Repeat of Rules: Eval

- **name** - Look up *name* in the current environment, if found return value, otherwise lookup in enclosing (parent) environment.

- **(lambda (params) body)** - Create double bubble with code ptr to *params* and *body* and env ptr to current environment.

- **(define name value)** - Evaluate *value* and then create/replace binding for *name* with the result.

- **(set! name value)** - Evaluate *value* and then replace the first binding for *name* in the chain of environments, starting with the current env.

- **(proc args ... )** - Evaluate *proc* and *args* in the current environment, then apply.

- Otherwise – Follow the correct rule (numbers, if, cond, begin, quote, etc.)

Repeat of Rules: Apply

- Step 1 - Drop a new frame

- Step 2 - Link frame pointer of new frame to environment pointed to by env pointer of double bubble being applied.

- Step 3 - Bind *params* of double bubble in the new frame.

- Step 4 - Eval the *body* in the new frame.
Three Counter Attempts

1. (define make-count-proc-1
   (lambda (f)
     (lambda (x)
       (let ((count 0))
         (cond ((eq? x 'count) count)
               (else
                (set! count (+ count 1))
                (f x))))))

(define sqrt-c-1
 (make-count-proc-1 sqrt))

(sqrt-c-1 4)
(sqrt-c-1 'count)

This counter won’t work: Each time sqrt is called within make-count-proc-1 the counter value count will be reset to 0. So passing the message 'count will always return 0 regardless of how many times sqrt-c-1 has been called.
2. (define make-count-proc-2
   (lambda (f)
     (let ((count 0))
       (lambda (x)
         (cond ((eq? x 'count) count)
               (else (set! count (+ count 1))
                   (f x)))))))

(define sqrt-c-2
  (make-count-proc-2 sqrt))
(define sqr-c-2
  (make-count-proc-2 square))

(sqrt-c-2 4) ==> 2
(sqrt-c-2 'count) ==> 1

(sqr-c-2 4) ==> 16
(sqr-c-2 'count) ==> 1

This version will result in independent counters for sqrt-c-2 and sqr-c-2. Each time make-count-proc-2 is applied.
3. (define make-count-proc-3
  (let ((count 0))
    (lambda (f)
      (lambda (x)
        (cond ((eq? x 'count) count)
              (else
               (set! count (+ count 1))
               (f x)))))))

(define sqrt-c-3
  (make-count-proc-3 sqrt))
(define sqr-c-3
  (make-count-proc-3 square))

(sqrt-c-3 4) ==> 2
(sqrt-c-3 'count) ==> 1

(sqr-c-3 4) ==> 16
(sqr-c-3 'count) ==> 2

In this version, there is only a single counter for any procedure that’s been composed with make-count-proc-3. The 'count message now returns the total number of times that either sqrt-c-3 or sqr-c-3 has been applied to a number.
4. The procedure `last-pair` returns the last pair of a list (guaranteed to have `()` in the cdr).

```
(define (list-inserters lst)
  (let ((last (last-pair lst)))
    (list (lambda (x)
      (set-cdr! lst (cons x (cdr lst)))
      lst)
    (lambda (y)
      (set-cdr! last (cons y '()))
      (set! last (cdr last))
      lst))))

(define the-list (list 1 3 4))

(let ((ins (list-inserters the-list)))
  ((list-ref ins 0) 2)
  ((list-ref ins 1) 5))
```

Finish the environment diagram.