CS 135 Fall 2015
Akinyemi, Heinle, Holtby, Istead, Roegiest, Tompkins

Assignment: 4

Due:	Tuesday, October 13, 2015 9:00pm
Language level:	Beginning Student
Allowed recursion:	(Pure) Structural recursion
Files to submit:	`numbers.rkt`, `rockets.rkt`, `cards.rkt`, `cards-bonus.rkt`
Warmup exercises:	HtDP 8.7.2, 9.1.1 (but use box-and-pointer diagrams), 9.1.2, 9.5.3
Practise exercises:	HtDP 8.7.3, 9.5.4, 9.5.6, 9.5.7

- You should note a new heading at the top of the assignment: "Allowed recursion". In this case the heading restricts you to pure structural recursion, i.e., recursion that follows the data definition of the data it consumes. See slide 05-38.
- Provide your **own examples** that are distinct from the examples given in the question description of each function.
- Of the built-in list functions, you may only use *cons*, *first*, *rest*, *empty?*, *cons?*, *length*, and *member?*. You cannot use other functions such as *list*, including in examples and tests.
- Your solutions must be entirely your own work.
- For this and all subsequent assignments, you must include the design recipe for all functions, including helper functions, as discussed in class. In addition, you must include a data definition for all user-defined types. Unless otherwise stated, if *X* is a known type then you may assume that (*listof X*) is also a known type. You do not need to provide template functions unless explicitly stated.
- Solutions will be marked for both correctness and good style as outlined in the Style Guide.
- 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 and number of parameters match ours. You must use the basic tests to be sure. 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.

Here are the assignment questions you need to submit.

1. Perform the assignment 4 questions using the online evaluation "Stepping Problems" tool linked to the course web page and available at

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

   The instructions are the same as those in assignment 3; check there for more information if necessary. Reminder: You should not use DrRacket's Stepper to help you with this question. First, DrRacket's evaluation rules are slightly different from the ones presented in class, and you must use the ones presented in class. Second, when writing an exam you will not have the Stepper to help you. Third, you can re-enter steps as many times as necessary to get them correct, so you might as well maximize the educational benefit.

2. This problem deals with lists of numbers.

   (a) Write a function *sum-list* which consumes a list of numbers, and produces the sum of that list. The sum of an empty list is 0. For example (*sum-list* (*cons* 1 (*cons* 2 (*cons* 3 *empty*)))) ⇒ 6.

   (b) Write a function *divide-list* which consumes a list of numbers and an additional non-zero number, and divides each element in the list by the consumed number.
   Example:
   (*divide-list* (*cons* 1 (*cons* 2 (*cons* 3 *empty*))) 3) ⇒ (*cons* 1/3 (*cons* 2/3 (*cons* 1 *empty*)))

   (c) Write a function *normalize-list* which consumes a list of positive numbers, and produces the list obtained when each element of the consumed list is divided by the sum of the consumed list. You should use your solutions for (a) and (b) here. The normalized empty list is the empty list (all elements must be divided by zero, but there aren't any so this isn't an issue).
   Example:
   (*normalize-list* (*cons* 1 (*cons* 2 (*cons* 3 *empty*)))) ⇒ (*cons* 1/6 (*cons* 1/3 (*cons* 1/2 *empty*)))

   (d) Write a function *list-replace* which consumes a list of numbers, a target number, and a replacement number. The function produces a new list, where each element is equal to the corresponding element of the consumed list, except for all occurrences of the target number (if any), which are replaced with the replacement number. For example (*list-replace* (*cons* 1 (*cons* 2 (*cons* 2 *empty*))) 2 3) ⇒ (*cons* 1 (*cons* 3 (*cons* 3 *empty*)))

   (e) Write a function *count-repeats* which consumes a list of integers, and produces the number of adjacent duplicates. That is, the number of elements in the list that are equal to the next value in the list.
   Examples:
   (*count-repeats* (*cons* 1 (*cons* 1 (*cons* 2 *empty*)))) ⇒ 1

(*count-repeats* (*cons* 1 (*cons* 1 (*cons* 1 *empty*)))) ⇒ 2

Note: This question requires a non-standard list template, because it must look at both the first and second elements of the list. You do not need to write this template.

