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 9.0 (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.

To use DSSL2, make sure DrRacket is set to "Determine language from source" in the bottom left corner[1] and start your program with the line: `#lang dssl2`

Background

As stated in the syllabus, final letter grades in the class are derived from grades on four *tracks* which correspond to the four learning objectives of the class: *implementation*, *integration*, *theory*, and *design*.

In this homework, you will write functions to calculate track grades, as well as to combine them into final grades. This handout outlines the structure of your grade calculator and the interfaces of its individual pieces. For the specifics of each part of the computation, please refer to the **syllabus**; now is a great time to get to know it!

This grade calculator assumes the student has received enough credit for in-class exercises as a minimum requirement to pass the class.

[1] If you see something else, like "Advanced Student Language" click the bottom left and choose "The Racket Language".

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`.

- `homework`s are represented as structs which have three fields: the number of test suites (0–4) that successfully passed, the percentage (0.0–1.0) of correct answers on the associated self-evaluation, and the percentage (0.0–1.0) of mutants that were killed by its test suite. Because homework 1 does not feature mutation testing, `None` is also an acceptable value for the third field; representing a homework without mutation testing.

  Both here and in the rest of this assignment, percentages are represented as floating-point numbers (real numbers, roughly) between 0 and 1. The percentage 36%, for example, would be represented as the number 0.36.

- The `project` is represented as a struct which has two fields: the number of test suites (0–6) that successfully passed, and a vector of three booleans representing whether each of the three project design documents was satisfactory or not.

- Worksheet and exam scores are represented as percentages (0.0–1.0).

- The possible grades for each track (**F**, **D**, **C**, **B**, and **A**) are represented as strings. They can be recognized using the `track_grade?` predicate.

- The letter grades that can result from combining tracks (**F**, **D**, **D+**, **C-**, **C**, **C+**, **B-**, **B**, **B+**, **A-**, and **A**) are also represented as strings. They can be recognized using the `letter_grade?` predicate.

  Note that **D+** is not a valid final grade at NU; we only use it as an intemediate result when computing grades, and it gets rounded to a **D**.

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

Part I: Tracks

You will need to implement a series of functions to compute grades for each of the tracks. Each individual function is fairly simple (no more than a dozen lines of code or so), but there are a good number of them: repetition is an effective way to achieve fluency!

1. `integration_grade(int?) -> track_grade?` takes a single integer as argument, which represents the number of test suites passed for the project. It returns the approriate integration track grade following the rules from the syllabus.

   Your function must check that the number of test suites is in the correct range (0–6); if that's not the case, it must raise an error with the message: `"score out of range"`.

2. `implementation_grade(VecC[int?]) -> track_grade?` takes a vector of integers as argument, which represents the number of passed test suites for each homework. It returns the approriate implementation track grade, following the rules from the syllabus.

   Your function must check that the input vector has the correct number of homeworks (5); if that's not the case, it must raise an error with the message: `"incorrect number of assignments"`.

   In addition, your function must also check that each homework has a valid number of passed test suites (0–4); if that's not the case, it must raise an error with the message: `"score out of range"`.

3. `exams_theory_points(num?, num?) -> int?` takes in two percentages as inputs, each one being the score for one of our two exams, and returns the total number of theory points earned by the two exams together.

   Your function must check that its inputs are valid percentages (between 0% and 100%) and if that's not the case, it should raise an error with the message `"score out of range"`.

4. `tally_design_points(VecC[AnyC], int?, FunC[AnyC -> bool?]) -> int?` takes three inputs: a vector of design-related assignments (of any type), an expected number of assignments, and a *function* that takes an assignment as input and returns whether it earns a design point or not. Your tally function must return the total number of design points earned by the given assignments, using the given function to check whether each one contributes to the total.

   In addition your function must also check that the number of assignments in the vector matches the expected number. If that's not the case, your function must raise an error with the message:
   `"incorrect number of assignments"`.

   Hint: remember what you learned in 111.

5. `self_evals_design_points(VecC[num?]) -> int?` takes a vector of self-evaluation scores represented as percentages (0.0–1.0) and returns the total number of design points earned from the self-evaluations, following the rules from the syllabus.

   Unless the correct number of self-evaluations (5) is passed in, your function must raise an error with the same message as earlier functions.

   Hint: you've already done of lot of the work for this one.

