A Style Guide for EECS345 Programming Language Concepts

January 14, 2019

Contents

1 Introduction **1**

2 Comments **2**

3 Whitespace **2**

4 Indentation **3**

5 Line Lengths **7**

6 Naming **7**

7 Other Conventions. **8**

1 Introduction

As with all code that involves others, style is important for communication. With proper style, what was once hard to decipher becomes easy to read. Numerous organizations have created there own style guides in order to achieve the goal of consistent and readable code.

Style includes indentation, comments, white-space, line length, function names, and variable/value names.

The first section of this document will outline what style one's Racket code should be in. Later sections (when added) will cover Haskell, C, OCaml, or other languages that may be covered in the course.

1. On Other Sources

   Examples from lectures and handouts, are often condensed into fewer lines for illustrative purposes. You should not imitate their condensed styles for assigns, because you do not have the same space restrictions.

2 Comments

In Racket, everything after a ; on a line is ignored and is considered a comment. Always follow a comment by a space for better readability

- For section headings use four or three semicolons
- For full line comments, use two semicolons
- For inline comments, use one semicolon

```
;;;; TEST SECTION, this is an example of a section headers

;; This is an example of a full line comment


;; Here is some value documentation: letter-grade......
(define letter-grade 'A) ; Hopefully your grade by the end of the course!
```

Note that one can add extra characters around their headings to make their sections more visible

```
;;;; ************************************************
;;;;   Jane Doe (JED665)
;;;;   EECS 345 Spring 2019
;;;;   Assignment 4
;;;; ************************************************



;;;; Helper Functions-----------------------------------------------
```

3 Whitespace

Whitespace is very important for the readability of Racket code. It is customary to put a space between arguments and other arguments and the function itself. Also when there is a long chain of closed parenthesis, no spaces between them nor newlines should be put. Note that for many beginners this takes getting used to, after time the parens will start to disapper and indentation will lead one's intuition.

```
;;;; Whitespace Examples

;; Good
```

```
(f (g x) (h y z) q)

;; Bad: there is no space between f and (,
(f(g x))

;; Bad: there is no space between ) and (
(f (g x)(h y))

;; Bad: there is no space x and (h y)
(f (g x(h y)))

;; Bad: there are extra spaces between ()'s
( f ( g x ( h y ) ) )

;; Bad: the )'s are on their own line, let them hug each other!
( f (g x (h z
         )
    )
)
```

4 Indentation

Indentation is quite a different beast in LISPS compared to C/Algol like languages. The `Dr.Racket` editor will indent you correctly for all these situations, so make sure to follow the defaults given when one hits enter in the editing environment.

1. Arguments should align with eachother

   ```
   ;; Notice how (time 562 1231) aligns with (square x)
   ;; very-long-name also algins with (square x)
   (function-name (square x)
                  (times 562 1231)
                  (very-long-name 562
                                  1231))
   ```

2. When dealing with arguments on the next line 1 or 2 spaces are used

   ```
   ;; Good: uses 1 space
   (long-function-name
    (more-code x y)
    more-arguments)

   ;; Good: uses 2 spaces
   (long-function-name
     (more-code x y)
   ```

```
  more-arguments)
```

3. When dealing with multiple arguments, one should put them on a single line unless the line becomes too long. When this happens, one should put each argument on their own line.

```
;;; from https://docs.racket-lang.org/style/Textual_Matters.html

;; 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)


;; This is good, as the 2 10's are related and short
#lang racket

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

4. Align code in `let`, `cond`, `match`, and anywhere where there is a clear dichotomy between one argument and the other. This rule only applies when the left side is relatively small, and the right side is only 1 line or so. Note this rule is optional, but greatly improves readability for the graders and oneself.

```
;; Good: this clearly shows the relation
;;       between the question (the predicate) and the answer
;;       Thus we can read each [] section as
;;       pred implies answer
```

```
(cond
  [(< foo 0)                            (negate-foo fooz)]
  [(> largest-int most-negative-value) (negate-ans fooz)]
  [else                                 (+ 23 123123)])

;; Good: here the 2ND predicate is super long,
;;       so aligning the answers would cause
;;       the answers to go over an acceptable
;;       character-line limit
(cond
  [(< foo 0)
   (negate-foo fooz)]
  [(super-super-super-duper-super-long-pred x)
   (negate-ans fooz)]
  [else (+ 23 123123)])

;; Good: arguments are aligned
;;      (note the body location is correct,
;;      and such exceptions are talked about in 5.)

