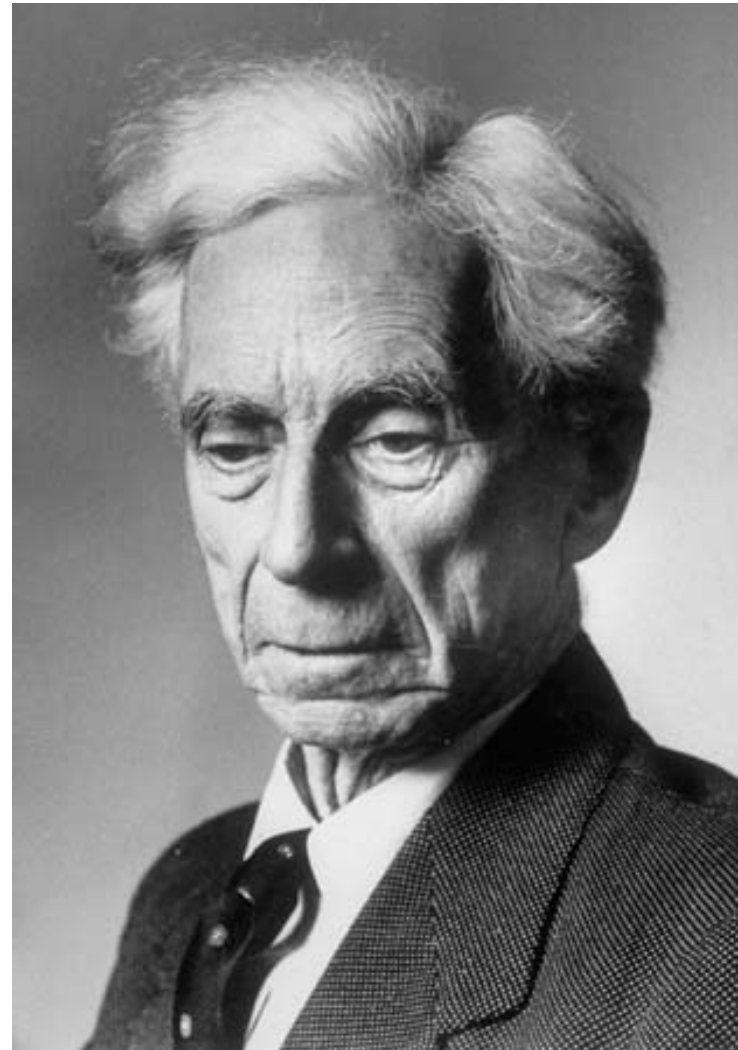


Turtles All The Way Down

Bertrand Russell had just finished giving a public lecture on the nature of the universe. An old woman said "Prof. Russell, it is well known that the earth rests on the back of four elephants, who stand on the back of a giant turtle." Russell replied, "Madame, what does the turtle stand on?" The woman replied, "That's easy. It's turtles all the way down."



Semantics of Programming Languages

- How do we define the meaning of computer programs?
 - Denotational
 - Construct mathematical objects which formally define the meaning of programs.
 - Operational
 - Translate the programs into a language with well defined semantics.

Semantics of Programming Languages


- How do we define the meaning of computer programs?
 - Denotational
 - Construct mathematical objects which formally define the meaning of programs.
 - Operational
 - Translate the programs into a language with well defined semantics.
 - *Metacircular*
 - *Write an interpreter for programs written in the same language as the programs themselves.*

assoc

```
(define assoc
  (lambda (key alist)
    (if (null? alist)
        #f
        (if (eq? (car (car alist)) key)
            (car alist)
            (assoc key (cdr alist))))))
```