6. `mutation_testing_design_points(VecC[num?]) -> int?` takes a vector of mutation testing scores represented as percentages (0.0–1.0) and returns the total number of design points earned from mutation testing, following the rules from the syllabus.

Unless the correct number of mutation testing results (4) is passed in, your function must raise an error with the same message as earlier functions.

7. `design_docs_design_points(VecC[bool?]) -> int?` takes a vector of booleans which correspond to whether each of the three project design documents is satisfactory or not, and returns the number of design points earned by the design documents, following the rules from the syllabus.

   Unless the correct number of design documents (3) is passed in, your function must raise an error with the same message as earlier functions.

Part II: Final Grades

Next, you will need to implement three functions to combine track grades into final grades.

8. `base_grade(TracksC) -> track_grade?` takes a vector of four track grades (checked using the `TracksC` contract) as input and returns the *base grade*, following the rules outlined in the syllabus.

   Hint: we provided the `grade_lt` function in the starter code.

9. `n_above_expectations(TracksC) -> int?` takes a vector of four track grades (checked using the `TracksC` contract) as input and returns the number of tracks which are *above expectations*, following the rules outlined in the syllabus.

10. `final_grade(track_grade?, int?) -> letter_grade?` takes the base grade and the number of tracks above expectations as arguments, and returns the final letter grade these would result in, following the rules from the syllabus.

    Hint: the possible letter grades are available in the `letter_grades` vector. You may find it useful when writing this function.

Part III: Students

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 the `letter_grade` method. You will need to implement the rest of the methods yourself.

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

12. `Student.get_homework_grades(self) -> VecC[int?]` takes no arguments (besides the receiver, `self`) and returns a vector storing the number of test suites passed for each of this student's homeworks. The order of this vector must follow the order of the `homeworks` field of the class.

13. `Student.get_project_grade(self) -> int?` takes no arguments (besides the receiver, `self`) and returns the number of test suites passed for the project.

14. `Student.resubmit_homework(self, int?, int?) -> NoneC` takes a homework number (1 to 5) and a number of passed test suites as arguments. It replaces the number of test suites for that homework with the new one if it is better than the existing one; otherwise it should do nothing. This method does not return anything.

    When attempting to resubmit a homework that does not exist (i.e., not 1 to 5), your function must raise an error with the message `"no such homework"`.

15. `Student.resubmit_project(self, int?) -> NoneC` takes a number of passed test suites as argument, and replaces this student's current for the project with the new one if it is better than the existing one; otherwise it should do nothing. This method does not return anything.

Testing

Untested code is broken code, you just don't know it's broken yet.

In this class, we expect you to write your own thorough test suites to gain confidence in your code's correctness *before* you submit. To get started, you can take inspiration from the tests in the starter code, as well as the examples in the syllabus.

Beyond that, the **"Testing" video** and the **"A Beginner's Guide to Testing.pdf" document** under "Supplemental Materials" on Canvas include detailed guidance regarding how and what to test.

This assignment is a perfect opportunity to get some practice writing good test suites. Starting with homework 2, we will be assessing the thoroughness of your testing using mutation testing; make sure you're prepared for that!

Honor code

Every programming assignment you hand in must begin with the following definition (taken from the Provost's website;[2] 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."]
```

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

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 in 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!**

Functional Correctness

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

- **Basic tracks**: `implementation_grade` and `integration_grade`.
- **Advanced tracks**: `exams_theory_points`, `tally_design_points`, `self_evals_design_points`, `mutation_testing_design_points`, and `design_docs_design_points`.
- **Basic grades**: `Student.__init__`, `Student.get_homework_grades`, `Student.get_project_grade`, `Student.resubmit_homework`, and `Student.resubmit_project`.
- **Advanced grades**: `base_grade`, `n_above_expectations`, and `final_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.

Non-Functional Correctness

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

- Reuse of code (*i.e.*, not copy-paste) where pertinent: if you find yourself repeating code or logic, abstract it instead.
- Using appropriate programming constructs to keep programming logic as simple as possible.
- Thorough testing, including of edge cases and error cases (`assert_error`).

The self-evaluation will "spot check" your implementation's non-functional correctness. For example, to evaluate how extensively you may have tested, it may spot check if you have written a specific test case with a question possibly worded like "Did you write a unit test for X function to test that Y input produces the correct value?". For such a question, you would receive credit if you did write such a test and we could verify its existence in your submission. Specific instructions will be given when the self-evaluation is released.

