# CS61A Notes 14 – Your Father's Worst Nightmare (v1.0)

**The Lazy Way Out**

The lazy evaluator implements normal order of evaluation that you learned about way back when. We do this by *delaying* execution as much as possible by *thunking* it; when we can't delay it further, we *force* the thunk to obtain the actual value.

A **thunk** in the lazy evaluator is a list whose first element is the word `thunk`, second element the expression we're delaying, and third element the environment in which to evaluate the expression when we need to force the thunk.

There's only one place where we delay evaluation: **we thunk all arguments to a compound procedure.**

There are four places in which we want to force:

1. We always force the operator of a procedure call
2. We always force arguments to a primitive procedure
3. We always force what we're printing out to the screen on the top level
4. In special cases of special forms, like the predicate of `if`, we also want to force thunks

The procedure that actually does the delaying is called `delay-it`:

```
(define (delay-it exp env)
   (list 'thunk exp env))
```

Obviously, this just creates a thunk that looks just as we described above. The procedure that we use to force an expression is called `actual-value`:

```
(define (actual-value exp env)
   (force-it (mc-eval exp env)))
```

Note that `actual-value` *always* takes in a valid Scheme expression, and never a thunk! Thus, it can first call `mc-eval` to evaluate the expression. Then, since `mc-eval` might return a thunk, it needs to call `force-it` to actually force the thunk to get the real answer.

Note that the lazy evaluator also memorizes forced thunks. Take a look at the memoizing version of `force-it`:

```
(define (force-it obj)
   (cond ((thunk? Obj)
          (let ((result (actual-value
                         (thunk-exp obj)
                         (thunk-env obj))))
            (set-car! obj 'evaluated-thunk)
            (set-car! (car obj) result)
            (set-cdr! (cdr obj) '())
            result))
         ((evaluated-thunk? obj)
          (thunk-value obj))
         (else obj)))
```

First, note the mutual recursion between `force-it` and `actual-value`; one calls the other. This is because forcing an object might yield another thunk, so we need to keep forcing it until we get some actual value. And note how we perform the memoization: when we find a thunk, we first call `actual-value` on the delayed expression to find the value. Then, we *transform* the thunk into an *evaluated-thunk*, and store the result of the forcing into the evaluated-thunk. From then on, if we try to force this evaluated-thunk again, we simply return the value we've already calculated.

**Of Pills And Other Euphoric Items**

It becomes headache-inducing, panic-striking and mind-numbing to trace through, in your head, what's being delayed and what's being forced at what time. So we give you the following extension to the environment diagram so that you may have a sound footing.

First, we need to add a new element to the happy family of environment diagrams: the PILL. The pill is used to represent a thunk, and it has two elements; the first is the expression that we're delaying, and the second is the environment in which to evaluate that expression should the thunk be forced.

Here are the changes to the environment diagrams:

1. To apply a **primitive procedure**: find the actual values of the arguments and the operator.
2. To apply a **compound procedure**: find the actual value of the operator. Then, create separate thunks for each of the arguments, each having its environment pointing to the current environment. Then, create a new frame as before pointing to the procedure environment. The parameters should point to the corresponding argument thunks, rather than argument values.
3. Before you return anything to the top level, find its actual value.

But how do we find the actual value?

1. The actual value of a **non-thunked expression** is just its value as usual under non-lazy evaluators.
2. The actual value of a **thunked expression** is its thunked expression evaluated in the thunk-environment. After finding out its actual value, shade the pill so that it becomes an *evaluated thunk*, and replace the thunked expression with the value that you just obtained.
3. The actual value of an **evaluated-thunk** is the value stored when the thunk was forced the first time.

