Fall 2022
Becker, Kamal, Nasr Esfahani, Tompkins, Van Leeuwen, Vasiga

Assignment: 2

Due: Tuesday, September 27, 2022 9:00 pm
Coverage: End of Module 04 (see Coverage note below)
Language level: Beginning Student
Files to submit: `median.rkt`, `cond.rkt`, `blood.rkt`, `box-office.rkt`

Assignment policies, Correctness and Readability

Please refer to the policies, correctness and readability sections at the top of A01.

Grading

- **You will not receive marks for** *A*02 **unless you have earned full marks for A00 before the** *A*02 **due date.** Assignments may be submitted as often as desired up to the due date.
- Your solutions for assignments will be graded on both correctness and on readability. This assignment will be auto-graded and hand marked.
- For this assignment, you will need the design recipe for all the questions except Questions 1 and 3.
- For each function in Questions 2, 4 and 5, you are required to submit the design recipe. Every design recipe has examples that are illustrated with the use of `check-expect`s. You are required to submit these examples by **Friday, September 23 at 8:00AM**. Note: this is before the due date of assignment A02. This early submission component encourages you to start thinking about the assignment early. To submit the examples, simply submit the files that you will eventually submit to MarkUs for each question, with the examples. **You do not need to have the functions themselves implemented, only the `check-expect`s.** This grading component will be worth approximately 5% of your grade for each of these questions. You are encouraged to also complete and submit the purpose statement for each function. More details about the design recipe grading will be specified on the A02 Official Post on Piazza.

Here are the assignment questions that you need to submit.

1. **Complete all the required stepping problems in Module 4a and Module 4b** at

   `https://www.student.cs.uwaterloo.ca/~cs135/assign/stepping/`

   You should refer to the instructions from A01 Question 1 for the stepper question instructions.

2. The following function produces the median of 3 numbers:

```
;; median-of-3:  Num Num Num -> Num
(define (median-of-3 a b c)
  (cond
    [(or (and (<= b a) (<= a c)) (and (<= c a) (<= a b))) a]
    [(or (and (<= a b) (<= b c)) (and (<= c b) (<= b a))) b]
    [(or (and (<= b c) (<= c a)) (and (<= a c) (<= c b))) c]))
```

Simplify this function so that it reduces the number of inequality comparisons that are performed. Inequalites include `>`, `>=`, `<` and `<=`. For example, the expression `(< a b)` performs a single comparison for numbers `a` and `b`. However, `(< a b c)` performs two comparisons and produces the same output as `(and (< a b) (< b c))`.

In addition to the above inequality operators, you may only use **`define`**, **`cond`**, `=`, **`and`**, **`or`**, or `not`. Note, **`else`** is allowed whenever **`cond`** is allowed.

Your simplified function should perform 8 or fewer inequality comparisons. There may be several solutions depending on how you decide to write the function. For your chosen approach, you should have as few question/answer pairs as possible and it should not be possible to further simplify the questions.

The name of the simplified function should be `median-of-3-simple`.

Place your solution code in the file `median.rkt`.

Reminder: The Grading section at the top of this document states which questions require the design recipe.

3. 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.)

Many 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.

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

- Each function uses **exactly one** `cond`.
- There may be several solutions depending on how you decide to write the function. For your chosen approach, you should have as few question/answer pairs as possible and it should not be possible to further simplify the questions.
- 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 has the same function name as the original in this question.
- The functions `q3a`, `q3b`, and `q3c` all have the contract `Bool Bool` → `Sym`.

Full marks will be awarded for solutions which have the fewest number of questions as possible in the **cond**.

(a)
```
(define (q3a p1? p2?)
  (cond
    [p2? (cond [p1? 'left]
               [else 'down])]
    [else (cond [p1? 'up]
                [else 'right])]))
```

(b)
```
(define (q3b p1? p2?)
  (cond [p1? (cond
               [p2? (cond
                      [p1? 'up]
                      [p2? 'down]
                      [else 'right])]
               [else (cond
                       [p2? 'down]
                       [else 'up])])]
        [(and p1? false) (cond
                           [p2? 'left]
                           [else 'right])]
        [else 'down]))
```

(c)
```
(define (q3c p1? p2?)
  (cond [(cond [p1? p2?]
               [else true])
         'up]
    [else 'down]))
```

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

4. Everyone has a "blood type" that depends on many things, including the antigens they were exposed to early in life. There are many different ways to classify blood; one of the most common is by group: O, A, B, and AB. This is augmented by the "Rh factor" which is either "positive" or "negative". This yields a set of eight relevant types. We'll use the following symbols to represent them: `'O-`, `'O+`, `'A-`, `'A+`, `'B-` `'B+`, `'AB-`, and `'AB+`.

   If a person needs a blood transfusion, the type of the donor's blood is restricted to types which the recipient's body can accept. In the following chart, a checkmark indicates which types are acceptable for each type of recipient. For example, a person with type `'O+` can donate to a person with type `'A+`, but not to someone with a type `'B-`. You can observe that `'O-` is sometimes referred to as the *universal donor* and `'AB+` is sometimes referred to as the *universal recipient*.

Recipient \ Donor	O-	O+	A-	A+	B-	B+	AB-	AB+
O-	√							
O+	√	√						
A-	√		√					
A+	√	√	√	√				
B-	√				√			
B+	√	√			√	√		
AB-	√		√		√		√	
AB+	√	√	√	√	√	√	√	√

   (a) Write the function `can-donate-to/cond?` which consumes a symbol denoting the donor's blood type as the first parameter and a symbol denoting the recipient's blood type as the second parameter. Produce `true` if the donor's blood type is acceptable for the recipient's blood type, according to the above chart, and `false` otherwise. `can-donate-to/cond?` **must** use **`cond`** expressions **without** using **`and`**, **`or`**, or `not`. Your solution will be marked for style. You should not hard-code every possible combination of blood types in your solutions. Try to find patterns in the data and use those patterns to make your program easier to read and understand.

   (b) Write the function `can-donate-to/bool?` which is identical to `can-donate-to/cond?` except that it uses **only** a Boolean expression. That is, it does *not* have a **`cond`** expression.

   As always, make sure you type all symbols exactly as they are written in the question. Place your answers in the file `blood.rkt`.

5. Movie studios have been scrambling to predict the success of their movies for decades. After watching hundreds of them, you believe that 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! Movies without explosions are boring!!!

  A movie with zero explosions receives a penalty of $20M. For each explosion in a movie, its box office profits increase by $6M.

  For example, if a movie has two explosions in it, it will receive a penalty of $8M. However, if it has four explosions, it will receive a bonus of $4M.

Write the 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`) in millions. Note that the profit is not always a positive number.

For example, your function will predict that Marvel's new "Thor: Love and Thunder" with 4 famous actors and 50 explosions will earn $980M at the box office. It has neither a penalty nor a bonus for the title. Coming from the Marvel studio, it receives a bonus of $500M. Four famous actors nets an additional $200M and 50 explosions adds $280M.

```
(check-expect (box-office-profits "Thor: Love and Thunder" "Marvel" 4 50 ) 980).
```

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

Hints:

- Consider writing more than one helper function for this question. It is in your best interest to keep the main function as simple as possible.
- Consult slide M04-37 for the complete list of functions available to solve this problem. Check out the string functions, in particular.

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 want to take a look at exceptions in DrRacket's help desk.