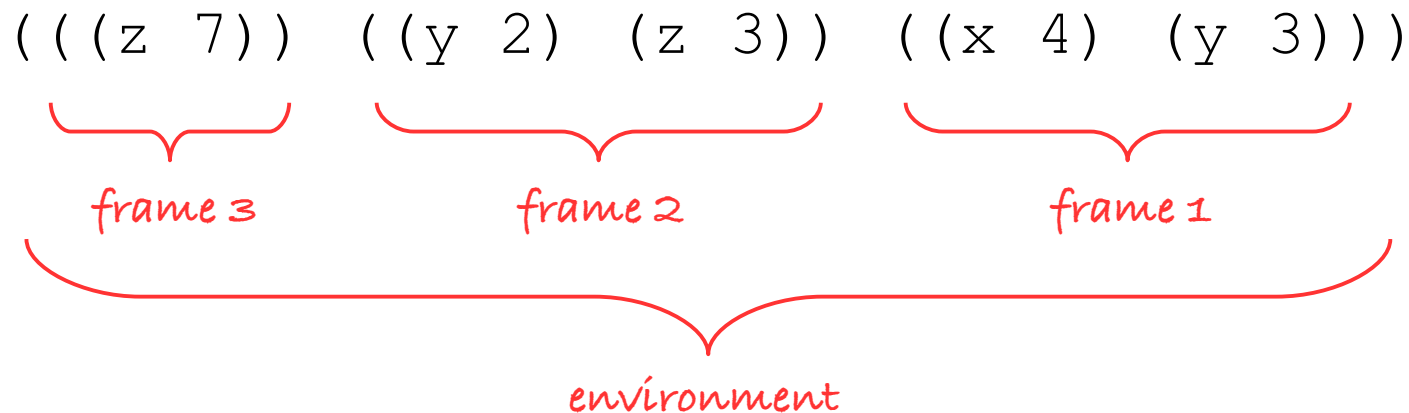
```
> (let ((x 2) (y 3)) y)
3
```

```
> (assoc 'y ' ((x 2) (y 3)))
(y 3)
```



Environments and Lexical Scope

```
(let ((x 4) (y 3))  
  (let ((y 2) (z 3))  
    (let ((z 7))  
      (+ x y z))))
```



lookup

```
(define lookup
  (lambda (var env)
    ((lambda (val) □ used instead of let
      (if val
          (car (cdr val))
          (lookup var (cdr env))))
      (assoc var (car env))))))
```

```
> (let* ((x 2) (y 3)) (let ((x 4)) y))
3
```

```
> (lookup 'y '(((x 4)) ((x 2) (y 3))))
3
```

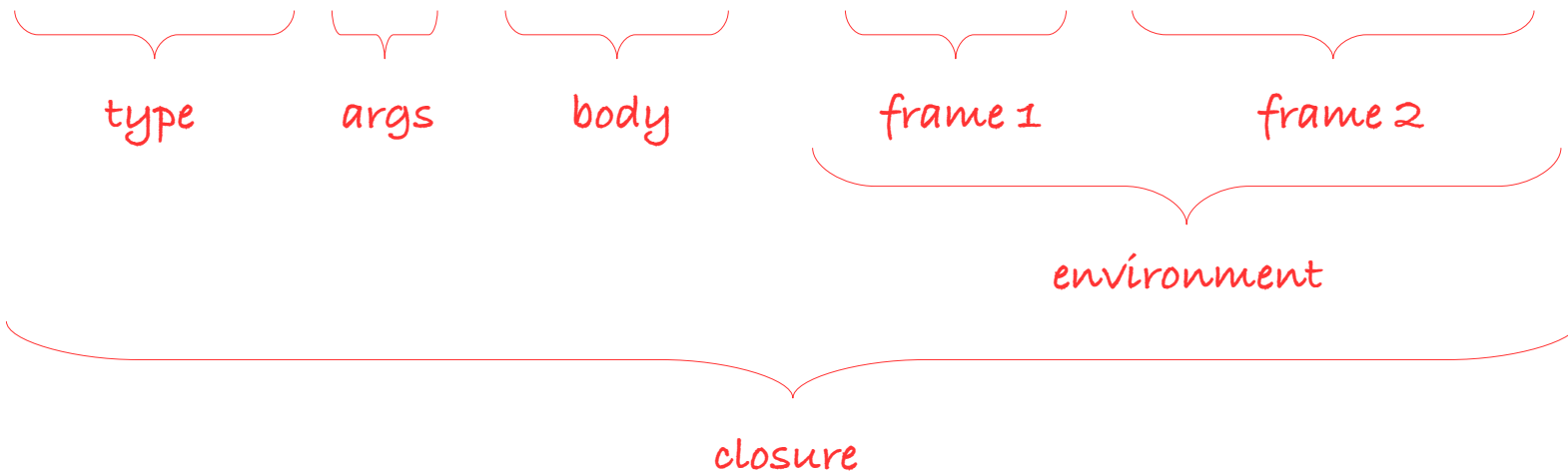
frame 2 *frame 1*

environment

Function Objects a.k.a. "Closures"

```
(define make-closure
  (lambda (args body env)
    (cons (quote closure)
          (cons args
                (cons body (cons env (quote ())))))))
```

```
> (let ((y 3)) (lambda (x) (+ x y)))
(closure (x) (+ x y) (((y 3)) global-env))
```



