Homework 1: Grade Calculator

The purpose of this assignment is two-fold:

- to get you programming fluently in DSSL2, the language that we'll be using for the course, and
- to get you acquainted with the course's grading system.

As a bonus, you'll be able to use the program you write to calculate your own grade as the quarter goes on, so you can know how things are going.

On Canvas, you will find starter code (`grade-calculator.rkt`) including some definitions, headers for the methods and functions that you'll need to write, along with an insufficient number of tests.

Installing DSSL2

To complete this homework assignment, you will first need to install the DrRacket programming environment, version 8.11 (available from `racket-lang.org`). Older versions you may have already installed will not work. Then you will need to install the DSSL2 language within DrRacket.

Once you have DrRacket installed, open it and choose "Package Manager" from the "File" menu. Type `dssl2` as the source, then click the "Install" button and wait for installation to finish. When it's finished, the "Install" button should change to "Update"; then close the window.

Background

Final grades in this class are composed of two components:

- A *base grade*, and
- A number of *modifiers*

Base grades are the starting point for a students' final grade, which modifiers then adjust up or down, one or more partial letter grades at a time. The result of this adjustment is the actual final grade.

This assignment handout outlines the structure of your grade calculator and the interfaces of its individual pieces. For the specifics of each step of the computation, please refer to the syllabus; now is a great time to get to know it!

Data Definitions

To represent the data you will need to compute grades, we will use the following definitions, which you can find near the top of `grade-calculator.rkt`.

- The possible outcomes a student can earn on a homework (or project) submission are represented as strings. They can be recognized using the `outcome?` predicate.
- `homework`s are represented as structs which have two fields: `outcome`, which is the highest outcome the student achieved on this homework across submissions, and `self_eval_score`, which is the score (out of 5) the student earned on this homework's self-evaluation.
- The `project` is a represented as a struct which has two fields: `outcome`, which is the highest outcome the student achieved on the project across their submissions, and `docs_modifier`, which is the modifier the student earned on their project design documents.
- Letter grades are represented as strings corresponding to the possible letter grades at Northwestern. They can be recognized with the `letter_grade?` predicate.

Our grading scripts will follow these definitions; if you change them, we will not be able to run your submission.

Part I: Modifiers

You will need to implement a series of functions to compute the different modifiers a student can earn in this class.

Modifiers are represented as integers—positive or negative—which denote the number of partial letter grades—up or down—by which a letter grade should be adjusted. As such, each of these functions must return an integer, as indicated by the `int?` contracts on these functions.

The in-class exercise modifier is already implemented for you, however, you will need to implement the remaining modifiers below yourself.

1. `worksheets_modifier(TupC[num?, num?]) -> int?` takes a vector of two real numbers between 0.0 and 1.0 (inclusively) as argument, whose elements represent percentage scores on our two worksheets. It returns the modifier (-1, 0, or 1) that these worksheet scores would earn.
2. `exams_modifiers(nat?, nat?) -> int?` takes two integers between 0 and 20 (inclusively) as arguments, each of which represents a score on one of our two exams. It returns the combined modifiers (this result can be an integer between -6 and 2) earned by both exams together, including the "exam improvement" modifier.
3. `self_evals_modifier(VecC[homework?]) -> int?`, takes a vector of five homework structs (see above) as an argument, and returns the modifier (-1, 0, or 1) that the five self-evaluations would earn. Because there are five homework assignments in the class, your function must raise an error if an incorrect number of homeworks are passed in.

Of course, you will also want to (and need to) write extensive tests for each of these functions: untested code is broken code; you just don't know it yet.

Part II: Base Grades

Your grade calculator must implement a `Student` class, which represents a student and their body of work in this class. We provided an outline for the class, as well as some of the methods. You will, however, need to implement the rest of the methods yourself.

4. The constructor, `Student.__init__(self)` takes a student's name, homeworks, project, worksheets, in-class exercises, and exams as arguments. Contracts for each of them can be found on the fields of the class.

5. `Student.get_homework_outcomes(self) -> VecC[outcome?]` takes no arguments (besides the receiver, `self`) and returns a vector storing the outcomes (and only the outcomes!) of each of this student's homeworks. The order of this vector should follow the order of the `homeworks` field of the class.

