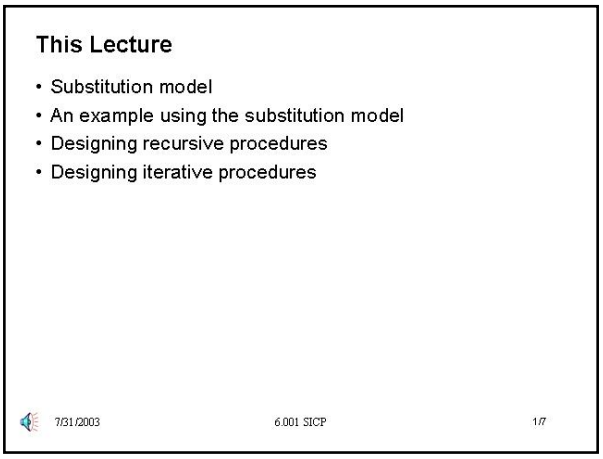
6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

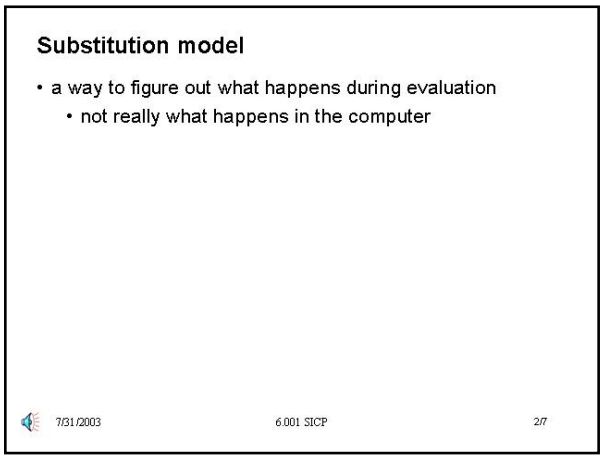
6.001 Notes: Section 3.1

Slide 3.1.1



In this lecture, we are going to put together the basic pieces of Scheme that we introduced in the previous lecture, in order to start capturing computational processes inside procedures. To do this, we will introduce a model of computation, called the **substitution model**. We will show how that model helps us relate the choices we make in designing a procedure to the actual process of evaluation that will occur when we use that procedure.

Thus we will first look at an example of the substitution model, to make sure you understand its mechanics, and then we will use to that examine two different approaches to creating procedures.



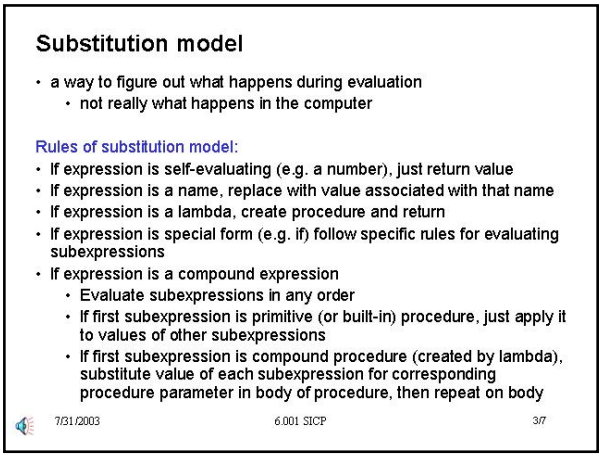
Slide 3.1.2

Now that we have the ability to create procedures and use them, we need some way of figuring out what happens during the evolution of a procedure application. For that we have something called the **substitution model**. You've actually seen this, but we just didn't call it that, and now we are going to make it quite explicit.

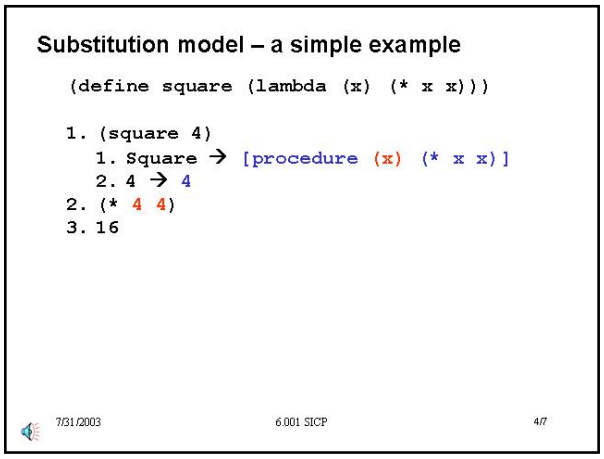
The role of the substitution model is to provide us with a means of determining how an expression evolves. In examining the substitution model, we stress that this is not a perfect model of what goes on inside the computer. In fact, later in the term we will see a much more detailed and accurate model of computation, but this model suffices for our purposes. Indeed, we don't really care about using the substitution model to figure out the actual value of an expression. We can use the computer to do that. Rather, we want to use the substitution model to understand the process of evaluation, so that we can use that understanding to reason about choices in designing procedures. Given that understanding, we can work backwards, using a desired pattern of evaluation to help us design the correct procedure to generate that evaluation pattern.

6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

Slide 3.1.3



So here are the rules for the substitution model. Given an expression, we want to determine its value. If the expression is a simple expression there are several possibilities. If it is a self-evaluating expression, like a number, we just return that value. If the expression is a name (something we created with a **define** expression) then we replace the name with the corresponding value associated with it.
If the expression is a special form, there are also several possibilities. If the expression is a **lambda** then we replace the expression with some representation of the actual procedure. If the expression is some other special form (such as an **if** expression) then we follow specific rules for evaluating the subexpressions of that special form.
Otherwise the expression is a compound expression (one of those expressions nested in parentheses that contains several subexpressions). Here we apply exactly the same set of rules to each of the subexpressions of the compound expression, in some arbitrary order. We then look at the value of the first subexpression. If it is a primitive procedure (one of the built-in procedures, like * or +) then we just apply that procedure to the values of the other expressions. On the other hand, if the value of the first subexpression is a compound procedure (something we created by evaluating a **lambda** expression) then we **substitute** the value of each subsequent subexpression for the corresponding procedure parameter in the body of the procedure. We replace the entire expression with this instantiated body, and repeat the process.



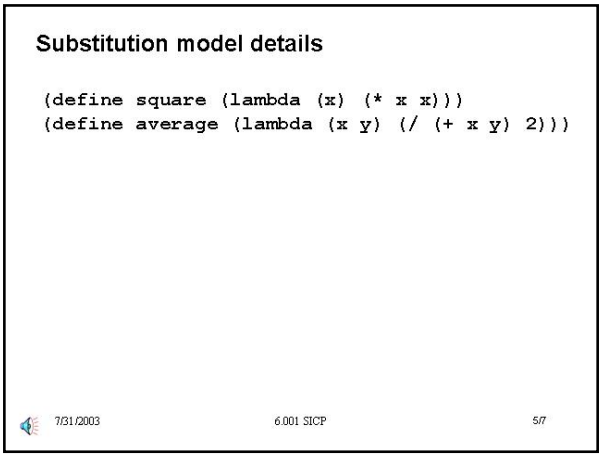
Slide 3.1.4

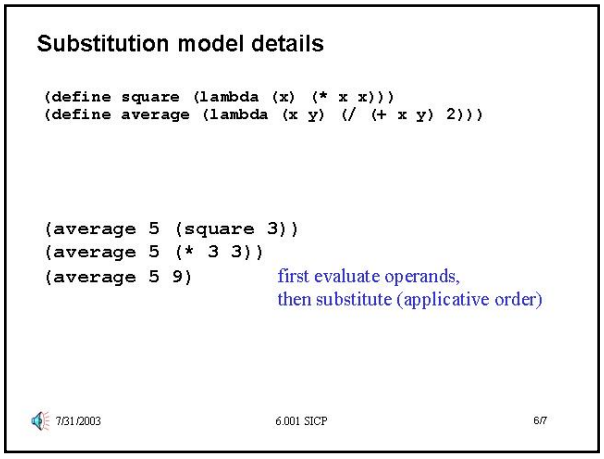
So here is a simple example. Suppose we have defined **square** to be the obvious procedure that multiplies a value by itself. Now, let's use the substitution model to trace out the evolution of applying that procedure.
The substitution model says that to trace the evaluation of **(square 4)**, we first determine that this is a compound expression. Since it is, we evaluate each of the subexpressions using the same rules. Thus, the first subexpression, **square,** is just a name, and we look up its value, getting back the procedure we created when we evaluated the **lambda.** The second subexpression is just a number, so that is its value. Now the rule says to substitute 4 everywhere we see the formal parameter **x** in the body of the procedure and replace the original expression with this instantiated body expression, as shown.
This reduces the evaluation of **(square 4)** to the evaluation of the expression **(* 4 4)**. This is also a compound expression, so again we get the values of the subexpressions. In this case, the application is of a primitive procedure, so we simply apply it to reduce the expression to its final value, **16**.

6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

Slide 3.1.5

Here is a more detailed example. First, let's create two procedures. These expressions will each generate a procedure, through the **lambda** expressions, and the **defines** will then associate a name with each of them. Thus, in our environment, we have pairings of **square** and **average** to the associated procedure objects, each with its own parameter list and body.





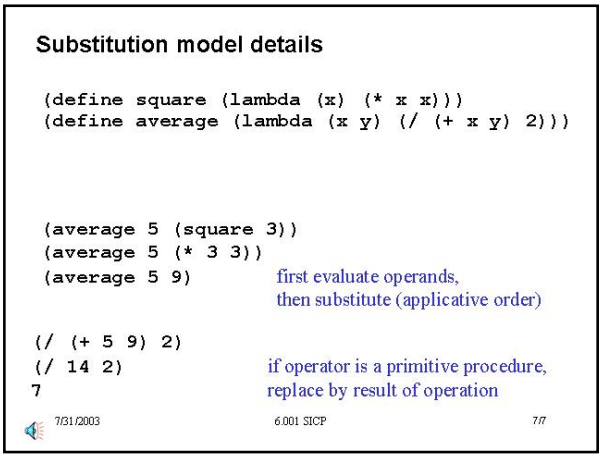
Slide 3.1.6

Here is an example using these procedures. To evaluate this first expression using the substitution model, I first need to get the values of the sub-expressions. The value of **average** I get by looking this name up in the environment, giving me the associated procedure object. The value of 5 is easy. The last sub-expression is itself a combination, so I recursively apply the rules. By the same reasoning I get the procedure associated with square and the value 3, then substitute 3 into the body of square for **x**. This reduces to another combination, which I can recognize is an application of a primitive procedure, so this reduces to the number 9. Thus the recursive application of the substitution model yields a simpler combination.

