James Lin
<jameslin@cory.eecs.berkeley.edu>

Drawing Environment Diagrams

"Why draw environment diagrams? What's the big deal?" you might be wondering. Maybe they seem unimportant now, but they *are* useful. Primarily, they will allow you to understand and to predict to the behavior of a programming language better . If you continue to take additional CS classes, you *will* see them again, particularly in CS164 (Programming Languages and Compilers) and perhaps in CS61C (Machine Structures). And though they may seem a bit arcane, if you follow their basic rules, they should be easy and straightforward to draw.

Terminology

to bind	to assign a value (which may be a procedure) to a name
binding	a name and its value
frame	a set of bindings created by calling a (compound) procedure
environment	a sequence of frames; a frame and all of its enclosing frames
bubble-pair	a representation of a procedure and the pointer to its frame

Rules

Here are the basic rules for drawing environment diagrams. Remember that even though some of the details are specific to Scheme syntax, these rules apply to any **lexically scoped** language (though in slightly different forms).

- Every `lambda` expression creates a procedure, represented as a "bubble-pair." The **left bubble** represents the **structure** of the procedure; it lists the **parameters** to the procedure and the **body** (the code it executes). The **right bubble** contains a pointer back to the **frame** in which the procedure was created.

- Whenever a (compound) procedure is **invoked**:
  1. Create a new frame. This frame becomes the **current frame**. The frame always points back to the *same* frame as the right bubble of the invoked procedure.
  2. Bind values to the parameters of the invoked procedure.
  3. Evaluate/execute the body of the invoked procedure.

- Every `define` expression **creates** a new binding in the current **frame**.

- Every `set!` expression **assigns** a new value to the closest *existing* binding in the current **environment**.

Also remember that Scheme evaluates expressions using **applicative order**. This means that the arguments to a procedure are evaluated *before* the procedure actually is invoked.

For our purposes, we will make the following distinction between `define` and `set!`: `define` will be used to declare or to *create* bindings only, and `set!` will be used to assign values to them afterward.

In Scheme, *only* `lambda` creates procedures. This includes **implicit** `lambda` expressions that occur in code such as:

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

(let ((a 1)                   ⇔   ( (lambda (a b c) (+ a b c))
      (b 2)                         1 2 3)
      (c 3) )
  (+ a b c) )
```

Because every `let` statement is **syntactic sugar** for a `lambda` expression *and* its invoked procedure, every `let` statement creates a bubble-pair *and* a new frame for it.

(If you are having trouble with `let` statements or with the syntactic sugar form of `define`, it may help to rewrite all the code first using their equivalent `lambda` forms.)

Garbage Collection

Frames consume memory. When a procedure is invoked and creates a frame, what happens after the procedure terminates?

The Scheme interpreter uses a **garbage collector**. If there exists a structure in memory—such as a frame, a procedure, or a `cons` pair—that can no longer be accessed from the current environment, the garbage collector will delete it and free the memory.

You'll learn more details about how a garbage collector works in CS61B (Data Structures) and in CS164, but for now just know that it discards stuff in memory that's no longer in use.

Suggestions/Reminders

- Label/number your frames. This makes it easier to keep track of the order in which the frames were created.
- Draw *all* frames and bubble-pairs, even those that are "garbage collected." Rather than erase temporary frames and bubbles, cross them out.
- The right bubble of a procedure always points to a single **frame**.
- A frame always points to another, single **frame**.
- A binding can only point to a **value** (a number, symbol, string, pair, procedure, etc.). It *cannot* point to a frame.
- `lambda` is a **constructor** for procedures. Evaluating a `lambda` expression **returns a new procedure**.
- Before invoking a procedure, evaluate all of its arguments *first*.
- We don't indicate return values unless we do something with them (such as binding them to a name).
- You will almost certainly be tested on drawing environment diagrams. Do *not* depend on using `envdraw`. Instead, try drawing some environments manually and use `envdraw` to *check* your work.

Examples

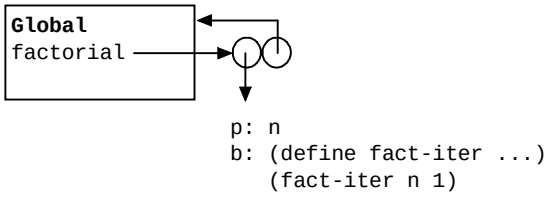
Let's draw the environment diagram for the iterative version of `factorial`:

```
(define (factorial n)
  (define (fact-iter i result)
    (if (= i 0)
        result
        (fact-iter (- i 1) (* i result)) ))
  (fact-iter n 1) )

(factorial 3)
```

It's a good idea to get rid of all the syntactic sugar:

```
(define factorial
  (lambda (n)
    (define fact-iter
      (lambda (i result)
        (if (= i 0)
            result
            (fact-iter (- i 1) (* i result)) )))
     (fact-iter n 1) ))

(factorial 3)
```

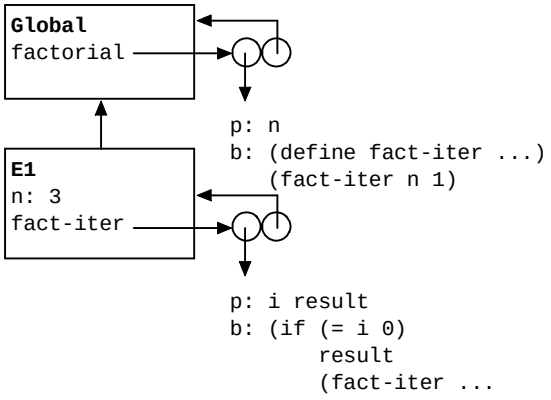
The rule for `define` is to evaluate the second argument—the `(lambda (n) ...)` part—and to bind the result to the first argument—in this case, `factorial`. We initially start off in the global frame, which we'll assume is empty. (It's really not empty; this is where all of the Scheme primitives are bound, but we don't bother showing them.) We evaluate the `(lambda (n) ...)`, which returns a procedure. This procedure was created in the global frame, so that's where the right bubble points. We create a variable `factorial` and bind it to that procedure, indicated with an arrow.



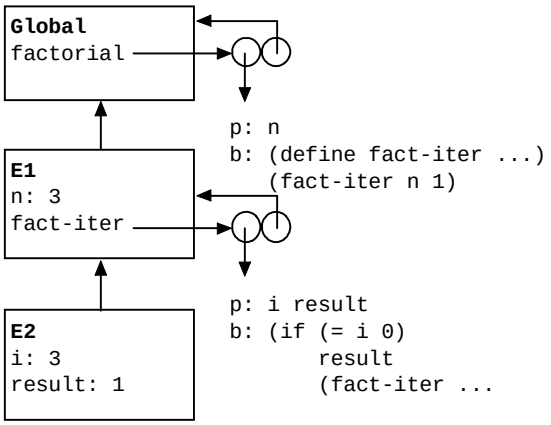
That's it for the for the `(define factorial ...)` part! Now let's handle the invocation `(factorial 3)`.

`(factorial 3)` is a procedure call. Therefore we create a new frame E1, pointing back to the global frame. E1 is now our current frame. We next assign values to the parameters, so we bind `n` to 3 inside of E1. Finally, we evaluate the body of the `factorial` procedure.

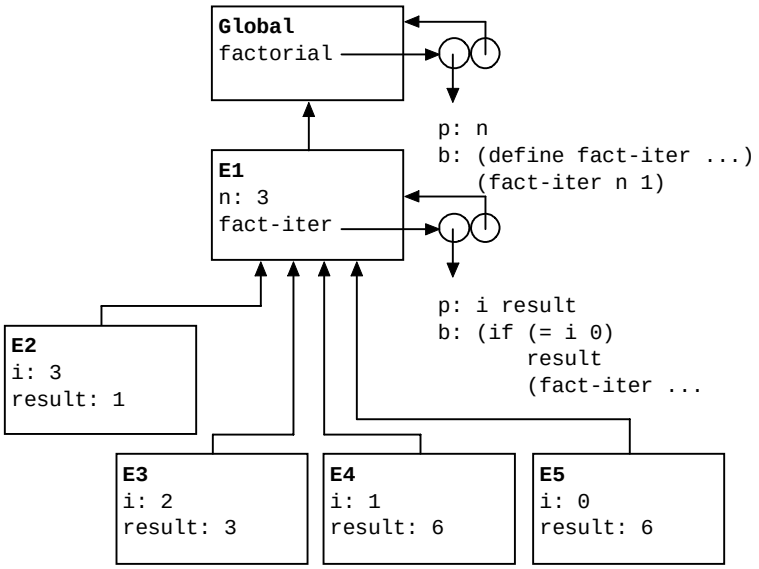
The first part of `factorial`'s body is another `define` expression. We do the same thing as we did before. This time the procedure points to E1, since that's where it was created.



The second part of `factorial`'s body is a procedure call to `fact-iter`. This creates a new frame, E2. E2 points to E1 because that is where the `fact-iter` procedure points. We assign values to its parameters. (Note that `i` is bound to 3 and *not* to `n`! We have to use applicative-order evaluation, which means that we evaluate all of the arguments *first*.)



Finally, we evaluate the body of the `fact-iter` procedure, which recursively calls `fact-iter`. Since `fact-iter` is not bound in the current frame E2, we instead look to the enclosing frame. We find the enclosing frame by following the arrow from E2. From here, the process should be fairly straightforward.



Note that *every* frame created by calling `fact-iter` always points back to E1. Also note that we don't write the return value of `(factorial 3)` anywhere, since we don't do anything with it.

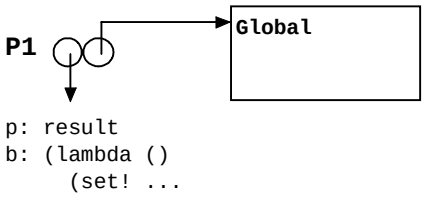
Now let's look at an example that maintains **local state**.

