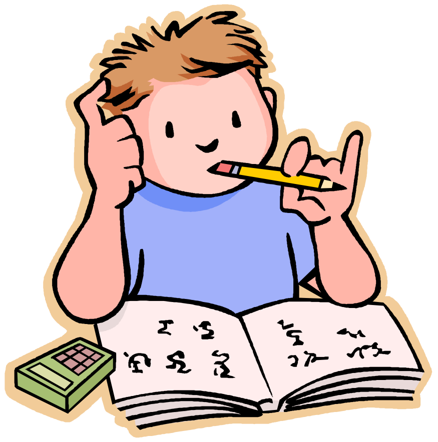
6.001 Final Exam Review

Spring 2005

By Gerald Dalley

Study Resources

- Lectures
  - 2005-05-12 lecture had some good summary portions (+ preview of future courses)
- Previous exams
  - Skip any problems with **ec-eval** or **(goto …)**
- Online tutor
- Tutorial problems



Topics

- Scheme
- Procedures and recursion
- Orders of growth
- Data abstractions
- Higher-order procedures
- Program methodology
- Symbols and quotation
- Abstract data types
- Mutation
- Environment model
- Object-oriented systems
- Interpretation / evaluation
- Lazy evaluation
- Asynchronous computing

Fall 1998 Exam: True/False

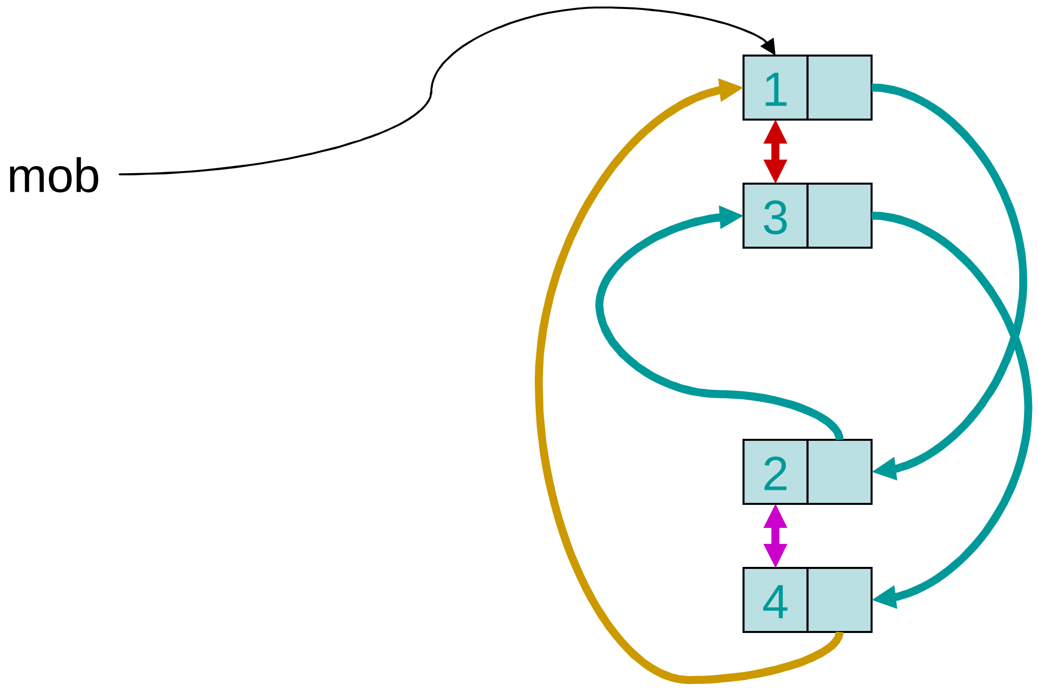
- If the Scheme evaluator supports the **delay** and **force** special forms, it is possible for the Scheme user to implement **cond** as a simple procedure without additional extensions to the evaluator.
  – False: delay requires that the user explicitly mark delayed expressions. The **cond** “procedure” needs to implicitly delay all of its arguments.
- Deadlock can occur between two processes that are running in parallel if a mutual exclusion approach is used (such as the synchronization approach discussed in class) in which both processes try to gain access to a single shared resource.
  – False: we need two shared resources for deadlock (given the scheme presented in class)

