

6.001 Tutorial 7 – Solutions

David Ziegler

October 25, 2004

1 Administrivia

- Problem set 6 is due tomorrow.

2 Feedback

- I can't move lab to west campus.
- I can't add a check button to quizzes.
- I won't give out solutions to projects.
- Solutions to tutorial problems are online.
<http://david.ziegler.ws/6.001/>
- If you bring in cookies/cake/coffee, I know everyone would be happy to eat it.

3 Mutation

Up to this point, everything we have done in Scheme has been *functional programming* – each function we write is a *function*; that is, a procedure that always returns the same value(s) for any set of inputs.

Mutation changes all that, specifically by giving us functions that can change things. Specifically, we have `set!`, `set-car!` and `set-cdr!`.

(set! name exp) `set!` evaluates the `exp`, and *replaces* the binding of `name` with the new value.

(set-car! p exp) `set-car!` evaluates `p` to get a pair, and the `exp`, and replaces the `car` of `p` with the value of `exp`.

(set-cdr! p exp) `set-cdr!` evaluates `p` to get a pair, and the `exp`, and replaces the `cdr` of `p` with the value of `exp`.

4 Examples

```
;; This function takes one argument
;; and returns the value of the argument
;; the LAST time it was called:
;; (buffer 1) => #f
;; (buffer 'foo) => 1
;; (buffer '(1 2 3)) => foo
(define buffer
  (let ((last #f))
    (lambda (x)
      (let ((old last))
        (set! last x)
        old))))
```

```
;; This function takes one argument
;; and returns #t if it has ever seen
;; that argument before
(define seen?
  (let ((seen nil))
    (lambda (x)
      (if (member x seen)
          #t
          (begin
             (set! seen (cons x seen))
             #f)))))
```

```
;; This function does destructive
;; append -- it changes the cdr of
;; the last pair to point to the
;; second list
;; Don't worry about empty x
(define (append! x y)
  (if (null? (cdr x))
      (set-cdr! x y)
      (append! (cdr x) y)))
```

5 Memoization

A useful optimization that some languages perform is *memoization*. The idea of memoization is to figure out when a function returns a constant answer for a given set of arguments. If it does, whenever we call a function, we remember the arguments and the result. Later, we can check if we've seen these arguments before and return the result without doing the computation again. For expensive functions (lots of work), this can save a lot of time. The following procedure takes a single argument function and returns a memoized version.

```
(define (memoize proc)
  (let ((vals '()))
    (lambda (arg)
      (let ((previous (assoc arg vals)))
        (if previous
            (cadr previous)
            (let ((temp (proc arg)))
              (set! vals
                    (cons (list arg temp)
                          vals))
              temp)))))))
```

A little hint – remember that `assoc` is a procedure that operates on association lists. It compares the `car` of each element of the list to the first argument and returns that element if it is. It has the following behavior:

```
(assoc 5 '((5 25) (6 36)))
; Value: (5 25)
```

```
(assoc 3 '((5 25) (6 36)))
; Value: #f
```

```
(assoc 3 '())
; Value: #f
```

Now imagine we memoize a simple procedure:

```
(define fast-square
  (memoize (lambda (x) (* x x))))
```

```
(fast-square 5)
```

What happens?