General Marking Guidelines (AXX Version) CS135 ☺

The primary goal of this document is to provide a consistent set of guidelines by which TA's should mark and provide feedback for student assignments. Every assignment marking scheme will be coupled with a general marking scheme whose contents will be more or less the same as previous ones. New content will be highlighted to ensure markers don't miss out on guidelines that apply to new course and assignment content.

It's important to keep in mind that the purpose of marking student assignments is not only to assign and deduct grades, but also to provide meaningful feedback. If you see a student who has done something particularly well in an assignment don't hesitate to leave a nice message. Similarly, if you see a student who has written a poor solution (but not necessarily incorrect according to our guidelines), **please** provide meaningful feedback!

General Marking Notes

- If you are marking more than 30 submissions, click All on MarkUs to show all the submissions you have.
- Do not leave submissions blank. All submissions should have annotations. If a student receives full marks, leave an annotation telling them that they did a good job!
- If you remove marks for any reason, state the reason with an annotation and highlight the most appropriate spot.
- Do not modify preset annotations; this will change them in *every* occurrence for all markers.
- If you notice a common mistake that is not listed under the annotations, please create one under an appropriate category. However, please consult the appropriate teams channel **prior to deducting** for the mistake.
- Do not duplicate any preset annotations.
- Remember to set each assignment's marking status to complete when you are finished.
- Please start marking early! The earlier you start the easier it is for us to help you.
- While marking if you encounter commonly occurring errors, please report them by replying to the Common Errors thread in the appropriate assignment channel on Teams. Please report them as you find them - say you have marked 10 submissions with the same error, even if you haven't completed marking yet you know by that point it's a common error so you can add it to the thread. Lastly, even if someone else reported the same common error, please report it again because that helps us determine just how common it is.
- Please email the course account "cs135@uwaterloo.ca" when you are finished marking each week. The email should include the amount of time you spent marking and the time it took you to complete the TA Assignment.

General Notes on Marking Rubric

- Unless otherwise stated, deduct one rubric level for each major error.
- Unless otherwise stated Do not deduct more than one level for the same error in multiple places.
- If the student did not submit anything for the questions we are checking, deduct *all* rubric levels.
- If you notice any potential error or controversial style not covered in the rubric, consult the appropriate Teams channel for existing clarifications or ask for new clarifications before deducting marks.

Design Recipe

Purpose Statement Correctness (Main Functions):

Select the appropriate rubric level based on how many **main function** purpose statements are correct. Please refer to the specific assignment marking scheme for the purpose statements. For example, **even if an error is repeated in 2 purpose statements, that means both are incorrect, so that is 2 deductions**. Any of the following errors make a purpose statement incorrect.

If students are missing all purpose statements, **deduct all rubric levels**, otherwise mark the present purpose statements.

- **Major Errors:**
  - Missing purpose statement.
  - Ambiguous purpose statement, not clearly explaining what the function does. This could be any of,
    - The purpose statement does not reference the produced value.
    - The purpose statement references the produced value with regards to a helper function called. For example, a purpose saying the function "produces the sum of this-helper and this-other-helper", or, "calls this-helper"
    - Saying it is a wrapper function for a different function.
    - Any confusing or ambiguous statement about the function.
  - The purpose describes how the function works instead of what it does (i.e. describing the procedure/mechanism instead of the outcome).

Purpose Statement Existence (Helper Functions):

This category checks for purpose statement existence in Helper Functions, not correctness or formatting. If the purpose statement for a helper function is present, but incorrect (which you should not be checking), **do not deduct here**. Deduct for missed purpose statements according to the rubric.

If a student has not defined any helper functions, deduct all rubric levels.

Purpose Format:

All purpose statements should follow the format in the style guide. If a student didn't submit anything for this question, or has **no** purpose statements, **deduct all rubric levels**. Otherwise, mark and provide feedback for the **required function purpose statements present**. If several functions contain the same issue, do not deduct multiple times.

You should still mark incorrect purpose statements for format.

- **Major Errors:**
  - Not referencing all parameters in the purpose description, or not mentioning which parameter they are referencing (eg. "consumes velocity" without stating it is the parameter v, or vice-versa).
  - Function headers at the beginning of the purpose (i.e. (fn-name param1 param2 ...)) does not exist or does not match the actual function header (do not deduct for obvious typos, if you are not sure what is a typo and what is an incorrect function header then please consult in the appropriate assignment Teams channel).
  - Function header at the beginning of the purpose is missing brackets.
- **Comment - don't deduct:**
  - If purpose is correct but it is unnecessarily long and convoluted, do not deduct marks, but leave a comment.
  - Multiple occurrences of the same type of error within one question count as one mistake (eg. two mislabeled parameters in a function instead of one).

Examples:

All functions should have **at least 1** illustrative example to test the basic functionality of the code. Some examples have been provided in the questions for students. They may choose to use them and/or write their own. If a student does not write examples, select the appropriate rubric level on MarkUs.

Constants:

Select the appropriate rubric level based on how many required constants are present in the code. Please see the Specific Marking Scheme for details about the constants we expect. Of course, if no constants are present, deduct all rubric levels.

