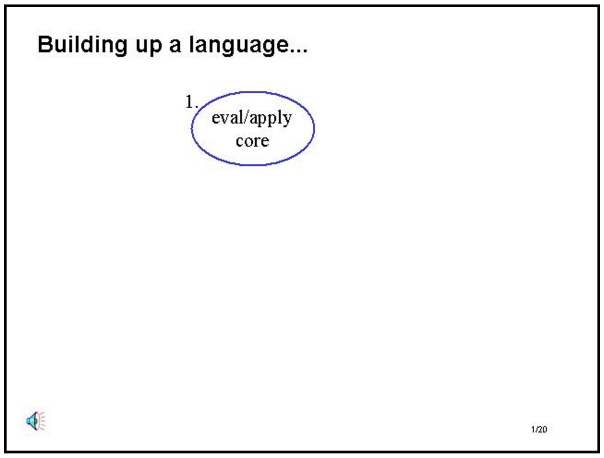
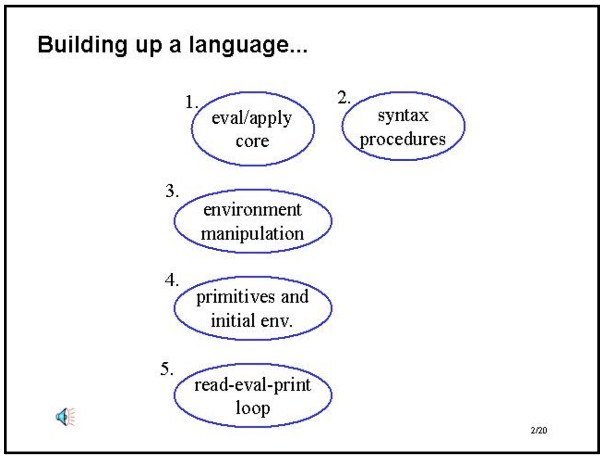
6.001 Notes: Section 16.1



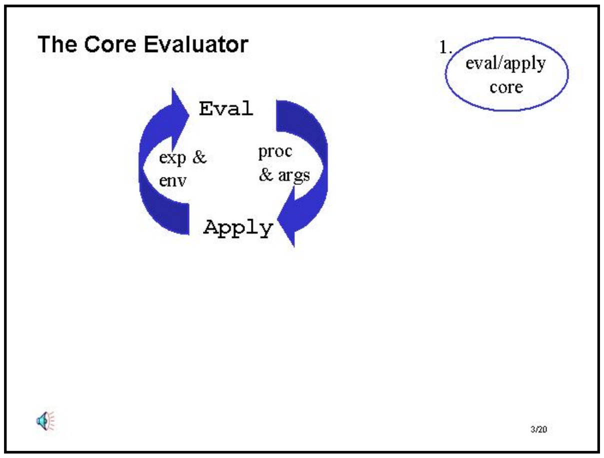
Slide 16.1.1
Last time, we completed building our evaluator. And as you saw, we slightly misled you. We started off saying we were going to user Scheme's lexical analyzer and parser, but then build our own evaluator, which we did initially for arithmetic expressions, then for defines, then for applications, and eventually we ended up with something that basically looked like Scheme's evaluator, written in Scheme!
Today, we are going to build on that idea, by examining the **actual** Scheme evaluator. We will run through a quick, but grand tour of the full evaluator, looking at several key ideas. First, remember that we are basically describing the process of evaluation, which in our case means making the environment model a concrete set of procedures. Second, the essential message is that by defining the process of evaluation, we are also defining our language. This means that the evaluator design then provides the basis on which we can create abstractions, especially procedural abstractions, as it provides the mechanism for unwinding the abstraction back down to primitive pieces when we want to get an actual value. And finally, given that designing an evaluator is essentially equivalent to defining a language, we are going to look at how variations in a Scheme evaluator can lead to very different language behavior.


Building up a language...
1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop
2/20

Slide 16.1.2
To do this, we are going to have to look at several different parts of the language design. We will start with the core of `eval` and `apply`, then look at how we support the syntax of the language, how we create and manipulate the environments that let us look up values that are associated with that syntax, and then how primitives are installed into the initial or global environment. Finally, we will put together the overall infrastructure for letting a user interact with the evaluator, and we will see all of these pieces as we quickly walk through our full evaluator.
As with previous lectures, there is a code handout that goes with this lecture, and we suggest you print out a copy and have it available as we walk through our discussion.

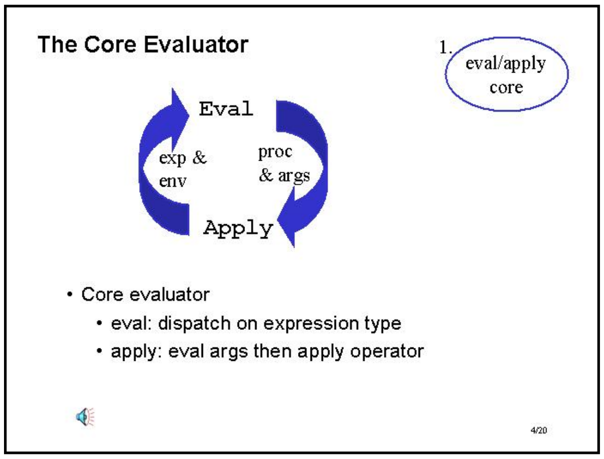
Slide 16.1.3

Let's start with the heart of the interpreter. We have already seen this with our simple interpreter from the last lecture. The essence of the evaluator is a tight loop, in which the **evaluation** of an expression with respect to an environment reduces in the general case to an **application** of a procedure to a set of arguments. This in turn generally reduces to an evaluation of a simpler expression (the body of the procedure) with respect to a new environment (one in which the formal parameters of the procedure have been bound to the arguments passed in).
This loop continues, unwinding expressions, until it reaches the application of a primitive procedure to primitive values or the evaluation of a primitive data object.



The Core Evaluator

3/20



The Core Evaluator

- Core evaluator
  - eval: dispatch on expression type
  - apply: eval args then apply operator

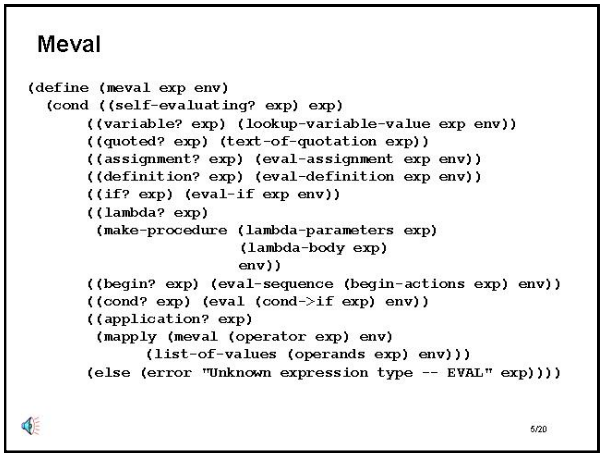
4/20

Slide 16.1.4

Our convention is that we will use an `eval` that does dispatch on type. Thus, it checks the expression type, and based on that, sends the expression to a procedure designed to handle that type of expression. Our convention on `apply` is that it will first evaluate the arguments, and then apply the procedure that is the value of the first argument to the values of the others.

Slide 16.1.5

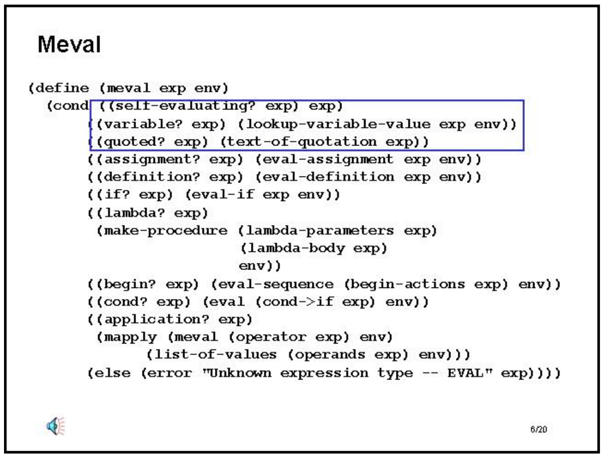
So here is our implementation of `eval`. Notice its form. First, it is a dispatch on type, so it checks the expression to figure out which type matches. Once it does, it sends the expression to a procedure specific to that type. Notice that we are assuming a data abstraction for checking types. We are not making any assumptions about how types are represented, but are rather using a set of procedures to check each type. This will allow us to cleanly add to or alter our types, without having to change the evaluator directly.



Meval

```
(define (meval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (mapply (meval (operator exp) env)
               (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))
```

5/20



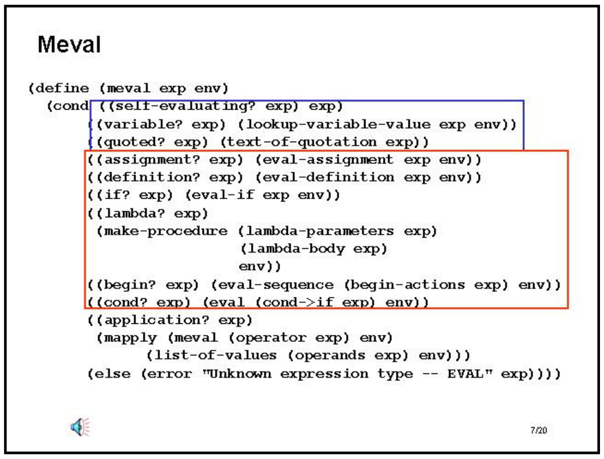
Meval

```
(define (meval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (mapply (meval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))
```

6/20

Slide 16.1.6
Notice the order in which we check things in `eval`. We first start out by checking for primitives, things like self-evaluating expressions, or variables. These are easy things to deal with.

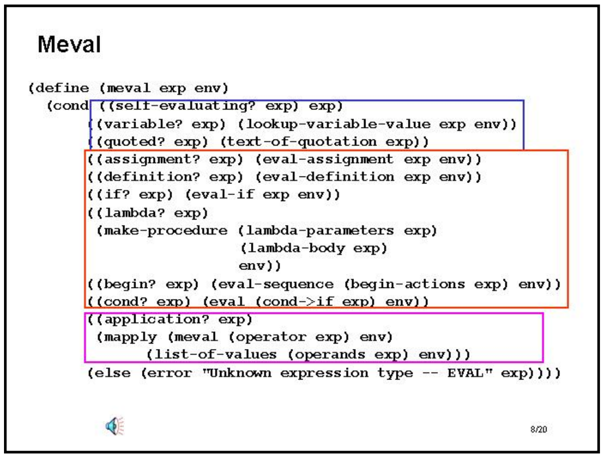
Slide 16.1.7
Next, we check out the **special forms**. Remember that this is an expression that does not follow the normal rules of evaluation for compound expressions. These are things like assignment or definition, in which we only want to evaluate one of the subexpressions, or things like `if`, where we know we want to evaluate in a different order. Each of these expressions we will treat separately, with a specific procedure to deal with that kind of expression. We've added a couple of new ones, and we will come back to those shortly. The issue to note is that this section deals with all the special forms.



Meval

```
(define (meval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (mapply (meval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))
```

7/20



Meval

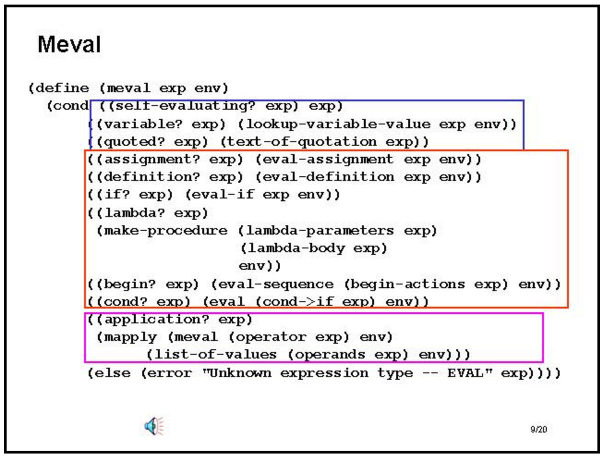
```
(define (meval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (mapply (meval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))
```

8/20

Slide 16.1.8
Finally, we get to an application. This is the case where we are treating an expression that has a set of subexpressions that we will evaluate, then apply the operator (or value of the first subexpression) to all the rest of them. We have left `application?` as an abstraction to check if something is an application, but as you saw with our earlier evaluators, most likely we will just make sure this is a combination, meaning we will assume that if the expression is a compound expression but not a known special form, then it must be an application.

Slide 16.1.9

In general, however, we see that our evaluator has exactly the kind of form that we slowly evolved in our previous examples. Given an expression and an environment, `eval` will check the type of the expression, using some abstractions that we will get to, to see if it is a primitive, or if it is a special form, or ultimately if it is an application. In each case, `eval` dispatches that expression to a specific procedure. We have already seen many of these pieces. We know that for variables, we should just look up the value of the variable in the environment. We know that for quoted expressions, we should simply grab the expression as tree structure and return it. Similarly for the special forms: a definition should create a new binding for a name and a value in the environment; an if should change the order of evaluation of the subexpressions.

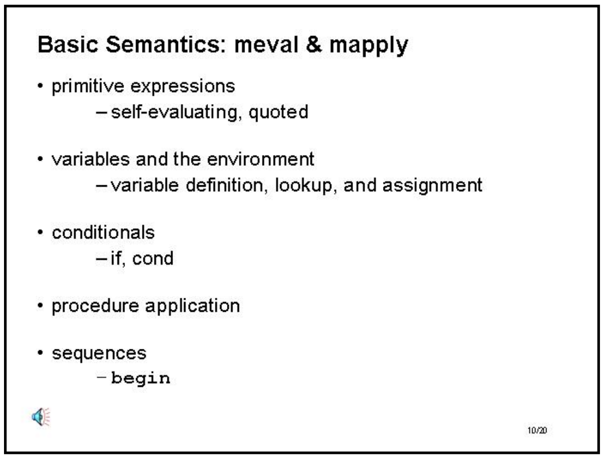


Meval

```
(define (meval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (mapply (meval (operator exp) env)
                 (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))
```

9/20

For an application, we should evaluate the first subexpression (the operator), then evaluate all the other subexpressions, and apply that operator to that list of values. Notice that I have slightly misled you because we don't have to assume that the operator is the **first** subexpression. `Operator` is simply some data abstraction that will get the appropriate subexpression but it could in fact be some piece other than the first or leftmost one.



