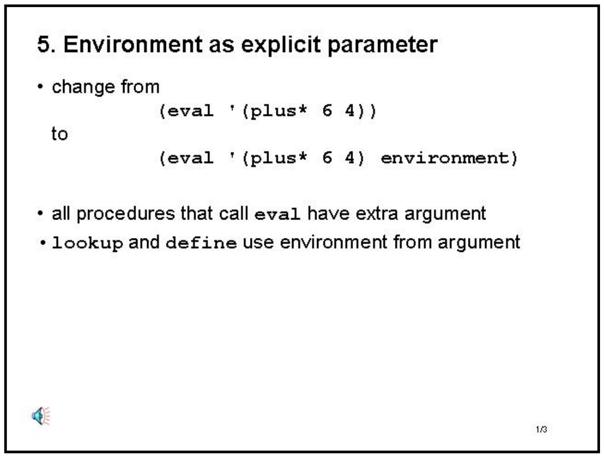
6.001 Notes: Section 15.6

Slide 15.6.1

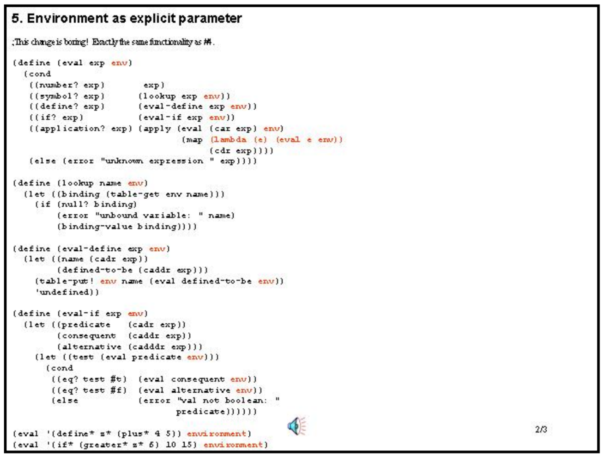


5. Environment as explicit parameter

- change from
  `(eval '(plus* 6 4))`
  to
  `(eval '(plus* 6 4) environment)`
- all procedures that call `eval` have extra argument
- `lookup` and `define` use environment from argument

1/3

The next stage in the evolution of our evaluator is to pull the **environment** out as an explicit parameter. Up until now we could rely on just having a single environment in which to store bindings for variables. It made sense to have a global environment in which to hold names for procedures and global variables. We are about to start moving towards application, in fact we built a simple version of application but we want to extend that to procedures that need local environments. Thus let's pull out the environment explicitly. So we will have to change our evaluator from simply evaluating expressions to evaluating expressions with respect to some environment.
What changes to we need? All procedures must now call `eval` with that extra argument of an environment. `Lookup` and `define*` are going to use that environment argument to specify where to work.



5. Environment as explicit parameter

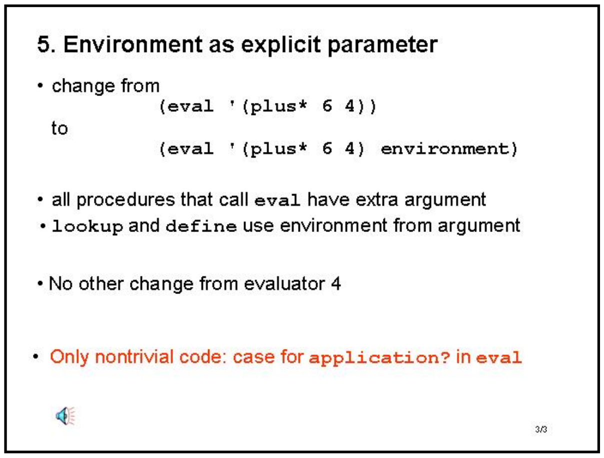
```
;This change is boring! Exactly the same functionality as #4.

(define (eval exp env)
  (cond
   ((number? exp)       exp)
   ((symbol? exp)      (lookup exp env))
   ((define? exp)      (eval-define exp env))
   ((if? exp)          (eval-if exp env))
   ((application? exp) (apply (eval (car exp) env)
                              (map (lambda (e) (eval e env))
                                   (cdr exp))))
   (else (error "unknown expression " exp))))

(define (lookup name env)
  (let ((binding (table-get env name)))
    (if (null? binding)
        (error "unbound variable: " name)
        (binding-value binding))))

(define (eval-define exp env)
  (let ((name (cadr exp))
        (defined-to-be (caddr exp)))
    (table-put! env name (eval defined-to-be env))
    'undefined))

(define (eval-if exp env)
  (let ((predicate   (cadr exp))
        (consequent  (caddr exp))
        (alternative (cadddr exp)))
    (let ((test (eval predicate env)))
      (cond
       ((eq? test #t)  (eval consequent env))
       ((eq? test #f)  (eval alternative env))
       (else           (error "val not boolean: "
                               predicate))))))

(eval '(define* x* (plus* 4 5)) environment)
(eval '(if* (greater* x* 6) 10 15) environment)
```

2/3

Slide 15.6.2

This is actually a **really boring** change! The functionality of this evaluator is **exactly** the same as the previous version. All we do is extend `eval` to take an expression and an environment as arguments, and then make sure that all the dispatch functions also take environments as arguments. Other than make those changes, everything else is exactly as before.
Of course now when we evaluate an expression we have to specify what environment, and we will need an initial or global environment to start with. This will contain bindings for built in procedures.
Otherwise this is just bookkeeping. It will allow us to extend our evaluator in a more general way.

Slide 15.6.3



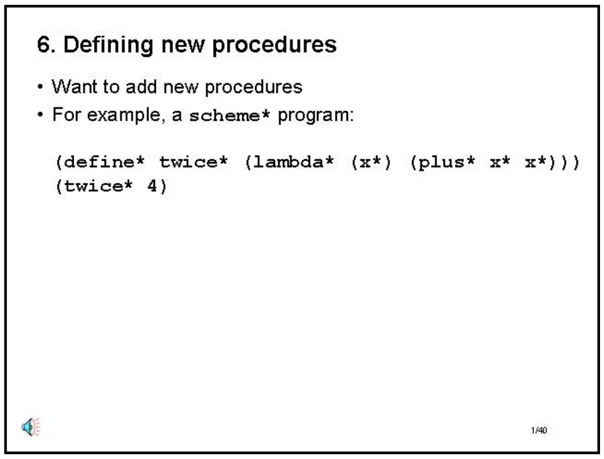
5. Environment as explicit parameter

- change from
  `(eval '(plus* 6 4))`
  to
  `(eval '(plus* 6 4) environment)`
- all procedures that call `eval` have extra argument
- `lookup` and `define` use environment from argument
- No other change from evaluator 4
- Only nontrivial code: case for `application?` in `eval`

3/3

Notice in fact that the only non-trivial case is that dealing with application inside of `eval`. Remember that this is the dispatch that says that we have to apply the object that represents the procedure to its arguments. Notice the change we need. As before, we will evaluate the first subexpression of the tree structure to get the procedure, now including the environment as an argument.
In order to get the values of the other expressions before we could just map `eval` down the list of expressions. Here, however, we have to evaluate each expression with respect to the environment so we need a lambda here to capture that procedure.
We will come back to this shortly, but you can see from a Scheme perspective this makes sense. We are mapping a procedure that evaluates each subexpression with respect to the environment passed in. This is the only non-trivial extension in this case.

6.001 Notes: Section 15.7



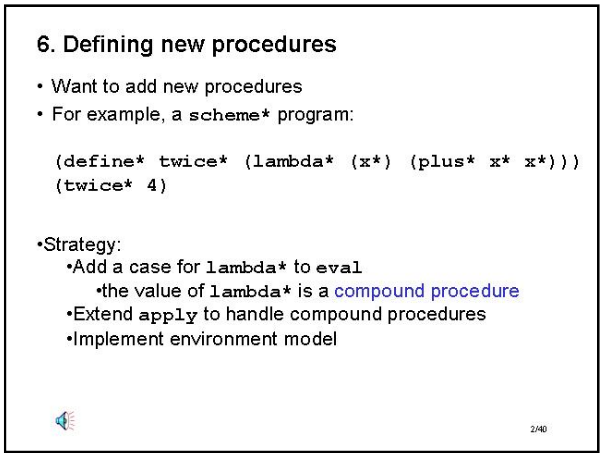
6. Defining new procedures

- Want to add new procedures
- For example, a `scheme*` program:

```
(define* twice* (lambda* (x*) (plus* x* x*)))
(twice* 4)
```

1/40

Slide 15.7.1
Now we are ready to make our last big extension to our interpreter. We would like to be able to add new procedures. If you think about it, once we add this ability we have all the basic components of an interpreter: we can deal with primitive expressions, with special forms, and with applications, and as along as we can create our own procedures, those applications could include arbitrary nesting of expressions.
How do we add new procedures? We would like to create a name for something we make with the equivalent of a `lambda`. We want to capture procedures within `lambda`'s and then use that procedure as an application.



