CS 135 Fall 2019
Becker, Clarke, Giesbrecht, Holtby, Lanctot, Reetz

Assignment: 02

Due:	Tuesday, September 24, 9:00 pm
Language level:	Beginning Student
Files to submit:	`cond.rkt`, `box-office.rkt`, `football.rkt`, `bonus-a02.rkt`
Warmup exercises:	HtDP 4.1.1, 4.1.2, 4.3.1, and 4.3.2
Practice exercises:	HtDP 4.4.1, 4.4.3, and 5.1.4

- Policies from Assignment 1 carry forward.
- This assignment covers concepts up to the end of Module 04. Unless otherwise specified, you may only use Racket language features we have covered up to that point.
- Frequently scan the OFFICIAL A02 post on Piazza for answers to frequently asked questions, *especially before you post to Piazza*.
- **It is very important that your function names, strings, and symbols match ours.** Basic tests results will catch many, but not necessarily all of these types of errors.
- A proportion of your marks will be for good coding style. You should be familiar with the style guide and apply relevant parts to each assignment. **Unless otherwise specified, for this and all subsequent assignments, you should include the design recipe as discussed in class.**
- A total of 80% of the marks are allocated as indicated at the beginning of each question. The remaining 20% are allocated in the following manner: 5% for Understandabilty (including formatting, naming, organization, helper functions), 5% for Purpose and Contract, and 10% for Testing (including examples).

Here are the assignment questions you need to submit.

1. (20 %) A **cond** expression can always be rewritten to produce *equivalent expressions*. These expressions always produce the same answer as the original when given the same arguments. 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 have 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 have 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.

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.

Below are four functions whose bodies are nested **`cond`** expressions. Write new versions of these functions subject to the following constraints:

- Each function uses **exactly one `cond`**.
- Each function always produces the same answer as the original regardless of the values of the parameters passed in.
- All of the **`cond`** questions are "useful", that is, there exists no question that could never be asked or that would always answer `false`.
- Each new function begins with the function name of the original but appends "-alt" to it. For example, your reimplementation of `q1a` would become `q1a-alt`.
- None of the functions shall use any helper functions.

Note: several of these functions contain **`cond`** expressions that immediately follow an **`else`**. This is terrible style, which is one reason why you will rewrite these functions. Furthermore, this question **does not require use of the design recipe**.

Hint: there are multiple possible approaches to simplify nested **`cond`** expressions, such as, truth tables.

As a warm-up, consider this function with the contract `Int Bool -> Int`

(a)
```
(define (q1a n a?)
  (cond
    [a?
      (cond
        [(>= n 0) (add1 n)]
        [else (sub1 n)])]
    [else 0]))
```

Now for three more difficult functions. For your solution, consider the fact that all functions below have the contract `Bool Bool Bool -> Sym`.

(b)
```
(define (q1b a? b? c?)
  (cond
    [a? (cond
          [b? 'elm]
          [(not c?) 'birch]
          [else 'cedar])]
    [else (cond
            [b? 'pine] ; Yuck. else followed by cond!
            [(not c?) 'birch]
            [else 'cherry])]))
```

(c)
```
(define (q1c a? b? c?)
  (cond
    [c? (cond
          [a? 'oak]
          [b? 'maple]
          [else 'willow])]
    [a? (cond
          [c? 'chestnut]
          [b? 'walnut]
          [else 'dogwood])]
    [(not b?) 'sumac]
    [else 'buckthorn]))
```

(d)
```
(define (q1d a? b? c?)
  (cond
    [(cond
       [c? b?]
       [else (not a?)]) (cond
                          [b? 'spruce]
                          [c? 'fir]
                          [else 'larch])]
    [else (cond
            [a? 'hazel]
            [else 'hickory])]))
```

Place solution code in the file `cond.rkt`.

2. (30 %) Movie studios have been scrambling to predict the success of their movies for decades. After watching hundreds of them, you believe you have found the perfect formula to predict box-office success. Here are your thoughts:

   - Names matter! The movie watching audience does not like boring movie titles or movie titles that take too long to read. As a result, all movies whose title is shorter than 10 characters (including spaces) receive a bonus of $25M (short for $25 Million), and movies that start with "The" receive a penalty of $50M.
   - The studio! "Marvel" movies will earn an extra $500M in the box office, "DC" movies receive a penalty of $250M; any other studio will not receive any bonus / penalty.

- Have famous actors! Each famous actor in a movie increases box-office profits by $50M!
- Explosions! The more the better! The following table shows your estimate on how the number of explosions affects box-office profits:

Number of explosions	Profits
0	-$20M
1	-$14M
2	-$8M
3	-$2M
4	+$4M
...	...

Create a function called `box-office-profits`. This function will consume four parameters:

- the name of the movie (as a `Str`),
- the name of a studio (also as a `Str`),
- the number of famous actors (as a `Nat`), and
- the number of explosions (also as a `Nat`).

It will then produce the predicted box-office profits (as an `Int`).

For example, you predict that Marvel's new "Avengers: more endgames" with 4 famous actors and 50 explosions will earn $980M at the box office `(check-expect (box-office-profits "Avengers: more endgames" "Marvel" 4 50 ) 980)`. By contrast, you predict that DC's new "Superman v Superman", with 2 famous actors and 100 explosions will only earn $430M `(check-expect (box-office-profits "Superman v Superman" "DC" 2 100 ) 430)`. As a final example, you predict "The Slog" by New Line Cinema with no famous actors and without any explosions will lose $45M at the box office `(check-expect (box-office-profits "The Slog" "New Line Cinema" 0 0 ) -45)`.

