Computer Science @ The University of Waterloo

CS 115 & CS 135
Course Specific Style Guide

Last Updated: 2019.12.12

1 Introduction

The code you submit for assignments, as with all code you write, can be made more readable and useful by paying attention to style. This includes the placement of comments, choice of variable and function names, whitespace and indentation. None of these things affect the execution of a program, but they affect its readability and extensibility. As in writing English prose, the goal is communication, and you need to think of the needs of the reader. This is especially important when the reader is assigning you a grade.

1.1 A Warning on Examples From Other Sources

The examples in the presentation slides, handouts and tutorials/labs are often condensed to fit them into a few lines; you should not imitate their condensed style for assignments, because you do not have the same space restrictions.

1.2 A Warning on `DrRacket`

`DrRacket(.rkt)` files can store rich content that can include images, extended formatting, comment boxes, and special symbols. Using this rich content may make your assignment unmarkable. Unfortunately, some of the content in the Interactions window may be "rich" (*e.g.,* rational numbers), and so you should avoid copy-and-pasting from your interactions window into your definitions window. In addition, code that appears in a `.pdf` document (*e.g.,* presentation slides and as-

signments) may contain hidden or unusual symbols, so you should not copy-and-paste from those sources either.

For your assignments, save in **plain text** format. In `DrRacket` you can ensure your `.rkt` file is saved using plain text format by using the menu items: `File > Save Other > Save Definitions as Text`.

Never include Comment Boxes or images in your submissions. Do not copy-and-paste from the Interactions window or from `.pdf` documents into the Definitions window, or your assignment may become unmarkable.

2 Assignment Formatting

2.1 Header

Your file should start with a header to identify yourself, the term, the assignment and the problem. There is no specifically required format, but it should be clear and assist the reader. The following is a good example in Racket.

```
;;
;; ************************************************
;;    Rick Sanchez (12345678)
;;    CS 135 Fall 2019
;;    Assignment 03, Problem 4
;; ************************************************
;;
```

2.2 Comments

There are two types of comments in Racket:

- `;;` starts a full-line comment
- `;` starts a comment at the end of a line

Comments should be used for documentation purposes and to explain *why* code does what it does.

Use in-line comments sparingly. If you are using standard design recipes and templates, and following the rest of the guidelines here, you should not need many additional comments. Any such comment can either be put on its own line, or tacked onto the end of a line of code, providing it fits.

2.3 Identifiers

2.3.1 Naming Functions, Parameters and Constants

Identifiers (names) should be meaningful, but not awkwardly long nor cryptically short. The first priority should be to choose a meaningful name. Names like `salary` or `remittance` would be appropriate in a program that calculates taxes.

Sometimes a function will consume values that don't have a meaning attached, for example, a function that calculates the maximum of two numbers. In that case, chose names that reflect the structure of the data. That is, `n` is for numbers, `i` for integers, `lst` for a list or `lon` for a list of numbers, and so on. Names that are proper nouns like Newton should always be capitalized. Otherwise, use the Racket convention:

In Racket, identifiers use lower-case letters and hyphens, eg.
`top-bracket-amount`

2.3.2 Constants

Constants should be used to improve your code in the following ways:

- To improve the readability of your code by avoiding "magic" numbers. For example, if you have code dealing with tax rates like Ontario's HST, you might want a constant such as `hst` or `taxes` and have this value set to 0.13.

```
(define hst 0.13) ; HST Rate in Ontario effective July 1, 2010
```

- To improve flexibility and allow easier updating of special values. If the value of `hst` changes, it is much easier to make one change to the definition of the constant than to search through an entire program for the value `0.13`. When this value is found, it may not be obvious if it refers to the tax rate, or whether it serves a different purpose and should not be changed.
- To define values for testing and examples. As values used in testing and examples become more complicated (*e.g.,* lists, structures, lists of structures), it can be very helpful to define named constants to be used in multiple tests and examples.

2.4 Whitespace, Indentation and Layout

Whitespace should be used to make it easier to read code. This means not having too much or too little whitespace in your code. In what follows, we give an example of what our whitespacing will look like in our model solutions though you can deviate from this slightly as long as the code is still readable.

