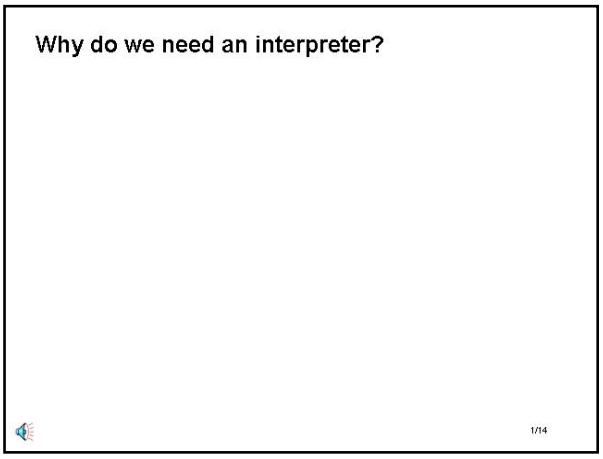
6.001 Notes: Section 15.1

Slide 15.1.1



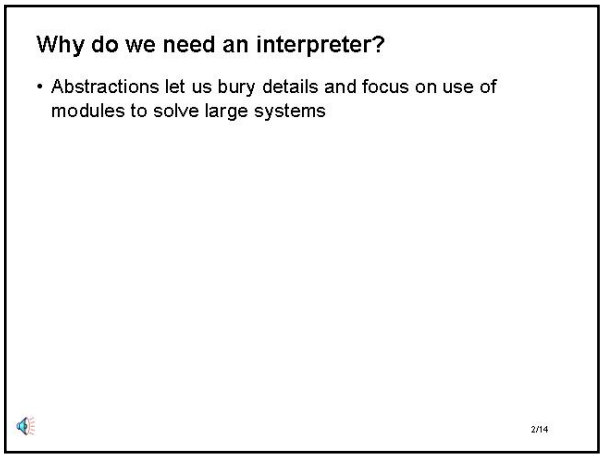
Our goal over the next few lectures is to build an **interpreter**, which in a very basic sense is the ultimate in programming, since doing so will allow us **to define our language**.

This is a somewhat surprising statement. But, in fact, as we will see through these lectures, it really is correct. The reason it is correct is the following:

Every expression we write in our language has a meaning associated with it. Deducing the meaning associated with an expression is the **process** of evaluation. And therefore, if we can define the program that executes that deduction for us, our definition of that program (i.e. our definition of the interpreter) provides for us the exact specification of what's legal in our language and how to deduce meanings of expressions within our language.

Our goal is to work through this understanding in stages. We will explore a series of examples, initially some very simple examples, but culminating in a full-scale Scheme evaluator. We are going work our way up to this, letting you see how to add the different pieces into an interpreter.

Before you proceed, however, there is a code handout that goes with this lecture. I suggest that you stop, go back to the web page, and print out a copy of the code to have next to you as we go through the lecture.

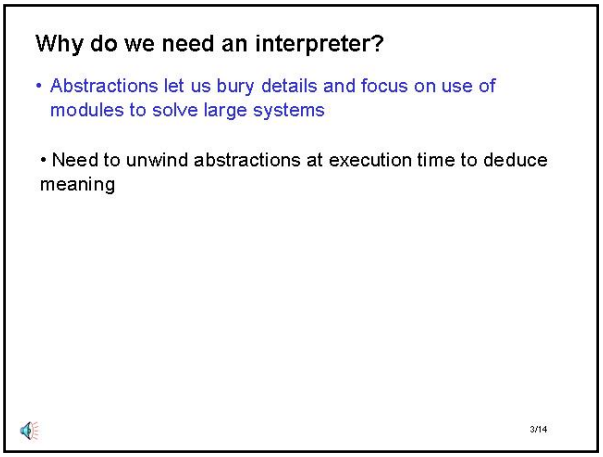


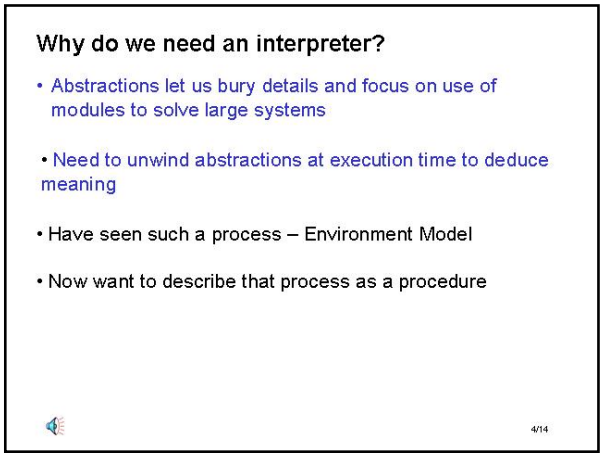
Slide 15.1.2

First, let's set the stage for why we want to do this, why do we want to build an interpreter? Why do we need an interpreter? Think about what we have seen so far in this course. We have spent a lot of time talking about, and using, the ideas of abstraction, both procedural and data. We know that this is a powerful idea, as it lets us bury details, and it certainly supports the expression of ideas at an appropriate level. Said another way, we can think about what we want to **do** and separate that from the **details** of how to actually make it happen.

Slide 15.1.3

But eventually, we need a process for unwinding all of those abstractions, to get the value that corresponds to an expression's meaning. That means we need to implement **semantics** of interpreting expressions.





Slide 15.1.4

Notice the words I used. I said, "we need a **process** for unwinding those abstractions". If we can have such a process, then we should be able to describe it in a procedure: that is our language for describing processes.

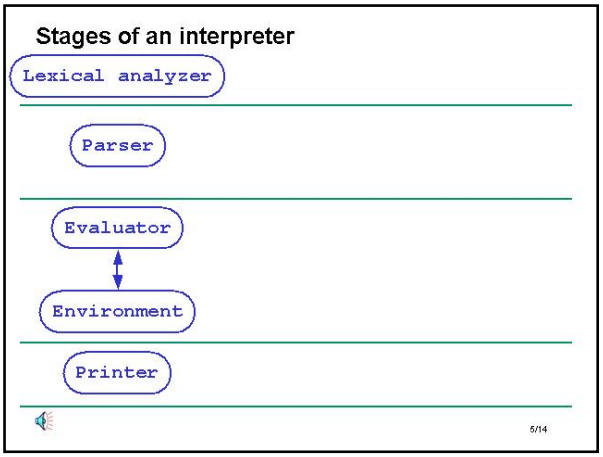
In fact, you have already seen a version of this description, just not as a procedure. What was the description? ... the environment model!

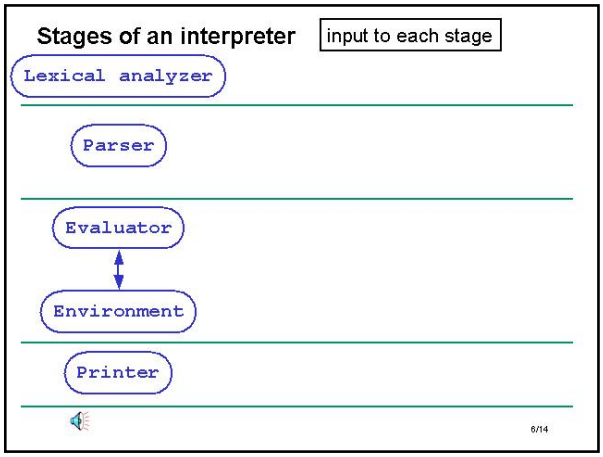
If you think about it, that makes sense. The environment model just described the process for how to determine a meaning associated with an expression, which in turn just unwrapped the abstractions down to the primitive operations. Today, we want to talk about how to actually build an evaluator as a procedure rather than as that abstract environment model.

Slide 15.1.5

First, what are the stages of an interpreter? For the kind of languages we are considering, Scheme or Lisp like languages, typically there are five stages. There is a **lexical analyzer**, a **parser**, an **evaluator** that typically works hand-in-hand with an **environment**, and there is a **printer**.

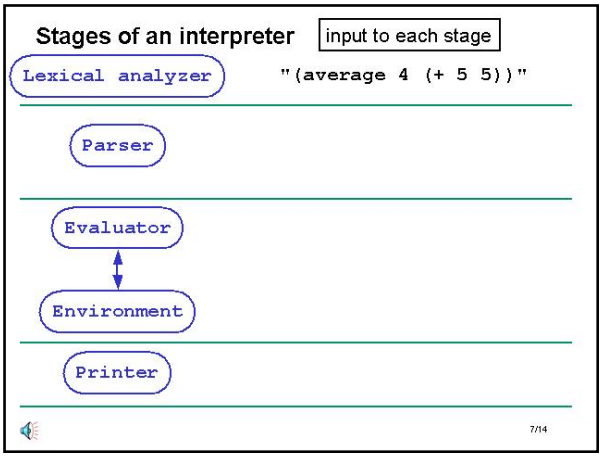
Let's look at what goes on in each of these at a high level, before we build them.

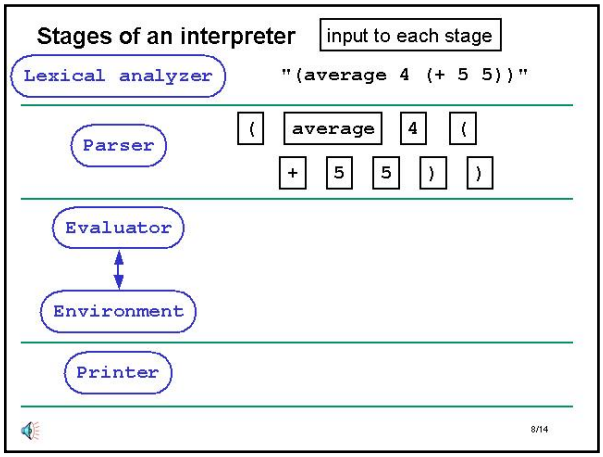




Slide 15.1.6
To do that, let's talk about the input and output characteristics of each of them.
By focusing on the input to each successive stage in the interpretation, we can get a sense of what should happen at each stage, and thus get a sense of how to build an interpreter.

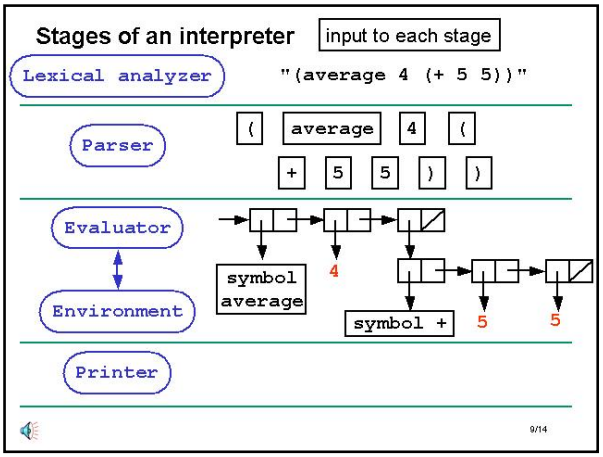
Slide 15.1.7
The initial input is a string of characters, which represents the typewritten version of the expression we want to evaluate. This is exactly the thing that we would type in at a terminal if we wanted to have an expression evaluated. So the initial input is a string of characters.

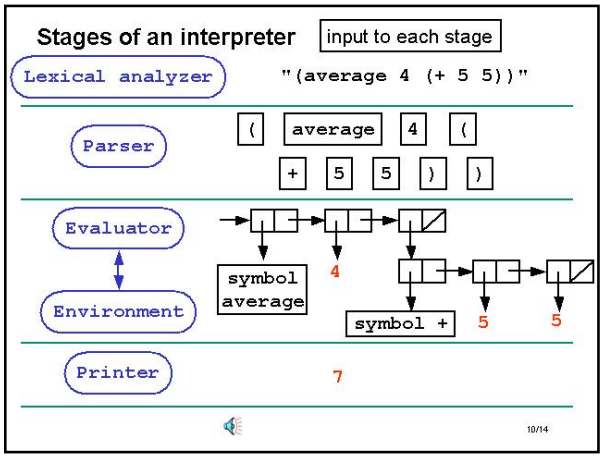




Slide 15.1.8
The first step is to use a **lexical analyzer** to convert that string of characters into units or words. This is shown here, where the string gets converted into a set of words or isolated characters like "(" and ")" and "+" and numbers. Thus the input to the next stage is an ordered sequence of these units or words.

Slide 15.1.9
The second stage then **parses** those words into a structure that we can use for evaluation. In particular, we convert the linear sequence of words into a tree structure. We are using pairs here for convenience but that is not required. We could use any other representation of trees as well.
As we do this, we are going convert the self-evaluating expressions into their internal values. So notice the form we get for the next stage: it's a tree structure, and hanging off of the leaves of the tree are numbers, symbols, or other objects that are represented as basic words. This is the input to the next stage.



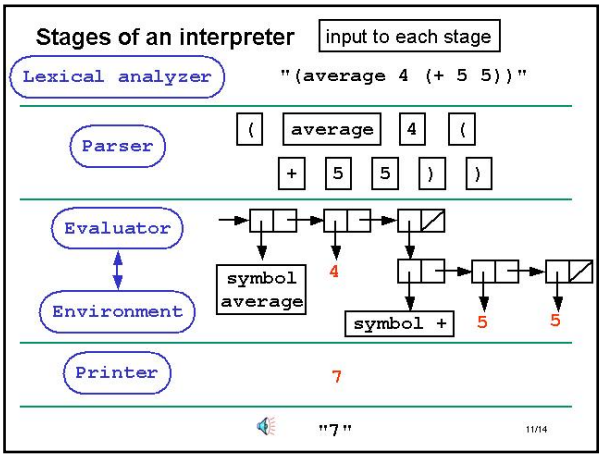


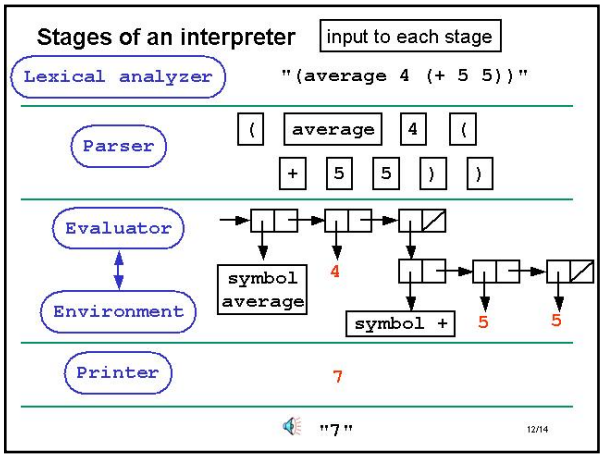
Slide 15.1.10

Now comes the heart of the interpreter. We want to take that tree structure and deduce its value. First, notice the form of tree structure. We will talk about this in detail later, but you can already see how the parser has converted things into a tree. Every time we see an "(", indicating the beginning of a new combination, we have created a new list structure. If the parser is already inside a list structure, it drops down a level, so that we build up a tree where each horizontal slice through the tree corresponds to some combination. Now what is the evaluator supposed to do? It wants to take that tree structure, plus an environment, and interpret it. And what does that mean? Think of the environment as a way of associating names with more primitive values. It acts much like a dictionary. The evaluator will use a set of rules to walk through this tree, looking up values associated with symbols from the environment, i.e. from that dictionary, and then using the rules to reduce complex expressions to simpler things, culminating in some simple value that we will return.

