

## Lexical vs. Dynamic Scope

...where do I point to?

### Intro...

- Remember in Scheme whenever we call a procedure we pop a frame and point it to where the procedure points to (its defining environment).
- This is called lexical scoping.

### Intro...

- So what's the difference between lexical and dynamic?
- In dynamic scope, when calling a procedure, the current environment is **NOT** extended where the procedure points to.

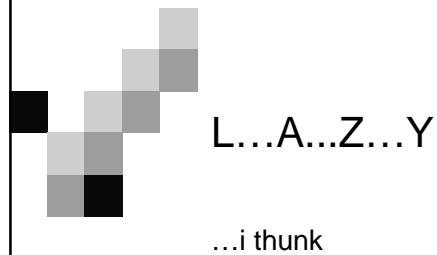
### Example.

- (define pie 'pie)
- (define (pie-maker pie)  
    (word pie n))
- (define (yum n)  
    (word (pie-maker n) pie))
- (yum 'apple)

## Practice...revisited.

- > (define x 10)
- > (define (foo y)  
 (let ((x 20)  
 (f (lambda (z) (y))))  
 (f 14)))
- > (foo (lambda () x))

What is the result using lexical scope?  
How about dynamic scope?



## So Far...

- We saw the MCE and all its glory...
- So how about changing the MCE so that we defer operations.
  - Create a normal order evaluation interpreter.
  - How to do that?

## Modify...

- So let's modify the MCE so that we defer evaluation of procedures until we need the value
  - This is only needed for compound procedures and not primitives (this will be later explained)
- Need to modify procedure calls
  - Delay arguments ("thunk")

## Delays...

- What should you delay???
- Arguments to procedures.
- THAT'S IT!

## Thinking...

- Let's take this expression  
`((lambda (x y) x) + -) 3 4`
- So we would think 3 & 4 and evaluate the the operator  
`((lambda (x y) x) + -)`
- We then think everything again and eval the lambda  
`#[closure args=(x, y) ...]`
- Then we pass in the thunked '+' and '-' and we replace x with the thunked '+'

## Thunks...

`(+ 3 4)`

- To evaluate a thunk is to **force** the argument.
- Since the operator is a primitive we **force** the operator and **force** all of it's arguments
- So we get the return value of 7.
- More explanations to come!

## Force...

- What should be forced???
- Arguments to primitives
  - ie `(+ 3 4)` ← 3 and 4 are forced.
- Operators in procedure calls
  - ie `((lambda (x) x) (foo x))` ← lambda is forced.
- Value to be printed by the driver loop
- Parts of special forms
  - ie. Predicate of an if-statement

## How do we change the evaluator?

- So what's the plan?
  - Change it so that we defer evaluation of the arguments
  - Modify the apply so that we call primitives and delay compound procedures
  - Implement the "thunk"

## Changing EVAL...

```
(define (eval exp env)
  (cond ...
    ((application? exp)
     (apply (eval (operator exp) env)
            (list-of-values (operands exp) env)))
    ...))
```

**Remember...**list of values recursively calls eval for each of the arguments, so instead let's create it so it delays all of the arguments!

## Changing EVAL...

```
(define (eval exp env)
  (cond ...
    ((application? exp)
     (apply (actual-value (operator exp) env)
            (operands exp) env))
    ...))
```

Here instead of doing eval on the operator, we use actual-value. This procedure **forces** a promise because eval may return a promise or an actual value.

## Changing APPLY...(primitive)

```
(define (apply proc args)
  (cond ((primitive-procedure? proc)
        (apply-primitive-procedure
         proc args))
        ...))
```

So we said that for primitives we **force** all the arguments to the primitive so...

## Changing APPLY...(primitive)

```
(define (apply proc args env)
  (cond ((primitive-procedure? proc)
        (apply-primitive-procedure
         proc (list-of-arg-values args env)
         ...))
```

So we call list-of-arg-values which forces all of the arguments to the primitive.

## List-of-arg-values

```
(define (list-of-arg-values exps env)
  (if (no-operands? exps)
      '()
      (cons (actual-value (first-operands exps) env)
            (list-of-arg-values
             (rest-of-operands exps) env))))
```

So this forces each argument by calling actual-value on each of the arguments and returns a list of values.

## Changing APPLY...(compound)

```
(define (apply proc args)
  (cond ...
        ((compound-procedure? proc)
         (eval-sequence
          ...
          (extend-environment
           (procedure-parameters proc)
           (list-of-delayed-args arguments env)
           (procedure-environment proc))))
        ...))
```

The list-of-delayed-args will use eval to get the value of each argument but this time will use delay and make a list of **thunks**

## List-of-delayed-values

- So list-of-delayed-args looks similar to list-of-values, but instead of just evaluating each argument. We delay each of the arguments

```
(define (list-of-delayed-args exps env)
  (if (no-operands? exps)
      '()
      (cons (delay-it (first-operand exps) env)
            (list-of-delayed-args exps env))))
```

## Creating delay and force...

- We can't just use the underlying Scheme's **delay**. This wouldn't make a thunk that we could use in the MCE.
- Let's create a delay and force.