Note the format here. I first evaluate the sub-expressions, then substitute and apply the procedure. This is called an **applicative order** evaluation model.

Slide 3.1.7

Notice in particular that under this model I need to get the value of the operands first, which caused me to evaluate that interior compound expression before I got around to applying **average**. Once I have simple values, I can continue the substitution. In this case, I use the body of the procedure associated with **average**, now substituting 5 and 9 for **x** and **y** in that body. Note that there is no confusion about which **x** I am referring to, it's the one in the procedure associated with **average**, not the one in **square**.



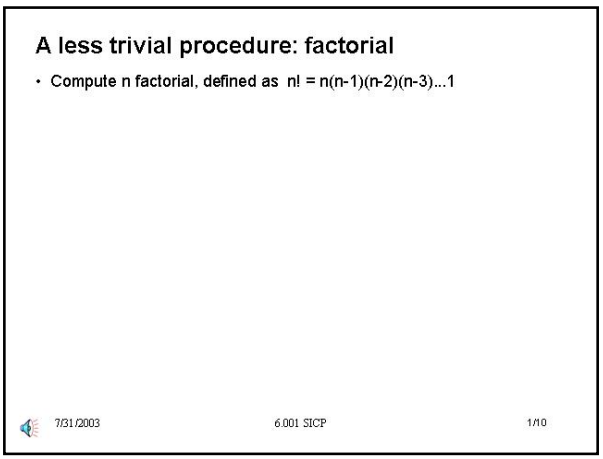
As before, I substitute into the body of this procedure, and then continue. I recursively apply the rules to each subexpression. In the case of a compound expression, I must reduce this to a simpler value before I can continue with the rest of the expression. Also note that when the application involves a primitive procedure, I just do the associated operation, eventually reducing this to the final answer.

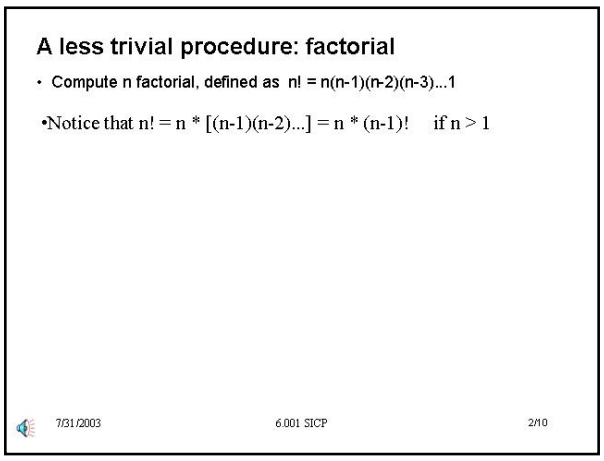
The key idea is to see how this notion of substitution lets us trace out the evolution of an expression's evaluation, reducing an application of a procedure to the evaluation of a simpler expression.

6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

6.001 Notes: Section 3.2

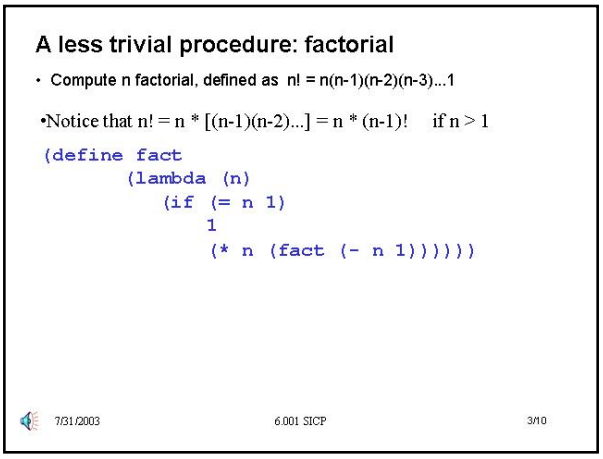
Slide 3.2.1
Now let's look at a little less trivial example. Suppose I want to compute factorial of **n**, which is defined as the product of **n** by **n-1** and so on down to 1. Note that I am assuming that **n** is an integer and is greater than or equal to 1.



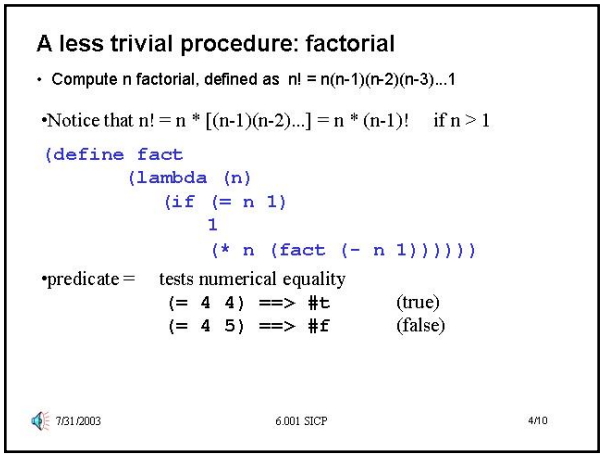


Slide 3.2.2
So how do I create a procedure to compute factorial of **n**? Well I need to think carefully here, since my choice in designing this procedure will impact the evolution of the process when I use it.
One way to look at factorial is to group the computation into multiplying **n** by all the other multiplications, but I can recognize that this latter computation is just factorial of **n-1**. So notice what this does, it reduces a computation to a simpler operation, multiplication, and a simpler version of the same problem, in this case factorial of a smaller number. So I have reduced one version of a problem to a simpler version of the same problem, plus some simple operations. This is a very common pattern that we are going to use a lot in designing procedures.

Slide 3.2.3
In fact, given this common pattern, we can write a procedure to capture this idea. Here it is. The first part says to give the name **fact** to the procedure created by the **lambda**, which has a single argument **n** and a particular body. Now, what does the lambda say to do? It has two different parts, which we need to look at carefully.

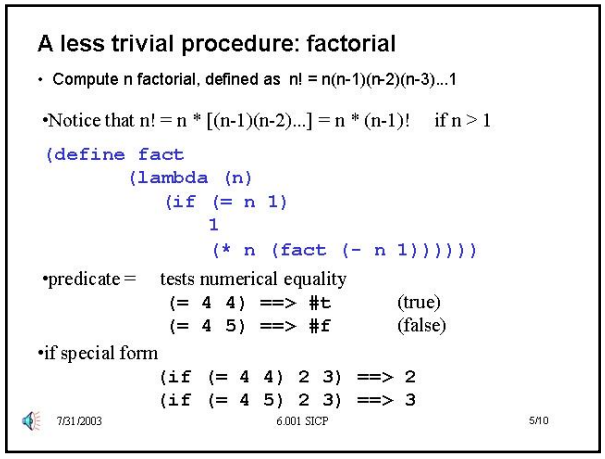


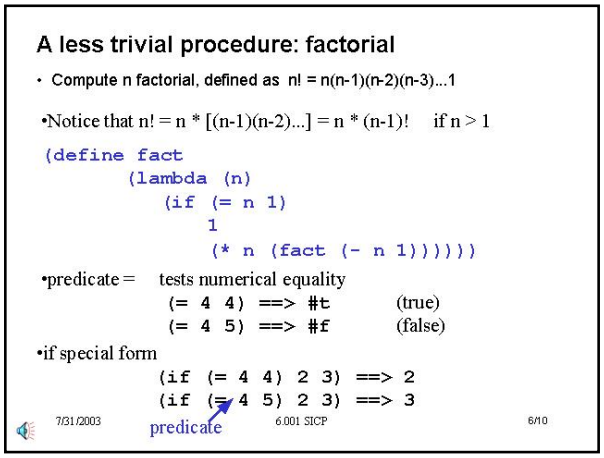
6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.



Slide 3.2.4
First of all, it has a new kind of expression in the interior, the **=**. This is an example of a predicate, a procedure that takes some arguments and returns either **true** or **false**, in this case based on the test for numerical equality.

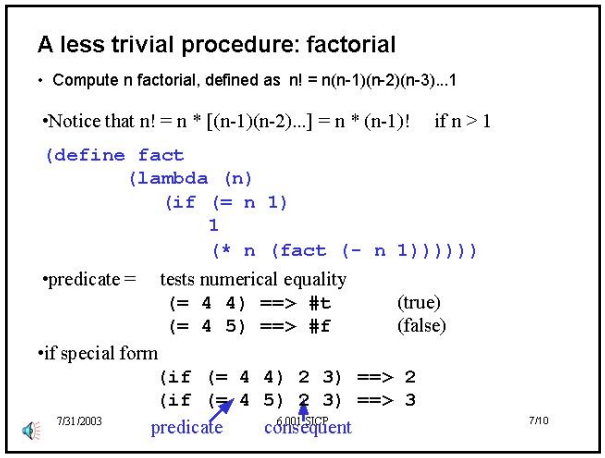
Slide 3.2.5
The more interesting part is the body of the lambda, and it is a new special form called an **if**. Because this is a special form, it means that evaluation does not follow the normal rules for combinations. **If**'s are used to control the order of evaluation, and contain three pieces.



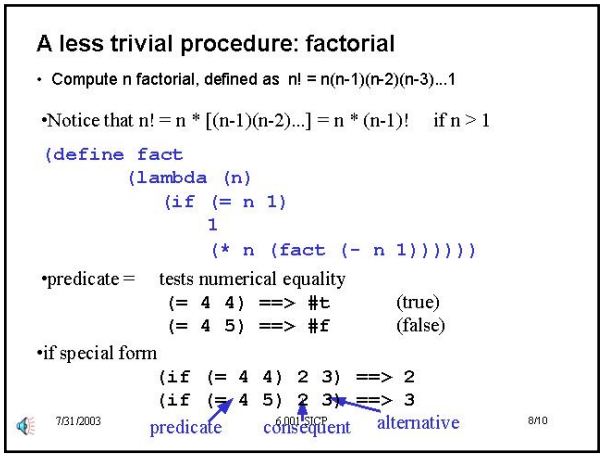


Slide 3.2.6
Its first subexpression we call a **predicate**.

Slide 3.2.7
Its second subexpression we call a **consequent** ...



6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.



