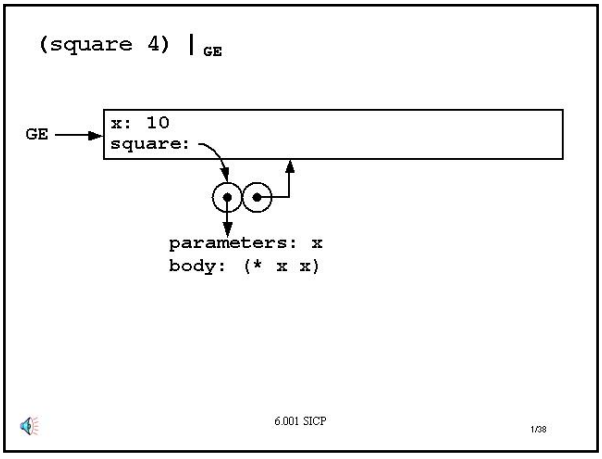
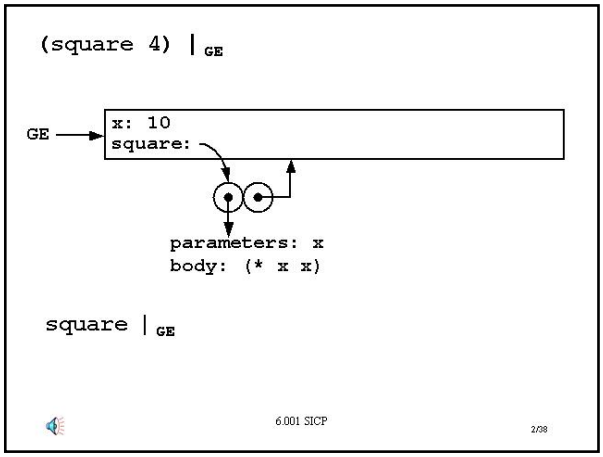
6.001 Notes: Section 12.3

Slide 12.3.1

Because this is the central part of the environment model, let's look in very **painful** detail at an example of an evaluation. In particular, let's look at the evaluation of `(square 4)` with respect to the global environment. Here is the structure we start with. Assume that `x` has already been bound to the value 4 by some `define` expression in the environment, and that we have created the definition of `square` as we just saw, it is pointing to the indicated procedure object.

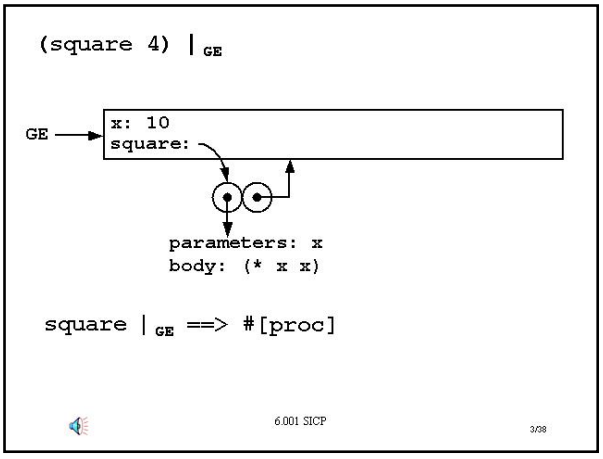
Now, we want to see is how the rules for the environment model tell us how to get the value associated with squaring 4 in this environment.

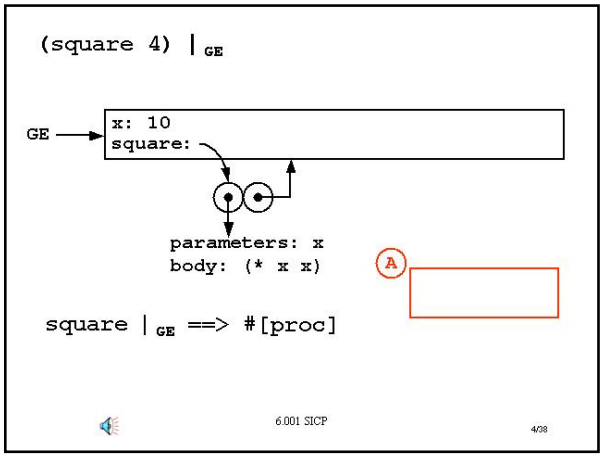




Slide 12.3.2
This is a compound expression, so first we have to get the values of the subexpressions with respect to the same environment. 4 is easy to evaluate, it's just 4. We also need to get the value of `square` with respect to the global environment.

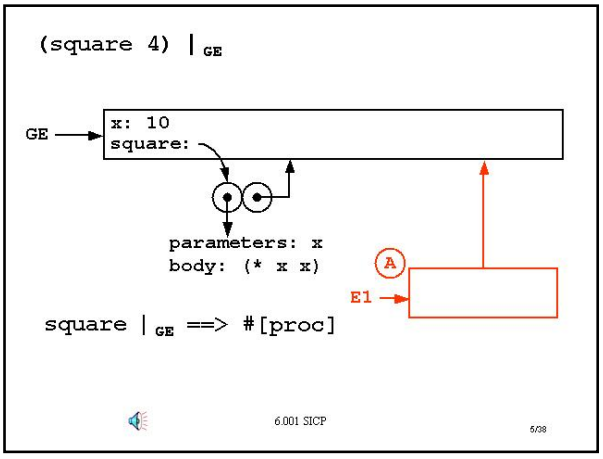
Slide 12.3.3
... and that just comes from applying the **name rule**. We look up the binding for `square` in this environment, and it simply points to that double bubble as shown.

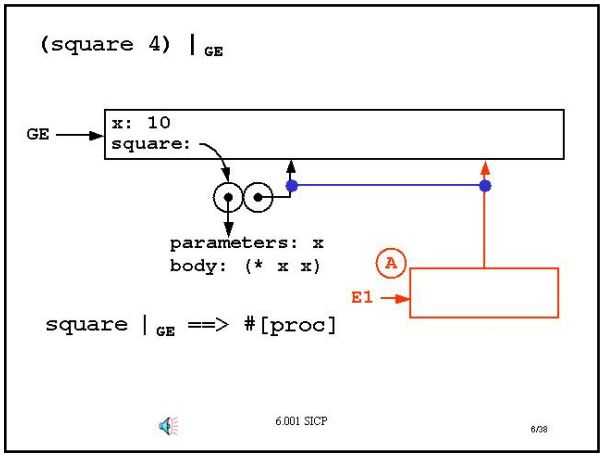




Slide 12.3.4
Aha! We are applying a procedure, one of those double bubbles, to a set of arguments, so our four-step rule now comes into play. **Step one:** create a new frame, lets call it A.

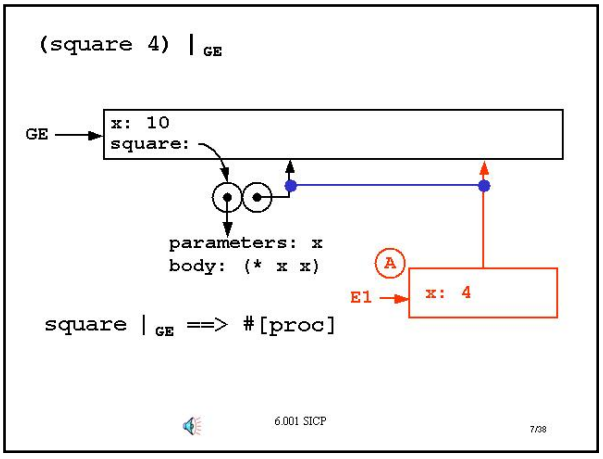
Slide 12.3.5
Step two: convert that frame into an environment, by having its enclosing environment pointer point to the same environment as the environment pointer of the procedure object.

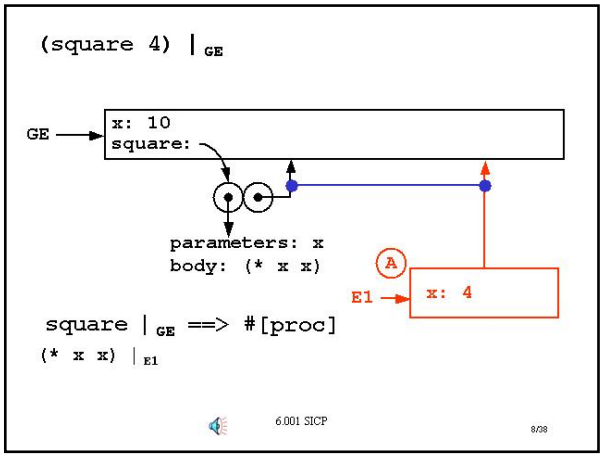




Slide 12.3.6
In fact, we can create a nice notation to keep track of this. I can link these two pointers together, to indicate that the enclosing environment of Frame A comes from the application of the indicated procedure object.

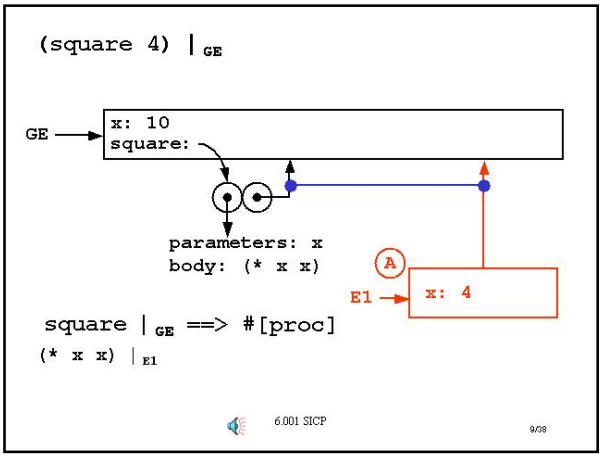
Slide 12.3.7
Step three: take the formal parameters of the procedure object being applied, in this case `x`, and bind them within that new frame to the corresponding procedure argument values, in this case `4`.

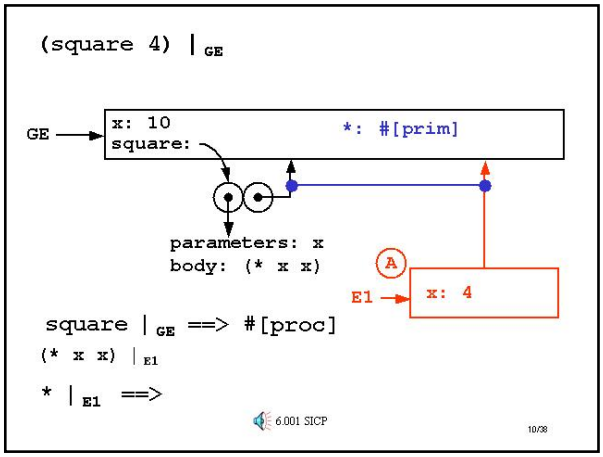




Slide 12.3.8
Now, with respect to that new environment, E1, evaluate the body of the procedure being applied. So notice, evaluating `(square 4)` with respect to one environment, has reduced to evaluating a simpler expression, `(* x x)`, with respect to a new environment.

