

6.001 Recitation 15: The Environment Model

RI: Gerald Dalley, dalleyg@mit.edu, 5 Apr 2007
<http://people.csail.mit.edu/dalleyg/6.001/SP2007/>

Gerald's Top 3

1. Higher-order procedures
2. THE ENVIRONMENT MODEL
3. ...I'll tell you next week...

I love higher-order procedures because they allow us to do some really powerful things and when using other languages like C++ I miss them immensely for their elegance and compactness. In particular, you just can't beat Scheme's `map`, `filter`, and `fold-right/fold-left` procedures.

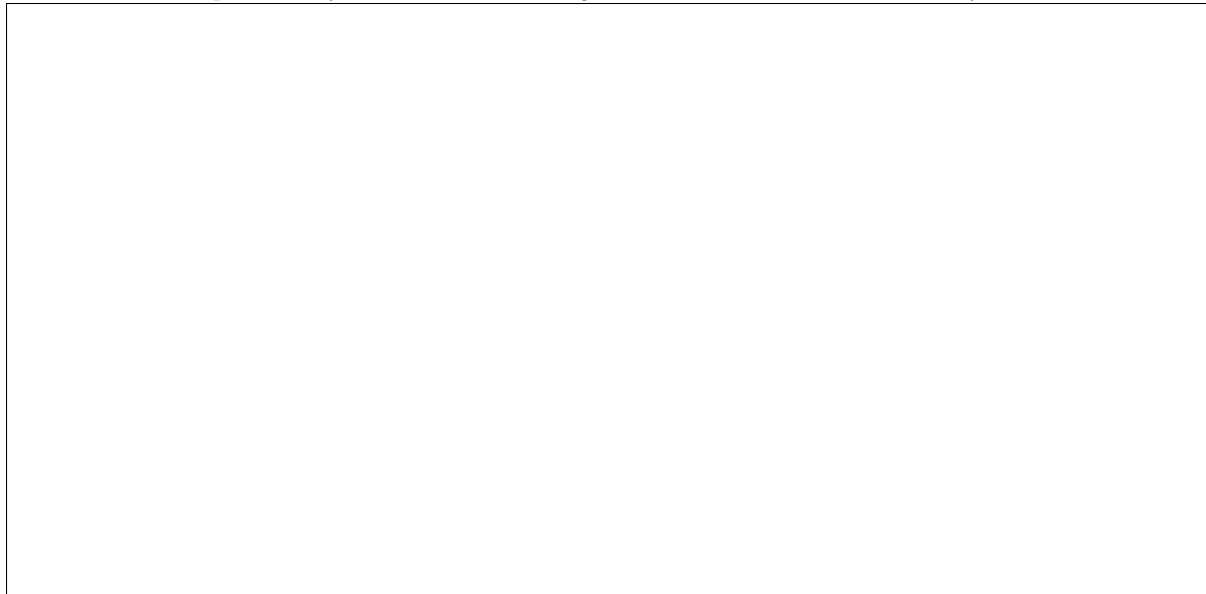
C++ does have HOPs, but they are not nearly as convenient nor as powerful. A good understanding of the environment model tells me why (IMHO) C++ will never have a fully-functional `map`, `filter`, and `fold-right/fold-left` procedures that are as convenient as Scheme's. On the other hand, some languages I use such as Python and recent versions of Matlab implement a similar environment model that allow for much or all of Scheme's power.

Environmental Facts

```
(define fact
  (lambda (n)
    (if (= n 1)
        1
        (* n (fact (- n 1))))))
(fact 3)
```

What is the value of this expression?

Show all relevant portions of the environment diagram used to evaluate this block of code.



