Assignment:	2
Due:	Tuesday, September 25, 9:00 pm
Language level:	Beginning Student
Files to submit:	`score.rkt`, `immigration.rkt`, `laundry.rkt`, `bonus.rkt`
Warmup exercises:	HtDP 4.1.1, 4.1.2, 4.3.1, 4.3.2
Practice exercises:	HtDP 4.4.1, 4.4.3, 5.1.4

- Policies from Assignment 1 carry forward.
- This assignment covers concepts up to the end of Module 02. Unless otherwise specified, you may only use Racket language features we have covered up to that point.
- Make sure you read the **OFFICIAL A02 post on Piazza** for the answers to frequently asked questions.
- For this assignment we have provided starter files to help you organize your functions. You are not obligated to use them, but they reflect the style and organization we expect of your work. Download them from the assignment page.
- Your solutions must be entirely your own work.
- Solutions will be marked for both correctness and good style. Follow the guidelines in the Style Guide.
- **Unless otherwise specified, for this and all subsequent assignments, you should include the design recipe as discussed in class.**
- The basic and correctness tests for all questions will always meet the stated assumptions for consumed values.
- You must use *check-expect* for both examples and tests.
- You must use the **cond** special form, and are not allowed to use **if** in any of your solutions.
- **It is very important that your function names match ours.** Use the basic tests to check this. In most cases, solutions that do not pass the basic tests will not receive any correctness marks. The names of the functions must be written exactly. The names of the parameters are up to you, but should be meaningful. The order and meaning of the parameters are carefully specified in each problem.
- Any string or symbol constant values must exactly match the description in the questions. Any discrepancies in your solutions may lead to a severe loss of correctness marks. Basic tests results will catch many, but not necessarily all of these types of errors.
- Since each file you submit may contain more than one function, it is very important that the code runs. If your code does not run, then none of the functions in that file can be tested for correctness.

- Do not send any code files by email to your instructors or ISAs. Course staff will not accept it as an assignment submission. Course staff will not debug code emailed to them.
- You may use examples from the problem description in your own solutions.

Here are the assignment questions you need to submit.

1. You may have heard of a video game called *Tetris*. The goal of *Tetris* is to collect as many points as possible. To score points, players have to fill an entire horizontal line with blocks (so-called *Tetrominos*). These *Tetrominos* appear on the top of the screen and start travelling downward until they collide with the bottom of the screen or another already placed *Tetromino*. While moving downward, players can move and rotate *Tetrominos*. With progressing play time (expressed as *levels*), *Tetrominos* will have increased downward speed, making the game more and more difficult. Two factors determine how many points players score:

    - the number of lines eliminated by a single *Tetromino* and
    - the game's level.

Level	Points for 1 line	Points for 2 lines	Points for 3 lines	Points for 4 lines
0	40	100	300	1200
1	80	200	600	2400
2	120	300	900	3600
...				
9	400	1000	3000	12000
10	440	1100	3300	13200
...				
n	?	?	?	?

    Write a Racket function `tetris-score` that consumes the current level (as an integer) as well as the number of lines eliminated (also as an integer) and then outputs the score. Due to the shape of *Tetrominos*, it is not possible to eliminate more than four lines at a time. As a result, `tetris-score` should produce `0` if the number of lines is 5 or larger. To give an example, `(tetris-score 19 2)` produces `2000` and `(tetris-score 19 6)` produces `0`.

    Place your function in the file `score.rkt`.

2. As part of its immigration policy, Canada uses a "Comprehensive Ranking System" to assess whether applicants qualify for permanent residency. In this question you will implement a simplified version of this system for the "Canadian Experience Class" (CEC) category[1]. In this ranking system, eligible applications score points for certain features that the government deems valuable to the Canadian society (all features relevant to this assignment are listed

[1]This is based on the data from `https://www.canada.ca/en/immigration-refugees-citizenship/services/immigrate-canada/express-entry/eligibility/criteria-comprehensive-ranking-system/grid.html`, as retrieved on 2018-09-03

below). Applicants are then ranked according to their score, and the highest-ranking ones are invited to apply for permanent residency.

