Assignment: 8
Due: Monday, November 26th, 9:00 pm
Language level: Intermediate Student with Lambda
Allowed recursion: None (see notes below)
Files to submit: alf.rkt, partition.rkt, ranking.rkt, match.rkt, bonus1.rkt,
bonus2.rkt
Warmup exercises: HtDP Without using explicit recursion: 9.5.2, 9.5.4
Practice exercises: HtDP 20.2.4, 24.3.1, 24.3.2

• Make sure you read the OFFICIAL A08 post on Piazza for clarifications and answers to frequently asked questions.
• Functions you may use unless indicated otherwise in the question or the next item:
  – The higher-order functions (also called “abstract list functions”) listed in the Help Desk. For example, build-list, filter, foldr, map, and quicksort. Note that the built-in versions you have access to are sometimes more flexible than the versions discussed in class. That flexibility is helpful on some of the questions. For example, (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)).
  – Built-in math functions, string functions (e.g. string->list, string-append, string-length, substring, comparisons), list functions (but note the specific exclusions, below).
  – Functions defined in earlier parts of the assignment. (E.g., you may use a function written for Q4 as part of your solution for Q5.)
• Functions you may not use unless indicated otherwise in the question:
  – Functions assoc, assq, for-each, append, reverse, list-ref, and list functions with names beginning with “ca”, “cd”, or “mem”.
  – 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.
• Your examples and tests must be different from the ones provided in the assignment.
• String comparisons, when required, should be case sensitive.
• Do not be alarmed if some of your definitions are very short. Remember, you may not use explicit recursion unless otherwise specified and functions written for earlier parts of the question may be used in later parts of the assignment.
• There are starter files available on the assignments page.
Here are some development tips:

- We have provided a file with helpers, `a08helpers.rkt`. Import this into `match.rkt` with (`require "a08helpers.rkt"`) at the top of the file. Both files must be in the same directory.

- Look at the `require` and `provide` expressions at the top of `a08helpers.rkt` to learn how you can access code you have written. In particular, code you write in `partition.rkt` and `ranking.rkt` will be useful in other parts of the assignment. Be sure MarkUs always has the most recent version of each file!

- You can add (`require racket/trace`) to the top of your program. Add `(trace X)`, where X is one or more functions, after those functions have been defined. When those functions are applied, the arguments they consume and the values they produce will be printed in the interactions window.

Here are the assignment questions that you need to submit. The weight of each question is given.

1. (10%) The stepping problems for Assignment 8 at
   [https://www.student.cs.uwaterloo.ca/~cs135/stepping/](https://www.student.cs.uwaterloo.ca/~cs135/stepping/)

2. (30%) Implement the following functions. You may not use `local`. You may not use helper functions other than with `lambda`. Place your solutions in `alf.rkt`.
   
   (a) `occurrences` consumes a list of numbers and a number, in that order, and produces the number of times that the given number occurs in the list of numbers.

   (b) `absolutely-odd` consumes a list of integers and produces the sum of the absolute values of the odd integers in the list. See A04 Q3a.

   (c) `zip` consumes two lists of equal length, and produces a list of pairs (two element lists) where the ith pair contains the ith element of the first list followed by the ith element of the second list.

   (d) `unzip` consumes a list of pairs, 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. Unzipping an empty list produces `'(()) ()`.

   (e) `dedup` (“de-duplicate”) consumes a list of numbers and produces a new list with only the first occurrence of each element of the original list. Note that this is different from `debounce` in the midterm.

   (f) `(subsequence lst from to)` consumes a list and two natural numbers. It produces the subsequence from `lst` that begins at index `from` and ends just before index `to`. Indexing starts at 0.
3. (10%) **partition** consumes a predicate and a list. It produces a two element list, \((\text{list } X \ Y)\), where \(X\) is a list of those items in the consumed list that satisfy the predicate and \(Y\) is a list of those items that don’t satisfy the predicate. The order of items in each list must be the same as the original list.

\[
\text{(check-expect (partition odd? '(1 2 3 4 5)) '((1 3 5) (2 4)))}
\]
\[
\text{(check-expect (partition boolean? empty) '(() ()))}
\]

Your solution **must** use explicit accumulative recursion and may **not** use abstract list functions. You may use **reverse**. Use **local** to encapsulate all helper functions. Place your solution in **partition.rkt**.