Basic Semantics: meval & mapply

- primitive expressions
  - self-evaluating, quoted
- variables and the environment
  - variable definition, lookup, and assignment
- conditionals
  - if, cond
- procedure application
- sequences
  - `begin`

10/20

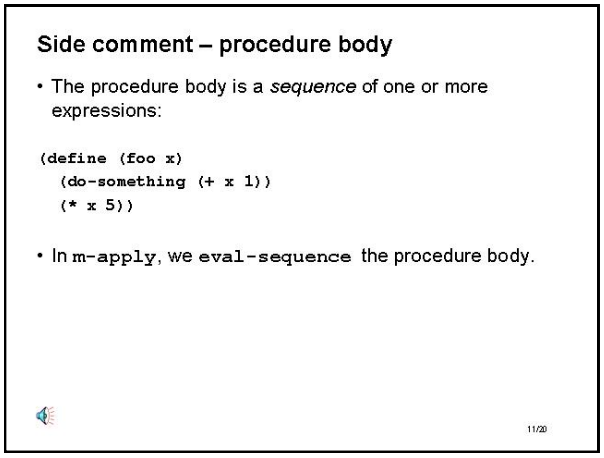
Slide 16.1.10

Here is a quick synopsis of what we see on that evaluator. It dispatches on type: first for primitives, either self-evaluating things or things that should not be evaluated; then for expressions that deal with variables and their manipulation within the environment (creating, looking up, or changing values of variables); then for conditionals, ways of branching depending on the value of an expression; then procedure application.

Note there are several new forms here that we will have to define: quoted objects, cond, and begin. Let's take a look at some of these.

Slide 16.1.11

In our evaluators from the previous lectures, when we applied a procedure to a set of arguments, we saw that reduced to simply evaluating the body of the procedure with respect to a new environment. Evaluation of the body was simply a case of using `eval` on that expression, and that was because we assumed the body was just a **single** expression. Before we introduced mutation into our language, this made perfect sense, because the value of the body would be the value of the last expression in it, and it only made sense to have one expression, since any other expression's value in a body would be lost.



Side comment – procedure body

- The procedure body is a *sequence* of one or more expressions:

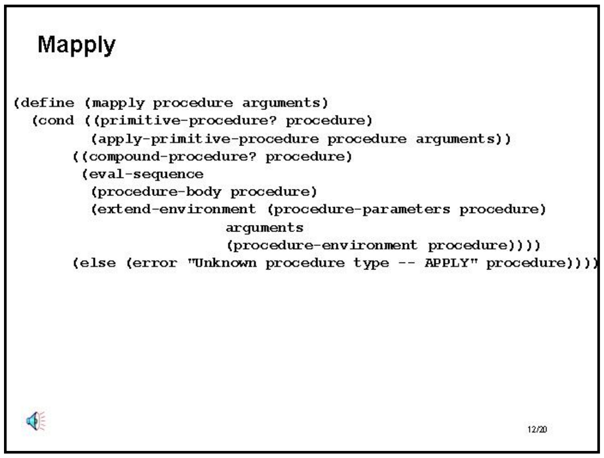
```
(define (foo x)
  (do-something (+ x 1))
  (* x 5))
```

- In `m-apply`, we `eval-sequence` the procedure body.

11/20

Once we have mutation, however, expressions can do things other than return values: they can create side effects. So a generalization for an evaluator is to let the body of a procedure be a sequence of one or more expressions. That is shown here, in which we define `foo` to be a procedure whose body has two expressions. The first does something, probably a side effect, and the second of

which computes and returns a value.
We still have to figure out how to do evaluation of a sequence (we'll get to that in a second) but given the idea that evaluating a sequence could take a series of expressions, evaluate them in order and return the value of the last one, we can now let bodies have multiple expressions, and inside of `apply` we can generally evaluate the body as a sequence, not a single expression.



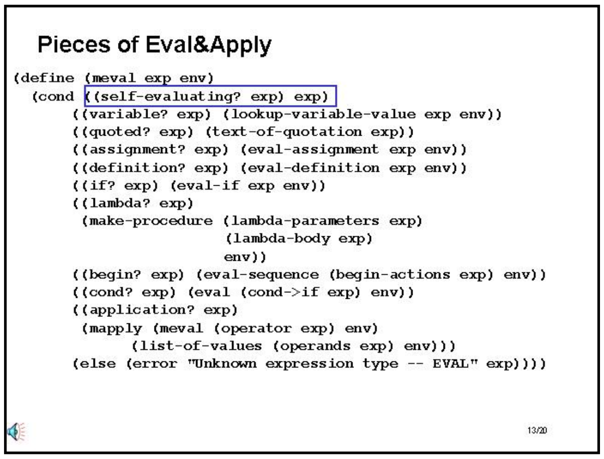
Mapply

```
(define (mapply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment (procedure-parameters procedure)
                              arguments
                              (procedure-environment procedure))))
        (else (error "Unknown procedure type -- APPLY" procedure))))
```

12/20

Slide 16.1.12
Thus, our `apply`, our way of dealing with applications of procedures to arguments, will be slightly different. As before, it will have a way of dealing with primitive procedures, in this case just sending the expression to the underlying machine primitive. For compound procedures (application of something we built using a `lambda`), we make a slight change. We still create a new environment, binding the parameters of the procedure (obtained using the appropriate data abstraction) to the values of the arguments in a frame that extends the environment specified by the procedure object. Within this new environment, we will evaluate the body of the procedure, but notice here we will evaluate it as a sequence, not as a single expression. Thus, we will treat the body as a set of expressions, evaluate each one in order, and then return the value of the last one as the value of the entire application.
With that, we now see the intertwining of `eval` and `apply`. Let's take a look at some of the specific pieces.

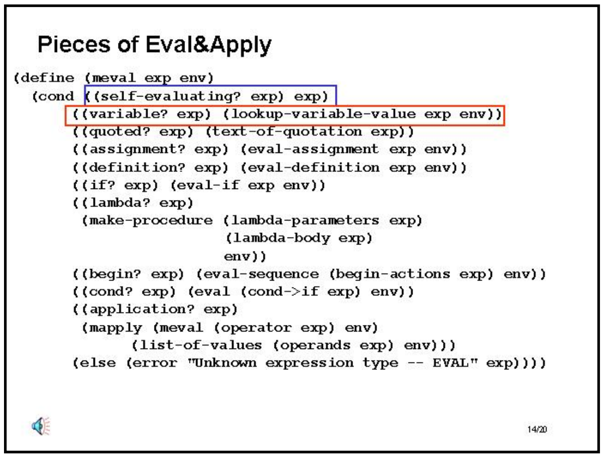
Slide 16.1.13
First, self-evaluating expressions. Remember that `exp` is just a tree structure, and in this case, we simply return that tree structure directly. Thus numbers get returned without any additional work.



Pieces of Eval&Apply

```
(define (meval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (mapply (meval (operator exp) env)
                 (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))
```

13/20



Pieces of Eval&Apply

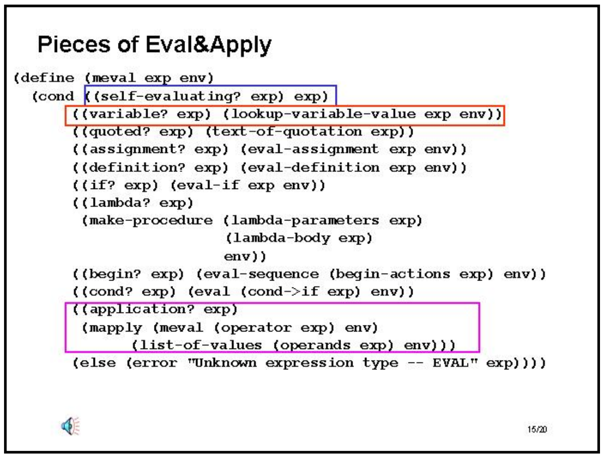
```
(define (meval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (mapply (meval (operator exp) env)
                 (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))
```

14/20

Slide 16.1.14
If an expression is just a variable (a symbol) we will just look up that variable's value in the environment, and we'll deal with details of that shortly.

Slide 16.1.15

Next, let's skip down to procedure applications. With our change, we can see that the evaluator gets the value of the operator by recursively evaluating that subexpression, then gets a list of values of the other expressions (note that we explicitly require a **list** here) by evaluating each piece in turn and constructing a list. Note, however, that our data abstraction isolates the issue of the order of the arguments from our evaluator. We don't know if the operator is the first expression or not in this case. `Operator` will simply select out whatever we decide on in terms of syntax for our language. And as we saw, `apply` will now evaluate the sequence of expressions that it assumes the body contains.



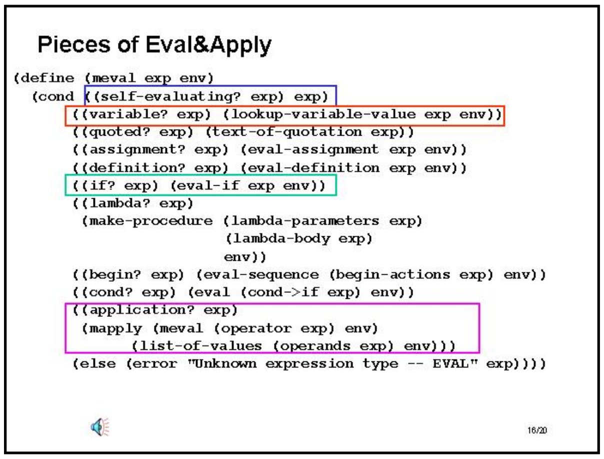
Pieces of Eval&Apply

```
(define (meval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (mapply (meval (operator exp) env)
              (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))
```

15/20

Slide 16.1.16

What about `if`s? We know that an `if` expression should take a set of expressions, evaluate the first subexpression (or rather we are assuming it is the first subexpression) and then depending on that value, either evaluates the consequent or the alternative.



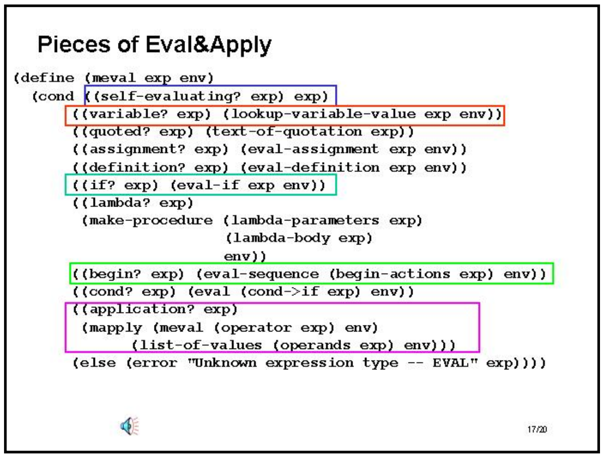
Pieces of Eval&Apply

```
(define (meval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (mapply (meval (operator exp) env)
              (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))
```

16/20

Slide 16.1.17

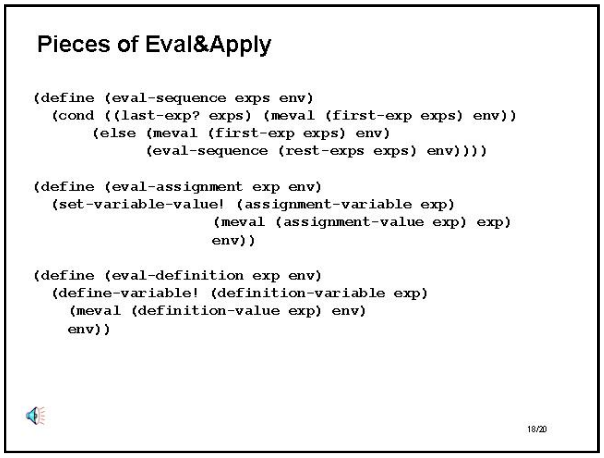
And what about `begin`s? Remember that a sequence is either something identified by an explicit `begin` statement or is used as the body of a procedure, as we just saw. In this case, we want to extract the set of expressions, and evaluate them. We use the following trick, shown on the next slide.



Pieces of Eval&Apply

```
(define (meval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (mapply (meval (operator exp) env)
              (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))
```

17/20



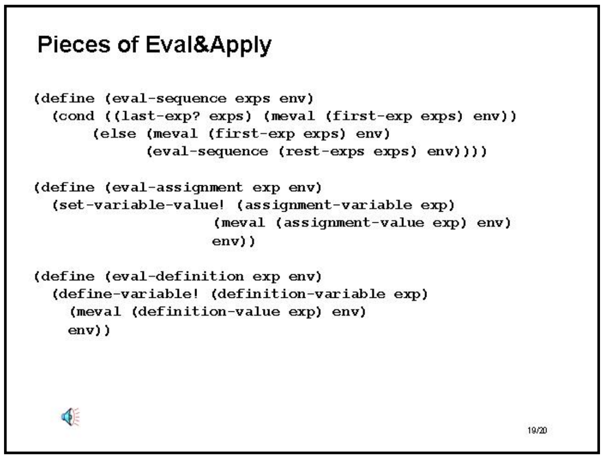
Pieces of Eval&Apply

```
(define (eval-sequence exps env)
  (cond ((last-exp exps) (meval (first-exp exps) env))
        (else (meval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (meval (assignment-value exp) exp)
                       env))

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
    (meval (definition-value exp) env)
    env))
```

18/20

Slide 16.1.18

Our plan is to evaluate each of the expressions in order. When we get to the last one, we should return its value as the value of the entire sequence. We will bury some of the details behind data abstractions, but we can see the general form for evaluating a sequence. If we have the last expression, we will simply evaluate it and return the value. If it is not, we evaluate it, then recursively walk down the sequence and do the same thing.