- **Notes:**
  - Students may include additional constants other than the required constants, leave a comment if they do.

Names:

Constant, parameter, and helper function names should be descriptive but not too long.

- **Major Errors:**
  - Ambiguous names, names that are not self-explanatory: For helper functions, parameters, and constants students must use relevant and descriptive names. Deduct marks if a student leaves a completely irrelevant or unrelated name for a helper function or parameter.
  - Using the value in the constant name, inflexible names (e.g. 75-threshold-percentage).
  - Violating the Style Guide Requirements in Section 2.4 (Identifiers) which is that names that are proper nouns like Newton should always be capitalized. Otherwise, use the Racket convention of lowercase letters and hyphens (i.e. top-tax-bracket).
- **Comment - don't deduct:**
  - Having names that are too long. Don't deduct marks but leave a comment.
  - Having filler words like, "the", "that", etc. Don't deduct marks but leave a comment.
  - If a student uses a name that is accurate but not specific enough, leave a note but do not deduct marks.
- **Notes:**
  - While it may be easy to check the required function parameter names and constant names, it can be hard to tell the names of helper functions and helper function parameter should be since those are up to the student. So only deduct if you're sure the names are incorrect such as clearly ambiguous helper function names or parameter names.

    In *this* specific case, **lean towards leniency** if you're not sure, but **leave a comment for feedback**.

Contract Correctness (Main Functions):

Select the appropriate rubric level based on how many main function contracts are correct. Please refer to the specific assignment marking scheme for the contracts. For example, **even if an error is repeated in 2 contracts, that means both are incorrect, so that is 2 deductions**. Any of the following errors make a contract incorrect.

If students are missing all contracts, **deduct all rubric levels**, otherwise mark the present contacts.

- **Major error:**
  - Incorrect type or incorrect number of types listed (other than the exceptions below).

  - Missing requirements **when necessary** (we will specify when they are not necessary).
  - False or incorrect requirements

- **Exceptions:**
  - An Int with a requirement that it must be a non-negative number is the same as a Nat (leave a comment, but do not deduct any marks).
  - If students specify true, but unnecessary requirements, leave a comment, but do not deduct marks. For example if they leave a requirement that a Nat must be non-negative.
  - Requirements can vary as long as they depict the right message. For example, stating that t is non-negative is the same as t >= 0 / 0 <= t.

Contract Existence (Helper Functions):

This category checks for contract existence in Helper Functions, not correctness or formatting. If the contract for a helper function is present, but incorrect (which you should not be checking), **do not deduct here**. Deduct for missed contracts according to the rubric.

If a student has not defined any helper functions, deduct all rubric levels.

Contract Format:

If no contracts are present, deduct all rubric levels; otherwise, mark the existing contracts for **all functions**.

You should still mark incorrect contracts for format.

- **Major Errors:**
  - Missing/Incorrect function name.
  - Surrounding function names in brackets.
  - Including parameter names, or using data types before the colon.
  - Missing colon after function name if the function name is provided.
  - Missing uppercase letter to begin type name (num instead of Num).
  - Using incorrect type names (Number instead of Num).
  - Missing (->) or incorrect arrow used (for example =>).
  - Adding any additional characters to the contract like "+" between types
  - Missing the word "requires" when there are requirements, or incorrectly placing it before the contract.
  - Requirements separated from Contract
  - Unnecessary brackets around function name or type names

    - Missing brackets around list type definitions, for example listof X or list X instead of (listof X) and (list X).

- **Comment - don't deduct:**
    - Student indicates contract explicitly (for example, using ";; contract:")

- **Notes:**
    - If you come across a formatting error which is not mentioned, consult on the appropriate Teams Channel.

Presentation and complexity

Note to future ISAs: You may either combine or keep separate the Whitespace and Layout categories. Here are examples of both:

Whitespace/Layout:

Solutions should be spaced **appropriately** for readability, and different function blocks should be separated.

- **Major Errors:**
    - Missing separators between function blocks. Separators can be at least 2 blank lines or a row of symbols (such as *), or question headings. For example,

```
;; ************************************************************************
                                 or
;; ***************************** Question 2 ****************************
                                 or
;; ----------------------------------------------------------------------
```

    If a student leaves *excessive* space between function blocks, deduct marks.

    - Consistently leaving more than 5 blank lines between function blocks.
    - Multiple lines that **exceed 102 characters**.
    - Design recipe **not in correct order,** purpose, examples, contract, requirements (if applicable), function definition, test cases. For example,

```
;; (mean x1 x2) consumes numbers x1 and x2, and produces their mean
   value.
;; Examples:
(check-expect (mean 0 1) 0.5)
(check-expect (mean 2 4) 3)

;; mean : Num Num -> Num
(define (mean x1 x2)
   (/ (+ x1 x2) 2))
```

```

;; Test cases:
(check-expect (mean 1 1) 1)
(check-expect (mean 1 0) 0.5)
(check-expect (mean -1 1) 0)
(check-expect (mean -1 -1) -1)
(check-expect (mean -2 -4)
```