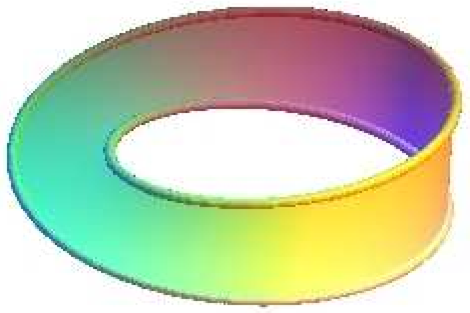
Box-and-Pointers

```
(define mob '(1 2 3 4))
(set-cdr! (cdddr mob) mob)
(define (l x y) (set-car! x y) (set-car! y x))
(l mob (cddr mob))
(l (cdr mob) (cdddr mob))
mob
```

Box-and-Pointers

```
(define mob '(1 2 3 4))
(set-cdr! (cdddr mob) mob)
(define (l x y) (set-car! x y) (set-car! y x))
(l mob (cddr mob))
(l (cdr mob) (cdddr mob))
mob
```





Listless Fun

- What is the final value of z?

```
(define x '(1 2 3))
(define y '(4 5 6))
(define z (list (list (list x y)) x 7))
(set-cdr! (cddr x) (third z))
```

- `((((1 2 3 . 7) (4 5 6))) (1 2 3 . 7) 7)`

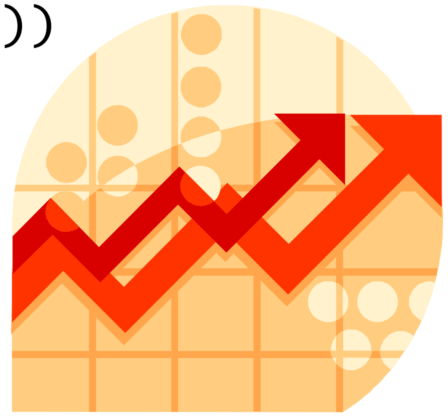
Orders of Growth

```
(define (find-e n)
  (if (= n 0)
      1.
      (+ (/ (fact n)) (find-e (- n 1)))))
```

- *Time: Θ(n^2)*
- *Space: Θ(n)*

```
(define (fast-expt x n)
  (cond ((= n 0)    1)
        ((even? n) (fast-expt (* x x) (/ n 2)))
        (else      (* x (fast-expt x (- n 1))))))
```

