CS 135 Style Guide

Computer Science @ The University of Waterloo
Last Updated: 2023-09-13

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.

This document is organized linearly to match the topics covered. The guide is cumulative; new guidelines are added to the old and only rarely replace previous guidelines. You're responsible for the style guide up to and including the current lecture module's material.

1.1 Some warnings

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.

The design recipe at Waterloo has evolved beyond what is presented in the course textbook *"How to Design Programs"* by Felleisen, Flatt, Fieldler and Krishnamurthi, MIT Press 2003 [HtDP]. Their presentation of the design recipe should not be used directly. However, the spirit of HtDP style remains and the examples in the textbook are good, particularly those illustrating the design recipe, such as Figure 3 in Section 2.5.

`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 assignments) 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`.

2 M02: The Basics

2.1 Comments

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

Racket comments begin with a semi-colon (`;`). By convention, full-line comments begin with two semi-colons:

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

Use in-line comments sparingly. If you are following this style guide, 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.

2.2 File 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.

```
;;
;; ***************************************************
;;    Rick Sanchez (12345678)
;;    CS 135 Fall 2020
;;    Assignment 03, Problem 4
;; ***************************************************
;;
```

2.3 Line length and indentation

Overly long lines are hard to read. That's why newspaper columns are generally short. Keep your line lengths no longer than 102 characters. (102 is the default maximum in `DrRacket`.)

`DrRacket` has a setting to show when you've exceeded the common style guide's recommended line length:

`Edit` → `Preferences` → `Editing tab` → `General Editing sub-tab` → `Maximum character width guide`.

(On MacOS you may need to use the `DrRacket` menu to see the `Preferences` menu.)

If your lines are getting too long, they should be broken at sensible places so that the code is readable. Code is hard to read if it's either too horizontal (lines are too long) or too vertical (lines are so short that there are needlessly many).

Indentation (appropriate spacing at the beginning of each line) 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).

Here are examples showing both good and bad indentation and line lengths:

```
;; 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".

2.4 Identifiers

Identifiers (names of functions, parameters, constants) 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.

The importance of a meaningful name increases with the amount of code in which it can appear. A parameter to a short function can be shorter and more cryptic than a constant appearing throughout a file, for example.

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, choose names that reflect the structure of the data. That is, `n` for numbers, `i` for integers, `lst` for a list or `lon` for a list of numbers, and so on.

Use the Racket convention of lowercase letters and hyphens such as `top-tax-bracket`. Although Racket allows the use of Unicode characters, it is best to use alphanumeric characters in your names.

If a name is given as part of a problem, be sure to use that name. It may be vital for our marking scripts to evaluate your function correctly or simply be a name that makes marking easier for the humans involved.

Pay attention to conventions. Conversion functions often have an arrow in the name (*e.g.,* `fahrenheit->celsius`). In course materials we sometimes typeset arrows as → but in your code you should simply use `->` (hyphen followed by greater-than).

2.5 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 "testing constants" to be used in multiple tests and examples.

Zero and one rarely need to be made a constant.

"Magic numbers" do not need to be defined as a constant if they are used only once

in a conversion function. For example, in the following neither 32 nor 5/9 needs to be defined as a constant.

```
;; (f->c degrees-f) produces the temperature in degrees Celsius
;;    given degrees-f degrees Farenheit

;; Examples:
(check-within (f->c -34) -36.7 0.1)
(check-within (f->c 80) 26.7 0.1)

;; f->c: Num -> Num
(define (f->c degrees-f)
  (* (- degrees-f 32) 5/9))
```

The rule of thumb is that if a magic number has a name or distinct meaning, it should be a constant. If it is tied to the specific formula, it can remain a number.

If sets of constants are related, group them together, and separate blocks of related constants with a line break. You may also include a comment to indicate the purpose of these constants:

```
;; Tax rates
(define hst 0.13)
(define gst 0.05)

;; Income levels
(define poor-income 10000)
(define rich-income 1000000)
```

When naming constants, be careful that the name of the constant is not tied to its value. If changing the value of the constant would make its name incorrect, then you have chosen a bad name.

For example, the following is a bad name for `hst` because if the HST value ever changes then the name becomes incorrect.