  - Design recipe for main function interrupted by helper functions, constants or data definitions. Helper functions or constant definitions should be at the top or bottom of main function block but should **never** appear in the middle of it. For example, in the previous example, if students inserted a comment definition or helper function anywhere between design recipe elements.
  - Awkward/incorrect indentation, or lines are consistently too vertical or horizontal. See style guide for guidelines.
  - Giving opening *or* closing brackets their own lines, for example,

```
(define (function x y)
    (
        operation x y
    )
```

- **Comment - don't deduct:**
  - If students fail to include a student header specifying their name, student number, and assignment number, and question number. For example, if they don't have something along the lines of:

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

  - If students consistently miss spaces in their code, for example,

```
(and(or(cond[])))
```

  - If students fail to indicate examples or tests (using ;;Examples and ;;Tests)
  - If students have excessive blank lines at the end of their files.
  - If a student defines a long list for test cases and results exceeding the 102 character limit.

Or, alternatively, separating Whitespace and Layout

Whitespace:

Note that we have split up whitespace and layout for this assignment to avoid the long list of annotations; otherwise, mark as you normally would, but notice there are now two separate categories in the rubric.

Solutions should be spaced **appropriately** for readability, and different function blocks should be separated.

- **Major Errors:**
  - Missing new lines between **main function** design recipe elements. Note that students do not need new lines between the purpose and the examples, as well as between contract and function headers.
  - Missing (at least a) **single** newline between local function definitions or constant blocks. See section 9 of the style guide.
  - Missing (at least a) **single** newline between local definitions and local return statement. See section 9 of style guide.
  - Consistently adding too many new lines between local function definitions or constant blocks. Use your judgement to decide whether it hinders the readability of their code, more than 5 lines is often too many within one local function.

Layout:

- **Major Errors:**
  - Giving opening or closing brackets their own lines.
  - Awkward/incorrect indentation, or lines are consistently too vertical or horizontal. See style guide for guidelines.

    Pay close attention to the line breaks in long predicate functions, if you cannot tell which part belongs which operation, deduct a mark and leave a comment.
  - Main function design recipe **not in correct order,** purpose, examples, contract, requirements (if applicable), function definition, test cases.
  - Local function design recipe in incorrect order (i.e., contract before purpose.) If they are missing either, do not deduct here, deduct under the respective existence category.
  - **Multiple** lines that **exceed 102 characters**.
- **Comment - don't deduct:**
  - If students fail to include a student header specifying their name, student number, and assignment number, and question number.
  - If students fail to indicate examples or tests (using ;; Examples and ;; Tests)
  - If a student defines a long list for test cases and results exceeding the 102 character limit.

Code Complexity:

Incorrect code should **still be marked for code complexity**. Students can and often should define helper functions to avoid repeating code and enhance readability.

- **Major Errors:**
  - Including a cond in the answer part of the else condition, for example,

```
(define (fun x y)
    (cond
        [(condition-to-be-satisfied x y)
        true]
        [else
        (cond
            [(another-condition x y) true]
            [else false])]))
```

in this example, we want the students to **rewrite** this as either of,

```
(define (fun x y)
    (cond
        [(or (condition-to-be-satisfied x y)
              (another-condition x y))
        true]
        [else false]))

(define (fun x y)
    (cond
        [(condition-to-be-satisfied x y) true]
        [(another-condition x y) true]
        [else false]))
```

If you are deducting for this error, **make sure it says cond, not cons**.

  - Using eq? or equal? instead of more specific predicates for comparison, students should be using the following instead of equal? when the contract takes in a specific data type.

```
                      (= num-1 num-2) (symbol=? sym-1 sym-2)
                (and bool-1 bool-2) (and (not bool-1) (not bool-2))
                   (char=? char-1 char-2) (string=? str-1 str-2)
```

Note that (and bool bool) and (and (not bool) (not bool)) replace boolean=?, which they should not be using.

  - Checking conditional criteria using

```
                    (boolean=? bool true) (boolean=? bool false)
                                OR
                    (equal? bool true) (equal? bool false)
```

These should instead be:

```
                    bool   (not bool)
```

- Other unnecessary code that adds code complexity. For example, if there is repeated code that can be simplified by re-factoring or grouping cases together, deduct a rubric level and leave a comment.

- **Comment - don't deduct:**
  - If a student defines any **unnecessary** helper functions that *only* perform simple calculations (e.g.add 2 numbers together).
  - Students may (and will) define more helpers than required. Do not deduct any marks if they have more (relevant) helpers. Use your judgement, but if you're not sure whether a helper function is relevant, consult the appropriate Teams channel.
  - If students use **unnecessary** nested conds in their return statements (that *isn't the else condition*). For example,

```
(define (fun x y)
    (cond
        [(condition-to-be-satisfied x y)
         (cond
            [(another-condition x y) true]
            [else false]]
        [else false])))
```

  In many cases, code like this can be simplified, if this is the case, leave a comment.

- **Notes:**
  - If students define *any* helper functions, make sure you are checking for **purpose statement and contract existence** and providing the appropriate feedback.
  - If you see any errors that are not on the list above or have any doubts regarding helper functions, please consult the appropriate Teams channel.