Slide 3.2.8

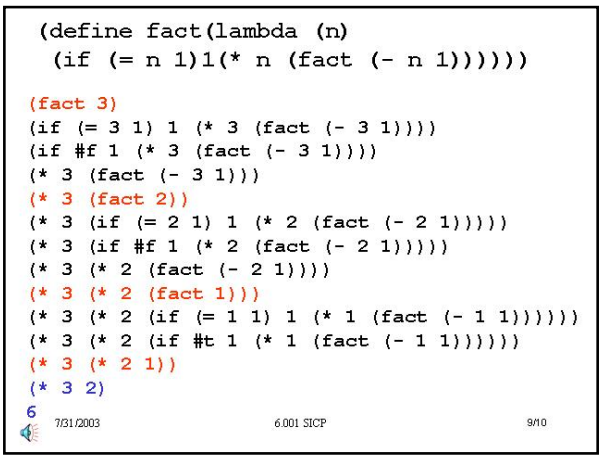
... and its third subexpression we call an **alternative**. Now here is why an **if** is a special form. An if expression first evaluates its predicate, using the same set of rules recursively on this expression. It does this before it ever looks at either of the two other expressions. Now, if the predicate evaluates to a true value, then the if expression takes the consequent, and evaluates it, returning that value as the value of the overall if expression. On the other hand, if the predicate evaluates to a false value, then the if expression takes the alternative expression, evaluates it, and returns that value. Notice that only one of the consequent and alternative expressions is evaluated during an if.

In the case of our **fact** procedure, we can now see how the evaluation will proceed. When we apply **fact** to some argument, we first test to see if that value is 1. The **if** expression does this by first evaluating the predicate, before ever considering the other expressions, thus changing the order of evaluation. If the predicate is true, then we simply return the value 1. If it is false, then we will evaluate the last expression with appropriate substitution, thus unwinding factorial of **n** into the multiplication of the value of **n** by the result of evaluating factorial of **n-1**.

Slide 3.2.9

Here is a tracing of the substitution model in action. To evaluate factorial of 3, we substitute into the body of fact, leading to the next expression. The **if** first evaluates the predicate, leading to the next expression. Since the predicate's value is false, the entire **if** expression is replaced by the alternative, with appropriate substitutions.

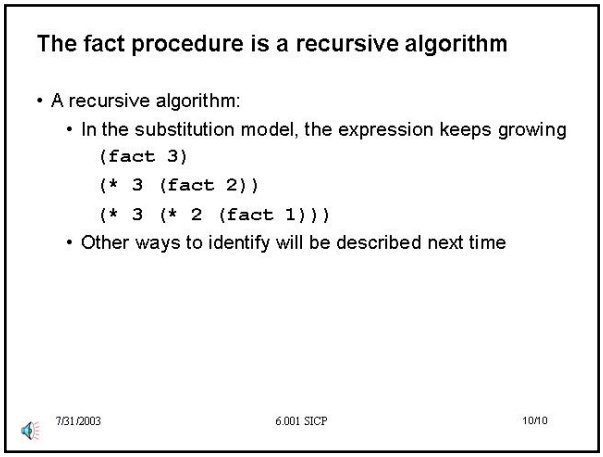
To evaluate this expression we need to get the values of the subexpressions, so this entire expression reduces to a multiplication of 3 by the value of **(fact 2)**. Notice how this has unwound the computation into a particular form, a deferred multiplication and a recursive call to a simpler version of the same problem.



This process repeats, using the **if** to unwind another level of the computation, and this continues, just using the substitution model rules, until finally the predicate is true. In this case, we are left with an expression just involving primitives, and we can complete the multiplications, thus reducing the computation to a simple answer.

Note the form shown in red, in which an expression is reduced to this pattern of a simpler operation and a simpler version of the same problem. This unwrapping continues until we are left with a nested expression whose innermost subexpression only involves simple expressions and operations, at which point the deferred operations are evaluated.

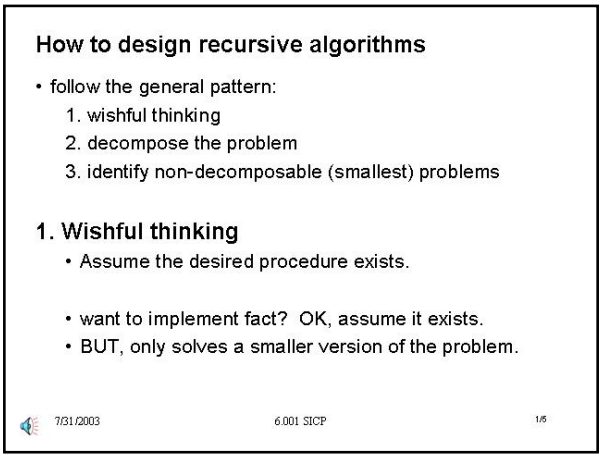
6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

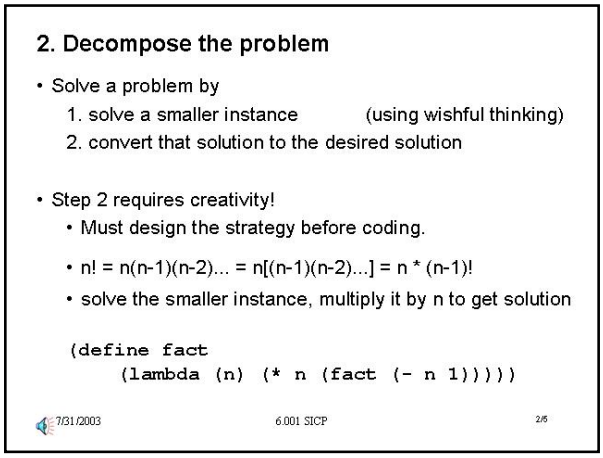


Slide 3.2.10
That means that **fact** gives rise to what we call a **recursive process**. In the substitution model we can see that this is characterized by a set of deferred operations, in which the multiplication is deferred while we go off to get the sub-computation of the simpler version of fact. Once we get to a simple case, we can start accumulating those stacked up operations. We will come back to this idea next time.

6.001 Notes: Section 3.3

Slide 3.3.1
Now that we have seen our first more interesting procedure, **fact**, let's step back and generalize the ideas we used to capture this computation in a procedure.
What are the general steps that we use in creating a recursive procedure like this? We have three stages: as shown here.
The first stage is called **wishful thinking**. The idea is as follows. If I want to create a solution to some problem, I first assume that I have available a procedure that will solve the problem, but only for versions of the problem smaller than the current one. In the case of factorial, I assume (wishfully) that I can solve factorial for problems of size smaller than **n**.

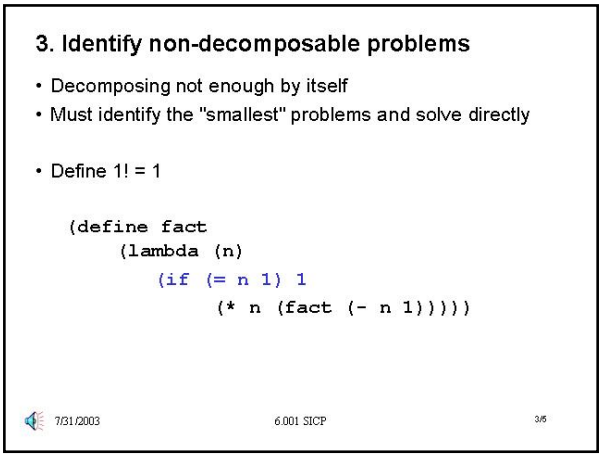




Slide 3.3.2
Given that assumption, the second stage proceeds to design a solution to the problem. In particular, I can use the existence of a solution to the smaller sized problem, plus a set of simple operations, to design the solution to the larger sized problem.
We saw this with factorial, where we combined the solution to a smaller version of factorial, plus a multiplication, to create the solution to the full version of factorial.
Notice that the second step here requires some ingenuity. One should be careful to think through this strategy, before beginning to code up a solution. In the case of factorial, we saw how we did this decomposition into a multiplication and a smaller version of factorial.
With this, we can build a first version of the procedure as shown at the bottom of the slide.

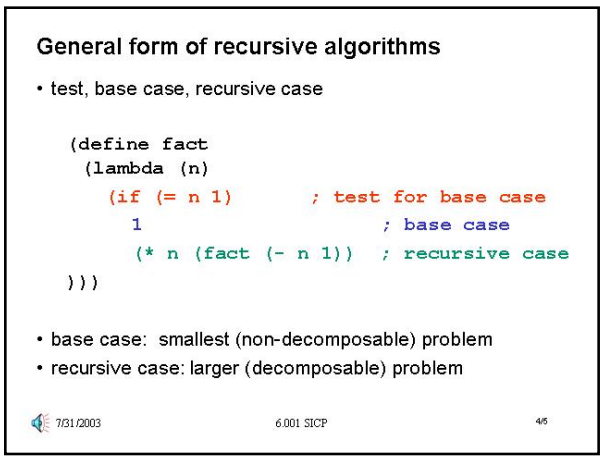
6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

Slide 3.3.3



Of course you already know that this won't quite work, but let's think about why. If we apply that version of **fact** to some argument, it will keep unwinding the application of factorial into a multiplication and a smaller version of factorial. But before we can compute that multiplication, we have to get the value of the smaller version of factorial, and that will continue to unwind to another version, ad infinitum.

The problem is that I didn't use my third step; I didn't consider the smallest sized problem that I can solve directly, without using wishful thinking. In the case of factorial, that is just knowing that factorial of 1 is just 1. Once I find that smallest sized problem, I can control the unwinding of the computation, by checking for that case and doing the direct computation when appropriate. In this way, I will terminate the recursive unwinding of the computation and accumulate all the deferred operations.

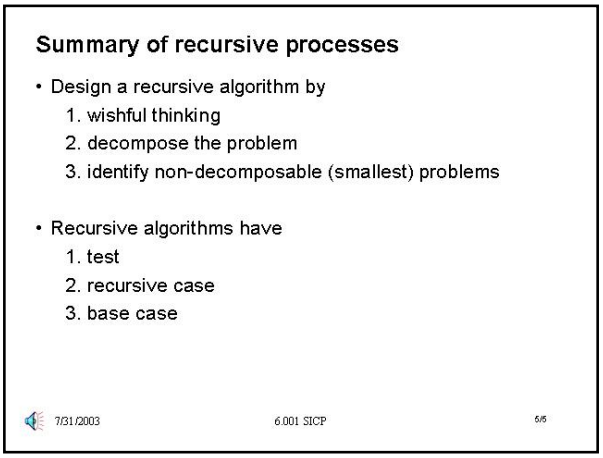


Slide 3.3.4

This in fact is a very common form, a common pattern for a recursive algorithm that has a **test**, a **base case** and a **recursive case**. The **if** controls the order of evaluation to decide whether we have the base case or the recursive case. The recursive case controls the unwinding of the computation, doing so until we reach the base case.

