Style and Submission Guide

1 Assignment Style Guidelines

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, whitespace, indentation, and choice of variable and function names. None of these things affect the execution of a program, but they affect its readability and extensibility. There isn't a single strict set of rules that you must follow; just 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 mark. A portion of the marks for each assignment will be allocated for readability.

1.1 General Guidelines

The examples in the presentation slides and handouts are often condensed to fit them into a few lines; you should not imitate their style for assignments, because you don't have the same space restrictions. The examples in the textbook are more appropriate, particularly those illustrating variations on the design recipe, such as Figure 3 in Section 2.5, Figure 11 in Section 6.5, and Figure 17 in Section 17.2. At the very least, a function should come with contract, purpose, examples, definition, and tests. After you have learned about them, and where it is appropriate, you should add data definitions. See the section below titled "The Design Recipe" for further tips.

You should prepare one file for each question in an assignment, containing all code and documentation. The file for question 3 of Assignment 8 should be called `a08q3.rkt`, and all the files for Assignment 8 should be in the folder/directory `a08`. It is recommended that you manage your lab questions in a similar manner; the file for question 2 of Lab 6 should be called `l06q2.rkt`, and all the files for Lab 6 should be in the folder/directory `l06`. 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 should be put with the assignment function they are helping, but whether they are placed before or after is a judgement you will have to make depending on the situation. In working with several functions in one file, you will find it useful to use the Comment Out With Semicolons and Uncomment items in DrRacket's Racket menu to temporarily block out successful tests for helper functions. Note: never include Comment Boxes or images in your submissions, as they will be rendered unmarkable.

The file for a given question should start with a file header, like the one below. The purpose of the file header is to assist the reader.

```
;;
;; ************************************************
;;
;; CS 115 Assignment 13, Question 3
;; Ursula Franklin
;; (Put a short description of the question here)
;;
;; ************************************************
;;
```

1.2 Block Comments and In-Line Comments

Anything after a semicolon on a line is treated as a comment and ignored by DrRacket. Functions are usually preceded by a block comment, which for your assignments will contain the contract, purpose, and examples. Block comments should be indicated by double semicolons at the start of a line, followed by a space.

```
;; distance: posn posn → num
;; Produces the Euclidean distance between posn1 and posn2.
;; Example: (distance (make-posn 1 1) (make-posn 4 5)) ⇒ 5
```

You may or may not choose to put a blank line between the block comment and the function header (probably for longer function headers it is appropriate), but there should be a blank line between the end of a function and the start of the next block comment. In your early submissions, you shouldn't need to put blank lines in the middle of functions; later, when we start using local definitions, they may be appropriate.

Use "in-line" comments sparingly in the middle of functions; if you are using standard design recipes and templates, and following the rest of the guidelines here, you shouldn't 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.

1.3 Indentation and Layout

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-fun* 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 and another for reindenting an entire file.

When to hit enter or return is a matter of judgement. At the very least, don't let your lines get longer than about 70 characters. You don't want your code to look too horizontal, or too vertical.

```
                                    ;; don't do this, either
                                   (define
                                     (squaresum x y)
                                       (+
                                         (*
                                           x
                                           x)                ;; this is all right
                                         (*                  (define (squaresum x y)
 ;; don't do this                        y                    (+ (* x x)
(define (squaresum x y) (+ (* x x) (* y y)))  y))                 (* y y))
```

If indentation is used properly to indicate level of nesting, then closing parentheses can just be added on the same line as appropriate, and you will see this throughout the textbook and presentations. Some styles for other programming languages expect you to put closing parentheses or braces by themselves on a separate line lined up vertically with their corresponding open parenthesis or brace, but in Scheme this tends to affect readability.

If you find that your indentation is causing your lines to go over 70 characters, consider breaking out some subexpression into a helper function, but do this logically rather than cutting out an arbitrary chunk.

For conditional expressions, you should place the keyword **cond** on a line by itself, and align the questions. It may be appropriate to align the answers, if they are short enough to do so. A long answer should be placed on a separate, indented line.

```
                                   (cond
                                     [(zero? n) 0]
(cond                                [(= n 1)   1]
    [(null? lon)  empty]             [else
    [else         (rest lon)])         (* n (fact (- n 1)))])
```

1.4 Variable and Function Names

Try to choose names for variables and functions that are descriptive, not so short as to be cryptic, but not so long as to be awkward. It is a Scheme convention to use lower-case letters and hyphens, as in the identifier *top-bracket-amount*. (DrRacket distinguishes upper-case and lower-case letters by default, but not all Scheme implementations do.) In other languages, one might write this as `TopBracketAmount` or `top_bracket_amount`, but try to avoid these styles in this course.

