CS61A Notes – Week 6a: Mutation, Vectors

Revenge of the Box-and-pointers

I've been waiting for several weeks to bring up something you've noticed all along – there are different degrees of "sameness" in Scheme. Or, more specifically, things can be `equal?`*, and things can be* `eq?`*. Now, you're finally old enough to know the truth.*

`equal?` Is used to compare **values**. We say two things are `equal?` if they evaluate to the same thing. For example, `(equal? '(2 3 4) '(2 3 4))` returns `#t`, since both are lists containing three elements: `2`, `3`, and `4`. This is the comparison method that you're all familiar with.

`eq?`, however, is used to compare **objects** (not in the OOP sense of the word). We say two things are `eq?` if they point to the same object. For those of you proficient in C, you may think that `(eq? x y)` if `x` and `y` are both pointers holding the same address values. In other words, `(eq? '(2 3 4) '(2 3 4))` returns `#f`, because, though the two lists hold the same values, **they are not the same list!**

Consider this:

```
> (define x (list 1 2 3))
> (define y (list 1 2 3))
> (define z x)
```

Then `(eq? x y)` returns `#f` but `(eq? z x)` returns `#t`. How many lists are created total?

QUESTION

We can also test if procedures are `equal?`. Consider this:

```
> (define (square x) (* x x))
> (define (sqr x) (* x x))
> (eq? square sqr) => #f
> (equal? square sqr) => #f
```

It's obvious that `square` and `sqr` are not `eq?`. But they're also not `equal?` because for procedures, `equal?` does the same thing as `eq?`. Why can't we tell that `square` and `sqr` really do the same thing – and thus, should be "`equal?`"? (Since you guys are always paranoid, no, this won't be on the test.)

Take a look at this procedure from *SICP*, exercise 3.14:

```
(define (mystery x)
   (define (loop x y)
      (if (null? x)
          y
          (let ((temp (cdr x)))
             (set-cdr! x y)
             (loop temp x))))
   (loop x '()))

(define v (list 'a 'b 'c 'd))
(define w (mystery v))
```

To help clarify, `loop` uses the "temporary" variable `temp` to hold the old value of the `cdr` of `x`, since the `set-cdr!` on the next line destroys the `cdr`.

QUESTION

Draw box-and-pointer diagrams that show the structures `v` and `w` after evaluating those expressions. What does Scheme print for the values of `v` and `w`?

Teenage Mutant Ninja... erm, Schemurtle (you try to do better)

Mutation refers to changing a data structure. Since our preferred data structure are pairs, naturally, then, to perform mutation on pairs, we have `set-car!` and `set-cdr!`. Note that `set-car!` and `set-cdr!` are NOT special forms! That's why you can execute things like `(set-car! (cdr lst) (+ 2 5))`.

To write procedures that deal with lists by mutation (rather than by constructing entirely new lists like we've done so far), here's a possible approach: first, try to do the problem without using mutation, as you normally would. Then, whenever you see `cons` used in your procedure, think about how you can modify the procedure to use `set-car!` or `set-cdr!` instead.

Do not confuse `set-car!` and `set-cdr!` with `set!`. `set!` is used to change the value of a *variable*, or, what some symbol in the environment points to. `set-car!` and `set-cdr!` are used to change the value *inside a* `cons` *pair*, and thus to change elements and structure of lists, deep-lists, trees, etc. They are *not the same!*

Also, in working with lists, you'll often find that you use `set-car!` to change elements of the list, and `set-cdr!` to alter the structure of the list. This shouldn't be a surprise – recall that in a list, the elements are the `car` of each pair, and the subsequent sublists are the `cdr`. But don't be fooled into thinking `set-car!` is always for element changes and `set-cdr!` is always for structural changes; in a richer data structure, either can be used for anything.

QUESTIONS

1.**Personally – and don't let this leave the room – I think `set-car!` and `set-cdr!` are useless; we can just implement them using `set!`. Check out my two proposals for `set-car!`. Do they work, or do they work? Prove me wrong:**

a.
```
(define (set-car! thing val)
  (set! (car thing) val))
```

b.
```
(define (set-car! thing val)
  (let ((thing-car (car thing)))
    (set! thing-car val)))
```

2.I'd like to write a procedure that, given a deep list, destructively changes all the atoms into the symbol `george`:

```
   > (define ls '(1 2 (3 (4) 5)))
   > (glorify! ls) => return value unimportant
   > ls => (george george (george (george) george))
```

Here's my proposal:

```
(define (glorify! L)
   (cond ((atom? L)
          (set! L 'george))
         (else (glorify! (car L))
               (glorify! (cdr L)))))
```

Does this work? Why not? Write a version that works.

3.We'd like to rid ourselves of odd numbers in our list:

```
(define my-lst '(1 2 3 4 5))
```

a.Implement `(no-odd! ls)` that takes in a list of numbers and returns the list without the odds, using mutation: `(no-odd! my-lst) => '(2 4)`

b.Implement `(no-odd! ls)` again. This time, it still takes in a list of numbers, but can return anything. But after the call, the original list should be mutated so that it contains no odd numbers. Or,

```
(no-odd! my-lst) => return value unimportant
my-lst => '(2 4)
```

(Try to consider if this is possible before you start!)