Slide 12.3.9
Now the same rules apply as before. This is a compound expression, so we need to get the values of the subexpressions with respect to E1. We start by getting the value of `*` with respect to E1.





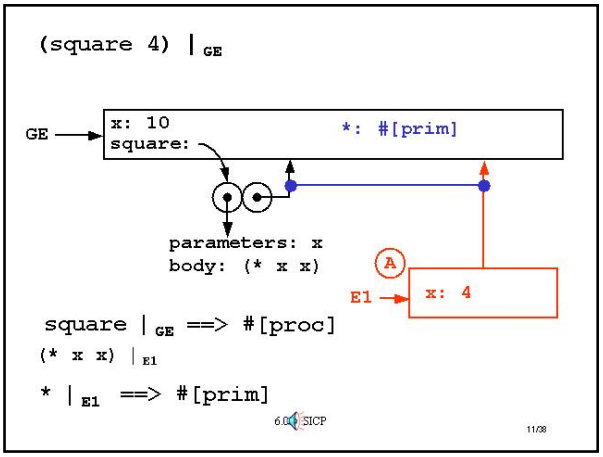
Slide 12.3.10

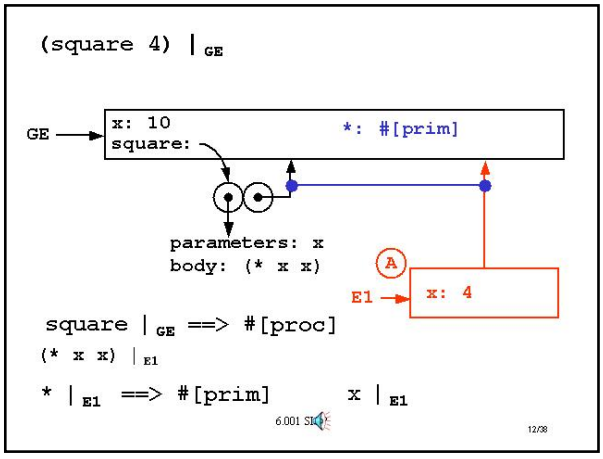
Well ... * certainly doesn't have a binding in the first frame of E1. That frame came from the application of the procedure object, and only formal parameters of the procedure are bound there. So our rule says to go up the enclosing environment pointer to the global environment and look for a binding in that environment.

We didn't tell you this, but in fact the global environment creates a bindings for all the built-in procedures. Thus, * is bound to the primitive multiplication procedure in that environment. Thus, our name rule looks up the value associated with * and returns a pointer to this primitive procedure.

Slide 12.3.11

So in this case we do get a value associated with *.



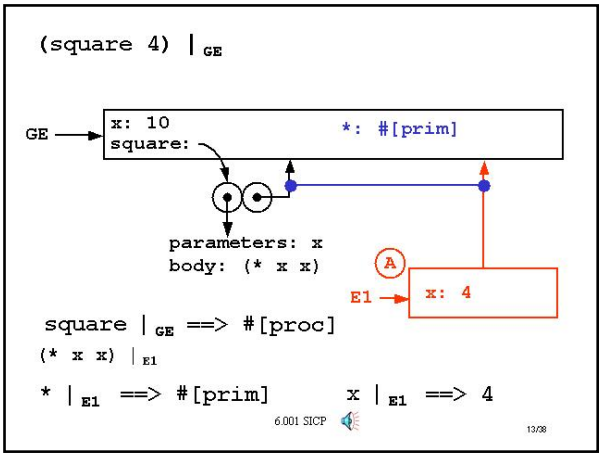


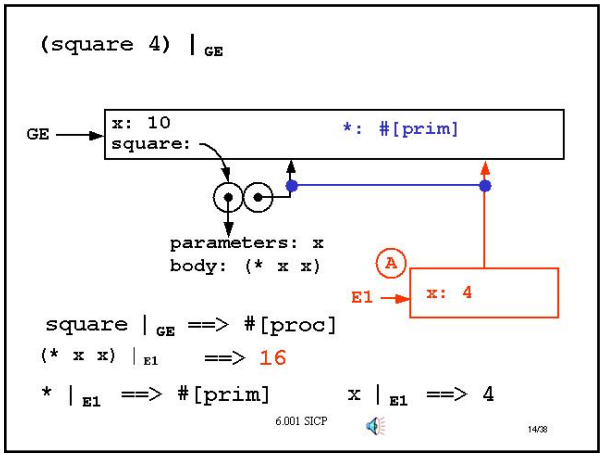
Slide 12.3.12

Remember where we were. We were getting the values of the subexpressions of (* x x) with respect to E1. We have the value of the first subexpression, so what about the value of x with respect to E1? This is just the name rule, so starting in E1, look for a binding of this variable. Of course, we find such a binding in the first frame of E1, so we get back the value associated with x in that frame.

Slide 12.3.13

In particular, we get back the value `4`. **Not** the value `10` to which `x` is bound in the global environment! Remember we start in the first frame of E1 looking for a binding for this variable. Since there is such a binding, it shadows the other binding in the global environment.





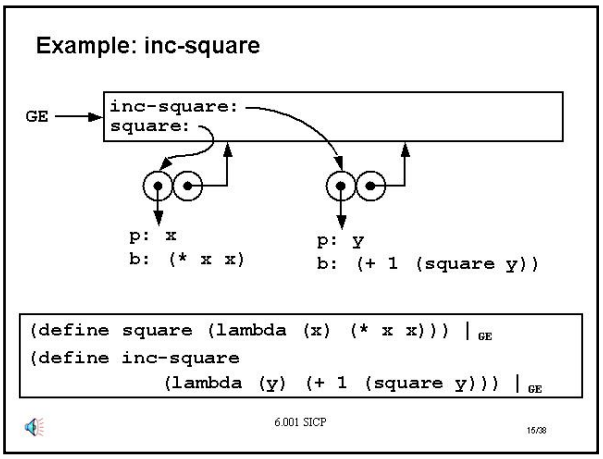
Slide 12.3.14

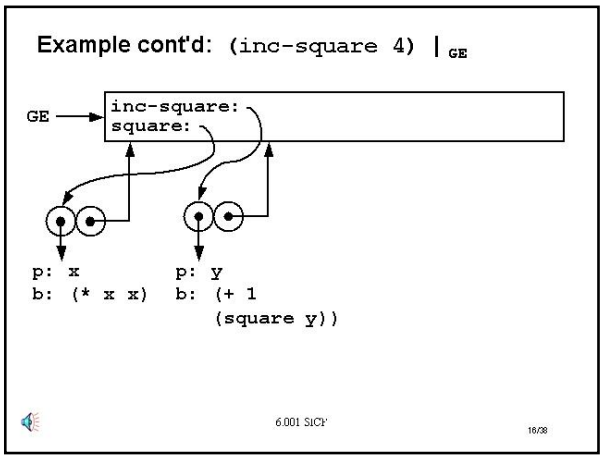
And now we complete this process. We get the value of the second `x` in the same way. Thus we are left with the application of a primitive procedure to numbers. This just returns the value `16`, which is the value of the **last** expression in the body of the procedure being applied. Thus, this is the value returned for the entire expression.

Although this was a long example, step back to notice how the mechanistic rules of the environment model simply tell us how to trace the evaluation of an expression with respect to an environment, reducing it to simpler expressions.

Slide 12.3.15

Now, let's be slightly more daring! Having seen the application of a simple procedure like `square`, let's look at something that involves a little more work. In particular, let's assume that we have defined `square` and `inc-square`, as shown in this environment structure.



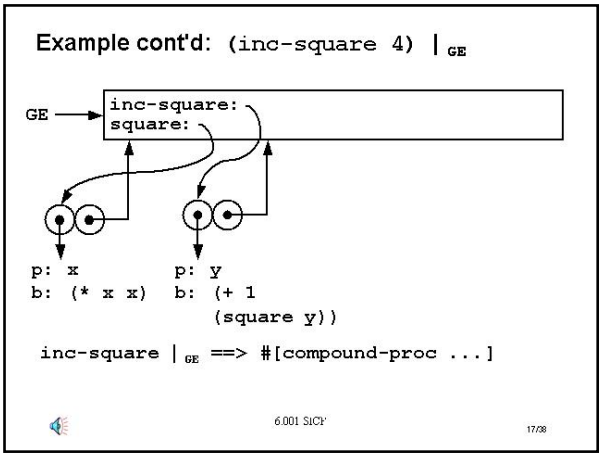


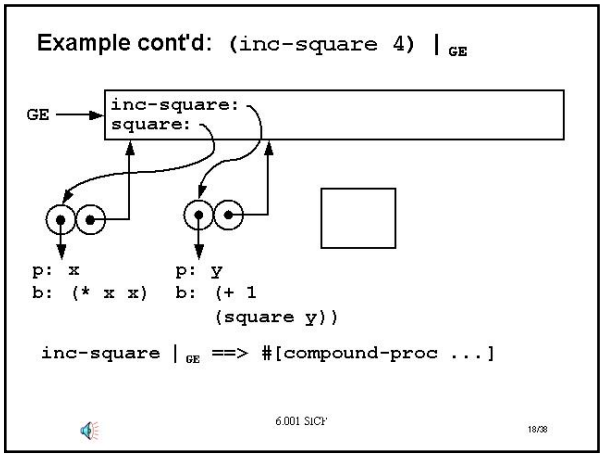
Slide 12.3.16

So lets evaluate `(inc-square 4)` with respect to the global environment, and here is the environment structure corresponding to that environment.

Slide 12.3.17

As in the previous case, this is a compound expression, so we need to first evaluate the subexpressions with respect to the same environment. The value of `inc-square`, by the name rule, is just the double bubble pointed to by that variable.



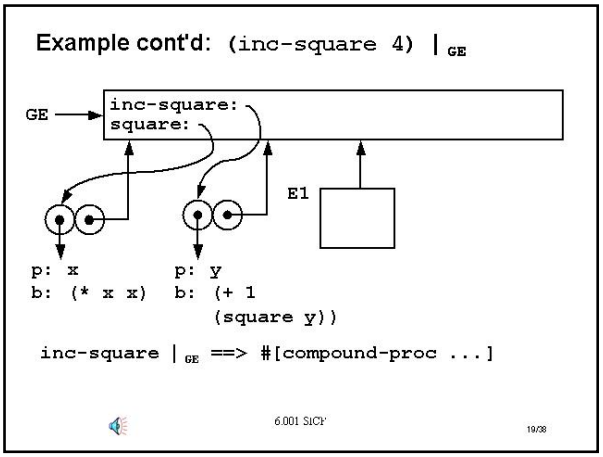


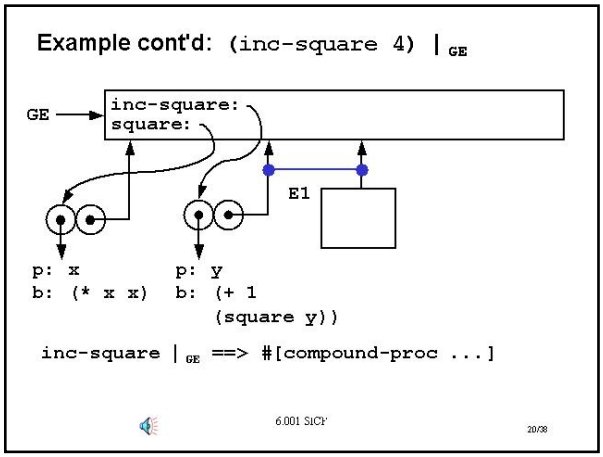
Slide 12.3.18

And as we saw before, our four-step rule kicks in. **Step one:** create a frame.