## Thunks...

- What's a thunk?
  - It's basically an expression we evaluate later in a certain environment
- So...

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

(define (thunk? obj)
  (tagged-list? obj 'thunk))

(define (thunk-exp thunk) (cadr thunk))

(define (thunk-env thunk) (caddr thunk))

in mc-eval...
((thunk? exp) exp)
```

## Evaluating thunks...

- Thunks can only return a value when forced.
- Actual-value:

```
(define (actual-value exp env)
  (force-it (eval exp env)))
```
- Force-it will evaluate the thunk until a value is reached.

## Force-it...

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

So what's evaluated-thunk???

## Memoizing...

- Once a thunk is evaluated, it's no longer a thunk and thus it's been evaluated...  
(define (evaluated-thunk? obj)  
 (tagged-list? obj 'evaluated-thunk))  
  
(define (thunk-value evaluated-thunk)  
 (cadr evaluated-thunk))

## Use Environment Diagrams!!!

- New Rules for Evaluation...
  - If the car's not a special form, then force the car and delay all the arguments
  - Bind the variables to the arguments
  - Evaluate the body
  - If an expression evaluates to a thunk, don't evaluate it! Just return the thunk, unless it's being printed by the read/eval/print loop
- Let's practice!

## Let's give it a go...

- > (define count 0)
- > (define (add x y)  
 (set! count (+ count 1))  
 (+ x y))
- > (define w (add 3 (add 4 5)))
- > count → ???
- > w → ???
- > count → ???

## Things to know...

- What changes to the MCE you need to do to implement lazy evaluation
- Practice doing lazy evaluation by environment diagrams
- Make sure you know the MCE!!!
- Chapter 4 is your friend!
- Don't understand? Come and talk with me.



## Lazy Terminology

- Normal Order vs. Applicative Order
  - refers to order of evaluation of arguments
  - Applicative Order: Scheme
  - Normal Order: lazy evaluation
- strict vs. non-strict
  - refers to procedures and arguments
  - strict: evaluate arguments before entering body of procedure (scheme procedures)
  - non-strict: evaluate arguments later

## More Lazy Terminology

- call-by-value, call-by-name (thunks), call-by-need (memoized thunks):
  - call-by-value: pass in values of arguments
  - call-by-name: values are “thunkified”, and passed in as thunks
  - call-by-need: thunks are memoized (or mini-memo for those that don’t believe that it’s memoized) so that the value isn’t computed again.

## Review Lazy...

- Delay
  - Only arguments to compound procedure calls
- Force
  - Arguments to primitive procedure calls
  - Operators to procedure calls (because what would you apply to the thunked arguments?)
  - The IF predicate
  - Values to the print loop



## Lazy Below the Line

- Things to change
  - **MC-EVAL – application? Clause**  
((application? exp)  
  (apply (**actual-value** (operator exp) env)  
          (operands exp) env))
  - **MC-APPLY**  
((primitive-procedure? proc)  
  (apply-primitive-procedure  
    proc (**list-of-arg-values** args env))  
(compound-procedure? proc)  
  (eval-sequence  
    ...  
    (extend-environment  
      (procedure-parameters proc)  
      (**list-of-delayed-args** arguments env)  
      (procedure-environment proc) ) )

## Lazy Environment Diagrams...

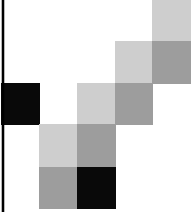
- The only change in evaluation...
  - When it's a procedure call, evaluate first argument (force the operator)
  - Then thunkify the arguments by drawing pills. Where the left side points to the argument being thunked and the right points to where the thunk is being evaluated.
  - Then do the normal popping of frame, binding the arguments to the thunks you created, and then evaluating the body of the function you're calling.
  - Evaluating thunks:
    - If memoizing thunks, you point the variable to the return value of the thunk
    - If unmemoized, you just force the thunk but leave the variable pointing to the thunk as is.

## Draw the environment diagram...

- > (define w 100)
- > (define (foo x y) (x y))
- > (define q (foo (lambda (z) (set! w 50) z)  
                  (begin (set! w 10) 3))))
- > w → ???
- > q → ???
- > w → ???

## Draw the environment diagram solutions...

- > (define w 100)
- > (define (foo x y) (x y))
- > (define q (foo (lambda (z) (set! w 50) z)  
                  (begin (set! w 10) 3))))
- > w → 50
- > q → 3
- > w → 10



**Next Time:  
Amb Eval**

...nondeterministic  
programming...the fun never  
stops.