- Insert two consecutive blank lines between functions or "function blocks", which include the documentation (*e.g.,* design recipe) if applicable for the function.
- Insert one blank line before the function header and one after the function body.
- Blank lines may appear in function blocks but do not use more than one consecutive blank line within a function block.

In `DrRacket`, if you have many functions and/or large function blocks, you may want to insert a row of symbols (such as the $*$'s used in the file header above) to separate your functions.

If the question asks you to write more than one function, the file should contain them in the order specified by the assignment. Helper functions are placed either all above the assignment function(s) they are helping or all below the functions they are helping. (CS135 prefers all below.) Remember that the goal is to make it easier for the reader to determine where your functions are located.

Indentation plays a big part in readability. It is used to indicate level of nesting, to align related subexpressions (*e.g.,* arguments of a function), and to make keywords more visible. `DrRacket`'s built-in editor will help with these. If you start an expression `(my-fn` and then hit enter or return, the next line will automatically be indented a few spaces. However, `DrRacket` will never break up a line for you, and you can override its indentation simply by putting in more spaces or erasing them. `DrRacket` also provides a menu item for reindenting a selected block of code (`Racket > Reindent`) and even a keyboard shortcut reindenting an entire file (`Ctrl+I` in Windows; `Cmd-I` on a Mac).

When to start a new line (hit enter or return) is a matter of judgment. Try not to let your lines get longer than about 70 characters, and definitely no longer than 80 characters. You do not want your code to look too horizontal nor too vertical.

In `DrRacket` there is a setting you can change to show you when you've exceeded the common style guide's recommended line length:
`Edit -> Preferences -> Editing tab -> General Editing sub-tab -> Maximum character width guide.`
Occasionally, your examples and tests may exceed the line length guidelines when there is no obvious place to break the line. You should still strive to keep the lines within the suggested limits though.

> Style marks may be deducted if you exceed 80 characters on any line of your assignment.

Indentation examples:

```
;; GOOD
(define (distance posn1 posn2)
  (sqrt (+ (sqr (- (posn-x posn2) (posn-x posn1)))
           (sqr (- (posn-y posn2) (posn-y posn1))))))

;; GOOD
(define (distance posn1 posn2)
  (sqrt (+ (sqr (- (posn-x posn2)
                   (posn-x posn1)))
           (sqr (- (posn-y posn2)
                   (posn-y posn1))))))

;; BAD (too horizontal)
(define (distance posn1 posn2)
  (sqrt (+ (sqr (- (posn-x posn2) (posn-x posn1))) (sqr (- (posn-y posn2) (posn-y posn1))))))
```

```
;; BAD (too vertical)
(define (distance posn1 posn2)
  (sqrt
   (+
    (sqr
     (-
      (posn-x posn2)
      (posn-x posn1)))
    (sqr
     (-
      (posn-y posn2)
      (posn-y posn1))))))
```

If indentation is used properly to indicate level of nesting, then closing parentheses can just be added on the same line as appropriate. You will see this throughout the textbook and presentations and in the GOOD example above. Some styles for other programming languages expect that you place closing brackets on separate lines that are vertically lined up with the opening brackets. However, in Racket this tends to negatively effect the readability of the code and is considered "poor style".

For `cond`itional expressions, placing the keyword `cond` on a line by itself, and aligning not only the questions but the answers as well can improve readability (provided that they are short). However, this recommendation is not strictly required. Each question must appear on a separate line, and long questions/answers should be placed on separate lines. Marks may be deducted if a `cond` is too dense or confusing.

`cond` examples:

```
;; Looks good for short cond questions and answers
(cond
   [(< bar 0) (neg-foo bar)]
   [else      (foo bar)])


;; Acceptable
(cond [(< bar 0) (neg-foo n)]
      [else (foo n)])

;; BAD: (place each question on a separate line)
(cond [(< bar 0) (neg-foo n)] [else (foo n)])


;; GOOD: Place long questions/answers on separate lines
(cond
```

```
[(and (>= bar 0) (<= bar 100))
 (really-long-function-or-expression (/ bar 100))]
[else
 (some-other-really-long-function-or-expression bar)])
```

2.5 Summary

- Use two comment symbols for full-line comments and use one comment symbol for in-line comments, and use them sparingly inside the body of functions.
- Provide a file header for your assignments.
- Make it clear where function blocks begin and end
- Order your functions appropriately.
- Avoid overly horizontal or vertical code layout.
- Use reasonable line lengths.
- Choose meaningful identifier names and follow our naming conventions.
- Avoid use of "magic numbers".

