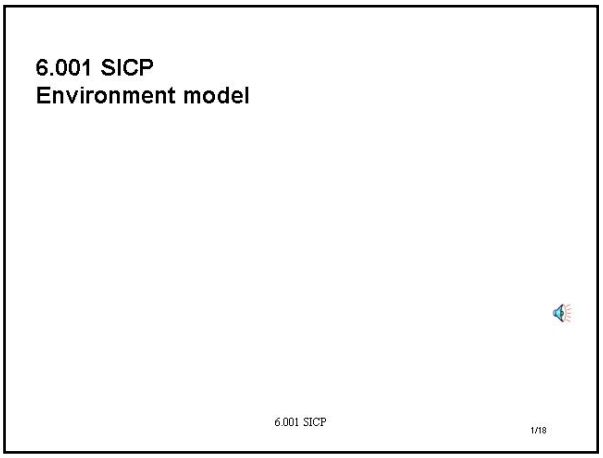
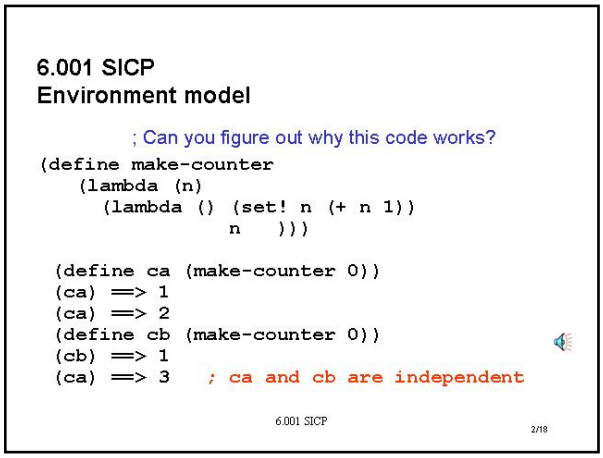
6.001 Notes: Section 12.1

Slide 12.1.1



In the last lecture, we introduced mutation as a component of our data structures. We saw, for example, that `set!` was a way of changing the value associated with a variable in our system, and we saw that `set-car!` and `set-cdr!` were ways of changing the values of parts of list structure.
Now, several important things happened when we introduced mutation. First, we introduced the notion of **time** and **context** into our interpretation Scheme. The order in which things were evaluated now mattered in terms of changes in returned values. As a consequence, secondly we shifted from a functional programming perspective to a more state-based programming perspective, a point to which we will return. Third, we unfortunately introduced the opportunity for bugs and errors in our system, since shared objects allow the mutation of one to affect the value of the other, another point to which we will return. And finally, fourth, we broke the substitution model.
This last point needs to be addressed, and indeed in addressing it, we will also address many of these other points. In this lecture, then, we are going to replace the substitution model with a stronger model, that incorporates the substitution model as a piece of it, but also accounts for state, time, context and mutation.

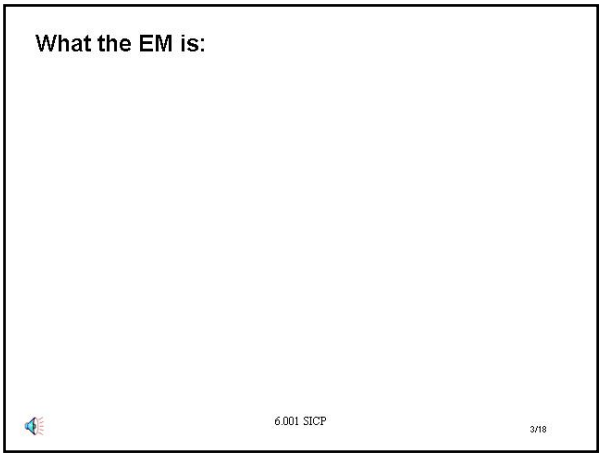


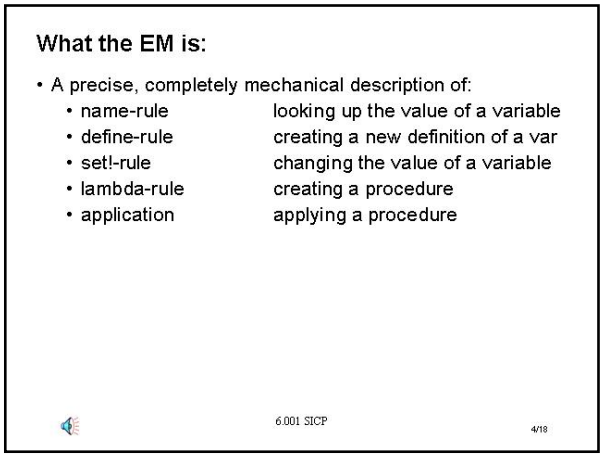
Slide 12.1.2

To stress this idea that the substitution model no longer holds, consider the following example. Why does this code behave in this manner?
We can see that `make-counter` is a higher-order procedure, that is, it returns a procedure as its value. Suppose we use it to create a counter, called `ca`. Now if we all this procedure (or evaluate its application) several times, we see that is behavior is to count starting at 1, increasing the returned value by one each time. Thus the standard substitution model (or functional programming model) no longer holds, because the **same** expression is being evaluated each time, but a **different** value results, depending on when we evaluate the expression.
If we create a second counter, `cb`, we end up with different structures. As shown in the last two examples, these two counters are independent, with `cb` now counting from 1, but `ca` continuing to count from its place. Thus, even though these objects were created by evaluating the same expression, they do not share any common state. So, our substitution model is broken, caused by the introduction of mutation. We thus need a better model that would explain how this code behaves.

Slide 12.1.3

To do this, we are going to introduce a better model of evaluation, known as the **environment model**. This model will explain things covered by the substitution model, as well as new effects such as mutation. And it is going to lead us towards a much better understanding of the evaluation process.



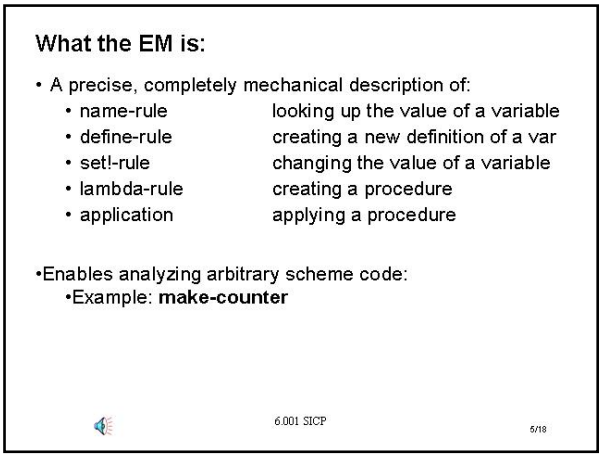


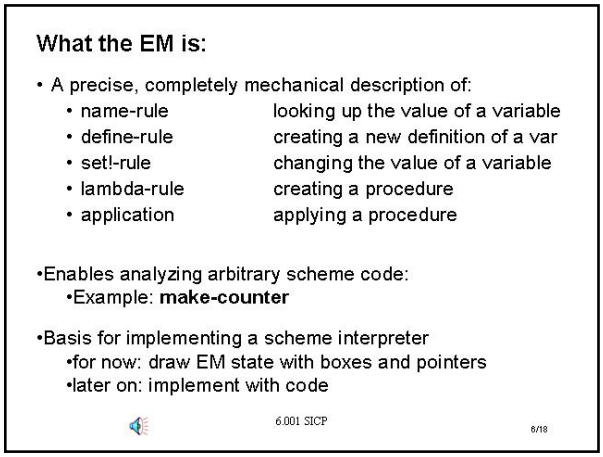
Slide 12.1.4

So what is the environment model? For now, think of it as a **very precise, very mechanical** description of a set of rules for determining the values associated with expressions in Scheme. Thus, similar to what we saw several lectures ago, we will have rules for dealing for **getting** the value of a variable, a rule for **creating** a value of a variable, and a rule for **changing** the value of a variable. We'll also have a rule for **creating** procedures, and a rule for **applying** procedures.

Slide 12.1.5

So our goal will first be to determine the specific rules for the environment model for these kinds of expressions. Once we have set out those details, we will be able to explain the evolution of the evaluation of arbitrarily complex expressions, such as the example we just saw. More importantly, by using the model to analyze code, we will learn how to associate particular coding choices with their effects of the evaluation process. This will allow us to reverse the process. By deciding the behavior we want in our evaluation process, we will be able to work backwards to determine the code components needed to achieve that behavior.