You will notice some conventions in naming functions: predicates that return a Boolean value usually end in a question mark (e.g. *zero?*), and functions that do conversion use a hyphen and greater-than sign to make a right arrow (e.g. *string->number*). This second convention is also used in contracts to separate what a function consumes from what it produces. A "double right arrow" (which we use to indicate the result of partially or fully evaluating an expression) can be made with an equal sign and a greater-than sign. In the typesetting in this document and in the presentations, we have fonts with single and double arrow symbols, which you can see in our contracts and comments, but DrRacket isn't equipped with them.

1.5 Summary

- Use block comments to separate functions.
- Use the design recipe.
- Comment sparingly inside body of functions.
- Indent to indicate level of nesting and align related subexpressions.
- Avoid overly horizontal or vertical code layout.
- Use reasonable line lengths.
- Align questions and answers in conditional expressions where possible.
- Choose meaningful identifier names and follow the naming conventions used in the textbook and in lecture.
- Do not include Comment Boxes or images.

1.6 The Design Recipe

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. Correctness is not the only aspect of code that we are evaluating, and our emphasis is on the process as well as the outcome of programming.

The design recipe comes in several variants, depending on the form of the code being developed. As we learn about new language features and ways of using them, we also discuss how the design recipe is adapted. Consequently, not everything in this section will make sense on first reading. We suggest that you review it before each assignment.

The design recipe for a function starts with contract, purpose, and examples. You should develop these before you write the code for the function.

;; distance: posn posn $\rightarrow$ num
;; Produces the Euclidean distance between posn1 and posn2.
;; Examples:
;; (distance (make-posn 1 2) (make-posn 1 2)) $\Rightarrow$ 0
;; (distance (make-posn 1 2) (make-posn 1 4)) $\Rightarrow$ 2
;; (distance (make-posn 1 1) (make-posn 4 5)) $\Rightarrow$ 5

(**define** (*distance posn1 posn2*)
  ...)

Since a contract is a comment, errors in contracts will not be caught by DrRacket. As this is not the case with statically typed languages such as Java, it is good practice to write them correctly. The contract consists of the name of the function, a colon, the types of the arguments consumed, an

arrow, and the type of the value produced. The types of the arguments should be in the same order as the arguments listed in the function header. It is also acceptable to use a specific value instead of a type, such as a (*union string false*) when a function produces either a string or the value *false*.

For full marks, be sure to follow the instructions here; we are being more strict about types than the textbook is. Types can be either built-in or user-defined. Built-in types include *num*, *image*, *boolean*, *char*, *string*, *symbol*, and *posn*. We also use *int* for integers and *nat* for natural numbers (non-negative integers). Any type that has been defined with a data definition can be used in a contract; examples from lecture include *mminfo*, *card*, and *potatohead*. For mixed data, use the notation (*union* ...) and for lists, use the notation (*listof* ...). Where the book uses types like *list-of-numbers*, we are being more careful and using the type (*listof num*). We use square brackets after a type to specify restrictions, such as *int*[*>=6*] for integers greater than or equal to six and (*listof num*)[*nonempty*] for a nonempty list of numbers. For more than one restriction, we use a comma to separate the restrictions, such as *int*[*>=0,<=5*] for the integers from 0 to 5. The length of a string or list can be indicated with the word *len*, as in *string*[*len>=1*] (a string of length at least 1) and (*listof int*)[*0<=len,len<=4*] (a list of integers of length from 0 to 4).

The purpose is a brief one- or two-line description of what the function should compute. Note that it does not have to be a description of how to compute it; the code shows how. Mention the names of the parameters in the purpose, so as to make it clear what they mean (choosing meaningful parameter names helps also).

The examples should be chosen to illustrate the uses of the function and to illuminate some of the difficulties to be faced in writing it. Many of the examples can be reused as tests (though it may not be necessary to include them all). The examples don't have to cover all the cases that the code examines; that is the job of the tests, which are designed after the code is written. For lengthy examples such as those using structures and lists, you may wish to define constants for use in examples and tests: this cuts down on typing, makes examples clearer, and helps ward off the temptation to cut answers from the Interactions window to paste in the Definitions window (see the note below).

Next come the function header and body of the function itself. You'd be surprised how many students have lost marks in the past because we asked for a function *my-fun* and they wrote a function *my-fn*. They failed all of our tests, of course, because they didn't provide the function we asked for.

To avoid this situation, use the provided "interface" files, such as the file `assninterface3.rkt` for Assignment 3. These contain the function headers of the functions asked for, and perhaps definitions of some structures. If you use these as a starting point, you are less likely to misspell key identifiers. A word of warning: if an assignment asks you to use a teachpack, do not copy definitions from the teachpack into your file. When your work is tested by our testing system using the teachpack, the tests will fail due to the definitions having been given twice (once in the teachpack and once in your file).