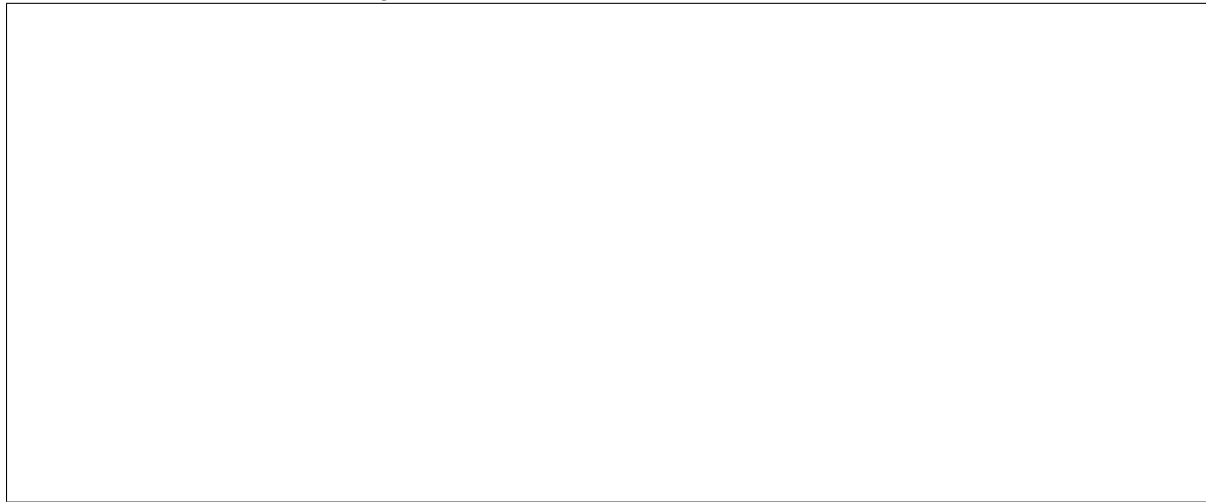
Scoping, `define` versus `set!`, and Shadowing

```
(define x 0)
(define f
  (lambda (y)
    (define x (+ y 10))
    x))
(define g
  (lambda (y)
    (set! x (+ y 10))
    x))
```

Find the values of:

(f 5) , x , (g 5) , and x

...and show the environment diagram:

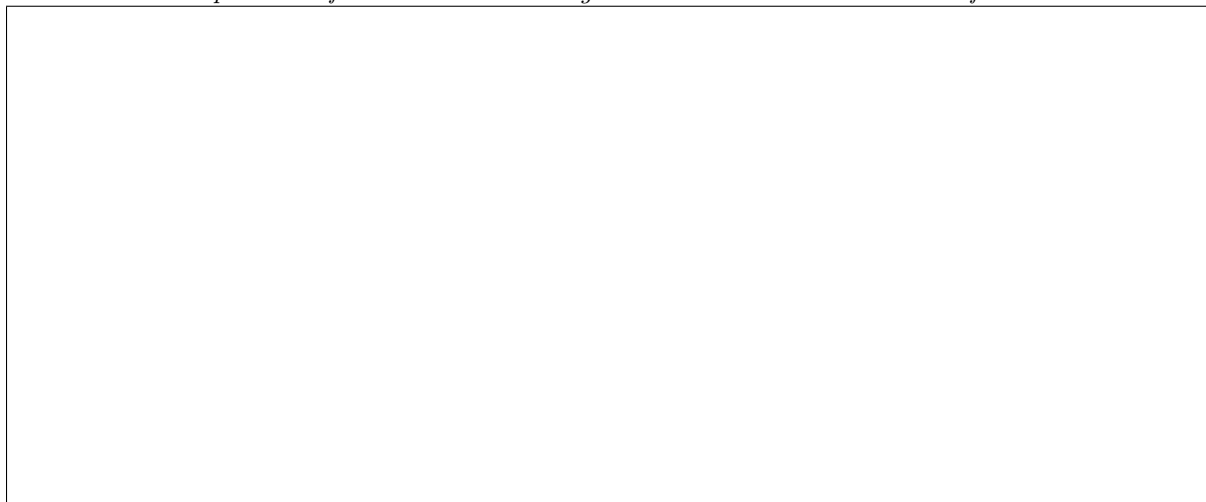


Nameless Wonders

```
(define x 3)
((lambda (x y) (+ (x 1) y))
 (lambda (z) (+ x 2))
 3)
```

What is the value of this expression?

Show all relevant portions of the environment diagram used to evaluate this block of code.