Slide 3.3.5

To summarize, we have now seen how to design recursive algorithms, by using this idea of wishful thinking to separate the problem into a simpler version of the same problem, do that decomposition and identify the smallest version of the problem that will stop the unwinding of the computation into simpler versions of the same problem. As a consequence, algorithms associated with this kind of approach have a **test,** a **base case,** and a **recursive case** to control the order of evaluation in order to unwind the computation into simpler versions of the same problem.

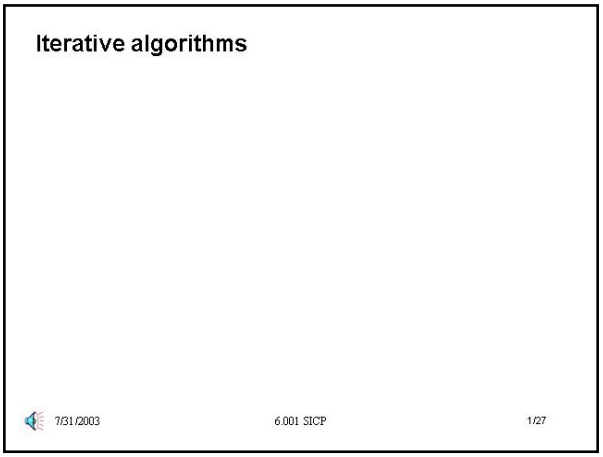


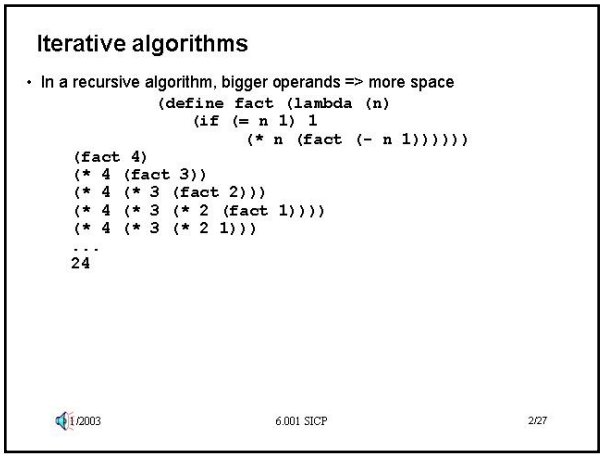
6.001 Notes: Section 3.4

6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

Slide 3.4.1

So we have introduced recursive procedures, and we have now seen an example of building a recursive procedure to implement factorial. Now, let's look at a different kind of procedure, an iterative procedure for computing factorial. In doing this, we are going to see how we can develop procedures with different evolutions that compute the same basic computation.



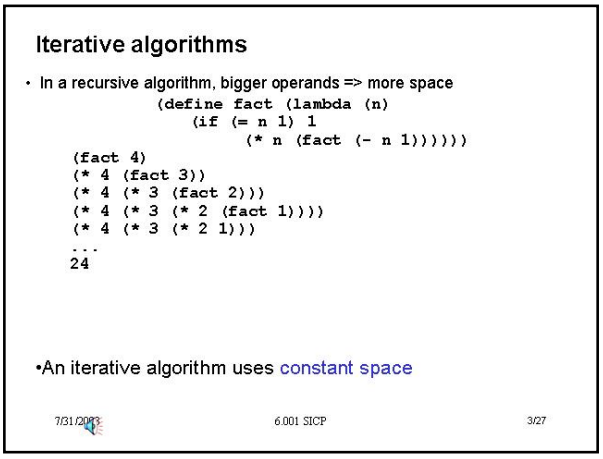


Slide 3.4.2

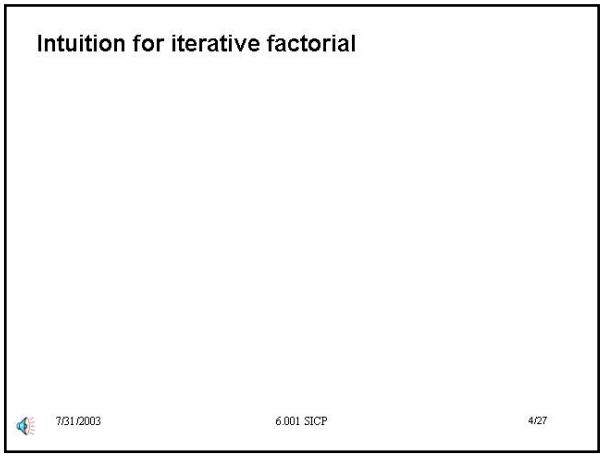
To set the stage for this, here is our **factorial** procedure from last time. Remember that this was a nice little procedure that first checked to see if we were in the base case, and if we were, returned the answer 1. If we were not, the procedure reduced the problem to multiplying **n** by a recursive call to **factorial** of **n-1**. One of the properties of this procedure was that it held a set of deferred operations. To compute factorial of n, it had to hold onto a multiplication while it went off to compute the subproblem of a simpler version of factorial, and that in turn required another deferred operation. Thus, the procedure had to keep track of something to do, once it was done computing a subproblem, and this meant that as the argument to the procedure gets larger, we would have more things to keep track of. We would like to see if there is another way of computing factorial, without all of those deferred operations.

Slide 3.4.3

The specific question we want to consider is whether we can design an algorithm for factorial that uses only constant space. By this we mean that the computation does not require keeping track of any deferred operations, but rather can perform its work without using up any additional memory to keep track of things to do.



6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

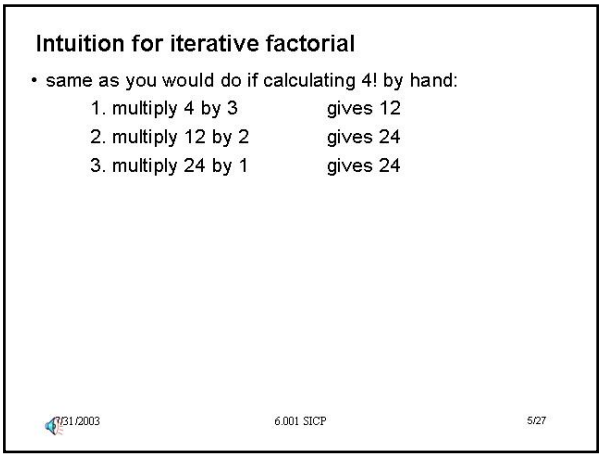


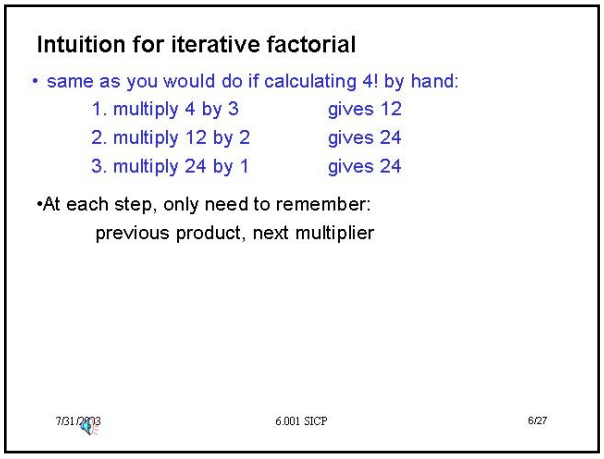
Slide 3.4.4

So that's our goal, to compute factorial in a different way that is not characterized by those deferred multiplications

Slide 3.4.5

... and here is the intuition behind this different approach. Let's think about how you might do this if you were executing the computation by hand. One way to do this is to multiply out the first two numbers, and keep track of that temporary product, plus the next stage in the computation. For example, for factorial of 4, you would multiply out 4 by 3, keep track of the result, 12, and the fact that the next thing to multiply is 2. You would then multiply the current result, 12, by 2, keeping track of that result and the next thing to multiply by.



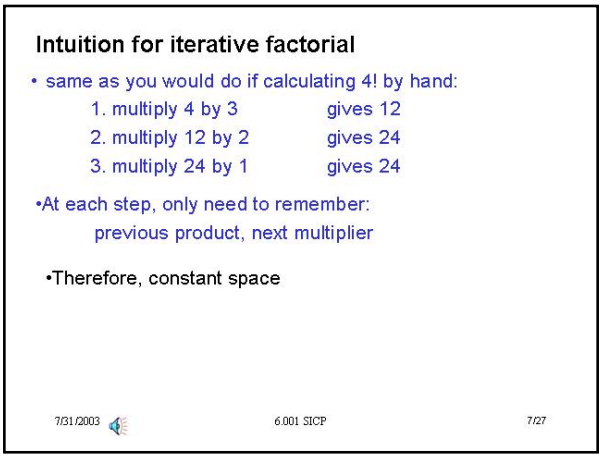


Slide 3.4.6

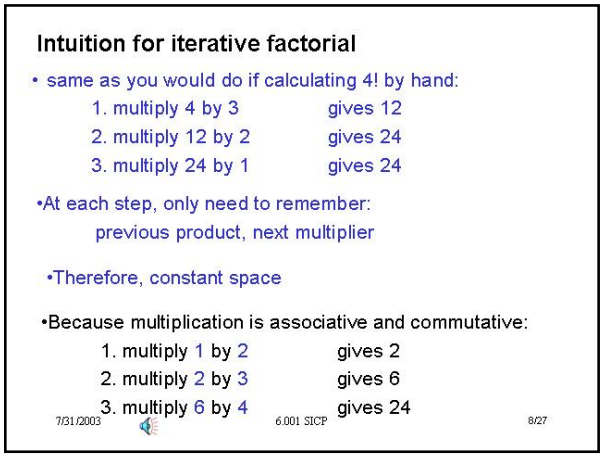
Notice that in this version, we only need to keep track of 2 things, the current product, and the next thing to do. This is different from the recursive case, where we were blindly keeping track of each pending multiplication. Here, we replace the previous product by the new one, and the next multiplier by the new one, whereas in the previous case we had to keep track of each deferred operation, i.e. I have to multiply by 2, then I have to multiply by 3, then ...

Slide 3.4.7

Thus, we only need a constant amount of space. To stress, think about the recursive case. There, we had to write down that I have a pending operation of multiplication by some number, plus another pending operation of multiplication by another number, and so on. Keeping track of each such pending operation takes a bit more space. In this case, I only need two pieces of information, the current product and the next multiplier.