6. Defining new procedures

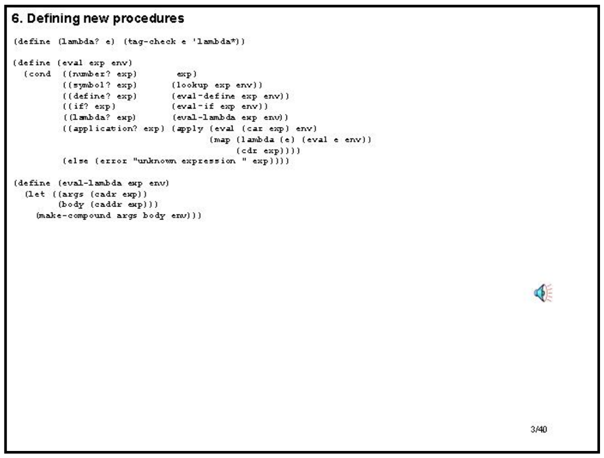
- Want to add new procedures
- For example, a `scheme*` program:

```
(define* twice* (lambda* (x*) (plus* x* x*)))
(twice* 4)
```

•Strategy:
  •Add a case for `lambda*` to `eval`
    •the value of `lambda*` is a compound procedure
  •Extend `apply` to handle compound procedures
  •Implement environment model

2/40

Slide 15.7.2
To do this, we need to add to our evaluator the ability to capture a computation in a procedure. Earlier in the term we saw this with Scheme: that a `lambda` expression could capture a common pattern of computation. Now we are going to see how to add that ability to an evaluator, by creating something that will deal with `lambda`-like expressions.
Here is our strategy for accomplishing this. First, we will need another dispatch in our evaluator, something that deals explicitly with `lambda*` expressions. We would like the value of the `lambda*` expression to return a compound procedure, however we decide to define it. Second, given the ability to create our own procedures, we will need to extend `apply` to handle such procedures. Right now `apply` is just set up to handle built in things we inherit from Scheme, but we need to have it also handle procedures we create. And then we should go back and finish the environment model, since it will become essential in the application of procedures.

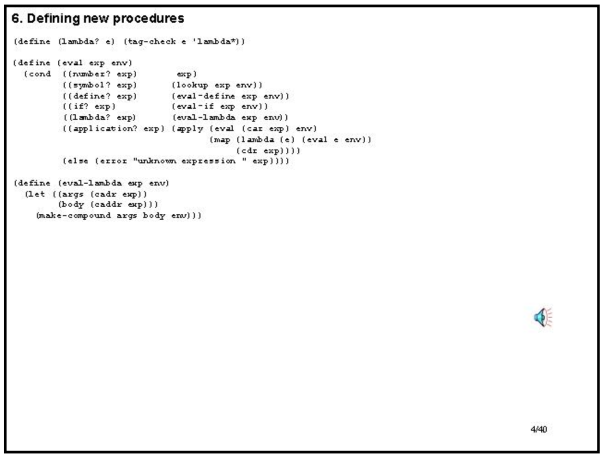


6. Defining new procedures

```
(define (lambda? e) (tag-check e 'lambda*))

(define (eval exp env)
  (cond  ((number? exp)        exp)
         ((symbol? exp)       (lookup exp env))
         ((define? exp)       (eval-define exp env))
         ((if? exp)           (eval-if exp env))
         ((lambda? exp)       (eval-lambda exp env))
         ((application? exp) (apply (eval (car exp) env)
                                    (map (lambda (e) (eval e env))
                                         (cdr exp))))
         (else (error "unknown expression " exp))))

(define (eval-lambda exp env)
  (let ((args (cadr exp))
        (body (caddr exp)))
    (make-compound args body env)))
```

3/40

Slide 15.7.3
Adding the dispatch to `eval` is straightforward. We'll need a new tag checker for `lambda*` expressions, and a new dispatch clause inside `eval` to a procedure to handle the create of `lambda*` objects. Notice where we add this dispatch clause: it **must** come before the application case, since `lambda*` is a special form. It should really also come after the checks for the primitive expressions (numbers or symbols). The ordering of where within the special forms it falls is not crucial.



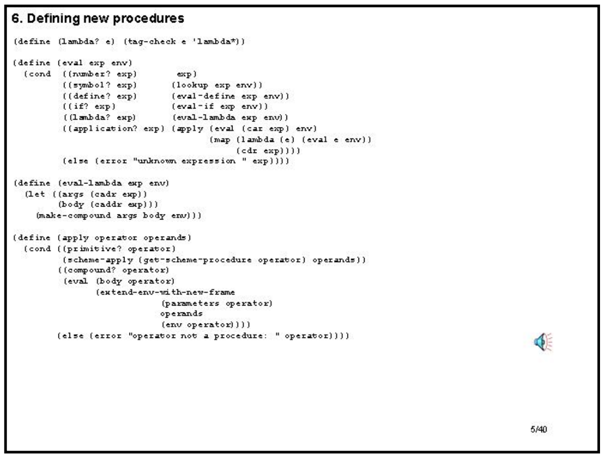
6. Defining new procedures

```
(define (lambda? e) (tag-check e 'lambda*))

(define (eval exp env)
  (cond  ((number? exp)        exp)
         ((symbol? exp)       (lookup exp env))
         ((define? exp)       (eval-define exp env))
         ((if? exp)           (eval-if exp env))
         ((lambda? exp)       (eval-lambda exp env))
         ((application? exp) (apply (eval (car exp) env)
                                   (map (lambda (e) (eval e env))
                                        (cdr exp))))
         (else (error "unknown expression " exp))))

(define (eval-lambda exp env)
  (let ((args (cadr exp))
        (body (caddr exp)))
    (make-compound args body env)))
```

4/40

Slide 15.7.4
So what should `eval-lambda` do? Remember that `eval` takes as input a tree structure representing a `lambda*` expression, which it then passes to this procedure, together with a pointer to the environment. So first, we need to walk down that tree structure, and grab off the arguments of the expression, the formal parameters if you like. That's what the `cadr` does. And we want to walk further along the tree and grab the body of the expression, that is the `caddr`. Having pulled off those two pieces, without evaluation, we then want to glue together some representation of a procedure as a set of arguments, a body, and an environment. This should sound familiar, as it sounds a lot like what we have when we do evaluation with our "double bubble" notation in the environment model. There we have a similar representation for a procedure: formal parameters, body and environment. `Make-compound` is doing a similar thing here. Note that there is **no** call to `eval` inside `eval-lambda`. None of these expressions is being evaluated; we are simply manipulating tree structure.
Also notice that we have made a decision about the body of a procedure. By using the `caddr` of the expression, we are assuming that the body contains exactly one expression. That is a design choice that we could change.

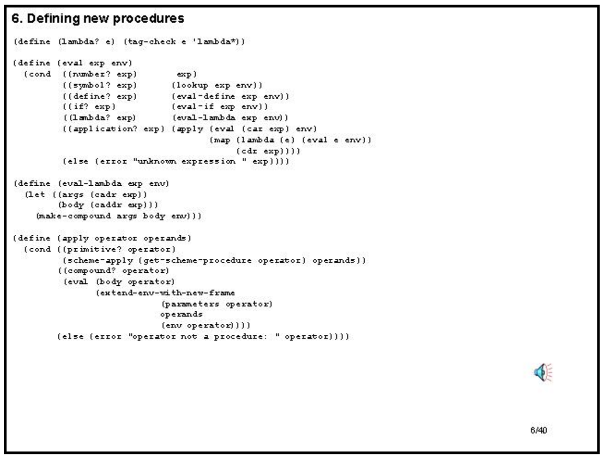
Slide 15.7.5
So adding the new form to create procedure objects is pretty straightforward. We will have to eventually work out what `make-compound` does, but that is basically an abstraction issue.
Now the question is: how do I change `apply`? Remember that `apply` is going to evaluate the first subexpression of a tree structure, get out the appropriate procedure, evaluate all the arguments getting back a list of those values, and then we want to **apply** that procedure to that list.
In the case of primitive things, inherited things from Scheme like `+, *` etc., apply was just a matter of doing the basic operation. Now, we need to extend apply.



6. Defining new procedures

```
(define (lambda? e) (tag-check e 'lambda*))

(define (eval exp env)
  (cond  ((number? exp)        exp)
         ((symbol? exp)       (lookup exp env))
         ((define? exp)       (eval-define exp env))
         ((if? exp)           (eval-if exp env))
         ((lambda? exp)       (eval-lambda exp env))
         ((application? exp) (apply (eval (car exp) env)
                                   (map (lambda (e) (eval e env))
                                        (cdr exp))))
         (else (error "unknown expression " exp))))

(define (eval-lambda exp env)
  (let ((args (cadr exp))
        (body (caddr exp)))
    (make-compound args body env)))

(define (apply operator operands)
  (cond ((primitive? operator)
         (scheme-apply (get-scheme-procedure operator) operands))
        ((compound? operator)
         (eval (body operator)
               (extend-env-with-new-frame
                     (parameters operator)
                     operands
                     (env operator))))
        (else (error "operator not a procedure: " operator))))
```

5/40



6. Defining new procedures

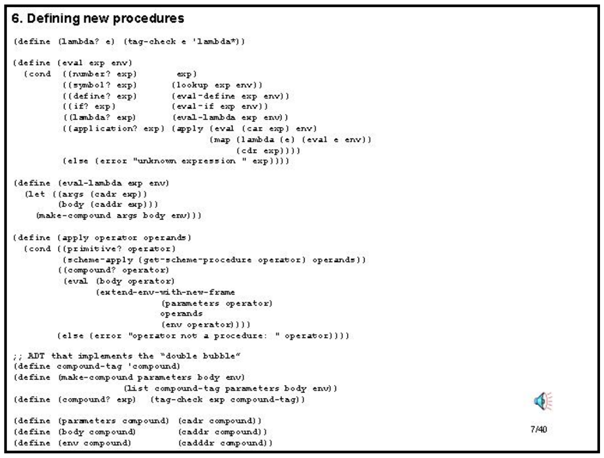
```
(define (lambda? e) (tag-check e 'lambda*))

(define (eval exp env)
  (cond  ((number? exp)        exp)
         ((symbol? exp)       (lookup exp env))
         ((define? exp)       (eval-define exp env))
         ((if? exp)           (eval-if exp env))
         ((lambda? exp)       (eval-lambda exp env))
         ((application? exp) (apply (eval (car exp) env)
                                   (map (lambda (e) (eval e env))
                                        (cdr exp))))
         (else (error "unknown expression " exp))))

(define (eval-lambda exp env)
  (let ((args (cadr exp))
        (body (caddr exp)))
    (make-compound args body env)))

(define (apply operator operands)
  (cond ((primitive? operator)
         (scheme-apply (get-scheme-procedure operator) operands))
        ((compound? operator)
         (eval (body operator)
               (extend-env-with-new-frame
                     (parameters operator)
                     operands
                     (env operator))))
        (else (error "operator not a procedure: " operator))))
```