Slide 15.1.11

That value becomes input to the final stage. The printer simply converts things back into the appropriate form for display on the monitor, and then ...



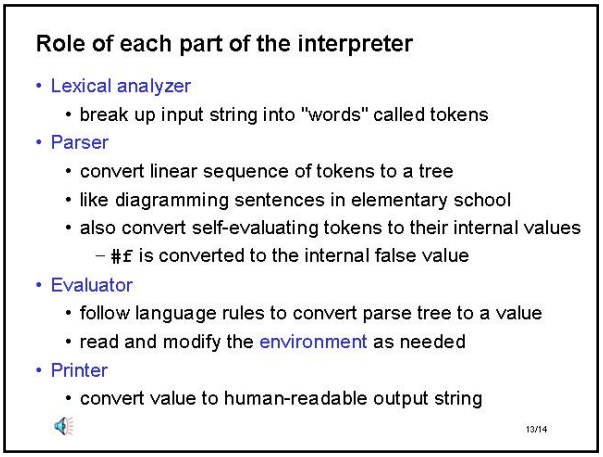


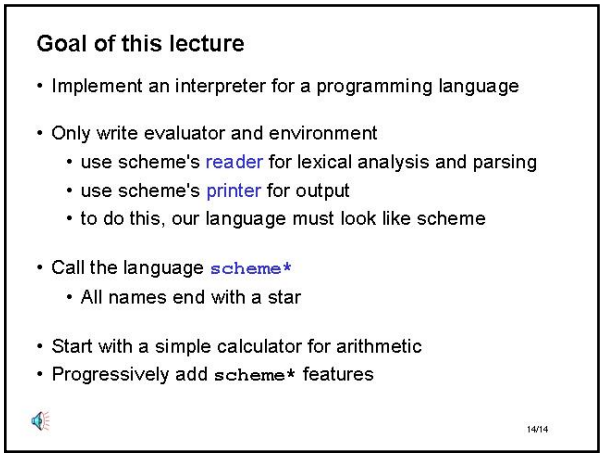
Slide 15.1.12

... that just gets displayed to the user.

Slide 15.1.13

So here is a summary of that process in words.





Slide 15.1.14

Our goal is to implement an interpreter. Actually, that's not quite right. Our goal is for you to **understand** what goes into an interpreter, which we will explore by implementing one. Since the key part of an interpreter, the crucial part, is the evaluator, we are going to concentrate almost exclusively on that. We are going to use Scheme for all the rest of the pieces, that is, we will use Scheme's lexical analyzer and parser, and Scheme's printer, rather than building them from scratch. This means of course that we will need to create an evaluator for a language that looks a lot like Scheme in the sense of having a tree structure as the output of its parser and a set of rules for manipulating that tree structure, as the way of dealing with the actual evaluation. It also says that the syntax of the language we are going to use to demonstrate an interpreter will need to look at lot like Scheme, in terms of things like using parentheses to delimit expressions and other related issues.

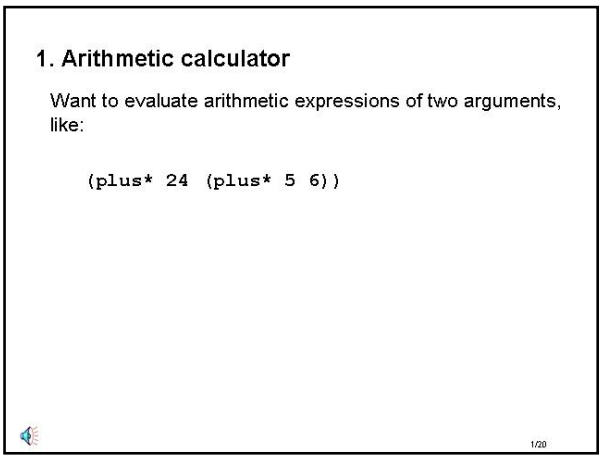
We say this because we don't want you to get confused between what is going in Scheme and the general ideas of building an evaluator and interpreter. Our goal is to build an interpreter, especially the evaluator part, and let you see how that occurs and use that to explore the idea of how these things implement the rules for a language. We are going to build our own simple language and its evaluator. For convenience, we are going to call this language, **Scheme***. It has a lot of the characteristics of Scheme, but we will use the convention that a * will be placed at the end of every expression in **our** language, to distinguish it from the corresponding Scheme expression.

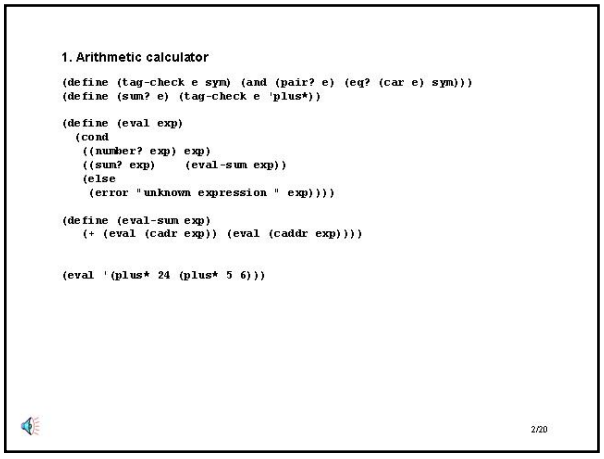
We'll start with a simple evaluator and work our way up. The first simple evaluator will be one that handles simple arithmetic expressions.

6.001 Notes: Section 15.2

Slide 15.2.1

Our plan is to start by building an evaluator that handles arithmetic expressions, and in fact we will restrict ourselves just arithmetic expressions of two or fewer arguments. We would like to be able to evaluate things like the example shown on the slide: adding 24 to whatever we get by adding 5 and 6. Notice the `*` at the end of the symbol `plus` to indicate that this is something that we will build within our language.





Slide 15.2.2

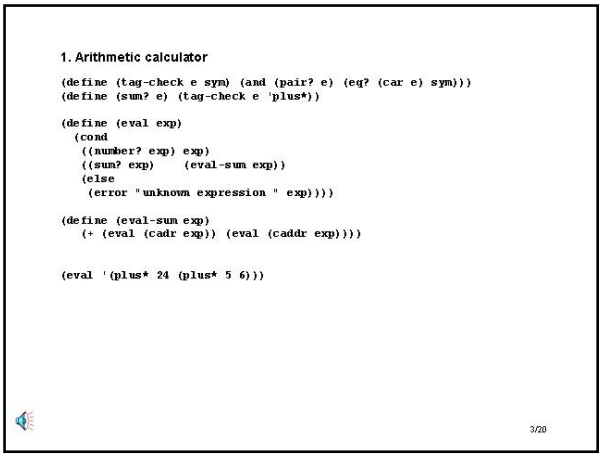
And here is some code that captures how we will evaluate expressions of this form. This is identical to the code listed in the separate code handout, and I suggest you have that page handy as we go through this development.

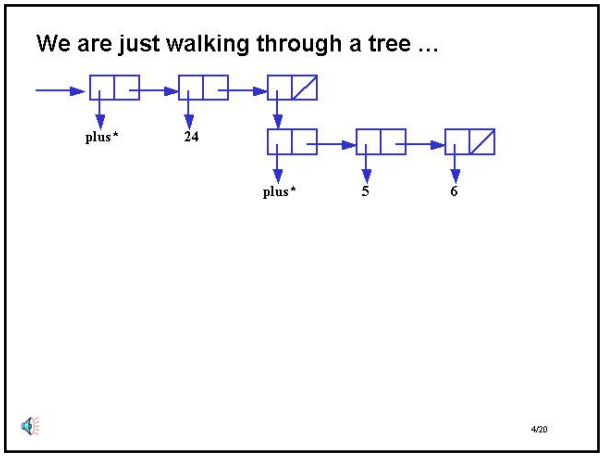
Slide 15.2.3

Notice what we are doing here. We are using our knowledge of Scheme to describe the process of evaluating expressions in this new language. We are writing, **in Scheme**, the description of that process.

Okay, what do we need? We have a procedure for evaluating expressions in our new language, called `eval`. Notice its form. It has a way of dealing with the base case, which is an expression that just consists of a number. And to do that it uses **type checking**.

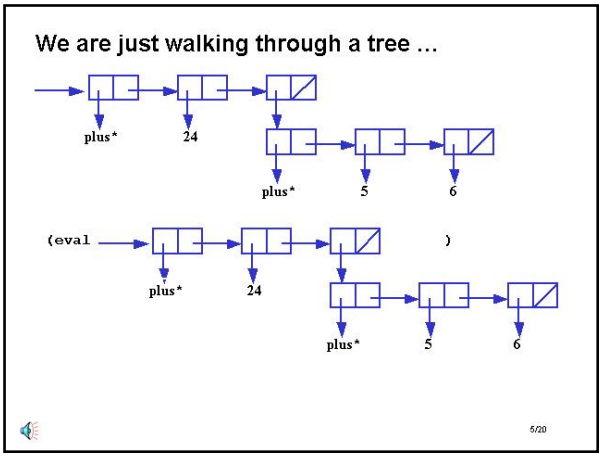
Then, we have a way of dealing with the compound case. Here, it uses type checking to see if we have a **sum** and notice how this works. It uses the keyword of the expression to determine the type of that expression. If the expression is a **sum** then we will just add, using the primitive operation of addition, the values of the subexpressions. But a key point arises here! To get those values we need to evaluate each subexpression as well, since we don't know at this stage if they are just numbers or are themselves compound expressions.

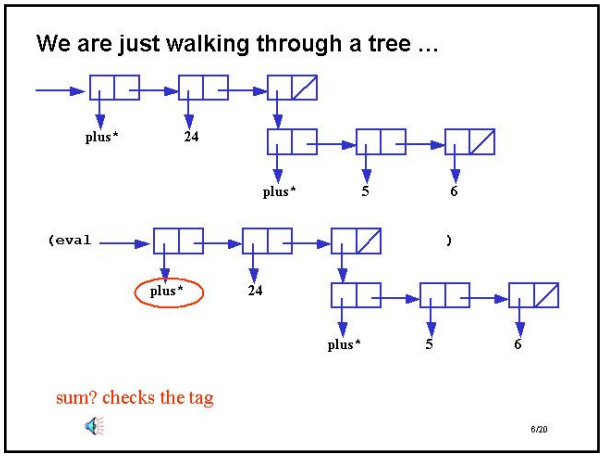




Slide 15.2.4
Let's look at this in more detail. First, let's look at the input to this evaluation process. Remember that our expression, which we typed in, is converted into list structure, as a tree of symbols and numbers. It looks like what is shown on the slide, and this is what gets handed to the evaluator as a representation of our example expression. So let's treat this as if this exact tree structure were passed in to `eval`.

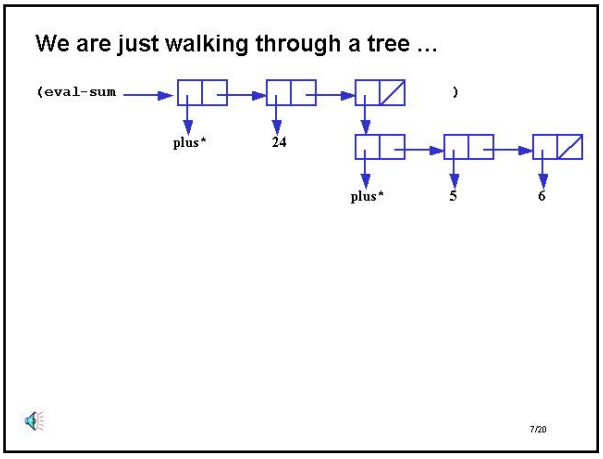
Slide 15.2.5
And what does `eval` do with this input? Check the code on the handout. `Eval` grabs the list and tests its tag. That means it first checks to see if this whole thing is a number. Since it is not, it takes the first element of this list structure and checks to see if it is the special symbol `plus*`.

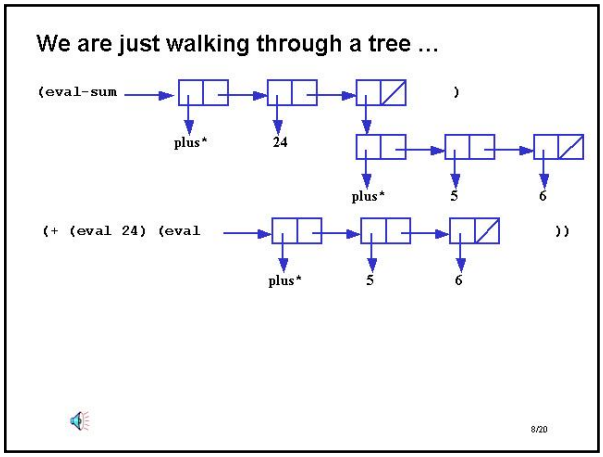




Slide 15.2.6
Having done that it dispatches on type to the right procedure to handle this kind of expression. Having determine that it is a `sum` by checking the tag, it sends it off to `eval-sum`, and this is (for now at least) just a normal procedure application. We apply the procedure to the expression.

