6.001 Tutorial 8 Notes

TA: Gerald Dalley
4–5 Apr 2005

Environment Model Problems

```
(define a 5)
((lambda (a) (set! a 2)) a)
a
```

```
(define counter
  (let ((count 0))
    (lambda ()
      (set! count (+ 1 count))
      count))))
(counter)
(counter)
```

```
(define (fact n) (if (= n 0) 1 (* n (fact (- n 1)))))
(fact 3)
```

```
(define ifact
  (lambda (n)
    (define (help n p)
      (if (= n 0)
          p
          (help (- n 1) (* p n))))
    (help n 1)))
(ifact 3)
```

```scheme
(define (make-counter)
  (let ((count 0))
    (lambda ()
      (set! count (+ 1 count))
      count)))
(define a (make-counter))
(define b (make-counter))
(a)
(a)
(b)
```

```scheme
(define (foo n)
  (lambda (y)
    (lambda ()
      (set! n (+ n y))
      n)))
(define bar (foo 0))
(define baz (bar 2))
(baz)
(baz)
```

```scheme
(define x 4)
(let ((x 5)
       (y (+ x 6)))
      (+ x y))
```

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

```
(define x
  (let ((a (list 9)))
    (let ((b (cons a a)))
      (cons b b))))
x
```

```
(define (compose f g)
  (lambda (x) (f (g x))))
((compose inc abs) -5)
```

```
(define (iter f n)
  (if (- n 1) f
      (compose f (iter f (- n 1)))))
(define plus3 (iter inc 3))
(plus3 4)
```

Number Procedures

`(quotient n1 n2)`
`(remainder n1 n2)`
`(modulo n1 n2)` These procs "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)` 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)` returns the `k`th element of `lst`, using 0-based indexing.

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

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

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

`(last-pair lst)` 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 `lst`s. 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 `lst`s, 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.

Environment Model Cheat Sheet

Elements of the Environment Model

- A frame consists of a list of variable bindings. Each binding associates a name (must be a symbol) with a value. We draw frames as boxes.
- Every frame except the GE frame has a "parent" pointer which points to another frame (also called the enclosing environment). The frames form a tree structure, with the GE as root.
- With each frame, there is an associated environment. The environment of a frame F consists of the chain of frames F, $parent(F)$, $parent(parent(F))$, until we hit the GE.
- GE = Global Environment. All initial bindings (*e.g.* for `+`, `map`) live in the GE. Whenever we evaluate something, we must specify the frame in which we evaluate it.
- New frames are created when a procedure is called, or when a `let` statement is evaluated.

The Hats

Double-bubble: In charge of the lambda rule (double-bubble creation)

Bind: In charge of step 5 of the combination rule, `set!` rule, `define` rule, symbol lookup rule

Trouble: In charge of steps 2–4 of the combination rule

Grand Evaluator: In charge of keeping track of evaluation, current environment, identifying the type of expression, and remembering the values of arguments

Evaluation Rules

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

`name`$|_e$

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

`(define name exp)`$|_e$

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)`$|_e$

Evaluate `exp` in e to get `val`, and replace the first binding for `name` in e with `val`. If no such binding is found, generate an error. Return unspecified.

`(lambda args body)`$|_e$

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.

`(exp1 exp2 exp3...)`$|_F$

·

1. Evaluate each expression `exp1`, `exp2`, `exp3`, ... in frame F, resulting in `val1`, `val2`, `val3`, ...
2. If `val1` does not point to a double-bubble, it is an error. Otherwise, let P be this double-bubble.
3. Create a new frame A
4. Make A into an environment E: A's enclosing environment pointer goes to the same frame as the environment pointer of P. Link these two pointers together with handcuffs.
5. In A, bind the parameters of P to the values `val2`, ... (`val1` is the double-bubble)
6. Evaluate the body of P with E as the current environment.

`(let ((var init) ...) body)`$|_F$

Either desugar the `let`, or:

1. Evaluate each `init` in F (in any order) to get `vals`
2. Drop a new frame A that points to F
3. In A, bind each `var` to the associated `val`
4. Evaluate `body` in frame A.
5. 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!