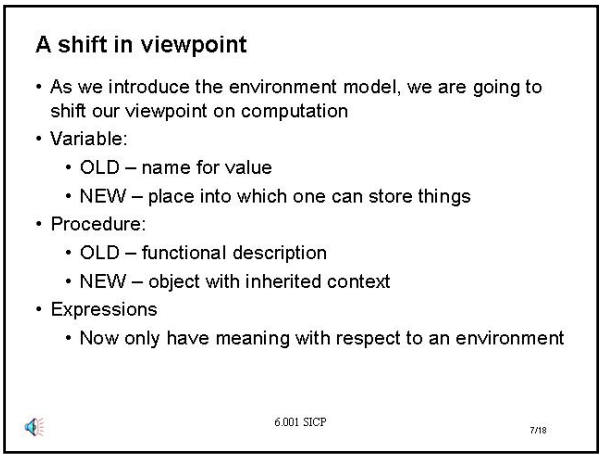


Slide 12.1.6

There are two main reasons for doing this. The first is simply to understand the effect of our design choices in creating procedures, for example how mutation will affect behavior. Ultimately, however, we are going to see that this environment model serves as a blueprint for building an actual interpreter for Scheme, that is, the actual mechanism for evaluating Scheme expressions inside a machine.

As a consequence, for now we will use abstract representations for our environment model, but we will eventually see how this leads to a very mechanical method that can be implemented to actually build a Scheme system.

Slide 12.1.7



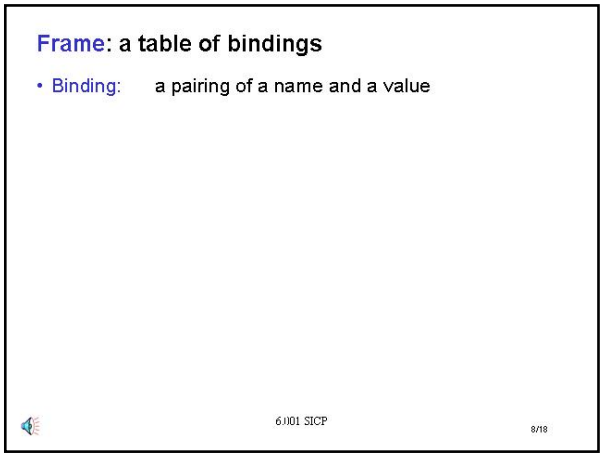
As we build up the pieces of our environment model, a key thing to keep in mind is that we are about to shift our viewpoint on what constitutes the basic units of computation.

Until we introduced mutation, we could think of a variable as just being a **name** for a value. That was exactly how it behaved, in the functional viewpoint of computation. Now we are going to change that viewpoint. We are going to think of a variable as a **place** into which one can store things, a name for a cubicle into which things can be placed.

The second change we are going to make deals with procedures. Up until now, we could really think of a procedure as a functional description of some computation. We stressed this idea that evaluating the same expression gave rise to the same value, it behaved as a **mapping** from input values to output values, like any ordinary function would. Now, we are going to change that viewpoint. We will instead think of a procedure as an object, with an inherited context. That context tells us how to interpret symbols in that computation, which will change our overall view of computation.

And finally, we are going to change the way we think about expressions. We now will say that an expression only has meaning with respect to a structure called an **environment**, something that we are about to define. This means that an expression now inherits its value, by inheriting information about what was occurring when they were created.

So we ask that you keep these ideas in mind as we build our new model of computation. Variables are now places into which one can store things, procedures are now objects with an inherited context, and expressions have meaning only with respect to an environment.

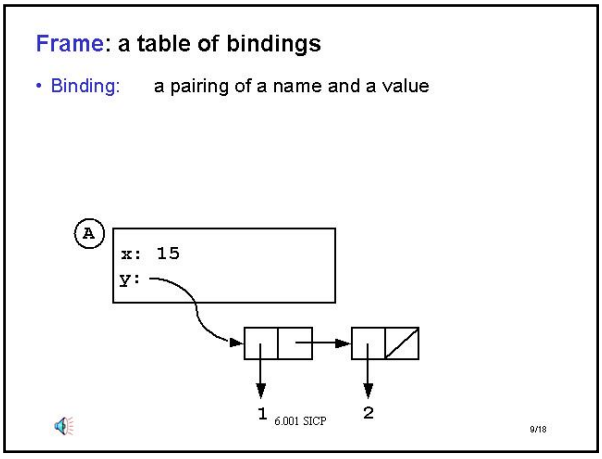


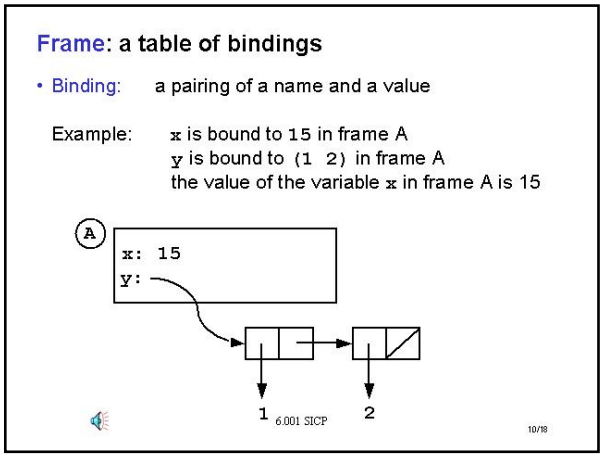
Slide 12.1.8

Now we can start building our **environment model**. First of all, if variables should now be thought of as places, we need a way of organizing them. So the first piece of an environment is something we call a **frame**. This just consists of a **table of bindings**, and a **binding** refers to a pairing of a name and a slot into which a value can be stored, which will be associated with that name.

Slide 12.1.9

In terms of an abstract schematic for keeping track of things, here is a table of bindings. Table A, which we also refer to as a frame, has two bindings in it: it has a binding for the variable `x`, and a binding for the variable `y`.





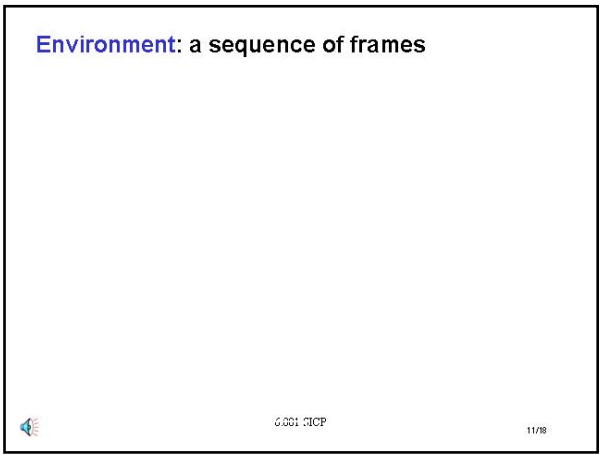
Slide 12.1.10

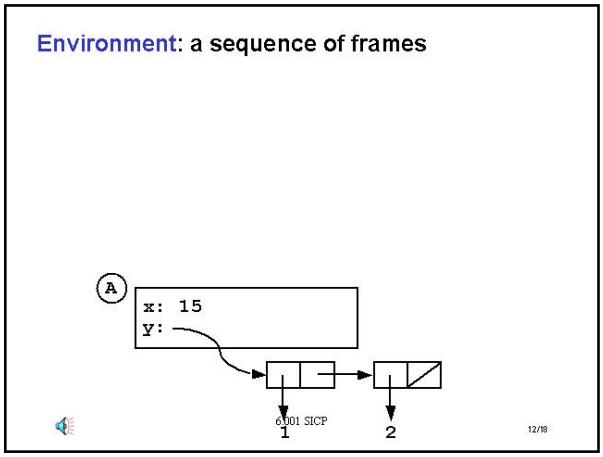
And in particular, we say that `x` is bound to the value `15` within Frame A, and `y` is bound to the value of the list `(1 2)` within Frame A.

Notice that the expression `x` has a value `15` associated with it in this frame, as determine by looking up the binding of `x` within this table. Shortly we will talk about how we actually establish bindings in this frames, but for now we have the first piece of our environment model. A frame is a table of bindings, and a binding is a pairing of a name and a value.

Slide 12.1.11

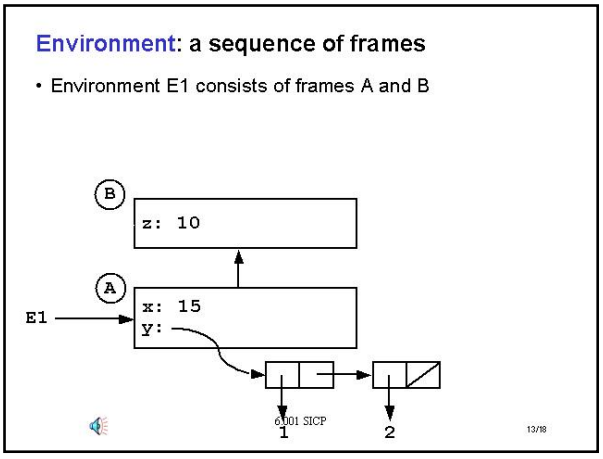
An **environment** consists of a **sequence of frames**, for reasons that we will see shortly.