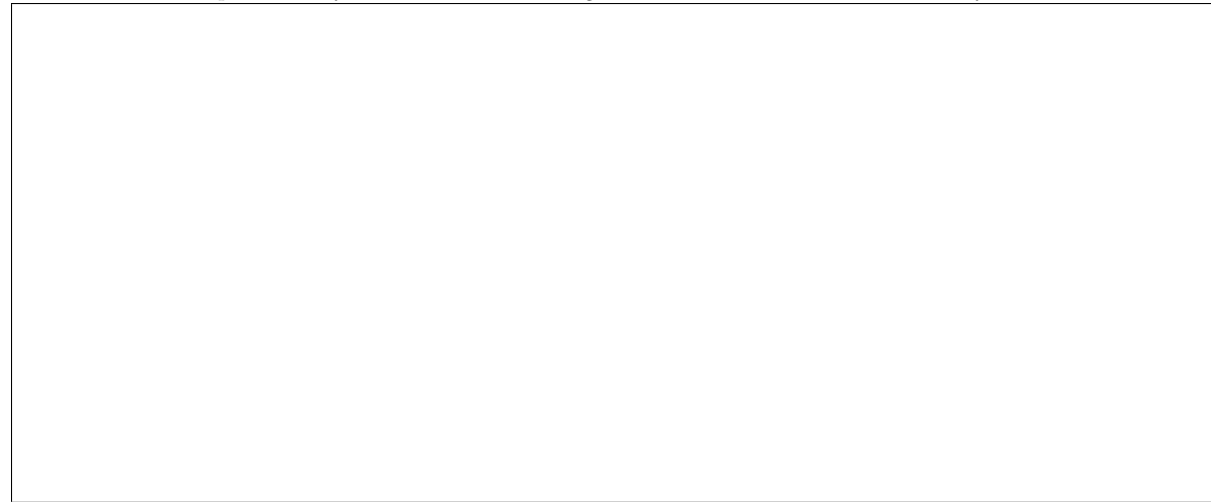


Aspartame (Desugaring `let`)

Desugar the following expression:

```
(define x 4)
(let ((x (+ 2 1))
      (y (square x)))
  (* x y))
```

Show all relevant portions of the environment diagram used to evaluate this block of code.

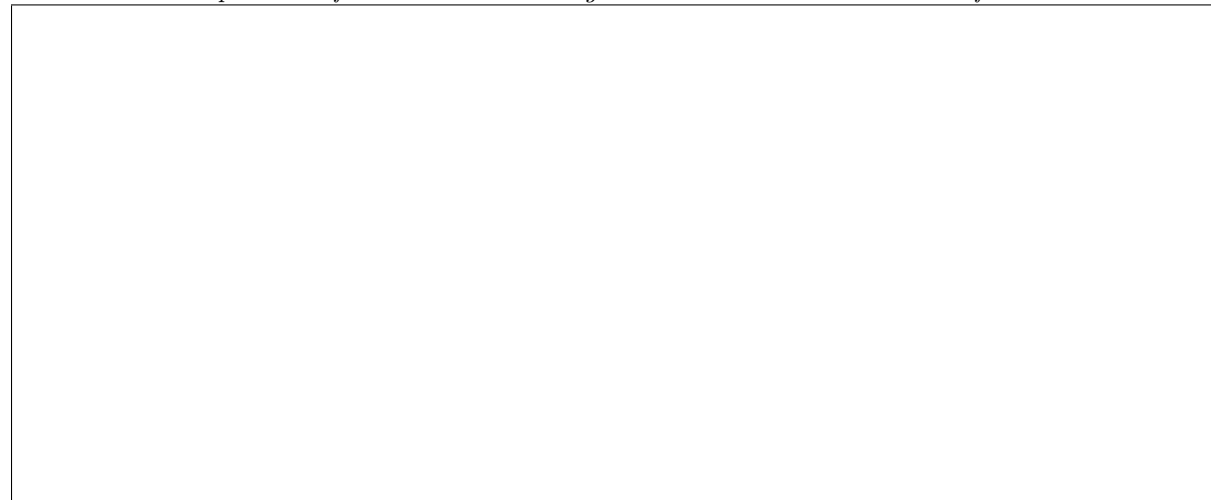


λ -let

```
(define x 5)
(let ((x (lambda (x) (+ 6 x)))
      (set! x (x 7))
      x)
```

What is the value of this expression?

Show all relevant portions of the environment diagram used to evaluate this block of code.



Yet More Complexity

```
(define a 5)
(define foo
  (let ((a 10))
    (lambda (x)
      (+ x a))))
(define (bar a) (foo 20))
(bar 100)
```

What is the value of this expression?