Pieces of Eval&Apply

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps) (meval (first-exp exps) env))
        (else (meval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (meval (assignment-value exp) env)
                       env))

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
    (meval (definition-value exp) env)
    env))
```

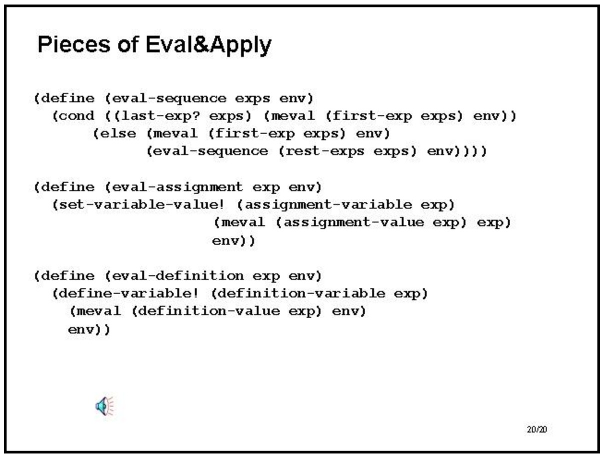
19/20

Slide 16.1.19

Expressions like definitions and assignments either create or change existing bindings of variables and values in the environment. Notice the form, however. In both cases, we get out the part of the expression that corresponds to the new value, and actually **evaluate** it, by recursively applying `meval` to it.

We simply get out the part of the expression that corresponds to the variable as a tree structure manipulation, however, without evaluation. Then we do the appropriate manipulation of the environment.

Notice how by using data abstractions we have not specified what order the expressions will take. While we will eventually decide that, from the perspective of this code, any change in that order will not affect the evaluation process.



Pieces of Eval&Apply

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps) (meval (first-exp exps) env))
        (else (meval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (meval (assignment-value exp) exp)
                       env))

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
    (meval (definition-value exp) env)
    env))
```

20/20

Slide 16.1.20

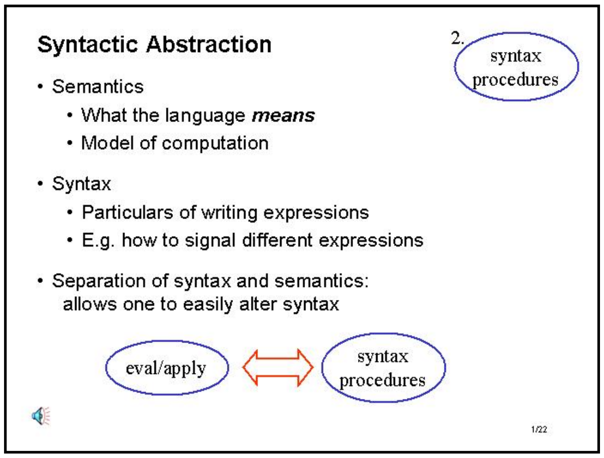
So there is the whirlwind tour of `eval` and `apply`. We have buried some details behind some data abstractions, which we will address shortly. The part that you see here really is the heart of evaluation.

The essence of the evaluator is a tight loop, in which the **evaluation** of an expression with respect to an environment reduces in the general case to an **application** of a procedure to a set of arguments. This in turn generally reduces to an evaluation of a simpler expression (the body of the procedure) with respect to a new environment (one in which the formal parameters of the procedure have been bound to the arguments passed in), while inheriting values from other environments.

6.001 Notes: Section 16.2

Slide 16.2.1

So what have we done so far? Basically we have defined the semantics of our language. By defining `eval` and `apply` we have specified what the language means and the model of computation we are going to use. Notice that we have done this in terms of data abstractions. Everything we have used to pull out list structure from expressions has used an abstraction. There are no `car`s and `cdr`s anywhere in the code so far.

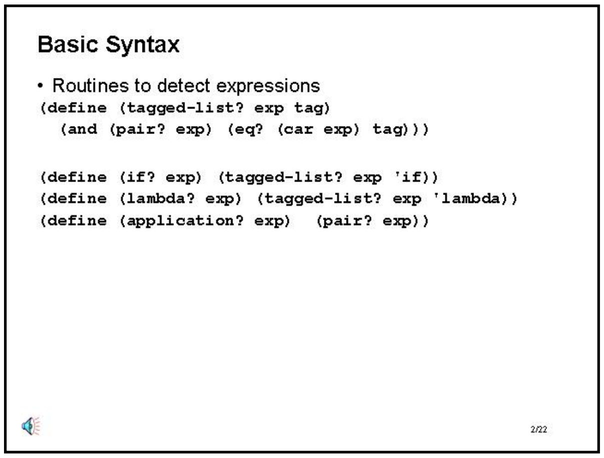


Syntactic Abstraction

- Semantics
  - What the language ***means***
  - Model of computation
- Syntax
  - Particulars of writing expressions
  - E.g. how to signal different expressions
- Separation of syntax and semantics: allows one to easily alter syntax

2.
syntax procedures
eval/apply
syntax procedures
1/22

This is important, because we have separated the syntax of the language from the semantics of the language. But eventually we do need to focus on the syntax, the details of how we write legal expressions in our language. This will force us to specify things like how to identify kinds of expressions, and the order of arguments. At the same time, this separation of syntax from semantics is extremely useful, as it allows us to very easily change the syntax without having to change the actual evaluator. We simply change the interface to the abstractions. We are now going to examine this issue, by looking at details of implementing the abstractions and by looking at changes to that abstraction.



Basic Syntax

- Routines to detect expressions

```
(define (tagged-list? exp tag)
  (and (pair? exp) (eq? (car exp) tag)))

(define (if? exp) (tagged-list? exp 'if))
(define (lambda? exp) (tagged-list? exp 'lambda))
(define (application? exp)  (pair? exp))
```

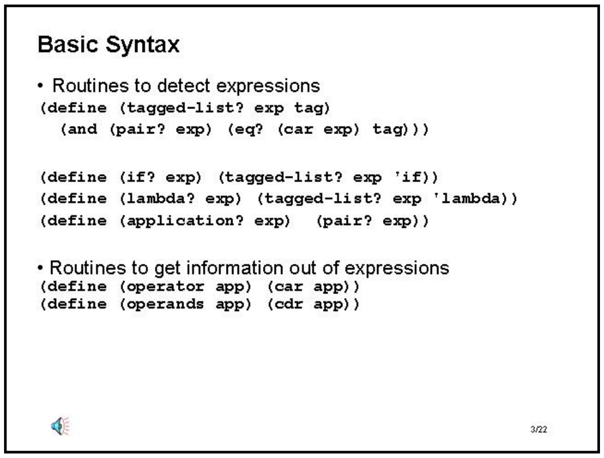
Slide 16.2.2

The second page of the code handout contains all of this in much more detail. Here we will simply highlight some of the key issues in syntax.

First, we need a set of routines to detect different kinds of expressions. An easy way to do this is to have each one of them check to see if it is a "tagged" list. So, for example, `if?` will check to see if the expression is a list, that is something constructed out of `cons` pairs. It will then check to see if the first element of that list is the symbol `if`. Ditto for `lambda`. For application, we will use a slightly different form. We will assume that anything that has not been caught by one of the tag checkers for a special form but is in fact a list of expressions, we can treat as an application. This means that we may get into some trouble but it is the easiest way of allowing us to generalize to having any kind of procedure applied to any set of expressions as long as that procedure was created using our `lambda`.

Slide 16.2.3

Given that we can detect different kinds of expressions and ship them off to the procedures that handle them, those procedures will also need to have ways of getting information out of expressions. They will have to have ways of pulling out pieces by walking down the list structure. So, for example, if we have an application, we need a way of getting out the operator, and the other expressions. Here we **will** assume that the first subexpression is the operator, so we use `car` to get it out of the expression. And we will assume that operands is a list of all the remaining expressions, so we use `cdr` to get that part. Of course we could have made a different design choice, as we will see. The key point is that we now have an abstraction that pulls out pieces to pass on to the evaluator.



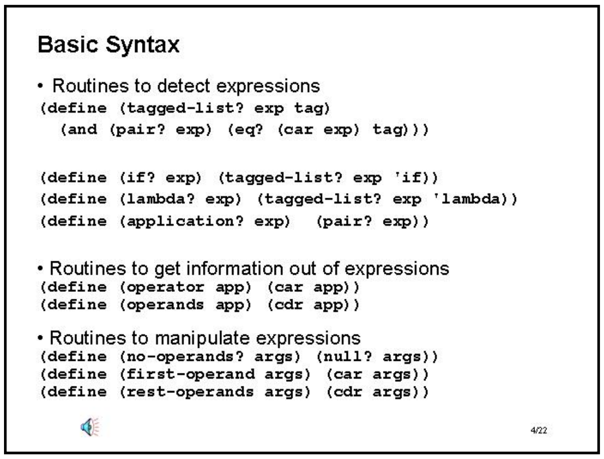
Basic Syntax

- Routines to detect expressions

```
(define (tagged-list? exp tag)
  (and (pair? exp) (eq? (car exp) tag)))

(define (if? exp) (tagged-list? exp 'if))
(define (lambda? exp) (tagged-list? exp 'lambda))
(define (application? exp)  (pair? exp))
```

- Routines to get information out of expressions

```
(define (operator app) (car app))
(define (operands app) (cdr app))
```


Basic Syntax

- Routines to detect expressions

```
(define (tagged-list? exp tag)
  (and (pair? exp) (eq? (car exp) tag)))

(define (if? exp) (tagged-list? exp 'if))
(define (lambda? exp) (tagged-list? exp 'lambda))
(define (application? exp)  (pair? exp))
```

- Routines to get information out of expressions

```
(define (operator app) (car app))
(define (operands app) (cdr app))
```

- Routines to manipulate expressions

```
(define (no-operands? args) (null? args))
(define (first-operand args) (car args))
(define (rest-operands args) (cdr args))
```

4/22

Slide 16.2.4

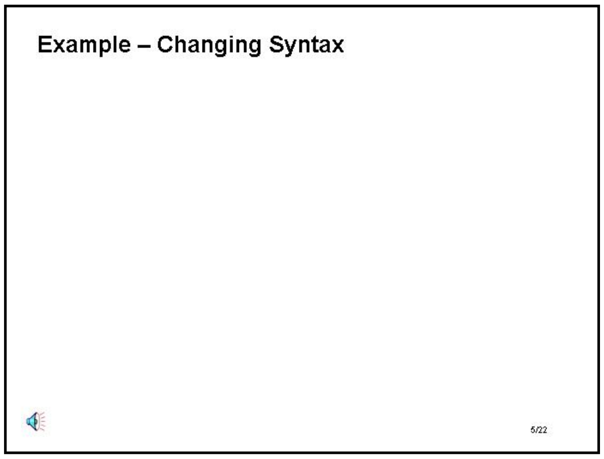
And we will need routines that manipulate expressions, that is, walk along the expressions finding pieces. For example, if we are applying a procedure to a set of arguments, we need to get out different parts of the argument list. So we will have something to check if there are arguments, something to get the next operand, and something to get the other operands. This is just list structure that we are manipulating. Notice that we have made an explicit choice about order, assuming the operands are arranged in a left-to-right ordering.

Overall, our syntax (see page 2 of the code) is defined by a set of expressions to deal with the list structure that is passed in to the evaluator. These procedures walk down the list structure to test for kind of expression, to get out pieces or otherwise manipulate that list structure.

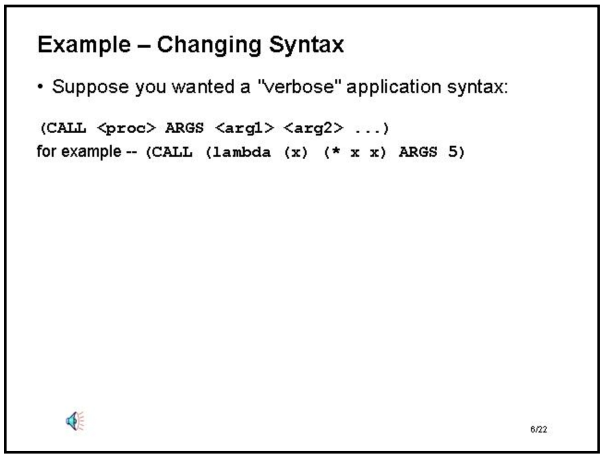
Slide 16.2.5

The rest of the code on page 2 of the handout just fills out the rest of the details of the syntax, the representation of expressions. Look it over to be sure you understand it, but it is basically the same sort of material: tagged lists for identifying types, list operations (car and cdr) to get out pieces.

As we said earlier, one of the reasons for using data abstractions everywhere within `eval` is to let us separate the syntax from the semantics. `Eval` and `apply` define the semantics, how expressions get their values in this language. The syntax tells us how to write legal expressions in this language. By doing that separation, we can make changes to the syntax without affecting the semantics. Here is one such example.


Example – Changing Syntax

5/22


Example – Changing Syntax

- Suppose you wanted a "verbose" application syntax:

```
(CALL <proc> ARGS <arg1> <arg2> ...)
```

for example -- `(CALL (lambda (x) (* x x)) ARGS 5)`

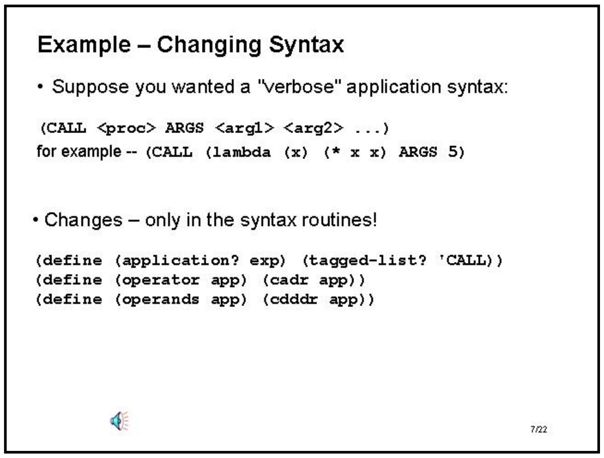
6/22

Slide 16.2.6

Suppose I decide that rather than having the convention that the operator is the first subexpression of a combination, and the operands are all the other subexpressions, instead I want to be much more verbose. I might want expressions like the one shown here, explicitly identifying the procedure and the arguments. How would I make this change to my language?

Slide 16.2.7

We have already hinted at the answer. **All** we need to do is change the syntax. For example, now checking to see if something is an application involves looking to see if the first element of the list is the special symbol `CALL`. This is clearly different from our earlier version, where we assumed any combination that was not a special form was an application. Here, we are explicitly identifying applications.