Submit your code for this question in a file named `numbers.rkt`

3. This problem deals with rockets, so please be careful!

The rockets used by NASA are designed with multiple stages. This allows empty tanks and heavy engines to be dumped when they're no longer useful.

For example, the Apollo moon-landing missions used three-stage Saturn V rockets. The first stage used five F-1 engines and a 2,100,000L fuel tank; the second, five smaller J-2 engines with 1,300,000L of fuel; the third, one J-2 and 340,000L. Further, the mission modules were situated atop the third stage with their own engines and fuel. The lander itself had both a descent stage for landing on the Moon, and an ascent stage for returning to lunar orbit.

We will define a rocket in Racket as a list of stages. i.e.:

;; A Rocket is a (listof Stage)

There are four important values for each stage, so we can represent a stage with a structure that has the following four fields:

- mass of fuel (in tonnes, 1000kg).
- "dry" mass of the stage (in tonnes), i.e., the mass of the tank not including the mass of fuel.
- thrust (in kilo-newtons, 1000N).
- specific impulse (abbreviated I_{sp}, in metres per second, m/s).

Some of these may be familiar, but specific impulse is not often seen outside of space exploration (real or fictional, e.g., Kerbal Space Program). Specific impulse measures the efficiency of the stage's engine(s). The background material at the end of this document contains interesting but **unnecessary** information; for a deeper extra-curricular understanding, see `https://en.wikipedia.org/wiki/Specific_impulse`. All values must be positive numbers.

The following sub-questions may make use of the "Silly Rocket" example: (**define** *silly-rocket* (*cons* (*make-stage* 90 30 1629.25 3333) (*cons* (*make-stage* 9 4 191.1 4500) *empty*))) You can find this sample rocket in the `example_rockets.rkt` file, along with a few other examples. You may copy these for examples and testing.

(a) You have been given that a Rocket is a (*listof Stage*). Complete this data definition by writing a structure definition for *stage* and a data definition for *Stage* with the exact four fields mentioned above, in order. You must use the field names *fuel-mass*, *dry-mass*, *thrust*, and *isp*.

(b) Write a function *rocket-mass* which consumes a Rocket and produces the total mass in tonnes (including fuel) of that rocket.
Example: The Silly Rocket has a mass of 133 tonnes ($90+30=120$ for the first stage, and $9+4=13$ for the second).

(c) Write a function *rocket-twr* which consumes a Rocket and produces a list of numbers. Each number in the list represents the minimum thrust-to-weight ratio (TWR) of the rocket when that stage is active. The TWR of a rocket is its thrust divided by its weight, and it is at a minimum when a stage is first ignited (the weight will decrease as fuel is consumed from the active stage).
The weight of an object is the force of gravity that acts upon it. You can get this value by multipling the object's mass by the acceleration due to gravity it experiences (as in previous assignments, use $g = 9.8N/kg$). For example, a person with a mass of $100kg$ has a weight of $100kg \cdot 9.8N/kg = 980N$, and a rocket with a mass of 100 tonnes has a weight of $980kN$.
Example: The Silly Rocket has a minimum TWR of 1.25 for the first stage ($1629.25kN/(133t \cdot 9.8N/kh) = 1.25$) and 1.5 for the second stage ($191.1kN/(13t \cdot 9.8N/kg) = 1.5$).

(d) The Tsiolkovsky rocket equation states that the total change in velocity v after burning a rocket stage is $\Delta v = I_{sp} \log \frac{m_0}{m_1}$, where I_{sp} is the specific impulse (as a speed), m_0 is the mass of the rocket when the stage starts burning, and m_1 is the mass of the rocket after it has finished (but before it has been detached). The logarithm is base e (as is Racket's built-in *log* function).
Example: Rounding to one decimal place (just for this example), the amount of Δv achievable by the Silly Rocket is computed by finding the amount for each of its stages.
The first stage has $\Delta v = 3333 \log \frac{133}{133-90} \approx 3763.5$.
The second stage has $\Delta v = 4500 \log \frac{13}{13-9} \approx 5303.9$.
Combined, this gives a total Δv of approximately 9067.4 m/s.
Write a function *rocket-delta-v* which consumes a Rocket and produces the total Δv for that rocket, in m/s.
Note: Do **not** round any values. The example does so because logarithms are inexact. Use a tolerance of 1 m/s when testing your function.