```
(define hst_13_percent 0.13)
```

2.6 Function comments

Each function should be preceded by comments that describe the function's purpose. Because these are whole-line comments, begin with two semi-colons.

Leave two blank lines before the comment to separate this function from those that precede it.

An example:

```
;; (twice n) produces a value that is twice as large as n
(define (twice n) (* 2 n))
```

```
;; (thrice n) produces a value that is three times as large as n.
(define (thrice n) (* 3 n))
```

We will have *much* more to say about documenting functions in the following sections (that correspond to lecture modules 04 and later).

2.7 Ordering your assignment

You should have an identifying header at the top (see above) of your file.

If the assignment asks for multiple problems in the same file, put them in the same order as the assignment, separated by a visual indicator. For example,

```
;;
;; Problem 3
;;
```

Constants that apply to a specific problem should be grouped with that problem. Constants that apply to more than one problem should be grouped near the top of your file, below the header.

2.8 Summary

1. Start your file with an identifying header.
2. Place problems in the order given with a visual separator.
3. Limit lines to 80-102 characters.
4. Use `DrRacket`'s indentation tool. Split lines so they are neither too vertical nor too horizontal.
5. Choose meaningful identifier names, particularly when they span a larger amount of code.
6. Name identifiers in Racket style (lowercase; dashes).
7. Use constants for "magic numbers" (except if used only once in a conversion function).

3 M03: Simple Data

3.1 Identifiers

Pay attention to naming conventions. Functions that produce a Boolean often have names ending in a question mark (`even?`, `can-vote?`).

3.2 Comparison Operators

You will almost never use `boolean=?` to compare boolean values. Instead you can use the boolean directly. For example, instead of

```
(cond
  [(boolean=? x true) ...]
  [...])
```

You can write:

```
(cond
  [x ...]
  [...])
```

Similarly you can replace `(boolean=? x false)` with `(not x)`. If you do not know whether `x` is a boolean value, you might use `(false? x)`, which produces `false` for non-Boolean values. However, you should use this sparingly – only when the contract of your function allows for `x` to be both `Bool` and something else.

Racket has an equality comparison called `eq?` but we never use it in CS 135 and you shouldn't either.

Be careful to use the most specific comparison operator the contract of your function allows. In particular, be careful about using `equal?` if a more specific comparison operator (such as `char=?` or plain old `=`) will suffice.

3.3 Tests

As you write your test suite, you may want to think about whether the following values deserve specific tests.

Parameter type	Consider trying these values
`Nat`	0, small values, large values, specific boundaries
`Int`	similar to `Nat`, but also negative values
`Num`	similar to `Int`, but also, non-integer values
`Bool`	`true`, `false`
`Str`	empty string (`""`), length 1, length > 1, extra whitespace, different character types, etc.

`Num` values can be *exact* (representable as a rational number) or *inexact*. If your function produces inexact values then you should test for inexact values; if it does not produce inexact numbers when provided exact arguments, such tests are optional and you may assume all `Num` arguments are exact.

The test suite should include tests for each question/answer pair in a **`cond`** expression, including the **`else`** statement. Also ensure that complex Boolean expressions are adequately tested.

3.4 Indentation

For **`cond`**itional expressions each question must appear on a separate line. If the answer clause is short it may be placed on the same line; otherwise, it should be placed on a separate line. The **`cond`** may be on its own line or share the line with the first question/answer pair.

Marks may be deducted if a **cond** is too dense or confusing.

cond examples:

```
;; GOOD: short question/answer pairs can be one line each
(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)])
```

It is considered poor style for the **else** clause to be a single **cond**. For example,

```
(define (count-char/list ch loc)
  (cond
    [(empty? loc) 0]
    [else (cond [(char=? ch (first loc))
                 (+ 1 (count-char/list ch (rest loc)))]
                [else (count-char/list ch (rest loc))])]))
```

can be trivially transformed into the simpler and more readable

