

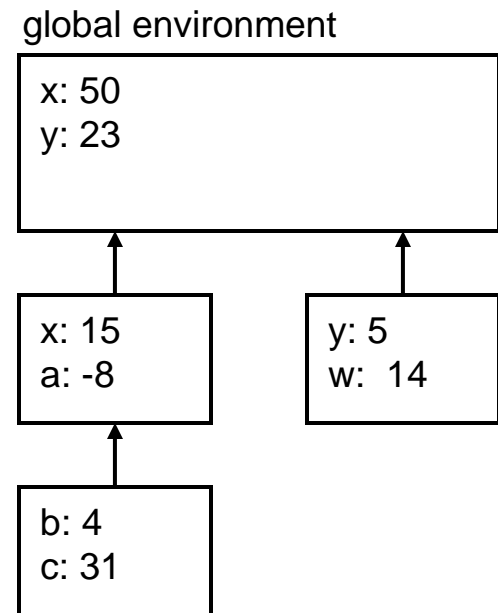


CS1 Recitation 7

Fall 2008

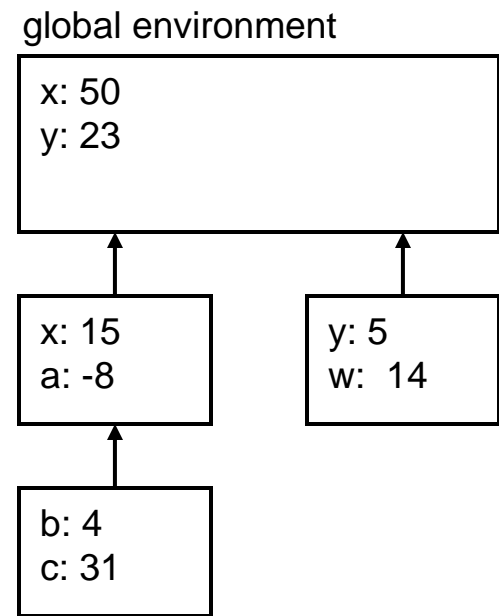
Environment Model

- Frames are collections of bindings
 - A “binding” associates a name and a value
- The global environment is the top-most frame
 - All other frames have a parent frame
 - Called the “enclosing environment”
- An environment starts with a particular frame
 - It includes all frames from that frame to the global environment



Rules for Environment Model

- All expressions are evaluated in the context of an environment
 - The environment for an operation starts with a particular frame and extends to the global environment
- Most important detail is to identify the environment associated with each expression
 - Follow the environment model rules unswervingly!



Evaluation Rules 1, 2

1. **define** creates a new binding in the current frame
 - The current environment starts with the current frame
2. **set!** changes an existing binding in the current environment
 - Frequently occurs in a different frame from the current frame
 - **set!** *never ever* creates a new binding!

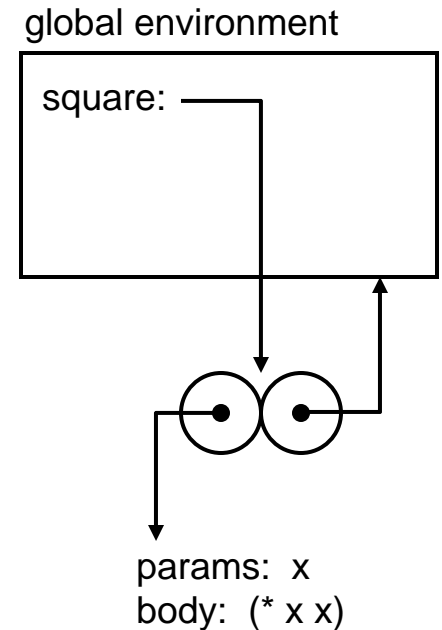
Evaluation Rule 3

3. Evaluating lambda expressions creates a pair

- *...not a cons pair...*
- Left arrow points to:
 - Parameter list
 - Body of procedure
- Right arrow points to:
 - Environment (frame) where lambda was created

■ Example:

- `(define (square x) (* x x))`
- `(define square (lambda (x) (* x x)))`



Evaluation Rule 4

- When a lambda expression is *applied*:
 - A new frame is created
 - Values of arguments are bound in new frame
 - Parent of new frame is the frame that the lambda expression points to
 - “Current environment” changes to the new frame, and body of lambda is evaluated in that context
- Most important detail:
 - Operands are evaluated first, *before* this step occurs!
 - Operands are not evaluated in the context of the new frame – it doesn’t exist when they are evaluated.

