CS 135 Fall 2012
Ali,Fraser,Goldberg,Hoey,
Kaplan,Kogan,Petrick,Tompkins

Assignment: 2

Due:	Tuesday, September 25, 9:00 pm
Language level:	Beginning Student
Files to submit:	`cond.rkt`, `mystery.rkt`, `grades.rkt`, `rpsls.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. For example, your solutions must be entirely your own work, and your solutions will be marked for both correctness and good style. Good style includes qualities such as descriptive names, clear and consistent indentation, appropriate use of helper functions, and documentation (design recipe).

For this and all subsequent assignments you are expected to use 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**.

It is very important that the function names and parameters match ours. You must use the public tests to be sure. The names of the functions will be given 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.

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.

Below are two functions whose bodies are 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 [(not (p1? x))
         (cond [(p2? x) (f1 x)]
               [(p3? x) (f2 x)]
               [(not (p3? x)) (f3 x)])]
        [else
         (cond [(p3? x) (f2 x)]
               [else x])]))
```

(b)
```
(define (q1b x)
  (cond [(number? x)
         (cond [(cond [(p3? x) (p1? x)]
                      [else (p2? x)]) 'alfa]
               [(p3? x) 'bravo]
               [else 'charlie])]
        [(symbol? x)
         (cond [(symbol=? x 'delta) x]
               [else 'echo])]
        [else 'foxtrot]))
```

You do not need to know what the predicates *p1?*, *p2?* and *p3?* actually do; your equivalent expressions should produce the same results for *any* predicates that can consume the same inputs. You can test your work by inventing different combinations of predicates and different functions *f1*, *f2* and *f3*, but you must 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*. Tip: you may find it helpful to try re-ordering your questions.

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 in the file `cond.rkt`. Note that you are not required to use the design recipe in your solution to this problem. You are not allowed to define any helper functions in this question. As with A1 question 1, you will *not* be able to Run this DrRacket file.

2. Consider the following mystery predicate function that consumes three Booleans and produces a Boolean:

   ```
   (define (cond-mystery? a b c)
     (cond
       [a b]
       [else c]))
   ```

   Write the scheme function *bool-mystery?* so that it is equivalent to *cond-mystery?* except that it uses only a Boolean expression (i.e.: it does *not* have a **cond** expression).

   Tip: Try to solve this question yourself without help or discussing it with your peers, so you do not ruin the mystery for yourself.

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

3. In the file `grades.rkt`, write the following functions:
   (Note: if you had trouble with your A1, you may use the posted solution as the starting point for your A2 solution, but you *must* indicate this in your comments.)

   (a) Write a function named *participation-mark* that consumes three values in the following order: the total number of clicker questions asked in class, the number of clicker questions answered correctly (by a particular student) and the number of clicker questions answered incorrectly (by that same student). Review the course web page on "Grading & Marks" for details on how points are awarded. The result will be a fractional value between 0 (nothing answered) and 100 (at least 75% of the questions answered correctly). You may assume that the total number of clicker questions asked in class is greater than zero and divisible by 4.

   (b) Rewrite the function *final-cs135-grade* from A1. Recall that it consumed four integers (in the following order): the final exam grade, the first midterm grade, the second midterm grade, and the overall assignments grade (all integers between 0 and 100, inclusive). Add a fifth parameter, the participation grade, also an integer between 0 and 100, inclusive. *final-cs135-grade* should produce the final grade (as a fractional number between 0 and 100, inclusive) in the course. If *either* the overall assignments grade *or* the weighted average of the three exams is below 50 percent, *final-cs135-grade* should produce either the value 46 or the weighted average of the five provided grades, whichever is *smaller*. You may need to review the mark allocation in the course.

   (c) Rewrite the function *final-cs135-exam-grade-needed* from A1, adding a fourth parameter of the participation grade as in the previous question. Your function should produce the minimum grade required to achieve at least 50% on the weighted exam component and achieve a grade of at least 60% overall. If the grade required is *greater* than 100, or if the overall assignments grade is below 50, your function should produce `'impossible`. Note: *final-cs135-exam-grade-needed* can produce either a *Symbol*, or a *Num* – we will address this in more detail in Module 4, but for now, you should use the *Any* type in your contract.

4. The game of "Rock-Paper-Scissors" has existed for centuries and in many different forms. Recently, an extension of the game known as "Rock-Paper-Scissors-Lizard-Spock" (RPSLS) has become popular:

   `http://en.wikipedia.org/wiki/Rock-paper-scissors-lizard-Spock`

   RPSLS is a two player game, where each player simultaneously chooses a symbol from the set {rock, paper, scissors, lizard, spock}. If the two players choose the same symbol, it is a tie. Otherwise, one player is determined to be the winner according to the following list, where for each pairing below, the first defeats the second: (*i.e.:* Scissors defeats paper).

   - Scissors cuts paper
   - Paper covers rock
   - Rock crushes lizard
   - Lizard poisons Spock
   - Spock smashes scissors
   - Scissors decapitates lizard
   - Lizard eats paper
   - Paper disproves Spock
   - Spock vaporizes rock
   - Rock crushes scissors

   Write the function *rpsls* that consumes two symbols where each symbol is one of: { 'rock, 'paper, 'scissors, 'lizard, 'spock}. The first symbol will be the action for player 1, and the second symbol will be the action for player 2. Your function should produce one of three symbols: { 'tie, 'player1, 'player2 } that corresponds to the winner of the game of RPSLS. Note that all of the symbols and the name of the function are in lower case. Place your answers in the file `rpsls.rkt`.

This concludes the list of questions for which you need to submit solutions. Don't forget to always check your email for the public 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 DrScheme, *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 DrScheme's help desk.