CS 450

Lecture 8: `set-car!`, `set-cdr!`, and building tables

Carl D. Offner

1 When do we allocate memory?

The basic memory allocator in Scheme is `cons`. The function `list` is really defined in terms of `cons`, so it also allocates memory.

For example, suppose you start out like this:

```
(define x (list 'a 'b))
(define z1 (cons x x))
```

Then `z1` evaluates to `((a b) a b)`. In fact, we have the data structure represented in Figure 1.

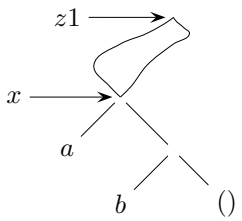

Figure 1: `(cons x x)`, where `x` is `(list 'a 'b)`

On the other hand, suppose we evaluate the following Scheme expression:

```
(define z2 (cons (list 'a 'b) (list 'a 'b)))
```

What is different here is that `list` is evaluated twice—that is, we are allocating memory two times instead of once. As a result, we will get the data structure represented in Figure 2.

Note that the two lists have the same terminal nodes. That's because symbols like `'a` and `'b`, and also the empty list, are guaranteed to be unique in Scheme's memory.

1.1 Exercise *If on the other hand, we defined*

```
(define z3 (cons (list 2 3)(list 2 3)))
```

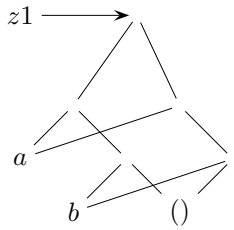

Figure 2: `(cons (list 'a 'b) (list 'a 'b))`

We would most likely get something else. Do you see why?

Note that we can't distinguish $z1$ from $z2$ by just printing them out:

```
==> z1

((a b) a b)

==> z2

((a b) a b)
```

Look at this, however:

```
==> (eq? z1 z2)

#f

==> (eqv? z1 z2)

#f

==> (equal? z1 z2)

#t
```

2 `set-car!` and `set-cdr!`

These two procedures are just pretty much what you think they are. So for instance, if we have

```
(define aaa (cons 3 4))
```

then `aaa` will evaluate to `(3 . 4)`. And if then we evaluate

```
(set-car! aaa 7)
```

then `aaa` will evaluate to `(7 . 4)`. And so on.

Here is a straightforward example, which is pretty typical of the way these procedures are used: Suppose we start out by defining

```
(define x '((a b) c d))
(define y '(e f))
```

Then Figure 3 shows what this looks like in memory.

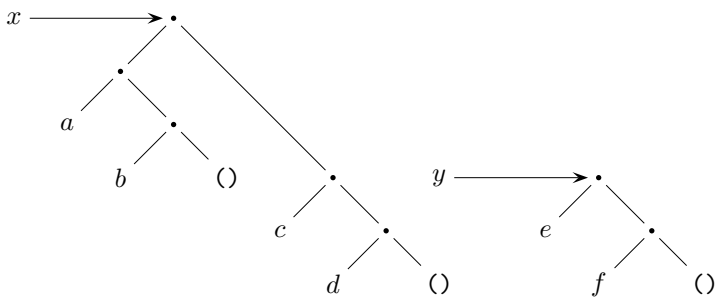

Figure 3: `x` and `y`

Now suppose we evaluate the expression

```
(set-car! x y)
```

We get what is pictured in Figure 4. In particular, `x` is now `((e f) c d)`.

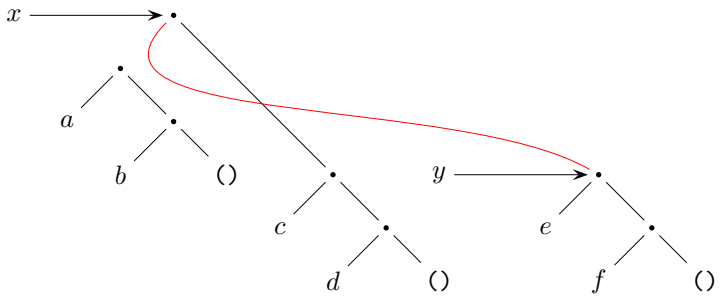

Figure 4: `x` and `y` after `(set-car! x y)`

2.1 Another tricky example

There are, however, some things you have to be careful of, and at first they may seem pretty tricky. For instance, suppose we define (as the book does) the following rather silly procedure:

```
(define (set-to-wow! x)
  (set-car! (car x) 'wow)
  x)
```

Then after evaluating

```
(set-to-wow! z1)
```

we will have

```
z1    ==>   ((wow b) wow b)
```

while after evaluating

```
(set-to-wow! z2)
```

we will have

```
z2   ==>   ((wow b) a b)
```

which shows that `z1` and `z2` really *are* different.