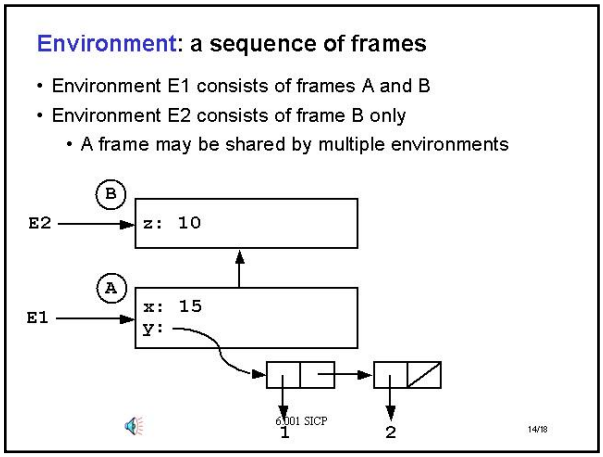




Slide 12.1.12
So here is our frame from before, with the bindings we had...

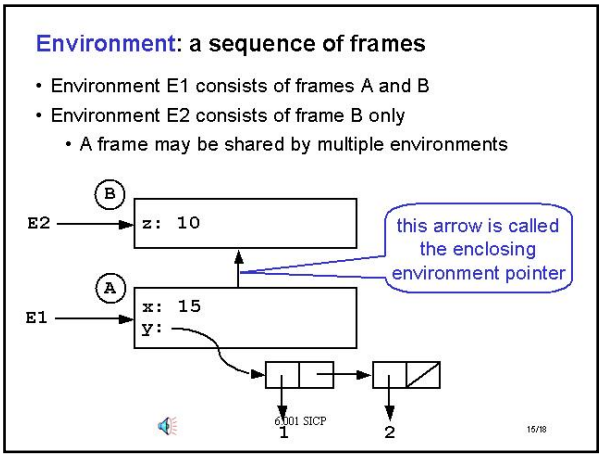
Slide 12.1.13
... and here is a second frame, with its own set of bindings. An environment is a nested **sequence** of frames, so environment E1 consists in this case of Frame A followed by Frame B.

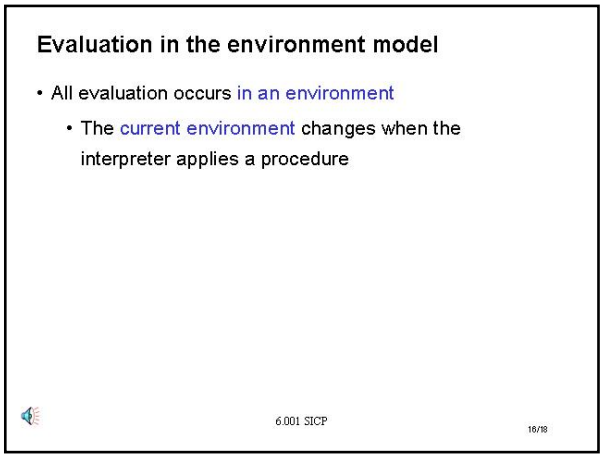




Slide 12.1.14
A second environment, E2, might just consist of Frame B, so note that a frame can be shared by more than one environment, and we will see shortly why that is a powerful tool.

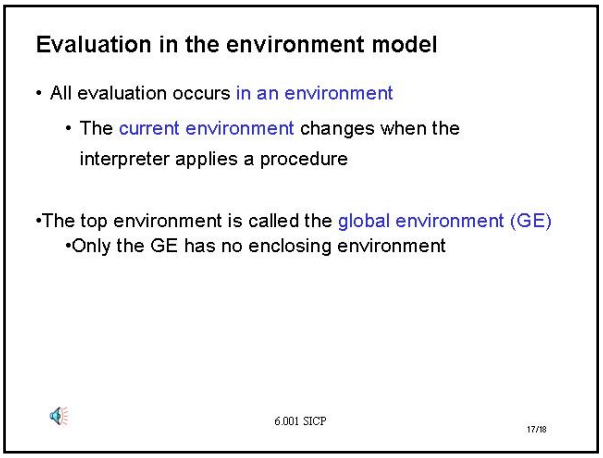
Slide 12.1.15
The connection between frames is important, and is called an **enclosing environment pointer**. So for example, E1 starts with Frame A, and inherits Frame B as its enclosing environment, which in fact is similar to E2. We will see shortly why all these pieces put together help us understanding exactly how evaluation is going to proceed.
So far, we have just been talking about details: we have frames, which are tables of bindings, we have connections between frames, and we have environments as sequences of frames.
With all of these pieces in mind, we can now start relating them to the actual evaluation of expressions.

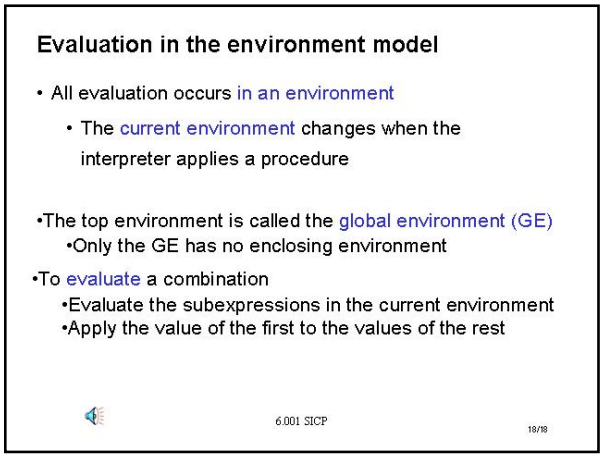




Slide 12.1.16
Now we are ready to see how to connect environments to evaluation. First, as we have already suggested, all evaluation is going to take place with respect to some environment. The environment will provide the context for how to interpret symbols and names. Notice that this won't always be the same environment, and a key issue is that whenever we apply a procedure, we will create a new environment, which captures the context for interpreting the variables of that procedure.

Slide 12.1.17
We have said that all evaluation of expressions is going to take place with respect to an environment. And we have also seen that environments can inherit other environments as part of the chaining process. Nonetheless, this means that eventually we have to terminate that chaining of environments, and to do that we have a special environment, called the **global environment**. It has no enclosing environment and is the only such environment with that property. This is the environment in which we will normally evaluate expressions, as it is our starting point. It is also going to hold the bindings for our basic expressions in our language.





Slide 12.1.18
So we are ready to start putting together rules for evaluating expressions with respect to an environment. Before we do, lets remind you that our central rule for evaluation deal with combinations, and in this new model things are nearly the same as they were in the substitution model.
As before, we will evaluate the subexpressions in any order, now with respect to the current environment. Then we will apply the value of the first subexpression to the values of the others. We will see shortly how to do that in a very mechanistic way. With all of this in place, lets start actually building the environment model.

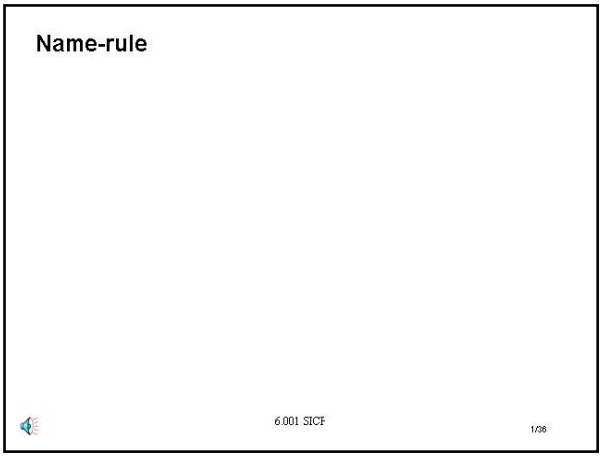
6.001 Notes: Section 12.2

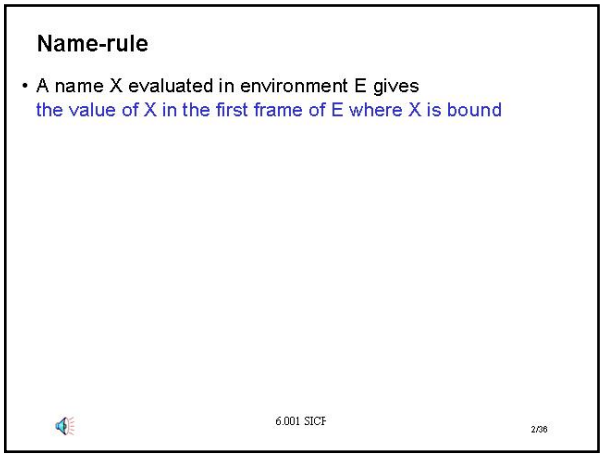
Slide 12.2.1

Okay, we are ready to start looking at our rules for evaluation in our new model, especially to how environments are created and manipulated as part of the evaluation process.

