MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall 2007

Recitation 17 — 11/1/2007 Solutions
Message Passing Objects

Earlier we've seen methods for abstracting data objects and also for procedures. Today we'll combine the two, using an object system quite similar to the objects used in the (forthcoming) project 4, also known as the adventure game. Objects combine data and procedural abstractions. What does that mean? Object-oriented programming allows us to combine the procedures with the data it manipulates. In effect, this is one way to enforce that the abstractions are followed.

Here's a start of defining a set of objects, in normal Scheme:

```
(define (make-named name)
  (lambda (msg)
    (cond
      ((eq? msg 'NAME) (lambda () name))
      (else (error "no handler for " msg)))))

(define (make-animal name)
  (let ((named-part (make-named name)))
    (lambda (msg)
      (cond
        ((eq? msg 'DRINK) (lambda () (display "slurp")))
        ((eq? msg 'EAT) (lambda (object)
                          (display "crunch crunch ")
                          (display ((object 'NAME)))
                          (display " was yummy")))
        (else (named-part msg))))))

(define fluffy (make-animal 'fluffy))
(define lunch (make-named 'pizza))
```

In objects terminology, we'd say that there are two classes, `named`, and `animal`. We also have two objects, `fluffy`, which is an instance of an `animal`, whereas `lunch` is an instance of the `named` class.

Problem: How would you ask `fluffy` to return his name?

`((fluffy 'NAME))`

Problem: How would you ask fluffy to eat lunch?

`((fluffy 'EAT) lunch)`

As we can quickly see, defining objects this way will work, but the interface is rather clunky.

Instead, we'll build on a set of abstractions for creating and manipulating objects. Here's what the same class definition would look in the project 4 system. We won't talk at all today about how the various provided procedures are defined.

```
(define (create-animal name)
  (create-instance animal name))

(define (animal self name)
  (let ((named-part (named-object self name)))
    (make-handler
     'ANIMAL
     (make-methods
      'DRINK (lambda () (display-message (list "slurp")))
      'EAT (lambda (x) (display-message
                        (list "crunch crunch "
                              (ask x 'NAME)
                              " was yummy")))
      )
     named-part)))

(define fluffy (create-animal 'fluffy))
(define lunch (create-named-object 'pizza))
```

Problem: How would you ask fluffy his name and to eat lunch now?

```
(ask fluffy 'NAME)
(ask fluffy 'EAT lunch)
```

Note that that the constructor for every class we will define takes as it's first argument an object called `self`. Draw a diagram of the methods that fluffy can respond to, and try to come up with examples of when the `self` pointer would be useful.

Now here's a `cat` class that derives from the `animal` class, and specific instance of a cat, `garfield`.

```
(define (create-cat name)
  (create-instance cat name))

(define (cat self name favorite-food)
  (let ((animal-part (animal self name))
        (mood 0))
    (make-handler
     'CAT
     (make-methods
      'FETCH
      (lambda (x)
        (display-message (list "Go fetch" (ask x 'NAME) "yourself"))
        (set! mood (- mood 3)))
      'MOOD
      (lambda () (if (>= mood 4) 'content 'angry))
      'INSTALL
      (lambda ()
        (ask animal-part 'INSTALL)
        (display-message (list "I am" (ask self 'NAME) "yaawn"))
        (ask our-clock
             'ADD-CALLBACK
             (create-clock-callback 'EAT-CB self 'EAT)))
      'EAT
      (lambda ()
        (ask animal-part 'EAT favorite-food)
        (set! mood (+ mood 1)))
      )
     animal-part
     )))

(define lasagna (create-named-object 'lasagna))
(define garfield (create-cat 'garfield lasagna))
```

Note that there are references to additional classes that aren't provided here (namely, clocks and clock callbacks).

1. What will the following evaluate to:

```
(ask garfield 'MOOD)   -> angry
(ask our-clock 'TICK)  -> crunch crunch lasagna was yummy
(ask our-clock 'TICK)  -> crunch crunch lasagna was yummy
(ask garfield 'MOOD)   -> angry
(ask our-clock 'TICK)  -> crunch crunch lasagna was yummy
(ask our-clock 'TICK)  -> crunch crunch lasagna was yummy
(ask garfield 'MOOD)   -> content
```

2. Add a method to the `cat` class called `'GREET`. If the cat is content, the cat should (display) "purr", otherwise "hiss".

```
'GREET (lambda (x) (display-message
                     (list "hello" (ask x 'NAME)
                           (if (eq? (ask self 'MOOD) 'content)
                               "purr" "hiss"))))
```

3. Write a class definition for a `dog`, and then create an instance of a dog named `odie`. Dogs should respond to the same methods as cats, but with opposite effects – Odie should grow more unhappy as time progresses, rather than being happier the longer he's left alone, and any attention (such as being asked to `'FETCH`) should improve his mood.