Slide 12.3.19

Step two: turn it into an environment, by having the enclosing environment pointer of the frame point to the environment specified by the procedure that is being applied...



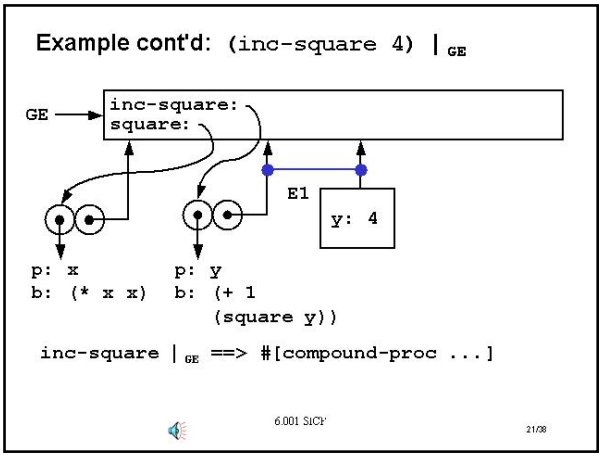


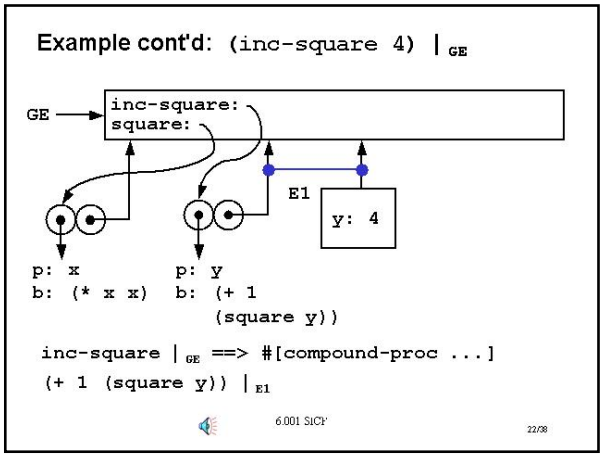
Slide 12.3.20

... and that we know is specified by the second part of the double bubble of the procedure being used.

Slide 12.3.21

Step three: take the formal parameter of this procedure and create a binding for it in the new frame, to the value of the argument passed in.





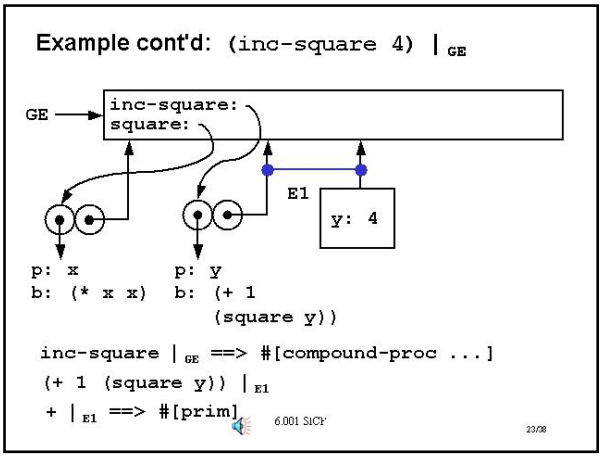
Slide 12.3.22

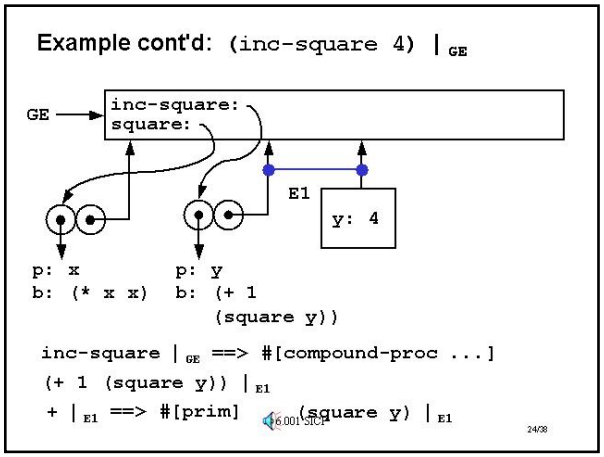
Step four: take the body of that procedure object and evaluate it with respect to this new environment, that is, `(+ 1 (square y))` with respect to E1.

Slide 12.3.23

Again, notice how we have reduced the evaluation of one compound expression with respect to one environment to the evaluation of a simpler compound expression with respect to another environment.

As before, to evaluate this compound expression, we first need the values of the subexpressions. The value of `+` with respect to E1 is determined by the name rule, chasing up the environment chain from E1 to the global environment to get the primitive addition operation. The value of 1 is just a self-evaluation rule.



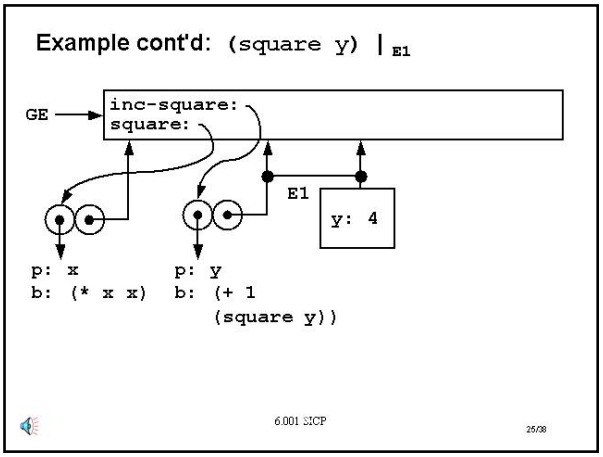


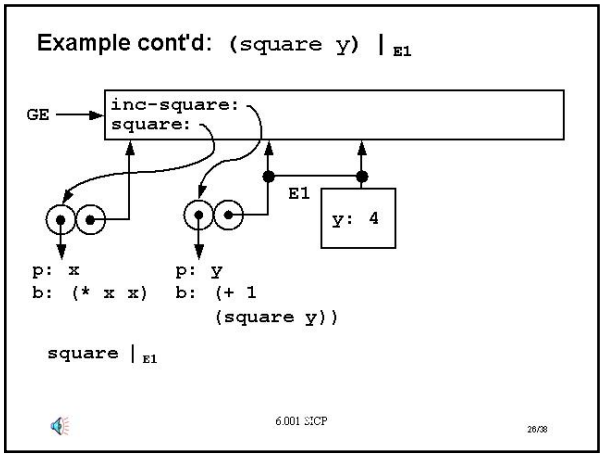
Slide 12.3.24

So all we have left to do is get the value of `(square y)` with respect to E1. This is again a compound expression, being evaluated with respect to this new environment.

Slide 12.3.25

To complete this evaluation, we need to get the value of `(square y)` with respect to E1, and here is a clean version of the environment structure developed so far.



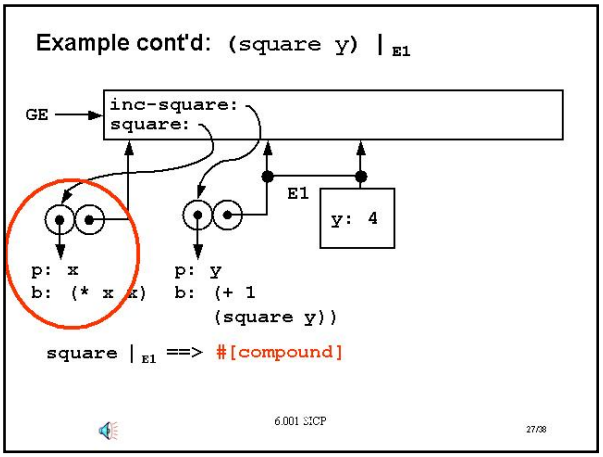


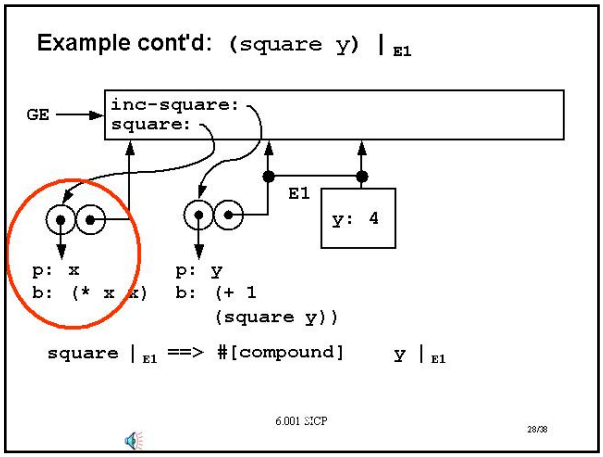
Slide 12.3.26

Well, let's do it in steps. We first get the values of each of the subexpressions, **now** with respect to E1. First, what is the value of `square` with respect to E1? This is a little different than last time. We start in E1. Since there is no binding for `square` there, we go up the environment chain to the global environment, where we find the binding, and thus return ...

Slide 12.3.27

... the appropriate procedure.

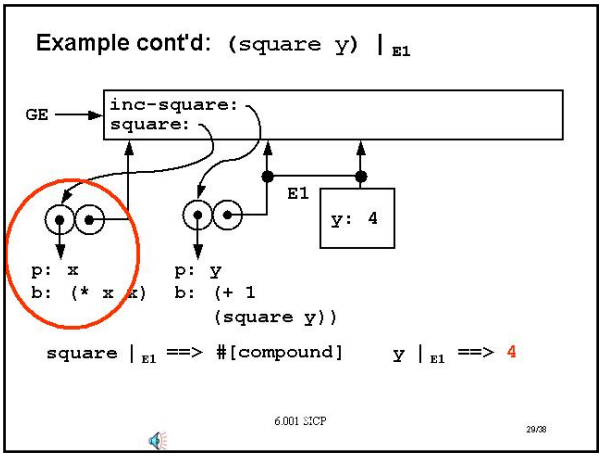


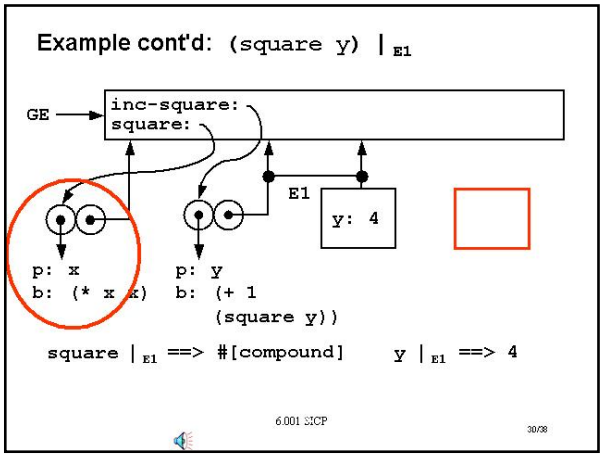


Slide 12.3.28

The second subexpression is `y` and we need its value with respect to E1.

