

Environmental model of evaluation

- Now that the values of variables can be changed, the substitution model of evaluation is no longer adequate
- Value of a formal parameter cannot be substituted everywhere in the body of a procedure definition because that may not be the value of the formal parameter everywhere in the procedure definition

Substitution model fails

Example: `(quadratic a b c x)` computes $a*x*x + b*x + c$

```
(define (quadratic a b c x)
  (set! a (* a x))
  (set! a (* (+ a b) x))
  (set! a (+ a c))
  a)
```

New interpretation of variables

- We need to interpret variables as names of places in memory where values are stored
- Data structures for storing variables and their values are called **environments**
- An environment is a linked list of **frames**
- A frame is a table of **bindings**, that is, a table of variable names and their associated values

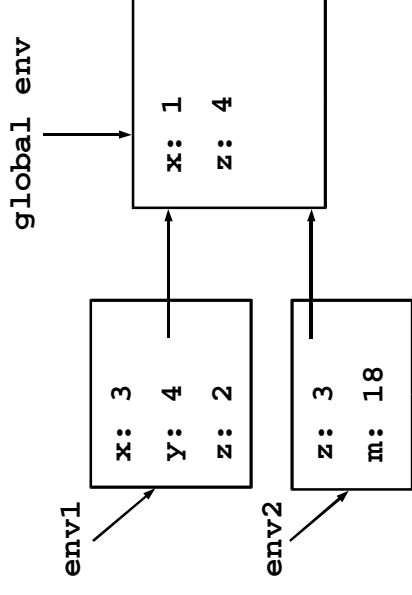
All environments share the same last frame

- The last frame of every environment is the same frame, which is called the **global environment**
- To find the value of a variable in an environment, the list of frames is searched until one is found that contains that variable name
- The value of that variable in that frame is the value of the variable in the environment

Shadowing

- The binding of a variable in the first frame in which the variable name occurs **shadows** any binding of that variable in later frames in the linked list
- The bindings that are shadowed are not visible in the environment

Three environments



Rules for evaluation

- 1) Numbers, strings evaluate to themselves
- 2) Variables (symbols) evaluate to their values in the current environment
- 3) Combinations (lists) are evaluated as calls to normal procedures (arguments are evaluated) or special forms (arguments are not evaluated)

Evaluation of normal procedures

- 1) Make new frame containing formal parameters of procedure
- 2) Assign argument values to corresponding formal parameters in frame
- 3) Link frame to the environment in which procedure was created
- 4) Evaluate body in the environment that begins with this new frame

Evaluation of lambda expressions

- Creates a procedure
- The procedure has two parts
 - text (parameter list and body) of the lambda expression
 - pointer to the environment in which the procedure is being created (the environment in which the lambda expression is being evaluated)

Evaluation of define

- The symbol (variable or procedure name) that is being defined, and its value, are added to the first frame of the current environment
- `(define (procname ...) ...)` is treated the same as `(define procname (lambda (...)))`

Evaluation of set!

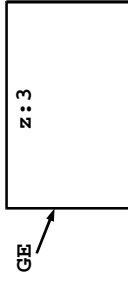
Starting with the first frame of the current environment, set! searches until it finds the variable it is changing and then changes the value of that variable in that frame to the new value

To illustrate:

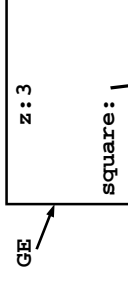
Consider what happens when the following sequence of expressions is evaluated in an empty global environment (GE)

```
(define z 3)
(define (square x) (* x x))
(square z)
(define (poly2 a b c x)
  (+ (* a (square x)) (* b x) c))
(poly2 1 2 3 (+ z 1))
```

(define z 3)

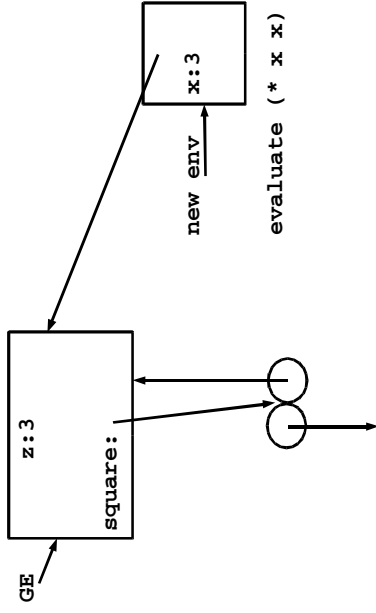


(define (square x) ...)



