Exercise 6: Object Oriented Macro

In this exercise, we will be working with pattern-based macros, including the OOP macro from clas. 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

Download these files from Quercus files.

- `Exercises/ex6/ex6.rkt`

Submission instructions

Submit the file `ex6.rkt` to MarkUs: https://mcsmark.utm.utoronto.ca/csc324f20/

But before you submit, check that:

- Your code does not have any syntax error. If you only completed part of the exercise, replace any possibly syntactically incorrect code with the starter code.
- The `(provide ...)` code in your Racket file is left intact. Otherwise we won't be able to run our tests!

Task 1: Macro Practice (4 pt)

Write the macros `my-or*`, `my-and*` in terms of `if`. This is an extension of the exercise from week 6 lectures. You may *not* use Racket `and`, `or`. The macro should properly short-circuit.

Then, write the macro `lazy-curry` that takes a function and some argument expressions. The argument returns a new procedure in terms of the remaining arguments to the function. This macro should *not* evaluate the argument expressions until the returned procedure is called.

You might find the following Racket syntax and functionalities helpful:

The higher-order function *apply* takes a function, and a list of arguments, and calls the function with those arguments.

```
(apply f '(3 4 5)) ; same as (f 3 4 5)
```

If we omit the brackets around the argument in an anonymous lambda definition, the parameter represents the list of arguments passed in. This syntax allows us to create *variadic functions*, or functions that can take a variable number of arguments:

```
> (define f (lambda xs xs)) ; NOT the same as (lambda (x) x)
> (f 3 4)
'(3 4)
> (f 3)
'(3)
```

Task 2: Working with the class macro (3 pt)

In this task, we will practice using the `my-class` macro from lecture. Use the `my-class` macro to create a class `Walrus` with the following attributes:

- `name`: the name of the Walrus (string)
- `age`: the age of the Walrus (integer)
- `bucket`: a list of item in the Walrus' bucket (a list of strings)

The class should also have the following methods:

- `bucket-empty?`: returns a boolean describing whether the Walrus' bucket is empty
- `same-name? other`: take a single parameter of type `Walrus`, and checks if the other Walrus has the same name

- `equal? other`: take a single parameter of type `Walrus`, and checks if the other Walrus has the same name and age. The other Walrus may have a different list of items in its bucket.
- `catch item`: take a single parameter of type string, and returns a new Walrus that as the new item inserted to the beginning of the list of bucket items.
- `have-birthday`: returns a new Walrus that is the same as the old Walrus, but with the age incremented by one, and the string "cake" inserted to the beginning of the list of its bucket items.

Task 3: Accessor Functions in `my-class` (3 pt)

Implement a new macro called `my-class-getter`. This macro has the same functionality as `my-class`, except that in addition to defining a class constructor function `<Class>`, it *also defines accessor functions* for each attribute of the class, using the name of the attribute as the name of its corresponding accessor. For example:

```
(my-class-getter Point (x y)
  ; Zero or more methods, omitted here
  )

(define p (Point 2 3))
(p 'x)            ; 2
(x p)             ; also 2
```

The starter code uses the Racket syntax `begin`, which takes several expressions as argument, evaluates *each* expression, and returns the result of the final one. For example, the expression `(begin (+ 1 2) (+ 3 4))` means to evaluate `(+ 1 2)`, then evaluate `(+ 3 4)`, and then return the final result 7. A `begin` expression can be used like this:

```
(begin
  (define a 3)
  (+ a 4))
```

(As a side note, this naming convention can lead to name collisions, a common problem with Haskell records. The Scheme convention is to name accessors `Point-x` and `Point-y` instead of `x` and `y`. However, doing so requires using a more powerful way of defining macros.)

One other strange issue that you might run into is that definitions inside macros handled differently depending on whether the new identifier being bound is user-defined.

For example, consider these two macros:

```
(define-syntax foo
  (syntax-rules ()
    [(foo <name>) (define <name> 3)]))

(define-syntax bar
  (syntax-rules ()
    [(bar <name>) (define myname 3)]))
```

In the first macro `<name>` is supplied by the macro's user, so Racket will make the new definition global:

```
> (foo a)
> a
3
```

In the second macro, `myname` is not supplied by the macro's user, so Racket will assume that `myname` should be local.

```
> (bar a)
> myname
. . myname: undefined;
 cannot reference an identifier before its definition
```