Exercise 3: More with Higher-Order Functions; Building Environments

In this exercise, you will gain a deeper understanding of using an *environment* to both create and resolve name bindings. You'll also use higher-order functions to implement *currying* in Racket, a pretty useful language feature that is actually built into Haskell by default. *Please complete Lab 3 before starting this exercise.*

Deadline: October 1, 2019 before 10:00pm

Starter code

- ex3.rkt

Task 1: Calculator III. Building an environment

Consider the following grammar that we introduced in lab 3:

```
Expanded Binary Arithmetic Expression Grammar

<expr> = NUM
       | ID
       | (<op> <expr> <expr>)
       | (if (<comp> <expr> <expr>) <expr> <expr>)
       | (let* ((ID <expr>) ...) <expr>)
<comp> = = | > | <
<op>   = + | - | * | /
```

Where `NUM` represents any integer literal in base 10, `ID` is a variable name (a symbol in Racket), and the arithmetic operations `+`, `-`, `*`, `/` have the standard mathematical meanings. Likewise, the comparison operations `=`, `>` and `<` have their standard mathematical meanings as well.

The semantics of a `let*` binding is just like in Racket:

- The expression `(let* ((ID <expr1>)) <expr2>)` means to evaluate `<expr2>`, with a new environment. This new environment contains all the bindings in the old environment and one more: `ID` bound to to the value of `<expr1>`.
- The expression `(let* () <expr>)` just evaluates to `<expr>`. This is the case with no new bindings.
- The expression `(let* ((ID <expr1>) (ID <expr>) ...) <expr2>)` is a shorthand for `(let* ((ID <expr1>)) (let* ((ID <expr>) ...) <expr2>))`, whose semantics we described earlier.

For example, `(let* ((a (* 4 3))) (+ a 1))` means:

- Create a new environment with all the prior bindings, but with `a` bound to the value of `(* 4 3)`
- Evaluate `(+ a 1)` with this new environment.

Modify your function `eval-calc` from lab 3 to handle `let*` expressions.

(Note: You should never use the built-in `eval` interpreter in Racket.)

Task 2: Currying

One of the common uses of higher-order functions that return functions is to enable the *partial application* of a function, in which we fix values for certain arguments to a function, returning a new function that just takes the remaining arguments.

You did this in Lab 2. We started with `(count-pred pred lst)`, which is a binary function that takes a predicate and a list, and returns the number of items in the list that satisfies the predicate. But we wanted to write functions that fixed the value of the predicate, and that just take in the list:

```
(define (make-counter pred)
  (lambda (lst)
    (count-pred pred lst)))

(define count-evens (make-counter even?))
```

If we abstract away this specific context, we get an interesting pattern for binary functions:

```
; A normal binary function
(define (f x y) ...)

; A version that takes arguments one at a time, using an intermediate function.
(define (f1 x)
  (lambda (y)
    (f x y)))

; Or to make the pattern more obvious, separating (f1 x) into pure `lambda` form:
(define f1
  (lambda (x)
    (lambda (y)
      (f x y))))
```

`f1` represents a function that takes a single argument `x`, and returns a new function that takes a single argument `y`, that returns `(f x y)`. In other words, the expressions `(f x y)` and `((f1 x) y)` are equal.

We call `f1` the *curried form* of `f`, a transformation of `f` into a sequence of unary functions that "fill in" the arguments to `f` one at a time, from left to right. Currying isn't the only kind of partial application—one could imagine filling in the arguments to `f` from right to left, for example—but it turns out to be a common and useful technique, and is implemented by default (!) for all functions in Haskell. Currying can be extended more generally to functions of any arity to convert them into a sequence of unary functions:

```
; Standard form
(define (g x1 x2 x3 ... xn) ...)

; Curried form
(define curried-g
  (lambda (x1)
    (lambda (x2)
      (lambda (x3)
        ...
          (lambda (xn)
            (g x1 x2 x3 ... xn)))))
```

Note the recursive structure here! Your task for this part is to implement a *currying function*, a higher-order function that transforms a given function into its curried form (roughly, takes in `g` and returns `curried-g`). As usual, see the starter code for details—this is a fairly abstract problem, so we've broken this down by having you implement some simpler functions first.