6/40

Slide 15.7.6
In this case, when we go to apply an **operator** to a list of **operands**, we will need to check for type. Is it a primitive operation? If yes, we just apply the underlying Scheme procedure as before. Is it a compound operator, that is, are we evaluating an expression in which we are using a procedure that we created in our evaluator using `lambda*`? In this case we need to implement the idea of the environment model.
Thus, we get the **body** of the operator, that is, we pull out the piece that we glued together when we made the procedure object. Now we want to evaluate that body. But we are going to evaluate that with respect to an environment, created by taking

the **operands** passed in, taking the formal parameters of the procedure object, and binding them together. This new environment should extend the previous one, in case we need things from there.
Thus when we have a compound application, we will grab the body of the procedure and evaluate it in a new environment, which has access to everything in the previous environment, plus bindings for the formal parameters of the procedure to the values passed in.
Note what this does. Given this version of `apply` we have a wonderfully cyclic structure. `Eval` of an expression with respect to an environment in the general case will reduce to `apply`ing a procedure to a set of arguments, and that will reduce in the general case to `eval`uating a new expression (the body of the operator) with respect to a new environment. In this way, `eval` and `apply` will work hand-in-hand in terms of unwrapping the abstractions of the procedures down to primitive operations on primitive objects.

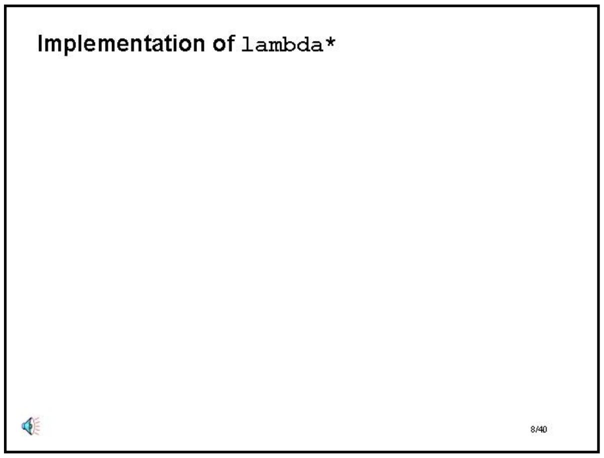
Slide 15.7.7
All that is left to do is to implement the "double bubble" idea as an abstract data type. We can do that by creating a tag for a compound procedure and having the constructor (`make-compound`) simply glue things together in a list, with the appropriate predicate and selectors associated with this data abstraction. This is not the only way to do this; all we need is some method for gluing things together and getting them back apart.



6. Defining new procedures

```
(define (lambda? e) (tag-check e 'lambda*))

(define (eval exp env)
  (cond ((number? exp)       exp)
        ((symbol? exp)       (lookup exp env))
        ((define? exp)       (eval-define exp env))
        ((if? exp)           (eval-if exp env))
        ((lambda? exp)       (eval-lambda exp env))
        ((application? exp) (apply (eval (car exp) env)
                                   (map (lambda (e) (eval e env))
                                        (cdr exp))))
        (else (error "unknown expression " exp))))

(define (eval-lambda exp env)
  (let ((args (cadr exp))
        (body (caddr exp)))
    (make-compound args body env)))

(define (apply operator operands)
  (cond ((primitive? operator)
         (scheme-apply (get-scheme-procedure operator) operands))
        ((compound? operator)
         (eval (body operator)
               (extend-env-with-new-frame
                          (parameters operator)
                          operands
                          (env operator))))
        (else (error "operator not a procedure: " operator))))

;; ADT that implements the "double bubble"
(define compound-tag 'compound)
(define (make-compound parameters body env)
                  (list compound-tag parameters body env))
(define (compound? exp)  (tag-check exp compound-tag))

(define (parameters compound)  (cadr compound))
(define (body compound)        (caddr compound))
(define (env compound)         (cadddr compound))
```

7/40

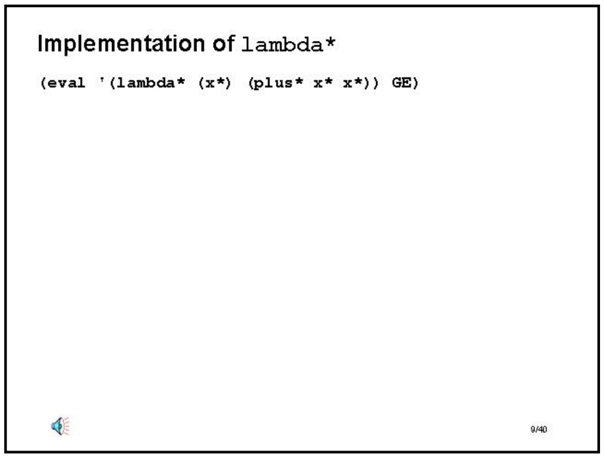


Implementation of `lambda*`

8/40

Slide 15.7.8
Looking at the code helps us understand our changes, but it is probably easier to see the changes by watching the evolution of the evaluation of an expression. Let's trace through the evaluation of a `lambda*` expression. That is, we are evaluating a tree structure whose first element is the symbol `lambda*`, whose second element is a list of parameters, and whose third element is an expression constituting the body of the procedure. We are evaluating this with respect to some environment, say the global environment.

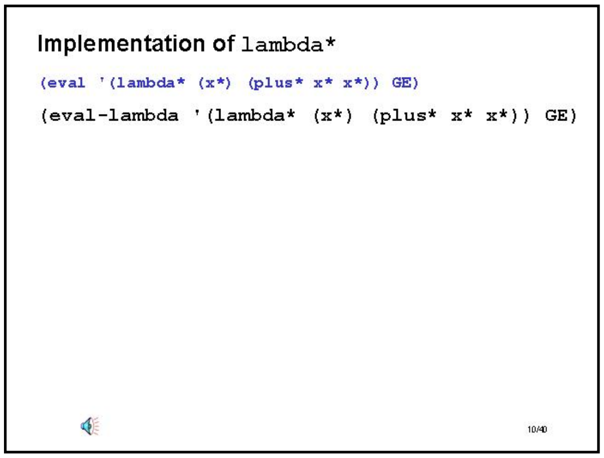
Slide 15.7.9
As before, please follow along in the code (for part 6).
So `eval` takes in that tree structure representing this expression and checks it for different types of expressions (number, symbol, etc.). Eventually `eval` reaches the case for handling `lambda*` expressions, Thus, we will dispatch on type to `eval-lambda`, the specific procedure designed to evaluate `lambda*`'s.



Implementation of `lambda*`

```
(eval '(lambda* (x*) (plus* x* x*)) GE)
```

9/40



Implementation of `lambda*`

```
(eval '(lambda* (x*) (plus* x* x*)) GE)
(eval-lambda '(lambda* (x*) (plus* x* x*)) GE)
```

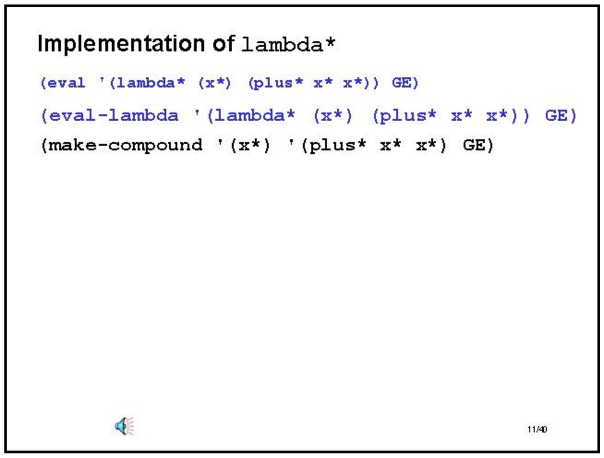
10/40

Slide 15.7.10

`Eval-lambda` starts with the same tree structure as an argument, plus some pointer to the global environment (GE). What does it do?
It walks down the tree structure of the first argument, and grab the arguments (that's the `cadr`). Similarly, walk down the tree structure and grab the body (that's the `caddr`) (note that we are assuming there is only one expression in the body). In this case, the `cadr` will return the list `(x*)` and the `caddr` will return the list `(plus* x* x*)`. Let me stress again that we are simply getting these as tree structure; there is no evaluation yet. Having acquired those two structures, we then glue them together using `make-compound`.

Slide 15.7.11

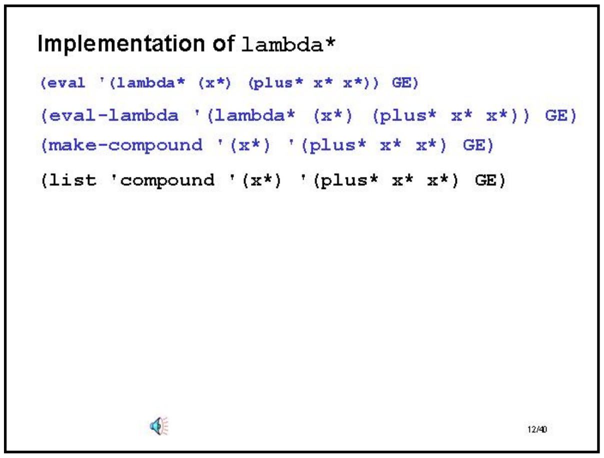
So the evaluation reduces to this expression. This is simply using our constructor to glue these pieces together, and in this case that converts to this form....



