CS 135 Fall 2016
Graham, Holtby, Petrick, Sakhnini

Assignment: 2

Due:	Tuesday, September 27, 9:00 pm
Language level:	Beginning Student
Files to submit:	`cond.rkt`, `bridgescore.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).**
- 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                    (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) 'down]
       [else    'up])]
    [else
     (cond
       [(p2? x) 'left]
       [else    'right])]))
```

(b)

```
(define (q1b x)
  (cond
    [(cond
       [(p1? x) false]
       [else    (p2? x)])
     'down]
    [else 'up]))
```

(c)

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

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. For this question you will submit the functions described in parts (a), (b) and (c) in a single file. Bridge is a card game with four players, where the players are divided into two pairs. After the cards are dealt, the players take part in a bidding auction that determines the contract for the hand. The last pair to say something in the auction, wins the bid and is known as the declarer. The other pair are known as the opponents.

   The contract that has been bid has two parts: a *level* and a *suit*. The *level* is a number between 1 and 7 representing the number of tricks the declarer is required to take. The *suit* is one of clubs, diamonds, hearts, spades, or no trump. At the end of the auction, the opponents may *double* the contract and the declarer can *redouble*. **A *redouble* can only happen after a *double* has been bid.**

   a) Write a function called *contract-points* that consumes an integer between 1 and 7 inclusive representing the *level* of the bid, and a symbol, one of 'clubs, 'diamonds, 'hearts, 'spades, or 'NT, representing the *suit* of the bid, and two Boolean values indicating if the contract has been *doubled* or *redoubled*. The function produces the contract points for that bid according to the following rules:

      - Clubs and diamonds are known as minor suits.
      - Hearts and spades are known as major suits.
      - Contract points table (from Wikipedia):

Denomination	**Contract Points Per Trick**		
	Undoubled	**Doubled**	**Redoubled**
No trump			
• First trick	40	80	160
• Subsequent tricks	30	60	120
Major suits	30	60	120
Minor suits	20	40	80

   For example,
   (*contract-points* 2 'clubs *true false*), representing a bid of 2 clubs, doubled, produces 80.
   (*contract-points* 4 'spades *true true*) representing 4 spades, redoubled, produces 480.
   (*contract-points* 3 'NT *false false*) representing 3 no trump, produces 100, (40 points for the first trick and 2 * 30 points for the remaining two tricks.)

b) After playing the hand, if the declarer did not take as many tricks as were bid, then it is said that the contract went down. The tricks short of the bid are known as *undertricks*. For example, 2 undertricks indicates that the declarer took 2 tricks less than what was bid. A declarer will lose points when the contract goes down. The number of points they lose will be determined by the number of undertricks, as well as their vulnerability. A declarer can be either *vulnerable* or *not vulnerable*.

Write a function called *penalty-points* that consumes a positive integer representing the number of undertricks followed by three Boolean values indicating if the contract is *vulnerable*, *doubled*, and/or *redoubled*. The function produces the penalty points (a negative number) according to the following table:

<table>
<tr><th rowspan="3">Number of Undertricks</th><th colspan="6">Points Per Undertrick</th></tr>
<tr><th colspan="3">Vulnerable</th><th colspan="3">Not Vulnerable</th></tr>
<tr><th>Undoubled</th><th>Doubled</th><th>Redoubled</th><th>Undoubled</th><th>Doubled</th><th>Redoubled</th></tr>
<tr><td>1st undertrick</td><td rowspan="3">100</td><td>200</td><td>400</td><td rowspan="3">50</td><td>100</td><td>200</td></tr>
<tr><td>2nd and 3rd, each</td><td>300</td><td>600</td><td>200</td><td>400</td></tr>
<tr><td>4th and each subsequent</td><td>300</td><td>600</td><td>300</td><td>600</td></tr>
</table>

For example, (*penalty-points* `3` *false true false*) represents a result of down 3 tricks, not vulnerable, doubled. It produces a score of -500 since there was a penalty of 100 points for the first undertrick, and 2 * 200 = 400 points for the second and third undertricks.