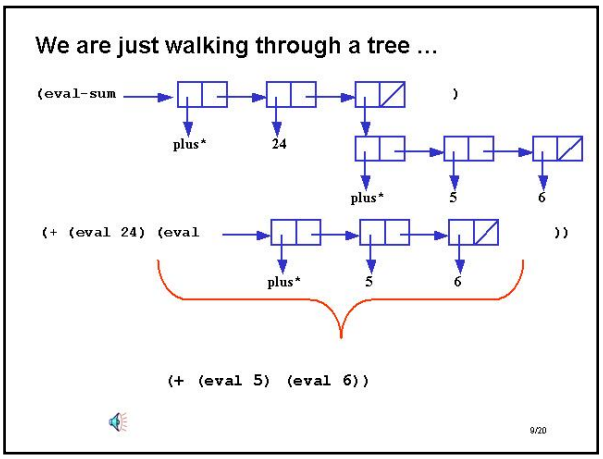
Slide 15.2.7
So now `eval` has reduced this to applying `eval-sum` to the tree structure shown. Notice what the body of `eval-sum` does. It walks down the tree, grabbing out the two subexpressions, that is the first and second components of this sum. `Eval-sum` then converts this into adding, using the built-in primitive, whatever I get by evaluating the first subexpression and whatever I get by evaluating the second subexpression.

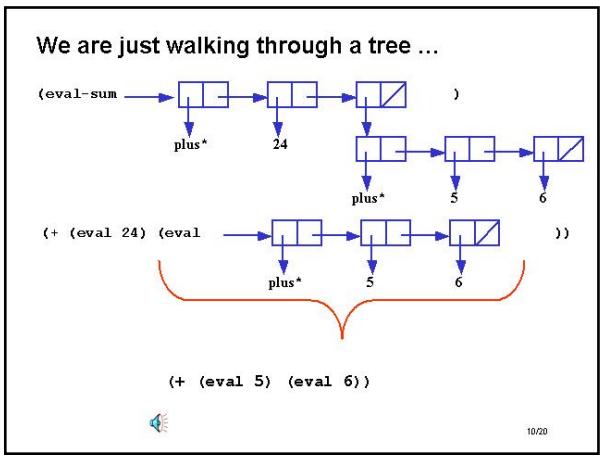




Slide 15.2.8
That leads to this form. Notice what we have done: we have reduced evaluation of one expression into some simple primitive operations on evaluation of other subexpressions. And what has that done? It has literally just walked down the tree structure that represents this expression, pulling out the right pieces. It has used the tag to tell us who to send the expression to, and then it has simply grabbed cars and cdrs of the list structure, and handed them off to new evaluations. Now at this stage, evaluating the first subexpression `(eval 24)` is easy. We see from our code that it will use type checking to determine this is a number and simply return that expression. And that's nice! This is just pointing to the number `24` so the number `24` gets returned. What about the other piece?

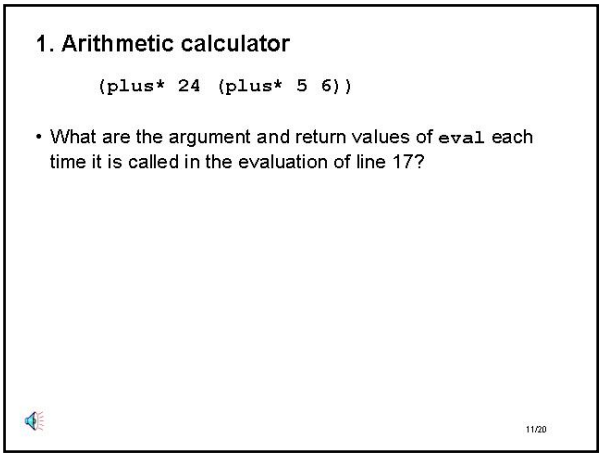
Slide 15.2.9
Well this just looks like the kind of expression we started with. We are evaluating some list structure that happens to represent a sum. It's got a tag at the front to say it is one of these `plus*` expressions, so we can do **exactly** the same thing. The evaluation of this expression will unwrap into an `eval-sum` of the same list structure, and that will reduce to a primitive application of `+` to whatever I get by evaluating the subexpressions, and that I get by walking down the tree, grabbing the right pieces, applying `eval` and getting back the numbers.

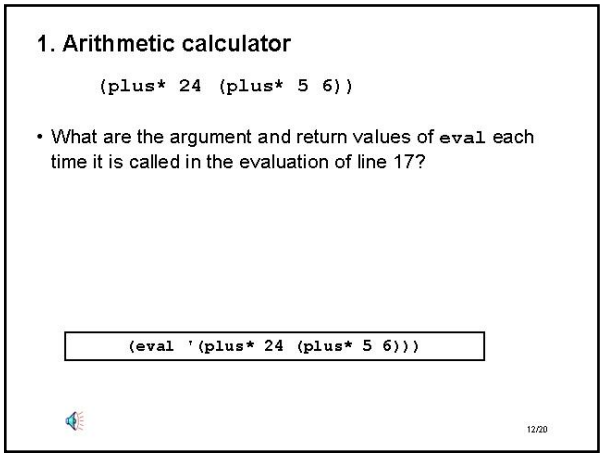




Slide 15.2.10
And now we see that we have unwrapped this abstraction down to some primitive operations, primitive application of addition to some simple expressions, in this case just numbers. And of course this will finally reduce to the answer we expect. However, a key thing to note is how this simple evaluator has taken in a tree structure representing an expression and has unwrapped it into successive evaluations until it reduces to a set of applications of primitive built-in procedures to primitive values.

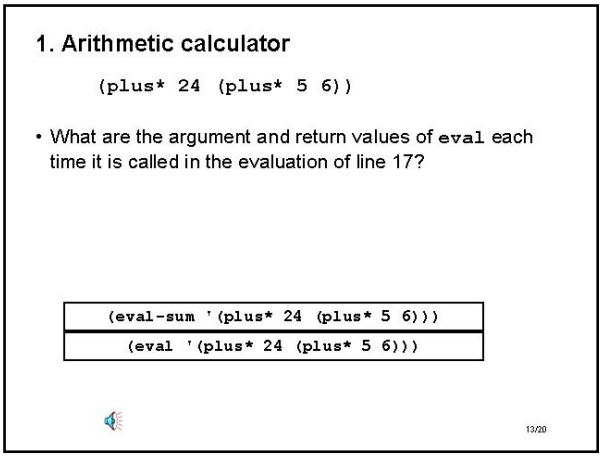
Slide 15.2.11
Since this is important, these stages of eval unwrapping into simpler and simpler things, and the dispatching on type to the correct procedure, let's look at this one more time. In this case, let's focus on how `eval` unwinds the abstraction, and what values are returned at each stage of the evaluation. As before, you may find it convenient to have a copy of the code in front of you as we go through this examination.

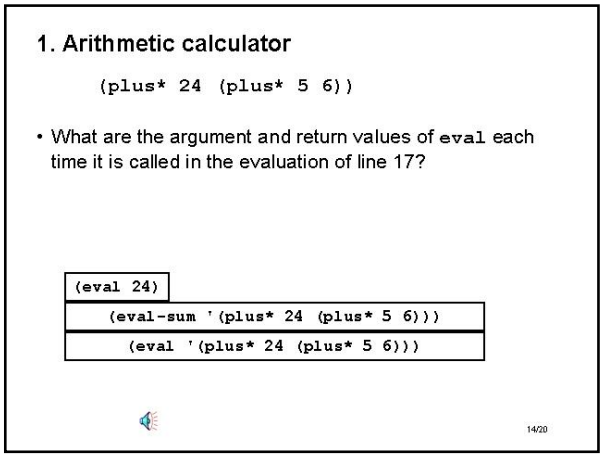




Slide 15.2.12
So we start with `eval` of this full expression. We've put a `'` in front of the expression to show that we want list structure equivalent to this expression.
Thus we start with an `eval` of this expression.

Slide 15.2.13
`Eval` first checks the type of this expression, deduces that it is not a number, but is a sum (because of the type tag), so this expression gets dispatched to `eval-sum`. `Eval` sends the expression to the procedure that is exactly set up to deal with this particular form of list structure.

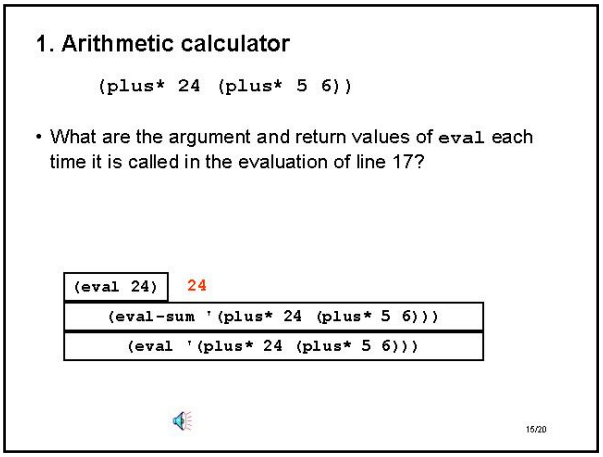


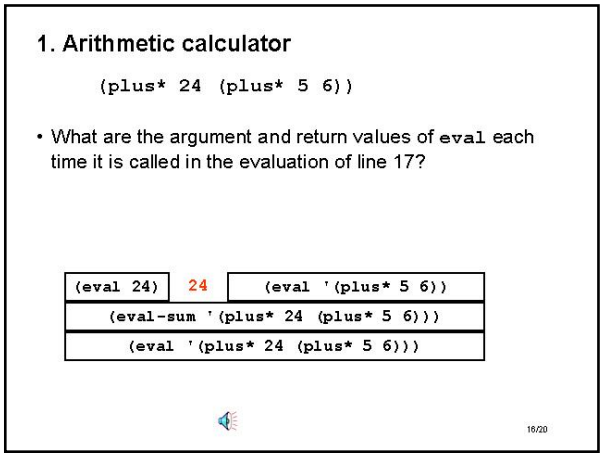


Slide 15.2.14
Now, `eval-sum` says, "go ahead and add whatever I get by `evaling` each of the pieces". We haven't actually specified in what order to do the subpieces, but for convenience assume that it is done from left to right. So we now need to trace down the tree, and get `(eval 24)`.

Slide 15.2.15

`Eval` once again checks the type of this expression, deduces it is a number, and just returns that expression, literally a pointer to that thing which is an internal representation for the number 24.



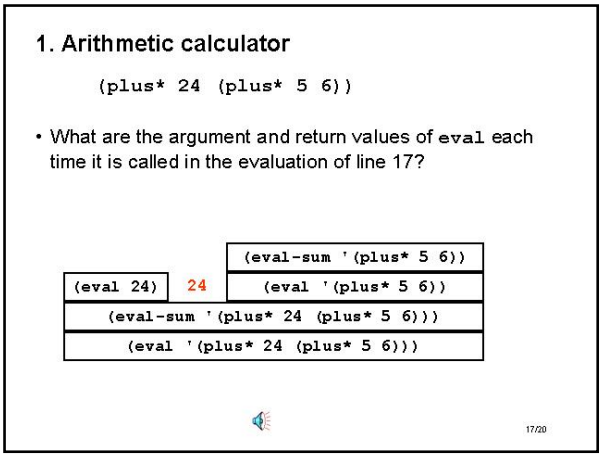


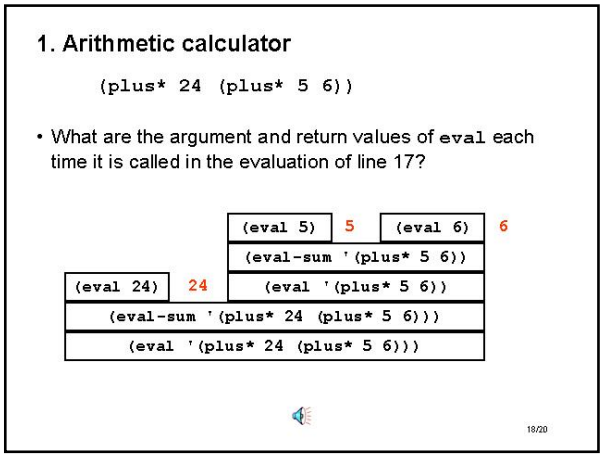
Slide 15.2.16

Next, `eval` has to evaluate the second subexpression, so this is the `eval` of the expression shown at top right. As before, we are going to dispatch on type, i.e. check to see what kind of "beast" this is, deduce that it is a "sum" and therefore pass this on to the right procedure to handle sums.

Slide 15.2.17

Once more, `eval-sum` will reduce to applying the addition operation to whatever it gets by evaluating the subpieces. Thus, we need to extract the subexpressions and once again apply `eval` to them. Notice the nice recursive unwinding that is going on here.



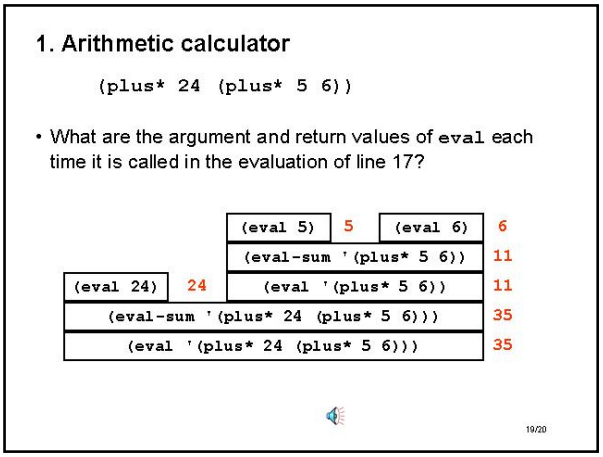


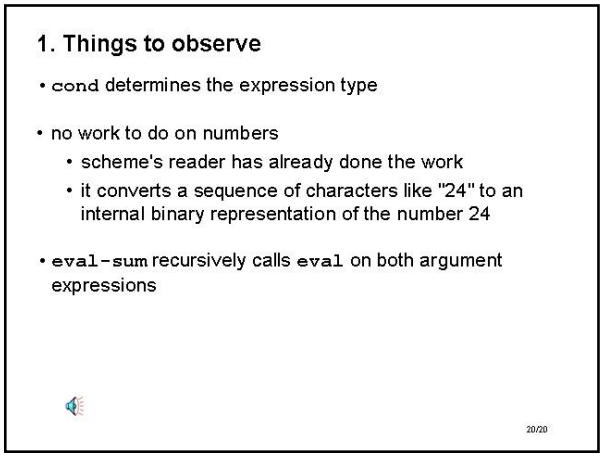
Slide 15.2.18

Well this just unwraps one more time. Again, we will apply `+` to whatever we get by evaluating the two pieces, and `eval` in both cases just dispatches on type, determines the expression is a number and returns the expression as the value.

