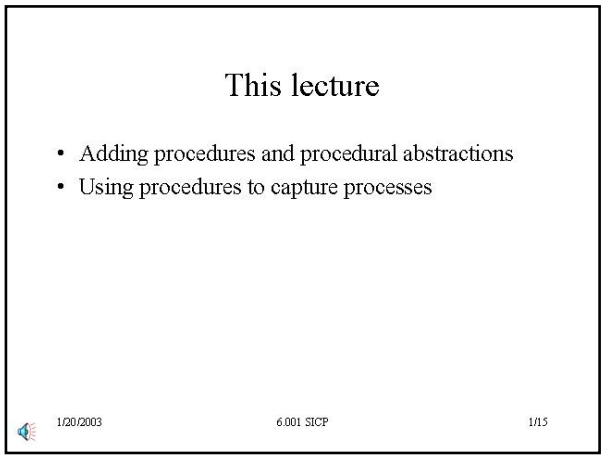
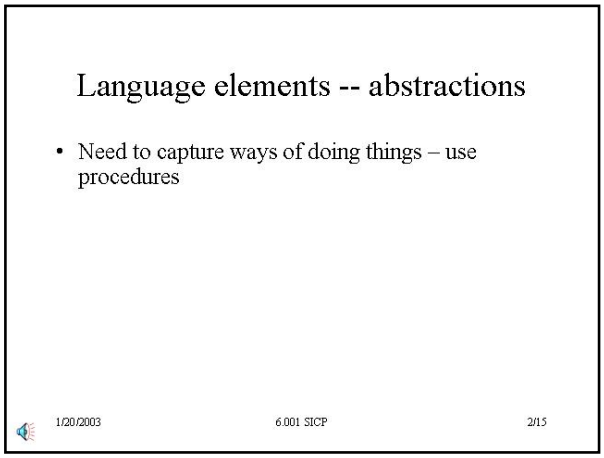
6.001 Notes: Section 2.1

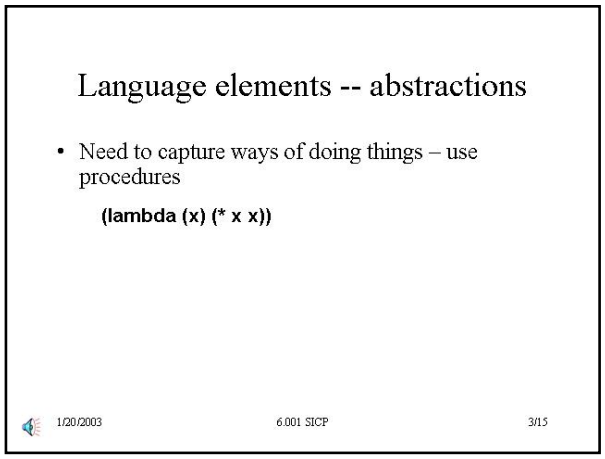
Slide 2.1.1
In the last lecture, we began looking at the programming language, Scheme, with the intent of learning how that language would provide a basis for describing procedures and processes, and thus for understanding computational metaphors for controlling complex processes. In this lecture, we look at how to create procedural abstractions in our language, and how to use those abstractions to describe and capture computational processes.

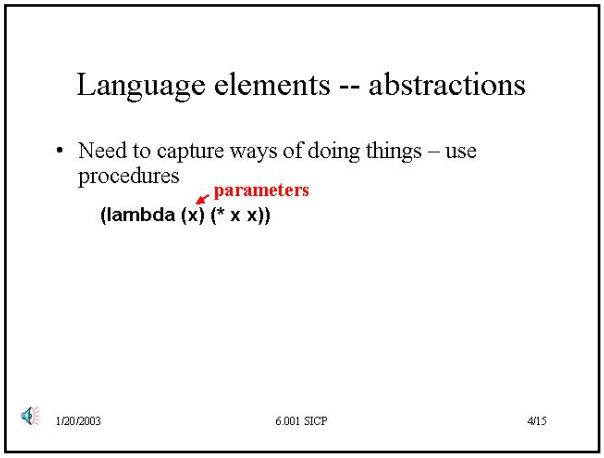




Slide 2.1.2
Well -- we've got primitives (numbers and built in procedures); we've got means of combination (ways of creating complex expressions); and we've got our first means of abstraction (a way of giving a name to something). But we are still stuck just writing out arithmetic expressions, as the only procedures we have are the built in ones. We need a way to capture our own processes in our own procedures. So we need another kind of abstraction -- we need a way of capturing particular processes in our own procedures, and that's what we turn to next.

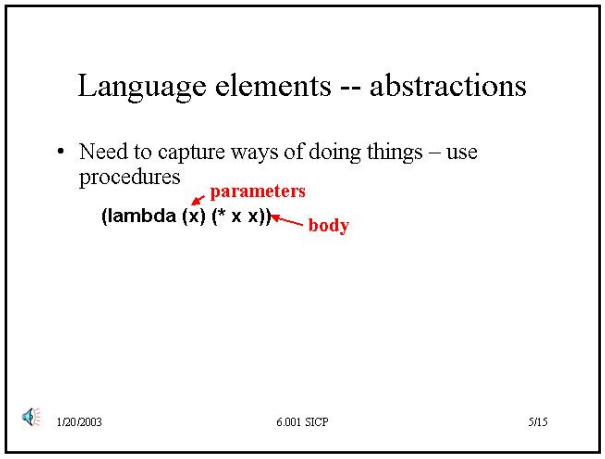
Slide 2.1.3
In Scheme, we have a particular expression for capturing a procedure. It's called a **lambda** expression. It has the form shown, an open parenthesis, the keyword lambda, followed by some number of symbols enclosed within parentheses, followed by one or more legal expressions, followed by a close parenthesis.

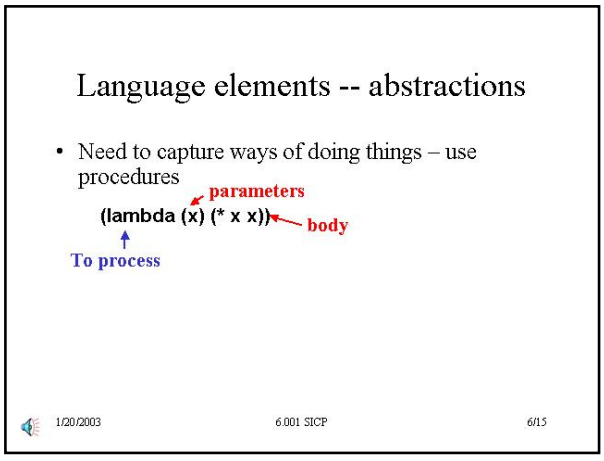




Slide 2.1.4
The keyword lambda identifies this expression as a particular special form. The set of symbols immediately following the lambda are called the formal parameters of the lambda, in this case, there is just one parameter, **x**.

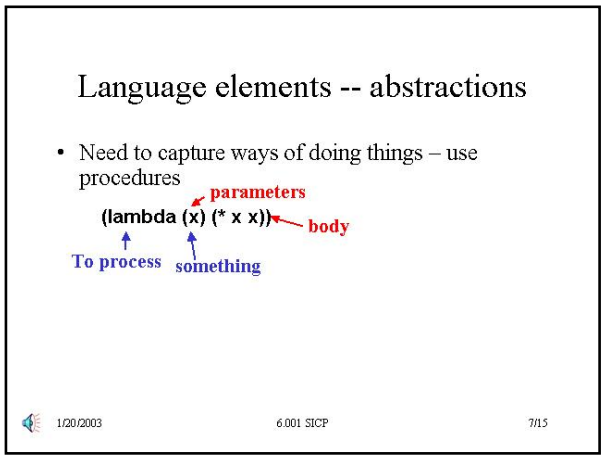
Slide 2.1.5
The subsequent expression we refer to as the body of the procedure. This is the particular pattern we are going to use to capture a process.