Show all relevant portions of the environment diagram used to evaluate this block of code.



Insanity!

```
(define (make-count-proc f)
  (let ((count 0))
    (lambda (x)
      (if (eq? x 'count)
          count
          (begin (set! count (+ count 1))
                  (f x))))))

(define sqrt* (make-count-proc sqrt))
(define square* (make-count-proc square))
```

Find the values of:

(sqrt* 4) ,

(sqrt* 'count) ,

(square* 4) ,

and (square* 'count)

...and show the environment diagram:



Environment Model Cheat Sheet

Elements of the Environment Model

- A frame consists of a list of variable bindings. Each binding associates a name (must be a symbol) with a value. We draw frames as boxes.
- GE = Global Environment. All initial bindings (e.g. for `+`, `map` live in the GE . Whenever we evaluate something, we must specify the frame in which we evaluate it.
- Every frame except the GE frame has a “parent” pointer which points to another frame (also called the enclosing environment). The frames form a tree structure, with the GE as root.
- Each frame has an associated environment. Frame F ’s environment consists of the chain of frames F , $parent(F)$, $parent(parent(F))$, until we hit the GE .
- New frames are created when a procedure is called, or when a `let` statement is evaluated.

The Hats

Double-bubble: In charge of the lambda rule (double-bubble creation)

Bind: In charge of step 5 of the combination rule, `set!` rule, `define` rule, symbol lookup rule

Trouble: In charge of steps 2–4 of the combination rule

Grand Evaluator: In charge of keeping track of evaluation, current environment, identifying the type of expression, and remembering the values of arguments

Evaluation Rules

To evaluate an expression in an environment e , follow the rule:

`name`| e

Lookup `name` in the current environment (e), moving up frames to find the `name`. Return the value bound to the `name`.

`(define name exp)`| e

Evaluate `exp` in e to get `val`, and create or replace a binding for `name` in the first frame of e with `name`. Return unspecified.

`(set! name exp)`| e

Evaluate `exp` in e to get `val`, and replace the first binding for `name` in e with `val`. If no such binding is found, generate an error. Return unspecified.

`(lambda args body)`| e

Create a double-bubble whose environment pointer (right half) is e , and set the left half to have the parameters `args` and body `body`. Return a pointer to the double-bubble.

`(exp1 exp2 exp3...)`| F

1. Evaluate each expression `exp1`, `exp2`, `exp3`, ... in frame F , resulting in `val1`, `val2`, `val3`, ...

2. If `val1` does not point to a double-bubble, it is an error. Otherwise, let P be this double-bubble.

3. Create a new frame A

4. Make A into an environment E : A ’s enclosing environment pointer goes to the same frame as the environment pointer of P . Link these two pointers together with handcuffs.

5. In A , bind the parameters of P to the values `val2`, ... (`val1` is the double-bubble)

6. Evaluate the body of P with E as the current environment.

`(let ((var init) ...) body)`| F

Either desugar the `let`, or:

1. Evaluate each `init` in F (in any order) to get `vals`

2. Drop a new frame A that points to F

3. In A , bind each `var` to the associated `val`

4. Evaluate `body` in frame A .

5. Return the value of the last expression in the `body`.

Common Environment Model Mistakes

- Be sure to set the parent pointer of new frames properly — it’s the same as the environment pointer of the procedure you’re applying!
- Keep track of what expression you’re evaluating, and remember what steps you have left to do. For example, when you have `(define foo bar)`, don’t forget to add the binding for `foo` after you finish evaluating `bar`!
- Don’t get ahead of yourself! A common mistake is for people to evaluate a `lambda` expression, giving a double bubble, and then immediately evaluate the body of the lambda. Be sure that you follow the rules carefully!

Solutions

Gerald's Top 3

1. Higher-order procedures
2. THE ENVIRONMENT MODEL
3. ...I'll tell you next week...

I love higher-order procedures because they allow us to do some really powerful things and when using other languages like C++ I miss them immensely for their elegance and compactness. In particular, you just can't beat Scheme's `map`, `filter`, and `fold-right/fold-left` procedures.