Implementation of `lambda*`

```
(eval '(lambda* (x*) (plus* x* x*)) GE)
(eval-lambda '(lambda* (x*) (plus* x* x*)) GE)
(make-compound '(x*) '(plus* x* x*) GE)
```

11/40



Implementation of `lambda*`

```
(eval '(lambda* (x*) (plus* x* x*)) GE)
(eval-lambda '(lambda* (x*) (plus* x* x*)) GE)
(make-compound '(x*) '(plus* x* x*) GE)
(list 'compound '(x*) '(plus* x* x*) GE)
```

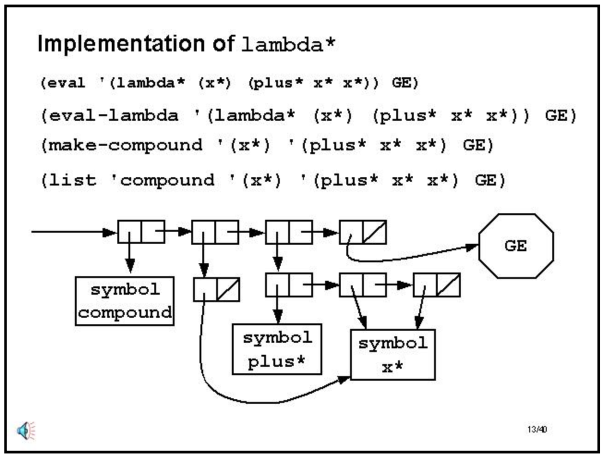
12/40

Slide 15.7.12

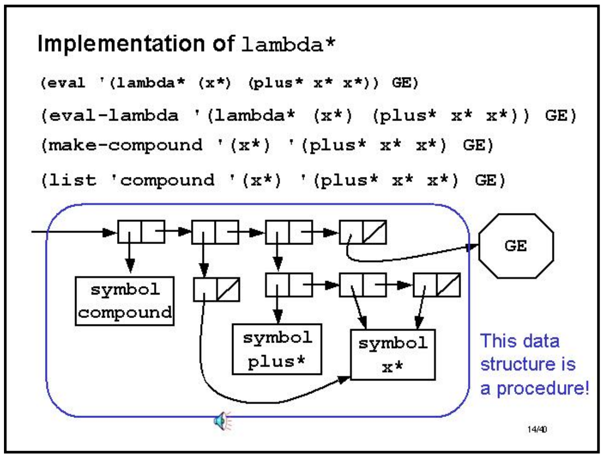
... and thus we are down to a basic primitive. We are going to construct a new list structure, with a type tag, a list of parameters, a body and the environment in which this `lambda*` expression was created.

Slide 15.7.13

And that says that the evaluation of this expression returns **this** list structure. That is actually quite nice! We have created a new object in our language, we have evaluated a `lambda*` and created a procedure object. When we go to do an application (as we will see shortly) we need an expression represented as list structure so that we can pull out the pieces, and here they are. All of the parts we will need for an application are represented as list structure within this list, so that we easily use those within the evaluator.


Implementation of `lambda*`

```
(eval '(lambda* (x*) (plus* x* x*)) GE)
(eval-lambda '(lambda* (x*) (plus* x* x*)) GE)
(make-compound '(x*) '(plus* x* x*) GE)
(list 'compound '(x*) '(plus* x* x*) GE)
```


Implementation of `lambda*`

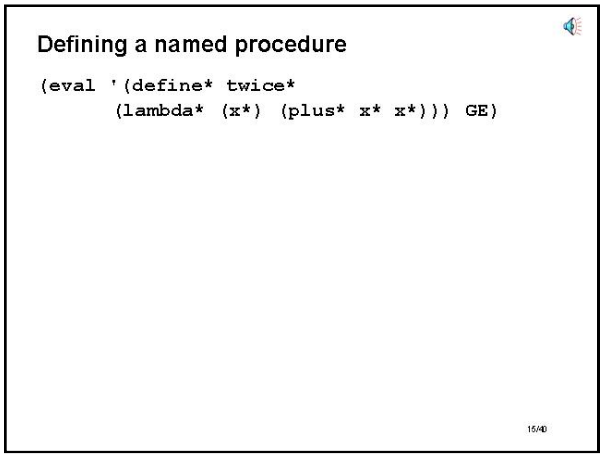
```
(eval '(lambda* (x*) (plus* x* x*)) GE)
(eval-lambda '(lambda* (x*) (plus* x* x*)) GE)
(make-compound '(x*) '(plus* x* x*) GE)
(list 'compound '(x*) '(plus* x* x*) GE)
```

Slide 15.7.14

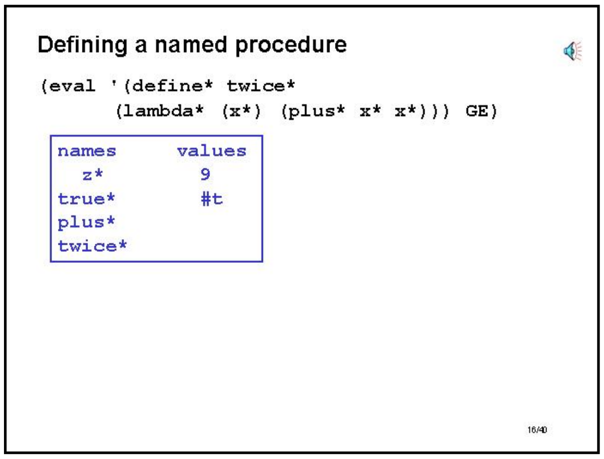
Thus, this piece of list structure represents a procedure. It is our particular way of creating a data structure for a procedure: it has a tag identifying it as a compound procedure, a set of parameters, and a body, all represented in the format we expect as a parsed tree.

Slide 15.7.15

Now that we can create procedures in our interpreter, we naturally want to give them names, and that just uses our `define*` expressions. How does this work? First of all, we will have some initial bindings in our environment ...


Defining a named procedure

```
(eval '(define* twice*
      (lambda* (x*) (plus* x* x*))) GE)
```


Defining a named procedure

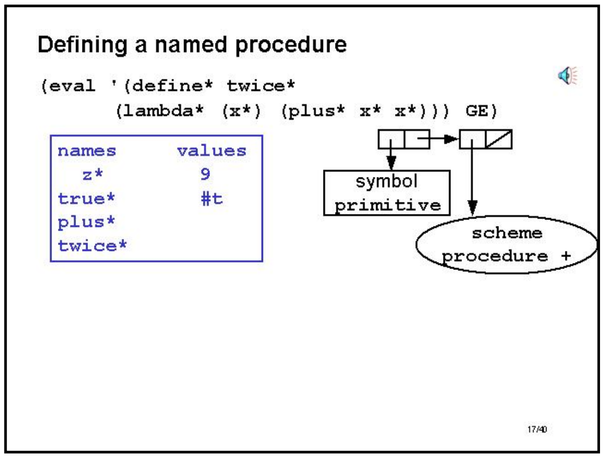
```
(eval '(define* twice*
      (lambda* (x*) (plus* x* x*))) GE)
```

names	values
z*	9
true*	#t
plus*	
twice*	

Slide 15.7.16

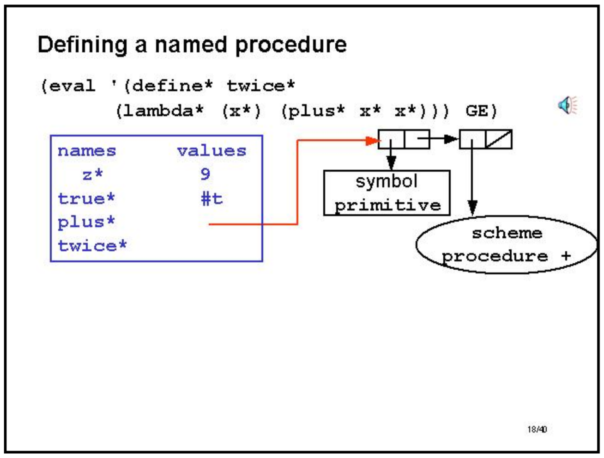
... as shown here in our abstract table representation for an environment. This environment includes prior bindings for names to numbers and for symbols to Boolean values. We might also have some other bindings based on built in procedures, for example, we may have installed a primitive procedure for `plus*` ...

Slide 15.7.17
... and as we saw from our earlier work, `plus*` would be represented by a primitive procedure, in this case as list structure with a tag identifying it as a primitive and a pointer to the actual primitive code. Remember that this was installed in the global environment when we loaded up our evaluator.