The simplest possible expressions are just numbers. These are self-evaluating which means that their value is just their value, independent of the environment.

The first interesting rule comes with **names**. We have a way of dealing with evaluation of names, our simplest kind of primitive, and that rule should tell us how to get the value associated with a variable.



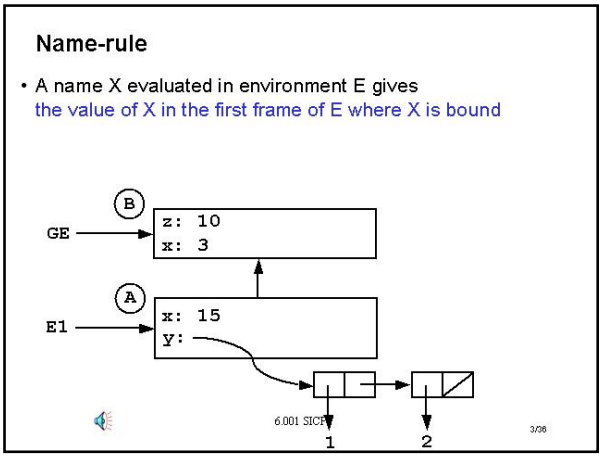


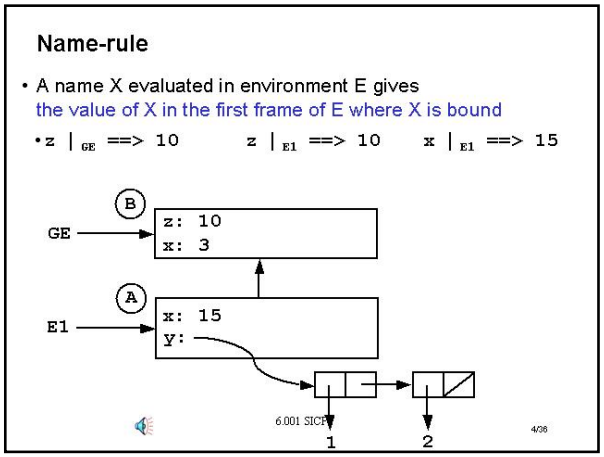
Slide 12.2.2

The rule is straightforward, but quite mechanistic. A name X evaluated in an environment E gives back the value of X in the first frame of E in which there is a binding for X. Remember, an environment is a sequence of frames, so we will start in the first frame of the environment, looking for a binding of that name. We will continue up the chain of frames until we find the first such binding, in which case we return the value associated with that binding.

Slide 12.2.3

So here are our example environments from before. We will call the top level environment, the global environment, or our stopgap environment. Environment E1 has its own frame, and inherits as its enclosing environment, the global environment.



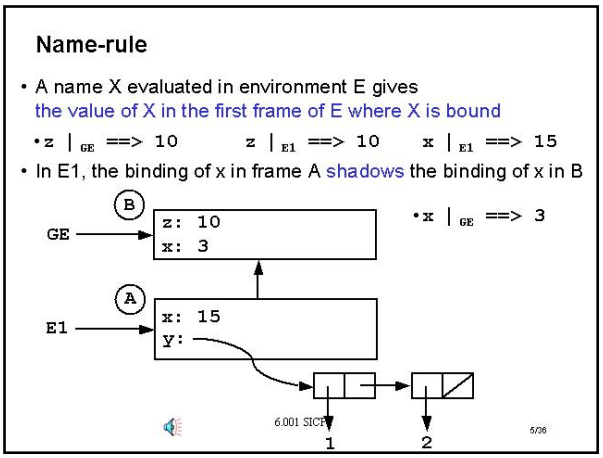


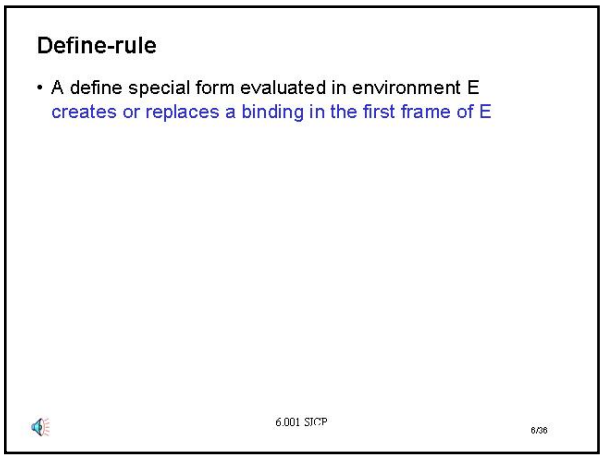
Slide 12.2.4

We can now talk about **the value of a variable with respect to an environment**; in fact we should **always** talk about the value of any expression with respect to some environment. For example, we can ask for the value of the variable `z` with respect to the global environment. Notice the notation we use for this, namely the expression, then a | to indicate the evaluation with respect to some environment, then a subscript to denote the actual environment. Note that normally we will evaluate expressions with respect to the global environment, since that is where we interact with the Scheme system. If I type in an expression to the Scheme interpreter, I am interacting with the

global environment.
Our rule can now be used. The value of variable `z` with respect to the global environment is clearly `10`, as that is the pairing associated with it. If I ask for the value of `z` with respect to the environment E1, I start in frame A. Since there is no binding for `z` in that frame, I move up the enclosing environment pointer to the next frame, in this case, the global environment, and look for the binding of `z` there. Since there is one, I return the value `10`. Finally, if I ask for the value of `x` with respect to E1, it points to the first frame A, and I look there for a binding. Since there is one, I return that value, `15`.

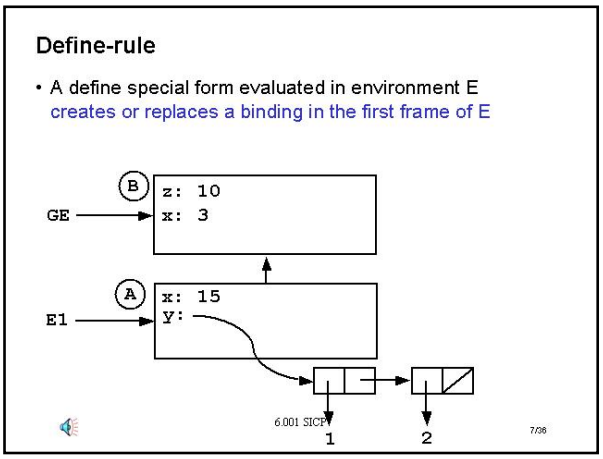
Slide 12.2.5
On the other hand, if I ask for the value of `x` with respect to the global environment, I first look in Frame B for a binding. This returns the value `3`, the value associated with `x` in that frame. In this case, `x` has a different value in different environments. In one, its value is `15`, in another it is `3`. We say that the binding of `x` in Frame A **shadows** the binding of `x` in Frame B, it hides it so that if we start in Frame A, we only see the binding visible there.
Thus, we see how the **name rule** tells us the mechanism by which we look up the value associated with a variable in some environment.

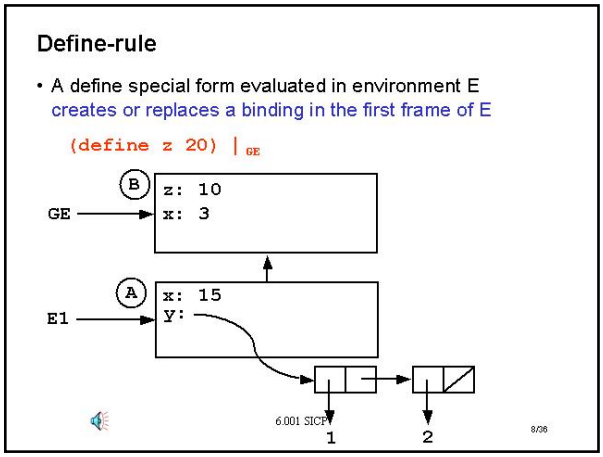




Slide 12.2.6
Given that we have a rule for looking up values associated with variables, what about a rule for **creating** bindings for variables? That, we know, is a `define` expression, and we have a rule for dealing with such expressions. A `define` special form, when evaluated in an environment E, creates or replaces a binding for that variable in the first frame of E, and it always does this in the **first** frame of E.

