CS 135
Winter 2014
Troy Vasiga

Assignment: 8

Due:	Tuesday, March 25, 2014 9:00pm
Language level:	Intermediate Student with Lambda
Allowed recursion:	**None**
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 in DrRacket 5.3.
- You may use primitive functions such as mathematical functions, *cons*, *first*, *rest*, *list*, *empty?*, *local*, etc.
- You may **not** use *length*, *append*, *reverse*, and similar "non-primitive" 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 8 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 (*listof* (*ne-listof Num*)), 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)

(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*) ...))))

To earn credit for this question, you must answer both of the following parts correctly:

(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). If you intend to attempt this question, you may wish to first read the enhancements.

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

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

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

⟨exp⟩ = ⟨var⟩|(**lambda** (⟨var⟩) ⟨exp⟩) | (⟨exp⟩⟨exp⟩)

Although Professor Temple does not trust **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 *addgen* example from class, slide 09-39). 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.

(**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 f as its argument and returns a function which applies f to its argument zero times. Then "one" would be the function which takes a function f as its argument and returns a function which applies f to its argument once.

(**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.

(**define** *my-true* (**lambda** (*x*) (**lambda** (*y*) *x*)))
(**define** *my-false* (**lambda** (*x*) (**lambda** (*y*) *y*)))

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

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

The function *my-sub1* is quite tricky. What we need to do is create the pair $(0, 0)$ by using *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 *my-sub1*. Make this more precise, and then figure out *my-zero?*.

If we don't have **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):

`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, the predecessor of Scheme, and is the reason that the teaching languages retain the keyword **lambda** for use in defining anonymous functions.