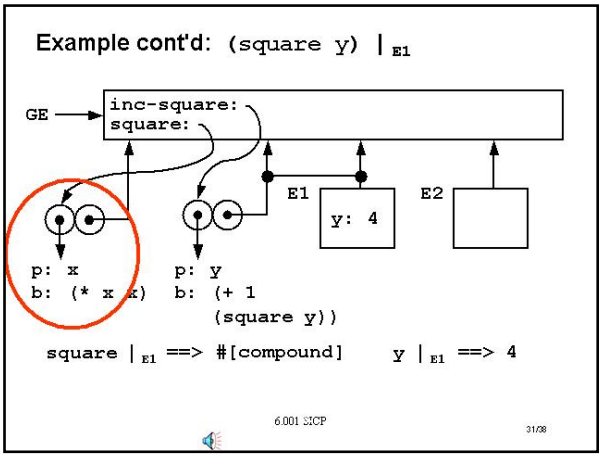
Slide 12.3.29
... and our name rule says to simply look it up with respect to this environment chain starting in E1. This, of course, returns the value 4.

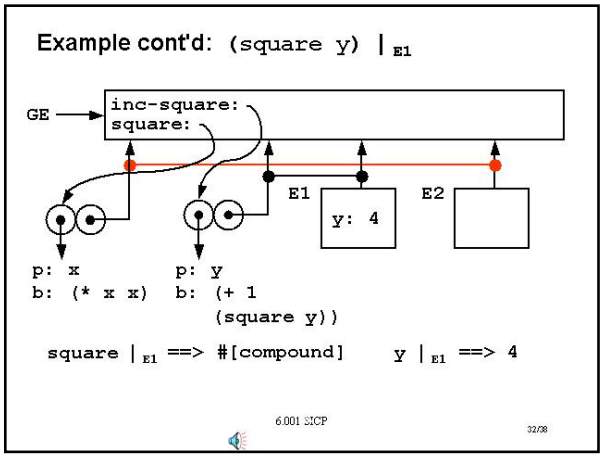




Slide 12.3.30
So we are set for our big rule again. We have the application of a procedure, one of those double bubbles, to a set of values. Step one says: drop a frame.

Slide 12.3.31
Extend that frame into an environment by having its enclosing environment pointer match that specified by the procedure object...

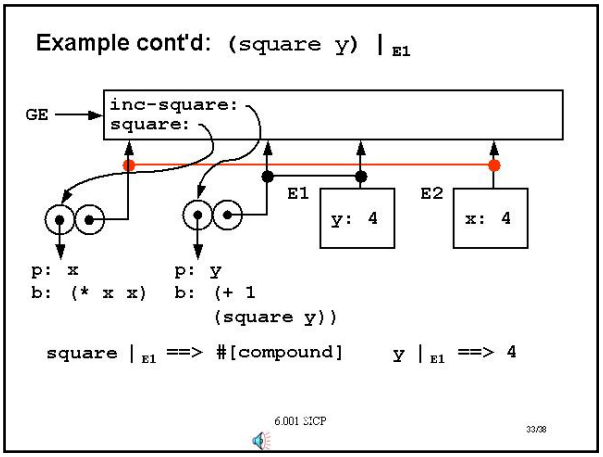


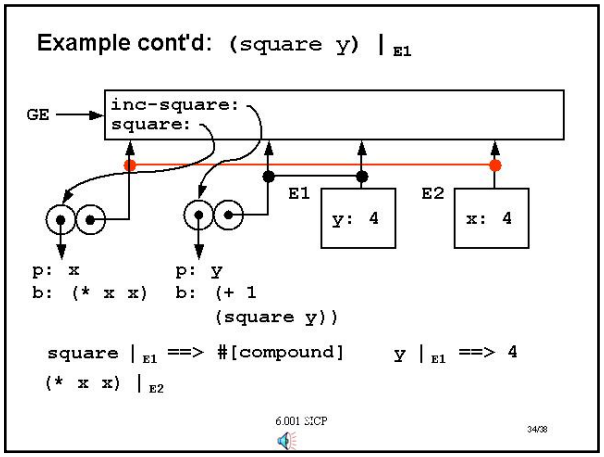


Slide 12.3.32
... which says we want this one. Notice an interesting point here. This new environment E2 is scoped by the **global environment**, not by E1. You might have thought this should be E1, because that was where we were doing the evaluation, but remember that rule says the enclosing environment is specified by the procedure being applied, **not** by the frame in which we are doing the evaluation.

Slide 12.3.33

Step three: create a binding, that is, take the formal parameter of the procedure, X, and bind it in that frame to the value passed in, 4.



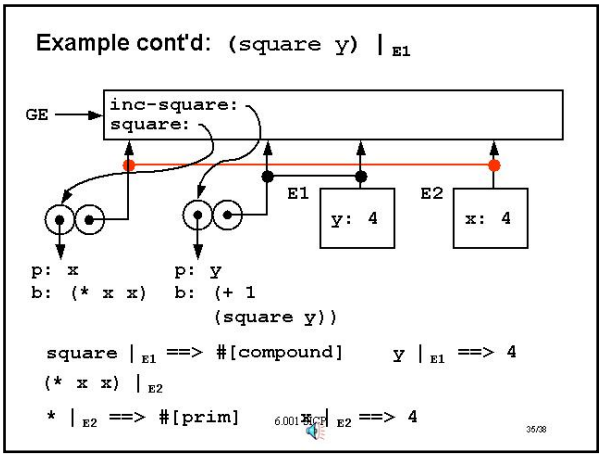


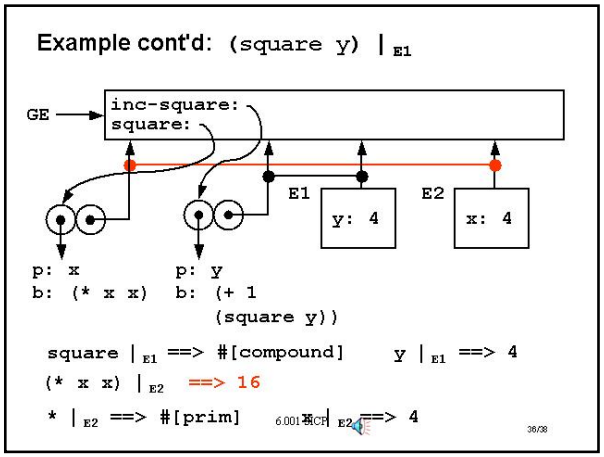
Slide 12.3.34

And step four: with respect to that new environment, evaluate the body of this procedure, (* X X). So we have reduced this evaluation to multiplying X by itself within environment E2.

Slide 12.3.35

So now we are almost done. The values of each of these subexpression with respect to E2 are obtained by applying the name rule, starting in the first frame of E2. We then apply the primitive multiplication procedure to the value of 4 and 4.





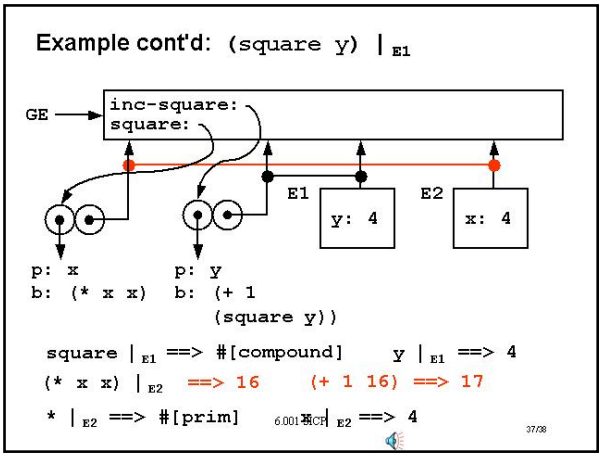
Slide 12.3.36

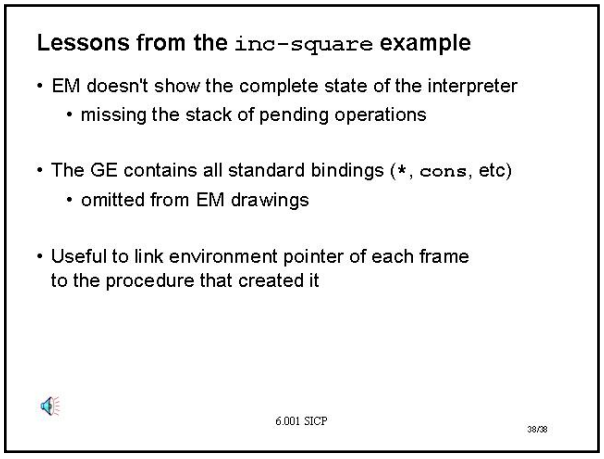
That of course just returns 16.

Slide 12.3.37

And remember what started all of this. That value then gets passed back out to where we were when we started this diagram, which is adding `1` to the value returned by this procedure application, finally resulting in `17`.

There are a lot of details here, but the goal was to show how the rules of the environment model very mechanistically specify exactly the order in which to evaluate expressions, and how to look up the bindings of variables in the appropriate environment in order to make sure expression has a legal value.





Slide 12.3.38

So here are the key points to take away from this example. The main point is to see how the rules for the environment model specify **almost** everything we need to know in order to understand how expressions are evaluated. It doesn't quite do all of it, in particular, it doesn't specify the pending operations, as we saw in the last example.

The other two points to notice are summarized on the slide.

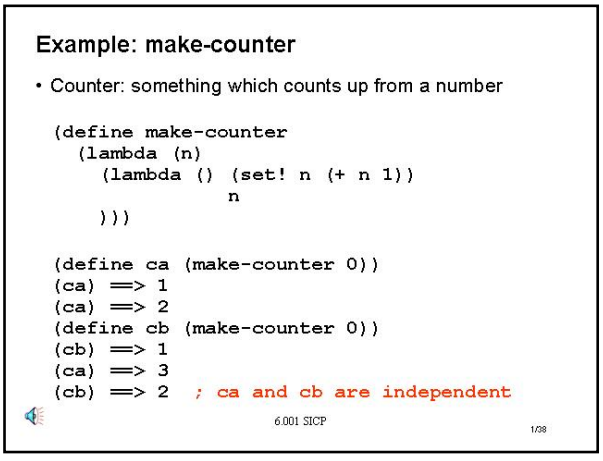
6.001 Notes: Section 12.4

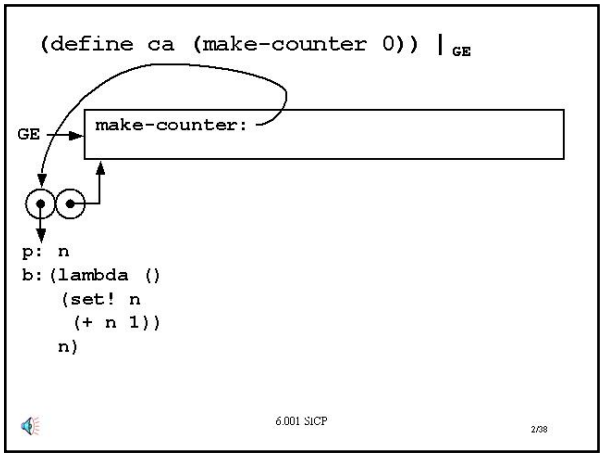
Slide 12.4.1

Now that we are reasonably comfortable with the environment model, let's go back and tackle the problem that started us off on this discussion. Remember this example from the beginning of lecture?