Example – Changing Syntax

- Suppose you wanted a "verbose" application syntax:

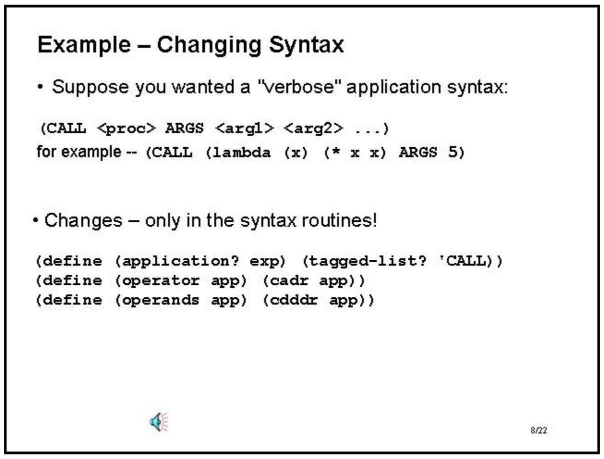
```
(CALL <proc> ARGS <arg1> <arg2> ...)
```

for example -- `(CALL (lambda (x) (* x x)) ARGS 5)`

- Changes – only in the syntax routines!

```
(define (application? exp) (tagged-list? 'CALL))
(define (operator app) (cadr app))
(define (operands app) (cdddr app))
```

7/22



Example – Changing Syntax

- Suppose you wanted a "verbose" application syntax:

```
(CALL <proc> ARGS <arg1> <arg2> ...)
```

for example -- `(CALL (lambda (x) (* x x)) ARGS 5)`

- Changes – only in the syntax routines!

```
(define (application? exp) (tagged-list? 'CALL))
(define (operator app) (cadr app))
(define (operands app) (cdddr app))
```

8/22

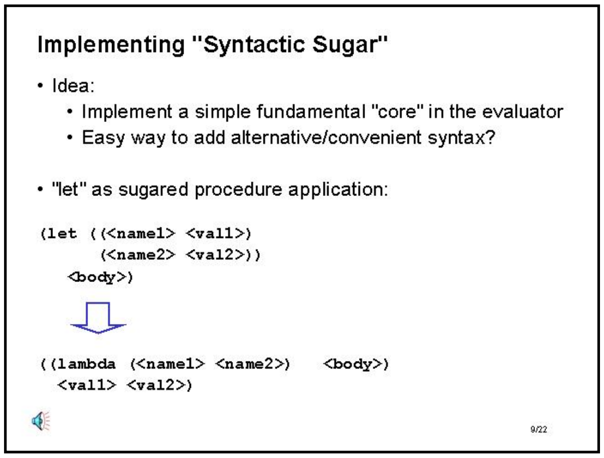
Slide 16.2.8

So how do I get out the pieces? Before, we would have gotten the operator as the first subexpression, here it is the **second** subexpression, since the first subexpression is the symbol `CALL`. And for the operands, we have to skip past the `ARGS` to get the right parts. Again, the key point: all I have changed is the syntax, the routines that walk down the tree structure and pull out the pieces. Nothing has changed in `eval` or `apply`.

Slide 16.2.9

The second reason for separating syntax from semantics is that it allows us to easily accommodate alternative forms for expressions. We often call this "syntactic sugar", meaning that it looks nicer, but it actually just coats the same underlying idea, with a sweeter way of expressing the same thing. Let me show you an example. Remember `let`?

We could treat `let` as a special form and write a handler for it. We could have something that handles dispatches, having detected expressions of this type (let). But we could also realize that `let` is just a cleaner way of creating a procedure, then applying it to capture some local state variables.



Implementing "Syntactic Sugar"

- Idea:
  - Implement a simple fundamental "core" in the evaluator
  - Easy way to add alternative/convenient syntax?
- "let" as sugared procedure application:

```
(let ((<name1> <val1>)
      (<name2> <val2>))
   <body>)
```

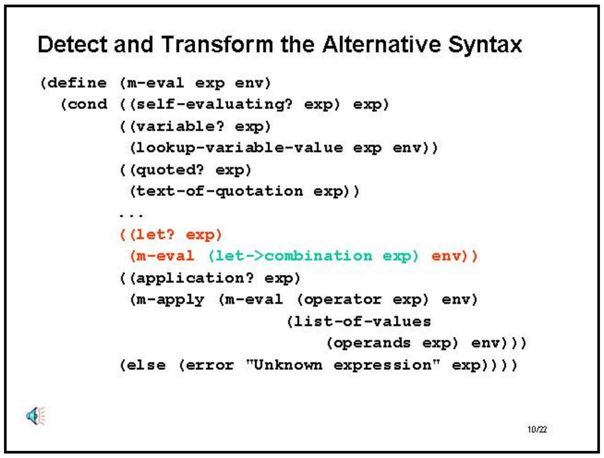
$\Downarrow$

```
((lambda (<name1> <name2>)   <body>)
  <val1> <val2>)
```

9/22

Said another way, a `let` expression is really just the same as the expression shown here. That is, we create a lambda, whose argument list is the same as the arguments for the bindings in the `let`, whose body is just the body of the `let`, and that `lambda` would then be applied to expressions whose values we want to bind to those names. These are equivalent forms. As we saw in the environment model, they create exactly the same kind of structure.

So rather than building a special form for `let`, let's see how we could use syntax and syntactical manipulation to convert this form of `let` into an application of a procedure to a set of arguments.



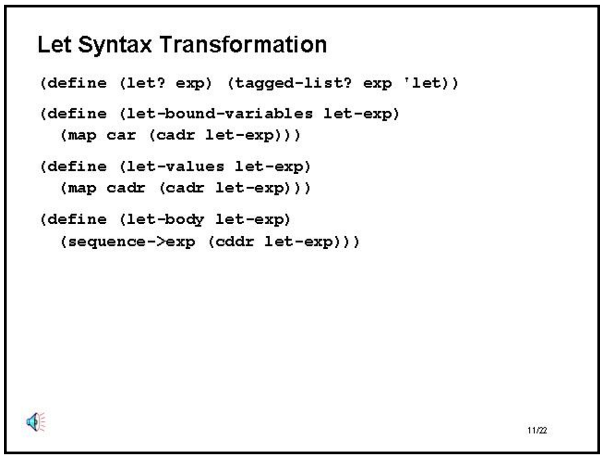
Detect and Transform the Alternative Syntax

```
(define (m-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp)
         (lookup-variable-value exp env))
        ((quoted? exp)
         (text-of-quotation exp))
        ...
        ((let? exp)
         (m-eval (let->combination exp) env))
        ((application? exp)
         (m-apply (m-eval (operator exp) env)
                  (list-of-values
                        (operands exp) env)))
        (else (error "Unknown expression" exp))))
```

10/22

Slide 16.2.10
First, here is the change we will make inside of our evaluator. We will have something that detects lets. What we will do in this case is write something that manipulates the syntax, turns a let into a combination, and then simply evaluates that combination, i.e. recursively calls `meval` treating this as a normal combination.
Thus in this case the dispatch is different. Rather than going off to a procedure, it manipulates syntax, then evaluates the new expression.

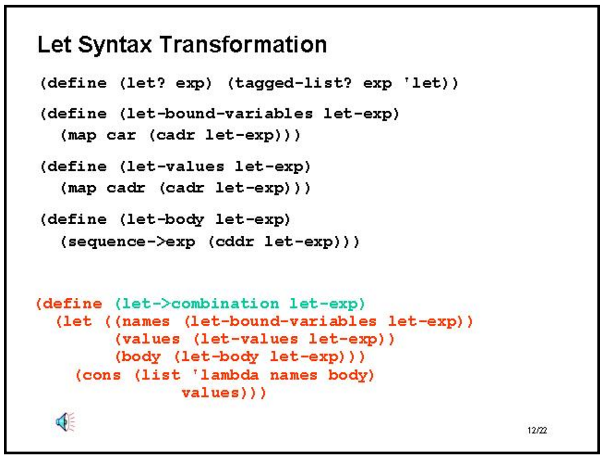
Slide 16.2.11
Thus now we just need to figure out how to rewrite the tree structure that represents a `let` expression into something that looks like a procedure application.
First, we will need a way to check if we have a `let` expression, which is just a tagged list check, as before. Next, we need to pull out the variables of the `let`, we need to pull out the values to which they will be bound, and we need to pull out the body. Notice how `let-bound-variables` takes in one of these tree structures, will get the `cadr` which gives us the set of let clauses, and then will map `car` down that list. As we will see, this pulls off all the variable names from the clauses.
Getting the values is almost the same, but maps `cadr` down the list of clauses. For the body, we simply take the `cddr` of the expression to get everything but the `let` tag, and the variable clauses. We convert that into a single expression by wrapping a sequence label (a `begin`) around it.
Ignoring for the moment the specific details, these three procedures are basically walking through the tree structure corresponding to a `let` expression, and pulling out the right pieces: the list of variables, the list of values and the body (which gets converted into a sequence).



Let Syntax Transformation

```
(define (let? exp) (tagged-list? exp 'let))

(define (let-bound-variables let-exp)
  (map car (cadr let-exp)))

(define (let-values let-exp)
  (map cadr (cadr let-exp)))

(define (let-body let-exp)
  (sequence->exp (cddr let-exp)))
```

11/22



Let Syntax Transformation

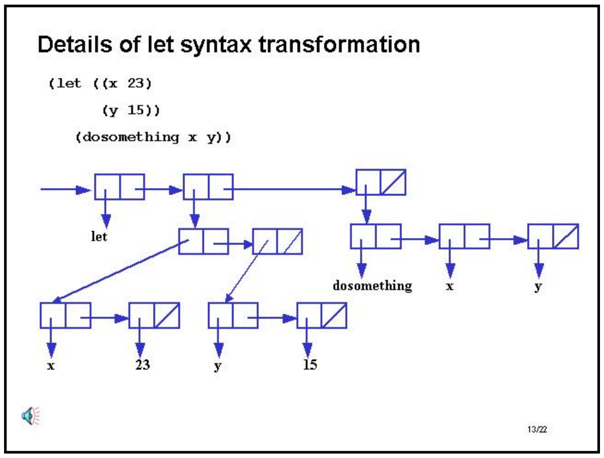
```
(define (let? exp) (tagged-list? exp 'let))

(define (let-bound-variables let-exp)
  (map car (cadr let-exp)))

(define (let-values let-exp)
  (map cadr (cadr let-exp)))

(define (let-body let-exp)
  (sequence->exp (cddr let-exp)))


(define (let->combination let-exp)
  (let ((names (let-bound-variables let-exp))
        (values (let-values let-exp))
        (body (let-body let-exp)))
    (cons (list 'lambda names body)
          values)))
```

12/22

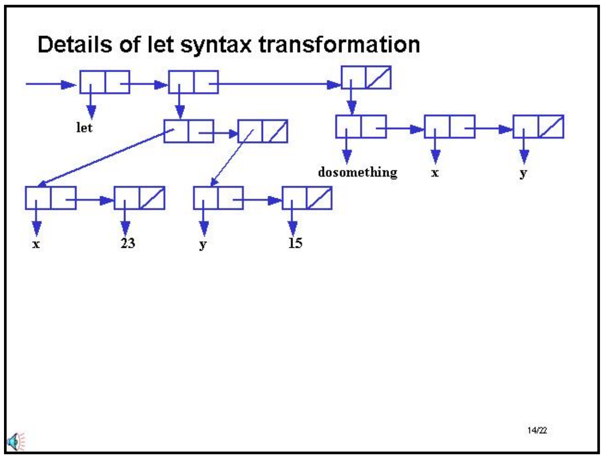
Slide 16.2.12
Now we can put all of this together. So `let-combination` is going to take in a tree structure representing a `let` expression. It will then convert this into a form that we can just evaluate.
To do this, it takes the tree structure corresponding to the variables, the tree structure corresponding to the values, and the tree structure corresponding to the body, and then it creates a **new tree structure**. Notice this structure! The first part of this list (because we are doing a `cons`) is a list that represents a procedure in our syntax. It is a list of the symbol `lambda`, followed by a list of names, followed by a body! That looks exactly like a `lambda` expression.
And then we put that at the beginning of a list of the values. So this will convert a `let` combination into a new tree structure, a tree structure that looks exactly like an application of a `lambda` to a set of values. That will then

be passed to `meval`, which will cause the evaluation of the whole expression, creating the thing we want.

Slide 16.2.13
So let's trace this through. Remember that a `let` expression such as that shown on the slide gets converted by the parser into a tree structure that looks exactly like this. The first part of the tree is the symbol `let`, the second part is a tree that captures the set of clauses, and the last part is another tree that represents the body of the `let`. This tree structure is what is passed into `meval` and let's see what happens when we do that.

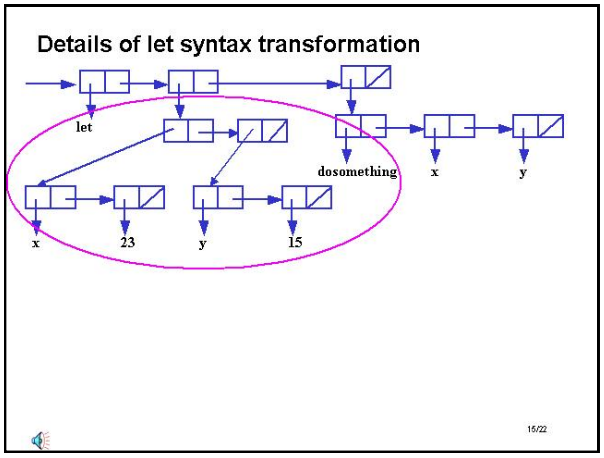

Details of let syntax transformation

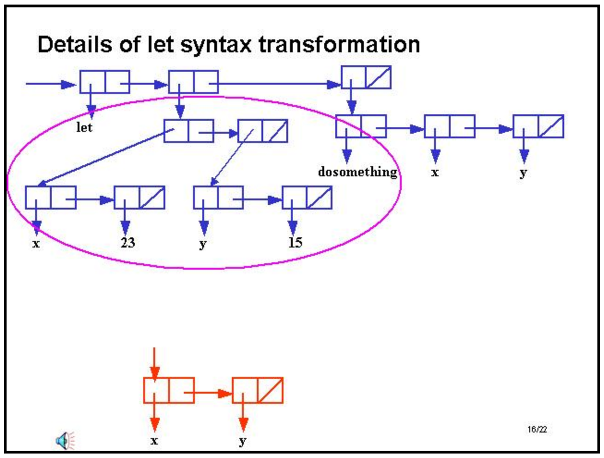
```
(let ((x 23)
      (y 15))
   (dosomething x y))
```


Details of let syntax transformation

Slide 16.2.14
What does the syntactic manipulation that converts a `let` expression into a combination do? First it wants to get the bound variables, and the code says to get the `cadr` of this tree. So it is going to grab this piece ...

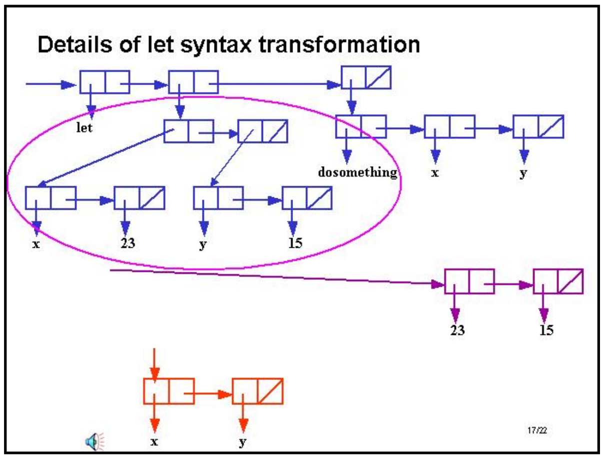
Slide 16.2.15
... and then it says to walk down this piece of list structure, using a `map`, and apply `car` to each element. That creates a new list, with the car of each element of the old list copied into that list, shown as ...