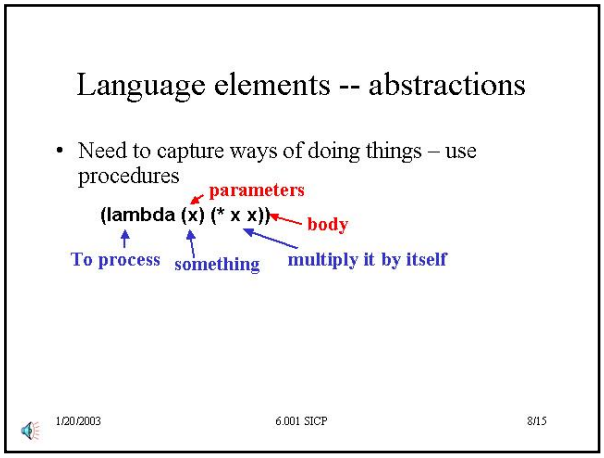




Slide 2.1.6
The way to think about this lambda expression is that it is going to capture a common pattern of computation in a procedure -- it will actually build a procedure for us.
The way to read the expression is: To process ...

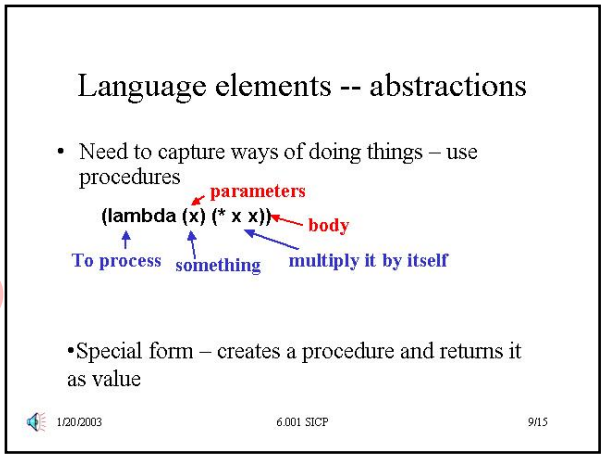
Slide 2.1.7
... something ...

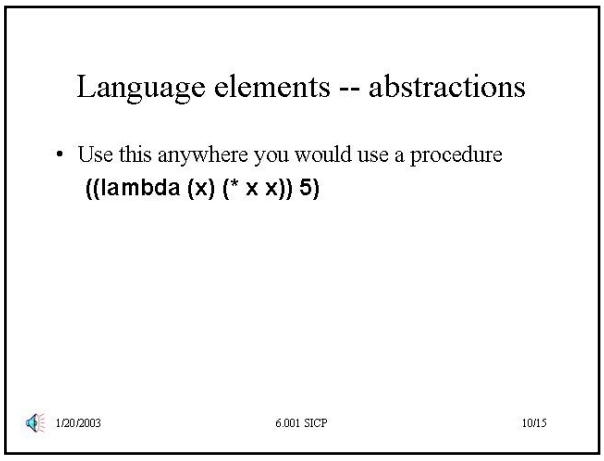




Slide 2.1.8
... multiply it by itself, and return that value.

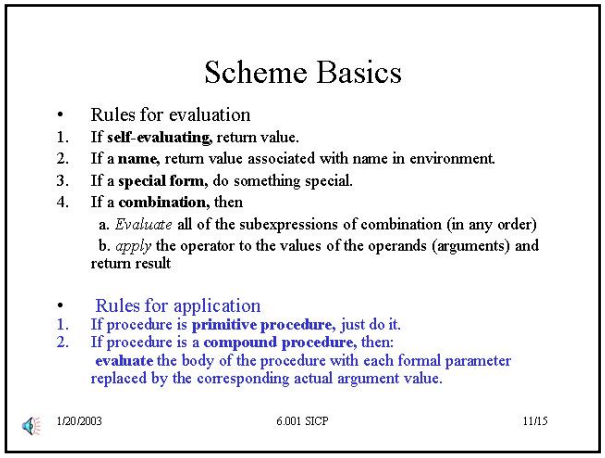
Slide 2.1.9
So in fact this particular lambda expression captures the process of "squaring". It says, if you give me a value for **x** I will return the value of multiplying that value by itself. In a second we will see how this happens.
Notice that lambda expressions must be special forms. The normal rules for evaluating a combination do not apply here. Instead, the value returned by evaluating a lambda expression is the actual procedure that it captures. Contained within that procedure will be a set of formal parameters, and a body that captures the common pattern of the process, as a function of those formal parameters.

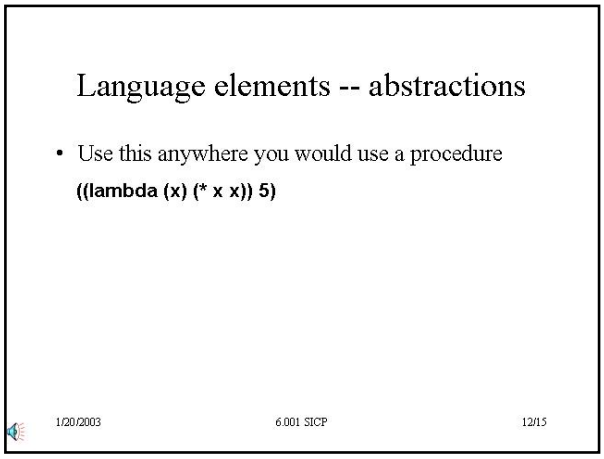




Slide 2.1.10
Now, where can we use such a procedure? Basically anywhere in our earlier expressions that we could use a built-in procedure, which for now means as the first element in a combination.
For example, here is a compound expression, with two subexpressions. What is the value or meaning associated with it? The value of the first subexpression we just saw was a procedure. The value of the second subexpression is just the number 5. Now we have something similar to our earlier cases, a procedure applied to a value. The only difference is that here we have a procedure we built, rather than a pre-existing one.
We need to specify how such a procedure is applied to a set of arguments.

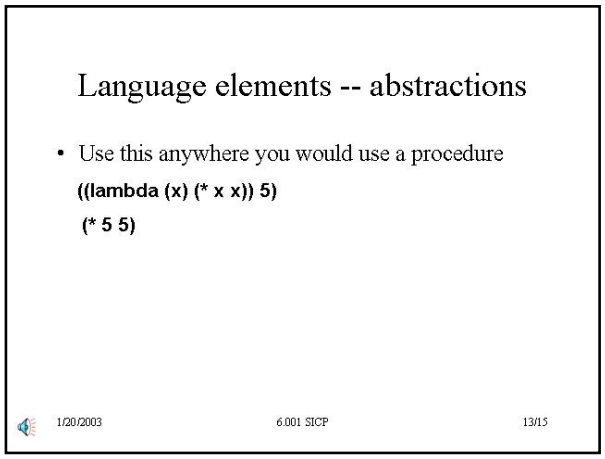
Slide 2.1.11
So here is a summary of our earlier rules for evaluating expressions. The only change is to amplify what it means to apply a procedure to a set of arguments. When the procedure was a built-in arithmetic operator, we just did the obvious thing. Now if the procedure is something built by evaluating a lambda expression, we have a new rule. We take the body of the procedure, substitute the value of the argument in place of the corresponding formal parameter, and then use the same rules to evaluate the resulting expression.

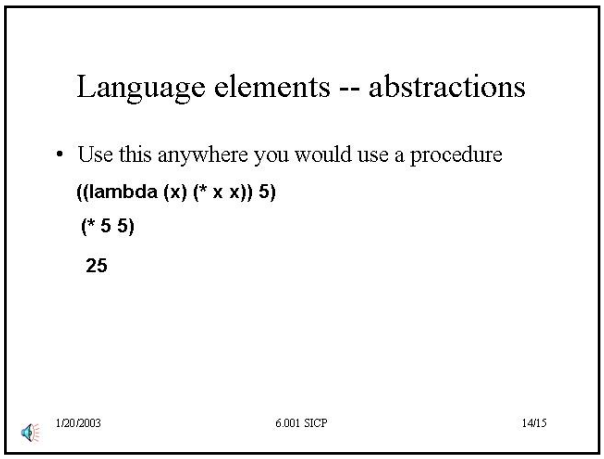




Slide 2.1.12
So let's go back to our example. Our rule says to replace the value of the second expression, **5**, everywhere in the body that we see the formal parameter, **x**. This then reduces the application of the lambda expression to a simpler expression ...