Counter, or something created by `make-counter`, was an object that should count up from a number. Every time we applied that procedure of no arguments, we would get as output the next number in sequence. We want to understand how evaluating this **same** expression could give rise to **different** values at different times. And how evaluating the same expression could give us different objects, with independent behavior.

Let's see if the environment model helps explain this computation.





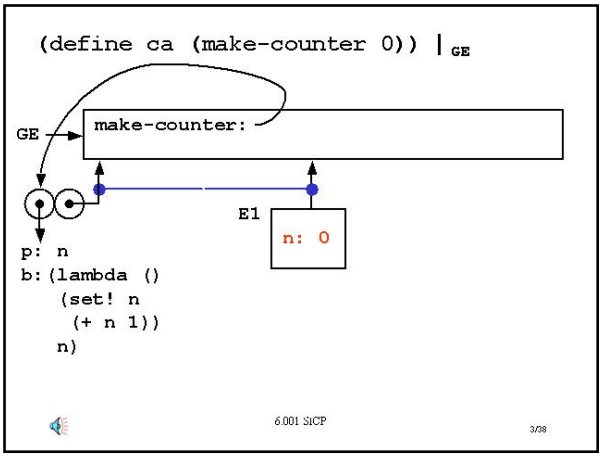
Slide 12.4.2

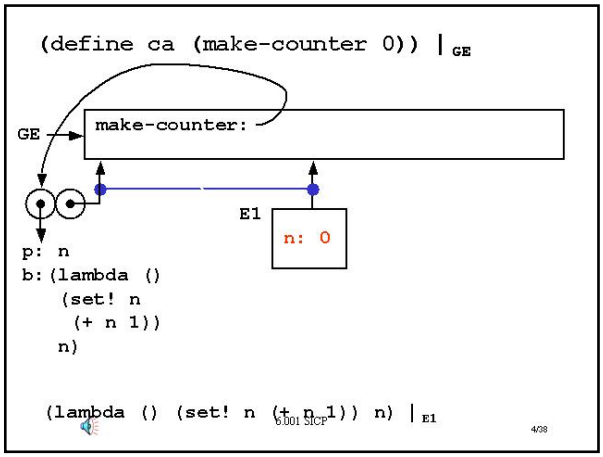
Although there will be a lot of details in this example, you should try to follow the high level view of how the model explains the computation.

First, lets evaluate this definition of `ca` to be the value of applying `make-counter` in the global environment. We have shown our structure for the global environment, with a binding for `make-counter` to a procedure object, as would be obtained when we evaluate the definition on the previous slide.

Slide 12.4.3

To do this, we know we need to get the value of applying `make-counter` to 0. `Make-counter` is a procedure so the next set of stages is something we have already seen. We will drop a frame, extend it into an environment by scoping it by the same environment as the environment pointer of the procedure, and within that frame, bind the formal parameters of the procedure to the arguments passed in. Thus we get the structure shown, which is exactly like our previous procedure applications.



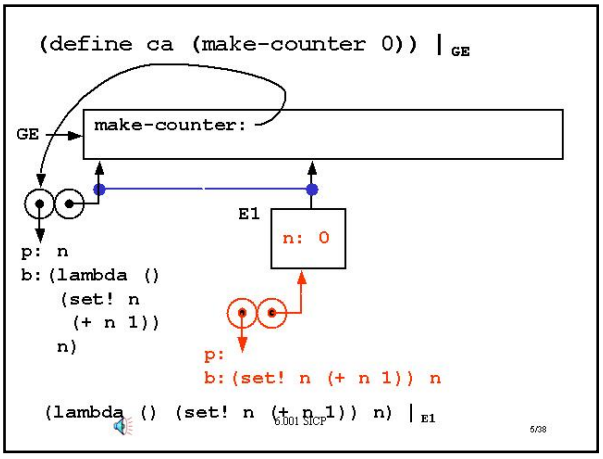


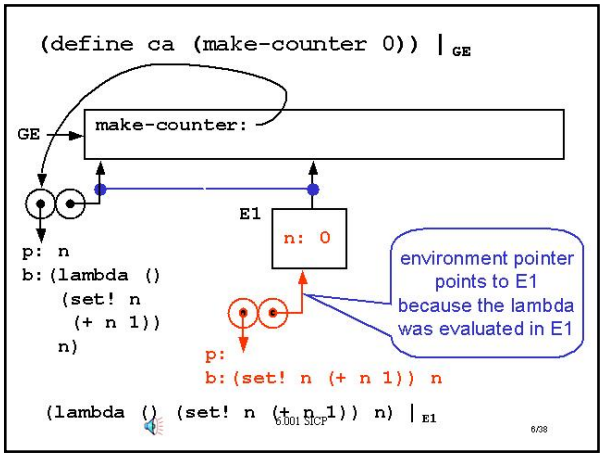
Slide 12.4.4

Now watch carefully. We now have an evaluation of the body of that procedure with respect to an environment, and what is that body? It is, itself, a `lambda`, with no parameters and a body that does a mutation and returns a value.

Slide 12.4.5

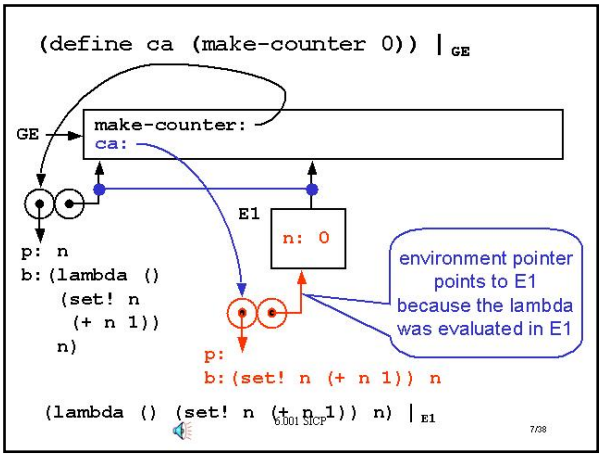
So we just apply our rule for `lambda`s. We create a double bubble for the procedure object created by the `lambda`. The code part of the object is just the formal parameter (in this case nothing) and the body (in this case `(set! n (+ n 1))` and then `n`). The key issue is where does the second part of the double bubble go? What is the environment pointer we want here?





Slide 12.4.6

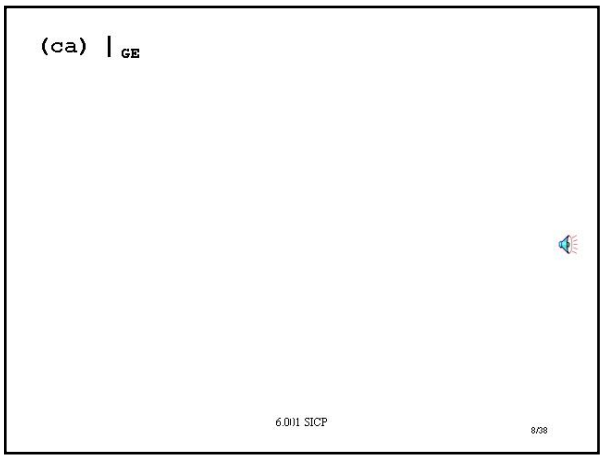
Right! It points to E1, because this is the environment in which we were evaluating the `lambda` expression. Notice that this is a **different** structure than we have seen before. For the first time, we have a procedure object whose environment pointer points to a frame or environment other than the global environment. And in a second we are going to see why that is crucial.



Slide 12.4.7

Now remember, that red (that double bubble) is the value actually returned by the evaluation of the body of the procedure we applied. So that is the value returned by `(make-counter 0)`. Therefore, we can complete our definition. The binding for `ca` up in the global environment, since that is where we were evaluating the `define`, is now created, and that variable points to the object returned by applying `make-counter`, which is that procedure object.

This is a useful structure. The variable `ca`, which is available to us in the global environment, points to a procedure, much as earlier things did, but this procedure has nested within it an internal environment. Its environment pointer points to an environment that is scoped relative to the global environment.

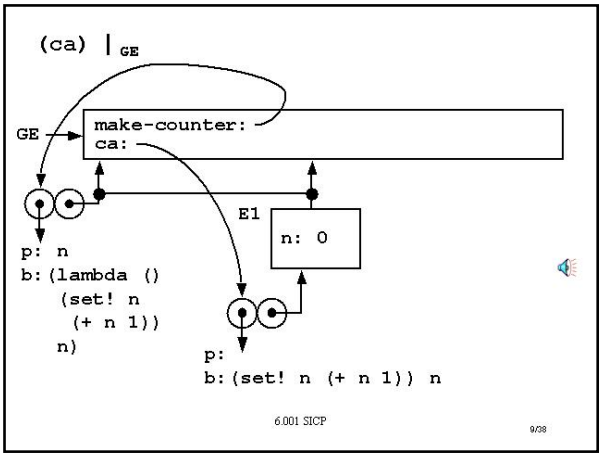


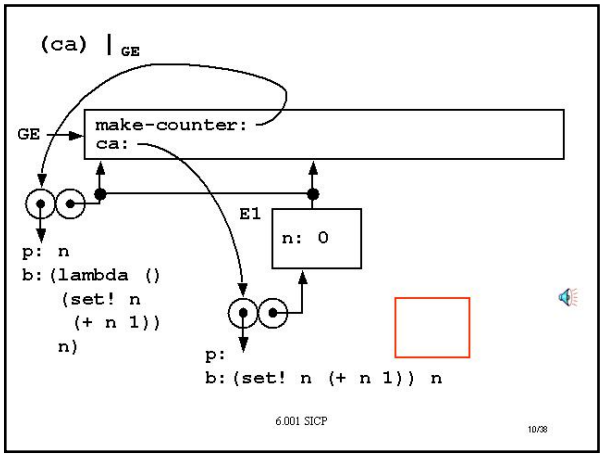
Slide 12.4.8

So having created this procedure object associated with `ca`, let's look at what happens when we apply it, when we evaluate it with no arguments.

Slide 12.4.9

Here's a recap of the environment structure. We have the global environment with a binding for `ca`, which points to a procedure object whose environment pointer points to a new frame, or environment, E1. This frame is scoped by the global environment, but contains within it its own local variable `n`.