Defining a named procedure

```
(eval '(define* twice*
      (lambda* (x*) (plus* x* x*))) GE)
```

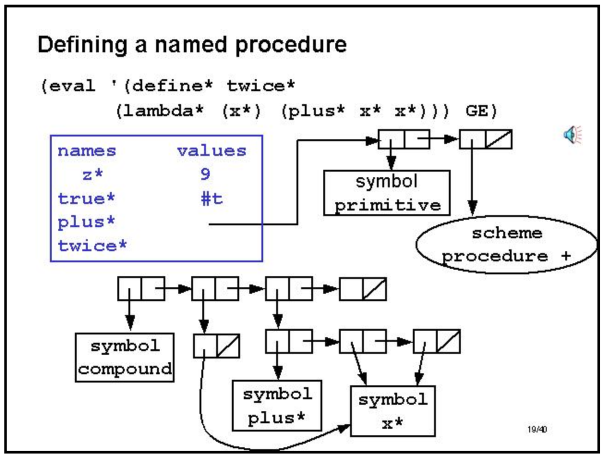


Defining a named procedure

```
(eval '(define* twice*
      (lambda* (x*) (plus* x* x*))) GE)
```

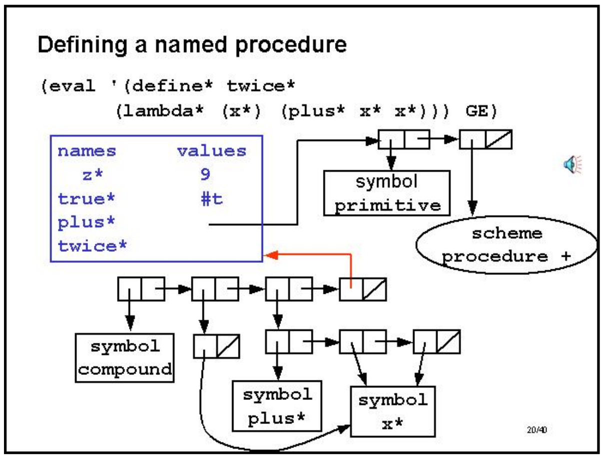
Slide 15.7.18
That is, there is a pointer from `plus*` in the global environment (that table) to the actual representation of the object.

Slide 15.7.19
So what happens when we evaluate the expression shown at the top? `Eval` will dispatch this to `eval-define` which will create a binding for the symbol `twice*` in the environment, to the value returned by evaluating `(lambda* (x*) (plus* x* x*))`. We know what that latter evaluation will return, that is the thing we just saw and it gives us back this structure.



Defining a named procedure

```
(eval '(define* twice*
      (lambda* (x*) (plus* x* x*))) GE)
```

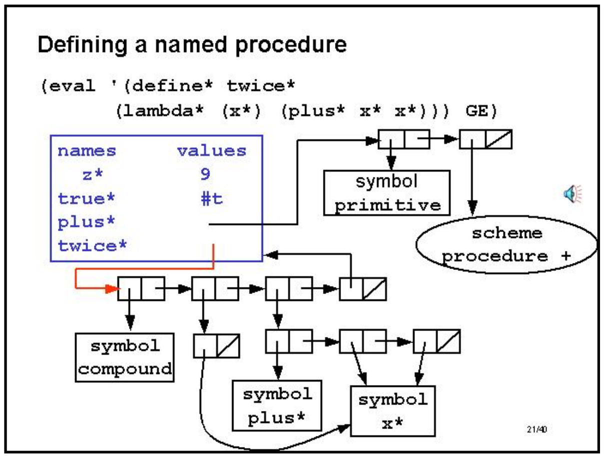


Defining a named procedure

```
(eval '(define* twice*
      (lambda* (x*) (plus* x* x*))) GE)
```

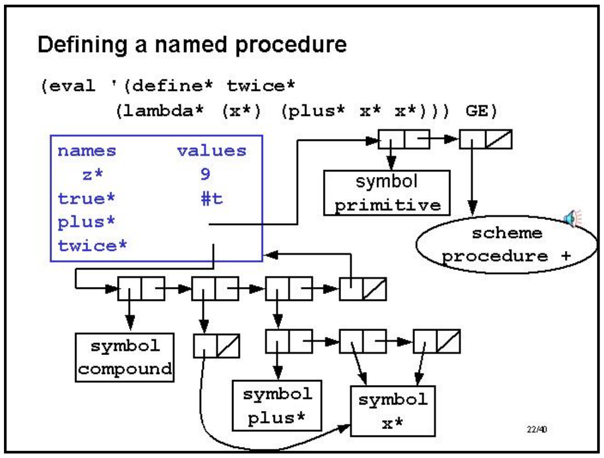
Slide 15.7.20
We do need to worry about where the environment part of the structure points to, and we know it should point to the environment in which the evaluation was done, hence to this frame...

Slide 15.7.21
... and the binding of the name `twice*` to this value will take place in this environment. Thus we add a pairing to the table of the symbol `twice*` and a pointer to this structure.



Defining a named procedure

```
(eval '(define* twice*
      (lambda* (x*) (plus* x* x*))) GE)
```

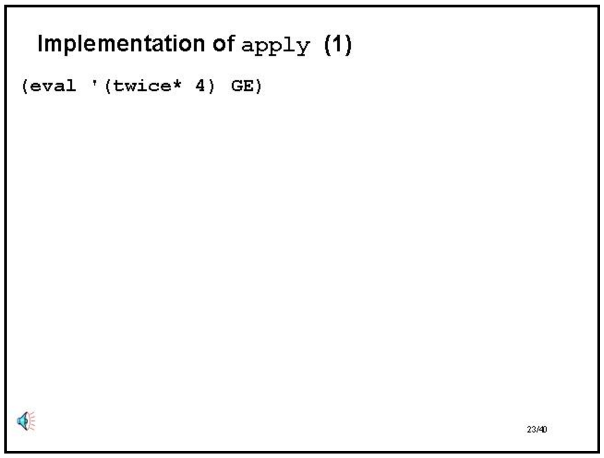


Defining a named procedure

```
(eval '(define* twice*
      (lambda* (x*) (plus* x* x*))) GE)
```

Slide 15.7.22
So notice what has happened. We have created the ability to create names for procedures in our environment, that point to one of these procedure objects. Thus we can now refer to that procedure object by name, thus building in a level of abstraction.

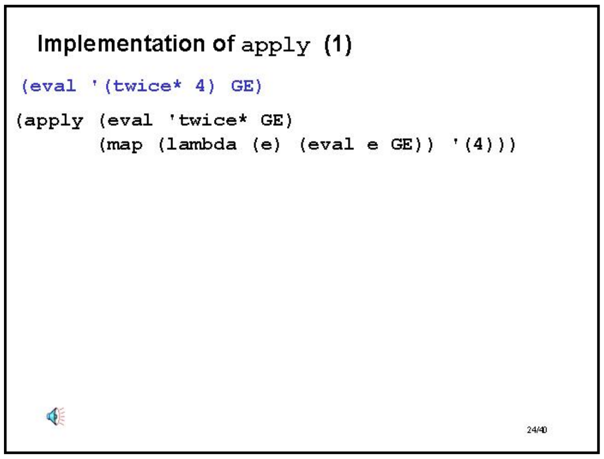
Slide 15.7.23
So let's look at the use of this abstraction, by looking at the evaluation of an expression that involves a name for one of these procedures. We want to see how `eval` and `apply` work together to unwrap the abstraction to more primitive expressions.
First, let's evaluate the expression shown with respect to the global environment.



Implementation of apply (1)

```
(eval '(twice* 4) GE)
```

23/40



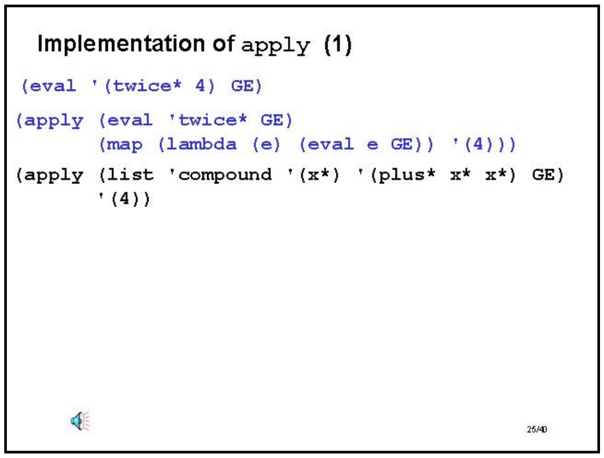
Implementation of apply (1)

```
(eval '(twice* 4) GE)

(apply (eval 'twice* GE)
       (map (lambda (e) (eval e GE)) '(4)))
```

24/40

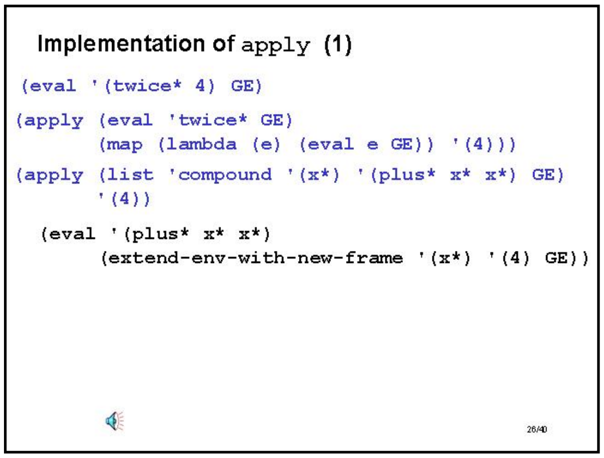
Slide 15.7.24
`Eval` will deduce that this is an application, thus reducing to this `apply` expression. We must evaluate the first subexpression (which we simply grab from the tree structure) and apply the result to the list we get by mapping `eval` down the list of arguments in the initial expression. Note that all of these evaluations take place with respect to the same environment.

Slide 15.7.25
And what does the evaluation of the pieces do? `Twice*` is just a symbol, so `eval` dispatches to a lookup, which returns this **list** structure that captures the pieces of the procedure (label, argument list, body and inherited environment). Evaluating the arguments simply returns the list `(4)` since numbers are self-evaluating. Thus we are ready to apply this representation of a compound procedure to a list of argument values.


