CS 135 Fall 2017
Becker, Goldberg, Kaplan, Nijjar, Tompkins, Vasiga

Assignment: 2

Due:	Tuesday, September 26, 9:00 pm
Language level:	Beginning Student
Files to submit:	`cond.rkt`, `airmiles.rkt`, `grades.rkt`, `blood.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.
- Your solutions must be entirely your own work.
- Solutions will be marked for both correctness and good style.
- Good style includes qualities such as meaningful names for identifiers, clear and consistent indentation, appropriate use of helper functions, and documentation (the design recipe).
- **For this and all subsequent assignments, you should include the design recipe as discussed in class (unless otherwise noted, as in question 1).**
- 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.** You must use the basic tests to be sure. 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 descriptions 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 will 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. A **cond** expression can always be rewritten to produce *equivalent expressions*. These are new expressions that always produce the same answer as the original (given the same inputs, of course). For example, the following are all equivalent:

```
(cond                  (cond                  (cond
  [(> x 0)   'red]       [(<= x 0) 'blue]       [(> x 0) 'red]
  [(<= x 0)  'blue])     [(> x 0)  'red])       [else    'blue])
```

(There is one more really obvious equivalent expression; think about what it might be.)

So far all of the **cond** examples we've seen in class have followed the pattern

```
(cond [question1 answer1]
      [question2 answer2]
      ...
      [questionk answerk])
```

where *questionk* might be **else**.

The questions and answers do not need to be simple expressions like we've seen in class. In particular, either the question or the answer (or both!) can themselves be **cond** expressions. In this problem, you will practice manipulating these so-called "nested **cond**" expressions.

In some cases, having a single **cond** results in a simpler expression, and in others, having a nested **cond** results in a simpler expression. With practice, you will be able to simplify expressions even more complex than these.

Below are three functions whose bodies are nested **cond** expressions. Write new versions of these functions subject to the following constraints:

- Each function uses **exactly one cond**.
- Each function always produces the same answer as the original regardless of the value of *x* or the definitions of the helper predicates *p1?* and *p2?*.
- All of the **cond** questions are "useful", that is, there exists no question that could never be asked or that would always answer *false*.
- Each new function has the same function name as the original.
- Does not use any helper functions (other than *p1?* and *p2?*).

(a)

```
(define (q1a x)
  (cond
    [(p2? x)
     (cond
       [(p1? x) 'left]
       [else    'down])]
    [else
     (cond
       [(p1? x) 'up]
       [else    'right])]))
```

(b)

```
(define (q1b x)
  (cond [(p1? x)
         (cond [(p2? x)
                (cond
                  [(p1? (+ x 1)) 'up]
                  [(p2? (* 2 x)) 'down]
                  [else 'right])]
               [else
                (cond
                  [(p2? 2) 'down]
                  [else 'up])])]
        [(p1? 0)
         (cond
           [(p2? x) 'left]
           [else 'right])]
        [else 'down]))
```

(c)

```
(define (q1c x)
  (cond
    [(cond
       [(p1? x) (p2? x)]
       [else   true])
     'up]
    [else 'down]))
```

The functions *q1a*, *q1b*, and *q1c* have contract *Num* → *Sym*. Each of the functions *p1?* and *p2?* is a predicate with contract *Num* → *Bool*. You do not need to know what these predicates actually do; the equivalent expressions should produce the same results for *any* predicates obeying the contract.

Test your code by inventing different combinations of predicates, but comment them out or remove them from the file before submitting it.

Place solution code in the file `cond.rkt`. This question does not require use of the design recipe.

2. Some credit cards and businesses offer AirMiles as a reward for using their card and/or purchasing from that particular business. A summary of what can be obtained is described below:

   - for "standard" credit cards, 1 AirMile is earned for every $15 spent at "sponsor" stores;
   - for "standard" credit cards, 1 AirMile is earned for every $20 spent at "non-sponsor" stores;
   - for "premium" cards, 1 AirMile is earned for every $10 spent at "sponsor" stores;
   - for "premium" cards, 1 AirMile is earned for every $15 spent at "non-sponsor" stores.

   Note that partial AirMiles are not given: for example, if $30 was spent at a non-sponsor store on a standard credit card, only 1 AirMile would be earned, rather than 1.5 AirMiles.

   In the file `airmiles.rkt`, write the function *calc-airmiles* which takes three parameters (in this order):

   - the dollar amount purchased, given as a decimal value, such as 10.72;
   - the symbol 'standard or 'premium to indicate the type of card used;
   - a boolean value which will be *true* if the store is a sponsor, and *false* otherwise.

   The function *calc-airmiles* should produce the number of AirMiles earned for the given purchase.