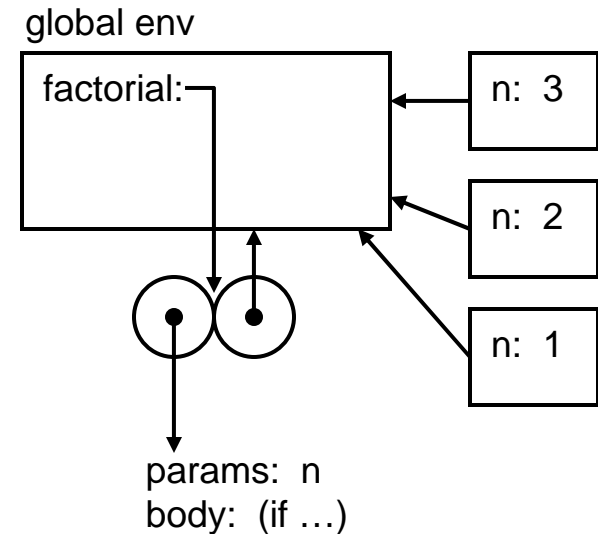
Simple Example

- Recursive function:

```
(define (factorial n)
  (if (< n 2)
      n
      (* n (factorial (- n 1)))))
(factorial 3)
```

- Main point:

- When a procedure is called, the new frame points to the same parent that the procedure points to!
- Don't get tripped up by recursive function invocations



Passing an Unnamed Procedure

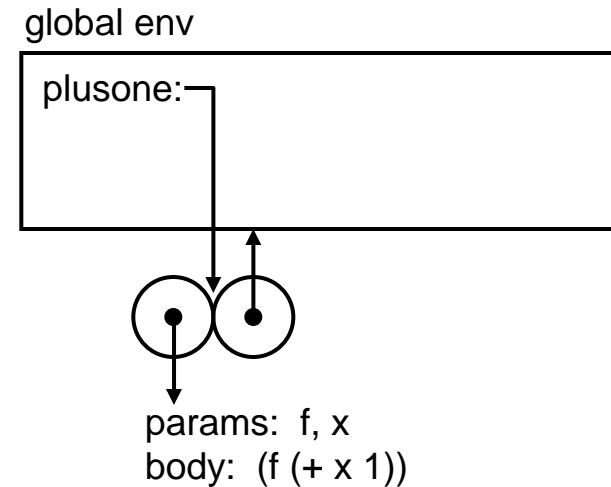
■ Example:

```
(define (plusone f x)
  (f (+ x 1))
  (plusone (lambda (x) (* 2 x)) 10))
```

■ Desugar to clarify:

```
(define plusone
  (lambda (f x)
    (f (+ x 1))))
```

□ Easy to diagram...



Passing Unnamed Procedure (2)

- Second part:

(plusone (lambda (x) (* 2 x)) 10)

- Evaluate operands first, just like substitution model

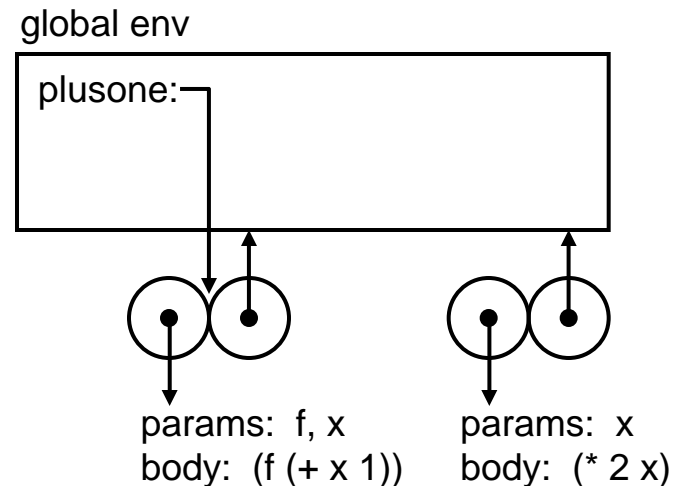
- Create lambda expression...

- Current environment is *still* the global environment

- Procedure points to global env.

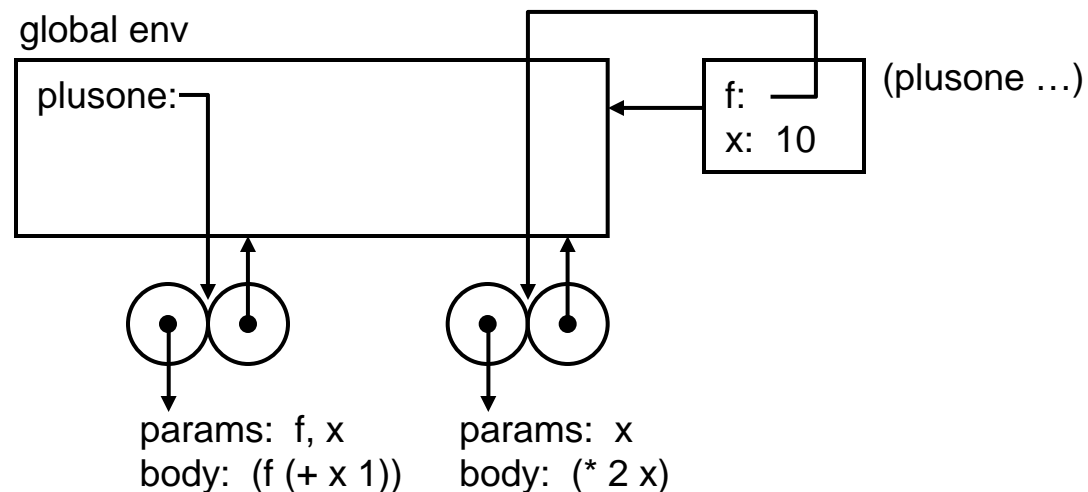
- Note: current environment hasn't changed yet!

