is empty")]
    [(cons? list1)
      (if (empty? list2) "list two is empty" "both lists are non-empty")])
        )
```

6 Verbosity

Simplify *if* expressions There are a number of equivalent ways to express the same conditional logic. In almost all cases, shorter expressions are preferred:

Verbose	Concise
(**if** expr **true false**)	expr
(**if** expr expr **false**)	expr
(**if** expr **false true**)	(not expr)
(**if** (not expr)x y)	(**if** expr y x)
(**if** x **true** y)	(**or** x y)
(**if** x y **false**)	(**and** x y)
(**if** x **false** y)	(**and** (not x)y)
(**if** x y **true**)	(**or** (not x)y)

When an **if** expression is used for argument selection, it can be embedded within a procedure application to improve readability, as follows:

```
;; Duplication of (f a b ..) application
(if c (f a b x) (f a b y))

;; Can be eliminated by embedding the if
(f a b (if c x y))
```

Don't rewrap procedures When applying a procedure to another procedure, don't rewrap the procedure if it already does what you need it to do. Here are two examples:

```
;; VERBOSE
(lambda (x y) (cons x y))

;; CONCISE
cons

;; VERBOSE
(define (select x y) (filter x y))

;; CONCISE
(define select filter)
```

7 Conclusion

Coding style is, of course, a matter of style, and some circumstances better lend themselves to one style than to another. Sometimes this distinction is a matter of opinion; if you are ever unclear about whether there is a preferred style, ask a TA.

And always remember: you are not writing code only for yourself. Code is something you will likely share with your peers and others. So it behooves you to follow the language conventions of

whatever language you find yourself programming in. Otherwise, you will confuse not only yourself, but anyone you share your code with.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17document by filling out the anonymous feedback form: `http://cs.brown.edu/courses/cs017/feedback`.