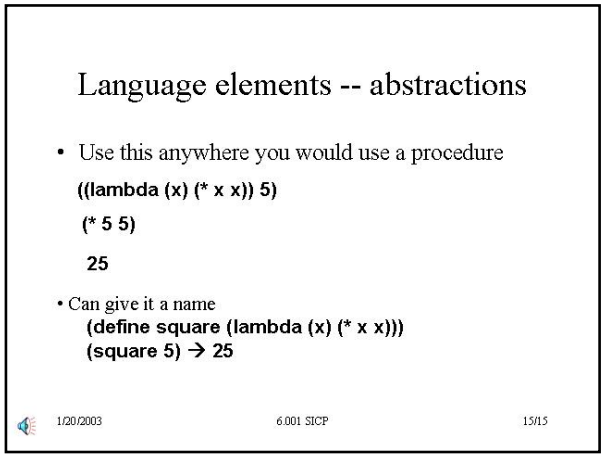
Slide 2.1.13
This reduces the application of a procedure to this simpler expression, and we can apply our rules again. The symbol ***** is just a name for the built-in multiplication operation, and **5** is just self-evaluating, so this all reduces to ...





Slide 2.1.14
... 25.
Thus we see that our rules now cover the evaluation of compound expressions that include the application of procedures created by lambda expressions. In particular, the rules tell us to substitute into the body of a procedure for the formal parameters, reducing to a new expression, and then apply the same set of rules all over again, until we reach a final answer.

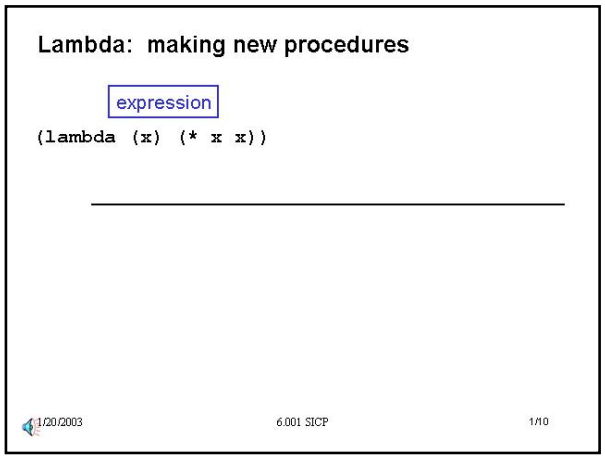
Slide 2.1.15



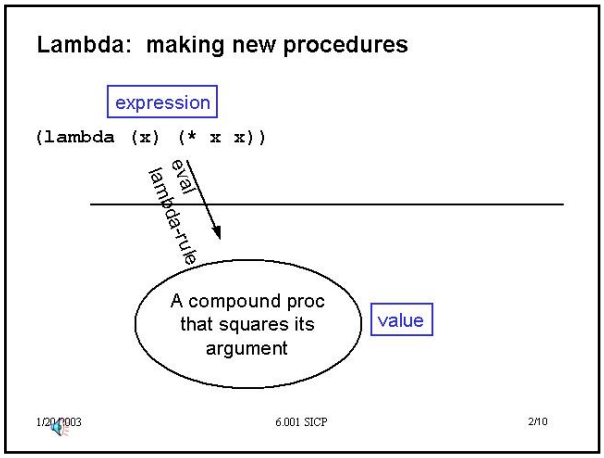
Thus, lambda gives us the ability to capture procedure abstractions -- patterns of computation in a single procedure. But we don't want to have to write out lambda expressions everywhere we need to use this procedure.
Instead, we can combine this procedural abstraction with our naming abstraction -- that is, we can use a define expression to give a name to a procedure. In this case, the name **square** will be paired with the value of the **lambda** expression, or quite literally with the actual procedure created by evaluating that **lambda**.
Then we can use the name **square** wherever we want the procedure, since its value is the actual procedure. If you follow the rules of evaluation for the last expression, you will see that we get a procedure applied to a number, and the substitution of the argument into the body of the procedure reduces to a simpler expression, just as we saw earlier.

6.001 Notes: Section 2.2

Slide 2.2.1



The second kind of special form we saw was a **lambda** expression, and **lambda** we said was our way of capturing processes inside procedures. **Lambda** is Greek for "procedure maker", and is our way of saying "Take this pattern, and capture in a way that will let me reuse it multiple times". In our two world view, if we type this expression into the computer and ask it to be evaluated, the computer uses the key word **lambda** to determine that this is a particular special form. It will then use the rule designed for this type of expression.

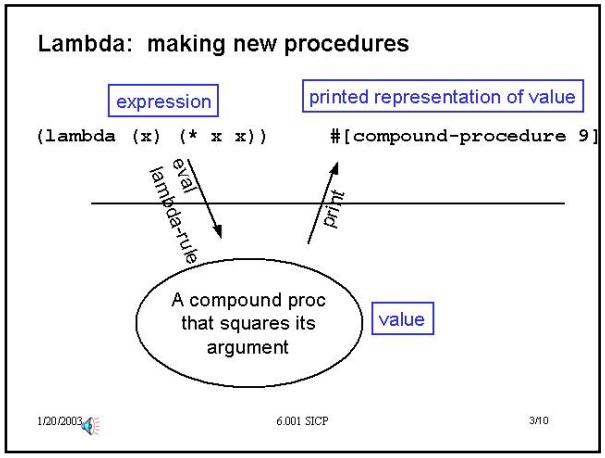


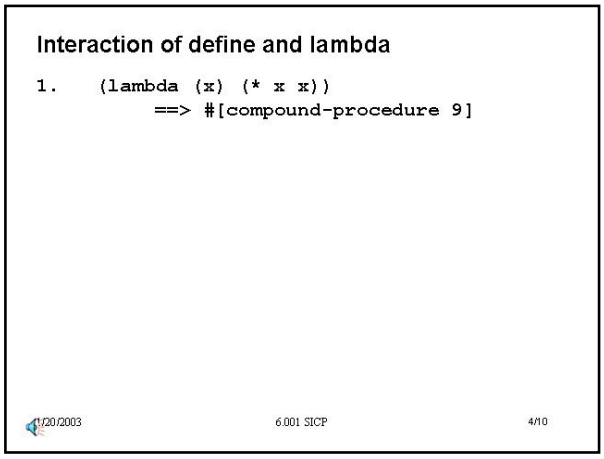
Slide 2.2.2

It is important to **stress** that the actual value created inside the machine is some representation for the procedure itself. This representation includes information about the parameters of the procedure, and the body of the procedure, that is, what pattern of computation is being captured, and what are the variables to be replaced in that pattern when we want to use this procedure. But it is the actual procedure representation that is associated as the value of the expression.

Slide 2.2.3

And what gets returned? Basically some representation like this, that identifies that this is a procedure created by evaluating a lambda, and an indication of where it lives inside the machine (i.e. how to get at the parameters and body of the procedure). It is important to stress that evaluating a lambda creates an actual procedure object inside the execution world, and that value is actually returned as the value of the expression.



