# CS115/116 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 assignment (and, in the case of CS116, the tutorial) solutions are more appropriate. At the very least, a function should come with contract, conditions, purpose, examples, definition, and tests. After you have learned about them, and where it is appropriate, you should add data definitions and templates. 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`.

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 (though, generally, helper functions are placed before the main function). 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. It is also useful when you (as the reader) are looking back at your own code to put it in the right context.

*File Header* →

```
;; ************************************************
;; CS 115 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,conditions, purpose, examples, and template. Block comments should be indicated by double semicolons at the start of a line, followed by a space.

*Block Comments* →

```
;; between: Num Num Num -> Boolean
;; Conditions:
;;    PRE: lower < upper
;; Produces true if lower <= x <= upper, and false otherwise.
```

You should put a blank line between the block comment and the function header, and 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.

```
;; this is good
(define (squaresum x y)
  (+ (* x x)
     (* y y)))
```

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

```
;; don't do this
(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 Use of Constants

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

- To improve the readability of your code by avoiding "magic" numbers. For example, the purpose of the following code using the constant `cost-per-minute`

  ```
  (define cost-per-minute 0.75)
  (* cost-per-minute total-minutes)
  ```

  is much clearer than when a constant is not used:

  ```
  (* 0.75 total-minutes)
  ```

- To allow easier updating of special values. If the value of `cost-per-minute` changes, it is much easier to make one change to the initialization of its constant, than to search through the entire program for the value 0.75 and then deciding if it must change because it refers to `cost-per-minute` or whether it serves a different purpose and should not be changed.
- To define values for testing purposes. As values used in testing become more complicated (for example, lists, structures, lists of structures), it can be very helpful to define named constants that can be used in multiple tests.

Constants should be placed before helper functions and the main function. Constant names should be chosen using the same guidelines as other identifiers (see the next section).

## 1.5 Variable and Function Names

Try to choose names for constants, variables, and functions that are descriptive, not so short as to be cryptic, but not so long as to be awkward. The detail required in a name will depend on context: if a function consumes a single number, calling that number `n` is probably fine; however, if it consumes multiple numbers, names reflecting the role (for example, `upper-limit` and `lower-limit`) are far more helpful than `n` and `m`. Similarly, if constants are used for testing purposes, the test will be much more understandable if the constants have names like `list-of-evens` or `list-no-matches` instead of `list1` and `list2`.

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, and functions that do conversion use a hyphen and greater-than sign to make a right arrow. 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.

## 1.6 Style Summary

- Use block comments to separate functions.
- 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.
- Use constants to improve the style of your program.
- Choose meaningful identifier names and follow the naming conventions used in the textbook and in lecture.
- Do not include Comment Boxes or images.
- Do not cut-and-paste from the Interactions window into the Definitions window.
- Use the design recipe as described in the following sections.

## 1.7 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 problem begins with the data analysis stage, which includes new structure and data definitions for the new data types required by the problem. The is followed by templates for functions consuming those new data values.

**Data Analysis** All new data types require a data definition. If this new type is a structure, then a structure definition is needed as well. Once a name has been assigned to a new type, that name can be used in the contracts that follow.

Suppose we have a new structure Card, to store information about playing cards. The new type requires both a structure defintion (to set up the new Scheme functions for this type), as well as a data definition, which provides details about how the type is to used.

```
(define-struct card (suit val))
;; A Card is a structure (make-card s v), where
;; - s is a Symbol for the suit of the Card
;;   (one of 'club, 'diamond, 'heart, 'spade)
;; - v is an Integer between 1 and 10, inclusive,
;;    for the value of the card
```

**Templates** When more complicated data (structures, lists, and trees) are used in a problem, it is essential to develop template functions from the associated data definitions. The template should consume one parameter, which will be of the new type. As the template is intended as a starting point for your functions, choose a generic name for it (do not use the name of the required function). Include a contract for the template function (use the generic type Any for the produced type).

The template will be formulated from the data definitions. When dealing with a structure type, the template will include references to all the fields of the parameter. When dealing with a recursive type, the template will include a conditional expression to cover both the base case(s) and the recursive case(s).

```
;; my-card-fun: Card -> Any
(define (my-card-fun c)
   ... (card-suit c) ...
   ... (card-val c) ... )
```

After these initial steps, the other design recipe steps should be followed for all the functions - both the main function and its helpers. The design recipe for a function starts with contract (and conditions, as needed), purpose, and examples. You should develop these before you write the code for the function.