Place your function in the file `box-office.rkt`.

Hint: you might have to use other string fuctions besides the ones presented on the lecture slides (e.g., `substring`).

3. (30 %) Football season just started! The rules of the game can, however, be quite complicated. Let's see if we can make sense of one particular penalty: "Intentional Grounding".

   Quarterbacks commit this penalty when they were throwing the ball and all of the following are true:

   - They were under imminent pressure.
   - They threw from inside the pocket.
   - They did not throw the ball towards a team member.

Note: you are not allowed to use `cond` to implement the predicates in Q3a and Q3b! Instead, may only use Boolean operators (i.e., `and`, `or`, and `not`). By definition, predicates produce Boolean values; therefore, using `cond` would imply that the answers to the questions must be Boolean values (i.e., `true` or `false`) as well. As a result, there is no need to use a `cond` structure, and it is better practise to use results from the questions directly.

(a) Create a predicate called `intentional-grounding?`. This predicate will consume three parameters (all `Bool`):

- whether the quarterback was under imminent pressure,
- whether the quarterback was inside the pocket, and
- whether the quarterback threw the ball towards a team member.

For example, throwing the football under imminent pressure from outside the pocket not towards a team member would not count as Intentional Grounding `(check-expect (intentional-grounding? true false false) false)`.

Hint: you do not have to know anything about football to answer this question! Abstract from the concrete context of the problem (football) and transfer it into a language you are more familiar with (i.e., Boolean logic).

(b) Unfortunately, the "Intentional Grounding"-rules is actually more complex:

Quarterbacks commit this penalty when they were throwing the ball and all of the following are true:

- They were under imminent pressure.
- They threw from inside the pocket.
- They did not throw the ball towards a team member, or threw the ball towards a team member who is not an "Eligible Receiver".

Create a predicate called `intentional-grounding-correct?`. This predicate will consume four parameters (all `Bool`):

- whether the quarterback was under imminent pressure,
- whether the quarterback was inside the pocket,
- whether the quarterback threw the ball towards a team member, and
- whether the team member was an Eligible Receiver.

For example, throwing the football under imminent pressure from inside the pocket to a teammate who is an Eligible Receiver would not count as Intentional Grounding `(check-expect (intentional-grounding-correct? true true true true) false)`. On the other hand, it would be a penalty if the teammate was not an Eligible Receiver `(check-expect (intentional-grounding-correct? true true true false) true)`.

(c) Intentional Grounding carries several penalties. The type of penalty depends on where on the field the Intentional Grounding infraction occurred: in the endzone the penalty is a so-called "Safety", while outside the endzone the penalty is 10 yards.

Create a function called `intentional-grounding-penalty`. This function will consume five parameters (all `Bool`):

- whether the quarterback was under imminent pressure,
- whether the quarterback was inside the pocket,
- whether the quarterback threw the ball towards a team member
- whether the team member was an Eligible Receiver, and
- whether the infraction occurred in the endzone.

It will then produce three possible results (all `Sym`): `'Safety` if the Intentional Grounding occurred in the endzone, `'10yds` if the Intentional Grounding occurred outside the endzone, or `'None` if there was no Intentional Grounding.

Hint: remember that you are allowed to call any of the functions that you have implemented previously (i.e., for part Q3a and Q3b). By reusing existing (and hopefully well-tested!) helper-functions, you can solve Q3c in only a few lines of code.

Place your solutions to questions Q3a - Q3c in the file `football.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.

Bonus (5%): Write a function `date->day-of-week` which consumes a natural number and produces a symbol corresponding to the day of the week, according to the Gregorian calendar.

An integer encodes a date as follows:

- The leading four digits correspond to the year
- The two digits after that correspond to the month of the year
- The final two digits correspond to the day of the month

The function should return one of seven symbols: `'Monday`, `'Tuesday`, `'Wednesday`, `'Thursday`, `'Friday`, `'Saturday`, or `'Sunday` according to the day of the week the consumed date corresponds to.

To give an example: consider Monday, July 1, 1867. The equivalent call would be `(date->day-of-week 18670701)`, which produces `'Monday` as output. Similarly, for September 24, 2019, `(date->day-of-week 20190924)` the function produces `'Tuesday`.

You can assume that all dates are correct. This means that your function does not have to check for invalid input, such as, April 65, 1900 (`(date->day-of-week 19000465)`). You can also assume that no dates before January 1, 1753 AD, will be tested.[1]

You may only use the Racket constructs we have discussed in lecture so far, and built-in mathematical functions. You may not use Racket's date functions.

[1] Fun fact: the Gregorian calendar was not fully established in the United Kingdom and her colonies before that date (`https://en.wikipedia.org/wiki/Adoption_of_the_Gregorian_calendar`, as retrieved on 2019-09-09).

It is acceptable (and encouraged) to consult rules or algorithms for computing the day of the week given a date, but it is not acceptable to copy and paste code you have found. You must cite any resources you use in your Racket file. Just like before, it might be helpful to define multiple functions.

Put your solution in `bonus-a02.rkt`.

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.