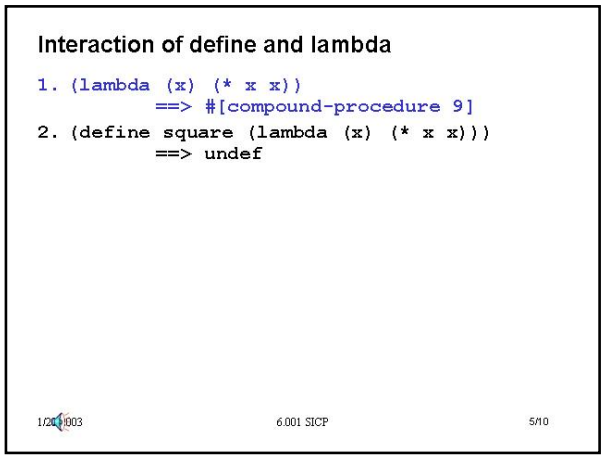


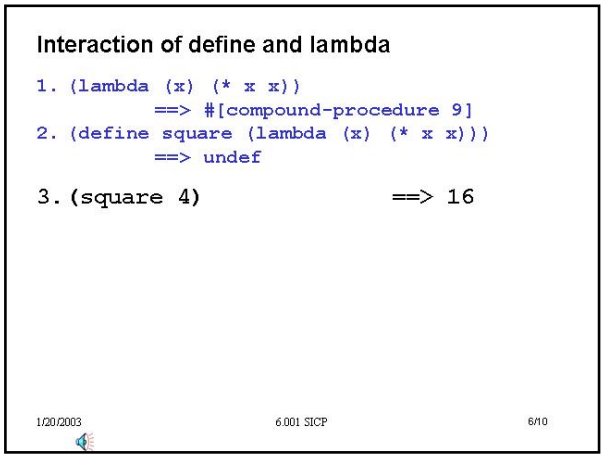
Slide 2.2.4

Now let's look at the interaction between creating procedures and giving them names. First of all, I can create a lambda expression, and the value returned by that evaluation is the procedure itself. Note that the procedure is not executed or run: there is no numeric value returned here as a square, rather the procedure itself is returned as the value.

Slide 2.2.5

If I actually want to use that procedure I need to refer to it, and the easiest way to do this is to give it a name. So I can define **square** to be the value returned by that lambda expression, which we just saw is a procedure. This then creates a pairing of the name **square** with the procedure that captures the pattern of multiplying a value by itself.





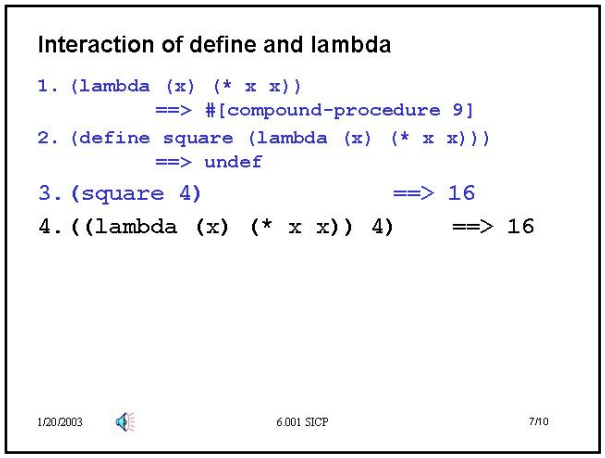
Slide 2.2.6

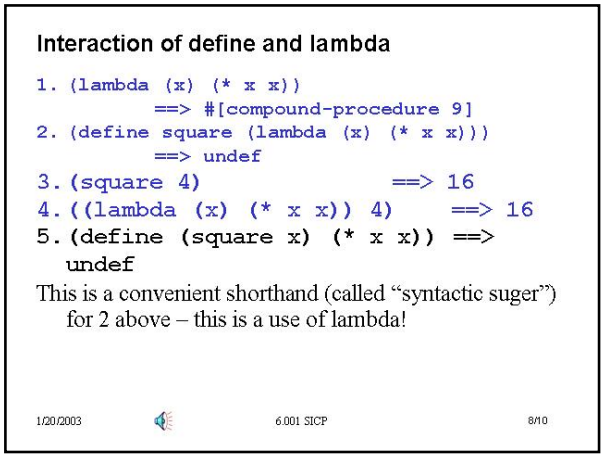
Having done that, I can now write any expression that uses the name **square** anyplace that the associated lambda expression would have been legitimate.

Note that the evaluation rules nicely cover this. This expression is a standard combination, so we just get the values of the sub-expressions. **Square** is a name, so we look up its value, getting back the procedure. This can then be applied to the value of the next sub-expression, using the standard rules for procedure application, namely substitute 4 for **x** everywhere in the body of the procedure, then evaluate that new expression using the same rules.

Slide 2.2.7

Of course, I could have used the full **lambda** expression in place of the name. In this case, evaluation of the first sub-expression will create the procedure, which is then applied to the value of the second sub-expression.



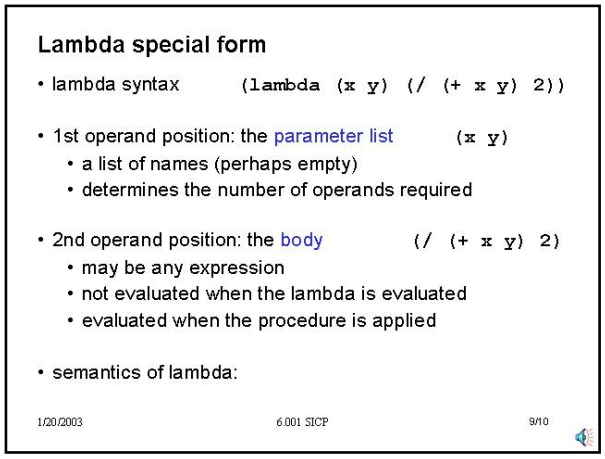


Slide 2.2.8

Because this action of creating a procedure and giving it a name is such a common activity, we have an alternative way of doing this. This last expression is really the same as the second one, but is simply written in a more convenient form (which we call "syntactic sugar" meaning that the meaning is the same but the form is sweeter). It is important to stress that in this last form, there is a hidden **lambda** that is evaluated to create the procedure, and then the **define** is used to give it a name.

Slide 2.2.9

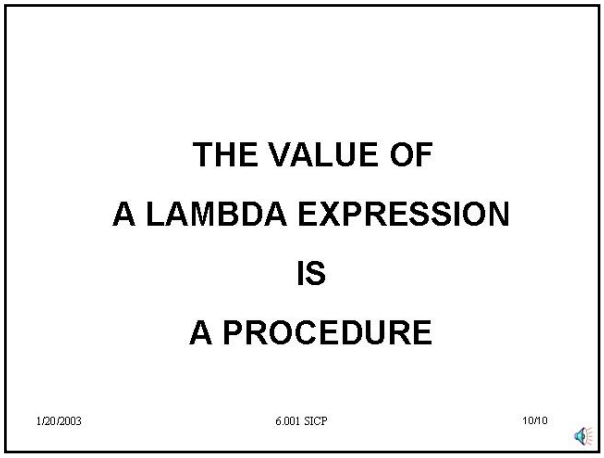
Let's look a little more carefully at this **lambda** special form. As we have seen, the syntax, as shown at the top, has three pieces.



There is the key word, **lambda**, which identifies the type of expression. The first operand position of this expression identifies the formal parameters of the procedure. This is a list (or a sequence of names enclosed in a set of parentheses). In this case, the procedure takes two arguments, which we will call **x** and **y**. Note that there can be an arbitrary number of arguments for a procedure, including zero (in which case we would use () as the parameter list), and that this component determines how many arguments must be passed to the procedure when it is applied.

The second operand position determines the body of the procedure, that is, the pattern of evaluation to be used. This may be any valid Scheme expression. Note that this expression is **NOT** evaluated when the procedure is created. It is simply stored away as a pattern. It is **ONLY** evaluated when the procedure is applied in some legal combination.

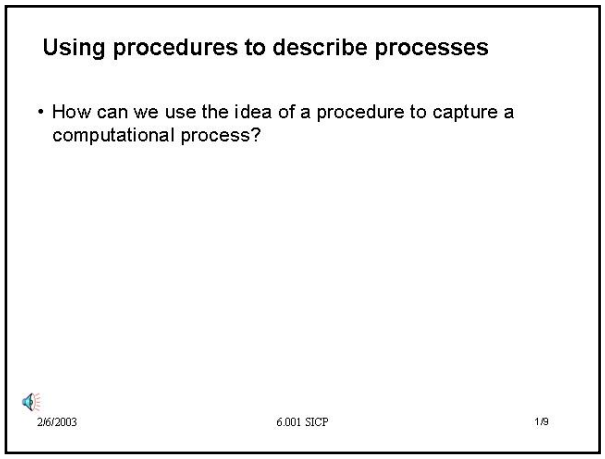
That determines the syntax of the **lambda**. The semantics of this kind of expression is very important, so we place it on a whole separate slide, which follows ...