6. `Student.get_project_outcome(self) -> outcome?` takes no arguments (besides the receiver, `self`) and returns the outcome which this student earned on the project.

7. `Student.resubmit_homework(self, nat?, outcome?) -> None` takes a homework number (1 to 5) and an outcome as arguments, and replaces the outcome on that homework with the given outcome. This method does not return anything. Attempting to resubmit a homework which does not exist is an error which your method should detect and report.
   Note: while the class's resubmission policy only counts resubmissions if they improve a homework's outcome, you don't need to handle this here; a simple update will do.

8. `Student.resubmit_project(self, outcome?) -> NoneC` takes an outcome as argument, and replaces this student's outcome on the project with the given outcome. This method does not return anything.

As before, you will also want to test these methods extensively.

Part III: Final Grades

Once you can compute both base grades and modifiers, you can combine them into final grades. For this, you'll need to implement one standalone function and two methods of the `Student` class:

9. `Student.total_modifiers(self) -> int?` is a method that takes no arguments, and returns the sum of all the modifiers this student earned across the entire class; see the syllabus for the complete list.
   Note: the modifier for the project documents is not computed; you can

find it directly in a field of `project` structs. We also provided a method that computes the "project program above expectations" modifier; you do not need to compute it yourself.

10. `apply_modifiers(letter_grade?, int?) -> letter_grade?` is a standalone function (i.e., not inside the `Student` class) which takes a base grade and an integer representing the total modifiers a student earned, and returns the final letter grade that corresponds to this base grade adjusted accordingly.
11. Finally, the `Student.letter_grade(self) -> letter_grade?` method brings everything together and computes a student's final letter grade.

As always, we expect you to write thorough test suites for these operations also.

Honor code

Every programming assignment you hand in must begin with the following definition (taken from the Provost's website;[1] see that for a more detailed explanation of these points):

```
let eight_principles = ["Know your rights.",
     "Acknowledge your sources.",
     "Protect your work.",
     "Avoid suspicion.",
     "Do your own work.",
     "Never falsify a record or permit another person to do so.",
     "Never fabricate data, citations, or experimental results.",
     "Always tell the truth when discussing your work with your instructor."]
```

If the definition is not present, you receive no credit for the assignment.

Note: Be careful about formatting the above in your source code! Depending on your pdf reader, directly copy-pasting may not yield valid DSSL2 formatting. To avoid surprises, be sure to test your code *after* copying the above definition.

Grading

Please submit your completed version of `grade-calculator.rkt`, containing:

- definitions for the methods and functions described above,
- sufficient tests to be confident of your code's correctness,
- and the honor code.

Be sure to remove any leftover debugging printing code, comment out any code that would cause infinite loops, and make sure that your submission can run successfully. If your submission produces excessive output, loops infinitely, or crashes, we will not be able to give either you feedback on it or credit for it. **Please run your code one last time before submitting!**

[1] `http://www.northwestern.edu/provost/students/integrity/rules.html`

Functional Correctness

We will use four separate test suites to test your submission, each covering a subset of the operations you must implement:

- **Basic modifiers**: `worksheets_modifier` and `exams_modifier`.
- **Advanced modifiers**: `self_evals_modifier`.
- **Basic grades**: `Student.__init__`, `Student.get_homework_outcomes`, `Student.get_project_outcome`, `Student.resubmit_homework`, and `Student.resubmit_project`.
- **Advanced grades**: `total_modifiers`, `apply_modifiers`, and `letter_grade`.

To get credit for a test suite, your submission must pass *all* its tests.
Be aware: our tests check edge and error cases, and yours should too; code that only works in the happy case is broken code.

The outcome your submission will earn will be determined as follows:

- **Got it**: passes all four test suites.
- **Almost there**: passes both basic test suites and fails a single advanced test suite.
- **On the way**: either passes both basic test suites, or passes both the basic and advanced test suite for one of the two topics.
- **Not yet**: does not achieve "on the way" requirements.
- **Cannot assess**: we could not successfully grade your submission.
- **Missing honor code**: your submission did not include the honor code.

Non-Functional Correctness

For this assignment, the self-evaluation will be specifically looking for:

- Thorough testing, including edge cases
- Rigorous checking of error cases
- Reuse of code (*i.e.*, not copy-paste) where pertinent: if you find yourself repeating code or logic, abstract out common code into a separate helper and use that