```
(define count
  (let ((result 0))
    (lambda ()
      (set! result (+ result 1))
      result) ))

(count)
(count)
```

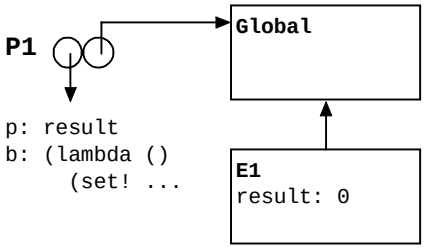
How does this work? Again, it's a good idea to get rid of all the syntactic sugar:

```
(define count
  ((lambda (result)
     (lambda ()
       (set! result (+ result 1))
       result) )
   0))

(count)
```

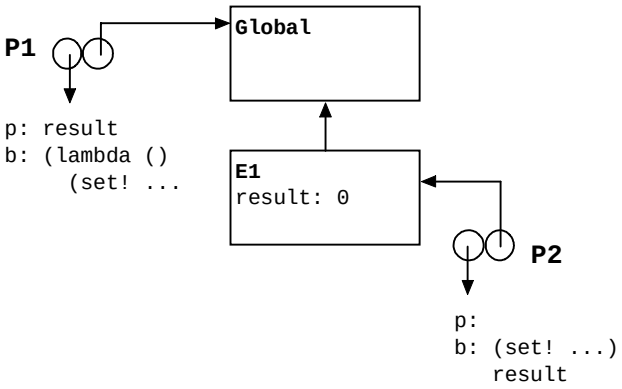
We first evaluate the second argument to the `define`, which is a procedure call. We evaluate `(lambda (result) ...)`, which creates a procedure. Note that *nothing* points to this procedure. For convenience, however, let's refer to this procedure as **P1**.



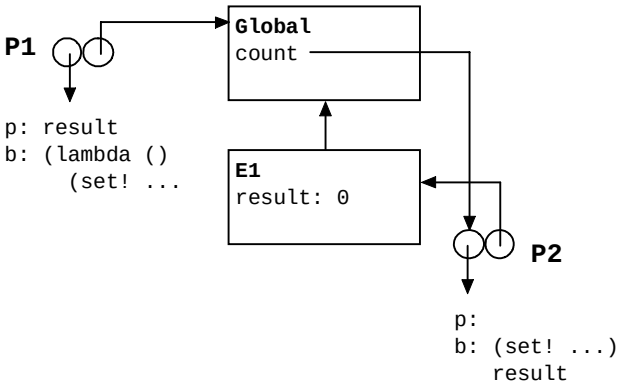
To call P1, we first create a new frame E1 and bind 0 to the parameter `result`. E1 is now our current frame.



Next we evaluate the body of P1. This causes us to evaluate another `lambda` expression. This creates a procedure of no arguments. This procedure points back to E1, since that is where it was created. For convenience, let's refer to this procedure as **P2**.

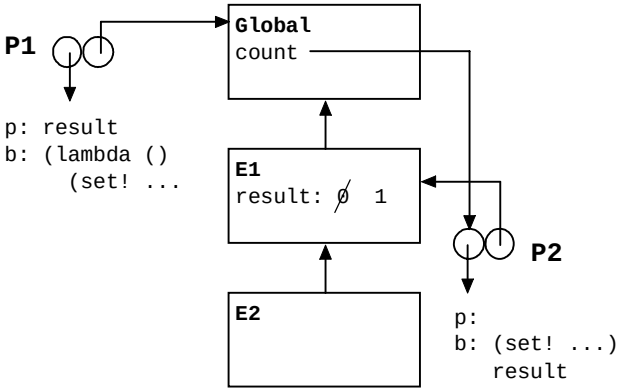


Now we finally get back to that `define`. The `define` is evaluated in the global frame, so we create a variable `count` there. Since we invoke P1, we bind `count` to whatever P1 returns. In this case, P1 returns P2.



From this we can see that the `count` procedure has a frame that maintains local state. When we call `count`, we create a new frame E2. E2 points to E1 because P2 points to E1.

There are no parameters to P2, so we then evaluate its body. (Note that we have to draw the frame E2 even though it is empty!) In the body, we assign a new value to `result` using `set!`. We see that there is no existing binding `result` in E2, so we follow the frame's pointer to its enclosing frame, E1. In E1 we find `result`, so we cross out the old value and replace it with the new one.



Exercises

Draw the environment diagrams that would result from evaluating the following expressions:

exercise 1