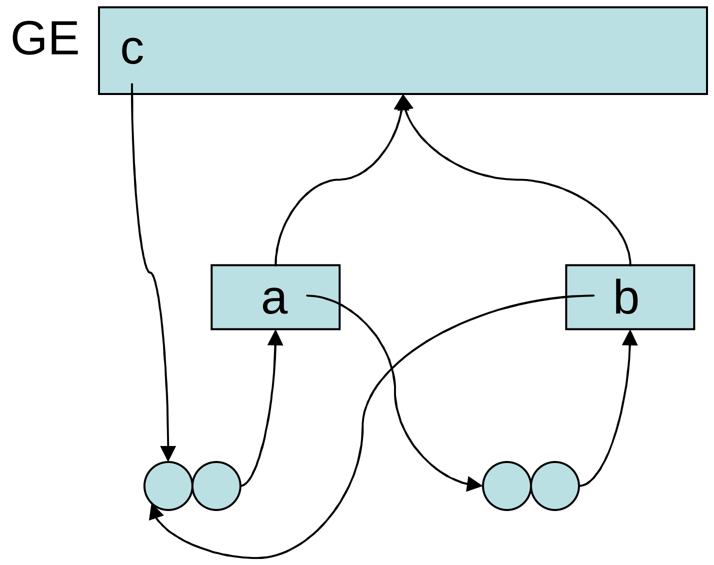
- *Time: Θ(log n)*
- *Space: Θ(log n)*

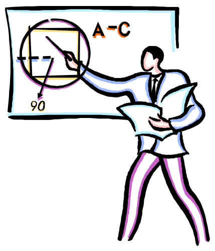


Environmental Trivia

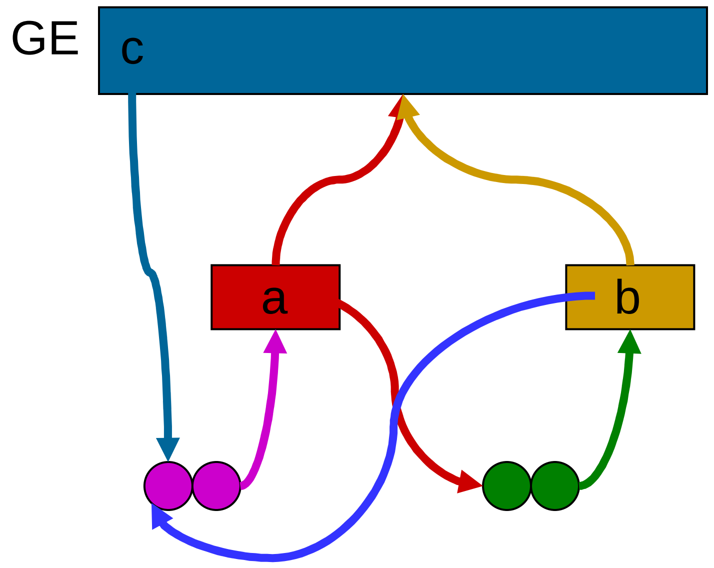
- To drop a frame…
  ```
  (let ((…) …) …)
  (proc …)
  ```
- To create a double-bubble…
  ```
  (let ((…) …) …)
  ```
  if desugaring
  ```
  (lambda (…) …)
  (define (foo …) …)
  ```
- Environments form what type of a graph (*e.g.* chain, tree, directed acyclic graph, general graph, …)?
  – Tree
- (Re)memorize the environment model!

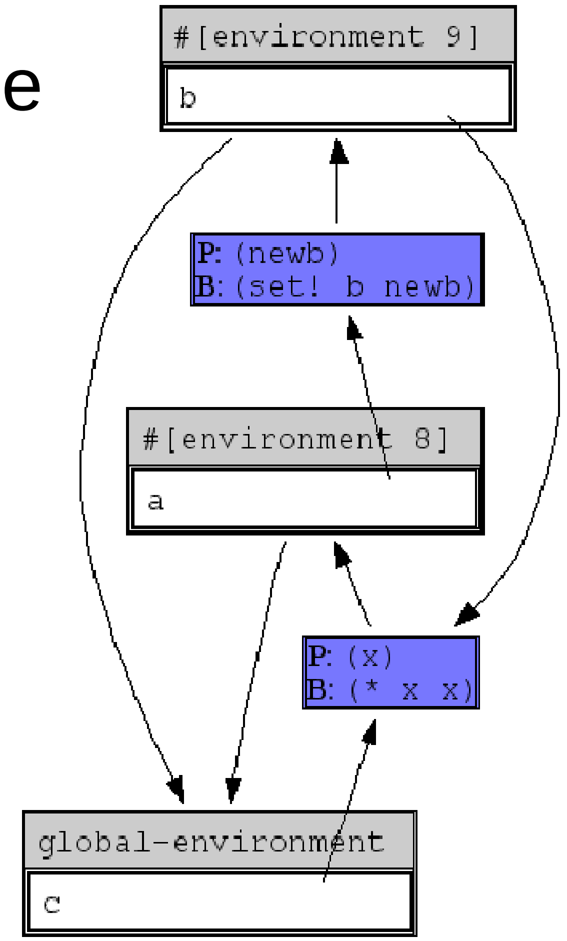
Write the Scheme Code to Create the Following Environment Diagram



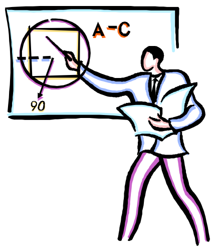


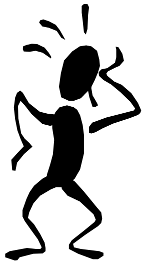
Write the Scheme Code to Create the Following Environment Diagram





```
(load "env-dia.scm")
(define c
  (let ((a (let ((b nil))
             (lambda (newb)
               (set! b newb)
               b))))
    (a (lambda (x) (* x x)))))
(env-dia 'render "tricky")
```





OOPs

- In the following code, how many references are created to the “anakin” symbol?

```
(define (walker self name speed)
  (let ((named-part (named-object self name))) …))
(define (biker self name speed)
  (let ((named-part (named-object self name))) …))
(define (swimmer self name speed)
  (let ((named-part (named-object self name))) …))
(define (tri-athlete self name walk-speed bike-speed swim-speed)
  (let ((walker-part (walker self name walk-speed))
        (biker-part (biker self name bike-speed))
        (swimmer-part (swimmer self name swim-speed))) …))
(define (jedi self name)
  (let ((tri-athlete-part (tri-athelete self name 20 50 15))) …))
(define anakin (create-jedi 'anakin))
```

- 8 (3 named-objects, walker, biker, swimmer, tri-athlete, jedi)
  - Moral: multiple personalities lead to Sithdom.