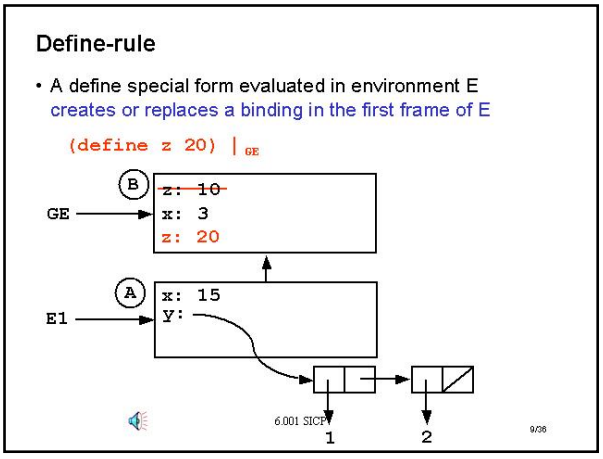
Slide 12.2.7
Here is our environment structure from before. Let's look at what `defines` do when they are evaluated with respect to this structure.

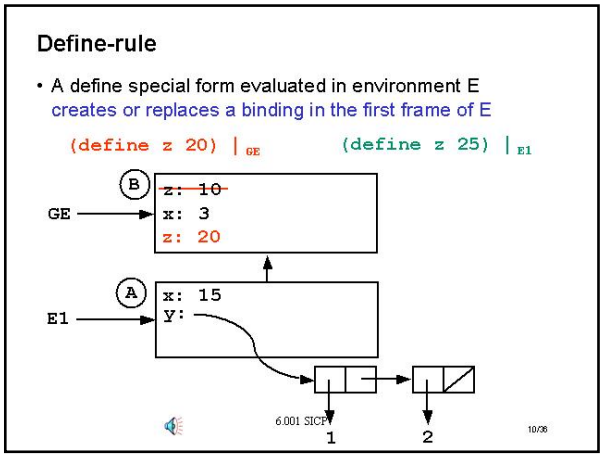




Slide 12.2.8
First, suppose we evaluate `(define z 20)` in the global environment. As I said earlier, this might be an expression that you typed into the computer, and we want to see what happens.

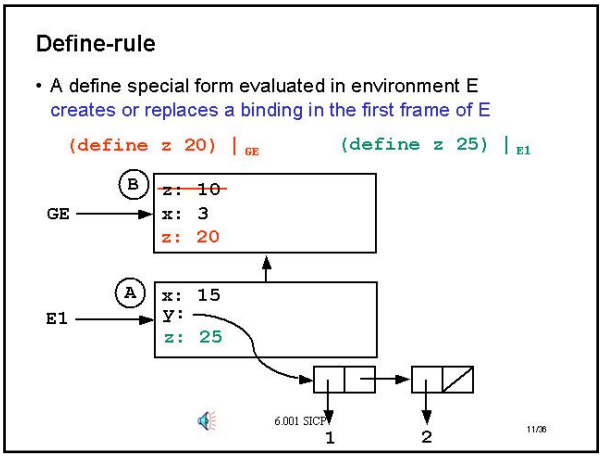
Slide 12.2.9
Note what the rule says: create a binding or replace a binding for the variable, in this case `z`, in the first frame of the environment, in this case, the global environment. So that old binding for `z` as 10 gets replaced by a new binding of `z` to 20.

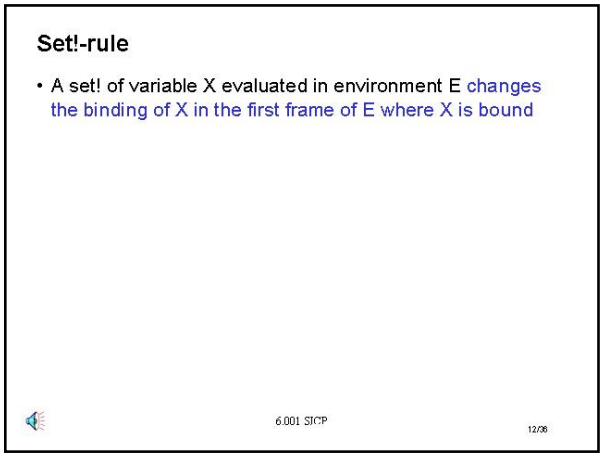




Slide 12.2.10
On the other hand, suppose we evaluate `(define z 25)` with respect to E1. What happens in this case?

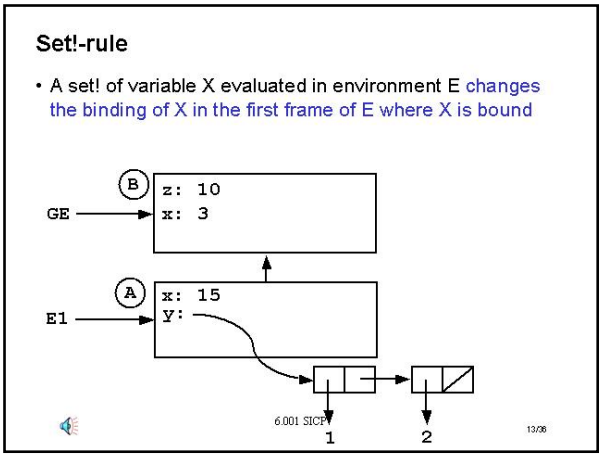
Slide 12.2.11
Remember our rule: create or replace a binding for the variable in the **first** frame of the environment. So in this case, we do this in frame A, the first frame of environment E1. Since there is no current binding for `z` in this frame, we create a new one, and we have now set up a shadowing of `z`. The `z` bound to 25 in Frame A will shadow the binding of `z` to 20 in Frame B, similar to what we saw in the previous slide.

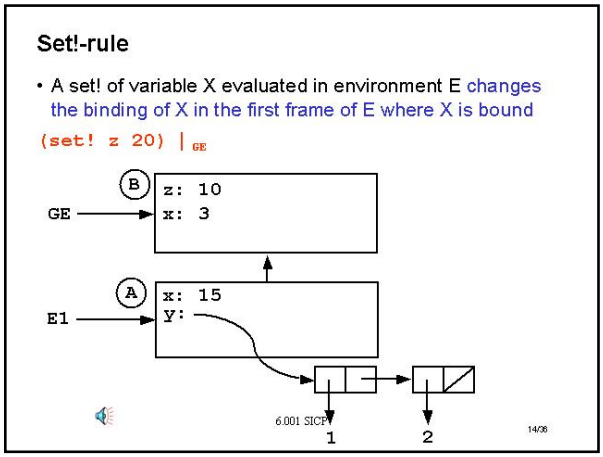




Slide 12.2.12
Thus, we have a rule for looking up the binding of a variable, and a rule for creating a binding of a variable. We should also have a rule for **mutation** of the binding of a variable. This is our **set!** rule. The rule says: mutating a variable `X` with respect to an environment E changes the binding for `X` in the first frame of E for which such a binding exists. Note carefully how this differs from `define`.

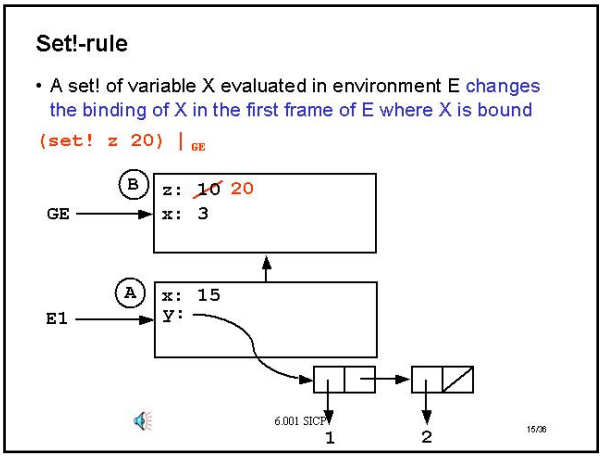
Slide 12.2.13
Here is our environment structure from before. Let's look at what happens we evaluate `set!` expressions with respect to this environment structure.