- Haven't applied plusone to its operands yet!



Passing Unnamed Procedure (3)

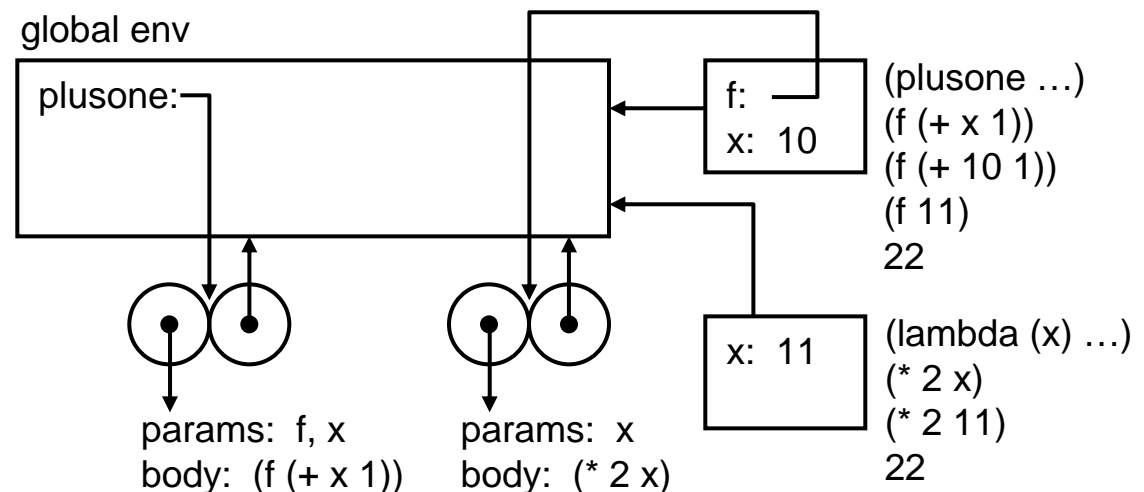
- Apply plusone to operands
 - Create new frame...
 - Bind arguments
 - Current environment changes
 - Evaluate body of procedure in new environment



Passing Unnamed Procedure (4)

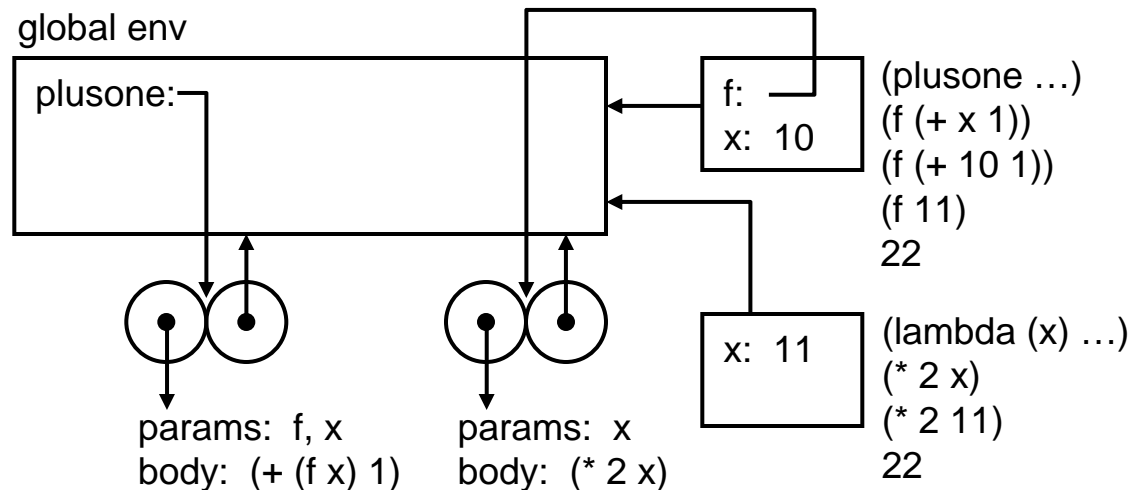
- Body of plusone invokes unnamed procedure
 - ...named f inside of plusone...
 - Create a new frame and bind arguments
 - Arguments are still evaluated in current env.
 - ...frame with both f and x bound...
 - Parent of new frame is global environment, again!