(a) Write a Racket function `pr-cec-score` that calculates an applicant's score. The function consumes the following parameters (in the given order):

- A natural number indicating the person's age. Points are awarded as follows:

Age	Points
17 years or younger	0
18 years	90
19 years	95
20 – 29	100
30 years	95
31 years	90
32 years	85
...	
48 years	5
49 years or older	0

- A symbol representing the applicant's education level; it takes one of the following values: `'graduate` which contributes 126 points, `'undergraduate` which contributes 112 points, or `'highschool` which contributes 28 points.
- A natural number between 1 and 10 inclusive that represents the applicant's language proficiency. A value of 9 and higher contributes 116 points, a value of 8 contributes 88 points, a value of 7 contributes 64 points, and any other value contributes 0 points.
- A natural number indicating the number of years of work experience the applicant has. One year contributes 35 points, 2–3 years contributes 56 points, and 4 years or more contributes 70 points. No work experience yields 0 points.
- A Boolean indicating whether the applicant has a job offer in Canada. It is `true` if the applicant has a job offer, and `false` otherwise. A job offer contributes 200 points, and no job offer contributes 0 points.

For example, `(pr-cec-score 22 'undergraduate 5 1 false)` produces `247`.

You might find helper functions useful in writing this function. Be clever about reducing the number of conditions you have to evaluate for each criterion (for example, evaluating the `age` criterion can be done in no more than four conditions). You can assume that all symbol passed to your function as parameters are valid.

(b) To be eligible for the "Canadian Experience Class" category in the last round, applicants needed one or more years of work experience, and they also had to score 350 points or more. Write a predicate `pr-cec-eligible?` which produces `true` if the given applicant is eligible for the CEC category, and `false` otherwise. This predicate consumes the same parameters as `pr-cec-score`.

For example, `(pr-cec-eligible?  28 'graduate 9 2 false)` produces `true`.

You do not have to check for applications that are impossible in combination so long as individual arguments obey their requirements. For example, we will not test cases where the applicant is 8 years old but has 10 years of work experience.

Place your solutions in `immigration.rkt`.

Notes about parameters: these functions require you to write functions with many parameters. To avoid lines being longer than 80 characters (and thus losing style marks) your function headers can be broken up over more than one line.

Notes about constants: somewhat unfortunately, this question will require you to define numerous constants (i.e., more than 10). Some guidelines:

- Constants for ranges of values should be named consistently. For example, naming one constant `education-highschool` and a closely related one `grad-points` is probably a bad idea. Using `education-highschool` and `education-graduate` would be more consistent. Choose a consistent naming scheme and stick with it. If your naming scheme is expressive, you even might be able to omit documentation for your constants.
- Related constants should be grouped together in your code. One line of whitespace should separate each group of constants.
- Whether to put values in a constant or not is often a judgement call. One of the guidelines is that values that might change frequently should be defined in constants and not directly coded into functions. In the context of this assignment question, it is reasonable to assume that points for education level, language proficiency, work experience, as well as age brackets might change often.

Notes about testing: Testing code with this many conditions can be a challenge. Here are some guidelines that might make your life easier:

- Start by finding a combination of parameters that produces zero points.
- Using this initial test as a base case, vary other parameters in isolation to vary the point totals for each condition.
- When testing intervals, be sure to test both boundary conditions and conditions within the range.
- Make sure you have sufficient tests that every line of code is executed: there should be no "black highlighting." This means that together, the tests must cover all conditions of your code.

3. You have been so busy studying that you have neglected to do your laundry. Now you are running late to get to CS 135, so you go through your hamper to find some not-clean-but-kinda-acceptable clothes. Write a Racket predicate to help you decide what to wear.

    - If the item is smelly, you should not wear it!

- Otherwise, if the item is of type `'socks` and has been in the hamper for 4 days or longer, you should not wear it.
- Otherwise, if the item is of type `'shirt` and has been in the hamper for 10 days or more, you should not wear it.
- Otherwise, if the item is of type `'shirt` and has been in the hamper for 2 days or less, you should not wear it.
- Otherwise, you might wear the item.