```
(define (foo x) (* x x))
(define (bar x) (* x x))
(define baz foo)
(define (qux x) foo)
(define (zot x) (foo x))

(let ((foo 3)
      (bar 5))
  (zot (+ foo bar)) )
```

exercise 2

```
(define (make-counter)
  (define result 0)
  (lambda ()
    (set! result (+ result 1))
    result) )

(define c1 (make-counter))
(define c2 (make-counter))

(c1)
(c1)
(c2)
```

exercise 3

```
(define (foo x) (- x))
(define (bar y) (foo (* y y)) )
(define (baz z)
  (define (foo x) (+ x 1))
  (set! foo bar)
  (foo z))

(baz 5)
```

exercise 4 (from the final exam of Fall 1997)

```
(define (kons a b)
  (lambda (m)
    (if (eq? m 'kar) a b) ))

(define p (kons (kons 1 2) 3)
```

exercise 5 (from midterm #3 of Fall 1998)

```
(define foo
  (let ((g (lambda (x) (* x 3))))
    (lambda (y) (g (- y 6)))))

(foo 5)
```

exercise 6 (from midterm #3 of Fall 1999)

```
(define x 4)
(define (baz x)
  (define (* a b) (+ a b))
  (lambda (y) (* x y)) )
(define foo (baz (* 3 10)))