Slide 15.2.19

Notice where we are at this stage. We have unwrapped this compound expression into a nested sequence of operations of primitive things to primitive values. At this stage we can gather up the things we have left to do. We have some deferred operations, for example in the topmost `eval-sum` we can now add 5 and 6 to get 11, and so on, reducing all the deferred primitive operations down to a single value.





Slide 15.2.20

Thus, we have built a simple evaluator that handles sums of no more than two arguments. Here are some key points to notice from this exercise, since our goal is to understand the process of evaluation.

First, `eval` does type tag checking. It dispatches based on type, much like we saw earlier in the course.

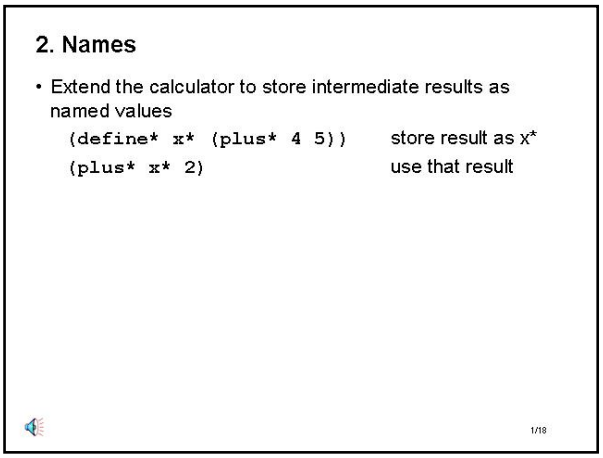
Second, numbers are just numbers, so there is nothing really to do.

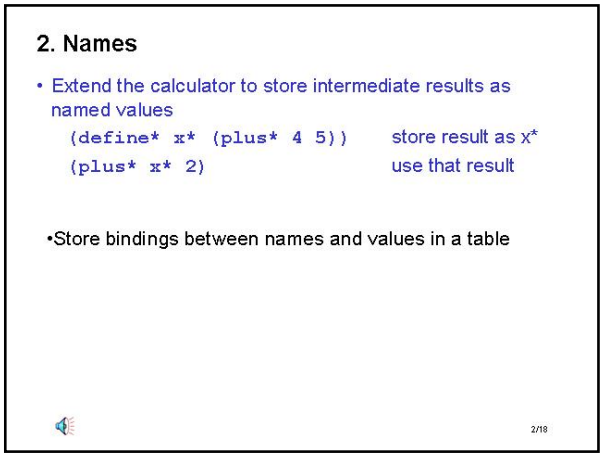
Third, complex expressions nicely get recursively evaluated in pieces. `Eval` unwraps a complex expression into an evaluation of the simpler parts, plus a deferred operation to gather the values back together. Numbers just get handled as numbers. And eventually we reduce this whole thing down to a set of primitive operations on primitive values.

6.001 Notes: Section 15.3

Slide 15.3.1

Okay, now let's build on this basic system. Suppose we want to give **names** to things. For example, suppose we want to have the behavior shown here, in which we can store intermediate results as named values, and then just use those names anywhere that we would want to use the actual expression and its resulting value. This is the kind of behavior we saw earlier in the term in Scheme, how would we add that behavior to the evaluator we are building for simple arithmetic expressions?





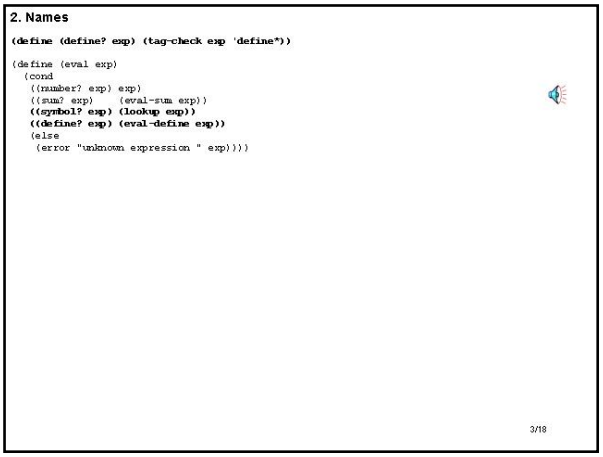
Slide 15.3.2

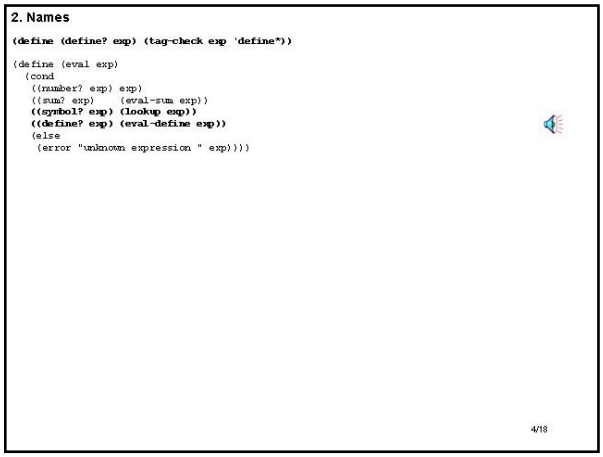
Of course, the first thing we realize is that this means we will need a way of storing **bindings** between names and values. We can certainly imagine getting the value for something, but now we have to have a way of storing the name and value together. `Define*` has to have some way of gluing pieces together. So we need to add this capability to our evaluator.

Slide 15.3.3

This is now the second page of your code handout. Don't be intimidated by this code, as we have highlighted the things we have changed from the first evaluator, shown here in **bold face font**.

So what have we added? First, we need another type checker, something that checks whether the expression is a `define*` expression, here called `define?`. We have also added two new pieces to the evaluator.



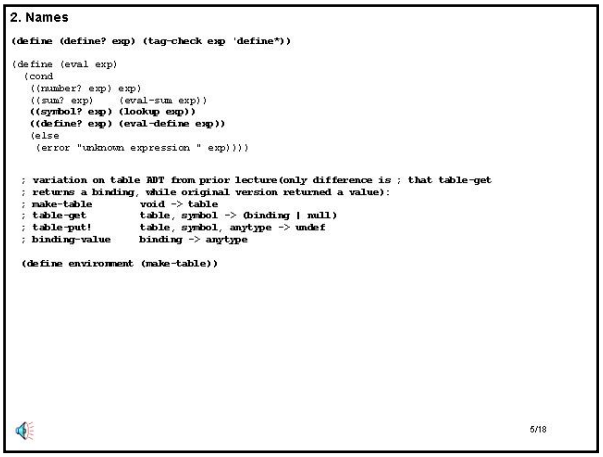


Slide 15.3.4

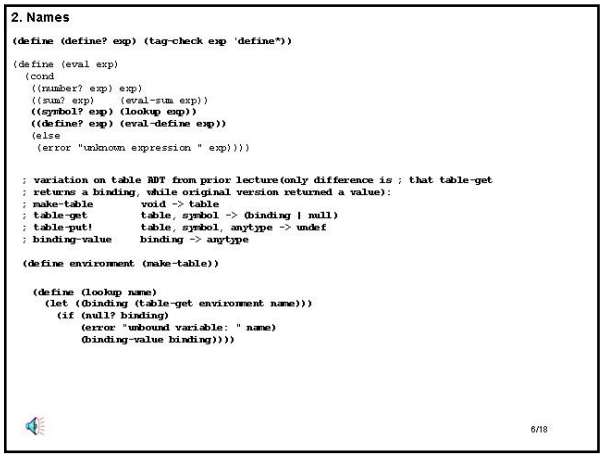
Note that in this version of `eval` we have a way of creating names for values, and a way of getting back the value associated with a name. Thus, we have two new dispatches in our evaluator: something that checks to see if the expression is a symbol, in which case we will lookup its value; and something that checks to see if the expression is a definition (checked using the special tag `define*`), in which case we dispatch to something the evaluates these special expressions.

Slide 15.3.5

Before we look at the procedures that will handle `lookup` and `eval-define`, let's first think about what we need. We will need a way of gluing things together, and we know how to do that. Let's just assume a data abstraction, called a **table**. It has a constructor, `make-table`. It has a way of getting things out of a table, thus given a table and symbol, `table-get` gives us back either a `nil` to indicate no binding for that symbol was present, or the actual binding. We have a way of putting things into the table, `table-put!`, and we have **binding-value** which when given a binding, returns the value part of that pairing.

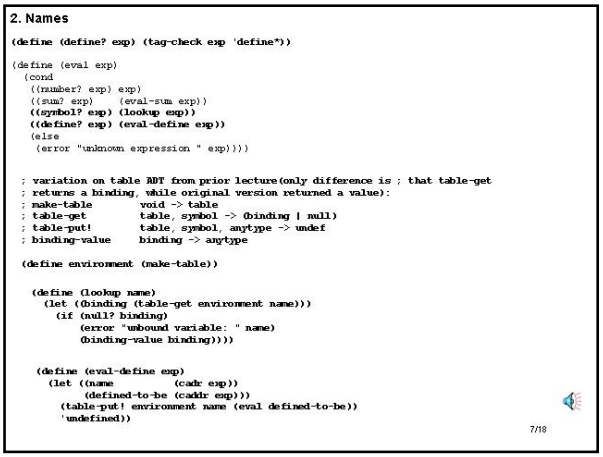


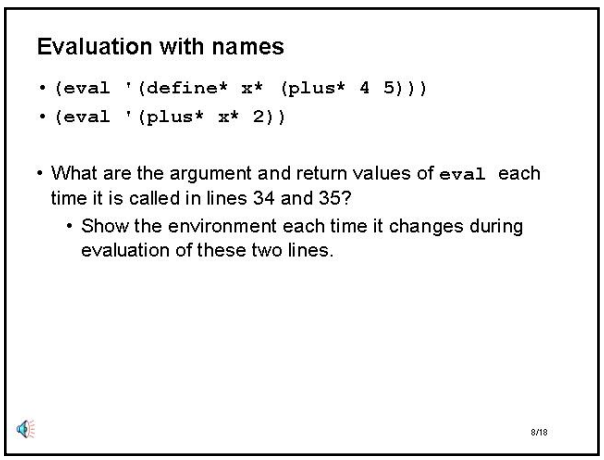
What is the point of this? We can simply assume that this table abstraction exists, and then we can build an **environment**. Let's define `environment` in our underlying Scheme to be a table, and then we can create procedures for `lookup` and `eval-define`.



Slide 15.3.6
Looking up the value of a symbol is simply a matter of manipulating the table. We find the binding in the table for this symbol, and then return that value part of the binding. This is simply manipulating the environment and we can abstract that away.

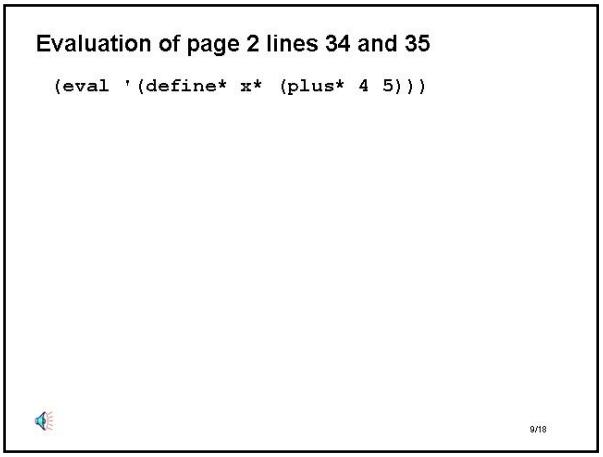
Slide 15.3.7
The real thing we have added is a way of dealing with a new kind of expression, something built by `define*` that is creating a binding of a name and a value. So what should this do?
This procedure says to walk the tree structure to get out the name (remember this is just walking the tree structure, there is **no** evaluation going on here). Then, **evaluate** the expression that will provide the value of the binding. Notice the use of `eval` which recursively returns to the top level, and evaluates this expression using the same rules! Once I get a value, I stick it into the table that represents the environment, paired with the name in a binding.

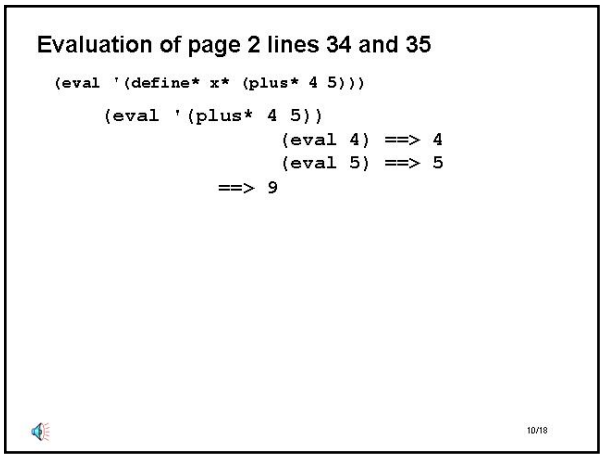




Slide 15.3.8
Since we have added a new component to our evaluator, let's again look at what happens if we evaluate these two expressions in our evaluator, especially watching to see what is returned each time we recursively call `eval` in this process. So keep track of the argument and return values of `eval` as we trace through this process, using the code handout to keep track of this process.

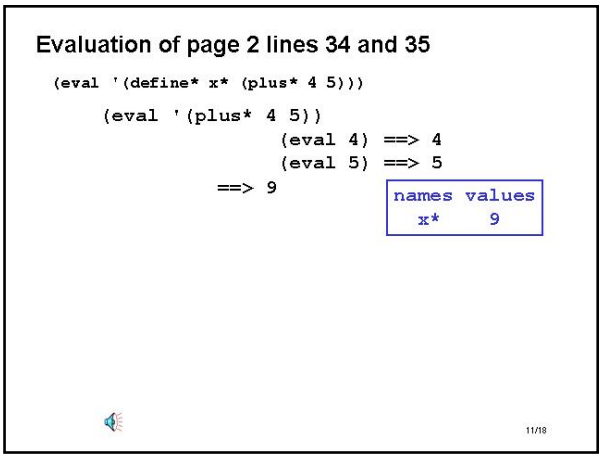
Slide 15.3.9
So let's use our extended evaluator, by evaluating this `define*` expression. Remember that this expression is just represented as tree structure, the `'` is just used here to remind that the parser will have reduced this input expression to the equivalent tree structure. So what does `eval` do with this expression? It is a dispatch, so it will deduce that this is a `define*` expression, and will thus send the expression to a handler. That procedure will decide to `eval` the value subexpression.