A Recursive Function

```
(define fact
  (lambda (n)
    (if (= n 0)
        1
        (* n (fact (- n 1))))))
```

CLOSURE

arguments

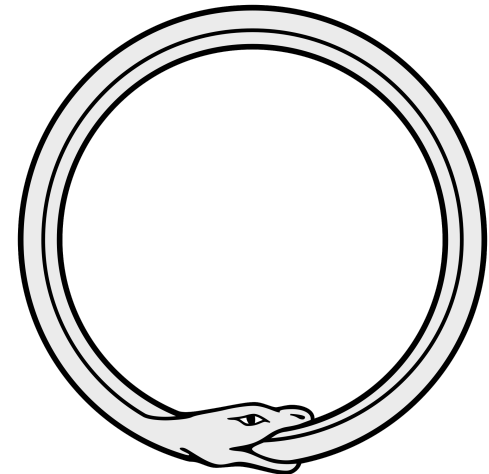
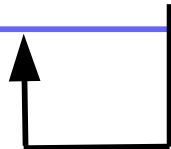
(n)

body

(if (= n 0) 1 (* n (fact (- n 1))))

environment

((fact) ...)

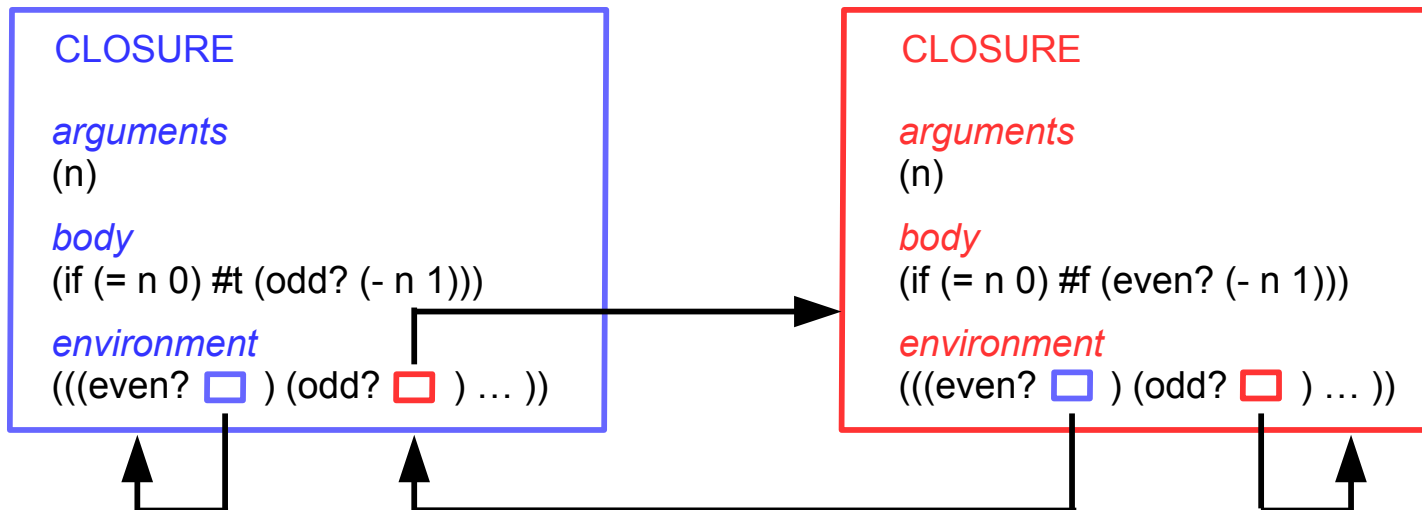
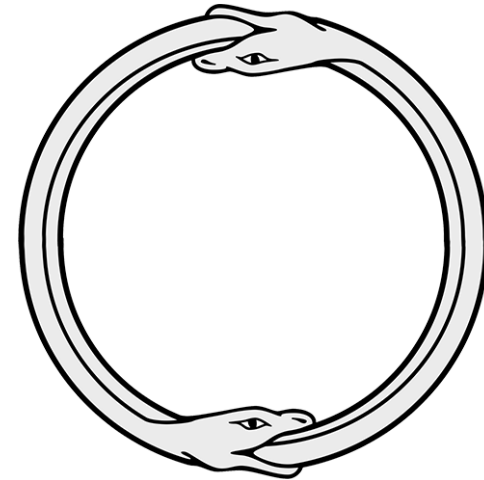


