Lab 4: A Basic Object-Oriented System

In this lab, you'll get started looking at a particularly famous application of lexical closures to implement a programming paradigm you're already familiar with: *object-oriented programming*. We'll build on what you do here during lecture this week.

Starter code

- lab4.rkt

Task 1: Closures as objects

One of the consequences of closures is that it is possible to store values in a function closure, in an analogous fashion to storing data in an object's attributes in object-oriented programming.

The simplest sort of object we can define is a *struct*, a compound data type that consists of one or more values that we call *fields* or *attributes*. You can see a simple example of a struct representing a point in the starter code. Your task here is to understand, use, and add to this definition by completing the following exercises.

0. Use `Point` to define a value representing the point (1, -3).
1. Write a function that takes in one `Point` value and returns the sum of its two coordinates.
2. Write a function that takes in two `Point` values and returns the distance between them.
3. Write a function that takes in a positive integer `n` and returns a list of points with coordinates (0, 0), (1, 1), ..., (n-1, n-1).
4. A *method* is an attribute whose type is a function. (Many languages clearly delineate non-function attributes and methods because they treat functions as different from other values. As we treat functions as first-class values, we won't need to make such a distinction.)

   Add to the definition of the `Point` function itself so that calling a point value on the argument `'scale` behaves as follows:

```
(define p (Point 2 3))      ; p is the point (2, 3)
(p 'scale)                  ; #<procedure...>

(define p2 ((p 'scale) 3))  ; p2 is the point (6, 9)
```

Task 2: The attribute `__dict__`

You might notice that our sequence of `cond` branches is really a drawn out version of implementing a key-value lookup. We can improve both the elegance and efficiency of our `Point` implementation by using an explicit hash table to store the attribute names and their corresponding values.

Your second task in this lab is to do exactly this! More concretely, in the starter code we've given you space to write a second implementation of our `Point` class called `PointHash` that should use this hash table-based approach. *`PointHash` should have the same public behaviour as `Point` as described in the previous section.*

Using the naming convention of Python, use the name `__dict__` to refer to the hash table you'll create. Note that for now, you should store both the simple attributes (`'x` and `'y`) as well as methods (`'scale`) in the same hash table. You might have questions here, especially if you've seen `__dict__` before in Python—more on this in lecture.