# Assignment: 2

| | |
|---|---|
| Due: | Tuesday, January 19, 9:00 pm |
| Language level: | Beginning Student |
| Files to submit: | `cond.rkt`, `grades.rkt`, `rpsls.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.

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 three functions whose bodies are nested **cond** expressions. You must write new versions of these functions, each of which uses **exactly one cond**. The new 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) 'up]
       [else 'down])]
    [else
     (cond
       [(p2? x) 'jump]
       [else (cond
               [(p3? x) 'left]
               [else 'right])])]))
```

(b)
```
(define (q1b x)
  (cond
    [(cond [(and (p1? x) (< x 2016)) true]
           [else false])
     'heart]
    [(cond [(p1? x) (p2? x)]
           [else (p3? x)])
     (cond [(p1? x) 'spade]
           [else 'club])]
    [else 'diamond]))
```

(c)
```
(define (q1c x)
  (cond
    [(cond
       [(p1? x) (p2? x)]
       [else    (p1? x)])
     (cond
       [(p3? x) 'alpha]
       [else    'bravo])]
    [else 'charlie]))
```

The functions *q1a*, *q1b*, and *q1c* have contract *Num -> Sym*. Each of the functions *p1?*, *p2?*, etc. 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 this works by inventing different combinations of predicates, but comment them out or remove them from the file before submitting it. Make sure that all of the **cond** questions are "useful", that is, there exist no questions that could never be asked or that would always answer *false*.

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 solution code in the file `cond.rkt`. Use the same function name given in each question. This question does not require use of the design recipe. Do not include any helper functions in the solution code.

2. Assignment A01 required functions to calculate CS 135 grades. This assignment requires a function that calculates the final grade while checking the requirement that you must pass both the assignment and weighted exam portion of the course.

   In the file `grades.rkt`, write the function *cs135-final-grade* that consumes four numbers (in the following order):

   (a) the midterm grade,

   (b) the participation grade,

   (c) the overall assignments grade, and

   (d) the final exam grade.

   *cs135-final-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, *cs135-final-grade* should produce either the value 46 or the calculated final grade, whichever is *smaller*. As with the previous assignment, assume that all grades are percentages and are given as numbers between 0 and 100, inclusive.

   You may use your solution (or portions of your solution) from A01 or the posted solution to A01, but you must indicate so in your submission with a comment.

3. 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:

   https://en.wikipedia.org/wiki/Rock-paper-scissors

   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: (*e.g.:* 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`.

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 2016:

   - '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.

**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.