2.2 Representing a pair as a dispatch procedure supporting `set-car!` and `set-cdr!`

In Lecture 2, we saw how the data structure produced by the primitive procedure `cons`—that is, a pair—could be regarded itself as a procedure with "methods" (named, of course, "`car`" and "`cdr`") that extracted its two components. We can easily extend that construction to include `set-car!` and `set-cdr!`. Here's how:

```
(define (cons x y)
  (define (set-x! v) (set! x v))
  (define (set-y! v) (set! y v))
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          ((eq? m 'set-car!) set-x!)
          ((eq? m 'set-cdr!) set-y!)
          (else (error "..."))))
  dispatch)

(define (car z) (z 'car))
(define (cdr z) (z 'cdr))
(define (set-car! z new-value)
  ((z 'set-car!) new-value)
  z)
(define (set-cdr! z new-value)
  ((z 'set-cdr!) new-value)
  z)
```

3 Tables

Using these new functions, let us see how to construct tables. We'll consider both 1-dimensional and 2-dimensional tables; they are both extremely useful, and this code is something you will find useful in the programming assignments (as well as any other Scheme code you end up writing).

3.1 A 1-dimensional table with lookup and insertion

```
(define (lookup key table)
  (let ((record (assoc key (cdr table))))
    (if record
        (cdr record)
        #f)))

(define (assoc key records)
  (cond ((null? records) #f)
        ((equal? key (caar records)) (car records))
        (else (assoc key (cdr records)))))

(define (insert! key value table)
  (let ((record (assoc key (cdr table))))
    (if record
        (set-cdr! record value)
        (set-cdr! table
                  (cons (cons key value) (cdr table)))))
  'ok)  ;;; maybe val would be better than 'ok

(define (make-table)
  (list '*table*))
```

3.2 A 2-dimensional table with lookup and insertion

```
(define (lookup key-1 key-2 table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable
        (let ((record (assoc key-2 (cdr subtable))))
          (if record
              (cdr record)
              #f))
        #f)))

(define (insert! key-1 key-2 value table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable
        (let ((record (assoc key-2 (cdr subtable))))
          (if record
              (set-cdr! record value)
              (set-cdr! subtable
                        (cons (cons key-2 value)
                              (cdr subtable)))))
        (set-cdr! table
                  (cons (list key-1
                              (cons key-2 value))
                        (cdr table)))))
  'ok)  ;;; maybe val would be better than 'ok
```

3.3 A 2-dimensional table represented as a set of procedures with internal state

This is one way of implementing `put` and `get`. `put` and `get` were present from the early days of Lisp, but are not part of standard Scheme. If you really wanted them, however (and I don't generally recommend it), this is how you could implement them.

```
(define (make-table)
  (let ((local-table (list '*table*)))
    (define (lookup key-1 key-2)
      (let ((subtable (assoc key-1 (cdr local-table))))
        (if subtable
            (let ((record (assoc key-2 (cdr subtable))))
              (if record
                  (cdr record)
                  #f))
            #f)))
    (define (insert! key-1 key-2 value)
      (let ((subtable (assoc key-1 (cdr local-table))))
        (if subtable
            (let ((record (assoc key-2 (cdr subtable))))
              (if record
                  (set-cdr! record value)
                  (set-cdr! subtable
                            (cons (cons key-2 value)
                                  (cdr subtable)))))
            (set-cdr! local-table
                      (cons (list key-1
                                  (cons key-2 value))
                            (cdr local-table)))))
      'ok)  ;;; maybe val would be better than 'ok
    (define (dispatch m)
      (cond ((eq? m 'lookup-proc) lookup)
            ((eq? m 'insert-proc!) insert!)
            (else (error "Unknown operation -- TABLE" m))))
    dispatch))

(define operation-table (make-table))
(define get (operation-table 'lookup-proc))
(define put (operation-table 'insert-proc!))
```