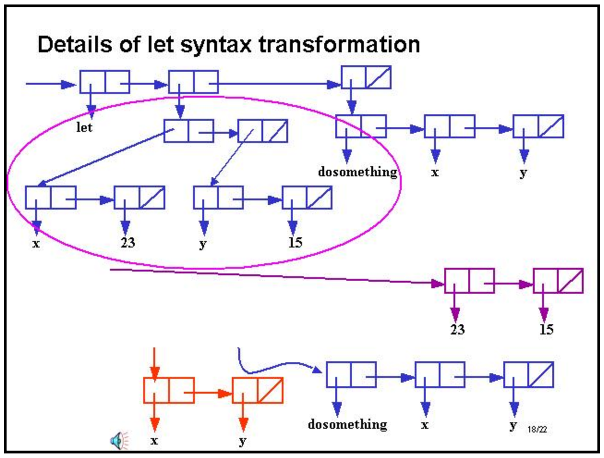

Details of let syntax transformation



Slide 16.2.16
... here. The next thing it does is take the same `cadr` structure, but now map `cadr` down it. This creates a new list structure, copying the `cadr` of each element onto it, creating this ...

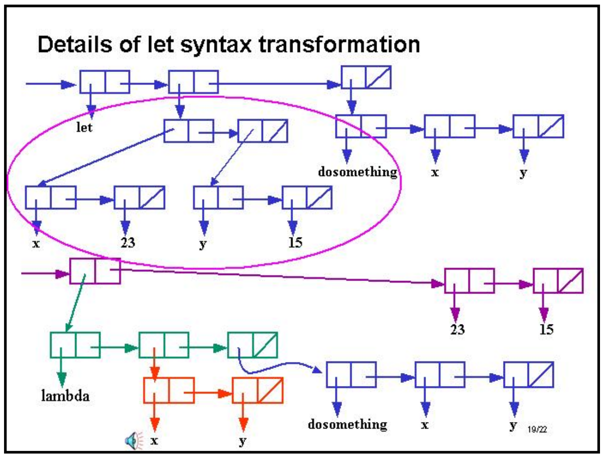
Slide 16.2.17
... structure. And then this syntactic manipulation procedure walks down the `let` tree structure and grabs the body, given us this ...



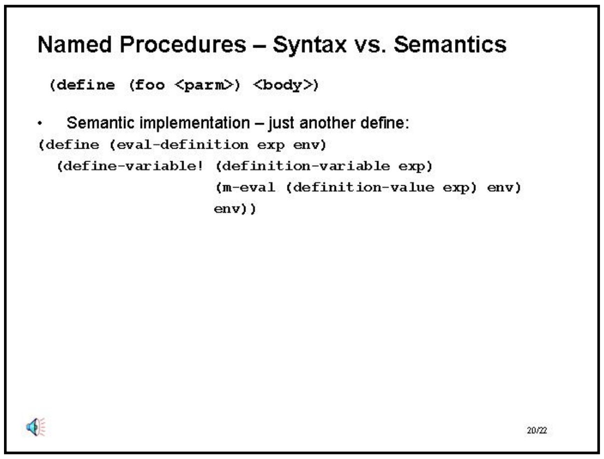


Slide 16.2.18
... structure. Note what we have done. We have created three new pieces of list structure here. We have done **NO** evaluation, we have simply manipulated pieces of the tree structure. Given those pieces, we now glue them together. The code says to create a list with the symbol `lambda`, followed by the set of names, followed by the body. Take that whole piece and put it on the front of the structure representing the values.

Slide 16.2.19
That gives us this list structure, which represents an expression, and check out its form. We have a list whose first element is a `lambda` expression, followed by a list of expressions, and we return a pointer to this whole list structure. Evaluating this tree structure, or if you like the expression that is represented by this tree structure turns into an evaluation of an application, as we wanted.
So the key point is that we can do syntactic manipulation of expressions to convert one form into another form, in this case converting a `let` into its underlying `lambda` application. Then we let our evaluator do the work to complete the



evaluation.



Named Procedures – Syntax vs. Semantics

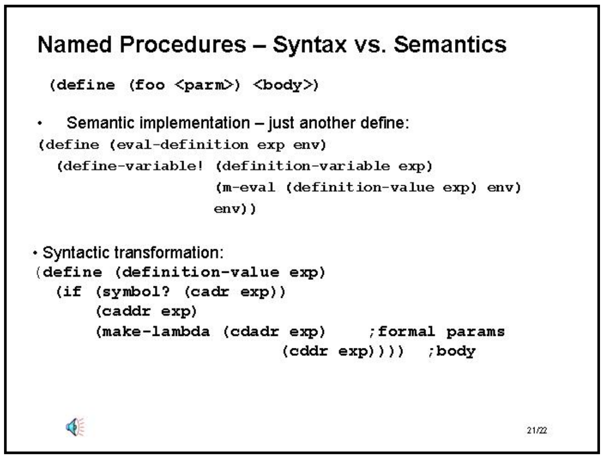
```
(define (foo <parm>) <body>)
```

- Semantic implementation – just another define:

```
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (m-eval (definition-value exp) env)
                    env))
```

20/22

Slide 16.2.20
What other syntactic variations are possible? Remember how we had named procedures, that is, forms such as that shown in the slide. Earlier, we would have had to write `(define foo (lambda ...))` to separate the `lambda` that created the procedure from the `define` that gave it a name.
We saw it was convenient, especially when thinking about the substitution model, to have this alternative form, in which we clearly identify the application of a procedure to a set of arguments. How can we add this form to our system, which so far `meval` cannot support.
In terms of semantics this is just another define. We are defining a variable to be a particular value. What else do we need?



Named Procedures – Syntax vs. Semantics

```
(define (foo <parm>) <body>)
```

- Semantic implementation – just another define:

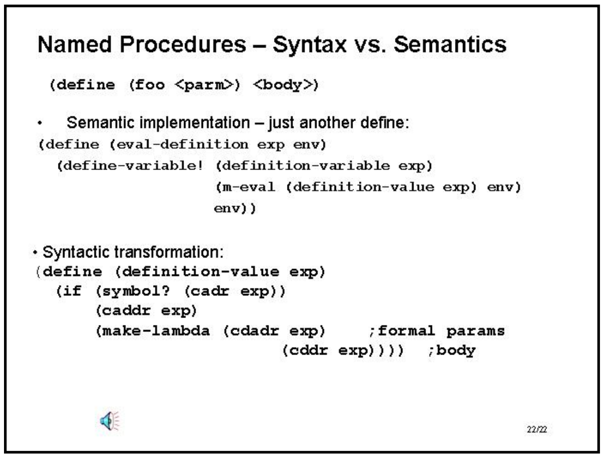
```
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (m-eval (definition-value exp) env)
                    env))
```

- Syntactic transformation:

```
(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)     ;formal params
                   (cddr exp))))  ;body
```

21/22

Slide 16.2.21
So we don't want to change the semantics. Evaluating a definition should behave exactly the same as before. All we need to do is change the syntax, that is, the thing that manipulates the tree structure, to support both kinds of definitions of procedures.
Before, getting the value out of a definition would have simply grabbed the third subexpression. But now we can be careful. We will first check to see if the second subexpression is a symbol. If it is, then we know we have the form of a define we handled before, and we will simply grab the right piece and return it. On the other hand, if it is not a symbol, then we can assume we have one of these new forms. What should we do in this case? We need to desugar or unwrap the hidden `lambda`. So we will construct a new `lambda` using a constructor for `lambda`s, passing in the formal parameters, which we get by walking down the tree structure of the expression, and the body, which we also get by walking down the tree structure of the expression. In this getting the value of the expression will convert the actual expression into a `lambda`. This is great, because that then gets passed back to `eval-definition`, which evaluates this expression to create the procedure, which is then returned to be bound to the name.
Notice how again we are simply manipulating the tree structure of an expression to convert it into a new expression.



Named Procedures – Syntax vs. Semantics

```
(define (foo <parm>) <body>)
```

- Semantic implementation – just another define:

```
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (m-eval (definition-value exp) env)
                    env))
```

- Syntactic transformation:

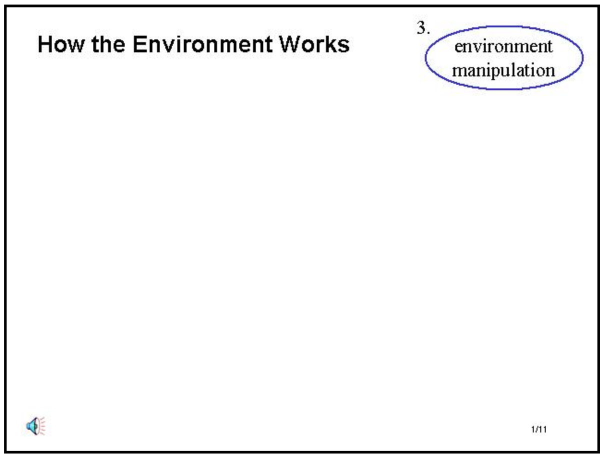
```
(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)     ;formal params
                   (cddr exp))))  ;body
```

22/22

Slide 16.2.22
In summary, we have both created `eval` and `apply` to define the semantics for our language, and we have separately defined the syntax, the legal expressions in our language. By using data abstractions between `eval` and `apply`, and the procedures that manipulate expressions, we give ourselves a clean way for changing syntax, for adding to the syntax, for manipulating syntactic variations on expressions, without ever changing `eval` or `apply`.
Thus we have seen both the semantics and the syntax of our language.

6.001 Notes: Section 16.3

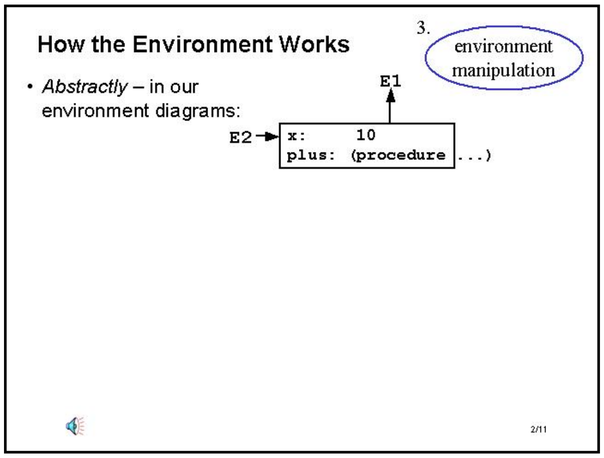
Slide 16.3.1
So you can see that we are making progress in building a complete evaluator for Scheme. We started by building `eval` and `apply`, on top of some data abstractions so that we could see the flow of evaluation: between evaluating an expression with respect to an environment; to applying a procedure to a set of arguments. We have also filled in the details of the evaluator, separating the semantics from the syntax, i.e. how we write and manipulate expressions in the language.
What else do we need to fill in? What about environments? So far, we have simply relied on there being some kind of abstract table for dealing with storing of bindings, and retrieving them. Let's make that explicit as well.



How the Environment Works

3. environment manipulation

1/11



How the Environment Works

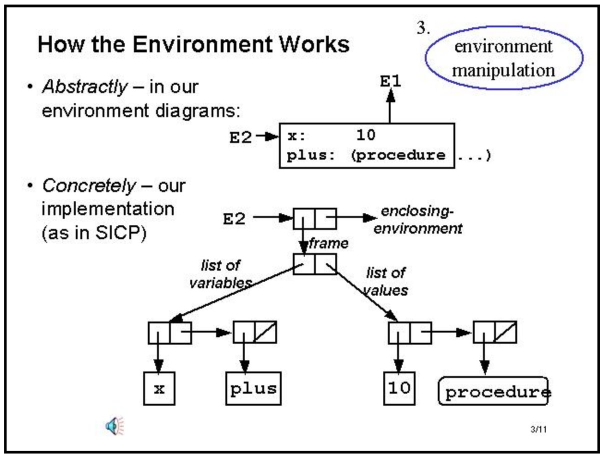
3. environment manipulation

- *Abstractly* – in our environment diagrams:

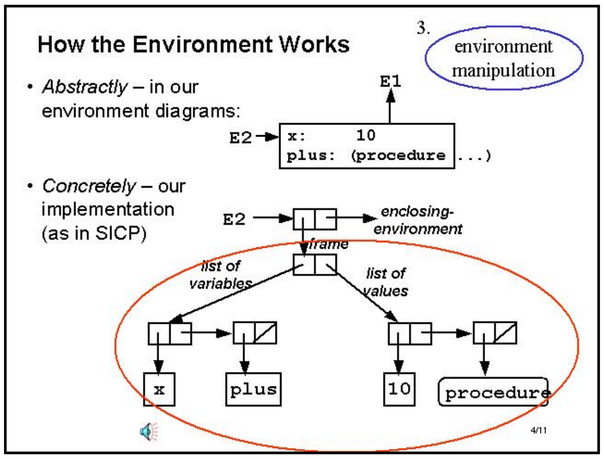
2/11

Slide 16.3.2
What do we need in an environment? Abstractly we just want to build the environment diagrams we had in our model. That is, we need a way of taking bindings (pairings of names and values) and gluing them together into tables. Those tables will need pointers to other tables, to allow for sequences of frames. Thus we need to glue things together as sequences of tables, and the tables need to glue together bindings of variables and values.

Slide 16.3.3
We can just do this using list structure! Page 3 of the code handout shows this. An environment will be created as a list of frames. Thus in this example E2 would be a list whose first element points to a frame and whose `cdr` points off to the enclosing environment. This list can continue until it terminates in the global environment, as the last element in the list.


How the Environment Works

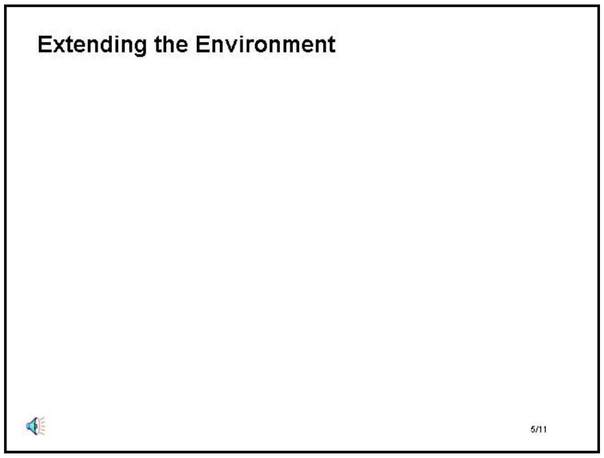
- *Abstractly* – in our environment diagrams:
- *Concretely* – our implementation (as in SICP)

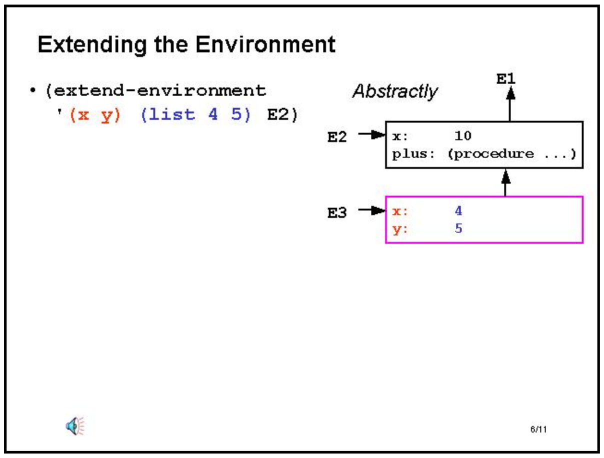

How the Environment Works

- *Abstractly* – in our environment diagrams:
- *Concretely* – our implementation (as in SICP)

Slide 16.3.4
To represent a frame, we have lots of choices. Abstractly, we need a paired collection of variables and values. We could just glue them together in that order: variable, value, variable, value. A second way however is to create two lists: The first component will be a list of the variables, and the second component will be a list of the corresponding values. The correspondence or binding is determined by the order: the first variable matches the first value, the second variable matches the second value, and so on.