Submit your code for this question in a file named `rockets.rkt`

4. Card games have a long history of entertainment. Archaeological evidence suggests that these games go back centuries to 12th century China. Nintendo itself started out as a card game company in 1889. Ironically, card games still enjoy cult-like followings after appearing as mini-games in much larger video games ("Triple Triad", Final Fantasy 8).

Drawing inspiration from "Triple Triad", consider a game where each card represents a combatant. Two combatant cards battle each other, resulting in a win, lose or draw. The outcome of each battle depends on the four positive integer values, (strength, speed, intelligence, charm), written on each card.

(a) Write a structure definition for *card* and a data definition for *Card*. The structure must have exactly four fields: strength, speed, intelligence, and charm (with exactly those names, in that exact order).

(b) Write a function *card-to-list* which consumes a Card and produces a list of the strength, speed, intelligence and charm values.

Write a function *list-to-card* which consumes a list of four numbers and produces a Card.

(c) The four values on a regular card must add to 10. Write a predicate *card-regular?* which consumes a Card and produces true if and only if the card matches the given constraints. For example, (*make-card* `7 1 1 1`) is regular, but (*make-card* `7 1 1 2`) and (*make-card* `6 1 1 1`) are not because they don't sum to 10.

(d) Battles are decided by comparing each of the strength, speed, intelligence and charm values for the two cards. The card with the majority of greater values wins and the other card loses; otherwise, the battle is a draw. For example, (*make-card* `4 2 2 2`) beats (*make-card* `7 1 1 1`) because it has greater speed, intelligence and charm; on the other hand, (*make-card* `4 2 2 2`) draws with (*make-card* `5 1 2 2`) since each card has exactly one value greater than the other card.

Write a function *card-battle* that consumes two cards and produces 'win if the first card beats the second, 'lose if the second card beats the first, or 'draw otherwise.

(e) Write a function *card-select* that consumes your hand of cards (i.e., a list of cards) and an opponent's card. This function will produce the first card from your hand that beats the opponent's card; if none of the cards from your hand beat it, the function will produce false.

(f) A special deck of collector cards was released by the CS135 instructors. A collector card is a card whose strength, speed, intelligence and charm values sum to any positive number greater than 3 except 10.

Write a function *count-collector-cards* that consumes a hand of cards and produces the number of collector cards in the hand.

Submit your code for this question in a file named `cards.rkt`

This concludes the list of questions for which you need to submit solutions. Don't forget to check your email for the basic test results after making a submission.

---

5. **5% Bonus**: This question will continue using the Card data type defined in Question 4.

(a) Write a function *card-better* that consumes a regular card and a value which is either 'strength, 'speed, 'intelligence or 'charm. The function produces a regular card that beats the consumed card along the specified value, or it produces false if no such card exists. For example, (*make-card* `3 3 3 1`) and 'strength may produce (*make-card* `4 4 1`

1) because (*make-card* 4 4 1 1) beats (*make-card* 3 3 3 1), and it has a higher strength value (this is not the only card that meets these two requirements). Also, (*make-card* 5 1 2 2) and 'strength produces false, since there is no valid card that beats (*make-card* 5 1 2 2) with strength value 6 or greater.

Recall: the four values of a regular card must add to 10.

(b) Write a function *card-allbetter* that consumes a regular card and produces a list containing all regular cards that beat the consumed card.

