

6.001 Tutorial 9

David Ziegler

November 8, 2004

1 Administrivia

- Quiz 2 is tomorrow, 7:30 – 9:30!
- Coverage is through environment models.
- One page of notes, both sides.
- Review session tonight, 5:00, 4-159.
- Office hours, tomorrow, 3-4, Stata, Gates tower, 9th floor lobby.

```
(define x 3)
((lambda (x y) (+ (x 1) y))
 (lambda (z) (+ x 2))
 3)
```

2 Environments!

```
(define (identity x) x)
(define square
  (identity (lambda (x) (* x x))))
(square 5)
```

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
(fact 2)
```

```
(define x 2)
(define proc
  (let ((x (* x 5))
        (f (lambda (y) (* x y))))
    f))
(proc 3)
```

```
(define x 2)
(define proc
  (let ((x (* x 5)))
    (let ((f (lambda (y) (* x y))))
      f)))
(proc 3)
```

Environment Model Cheat Sheet

To evaluate an expression in an environment e , follow the rule:

name Lookup name in the current environment, moving up frames to find the name. Return the value bound to the name.

(define name exp) Evaluate exp in e to get val , and create or replace a binding for name in the first frame of e with name. Return unspecified.

(set! name exp) Evaluate exp in e to get val , and replace the first binding you come to for name in e with val . Return unspecified.

(lambda args body) Create a double bubble whose environment pointer (right half) is e , and set the left half to have the parameters $args$ and body $body$. Return a pointer to the double bubble.

(let ((var init) ...) body) Evaluate each $init$ in e (in any order) to get $vals$, drop a frame that points to e , bind each var to the associated val in the new frame, evaluate the $body$ in the newly created environment. Return the value of the last expression in the $body$.

(proc arg arg arg) Evaluate each expression in the combination in e , in any order. Apply the first value to the rest of the values. Return the value from applying.

To apply a procedure p :

- Create a new frame f .
- Set f 's parent pointer to be the same as p 's environment pointer — they are “hand-cuffed” together.
- In f , bind the parameters of p to the values of the arguments it was applied to.
- Evaluate the body of p in the newly created environment.
- Return the value of the last expression in the body.

Common Environment Model Mistakes

- Be sure to set the parent pointer of new frames properly — it's the same as the environment pointer of *the procedure you're applying!*
- Keep track of what expression you're evaluating, and remember what steps you have left to do. For example, when you have `(define foo bar)`, don't forget to add the binding for `foo` after you finish evaluating `bar!`
- Don't get ahead of yourself! A common mistake is for people to evaluate a lambda expression, giving a double bubble, and then immediately evaluate the body of the lambda. Be sure that you follow the rules carefully!

Number Procedures

(quotient n1 n2)

(remainder n1 n2)

(modulo n1 n2) These procedures “do the right thing.” `remainder` always returns a number with the sign of $n1$, `modulo` always returns a number with the sign of $n2$.

(gcd n ...)

(lcm n ...) These procedures return the greatest common divisor or least common multiple (respectively) of their arguments. The result is always non-negative.

(floor x)

(ceiling x)

(truncate x)

(round x) These procedures return integers. `floor` returns the largest integer not larger than x . `ceiling` returns the smallest integer not smaller than x . `truncate` returns the integer closest to x whose absolute value is not larger than the absolute value of x . `round` returns the closest integer to x , rounding to even when x is halfway between two integers.

(random modulus) `random` returns a pseudo-random number between zero (inclusive) and `modulus` (exclusive). The exactness of the result is the same as the exactness of `modulus`.

List Procedures

(cons* obj obj ...) cons* is like list, except it conses together the last two arguments rather than consing the last argument with the empty list.

```
(cons* 'a 'b 'c) => (a b . c)
(cons* 'a 'b '(c d)) => (a b c d)
(cons* 'a) => a
```

(list-copy lst) list-copy returns a newly allocated copy each of the pairs comprising lst. It does not touch the elements of the list. You can use tree-copy to make a deep copy of a list.

(list-ref lst k) list-ref returns the kth element of lst, using 0-based indexing.

(sublist lst start end) sublist returns a newly allocated list of the elements of lst beginning at index start (inclusive) and ending at end (exclusive).

(list-head lst k) list-head returns a newly allocated list of the first k elements of lst.

(list-tail lst k) list-tail returns the sublist of lst obtained by omitting the first k elements.

(last-pair lst) last-pair returns the last pair in lst.

(list-transform-positive lst pred)

(list-transform-negative lst pred)
These procedures return a newly allocated copy of lst containing only those elements for which pred returns (respectively) true or false.

(delq element lst)

(delv element lst)

(delete element lst) Returns a newly allocated copy of lst with all elements equal to element removed. delq uses eq? to compare elements, delv uses eqv?, and delete uses equal?.

(memq obj lst)

(memv obj lst)

(member obj lst) Returns the first pair of lst whose car is obj. If obj does not appear in lst, #f is returned. memq uses eq? to compare obj, memv uses eqv?, and member uses equal?.

(map proc lst lst ...) proc must be a procedure that takes as many arguments as there are lsts. If there is more than one lst given, they must all be the same length. map applies proc element-wise to the elements of the lsts, and returns a list of the results. The order in which proc is applied is unspecified.

(for-each proc lst lst ...) Just like map, but proc is applied in order, from left to right, and the result is unspecified.

(fold-right proc init lst) Combines all the elements of lst using the binary operation proc.

(sort lst proc)

(merge-sort lst proc)

(quick-sort lst proc) Returns a newly allocated list whose elements are those of lst, rearranged to be in the order specified by proc. sort is an alias for merge-sort. quick-sort is an alternative sorting implementation.

Miscellaneous Procedures

(apply proc obj obj ...) Calls proc with the elements of the list (cons* obj obj ...) as arguments.