Assignment: 9

Due: Tuesday, March 22, 2016 9:00pm
Language level: Intermediate Student with Lambda
Allowed recursion: **None, unless otherwise stated**
Files to submit: `alfs.rkt`, `composite.rkt`, `bonus.rkt`
Warmup exercises: HtDP 19.1.5, 20.1.1, 20.1.2, 24.0.7, 24.0.8, *Without using explicit recursion:* 9.5.2, 9.5.4
Practise exercises: HtDP 19.1.6, 20.1.3, 21.2.3, 24.0.9, 20.2.4, 24.3.1, 24.3.2

For this assignment:

- You may use 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`), except for *assf* which is not defined in Intermediate Student with Lambda.
- You may use primitive functions such as mathematical functions, *cons*, *first*, *rest*, *list*, *empty?*, **local**, etc.
- You may **not** use *length*, *append*, *reverse*, *member?* and similar "non-primitive" functions.
- All helper functions must be declared using **local**, except in Question 3, where previous parts may be used as global helper functions.

Nesting abstract list functions can become confusing. Use appropriately documented helper functions liberally to help clarify your intentions.

Here are the assignment questions you need to submit.

1. The stepping problems for Assignment 9 at:

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

2. Construct the following functions using abstract list functions. Do *not* use explicit recursion (functions that call themselves, either directly or indirectly). No credit will be given for explicitly recursive solutions:

   (a) The function *x-coords-of-posns*, which consumes a list of anything at all, and produces the list of all of the x-coordinates of the *Posn*s in the list, in the same order which they occur in the consumed list.

   (b) The function *alternating-sum*, which consumes a list of numbers (*list* a_1 a_2 ... a_n) and produces the alternating sum $a_1 - a_2 + a_3 - a_4 + \cdots (-1)^{n-1} a_n$; if the list is empty, produce 0.

   (c) The function *remove-duplicates*, which consumes a list of numbers, and produces the list with all but the first occurrence of every number removed. For example,

   ```
   (remove-duplicates '(1 4 2 1 5 4)) => '(1 4 2 5)
   ```

   (d) The function *first-col* which consumes a list of non-empty list of numbers, denoting a rectangular matrix of numbers, and produces the first column of the matrix, as a list. For example,

   ```
   (first-col '((1 2 3 4)
               (5 6 7 8)
               (9 10 11 12))) => (list 1 5 9)
   ```

   (e) The function *add1-mat* which consumes a (*listof* (*listof Num*)), denoting a rectangular matrix of numbers, and produces the matrix resulting from adding 1 to every entry of the matrix. For example,

   ```
   (add1-mat '((1 2 3 4)
               (5 6 7 8)
               (9 10 11 12)))
   =>
   (list (list 2 3 4 5)
         (list 6 7 8 9)
         (list 10 11 12 13))
   ```

   (f) The function *sum-at-zero*, which consumes a list of functions $(f_1, \ldots, f_n)$ (where each consumes and produces a number), and produces the value $f_1(0) + \cdots + f_n(0)$. For example,

   ```
   (sum-at-zero (list add1 sqr add1)) => 2
   ```

   If the consumed list of functions is empty, produce 0.

   Place your solutions in the file `alfs.rkt`.

3. In mathematics, function composition is the application of one function to the result of another. Given two functions f and g, one can obtain the composite function $(f \circ g)(x)$ by computing $f(g(x))$ for any given x. For this question, we assume f and g consume one parameter each, and the result of g can be consumed by f.

   (a) Write the function *composite* that consumes two functions f and g and produces the composite function $(f \circ g)$. For example, (*composite add1 abs*) produces a function that maps a number n to the number $|n| + 1$.

   (b) Use *composite* as a helper function and write a function *inverse-of-square-list* that consumes a list of *positive* numbers and produces a new list of positive numbers with each element being the inverse of the square of the corresponding element in the original list. For example,

   (*inverse-of-square-list* '(1 2 5)) => '(1 0.25 0.04)

   You *must* use *composite* in a non-trivial way for full marks.

   (c) Write the function *composite-list* that consumes a list of functions $(f_1, \ldots, f_n)$ (where each consumes and produces a number), and produces the composite function $(f_1 \circ \ldots \circ f_n)$. If the list is empty, *composite-list* should produce the identity function $f(x) = x$. You may use *composite* as a helper.

   You may **not** use any explicit recursion in writing the above functions. Note that when it comes to testing the functions from parts (a) and (c), you will run into difficulties, since *check-expect* cannot verify that two functions are the same. Instead, you can test the functions you get from *composite* and *composite-list* by calling them and comparing the result with your expectations. For example:

   (*check-expect* ((*composite abs sub1*) −5) 6)

   Place your solutions in the file `composite.rkt`.

4. **5% Bonus**: Consider the following family of functions:

   (**define** *c0* (**lambda** (*f*) (**lambda** (*x*) *x*)))
   (**define** *c1* (**lambda** (*f*) (**lambda** (*x*) (*f x*))))
   (**define** *c2* (**lambda** (*f*) (**lambda** (*x*) (*f* (*f x*)))))
   (**define** *c3* (**lambda** (*f*) (**lambda** (*x*) (*f* (*f* (*f x*))))))
   (**define** *c4* (**lambda** (*f*) (**lambda** (*x*) (*f* (*f* (*f* (*f x*)))))))
   . . .
   (**define** *ck* (**lambda** (*f*) (**lambda** (*x*) (*f* . . . (*f x*) . . .))))

   where *f* is applied *k* times in *ck*.

   To earn credit for this question, you must answer both of the following parts correctly, and for this question we will be looking for the full-design recipe:

   (a) Write the function *c->nat* that consumes a function *cj* above and produces the natural number *j*.
   (b) Write the function *nat->c* that consumes a natural number *j* and produces the function *cj*.

   You may use structural recursion for part (b), but do **not** use recursion for part (a).

   Place your solution in the file `bonus.rkt`.