(foo (* 2 x))
```

Extra

What if Scheme used **dynamic scope** instead of lexical scope?

What if Scheme used **normal-order evaluation** instead of applicative-order?

Recursion, Iteration, and the Program Stack

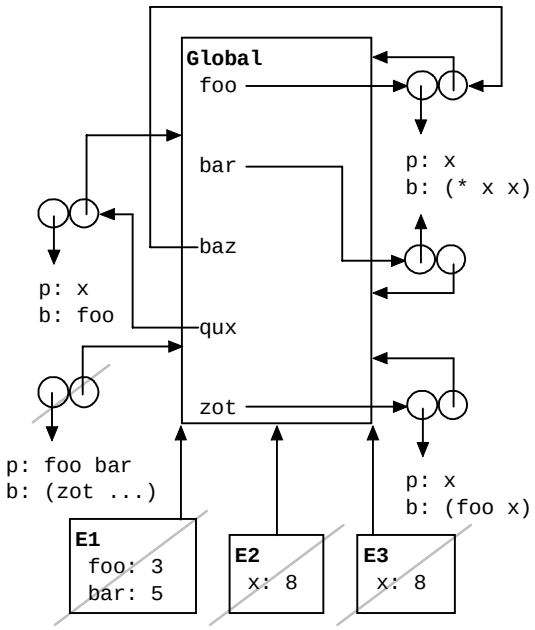
Those of you taking CS61C have encountered the **program stack**. When we call procedures, we allocate space on the stack to store its arguments and other local variables. The space we allocate on the stack—the **stack frame**—is the same as a frame in environment diagramming! When the procedure terminates, we (usually) can pop the frame off the stack and return to the previous one. (This is *not* entirely true; if there is an active procedure that points to a frame, that frame cannot be discarded yet. Most other programming languages don't have first-class procedures, so this isn't usually a problem.)

What happens with a recursive process? In a recursive process, every recursive call pushes a new stack frame on top of the current one. This is why recursive processes generate **stack overflow** errors when they run out of memory.

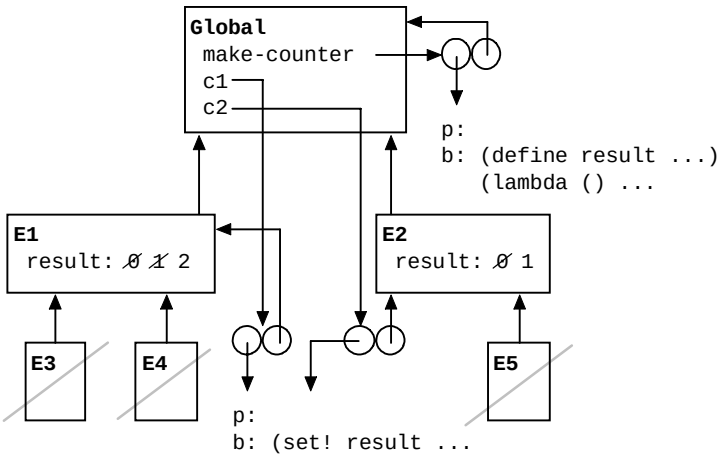
What about iterative processes? They are also recursive, but the difference is that tail-recursive procedures *replace* the current stack frame instead of pushing a new one on top. Even though we draw multiple frames in environment diagrams, they really overlap each other and occupy the same space in memory. This is why iterative processes use **constant space**.

Exercise Solutions

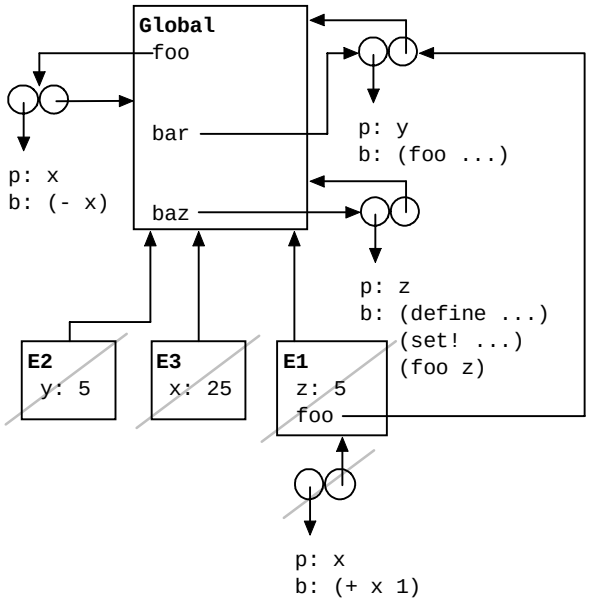
exercise 1



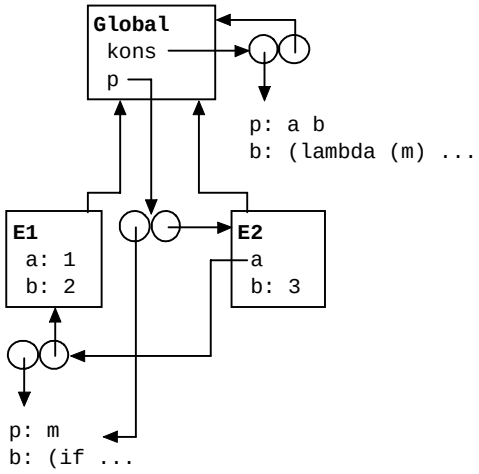
exercise 2



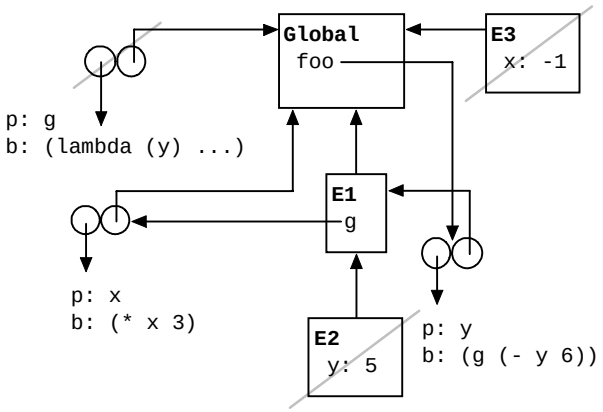
exercise 3



exercise 4



exercise 5



exercise 6