Assignment: 09
Due: Friday, April 3, 2020 11:59 pm
Language level: Intermediate Student with Lambda
Allowed recursion: No explicit recursion, see below
Files to submit: debug-a09.rkt, hoarding.rkt
Warmup exercises: Without using explicit recursion complete HtDP 9.5.2, 9.5.4, and write your own versions of member? and append.
Practice exercises: HtDP 21.2.1, 21.2.2, and 21.2.3

For this assignment:
• Make sure you read the OFFICIAL A09 post on Piazza for the answers to frequently asked questions.
• You may use only the following abstract list functions: build-list, filter, map, foldr, and foldl.
• You may use any primitive functions, such as mathematical functions, cons, first, second, third, list, empty?, and equal?.
• You may use cond.
• You may use lambda.
• You may not use explicit recursion for any question. That is, functions that involve an application of themselves, either directly or via mutual recursion. Use the abstract list functions instead.
• You may not use any non-primitive list functions, including length, member?, reverse, append, and list-ref.
• You may only define non-local constants and helper functions if they will be used to answer multiple parts of the same question. Otherwise, constants and helper functions should be defined locally. Local helper functions do not require purposes, contracts, examples or tests.
• You may use functions defined in earlier parts of a question to help write or test functions later in the same question.

1. [20% Correctness] Question 1

Consider the code in debug-a09.rkt, which may contain syntax, mathematical, and style errors. The code may also be difficult to read and/or maintain, lack helper functions and constants. Correct the code.
2. [60% Correctness] Question 2

A panic has struck your city. Grocery stores everywhere are under attack. Let’s build some quick functions to help them out.

```racket
(define-struct item (name cost taxable count))
;; An Item is a (make-item Str Num Bool Nat)
;; requires:  Num >= 0, Str does not equal "", Nat > 0
;; where: Str is the name
;;        Num is the price
;;        Bool is true if the item is taxable
;;        Nat is the quantity to purchase

;; A GroceryList (GL) is a (listof Item)
;; requires:  item names are unique

;; A GItem is a (list Str Num Bool Nat)
;; requires:  Num >= 0, Str does not equal ""
;; where: Str is the name
;;        Num is the price
;;        Bool is true if the item is taxable
;;        Nat is the count in stock

;; An Inventory is a (listof GItem)
;; requires:  item names are unique
```

Note that the names of items in GroceryList are unique, as are those in an Inventory.

(a) Write a function `sum` that consumes a GroceryList and computes the total cost of the items in the list before tax.

(b) Write a function `total-bill` that consumes a GroceryList and computes the total cost of the items in the list after tax. The only abstract list functions you may use are `map` and `foldl`.

Note that tax does not apply to all items in the list. The tax rate is given in the starter file as `(define tax 0.13)`.

(c) Repeat (b), but the only abstract list function you may use is `foldr`. Call this function `total-bill-alt`.

(d) Write a function `in-stock?` that consumes the name of an item, as a string, and an Inventory, and produces true if the item is in stock.

Note: an item is in stock if there is at least one of that item in the inventory.

(e) Write a function `can-fulfill?` that consumes a GroceryList and an Inventory and produces true if the order can be fulfilled, that is, all items in the grocery list are in stock in the requested quantities.

Place your solutions in `hoarding.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 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 `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 `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.

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

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.

\[
\text{(define my-one (lambda (f) (lambda (x) (f x))))}
\]

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.

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

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

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

If we don’t have \textit{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 \textit{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 \textit{lambda} for use in defining anonymous functions.