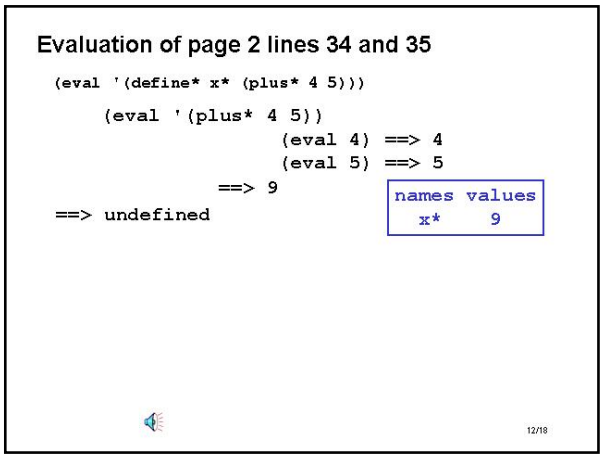




Slide 15.3.10
Well that is a lot like what we saw before! We are now doing an `eval` on **this** list structure. And we have literally walked down the list structure for the first expression to grab off this piece and now evaluate it. This just recursively determines the type of expression (a `sum`) and thus passes it off to `eval-sum` which evaluates each of the subexpressions in turn, then applies the primitive `+` operation to return a value (9). So the value returned by evaluating this subexpression is just `9`.

Slide 15.3.11
And then what does `eval-define` say to do (remember this is where we were when we went off to evaluate the subexpression). It simply takes the name (or symbol) `x*`, grabbing that off of the tree structure, it takes the returned value, and it binds them together in the table somewhere.



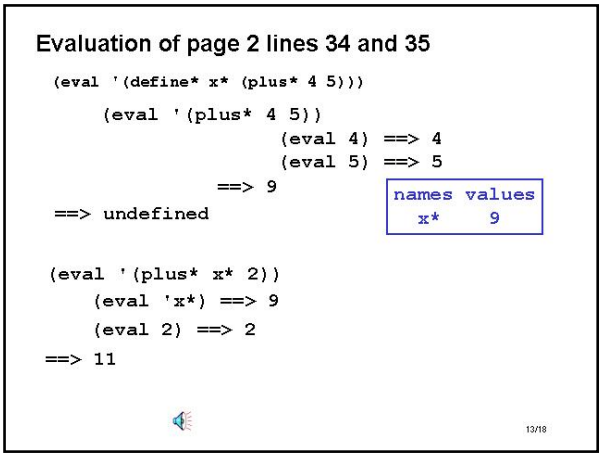


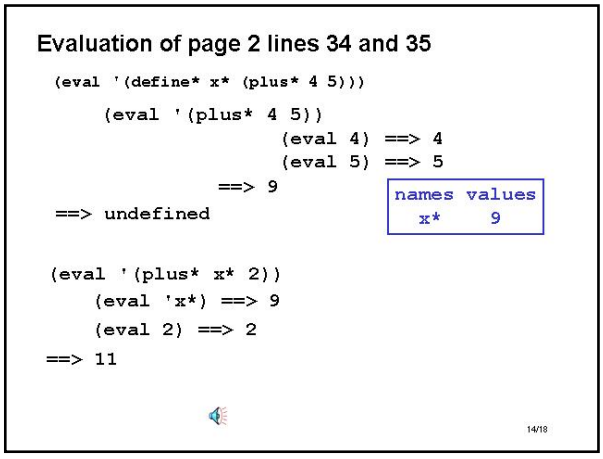
Slide 15.3.12
And having done its work, it just returns the symbol `undefined` to tell us it has completed the evaluation of this definition.

Slide 15.3.13

So now let's evaluate an expression that includes one of these names, such as the expression shown. As before, this expression is converted to tree structure, which is then evaluated. The tag checking dispatches this to `eval-sum`. That procedure says to add together (using `+`) whatever I get by recursively evaluating the subpieces.

First is `(eval x*)`. Notice in line 9 of evaluator that because this part of the tree structure is a symbol, we do a `lookup`. That is, we use the table abstraction to get the binding associated with this symbol, and returns the value associated with that binding in the table, in this case `9`. The rest of the evaluation proceeds as in previous cases.





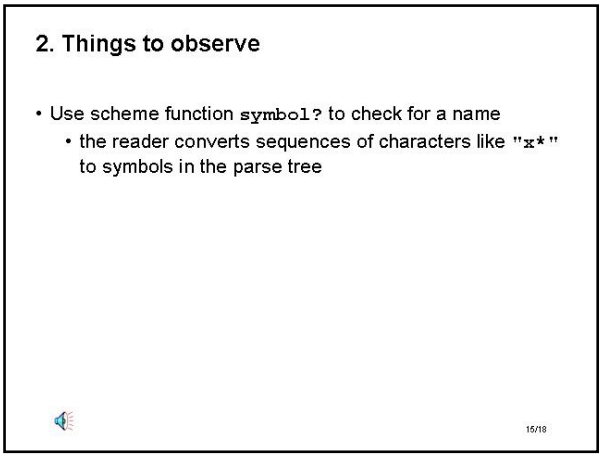
Slide 15.3.14

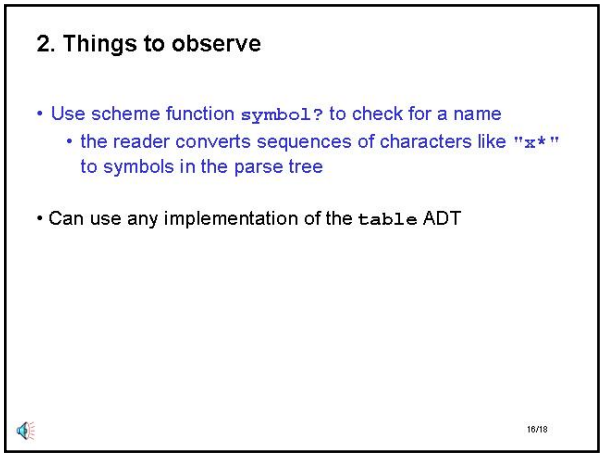
So in fact this was fairly straightforward. As long as we have the abstraction of the table, we see how we can add bindings to it, through definitions. Notice also how just walking the tree structure of the expression allows us to grab off symbols without evaluating them, in this case, getting `x*` so that we could bind it together with a value in the table. Other than that, all the other things still hold for our evaluator, we can simply now give names to values.

Slide 15.3.15

Having now extended our evaluator by adding in this new capability, let's step back and extract some key messages. First of all, notice that we added two new **dispatch** clauses to our evaluator. One of them just dispatches to `eval-define` using standard type checking of tags. The other clause, however, relies on using `symbol?`, the underlying Scheme function, to check for a name. Why is this reasonable?

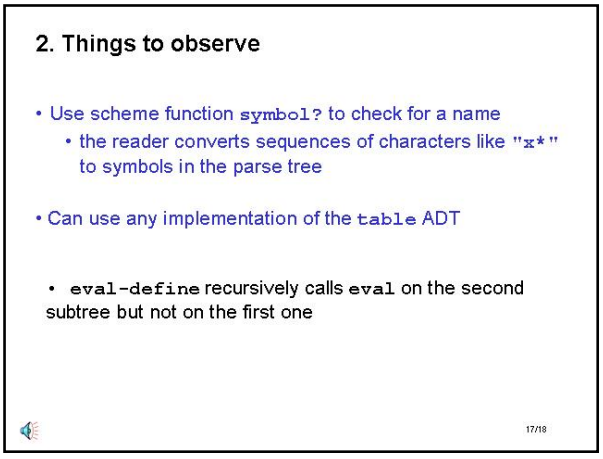
Note that we are relying on the underlying Scheme reader to convert things into a parse tree, and in particular, that reader will convert sequences of characters like `x *` into symbols in the parse tree, which then gets passed on to `eval`. Thus, `eval` will get a structure that can be handled by `symbol?`.





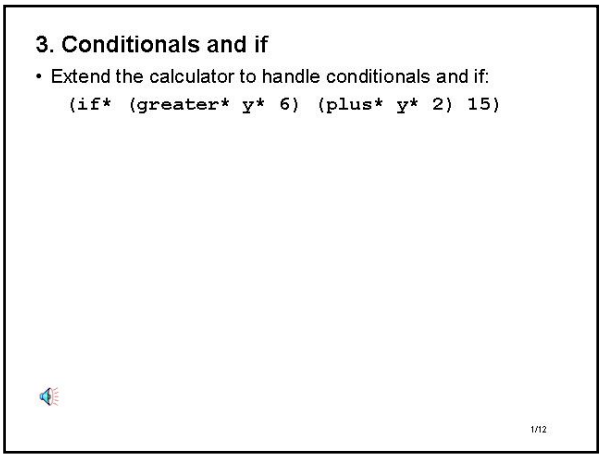
Slide 15.3.16
Note the second thing we added, a table. Or rather, we added an abstract data type of a table to represent our environments. Note that we could have used **any** implementation of a table. We are not relying in **any way** on the specifics on a table implementation, and that is exactly the point. As long as we have a way of creating tables, we can build an evaluator that simply manipulates them.

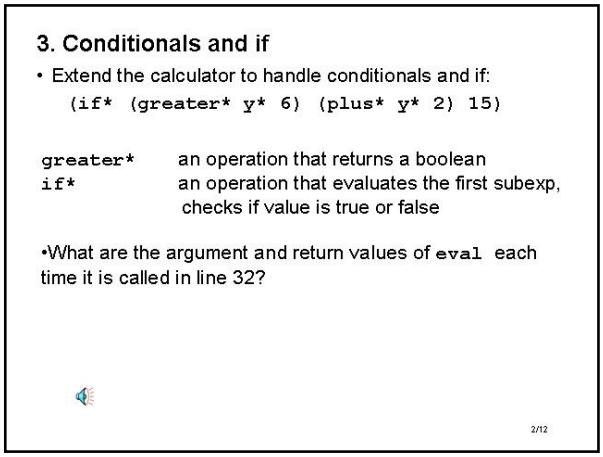
Slide 15.3.17
A key thing to note as well is that we have now added a **special form** to our little evaluator! If you think carefully about it, you will note that prior to this we were simply evaluating normal combinations, which for this language were sums of two arguments. However, `eval-define` does something different. It recursively evaluates only the **second** subexpression! In other words, `eval-define` takes the first subtree of the parse tree passed in, without evaluation, and treats it as a symbol. It only does evaluation on the second subtree.



6.001 Notes: Section 15.4

Slide 15.4.1
What else can we add to our evaluator? To this point, we have no way of making decisions. We can't branch to do different things depending on a value. So let's extend our evaluator to handle this, i.e. let's extend our calculator to handle conditions and ifs: Statements such as the example shown, in which if something is true we want to do one thing, otherwise we want to do something else.





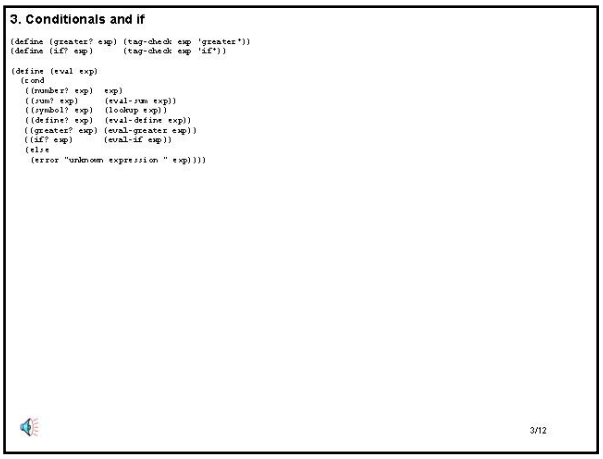
Slide 15.4.2

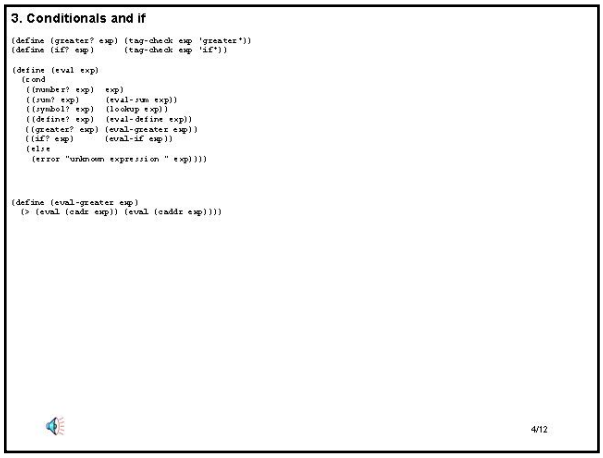
We can already see from our example that we are going to need to add two kinds of things. First, we will need procedures that return Boolean values. Thus, `greater*` should evaluate a pair of expressions, then return `true` if the value of the first expression is larger than the value of the second expression. Of course, we can easily imaging adding other such expression to our language, so long as we have some expressions that evaluate expressions and return Boolean values.

The second thing we will need is an operation that lets us control the branching of our code. We know that this should be a special form in which we should first evaluate the first subexpression, check its value, and then based on that value either evaluate and return the second subexpression or evaluate and return the third subexpression. This means we need to add an `if*` expression to our language.

Slide 15.4.3

As before, there is a fair amount of code here, but we have **highlighted** the changes from the earlier version. First, we need two new ways of dispatching on `greater*` and `if*` expressions, just using type checking as before. This means we need ways of checking the type of such expressions, and we need to add dispatch clauses to the `cond` expression inside `eval`, one for each new type of expression. This is just like before.