ouroboros

Mutually Recursive Functions

```
(define odd?  
  (lambda (n)  
    (if (= n 0)  
        #f  
        (even? (- n 1)))))
```

```
(define even?  
  (lambda (n)  
    (if (= n 0)  
        #t  
        (odd? (- n 1)))))
```

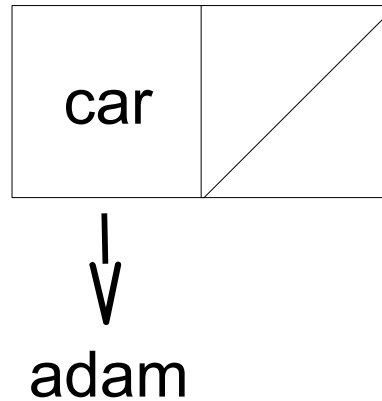


Functional Programs Can Only Make Trees

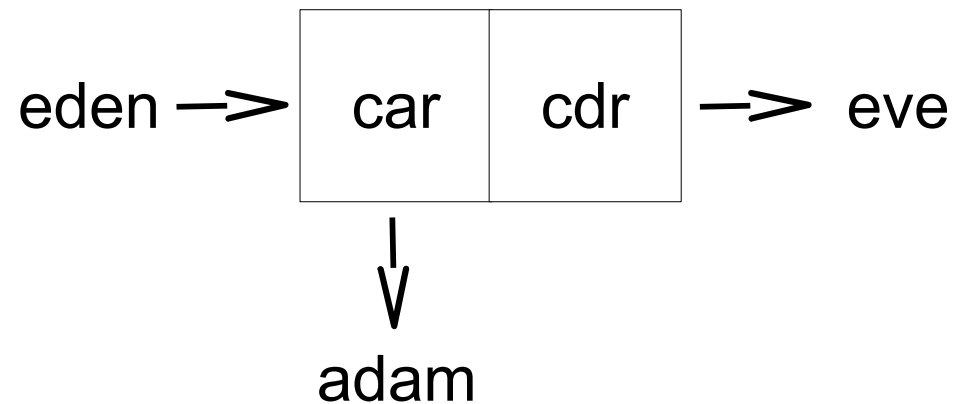


Box and Pointer Notation

```
> (cons 'adam '())  
(adam)
```

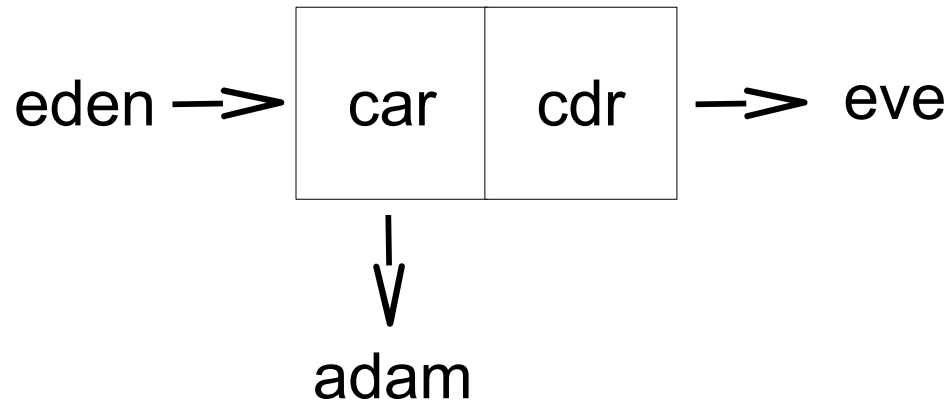


```
> (define eden (cons 'adam 'eve))  
(adam . eve)
```