Slide 16.3.5
As we have seen, the integral loop of `eval` and `apply` is to reduce the evaluation of one expression with respect to some environment to evaluation of another expression with respect to a new environment that extends the original environment. So we need to specify how to extend an environment to allow for this new evaluation.

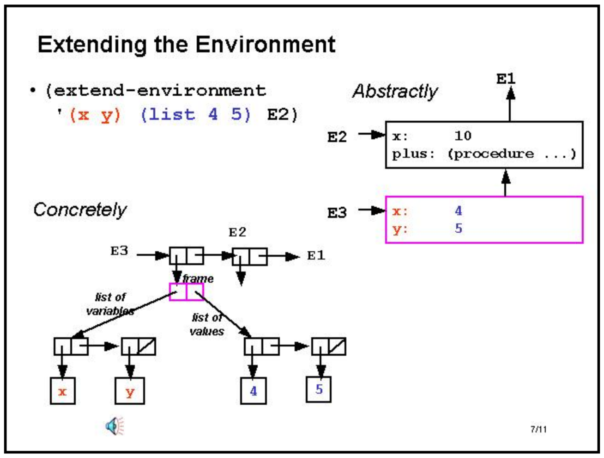

Extending the Environment


Extending the Environment

- (extend-environment
  '(x y) (list 4 5) E2)

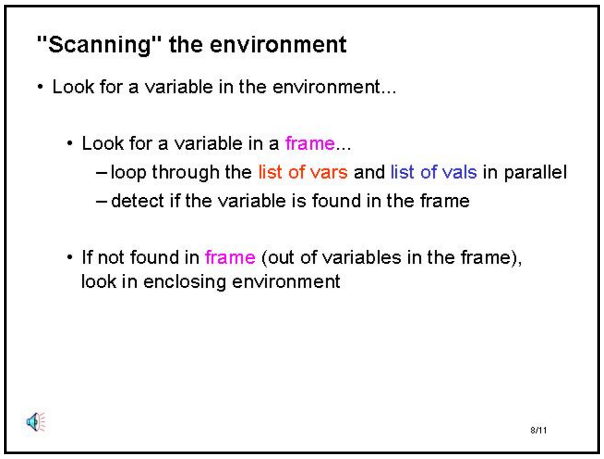
Slide 16.3.6
Abstractly, we should take a list of variable names, a list of corresponding values, and a current environment, and we should create a new frame that is enclosed or scoped by the original environment, and within which the list of parameters is bound to the corresponding list of variables. This is just our environment model.

Slide 16.3.7
Concretely, our environment was just a list of frames, so to extend an environment we simply want to add to the front of that list a new frame. Check the code, which simply "conses" a new frame onto the existing environment. And what is a frame? It just constructs one of these pairings of a list of variables and a list of values. The code is careful to ensure that number of variables and values matches.
Thus in our concrete implementation, extending an environment just creates a new frame by pairing up two lists, then puts that frame at the front of the list of frames that represents our environment. We now start evaluation with respect to the new frame at the front of the list.



Extending the Environment

- (extend-environment '(x y) (list 4 5) E2)



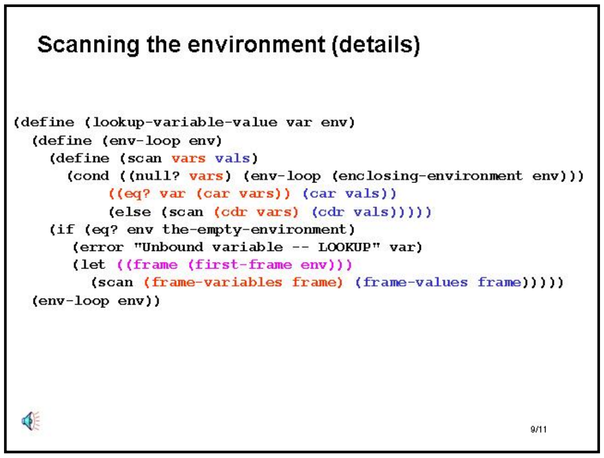
"Scanning" the environment

- Look for a variable in the environment...
  - Look for a variable in a frame...
    - loop through the list of vars and list of vals in parallel
    - detect if the variable is found in the frame
  - If not found in frame (out of variables in the frame), look in enclosing environment

8/11

Slide 16.3.8
Once we have an implementation for building environments, we can turn to using them. In particular, environments are intended to help us look up values of variables. We now need some procedures to support value lookup. So what do we need? First, we should look for a binding in a frame. Thus we will take the current frame and loop through the list of variables and the list of values in parallel (remember that we stored those as two separate lists in our implementation). Thus we just walk down these lists in synchrony. If we find the variable for which we are looking, we return the associated value. If we reach the end of that pairing of lists without finding the variable, then we know there is no binding for that variable in this frame. Thus
we move on to the next enclosing environment. And we know all of this should just be manipulation of list structure. The next slide shows the detailed code to do this for us.

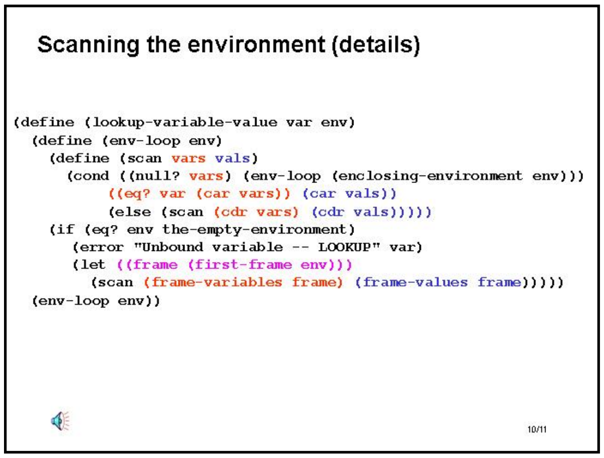
Slide 16.3.9
To implement this idea in our particular structure, we just need two different looping mechanisms. The first one, when we are going to look up a variable value, loops over environments. So notice what `env-loop` does: it takes in an environment, and if it is empty, complains about an unbound variable; otherwise it will look in the first frame of that environment for a binding, where `first-frame` is just a data abstraction for getting the first frame of the sequence. If it doesn't find a binding there, it should move on to the next frame in the sequence. Thus, `env-loop` is the top-level loop for looking up bindings in a frame.



Scanning the environment (details)

```scheme
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) (env-loop (enclosing-environment env)))
            ((eq? var (car vars)) (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
       (error "Unbound variable -- LOOKUP" var)
       (let ((frame (first-frame env)))
         (scan (frame-variables frame) (frame-values frame)))))
  (env-loop env))
```

9/11



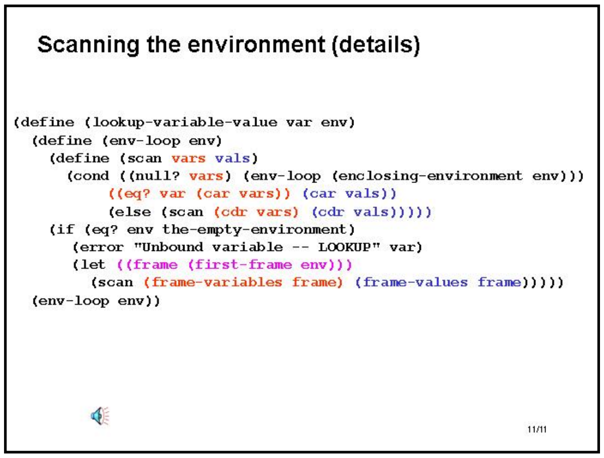
Scanning the environment (details)

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) (env-loop (enclosing-environment env)))
            ((eq? var (car vars)) (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- LOOKUP" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame) (frame-values frame)))))
  (env-loop env))
```

10/11

Slide 16.3.10
Once it gets the first frame of the environment, the method runs a second loop, `scan` on that frame's list of variables and list of corresponding values. `Scan` works its way down both lists in synchrony. If it runs out of variables, it must not have found a binding in this frame, and we go on to the next frame. Notice that it calls `env-loop` to go back up to the top level to get the next frame and rescan.
If there are variables left to check, then we compare our search variable against the next one in the list. If it is, I return the next element from the values list. If it is not, I `scan` again, moving in synchrony down both lists.
There are the two loops that let us look up the value of a variable in an environment.

Slide 16.3.11
Other aspects of manipulating variables in environments, e.g. setting a variable, will have a very similar form. In the case of setting a value, we similarly look for the right pairing, then change it (see the code for details).
Thus we can build environments just out of list structure.



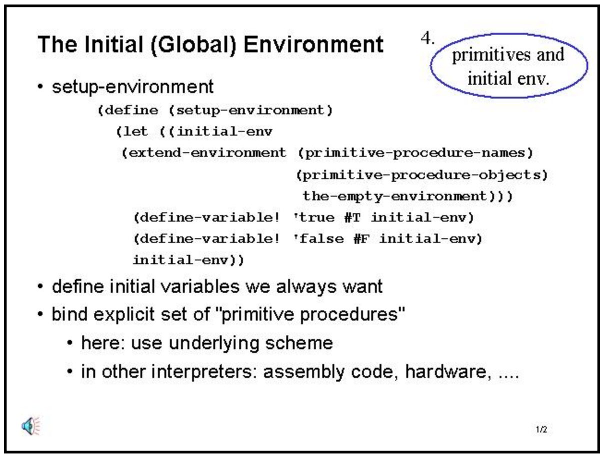
Scanning the environment (details)

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) (env-loop (enclosing-environment env)))
            ((eq? var (car vars)) (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- LOOKUP" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame) (frame-values frame)))))
  (env-loop env))
```

11/11

6.001 Notes: Section 16.4

Slide 16.4.1
To get things going, we need an initial environment, also called the **global environment**. This will be our default environment, where we will normally evaluate expressions from the user. Therefore, in this environment we need bindings for the built-in names of procedures. This will also be our backstop, meaning if we look through all the environments for a binding, and get to and through the global environment, without out finding a binding, we will stop and signal an unbound variable error. Remember that we are representing our environment just as lists of frames, so to get the initial environment, we will take an empty environment (just the empty list) and extend it, building a single frame in which we do the following. We take a set of possible names for built-in primitives and a set of procedures to go with those names, and we will install them into the environment. Look at the code handout on page 4 to see how we give our own names to the built-in Scheme primitives. Remember that the names are our choice: here we have used the same ones as those in Scheme but we could have made other selections. Also notice that the set of things we choose to have as primitives is also our



The Initial (Global) Environment

4. primitives and initial env.

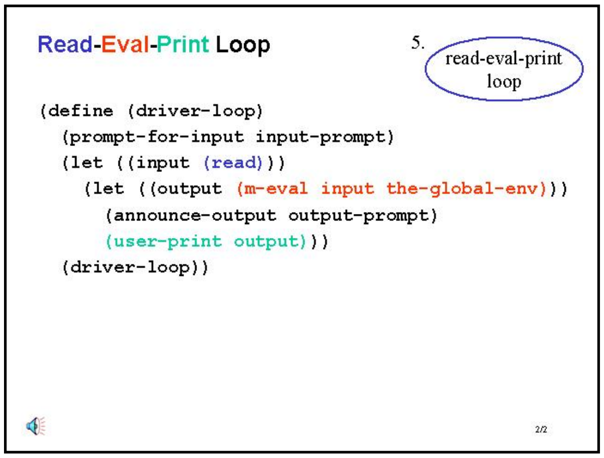
- setup-environment

```
(define (setup-environment)
  (let ((initial-env
   (extend-environment (primitive-procedure-names)
                       (primitive-procedure-objects)
                        the-empty-environment)))
    (define-variable! 'true #T initial-env)
    (define-variable! 'false #F initial-env)
    initial-env))
```

- define initial variables we always want
- bind explicit set of "primitive procedures"
  - here: use underlying scheme
  - in other interpreters: assembly code, hardware, ...

1/2

choice. In addition to creating bindings for our primitive procedures, we also create bindings for `true` and `false` to the Boolean values used by the machine to represent these values. Finally we return this environment as our initial environment.
For details, check out the code handout.



Read-Eval-Print Loop

5.
read-eval-print loop

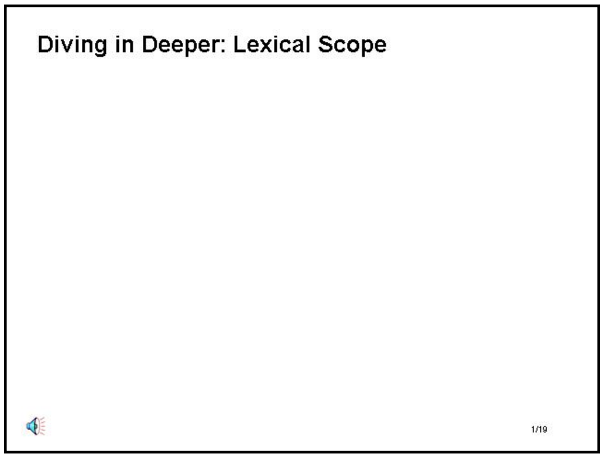
```
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (m-eval input the-global-env)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))
```

2/2

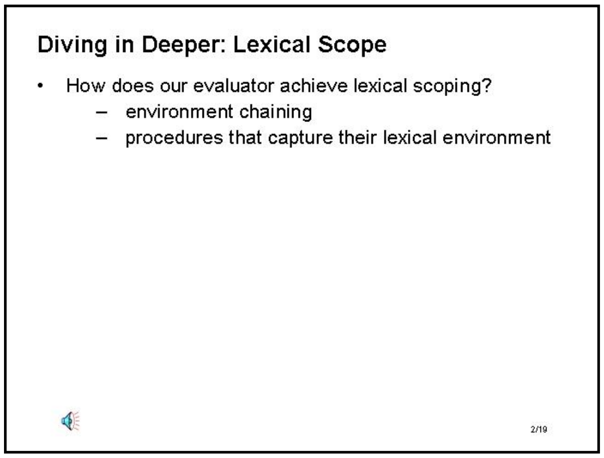
Slide 16.4.2
To interact with our evaluator, we need some mechanisms for getting expressions into it. For this, we build a simple little driver loop, which is often known as an **REP** or read-eval-print loop. This loop runs through a constant cycle of reading in an expression, evaluating it using the interpreter, then printing out the result and cycling to read the next expression. Note how the code on the slide runs through exactly that cycle.
It prints a prompt to the user on the screen, then reads in an expression using Scheme's **read** operator. It then passes that parsed tree structure to our evaluator for interpretation, with respect to our global environment. When done, it prompts the user with a note that the value follows, and prints out the value.
It then cycles through this process again.
With this piece we have completed our construction of an evaluator. We have `eval` and `apply` to unwrap our abstractions to primitive operations, we have a definition of the syntax of the language which is abstracted away from the interpreter to allow ease of modification, we have details of how to create and use environments to hold our values for variables, and we have a simple interface to the interpreter.

