CS 61A Week 1

Topic: Functional programming

Reading: Abelson & Sussman, Section 1.1 (pages 1–31)

Note: With the obvious exception of this first week, you should do each week's reading *before* the Monday lecture. So also start now on next week's reading, Abelson & Sussman, Section 1.3

Homework:

People who've taken CS 3: Don't use the CS 3 higher-order procedures such as `every` in these problems; use recursion.

1. Do exercise 1.6, page 25. This is an essay question; you needn't hand in any computer printout, unless you think the grader can't read your handwriting. If you had trouble understanding the square root program in the book, explain instead what will happen if you use `new-if` instead of `if` in the `pigl` Pig Latin procedure.

2. Write a procedure `squares` that takes a sentence of numbers as its argument and returns a sentence of the squares of the numbers:

```
> (squares '(2 3 4 5))
(4 9 16 25)
```

3. Write a procedure `switch` that takes a sentence as its argument and returns a sentence in which every instance of the words `I` or `me` is replaced by `you`, while every instance of `you` is replaced by `me` except at the beginning of the sentence, where it's replaced by `I`. (Don't worry about capitalization of letters.) Example:

```
> (switch '(You told me that I should wake you up))
(i told you that you should wake me up)
```

4. Write a predicate `ordered?` that takes a sentence of numbers as its argument and returns a true value if the numbers are in ascending order, or a false value otherwise.

5. Write a procedure `ends-e` that takes a sentence as its argument and returns a sentence containing only those words of the argument whose last letter is E:

```
> (ends-e '(please put the salami above the blue elephant))
(please the above the blue)
```

Continued on next page.

Week 1 continued...

6. Most versions of Lisp provide `and` and `or` procedures like the ones on page 19. In principle there is no reason why these can't be ordinary procedures, but some versions of Lisp make them special forms. Suppose, for example, we evaluate

```
(or (= x 0) (= y 0) (= z 0))
```

If `or` is an ordinary procedure, all three argument expressions will be evaluated before `or` is invoked. But if the variable `x` has the value `0`, we know that the entire expression has to be true regardless of the values of `y` and `z`. A Lisp interpreter in which `or` is a special form can evaluate the arguments one by one until either a true one is found or it runs out of arguments.

Your mission is to devise a test that will tell you whether Scheme's `and` and `or` are special forms or ordinary functions. This is a somewhat tricky problem, but it'll get you thinking about the evaluation process more deeply than you otherwise might.

Why might it be advantageous for an interpreter to treat `or` as a special form and evaluate its arguments one at a time? Can you think of reasons why it might be advantageous to treat `or` as an ordinary function?

Unix feature of the week: `man`

Emacs feature of the week: `C-g`, `M-x apropos`

There will be a "feature of the week" each week. These first features come first because they are the ones that you use to find out about the other ones: Each provides documentation of a Unix or Emacs feature. This week, type `man man` as a shell command to see the Unix manual page on the `man` program. Then, in Emacs, type `M-x` (that's meta-`X`, or `ESC X` if you prefer) `describe-function` followed by the Return or Enter key, then `apropos` to see how the `apropos` command works. If you want to know about a command by its keystroke form (such as `C-g`) because you don't know its long name (such as `keyboard-quit`), you can say `M-x describe-key` then `C-g`.

You aren't going to be tested on these system features, but it'll make the rest of your life a *lot* easier if you learn about them.