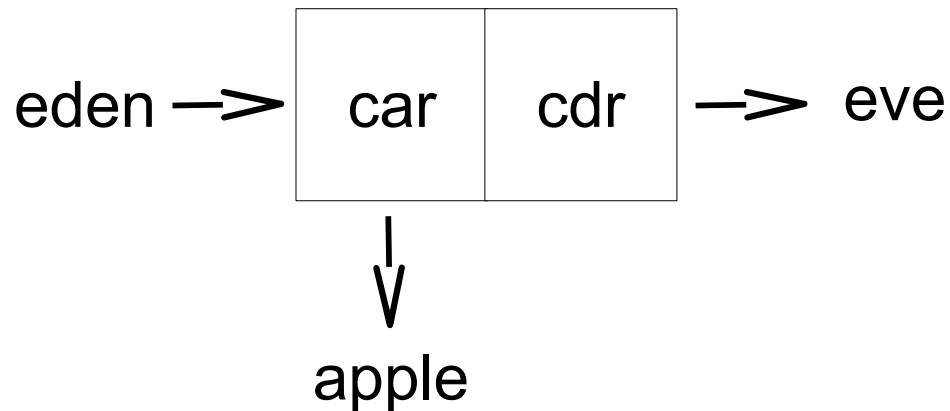


set-car!

```
> eden  
(adam . eve)
```



```
> (set-car! eden 'apple)  
(apple . eve)
```



Procedures With Side Effects

```
> (display "Hello World")  
Hello World
```

```
> (define x 7)
```

```
> x
```

```
7
```

```
> (set! x 9)
```

```
> x
```

```
9
```

```
> (define ls '(1 2 3 4))
```

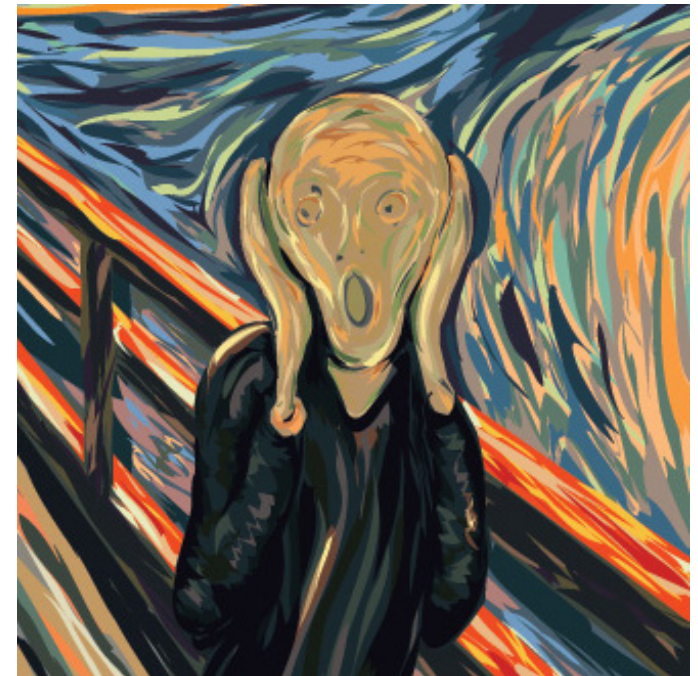
```
> ls
```

```
(1 2 3 4)
```

```
> (set-cdr! ls 0)
```

```
> ls
```

```
(1 . 0)
```



