Lab 5: Macros, Macros, Macros

In this lab, you'll get some valuable hands-on experience working with pattern-based macros. Note that this kind of macro is conceptually similar to the pattern-matching we've seen so far, but there are some differences that you'll encounter as you write your own macros!

Starter code

As you might expect, this week is Racket-only, as we're taking advantage of Racket's very impressive macro system.

- `lab5.rkt`

Task 1: Macro practice

Recall the following four syntactic forms in Racket: `or`, `and`, `if`, and `cond`. Earlier in the course, we discussed that due to *strict function call semantics* in Racket (and most other programming languages), these four cannot be implemented as functions because we want each one to exhibit some kind of "short-circuiting" behaviour. However, you might wonder if all four of these are really necessary as fundamental language primitives, and unsurprisingly, the answer is no. In fact, `if` can be used to implement each of the other three!

For example, consider "P or Q". We can express the exact same statement as "if P then true else Q": that is, first evaluate P, and if it's `true` then return `true`, else return whatever Q's value is. Take a minute to convince yourself that indeed the second statement is logically equivalent to the first (there are only 4 cases).

But notice that "implement" here means something different than simply writing functions. **What's the problem with this implementation of `or`?**

```
(define (my-or p q)
  (if p
      #t
      q))
```

Rewriting

Even though our intentions are good, using a function to "transform" an instance of `or` into an instance of `if` doesn't quite work. Technically speaking, the problem is that even though we're used to modeling function evaluation as substitution, the function call semantics of Racket interferes with the desired short-circuiting behaviour for `or`.

So instead, we use a **macro** to transform the `or` expression into an `if` expression, **before the program is evaluated**. If you're familiar with macros in other languages like C, you can think about the Racket macro system as a search-and-replace source code rewriting system, but better.

Here's the syntax for defining a macro, as we saw in lecture. (There's a slightly shorter version that can be used some of the time, but we're just covering the general technique here.)

```
(define-syntax my-or
  (syntax-rules ()
    [(my-or <p> <q>)
     (if <p> #t <q>)]))
```

The first line is straight-forward: it tells Racket the name of the macro: in this case, `my-or`. The second line uses the `syntax-rules` form, which takes some *literal keywords* (none in this example), and then pairs of `[<pattern> <template>]` expressions, representing the rules for our macro. *Before* the program is run, Racket takes every occurrence of the macro pattern, and replaces it with its corresponding template—without evaluating anything!

Using this idea, complete the described macros under the "Task 1" heading of the starter code, using only the built-in `if`, and *not* using any of `or`, `and`, and `cond`.

Task 2: Extending the `my-class` macro

Under the Task 2 heading, we've provided the `my-class` macro from the end of last week's lecture. One of the limitations of the current macro is that it hard-codes the exact form of the *constructor* of the class: it must always take in arguments corresponding exactly to the fields of the class, making the fields entirely transparent to the programmer and therefore breaking encapsulation.

A first step in addressing this issue is to decouple the declaration of the attributes from the constructor itself. In object-oriented programming, the main purpose of a constructor is to *initialize instance attributes*, typically done through statements like `this.x = 3`. (We ignore additional purposes for the constructor that appear in other languages, such as allocating memory for new objects.) But because we still aren't using mutation, we'll take a different view of constructors: a constructor is a function that *returns a hash table* mapping attribute names to values. Note that we've already given this hash table a name in our macro: `__dict__`.

For example, a typical Python constructor like the following:

```
class A:
    def __init__(self, x, y):
        self.a = x + y
        self.b = y * 4
        self.c = []
```

would be translated into something like

```
class A:
    def __init__(self, x, y):
        return {'a': x + y,
                'b': y * 4,
                'c': []}
```

Your task here is to add a new rule to the given `my-class` macro so that it supports the following syntax (`new` is a *literal keyword*):

```
(my-class Point
  ; The constructor must be the *first* method in the class.
  (method (new x y)
    (hash 'x x
          'y y
          'z 0
          'extra (+ x y)))
  ; Zero or more other methods can be included...
  )


; Sample usage
(define p (Point 2 3))  ; `Point` is a function with the same signature as `new`.
(p 'x)                  ; 2
(p 'y)                  ; 3
(p 'z)                  ; 0
(p 'extra)              ; 5
```

You'll likely find that your implementation no longer allows methods to access all instance attributes (*why?*)—we'll fix that when we introduce `self` next week!