

## Homework 17: Building Objects

Assigned: Tuesday, May 8, 2007

Due: Friday, May 11, 2007

*This homework is optional!*

**Summary:** In this assignment, you will explore issues of metaprogramming as you extend a procedure that creates procedures that create objects.

**Purpose:** To help you think more about metaprogramming and objects.

**Expected Time:** One to two hours.

**Collaboration:** You may work in a group of any size between one and four, inclusive. You may consult others outside your group, provided you cite those others. You need only submit one assignment per group.

**Submitting:** Email me your work, using a subject of *CSC151 Homework 17*.

**Warning:** So that this exercise is a learning assignment for everyone, I may spend class time publicly critiquing your work.

## Background: Objects

As you may recall, the structure of a procedure that creates objects is fairly uniform. In general, it looks something like the following:

```
(define make-object
  (lambda ()
    (let ((field1 (vector init1))
          (field2 (vector init2))
          ...
          (fieldn (vector initn)))
      (lambda (message . params)
        (cond
          ((eq? message ':->string)
           code-for-making-a-string)
          ((eq? message ':store)
           code-for-writing-to-a-file)
          ((eq? message ':restore)
           code-for-restoring-from-a-file)
          ((eq? message some-message)
           instructions-for-supporting-message)
          ...
          (else
           (error "object" "unsupported message '" message "'"))))))))
```

## Background: Metaprogramming

As you may recall from our discussion of metaprogramming, when we find that we write very similar code in a variety of situations, we have two primary options: We can rely on higher-order techniques, and write a procedure that encapsulates the common code; or we can rely on metaprogramming, and write a procedure that generates the common code. When the similarities are in terms of how we write the code, rather than in the absolute structure, we are more likely to use metaprogramming. For example, we have different numbers of messages in the body of objects, so it is harder to write a higher-order procedure to unify the construction of objects.

In class, we started to write a procedure to generate procedures that make objects. Here is an extended version of what we wrote:

```
;;; Procedure:
;;; make-maker
;;; Parameters:
;;; name, a symbol that names the class of object we are making
;;; fields, a list of two-element lists
;;; methodl ... methodn, 0 or more methods of the form (NAME (PARAMS) BODY)
;;; Purpose:
;;; Creates the code for a procedure that can make objects.
;;; Produces:
;;; code-for-maker, a list that can be treated as Scheme code.
;;; Preconditions:
;;; Each element of fields is a two element list, the car of which is a symbol and the
;;; cadr of which is a value.
;;; Each method is a list of length two or more. The car of the method is a symbol. The
;;; cadr of the method is a list of symbols. The remainder of the method is a list of
;;; Scheme code that could serve as the body of a procedure. That code may use the
;;; names introduced in fields and in the parameters to this method.
;;; Postconditions:
;;; code-for-maker is appropriate code for an object maker that makes objects with
;;; the given fields and methods.
(define make-maker
  (lambda (name fields . methods)
    (list 'define (string->symbol (string-append "make-" (symbol->string name)))
          (list 'lambda ()
                (list 'let (list-fields fields)
                      (list 'lambda (cons 'method 'params)
                            (append
                             (list 'cond
                                   (list (list 'eq? 'method (quote ':save))
                                           (save-method fields))
                                   (list (list 'eq? 'method (quote ':restore!))
                                           (restore-method fields)))
                             (select-methods name methods))))))))))

;;; Procedure:
;;; list-fields
;;; Parameters:
;;; fields, a list of two-element lists
;;; Purpose:
;;; Make a list of fields of the appropriate form.
;;; Produces:
;;; field-code, a list that represents Scheme code to initialize fields as vectors.
;;; Preconditions:
;;; Each element of fields is a two element list, the car of which is a symbol and the
;;; cadr of which is a value.
;;; Postconditions:
;;; field-code could serve as the declarations portion of a let clause.
(define list-fields
  (lambda (fields)
    (if (null? fields)
```

```

    null
    (let ((field (car fields)))
      (cons (list (car field) (list 'vector (cadr field)))
            (list-fields (cdr fields))))))

;;; Procedure:
;;; save-method
;;; Parameters:
;;; fields, a list of two-element lists
;;; Purpose:
;;; Make the code to save an object to a file.
;;; Produces:
;;; save-code, a list that represents Scheme code to save the fields to a file.
;;; Preconditions:
;;; Each element of fields is a two element list, the car of which is a symbol and the
;;; cadr of which is a value.
;;; Postconditions:
;;; save-code can serve as the code for the save method of an object with the given fields.
(define save-method
  (lambda (fields)
    (list 'begin (list 'display "Save is not yet implemented") (list 'newline))))

;;; Procedure:
;;; restore-method
;;; Parameters:
;;; fields, a list of two-element lists
;;; Purpose:
;;; Make the code to restore an object from a file.
;;; Produces:
;;; restore-code, a list that represents Scheme code to save the fields to a file.
;;; Preconditions:
;;; Each element of fields is a two element list, the car of which is a symbol and the
;;; cadr of which is a value.
;;; Postconditions:
;;; restore-code can serve as the code for the save method of an object with the given fields.
(define restore-method
  (lambda (fields)
    (list 'begin (list 'display "Restore is not yet implemented") (list 'newline))))

;;; Procedure:
;;; select-methods
;;; Parameters:
;;; name, a symbol
;;; methods, a list of method
;;; Purpose:
;;; Build the section of an object that selects a method based on the symbol given.
;;; Produces:
;;; methods-code, a list that represents Scheme code.
;;; Preconditions:
;;; methods have the form specified in make-maker
;;; Postconditions:
;;; methods-code can be plugged into a cond, and has the result of
;;; selecting between methods based on the name of the procedure.
(define select-methods
  (lambda (name methods)
    (letrec ((make-args
              (lambda (args params)
                (if (null? args)
                    null
                    (cons (list (car args) (list 'car params))
                          (make-args (cdr args) (list 'cdr params))))))
      (select-method
       (lambda (name method)
         (let ((method-name (car method))
               (args (cadr method))
               (body (caddr method)))
           (list (list 'eq? 'method (list 'quote method-name))
                 (list 'if (list 'not (list '= (list 'length 'params) (length args)))
                       (list 'error (string-append (symbol->string name) (symbol->string method-name))
                             "invalid number of parameters"))))))))