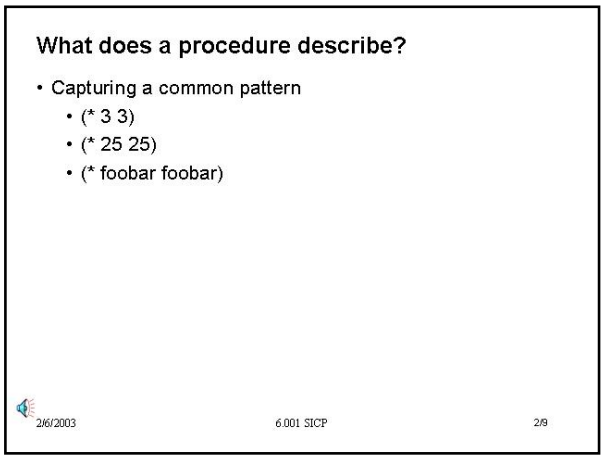


Slide 2.2.10
... and I feel like I should be shouting this out!!
The semantics says that the value of a **lambda** expression is a procedure object. It is some internal representation of the procedure that includes information about the formal parameters and the body, or pattern of computation.

6.001 Notes: Section 2.3

Slide 2.3.1
We have now seen most of the basic elements of Scheme. We will continue to add a few more special forms, and introduce some additional built in procedures, but we now have enough elements of the language to start reasoning about processes, and especially to use procedures to describe computational processes.
So let's look at some examples of describing processes in procedures.

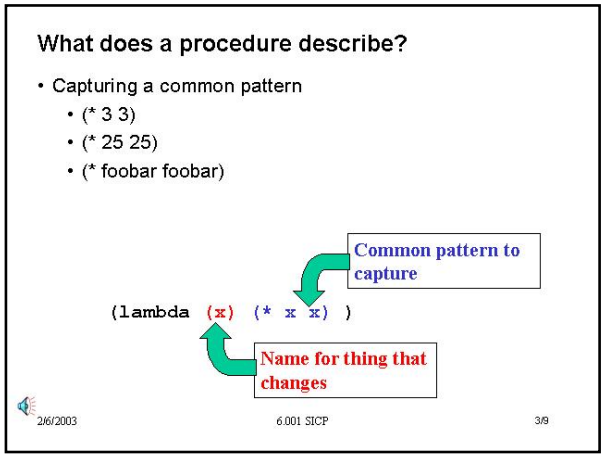


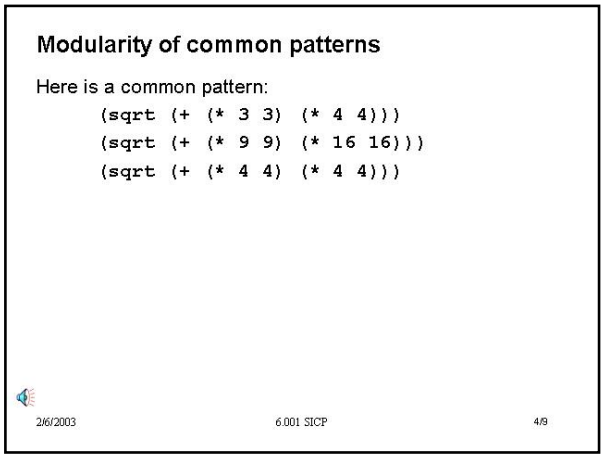


Slide 2.3.2
First, what does a procedure describe?
One useful way of thinking about this is as a means of generalizing a common pattern of operations. For example, consider the three expressions shown here. The first two are straightforward. The third is a bit more general, since `foobar` is presumably a name for some numerical value.
However, each of these is basically just a specific instantiation of a process: the process of multiplying a value by itself, or the process of "squaring".

Slide 2.3.3

So we can capture this by giving a name to the part of the pattern that changes with each instantiation; identifying that name as a formal parameter; and then capturing that pattern as the body of a `lambda` expression, together with the set of formal parameters, all within a `lambda` expression.



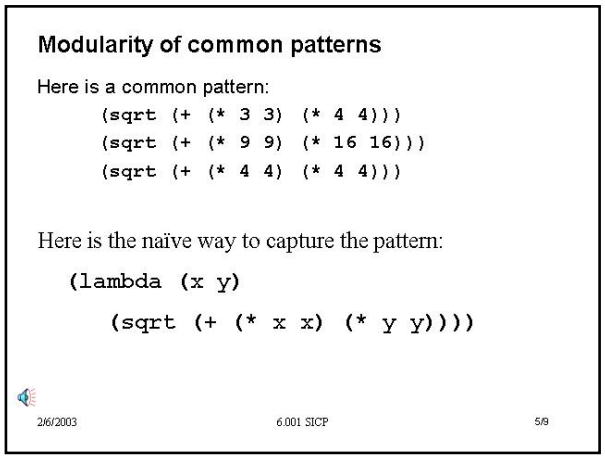


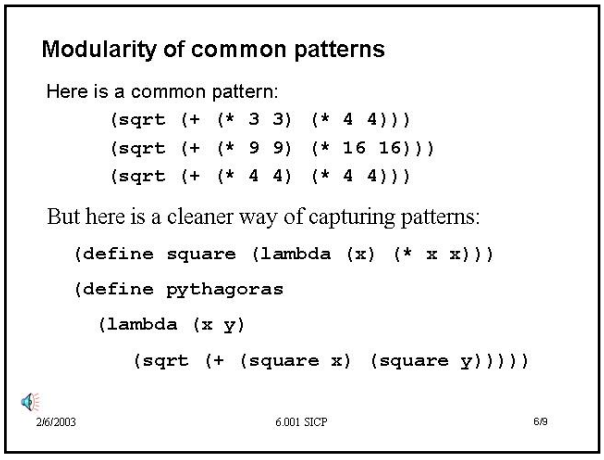
Slide 2.3.4

Now lets consider a more complex pattern, as shown here.

Slide 2.3.5

In this case, there are two things that vary, so we will need two parameters to capture this. Otherwise we could just do the same thing we did last time, and replicate the pattern, with the parameters in place of the things that change, as shown.

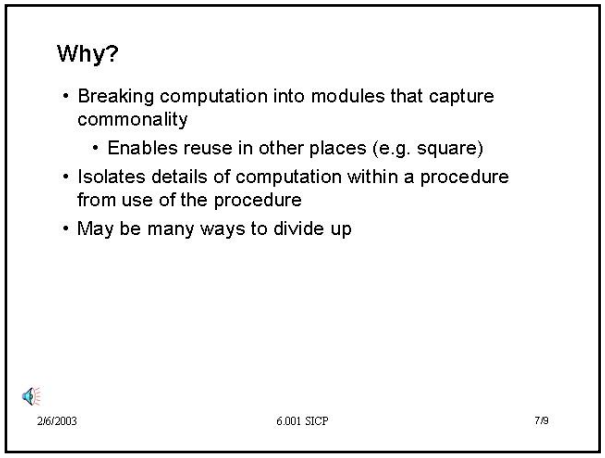




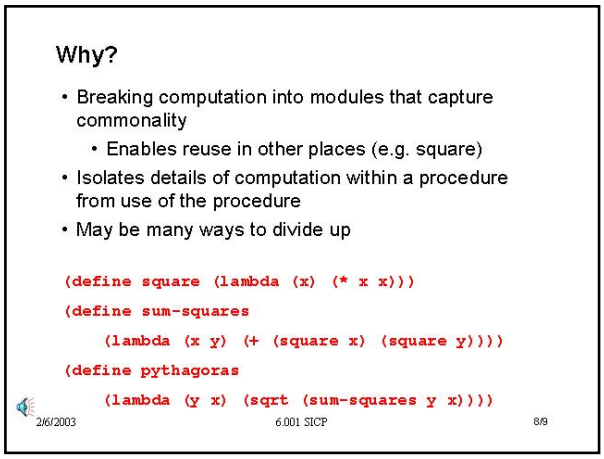
Slide 2.3.6

But a better way to capture this pattern is to realize that there are really two things going on. One is the sub-pattern of squaring things. The other is the use of the results of two different squaring operations within the larger pattern.
So we could capture each of those patterns within its own procedure, with its own parameters and body.
Note that in doing this, we are relying on a property of a combination, namely that a combination involving a named procedure is equivalent to the pattern captured by the procedure, with values substituted for the formal parameters.