Implementation of apply (1)

```
(eval '(twice* 4) GE)
(apply (eval 'twice* GE)
       (map (lambda (e) (eval e GE)) '(4)))
(apply (list 'compound '(x*) '(plus* x* x*) GE)
       '(4))
```

25/40

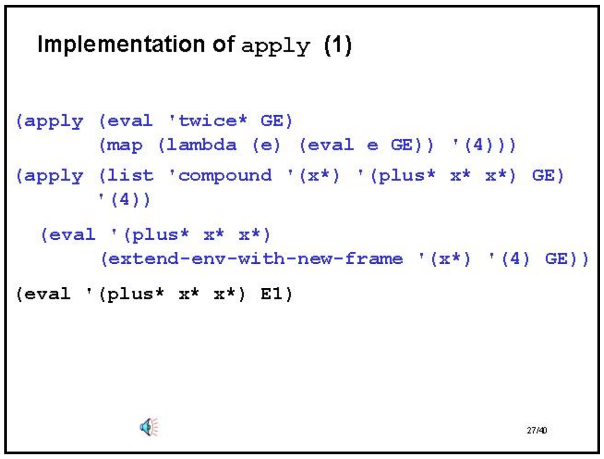

Implementation of apply (1)

```
(eval '(twice* 4) GE)
(apply (eval 'twice* GE)
       (map (lambda (e) (eval e GE)) '(4)))
(apply (list 'compound '(x*) '(plus* x* x*) GE)
       '(4))
  (eval '(plus* x* x*)
       (extend-env-with-new-frame '(x*) '(4) GE))
```

26/40

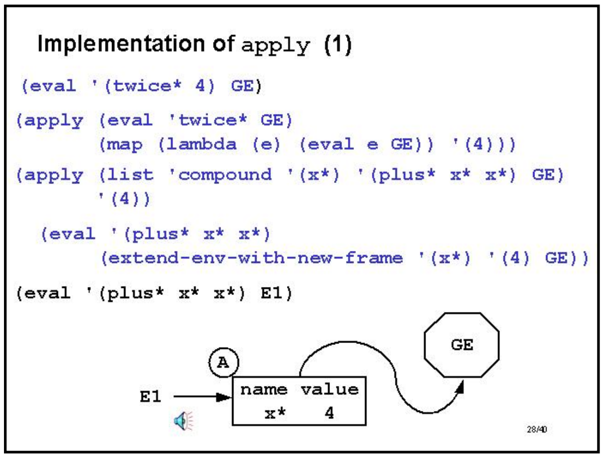
Slide 15.7.26
And the code for `apply` indicates that we should grab (using list manipulation procedures) the formal parameters of the procedure, take the list of argument values supplied, and glue them together with respect to the original environment in which the procedure was created, to create a new frame. That is just some list manipulation that we haven't detailed yet. Conceptually, we know what this should do: create a new table (or environment) with the formal parameters bound to the argument list, and which is scoped by the environment in which the procedure was created.
And then, we walk down the tree representation for the procedure object (this version of a "double bubble"), grab off the body, and we will evaluate that new expression in this new environment. Note that this is all just list manipulation, we have **not** evaluated any of the pieces, we are simply setting up to do that.

Slide 15.7.27
So extending the environment will be some kind of abstract data type manipulation that creates a new environment, call it E1, in which those variables have been bound. We are now going to evaluate this new expression with respect to that environment.


Implementation of apply (1)

```
(apply (eval 'twice* GE)
       (map (lambda (e) (eval e GE)) '(4)))
(apply (list 'compound '(x*) '(plus* x* x*) GE)
       '(4))
  (eval '(plus* x* x*)
       (extend-env-with-new-frame '(x*) '(4) GE))
(eval '(plus* x* x*) E1)
```

27/40



Implementation of apply (1)

```
(eval '(twice* 4) GE)
(apply (eval 'twice* GE)
       (map (lambda (e) (eval e GE)) '(4)))
(apply (list 'compound '(x*) '(plus* x* x*) GE)
       '(4))
  (eval '(plus* x* x*)
       (extend-env-with-new-frame '(x*) '(4) GE))
(eval '(plus* x* x*) E1)
```

GE
A
E1
name value
x* 4
28/40

Slide 15.7.28

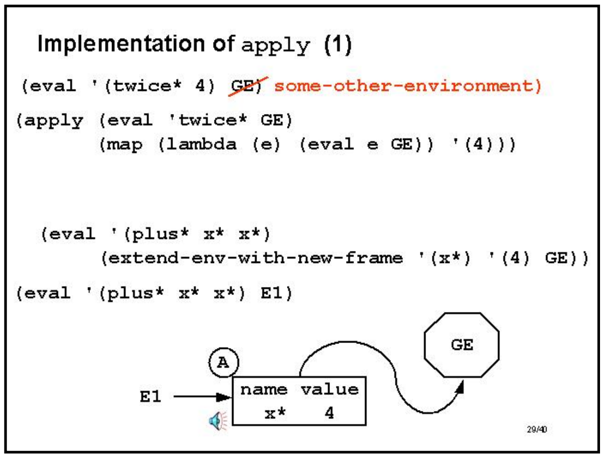
Conceptually, we have some new table structure, as shown here, with a new table for the binding of `x*` and a pointer to the enclosing environment (GE). We are now evaluating with respect to this environment E1.

Notice how we have reduced evaluation of one expression with respect to some environment to evaluation of a simpler expression with respect to a new, extended environment. There is a basic cycle of the environment model, and more specifically, of our evaluator.

Slide 15.7.29

Before we carry on tracing through the evaluation of this simpler expression with respect to this new environment, let's stop and look at an important point. Suppose we started off the overall evaluation of `(twice* 4)`, not in the global environment, but in some other environment. What would happen?

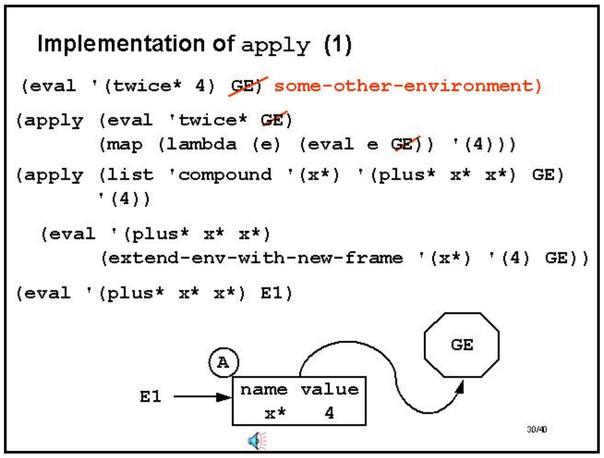
The application would have to do an evaluation of the subexpressions, and those would also be with respect to this new environment since that is passed down as part of the evaluation.



Implementation of apply (1)

```
(eval '(twice* 4) ~~GE~~ some-other-environment)
(apply (eval 'twice* GE)
       (map (lambda (e) (eval e GE)) '(4)))


  (eval '(plus* x* x*)
       (extend-env-with-new-frame '(x*) '(4) GE))
(eval '(plus* x* x*) E1)
```

GE
A
E1
name value
x* 4
29/40



Implementation of apply (1)

```
(eval '(twice* 4) ~~GE~~ some-other-environment)
(apply (eval 'twice* ~~GE~~)
       (map (lambda (e) (eval e ~~GE~~)) '(4)))
(apply (list 'compound '(x*) '(plus* x* x*) GE)
       '(4))
  (eval '(plus* x* x*)
       (extend-env-with-new-frame '(x*) '(4) GE))
(eval '(plus* x* x*) E1)
```

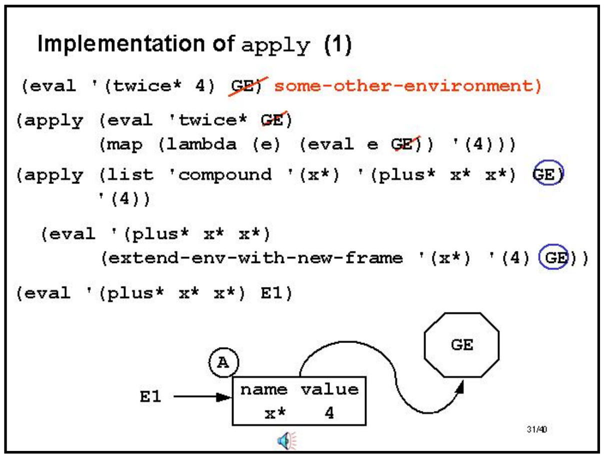
GE
A
E1
name value
x* 4
30/40

Slide 15.7.30

So at this stage we would use that same new environment.

Slide 15.7.31

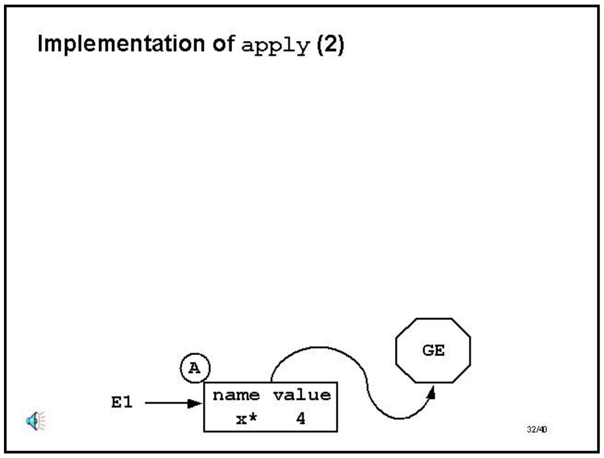
But when we get to the actual application, notice that here the environment **doesn't** change, because the global environment used here comes from the creation of the procedure. When we evaluated the `lambda*` expression that created the procedure object, part of that creation as a link to the environment in which the `lambda*` was evaluated. And that means that evaluating the body is going to extend this environment, not the environment we called it in. Thus, although we can evaluate this expression in any environment, the environment that gets inherited by the procedure is exactly that one in which it was created.



