Akinyemi, Heinle, Holtby, Istead, Roegiest, Tompkins

Assignment:	9
Due:	Tuesday, November 24th, 9:00 pm
Language level:	Intermediate Student with Lambda
Allowed recursion:	None (see notes below)
Files to submit:	`a09.rkt`, `alfun.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 call 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.**
- You **may** use functions defined in earlier parts of the assignment to help write functions later in the assignment. (E.g., you may use a function written for part 2 (a) as part of your solution for 3 (b).)
- Your solutions must be entirely your own work.
- Unless otherwise stated, the lists produced for Questions 2 and 4 may be in any order.

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. Racket has many list functions but other programming languages implement various other useful list functions, which can conveniently be implemented in terms of Racket's built-ins. In this question, you will implement several such functions as well as some other functions that may prove useful in the remainder of this assignment. Some of your solutions will be very short, this is okay. Remember, **you may not use explicit recursion** unless otherwise specified, and remember that functions written for earlier parts of the question may be used in later parts of the question. Place your answers in the file `a09.rkt`.

   (a) *intersection* consumes two lists, and produces a list of all values shared by both lists, with no duplicates. For example,
   (*check-expect* (*intersection* '(3 4 2 1) '(1 5 7 3)) '(3 1))

   (b) *union* consumes two lists, and produces a list containing all the elements of both lists, with no duplicates. For example, (*check-expect* (*union* '(3 4 2 1) '(1 5 7 3)) '(3 4 2 1 5 7)).

   (c) *unique-fn* consumes a list and a predicate equality function and produces the same list such that all duplicates, according to the provided equality predicate, are removed. No sorting may be used. For example,
   (*check-expect* (*unique-fn* '(3 1 3) =) '(3 1))

   (*check-expect* (*unique-fn* '(1 1.05 2 1.2) (**lambda** (*x y*) (> 0.1 (*abs* (− *x y*))))) '(1 2 1.2))

   (d) *cross* consumes two lists, and produces a list of all possible pairs of elements from the two lists. For example: (*check-expect* (*cross* '(1 2 3 4) '(3 2)) '((1 3) (1 2) (2 3) (2 2) (3 3) (3 2) (4 3) (4 2)))

   (e) A similarity measure [1] is a function that consumes two arguments of the same type and produces a number that represents how similar they are (smaller typically implies dissimilarity). Similarity can be defined on different types of objects; however, they are most commonly applied to lists of numbers. Write a function *jaccard*, which computes the Jaccard Index[2] of two non-empty lists of numbers (of equal length). Formally, the Jaccard Index is defined as:

$$jaccard(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

   Note that you should not need to use check-within to test this function. You may also assume that either A or B will be non-empty.

   (f) *take* consumes a list and a natural number n, and produces a list containing only the first n elements of the list, or the entire list if it contains fewer than n elements. **You may use explicit recursion for this question.** Note however that it can be done without explicit recursion.

[1] https://en.wikipedia.org/wiki/Similarity_measure
[2] https://en.wikipedia.org/wiki/Jaccard_index

3. *Duplicate Detection* is the problem of determining whether two objects are the same based upon the *features* of those objects. For example, two documents (e.g., tweets, web pages, stories, news reports) are the the same if they have the same title[3], author, and words used in the body of the document. These various parts of a document are *features* that can be used to tell if two documents are the same[4], i.e., duplicates.

   The features describing an individual document are typically collected into a *feature vector*; but for the sake of simplicity, we will judge document similarity based upon the words in the document [5]. We will also assume these words have been mapped to the natural numbers using some magical algorithm (read: you only have to worry about lists of natural numbers; do not worry about how one might actually produce this mapping).

   For this question, you will write code to perform two types of duplicate document detection; exact duplicate detection and near-duplicate detection. We will do this by comparing the feature vectors of documents using a similarity measure. Exact duplicate detection is the process of finding documents that are identical, at least according to our similarity measure (e.g., whose Jaccard Index is 1). Near-duplicate detection is the process of finding documents whose similarity, again according to some measure, is within a particular threshold (or tolerance).

   Use the following data definition for a Feature Vector Association List (FV-AL):

   ```
   ;; A Feature Vector (FV) is a (listof Num)

   ;; A Document Identifier (DI) is a String

   ;; A Document Vector (DV) is a (list DI FV)

   ;; A Feature-Vector Association List (FV-AL) is
   ;; * empty
   ;; * (cons DV FV-AL)
   ;; require: each FV must be of the same length
   ;; each DV must be unique

   ;; A Document Pair Tuple (DPT) is a (list DI DI Num)
   ;; where Num corresponds to some similarity between
   ;; the feature vectors associated with each DI
   ```

   Place your answers for Q3 in the file `a09.rkt` as well, *after* your answers for Q2.