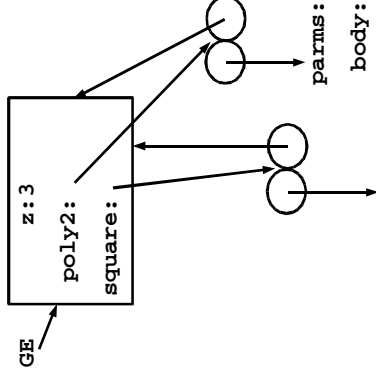
params: x
body: (* x x)

(square z)



params: x
body: (* x x)

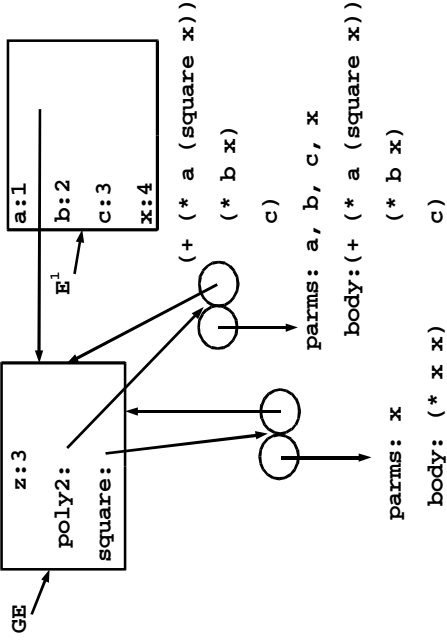
(define (poly2 ...) ...)



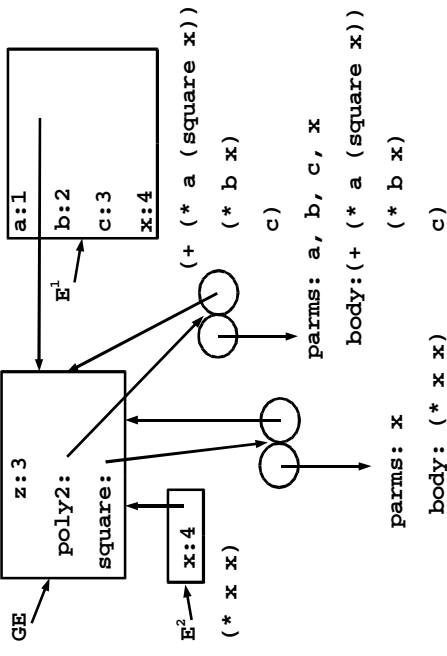
params: a, b, c, x
body: (+ (* a (square x))
 (* b x)
 c)

params: x
body: (* x x)

`(poly2 1 2 3 (+ z 1))`



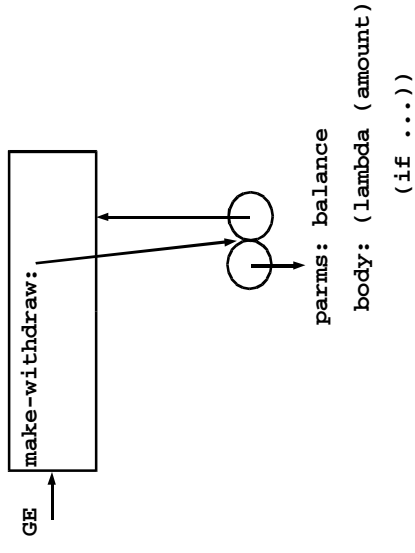
Evaluating (square x)