```
(define (count-char/list ch loc)
  (cond
    [(empty? loc) 0]
    [(char=? ch (first loc)) (+ 1 (count-char/list ch (rest loc)))]
    [else (count-char/list ch (rest loc))]))
```

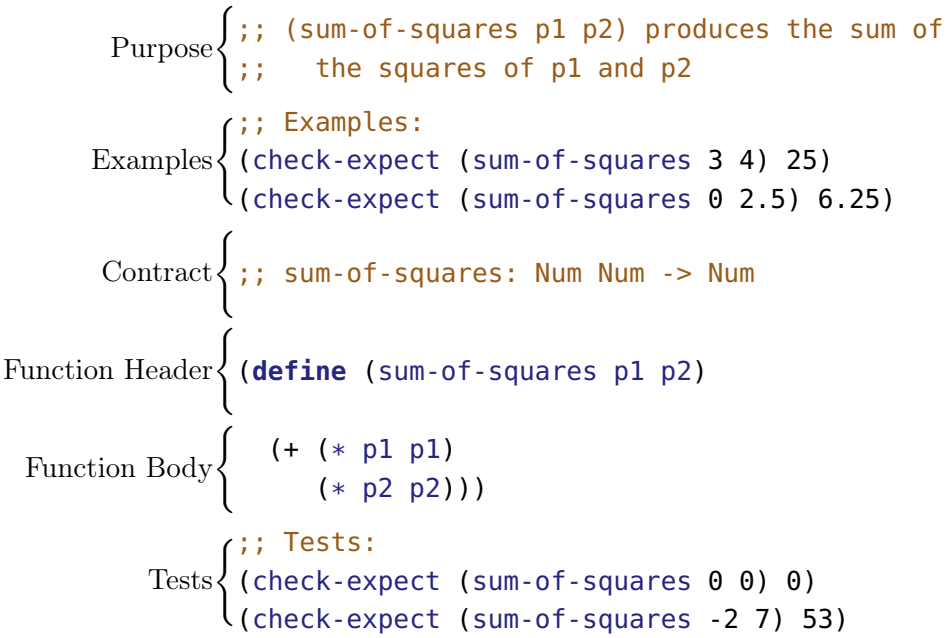
4 M04: Introductory Design Recipe

The design recipe is first and foremost a process that helps you develop working, trusted code. It leaves behind some artifacts that are useful for future readers (yourself, markers, profs) to understand and trust your code.

This section of the Style Guide describes the artifacts. But just putting those artifacts into your code after you've written it misses the point. Refer to lectures for the process. Then touch up the artifacts left behind to match this style.

Course staff will be reluctant to assist you unless the appropriate artifacts from the Design Recipe process are present.

4.1 Design Recipe Sample



4.2 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. This will include parentheses: `(sum-of-squares p1 p2)`
- Do not write the word "purpose".
- The description must make clear the purpose of each parameter. Sometimes that can be done only with well-chosen names but usually the parameter names are referenced in the function description to relate the parameter to what the function does.

  Do not include parameter types and requirements in your purpose statement — the contract already contains that information.
- It's often helpful to use the word 'produces' to highlight what the function creates as an output.
- If the description requires more than one line, indent the next line of the purpose 2 or 3 spaces.

- A function should have a single purpose. If you find the purpose description of one of your functions is too long or too complicated, you might:
  - Describe the function at a higher level of abstraction
  - Reorganize the function using the problematic function, perhaps to use additional helper functions

4.3 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. If the purpose indicates that some values are handled differently (perhaps zero or negative numbers or empty strings), include them in your examples.

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.

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.

In `DrRacket`, examples are written using `check-expect` (and much less frequently, `check-within` and `check-error`) so they can serve both as examples and as executable tests for your function.

The arguments used in your examples need not be big or hard to calculate. In fact, choose values so that you can easily calculate the expected result yourself. Just don't hide any complexities by consistently choosing arguments that trivialize the function's operation (for example, always multiplying by zero).

4.4 Contract

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

The contract contains the name of the function, a colon, the types of the arguments it consumes, an arrow, 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). Unlike purpose statements, the function is not enclosed in parentheses.

Use `->` to indicate the arrow. Do not use `=>`.

Contracts sometimes get longer than the recommended line length. If they do then split them up over two lines. Breaking the line just after the arrow usually works. If you break a contract over a second line indent subsequent lines as you do for purpose statements.

The following is a list of types that are valid in Racket (at this point in the course; we'll be adding to this list):

Parameter Type	Sample values
`Any`	Any value is acceptable
`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`, `'earth`)
`Str`	String (*e.g.,* `"Hello There"`, `"a string"`)
`Char`	Character (*e.g.,* `#\a`, `#\A`, `#\newline`)

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 or a `Str` must be of a specific length.

Single requirements can be on the same line as `Requires:`. If there are only a few requirements that can be specified in less than the line length guidelines, you may separate them with commas:

```
;; neeblu: Nat Nat -> Nat
;; Requires: n1 > 0, 8 <= n2 < 100
(define (neeblu n1 n2) ...)
```

Otherwise multiple requirements should start on a separate line and each line should be indented several spaces. For example:

```
;; neeblu: Nat Nat -> Nat
;; Requires:
;;    n1 > 0
;;    8 <= n2 < 100
(define (neeblu n1 n2) ...)
```

Requirements are often mathematical in nature and can be easily expressed mathematically. But they don't need to be. For example:

```
;; neeyel: Str Nat -> Str
;; Requires:  s is non-empty, n < length(s)
(define (neeyel s n) ...)
```

The guiding principle is that your requirements should be easy for humans to read and understand.

4.5 Tests

```
;; 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, include tests that check each of the

possible outcomes.

