CS 61A, Summer 2006 June 28, 2006

HOMEWORK ASSIGNMENT 1B
DUE WEDNESDAY JULY 5, 2006 AT 11:59 PM

You should probably read section 1.3 of the textbook before (or while) you do this homework assignment. Note that we are skipping 1.2; we'll get to it later. Because of this, never mind for now the stuff about iterative versus recursive processes in 1.3 and in the exercises from that section.

Don't panic if you have trouble with the half-interval example on pp. 67–68; you can just skip it. Try to read and understand everything else.

(All numbered exercises are from *Structure and Interpretation of Computer Programs*, 2nd edition.)

1. Exercise 1.31(a), 1.32(a), 1.33, 1.40, 1.41, 1.42, 1.43. Optional exercise: 1.46.

2. In lecture we have seen `every`, which takes a procedure and a sentence and applies the procedure to every word of that sentence, e.g.

```
> (every square '(1 2 3 4))
(1 4 9 16)
> (every pigl '(dogs and cats))
(ogsday anday atscay)
```

Now write a procedure `make-every` that takes a procedure and *returns another procedure* that takes a sentence as argument and applies the procedure to every word of the sentence, e.g.

```
> (make-every square)
#[closure ...]
> ((make-every square) '(1 2 3 4))
(1 4 9 16)
> ((make-every pigl) '(dogs and cats))
(ogsday anday atscay)
```

Optional Extra For Experts exercise on the next page ...

Extra For Experts. This is a **totally optional exercise**, worth no course credit whatsoever, intended only for people who have finished the assignment and are looking for something fun and/or intellectually challenging to do. You will never be expected to know anything about "extra for experts" problems in this course.

In principle, we could build a version of Scheme with no primitives except `lambda`. Everything else can be defined in terms of `lambda`, although it's not done that way in practice because it would be so painful. But we can get a sense of the flavor of such a language by eliminating one feature at a time from Scheme to see how to work around it.

In this problem we explore a Scheme without `define`. We can give things names by using argument binding, as `let` does, so instead of

```
(define (sumsq a b)
  (define (square x) (* x x))
  (+ (square a) (square b)))

(sumsq 3 4)
```

we can say

```
((lambda (a b)
   ((lambda (square)
      (+ (square a) (square b)))
    (lambda (x) (* x x))))
 3 4)
```

This works fine as long as we don't want to use *recursive* procedures. But we can't replace

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))

(fact 5)
```

by

```
((lambda (n)
   (if ...))
 5)
```

because what do we do about the invocation of `fact` inside the body?

Your task is to find a way to express the `fact` procedure in a Scheme without any way to define global names.