Style marks may be deducted if you have poor headers, identifier names, whitespace, indentation or layout.

3 The Design Recipe: Functions

Warning! The format of the Design Recipe is different than the one used in the HtDP textbook and sometimes changes between course offerings. This style guide will be used for assessment (*i.e.,* assignments and exams).

We hope you will use the design recipe as part of the process of working out your solutions to assignment questions. If you hand in only code, even if it works perfectly, you will earn only a fraction of the marks available. Elements of the

design recipe help us to understand your code. But the big reason to use the design recipe is that the process will help you write working code more quickly.

Not everything in this section will make sense on first reading, and some of the following details will only appear at the end of the course. We suggest that you review it before each assignment.

3.1 Design Recipe Sample

In your final version, the content including and after `;<---` should be removed and is included here to highlight the different components of the design recipe.

```
;; (sum-of-squares p1 p2) produces the sum of    ; <--- Purpose
;;   the squares of p1 and p2
;; Examples:                                     ; <--- Examples
(check-expect (sum-of-squares 3 4) 25)
(check-expect (sum-of-squares 0 2.5) 6.25)

;; sum-of-squares: Num Num -> Num                ; <--- Contract
(define (sum-of-squares p1 p2)                   ; <--- Function Header
  (+ (* p1 p1)                                   ; <--- Function Body
     (* p2 p2)))

;; Tests:                                        ; <--- Tests
(check-expect (sum-of-squares 0 0) 0)
(check-expect (sum-of-squares -2 7) 53)
```

3.2 Purpose

```
;; (sum-of-squares p1 p2) produces the sum of
;;   the squares of p1 and p2
```

The purpose statement has two parts: an illustration of how the function is applied, and a brief description of what the function does. The description does not have to explain how the computation is done; the code itself addresses that question.

- The purpose starts with an example of how the function is applied, which uses the same parameter names used in the function header.
- Do not write the word “purpose”.

- The description must include the names of the parameters in the purpose to make it clear what they mean and how they relate to what the function does (choosing meaningful parameter names helps also). Do not include parameter types and requirements in your purpose statement — the contract already contains that information.
- If the description requires more than one line, "indent" the next line of the purpose 2 or 3 spaces.
- If you find the purpose of one of your helper functions is too long or too complicated, you might want to reconsider your approach by using a different helper function or perhaps using more than one helper function.
- It's often helpful to use the word 'produces' to highlight what the function creates as an output.

3.3 Examples

```
;; Examples:
(check-expect (sum-of-squares 3 4) 25)
(check-expect (sum-of-squares 0 2.5) 6.25)
```

The examples should be chosen to illustrate "typical" uses of the function and to illuminate some of the difficulties to be faced in writing it. Examples should cover each case described in the data definition for the type consumed by the function. The examples do not have to cover all the cases that the code addresses; that is the job of the tests, which are designed after the code is written. It is very useful to write your examples before you start writing your code. These examples will help you to organize your thoughts about what *exactly* you expect your function to do. You might be surprised by how much of a difference this makes.

For recursive data, your examples **must** include *each* base case and at least one recursive case. Examples should cover edge cases whenever possible (for example, empty lists).

In `DrRacket`, examples serve both as examples and as tests for your function.

3.4 Contract

```
;; sum-of-squares: Num Num -> Num
```

The contract contains the name of the function, the types of the arguments it consumes, and the type of the value it produces. The contract is analogous to functions defined mathematically that map from a domain to a co-domain (or more loosely to the range of the function).

