CS 135 Fall 2016
Graham, Holtby, Petrick, Sakhnini

Assignment:	9
Due:	Tuesday, November 22th, 9:00 pm
Language level:	Intermediate Student with Lambda
Allowed recursion:	None (see notes below)
Files to submit:	`listsa09.rkt`, `eightsa09.rkt`, `sequences.rkt`, `numal.rkt`, `bonus.rkt`
Warmup exercises:	HtDP *Without using explicit recursion:* 9.5.2, 9.5.4
Practise exercises:	HtDP 20.2.4, 24.3.1, 24.3.2

- You may use most of the abstract list functions in Figure 57 of the textbook (`http://www.htdp.org/2003-09-26/Book/curriculum-Z-H-27.html#node_sec_21.2`) unless indicated otherwise in the question, but you may **not** use *foldl* and *assf*.
- Note that some versions of the textbook have different signatures for the abstract list functions. If in doubt, refer to the function signatures from the given link.
- You may use primitive functions, such as mathematical functions, *cons*, *first*, *second*, *third*, *rest*, *list*, *empty?* and *equal?*.
- You may provide constant definitions where appropriate.
- You may **not** use *append* or *reverse* unless indicated otherwise in the question. Other non-primitive list functions, including *length*, *member?* and *sort*, *are* permitted.
- **You may not write explicitly recursive functions; that is, functions that involve an application of themselves, either directly or via mutual recursion, unless specifically permitted in the question.**
- You may **not** use any other *named* user-defined helper functions, local or otherwise, and you should use **lambda** instead. (Although (**define** *my-fn* (**lambda** (*x*) ...)) is *not* permitted.) **However, local constants are permitted.** If you violate this restriction you risk receiving zero on the assignment question.
- You **may** use functions defined in earlier parts of a question to help write functions later in the same question.
- Your solutions must be entirely your own work.

Here are the assignment questions that you need to submit.

1. The stepping problems for Assignment 9 at:

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

2. Complete Question 2 of Assignment 4/5 following the restrictions described on the first page of this assignment. Your solution may only use the abstract list functions *map*, *filter*, and *foldr*. Since the purposes, contracts, tests, and examples for these functions would be the same as those used in the original assignment, you are not required to include them in your submission. However, you are encouraged to do thorough testing of your own solutions to ensure they are correct.

   Place your solution in `listsa09.rkt`.

3. Complete Question 4 (b), (c), (d), and (e) of Assignment 4/5 following the restrictions described on the first page of this assignment. Since the purposes, contracts, tests, and examples for these functions would be the same as those used in the original assignment, you are not required to include them in your submission. However, you are encouraged to do thorough testing of your own solutions to ensure they are correct.

   Place your solution in `eightsa09.rkt`.

4. Recall the description of arithmetic and geometric sequences from midterm 1. The terms of an arithmetic sequence have a common difference; the terms of a geometric sequence have a common ratio.

   (a) Either an arithmetic sequence or a geometric sequence can be described by the first two terms of the sequence. Write a function called *sequence* that consumes two numbers (the first two numbers of the potential sequence), one natural number (the number of elements in the sequence), and a symbol (either ‘arithmetic or ’geometric) and produces a list containing the numbers in the sequence. For example:

   - (*sequence* 5 4.9 10 ‘arithmetic) produces (*list* 5 4.9 4.8 4.7 4.6 4.5 4.4 4.3 4.2 4.1)
   - (*sequence* 1 5 4 ‘geometric) produces (*list* 1 5 25 125)

   A sequence with 0 elements would produce the *empty* list. You may assume that the values given will always produce a valid sequence.

   (b) Write a function *arithmetic?* that consumes a list of numbers, and produces *true* if the list is an arithmetic sequence, and *false* otherwise. You solution may not use *build-list*, nor may it use any functions that contain *build-list*.

   Place your solution in `sequences.rkt`.

5. Implement the following helpful list functions. Do not be alarmed if some of your definitions are very short. Remember, that functions written for earlier parts of the question may be used in later parts of the question.

   Use the following data definitions:
   ;; A Number Association List (NAL) is one of:
   ;; * empty
   ;; * (cons (list Num Any) NAL)

   ;; A Pair is a (list Any Any).

   Helpful hint: (*map f* (*list x1 x2* ... *xn*) (*list y1 y2* ... *yn*)) has the same effect as evaluating (*list* (*f x1 y1*) (*f x2 y2*) ... (*f xn yn*)).

   (a) *zip* consumes two lists of equal length, and produces a list of Pair(s) where the *i*th Pair contains the *i*th element of the first list followed by the *i*th element of the second list.

   (b) *unzip* consumes a list of Pair(s), and produces a list of two lists. The first list contains the first element from each Pair, and the second list contains the second element from each Pair, in the original order.

   (c) *de-dup* ("de-duplicate") consumes a list of any values and produces a new list with only one occurrence of each element of the original list, in any order.

   (d) *occurrences* consumes a list of any values and a value, in that order, and produces the number of times that the given value occurs in the list.

