MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1998

(Non)-Problem Set 9

- Issued: Tuesday, April 7

Quiz II announcement

Quiz 2 is on Tuesday, April 14. The quiz will be held in Walker Memorial Gymnasium (50-340) from 5–7PM xor 7–9PM. You may take the quiz during either one of these two periods, but students taking the quiz during the first period will not be allowed to leave the room until the end of the period. Optional review sessions for the quiz will be held on Sunday night, April 12, from 7-10PM in rooms 34-310, 302, 303 and 304.

The quiz is open book. The quiz will cover material presented from the beginning of the semester through recitation on April 3, the textbook through Chapter 3, and the general material covered in the problem sets through problem set 8.

The following material was found near the printer in the 6.001 lab. It appears to have something to do with quiz II.

Hey Alyssa!

So – I've got a great idea for our submission to the 50K Entrepreneurship Competition. Let's build a tax preparation system – you know, something that computer dummies could use to fill out their forms for the IRS. We could have "smart forms". You know, when you fill in a slot on the form with some figures, the forms know what other slots have to change, and do so automatically. This way, all the numbers stay consistent, and you don't have to explicitly do the work to make sure. I figure we could do this with some kind of data abstractions, and some smart procedures to handle the tax forms and their slots.

Here's a quick version of what I have in mind. Let me know what you think!

Ben

```
;; we'll use an abstraction called an entry to represent each slot in
;; a form, and we'll use another abstraction called a form to collect
;; all the entries together

;; for example
(define make-form list)
(define next-entry car)
(define rest-entries cdr)
(define empty-form? null?)

(define make-tagged-slot list)
(define slot-name car)
(define slot-value cadr)

(define (make-entry name value updates)
  (list
   (make-tagged-slot 'name name)
   (make-tagged-slot 'value value)
   (make-tagged-slot 'updates updates)))

(define entry-name car)
(define entry-value cadr)
(define entry-updates caddr)

(define (update-form form key op value)
  (define (loop fm key op value)
    (cond ((empty-form? fm) '())
          ((eq? key (slot-value (entry-name (next-entry fm))))
           (change-value (entry-value (next-entry fm))
                         op
                         value)
           (map (lambda (update) (update form))
                (slot-value
                 (entry-updates (next-entry fm)))))
          (else
           (loop (rest-entries fm) key op value))))
  (loop form key op value))

(define (lookup entry-key slot-key form)
  (lookup-slot slot-key (lookup-entry entry-key form)))
```

```
(define (income-value form)
  (lookup 'income 'value form))

(define (deductions-value form)
  (lookup 'deductions 'value form))

(define (taxes-paid-value form)
  (lookup 'taxes-paid 'value form))

(define (taxes-owed-value form)
  (lookup 'taxes-owed 'value form))

(define (refund-value form)
  (lookup 'refund 'value form))

;; temporary trial

(define income-1040
  (make-entry 'income 0 (list exp1)))

(define deductions-1040
  (make-entry 'deductions 0 (list exp2)))

(define taxes-paid-1040
  (make-entry 'taxes-paid 1000 (list exp3)))

(define taxes-owed-1040
  (make-entry 'taxes-owed 0 (list exp4)))

(define refund-1040
  (make-entry 'refund 0 '()))

(define trial-1040
  (make-form income-1040 deductions-1040 taxes-paid-1040 taxes-owed-1040 refund-1040))

;; I've got a bunch of other code -- abstractions, interfaces -- all
;; that stuff.  Here's an example of the system running:

1 ]=> (taxes-owed trial-1040)
;Value: 0

1 ]=> (update-form trial-1040 'income + 10000)
;Value: changed

1 ]=> (income trial-1040)
;Value: 10000

1 ]=> (taxes-owed trial-1040)
;Value: 2500.

1 ]=> (refund trial-1040)
;Value: -1500.
```

Hi Ben,

I got your note – frankly, you blew it. You're describing a spread sheet system. Lotus has a ton of patents on that stuff, and we'll be infringing if we use your version. Here's a much better way – use Object Oriented Programming.

– just treat each slot in a form as an object with some state

– treat each form as another kind of object with some state

– send messages between the forms and the slots to keep the data consistent

Here's a rough sketch of how we could do this:

```
(define (create-1040-form paid)
  (define income (create-income))
  (define deductions (create-deductions))
  (define taxes-owed (create-taxes-owed))
  (define taxes-paid (create-taxes-paid paid))
  (define refund (create-refund))
  (ask refund 'recompute taxes-paid taxes-owed)
  (lambda (message)
    (case message
      ((CHANGE-INCOME)
       (lambda (self new)
         (ask income 'add new)
         (ask taxes-owed 'recompute income deductions)
         (ask refund 'recompute taxes-paid taxes-owed)))
      ((CHANGE-DEDUCTIONS)
       (lambda (self new)
         (ask deductions 'add new)
         (ask taxes-owed 'recompute income deductions)
         (ask refund 'recompute taxes-paid taxes-owed)))
      ((INCOME)
       (lambda (self)
         (ask income 'value)))
      ((DEDUCTIONS)
       (lambda (self)
         (ask deductions 'value)))
      ((TAXES-OWED)
       (lambda (self)
         (ask taxes-owed 'value)))
      ((TAXES-PAID)
       (lambda (self)
         (ask taxes-paid 'value)))
      ((REFUND)
       (lambda (self)
         (ask refund 'value)))
      (else (no-method)))))
```

and here are some templates for the other pieces:

```
(define (create-income)
  (let ((value 0))
    (lambda (message)
      (case message
         ;; need a bunch of methods here, like
        ((VALUE) (lambda (self) value))
         ;; and some other special ones here, then end with
        (else (no-method))))))
```

This will work a lot better, and we avoid legal difficulties as well. Note, by the way, that we can easily make this more efficient by using streams, or by using concurrent processing.

Alyssa