6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

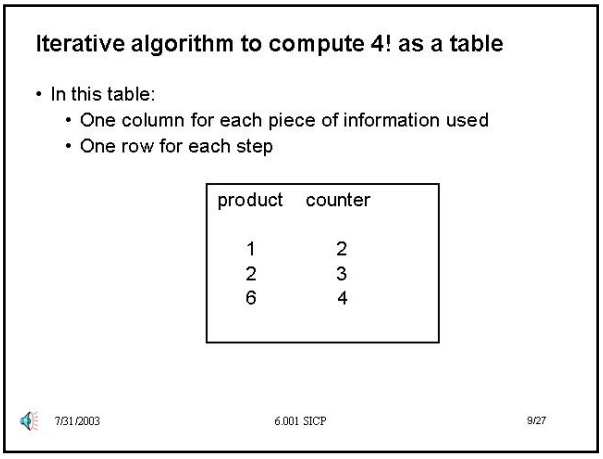


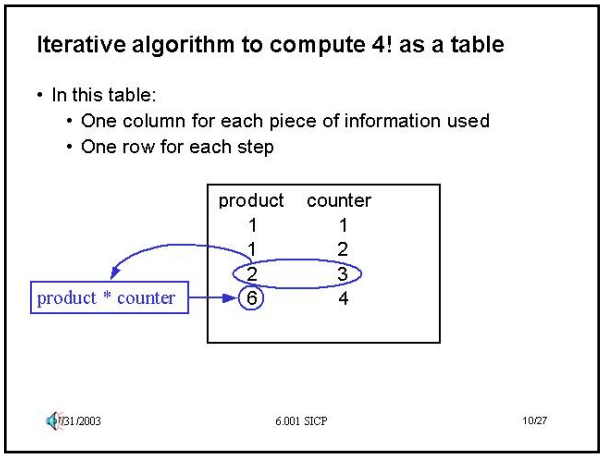
Slide 3.4.8

The reason I can do this is because multiplication satisfies two nice properties from mathematics, that in essence say I can do the multiplication in any order, and accumulating partial answers along the way will not affect the result. Thus, I can come up with an algorithm that only uses constant space.

Slide 3.4.9

So now let's capture that idea a little more formally. I can use the same sort of notion in a table, in which I have one column for each piece of information I am going to need, and one row for each step of the computation. Each step here means how to do I go from a current state, captured by a temporary product and a counter that tells me the next thing to multiply, to the next state of those parameters.



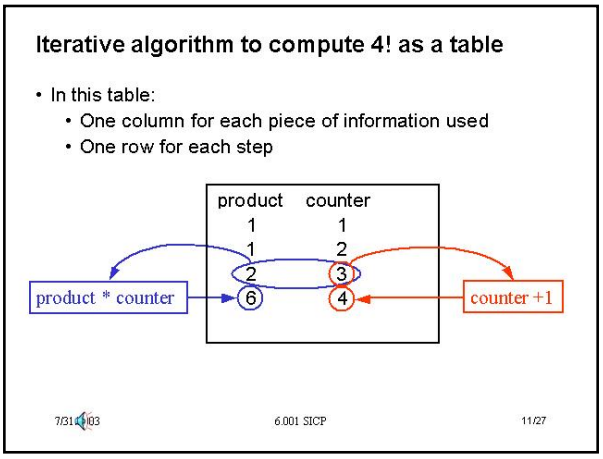


Slide 3.4.10

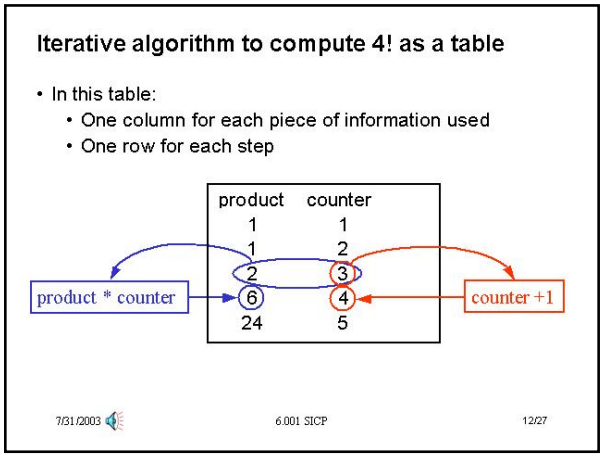
Now what I need to do is come up with a rule for how I am going to change the values in the table. In particular, to get the next value for product, I take the current value of product and the current value of counter, multiply them, and that becomes the next entry in the product column.

Slide 3.4.11

Similarly, I need a rule for getting the next value for counter. That, we know, is simply given by adding 1 to the current row's value for counter, thus generating the next row's value for counter.

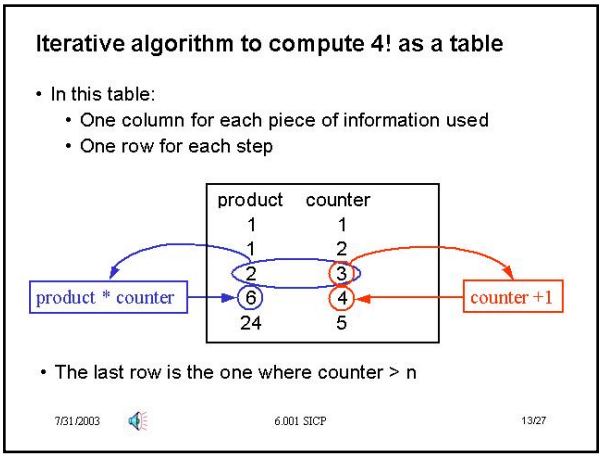


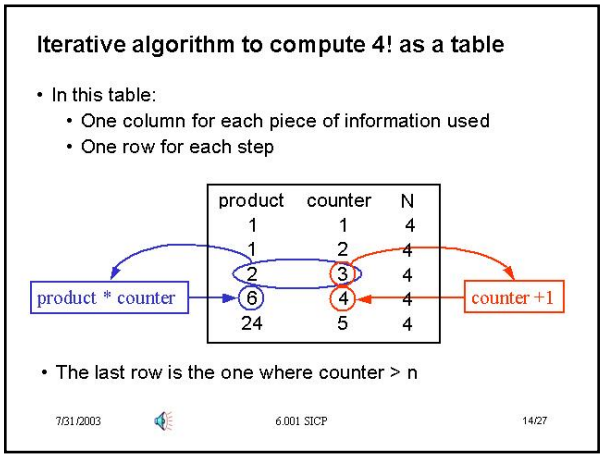
6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.



Slide 3.4.12
Of course, I can keep repeating this process, updating the values for the product column and the counter column until I am ready to stop.

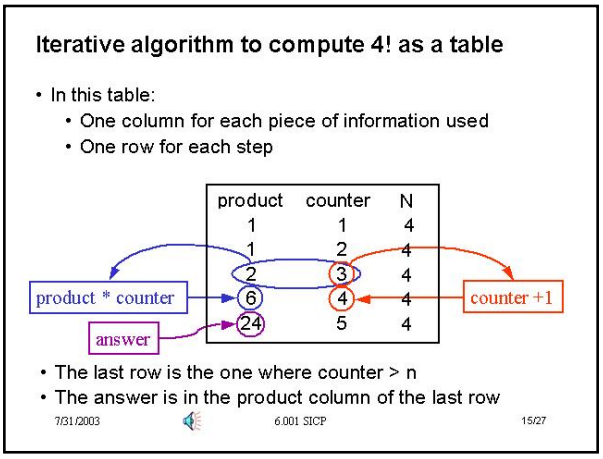
Slide 3.4.13
And how do I know when I am ready to stop? Well that's easy. The last row I want in the table is the one when counter is bigger than n.



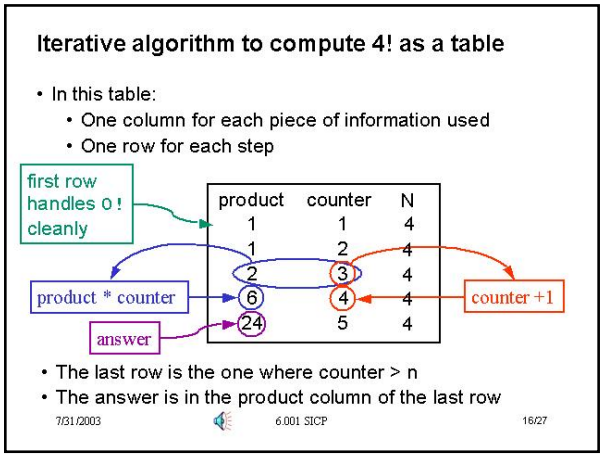


Slide 3.4.14
Ah -- and that indicates one more thing. I will need another column to keep track of **n**, since each column specifies a needed piece of information and that is one more. The fact that the value doesn't change is irrelevant, what matters is that I need to keep track of this information.

Slide 3.4.15
Once I am done, I still need to get the answer out, but I know where that is. It's sitting in the element at the intersection of the product column and the last row.

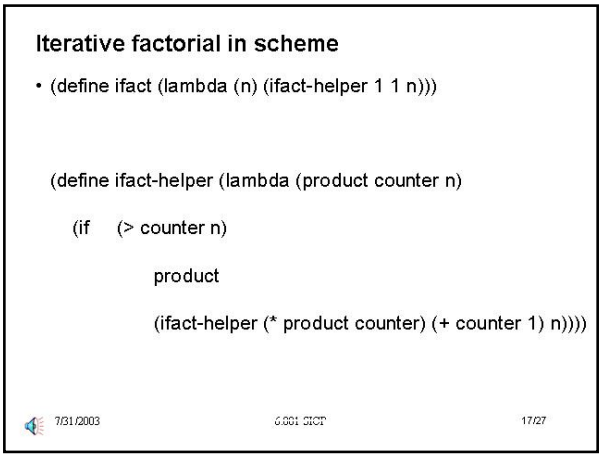


6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.