(let ([x           5]
      [val         23]
      [line-length 110])
  (+ x val line-length))


;; Good: arguments are all aligned, and the matched expression on the left
;;       is used to imply the result on the right
(require racket/match)

(define m-value-int
  (lambda (xs)
    (let ((ops (hash '+ + '- - '* * '/ quotient '% remainder)))
      (match xs
        ((list a b op) ((hash-ref ops op) (m-value-int a) (m-value-int b)))
        ((list* _ _)   (error "unvalid operation"))
        (x             x)))))
```

5. Special forms such as `let` and `define` have their own indentation rules for maximum readability. Please rely on your editor for deciding indentation, as it will always get these right!

   For `cond`'s it's good style to use [] for each question answer section. and for `let`'s it's customary to use them for each value binding section

```
;; Good: the lambda after f is spaced with =2 spaces=
```

```
;;       The body of lambda is spaced 2 spaces as well
;;       Note that the argument to lambda should be
;;       on the same line
(define f
  (lambda (x)
    (+ x 2)))

;; Good: this is short hand for the first and is acceptable
(define (f x)
  (+ x 2))

;; Bad: This is poor style as the body of the lambda
;;      is on the same line as the arguments
;;      (this is a problem in lisp code due to many
;;      lispers shortening lambda to λ in their code)
(define f
  (lambda (x y) (- x
                   y)))

;; this code ends up looking like this!
(define f
  (λ (x y) (- x
                   y)))

;; Bad: place each question on their own line
(cond [(< bar 0) 'foo] [else 'bar])

;; Bad: place each true/false branch on their own line
(if (pred x) x
    3)

;; Good
(if (pred x)
    x
    3)

;; Also fine for short if's only
(if (pred x) x 3)

;; Good: note that DR.Racket will format the code
;;       like this when one presses enter
;;       Notice the [] in [x 2]
(let ([x 2]
      [y 3])
  (+ x y))
```

```
;; Bad: Overuse of []
(let [[x 2]
      [y 3]]
  (+ x y))

;; Good: proper use of []
(cond
  [(< foo 0)                            (negate-foo fooz)]
  [(> largest-int most-negative-value) (negate-ans fooz)]
  [else                                 (+ 23 123123)])
```

5 Line Lengths

Try not to let the line get longer than `110` characters long. You do not want your lines to look too horizontal nor vertical. Sometimes this is hard to achieve, so use your best judgment

```
;; Good
(define sum-square
  (lambda (x y)
     (+ (* x x)
        (* y y))))

;; Bad: Too horizontal
(define sum-square (lambda (x y) (+ (* x x) (* y y))))

;; Bad: Too vertical
(define
  sum-square
  (lambda (x
           y)
    (+
      (*
        x
        x)
      (*
        y
        y))))
```

6 Naming

In Lisps we use the following convention `word1-word2`, so in the section above it's `sum-square` not `sumSquare` nor `sum_square`. Also names should be proportional to their scope and lengths. so `(lambda (x) (* x 2))` is perfectly fine, as this illustrates the meaning better than if we were to write `(lambda`

`(number) (* number 2))` as it is the form that matters, not the value itself. However if we had a longer example, then x would be too cryptic.

```
#lang racket

;; Here the name list->hash-bag is descriptive, an alternative acceptable
;; name would be list->hash-immutable-bag
;; Here the name xs is used for any arbitrary list, as it's a list of x's.
;; Since this list has no other property, we need not give it a more descriptive
;; name like, odd-list or even-list or whatever the implicit constraint may be

;; the name increment-bag, properly describes that this function
;; takes a bag and a key and increments that key returning a new bag
;; the names bag and key are small yet descriptive of what they do
(define list->hash-bag
  (lambda (xs)
    (let ([increment-bag
           (lambda (key bag)
             (hash-update bag key add1 0))]) ; 0 is the default value
      (foldl update-bag (make-immutable-hash) xs))))


;; Here is an example output of the function!
;; (list-to-hash-bag '(1 2 3 4 5 1 2 3))
;; '#hash((1 . 2) (2 . 2) (3 . 2) (4 . 1) (5 . 1))
```

7 Other Conventions.

- prefer `let`'s to inner `define`'s

  ```
  ;; Bad: although considered good Racket style
  ;;      this is considered poor style in other
  ;;      Lisps, including scheme
  (define f
    (define x 2)
    (+ x 2))

  ;; Good: A universal good style amongst all lisps
  (define f
    (let ([x 2])
      (+ x 2)))
  ```

- Don't nest `if`'s, instead use a cond.

  ```
  ;; Bad
  (if x
  ```

```
    y
    (if z
        f
        y))

;; Good
(cond [x    y]
      [z    f]
      [else y])

;; Fine: sometimes this can't be formed into a cond nicely
(if x
    (if z
        y
        1)
    a)

;; This is fine, as turning this into a cond means
;; that one would have to do g and b first before the check
;; wasting computation or writing something untrue!
(if x
    y
    (let ([g 3]
          [b 4])
      (if z
          g
          b)))
```