CS61A Notes 9 – Now The Mutants Attack [Solutions v1.0]

That Which Look The Same May Not Be The Same (Thy eyes are devil's idle playthings)

(weird) 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?`"?

The problem wants to check that square and sqr are "equal" in the sense that, given the same input, they always return the same output. This is impossible, and has been proven to be impossible! (You will see more of this in CS70). Therefore the most we can do is compare whether two things are the same *procedure*, not whether they're the same *function*.

Teenage Mutant Ninja… err, Schemurtle (you try to do better)

QUESTIONS

1. **Personally, I think `set-car!` and `set-cdr!` are pretty useless too; 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 by drawing box-and-pointer diagrams.**

   a.
   ```
   (define (set-car! thing val)
      (set! (car thing) val))
   ```
   Doesn't work – set! is a special form! It cannot evaluate what (car thing) is.

   b.
   ```
   (define (set-car! thing val)
      (let ((thing-car (car thing)))
         (set! thing-car val)))
   ```
   Doesn't work. thing-car is a new symbol bound to the value of (car thing), and the set! statement simply sets the value of thing-car to val, without touching the original thing at all.

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

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

   Here's my proposal:

   ```
   (define (glorify! ls)
      (cond ((atom? ls)
             (set! ls 'scheme))
            (else (glorify (car ls))
                  (glorify (cdr ls)))))
   ```

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

   No. Remember, to manipulate elements of a list, you need to use set-car! or set-cdr!. set! sets ls to 'scheme when ls is an atom, but once we return, the new value for ls is lost. Here's a way to do this:

```
(define (glorify! ls)
   (cond ((null? ls) '())
         ((atom? ls) 'scheme)
         (else (set-car! ls (glorify! (car ls)))
               (set-cdr! ls (glorify! (cdr ls)))
               ls)))
```

We need to return ls because the set-car! and set-cdr! expressions expect glorify! to return something – namely, the transformed sublist.

3. **We'd like to rid ourselves of odd numbers in our list:**
   `(define my-lst '(1 2 3 4 5))`
   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)`

```
(define (no-odd! ls)
   (cond ((null? ls) '())
         ((odd? (car ls)) (no-odd! (cdr ls)))
         (else (set-cdr! ls (no-odd! (cdr ls)))
               ls)))
```

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 this work? If not, can you write one that does?

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

Nope; this should be automatic by now: set! does not change elements or structure of a list! As for glorify!, you might try returning partial answers like this:

```
(define (insert! ls val)
   (cond ((null? ls) (list val))
         (else (set-cdr! ls (insert! (cdr ls) val))
               ls)))

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

This almost works. But what if ls is null?

```
(define ls '())
(insert! ls 3) ==> (3)
ls ==> '()
```

Why doesn't this work? Think carefully. The answer lies in #3b.

5. **Implement (`deep-map! proc deep-ls`) that maps a proc 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))
(define (deep-map! proc ls)
   (cond ((null? ls) '())
         ((not (pair? ls)) (proc ls))
         (else (set-car! (deep-map! proc (car ls)))
               (set-cdr! (deep-map! proc (cdr ls))))))
```

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

```
(define (interleave! ls1 ls2)
   (cond ((null? ls1) ls2)
         ((null? ls2) ls1)
         (else (set-cdr! ls1 (interleave! ls2 (cdr ls1)))
               ls1)))
```

Just When You Were Getting Used To Lists…

QUESTIONS

1. **Write procedure (`sum-of-vector v`) that adds up the numbers inside the vector.**

   You can use an accumulator or a helper like this:

```
(define (sum-of-vector v)
   (define (helper index)
      (cond ((= index (vector-length v)) 0)
            (else (+ (vector-ref v index) (helper (+ index 1))))))
   (helper 0))
```

2. **Write 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 b 5 2 3) ==> okay
STk> a ==> #(1 2 3 4 5 c d e 9 10)
STk> b ==> #(a b c d e f g h i j k)
(define (vector-copy! src src-start dst dst-start length)
   (if (> length 0)
       (begin (vector-set! dst dst-start (vector-ref src src-start))
            (vector-copy! src (+ src-start 1) dst
                  (+ dst-start 1) (- length 1)))))
```

3. **Write 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 disgarded.**

```
STk> a ==> #(cs61a is cool #[unbound] #[unbound])
STk> (insert-at! a 2 'very) ==> okay
STk> a ==> #(cs61a is very cool #[unbound])
(define (insert-at! v i val)
   (let* ((amt-to-shift (- (vector-length v) i 1))
          (temp-v (make-vector amt-to-shift)))
      (vector-copy! v i temp-v 0 amt-to-shift)
      (vector-copy! temp-v 0 v (+ i 1) amt-to-shift)
      (vector-set! v i val)))
```

NOTE: why didn't we just do (vector-copy! v i v (+ i 1) amt-to-shift)?

4. **Write 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)
```

IMPOSSIBLE! Hope you weren't fooled by this. To double the size of a vector, you'd have to allocate a new vector. However, recall that you cannot change what "a" points to from within a procedure! You can at most *return* a new vector double in size.

5. **Write procedure `(reverse-vector! v)` that reverses the elements of a vector, obviously.**

```
;; for elements from start to stop, do body
(define (from-to-do start stop body)
   (if (> start stop) 'done
       (begin (body start) (from-to-do (+ 1 start) stop body)) ))


(define (reverse-vector! v)
   (from-to-do 0 (- (quotient (vector-length v) 2) 1)
      (lambda (i)
         (let ((temp (vector-ref v i)))
            (vector-set! v i (vector-ref v (- (vector-length v) i 1)))
            (vector-set! v (- (vector-length v) i 1) temp)))))
```

6. **Write procedure `(square-table! t)` that takes in an `n`-by-`m` table and squares every element.**

```
(define (square-table! t)
   (let ((n (vector-length t))
         (m (vector-length (vector-ref t 0))))
      (from-to-do 0 (- n 1)
         (lambda(i)
            (from-to-do 0 (- m 1)
               (lambda(j)
                  (vector-set!
                     (vector-ref! t i)
                     j
                     (square (vector-ref! (vector-ref! t i) j)))))))))
```