Slide 3.4.16

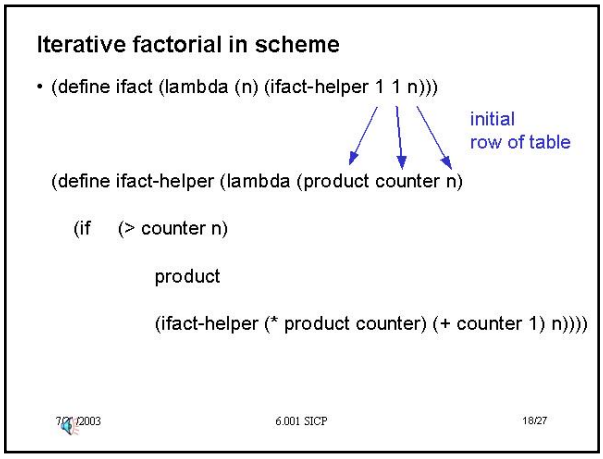
The only other thing I need to do is figure out how to start this process off. And this is also easy. My first row will deal with the equivalent of a base case. 0! is just 1, so I can start my product at 1, my counter at 1, set up N to be whatever value I want to compute, and I can then start this table process underway using the rules I just described.



Slide 3.4.17

Now let's write a procedure to do this. I am going to define a procedure called `ifact`. As you can see, I use a `lambda` to create the procedure of one argument, and I use a `define` to give it a name. Notice that the body of the procedure is just a call to another procedure, a helper procedure that I am also going to define.

The reason I need a helper procedure is clear from my previous discussion. I need to keep track of three things during the computation, so I need a procedure with three arguments, as shown, one for each column of my table. The body of this helper procedure looks a lot like my earlier recursive procedure, but with some important differences. First, there is a test case, to tell when we are done. Here we don't have a base case, but rather there is a return value, an answer to give when the test case holds. The iterative case, the thing to do if the test case is not true, is slightly different from our recursive case. Here, the call to the procedure has new arguments, one for each parameter. Notice how the new argument in each case captures the update rule from the table.



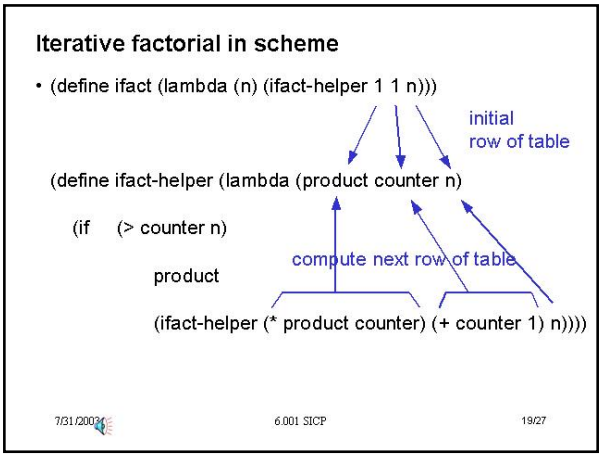
Slide 3.4.18

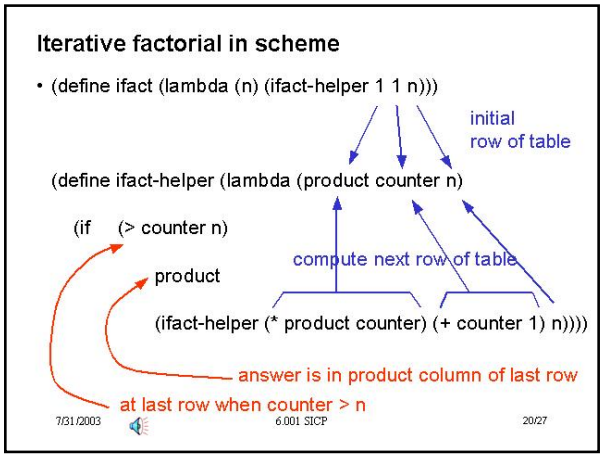
So now let's think about what happens if I evaluate `ifact`. The value of `ifact` with some argument `n` will be the same as evaluating the body of the procedure, that is `ifact-helper` with arguments 1, 1, and n. Notice that this is just setting up the initial row of my table.

6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

Slide 3.4.19

Then look at what happens when we evaluate `ifact-helper`. In fact, it just computes the next row of that table. It says that the value of `ifact-helper` when called on arguments for product, counter and n, is the same as evaluating `ifact-helper` with new values for product, counter and n, each of which exactly captures the update rules from my table.



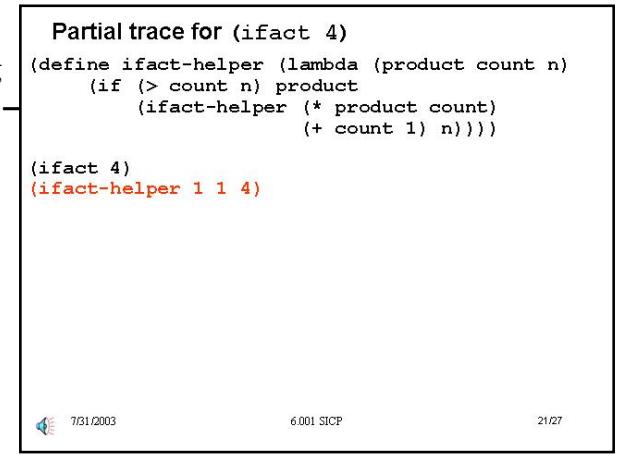


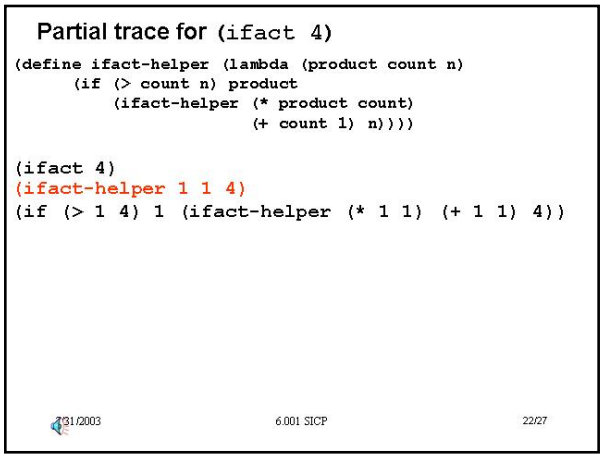
Slide 3.4.20

And what else do I need? Just to determine when to stop. So I have a test to determine when that happens, and at that stage, I know that the answer is in the product column, so I simply return that value.

Slide 3.4.21

To trace how this works, we use our substitution model. Calling `ifact` on some argument, say 4, reduces to calling `ifact-helper` on arguments 1, 1, and 4, since that is what the body of `ifact` contains, where we have substituted 4 for the parameter `n`.





Slide 3.4.22

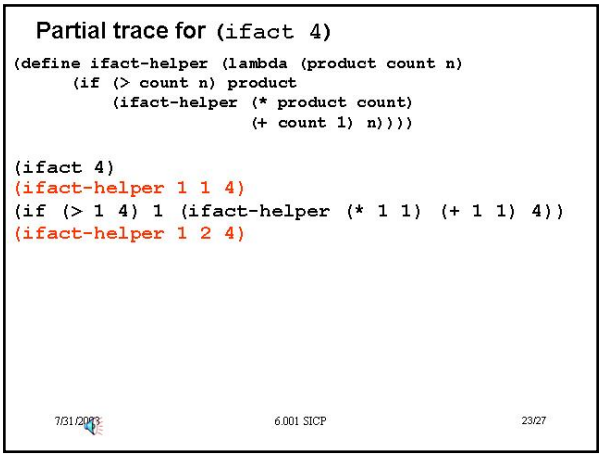
The substitution model then says to substitute the arguments for the formal parameters in the body of `ifact-helper` and evaluate that instantiated body, namely this `if` expression. We saw that `if` is a special form, and this means we first evaluate the predicate. Since in this case it returns a false value, the value of the whole if expression reduces to the value of the alternative clause.

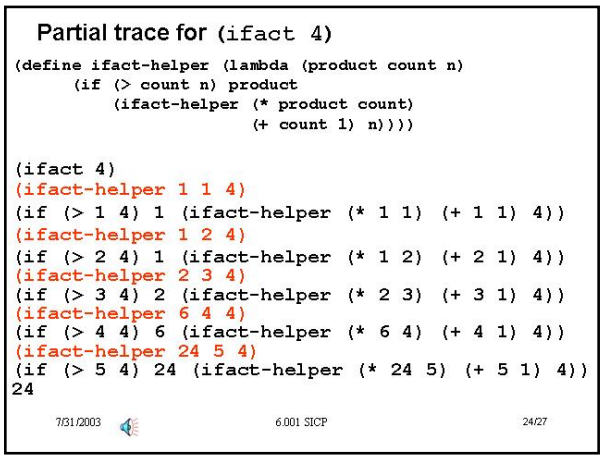
6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

Slide 3.4.23

Evaluating the expressions for the arguments then reduces to this expression.

Notice an important thing. We have reduced the evaluation of one expression involving `ifact-helper` to a different expression involving `ifact-helper`, with a different set of arguments, or parameters. But there are no deferred operations here, that is, there are no operations that we have to keep track of while we go off and compute some subproblem. Rather the answer to evaluating this expression is the answer to evaluating the first one directly.





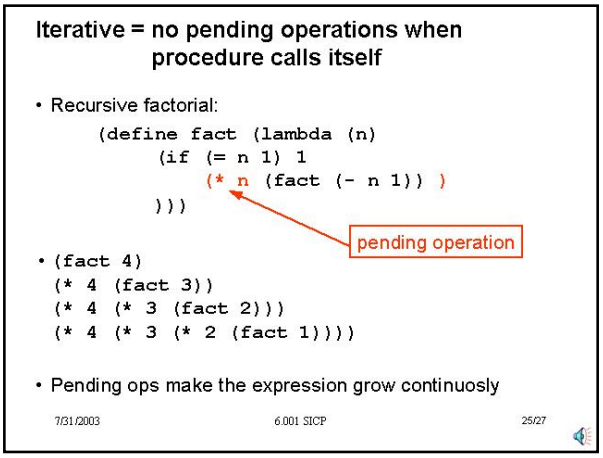
Slide 3.4.24

And we can just repeat this process, with the same behavior, until the predicate is true.