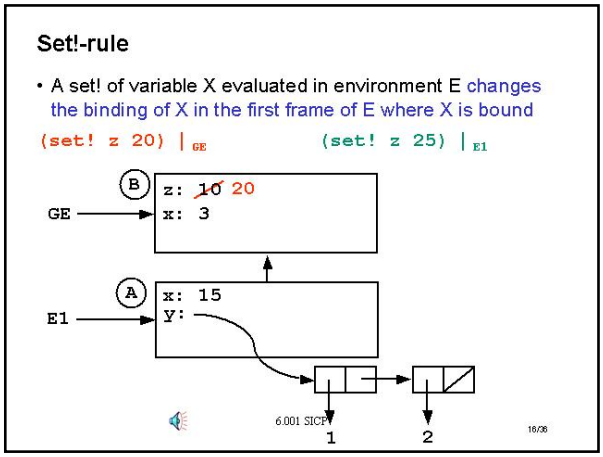




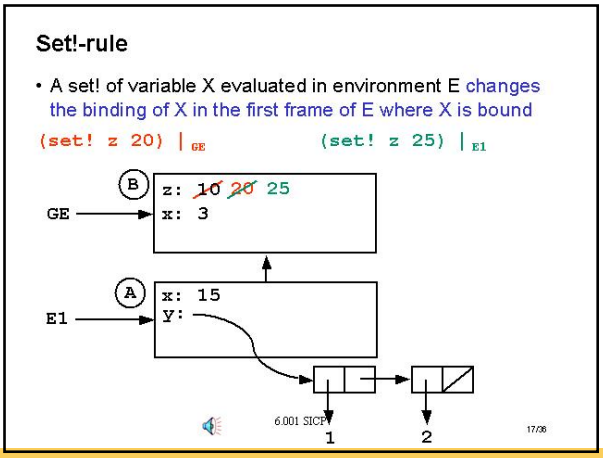
Slide 12.2.14
Let's evaluate `(set! z 20)` with respect to the global environment.

Slide 12.2.15
The rule says: find the first binding for `Z` in that environment. Since we are using the global environment, we look there first, and fortunately there is a binding for `Z` in that environment. So we change it. In place of the binding for `Z` we substitute the value 20, causing a mutation. If we look back at the previous slide, we can see that in this case the effect is exactly the same as a `define`. So why do we have two different forms?

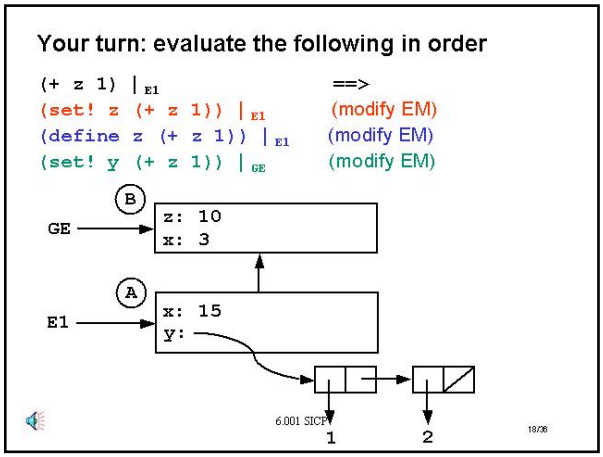




Slide 12.2.16
Well let's evaluate `(set! z 25)` in environment E1, instead.



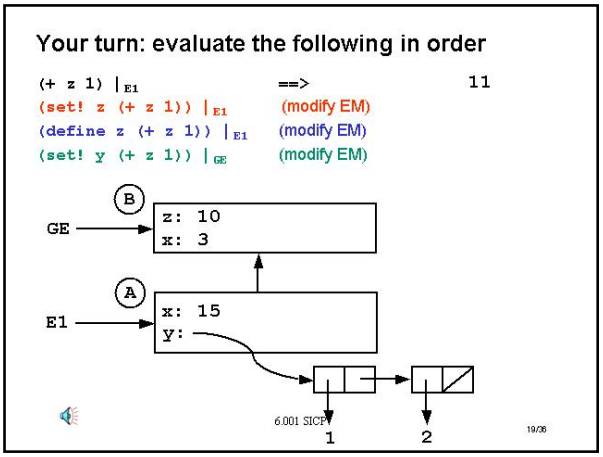
Slide 12.2.17
Remember what the rule says: starting in the first frame of E1, find a binding for `z`. Since there isn't one in the first frame of E1, we chase up the pointer chain to the next frame, B. Fortunately, there is a binding here, so we change **that** binding, that is, we actually change the value in the slot associated with `z`, removing the old value, and inserting the new one. Compare this to what happened when we used `defines` on the previous slide. Notice that now the change has taken place in a different environment from where we started. `Set!` always finds an existing binding for the variable, walking up the chain of environment pointers as needed until it finds a binding and changes it. `Define` always creates a binding in the current frame, even if there was a previous binding there.

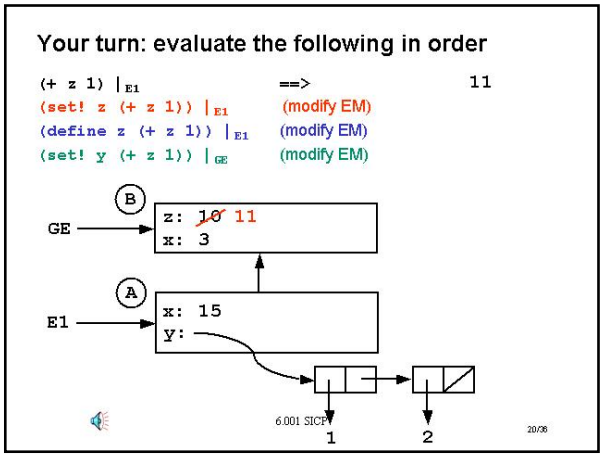


Slide 12.2.18
Okay, let's see if you are getting these ideas. Here is a set of expressions that we would like you to evaluate in order. Take a second to see what each one does, then move to the next slide to see what the effect should be in each case.

Slide 12.2.19

Well the first one is straightforward, but let's work through it carefully anyway. Evaluating this expression with respect to E1 says: get the value of `z`, which says start in E1 and move up the enclosing environment pointers until we find a binding for `z`, which happens to be 10. Get the value for 1, which is just 1. Then, add them together to return the value 11.



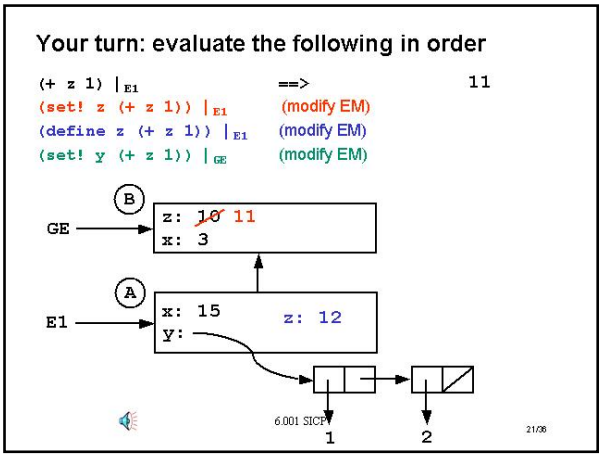


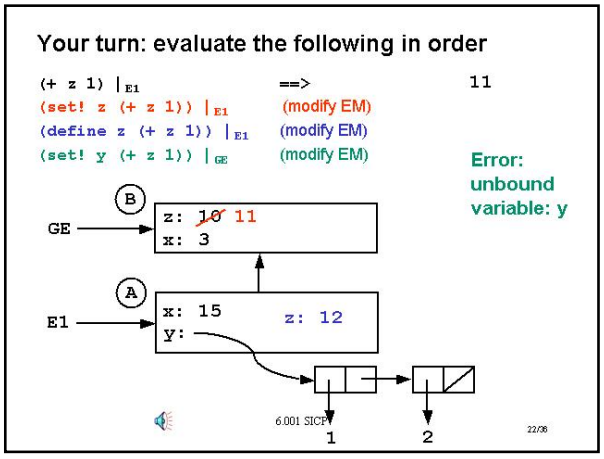
Slide 12.2.20

Doing the mutation of `z` with respect to E1 says: First, get the value of `(+ z 1)` with respect to E1. We just did that, and we know that will be 11. Then the rule says, starting in the first frame of E1, find a binding for `z`. There isn't one in Frame A, so go up the enclosing environment pointer to Frame B. Ah, there is the binding, so change this binding to the value we just computed.

Slide 12.2.21

Now, let's do a `define` with respect to E1. Remember, we get the value of the subexpression `(+ z 1)` with respect to E1, and just as we did before, we start in E1, and since there is no binding for `z` there, we go up the environment pointer to find the binding for `z`, which is now 11. We then add that to the value 1, and then create a binding for `z` in the environment E1. This means we create a pairing of `z` and `12` in Frame A.



