CS 135 Fall 2017
Becker, Goldberg, Kaplan, Nijjar, Tompkins, Vasiga

Assignment: 3

Due:	Tuesday, October 3rd, 9:00 pm
Language level:	Beginning Student
Files to submit:	`nutrition.rkt`, `creditcheck.rkt`, `battle.rkt`, `battle-bonus.rkt`
Warmup exercises:	6.3.2, 6.4.1, 7.1.2
Practice exercises:	6.3.3, 6.4.2, 6.4.3, 6.5.2, 7.1.3, 7.5.1, 7.5.2, 7.5.3

- **Make sure you read the OFFICIAL A03 post on Piazza** for the answers to frequently asked questions.
- Policies from Assignment 2 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.**
- The basic and correctness tests for all questions will always meet the stated assumptions for consumed values.
- You must use *check-expect* for both examples and tests of functions that produce exact values.
- You may use the **cond** special form. You are **not** allowed to use **if** in any of your solutions.
- **It is very important that your 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 in that file can be tested for correctness.
- Do not send any code files by email to your instructors or other course staff (e.g,. ISAs). 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. In this question you will perform step-by-step evaluations of Racket programs, by applying substitution rules until you either arrive at a final value or you cannot continue. You will use an online evaluation tool that we have created for this purpose.

   To begin, visit this web page:

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

   Note: the use of `https` is important; that is, the system will not work if you omit the `s`. This link is also in the table of contents on the course web page.

   You will need to authenticate yourself using your Quest/WatIAm ID and password. Once you are logged in, try the questions in the "Warmup questions" category under "CS 135 Assignment 3," in order to get used to the system. Note the "Show instructions" link at the bottom of each problem. **Read the instructions before attempting a question!**

   When you are ready, complete the six stepping problems in the "Assignment questions" category, using the semantics given in class for Beginning Student. You can re-enter a step as many times as necessary until you get it right, so keep trying until you completely finish every question. All you have to do is complete the questions online—we will be recording your answers as you go, and there is no file to submit. The basic tests for this assignment will tell you whether or not we have a record of your completion of the stepper problems. **Note that you are not done with a question until you see the message** `Question complete!` You should see this once you have arrived at a final value and clicked on "simplest form" (or "Error," depending on the question).

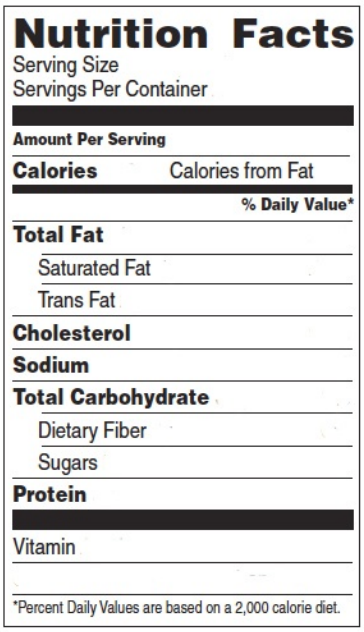
   You are stepping through the given expressions assuming that all function and constant definitions that appear above the expression exist. This means that you are not required to do any simplification of the constant definitions as part of the stepping.

   You should **not** use DrRacket's Stepper to help you with this question for several reasons. First, as mentioned in class, DrRacket's evaluation rules are slightly different from the ones presented in class, but we need you to use the evaluation rules presented in class. Second, in an exam situation, you will not have DrRacket's Stepper to help you, and there will definitely be step-by-step evaluation questions on at least one of the exams.

   Note: If you get stuck on a stepping question, **do not post to Piazza requesting the next step.** This is a violation of the academic integrity policy. Review the substitution rules carefully from Module 3 to try to solve the problem yourself. If you still cannot find your error, then you are encouraged to ask a question in person during office hours. As a last resort, you may make a private post to Piazza describing where you are stuck. Course staff will provide guidance directing you to the next step, but they will not give you the answer.