■ Complete diagram:



Passing Unnamed Procedure (5)

■ Complete diagram:



■ Notes:

- Remember that “current environment” doesn’t change until the procedure is applied to its arguments
- If an argument is a lambda expression, it *will not* point to the frame created by calling the procedure

Additional Rules

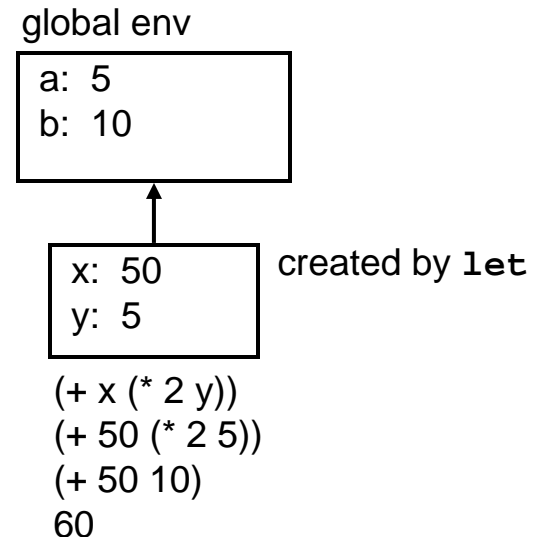
- Four additional “rules” or guidelines
 - These help simplify and clarify evaluation
 - Easily “derived” from first 4 rules
- Fifth rule deals with `let` expressions
 - `let` desugars into a `lambda`...
 - Simplification of rule for evaluating lambdas
- Rules 6 through 8 are just important points to keep in mind

Rule 5: `let` Expressions

5. Evaluating a `let` expression generates a new frame with the specified bindings
- Body of `let` is evaluated in context of the new frame
 - Just like creating a lambda and immediately applying it
 - ...without actually diagramming that lambda pair thingy

■ Simple example:

```
(define a 5)
(define b 10)
(let ((x (* a b))
      (y (- b a)))
  (+ x (* 2 y)))
```



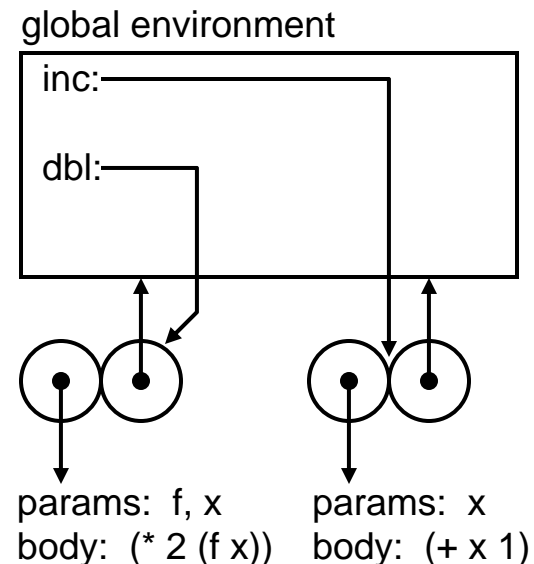
Rule 6: Bindings

- Bindings are *always* between a name and a value
 - If you have a binding between two names, there's an error in your diagram.

- Example:

```
(define (inc x) (+ x 1))  
(define (dbl f x) (* 2 (f x)))  
(dbl inc 3)
```

After creating
both lambdas...



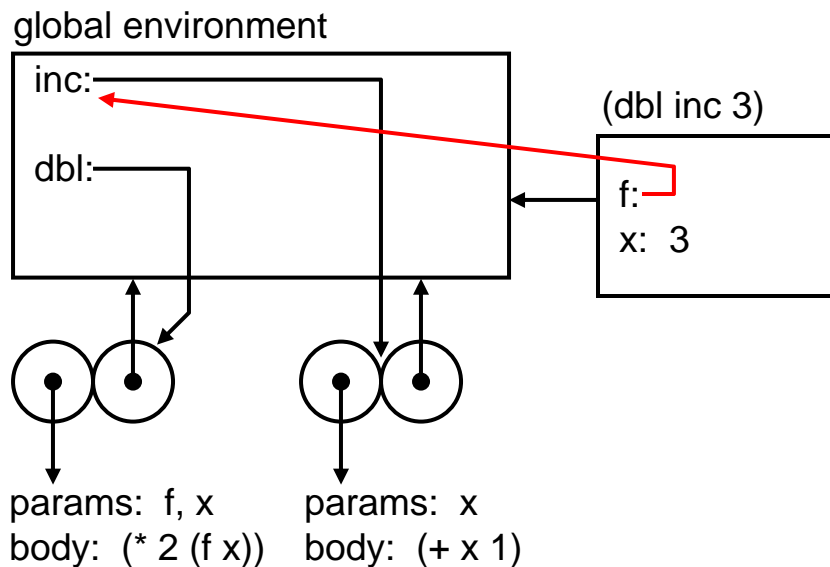
Rule 6: Bindings (2)

■ Example:

```
(define (inc x) (+ x 1))
```

```
(define (dbl f x) (* 2 (f x)))
```

```
(dbl inc 3)
```



WRONG: Violates rule 6!

Value of inc is a procedure.