Slide 2.3.7



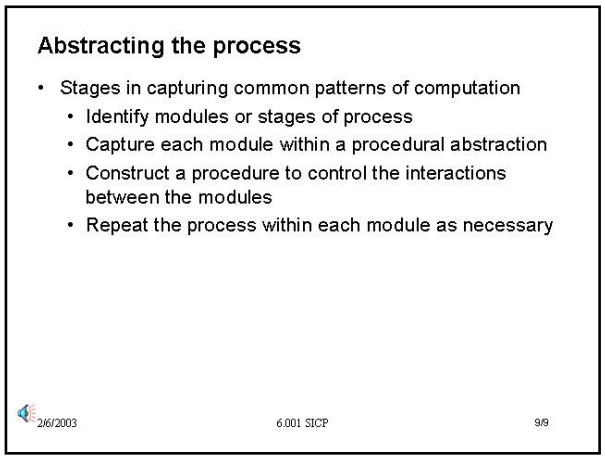
Why is this a better way of capturing the pattern?
The primary reason is that by breaking the pattern up into smaller modules, we isolate pieces of the computation in separate abstractions. And these modules can then be reused by other computations. In particular, the idea of `square` is likely to be of value elsewhere, so it makes sense to capture that in its own procedure, and then use within this larger one.
As well, by doing this, we create code that is easier to read, as we use a simple name to capture the idea of `square`, and suppress the unnecessary details. By doing so, we isolate out the **use** of a procedure from the **details** of its actual implementation, a trick to which we will return later in the term.
Of course, there may be many different ways of modularizing a computational pattern.



Slide 2.3.8

Here is a finer scale modularization of the pattern into procedures. Note how each procedure uses the previous one within its body, using that idea of abstraction to separate the use of a procedure from its details.

Slide 2.3.9

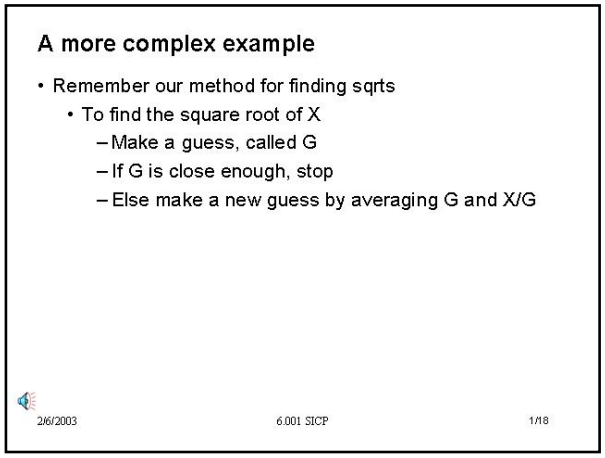


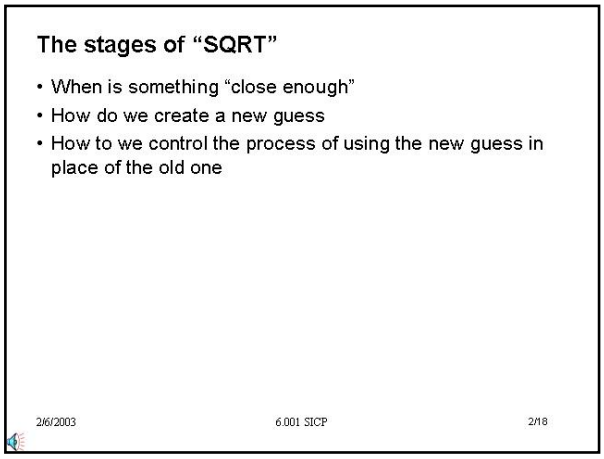
Now, let's step away from the specifics of this example, and talk about the process we just used.
In essence, we did several things: we identified modules or parts of the computational process, which we could usefully isolate; we then captured each of those within their own procedural abstraction; and finally we created a procedure to control the interactions between the individual modules. Of course, we could apply this process within each of the modules, in a recursive fashion.
Our goal now is to see how we can use this general approach to capture computational processes in procedures.

6.001 Notes: Section 2.4

Slide 2.4.1

Let's take another look at this idea of capturing a procedural description with our language for describing processes, namely `lambda`s. Remember our description of the process of finding square roots, from the first lecture, shown here.





Slide 2.4.2

Our first step in building code to compute "square roots" is to determine some good modules, or stages, within this process. Here, we can see several.

There is the idea of measuring whether our guess is good enough that we can stop and return an answer.

There is the idea of creating a new guess if we are not close enough.

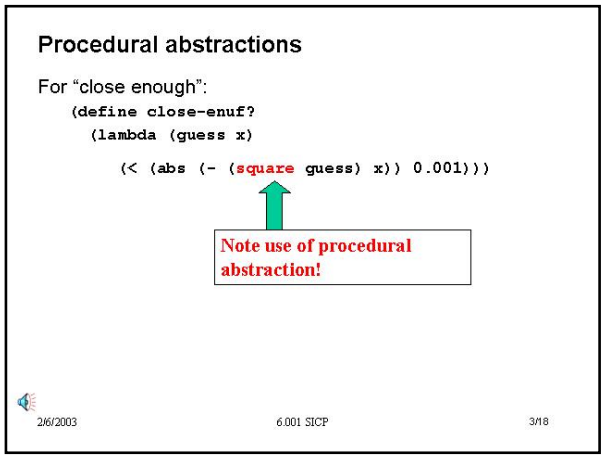
And there will need to be a way of controlling the process, in which we use our new guess as if it were the original one, and continue the process.

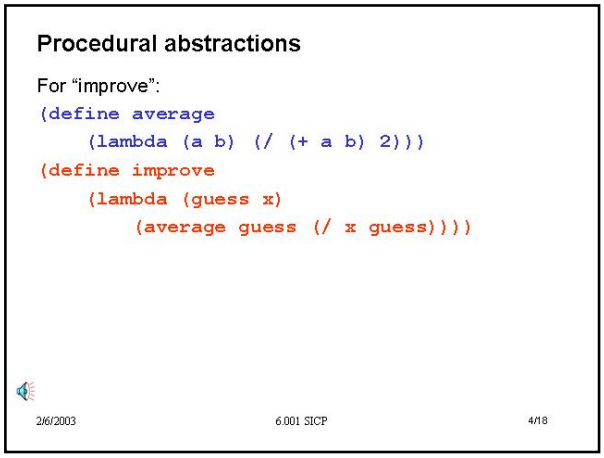
So let's build each of these abstractions, using our idea of capturing common patterns within `lambda` expressions.

Slide 2.4.3

Here is a rather naive way of deciding if a **guess** is *`close enough`*. We take the **guess**, and square it. If the **guess** is good, then that should give us a value close to the number whose square root we are seeking. Here `abs` is a built in procedure that returns the absolute value of its argument, and we simply test to see if that absolute difference is small.

Note how we are already using procedural abstractions: we have assumed that `square` is an abstraction for the process of squaring numbers.





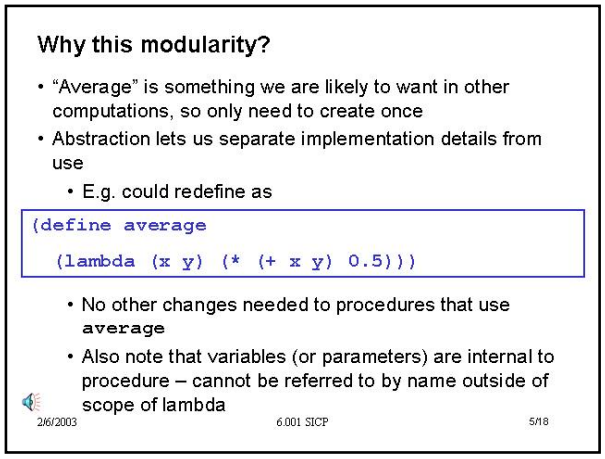
Slide 2.4.4

For improving the **guess**, we can just use the same approach we did with **pythagoras**: we capture the idea of **average** and we use it to improve a guess as described by the process.