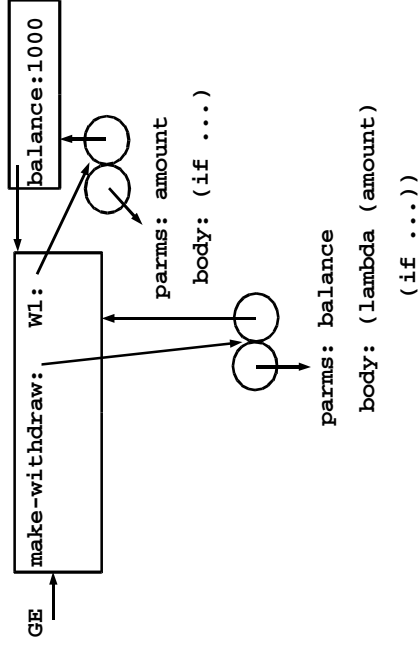
Local variables in frames

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin
          (set! balance (- balance amount))
          balance)
        "insufficient funds")))
(define W1 (make-withdraw 1000))
(W1 30)
(define W2 (make-withdraw 1200))
```

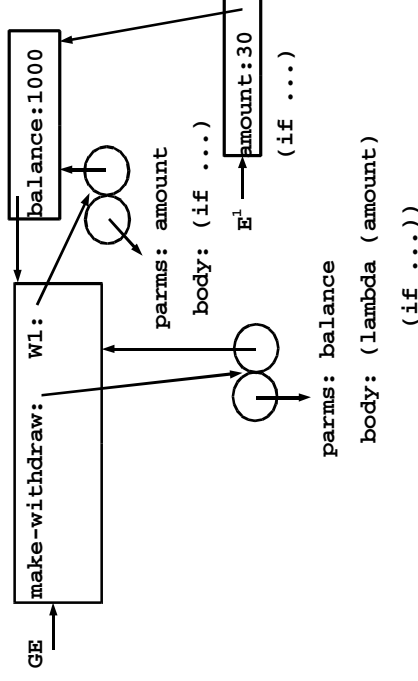
(define (make-withdraw...))



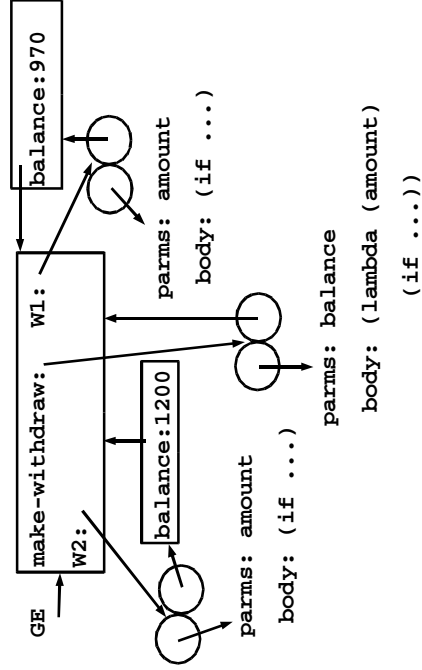
(define w1 ...)



(w1 30)



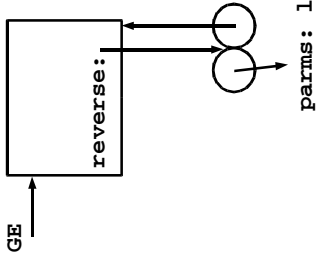
(define w2 ...)



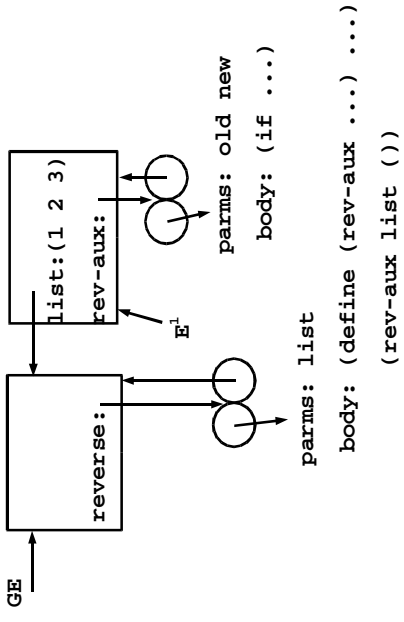
Nested definitions

```
(define (reverse list)
  (define (rev-aux old new)
    (if (null? old)
        new
        (rev-aux (cdr old)
                  (cons (car old) new))))
  (rev-aux list ()))
(reverse '(1 2 3))
```

(define (reverse ...



(reverse '(1 2 3))



Evaluating (rev-aux ...)