2. Nearly all packaged foods in Canada are required to display a nutritional facts table. You can find more about this program online. However, you do not require full details about nutritional facts to complete this question.

   http://Canada.ca/NutritionFacts



   This question uses the following *simplified* structure to represent a nutritional facts table:

```
(define-struct nutri-fact (name serving fat carbs sugar protein))
;; A Nutri-Fact is a (make-nutri-fact Str Num Num Num Num Num)
;; requires: 0 < serving
;;           fat + carbs + protein <= serving
;;           0 <= sugar <= carbs
;;           0 <= fat, protein
```

   All of the numerical fields are measured in grams. Note that the fields for the nutrients (`fat`, `carbs` (short for *carbohydrates*), `sugar` and `protein`) correspond to the number of grams per `serving`. In addition, `carbs` includes any amount of `sugar`. For example, consider the following `Nutri-Fact`:

```
(make-nutri-fact "Honey Nut Cheerios" 29 1.5 23 9 2)
```

   For *Honey Nut Cheerios (HNC)* with a `serving` size of 29g, there are 1.5g of `fat`, 23g of `carbs` and 2g of `protein`. Of the 23g of `carbs`, 9g of those are `sugar`.

You must include the above structure and data definition at the beginning of your solution file `nutrition.rkt`, along with the following functions:

(a) `my-nutri-fact-fn` is a **template function** that consumes a `Nutri-Fact` and produces `Any`. Note that for this template function, you only need to provide a contract and a body.

(b) `resize` consumes a `Nutri-Fact` and a new `serving` size (a positive number) and produces a new `Nutri-Fact` with the new `serving` size and all of the other numerical nutrient fields (*e.g.,* `fat`) changed to reflect the new `serving` size. For example, if the `serving` size of HNC is changed from 29g to 58g, then there are now 3g of `fat` per `serving`.

(c) `calories` consumes a `Nutri-Fact` and produces the number of calories there are in a `serving`. Each gram of `fat` is 9 calories, each gram of `carbs` is 4 calories, and each gram of `protein` is also 4 calories. For HNC, the number of calories per `serving` is 113.5. Note that the calories shown in the "real world" are rounded (often to the closest multiple of 10) but you should not perform any rounding.

(d) `choose-for-diet` consumes two `Nutri-Fact`s and produces the one that is the most appropriate for your friend's diet (or, if there is no difference for their diet, the first one). When comparing the two `Nutri-Fact`s, the `serving` size should not matter, so you should consider the quantity of a nutrient as a proportion of the entire `serving` size. Your friend is trying to reduce the amount of `sugar` in their diet, so they would prefer the one with the smallest proportion of `sugar`. For HNC, the proportion of `sugar` is 9/29 (9g out of 29g – which incidentally, is a lot). If the proportion of `sugar` is the same, they prefer the one with the larger proportion of `protein`. If they have both the same proportion of `sugar` and `protein`, then they prefer the one with smallest proportion of `carbs`, and if still tied, the one with the smallest proportion of `fat`.

(e) `valid-nutri-fact?` consumes an arbitrary value (`Any`) and produces `true` if it is a valid `Nutri-Fact` according to the provided data definition (and `false` otherwise).

3. This question uses the following structures to represent a date, a credit card transaction, and a customer's account information:

```
(define-struct date (year month day))
;; A Date is a (make-date Nat Nat Nat)
;; requires: year/month/day corresponds to a valid date
;;           (in the Gregorian calendar)

(define-struct transaction (tdate amount category))
;; A Transaction is a (make-transaction Date Num Sym)

(define-struct account (name expires limit threshold exception))
;; An Account is a (make-account Str Date Num Num Sym)
;; requires: 0 < threshold < limit
```

A credit card `transaction` stores the date of the transaction (`tdate`) along with the `amount` of the transaction (in dollars) and the `category` of the transaction. The `category` may be something like `'food`, `'gas` or `'entertainment`.

