Assignment: 4
Due: Tuesday, January 31, 2017 9:00pm
Language level: Beginning Student
Allowed recursion: (Pure) Structural recursion
Files to submit: faculty.rkt, lists.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

- Coverage: until M5-39
- 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.
- Test data for all questions will always meet the stated assumptions for consumed 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 the 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 can be tested for correctness.
- Do not send any code files by email to your instructors or tutors. 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.
- Of the built-in list functions, you may only use cons, first, rest, empty?, cons?, length, and member?. You may not use list abbreviations.
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


   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. In this question we will use the following data definitions:

```
(define-struct lecturer (name salary faculty))
;; A Lecturer is a (make-lecturer Sym Num Sym)
;; requires: salary is a non-negative number
;;
(define-struct professor (name salary faculty research-area))
;; A Professor is a (make-professor Sym Num Sym Sym Sym)
;; requires: salary is a non-negative number
;;
;; A FacultyMember is one of:
;; * a Lecturer
;; * a Professor
```

(a) Write the template function for FacultyMember.

(b) Write a function same-faculty? which consumes two FacultyMember structures and produces true if both are working in the same faculty, and false otherwise.

(c) Write a function update-salary which consumes a FacultyMember structure, and produces an updated FacultyMember structure as follows:
   If the FacultyMember is a Lecturer then his/her salary increases by 5%, otherwise his/her salary increases by 7%.

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

3. (a) Write a function all-positives? which consumes a list of integers, and produces true if all integers in the consumed list are positives, and false otherwise. The function produces true for empty list.
(b) Write a function `max-difference` which consumes a list of integers and produces the difference (as positive value) between the highest and the lowest values in the consumed list. For empty list the function should produce 0. For example, `(max-difference (cons 0 (cons -3 (cons 10 empty)))) ⇒ 13`

(c) Write a function `ascending?` which consumes a list of integers, and produces `true` if all integers in the consumed list are in increasing order, and `false` otherwise. A list of length 0 or 1 is ascending. For example, `(ascending? (cons 4 (cons 7 (cons 10 empty)))) ⇒ true` while `(ascending? (cons 4 (cons 7 (cons 10 (cons 10 empty))))) ⇒ false`

(d) Write a function `multiple-of?` which consumes a list of integers `nlst`, and a non-zero integer `n`, and produces `true` if all values in `nlst` are multiplication of `n`, and `false` otherwise. The function produces `true` for empty list. For example, `(multiple-of? (cons 15 (cons 10 (cons 30 (cons -35 empty)))) 5) ⇒ true`, while `(multiple-of? (cons 15 (cons 17 (cons 30 (cons 0 empty)))) 2) ⇒ false`

Submit your code for this question in a file named `lists.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 \textit{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 \textit{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.