

# 6.001 Tutorial 6 – Solutions

David Ziegler

October 18, 2004

## 1 Administrivia

- Office hours will be on Tuesdays, 3–4, Stata Center, Gates tower, 9th floor lobby.
- Problem Set 5 is due tomorrow night.
- Project 3 has been pushed back (yay!) to November 5.
- Project 2 will be handed back next tutorial.

## 2 The Read, Eval, Print Loop

We talked before about the read, eval, print loop (REPL), but we've focused on the evaluation part of it. The reader takes an expression you've typed in and produces something that represents it:

```
;; Input 1  
;; Result:
```

```
;; Input (1 2 3)  
;; Result:
```

```
;; Input (define a 1)  
;; Result:
```

```
;; Input (lambda (x y) (+ x y))  
;; Result:
```

### 3 quote

The `quote` special form just returns exactly the argument it was given – whatever the reader built.

```
(quote 1)
; Value:

(quote (1 2 3))
; Value:
```

You can also use the sugared form of `quote`, `'`:

```
(quote 1) <=> '1
(quote (1 2 3)) <=> '(1 2 3)
```

### 4 Symbols

Symbols are another data type in Scheme. They are similar to strings but they are *interned*. When Scheme sees a symbol, it checks to see if it has ever seen a symbol with the same character string before. If it has not, it creates a new symbol. If it has, it returns a pointer to the symbol it created last time it saw the character string.

Because of this, any two symbols of the same name refer to the same *object*. Unlike strings, we can compare two symbols for equality in constant time, by checking if they point to the same object.

We use symbols for many purposes:

- Tagged data – data types of the form `(type-of-data data data data)`.
- Deferring evaluation – we'll see this later in the term.
- Giving names to values – `(define a b)` associates the value of `b` with the *symbol* `a`.

### 5 Equality

We've talked about `=`, used to compare numbers for equality. We now have `eq?` and `equal?`.

`(eq? a b)` is the simplest test. It checks whether `a` and `b` are the same object (for C programmers, whether `a` and `b` are the same pointer). It works for booleans and symbols, but *not for numbers!*

```
(eq? #t #t) => #t
(eq? 'foo 'foo) => #t
(eq? 'foo 'FOO) => #t
(eq? (list 1 2 3) (list 1 2 3)) => #f
```

`(equal? a b)` checks whether `a` and `b` print out the same and works for almost everything.

```
(equal? #t #t) => #t
(equal (list 1 2 3) (list 1 2 3)) => #t
```

## 6 Tagged Data

The idea of tagged data is to attach an identification to non-trivial data values and to check the identification before operating on the data.

```
;; This procedure creates a piece of tagged data
(define (tag type data)
  (cons type data))

;; This procedure checks if an object is of a certain tagged type
(define (tagged-with? obj type)
  (and (pair? obj)
       (eq? (car obj) type)))

;; This procedure returns the type of an object
(define (type obj)
  (if (pair? obj)
      (car obj)
      (error "Bad object!")))

;; This procedure returns the data associated with an object
(define (data obj)
  (if (pair? obj)
      (cdr obj)
      (error "Bad object!")))
```

We can gain a lot by doing so – two important things:

- Data-directed programming.
- Defensive programming.

## 7 A Payroll System

We're going to create a set of procedures for working with timecards and calculating payroll. Some timecards are hourly-based:

```
(hourly? <obj>) => #t ;; if <obj> is an hourly timecard
(hourly-rate <card>) => dollars per hour
(hourly-hours <card>) => list of hours worked, e.g. (7 8)
```

and some are daily-based:

```
(daily? <obj>) => #t ;; if <obj> is a daily timecard
(daily-rate <card>) => dollars per day
(daily-days <card>) => number of days worked this week
```