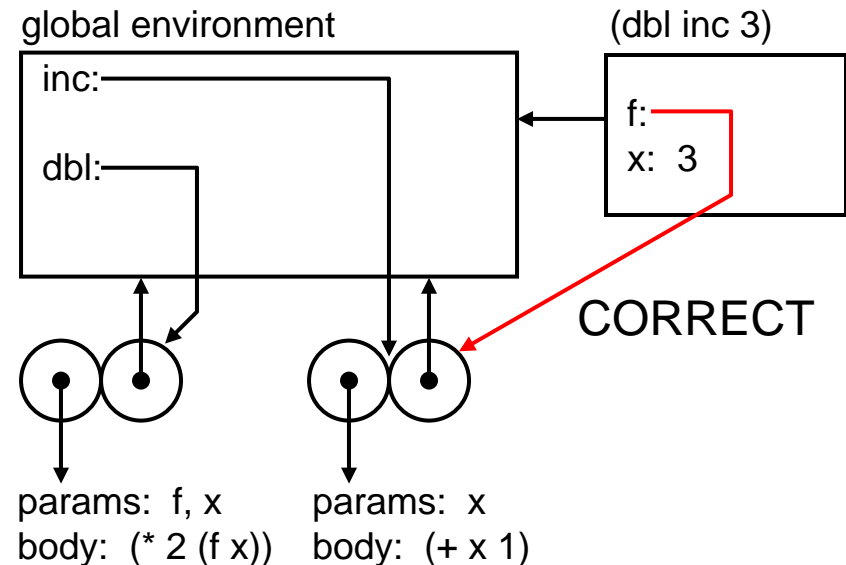
Rule 6: Bindings (3)

■ Example:

```
(define (inc x) (+ x 1))  
(define (dbl f x) (* 2 (f x)))  
(dbl inc 3)
```

■ Next,

- dbl invokes f...
- New frame is created
- Parent of new frame is *global environment*
 - lambda expression points to global environment



Rule 6: Bindings (4)

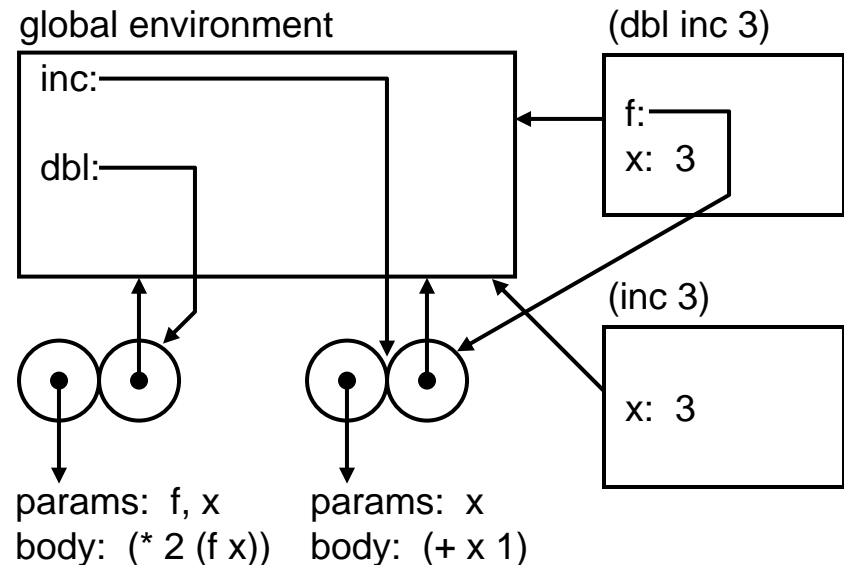
■ Code:

```
(define (inc x) (+ x 1))
```

```
(define (dbl f x) (* 2 (f x)))
```

```
(dbl inc 3)
```

■ Final diagram:



Rules 7 and 8

7. Always write procedures out in desugared form
 - Clarifies what is going on in the code...
 - Will help with diagramming procedures
8. Values bound in the “bindings” section of a `let` expression are evaluated in same environment that the `let` is evaluated in
 - i.e. values in the bindings section are *not* evaluated in the new frame created by the `let` expression!

```
(let ((x 10)
      (y 20))
      (+ x y))
```

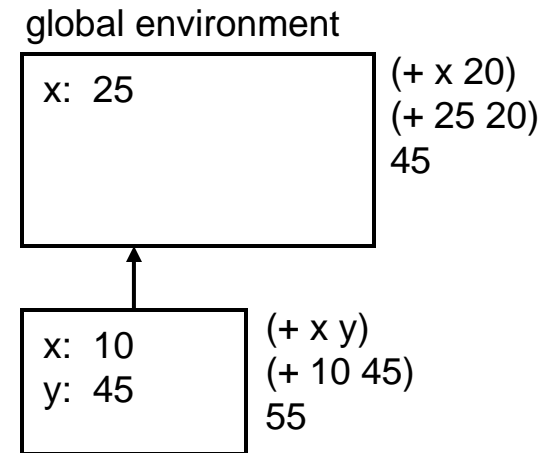
← “bindings section”
of `let` expression

Rule 8 Example

- **Code:**

```
(define x 25)
(let ((x 10)
      (y (+ x 20)))
  (+ x y))
```

Evaluated in context
of global environment!