   (e) *occurrences-al* consumes a list of numbers and produces a NAL where the keys are all the unique elements of the list and the values are the corresponding number of occurrences. For example,
   (*check-expect* (*occurrences-al* '(1 1 2 3)) '((1 2)(2 1)(3 1)))
   The NAL may be in any order.

   (f) *find-max-pair* produces the pair from a NAL that has the largest key. If the maximum key is not unique, produces any pair with a maximal key. If the given NAL is empty, produces *empty*.

   (g) A *distance function* (or "metric" `http://en.wikipedia.org/wiki/Metric_(mathematics)`) takes two arguments of the same type and produces a nonnegative number that tells how far apart they are. Distances may be defined on many different types of values; write a function *euclidean-dist* that consumes two lists of numbers of equal length, and computes their Euclidean distance `http://en.wikipedia.org/wiki/Euclidean_distance`, treating the lists as if they are points in *n*-dimensional space. (Here, *n* is the length of either list.) The Euclidean distance between two empty lists is 0.

   Place your solution in `numal.rkt`.

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

---

6. **5% Bonus (each part worth 1%)**:

   In this question, you will write some convenient functions that operate on functions, and demonstrate their convenience. Note that to receive full marks, you must write the correct **contract and function definition** for these functions. Though it will likely be helpful to complete the full design recipe for each of the functions. In addition to the other restrictions in this assignment, you may not use the built-in *compose* function. Place your solution in the file `bonus.rkt`.

   (a) Write the function *my-compose* that consumes two functions f and g in that order, and produces a function that when applied to an argument x gives the same result as if g is applied to x and then f is applied to the result (i.e., it produces (f (g x))).

   (b) Write the function *curry* that consumes one two-argument function f, and produces a one-argument function that when applied to an argument x produces another function that, if applied to an argument y, gives the same result as if f had been applied to the two arguments x and y.

   (c) Write the function *uncurry* that is the opposite of *curry*, in the sense that for any two-argument function f, (*uncurry* (*curry* f)) is functionally equivalent to f.

   (d) Using the new functions you have written, together with *filter* and other built-in Racket functions, but no other abstract list functions, give a nonrecursive definition of *eat-apples* from Module 09. You may not use any helper functions or **lambda**.

   (e) Using the new functions you have written, together with *foldr* and other built-in Racket functions, but no other abstract list functions, give a nonrecursive definition of *my-map*, which is functionally equivalent to *map* (as described in the textbook here: (`http://www.htdp.org/2003-09-26/Book/curriculum-Z-H-27.html#node_sec_21.2`)). You may not use *map* to solve this problem, perhaps not surprisingly. You may not use any helper functions or **lambda**.

   The name *curry* has nothing to do with spicy food in this case, but it is instead attributed to Haskell Curry, a logician recognized for his contribution in functional programming. The technique is called "currying" in the literature, and the functional programming language Haskell, which provides very simple syntax for currying, was also named after him. The idea of currying is actually most correctly attributed to Moses Schönfinkel. "Schönfinkeling" however does not have quite the same ring.

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

Consider the function (*euclid-gcd*) from slide 7-16. Let f_n be the nth Fibonacci number. Show that if $u = f_{n+1}$ and $v = f_n$, then (*euclid-gcd* u v) has depth of recursion n. Conversely, show that if (*euclid-gcd u v*) has depth of recursion n, and $u > v$, then $u \geq f_{n+1}$ and $v \geq f_n$. This shows that in the worst case the Euclidean GCD algorithm has depth of recursion proportional to the logarithm of its smaller input, since f_n is approximately ϕ^n, where ϕ is about 1.618.

You can now write functions which implement the RSA encryption method (since Racket supports unbounded integers). In Math 135 you will see fast modular exponentiation (computing $m^e \bmod t$). For primality testing, you can implement the little Fermat test, which rejects numbers for which $a^{n-1} \not\equiv 1 \pmod{n}$, but it lets through some composites. If you want to be sure, you can implement the Solovay–Strassen test. If $n-1 = 2^d m$, where m is odd, then we can compute $a^m \pmod{n}$, $a^{2m} \pmod{n}, \ldots, a^{n-1} \pmod{n}$. If this sequence does not contain 1, or if the number which precedes the first 1 in this sequence is not -1, then n is not prime. If n is not prime, this test is guaranteed to work for at least half the numbers $a \in \{1, \ldots, n-1\}$.

Of course, both these tests are probabilistic; you need to choose random a. If you want to run them for a large modulus n, you will have to generate large random integers, and the built-in function *random* only takes arguments up to 4294967087. So there is a bit more work to be done here.

For a real challenge, use Google to find out about the recent deterministic polynomial-time algorithm for primality testing, and implement that.

Continuing with the math theme, you can implement the extended Euclidean algorithm: that is, compute integers a, b such that $am + bn = \gcd(m, n)$, and the algorithm implicit in the proof of the Chinese Remainder Theorem: that is, given a list $(a_1, \ldots, a_n)$ of residues and a list $(m_1, \ldots, m_n)$ of relatively coprime moduli ($\gcd(m_i, m_j) = 1$ for $1 \leq i < j \leq n$), find the unique natural number $x < m_1 \cdots m_n$) (if it exists) such that $x \equiv a_i \pmod{m_i}$ for $i = 1, \ldots, n$.