Slide 2.4.5

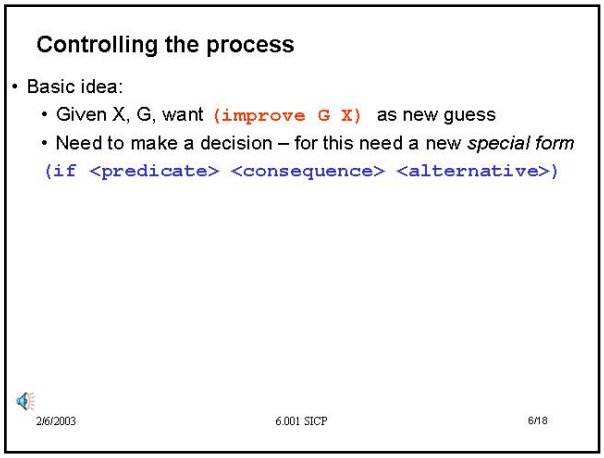
As before, we can see that `average` is likely to be a process that we will want to use in other places, so creating an abstraction allows us to avoid replicating the code in those places.

Moreover, we stress that by building this abstraction, we seal off the details of the implementation from the actual use of the abstraction. For example, we could decide to change the implementation of `average`, such as that shown here.



Doing so does not require us to make any changes to procedures that use `average` however, since those simply refer to the procedure, not the internal specifics.

Also note that the names of the parameters are internal to the `lambda` expression: we cannot refer to them outside the scope of the `lambda`. Here, we have changed the names of the parameters, but this does not affect those procedures that use `average`.



Slide 2.4.6

The last step is to decide how to integrate these pieces together into a process that controls the steps of the computation. The basic idea is already captured in the process description: given a number and a guess, we want to use `improve` to derive a new guess.

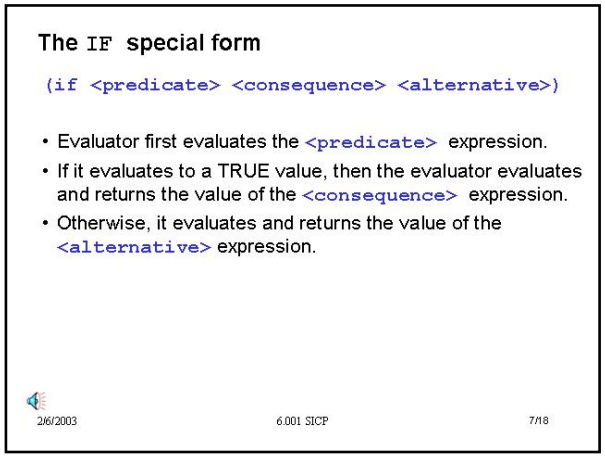
But now we have to make a decision: is the guess good enough, in which case we can stop? Or should we continue the process? In order to make such a decision, we need a new **special form**, called an `if` expression, which has three sub-expressions: a predicate, a consequence, and an alternative.

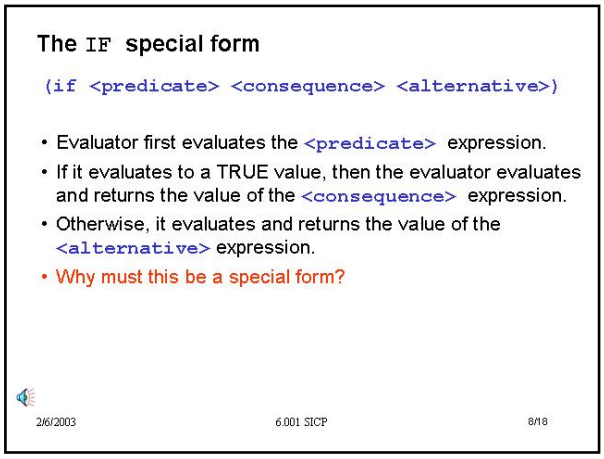
Slide 2.4.7

Here is how an `if` expression is evaluated. First, the evaluator uses its rules (such as those we have described) to determine the value of the **predicate** expression.

If that value is **true**, then the evaluator uses its rules to evaluate the **consequence** expression, and returns that value as the value of the entire `if` expression.

On the other hand, if that value is not **true**, then the evaluator uses its rules to evaluate the **alternative** expression, and returns that value as the value of the entire `if` expression.





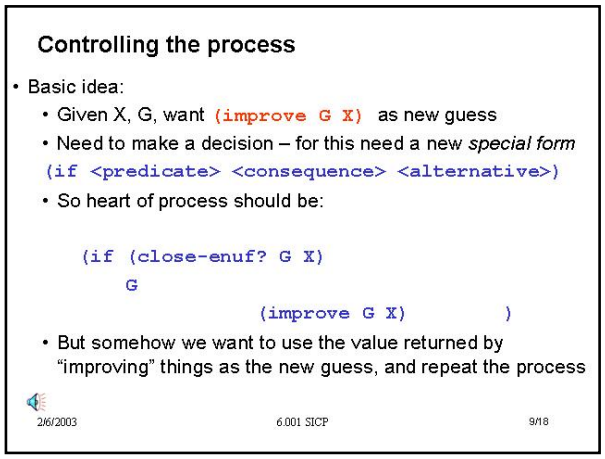
Slide 2.4.8

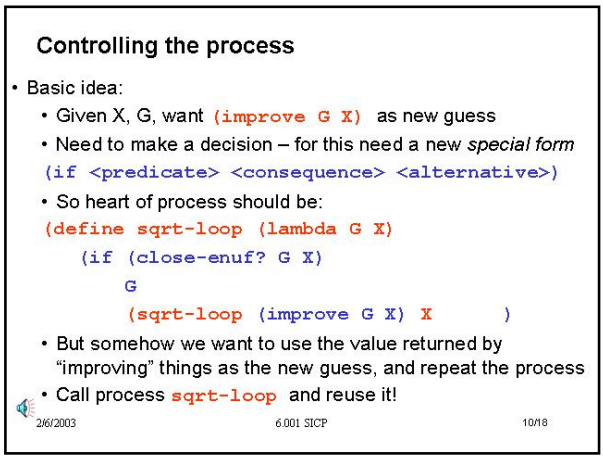
So why do we say that this is a special form, rather than just a procedure? We'll let you think about this, but after the next lecture you should be able to answer this question. For now, we will simply accept that `if` expressions are evaluated in this particular manner.

Slide 2.4.9

So back to our process. We know from our description that the heart of the process should look something like this. We check to see if we are close enough, and if so, then we just return the value of the guess. Our `if` expression will handle that for us.

If we are not close enough, we want to improve our guess, using the **improve** procedural abstraction we built. But somehow we want to then use that value as a new guess, and repeat the process.





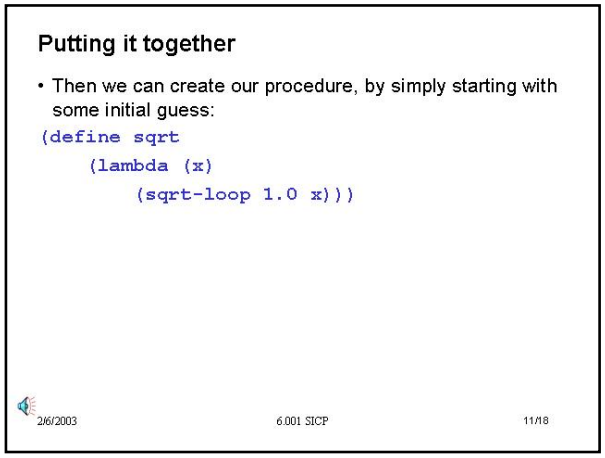
Slide 2.4.10

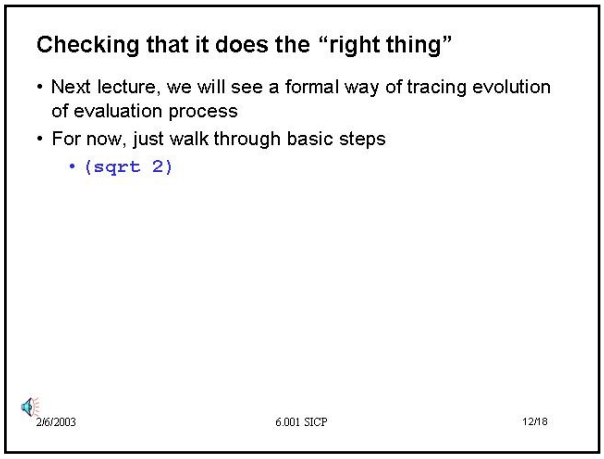
How do we do that?

