CS 1102. Fall 2013

Assignment #6

Due: October 16 @ 12:00 p.m. (noon)

Assignment Goals

- To give you practice with macros

  Hint: If you are having trouble debugging your macros, you can try using the Macro Stepper in DrRacket. If you click "Next Term", it will sequence through your toplevel expressions and show you what each use of macro expanded to.

You should use the full Racket language for this assignment!

The Assignment

1. **Implementing Objects Through Macros**

   Despite their syntactic differences, functional and object-oriented programs are more similar than you probably think they are. What do objects do? They group together data and functions into one piece of data, and you access methods by sending messages to objects. Whereas in functional programs, we might write

```
;; A dillo is a (make-dillo number boolean)
(define-struct dillo (length dead?))

;; longer-than? : dillo number -> boolean
;; is dillo longer than given length
(define (longer-than? adillo len)
  (> (dillo-length adillo) len))

;; run-over : dillo -> dillo
;; return dead dillo one unit longer than given dillo
(define (run-over adillo)
  (make-dillo (+ (dillo-length adillo) 1) true))
```

   We could also have written this in object-style in Racket using functions to support messages:

```
(define make-dillo-obj
```

```
  (lambda (length dead?)
    (lambda (message)
      (cond [(symbol=? message 'longer-than?)
             (lambda (len) (> length len))]
            [(symbol=? message 'run-over)
             (lambda () (make-dillo-obj (+ length 1)
true))])))))

(define d1 (make-dillo-obj 5 false))
((d1 'longer-than?) 6)
((d1 'longer-than?) 5)
(define d2 ((d1 'run-over)))
((d2 'longer-than?) 5)
```

While you may believe that this example has the *spirit* of objects, it certainly doesn't look very convincing. Your job is write macros that provide a better syntax for defining object-oriented classes in Racket. Your macros should support the following alternative syntax:

```
(define dillo-class
  (class (initvars length dead?)
         (method longer-than? (len) (> length len))
         (method run-over () (dillo-class (+ length 1)
true))))

(define d3 (dillo-class 5 false))
(send d3 longer-than? 6)
(send d3 longer-than? 5)
(define d4 (send d3 run-over))
(send d4 longer-than? 5)
```

To do this, you should implement two macros, one for `class` (which, for example, converts the class expression above to the toplevel `lambda` in the `make-dillo-obj` definition above) and one for `send` (which you can use in your interactions).

Your macros should allow someone to define a class with any number (including 0) of initvars and any number (including 0) of methods. Your macros should be such that running the two versions of the dillos code shown above (the `make-dillo-obj` version and the `dillo-class` version) with their sample interactions should produce the same answers.

For full credit add *error checking* to your macro implementation to prevent "abstraction leakage". For example, what happens if you try to invoke an undefined method, e.g, `(send d3 run-dover)`?

2. **Macros for Checking Access Policies**

A software company wants to develop a program to restrict the updates that its employees can make to code, documentation, and status reports. A set of company-defined access rules determine which employees can update which types of files. For example, the company might specify that

- programmers can read and write code and documentation
- testers can read code and write documentation
- managers can read and write reports
- managers can read documentation
- ceos can read and write all files

The company proposes the following language for writing down policies (this example captures the above set of 5 rules).

```
(define check-policy
  (policy-checker
   (programmer (read write) (code documentation))
   (tester (read) (code))
   (tester (write) (documentation))
   (manager (read write) (reports))
   (manager (read) (documentation))
   (ceo (read write) (code documentation reports))))
```

Each policy specifies a program that can be used to check whether a particular job can perform an access to a certain kind of file. For example, the following interaction uses the policy defined above:

```
> (check-policy 'programmer 'write 'code)
true
> (check-policy 'programmer 'write 'reports)
false
```

Write a macro for `policy-checker` such that this example and interaction works as shown. Your macro should be able to support any policy of this general format (in other words, it should allow different numbers of roles, rules, kinds of files, and kinds of accesses). Note that you are *not* writing a macro `check-policy`: in the example above, `check-policy` is the name given to the program produced by the macro, so you can use it to check access permissions as shown in the sample interaction.

For full credit add *error checking* to your implementation to prevent "abstraction leakage". For example, what happens if the programmer mistypes `write` as `wite`? Do as much error checking as you

can *without* changing the policy language above and *without* assuming any fixed set of roles, kinds of files or kinds of access.

What to Turn In

Turn in a single file *hwk6.rkt* containing your answers. Make sure that all students' names are in a comment at the top of the file.