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 `a8q3.rkt`, and all the files for Assignment 8 should be in the folder/directory `a8`. 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 116 Assignment 13, Question 3
;; Ursula Franklin
;; (Solving the problem of world hunger)
;;
;; ***********************************************
;;
```

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
(define (squaresum x y) (+ (* x x) (* y y)))

 ;; don't do this, either
(define
  (squaresum x y)
    (+
      (*
        x
        x)
      (*
        y
        y))
 ;; this is all right
(define (squaresum x y)
  (+ (* x x)
     (* 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
   [(empty? lon)   empty]
   [else (rest lon)])

(cond
  [(zero? n) 0]
  [(= n 1)   1]
  [else
    (* 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 → num
;; Produces the Euclidean distance between posn1 and posn2.
;; Examples:
;;  (distance (make-posn 1 2) (make-posn 1 2)) ⇒ 0
;;  (distance (make-posn 1 2) (make-posn 1 4)) ⇒ 2
;;  (distance (make-posn 1 1) (make-posn 4 5)) ⇒ 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 116 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 116 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

There are two steps in electronically submitting your assignment. The first is to make sure that all of the files associated with the assignment are in your account on the undergraduate Unix environment. The second step is to send the files to us. You must execute both steps for your work to be marked.

Every term there are students who lose marks by missing the deadline or by executing Step 1 but not Step 2. Read these instructions carefully to avoid having this happen to you.

2.1 Step 1: Putting your files in your account

If you are on a lab machine and have done your work on your Macintosh account, your work is on the right machine. Check Section 2.1.1 to be sure you have the right structure and then go to step 2.

If your work is on your home computer, you need to move your files to a directory (what folders are called in Unix) in your account on the undergraduate Unix environment. Everything below assumes the machine on which you are working is connected to the Internet. If you have made changes to a file and move it a second time, you may be prompted with a question asking if you wish to overwrite your files. You need to answer yes in order for your new work to replace your old work.

The easiest way to move files and submit is to use a Web-based interface called Odyssey. However, it is possible that it will not work for you at some future point in time. For that reason, we have described Odyssey in Sections 2.1.2 and 2.2.1 below, but the other sections describe backup methods.

We describe the necessary file and folder/directory structure, two ways of moving files (Odyssey and the MyWaterloo service) that have the advantage of being identical on Linux, Mac, and Windows, and (briefly) other methods. Feel free to use another method if you have a favorite.

Note: if you have taken CS 116 a previous term, remove your work from that term from your CS 116 folders.

2.1.1 Unix directory structure for assignments

A directory in Unix is the same as a folder on a Mac or in Windows.

You should have a directory called "cs116" in your home directory. Within that, you will need a separate directory for each assignment: a0 for Assignment 0, a1 for Assignment 1, and so on. Inside, say, the directory called a1, you will place a file a1q3.rkt containing your answer to question 3. Each question requires a separate file.

2.1.2 Moving and Submitting Files Using Odyssey

The URL for Odyssey is `https://www.cs.uwaterloo.ca/odyssey-local/`. Note the "s" after "http" and the slash ending the URL; this is a secure page, and it asks you for your UWdir/Quest userid and password. It then presents you with a menu of options; choose the first one, which is "Odyssey CS (student version)". On the next page, click on "Odyssey student.cs", and on the following page, click on the name of your home directory in the section titled "CS Student Environment File Access". This gets you to a list of the files and directories in your home directory on the Unix system.

You can use Odyssey to navigate through your directory structure and to create new directories. The top of the page, in yellow, will have the title "File Manager:" followed by the path to the directory you are currently in. Directly below the title the path is listed again, this time with links: clicking on any part will bring you to a page representing that level in the directory structure. After the bulleted list you will either see the line This path is an empty directory or This path is a directory containing x items for the appropriate value of x, followed by a table listing the items.

If your home directory contains a directory called `cs116`, click on the link in the table to get to that directory. Otherwise, in the section titled "Create a directory" type in the name `cs116` and click on the button.

You should now be on a page describing the `cs116` directory. You can set up the needed directories as described in Section 2.1.1 above, using the section "Create a directory". After creating each assignment directory, remember to go back to the `cs116` page before creating the next one (you can do this by clicking on the link under the title). In the future, you can get back to directory `a1` by following links from your home directory to `cs116` to directory `a1`.

For assignment submission to work properly, files must be in the correct directory in your Unix account. You need to make sure that the files you have created are listed on the page for the directory for the current assignment (e.g. `a1` for Assignment 1). Your home directory on the lab machines is the same as your home directory in your Unix account. If you have done your work in the labs, and your files are listed on the correct page, you can skip the next paragraph and go directly to the paragraph on submission.

If you have worked on a laptop or other non-lab machine, you need to copy your files to the appropriate directory in your Unix account. Using the instructions above, follow links from your home directory to cs116 to the page for the current assignment (e.g. `a1` for Assignment 1). To copy a file from the computer you are using to your Unix account, use the section "Upload a File" to first select a file (using the "Browse" button) and then upload it (using the "Upload!" button). To copy a file from your Unix account to another computer, follow links from your home directory to the file. After the sentence "This path is a file" is a "download" link beside it. Clicking on the link copies the file from your Unix account to the computer you are using.

You can submit all the files in one of your assignment directories by following the directions given on the page found by clicking on the "Assignment Submission" link on the left menu, on the "cs116" link on the next page, and on the link for the current assignment on the page after that. The page associated with a particular assignment will have a button allowing you to submit all the files in your directory for that assignment. This will overwrite any previous submissions of files with the same name. You can do this as often as you like before the submission deadline; late submissions are not permitted in CS 116 (for exceptions that might occur due to network problems, see Section 3.1.2). You must push the submit button to actually submit your assignment. If you fail to do this, your assignment will not be submitted. Every term, we have some students who move their files but do not submit, and they lose the marks associated with that assignment.

The other options we describe below are more complicated, but may be necessary if your files are not on a lab machine and Odyssey in not working.

2.1.3 Moving files using MyWaterloo

MyWaterloo can be accessed at `http://getmail.uwaterloo.ca`, using your favourite Web browser. You can also use this website to check your school e-mail. The site asks you to log in, which you do using your Unix/Mac userid and password.

In addition, you should select the server `mail.student.cs`. You might have two accounts, one for your faculty and one for CS. These accounts each have their own filespace and you wont be able to submit files that are on your faculty account.

Once you have logged in, click the link titled "File Manager" on the menu to the left of the screen. This will display the contents of your home directory on Unix, and will allow you to manipulate the files in various ways.

2.1.4 Other methods of moving files

There are many graphical file browsers and transfer applications available for Windows (e.g. WinSCP), Macs (e.g. Fugu), and Linux (gftp). Make sure you get one that implements secure FTP. Mac OS X and Linux also have commands you can type into a terminal window (`scp`, `sftp`). If you are experiencing computer difficulties, another simple option is to put your files on a data key to transfer them to a lab machine.

2.2 Step 2: Submitting Your Files

Once you have gotten all the files associated with a particular assignment into a directory in your Unix undergraduate account, you must issue the submit commands necessary to submit your assignment, and check that it has been submitted. Both Macs and Linux are built on Unix, but the submit commands are available only on the Waterloo Math undergraduate Unix environment. Although this is a very useful resource and you should eventually learn more about it, this document contains the minimum you need to do the submission. You can avoid direct use of Unix by using Odyssey as described in Section 2.2.1 below. The other methods of submission require typing commands into a Unix session. MyWaterloo cannot be used to submit files, only to move them to/from your Unix account.

2.2.1 Submission using Odyssey

This has already been described at the end of Section 2.1.2 above. You can check the status of what you have submitted using Odyssey by following the links to the assignment submission page for the current assignment.

2.2.2 Submission through a Unix session

To issue the submit commands, you need a command-line terminal window. On the Macs, look for the X11 application (in /Applications/Utilities, or in your Dock in the labs). Double-clicking this brings up a "xterm" window in which you type commands and receive responses. On Linux, look for an icon that looks like a screen in your bottom iconbar, or menu items that look like "xterm", "term", or "terminal". If all else fails, press Alt-Ctrl-F1 to switch to text mode and Alt-Ctrl-F7 to switch back. On Windows, use "Run. . ." under the Start menu. You can also log directly into your Unix account from the X-window client machines located in MC 3008 and MC 6080.

2.2.3 Using `ssh` in a terminal window

Using the terminal window, you can log into your Unix account. The preferred way of doing this is by using the command `ssh`, which is available on Macs and Linux but not on Windows. (For Windows, you can download the free PuTTY terminal application.) To do this, just type:

```
ssh userid@linux.student.cs.uwaterloo.ca
```

into the terminal window (replacing userid with your user id) and hit Return/Enter. (We will stop saying "hit Return/Enter", as it is the way to send all commands in a command-line interface. Just remember to keep doing it.) You will then be prompted to submit a password; your Quest password should work.

After that, if it is the first time that you are using your Unix account, it will ask you to enter publicly-displayable information. You can change this later, so if you dont want to bother, just hit Return to accept the default values. Then it will show you system news, which you can quit out of by typing 'q'. Finally, it will give you a prompt, which may look like

```
[cpu08 1%]
```

This means you are logged in. To log out, enter "logout" in response to this or another prompt.

If you encounter difficulty using the instructions above, it could be that there is a discrepancy between names used on different machines. To solve the problem, use the following command instead of the one above:

```
ssh -Y -l userid linux.student.cs.uwaterloo.ca
```

The userid will be your Unix/Mac userid and the `l` is a lower-case L, not the number `1`. After this you will receive a prompt for your Unix/Mac password.

Here are the commands useful for submission. We have used assignment 0 as an example, which results in `a0` showing up in the commands; you must replace this with `a1`, `a2`, and so on for subsequent assignments.

To list what we expect as submitted files:

```
submit cs116 a0 -L
```

To submit a directory containing all the files for an assignment:

```
submit cs116 a0 <mydir>
```

You must replace `<mydir>` by the relative path to the directory you wish to submit. The path is the sequence of directories to be navigated from the directory you are currently in (which initially is your home directory) to the one you wish to submit, separated by forward slashes. For example, the directory `a0` inside the `cs116` directory has path `cs116/a0`. This command can also be used to resubmit if you make changes after your first submission (if your changes have been made on a home or lab machine, make sure to transfer the new versions to your Unix account first).

To navigate around directories, use the command `cd`. For example, if you are in your home directory and wish to move into the subdirectory `cs116`, you could type `cd cs116`. To get directly from your home directory to your assignment 0 directory, `cd cs116/a0` works. To move up one level, type `cd ..` (two periods is the symbolic name of the parent directory). Remember, when issuing commands with relative paths like the one just above, to adjust the relative path appropriately given your current position.

To check what was submitted:

```
submit cs116 a0 -l
```

The last character in this command is a lower-case L.

3 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.

3.1 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/~cs116`.

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.

3.2 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.