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 comprehensive test suite that is no larger than necessary.

Examples also serve as tests for your function.

If you have many tests for your function you may break them up into blocks and organize them thematically. Separate each block with one line, and comment the theme of the block.

The submission process in CS135 includes some basic tests to ensure, for example, that your code uses the expected function names. If it doesn't, it is unmarkable by our testing software and you will lose many marks. Be sure to submit early enough to correct mistakes the basic tests catch.

4.6 Helper Functions

Do not use the word "helper" in your function name: use a descriptive function name.

Helper functions only require a purpose, a contract and at least one illustrative example. You are not required to provide additional tests for your helper functions (but often it is a very good idea).

In the past, we have seen some 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.

Place helper functions after the function they help so that the most interesting function (the one requested on the assignment) appears first. Exceptions to this ordering:

- An assignment that specifies functions in a different order (e.g. write two helper functions and then the main function that uses them) should place the functions in the order specified.
- Functions used in multiple assignment problems contained in the same file should be placed towards the beginning of the file, grouped, and with a header to indicate they are commonly used helpers.
- A helper function that is used to define constants must be placed before it is used.

The guiding principle is to organize your code so that it is easy for others (*e.g.,* the TAs marking your assignment) to read and understand.

Be sure to keep design recipe components together. Do not define new helper functions or constants in the middle of the design recipe for your main function; instead define

them before (for constants) or after (for helpers) all design recipe components for the function you are writing.

5 M05: Data definitions, templates, lists

5.1 Data definitions and templates

Data definitions introduce a new type and should be included early in the file, certainly before they are used, and above blocks of constants or helper functions.

Templates should be placed immediately after the data definition they consume, unless specified otherwise in the assignment. Including templates for data definitions is optional unless we ask for them on the assignment.

It is permissible (encouraged!) to define data definitions purely to increase the readability and understandability of your code. Such user-defined types should have capitalized names. Those names may be used in contracts. For example:

```
;; A NumGrade is an Int in the range [0, 100] representing a grade in
;; a course.

;; cumulative-average: (listof NumGrade) -> NumGrade
(define (cumulative-average grades) ... )
```

Because the limits on a `NumGrade` are included in the data definition, the contract for `cumulative-average` need not specify that grades are between 0 and 100.

5.2 Examples

For recursive data, your examples **must** include *each* base case and at least one recursive case. In all likelihood, this will include each case described in the data definition for the type consumed by the function.

5.3 Contracts

In addition to our previous types (`Any`, `Num`, `Int`, `Nat`, `Bool`, `Str`, `Char`, and `Sym`), we now add:

Parameter Type	Sample values
`(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)`.
`(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).
`(anyof T1 T2...)`	Mixed data types. For example, `(anyof Int Str)` can be either an `Int` or a `Str`. The types `T1, T2, ...` can also be specific values such as `false` (*e.g.,* a search function's failure indicator).
`UserDefined`	For types that you define (and `UserDefined` is replaced by your chosen name).

5.4 Wrapper Functions

When using wrapper 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 judgment to choose the function where examples and tests seem most appropriate. The other function is considered to be a helper and requires one example.

Helper functions that are wrapped by another function sometimes are named in terms of the main function, with a `/` character indicating the specialization. In the example below, `remove-from/char` is a helper function for `remove-from` that consumes a `(listof Char)`. This naming style is optional but you are encouraged to use it for your functions.

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

```
;; (remove-from s c) produces a new string like s, instances of
;;    the character c removed
;; Examples:
(check-expect (remove-from "" #\X) "")
(check-expect (remove-from "ababA" #\a) "bbA")
(check-expect (remove-from "Waterloo" #\x) "Waterloo")

;; remove-from: Str Char -> Str
(define (remove-from s c)
  (list->string (remove-from/char (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) "")


;; (remove-from/char loc c)  produces a new list, like loc, but with all
;;    occurrences of c removed
;; Example:
(check-expect (remove-from/char (string->list "Camera") #\a)
              (string->list "Cmer"))

;; remove-from/char: (listof Char) Char -> (listof Char)
(define (remove-from/char loc c)
  (cond [(empty? loc) empty]
        [(char=? (first loc) c) (remove-from/char (rest loc) c)]
        [else (cons (first loc) (remove-from/char (rest loc) c))]))
```

6 M07: More Lists

6.1 Fixed-length List Accessor Functions

In some cases we model data with fixed-length lists:

```
;; A PlayerScore is a (list Str Nat)
```

In these cases it can be helpful to use short helper functions to identify the elements of the fixed-length lists:

```
(define (name player-score) (first player-score))
(define (points player-score) (second player-score))
```

Such accessor functions do not require additional design recipe components, but you may include them if you wish.