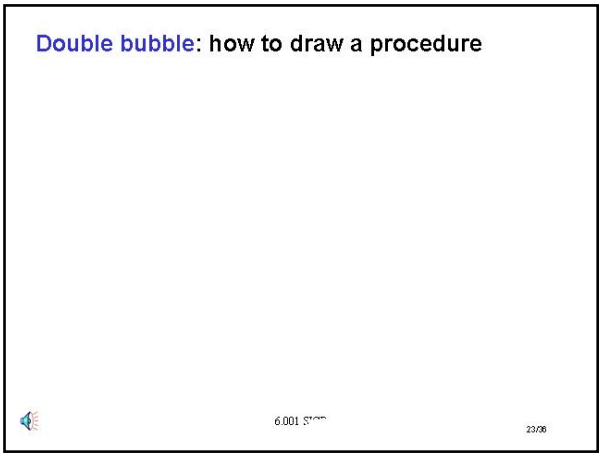


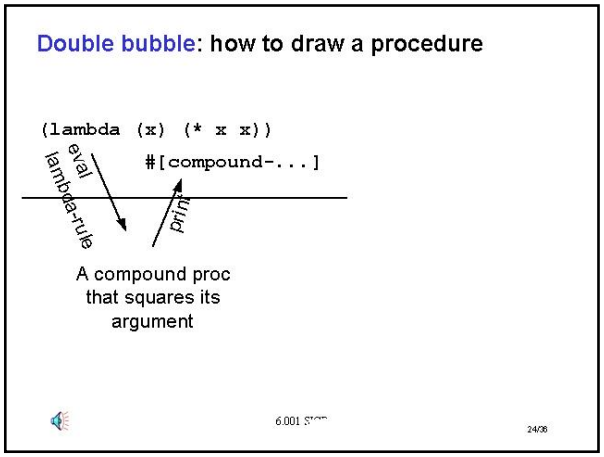
Slide 12.2.22

And the last one we tried to sneak by you! We can still get the value of `(+ z 1)` with respect to the global environment. But trying to do a `set!` of `y` with respect to the global environment will fail. Since we start in the global environment looking for an existing binding for `y`, we never see the binding for `y` in E1, and since we find no binding in GE, we reach an error.

Slide 12.2.23

Next, we need to deal with `lambda` expressions, and here we see a big change in our view of evaluation. First, we are going to introduce some more explicit notation for the procedure created by the evaluation of a `lambda` expression.



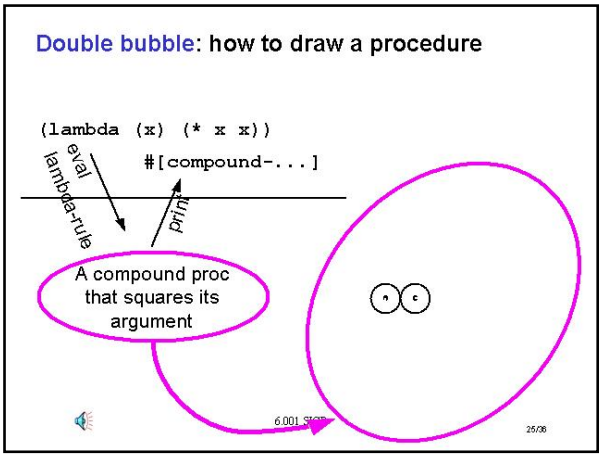


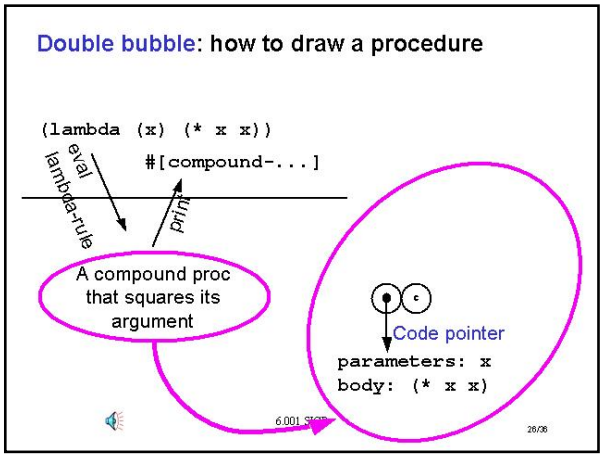
Slide 12.2.24

If we go back to our two-view world, which we used when talking about the substitution model, we said that evaluation of a `lambda` created a compound procedure in the evaluation world, which had a particular printed form in the visible world.

Slide 12.2.25

In the environment model, we are going to be much more explicit and mechanistic about what it means to create a compound procedure. In particular, we are going to create a notation for it, called a **double bubble**.





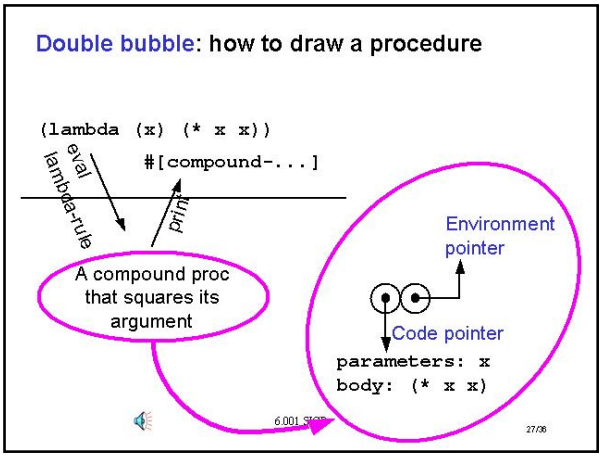
Slide 12.2.26

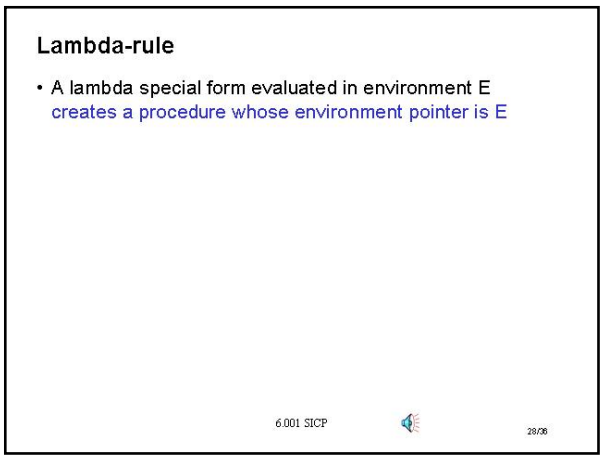
This object contains two parts. The first part, or the first bubble, points to the actual code associated with the procedure. It has two parts, the formal parameters of the procedure and the actual body of the procedure, just as we saw inside a `lambda` expression.

Slide 12.2.27

Note, by the way, that the body of the procedure, `(* x x)` in this case, is just list structure. It has not yet been evaluated, and we have no value associated with it. It is simply copied as the code associated with this part of the double bubble.

The new part is the second bubble. It points to an environment, and in particular to the environment in which the `lambda` expression was evaluated. This **environment pointer** captures a context, the context that was in place when the `lambda` expression was evaluated. This is the context in which the symbols of the code of the `lambda` will be evaluated.





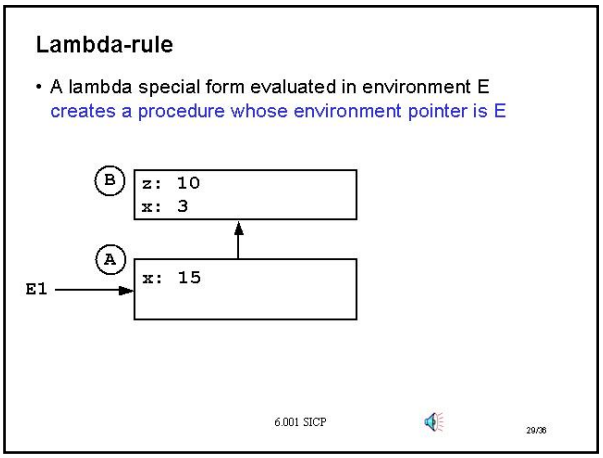
Slide 12.2.28

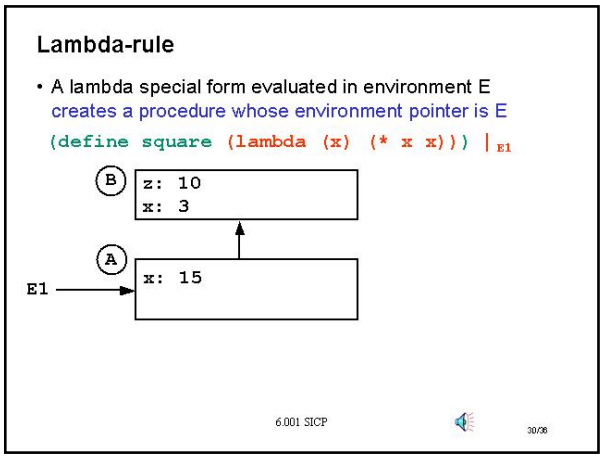
Now we can define the `lambda` rule for evaluation in our environment model. We know that this should create a procedure object; the key is that the environment pointer points to the current environment.