```

```

      (cons 'let
            (cons (make-args args 'params)
                  body))))))
(kernel (lambda (remaining-methods)
          (if (null? remaining-methods)
              (list (list 'else
                          (list 'error (symbol->string name) "invalid method" 'method)))
              (cons (select-method name (car remaining-methods))
                    (kernel (cdr remaining-methods))))))
(kernel methods)))

```

Here is an example of how we might use `make-maker` to build a simple two-light switch.

```

> (make-maker 'twolight '((red #f) (blue #f))
  '(:->string () (string-append "<twolight>("
                                "red:" (if (vector-ref red 0) "on" "off")
                                ","
                                "blue:" (if (vector-ref blue 0) "on" "off")
                                "))))
  '(:switch-red! () (vector-set! red 0 (not (vector-ref red 0))))
  '(:switch-blue! () (vector-set! blue 0 (not (vector-ref blue 0)))))
(define make-twolight
  (lambda ()
    (let ((red (vector #f)) (blue (vector #f)))
      (lambda (method . params)
        (cond
         ((eq? method ':save) (begin (display "Save is not yet implemented") (newline)))
         ((eq? method ':restore!) (begin (display "Restore is not yet implemented") (newline)))
         ((eq? method ':->string)
          (if (not (= (length params) 0))
              (error "twolight:->string" "invalid number of parameters")
              (let ()
                (string-append
                 "<twolight>("
                 "red:"
                 (if (vector-ref red 0) "on" "off")
                 ","
                 "blue:"
                 (if (vector-ref blue 0) "on" "off")
                 "))))))
         ((eq? method ':switch-red!)
          (if (not (= (length params) 0))
              (error "twolight:switch-red!" "invalid number of parameters")
              (let () (vector-set! red 0 (not (vector-ref red 0))))))
         ((eq? method ':switch-blue!)
          (if (not (= (length params) 0))
              (error "twolight:switch-blue!" "invalid number of parameters")
              (let () (vector-set! blue 0 (not (vector-ref blue 0))))))
         (else (error "twolight" "invalid method" method))))))
  )
> cut-and-paste previous code
> (define rb (make-twolight))
> (rb ':->string)
"<twolight>(red:off,blue:off)"
> (rb ':switch-red!)
> (rb ':->string)
"<twolight>(red:on,blue:off)"
> (rb ':switch-red! 'off)
twolight:switch-red! "invalid number of parameters"
> (rb ':switch-red!)
> (rb ':switch-blue!)
> (rb ':->string)
"<twolight>(red:off,blue:on)"

