CS61A – Homework 4.1	**Kurt Meinz**
University of California, Berkeley	Summer 2003

Topic: Object-oriented programming

Lectures: Monday July 14, Tuesday July 15

Reading: "Object-Oriented Programming—Above-the-line view" (in course reader)

This homework gives your practice with our OOP system for Scheme. To use it, you must load `~cs61a/lib/obj.scm`. The assignment is due at **8 PM Sunday, July 20**. Put your solutions into a file `hw4-1.scm`, yadda, yadda, yadda ... you know the drill.

Question 1. Create a class called `random-generator` that takes one instantiation argument, a number n. An instance of this class should respond to the message `new` by returning a random number that is less than n. (Recall that `(random 10)` returns a random number between 0 and 9.) Any other message should cause the instance to spit out the number it returned the last time:

```
STk> (define rand1 (instantiate random-generator 10))
rand1
STk> (ask rand1 'new)
4
STk> (ask rand1 'new)
9
STk> (ask rand1 'foo)
9
STk> (ask rand1 'bar)
9
STk> (ask rand1 'baz)
9
```

If a newly instantiated `random-generator` is given a message that is not `new`, it may return anything.

Question 2. Create a `coke-machine` class. Instances of this class have one instantiation variable, the price (in cents) of a coke, and respond to five messages:

- `num-cokes` — Returns the number of cokes currently in the machine. Initially, zero.
- `fill` n — Fills the machine with n cokes. Machines start out empty. Returns anything.
- `price` — Returns the price of a coke.
- `deposit` n — Deposits n cents into the machine toward the purchase of a coke. You can deposit several coins and the machine should remember the total. Return value is up to you.
- `coke` — Returns the string `"Machine empty"`, the string `"Not enough money")` or your change, which signifies the successful purchase of a beverage. Decreases the number of cokes in the machine by one and clears the money in the machine.

Here's an example:

```
STk> (define my-machine (instantiate coke-machine 70))
STk> (ask my-machine 'num-cokes)
0
```

The question continues on the next page.

```
STk> (ask my-machine 'coke)
"Machine empty"
STk> (ask my-machine 'fill 60)              ;; return value up to you
STk> (ask my-machine 'deposit 25)           ;; return value up to you
STk> (ask my-machine 'coke)
"Not enough money"
STk> (ask my-machine 'deposit 25)           ;; now there's 50 cents in there
STk> (ask my-machine 'deposit 25)           ;; now there's 75 cents
STk> (ask my-machine 'coke)
5                                           ;; 5 cents change
STk> (ask my-machine 'num-cokes)
59
```

You may assume that the machine has an infinite supply of change and infinite space to store cokes.

Question 3. The OOP construct `usual` forwards a message to the parent class, up exactly one level in the inheritance hierarchy. Extend this capability by writing a *method* called `n-usual` that sends a message to the nth ancestor in the inheritance hierarchy. This feature need only work with single inheritance. The method will take two arguments: n and a message. If n is zero, the message should be given to `self`. In order for this to work, each class in the hierarchy must have the same `n-usual` method. Here is the desired behavior (with return values omitted for clarity):

```
STk> (define-class (a)
        (method (foo) (display "Foo in A") (newline))
        (method (n-usual n message) ... ))
STk> (define-class (b)
        (parent (a))
        (method (foo) (display "Foo in B") (newline))
        (method (n-usual n message) ... ))
STk> (define (c)
        (parent (b))
        (method (foo) (display "Foo in C") (newline))
        (method (n-usual n message) ... ))
STk> (define a1 (instantiate a))
STk> (define b1 (instantiate b))
STk> (define c1 (instantiate c))
STk> (ask c1 'n-usual 0 'foo)
Foo in C
STk> (ask c1 'n-usual 1 'foo)
Foo in B
STk> (ask c2 'n-usual 2 'foo)
Foo in A
```

Assume the nth ancestor can handle the message, and that the message takes no arguments. This problem is trickier than it looks. You'll need more than one base case.

The assignment continues on the next page.

Question 4. This exercise is mindblowingly cool. We can use OOP to represent `cons` pairs, and out of these OOP pairs we can make lists! For simplicity, assume throughout this exercise that our OOP lists will contain only *atomic* data, such as words and numbers. We'll need two classes, `oop-pair` and `the-null-list`:

```
(define-class (oop-pair the-car the-cdr)
       (method (length)
          (+ 1 (ask the-cdr 'length)))
       (method (list-ref n)
          (if (= n 0)
              the-car
              (ask the-cdr 'list-ref (- n 1)))))

(define-class (the-null-list)
       (method (length) 0)
       (method (list-ref n)
          (error "Can't LIST-REF into null list")))
```

Just like a proper list made of primitive `cons` pairs must end in `nil`, a proper OOP list must end in an instance of `the-null-list` class. Here is how you can use these definitions to construct the list `(a b c)`:

```
STk> (define my-oop-list (instantiate oop-pair 'a
                             (instantiate oop-pair 'b
                                (instantiate oop-pair 'c
                                    (instantiate the-null-list)))))
my-oop-list
STk> my-oop-list
#[closure arglist=(message) 32ddec]   ;; it's an object!
STk> (ask my-oop-list 'length)
3
STk> (ask my-oop-list 'list-ref 2)
c
```

Pause here to make sure you understand how this works.

A. It's not very convenient to construct these OOP lists as above. Define a procedure `regular->oop-list` that takes a regular Scheme list and returns the equivalent OOP list:

```
STk> (define oop-list-1 (regular->oop-list '(holy cow)))
oop-list-1
STk> oop-list-1
#[closure arglist=(message) d2d88c]       ;; it's an object!
STk> (ask oop-list-1 'length)
2
STk> (ask oop-list-1 'list-ref 0)
holy
```

The assignment continues on the next page.

B. It's also not very convenient to view the contents of an OOP list. Add a `print` method to the `oop-pair` and `the-null-list` classes that has this behavior:

```
STk> (define oop-list-2 (regular->oop-list '(2 soon 2 tell)))
oop-list-2
STk> (ask oop-list-2 'print)
[2 soon 2 tell ]
okay                                              ;; return value up to you
STk> (ask (instantiate the-null-list) 'print)
[]
okay
```

Use the `display` procedure to print the elements of the list. The return value of the `print` method is up to you. We only care about its side-effect. Don't worry about extra spaces in the output.

C. Lastly, add a `member?` method to the two class definitions:

```
STk> (define oop-list-3 (regular->oop-list '(a prison for your mind)))
oop-list-3
STk> (ask oop-list-3 'member? 'prison)
#t
STk> (ask oop-list-3 'member? 'jail)
#f
```