**QUESTIONS: What is printed at each line?**

1.
```
> (define x (+ 2 3))
> x ==> ?
> (define y ((lambda(a) a) (* 3 4)))
> y ==> ?
> (define z ((lambda(b) (+ b 10)) y))
> z ==> ?
```

2.
```
> (define count 0)
> (define (foo x y) (x y))
> (define z (foo (lambda(a) (set! count a) (* a a))
                 (begin (set! count (+ 1 count)) count)))
> count ==> ?
> z ==> ?
> count ==> ?
```

3.
```
> (define count 0)
> (define (incr!) (set! count (+ count 1)))
> (define (foo x)
    (let ((y (begin (incr!) count)))
       (if (<= count 1)
           (foo y)
           x)
> (foo 10) ==> ?
```

**Nondeterministic and Indecisive**

The nondeterministic evaluator extends the metacircular evaluator with nondeterministic searching. It’s good, clean American fun. First we take a look at the new special form, amb, which takes in any number of arguments. If there are no arguments, it *fails*. If there are arguments, it chooses the first one; if the first one causes a failure, it chooses the second one; if that one causes a failure, it chooses the third, etc., until it runs out of arguments, at which point it itself signals a failure.

A companion to `amb` is `require`, which takes in a predicate value, and causes a failure if the value is `#f`.

```
(define (require pred?)
   (if (not pred?) (amb)))
```

Since you are trained on the imperative and functional ways of programming, this makes the structure of your program very unintuitive. But just imagine that the ambevaluator works magically; when an error is signaled by `(amb)`, you are allowed to fly to the nearest non-empty call to `amb`, try the next option and start over.

The book and the lectures already present several examples of programs written for the nondeterministic evaluator. But I will provide another motivating example…

Consider the problem of finding a path in a binary tree from the root to an element. Consider this tree `t`:

```
                                3
                   /                         \
                  5                           6
             /        \                 /          \
         1               30          2                 5
       /               /   \       / \
   20                 0     12     1  32
```

Note that this is not a binary search tree! Here’s what we’d like:

```
> (path-to t 12)
(l r r) ;; to get from root to 12, go left, right, right
> (path-to t 32)
(r l r) ;; to get to 32, go right, left, right
> (path-to t 19)
#f ;; 19 is not in the tree
```

Here’s how we’d need to write path-to in good old Scheme:

```
(define (path-to tree x)
   (cond ((empty-tree? tree) #f)
         ((eq? (datum tree) x) ‘())
         (else
            (let ((result (path-to (left-branch tree) x)))
               (if result
                   (cons ‘L result)
                   (let ((result (path-to (right-branch tree) x)))
                     (if result
                         (cons ‘R result)
                         #f)))))))
```

An ugly little thing! But suppose you take advantage of the amb evaluator:

```
(define (path-to tree x)
   (cond ((empty-tree? tree) (amb))
         ((eq? (datum tree) x) '())
         (else (amb (cons 'L (path-to (left-branch tree) x))
                    (cons 'R (path-to (right-branch tree) x))))))
```

In the recursive case, we use `amb` to choose *either* walking down the left branch or the right branch of the tree. Suppose we chose left; then, if, eventually, we reach an empty tree, we're going to signal an error, causing us to switch to right. If that throws another error, we ourselves will signal an error (since our `amb` has run out of options), and our parent will explore other options (either trying the right branch or signaling failure to its parent).

Even better – if there are two elements in the tree, we can get the path to one and, after entering `try-again`, can get the path to the other. Think of the nightmare of doing that with regular Scheme!

**QUESTIONS**

1. **Suppose we type the following into the amb evaluator:**

```
> (* 2 (if (amb #t #f #t)
           (amb 3 4)
           5))
```

**What are all possible answers we can get?**

2. **Write a function `an-atom-of` that dispenses the atomic elements of a deep list (not including empty lists). For example,**

```
> (an-atom-of '((a) ((b (c))))) ==> a
> try-again ==> b
```

3. **Use `an-atom-of` to write `deep-member?`.**

4. **Fill in the blanks:**

```
> (define (choose-member L R)
    (cond ((null? R) (amb))
          ((= (car L) (car R)) (car L))
          (else (amb (choose-member L (cdr R))
                     (choose-member (cdr L) R)))))
> (choose-member '(1 2 3) '(4 2 3))
```

_________________________

```
> try-again
```

_________________________

```
> try-again
```

_________________________