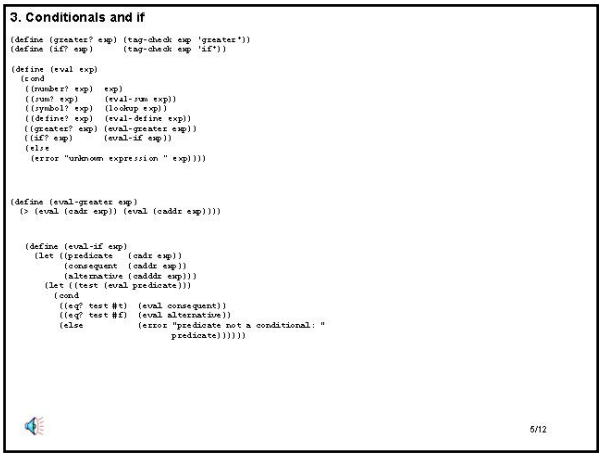


Slide 15.4.4

First, `greater*` expressions. Remember that `exp` will be a pointer to some tree structure that has been created by the parser. The first subexpression in that tree structure will be the symbol `greater*`. The second subexpression will be some other expression, as will the third subexpression. Our plan is to get the values of the second and third subexpressions. We will use tree operations to extract the right pieces, the recursively `eval`uate them. Once we have the values of these subexpressions, we can use the underlying Scheme primitive operation to compare the numerical values and return a Boolean answer.

This may seem a little odd; why not use Scheme's operation directly? Remember, our goal is to understand how to connect the evaluator to built-in primitives, and to allow for alternative ways of executing operations. So remember what we do, we walk the tree, grab the pieces, recursively evaluate them, then pass the pieces on to primitive procedures to complete the reduction to a value.

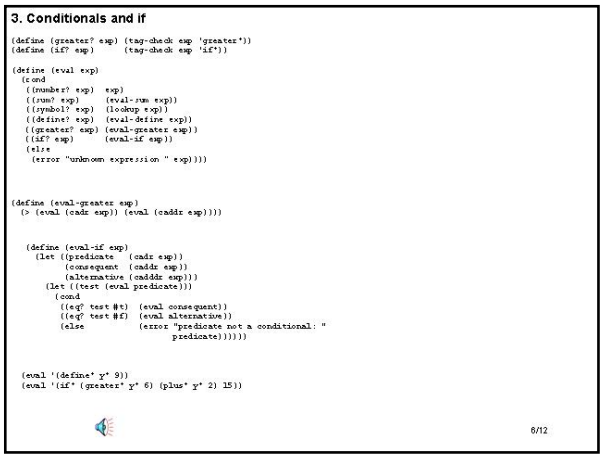
Slide 15.4.5



What about `if*` expressions? What happens when we dispatch to `eval-if`?

Once more, remember that what is passed in to this procedure is tree structure, created by the parser. This tree structure will several pieces: the tag `if*`; then tree structure representing the predicate of the expression; then tree structure representing the consequent of the expression; then tree structure representing the alternative of the expression. Notice how in our procedure we use list manipulation to pull off each piece of the tree, and then we evaluate the first one.

Once we have the value of `test`, we can then use the underlying primitive mechanism for branching, `cond`, to decide that if `test` is `true`, then and only then, take the consequent and evaluate it. If `test` is `false`, then and only then, take the alternative and evaluate it. Notice the order in which things are done. This is a special form in which we first evaluate the predicate. Depending on that value, we either evaluate the consequent or the alternative, and return that value as the value of the overall expression.

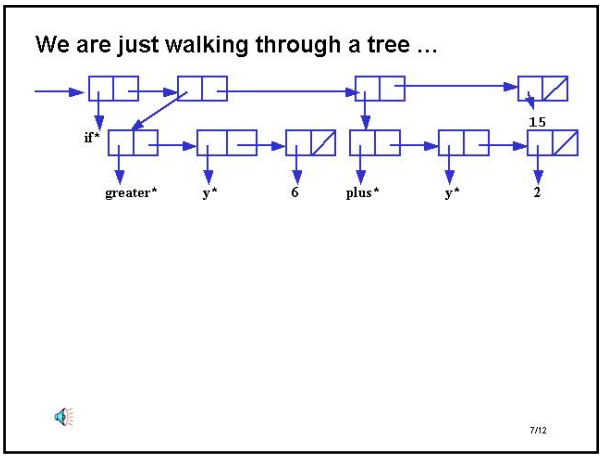


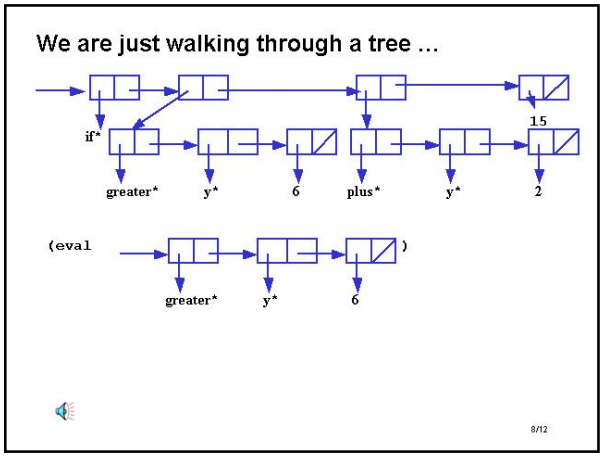
Slide 15.4.6

So having added these two new expressions to our evaluator, let's look at some examples of evaluation using them, as shown on the slide, tracing through the process of recursively using `eval`.

Slide 15.4.7

We can assume that we have done the evaluation of `(define* y* 9)`, since we know what that does. Now let's look at evaluating the `if*` expression.



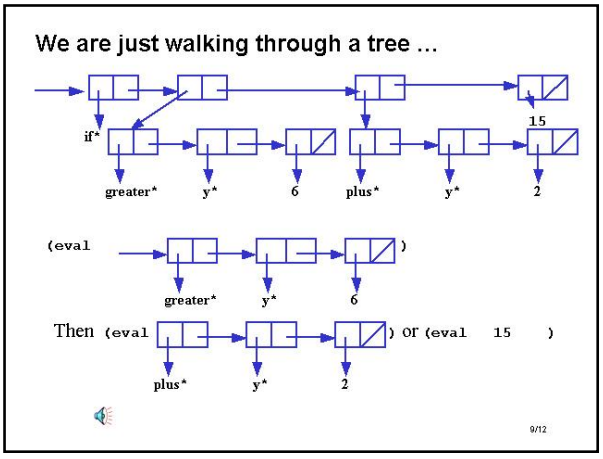


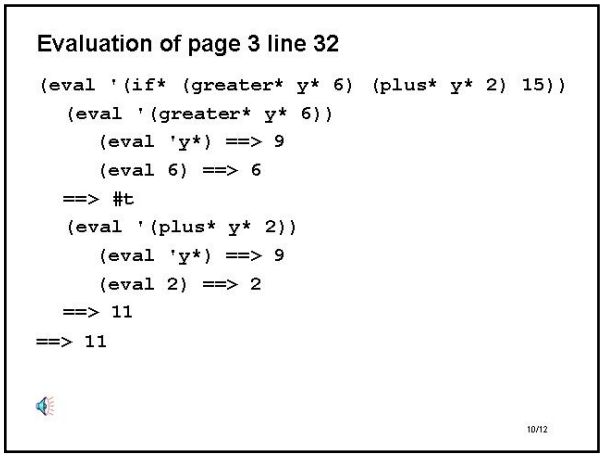
Slide 15.4.8

The first thing `eval` does is check this expression. Is it a number? Is it a sum? Is it a symbol? Is it a define? Is it a greater? Is it an if? In other words, it checks the tag on the expression, eventually determining that this is an `if*` expression. Thus it dispatches the expression to `eval-if`. If you look at the code handout, you will see that `eval-if` first walks down the tree structure and pulls out the predicate expression. It then `eval`uates that piece, as shown.

Slide 15.4.9

Depending on what value is returned by evaluating that first piece, the evaluator will either walk the tree structure to get out the consequent piece or the alternative piece, and it will evaluate that piece. Thus, this special form will first evaluate one piece, and then branch to an evaluation of one of the other two pieces, but only one of them.





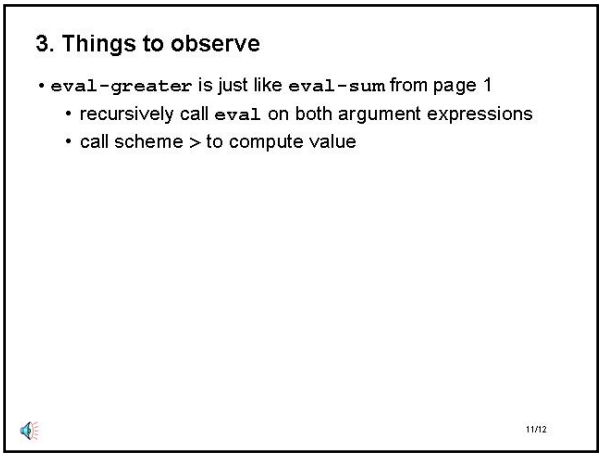
Slide 15.4.10

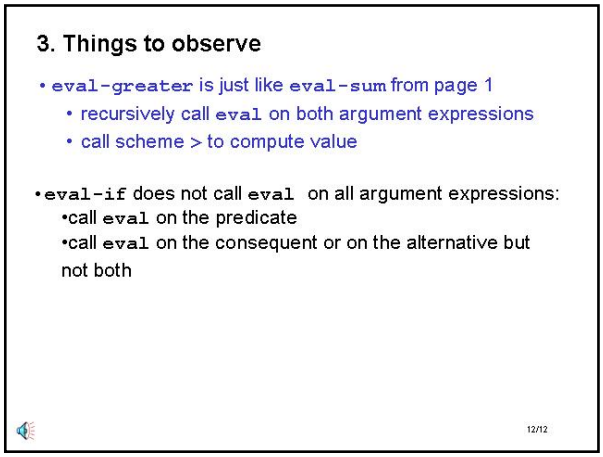
Then we can just step through the stages. The first stage evaluates the full expression. This dispatches to `eval-if`, which evaluates the first subexpression in the tree structure. That does a second dispatch to the procedure that handles `greater*` expressions. Here we walk further down the tree to get the subexpressions to this procedure, and recursively evaluate them. Having returned values (one by lookup, the other as a number), we actually apply the primitive comparison operator, and since the returned value is `true` we then extract the appropriate piece of the tree (the consequent) and we evaluate that piece. This just becomes a dispatch like the previous cases, and you can see the recursive calls to `eval` used to get the values of the subexpressions before the application of the primitive procedure.

Slide 15.4.11

Once more, let's step back and extract some messages from this exercise.

Note that `eval-greater` is really just like `eval-sum`. Both recursively call `eval` on the arguments, then apply a primitive procedure to the resulting values.





Slide 15.4.12

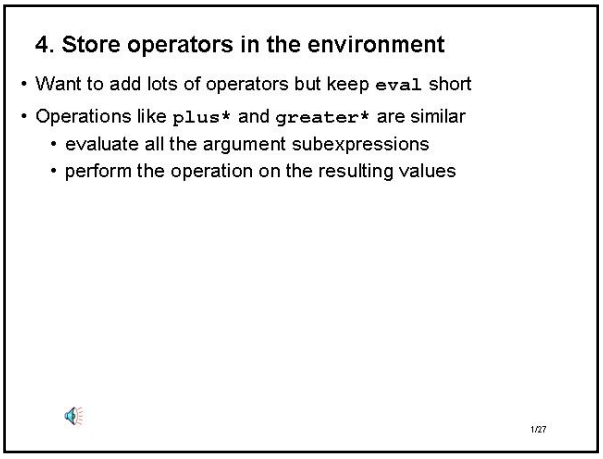
On the other hand, our conditional, `eval-if`, behaved differently. It is a special form. Rather than evaluating all of its arguments, it only evaluates some of them, and in a particular order, as shown on the slide.

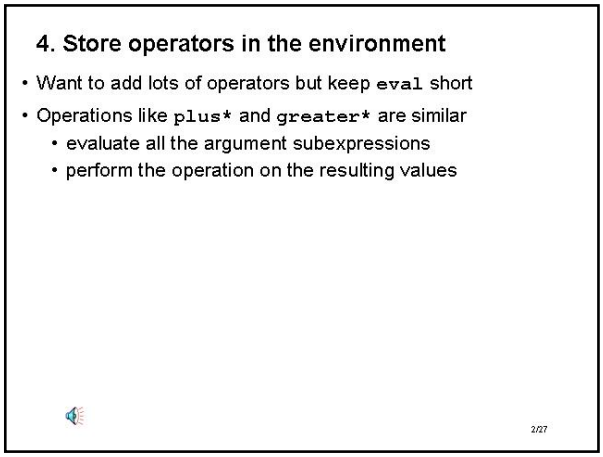
6.001 Notes: Section 15.5

Slide 15.5.1

In the last example, when we introduced a new kind of expression into our evaluator, we had to add a new dispatch into `eval`. We created a new type checker for the expression, and then we added a new clause to the `cond` to dispatch off to a procedure to handle this kind of expression.

That's okay with a small number of things, but we would like to add lots of operators, for multiplication, exponentiation, less than, and so on. How can we add lots of operators to our evaluator but still keep `eval` compact, and more importantly easy for us to extend later?



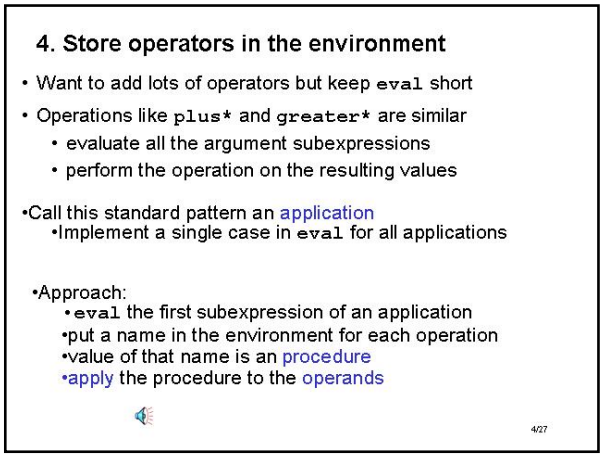


Slide 15.5.2

We have already hinted at how to do this. We have noted that operations like `plus*` and `greater*` are quite similar. They evaluate all their arguments, and then they perform a particular operation on the resulting values. That sounds like a common pattern, so we would like to capture that pattern and take advantage of that abstraction.