An `account` stores the `name` of the customer, the date the credit card `expires` and the largest possible transaction amount (`limit`). The account also has a `threshold` amount so the customer can receive an *alert* if a single transaction *exceeds* the `threshold` amount. However, customers can add an `exception` category to their account so they do not receive alerts for transactions in the `exception` category. For example, consider the following `account`:

```
(make-account "Bob Smith" (make-date 2017 10 31) 100 25 'food)
```

Bob Smith's credit card account expires on October 31st, 2017. Transactions exceeding $100 will not be approved. Transactions exceeding $25 will trigger an alert for Bob, *unless* the transaction is in his exception category of `'food`, in which case no alert will be sent.

You must include the above structures and data definitions at the beginning of your solution file `creditcheck.rkt`, along with the following functions:

(a) `date<=?` consumes two `Date`s and produces `true` if the first `Date` occurs before the second `Date` or they are the same `Date`. Otherwise, the function produces `false`.
Note: You may be familiar with some "tricks" to convert a `Date` to single number (or string) to make comparisons easier. For example, you could convert the `Date` October 31, 2017 to the number 2017.304 because October 31st is the 304th day in the year. Alternatively you could convert the `Date` to 20171031 or `"2017-10-31"`. This approach (converting a structure to a single number or string) can also be seen in the sample submission in the style guide (`time->seconds`). For this question you are **not** allowed to convert the `Date` to a single number (or string). Instead, use a combination of `cond` and/or Boolean functions.

(b) `approve?` consumes a `Transaction` as the first parameter and an `Account` as the second parameter. `approve?` produces `true` if the transaction `amount` does not exceed the account `limit` and the transaction date (`tdate`) is not after the date the card `expires` (and `false` otherwise).

(c) `alert?` also consumes a `Transaction` as the first parameter and an `Account` as the second parameter. `alert?` produces `true` if the transaction is approved, but the transaction amount exceeds the `threshold` and the transaction `category` is not the `exception` category in the `Account` (and `false` otherwise).

4. The elegant card game *Schotten Totten* designed by mathematician Dr. Reiner Knizia is a modern classic (you may also know the game by its nearly identical variant named *Battle Line*). You can read the rulebook for the game online, but it is not necessary and it contains many rules not pertinent to this assignment question (*e.g.,* the Tactic variant).

   http://www.iellogames.com/downloads/schottentotten_rulebook.pdf

   In the game, players have nine battles, but for this question we are only concerned with a single battle. To battle, each player builds a `Hand` of three `Card`s. The `Card`s are not from the traditional 52-card deck, but rather a specialized 54-card deck. Each `Card` has a `strength` (1 . . . 9) and one of six `colour`s (red, yellow, green, blue, purple, brown). The following data definitions describe a `Card` and a `Hand`:

```
(define-struct card (strength colour))
;; A Card is a (make-card Nat Sym)
;; requires: 1 <= strength <= 9
;;           color is one of: 'red 'yellow 'green 'blue 'purple 'brown

(define-struct hand (c1 c2 c3))
;; A Hand is a (make-hand Card Card Card)
```

   There are three special *attributes* (or "combinations") that a `Hand` may have:

   - The three `Card`s have the same `colour`.
   - The three `Card`s are *"three-of-a-kind"* (they have the same `strength` and their colours are not relevant).
   - The three `Card`s form a *run*, which is when the three `strength` of the `Card`s are sequential (*e.g.,* 3-4-5). Note that a run can appear in the `Hand` in any order (*e.g.,* 5-3-4).

   In addition, every `Hand` also has a *sum*, which corresponds to the sum of the three `strength`s.

   To determine who wins a battle, the two `Hand`s are compared and the `Hand` with the best *attributes* wins. If both `Hand`s have the same *attributes*, then the `Hand` with the larger sum wins. If there is still a tie, then the player who completed their `Hand` first wins (for this assignment, that is `'player1`).