3.4.1 Types in Racket

The following is a list of types that are valid in Racket:

`Num`	Any number, including rational numbers
`Int`	Integers: `...-2, -1, 0, 1, 2...`
`Nat`	Natural Numbers (non-negative Integers): `0, 1, 2...`
`Bool`	Boolean values (`true` and `false`)
`Sym`	Symbol (*e.g.,* `'Red`, `'Rock`)
`Str`	String (*e.g.,* `"Hello There"`, `"a string"`)
`Char`	Character (*e.g.,* `#\a`, `#\A`, `#\newline`)
`Any`	Any value is acceptable
`(anyof T1 T2...)`	Mixed data types. For example, `(anyof Int Str)` can be either an `Int` or a `Str`. It may also include specific values such as `false` (*e.g.,* a search function's failure indicator).
`(listof T)`	A list of **arbitrary length** with elements of type T, where T can be any valid type For example: `(listof Any)`, `(listof Int)`, `(listof (anyof Int Str))`.
`(list T1 T2...)`	A list of **fixed length** with elements of type T1, T2, etc. For example: `(list Int Str)` always has two elements: an `Int` (first) and a `Str` (second).
`User-Defined`	For structures and user-defined types (see section below). Capitalize your user defined types.
`X, Y, ...`	Matching types to indicate parameters must be of the same type. For example, in the following contract, the `X` can be any type, but all of the `X`'s must be the same type: `my-fn: X (listof X) -> X`

3.4.2 Additional Contract Requirements

If there are important constraints on the parameters that are not fully described in the contract, add an additional **requires** section after the contract. For example, this can be used to indicate a `Int` must be in a specific range, a `Str` must be of a specific length, or that a `(listof ...)` cannot be empty. Single requirements can be on one line. Multiple requirements should start on a separate line and each line should be indented several spaces. For an example in Racket:

```
;; quot: Nat Nat -> Nat
;; Requires:
;;    n1 >= 0
;;    n2 > 0
(define (quot n1 n2) ...)
```

3.5 Tests

It is extremely important that you submit your code and check your basic tests to make sure the format of your submission is readable by the autograder!

```
;; Tests:
(check-expect (sum-of-squares 0 0) 0)
(check-expect (sum-of-squares -2 7) 53)
```

Make sure that your tests are actually testing every part of the code. For example, if a conditional expression has three possible outcomes, you have tests that check each of the possible outcomes. Furthermore, your tests should be directed: each one should aim at a particular case, or section of code. Some people write tests that use a large amount of data; this is not necessarily the best idea, because if they fail, it is difficult to figure out why. Others write lots of tests, but have several tests in a row that do essentially the same thing. It's not a question of quantity, but of quality. You should design a small, comprehensive test suite.

In `DrRacket`, examples serve both as examples and as tests for your function. Notice that in other languages, this is not necessarily true!

Never figure out the answers to your tests by running your own code. Work out the correct answers independently (*e.g.,* by hand).

3.5.1 Testing Tips

Parameter type	Consider trying these values
`Num`	positive, negative, 0, non-integer values, specific boundaries, small and/or large values
`Int`	positive, negative, 0
`Bool`	`true`, `false`
`Str`	empty string (`""`), length 1, length > 1, extra whitespace, different character types, etc.
`(anyof ...)`	values for each possible type
`(listof T)`	empty, length 1, length > 1, duplicate values in list, special situations, etc.
`User-Defined`	special values for each field (structures), and for each possibility (mixed types)

3.6 Additional Design Recipe Considerations

3.6.1 Helper Functions

Do not use the word "helper" in your function name: use a descriptive function name. Depending on which course you are taking, all design recipe elements might not be necessary for helper functions. Purpose and contracts however should always be included. You are not required to provide tests for your helper functions (but often it is a very good idea). In the past, we have seen students avoid writing helper functions because they did not want to provide documentation for them. This is a bad habit that we strongly discourage.

Writing good helper functions is an essential skill in software development and having to write a purpose and contract should not discourage you from writing a helper function. Marks may be deducted if you are not using helper functions when appropriate. Helper functions should be placed before the required func-

tion(s) in your submission.

> Functions we ask you to write require a full design recipe.
>
> Helper functions need a purpose, contract, and at least one example.