(a) Write a predicate `acceptable-to-wear/cond?` which consumes a Boolean parameter (`true` for a smelly, `false` for a non-smelly item), a symbol representing the item's clothing-type (`'socks` or `'shirt`), and a natural number indicating how many days the item has stayed in the hamper already. It then produces `true` if the item is acceptable to wear and `false` otherwise. The function should use **`cond`** and no Boolean operations (so no **`and`**, **`or`** or `not`).

For example, `(acceptable-to-wear/cond? false 'shirt 8)` produces `true`. Similarly, `(acceptable-to-wear/cond? true 'socks 2)` produces `false`.

Carefully consider the order of your conditions to minimize duplicate code.

You may not use additional helper functions in this question. You do not need to check that `clothing-type` is actually a valid type of clothing.

(b) Write a Racket predicate `acceptable-to-wear/bool?` that consumes the same parameters as `acceptable-to-wear/cond?` but uses Boolean operations only, and no instances of **`cond`**.

Place these functions into the file `laundry.rkt`

This concludes the list of questions for which you need to submit solutions. Don't forget to always check your email for the basic test results after making a submission.

Bonus (5%): Write a function `date->day-of-week` which consumes a natural number and produces a symbol corresponding to the day of the week, according to the Gregorian calendar.

An integer encodes a date as follows:

- The leading four digits correspond to the year
- The two digits after that correspond to the month of the year
- The final two digits correspond to the day of the month

The function should return one of seven symbols: `'Monday`, `'Tuesday`, `'Wednesday`, `'Thursday`, `'Friday`, `'Saturday`, or `'Sunday` according to the day of the week the consumed date corresponds to.

To give an example: consider Monday, July 1, 1867. The equivalent call would be `(date->day-of-week 18670701)`, which produces `'Monday` as output. Similarly, for September 25, 2018, `(date->day-of-week 20180925)` the function produces `'Tuesday`.

You can assume that all dates are correct. This means that your function does not have to check for invalid input, such as, April 65, 1900 (`(date->day-of-week 19000465)`). You can also assume that no dates before January 1, 1753 AD, will be tested.[2]

You may only use the Racket constructs we have discussed in lecture so far, and built-in mathematical functions. You may not use Racket's date functions.

It is acceptable (and encouraged) to consult rules or algorithms for computing the day of the week given a date, but it is not acceptable to copy and paste code you have found. You must cite any resources you use in your Racket file. Just like before, it might be helpful to define multiple functions.

Put your solution in `bonus.rkt`.

Challenges and Enhancements: *Reminder—enhancements are for your interest and are not to be handed in.*

check-expect has two features that make it unusual:

1. It can appear before the definition of a function it calls (this involves a lot of sophistication to pull off).
2. It displays a window listing tests that failed.

However, otherwise it is a conceptually simple function. It consumes two values and indicates whether they are the same or not. Try writing your own version named *my-check-expect* that consumes two values and produces 'Passed if the values are equal and 'Failed otherwise. Test your function with combinations of values you know about so far: numbers (except for inexact numbers; see below), booleans, symbols, and strings.

Expecting two inexact numbers to be exactly the same isn't a good idea. For inexact numbers we use a function such as *check-within*. It consumes the value we want to test, the expected answer, and a tolerance. The test passes if the difference between the value and the expected answer is less than or equal to the tolerance and fails otherwise. Write *my-check-within* with this behaviour.

The third check function provided by DrRacket, *check-error*, verifies that a function gives the expected error message. For example, (*check-error* (/ 1 0) "/: division by zero")

Writing an equivalent to this is well beyond CS135 content. It requires defining a special form because (/ 1 0) can't be executed before calling *check-error*; it must be evaluated by *check-error* itself. Furthermore, an understanding of *exceptions* and how to handle them is required. You might take a look at exceptions in DrRacket's help desk.

[2] Fun fact: the Gregorian calendar was not fully established in the United Kingdom and her colonies before that date (https://en.wikipedia.org/wiki/Adoption_of_the_Gregorian_calendar, as retrieved on 2018-09-03).