The rankings of the *attributes* from best to worst are:

i **colour-run**: The `Card`s all have the same `colour` **and** form a run

ii **three-of-a-kind**: The `Card`s all have the same `strength`

iii **colour**: The `Card`s all have the same `colour` (but do not form a run)

iv **run**: The `Card`s form a run (but do not have the same `colour`)

v **sum**: The `Card`s have no special *attributes*, and so they are evaluated only according to their sum.

You must include the above structures and data definitions at the beginning of your solution file `battle.rkt`, along with the following function:

`battle` consumes two `Hand`s and produces `'player1` if the either the first `Hand` defeats the second `Hand` or there is a tie. Otherwise, it produces `'player2`.

For our tests, we will ensure that all six `Card`s in the two `Hand`s are unique. You do not have to check for invalid inputs.

In the previous question you were not allowed to convert a `Date` to a single number (or string) to make comparisons easier. For this question you may use that approach if you wish (*i.e.,* convert a `Hand` to a single number or string). However, it is not necessary to use that approach to solve this problem.

Also, for those familiar with poker, note that the `Hand` rankings are very similar but there is one important difference. In poker, when two `Card`s have the same *attributes*, ties are broken by the highest `Card` (and then the second highest `Card`, and so forth). In *Schotten Totten*, ties are broken by the sum.

If you are interested in playing *Schotten Totten*, MathSoc has a large collection of modern board and card games you can sign out with your student card. They also hold weekly game nights every Thursday evening at 6:30pm in the C&D (3rd floor MC). All students are welcome, and they often have free snacks. Some CS instructors have even been known to attend, and there will be a "Games with Profs" event scheduled later this term.

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.

---

5. **BONUS QUESTION (5%)**

Imagine that you are playing a game of *Schotten Totten* and your opponent has completed their `Hand` and you have already played one of the three `Card`s in your `Hand`. You want to construct a `Hand` that will defeat your opponent.

Here are some examples.

- Your opponent has played 5 red, 8 red, 9 red and you have already played 6 blue. You can defeat their `Hand` with three `Card`s that are the same `colour` but have a greater sum (*e.g.,* 6 blue, 8 blue, 9 blue). For this example, there are many possible `Hand`s that you could construct to defeat your opponent such as a three-of-a-kind (*e.g.,* 6 blue, 6 yellow, 6 green).
- Your opponent has played 7 red, 8 red, 9 red and you have already played 9 blue. You cannot defeat their `Hand`. The best `Hand` you could construct would be another colour-run with the same sum (*e.g.,* 7 blue, 8 blue, 9 blue) but because they completed their `Hand` before you, they would win the tie.
- Your opponent has played 3 red, 3 blue, 3 purple and you have already played 2 blue. You cannot defeat their `Hand`. You could construct a three-of-a-kind with 2s, but that will not defeat their 3s. You cannot construct a colour-run because your 2 blue would require the 3 blue to complete the colour-run and your opponent has the 3 blue in their `Hand`.

In the file `battle-bonus.rkt`, write a function `find-winner` that consumes your played `Card` and your opponent's `Hand` and produces either a `Hand` that defeats your opponent's `Hand`, or `false` if no such `Hand` is possible. You can assume that the `Hand` provided to you has no duplicates, and the `Card` provided does not appear in the `Hand`. The `Hand` you produce must contain your played `Card` (as `c1`) and two *unique* `Card`s that are one of the valid 54 `Card`s, but do not duplicate any of the `Card`s in your opponent's `Hand` (or your played `Card`). As in the first example above, there may be many possible solutions and any of them would be acceptable.

If you wish to re-use your solution or helper functions from `battle.rkt` simply "cut & paste" them into your `battle-bonus.rkt` file. No design recipe is required for the bonus. Solutions that we deem are too complex or inelegant (*e.g.,* enumerating every possible card combination) will not receive any marks. You may **not** use recursion to solve this problem.