3.6.2 Mutually Recursive Functions

If we ask you to write two mutually recursive functions, only one of the functions requires tests. In the following example, the tests are included with the function `is-odd?`.

```
;; (is-even? x) produces true if x is even, and false otherwise
;; Example:
(check-expect (is-even? 2) true)

;; is-even?: Nat -> Bool
(define (is-even? x)
  (cond [(= x 0) true]
        [else (is-odd? (sub1 x))]))


;; (is-odd? x) produces true is x is odd and false otherwise
;; Examples:
(check-expect (is-odd? 0) false)
(check-expect (is-odd? 45) true)

;; is-odd?: Nat -> Bool
(define (is-odd? x)
  (cond [(= x 0) false]
        [else (is-even? (sub1 x))]))

;; Tests
(check-expect (is-odd? 1) true)
(check-expect (is-odd? 10) false)
```

3.6.3 Wrapper Functions

When using wrapper (primary) functions, only one of the functions requires tests. If the required function is the wrapper function, then include the examples and tests with it. Otherwise, use your judgement to choose the function where ex-

amples and tests seem most appropriate. In the following example, the tests and examples are included with the wrapper function `remove-from`.

```
;; (remove-from-list loc c)  produces a new list, like loc, but with
;;     all occurrences of c removed
;; remove-from-list: (listof Char) Char -> (listof Char)
;; Examples and tests: see wrapper function remove-from

(define (remove-from-list loc c)
  (cond [(empty? loc) empty]
        [(char=? (first loc) c) (remove-from-list (rest loc) c)]
        [else (cons (first loc) (remove-from-list (rest loc) c))]))


;; (remove-from s) produces a new string like s, but with all
;;    A and a characters removed
;; remove-from: Str -> Str
;; Examples:
(check-expect (remove-from "" #\X) "")
(check-expect (remove-from "ababA" #\a) "bbA")
(check-expect (remove-from "Waterloo" #\x) "Waterloo")

(define (remove-from s c)
  (list->string (remove-from-list (string->list s) c)))

;; Tests:
(check-expect (remove-from "X" #\X) "")
(check-expect (remove-from "A" #\y) "A")
(check-expect (remove-from "Waterloo" #\o) "Waterl")
(check-expect (remove-from "OOOOO" #\O) "")
```

3.6.4 `Local` Helper Functions

For functions defined with a `local` block, no tests or examples are necessary, however include the other design recipe elements as illustrated in the following code. Add a blank line after each `local` function definition. Add a blank line after each block of `local` constant definitions, as you would for non-`local` constant definitions.

```
;; (remove-short los len) produces the list of strings in los
;;    which are longer than len.
;; remove-short: (listof Str) Nat -> (listof Str)
;; Examples:
(check-expect (remove-short empty 4) empty)
(check-expect (remove-short (list "ab" "1234" "hello" "bye") 3)
              (list "1234" "hello"))

(define (remove-short los len)
```

```
  (local
    [;; (long? s) produces true if s has more
     ;; long?: Str -> Bool
     (define (long? s)
       (> (string-length s) len))]

    (filter long? los)))

;; Tests
(check-expect (remove-short (list "abc") 4) empty)
(check-expect (remove-short (list "abcdef") 2) (list "abcdef"))
(check-expect (remove-short (list "ab" "1234" "hello" "bye") 1)
              (list "ab" "1234" "hello" "bye"))
(check-expect (remove-short (list "ab" "1234" "hello" "bye") 20)
              empty)
```

3.7 Summary

- Purposes should be brief and describe everything your function is doing.
- Parameters names must be in the purpose and explained.
- Contracts should use appropriate types
- Requirements should be used to explain restrictions on input not able to be captured by a contract.
- Examples for code should cover edge/base cases.
- Tests should be a small comprehensive suite covering both edge and typical cases.
- Helper functions do not need examples and tests but require all other design recipe elements.

4 User-Defined Types and Templates

A user-defined type can be a structure (defined with `define-struct`) or simply a descriptive definition.

- With the exception of the Racket built-in types (*e.g.,* `Posn`), every user-

defined type used in your assignment must be defined at the top of your file (unless an assignment instructs you otherwise).

- User-Defined types that appear in contracts and other type definitions should be capitalized (use `My-Type`, not `my-type`).