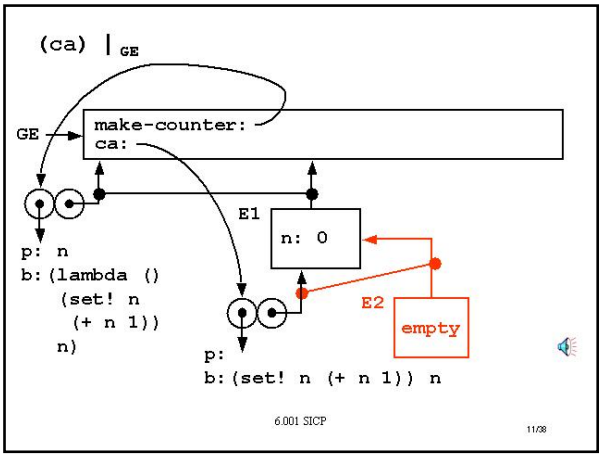


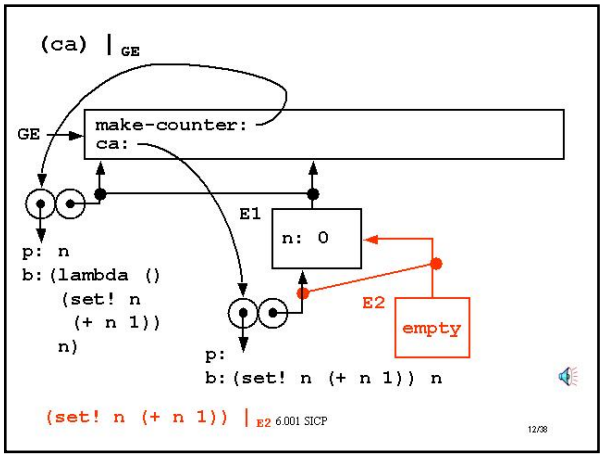
Slide 12.4.10

The value of `ca` is just that procedure object, so we are going to apply a procedure, and we know what rules to use. Drop a frame.

Slide 12.4.11

Within that frame, bind the formal parameters of this procedure. There aren't any, so there is nothing to put in the frame.





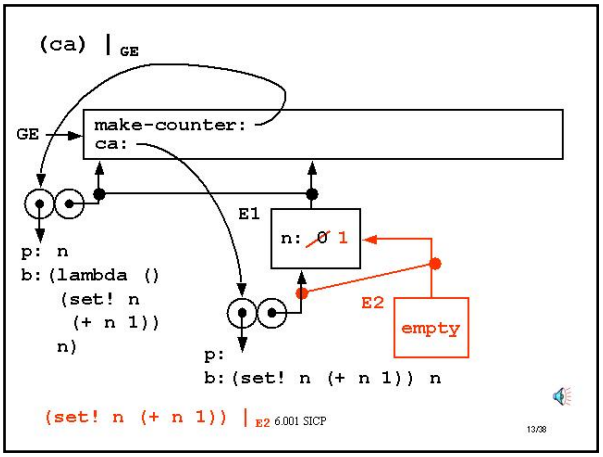
Slide 12.4.12

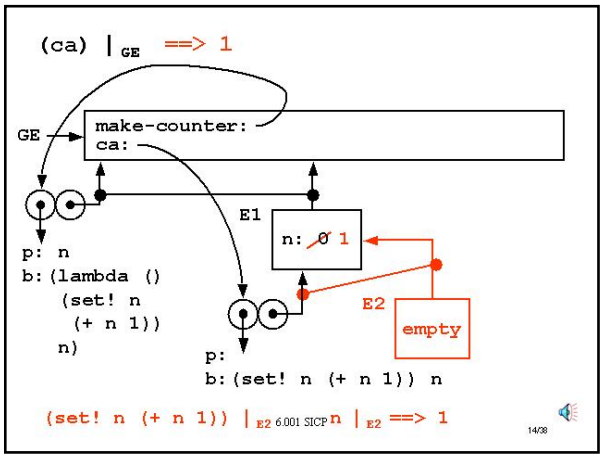
And relative to this new environment, evaluate the body of the procedure, which says we are going to evaluate `(set! n (+ n 1))` and then `n`, with respect to E2. Notice the structure here. We now have a frame, E2, that points to a frame E1, that points to the global environment.

Slide 12.4.13

First, we know what a `set!` does, but let's look at this carefully. First, we will get the value of `(+ n 1)` with respect to E2. This means we look up the value of `n` with respect to E2. Since it is not bound there, we go up the enclosing environment pointer, to find the binding in E1. This gives us the value `0`. Thus we add 0 to 1, to get 1.

Notice how that local frame has capture `n` for us. Given that, we can now evaluate the `set!` with respect to E2. This says, starting in E2, trace up the environment chain until we find a binding for `n`, which we find in E1. We then change **that** binding to the newly computed value, which is 1.





Slide 12.4.14

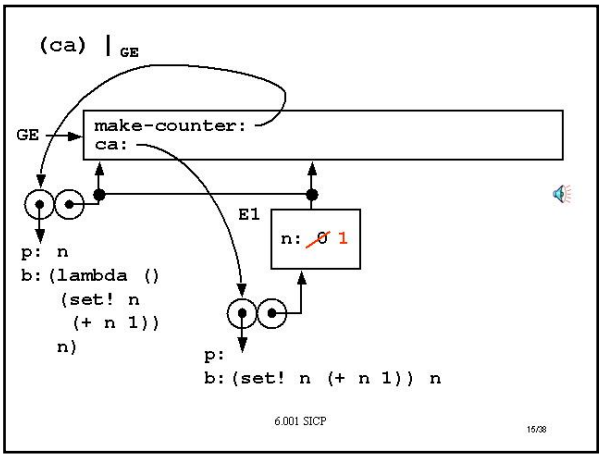
Having evaluated that part of the body of the procedure, we then evaluate that next part, `n`, with respect to E2. Using the same rules, we chase up the chain of environment pointers until we find a binding for `n`, returning the value 1 as the value for this expression. Since this is the last expression in the body of the procedure, this is also the value we return for the application of the procedure itself.

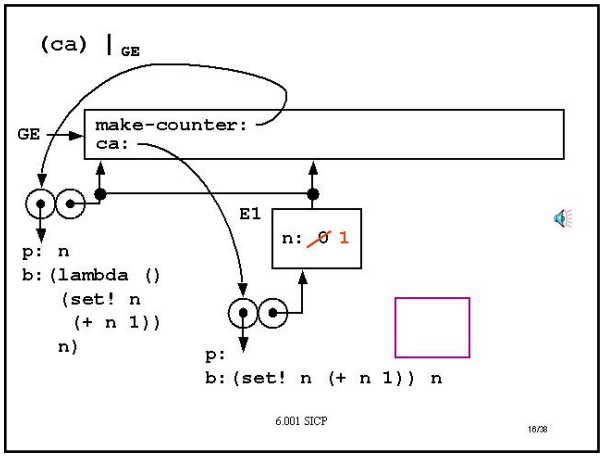
The key thing to note here is how that local frame, E1, captures some state information that is accessible only by this procedure. Calling `ca` gives us the ability to get this value of `n`, change or mutate it, and then return that value.

Slide 12.4.15

Now let's see what happens if we call `ca` again. We should expect to see the behavior of the value incrementing, so let's evaluate `(ca)` again with respect to the global environment.

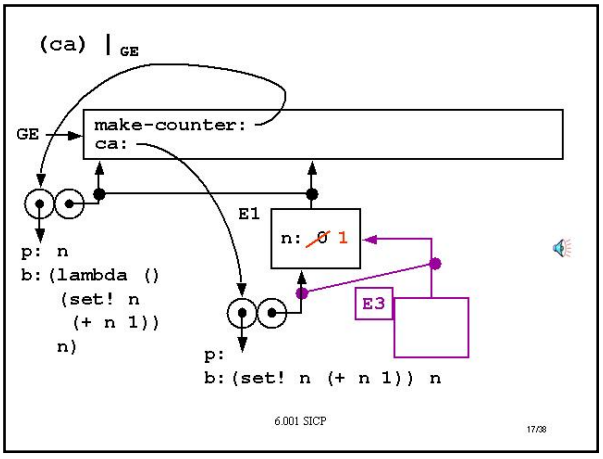
Shown is a replication of the environment structure left from the last evaluation, with `n` having been mutated from 0 to 1 as part of that process.

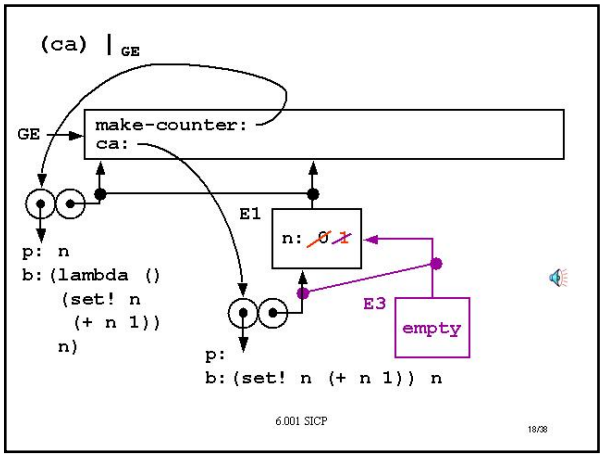




Slide 12.4.16
The value of `ca` is still the same procedure, and since we are applying a procedure, we use our four step rule. We drop a frame.

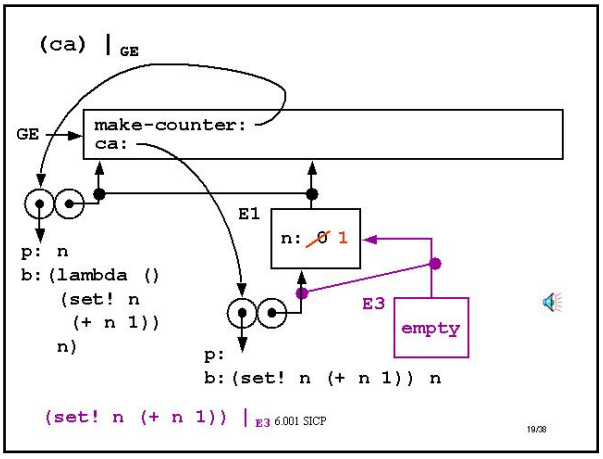
Slide 12.4.17
We then scope it with the same environment pointer as the procedure object being applied, thus E1 as we did before.