We want to be able to calculate the total payroll for the week.

```
;; This procedure returns the total payroll
(define (payroll cards)
  (sum paycheck-amount cards))

;; This procedure applies proc to each element of lst and adds the results
(define (sum proc lst)
  (if (null? lst)
      0
      (+ (proc (car lst)) (sum proc (cdr lst)))))

;; This procedure returns the amount of the paycheck for a timecard
(define (paycheck-amount card)
  (cond ((hourly? card) (pay-for-hourly card))
        ((daily? card) (pay-for-daily card))
        (else (error "Bad timecard!"))))

;; This procedure returns the pay for a daily timecard
(define (pay-for-daily card)
  (* (daily-rate card)
     (daily-hours card)))

;; This procedure returns the pay for an hourly timecard
(define (pay-for-hourly card)
  (sum (lambda (x)
        (* x (hourly-rate card)))
       (hourly-hours card)))
```

Next, we need to implement the actual data abstractions.

```
;; The hourly timecard constructor
(define (make-hourly rate hours)
  (tag 'hourly (list rate hours)))

;; The hourly timecard predicate
(define (hourly? obj)
  (tagged-with? obj 'hourly))

;; The rate for the hourly timecard
(define (hourly-rate card)
  (if (hourly? card)
      (car (data card))
      (error "Not a card: " card)))

;; The list of hours on an hourly timecard
(define (hourly-hours card)
  (if (hourly? card)
      (cadr (data card))
      (error "Not a card: " card)))
```

Then we realize that we'd like to know how many total hours are on an hourly timecard. We write `hourly-total-hours`, which returns the total number of hours worked during the week on an hourly timecard. This makes `pay-for-hourly` much simpler.

```
;; This procedure returns the total number of hours worked on an
;; hourly timecard
(define (hourly-total-hours card)
  (sum (lambda (x) x) (hourly-hours card)))

;; This procedure is a simplified version of pay-for-hourly from before
(define (pay-for-hourly card)
  (* (hourly-rate card)
     (hourly-total-hours card)))
```

Our payroll system is working wonderfully so far, but then a new CEO is hired, and he requests we add weekly timecards. Weekly employees are paid a weekly rate, regardless of how many days or hours they show up, as long as they turn in a timecard.

```
;; This procedure creates a weekly timecard
(define (make-weekly rate)
  (tag 'weekly (list rate)))

;; Other procedures?
(define (weekly? obj)
  (tagged-with? obj 'weekly))

(define (weekly-rate card)
  (if (weekly? card)
      (car (data card))
      (error "Not a card: " card)))

;; This procedure returns the amount of pay for a weekly timecard
(define (pay-for-weekly card)
  (weekly-rate card))
```

Of course, now we need to modify our payroll system. How do these change?

```
;; This procedure returns the total payroll
(define (payroll cards)
  (sum paycheck-amount card))

;; This procedure returns the amount of the paycheck for a timecard
(define (paycheck-amount card)
  (cond ((hourly? card) (pay-for-hourly card))
        ((daily? card) (pay-for-daily card))
        ((weekly? card) (pay-for-weekly card))
        (else (error "Bad timecard!"))))
```

The new CEO also wants to know how much work is being done in the company per week. Assume that someone has written `daily-total-hours` and `weekly-total-hours` (how these two work, we're not exactly sure...).

```
;; This procedure returns the total number of hours worked in the company
(define (total-company-hours cards)
  (sum (lambda (card)
        (cond ((hourly? card) (hourly-total-hours card))
              ((daily? card) (daily-total-hours card))
              ((weekly? card) (weekly-total-hours card))
              (else (error "Bad timecard!")))))
  cards))
```

## Feedback

1. If you had to choose one thing to change about tutorial, what would it be?
2. Are the examples good? Bad? Unintelligible? Greek?
3. How are explanations in tutorial? Are there things that I do a poor job of explaining?
4. Are there things that you would like to see online that would be useful?  
(As a reminder — <http://david.ziegler.ws/6.001/>)
5. Are there things I could do to make 6.001 more pleasant? Preemptively — I can't get rid of projects, problem sets, quizzes, ... Sorry.
6. Any other comments?

Thanks for your feedback!