- **Rule 8:**
 - Values in bindings section are evaluated in same environment that the let is evaluated in
 - `(+ x 20)` is also evaluated in global environment, where `x = 25`
 - New frame contains `x = 10`, `y = 45`
 - `(+ x y)` evaluated in new frame to produce `55`

Trapped Frames

- If a frame is (directly or indirectly) accessible from global environment:
 - That frame is “trapped”
 - It doesn’t get reclaimed by the Scheme interpreter
- Lambda expressions can point to a frame
 - Other frames can point to a frame, but you can’t bind a name to a frame directly
 - Frames are trapped by binding a lambda to a name in the global environment

Trapped Frames (2)

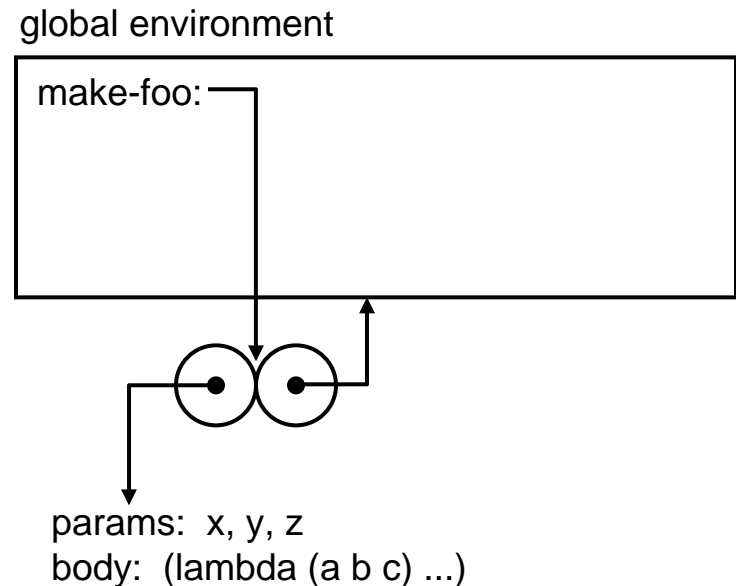
- Example:

```
(define (make-foo x y z)
  (lambda (a b c) ...))
(define foo (make-foo 5 10 15))
```

- Desugar first define:

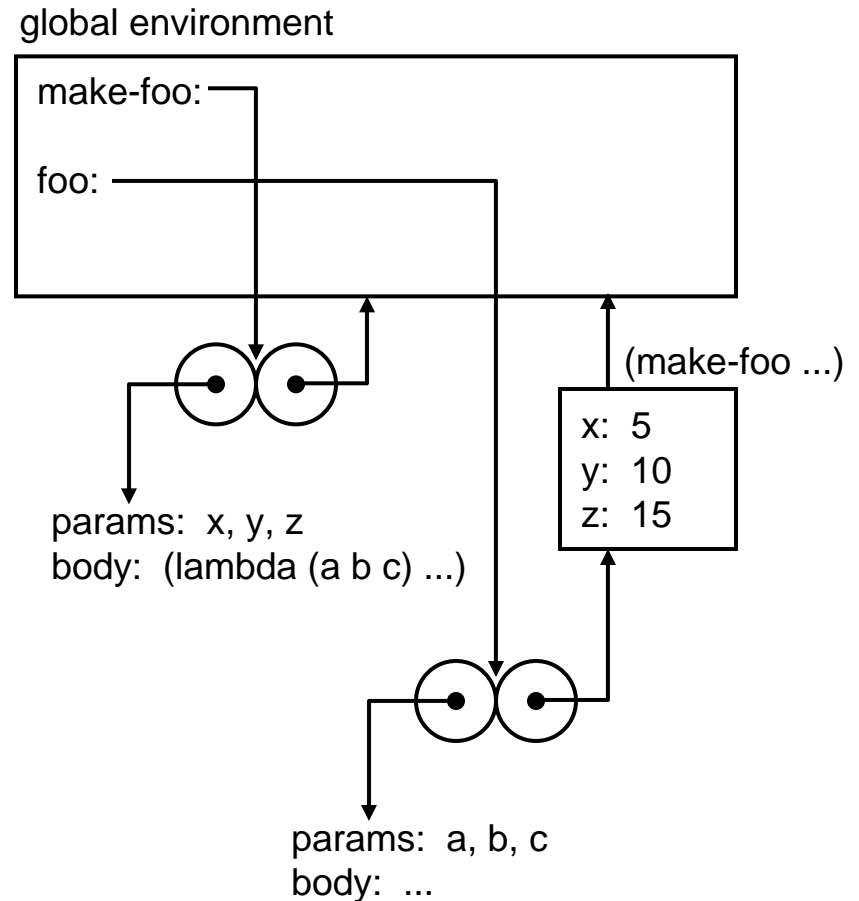
```
(define make-foo
  (lambda (x y z)
    (lambda (a b c) ...)))
```