Implementation of apply (1)

```
(eval '(twice* 4) ~~GE~~ some-other-environment)
(apply (eval 'twice* ~~GE~~)
       (map (lambda (e) (eval e ~~GE~~)) '(4)))
(apply (list 'compound '(x*) '(plus* x* x*) GE)
       '(4))
  (eval '(plus* x* x*)
       (extend-env-with-new-frame '(x*) '(4) GE))
(eval '(plus* x* x*) E1)
```

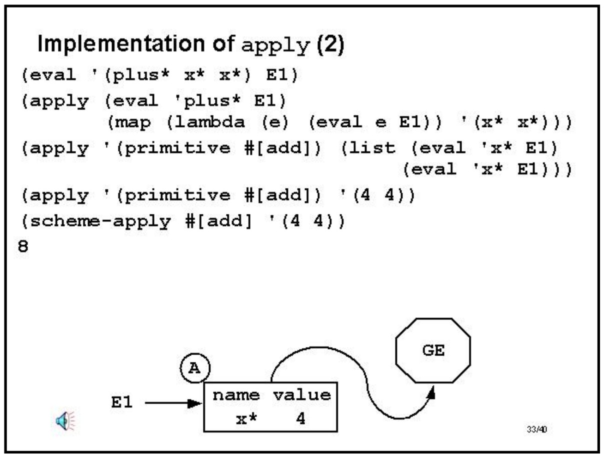
GE
A
E1
name value
x* 4
31/40



Implementation of apply (2)

Slide 15.7.32
So now let's finish off tracing out the evaluation of our expression. We now have a new environment, the one we just created. In it there is a local binding of the name `x*` to the value 4, and which is scoped by the global environment.

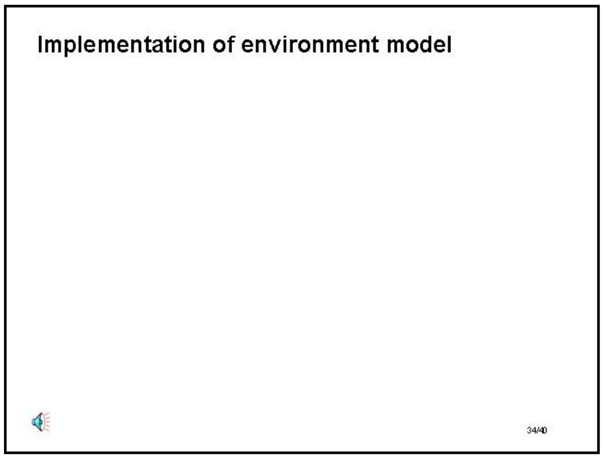
Slide 15.7.33
Within that environment we are evaluating the body of that procedure, `(plus* x* x*)`. Of course, the evaluator doesn't know that this is a body of a procedure; it has simply unwrapped previous evaluations to the stage of evaluating this procedure with respect to this environment. We can now quickly step through the evolution since it is much like earlier examples.
This reduces to an application. Getting the first value is simply a lookup of a binding, and the map operation will simply lookup the bindings of the other expressions. Note that the first binding will be found in GE, while the other bindings will be found in E1.
This reduces to the application of a primitive procedure to a set of values. This then becomes an application of a built in operation, which just reduces to the expected value.



Implementation of apply (2)

```
(eval '(plus* x* x*) E1)
(apply (eval 'plus* E1)
       (map (lambda (e) (eval e E1)) '(x* x*)))
(apply '(primitive #[add]) (list (eval 'x* E1)
                                 (eval 'x* E1)))
(apply '(primitive #[add]) '(4 4))
(scheme-apply #[add] '(4 4))
8
```

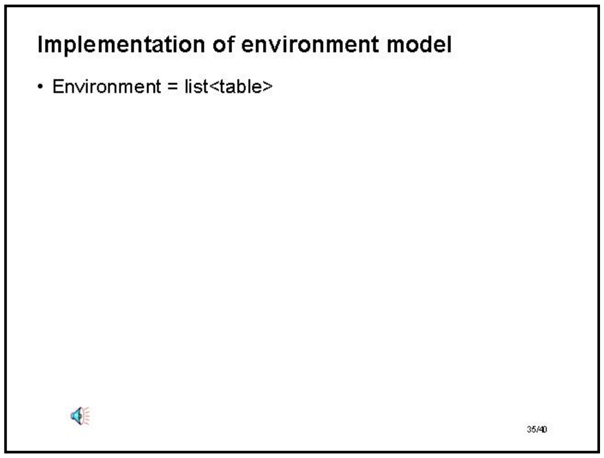
A
GE
E1
name value
x* 4
33/40



Implementation of environment model

Slide 15.7.34
The only thing left to be done is to actually implement the environment model, that is, dealing with frames and environments. Here, we get to make some design choices. Up till now we have treated it as an abstract data type, but we can make it much more concrete.

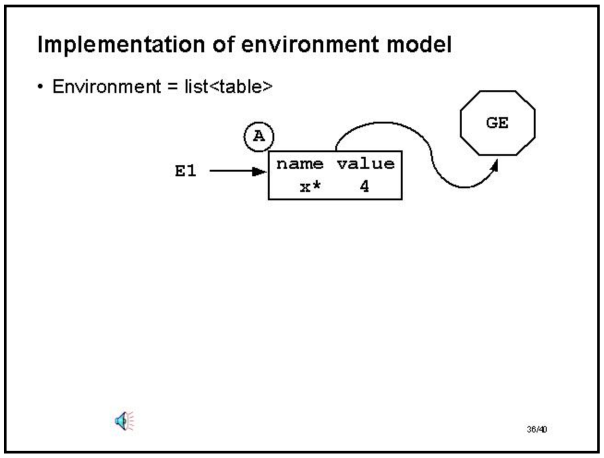
Slide 15.7.35
For the purposes of this exposition, we can just choose to represent an environment as a **list** of **tables**. We will use the table abstraction, as before (which we know we can implement in terms of list structure), and an environment will just be a list of these tables, sequenced together.



Implementation of environment model

- Environment = list<table>

35/40

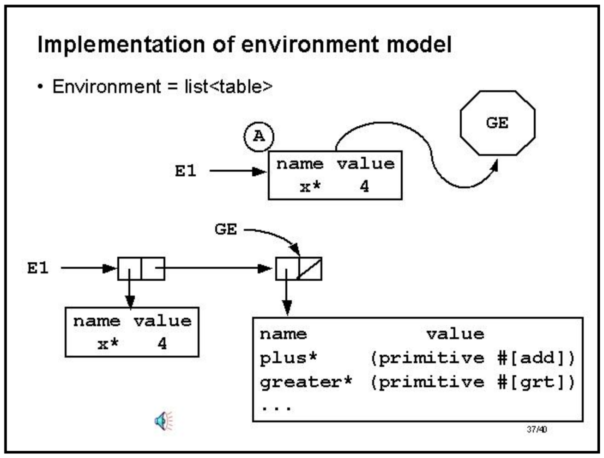


Implementation of environment model

- Environment = list<table>

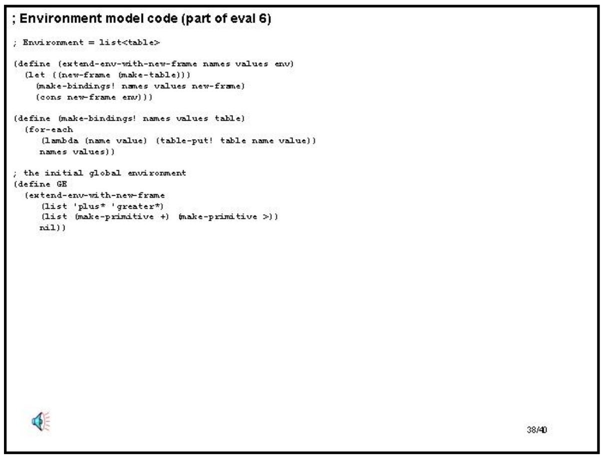
Slide 15.7.36
So, for example, given this abstract representation for the environment as a nested chain of tables starting with Frame A as a table, scoped by the global environment which is also a table, we can simply represent this much more concretely as ...

Slide 15.7.37
... this list structure. Now an environment points to a list, the first element of which is the first frame, the second element of which points to the remaining list of tables, which might be arbitrarily long. Eventually, the list will terminate in the global environment, which will contain the bindings for all the things that are established in the installation of the original environment. This is one way of representing an environment chain, which clearly illustrates the chaining together of frames in sequence.



Implementation of environment model

- Environment = list<table>



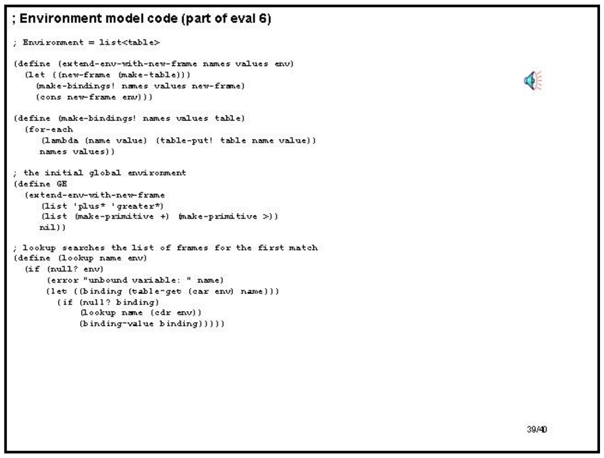
```
; Environment model code (part of eval 6)

; Environment = list<table>

(define (extend-env-with-new-frame names values env)
  (let ((new-frame (make-table)))
    (make-bindings! names values new-frame)
    (cons new-frame env)))

(define (make-bindings! names values table)
  (for-each
     (lambda (name value) (table-put! table name value))
     names values))

; the initial global environment
(define GE
  (extend-env-with-new-frame
     (list 'plus* 'greater*)
     (list (make-primitive +) (make-primitive >))
     nil))
```

38/40

Slide 15.7.38
Here is the code to actually implement environments. Most of this is bookkeeping detail, using lists.
Creating the initial global environment is simply a matter of starting with an empty environment, and then building a new frame that creates pairings of names for built in procedures with the actual representations of the primitives.

