CS61A – Homework 5.1 **Kurt Meinz**
University of California, Berkeley Summer 2005

Topic: Mutation

Lectures: Monday July 21, Tuesday July 22

Reading: Abelson & Sussman, Section 3.3.1–3

This assignment gives you practice with mutation of pairs and circular structures made of pairs. Additionally, several of problems require an understanding of last week's material. Question 4, `count-pairs`, is a classic problem in computer science. Make sure to spend enough time on it. This homework is due at **8 PM on Sunday, July 24**. Put your answers into a file `hw5-1.scm` and submit it electronically.

Question 1. This question isn't so much about how tables work, but about using them. *Memoization* is a technique for increasing the efficiency of a program by recording previously computed results in a local table. The keys are the arguments to the memoized procedure. When the memoized procedure is asked to compute a value, it first checks the table. If the value has already been computed, just pull it out of the table. Otherwise, compute the value and store it in the table for future use. (Note that memoization only benefits procedures that are strictly *functional*; it would not make sense to memoize `random` or other procedures with side-effects.)

A. Here is the familiar procedure for computing Fibonacci numbers:

```
(define (fib n)
   (cond ((= n 0) 1)
         ((= n 1) 1)
         (else (+ (fib (- n 1)) (fib (- n 2))))))
```

Its order of growth is $\Theta(2^n)$. Here is a memoized version:

```
(define memo-fib
  (let ((history (make-table)))
    (lambda (n)
       (let ((previously-computed (lookup n history)))
          (or previously-computed
              (cond ((= n 0) 1)
                    ((= n 1) 1)
                    (else
                      (let ((result (+ (memo-fib (- n 1)) (memo-fib (- n 2)))))
                        (insert! n result history)
                        result))))))))
```

Code for one-dimentional tables is in `~cs61a/lib/tables.scm`. To get a rough idea of how much work is saved, `trace` both versions and compute the 11th Fibonacci number. Explain why `memo-fib` computes the nth Fibonacci number in a number of steps proportional to n. That is, show that `memo-fib` has roughly a linear order of growth. Treat `lookup` and `insert!` as constant-time operations.

B. Memoize the `count-change` procedure defined on Page 40 of SICP. Actually, memoize its helper `cc`. Model your `memo-cc` procedure on `memo-fib`. Notice that `cc` has a structure that is very similar to `fib`: two base cases, one recursive case but with two recursive calls. The only difference is that `cc` takes two arguments, `amount` and `kinds-of-coins`. While you can use a two-dimentional table to deal with this, it is probably easier to use a one-dimentional table and `list` both arguments for the key. Test your `memo-cc` against the original `cc` procedure to make sure it returns the same answer—but faster! You'll find the original `count-change` procedure in `~cs61a/lib/change.scm`.

The adventure continues on the next page.

Question 2. In this question we look at destructive removal of elements from a proper list.

A. Write the procedure `remove-nth!` that takes a list and a number *n*. It should *destructively* remove the *n*th element of the list (counting from zero). The return value of `remove-nth!` is up to you; it's the side-effect we're after. You may assume that *n* will be within the length of the list. Additionally, you may assume that *n* will never be zero; that is, we'll never ask `remove-nth!` to get rid of the very first list element. The desired behavior is this:

```
STk> (define red-pill (list 'how 'deep 'the 'rabbit 'hole 'is))
red-pill
STk> (remove-nth! red-pill 1)
ok                                ;; return value is garbage
STk> red-pill
(how the rabbit hole is)
STk> (remove-nth! red-pill 3)
ok
STk> red-pill
(how the rabbit is)
```

B. Now the interesting part: why can't *n* be zero? Specifically, why is it **impossible** to write a `remove-nth!` function that can remove *all* the elements of a given list? For example:

```
STk> (define a (list 'hello))
a
STk> (remove-nth! a 0)
ok
STk> a
()
```

Assuming you have a working `remove-nth!` from Part A, why does the following definition of `remove-nth-with-zero!`, which attempts to handle the case when *n* is zero, fail?

```
(define (remove-nth-with-zero! lst n)
   (if (= n 0)
       (set! lst (cdr lst))
       (remove-nth! lst n)))    ;; call remove-nth! if n is nonzero
```

You may find it useful to draw an environment diagram (or have EnvDraw draw it for you).

Question 3. Write a function `interleave!` that takes two lists, the first of which is non-empty, and interleaves their elements using mutation. That is, `interleave!` should insert an element of the second list between every two elements of the first list. The return value of `interleave!` is up to you. Here is a sample call (with some return values omitted for clarity):

```
STk> (define numbers (list 1 2 3 4 5))
STk> (define letters (list 'a 'b))
STk> (interleave! numbers letters)
STk> numbers
(1 a 2 b 3 4 5)
STk> letters
(a 2 b 3 4 5)
```

Test `interleave!` thoroughly and include your test cases in your submission (but comment them out). **Do not allocate any new pairs!** The point of this problem is to reuse existing pairs, not make new ones. Hence, `cons` and friends are illegal.

The homework continues on the next page.

Question 4. We'd like to write a procedure `count-pairs` that returns the number of pairs in an arbitrary structure. The following is a version that would work for any structure of pairs that can be constructed *without* mutation:

```
(define (count-pairs x)
  (if (not (pair? x))
      0
      (+ (count-pairs (car x))
         (count-pairs (cdr x))
         1)))
```

Let's take it out for a spin:

```
STk> (count-pairs (list 'a 'b 'c))
3
STk> (count-pairs (cons 'a (cons 'b 'c)))
2
STk> (count-pairs (list (list (list (list 'a)) 'b) 'c))
6
```

Mutation, however, allows us to fool `count-pairs` into thinking a structure has more pairs than it really does:

```
STk> (define test (list 'a 'b 'c))   ;; 3 pairs
test
STk> (set-car! test (cdr test))      ;; still 3 pairs
okay
STk> (count-pairs test)
5
```

Worse still, `count-pairs` will go into an infinite loop on circular structures:

```
STk> (define test (list 'a 'b 'c))
test
STk> (set-car! test test)
okay
STk> (count-pairs test)
```
doesn't return

Fix `count-pairs` so it correctly returns the number of pairs in *any* structure, circular or not. Do this by having `count-pairs` keep track of pairs it has already visited in a local list. (Yes, this means you'll need to maintain local state somewhere.) When facing a new pair, check if it is already in the list with `memq`, which is is like `member` but uses `eq?` to perform comparisons. You will need a helper.

Test the new `count-pairs` on the nastiest circular structures you can come up with.