CS 61A Week 2

Topic: Higher-order procedures

Reading: Abelson & Sussman, Section 1.3

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.

Homework:

1. Abelson & Sussman, exercises 1.31(a), 1.32(a), 1.33, 1.40, 1.41, 1.43, 1.46

(Pay attention to footnote 51; you'll need to know the ideas in these exercises later in the semester.)

2. Last week you wrote procedures `squares`, that squared each number in its argument sentence, and saw `pigl-sent`, that `pigl`ed each word in its argument sentence. Generalize this pattern to create a higher-order procedure called `every` that applies an *arbitrary* procedure, given as an argument, to each word of an argument sentence. This procedure is used as follows:

```
> (every square '(1 2 3 4))
(1 4 9 16)
> (every first '(nowhere man))
(n m)
```

3. Our Scheme library provides versions of the `every` function from the last exercise and the `keep` function shown in lecture. Get familiar with these by trying examples such as the following:

```
(every (lambda (letter) (word letter letter)) 'purple)
(every (lambda (number) (if (even? number) (word number number) number))
       '(781 5 76 909 24))
(keep even? '(781 5 76 909 24))
(keep (lambda (letter) (member? letter 'aeiou)) 'bookkeeper)
(keep (lambda (letter) (member? letter 'aeiou)) 'syzygy)
(keep (lambda (letter) (member? letter 'aeiou)) '(purple syzygy))
(keep (lambda (wd) (member? 'e wd)) '(purple syzygy))
```

Continued on next page.

Week 2 continued...

Extra for experts:

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.

Unix feature of the week: `pine`, `mail`, `firefox`

Emacs feature of the week: `M-x info`, `C-x u` (undo)