Slide 15.5.3

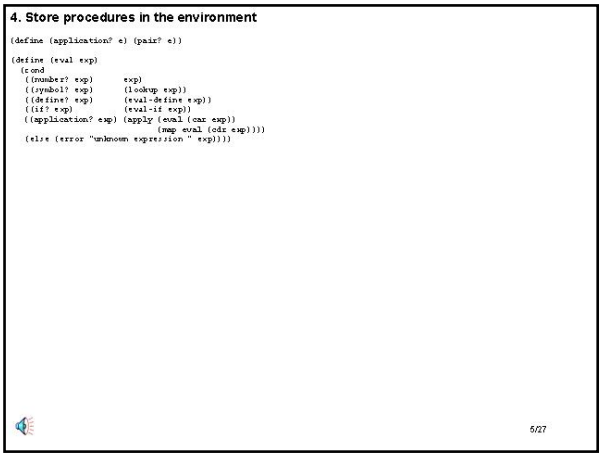
In particular, we will call this standard pattern an **application**. It is an application of a particular operator to a set of arguments. We can therefore implement a single case in our evaluator for all applications: just one way of dispatching. That will allow us to easily extend other operations to our evaluator, by simply creating the operation and having the same dispatch handle the application.



Slide 15.5.4

Here is our plan for accomplishing this. We will now first evaluate the first subexpression of an application. Note that in the previous cases we did not do this, we used the name to dispatch in `eval` to the right primitive procedure. Now we will evaluate the first subexpression to deduce the operation to apply. But that means we have to have a way of getting access to the appropriate operation, so we will put a name in the environment for each operation that we want to treat as an application. The value of that name will be a procedure and then we can simply apply the procedure to the values of the other subexpressions. How can we make this extension to our evaluator?

Slide 15.5.5

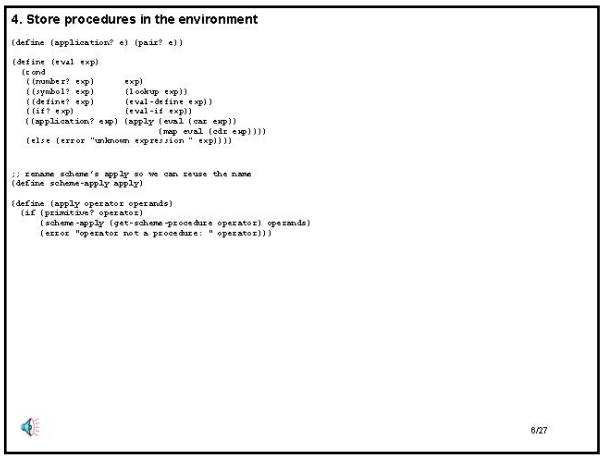


As with the previous cases, we suggest that you look at a printout of the code as we walk through this exposition. As before, we will try **highlighting** the changes we make to our existing evaluator, as we evolve to handle new kinds of expressions.

First, we need a way of detecting an application, and we are going to simply rely on the expression being a **pair**, that is, if it is a combination, then it is an application. But if we are going to assume that, then in our dispatch cases inside `eval`, applications had best be things we do **after** we check for the special kinds of things. And thus we can see a form evolving. In `eval` we first check for the primitives (numbers and symbols), and then we consider compound expressions and check for any that are special forms (things that do not obey the normal rules for evaluation, such as `define*` and `if*`). Finally, we check for applications, that is, we make sure we do have a compound expression, in which case we treat it as an application. Notice that we can now drop the special checks for `greater*` and similar operations, since they will now be caught by the application case.

Notice what we do if we have an application. Remember that the input is one of those tree structures representing the expression. We grab the first element of the tree and **evaluate** it. That should give us a procedure. We then walk down all the remaining elements of the tree and we will **map** `eval` down that list. We are just treating `map` as a primitive here, so that it applies `eval` to each element of that list and create a new list of the resulting values.

Then we will **apply** that procedure value to that list of arguments.

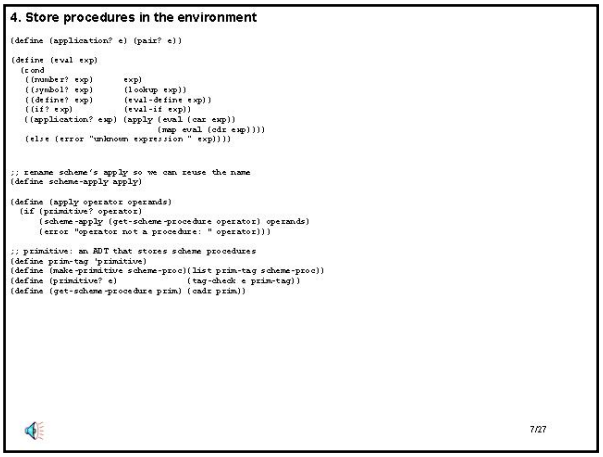


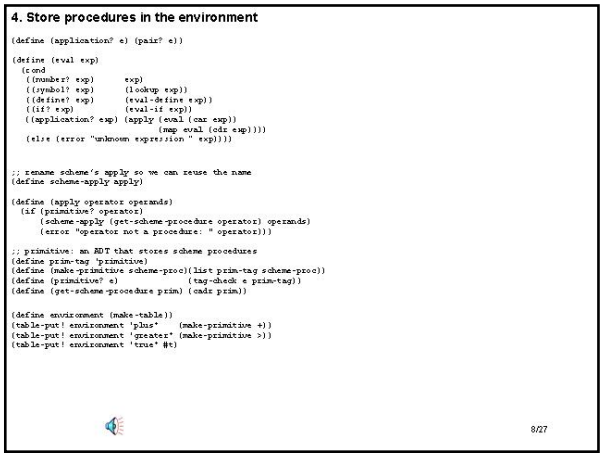
Slide 15.5.6

We know that recursively `eval` should do the right thing. If our expression is an application of a named procedure to some arguments, it will lookup the value of the name, which presumably is attached to some procedure in the environment. Similarly, mapping `eval` down the list of other expressions will return a list of values. Now, how do we do an **application**? Conceptually, **apply** says: check to see if the operator, the first subexpression, is a primitive, that is, one of the things built into my simple calculator (e.g. greater*, plus*). If it is, then get the corresponding primitive and apply it to the arguments, that is just do the normal Scheme thing to this primitive application.

Slide 15.5.7

So to use this, I simply need to create primitive operators, and I can do that with an abstract data type that stores away underlying procedures (basic things we are inheriting from Scheme) into a structure. We will make a primitive object by taking an underlying procedure, tagging it, so that I can check tags to find primitives and get out the actual procedure from the data abstraction.





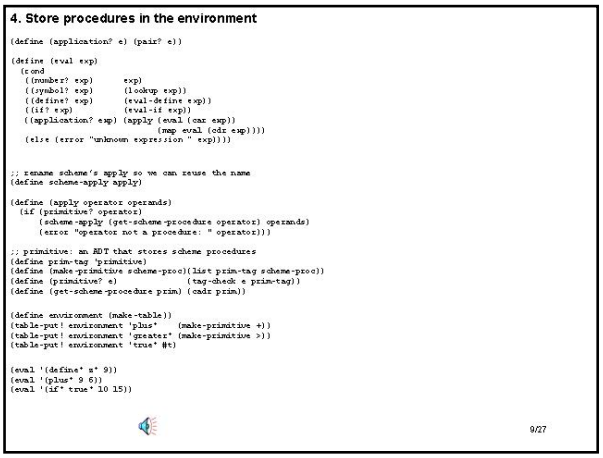
Slide 15.5.8

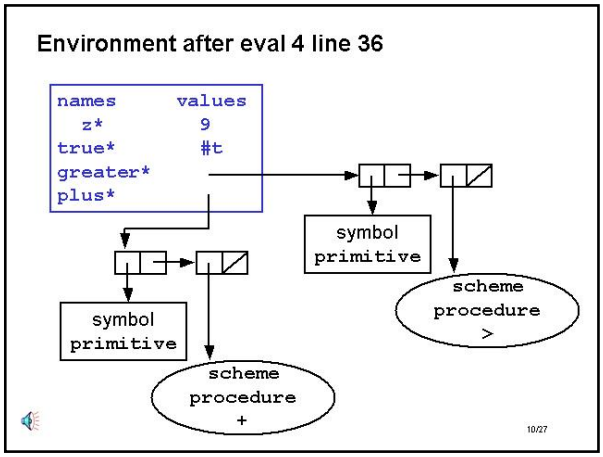
Remember that we have an environment, a little table abstraction that contains bindings for things, so initially, before I start using my evaluator, I can establish in that environment some built in primitives. For example, I can put in a built in primitive for `plus*` and one for `greater*`, and one for the symbol `true*`, which is bound to the machine representation for the Boolean value of true.

This should start looking familiar to you. Note that we are setting up an environment that contains bindings of built in names to built in primitive procedures. We also have for those procedures a way of tagging them to tell that they are procedures we have created. Then our evaluator is going to be able to use those built in primitives to execute the application of a procedure to some arguments.

Slide 15.5.9

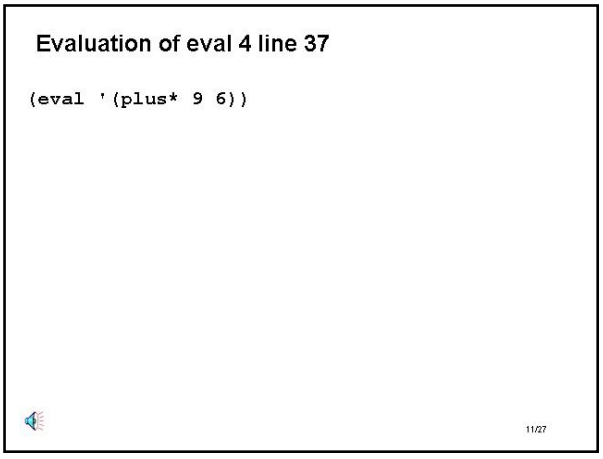
Okay, let's check it out. Let's evaluate the three shown expressions in order in our little evaluator. This will check both application and special forms in our new evaluator.

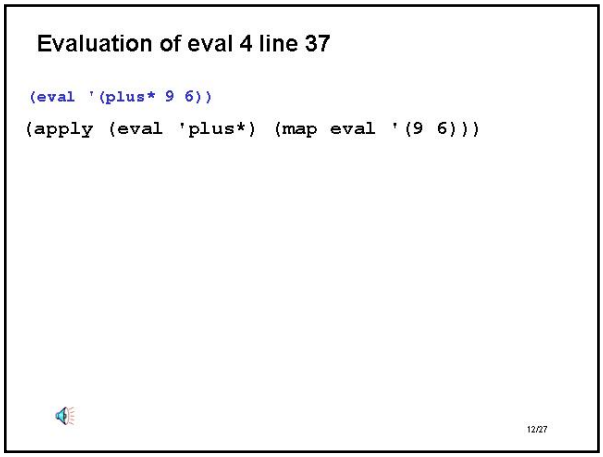




Slide 15.5.10
First, what does our environment look like? It is just an abstract data type of a table, containing names and values. This will include a binding of `z*` to `9` which we know comes from evaluating the `define*` expression. We also have a binding for `true*`, since that was one of the built in definitions. We also established bindings for names to representations of some primitive procedures.

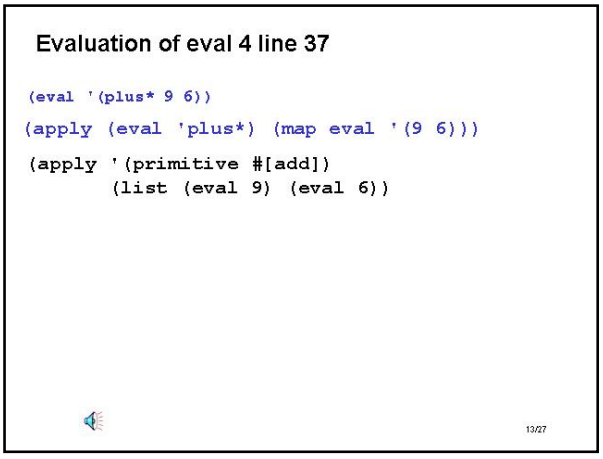
Slide 15.5.11
So let's step through the stages of evaluation of a simple expression. As before, keep the code handy to follow along. We start with the evaluation of the tree structure associated with `(plus* 9 6)`, the `'` is just use to remind us that this is tree structure along which we are walking. `Eval` steps through a big dispatch checking the type of this expression against each of its cases. Having decided it is not one of the explicit things about which it knows, it checks that it is a compound procedure, and since this is, `eval` assumes we have an application.

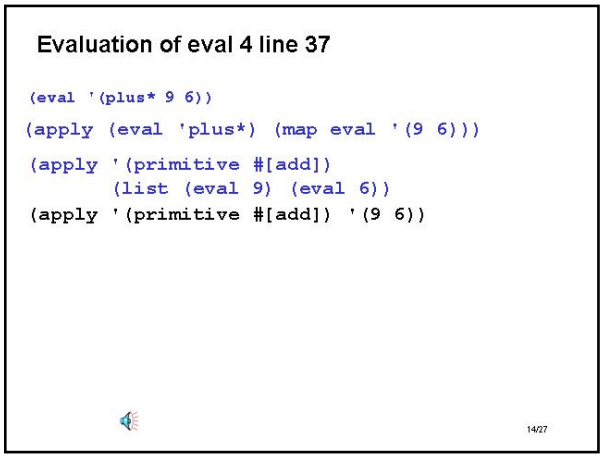




Slide 15.5.12
So `eval` unwinds this into an application of whatever we get by evaluating the first part of the tree to the list of the evaluation of the remaining parts of the tree. Notice how we are just walking through the tree, grabbing the pieces, and recursively evaluating each one in turn.

Slide 15.5.13
So what happens with the `eval` of the symbol `plus*`? Again, `eval` walks through each of its cases, checking the type of this expression against those cases. It decides this is a symbol, so it just does a look up in the environment. This just returns the representation associated with that name, in this case a tagged list, as shown. We now have the value of the operator. We can move on to mapping `eval` down the list of other pieces, which we know turns into a list of applying `eval` to each piece in turn.