*Contract* →
*Conditions* →
*Purpose* →
*Examples* →

```
;; distance: Posn Posn -> Num
;; Conditions:
;;   POST: Produced value >= 0
;; Purpose: Consumes 2 posns, posn1 and posn2, and 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)
  ...)
```

**Contract** 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. 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. The table below shows the types that can be used in the contract.

| | |
|---|---|
| `Num` | for numbers |
| `Int` | for integers |
| `Nat` | for natural numbers (non-negative integers) |
| `Boolean` | for Boolean values (`true` and `false`) |
| `Char` | for characters |
| `String` | for strings |
| `Symbol` | for symbols |
| `(list <t1>...<tn>)` | for a list of exactly `n` elements, where the first element has type `<t1>`, the second has type `<t2>`, etc. |
| `(listof <type>)` | for a list of arbitrary length with elements of `<type>` (could be `Int`, `String`, a `union`, etc.). Do not make `<type>` plural, for example it is `(listof Posn)` not `(listof Posns)`. |
| `Any` | for values of unknown type (i.e. any type is acceptable) |
| `X, Y, etc.` | for values of unknown type, but must match. For example, use `X` if a function consumes two values where the only restriction is that the two values must have the same type. (special to cs116) |
| `<structure name>` | for structures. For example, built-in structures such as `Image` and `Posn`, or user-defined structures. In your contracts, capitalize the name of the structure. |
| `<other>` | for other types; any type that is defined with a data definition in the question document or in a comment in your code can be used as a type in your contracts. For example, the `Card` data definitions defined in CS 115. |
| `union` | for mixed data types. For example, `(union Int false)` could be used to when a function produces either an integer or the boolean value `false`. |

**Conditions** Sometimes the parameters to a function must meet certain conditions (for example, an integer value must be between 4 and 12, inclusive, a string must contain fewer than 5 characters, or the first parameter must be greater than the second). These conditions must be stated as part of the Conditions step of the design recipe, using the title PRE (for pre-conditions - what must be true when the function is called).

Other times, the produced value is known to satisfy certain conditions (for example, the produced value may be all lowercase characters). These restrictions must be stated as the Conditions step, using the title POST (for post-conditions - what must be true about the produced value). Be sure to use parameter names to clarify which condition applies to which parameter. For example,

```
;; f: Int Int String -> String
;; Conditions:
;;    PRE: 4 <= p1 <= 12
;;         4 <= p2 <= 12
;;         p1 >  p2
;;         (string-length p3) <= 5
;;    POST: the produced String contains only lowercase characters
(define (f p1 p2 p3)
  ...)
```

Place one restriction per line, as shown in the sample above. If there are no further restrictions (other than type) on the parameters or the produced value, then the relevant condition statement can be omitted.

**Purpose** 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) and how they relate to the value being produced.

**Examples** 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. 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. However, they should include a base case and a non-base case, when appropriate, and each example should cover a different situation. 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, which can cause your program to fail all our tests.

**Body** 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 proper function.

To avoid this situation, use the provided "interface" files, such as the file `assninterface03.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. Be sure to read your public test results, as this type of error will result in a failed public test.

Teachpacks

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).

**Test** 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 one 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. Consider the following situations when testing (of course, not all these tests will be needed in a given situation).

| Parameter type | Consider trying these values |
|---|---|
| Numerical value | positive, negative, 0, at and between specific boundaries, etc. |
| Boolean | true, false |
| String | empty string, length 1, length >1, extra whitespace, different types of characters, etc. |
| union | values for each possible type |
| list | empty, length 1, length >1, duplicate values in list, special situations, etc. |
| structure | special values for each field |

Remember to test each path through a conditional expression.

When determining the expected value for a test, do not simply run your code and use the produced value. That is the actual value. You must determine the expected value without using your code.

**Note** It is tempting to cut and paste from the Interactions window into the Definitions window, or from the assignment pdf. Don't. What the Interactions window produces is not plain text, and may interfere with automated marking of your work. Similarly, don't cut and paste from the assignment pdf.