4. (30%) Waterloo’s Coop students are matched with employers with an algorithm (see Q5). It uses the following data definitions:

- An employer ID, **EmpId**, is a Str (e.g. “Google”)
- A student ID, **StdId**, is a Str (e.g. “Anna”)
- An Id is (anyof EmpId StdId)
- A preference, **Pref**, is a Nat (e.g. 2)
  - requires: a preference is \(\geq 1\)
- An **EmpRanking** is (list EmpId (listof (list StdId Pref)))
  - requires: The list of preferences is in non-decreasing order
  - and non-empty.
- A **StdRanking** is (list StdId (listof (list EmpId Pref)))
  - requires: The list of preferences is in non-decreasing order
  - and non-empty.
- A **Ranking** is (anyof EmpRanking StdRanking)

Some sample data:

\[
\text{(define employers}
  '(("Manulife" ("Anna" 1) ("Feisal" 2) ("Zihan" 2) ("Hannah" 2))
    ("Google" ("Anna" 1) ("Feisal" 2) ("Rafelia" 2))
    ("Apple" ("Rafelia" 1) ("Zihan" 2) ("Feisal" 3)))
\]
\[
\text{(define students}
\)
We’ll sometimes refer to the “ranking ID” and the “ranked ID”. In an \texttt{EmpRanking}, the ranking IDs are the employers and the ranked IDs are the students. In a \texttt{StdRanking} the situation is reversed.

Sometimes this data definition is hard to work with and the following will be preferable:

\texttt{;; A FlatRanking is a (list Str Str Num).}

\texttt{(define flat-employers '(("Manulife" "Anna" 1) ("Manulife" "Feisal" 2) ("Manulife" "Zihan" 2) ("Manulife" "Hannah" 2) ("Google" "Anna" 1) ("Google" "Feisal" 2) ("Google" "Rafelia" 2) ("Apple" "Rafelia" 1) ("Apple" "Zihan" 2) ("Apple" "Feisal" 3)))}

\texttt{flat-employers} is equivalent to \texttt{employers}. The first string is the ranking ID; the second string is the ranked ID.

In the sample data we’ve simply used first names to identify students. In real life we would use unique identifiers such as student numbers or userids. Assume \texttt{StdIds} and \texttt{EmpIds} are unique.

Implement the following list functions in \texttt{ranking.rkt}. Use of \texttt{local} is prohibited except in \texttt{unfold}. Use of external helper functions (including from previous questions) is also prohibited.

(a) \texttt{expunge} consumes a (\texttt{listof FlatRanking}), a ranking ID and and a ranked ID. It produces the same list of flat rankings but without elements containing the provided ranking ID or the provided ranked ID.

\texttt{(check-expect (expunge flat-employers "Manulife" "Rafelia") '(('"Google" "Anna" 1) ("Google" "Feisal" 2) ("Apple" "Zihan" 2) ("Apple" "Feisal" 3)))}

(b) \texttt{ranking-sort} consumes a (\texttt{listof FlatRanking}) and produces the same data but reordered so that the ranking IDs are in increasing alphabetical order. Where two ranking IDs are the same, they are ordered by the ranked IDs. The effect should be the same as appending the ranking ID and ranked ID of each item and then sorting, but you are not permitted to append the IDs.

\texttt{(check-expect (ranking-sort '(('"A" "c" 123) ("B" "a" 456) ("A" "a" 789))) '(('"A" "a" 789) ("A" "c" 123) ("B" "a" 456)))}
(c) **unfold** consumes a (listof Ranking) and produces an equivalent (listof FlatRanking). Use of **append** is permitted.

(check-expect (unfold employers) flat-employers)

(d) **find** consumes a key and a list of lists. It produces the first element within the list that has a first element equal to the key; false if there is no such element.

(check-expect (find "A" `(("B" 1 2 3) ("A" a) ("C"))) `("A" a))
(check-expect (find "D" `(("B" 1 2 3) ("A" X) ("C"))) false)
(check-expect (find 3 `((1 'a 'b 'c) (2 Y) (3 "X"))) `(3 "X"))

This could be done with **filter** but you **must** use **foldr**, just for fun.

(e) **find-pref** consumes a (listof Ranking), a ranking ID, and a ranked ID. It produces the preference of the ranking ID for the ranked ID. The preference must exist within the list of rankings.

(check-expect (find-pref students "Rafelia" "Google") 2)
(check-expect (find-pref employers "Apple" "Zihan") 2)

5. (20%) Waterloo’s Cooperative Education (“Coop”) has potential employers interview students for jobs. After the interviews, employers rank the students they interviewed with positive integers. A 1 signifies the most preferred student, larger numbers signify less preferred students. Similarly, students rank the employers that ranked them (employers may decline to rank a student they don’t want to hire). Again, 1 is used to signify the most preferred employers and larger numbers are used to signify less preferred employers.

