Exercise 3: More with higher-order functions, building environments

In this exercise, 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.* This exercise is a bit longer, but Task 1 in particular will be very useful for Assignment 1!

Deadline: Sunday January 28 before 10pm

Starter code

This week's exercise is Racket-only.

- ex3.rkt

Task 1: Building an environment

Consider the following grammar for a simple set of name bindings:

```
<prog> = <binding> [<binding> ...]   # A program consists of one or more name bindings.
<binding> = (define ID <expr>)
<expr> = NUMBER | ID
```

That is, we permit bindings of names to integer literals or to the same value as another name. In order for the program to be semantically-valid, names must be defined *before* they are referenced. For example,

```
(define x 3)
(define y x)
```

is permitted, while

```
(define y x)
(define x 3)
```

is not. Also, no name may be bound more than once.

For this task, the *denotational value* of a program is considered to be its environment: a mapping of names to *integer* values determined by these bindings. Your task is to write a function that takes a list bindings, and *evaluate* them to produce an environment.

Task 2: Currying

One of the most 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: `(count-pred pred lst)` 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 f2
  (lambda (x)
    (lambda (y)
      (f x y))))
```

Note that `f1` and `f2` represent the exact same value: 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)`, `((f1 x) y)`, and `((f2 x) y)` are all equal.

We call the `f1`/`f2` function 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 arbitrary-arity functions 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.

For course-related questions, please contact **david at cs.toronto.edu**.