Assignment: 2
Due: Tuesday, January 17, 9:00 pm
Language level: Beginning Student
Files to submit: cond.rkt, rpsls.rkt, transportation.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

- Coverage: until M2-47
- 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, 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).

- Test data 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 the 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 can be tested for correctness.
- Do not send any code files by email to your instructors or tutors. 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
     [(> x 0) 'red]
     [(<= x 0) 'blue])

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

   (cond
     [(> 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) 'right]
     [else 'left])]]))

(b)

(define (q1b x)
  (cond
   [(cond
     [(p1? x) false]
     [else (p2? x))]
    'red]
   [else 'blue]))

(c)

(define (q1c x)
  (cond
   [(cond
     [(p1? x) (p2? x)]
     [else (p1? x))]
    (cond
     [(p3? x) 'down]
     [else 'up])]
   [else 'right]]))

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. 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:


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.
3. In the following chart, a checkmark indicates whether there is a direct transportation between 8 cites in the region. For example, there is a direct transportation from cite C to cite D where there is no direct transportation between C and F.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>H</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(a) Write the function `direct-transportation/cond?` which consumes two symbols (`orig` and `dest` representing two cites). Produce `true` if there is a direct transportation from `orig` to `dest`, according to the above chart, and `false` otherwise. `direct-transportation/cond?` should use `cond` expressions without using `and`, `or`, or `not`.

(b) Write the function `direct-transportation/bool?` which is identical to `direct-transportation/cond?` except that it uses only a boolean expression (i.e.: it does not have a `cond` expression).

Place both functions into the file `transportation.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.