Sequencing Effects

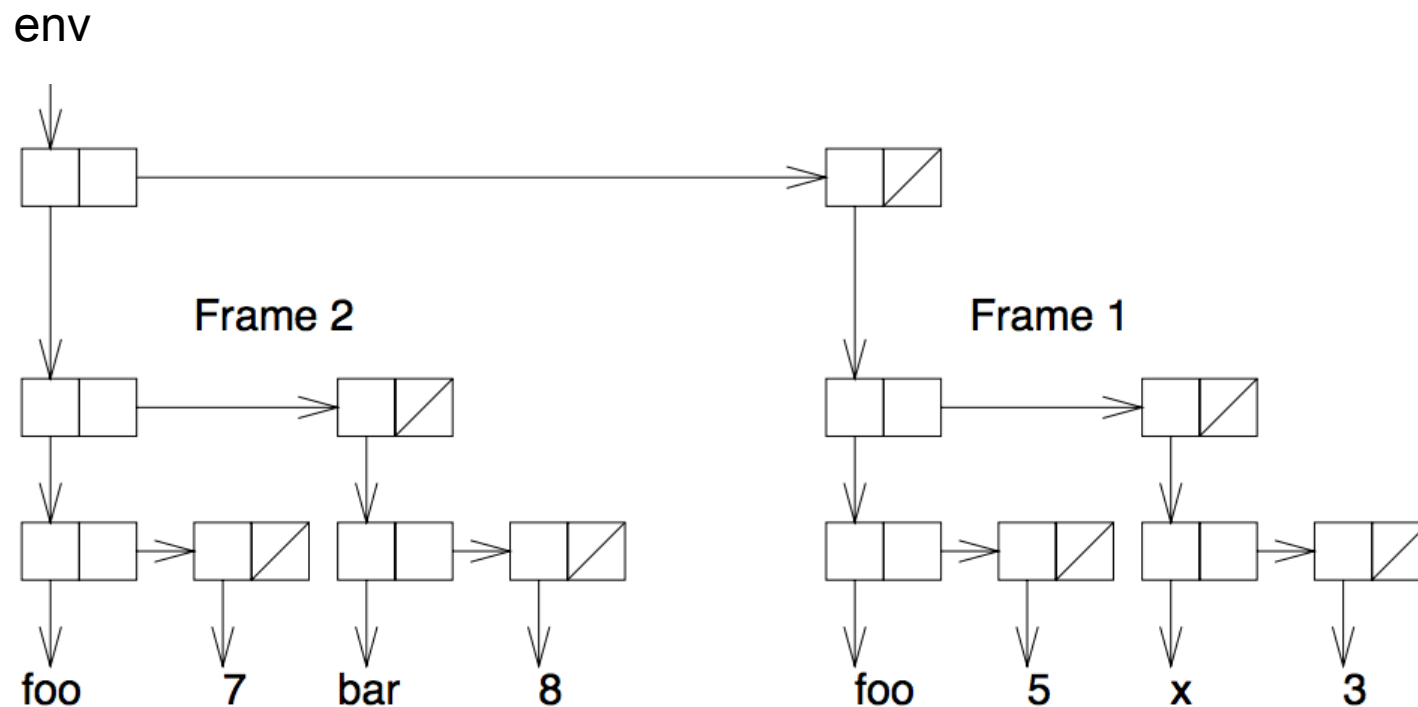


```
(define bonjour
  (lambda (name)
    (if (equal? name "Napoleon")
        (begin
          (display "Vive L'Empereur!")
          (newline))
        (begin
          (display "Bonjour ")
          (display name)
          (display ".")
          (newline))))))
```

```
> (bonjour "Napoleon")
Vive L'Empereur!
```

```
>
```