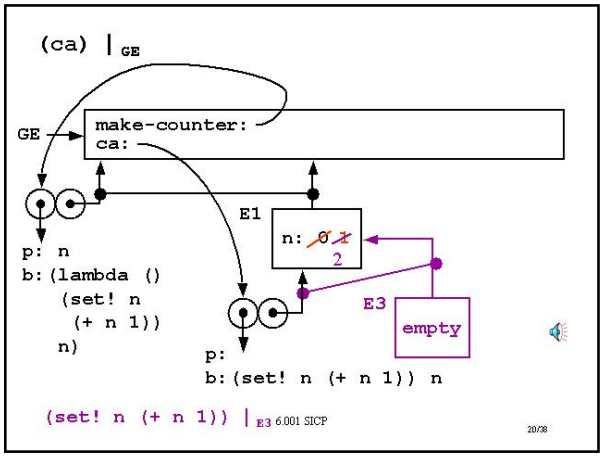




Slide 12.4.18
There are no parameters to bind in this frame.

Slide 12.4.19
So evaluating `(ca)` with respect to the global environment reduces to evaluating the body, `(set! n (+ n 1))` with respect to this new frame E3.





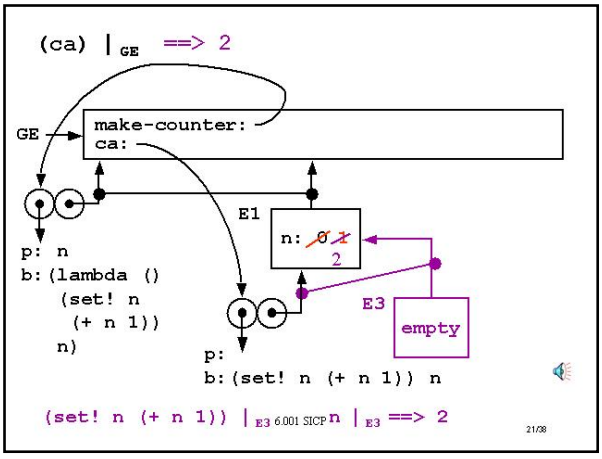
Slide 12.4.20

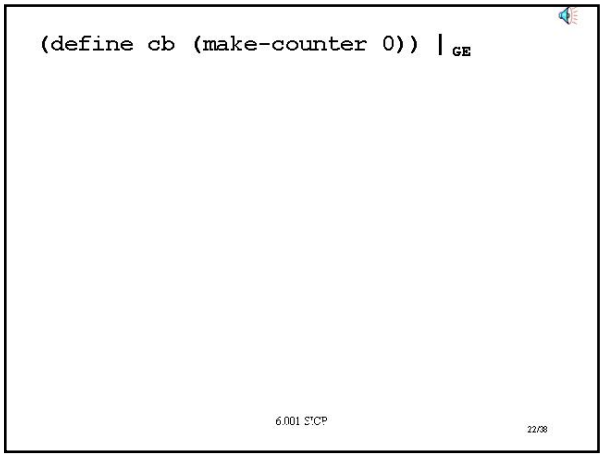
We know what should happen in this case. We just did this! The evaluation will again mutate the value of `n` to be one more than its current value, thus changing the value from 1 to 2 in this frame.

Slide 12.4.21

And having evaluated that part of the body, we now take the second part, `n`, and evaluate that with respect to E3. Chasing up the frame pointers finds the binding in E1, and returns that value, `2`. Since this is the value of the last expression in the body of `ca`, this is the value returned by the whole thing.

Thus we see how this local piece of state allows us to have a procedure, which when evaluated in successive turns, returns a different value.





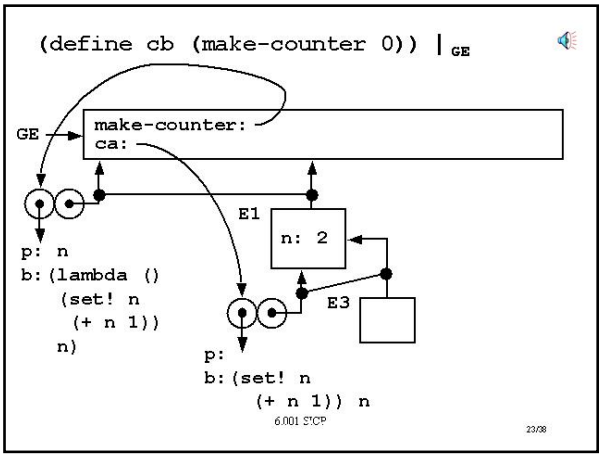
Slide 12.4.22

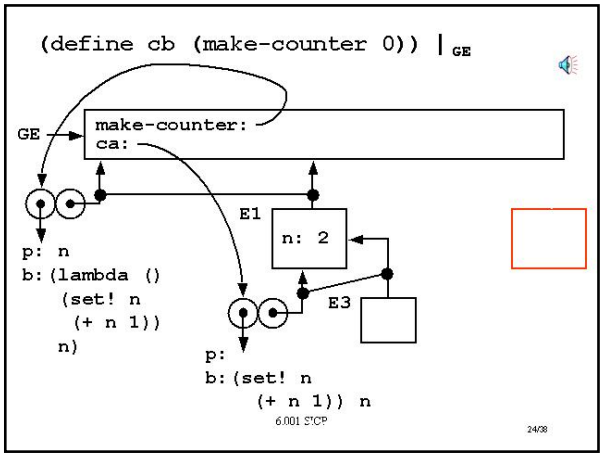
So our environment model helps us understand how one counter, `ca`, can have some local state which it can keep mutating. Thus, it can return a different value each time it is applied.

What happens when we call `make-counter` again, starting from 0, but giving it a different name.

Slide 12.4.23

And just to recap, here is the environment structure we have to this point. There is a variable `ca` pointing to the procedure shown, with local state of `n` equal to 2.



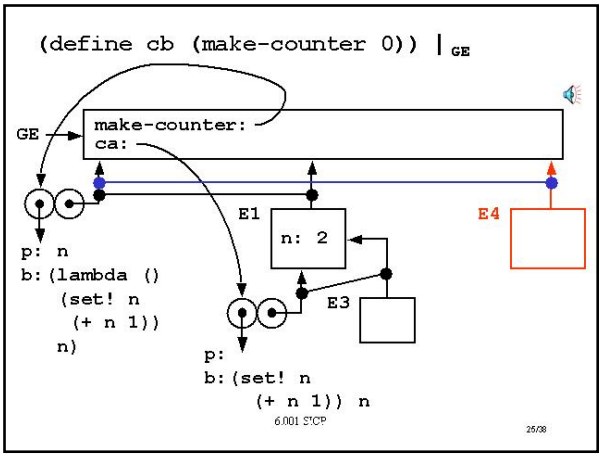


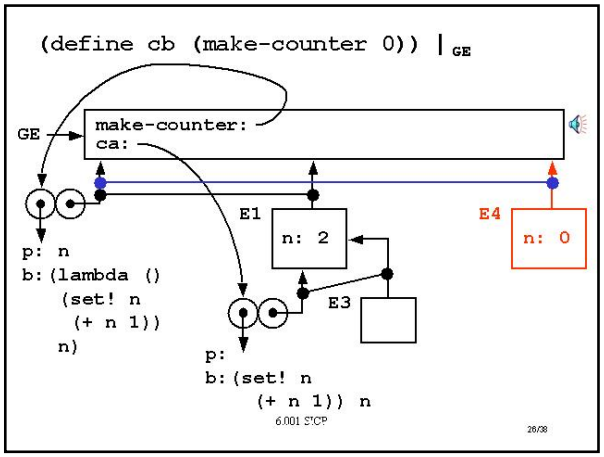
Slide 12.4.24

Since we want to apply `make-counter` to `0`, we first look up the value of `make-counter` with respect to the global environment. Since it is a procedure, which is being to applied to `0` our four-step rule applies. Step one: drop a frame.

Slide 12.4.25

Step two: convert it into an environment by scoping that frame with the same environment pointer as the procedure being applied. Note that this means in this case pointing to the global environment.



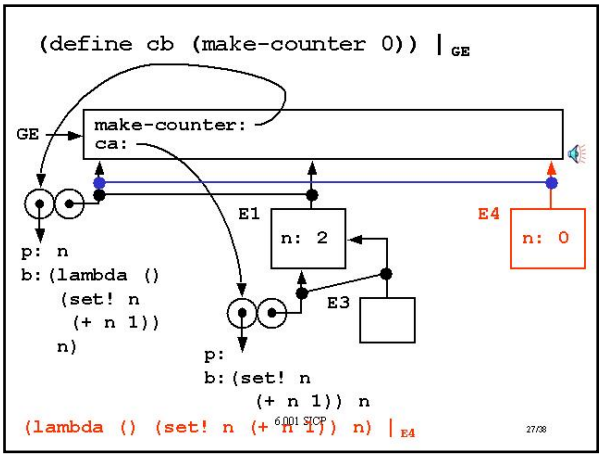


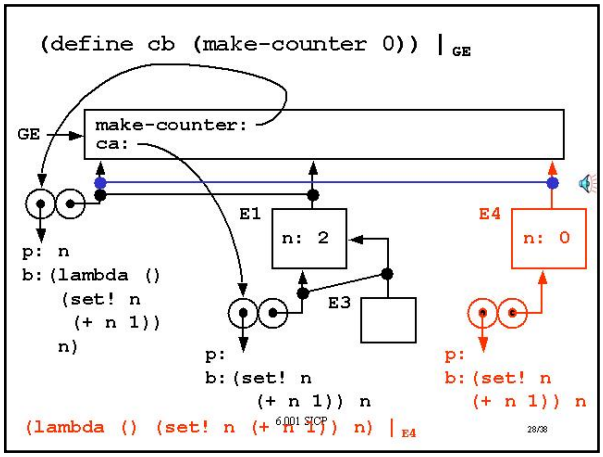
Slide 12.4.26

Step three: bind the formal parameter of this procedure, `n`, to the value passed in, `0`. Note that this is a **different** `n` than the one we had before. This `n` lives in environment E4. The `n` we had for `ca` lives in environment E1, so we have different bindings for the same name in different environments.

Slide 12.4.27

Relative to this environment, evaluate the body of the procedure. Remember the procedure we are applying, `make-counter`, so we are going to evaluate that `lambda` expression with respect to E4.





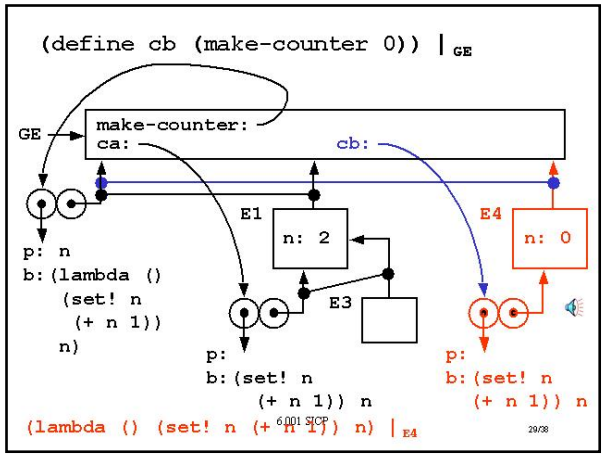
Slide 12.4.28

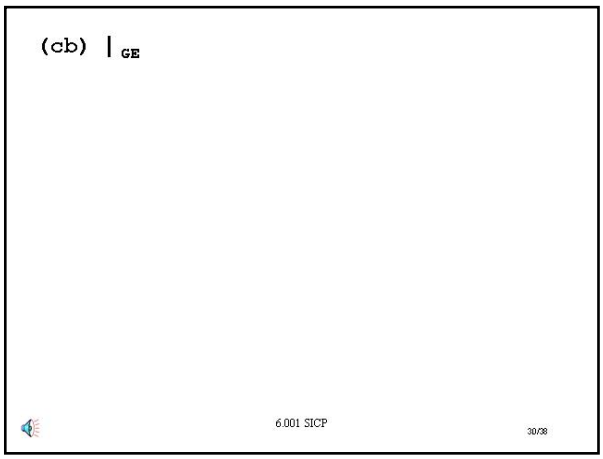
And we know what evaluating a `lambda` should do. It creates a procedure object, a double bubble, the code of which has no parameter and a body that is the same as the earlier body we created for `ca`. The key point is: where does the procedure object's environment pointer point to? To the environment in which the `lambda` is being evaluated, thus to E4. Note that this procedure thus has a different scoping than the one created for `ca`. It is going to capture E4, with **that** version of `n`, which is different than the one we had before.