   Note that using the built-in Racket function *floor* will be helpful.

3. In the file `grades.rkt`, re-write the function *final-cs135-grade* from Assignment 1, with some minor modifications. The function will now consume five integers (in the following order):

   1. the first midterm grade,
   2. the second midterm grade,
   3. the final exam grade,
   4. the overall assignments grade,
   5. the participation grade.

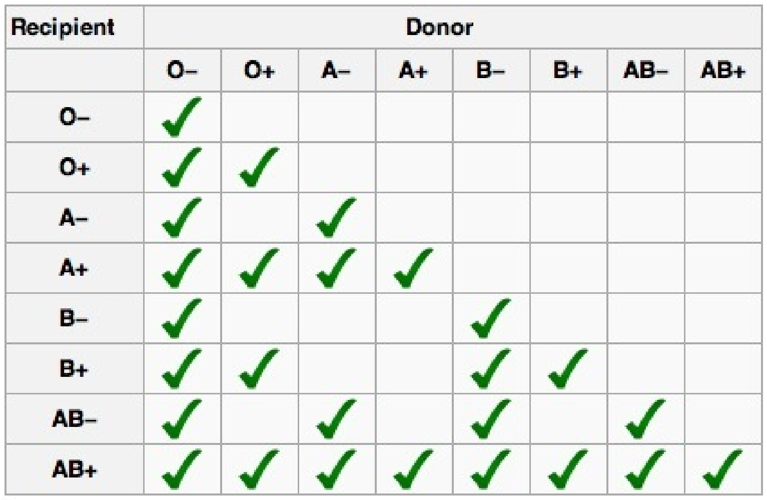
   You can assume that each of the above five integers is in the range between 0 and 100, inclusive. For the participation grade, assume that this is the computed mark, which has already taken the "top 75% of all responses" into account.

   The function *final-cs135-grade* should produce the final grade (as a number between 0 and 100, inclusive) in the course. If *either* the assignments grade *or* the weighted exam average is below 50 percent, *final-cs135-grade* should produce either the value 46 or the calculated final grade, whichever is *smaller*. If you had trouble completing your *final-cs135-grade* function in A1, you may use the posted solution as the starting point for your A2 solution, but you *must* clearly indicate this in your solution with a comment.

Note that you must use **cond** in this question: that is, a correct solution to the Bonus question on Assignment 1 will receive very few, if any, marks for this question. Moreover, you may not use *sgn* or *min* in your solution.

4. Everyone has a "blood type" that depends on many things, including the antigens they were exposed to early in life. There are many different ways to classify blood; one of the most common is by group: O, A, B, and AB. This is augmented by the "Rh factor" which is either "positive" or "negative". This yields a set of eight relevant types. We'll use the following symbols to represent them: 'O-, 'O+, 'A-, 'A+, 'B- 'B+, 'AB-, and 'AB+.

   If a person needs a blood transfusion, the type of the donor's blood is restricted to types which the recipient's body can accept. In the following chart (from `https://en.wikipedia.org/wiki/Blood_type`), a checkmark indicates which types are acceptable for each type of recipient. For example, a person with type 'O+ can donate to a person with type 'A+, but not to someone with a type 'B-. You can observe that 'O- is sometimes referred to as the *universal donor* and 'AB+ is sometimes referred to as the *universal acceptor*.



Recipient	Donor							
	O–	O+	A–	A+	B–	B+	AB–	AB+
O–	✓							
O+	✓	✓						
A–	✓		✓					
A+	✓	✓	✓	✓				
B–	✓				✓			
B+	✓	✓			✓	✓		
AB–	✓		✓		✓		✓	
AB+	✓	✓	✓	✓	✓	✓	✓	✓

   (a) Write the function *can-donate-to/cond?* which consumes a symbol denoting the donor's blood type as the first parameter and a symbol denoting the recipient's blood type as the second parameter. Produce *true* if the donor's blood type is acceptable for the recipient's blood type, according to the above chart, and *false* otherwise. *can-donate-to/cond?* should use **cond** expressions without using **and**, **or**, or *not*.

   (b) Write the function *can-donate-to/bool?* which is identical to *can-donate-to/cond?* except that it uses only a Boolean expression (i.e.: it does *not* have a **cond** expression).

As always, make sure you type all symbols exactly as they are written in the question. Your solution will be marked for style: in particular, writing 27 or so separate conditions will be penalized. Try to find patterns in the data and use those patterns to make your program easier to read and understand.

Place your solutions in the file `blood.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.

5. **5% Bonus**: Redo Question 4(b) (regarding blood types with booleans), except do not use **or** (or **cond**!) in your solution. Name your function *can-donate-to/bonus?* and place it in the file `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.