Lab 2: Higher-Order Functions, and More List Practice

Last week, we looked at performing computations using *pure functions*, even when the input was unbounded (requiring us to use recursion). In this lab, you'll start looking at how to move away from writing recursion manually, by using higher-level abstractions central to functional programming.

Starter code

- lab2.rkt

Task 1: Practicing with higher-order list functions

Review the code you wrote for `num-evens` and `num-many-evens` on Exercise 1. The structure should look basically identical, except one part: the predicate (boolean function) you are using to check the list elements. This sort of repeated functionality begs to be abstracted, and (so far) we only have one tool for code abstraction: writing a function, with parameters for the parts of functionality that change. In this case, the part we want to change is the predicate—this means we'll need to pass these predicates directly to our generalized function. In other words, we'll need to write a function that *takes another function as an argument*. This is a powerful idea. In functional programming, functions are just another type of value, and so if other values like numbers and strings can be passed as arguments, then so can functions.

1. To check that you understand this, write a function `num-pred`, which takes a (unary) boolean function and a list, and returns the number of items in the list that make the function return `#t`.

   We call `num-pred` a *generalization* of `num-evens` and `num-many-evens` because we should be able to implement both of the latter two functions using `num-pred`, just by passing in the right predicate as an argument. For example:

   ```
   (test-equal? "num-pred/num-evens"
     (num-evens (list 1 2 3 4 5))
     (num-pred even? (list 1 2 3 4 5)))

   (test-equal? "num-pred/num-many-evens"
     (num-many-evens (list (list 1 2 3 4 5) (list 2 4 6 10))
     (num-pred ...   (list (list 1 2 3 4 5) (list 2 4 6 10))))
   ```

2. In the second test above, replace the `...` with either a lambda expression or a helper function so that the test passes. That is, we want you to understand how to *really* generalize `num-many-evens` to `num-pred`.

Task 2: Another type of higher-order function

The work in the previous task is a good, but slightly unsatisfying, generalization. Suppose we have `num-pred`, and use it to count the number of even elements in several different lists:

```
(num-pred even? (list 1 2 3 4 5))
(num-pred even? (list))
(num-pred even? (list 100 300))
(num-pred even? (list 1 2 3 4 5 6 7 8 9 10))
(num-pred even? (list -1 -2 -3 -4 -5 -6 -7))
```

Instead of repeating the `num-pred even?` part over and over, we can define the following function:

```
(define (num-evens lst)
  (num-pred even? lst))
```

Of course, we could do the same for `num-many-evens`:

```
(define (num-many-evens lst)
  (num-pred ... lst))
```

This is actually a fairly standard idea in functional programming: if you have a multi-parameter function in which one (or more) parameters are frequently passed on the same value (e.g., passing `even?` for `pred`), we can define a new function that *fixes* some of the arguments to the old function, and takes in only the remaining ones. (This is very much related to the idea of *currying*, which we'll discuss in lecture.)

But if you look at these two definitions for `num-evens` and `num-many-evens`, and consider taking the same approach for defining, say, a `num-odds` or `num-greater-than-3`, there is still some repetition: `lst`. Even though `num-pred` has allowed us to abstract away the `(if ... (+ 1 ...) ...)` part, it hasn't abstracted the process of function creation itself.

In other languages, we might be forced to give up here; but because Racket functions are first-class values, there is nothing stopping us from defining a function that *returns another function*.

1. Define a function `make-counter`, which takes as input a predicate, and returns a *function* that takes a list and returns the number of elements in the list satisfying that predicate.

   ```
   (define (make-counter pred)
     ...)
   ```

2. Show how to use `make-counter` to define `num-evens` and `num-many-evens` in a very concise way.

The ability of a function to return other functions is one of the hallmarks of functional programming; it opens up new opportunities for abstraction, but raises some interesting questions when it comes to *how* these things are actually implemented. More on this in lecture.

Task 3: Racket code as data

One of the features of the Racket programming language is its *homoiconicity*, which roughly means that the source text has identical structure to the program it represents. In Racket, this comes through the fact that its primary unit of code, the expression, is recursive: it is either an atomic value (e.g., identifier, literal) or a pair of parentheses containing zero or more expressions. That is, program text is a *nested list*, which we think of as being a tree structure, where each leaf is an atomic value, and each internal node represents a parenthesized expression. Because of homoiconicity, it is easy to obtain the nested list representation of a Racket expression: we simply add an apostrophe to the start of the expression, which we call *quoting* the expression:

```
> (+ 1 2)  ; A regular Racket expression
3
> '(+ 1 2) ; Quoting the expression: results in a list of three elements.
'(+ 1 2)
> (list? '(+ 1 2))
#t
> (first '(+ 1 2))
'+
> (second '(+ 1 2))
1
> (third '(+ 1 2))
2
```

We call the result of quoting a Racket expression a *datum*, to distinguish it from other Racket lists. Here, the datum `'(+ 1 2)` is a list containing three elements: the *symbol* `'+`, and the *numeric literals* `1` and `2`. Symbols are a primitive datatype in Racket that can be compared against other symbols for equality, and that's about it—they are similar to

enumerations in other languages. Quoting has no effect on literals (because they're already evaluated as-written): `(equal? '1 1)` returns true.

You have two subtasks here, both practicing using recursion on nested lists. See the starter code for details.

1. Perform a basic *static analysis*: given a datum, return a list of all numeric literals it contains. Duplicates are permitted.
2. Perform a basic *code transformation*: given a datum, return a copy of it with every occurrence of the numeric literal `324` replaced by `9000`.