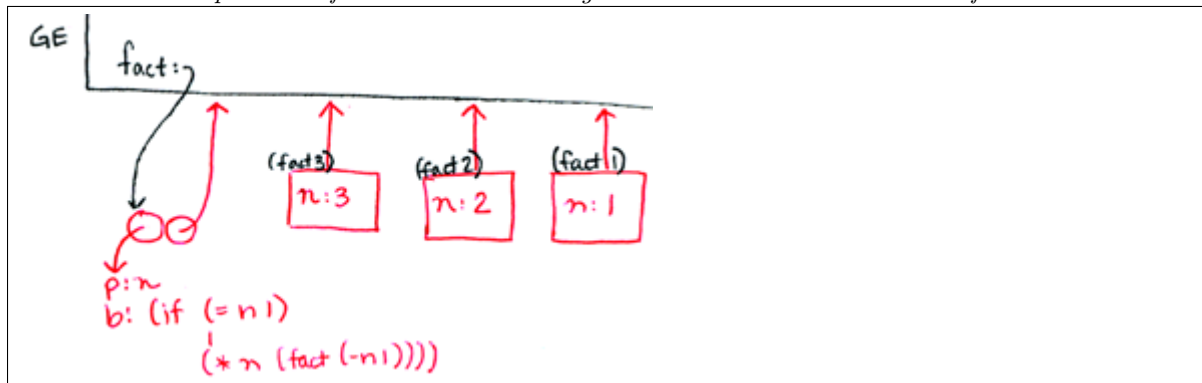
C++ does have HOPs, but they are not nearly as convenient nor as powerful. A good understanding of the environment model tells me why (IMHO) C++ will never have a fully-functional `map`, `filter`, and `fold-right/fold-left` procedures that are as convenient as Scheme's. On the other hand, some languages I use such as Python and recent versions of Matlab implement a similar environment model that allow for much or all of Scheme's power.

Environmental Facts

```
(define fact
  (lambda (n)
    (if (= n 1)
        1
        (* n (fact (- n 1)))))
(fact 3)
```

What is the value of this expression? 6

Show all relevant portions of the environment diagram used to evaluate this block of code.



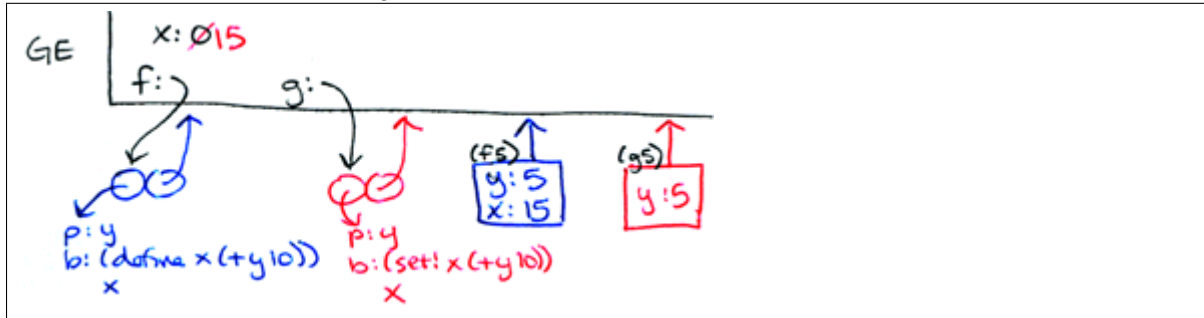
Scoping, `define` versus `set!`, and Shadowing

```
(define x 0)
(define f
  (lambda (y)
    (define x (+ y 10))
    x))
(define g
  (lambda (y)
    (set! x (+ y 10))
    x))
```

Find the values of:

`(f 5)` , `x` , `(g 5)` , and `x`

...and show the environment diagram:

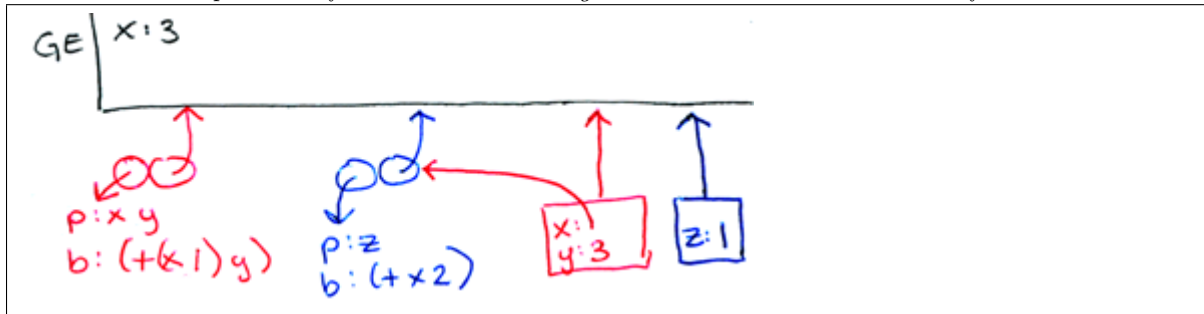


Nameless Wonders

```
(define x 3)
((lambda (x y) (+ (x 1) y))
 (lambda (z) (+ x 2))
 3)
```

What is the value of this expression?

Show all relevant portions of the environment diagram used to evaluate this block of code.



Aspartame (Desugaring `let`)

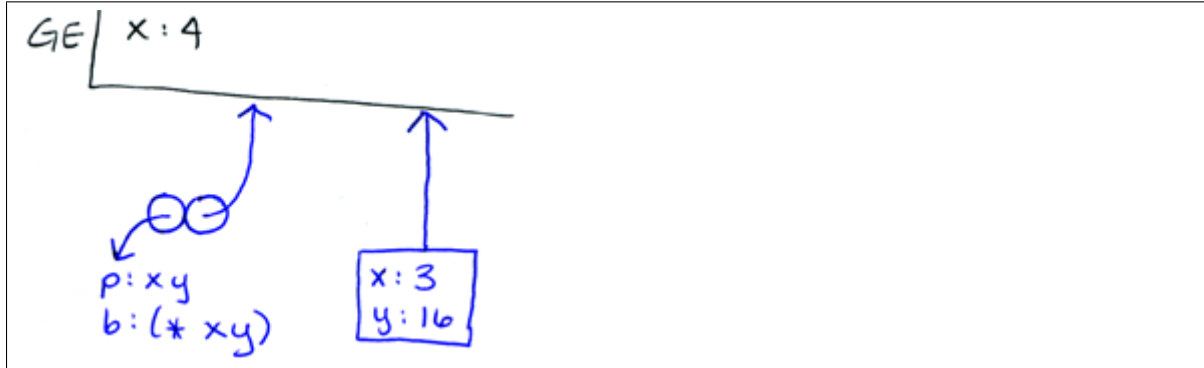
Desugar the following expression:

```
(define x 4)
(let ((x (+ 2 1))
      (y (square x)))
  (* x y))

;DESUGARS TO:

((lambda (x y) (* x y))
 (+ 2 1)
 (square x)) ;==> 48
```

Show all relevant portions of the environment diagram used to evaluate this block of code.

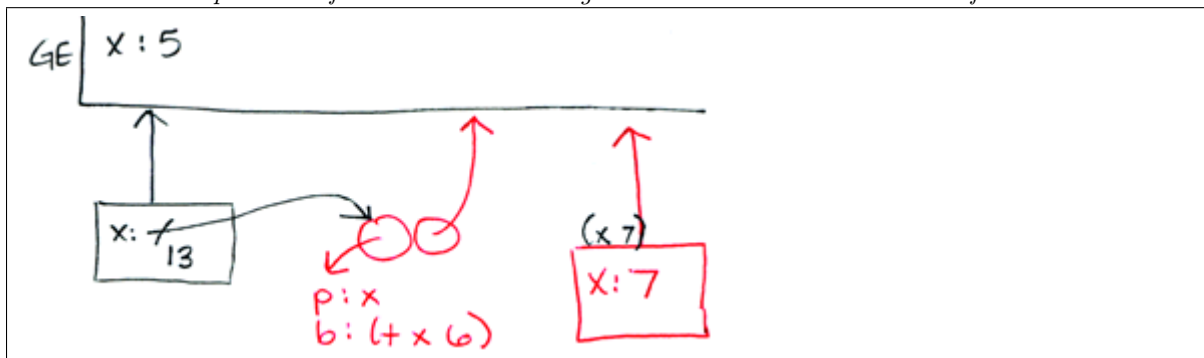


λ -let

```
(define x 5)
(let ((x (lambda (x) (+ 6 x)))
      (set! x (x 7))
      x)
  x)
```

What is the value of this expression? 13

Show all relevant portions of the environment diagram used to evaluate this block of code.