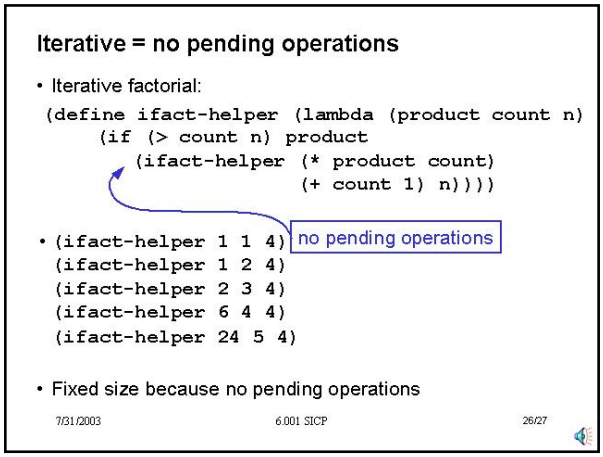
Notice the form of the red expressions. They are constant in size. There are no deferred operations, no storage of expressions while I go get the value of a subexpression. Instead, the variables that capture the state of the computation change with each evaluation. This is very different from the shape of the previous version of factorial. There, we had a set of deferred operations, and the size of the expressions grew with each step. Here there are no deferred operations, and a constant amount of space is needed to keep track of the computation.

Slide 3.4.25

Let's compare this to the recursive version. Remember that we had that pending operation, and it caused the shape of the procedure's evolution to be different. It grew with each new evaluation, because that extra pending operation had to be tracked, including information relevant to each operation.

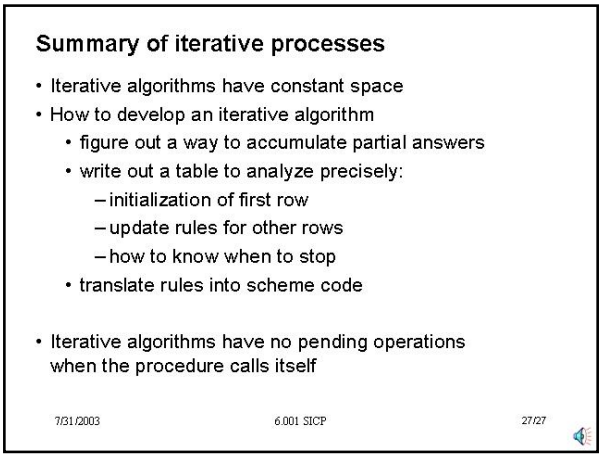


6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.



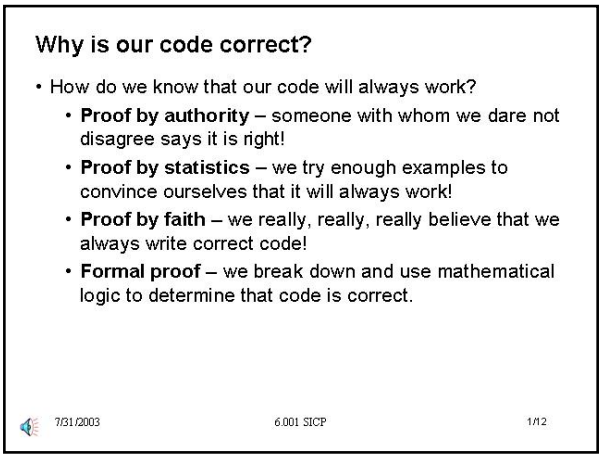
Slide 3.4.26
Compare that to the iterative version. Here there are no pending operations. The value of one call to the procedure is the same as the value of a different call to the same procedure, that is, different arguments but nothing deferred. As a consequence, this procedure has a fixed size, or constant, evolution.
So we have seen two different implementations of the same computation, with different behaviors.

Slide 3.4.27
And this summarizes what we have seen so far.



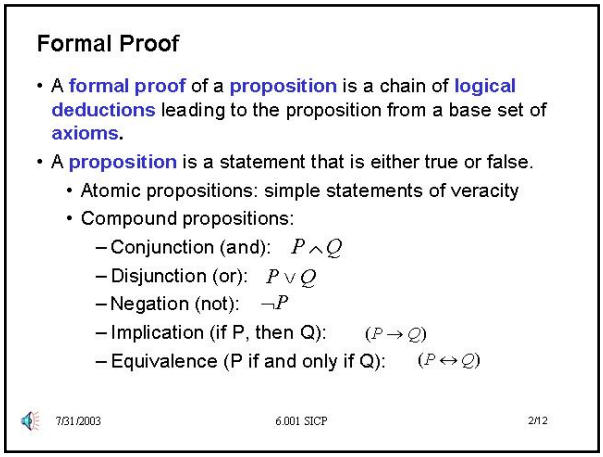
6.001 Notes: Section 3.5

Slide 3.5.1
So how do we know that our code is actually correct? It would be nice if we could guarantee that our code would run correctly, provided we give it correct inputs. So how do we ensure this? Well there are several possibilities. We could accept the word of someone with power over our future (e.g. a 6.001 professor) that the code is right. Clearly not a wise idea, though it is shocking how often this approach is used! A more common method is to rely on statistics, that is, to try our code on a bunch of example inputs, and hope that if it works on all the ones we try, it will work on every input. Sometimes this is the best you can do, but clearly there are no guarantees with this approach. All too often, programmers rely on the third method, also known as arrogance. They just assume that they wrote correct code, and are stunned when it fails.



The best method, when possible, is to actually formally prove that our code is correct. And it is this method that we are going to briefly explore here. We won't show you all the elements of proving programs correct, but we will provide you with a basis for understanding how this is done, and more importantly for using the reasoning pattern underlying this proof method to help you design good code.

6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.



Slide 3.5.2

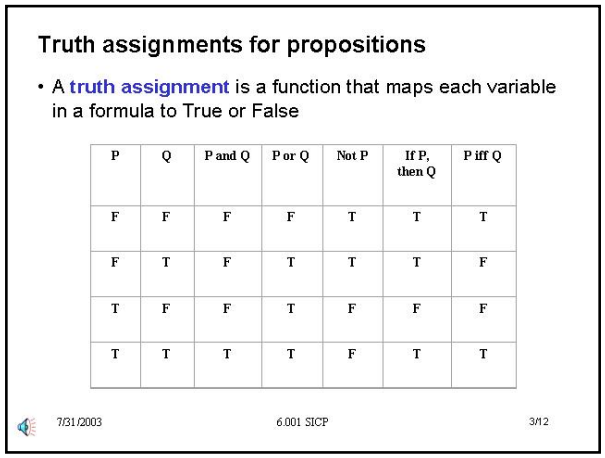
First, we need to talk about what constitutes a formal proof. Technically, a proof of a mathematical or logical proposition is a chain or sequence of logical deductions that starts with a base set of axioms and leads to the proposition we are attempting to prove. Now, we need to fill in the details on these pieces.

First, a proposition is basically a statement that is either true or false. Typically, in propositional logic, we have a set of atomic propositions, i.e., simple statements of fact. For example, "n=0" is an atomic proposition. Given some value for the variable n, this statement is either true or false (assuming that n is a number, of course).

More interesting propositions (and things that we are more likely to be interested in proving) are created by combining simpler propositions. There are five standard ways of creating compound propositions: one can take the conjunction (which acts like an "and", meaning the compound proposition is true if and only if both of the individual propositions are true); one can take the disjunction (which acts like an "or", meaning the compound proposition is true if and only if one or the other of the individual propositions is true); one can take the negation of a proposition (which means the proposition is true if and only if the initial proposition is false); we can create the implication (which means that if P is true, then Q is true); and we can establish an equivalence (which means that P is true if and only if Q is true).

Note that compound propositions can have elements that are themselves compound propositions, so that we can create deeply nested propositions.

Slide 3.5.3



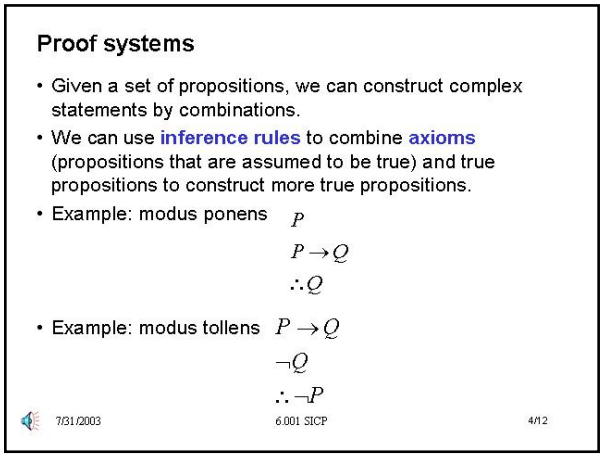
P	Q	P and Q	P or Q	Not P	If P, then Q	P iff Q
F	F	F	F	T	T	T
F	T	F	T	T	T	F
T	F	F	T	F	F	F
T	T	T	T	F	T	T

We can capture the logical structure of these compound propositions by simply looking at the possible cases. If we consider all possible combinations for the base propositions, we can chart out the veracity of the compound proposition. Thus, we see that "P and Q" behaves as we would expect; it is true as a statement if and only if both P and Q are true. Negation and disjunction also behave as expected. So does equivalence, since the combination can only be true when both simpler elements are.

Implication is a bit more puzzling. The proposition is defined to be true if either P is false or Q is true. To see this, take a simple example. Suppose we consider the proposition: "If n is greater than 2, then n squared is greater than 4". Then for example, if n equals 4, both propositions are true, and our truth table also states that the implication is true. If for example, n equals -4, then the first proposition is false and the second proposition is true. The compound statement is true, as we can see from the truth table, and this makes sense, since the consequent can clearly still be true even if the precedent is not. Finally, suppose n equals 1. In this case both propositions are false, however the statement as a whole is true.

This particular combination thus has a rather counterintuitive aspect: it is always true if the first part is false.

6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.



Slide 3.5.4

Now, given that we can construct propositions, we want to be able to prove their correctness by chaining together a sequence of logical deductions, starting from a set of axioms (which are propositions that are assumed to be true). The basic idea is that we start with statements that are correct, and then use standard deductions to reason about statements whose veracity we are attempting to establish.

For example, a common inference rule (which has been around at least since Aristotle) is `modus ponens`. This basically says that if we have a proposition P that is true. And we have an implication (P implies Q) that is also true, and then we can deduce that Q is a true proposition.

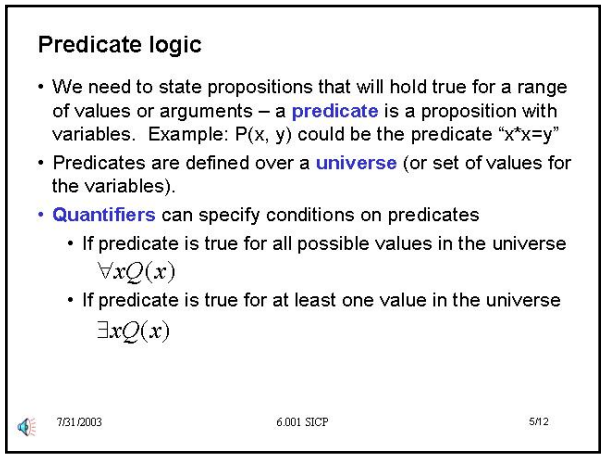
A related (and equally old) inference rule is `modus tollens`. This basically states that if we have an implication that P implies Q, and we also know that Q is not true, then we can infer that P is not true.