We’re going to simplify things just a bit and assume that each employer has only one job to fill. They are required to give exactly one student a rank of 1. They may give multiple students a rank of 2 or larger. Students, on the other hand, can rank as many employers as they want with any positive integer (including 1).

At this point a computer algorithm takes over. It constructs a list of potential matches. Each potential match is a list with three elements: the employer, the student, and $e+s+r$ where $e$ is the employer’s ranking of the student, $s$ is the student’s ranking of the employer, and $r$ is a random number between 0 and 1 used to break ties.

While the list of potential matches is not empty, the algorithm:

- Selects the potential match with the smallest sum, declaring that employer and student “matched”.
- Removes that employer and student from the list of potential matches.

**Write the function** (match employers students) where employers is a (listof EmpRanking) and students is a (listof StdRanking). Every student ranked in employers is present in...
students and ranks that employer. match produces a (listof (list EmpId StdId)) as described in the algorithm, above.

(check-expect (match employers students)
  '(('Manulife" "Zihan") ('Apple" "Rafelia") ('Google" "Anna")))

Reminder: You must come up with different data and different tests.

The order of the list produced by match does not matter.

You may be wondering why so many employers are ranked 1 in the sample data. The reason is that the ranking algorithm shows a very definite preference to students who rank with small numbers. As a student, the larger the number you assign to an employer, the more jobs that will be matched before you in the algorithm – and the less likely you are to be matched. This is implicitly acknowledged on the Coop website. For what it’s worth, employers face the same dilemma. See the first bonus problem for another approach.

Hints and requirements for implementing match:

• If you’ve completed questions 3 and 4, you’ve already done a lot of the work required for match.

• If you haven’t already downloaded a08helpers.rkt from the Assignments page, now is the time to do so.

• The EmpRanking and StdRanking data definitions are hard to work with. Consider transforming them to other data definitions that have all the same information (obviously!) but are easier to work with.

• At some point you’ll want a list of lists where each sublist has an EmpId, a StdId, and the sum of their preferences (including the random value). Notice that a FlatRanking has a Num rather than a Nat.

• For testing purposes, we don’t actually want a random number; we want something that is reproducible. Use the function random-epsilon contained in the starter file a08helpers.rkt.

• match may use helper functions that are not encapsulated with local.

• match requires one helper function that is recursive. All other functions are either not recursive at all or use abstract list functions to implement the recursion. This recursive helper function consumes a list. The list does not proceed towards empty one step at a time; it always gets smaller, but usually has several elements removed upon each recursive application. That is, it is not pure structural recursion.

This recursive helper function is small (about 5 lines of code in the sample solution) and uses its own helper function (also quite small; recursion is via abstract list functions). You should strive for a similar solution. That is, you are permitted one relatively small function that calls itself (either directly or indirectly). There is a reasonable amount of work to do to set up the first application of this function.
• The sample solution (excluding reused code from previous questions, comments and testing) is only about 25 lines of code. If your approach uses several multiples of this, it is probably needlessly complex. Perhaps you should think more carefully about it.

• Place your solutions in `match.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 #1:** There is an alternative to Coop’s current algorithm called “the stable marriage algorithm”, invented in 1962 by David Gale and Lloyd Shapley. It formed the basis for the 2012 Nobel Prize in Economics awarded to Shapley and Alvin E. Roth for their work in applying it to economic markets.

In our context, the algorithm matches employers and students such that there is no pair of matches \((E_i, S_i)\) and \((E_j, S_j)\) where \(E_i\) would prefer to be matched with \(S_j\) and \(S_j\) would prefer to be matched with \(E_i\). If these were marriages, there would be no affairs!

It works like this:

• Employers rank the \(n\) students they are willing to hire in order from 1 (most preferred) to \(n\) (least preferred).

• Students rank the \(m\) employers that ranked them (that is, students don’t rank employers that aren’t willing to hire them) from 1 (most preferred) to \(m\) (least preferred). For both employers and students these can be honest preferences; there is no penalty for ranking someone higher or lower than your true preference.