Well, let's call this overall process, of repeated improving a guess until we are close enough, `sqrt-loop`. Then if we are not close enough, we get an improved guess, and simply do this again!

Slide 2.4.11

Finally, we can assemble our `sqrt` procedure, we just use this repeated process of improving a guess: all we need to do is get it started with some initial guess.





Slide 2.4.12

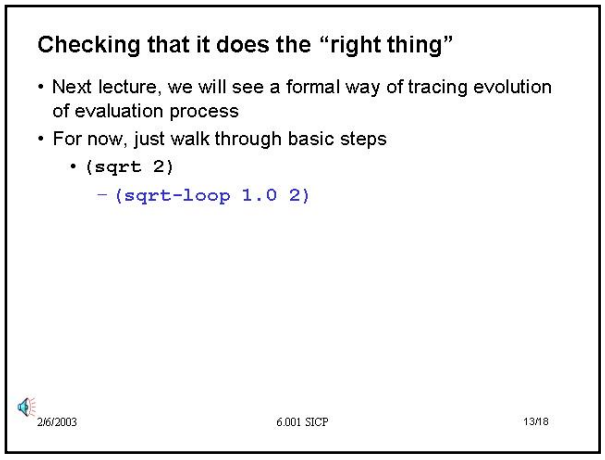
This may look a bit unusual. We have a procedure that refers to itself within its body, in a recursive fashion. Can we be sure that this procedure will correctly evolve, as we saw in the first lecture?

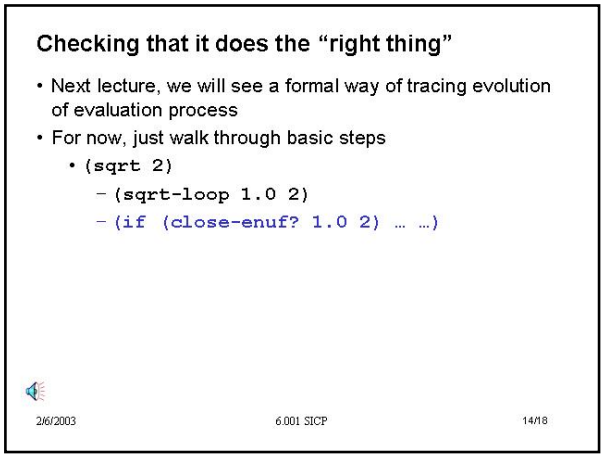
In the next lecture, we are going to introduce a formal model for tracing the evaluation of expressions, especially expressions involving the application of procedures. For now, however, we can be a bit informal and walk through the steps of the computation.

As an example, let's suppose we try to find the **sqrt** of 2.

Slide 2.4.13

Basically, we can replace this expression with the body of the procedure associated with `sqrt`, where the formal parameter is replaced with the specific value. In other words, we reverse the process of capturing a pattern, so we are going to loop through the process, using an initial guess of 1.

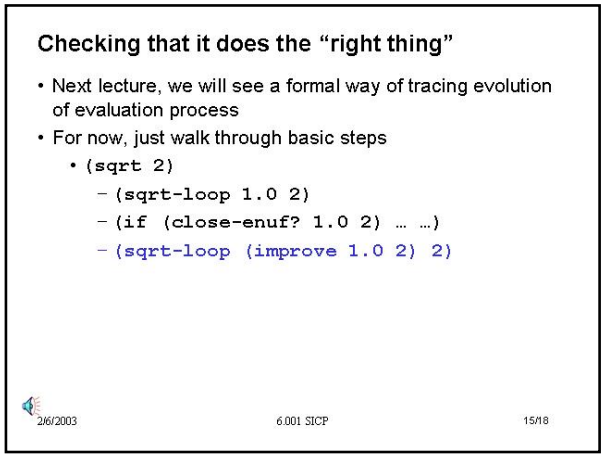


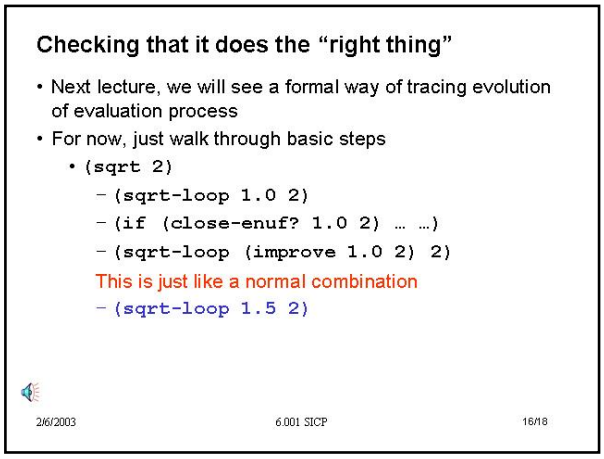


Slide 2.4.14

Now what does `sqrt-loop` do? Well, the pattern it captured was to see if we are close enough: here we have the body of the `sqrt-loop` procedure with the specific values in place. For now, we don't care about the other sub-expressions of the `if` expression.

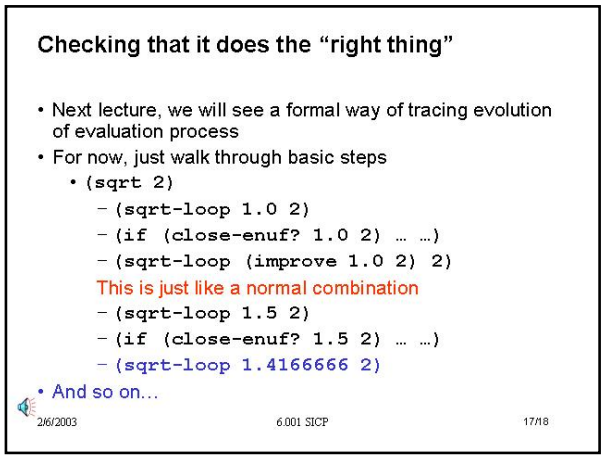
Slide 2.4.15
In this case, we are not close enough, so we move to the **alternative** stage of the `if` expression, as shown.

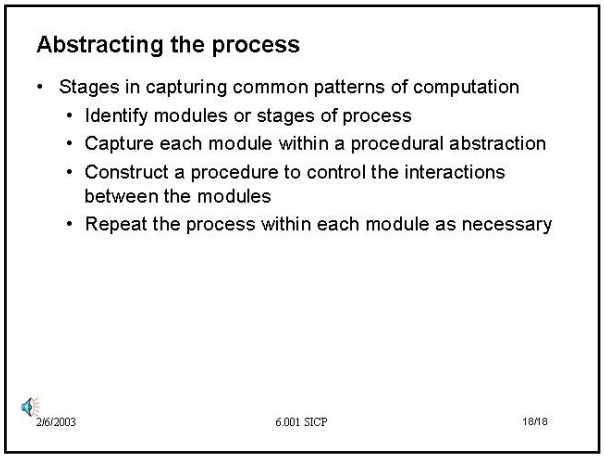




Slide 2.4.16
Now this is just like a normal combination. We reduce the values of the subexpressions, so we get the value of the **improve**d guess...

Slide 2.4.17
… and then repeat the process. We keep doing this until we get a value that is close enough that we can stop.





Slide 2.4.18
So to summarize, we have seen that we can use the idea of a procedure to capture a computational process: by finding good components or modules of the process; capturing each within its own procedure; and then deciding how to control the overall process of the computation. In the next lecture, we will return to this idea, looking at different ways to break a problem down into these steps.