Slide 15.7.39

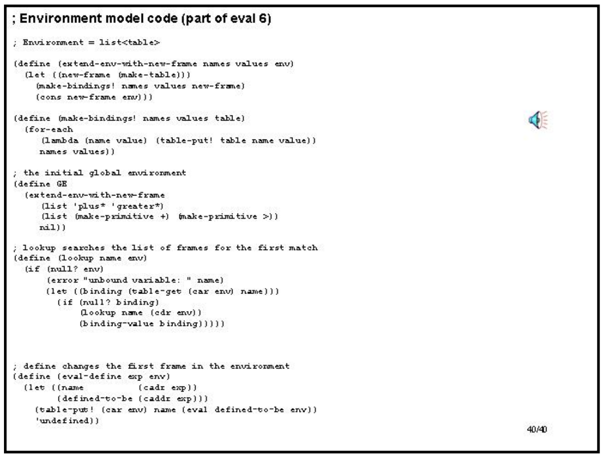
What else do we still need? `Lookup` needs to be a bit more explicit. To look up a value in an environment, we start by asking if the environment is empty. If it is, we have a problem, and we complaing that there cannot be a binding for this variable. Otherwise we do what we did before, the only difference is that we do the lookup with a `table-get` in the first frame of the sequence. If there is no binding there, we recursively move on to doing the lookup in the remainder of the environment. Thus we are just walking down the list of frames, looking in each one for a binding of the variable.



; Environment model code (part of eval 6)

```
; Environment = list<table>

(define (extend-env-with-new-frame names values env)
  (let ((new-frame (make-table)))
    (make-bindings! names values new-frame)
    (cons new-frame env)))

(define (make-bindings! names values table)
  (for-each
     (lambda (name value) (table-put! table name value))
     names values))

; the initial global environment
(define GE
  (extend-env-with-new-frame
     (list 'plus* 'greater*)
     (list (make-primitive +) (make-primitive >))
     nil))

; lookup searches the list of frames for the first match
(define (lookup name env)
  (if (null? env)
      (error "unbound variable: " name)
      (let ((binding (table-get (car env) name)))
        (if (null? binding)
            (lookup name (cdr env))
            (binding-value binding)))))
```

39/40



; Environment model code (part of eval 6)

```
; Environment = list<table>

(define (extend-env-with-new-frame names values env)
  (let ((new-frame (make-table)))
    (make-bindings! names values new-frame)
    (cons new-frame env)))

(define (make-bindings! names values table)
  (for-each
     (lambda (name value) (table-put! table name value))
     names values))

; the initial global environment
(define GE
  (extend-env-with-new-frame
     (list 'plus* 'greater*)
     (list (make-primitive +) (make-primitive >))
     nil))

; lookup searches the list of frames for the first match
(define (lookup name env)
  (if (null? env)
      (error "unbound variable: " name)
      (let ((binding (table-get (car env) name)))
        (if (null? binding)
            (lookup name (cdr env))
            (binding-value binding)))))


; define changes the first frame in the environment
(define (eval-define exp env)
  (let ((name          (cadr exp))
        (defined-to-be (caddr exp)))
    (table-put! (car env) name (eval defined-to-be env))
    'undefined))
```

40/40

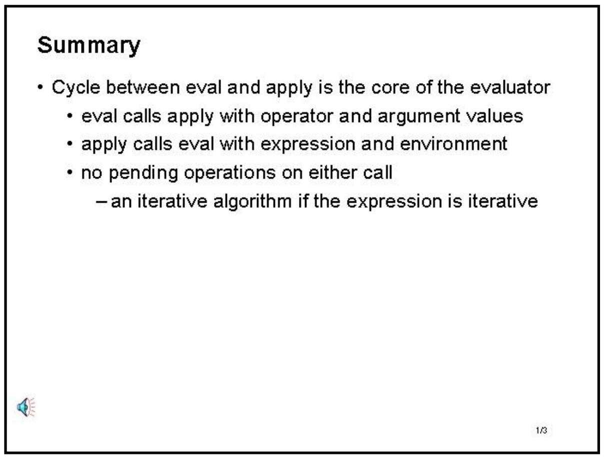
Slide 15.7.40

The only other change we need to make is to `define*`. We know this should change the first frame of the environment chain. Thus we should do a `table-put!` into the first frame in this sequence. Note that this is just list manipulation to create a proper sequence of frames constituting an environment.

6.001 Notes: Section 15.8

Slide 15.8.1

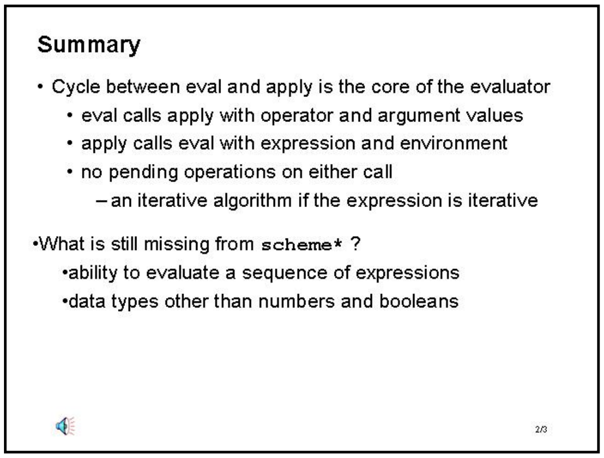
Thus, we have implemented our interpreter.
The key thing to note is the cycle that occurs with application of a procedure. `Eval` and `apply` form the core of the evaluator. `Eval` takes an expression and an environment and reduces this to `apply`ing an operator to a set of values, which in turn reduces to `eval`uating a new expression with respect to a new environment. They continue this cycle, unwinding the abstractions of the procedures, until it reduces the expression to primitive components.
In particular, notice that there are no pending operations on either call. `Eval` does not have any pending operations, and is inherently iterative. `Apply` similarly has no pending operations. Thus, if we are evaluating an iterative procedure, we will get an iterative behavior.



Summary

- Cycle between eval and apply is the core of the evaluator
  - eval calls apply with operator and argument values
  - apply calls eval with expression and environment
  - no pending operations on either call
    - an iterative algorithm if the expression is iterative

1/3

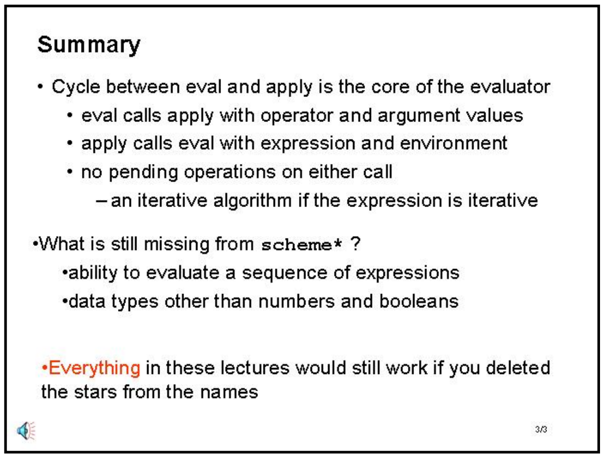

Summary

- Cycle between eval and apply is the core of the evaluator
  - eval calls apply with operator and argument values
  - apply calls eval with expression and environment
  - no pending operations on either call
    - an iterative algorithm if the expression is iterative

•What is still missing from `scheme*` ?
- •ability to evaluate a sequence of expressions
- •data types other than numbers and booleans

2/3

Slide 15.8.2
So it appears that we have worked up to a fairly full-fledged evaluator. What's is still missing from this interpreter? Only a couple of things: right now we can't evaluated a sequence of expressions. For example the bodies of our procedures expect only one expression. It would be fairly easy to add such a capability into our system. What else are we missing? We only have a couple of data types right now. So for example we might add in strings or other data types.


Summary

- Cycle between eval and apply is the core of the evaluator
  - eval calls apply with operator and argument values
  - apply calls eval with expression and environment
  - no pending operations on either call
    - an iterative algorithm if the expression is iterative

•What is still missing from `scheme*` ?
- •ability to evaluate a sequence of expressions
- •data types other than numbers and booleans

•Everything in these lectures would still work if you deleted the stars from the names

3/3

Slide 15.8.3
This then leads to the major punch line of this lecture. **Everything** in this lecture would still work if we just removed the *'s from the names! So what does that mean? It says that what we have done is describe the process of evaluation and interpretation in Scheme. We have really built a Scheme interpreter. Not as complete as the full-blown one, but the things missing are pretty minor. Literally everything we've done would still work if we removed the *'s which says we can build a Scheme interpreter. We happen to have done it on top of Scheme, but we could have done it on top of a more primitive language, or we could have done it in some other language, and this is **the central point**. By defining an evaluator or an interpreter we are defining the language. We are specifying what it means to evaluate expressions in this language, what is the legal syntax and the legal semantics, and once we have done that we have created a language that we can then use to solve other problems. This point we will return to, but this is the "take home" message of this lecture.