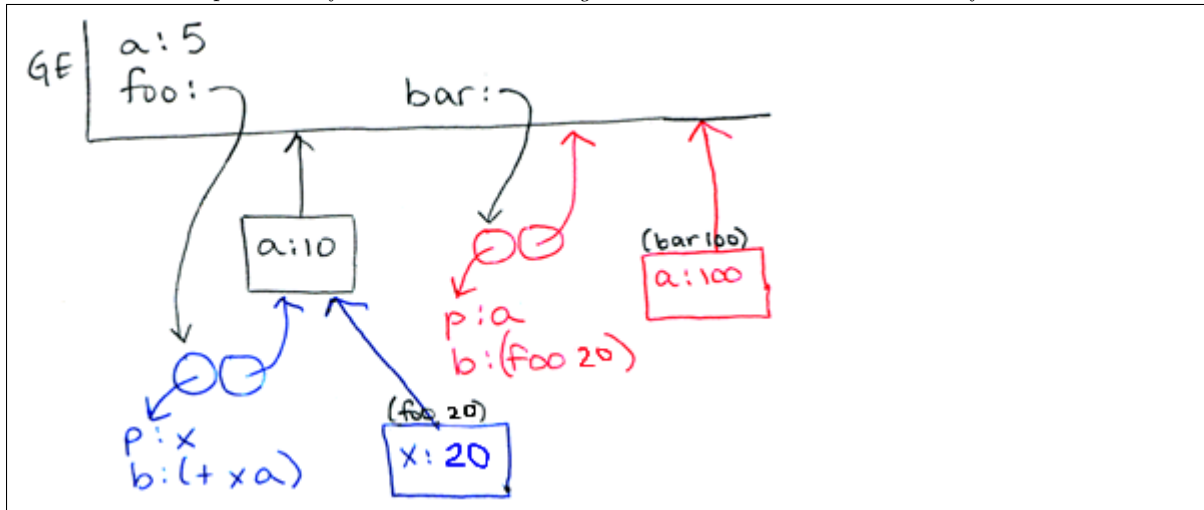


Yet More Complexity

```
(define a 5)
(define foo
  (let ((a 10))
    (lambda (x)
      (+ x a))))
(define (bar a) (foo 20))
(bar 100)
```

What is the value of this expression? 30

Show all relevant portions of the environment diagram used to evaluate this block of code.



Insanity!

```
(define (make-count-proc f)
  (let ((count 0))
    (lambda (x)
      (if (eq? x 'count)
          count
          (begin (set! count (+ count 1))
                  (f x)))))))
```

```
(define sqrt* (make-count-proc sqrt))
(define square* (make-count-proc square))
```

Find the values of:

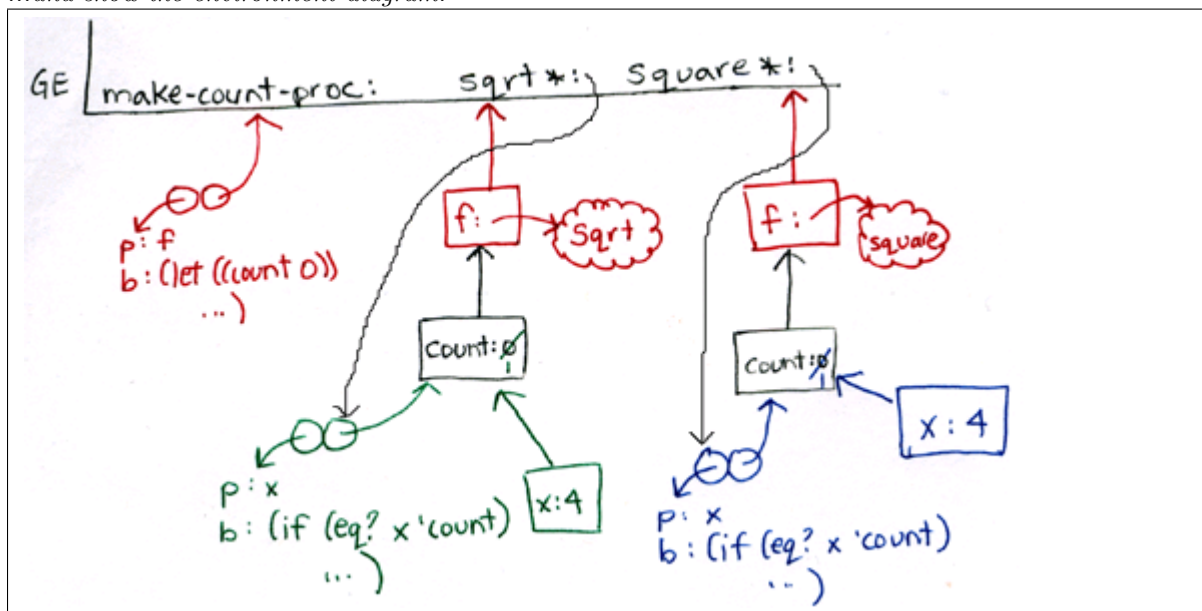
(sqrt* 4) 2,

(sqrt* 'count) 1,

(square* 4) 16,

and (square* 'count) 1

...and show the environment diagram:



Environment Model Cheat Sheet

Elements of the Environment Model

- A frame consists of a list of variable bindings. Each binding associates a name (must be a symbol) with a value. We draw frames as boxes.
- GE = Global Environment. All initial bindings (e.g. for `+`, `map` live in the GE . Whenever we evaluate something, we must specify the frame in which we evaluate it.
- Every frame except the GE frame has a “parent” pointer which points to another frame (also called the enclosing environment). The frames form a tree structure, with the GE as root.
- Each frame has an associated environment. Frame F ’s environment consists of the chain of frames F , $parent(F)$, $parent(parent(F))$, until we hit the GE .
- New frames are created when a procedure is called, or when a `let` statement is evaluated.

The Hats

Double-bubble: In charge of the lambda rule (double-bubble creation)

Bind: In charge of step 5 of the combination rule, `set!` rule, `define` rule, symbol lookup rule

Trouble: In charge of steps 2–4 of the combination rule

Grand Evaluator: In charge of keeping track of evaluation, current environment, identifying the type of expression, and remembering the values of arguments

Evaluation Rules

To evaluate an expression in an environment e , follow the rule:

$name|_e$

Lookup `name` in the current environment (e), moving up frames to find the `name`. Return the value bound to the `name`.

$(define\ name\ exp)|_e$

Evaluate `exp` in e to get `val`, and create or replace a binding for `name` in the first frame of e with `name`. Return unspecified.

$(set!\ name\ exp)|_e$

Evaluate `exp` in e to get `val`, and replace the first binding for `name` in e with `val`. If no such binding is found, generate an error. Return unspecified.

$(lambda\ args\ body)|_e$

Create a double-bubble whose environment pointer (right half) is e , and set the left half to have the parameters `args` and body `body`. Return a pointer to the double-bubble.

$(exp1\ exp2\ exp3\ \dots)|_F$

1. Evaluate each expression `exp1`, `exp2`, `exp3`, ... in frame F , resulting in `val1`, `val2`, `val3`, ...
2. If `val1` does not point to a double-bubble, it is an error. Otherwise, let P be this double-bubble.
3. Create a new frame A
4. Make A into an environment E : A ’s enclosing environment pointer goes to the same frame as the environment pointer of P . Link these two pointers together with handcuffs.
5. In A , bind the parameters of P to the values `val2`, ... (`val1` is the double-bubble)
6. Evaluate the body of P with E as the current environment.

$(let\ ((var\ init)\ \dots)\ body)|_F$

Either desugar the `let`, or:

1. Evaluate each `init` in F (in any order) to get `vals`
2. Drop a new frame A that points to F
3. In A , bind each `var` to the associated `val`
4. Evaluate `body` in frame A .
5. Return the value of the last expression in the body.

Common Environment Model Mistakes

- Be sure to set the parent pointer of new frames properly — it’s the same as the environment pointer of the procedure you’re applying!
- Keep track of what expression you’re evaluating, and remember what steps you have left to do. For example, when you have `(define foo bar)`, don’t forget to add the binding for `foo` after you finish evaluating `bar`!
- Don’t get ahead of yourself! A common mistake is for people to evaluate a `lambda` expression, giving a double bubble, and then immediately evaluate the body of the lambda. Be sure that you follow the rules carefully!