6.001 Notes: Section 16.5

Slide 16.5.1
So there you have it! We have built an evaluator for Scheme, implementing `eval` and `apply`. Although there is a lot of code here, you should try to step back from the details to see the general outline of this evaluator.
Having built our evaluator, we can ask some questions about choices that were made in designing this evaluator. In particular, we said much earlier in the term that Scheme uses **lexical scoping**. So what does that mean?



Diving in Deeper: Lexical Scope



Diving in Deeper: Lexical Scope

- How does our evaluator achieve lexical scoping?
  - environment chaining
  - procedures that capture their lexical environment

2/19

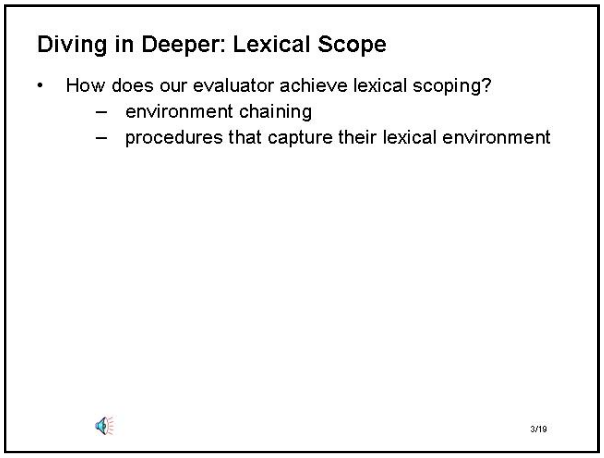
Slide 16.5.2

Think about what happens when we apply a procedure to a set of arguments. In detail, we get the values of the arguments and the value of the procedure, then we create a new environment in which the variables of the procedure are bound to the arguments passed in, and relative to that environment we evaluate the body of the procedure. So what happens inside that body? In particular, that body will be an expression that contains lots of names, and the issue is: how do we find the values associated with those names?

First, any name that was a formal parameter of the procedure gets its value from the frame we just created. Those are called **bound parameters**. But any variable in that body that is not one of the formal parameters is known as a **free variable**. How do we find the bindings for those? We have said that if we have a procedure that has free variables, the values for them come from bindings made by enclosing procedure definitions. In other words, they are looked up in the environment in which the procedure was defined.

Slide 16.5.3

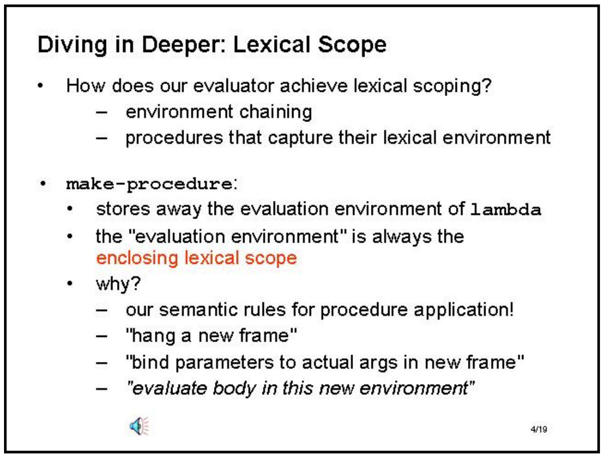
Or said another way, it says we walk up that chain of environments looking for a binding of the variable. And we know that chain of environments comes from sets of procedures. So if our procedure is defined inside another procedure we know that if we can't find a scoping (a formal parameter) binding the thing we are looking for in the initial `lambda` we go outside the scope of that `lambda` to the enclosing `lambda`, looking for a corresponding formal parameter here. Thus, the boundaries of the `lambda` expressions define the chain of frames we are going to see in our environment model, and that is how we are going to capture our lexical scoping to determine the values of variables. How does this happen in practice?



Diving in Deeper: Lexical Scope

- How does our evaluator achieve lexical scoping?
  - environment chaining
  - procedures that capture their lexical environment

3/19



Diving in Deeper: Lexical Scope

- How does our evaluator achieve lexical scoping?
  - environment chaining
  - procedures that capture their lexical environment
- `make-procedure`:
  - stores away the evaluation environment of `lambda`
  - the "evaluation environment" is always the enclosing lexical scope
  - why?
    - our semantic rules for procedure application!
    - "hang a new frame"
    - "bind parameters to actual args in new frame"
    - *"evaluate body in this new environment"*

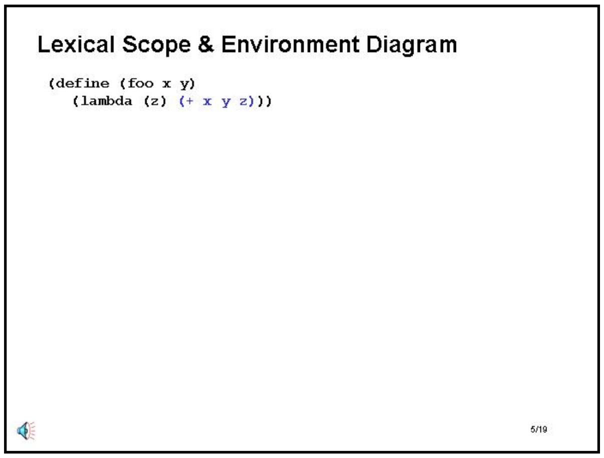
4/19

Slide 16.5.4

Look at the code for `make-procedure`, which you will find on page 3 of the handout. It glues together a tag, the parameters, the body, and the environment in which it is evaluated. As a consequence, the **evaluation environment** is stored away and is always going to be the enclosing lexical scope. It is going to tell use where to look to find a binding for a free variable. Why?

Well that was just a design choice! This came from our design of procedure application, which said: hang a frame, bind the parameters to the actual values passed in, then evaluate the body in this new environment. The choice here was to scope together frames based on where the procedures were actually evaluated.

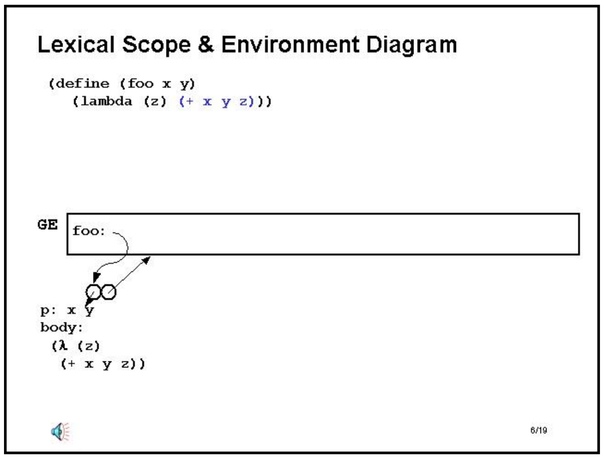
Slide 16.5.5
So let's remind ourselves of how that works. This is just reviewing the environment model, but with our detailed implementation now in mind. Suppose we consider the definition shown here. In the environment model, we know what happens ...



Lexical Scope & Environment Diagram

```
(define (foo x y)
  (lambda (z) (+ x y z)))
```

5/19



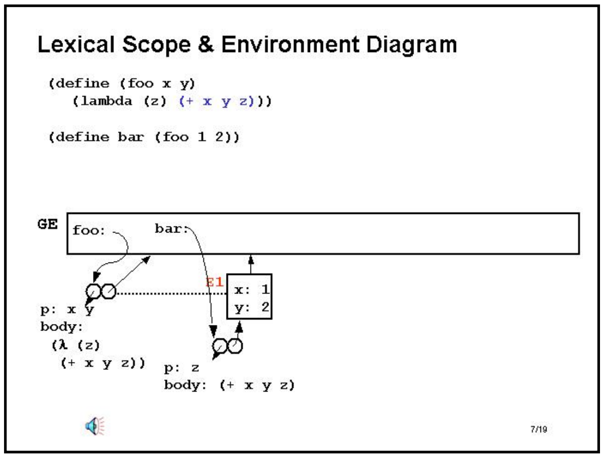
Lexical Scope & Environment Diagram

```
(define (foo x y)
  (lambda (z) (+ x y z)))
```

6/19

Slide 16.5.6
Evaluation of this expression will create a binding for `foo` in the global environment to a procedure of arguments `x` and `y` coming from that hidden `lambda` (due to the desugaring of the syntactic sugar). Note that the body of this procedure is another `lambda` expression that has not yet been evaluated, but just exists as tree structure.

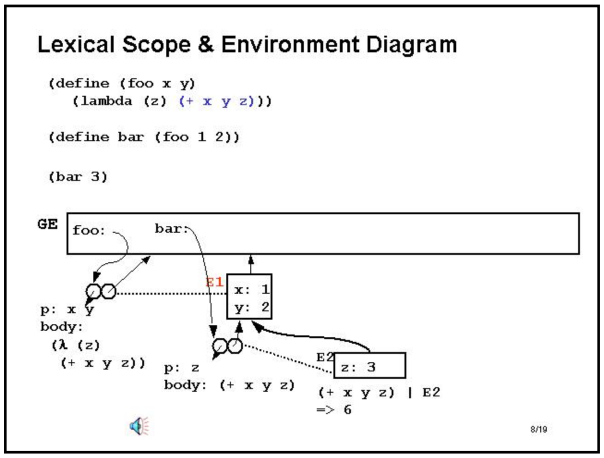
Slide 16.5.7
Let's apply `foo` to a couple of arguments, and give the result the name `bar`. `Foo` then hangs a new frame scoped by the same environment as the procedure, binds `x` and `y` to the values 1 and 2, and relative to this frame we evaluate the body. This is another `lambda` so we create the double bubble scoped by **this** frame since that is where the evaluation takes places. `Bar` is bound to this value, i.e. this procedure object.



Lexical Scope & Environment Diagram

```
(define (foo x y)
  (lambda (z) (+ x y z)))

(define bar (foo 1 2))
```

7/19



Lexical Scope & Environment Diagram

```
(define (foo x y)
  (lambda (z) (+ x y z)))

(define bar (foo 1 2))

(bar 3)
```

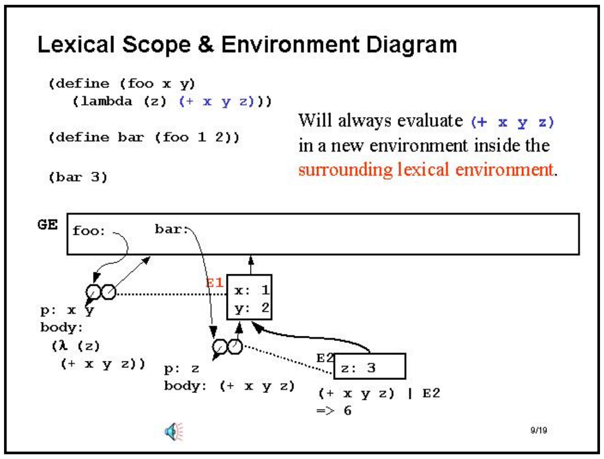
8/19

Slide 16.5.8
Now let's apply `bar`. We can trace through our evaluator, or alternatively our environment model, to drop a frame whose enclosing environment is identical to the procedure's, within which we bind the formal parameter to its argument, and relative to which we evaluate the body of this procedure. Notice that we are getting the values of `x, y` and `z` with respect to E2. Thus the binding for `z` will come from this frame as it is scoped by the `lambda` but to get the bindings for `x` and `y` we move up the chain to the next frame which comes from the enclosing `lambda`. To get the value of `+` we move further up the chain to the global environment.

Slide 16.5.9

As a consequence of this, we can see that we will always evaluate the expression `(+ x y z)` (the body of `bar`) in a new environment within the lexical scope established by the enclosing procedure. Every time we apply this procedure, we will hang a new frame that will always scope back to E1, which is the surrounding lexical environment, meaning we will always get the same bindings for `x` and `y`.

This was a particular choice we made in constructing our evaluator, but it is not the only such choice.

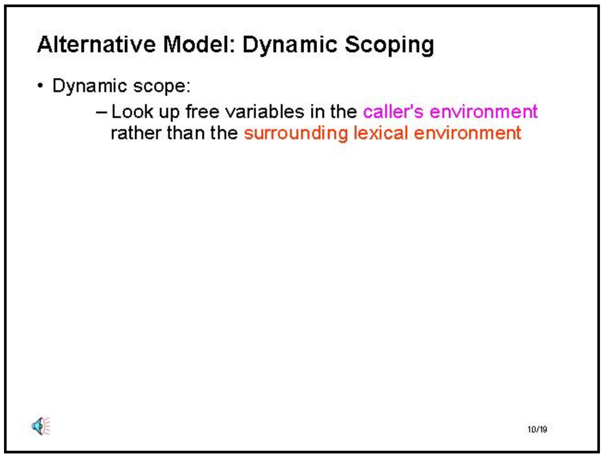


Lexical Scope & Environment Diagram

```
(define (foo x y)
  (lambda (z) (+ x y z)))

(define bar (foo 1 2))

(bar 3)
```

Will always evaluate (+ x y z) in a new environment inside the surrounding lexical environment.

