CS61A – Homework 3.2	**Kurt Meinz**
University of California, Berkeley	Summer 2003

Topic: Representing abstract data

Lectures: Wednesday July 10, Thursday July 11

Reading: Abelson & Sussman, Sections 2.4 through 2.5.2 (Pages 169–200)

This assignment gives you practice with data-directed programming and message-passing. Part of the assignment involves understanding and modifying the generic arithmetic system described in the book.

The file `~cs61a/lib/packages.scm` contains the code from the book that implements generic arithmetic. It has the definitions of `install-rectangular-package`, `install-polar-package`, `install-scheme-number-package`, `install-rational-package` and `install-complex-package`. But remember, these are just procedure definitions. **You have to invoke them to populate the table!** For convenience, we've provided the function `install-all-packages` which is defined as:

```
(define (install-all-packages)
  (install-polar-package)
  (install-rectangular-package)
  (install-complex-package)
  (install-scheme-number-package)
  (install-rational-package)
  'engage-warp-9)
```

The file also contains `apply-generic` from Page 184, the generic procedures from Pages 184 and 189, the relevant constructors and other supporting code.

This assignment is due at **8 PM on Sunday, July 13**. Put your answers into a file called `hw3-2.scm` and turn it in electronically. Comment out your test cases so the file loads smoothly.

Question 1. Write a function `add-up-complex` that takes a list of complex numbers (in polar or rectangular form) and returns a complex number representing their sum. The result should be in rectangular form.

```
STk> (define x (make-complex-from-real-imag 3 4))
x
STk> (define y (make-complex-from-real-imag 10 0))
y
STk> (define z (make-complex-from-mag-ang 5 1.2))
z
STk> (add-up-complex (list x y z))
(complex rectangular 14.8117887723834 . 8.66019542983613)
STk> (add-up-complex '())
(complex rectangular 0 . 0)
```

The adventure continues on the next page.

Question 2. Read and complete Exercise 2.77 on Page 192. You might want to `trace` the `apply-generic` procedure.

In case this is not clear, when Louis types

```
(put 'magnitude '(complex) magnitude)
```

the `magnitude` procedure actually inserted into the table is the one defined on Page 184. The definitions of `real-part`, `imag-part` and `angle` are there, too.

A good place to start is by reproducing Louis' error:

```
STk> (load "~cs61a/lib/packages.scm")
okay
STk> (install-all-packages)
engage-warp-9
STk> (define z (make-complex-from-real-imag 3 4))
z
STk> (magnitude z)
*** Error:
    No method for these types -- APPLY-GENERIC (magnitude (complex))
```

Question 3. We'd like to create a generic procedure `zero?` that tests if its argument is equal to zero. We're going to use `apply-generic` to define it:

```
(define (zero? x) (apply-generic 'zero? x))
```

Your job is to add something to the `complex`, `rational` and `scheme-number` packages to make this generic definition work. Here is the desired behavior:

```
STk> (zero? (make-rational 1 2))
#f
STk> (zero? (make-complex-from-real-imag 0 0))
#t
STk> (zero? (make-complex-from-mag-ang 0 1.4))   ;; zero magnitude
#t
STk> (zero? (make-scheme-number 43))
#f
STk> (zero? (make-rational 0 234))
#t
STk> (zero? (make-scheme-number 0))
#t
```

Show just the parts you added.

The learning continues on the next page.

Question 4. Berkeley is a great place to buy coffee. So many vendors to choose from: Starbucks, Tullys, Peets, Strada, etc. Some of these places offer a bulk discount: the more coffee you buy the less it costs. We'll model the pricing scheme of a given coffee vendor with a function that takes the quantity of coffee you'd like to purchase and returns the total price. Suppose we've set up a table keyed by vendor name and coffee type like this:

```
STk> (put 'starbucks 'frap (lambda (n) (* n 3.50)))
STk> (put 'starbucks 'mocha (lambda (n) (* n 1.50)))
STk> (put 'coffee-source 'frap (lambda (n)
                                 (cond ((< n 5) (* n 4.00))
                                       ((< n 10) (* n 3.00))
                                       (else (* n 2.50)))))
STk> (put 'tullys 'frap (lambda (n)
                          (cond ((< n 10) (* n 3.50))
                                ((< n 30) (* n 3.00))
                                (else (* n 2.50)))))
STk> (put 'peets 'frap (lambda (n)
                          (if (< n 50)
                              (* n 4.00)
                              (* n 2.00))))
```

To find out how much ten Starbucks fraps cost you'd type:

```
STk> ((get 'starbucks 'frap) 10)
35.0
```

Write a function `best-deal` that takes three arguments: the type of coffee, the quantity you want to purchase and a list of vendors **at least one of which sells the desired item**. It should return the *name* of the vendor with the best price for that quantity of goods. If multiple vendors exists with the same low price `best-deal` should return the first one in the list.

```
STk> (best-deal 'frap 1 '(starbucks tullys office-depot))
starbucks
STk> (best-deal 'frap 10000 '(starbucks walmart peets tullys strada))
peets
STk> (best-deal 'frap 13 '(coffee-source tullys))
coffee-source
STk> (best-deal 'mocha 87 '(peets starbucks coffee-source))
starbucks
```

As you can see, not all the vendors will sell the product desired. Some of the vendors might not even be in the table! At least one will. Recall that `get` returns #f if it does not find anything in the table matching *both* keys.

You may want to use the following helper function, which returns the first vendor in a list of vendors that sells a specific good.

```
(define (vendor-that-sells good vendors)
  (if (get (car vendors) good)
      (car vendors)
      (vendor-that-sells good (cdr vendors))))

STk> (vendor-that-sells 'mocha '(copy-central peets starbucks))
starbucks
```

The excitement continues on the next page.

Question 5. In the last homework, you implemented a Mobile ADT and wrote functions `total-weight` and `balanced?` that worked on Mobiles. Here is the Mobile constructor:

```
(define (make-mobile left-branch right-branch)
    (list 'mobile left-branch right-branch))
```

We'd now like to implement Mobiles as *message-passing objects*, similar to `make-from-real-imag` on Page 186. Here is the new Mobile constructor:

```
(define (make-mobile left-branch right-branch)
   (define (dispatch op)
      (cond ((eq? op 'left-branch) left-branch)
            ((eq? op 'right-branch) right-branch)
            (else (error "I don't understand -- MAKE-MOBILE: " op))))
   dispatch)
```

Implement `make-branch`, the constructor for branches, in message-passing style. Then write the four selectors `left-branch`, `right-branch`, `branch-structure` and `branch-length` to work with this implementation of Mobiles and branches. Test them on `mobile-3`, defined in the last homework. Your `total-weight` and `balanced?` functions should work **without modification** with this new representation of Mobiles.