Slide 12.4.29

And finally, that procedure object is the value returned by applying `(make-counter 0)`, and therefore the definition creates a binding for `cb` up in the global environment to that new procedure object.

Look carefully at the structure we have here. We have `ca` pointing to a procedure object that inherits a frame E1. We have `cb` pointing to a similar procedure object that inherits a frame E4. They have different bindings for the parameter `n`, and that is going to allow these things to behave differently.



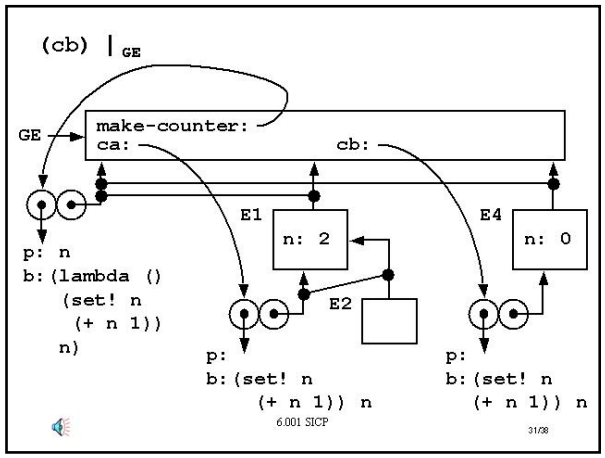


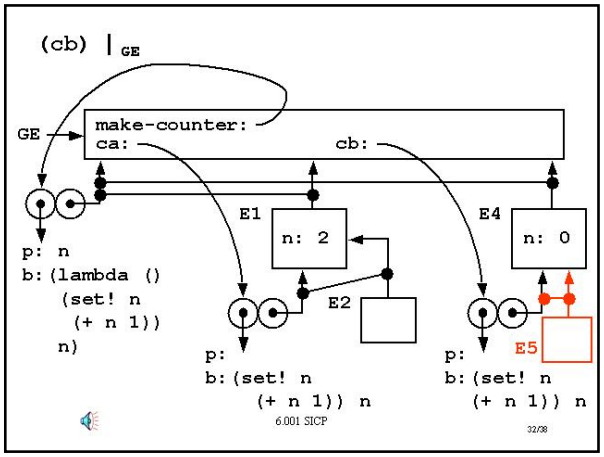
Slide 12.4.30

So lets finish our understanding of this process by looking at what happens when we evaluate `(cb)>` within the global environment.

Slide 12.4.31

Here, again, is the environment structure we have so far. We have a global environment with a binding for `ca` to one procedure object with its own frame, and a binding for `cb` to another procedure object with its own frame. What happens when we apply `cb`?



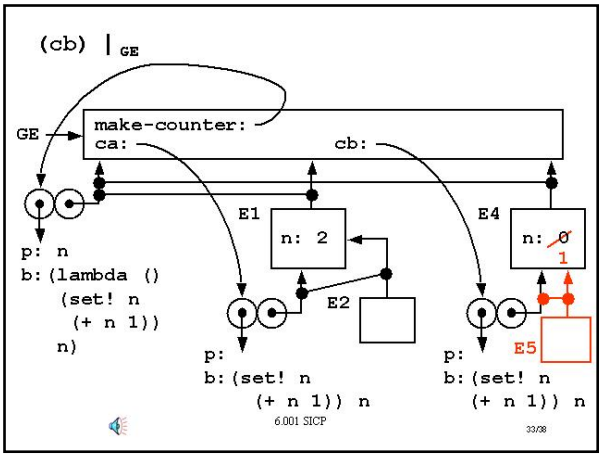


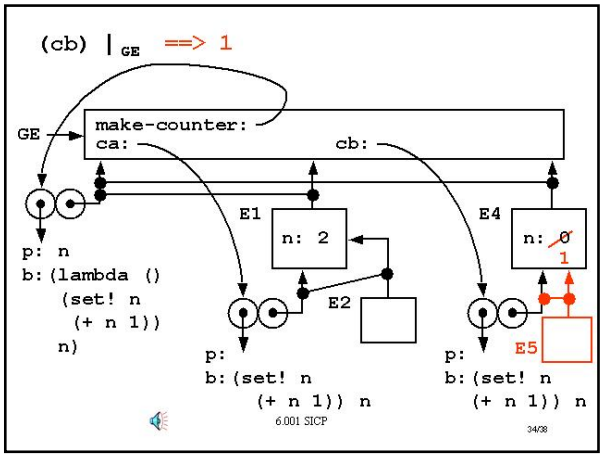
Slide 12.4.32

Well, the rules make it very clear. We take the procedure object associated with the binding for `cb`, and we apply it. This will drop a frame, scope it by using the same environment pointer as the procedure object, which says E5 will be scoped by E4. **Not** E1, **not** global environment, but **E4**.

Slide 12.4.33

This says that using exactly the same reasoning as before, the value of `n` that will get mutated is the one that is seen from E5, namely the one sitting in E4. So it gets changed by 1, and then we return that value, of 1.





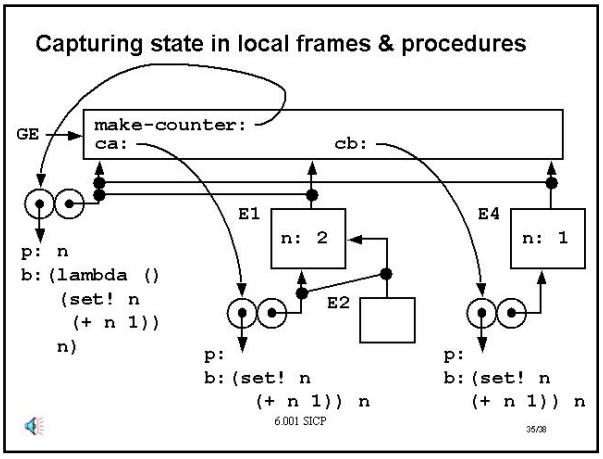
Slide 12.4.34

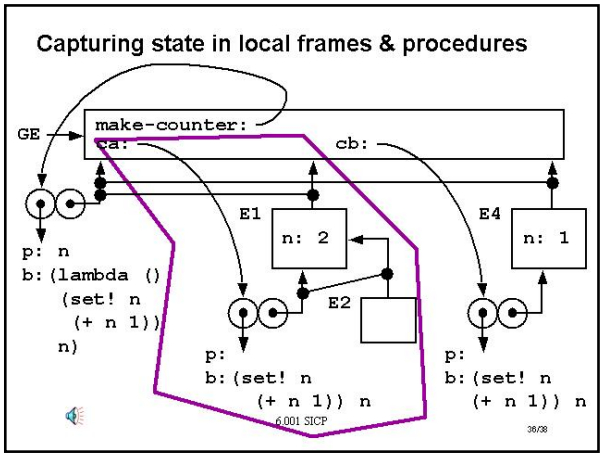
This says the value of the overall expression is just 1.

Slide 12.4.35

So what was the point of this long, drawn-out, exercise? Part of it was to let you see how the rules for the environment model evolve. They specify what happens during a computation, they specify how expressions get meanings assigned to them through a very mechanistic set of rules.

But the second point was to let us understand things that involve mutation, especially mutation associated with procedures. So think about what happened with our little counter example. Here is the environment model we had.





Slide 12.4.36

And notice that there is a structure associated with it. What does `ca` look like? It's a name, bound in the global environment, that points to a procedure that has as its environment pointer, a pointer into a frame, in this case E1, that has some local state in it.

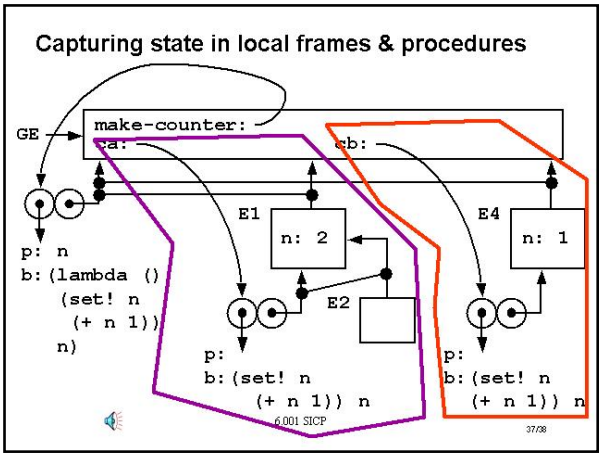
The only way we can get at that value of `n` is through a procedure whose environment pointer, through some chain of frames, points into that frame. In this case, that is only through `ca`.

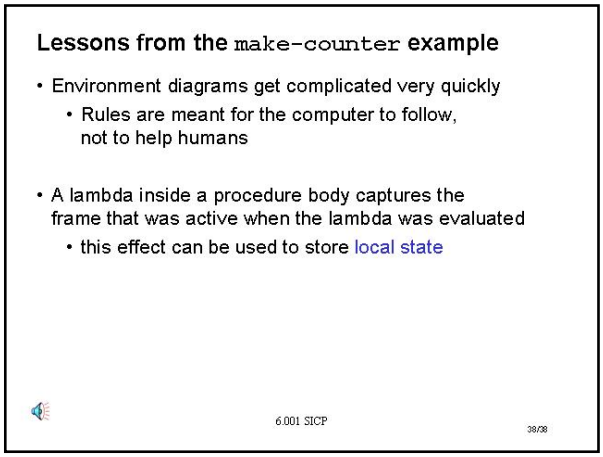
So this structure or pattern is a common pattern.

Slide 12.4.37

And in fact when we evaluated `make-counter` a second time and gave the result returned by it the name `cb`, we created a similar pattern, but now with its own local state. Thus, each of these names specifies a procedure that has associated with it some information that belongs only to that procedure.

This kind of framework, that is, procedures that capture local state and can manipulate local state, is going to be a very useful programming tool, as we are going to see in the next few lectures.





Slide 12.4.38

Here is a summary of the key points to take away from this exercise.