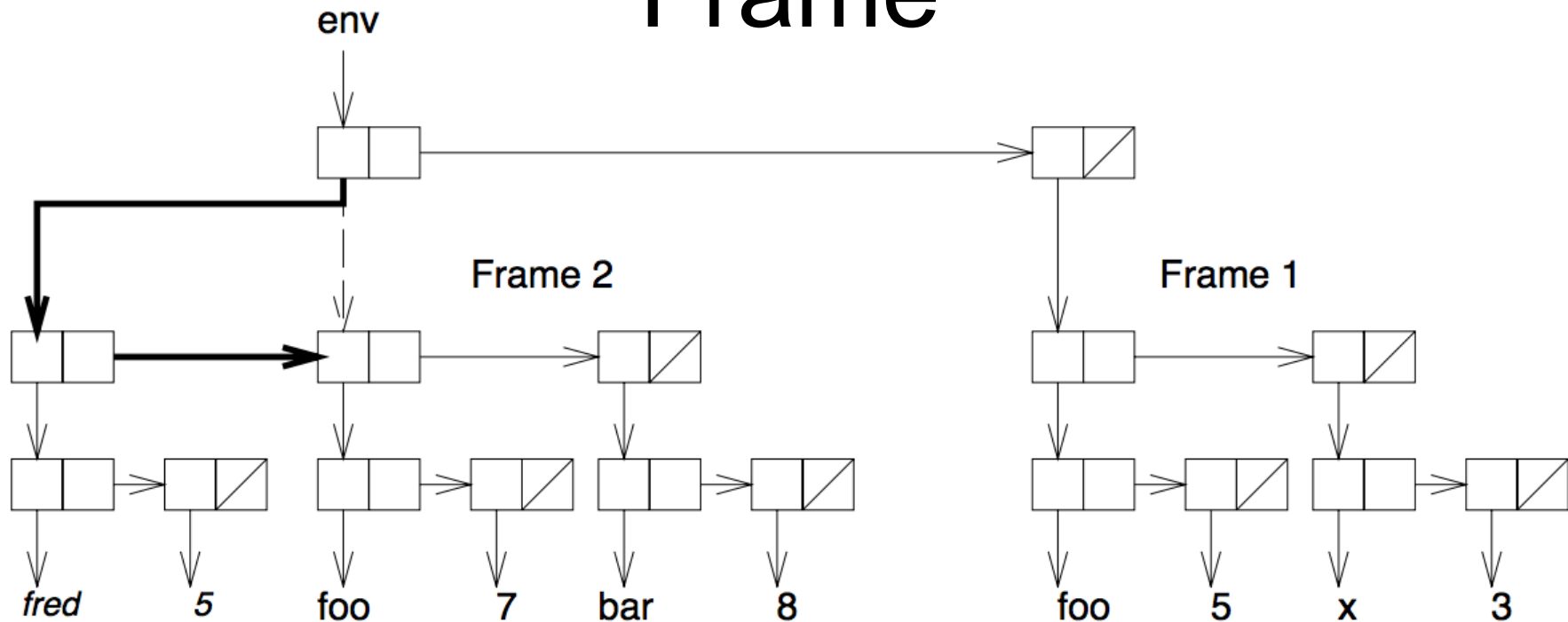
Environment in Box and Pointer Notation



`(((foo 7) (bar 8)) ((foo 5) (x 3)))`

frame 2 *frame 1*

Adding a Definition to the Front Frame

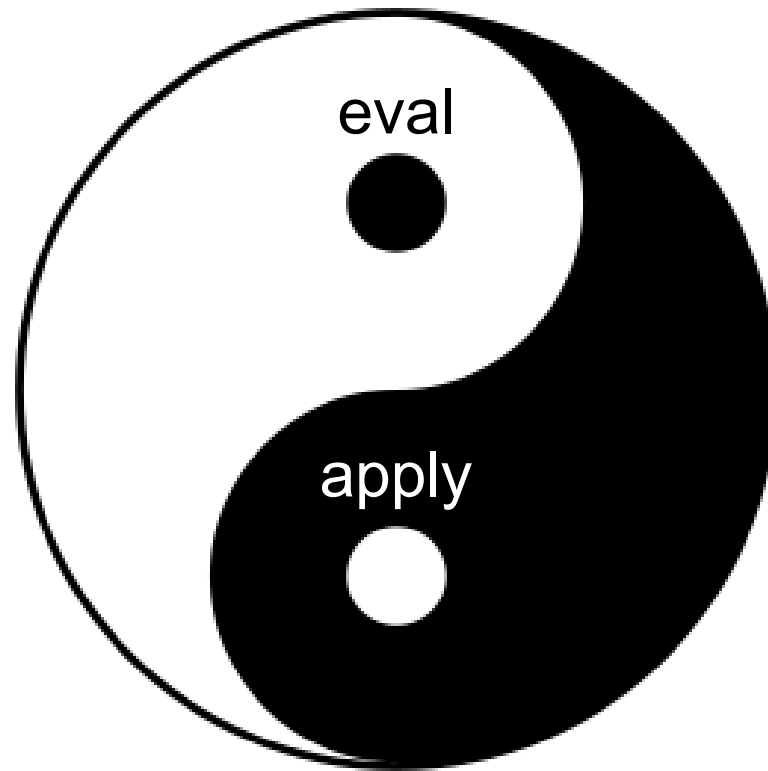


```
(define add-defn!
  (lambda (var val env)
    (set-car! env
      (cons (cons var (cons val (quote ()))) (car env)))
    (quote ok)))
```

```
> (add-defn! 'fred 5 '(((foo 7) (bar 8)) ((foo 5) (x 3))))
ok
```


Mutual Recursion Between Eval and Apply

Expressions are evaluated by applying functions to values...