## 1.8 A Sample Submission

Here is the code from the second phone bill example done in CS 115 (Module 3).

```
;; ************************************************
;;   CS 115 Assignment 92, Question 1
;;   John A. MacDonald
```

```
;;   (Cell phone bill calculations)
;; ***********************************************

;; 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
;; Conditions:
;;   PRE:
;;        minutes >= 0
;;        freelimit >= 0
;;        rate >= 0
;;   POST:
;;        Produced value >= 0
;; Purpose: Consumes the number of free minutes, freelimit, and the
;; charge for each minute that is not free, rate. Produces
;; charges computed for minutes, where the first freelimit minutes
;; are free and the remaining minutes are charged at rate dollars
;; per minute.
;; 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)
;; Minutes at free limit
(check-expect (charges-for 75 75 134.50) 0)

;; cell-bill: Num Num -> Num
;; Conditions:
;;    PRE:
;;        day >= 0
;;        eve >= 0
;;    POST:
;;        Produced value >= 0
;; Purpose: Consumes day and eve minutes and produces the
;; cell phone bill for day daytime minutes and eve evening minutes.
;; Examples:
;; (cell-bill 150 300) => 100
;; (cell-bill 500 150) => 400
```

```
(define (cell-bill day eve)
     (+ (charges-for day day-free day-rate)
        (charges-for 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)
;; Day and evening at free limits
(check-expect (cell-bill 100 200) 0)
```

## 1.9 Sample Submission with Structures and Lists

Here is the code from the student example done in CS 115 (Module 5), showing how to incorporate data definitions for structures and lists. As in this example, templates will be incorporated into your final code.

```
;; ************************************************
;;   CS 116 Assignment 25, Question 3
;;   John A. MacDonald
;;   (Student grade calculations)
;; ************************************************

;; DATA ANALYSIS

(define-struct student (name assts mid final))
;; A Student is a structure (make-student n a m f), where
;; n is a string (for the student's name), and
;; a, m, f are numbers between 0 and 100 (for the student's assignment,
;;  midterm and final exam grades).
;;
(define-struct grade (name mark))
;; A Grade is a structure (make-grade n m), where
;; n is a string (for the student's name), and
;; m is a number between 0 and 100 (for the student's overall grade).
;;
;; A Student-List is
;; * empty, or
;; * (cons s sl), where s is a Student and sl is a Student-List

;; TEMPLATES

;; student-fn: Student -> Any
;; (define (student-fn s)
;;     ... (student-name s) ... (student-assts s)
;;     ... (student-mid s) ... (student-final s) ...)
;;
```

```
;; studentlist-fn: Student-List -> Any
;; (define (studentlist-fn sl)
;;     (cond [(empty? sl) ...]
;;           [else ... (student-fn (first sl))
;;                 ... (studentlist-fn (rest sl)) ...]))
;;
;; Note: No template is included for the Grade structure because
;;       no function consumes a Grade value.

;; 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
;; Purpose: Consumes a student, astudent, and produces the calculated grade
;; for 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: Student-List -> (listof Grade)
;; Purpose: Consumes a Student-list, slist, and produces a grade list
;; consisting of the grade for each student in 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 empty))
              (cons (make-grade "Virginia Woolf" 100) 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))))
```

## 1.10 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 or tutorial question, the complete design recipe is still required.

**Wrapper Functions** For a function that has the main work done by a helper function, it is acceptable that your examples and tests are provided for the main function only.

**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, conditions, 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.

# 2 Assignment Submission

## 2.1 Login to MarkUs

To log in to MarkUs, please ensure that you are doing the following:

- Use Firefox to access Markus. MarkUs does not work properly with Internet Explorer.
- Use all lowercase letters when entering your userid on the CAS page. Using uppercase letters will render you unrecognizable to the system.
- Use only the first 8 letters of your userid if your userid is longer than 8 letters.
- Use the link to MarkUs provided on the menu at the left. You can also watch how to submit files on MarkUs by downloading the video provided on the course website. (Every time that the video shows CS 115, make sure you change it to CS 116, if appropriate.)

If you encounter a "Login Failed" message, check that you meet the above three criteria before contacting course personnel. If you are still experiencing issues or have run into different errors (such as a redirect problem), please e-mail the course account with your Quest userid and a brief description of your error.

## 2.2 Submitting Your Files

To submit files for Assignments, follow the instructions below:

1. On your MarkUs homepage, click the assignment name to go to its submission page.

   **Submissions** This box shows you how many files you have submitted and how many of the required files you are missing. The Missing Required Files will initially be equal to the number of files you have to submit. As you submit the properly named files, this number will decrease.