Submit your code for this question in a file named `cards-bonus.rkt`

Enhancements: *Reminder—enhancements are for your interest and are not to be handed in.*

Racket supports unbounded integers; if you wish to compute 2^{10000}, just type (*expt* 2 10000) into the REPL and see what happens. The standard integer data type in most other computer languages can only hold integers up to a certain fixed size. This is based on the fact that, at the hardware level, modern computers manipulate information in 32-bit or 64-bit chunks. If you want to do extended-precision arithmetic in these languages, you have to use a special data type for that purpose, which often involves installing an external library.

You might think that this is of use only to a handful of mathematicians, but in fact computation with large numbers is at the heart of modern cryptography (as you will learn if you take Math 135). Writing such code is also a useful exercise, so let's pretend that Racket cannot handle integers bigger than 100 or so, and use lists of small integers to represent larger integers. This is, after all, basically what we do when we compute by hand: the integer 65,536 is simply a list of five digits (with a comma added just for human readability; we'll ignore that in our representation).

For reasons which will become clear when you start writing functions, we will represent a number by a list of its digits starting from the one's position, or the rightmost digit, and proceeding left. So 65,536 will be represented by the list containing 6, 3, 5, 5, 6, in that order. The empty list will represent 0, and we will enforce the rule that the last item of a list must not be 0 (because we don't generally put leading zeroes on our integers). (You might want to write out a data definition for an extended-precision integer, or EPI, at this point.)

With this representation, and the ability to write Racket functions which process lists, we can create functions that perform extended-precision arithmetic. For a warm-up, try the function *long-add-without-carry*, which consumes two EPIs and produces one EPI representing their sum, but without doing any carrying. The result of adding the lists representing 134 and 25 would be the list representing 159, but the result of the lists representing 134 and 97 would be the list 11, 12, 1, which is what you get when you add the lists 4, 3, 1 and 7, 9. That result is not very useful, which is why you should proceed to write *long-add*, which handles carries properly to get, in this example, the result 1, 3, 2 representing the integer 231. (You can use the warmup function or not, as you wish.)

Then write *long-mult*, which implements the multiplication algorithm that you learned in grade school. You can see that you can proceed as far as you wish. What about subtraction? You need to figure out a representation for negative numbers, and probably rewrite your earlier functions to deal with it. What about integer division, with separate quotient and remainder functions? What about rational numbers? You should probably stop before you start thinking about numbers like 3.141592653589...

Though the basic idea and motivation for this challenge goes back decades, we are indebted to Professor Philip Klein of Brown University for providing the structure here.

Rocketry Background.

Specific Impulse - Specific impulse is the efficiency of the engine–the amount of thrust produced per unit flow of fuel. If fuel is measured as a mass, then the units are newtons per kilogram per second ($N/(kg/s)$ or $N \cdot s/kg$) which simplifies to metres per second. This speed is the speed at which the engine emits exhaust (in an idealized model), so it is often called "(effective) exhaust velocity". Measuring the fuel by weight instead of mass will result in a specific impulse measured in seconds. This is the amount of time for the engine to consume an amount of fuel with weight equal to the thrust being produced. Translating between these two requires converting between mass and weight (multiplying or dividing by Earth's surface gravity $g = 9.8N/kg$).

TWR - Most designs call for a TWR of around 1.2 during ascent (going from the ground to space). Too high and you lose too much energy to drag. Too low and you waste fuel fighting gravity (less than 1 and you cannot even fight gravity and won't go up at all!) Once in space it is OK to have a much lower TWR. For example, the New Horizons probe was launched using an Atlas V rocket with an initial TWR somewhere around 1.2, but the probe itself has a TWR of about 0.004. This means minor course corrections, like the one it made on the way to Jupiter, can take 15 minutes to complete! Since its journey to Jupiter was a long one, this was not important.

Delta V - It takes somewhere in the range of 8,600 to 10,000 m/s worth of Δv to make it into orbit around the Earth (the exact number depends on what path you take through the atmosphere). Once in orbit it takes about 3300 m/s to get into orbit around the Moon, or about 4300 to get into orbit around Mars. So getting to Mars does not require that much additional fuel despite the major difference in distance. It does, however, require a lot more in terms of snacks. As Robert Heinlein put it, "Once you're in orbit, you're halfway to anywhere."