Lab 4: A Basic Object-Oriented System; Syntax Trees as Data Types

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. In the second part of the lab, you'll see how Haskell's data types can make explicit the *structural recursion* specified by language grammars, and then naturally lead to pattern-matching implementations of the corresponding functions. Hopefully this will help clarify some aspects of this style of programming to help with Assignment 1!

Starter code

- lab4.rkt
- Lab4.hs

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 type 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 (i.e., the [function] values returned by calling `Point`) 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. Because 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))     ; The point (2, 3)
(p scale)                  ; #<procedure...>

(define p2 ((p scale) 3))  ; The point (6, 9)
```

Task 2: Abstract Syntax Trees as types

So far, we have been taken advantage of the homoiconicity of Racket to directly analyse and manipulate Racket programs in the form of *nested lists*. One disadvantage of this approach is that

the representation is built upon on the list data type, which is often *too flexible* for our purposes (think back to your previous exercises and bugs you found by confusing a datum with a list of datum, of example).

We have previously seen how to formally represent recursive structures in Haskell's type system. In this task, you'll begin to see how to do the same to represent a program in the same style, defining and using what we can an *Abstract Syntax Tree*.

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