9/19



Alternative Model: Dynamic Scoping

- Dynamic scope:
  - Look up free variables in the caller's environment rather than the surrounding lexical environment

10/19

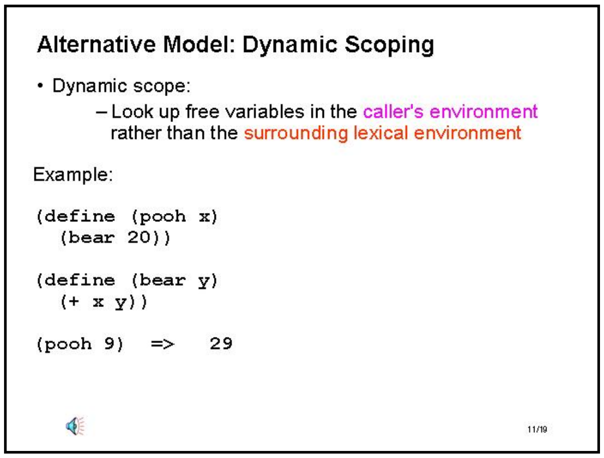
Slide 16.5.10

An alternative way to get bindings for free variables would be to look them up in the **caller's** environment, meaning in the environment that corresponds to the procedure asking for the values, rather than in the surrounding lexical environment. This will lead to a different behavior, and this style of handling free variables is known as **dynamic scoping**, because it is based on the values in place when the caller asks for them.

Slide 16.5.11

This model leads to a different behavior. Consider this definition for `pooh`. Notice that there is no explicit reference to the parameter `x` within this body. We can define `bear` as shown. Notice that `x` is used in the body of `bear` but there is no explicit parameter for `x` in the definition. Thus, `bear` needs to get the value from somewhere else.

If we call `(pooh 9)` it will call `(bear 20)` and given that the value of `x` was specified to be 9 in the call to `pooh` we could get out the value shown.



Alternative Model: Dynamic Scoping

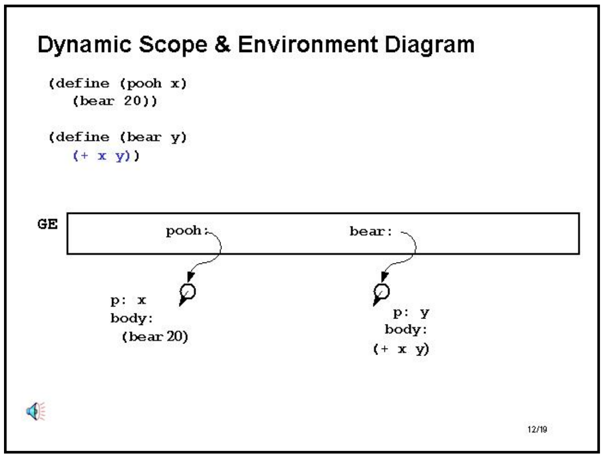
- Dynamic scope:
  - Look up free variables in the caller's environment rather than the surrounding lexical environment

Example:

```
(define (pooh x)
  (bear 20))

(define (bear y)
  (+ x y))

(pooh 9)   =>   29
```

11/19

This looks strange, as this certainly would **not** work under lexical scoping, or our normal Scheme. We are now changing the behavior of our evaluator, we are asking `pooh` to set a binding for its parameter `x` so that when `bear` is used, it can use that value of `x` created in the caller's environment.

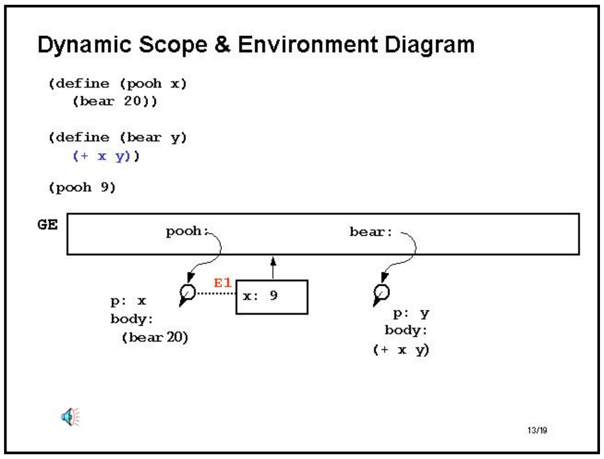


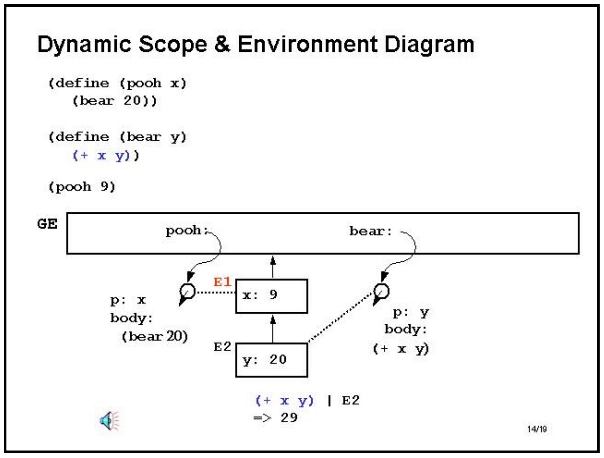
Slide 16.5.12

Suppose we want to change our evaluator to use **dynamic** scoping rather than **lexical** scoping. Conceptually, how would this change our environment model? First, note that our double bubble becomes a single bubble. Under dynamic binding we don't need to keep track of the enclosing environment in which a procedure was created, as it is no longer used to define the scoping for free variables.

Slide 16.5.13

Now, let's apply `pooh` to some argument. Applying `pooh` under this **new** model says to drop a frame in which we bind the formal parameter `x` to the value `9`. That frame **now** gets scoped by the **caller's** environment, which in this case happens to be the global environment, since that is where we are asking for the evaluation. So far this looks like before. Now, relative to that frame we are going to evaluate the body of `pooh`.

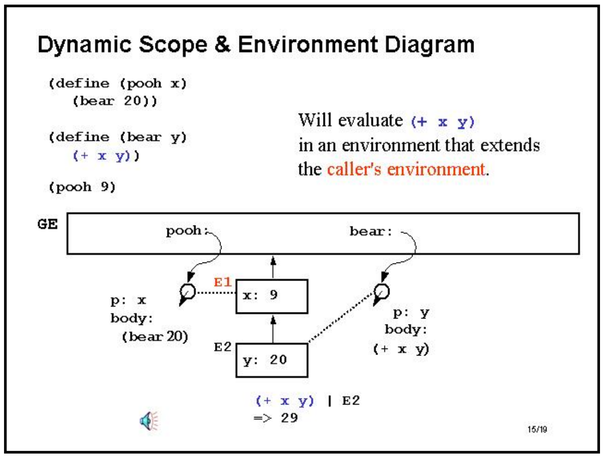




Slide 16.5.14

And the body of `pooh` says to apply `bear`. Thus, we apply a procedure so we build a frame in which `y` is bound to `20`. But now, the scoping environment for this frame is the environment in place when the caller asked, i.e. E1. It does not inherit the environment that was there when `bear` was created but rather it inherits the environment that was used when the value of applying `bear` was asked for. Now we can evaluate the body `(+ x y)` in this frame, and thus it will inherit the value of `x` that was passed in back when we evaluated `pooh`.

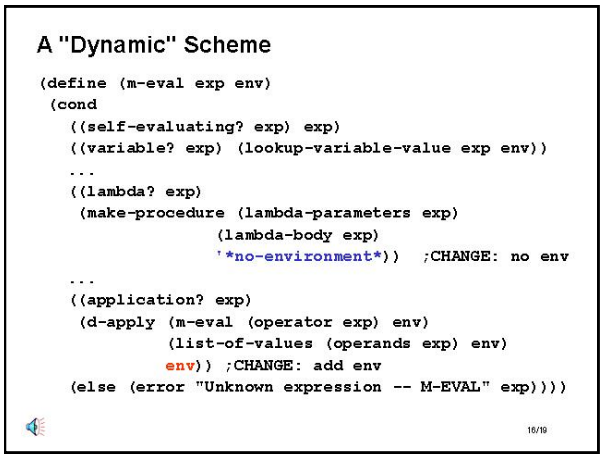
Slide 16.5.15
The key thing to note is that we will evaluate the expression `(+ x y)` in an environment that extends the caller's environment. This means that as we call this at different times, we will have different environments, as opposed to the lexical case in which we would always point back to the environment created when the procedure was created.
So we can see we can get a very different behavior when we shift to dynamic scoping.



Dynamic Scope & Environment Diagram

```
(define (pooh x)
  (bear 20))

(define (bear y)
  (+ x y))

(pooh 9)
```

Will evaluate `(+ x y)` in an environment that extends the caller's environment.

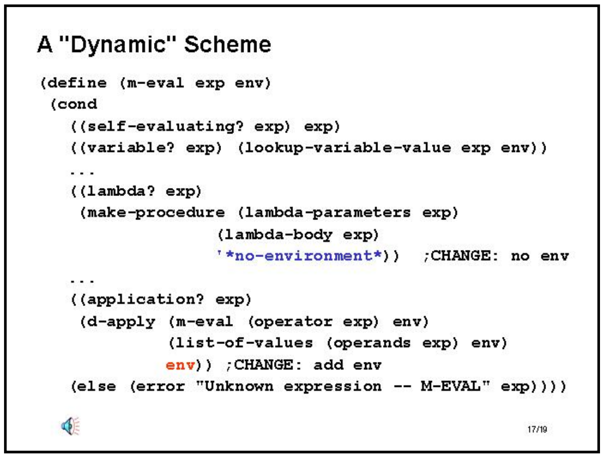


A "Dynamic" Scheme

```
(define (m-eval exp env)
 (cond
   ((self-evaluating? exp) exp)
   ((variable? exp) (lookup-variable-value exp env))
   ...
   ((lambda? exp)
    (make-procedure (lambda-parameters exp)
                    (lambda-body exp)
                    '*no-environment*))  ;CHANGE: no env
   ...
   ((application? exp)
    (d-apply (m-eval (operator exp) env)
             (list-of-values (operands exp) env)
             env)) ;CHANGE: add env
   (else (error "Unknown expression -- M-EVAL" exp))))
```

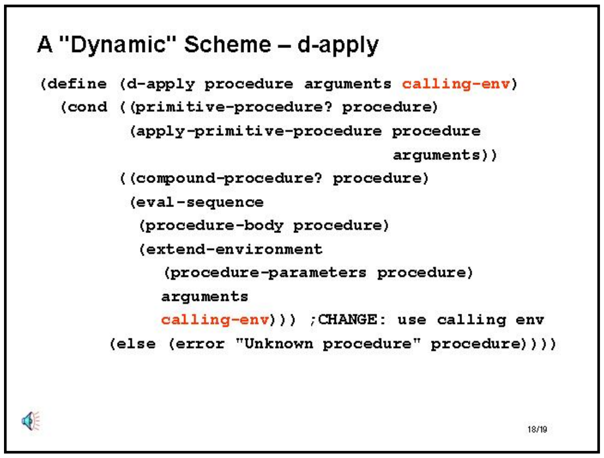
Slide 16.5.16
We will return to the issues of why one might want to have dynamic binding rather than lexical binding. Here, we want to focus on how **simple** it is to make this change in our evaluator. In fact, only a very small number of changes can cause this drastic change in behavior.
First, when we evaluate a `lambda` we will make a procedure as before, but now we don't need to include the enclosing environment. We don't make a double bubble, we make a single bubble. Thus we can use the same form, but just pass in a symbol indicating that no environment is stored.

Slide 16.5.17
The second change occurs when we go about actually applying a procedure (something we built with a `lambda`) to a set of arguments. Remember that in the previous case, we got the value of the operator, we got the values of the operands, and then we apply the operator, which meant evaluating the body of the operator in a new environment that extended the environment part of the procedure with a frame binding the parameters to the values. Here, we change that. Now our application says: get the value of the operator, get the list of values of the operands, and then apply that operator in the **current** environment. Thus we have added one more argument to our `apply`.



A "Dynamic" Scheme

```
(define (m-eval exp env)
 (cond
   ((self-evaluating? exp) exp)
   ((variable? exp) (lookup-variable-value exp env))
   ...
   ((lambda? exp)
    (make-procedure (lambda-parameters exp)
                    (lambda-body exp)
                    '*no-environment*))  ;CHANGE: no env
   ...
   ((application? exp)
    (d-apply (m-eval (operator exp) env)
             (list-of-values (operands exp) env)
             env)) ;CHANGE: add env
   (else (error "Unknown expression -- M-EVAL" exp))))
```



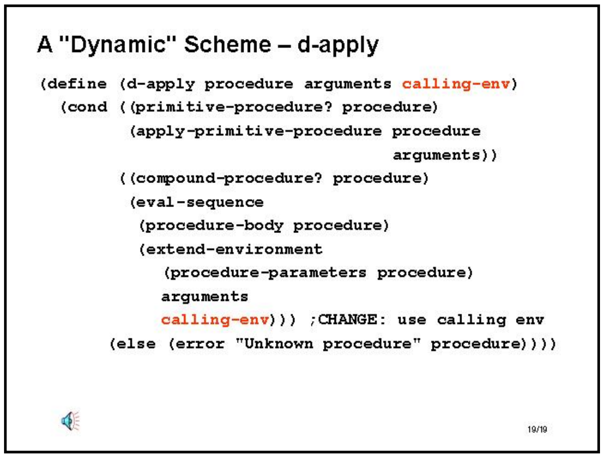
A "Dynamic" Scheme – d-apply

```
(define (d-apply procedure arguments calling-env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure
                                    arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
            (procedure-parameters procedure)
            arguments
            calling-env))) ;CHANGE: use calling env
        (else (error "Unknown procedure" procedure))))
```

18/19

Slide 16.5.18
The only other change we need is a dynamic `apply`. This `apply` takes in a procedure, a list of values, and an environment, the environment that the caller is in.
For primitives, this `apply` does the same thing. However, application of a compound procedure (something we built with a `lambda`) will evaluate the body (which we expect to be a sequence) in an environment, but **here** the environment comes from extending the calling environment.
The key thing to note is how an incredibly small set of changes to `eval` and `apply` can dramatically change the way in which variables are interpreted.

Slide 16.5.19
This was really the whole point of this example of dynamic scoping. It leads to a different kind of behavior and it is worth thinking about what kinds of advantages one might get from such a change. But what we have really done is show how by making a few small changes to `eval` and `apply` we have changed the semantics of the language. That is exactly the point of having `eval` and `apply`. They define what it means to evaluate expressions in a language.
The second point is that by cleanly separating the syntax of a language from the semantics, we have enabled making that kind of change in an easy manner. The data abstractions that define the syntax separate them from the rules for interpretation. Thus we can change the evaluation rules, but not have to make any changes to the syntax.
In the next lecture, we will return to the idea of changing the rules of evaluation, and examining the impact of those changes on the behavior of the language.



A "Dynamic" Scheme – d-apply

```
(define (d-apply procedure arguments calling-env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure
                                    arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
            (procedure-parameters procedure)
            arguments
            calling-env))) ;CHANGE: use calling env
        (else (error "Unknown procedure" procedure))))
```

19/19