CS 135 Winter 2015
Troy Vasiga

Assignment: 2

Due:	Tuesday, January 20 at 9:00 pm
Language level:	Beginning Student
Files to submit:	`cond-a.rkt`, `cond-b.rkt`, `grades.rkt`, `blood-cond.rkt`, `blood-bool.rkt`, `pizza.rkt`
Warmup exercises:	HtDP 4.1.1, 4.1.2, 4.3.1, 4.3.2
Practise 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).**
- 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 the function names match ours.** You must use the basic tests to be sure. 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.
- For questions that use symbols, **you must type the symbols exactly as they are written in the question**. If you make spelling mistakes or use the wrong capitalization, you will get a low mark.

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 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.

   Below are two functions whose bodies use nested **cond** expressions. You must write new versions of these functions, each of which uses **exactly one cond**. Your versions must be equivalent to the originals—they should always produce the same answers as the originals, regardless of *x* or the definitions of the helper predicates *p1?*, *p2?* and *p3?*.

   (a)

```
(define (q1a x)
  (cond
    [(p1? x)
     (cond
       [(p2? x) 'red]
       [else 'blue])]
    [else
     (cond
       [(p2? x) 'green]
       [else (cond
               [(p3? x) 'pink]
               [else 'yellow])])]))
```

(b)

```
(define (q1b x)
  (cond
    [(cond [(and (p1? x) (< x 2015)) true]
           [else false])
     'brown]
    [(cond [(p1? x) (p2? x)]
           [else (p3? x)])
     (cond [(p1? x) 'black]
           [else 'white])]
    [else 'purple]))
```

The functions *q1a* and *q1b* have contract *Num -> Sym*. Each of the functions *p1?*, *p2?*, and *p3?* is a predicate with contract *Num -> Bool*. You do not need to know what these predicates actually do; your equivalent expressions should produce the same results for *any* predicates obeying the contract. You can test your work by inventing different combinations of predicates, but comment them out or remove them from the file before you submit it. Make sure that all your **cond** questions are "useful", that is, that there does not exist a question that could never be asked or that would always answer *false*.

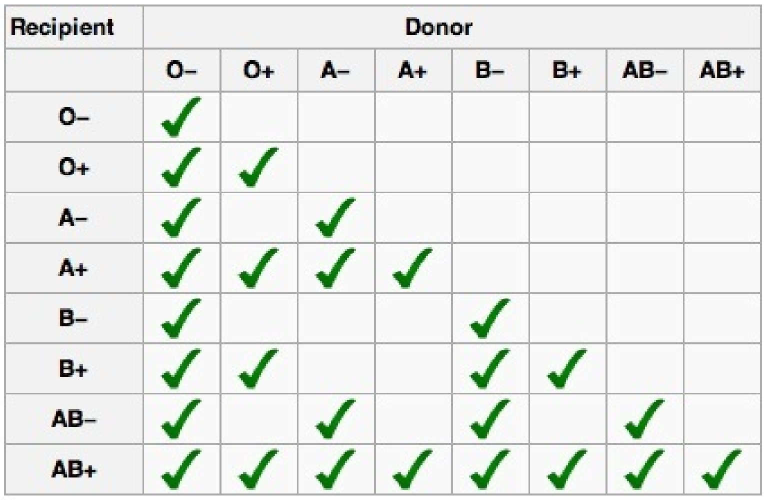
You may find that in some cases, having a single **cond** results in a simpler expression, and in others, having a nested **cond** results in a simpler expression. With practice, you will be able to simplify expressions even more complex than these.

Place your answers to parts (a) and (b) in two separate files, `cond-a.rkt` and `cond-b.rkt`, respectively. Use the same function name given in each question. For this question, you are not required to use the design recipe. You are not allowed to define any helper functions in this question. Make sure you spell your symbols correctly.

2. In the file `grades.rkt`, re-write the function *cs135-final-grade* from A1. Recall that it consumed three integers (in the following order): class participation grade, weighted assignment grade, and the weighted exam grade (all integers between 0 and 100, inclusive). *cs135-final-grade* should produce the final grade (as a number between 0 and 100, inclusive) in the course. If *either* the weighted assignments grade *or* the weighted exam average is below 50 percent, *cs135-final-grade* should produce either the value 46 or the calculated final grade, whichever is *smaller*. If you had trouble with your 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.

3. 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 only being a type which the recipients body can accept. In the following chart (from the "Blood Type" article on Wikipedia), 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 two symbols denoting the donor's blood type (as the first parameter) and 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*. Place your solutions in the file `blood-cond.rkt`.

(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). Place your solutions in the file `blood-bool.rkt`.

As always, make sure you type all symbols exactly as they are written in the question.

4. At Racket Pizza, you can order pizzas in 3 sizes: 'small, 'medium and 'large. Pizzas can also have standard toppings (onions, tomatoes, pepperoni, bacon, etc...) and premium toppings (chicken, feta cheese, etc...). The prices for the different pizza sizes and the toppings are below:

   - 'small is \$6
   - 'medium is \$8
   - 'large is \$9.50
   - Standard Toppings: \$1 each
   - Premium Toppings: \$1.50 each

   Racket Pizza currently has a variety of promotions (coupon codes) to attract Waterloo students. The following coupon codes are available in January 2015:

   - 'half-off gives you a half off your total order (a 50% discount).
   - 'big-eater gives you any pizza for \$18. The cost will be \$18 regardless of the size or the number of toppings.
   - 'supersize gives you a 'medium or 'large pizza for the price of a 'small pizza. All toppings are still regular price.
   - 'solo gives you a 'small pizza with 2 premium toppings for \$8. Only valid if the pizza is a 'small with 2 premium toppings and no standard toppings.

   Write the function *pizza-price* that produces the price of the pizza specified by the four values consumed in the following order: the size of the pizza (a symbol), the number of standard toppings, the number of premium toppings, and a coupon code (a symbol). The number of standard and premium toppings can be any natural number. Only one coupon code can be used, and the value 'none is used to represent no coupon code. If a coupon code doesn't match the order (e.g., 'solo only applies to size 'small with exactly 0 standard toppings and 2 premium toppings) or the coupon code provided is not listed above, then the coupon is ignored and the pizza is the regular price. Do not perform any rounding and do not add taxes. Make sure you watch your spelling: the symbols are case-sensitive. Place the function into the file `pizza.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.

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.