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

Assignment:	4
Due:	Tuesday, October 17, 2017 9:00pm
Language level:	Beginning Student
Allowed recursion:	(Pure) Structural recursion
Files to submit:	`lists.rkt`, `div-by-3.rkt`, `trains.rkt`, `bonus.rkt`
Warmup exercises:	HtDP 8.7.2, 9.1.1 (but use box-and-pointer diagrams), 9.1.2, 9.5.3, 10.1.4, 10.1.5, 11.2.1, 11.2.2, 11.4.3, 11.5.1, 11.5.2, 11.5.3
Practice exercises:	HtDP 8.7.3, 9.5.4, 9.5.6, 9.5.7, 10.1.6, 10.1.8, 10.2.4, 10.2.6, 10.2.9, 11.4.5, 11.4.7

- **Make sure you read the OFFICIAL A04 post on Piazza** for the answers to frequently asked questions.
- Of the built-in list functions, you may only use *cons*, *first*, *rest*, *empty?*, and *cons?*. You may not use list abbreviations.
- Policies from Assignment 3 carry forward.
- 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. Submissions that do not follow this restriction will be heavily penalized.
- In addition, you must include a data definition for all user-defined types. You do not need to include templates in your solutions unless specifically required by the question.
- You may use the **cond** special form. You are **not** allowed to use **if** in any of your solutions.

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.

2. (a) Write a function *sum-positive* which consumes a list of integers and produces the sum of the positive integers in that list. The empty list sums to 0. For example, (*sum-positive* (*cons* 5 (*cons* -3 (*cons* 4 *empty*)))) $\Rightarrow$ 9.

   (b) Write a predicate *contains?* which consumes a value, *elem*, and a (*listof Any*). It produces *true* if *elem* is in the list and *false* otherwise. For example, (*contains?* 'fun (*cons* 'racket (*cons* 'is (*cons* 'fun *empty*)))) $\Rightarrow$ *true*.

   Note that Racket contains the built-in functions *member?* and *member* which are identical to *contains?* except for the name. Of course, **you can't use them on this assignment**. In class we've told you to use *equal?* sparingly. This is an appropriate place to use it because you don't know the types of what you are comparing.

   (c) Write a predicate *has-duplicate?* which consumes a (*listof Any*) . It produces *true* if any element in the list appears more than once. For example, (*has-duplicate?* (*cons* 1 (*cons* 2 (*cons* 2 *empty*)))) $\Rightarrow$ *true*.

   (d) Write a function *keep-ints* which consumes a (*listof Any*). It produces a list that contains only the integers in the given list, in their original order. For example, (*keep-ints* (*cons* 'a (*cons* 1 (*cons* "b" (*cons* 2 *empty*))))) $\Rightarrow$ (*cons* 1 (*cons* 2 *empty*)). You may find the predicate *integer?* useful. For full marks you may not build and then reverse a list.

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

3. An alternative definition for natural numbers is that 0, 1, and 2 are all natural numbers. Any number that is three larger than a natural number is also a natural number.

   (a) Write a data definition for natural numbers using the above observation. Name this data definition *Nat3* to distinguish it from *Nat*. Examples of data definitions are on slides 05-5, 06-5, 06-13 and were discussed in association with 05-44.

   (b) Write a function template for *Nat3*. Call it *nat3-template*. You might consult the templates on slides 05-20 and 06-6.

   (c) 0 is divisible by three. So is $0+3$, $0+3+3$, $0+3+3+3$, and so on.

   Write a predicate, *div-by-3?* based on *nat3-template* which consumes a *Nat3* and produces *true* if it is divisible by 3 and *false* otherwise.

   You may not use any form of multiplication or division.

Submit your data definition, template, and code in a file named `div-by-3.rkt`.

4. Trains have locomotives, cars, and cabooses. There are three kinds of cars: box cars, tank cars, and passenger cars. We'll use the term *unit* to refer to a single locomotive, car or caboose and the term *train* to refer to zero or more units coupled together. Each unit has a unique serial number.

   We'll represent each unit with a structure, (*define-struct unit* (*type serial*)) where type is a *Unit-Type* that is one of 'L, 'B, 'T, 'P, or 'C [1] (for locomotives, box cars, tank cars, passenger cars, and cabooses, respectively) and *serial* is a natural number.

   Download (that's not the same as cutting and pasteing!) the file `trains.rkt` and save it your assignment 04 directory. This is where you'll write your code for this problem. Also download `a04lib.rkt` and place it in the same directory as `trains.rkt`. This file contains a *string→train* function to make testing easier. Remember to type → as ->. The *requires* at the top of `trains.rkt` is all that needs to be done to use it from that file.

   (a) In `trains.rkt` complete the data definitions for *Unit-Type*, *Unit*, and *Train*.

   (b) Read the code in `a04lib.rkt` for *string→train*. In a comment in `trains.rkt`, write a brief, 2-4 sentence description of **how** it works. That's different from the purpose statement, which describes what it does. *string→train* should save you lots of time and effort in testing your train-related code. Use it!

   (c) Write a predicate *headed-by?* that consumes a *Train* and a *Unit-Type*. It produces *true* if the first unit of the train has the given unit type and *false* otherwise.

   (d) Write a predicate *ends-with-caboose?* which consumes a *Train* and produces *true* if and only if there is exactly one caboose and it is the last unit in the train.

   (e) Write a function *remove-unit* which consumes a *Train*, *t*, and a serial number, *s*. It produces a *Train* that is identical to *t* except that the unit with the serial number *s* is removed[2]. Recall that serial numbers are unique. It could be that no unit in the train has the given serial number.

   (f) A *proper train* is a *Train* with zero or more locomotives followed by zero or more cars followed by zero or more cabooses (and nothing after the last caboose).

   Write a predicate *proper-train?* which consumes a *Train* and produces *true* if it is a proper train and *false* if it is not.

   Hints: (which you may ignore) There are at least three distinct approaches to this problem, some of which may be easier than others:

   - Given that the first unit is an *X*, write helper functions to determine whether the rest of the train is a proper train.
   - Given that the first unit is an *X*, is the following unit allowed in a proper train?

[1] These symbols are chosen to make *string→train* convenient. It sacrifices expressivity and clarity, unfortunately.
[2] *t* and *s* are used to make the sentence precise. They aren't good parameter names.

- Is the train composed of a block of locomotives followed by a block of cars followed by a block of cabooses followed by nothing?

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

---

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

   Recall *div-by-3?* from problem 3. A computationally more efficient method to determine whether a natural number is divisible by three is to sum the digits. If the sum of the digits mod 3 is 0 then the number itself is divisible by three. For example, $1+2+3+4+5 \mathit{ mod } 3 = 0$ and so we know that 12345 is divisible by 3.

   Write *div-by-3-alt?* which consumes a *Nat* and produces *true* if it is divisible by 3 and *false* otherwise.

   Restrictions: Your solution must use the observations above about summing digits. You may use *last-digit* and *other-digits* from `a04lib.rkt`. You may use boolean expressions, numerical comparison functions like $<$, **cond**, **define**, $+$, and $-$ (and nothing else).

   A data definition for decimal integers may be helpful:
   ;; A Digit is one of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
   ;; A DecInt is one of:
   ;; ∘ a Digit
   ;; ∘ (+ (* b 10) d) where b is a DecInt and d is a Digit

   Submit your code in the file `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.