Meandering Streams

```
(define ones (cons-stream 1 ones))
(define ints
  (cons-stream 1 (add-streams ones ints)))

(define (row rnum col-stream)
  (if (null? col-stream) '()
      (cons-stream
       (list rnum (stream-car col-stream))
       (row rnum (stream-cdr col-stream)))))

(define (block started-rows next-row col-stream)
  (define (helper sr)
    (if (null? sr)
        (block (append
                (map stream-cdr started-rows)
                (list (row (stream-car next-row)
                           col-stream)))
               (stream-cdr next-row)
               col-stream)
        (cons-stream (stream-car (car sr))
                     (helper (cdr sr)))))
  (helper (reverse started-rows)))

(show-stream (block nil ints ints) 15)
```

What gets displayed by **show-stream**?

	1	2	3	4	…
1	(1 1)	(1 2)	(1 3)	(1 4)	…
2	(2 1)	(2 2)	(2 3)	(2 4)	…
3	(3 1)	(3 2)	(3 3)	(3 4)	…
4	(4 1)	(4 2)	(4 3)	(4 4)	…
…	…	…	…	…	…

(1 1)
(2 1) (1 2)
(3 1) (2 2) (1 3)
(4 1) (3 2) (2 3) (1 4)
(5 1) (4 2) (3 3) (2 4) (1 5)

Fall 1998 Exam: Interleavings

- What are the possible values for **z** at the completion of the parallel-execution below?

```
(define z 5)
(define (P1) (set! z (+ z 10)))
(define (P2) (set! z (* z 2)))
(parallel-execute P1 P2)
```

Fall 1998 Exam: Interleavings

```
        (+ z 10))
(set! z
                           (* z 2))
                   (set! z
→
```

```
                           (* z 2))
        (+ z 10))
(set! z
                   (set! z
→
```

```
        (+ z 10))
                           (* z 2))
(set! z
                   (set! z
→
```

```
                           (* z 2))
        (+ z 10))
                   (set! z
(set! z
→
```

```
        (+ z 10))
                           (* z 2))
                   (set! z
(set! z
→
```

```
                           (* z 2))
                   (set! z
        (+ z 10))
(set! z
→
```

foreach special form

```
(foreach var exp
  body-exp
  body-exp)
(foreach x (list 1 2 3 4 5)
  (display x)
  (display " "))
```

```
(define foreach-variable
   second)
(define foreach-list third)
(define foreach-body cdddr)
```

```
(define (desugar-foreach exp)
  `(let loop ((lst ,(foreach-list exp)))
     (if (null? lst)
         #f
         (let ((,(foreach-variable exp) (car lst)))
           ,@(foreach-body exp)
           (loop (cdr lst))))))
```

foreach special form

```
(foreach var exp
  body-exp
  body-exp)
(foreach x (list 1 2 3 4 5)
  (display x)
  (display " "))
```

```
(define foreach-variable
   second)
(define foreach-list third)
(define foreach-body cdddr)
```