4.It would also be nice to have a procedure which, given a list and an item, inserts that item at the end of the list by making only one new `cons` cell. The return value is unimportant, as long as the element is inserted. In other words,

```
   > (define ls '(1 2 3 4))
   > (insert! ls 5) => return value unimportant
   > ls => (1 2 3 4 5)
```

Does the following procedure work? If not, can you write one that does?

```
(define (insert! L val)
   (if (null? L)
       (set! L (list val))
       (insert! (cdr L) val)))
```

5.Write a procedure, `remove-first!` which, given a list, removes the first element of the list destructively. You may assume that the list contains at least two elements. So,

```
> (define ls '(1 2 3 4))
> (remove-first! ls) => return value unimportant
> ls => (2 3 4)
```

And what if there's only one element?

6.Implement our old friend's ruder cousin, `(reverse! ls)`. It reverses a list using mutation. (This is a standard programming job interview question.)

7.Implement `(deep-map! proc deep-ls)` that maps a procedure over every element of a deep list, without allocating any new `cons` pairs. So,

```
(deep-map! square '(1 2 (3 (4 5) (6 (7 8)) 9))) =>
'(1 4 (9 (16 25) (36 (49 64)) 81))
```

8.Implement `(interleave! ls1 ls2)` that takes in two lists and interleaves them without allocating new `cons` pairs.

Just When You Were Getting Used to Lists…

Finally we are now introducing to you what many of you already know – arrays. Roughly, an array is a contiguous block of memory – and this is why you can have "instantaneous", random access into the array, instead of having to traverse down the many pointers of a list.

Recall the vector operators:

`(vector [element1] [element2] ...)` => works just like `(list [element1] ...)`
`(make-vector [num])` => creates a vector of `num` elements, all unbound
`(make-vector [num] [init-value])` => creates a vector of `num` elements, all set to `init-value`
`(vector-ref v i) => v[i]`; gets the `i`th element of the vector `v`
`(vector-set! v i val) => v[i] = val`; sets the `i`th element of the vector `v` to `val`
`(vector-length v)` => returns the length of the vector `v`

Beyond using different operators, there are a few big differences between vectors and lists:

Vectors of length N

- a contiguous block of memory cells
- O(1) for accessing any item in the vector
- O(N) for adding an item to the middle of the vector, since you have to move the rest of the vector down
- O(N) for growing a vector; you have to *reallocate* a new, larger block of memory!
- add 1 to index to get next element
- you may have "unbound" elements in the vector; that is, length of vector is not the same as length of valid data

Lists of length N

- many units of two cells linked together by pointers
- O(N) for accessing an item
- O(1) for inserting an item anywhere in the list, assuming we have a pointer to the location
- O(1) for growing a list; just add it at the beginning or the end (if you have a pointer to the end)
- `cdr` down a list
- length of list is exactly the number of elements you've put into the list

Note the last bullet. With lists, you allocate a new piece of memory (using `cons`) when you need to add an element, but with vector, you allocate all the memory you need first, even if you don't have enough data to fill it up.

Also, just as you can have deep lists, where elements of a list may be a list as well, you can also have "deep" vectors, often referred to as n-dimensional arrays, where n refers to how "deep" the deep vector is. For example, a table would be a 2-dimensional array – a vector of vectors. Note that, unlike in, say, C, your each vector in your 2D table does NOT have to have the same size! Instead, you can have variable-length rows inside the same table. In this sense, the vectors of Scheme are more like the arrays of Java than C.

QUESTIONS

1. **Write a procedure `(sum-of-vector v)` that adds up the numbers inside the vector. Assume all data fields are valid numbers.**

2. **Write a procedure `(vector-copy! src src-start dst dst-start length)`. After the call, `length` elements in vector `src` starting from index `src-start` should be copied into vector `dst` starting from index `dst-start`.**

```
STk> a => #(1 2 3 4 5 6 7 8 9 10)
STk> b => #(a b c d e f g h i j k)
STk> (vector-copy! a 5 b 2 3) => okay
STk> a => #(1 2 3 4 5 6 7 8 9 10)
STk> b => #(a b 6 7 8 f g h i j k)
```

3. **Write a procedure `(insert-at! v i val)`; after a call, vector `v` should have `val` inserted into location `i`. All elements starting from location `i` should be shifted down. The last element of `v` is discarded.**

```
STk> a => #(i'm like you #[unbound] #[unbound])
STk> (insert-at! a 1 'bohemian) => okay
STk> a => #(i'm bohemian like you #[unbound])
```

4. **Write a procedure `(vector-double! v)`. After a call, vector `v` should be doubled in size, with all the elements in the old vector replicated in the second half. So,**

```
STk> a => #(1 2 3 4)
STk> (vector-double! a) => okay
STk> a => #(1 2 3 4 1 2 3 4)
```

5. **Write a procedure `reverse-vector!`. Do I have to explain what it does?**

6. **Write a procedure `(square-table! t)` that takes in a rectangular table and squares every element.**