Environment Model

1. To evaluate a combination: evaluate subexpressions then **apply** value of operator subexpression to values of operand subexpressions.

2. Value of a variable w.r.t. an environment is the value given by the binding of the variable in the first frame in the environment that contains such a binding.

3. A lambda expression produces a procedure object:
   - **code** (parameters and body) are given by the text of the lambda and are stored away for later use
   - **environment pointer** points to the environment in which the lambda expression was evaluated

4. **`Define`** adds a binding to the current frame

5. To **apply** a procedure object to a set of arguments:
   - Create a new frame
   - Hang the frame from the environment part of the procedure object being applied
   - In the new frame, bind the formal parameters of the procedure to the actual arguments
   - Evaluate the body of the procedure in the context of the new environment

6. To evaluate **`(set! <var> <exp>)`** w.r.t. an environment E:
   - Evaluate **`<exp>`** w.r.t. E
   - Find and change the nearest binding for **`<var>`** in E to hold value of **`<exp>`**

Use and Mis-Use of Set!

```
;; Functional programming style
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

```
;; Imperative programming style -- DEPRECATED
(define (factorial n)
  (let ((product 1)
        (counter 1))
    (define (iter)
      (if (> counter n)
          product
          (begin (set! product (* counter product))
                 (set! counter (+ counter 1))
                 (iter))))
    (iter)))
```

```
;; Imperative programming style -- BUGGY!
(define (factorial n)
  (let ((product 1)
        (counter 1))
    (define (iter)
      (if (> counter n)
          product
          (begin (set! counter (+ counter 1))
                 (set! product (* counter product))
                 (iter))))
    (iter)))
```

Implications of Mutation

- Must worry about **time** and **change**: order of evaluation matters!
- Variables no longer stand for values
  - Become places whose contents may change
- Must worry about **identity**: lose referential transparency
- Natural for modeling objects/systems with **state**

Message-Passing Ship Implementation

```
(define (make-ship x-pos y-pos time-left)
  (define (move dx dy)
    (set! x-pos (+ x-pos dx))
    (set! y-pos (+ y-pos dy))
    (list x-pos y-pos))
  (define (count-down)
    (set! time-left (- time-left 1))
    (if (<= time-left 0)
        ’blast-off
        time-left))
  (define (dispatch message)
    (cond ((eq? message ’move) move)
          ((eq? message ’count-down) count-down)
          (else (error "No method" message))))
  dispatch)


(define enterprise (make-spaceship 0 0 10))


((enterprise ’move) 1 2) ==> (1 2)
```

Data-Directed Ship Implementation

```
(define (install-ship-package)
  ;; Internal representation
  (define (make-ship x y time) (list x y time))
  ; Accessors
  (define (ship-x ship) (car ship))
  (define (ship-y ship) (cadr ship))
  ; Mutators
  (define (set-ship-x! ship new-x)
    (set-car! ship new-x))
  (define (set-ship-y! ship new-y)
    (set-car! (cdr ship) new-y))
  ; Operations
  (define (move ship dx dy)
    (set-ship-x! (+ (ship-x ship) dx))
    (set-ship-y! (+ (ship-y ship) dy))
    (list (ship-x ship) (ship-y ship)))

  ;; External representation - tagged object
  (define (tag x) (attach-tag ’spaceship x))
  (put ’make ’spaceship
    (lambda (x y t) (tag (make-ship x y t))))
  (put ’move ’spaceship
    (lambda (s dx dy) (tag (move s dx dy))))
  ’done
  )

(define (move obj dx dy)
  (apply-generic ’move obj dx dy)
```