4.1 Structures

When you define a structure, it should be followed by a type definition, which specifies the type for each of the fields. For example:

```
(define-struct date (year month day))
;; A Date is a (make-date Nat Nat Nat)
```

If there are any additional requirements on the fields not specified in the type definition, a **requires** section can be added. For example:

```
(define-struct date (year month day))
;; A Date is a (make-date Nat Nat Nat)
;; Requires:
;;   fields correspond to a valid Gregorian Calendar date
;;   year >= 1900
;;   1 <= month <= 12
;;   1 <= day <= 31
```

4.2 Descriptive Definitions

A descriptive definition can be used to define a new user-defined type to improve the readability of contracts and other type definitions

```
;; A StudentID is a Nat
;; Requires: the value is between 1000000 and 99999999

;; A Direction is an (anyof 'up 'down 'left 'right)
```

A descriptive definition can also be a mixed type, with more than one possible type:

```
;; A CampusID is one of:
;; * a StudentID
;; * a StaffID
;; * a FacultyID
;; * 'guest
```

4.3 Templates

For any non-trivial user-defined type, there is a corresponding template for a function that consumes the new type. Since a template is a general framework for a function that consumes the new type, we also **always write a contract as part of the template**. For each new data definition, writing a template is recommended, but not required unless otherwise specified.

For structures, the template for a function that consumes the structure would have placeholders for each field:

```
(define-struct date (year month day))
;; A Date is a (make-date Nat Nat Nat)

;; date-template: Date -> Any
(define (date-template d)
  ( ... (date-year d) ...
    ... (date-month d) ...
    ... (date-day d) ...))
```

For mixed user-defined types, there is a corresponding `cond` question for each possible type:

```
;; A CampusID is one of:
;; * a StudentID
;; * a StaffID
;; * a FacultyID
;; * 'guest

;; campusid-template: CampusID -> Any
(define (campusid-template cid)
  (cond
    [(studentid? cid)       ...]
    [(staffid? cid)         ...]
    [(facultyid? cid)       ...]
    [(symbol=? 'guest cid)  ...]))
```

As you combine user-defined types, their templates can also be combined. See the course handouts for more examples.

4.4 Summary

- Structures should be followed by a type definition specifying the types of each field.
- Descriptive definitions can be used to define a new user-defined type to improve readability.
- We also always write a contract as part of the template.

5 A Sample Submission

Problem: Write a Racket function `earlier?` that consumes two times and will produce `true` if the first time occurs earlier in the day than the second time, and `false` otherwise.

Note how the named constants makes the examples and testing easier, and how the introduction of the `time->seconds` helper function makes the implementation of `earlier?` much more straightforward.

```
;;
;; *****************************************
;;   Rick Sanchez (12345678)
;;   CS 135 Fall 2017
;;   Assignment 03, Problem 1
;; *****************************************
;;

(define-struct time (hour minute second))
;; A Time is a (make-time Nat Nat Nat)
;; Requires: 0 <= hour < 24
;;           0 <= minute, second < 60


;; time-template: Time -> Any
(define (time-template t)
  ( ... (time-hour t) ...
    ... (time-minute t) ...
    ... (time-second t) ... ))
```

```
;; Useful converters
(define seconds-per-minute 60)
(define minutes-per-hour 60)
(define seconds-per-hour (* seconds-per-minute minutes-per-hour))


;; Useful constants for examples and testing
(define midnight (make-time 0 0 0))
(define just-before-midnight (make-time 23 59 59))
(define noon (make-time 12 0 0))
(define eight-thirty (make-time 8 30 0))
(define eight-thirty-and-one (make-time 8 30 1))



;; (time->seconds t) Produces the number of seconds since midnight
;;   for the time t
;; time->seconds: Time -> Nat
;; Example:
(check-expect (time->seconds just-before-midnight) 86399)

(define (time->seconds t)
  (+ (* seconds-per-hour (time-hour t))
     (* seconds-per-minute (time-minute t))
     (time-second t)))


;; (earlier? time1 time2) Determines if time1 occurs before time2
;; earlier?: Time Time -> Bool
;; Examples:
(check-expect (earlier? noon just-before-midnight) true)
(check-expect (earlier? just-before-midnight noon) false)

(define (earlier? time1 time2)
  (< (time->seconds time1) (time->seconds time2)))

;; Tests:
(check-expect (earlier? midnight eight-thirty) true)
(check-expect (earlier? eight-thirty midnight) false)
(check-expect (earlier? eight-thirty eight-thirty-and-one) true)
(check-expect (earlier? eight-thirty-and-one eight-thirty) false)
(check-expect (earlier? eight-thirty-and-one eight-thirty-and-one) false)
```