CS 61A Week 9

Topic: Mutable data, vectors

Reading: Abelson & Sussman, Section 3.3.1–3

(If you are a hardware type you might enjoy reading 3.3.4 even though it isn't required.)

Homework:

Abelson & Sussman, exercises 3.16, 3.17, 3.21, 3.25, 3.27

You don't need to draw the environment diagram for exercise 3.27; use a trace to provide the requested explanations. Treat the table procedures `lookup` and `insert!` as primitive; i.e. don't trace the procedures they call. Also, assume that those procedures work in constant time. We're interested to know about the number of times `memo-fib` is invoked.

Vector questions: In all these exercises, don't use a list as an intermediate value. (That is, don't convert the vectors to lists!)

1. Write `vector-append`, which takes two vectors as arguments and returns a new vector containing the elements of both arguments, analogous to `append` for lists.

2. Write `vector-filter`, which takes a predicate function and a vector as arguments, and returns a new vector containing only those elements of the argument vector for which the predicate returns true. The new vector should be exactly big enough for the chosen elements. Compare the running time of your program to this version:

```
(define (vector-filter pred vec)
  (list->vector (filter pred (vector->list vec))))
```

3. Sorting a vector.

(a) Write `bubble-sort!`, which takes a vector of numbers and rearranges them to be in increasing order. (You'll modify the argument vector; don't create a new one.) It uses the following algorithm:

[1] Go through the array, looking at two adjacent elements at a time, starting with elements 0 and 1. If the earlier element is larger than the later element, swap them. Then look at the next overlapping pair (0 and 1, then 1 and 2, etc.).

[2] Recursively bubble-sort all but the last element (which is now the largest element).

[3] Stop when you have only one element to sort.

(b) Prove that this algorithm really does sort the vector. Hint: Prove the parenthetical claim in step [2].

(c) What is the order of growth of the running time of this algorithm?

Note: Part II of programming project 3 is also due next week.

Continued on next page.

Week 9 continued...

Extra for experts:

1. Abelson and Sussman, exercises 3.19 and 3.23.

Exercise 3.19 is incredibly hard but if you get it, you'll feel great about yourself. You'll need to look at some of the other exercises you skipped in this section.

Exercise 3.23 isn't quite so hard, but be careful about the O(1)—i.e. *constant*—time requirement.

2. Write the procedure `cxr-name`. Its argument will be a function made by composing `car`s and `cdr`s. It should return the appropriate name for that function:

```
> (cxr-name (lambda (x) (cadr (cddar (cadar x)))))
CADDDAADAR
```

Unix feature of the week: `alias`, `unalias`

Emacs feature of the week: `C-x 4` (split window)