CS61A Notes – Week 9: Mutation (solutions)

Revenge of the Box-and-pointers

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.)

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.

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`?

```
w => (d c b a)
v => (a)
```

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

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)
b.     (set! (car thing) val))
c.
d.  Doesn't work – set! is a special form! It cannot
    evaluate what (car thing) is.

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

e. 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.

1. **I'd like to write a procedure that, given a deep list, destructively changes all the atoms into the symbol** scheme:
2. **> (define ls '(1 2 (3 (4) 5)))**
3. **> (glorify! ls) => return value unimportant**
4. **> ls => (scheme scheme (scheme (scheme) scheme))**
5. **Here's my proposal:**
6. **(define (glorify! L)**
7. **(cond ((atom? L)**
8. **(set! L 'scheme))**
9. **(else (glorify! (car L))**
10. **(glorify! (cdr L)))))**
11.
12. **Does this work? Why not? Write a version that works.**
13.
14. 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) 'chung)
         (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.

1. **We'd like to rid ourselves of odd numbers in our list:**
2. **(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:
   b. **(no-odd! my-lst) => '(2 4)**
   c.
   d. (define (no-odd! ls)
   e. (cond ((null? ls) '())
   f. ((odd? (car ls)) (no-odd! (cdr ls)))
   g. (else (set-cdr! ls (no-odd! (cdr ls)))
   h. ls)))

   a. **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,**

b. **(no-odd! my-lst) => return value unimportant**
c. **my-lst => '(2 4)**
d. **(Try to consider if this is possible before you start!)**
e.
f. This, I fear, is not possible. Note that we need to skip the first element (the 1), so we'd like to do something like (set! my-lst ...) to have it point to the solution instead of the cons pair containing 1 as the car. However, inside the procedure no-odd!, we have no way of doing that; we can set! ls to all our heart's content, but we have no way of altering the value of my-lst. Make sure you understand why; this point is crucial.
g.
h. There is hope – with use of sentinels. If we always represent lists like so:
i.

```
j. (define (new-list first . rest)
k.    (cons #f (cons first rest)))
```

l.
m. where (new-list 1 2 3 4 5) => (#f 1 2 3 4 5), then we can do this:
n.

```
o. (define (no-odd! ls)
p.    (cond ((null? ls) '())
q.          ((null? (cdr ls)) ls)
r.          ((odd? (cadr ls)) (set-cdr! ls (no-odd!
   (cddr ls))) ls)
s.          (else (set-cdr! ls (no-odd! (cdr ls)))
   ls)))
```

t.
u. Carefully consider how the new representation of lists allowed us to do that. What we wanted to be able to do was to have my-lst point to something else (other than the pair containing 1). But we had no way to do that. So instead, we have my-lst point to some useless thing – like a pair containing #f – and, with that, we'll be able to set! the first element of the list to something else. If that's clear as mud, draw a few box-and-pointer diagrams!
v.
w. We used the word "sentinel" to refer to #f. A "sentinel" is a meaningless value at the start of a data structure that allows us to do what we did above. It also makes the code look prettier, as in the remove! procedure presented in lecture.

1. 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,

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

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

```
6.  (define (insert! L val)
7.     (if (null? L)
8.          (set! L (list val))
9.          (insert! (cdr L) val)))
10.
11. 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:
12.
13. (define (insert! ls val)
14.    (cond ((null? ls) (list val))
15.          (else (set-cdr! ls (insert! (cdr ls) val))
16.                ls)))
17.
18. (define ls '(1 2 3 4))
19. (insert! ls 5) => (1 2 3 4 5)
20. ls => (1 2 3 4 5)
21.
22. This almost works. But what if ls is null?
23.
24. (define ls '())
25. (insert! ls 3) => (3)
26. ls => '()
27.
28. Why doesn't this work? Think carefully. The answer lies
    in #3b.
```

1. 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,

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

5. **And what if there's only one element?**
6. **From 3 and 4, we know that we can't change what ls points, and so we can't do it the obvious way:**

```
7.
8.  (define (remove-first! L)
9.     (set! L (cdr L)))
10.
11. This doesn't work for reasons we mentioned in 2.
    Instead, our strategy will be to copy all the elements
    up one spot, and cut off the cons cell at the end of the
    list rather than the beginning:
12.
```

```
13. (define (remove-first! L)
14.    (cond ((null? (cdr (cdr L)))
15.           (set-car! L (cadr L))
16.           (set-cdr! L '()))
17.          (else (set-car! L (cadr L))
18.                (remove-first! (cdr L)))))
19.
```

20. Note that having at least two elements in the list allows us to do (cdr (cdr ls)) in the base case. If there's only one element, this wouldn't work, and in fact there's no general procedure that can remove the first element given any list. Consider:

```
21.
22. (define ls '(1))
23. (remove-first! ls)
24. ls => '()
25.
```

26. This means we'll need to change what ls points to (from a cons cell to a null list), which we know we can't do from within remove-first!.

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

```
2.
3.  ;; here's a way using state
4.  (define (reverse! ls)
5.     (define (helper! prev rest)
6.        (cond ((null? rest) prev)
7.              (else (let ((rest-of-rest (cdr rest)))
8.                 (set-cdr! rest prev)
9.                 (helper! rest rest-of-rest)))))
10.    (helper! '() ls))
11.
12. ;; this way is similar to the way we approached the old
    reverse
13. ;; problem; reverse! the rest of the list, and then
    attach the
14. ;; first element to the last
15. (define (reverse! ls)
16.    (define (last ls)
17.       (cond ((null? ls) '())
18.             ((null? (cdr ls)) ls)
19.             (else (last (cdr ls)))))
20.    (cond ((null? ls) '())
21.          ((null? (cdr ls)) ls)
22.          (else (let ((rest-reversed (reverse! (cdr
    ls))))
23.                   (set-cdr! ls '())
24.                   (set-cdr! (last rest-reversed) ls)
```

```
                      rest-reversed))))
```

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))

(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))))))
```

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)))
```