• While the list of unmatched jobs\(^1\) is not empty:
  – Take the first job off the unmatched jobs list and offer it to the student most preferred by the employer
  – The student can respond in one of three ways:
    * **accept:** The student hasn’t been matched yet and thus is willing to accept any job that comes along. The student and job are matched; the job is removed from the list of unmatched jobs and added to the list of matched jobs.
    * **replace:** The student has already been matched with a job but prefers the job just offered (i.e. it has a lower preference; lower numbers are better). The old match is taken off the matched jobs list and put back on the unmatched jobs list. The student that just jilted that employer is removed from that job’s list of preferences because it’s known they won’t accept the job. The student with their newly accepted job is added to the matched jobs list.

---

\(^1\)Remember that each employer has exactly one job to fill; we’ll use employer and job interchangeably.
* **decline:** The student has already been matched and prefers their current job to the one just offered. The student is removed from the offered job’s list of preferences and the job is added back to the list of unmatched jobs.

For example (but you need different data/tests for your code!),

```rkt
(define alt-employers
 '(((("Manulife" (("Anna" 1) ("Feisal" 2) ("Zihan" 3) ("Hannah" 4)))
  ("Google" (("Anna" 1) ("Feisal" 2) ("Rafelia" 3)))
  ("Apple" (("Anna" 1) ("Zihan" 2) ("Feisal" 3)))))

(define alt-students
 '(((("Anna" (("Google" 1) ("Apple" 2) ("Manulife" 3)))
  ("Feisal" (("Google" 1) ("Apple" 2) ("Manulife" 3)))
  ("Rafelia" (("Google" 1)))
  ("Zihan" (("Apple" 1) ("Manulife" 2)))
  ("Hannah" (("Manulife" 1))))

The algorithm starts by offering the Manulife job to Anna, that employer’s most preferred student. Anna accepts. Next, the Google job is offered to Anna. Anna has ranked it as more preferrable to Manulife, so she jilts Manulife and accepts Google. The Manulife job goes back onto the list of unmatched employers without Anna as one of its preferences.

Assuming the Manulife job was added to the front of the list, it’s now offered to Feisal, Manulife’s most preferred student who hasn’t already turned it down. Feisal accepts because he doesn’t have a job.

The only job left is Apple. It’s offered to Anna, who declines because she likes her current match with Google better. The job goes back on the list but without Anna. So now it’s offered to Apple’s second preference, Zihan. Zihan accepts.

So Apple is matched with Zihan, Manulife with Feisal, and Google with Anna. Rafelia and Hannah were unfortunately not matched at this time.

Note that there is a change from the data defitions used earlier, namely that the preferences for both `EmpRanking` and `StdRanking` must be in increasing order and there must be at least one.

Write the following functions in `bonus1.rkt`.

- You may **not** use explicit recursion with the exception of one function in Q6a and `partition` (if you use it).
- You may **require** functions developed in previous questions.
- You are not required to encapsulate helper functions with `local`.

(a) `(match/acc emp-unmatched emp-matched students)` consumes three `(listof Ranking)`. `emp-unmatched` is a list of all the unmatched employers. `emp-matched` is a list of all
the matched employers. students is a list of all the students. Each student ranked by an employer has also ranked that employer.

If an employer is on the emp-matched list, it is matched with the student having the lowest preference in that ranking. Students who have previously declined the employer have been removed.

If an employer is on the emp-unmatched list, any students who have previously declined an offer from this employer will have been taken off the employer’s list of preferences. When emp-unmatched becomes empty, the function produces the current set of matches (emp-matched).

match/acc may use explicit recursion. Helper functions may not.

At each step the emp-unmatched list will get smaller, but sometimes it will be because a job is removed and sometimes it’s because a job is replaced but the list of preferences is shorter than it was previously. This is not pure structural recursion!

At each step the emp-matched accumulator will either grow (the job is accepted), remain unchanged (the job is declined) or change (one job is replaced with another).

(b) (match employers students) consumes a list of employer rankings and a list of student rankings. It produces a list of employer-student pairs representing the matches.

(check-expect (match employers students)
'(("Apple" "Zihan") ("Manulife" "Feisal") ("Google" "Anna"))

The order of the pairs produced does not matter.

Put your code in bonus1.rkt.

7. 5% Bonus #2 (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 include a correct contract for each function. In addition to the other restrictions in this assignment, you may not use the built-in compose function. Place your solution in the file bonus2.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.
Using the new functions you have written, together with \texttt{foldr} and other built-in Racket functions, but no other abstract list functions, give a nonrecursive definition of \texttt{my-map}, which is functionally equivalent to \texttt{map} with one list (the built-in version can be applied to multiple lists). You may not use \texttt{map} to solve this problem, perhaps not surprisingly. You may not use any helper functions or \texttt{lambda}.