...functions are applied to values by evaluating expressions in environments.

Primitive Functions

```
(define global-env
  (cons (cons (cons (quote car) (cons car (quote ())))
             (cons (cons (quote cdr) (cons cdr (quote ())))
                   (cons (cons (quote cons) (cons cons (quote ())))
                           (cons (cons (quote set-car!) (cons set-car! (quote ())))
                                   (cons (cons (quote null?) (cons null? (quote ())))
                                           (cons (cons (quote eq?) (cons eq? (quote ())))
                                                 (cons (cons (quote apply) (cons apply (quote ())))
                                                         (cons (cons (quote symbol?) (cons symbol? (quote ())))
                                                                 (cons (cons (quote pair?) (cons pair? (quote ())))
                                                                 (quote ())))))))))))))
  (quote ()))
```

minimum necessary to define eval

make-frame

```
(define make-frame
  (lambda (vars vals)
    (if (null? vars)
        (quote ())
        (cons (cons (car vars)
                    (cons (car vals) (quote ())))
              (make-frame (cdr vars) (cdr vals))))))
```

```
> (make-frame '(x y) '(1 2))
((x 1) (y 2))
```



eval-list and sequence

```
(define eval-list map eval
  (lambda (ls env)
    (if (null? ls)
        (quote ())
        (cons (eval (car ls) env)
              (eval-list (cdr ls) env))))))
```

```
(define sequence map eval for effect and return last
  (lambda (ls env)
    (if (null? (cdr ls))
        (eval (car ls) env)
        (begin
          (eval (car ls) env)
          (sequence (cdr ls) env))))))
```

Two More Helper Functions

```
(define self-evaluating?  
  (lambda (sexpr)  
    (if (pair? sexpr)  
        (if (eq? (car sexpr) (quote closure)) #t #f)  
        (if (symbol? sexpr) #f #t))))
```

```
(define apply-function  
  (lambda (proc vals)  
    (if (pair? proc) closure-body  
        (sequence closure-args  
                  (car (cdr (cdr proc)))  
                  (cons (make-frame (car (cdr proc)) vals)  
                        (car (cdr (cdr (cdr proc))))))  
                  (apply proc vals)))  
        (primitive function closure-env)))
```

eval

```
(define eval
  (lambda (sexpr env)
    (if (self-evaluating? sexpr)
        sexpr
        (if (symbol? sexpr)
            (lookup sexpr env)
            ((lambda (first)
              (if (eq? first (quote quote))
                  (car (cdr sexpr))
                  (if (eq? first (quote define))
                      (add-defn! (car (cdr sexpr))
                                  (eval (car (cdr (cdr sexpr))) env)
                                  env)
                      (if (eq? first (quote if))
                          (if (eval (car (cdr sexpr)) env)
                              (eval (car (cdr (cdr sexpr))) env)
                              (eval (car (cdr (cdr (cdr sexpr)))) env))
                          (if (eq? first (quote lambda))
                              (make-closure (car (cdr sexpr))
                                              (cdr (cdr sexpr))
                                              env)
                              (if (eq? first (quote begin))
                                  (sequence (cdr sexpr) env)
                                  (apply-function (eval (car sexpr) env)
                                                  (eval-list (cdr sexpr) env))))))))))
        (car sexpr))))))
```

self-evaluating

symbol

quote

define

if

lambda

begin

special-forms

CALL ME WEAK-MINDED FOR BELIEVING
THE WORLD SITS ON A TURTLE,



AND THAT TURTLE IS
SITTING ON ANOTHER
TURTLE,

AND IT'S TURTLES
ALL THE WAY
DOWN,

BUT DON'T TELL ME
THE UNEXAMINED
LIFE ISN'T
WORTH
LIVING

(Socrates)

AS I RIDE MY INFINITE TURTLE
FORTRESS ACROSS THE SKY.