- Evaluating first define:



Trapped Frames (3)

- Example:
(define (make-foo x y z)
 (lambda (a b c) ...))
(define foo (make-foo 5 10 15))
- Second define isn't a sugared form
- foo in global environment refers to the new procedure
 - Frame created by make-foo invocation is trapped
 - The frame won't be reclaimed



Making Trapped Frames

- Simple recipe for creating trapped frames:
 - Write a procedure that returns another procedure

```
(define (make-foo x y z)
  (lambda (a b c) ...))
```
 - Invoke maker and store newly created procedure

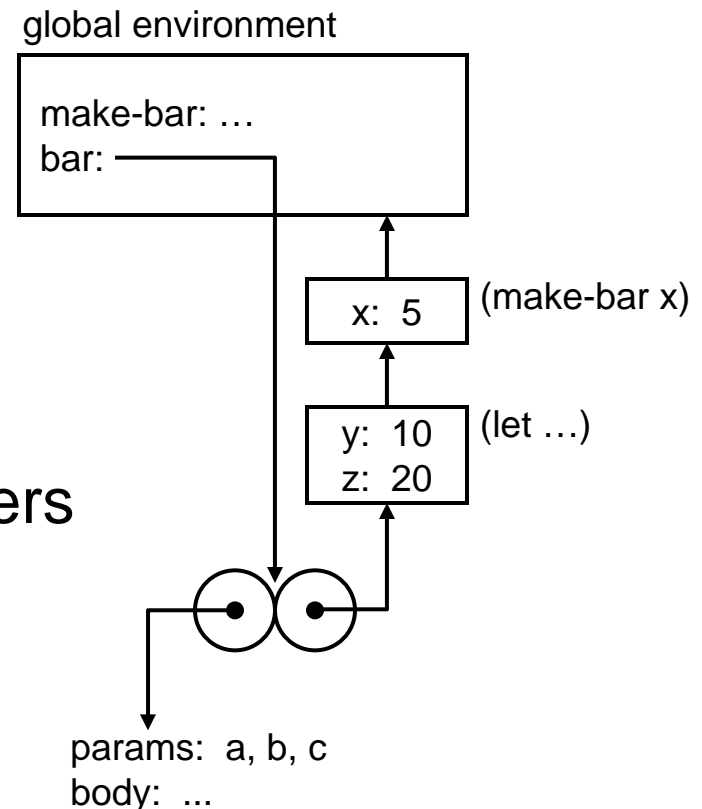
```
(define foo (make-foo 5 10 15))
```
 - Global environment now references new procedure, which traps the frame created by invoking the maker

Trapped Frames with `let`

- Can wrap inner procedure with `let` expressions too

```
(define (make-bar x)
  (let ((y 10)
        (z 20))
    (lambda (a b c) ...)))
(define bar (make-bar 5))
```

- Traps an additional frame with extra bindings
- Useful when function parameters don't have all necessary state



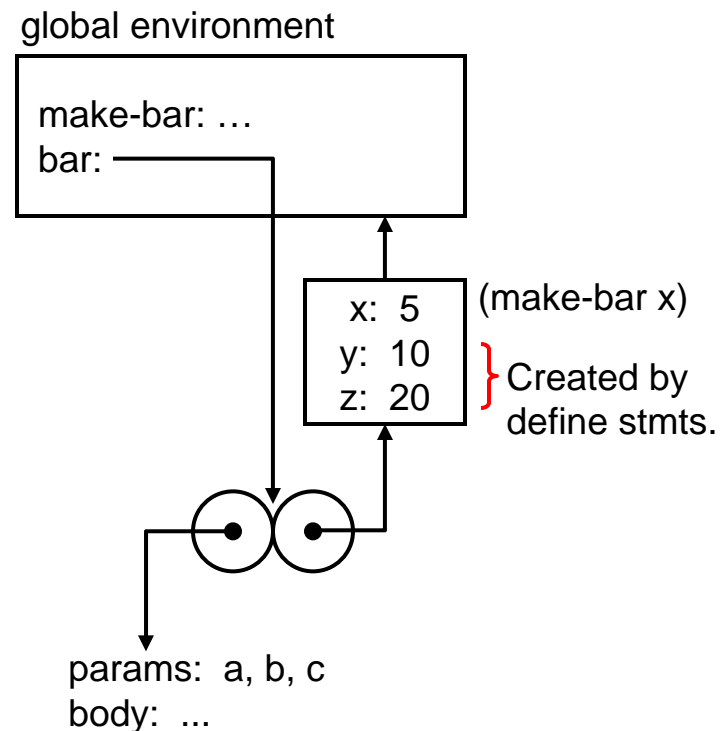
Trapped Frames without `let`

- Of course, could also use `defines` to accomplish the same thing!

```
(define (make-bar x)
  (define y 10)
  (define z 20)
  (lambda (a b c) ...))
```