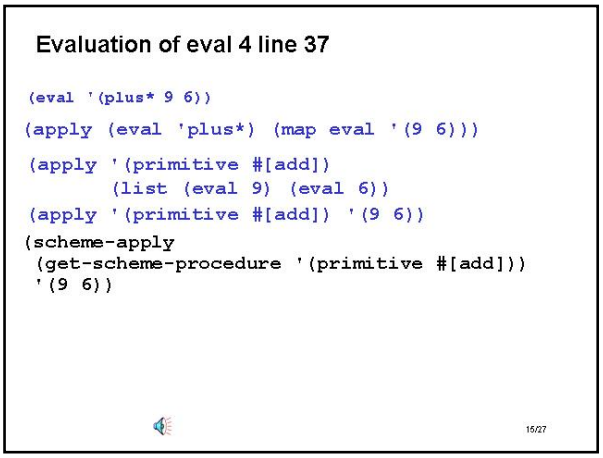
Slide 15.5.14

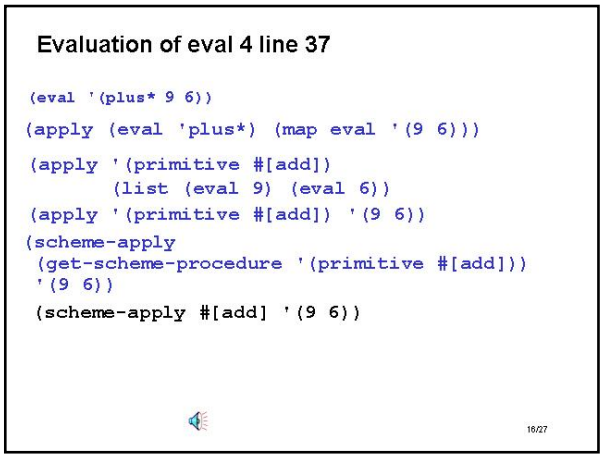
And of course we have seen what this does. For each expression, `eval` checks the type of the expression, decides it is a number, and just returns that expression as the value. This results in a list of the values 9 and 6.

Notice the form we now have: an application of a representation of a primitive procedure to a list of values. So we can now do the apply.

Slide 15.5.15

`Apply` first checks the argument to make sure it is a primitive. Since this one does have the right tag, it can reduce this to the underlying Scheme application of the procedure associated with addition to the values. And that just reduces to ...

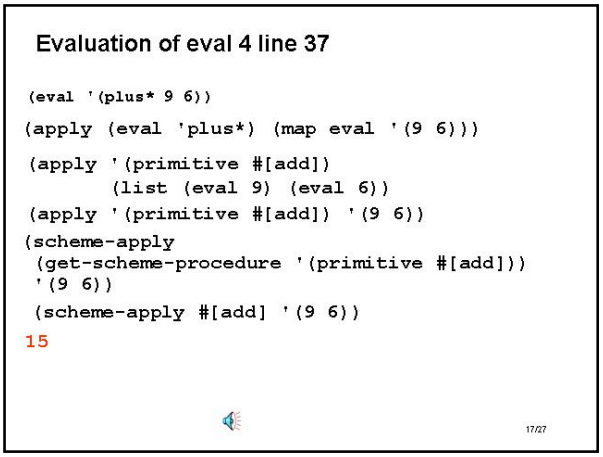


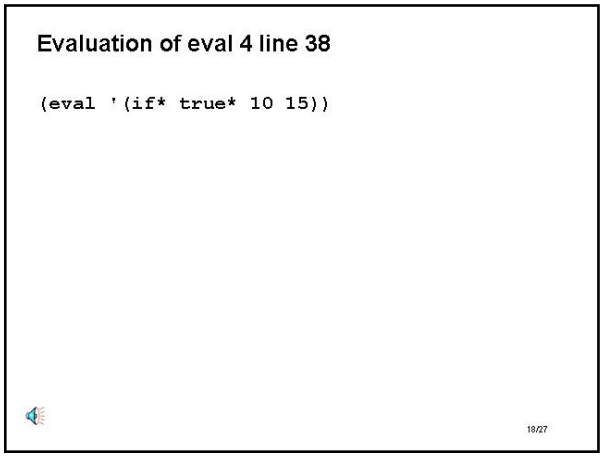


Slide 15.5.16

...this case. We just have a Scheme application of one of its primitive procedures to a list of values. Notice what we have done. We have reduced evaluation of an expression to an application of a procedure to a set of values, where we have recursively evaluated each subexpression to get that procedure and list of arguments. If the arguments were themselves combinations, we would have done the same process recursively on each of them, continually unwinding the evaluation down to an application, in this case of addition to some values ...

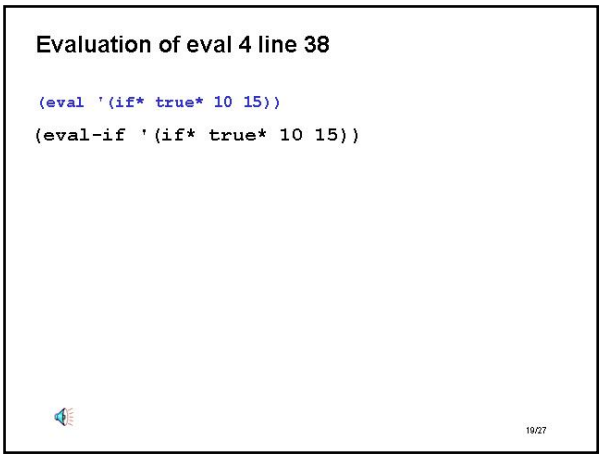
Slide 15.5.17
... returning the value we expect. The key thing to see here is the evolution of evaluation of a combination reducing to application of a procedure to a set of values.

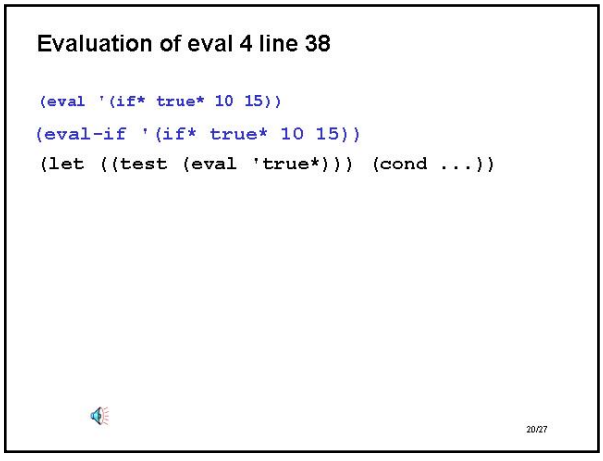




Slide 15.5.18
That example demonstrates that our normal procedure applications now work. Let's make sure that special forms still work, by considering the example of an `if*` expression.
Of course we get the behavior we expect. Given the tree structure associated with this expression, `eval` checks each case in its set of possibilities, checking the type of expression until it deduces that this is an `if*` expression. In this case, it dispatches off to a procedure designed to handle such expressions. Note that it reaches this case before it gets to normal applications.

Slide 15.5.19
That reduces to using the special procedure `eval-if` on this expression, and check the code to remind yourself what `eval-if` does.

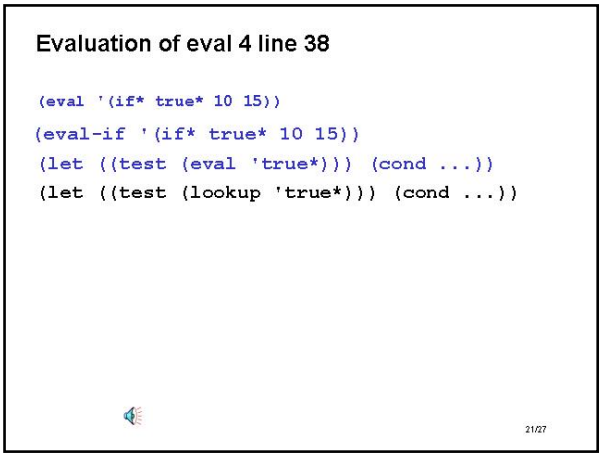


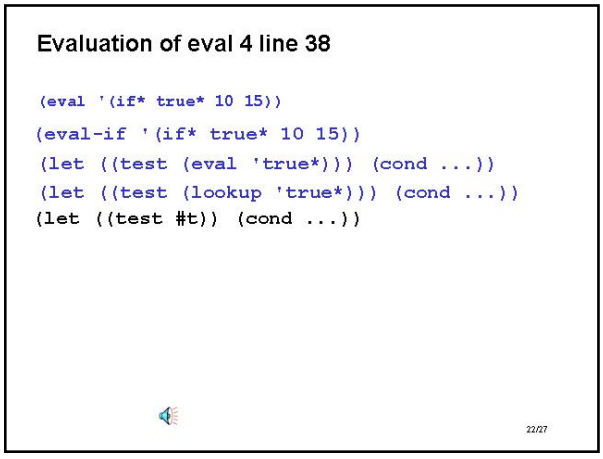


Slide 15.5.20
`Eval-if` first evaluates the predicate. It takes the expression, walks down the tree structure, pulls off the first subexpression, and evaluates it. We will temporarily give the result the name test. Key point is that we only evaluate this first subexpression, before we look at anything else.

Slide 15.5.21

`Eval` on this expression again walks through its cases, decides this is a symbol, and does a look up on it. It goes into the environment, finds the binding of this symbol in that table abstraction, and returns the binding, in this case the underlying representation for the Boolean value of true.



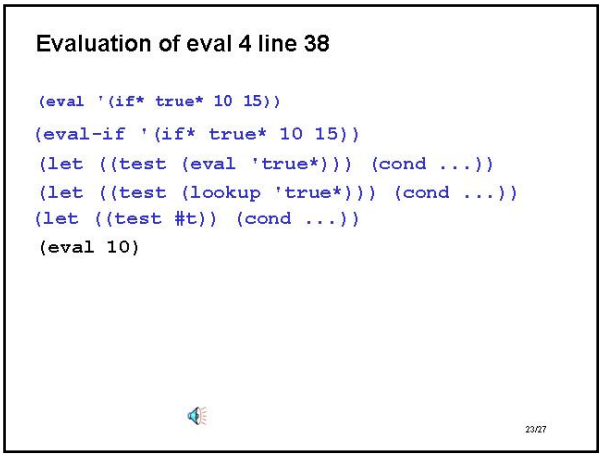


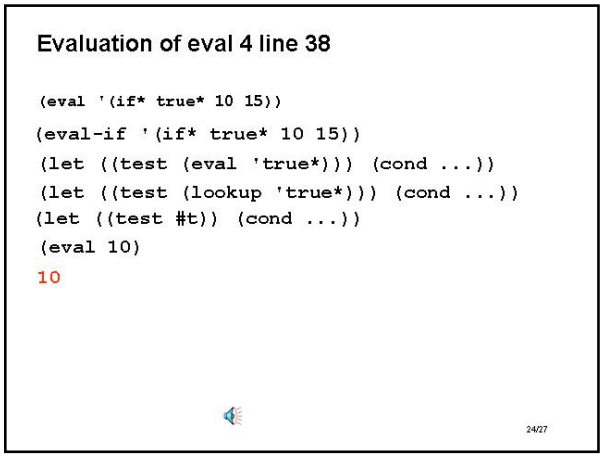
Slide 15.5.22

Having now obtained the value for this lookup, we can now proceed to the conditional expression. Thus, based on the value, it walks through the rest of the tree structure representing the expression, and grabs the appropriate piece.

Slide 15.5.23

and this reduces to simply evaluating this piece of the expression tree, and will return the value of this expression as the value of the overall `if*` expression. Because this is a number, this is self-evaluating and returns ...



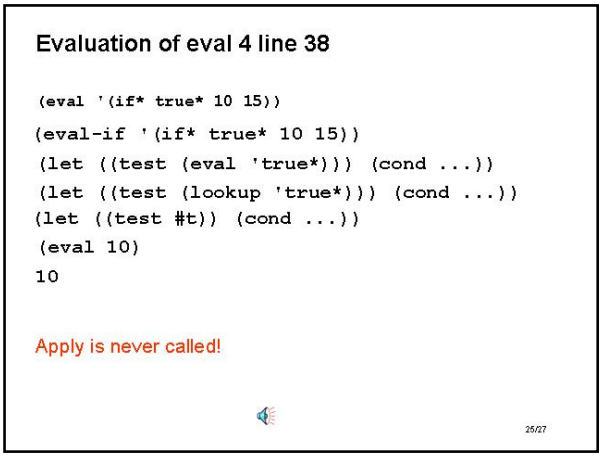


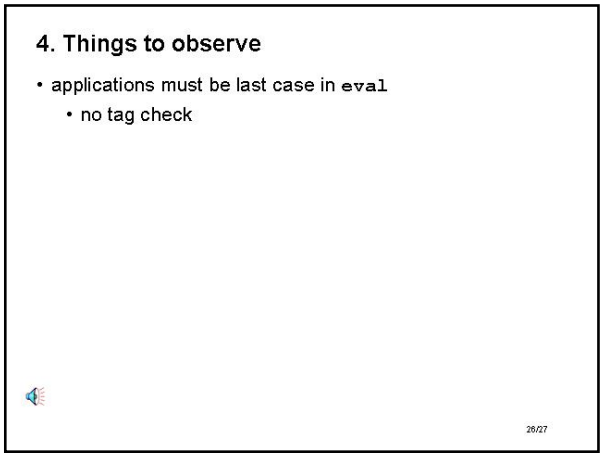
Slide 15.5.24

...just the expression itself.
Key thing to notice is how `eval` of an `if*` has walked through the tree structure representing the expression, in a particular order, and it has reduced the evaluation of that expression to the evaluation of one of the subexpressions. Also notice how we did not walk through all of the tree evaluating all of the pieces. We only evaluated "on demand", which is very different than an application.

Slide 15.5.25

In fact, notice that `apply` is **never** called here. We only do evaluation of pieces of the expression. Of course if one of those pieces had been an application we would have used `apply` but in terms of the parts of `if*` we never used `apply`. `If*` is a special form that handles evaluation of subpieces in a particular order, but never relies on an application of a procedure since it is not a procedure application.





Slide 15.5.26

Once more, let's step back from our example and extract some key observations. We have now extended our evaluator to include applications, which means we can create other kinds of expressions. But to do this, we must have applications be the last thing considered as an option in `eval`. We are not checking tags for applications, we are simply relying on the fact that is something is not one of the known special forms, it must be an application.

Also notice the form of `eval`, in which we first check for primitives, then for special forms, then for applications.

Slide 15.5.27

And as we just said, we never used `apply` in the last example. Applications evaluate all of their arguments before proceeding. Special forms handle things in a different order, exactly to control when arguments are evaluated.

So now we have seen a basic way of structuring an evaluator. We have primitive expressions, primitive means of combination, primitive conditionals, and primitive means of application, which enable creation of other expressions. Next time, we will look at further extending our evaluator.