```

Here's another example, one in which we build a simple grade that we can increment and scale. (Note that increment and scale both take parameters.)

```
> (make-maker 'grade '((grade 0))
  '(:->string () (string-append "<grade>(" (number->string (vector-ref grade 0)) ")"))
  '(:A! () (vector-set! grade 0 95))
  '(:B! () (vector-set! grade 0 85))
  '(:C! () (vector-set! grade 0 75))
  '(:F! () (vector-set! grade 0 60))
  '(:scale! (amt) (vector-set! grade 0 (round (* amt (vector-ref grade 0)))))
  '(:increment! (amt) (vector-set! grade 0 (+ amt (vector-ref grade 0))))
  '(:->letter () (vector-ref (vector "F" "F" "F" "F" "F" "F" "F" "C" "B" "A" "A")
                             (quotient (vector-ref grade 0) 10))))

(define make-grade
  (lambda ()
    (let ((grade (vector 0)))
      (lambda (method . params)
        (cond
         ((eq? method 'save) (begin (display "Save is not yet implemented") (newline)))
         ((eq? method 'restore!) (begin (display "Restore is not yet implemented") (newline)))
         ((eq? method ':->string)
          (if (not (= (length params) 0))
              (error "grade:->string" "invalid number of parameters")
              (let () (string-append "<grade>(" (number->string (vector-ref grade 0)) ")")))))
         ((eq? method 'A!)
          (if (not (= (length params) 0))
              (error "grade:A!" "invalid number of parameters")
              (let () (vector-set! grade 0 95))))
         ((eq? method 'B!)
          (if (not (= (length params) 0))
              (error "grade:B!" "invalid number of parameters")
              (let () (vector-set! grade 0 85))))
         ((eq? method 'C!)
          (if (not (= (length params) 0))
              (error "grade:C!" "invalid number of parameters")
              (let () (vector-set! grade 0 75))))
         ((eq? method 'F!)
          (if (not (= (length params) 0))
              (error "grade:F!" "invalid number of parameters")
              (let () (vector-set! grade 0 60))))
         ((eq? method 'scale!)
          (if (not (= (length params) 1))
              (error "grade:scale!" "invalid number of parameters")
              (let ((amt (car params)))
                (vector-set! grade 0 (round (* amt (vector-ref grade 0)))))))
         ((eq? method 'increment!)
          (if (not (= (length params) 1))
              (error "grade:increment!" "invalid number of parameters")
              (let ((amt (car params)))
                (vector-set! grade 0 (+ amt (vector-ref grade 0))))))
         ((eq? method ':->letter)
          (if (not (= (length params) 0))
              (error "grade:->letter" "invalid number of parameters")
              (let ()
                 (vector-ref
                  (vector "F" "F" "F" "F" "F" "F" "F" "C" "B" "A" "A")
                  (quotient (vector-ref grade 0) 10))))))
          (else (error "grade" "invalid method" method)))))))

> cut-and-paste of definition of make-grade
> (define exam1 (make-grade))
> (exam1 'B!)
> (exam1 ':->string)
"<grade>(85)"
> (exam1 ':->letter)
```

```

"B"
> (exam1 'scale 105/100)
grade "invalid method" :scale
> (exam1 'scale!)
grade:scale! "invalid number of parameters"
> (exam1 'scale! 105/100)
> (exam1 ':->letter)
"B"
> (exam1 ':->string)
"<grade>(89)"
> (exam1 'scale! 105/100)
> (exam1 ':->letter)
"A"
> (exam1 'increment -20)
grade "invalid method" :increment
> (exam1 'increment! -20)
> (exam1 ':->letter)
"C"
> (exam1 ':->string)
"<grade>(73)"

```

## Assignment

As you may note, the procedures that generate the code for saving an object to a file and restoring an object to a file are not yet written. Here's what happens when we try to save and restore, using the last object created above

```

> (exam1 'save "sams-exam")
Save is not yet implemented
> (exam1 'restore "sams-exam")
grade "invalid method" :restore
> (exam1 'restore! "sams-exam")
Restore is not yet implemented

```

**Write `save-method` and `restore-method` so that they generate appropriate code to save and restore an object.**

---

Copyright © 2007 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.