7 M09: Structures

7.1 Data Definitions

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
```

The data definition's name should be the same as the structure name but using Camel-Case. It should be used in applicable contracts.

The meaning or purpose of each field should be made clear. In the `date` example the field names together with the requirements are sufficient. If additional explanatory text is necessary, one suggested approach is to follow the types with a "where" clause: "A Foo is a (make-foo Nat Int) where bar is ...". Another option is to elaborate on the field meanings as part of the requires section.

7.2 Templates

We always recommend writing a template for each new data definition, but it is not required unless explicitly stated in the assignment. When including a template it is placed immediately after the data definition.

For structures, the template for a function that consumes the structure will have place-holders for each field in the same order as they are listed in the **`define-struct`**.

```
(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 lecture materials for more examples.

8 M11: Mutual Recursion

8.1 Mutually Recursive Functions

If we ask you to write mutually recursive functions, only the function we ask you to write (the "main function") requires the full design recipe. Other functions participating in the mutual recursion need a purpose statement, contract, and at least one example.

9 M13: Local

For functions defined with a **local** block, no tests or examples are necessary[1], 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.
;; Examples:
```

[1] It isn't possible to write a `check-expect` as part of a **local**.

```
(check-expect (remove-short empty 4) empty)
(check-expect (remove-short (list "ab" "1234" "hello" "bye") 3)
              (list "1234" "hello"))

;; remove-short: (listof Str) Nat -> (listof Str)
(define (remove-short los len)
  (local
    [;; (long? s) produces true if s has more than len characters
     ;; 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)
```

10 M14, M15: Functions that produce functions

lambda produces functions. It is often used with higher order functions, and to bind identifiers to values or functions. The design recipe components required depend on the situation.

10.1 Testing functions that produce functions

When testing a function that produces a function, it can be helpful to define a testing constant. For example, we might want to test the function `make-adder` from the notes:

```
;; Tests
(define add-3 (make-adder 3))
(check-expect (add-3 4) 7)
(check-expect (add-3 0) 3)
```

In this situation, it is okay to define the constant `add-3` for testing inside the design recipe for `make-adder`, even though doing so interrupts the design recipe.

Note that you cannot define such a constant before the function definition of `make-adder`, so you cannot use this technique for examples. Instead, create functions and test them within the same `check-expect`:

```
;; Examples
(check-expect ((make-adder 3) 4) 7)
(check-expect ((make-adder 3) 0) 3)
```

10.2 Anonymous functions for higher order functionss

When used to generate anonymous functions for use with higher order functions, no design recipe components for the **lambda** expression are required.

```
;; Design recipe for passing function goes here

(define (passing grade-list)
  (filter (lambda (x) (>= x pass-threshold)) grade-list))
```

Considering the contract of these **lambda** functions can still be helpful as your expressions get more complicated, however.

10.3 Binding identifiers

If a **lambda** expression binds an identifier to a value, then that expression defines a constant, which does not require design recipe components. For example, if `step` and `initial-value` are in scope, then `increment` is a constant:

```
(define increment ((lambda (x) (+ x step)) initial-value))
```

However, if a lambda expression produces a function, that function requires design recipe components as if it was defined as a function normally:

```
;; (double x) produces the double of x

;; Example:
(check-expect (double 6.3) 12.6)

;; double: Num -> Num
(define double (lambda (x) (+ x x)))
```

Sometimes higher order functions with **lambda** expressions are used to produce constant values:

```
(define smalls (filter (lambda (x) (< x (first lon)) lon)))
```

These constants follow the usual style rules, and do not require additional design recipe components.

11 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))


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

;; earlier?: Time Time -> Bool
(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)
```

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

;; time->seconds: Time -> Nat
(define (time->seconds t)
  (+ (* seconds-per-hour (time-hour t))
     (* seconds-per-minute (time-minute t))
     (time-second t)))
```