   **Assignment Rules** This box has the assignment name, due date, and the names of the required files for the assignment. You must name your assignment submissions as indicted by the list: a lowercase a followed by the two-digit assignment number, then a lowercase q followed by the question number. The file extension must be .rkt. For example, for Assignment 02 Question 3, your file should be named a02q3.rkt

2. Click the "Submissions" tab at the top of the page.
3. Click "Add A New File". A new row will be added to the table. Click "Choose File" or "Submit" in the new row. (The button's name will depend on your web browser.) In the window that appears, browse to where you have saved your file and select it. Click the "Submit" button.
4. Your submitted file will now appear in the table. Check that the filename is correct. If you click on the filename, you will see the contents of the file. You should check that the contents are correct, and that you have submitted what you wanted to submit.

It is best to submit each file individually (add a single file and submit, then repeat for all the files you have). While you can submit all your files at once, there have been instances in the past where the files were not properly submitted.

## 2.3 Replacing a File

You can only replace a file with one that has the same name.

**Warning** Do not use Internet Explorer to replace files. It will appear as though the change has gone through when it has not. Your files will not be replaced.

- Under the "Replace" column, click "Choose File" or "Browse" for the file you want to replace. In the window that appears, browse to the replacement file and select it. Click the "Submit" button.

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, as updates are made. This has several benefits:

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

## 2.4 Deleting a File

If you have submitted the incorrect file, you can delete it by selecting the "Delete" box for the file and then clicking "Submit". This is also an alternative way to replace a file: delete the file you want to replace and submit the correct version/file.

## 2.5 Viewing Assignment Marks and Results

Once marking for an assignment is completed, you can see your mark for each assignment on your MarkUs home page. For a more detailed breakdown of your mark along with marker comments, click the "Results" link.

- On the right is your mark breakdown:

  **Marks** Under this tab you can see the level you received for each criterion.

  **Summary** Under this tab you can see the weight for each catagory and how your mark was added up.

- On the left is the code for one of your files.

  1. To see marker comments, click on the left drop-down box (it is above and to the right of "Annot. Summary").
  2. Select "GRADED_ASSIGNMENT.ss". This file contains all of your submissions.
  3. Put your mouse over any lines that are yellow. A box will appear with the marker's comments about that particular line of code, or a general note about your submission. The "Annot. Summary" tab contains a list of all the comments a marker has made to your assignment. Click on the link at the top left of each comment to go to the associated code.
  4. Read the annotations carefully to avoid losing marks for similar problems on future assignments.

- If you feel an error has been made in the marking of your assignment, follow the instructions on the course web page to request a re-mark. Act promptly, however, as re-mark requests will only be accepted for a short time after the assignment has been marked.

# 3 Lab Style and Submission (CS115 only)

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 l01 (lowercase 'L', followed by digits 01) for Lab 01, l02 for Lab 02, and so on.

Although we recommend completing lab questions during the week of the assigned lab (especially since assignment questions may build on previous lab questions), there is no deadline for submitting your work and receiving public test feedback for labs.

# 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`
or
`http://www.cs.uwaterloo.ca/~cs116`
for CS115 and CS116, respectively.

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. In the case of a system-wide problem with MarkUs, do not email your code to course staff. It will not be accepted. Wait for an announcement in this case.

Keep in mind that 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. If you wish to receive feedback on a late assignment, send email to the tutors so that the assignment will be reviewed.

## 4.2 Public Test 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, the assignment pdf, or the course notes, you may fail the public tests. Remove all non-text and try again; the problem is likely to be in your tests or your examples. There is a problem even if the violating characters are included in comments, not executable code. It may be hard to identify non-text characters, so it is simply best to avoid all cutting-and-pasting (other than from the provided interface file).

For your work to pass public tests and 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.
- Use public tests to ensure that your solutions have the correct format. Make sure that you are receiving the public test emails. The emails are sent to your uwaterloo email address, so you may want to set up email forwarding to your main email address or change your preferred email on WatIAM. Do not mark public test emails as SPAM, as that action can cause problems for other students as well as for yourself.
- Test your code extensively. Passing public tests does not mean that your solution will pass all tests!
- Check MarkUs to see that your submissions are there.
- Complete Assignment 0 to ensure that you are following the submission procedure correctly.