CS61A – Homework 2.2 | **Kurt Meinz**
University of California, Berkeley | Summer 2005

Topic: Data abstraction

Lectures: Wednesday July 2, Thursday July 3

Reading: Abelson & Sussman, Sections 2.1 and 2.2.1 (Pages 79–106)

In this assignment you'll practice working with Scheme lists. Although lists look like sentences, you should treat them as a completely separate type. Do not use sentence operators on lists! Below is a table of sentence functions and their list counterparts:

Sentences	Lists
`first butfirst`	`car cdr`
`last butlast`	`none`
`sentence`	`list append cons`
`every`	`map`
`keep`	`filter`
`member?`	`member`
`item`	`list-ref`
`empty?`	`null?`
`count`	`length`

The homework is due at **8 PM Sunday, July 3**. Please put your solutions into a file called `hw2-2.scm` and submit them online. Include test cases, but comment them out so the file loads cleanly.

Question 1. In the sentence world, `keep` can be written in terms of `every`, like this:

```
STk> (define (keep pred sent)
       (every (lambda (e) (if (pred e) e '())) sent))
STk> (keep number? '(in 1 day it will be 1999))
(1 1999)
```

Can you use a similar trick to write `filter` using `map`, which is defined on Page 105? Why or why not?

Question 2. Read and complete Exercise 2.12 in SICP. Here is the interval ADT:

```
(define (make-interval a b) (cons a b))
(define (upper-bound interval) (car interval))
(define (lower-bound interval) (cdr interval))
```

We'll represent percentages as decimal values in the range $[0, 1]$. Here is the desired behavior:

```
STk> (define my-interval (make-center-percent 10 .1))   ;; 10% tolerance
STk> (center my-interval)
10
STk> (percent my-interval)
.1
STk> (lower-bound my-interval)
9
STk> (upper-bound my-interval)
11
```

The fun continues on the next page.

Question 3. The great thing about lists is that they can hold *any* Scheme value: numbers, booleans, even procedures! Watch:

```
STk> (define procs (list + * =))    ;; why doesn't '(+ * =) work?
procs
STk> ((car procs) 10 10 10)
30
STk> ((caddr procs) 9 11)
#f
```

A. We'd like to exploit this feature by writing a function `call-all` that takes two arguments: a list of unary procedures and an arbitrary Scheme value x. It should invoke all the procedures in the list on x in the intuitive order (the rightmost procedure is the last one invoked):

```
STk> (call-all (list (lambda (x) (- x 6)) abs sqrt) -10)
4
STk> (call-all (list cdr cdr cdr null?) '(free your mind))
#t
STk> (call-all nil 'foo)   ;; identity function
foo
STk> (call-all (list list list list) 'foo)
(((foo)))
```

Write `call-all`; there is a very simple recursive solution.

B. It'd be a lot nicer if instead of taking two arguments `call-all` could take *any number* of arguments, the last of which is x, like this:

```
STk> (call-all cdr cdr cdr null? '(free your mind))
#t
STk> (call-all 'foo)   ;; identity function
foo
```

Scheme provides a special *dotted-tail notation* for definitions that allows procedures to take an arbitrary number of arguments. For example:

```
STk> (define (f x y . z) (list x y z))
```

The procedure `f` can be called with two or more arguments. The first will be in `x`; the second in `y` and any remaining arguments will be put into a list `z`:

```
STk> (f 1 2 3 4)
(1 2 (3 4))
STk> (f 1 2)
(1 2 ())
STk> (f 1)
*** Error: wrong number of arguments to procedure: (f 1)
```

As another example, the primitive operator `list` can be defined like this:

```
STk> (define (list . args) args)
list
STk> (list 1 2 'three)
(1 2 three)
```

Use dotted-tail notation to create a new version of `call-all` that behaves as above. It should accept one or more arguments.

The fun continues on the next page.

Question 4. Since lists can contain lists, it becomes possible to create *nested* lists. Next week we'll explore nested lists and other hierarchical data structures. As a prelude, write a function `group-2` that takes a single list as argument. The number of things in the list will always be a multiple of two. The function should group every two consecutive elements into a list, returning a list of lists:

```
STk> (group-2 '(a b c d e f g h))
((a b) (c d) (e f) (g h))
STk> (group-2 '(hello (mr)))
((hello (mr)))
```

First write `group-2` so it generates a recursive process. Next write a version that generates an iterative process. The iterative solution is a bit more difficult; you may find `append` useful.