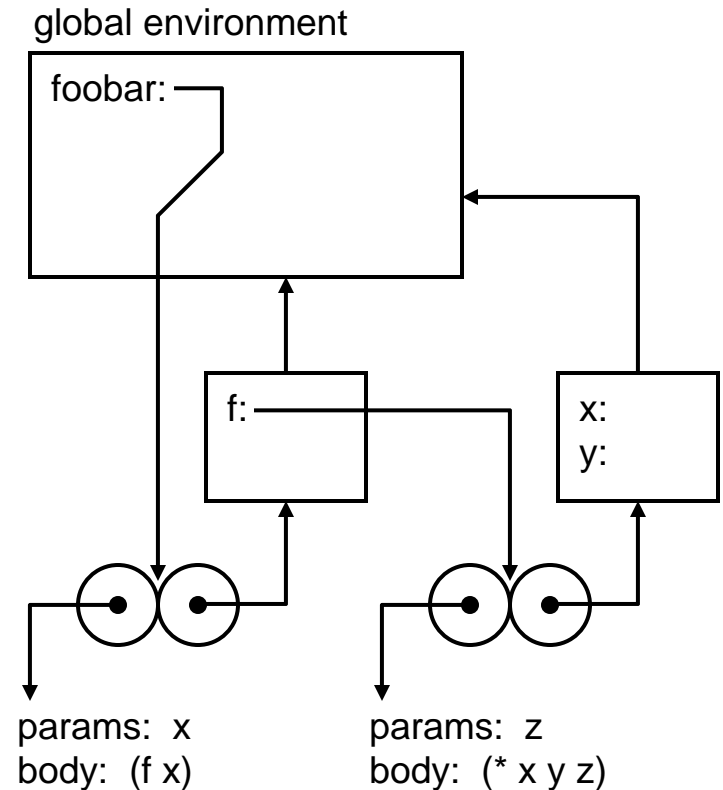
- Main difference between this and `let`?

- One less frame is created
- `y`, `z` are stored in frame created by calling `make-bar`



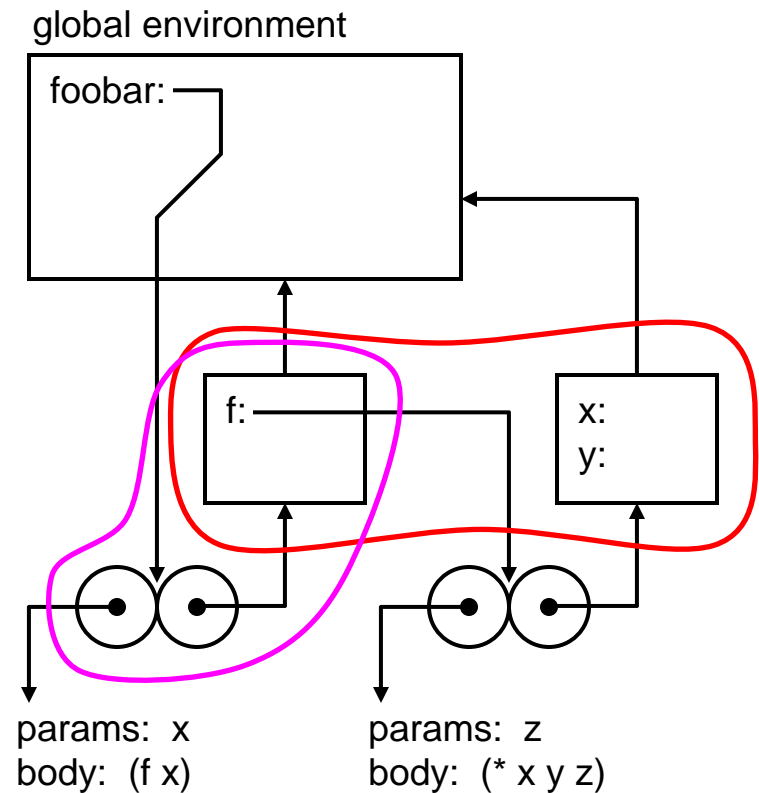
More Trapped Frames...

- Frame with x , y is trapped indirectly
 - Reachable through `foobar` binding in global env.
- What Scheme code would create this?
 - Assignment has several problems like this.



Diagrams → Scheme

- As HW says, feel free to add other things to environment besides what is in the diagram
- What causes the various configurations?
 - Frames are created by:
 - Calling a procedure
 - Evaluating a `let` expression
 - For a lambda to point to a non-global frame:
 - A procedure must be returned from another procedure-call
 - A procedure must appear within a `let` expression



Diagrams → Scheme (2)

- What causes the various configurations?
 - For a non-global frame to point to a procedure:
 - The procedure must be passed as an argument to another procedure
- Keep these concepts in mind
 - Gives the general form of the solution...
 - Once you have some Scheme code, try diagramming it to see what it actually produces