The name \texttt{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.

\textbf{Enhancements:} Reminder—enhancements are for your interest and are not to be handed in.

Professor Temple does not trust the built-in functions in Racket. In fact, Professor Temple does not trust constants, either. Here is the grammar for the programs Professor Temple trusts.

\[
\langle \text{exp} \rangle = \langle \text{var} \rangle | (\lambda (\langle \text{var} \rangle) \langle \text{exp} \rangle) | (\langle \text{exp} \rangle \langle \text{exp} \rangle)
\]

Although Professor Temple does not trust \texttt{define}, we can use it ourselves as a shorthand for describing particular expressions constructed using this grammar.

It doesn’t look as if Professor Temple believes in functions with more than one argument, but in fact Professor Temple is fine with this concept; it’s just expressed in a different way. We can create a function with two arguments in the above grammar by creating a function which consumes the first argument and returns a function which, when applied to the second argument, returns the answer we want (this should be familiar from the \texttt{make-adder} example from class, slide 10-47). This generalizes to multiple arguments.

But what can Professor Temple do without constants? Quite a lot, actually. To start with, here is Professor Temple’s definition of zero. It is the function which ignores its argument and returns the identity function.

\[
(\texttt{define my-zero (lambda (f) (lambda (x) x))})
\]

Another way of describing this representation of zero is that it is the function which takes a function \texttt{f} as its argument and returns a function which applies \texttt{f} to its argument zero times. Then “one” would be the function which takes a function \texttt{f} as its argument and returns a function which applies \texttt{f} to its argument once.

\[
(\texttt{define my-one (lambda (f) (lambda (x) (f x)))})
\]
Work out the definition of “two”. How might Professor Temple define the function add1? Show that your definition of add1 applied to the above representation of zero yields one, and applied to one yields two. Can you give a definition of the function which performs addition on its two arguments in this representation? What about multiplication?

Now we see that Professor Temple’s representation can handle natural numbers. Can Professor Temple handle Boolean values? Sure. Here are Professor Temple’s definitions of true and false.

\[
\text{(define } \text{my-true } (\lambda x (\lambda y x))) \\
\text{(define } \text{my-false } (\lambda x (\lambda y y)))
\]

Show that the expression \((c a b)\), where \(c\) is one of the values \text{my-true} or \text{my-false} defined above, evaluates to \(a\) and \(b\), respectively. Use this idea to define the functions \text{my-and}, \text{my-or}, and \text{my-not}.

What about \text{my-cons}, \text{my-first}, and \text{my-rest}? We can define the value of \text{my-cons} to be the function which, when applied to \text{my-true}, returns the first argument \text{my-cons} was called with, and when applied to the argument \text{my-false}, returns the second. Give precise definitions of \text{my-cons}, \text{my-first}, and \text{my-rest}, and verify that they satisfy the algebraic equations that the regular Scheme versions do. What should \text{my-empty} be?

The function \text{my-sub1} is quite tricky. What we need to do is create the pair \((0, 0)\) by using \text{my-cons}. Then we consider the operation on such a pair of taking the “rest” and making it the “first”, and making the “rest” be the old “rest” plus one (which we know how to do). So the tuple \((0, 0)\) becomes \((0, 1)\), then \((1, 2)\), and so on. If we repeat this operation \(n\) times, we get \((n - 1, n)\). We can then pick out the “first” of this tuple to be \(n - 1\). Since our representation of \(n\) has something to do with repeating things \(n\) times, this gives us a way of defining \text{my-sub1}. Make this more precise, and then figure out \text{my-zero}.

If we don’t have \text{define}, how can we do recursion, which we use in just about every function involving lists and many involving natural numbers? It is still possible, but this is beyond even the scope of this challenge; it involves a very ingenious (and difficult to understand) construction called the Y combinator. You can read more about it at the following URL (PostScript document):

\[
\text{http://www.ccs.neu.edu/home/matthias/BTLS/tls-sample.ps}
\]

Be warned that this is truly mindbending.

Professor Temple has been possessed by the spirit of Alonzo Church (1903–1995), who used this idea to define a model of computation based on the definition of functions and nothing else. This is called the lambda calculus, and he used it in 1936 to show a function which was definable but not computable (whether two lambda calculus expressions define the same function). Alan Turing later gave a simpler proof which we discussed in the enhancement to Assignment 7. The lambda calculus was the inspiration for LISP, a predecessor of Racket, and is the reason that the teaching languages retain the keyword \text{lambda} for use in defining anonymous functions.