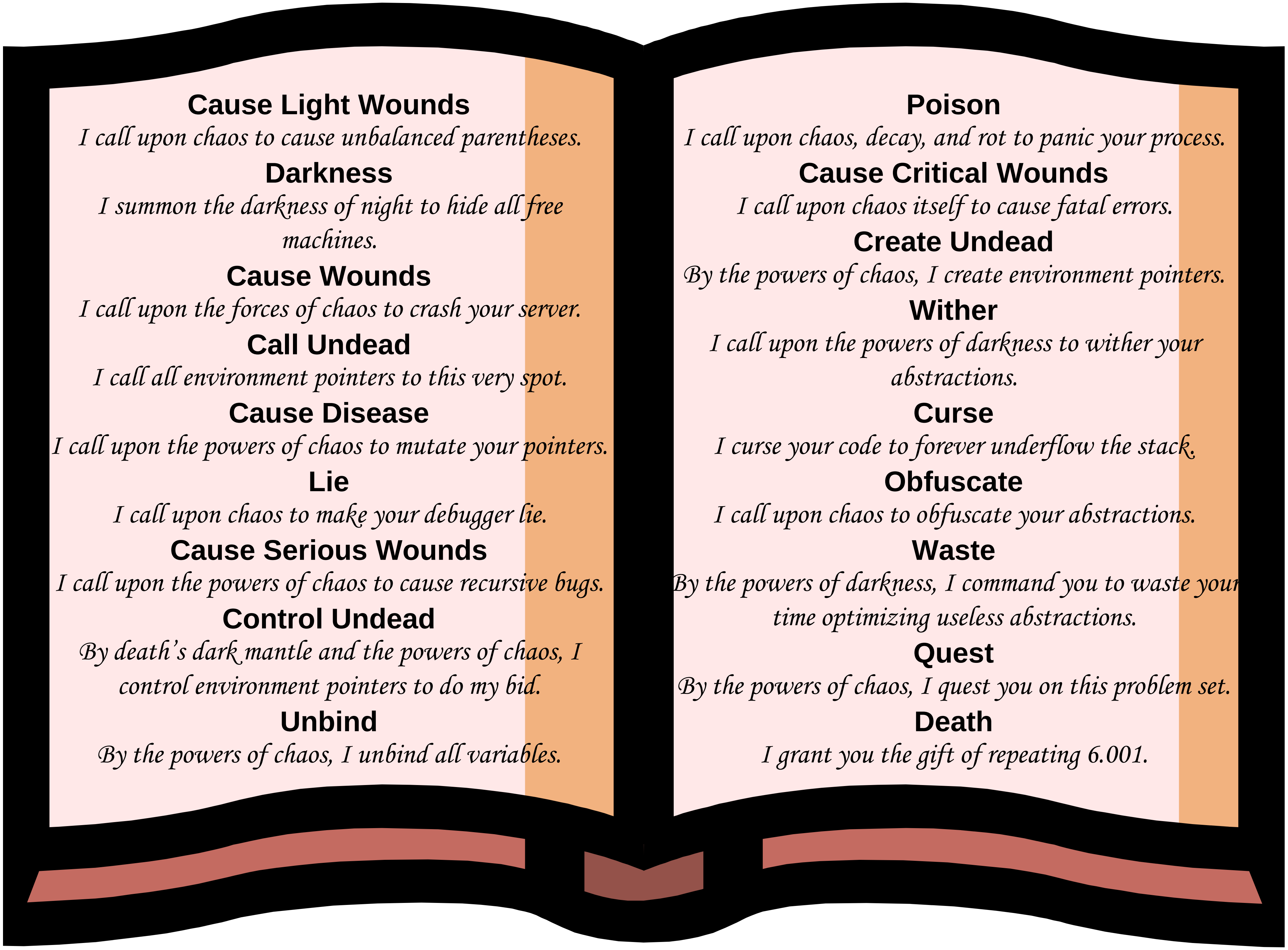
```
(define (eval-foreach exp env)
  (let loop ((lst (foreach-list exp)))
    (if (null? lst)
        #f
        (begin
          (m-eval `(let ((,(foreach-variable exp) ,(car lst)))
                          ,@(foreach-body exp))
                      env)
          (loop (cdr lst))))))
```

foreach special form

```
(foreach var exp
  body-exp
  body-exp)
(foreach x (list 1 2 3 4 5)
  (display x)
  (display " "))
```

```
(define foreach-variable
   second)
(define foreach-list third)
(define foreach-body cdddr)
```

```
(define (eval-foreach-nocapture exp env)
  (for-each (m-eval `(lambda (,(foreach-variable exp))
                       ,@(foreach-body exp)) env)
            (m-eval (foreach-list exp) env)))
```



Cause Light Wounds

I call upon chaos to cause unbalanced parentheses.

Darkness

I summon the darkness of night to hide all free machines.

Cause Wounds

I call upon the forces of chaos to crash your server.

Call Undead

I call all environment pointers to this very spot.

Cause Disease

I call upon the powers of chaos to mutate your pointers.

Lie

I call upon chaos to make your debugger lie.

Cause Serious Wounds

I call upon the powers of chaos to cause recursive bugs.

Control Undead

By death's dark mantle and the powers of chaos, I control environment pointers to do my bid.

Unbind

By the powers of chaos, I unbind all variables.

Poison

I call upon chaos, decay, and rot to panic your process.

Cause Critical Wounds

I call upon chaos itself to cause fatal errors.

Create Undead

By the powers of chaos, I create environment pointers.

Wither

I call upon the powers of darkness to wither your abstractions.

Curse

I curse your code to forever underflow the stack.

Obfuscate

I call upon chaos to obfuscate your abstractions.

Waste

By the powers of darkness, I command you to waste your time optimizing useless abstractions.

Quest

By the powers of chaos, I quest you on this problem set.

Death

I grant you the gift of repeating 6.001.