Thus, the `lambda` special form, when evaluated with respect to some environment E, creates a procedure object, whose environment pointer is that environment E, and whose code pointer points to the body of the `lambda` expression.

Slide 12.2.29

Let's look at this in more detail. Here, again, is our environment structure that we have used in previous examples. Let's look at what happens if we evaluate a `lambda` expression with respect to this structure.



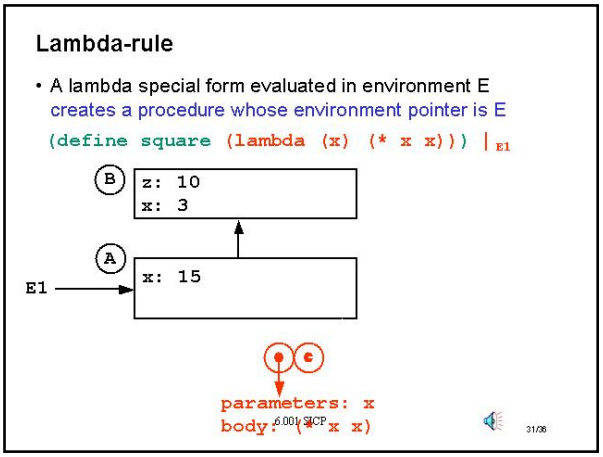


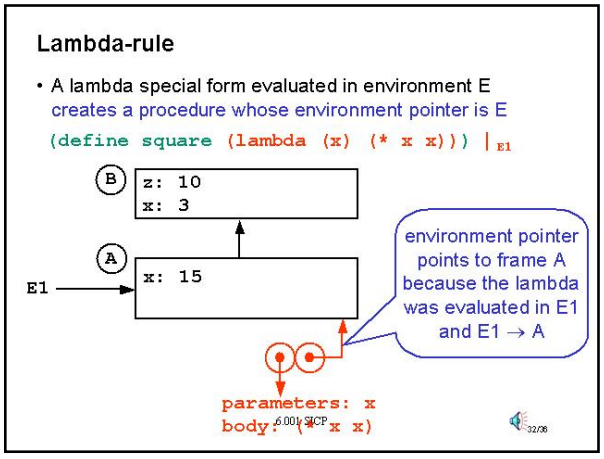
Slide 12.2.30

In particular, let's evaluate the indicated expression with respect to environment E1. Let's start with the `lambda` itself, the inner part.

Slide 12.2.31

The rule says to evaluate this `lambda` with respect to E1. So that will create one of these procedure objects, this little double bubble. The code part of it points to the formal parameter `x` and the body, `(* x x)`. The environment pointer we also know should point to a specific place ...



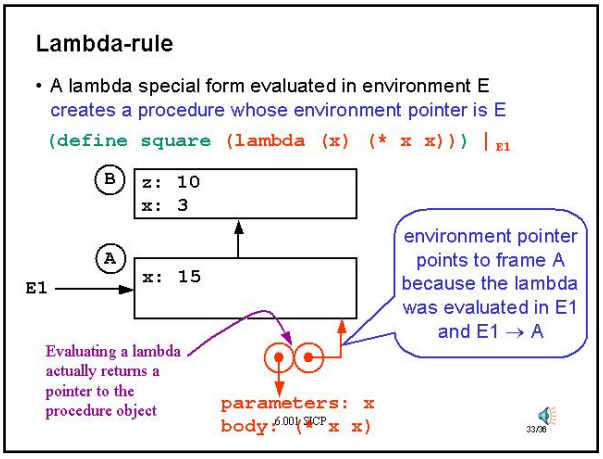


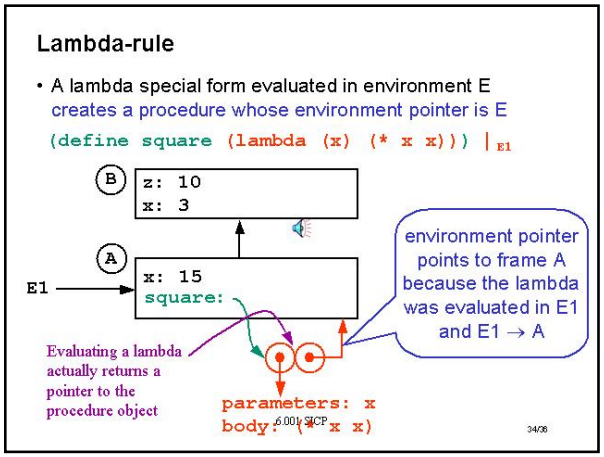
Slide 12.2.32

... specifically, it should point to **Frame A**, because the `lambda` was evaluated with respect to E1, and Frame A is the first frame of the environment. Notice that this rule very mechanistically specifies to where the environment pointer should go. That is the context in which `lambda` was evaluated, that is therefore the context that will define how to interpret other symbols within the body of the procedure.

Slide 12.2.33

The next important point to note is that evaluating a `lambda` actually returns a pointer to the procedure object. That is, evaluating that red `lambda` expression returns as its value a pointer to that red double bubble. Why is this relevant?

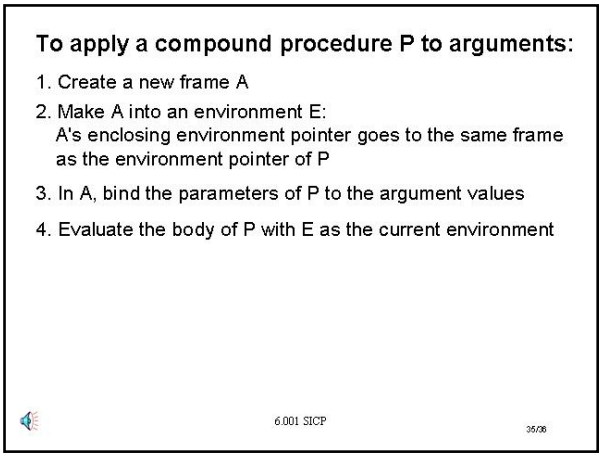




Slide 12.2.34

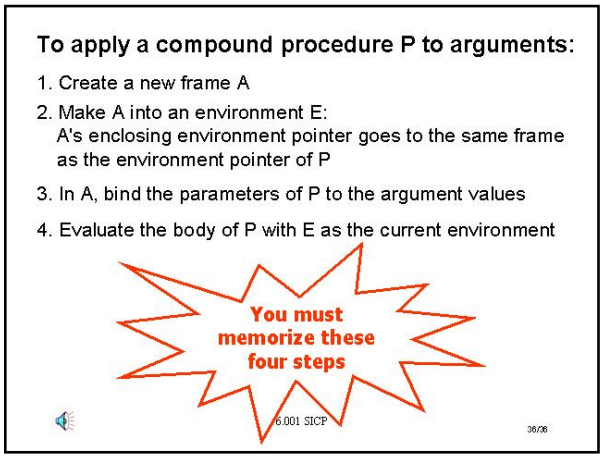
... because we can now complete the evaluation of the `define` expression. Remember what a `define` should do. It should create a binding for the symbol `square` in the first frame of environment E1, with an associated value of the pointer to the double bubble. Thus we will create a pairing of the name `square` and this object in the first frame of E1.

Slide 12.2.35



Now comes the big change. Remember that the central part of our evaluation model is how to apply a compound procedure to a set of arguments. That is, how to apply a procedure that we have created using a `lambda` to a set of arguments, to execute some computation.

We saw that in the substitution model. Now we need to define this for the environment model. This process has four steps. To apply a compound procedure to a set of arguments, do the following. **Step one:** create a new frame, A. **Step two:** convert that frame into an environment, by taking A, and having its enclosing environment pointer go to the same frame as the environment pointer of the procedure object being applied, P. This is very important, as Frame A will inherit the same environment as the procedure inherited. **Step three:** within Frame A, bind the formal parameters of the procedure to the associated arguments passed in. Finally, **Step four:** evaluate the body of the procedure P in this new environment.



Slide 12.2.36

This is the central step of the environment model. These four mechanistic steps are **essential** to your understanding of this model.