   You may use any of the functions you defined in Question 2 to help you solve this question (e.g., the Jaccard Index function from Question 2(e)). Again, you may not use recursion. In

[3]Yes, strictly speaking tweets don't have a title
[4]Of course, we might be more specific and require that all words fall in the same order but that's more complicated.
[5]Formally, this is called a bag-of-words model.

any case where there would be a "tie" (e.g., equal similarity, etc.) the ties may be broken arbitrarily.

(a) *cmp-with-sim* consumes a DV, a FV-AL, and a similarity measure (as defined in Question 2) in order, and produces a list of DPTs, one for each DV in the FV-AL. For each DV in the FV-AL, the corresponding DPT should be created as follows: the first DI comes from the provided DV, the second DI comes from the DV being processed, and the similarity comes from applying the provided similarity measure to the FVs of the provided DV and the DV being processed, respectively. For example,

```
(check-expect (cmp-with-sim (list "t1" '(1 2 3 4)) '(("t1" (1 2 3 4)) ("t2" (2 5 1 6))) jaccard)
              '(("t1" "t1" 1) ("t1" "t2" 1/3)))
```

(b) *find-all-exact* consumes a FV-AL and a similarity measure, and produces the list of DPTs that correspond to all ordered pairs of DIs in FV-AL that are *exact duplicates* of each other (and report their similarity as a Num), excluding DPTs with identical DIs (e.g., '("t1" "t1" 1) should not be included in the produced list). For this function, the order of the DPTs does not matter.

```
(check-expect (find-all-exact '(("t1" (1 2 3 4)) ("t3" (4 3 2 1))) jaccard)
              '(("t1" "t3" 1) ("t3" "t1" 1)))
```

(c) Write a simple predicate function, *redundant?*, that consumes two DPTs and determines if the tuples are redundant (i.e., represent the same set of DIs). For example, this predicate should produce true for the following: '("t1" "t2" 1) and '("t2" "t1" 1).

(d) *find-similar-within-d* consumes an FV-AL, a threshold $d \in [0,1]$, and a similarity measure and produces the DPTs of all combinations of pairs in the provided FV-AL whose similarity is above the threshold d. This function should not return DPTs with identical DIs. In addition, it should return only tuples with non-redundant DIs. For this function, the order of DIs in the DPT does not matter, and the order of the DPTs does not matter.

```
(check-expect (find-similar-within-d '(("t1" (1 2 3 4)) ("t2" (5 3 2 1)) ("t3" (4 3 2 1)))
                                     .5 jaccard)
              '(("t1" "t2" 0.6) ("t1" "t3" 1) ("t2" "t3" 0.6)))
```

(e) *find-k-similar-within-d* consumes an FV-AL, a threshold $d \in [0,1]$, a positive integer k, and a similarity measure. The function produces a list of DPTs for the k most similar pairs of FVs in the FV-AL whose similarity is above the threshold d. You may use sorting for this question. Again ensure that no identical or redundant DPTs are returned. For this function, the order of DIs in the DPT does not matter, and the order of the DPTs does not matter.

```
(check-expect (find-k-similar-within-d '(("t1" (1 2 3 4)) ("t2" (5 3 2 1)) ("t3" (4 3 2 1)))
                                       .5 2 jaccard)
              '(("t1" "t3" 1) ("t1" "t2" 0.6)))
```

4. Question 2 required an implementation of *unique-fn*, this function could be used to remove all the duplicate entries in a list. However, it is also the case that we may want to produce that list with only those numbers that didn't have duplicates or those numbers that had duplicates. In this question, you will implement two functions, *singletons* and *duplicates*.

   singletons will consume a list of numbers and produce a list containing only those numbers that had no duplicates in the original list. *duplicates* will consume a list of numbers and produce a list of numbers containing only those numbers that had duplicates in the original list. Note that the list returned by *duplicates* should contain no duplicates itself.

   You **may not** use sorting or explicit recursion in your solution, though you may define local helper functions. You should not use any functions you implemented in Question 2 (nor should you reimplement them).

   Place your solution in `alfun.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.

---

5. **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 only the contract and function defintion 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 $i = 1, \ldots, n$.