Finally, we have tests. You should make sure that the tests exercise every line of code, and furthermore, that the tests are 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.

Note: it is tempting to cut and paste from the Interactions window into the Definitions window. Don't. What the Interactions window produces is not plain text, and may interfere with automated marking of your work.

1.7 A Sample Submission

Here is the code from the second phone bill example done in class (Module 3).

```
;;
;; ************************************************
;;
;; CS 115 Assignment 92, Question 1
;; John A. MacDonald
;; (Cell phone bill calculations)
;;
;; ************************************************

;;
;; Defined constants
;;

;; Free limits
(define day-free 100)
(define eve-free 200)

;; Rates per minute
(define day-rate 1)
(define eve-rate .5)

;; charges-for: num num num → num
;; Produces charges computed for minutes, when the number
;; free minutes is freelimit and rate is the charge for
;; each minute that is not free.
;; Examples: (charges-for 101 100 5) ⇒ 5
;; (charges-for 99 100 34) ⇒ 0
(define (charges-for minutes freelimit rate)
  (max 0 (* (- minutes freelimit) rate)))
;; Tests for charges-for
;; Minutes above free limit
(check-expect (charges-for 101 100 5) 5)
;; Minutes below free limit
(check-expect (charges-for 99 100 34) 0)
```

```
;; cell-bill: num num → num
;; Produces cell phone bill for day and eve minutes.
;; Examples: (cell-bill 150 300) ⇒ 100
;; (cell-bill 500 150) ⇒ 400
(define (cell-bill day eve)
  (+
   (cond
     [(< day day-free) 0]
     [(>= day day-free) (* (- day day-free) day-rate)])
   (cond
     [(< eve eve-free) 0]
     [(>= eve eve-free) (* (- eve eve-free) eve-rate)])))
;; Tests for cell-bill
;; Day and evening below free limits
(check-expect (cell-bill 10 10) 0)
;; Day only below free limit
(check-expect (cell-bill 100 250) 25)
;; Evening only below free limit
(check-expect (cell-bill 200 100) 100)
;; Day and evening above free limits
(check-expect (cell-bill 101 202) 2)
```

1.8 Guidelines for Adaptations to the Design Recipe

Later on in the course, there are other components to the design recipe, such as data definitions and templates. We usually discuss these briefly in class, but the book goes into more detail. If you keep up with lecture attendance and reading, you will know how to adapt the design recipe for each new assignment.

In general, you should provide the complete design recipe for every function written in the course, whether it is a main function or a helper function. There are a few exceptions to this rule, detailed below:

Helper Functions from Lecture If you are using a helper function that was presented in lecture, you do not need to provide the complete design recipe. Note: if you are using a helper function that appeared as a lab question, the complete design recipe is still required.

Wrapper Functions for Strings For a function that consumes strings, if the main work is done by a helper function that consumes a list of characters, it is acceptable that your examples and tests be provided for the wrapper function only. That is, you can write your examples and tests using strings rather than lists of characters.

Locally-Defined Functions Every locally-defined function is a helper function. Although you should use examples and tests when developing your code, in your completed submission

you should provide only the contract, purpose, and definition of each locally-defined function. The contract and purpose should appear right above the function header, indented to be at the same level as the function header.

Mutually-Recursive Functions It is permissible to have a set of examples and tests that work for a pair or group of functions working together, rather than developing separate examples and tests for each component.