c) Duplicate bridge has a specific set of scoring rules. A declarer gets credit for contracts that are bid and made and is penalized for contracts that are not made. After the hand is played out, you record the *result*. If the declarer makes the contract, then the *result* is recorded as a positive integer that is greater than or equal to the number of tricks bid, representing the number of tricks actually taken. If the declarer did not make the contract, the *result* is recorded as a negative number that represents the number of *undertricks* (e.g. -2 indicates the result was 2 undertricks). Here is how the duplicate score is calculated:

If the contract was not made (i.e. the result is a negative number), produce the penalty points as described in part (b).

If the contract is made (i.e. the result is a positive number), the points are the sum of the *contract points* of the bid as calculated in part (a), plus the *game* bonus, plus the *insult* bonus (if applicable), plus the *overtrick* points (if applicable). Details of this scoring is on the next page.

- The *game* bonus is calculated as follows:
  - If the *contract points*, as described in part (a), of the bid are less than 100 points, there is a 50 point bonus.
  - If the *contract points* of the bid are at least 100 points, and the pair is not vulnerable, there is a 300 point bonus.
  - If the *contract points* of the bid are at least 100 points, and the pair is vulnerable, there is a 500 point bonus.
- The *insult* bonus is calculated as follows:
  - If the contract is *redoubled*, the players receive a 100 point bonus.
  - If the contract is *doubled* but not *redoubled*, the players receive a 50 point bonus.
- The *overtrick* points are calculated as follows:
  - Overtricks are the number of tricks taken beyond the bid contract. For example, if you bid 2 spades but made 5, then you have 3 overtricks.
  - Overtrick points table (from Wikipedia)

	Overtrick Points Per Trick	
Contract	**Vulnerable**	**Not Vulnerable**
Undoubled in		
• No trump	30	30
• Major suit	30	30
• Minor suit	20	20
Any doubled	200	100
Any redoubled	400	200

Write a function called *duplicate-score* that consumes a positive integer representing the level of the bid, a symbol, (one of 'clubs, 'diamonds, 'hearts, 'spades, or 'NT), representing the suit of the bid, an integer representing the result, (negative number for unmade contracts and positive number for made contracts), and three Boolean values indicating if the contract is *vulnerable*, *doubled*, and/or *redoubled*. The function produces the duplicate score according to the information described in this question.

For example: (*duplicate-score* 3 'NT 4 *false true false*) indicates the bid was 3 no trump, made 4 (i.e. 1 overtrick), not vulnerable and doubled. This produces 650: contract points of the bid are $80+60+60=200$, plus 300 points for the game bonus, plus 50 points for the insult bonus, plus 100 points for the overtrick.

You may use the function *contract-points* from part (a) and *penalty-points* from part (b) as a helper functions for this question if you want.

Notes:

- These rules follow the actual scoring rules for duplicate bridge with the exception of contracts at levels 6 and 7. These contracts are special and have extra rules for scoring not included in this question. You can probably find many scoring calculators online to confirm that your function is working properly. Here is a simple one: `http://www.rpbridge.net/cgi-bin/xsc1.pl`
- Since you are required to test the *contract-points* and *penalty-points* functions, it is not necessary for you duplicate all of these cases when testing the *duplicate-score* function. This means that for part (c) you only need one test for this function where the *result* is a negative number, and one test where the *result* is equal to the *bid*. The remaining tests for this function should have the *result* greater than the *bid*.
- This question includes three tables that each describe some portion of bridge scoring. There are mathematical relationships between some of the values within each table. A good solution should take advantage of these relationships. Solutions that use a separate question/answer pair for **each** entry in the table will not receive full marks for *Solution Complexity*

Place these functions into the file `bridgescore.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.