1.9 Sample Submission with Structures and Lists

Here is the code from the student example done in class (Module 5), showing how to incorporate data definitions for structures and lists. In assignments, you do not need to repeat data definitions that we provide to you. As in this example, templates will be incorporated into your final code. You do not need to provide a blank template in the comments.

```
;;
;; ************************************************
;;
;; CS 115 Assignment 302, Question 3
;; John A. MacDonald
;; (Student grade calculations)
;;
;; ************************************************

(define-struct student (name assts mid final))
;; A student is a structure (make-student n a m f) where n is a string and
;; a, m, and f are numbers.

(define-struct grade (name mark))
;; A grade is a structure (make-grade n m) where n is a string and m is
;; a number.

;; A list of students is either empty or (cons stud slist) where stud is a
;; student and slist is a list of students.

;; A list of grades is either empty or (cons gr glist) where gr is a grade and
;; glist is a list of grades.

;; Constants
(define assts-weight .20)
(define mid-weight .30)
(define final-weight .50)

;; Sample data for examples and tests
(define vw (make-student "Virginia Woolf" 100 100 100))
```

```
(define at (make-student "Alan Turing" 90 80 40))
(define an (make-student "Anonymous" 30 55 10))

;; final-grade: student → grade
;; Produces a grade from student record astudent.
;; Examples: (final-grade vw) ⇒
;; (make-grade ”Virginia Woolf” 100)
;; (final-grade an) ⇒ (make-grade ”Anonymous” 27.5)
(define (final-grade astudent)
  (make-grade
   (student-name astudent)
   (+ (* assts-weight (student-assts astudent))
      (* mid-weight (student-mid astudent))
      (* final-weight (student-final astudent)))))
;; Tests for final-grade
(check-expect (final-grade vw) (make-grade "Virginia Woolf" 100))
(check-expect (final-grade an) (make-grade "Anonymous" 27.5))

;; compute-grades: (listof student) → (listof grade)
;; Produces grade list from student list slist.
;; Examples: (compute-grades empty) ⇒ empty
;; (compute-grades (cons vw (cons at (con an empty) ⇒
;; (cons (make-grade ”Virginia Woolf” 100)
;; (cons (make-grade ”Alan Turing” 62)
;; (cons (make-grade ”Anonymous” 27.5 empty)))
(define (compute-grades slist)
  (cond
    [(empty? slist) empty]
    [else (cons (final-grade (first slist))
                (compute-grades (rest slist)))]))
;; Tests for compute-grades
(check-expect (compute-grades empty) empty)
(check-expect (compute-grades (cons vw (cons at (cons an empty))))
              (cons (make-grade "Virginia Woolf" 100)
                    (cons (make-grade "Alan Turing" 62)
                          (cons (make-grade "Anonymous" 27.5)
                          empty))))
```

2 Assignment Submission

Submit your assignments to the MarkUs server at `https://markus002.cs.uwaterloo.ca`. You will need to log in using your Waterloo userid and password. This is likely the same as your Quest userid and password.

MarkUs allows you to resubmit your files as many times as you want. You are strongly encouraged to submit as soon as you have something that you think is worth marks. Resubmit often thereafter. This has several benefits:

- It serves as a backup in case something goes wrong with your copy of your work. Just download it from MarkUs!
- You have *something* for us to mark, even if something bad happens that prevents you from submitting before the deadline (your computer crashes, the server crashes, you get sick, etc)
- If you have a really weird problem, course staff may look at your submission and be better able to help you.

3 Lab Style and Submission

Your lab questions give you opportunities to practice following the style and submission guidelines without having to worry about losing marks for making errors.

Like assignment questions, lab questions come with provided "interface" files for you to use.

Like assignment questions, lab questions can be submitted on MarkUs and then checked using the public test facility.

Follow the instructions given for assignment submission, using directory `l1` for Lab 1, `l2` for Lab 2, and so on.

Although we recommend completing lab questions during the week of the assigned lab (especially since assignment questions build on previous lab questions), there is no deadline for submitting your work nor for using the public tests.

4 Avoiding Submission Pitfalls

Since we cannot give marks for an assignment that has not been properly submitted, it is important that you be aware of dangers along the way. Here are a few common errors and how to avoid them.

4.1 Network problems

On rare occasions, MarkUs may break down. The best protection against network problems is to submit your work well before the deadline. Remember that you can repeatedly submit work, so it doesn't hurt to submit everything early just in case. If you are working close to the deadline, submit periodically so that you can at least get part marks for the work you complete on time.

If something unusual does happen with MarkUs, we will post an announcement to the course Web site. The regular Web site might not be readable; if that is the case, you can read the announcement at `http://www.cs.uwaterloo.ca/~cs115`.

On occasion, a problem occurs that will result in our accepting assignments past the deadline. If you are only able to successfully access MarkUs after the deadline, submit your work as normal.

If we need to change the deadline, we will accept assignments submitted by the given deadline. If we do not change the deadline, you will receive feedback but no marks on the assignment.

Keep in mind that your submitting an assignment late is not a guarantee that it will be marked. Moreover, remember that if you repeatedly submit, only the last submission will be listed. If the time of the last submission is past our new deadline, you will not receive any credit for your work.

4.2 Submission failure

There are a host of problems that can occur with your submission, even when the network is working properly.

If you have included non-text such as Comment Boxes or information cut and pasted from the Interactions window, your attempt to submit may fail. Remove all non-text and try again; the problem is likely to be in your tests.

For your work to pass auto-testing, you need to be sure to have the correct names of functions and parameters. Here are some easy ways to ensure correctness:

- Use the provided interfaces for assignments. These have correctly-spelled names of functions and parameters.
- Remember to press the "Submit" button in MarkUs!
- Use public tests (discussed in the Survival Guide) to ensure that your solutions have the correct format.
- Complete Assignment 0 to ensure that you are following the submission procedure correctly.