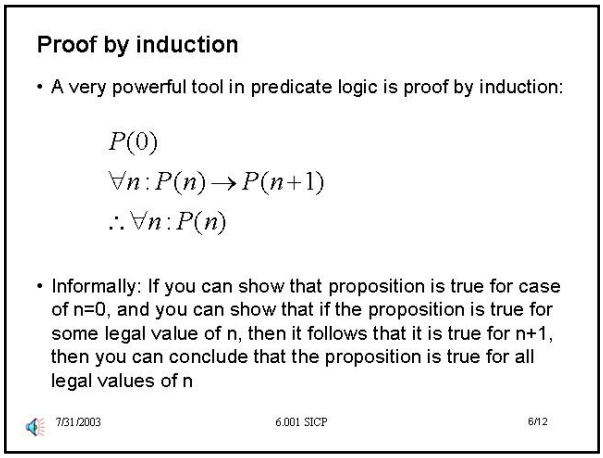
Slide 3.5.5

Now, given these tools, we could go off and prove statements of fact. Unfortunately, the things we want to prove about programs (e.g. does this code run for all correct inputs) require potentially infinite propositions, since we would have to take a conjunction of propositions, one for each input value. So we need to extend our propositional logic to predicate logic.

We handle this by creating propositions with variables, and then specifying the set of values (called the universe or domain of discourse) over which the variable may range.

Since predicates apply over ranges, we also can put conditions on those predicates. The two basic **quantifiers** are known as "for all" and "there exists". They capture the cases in which the predicate is true for all possible values in the universe, and in which the predicate is true for at least one value in the universe.





Slide 3.5.6

Now we are ready to put all these tools together to prove things about predicates. We are particularly interested in a proof method called **proof by induction**, which is shown here.

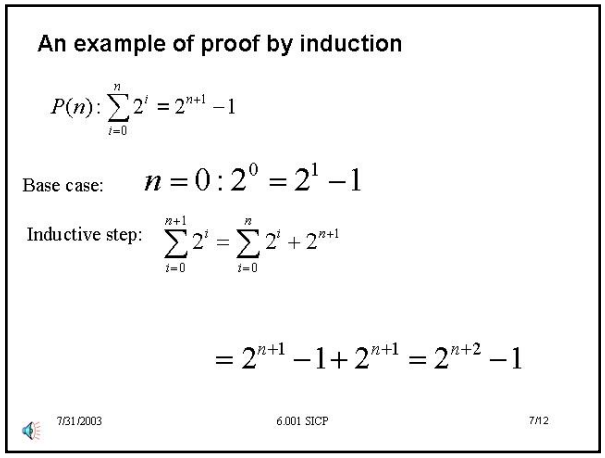
Less formally, this method basically says the following. Suppose you can show that the predicate is true for some initial case (here we are assuming the universe is the set of non-negative integers, so this initial case is n equal 0, if the universe were different the initial case would also be different). Now suppose you can somehow also show that for any value of n in the universe, the implication that if the proposition is true for that n it is then also true for an incremented n is true. In this case, induction allows you to deduce that the proposition is true for all values of n in the universe.

Note what this buys you. It says that you need only show that the statement is true for some base case; and you need only show that for some arbitrary value of n, if the proposition is true for that n then you can show it is true for the next value of n. You don't need to check all values; you just need to show that the implication holds. In

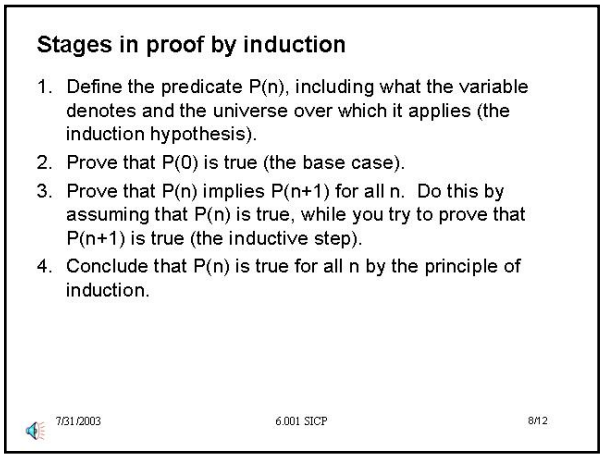
6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

essence, you can "bootstrap" a proof that the statement is always true.

Slide 3.5.7



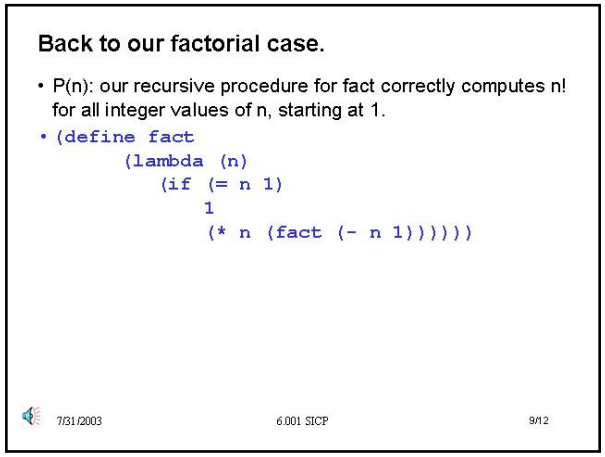
Let's make this a bit less murky with an example. Suppose I want to prove the predicate shown here for all non-negative values of n, that is, that the sum of the powers of 2 can be captured by the simple expression shown on the right.
To prove this by induction, I should first really specify the universe, which I will take to be all non-negative integers. Then, I need to show that this is true for the base case, when n is zero. That is easy. I just substitute for n on both sides of the equation, and check that 1 is the same as 2 - 1.
Now for the inductive step. Let's start with the left side of the equation for the n+1 case. I can rewrite this as shown in the first step. Now, I want to rewrite this way because the first term on the right is just the same predicate, but now for argument n. But by induction, I simply want to show that if this version of the predicate is true, so is the version I am considering. So I can replace that summation with the implication from the predicate, and then some simple arithmetic shows that I get the expression I want. I have now shown that if the proposition is true for n, then it must also be true for n+1. And now by induction I can conclude it is true for all legal values of n.



Slide 3.5.8

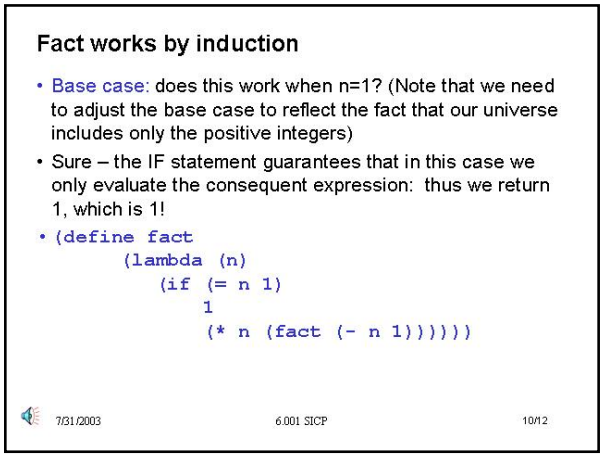
So what we have is a way of proving things by induction. The steps are as shown.
We first define the predicate we want to prove correct, including specifying what the variable denotes and the range of legal values of the variable.
We then prove that the predicate is true in the base case (which we have here assumed is for n equal to 0)
We then prove that the implication holds, and this allows us to conclude that the predicate is always true.

Slide 3.5.9



So finally we can relate this back to programs. Suppose we want to prove that our code for factorial is correct, that is, that it will always run correctly. Note that there is a hidden assumption here, namely that we give it a correct input. Note that here, we have made the unstated assumption that factorial only applies to integers greater than or equal to one. If we provide some other value, all bets are off. In other words, our universe here is positive integers.
So here is our code, and our proposition is that this code correctly computes n! given a legal value for n.

6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.



Slide 3.5.10

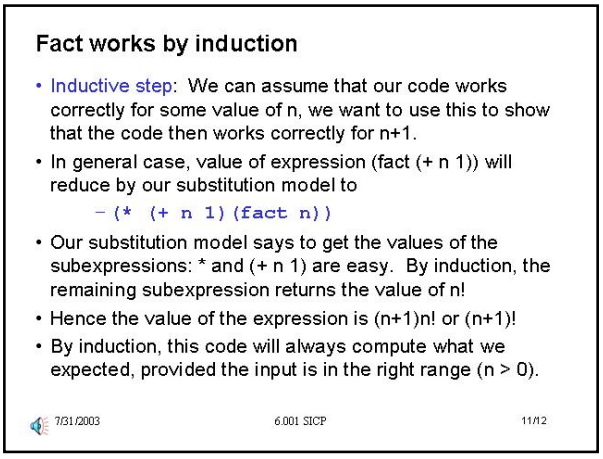
Thus to prove it we first consider the base case. Remember that here our base case is n equal 1, since that is the starting point for our universe of values. Is our proposition valid?

Sure! We know that our "if" statement will in this case take the path through the consequent, and just return the value 1, which we also know is the desired answer.

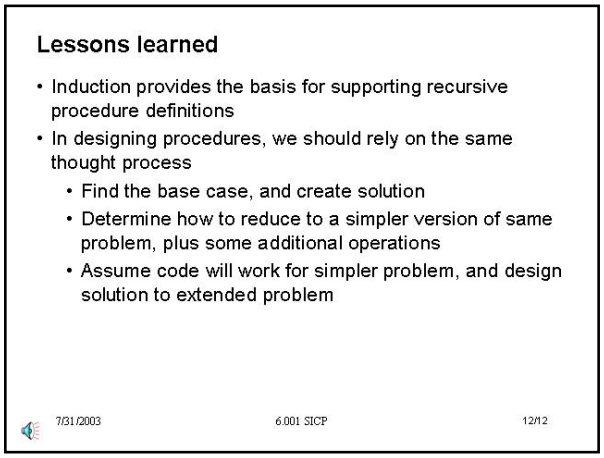
Slide 3.5.11

So what about the inductive step? In this case we can assume that our code works correctly for some arbitrary, but legal, value of n, and we want to show that under that assumption, it also works correctly for n+1.

In this case, we will take the consequent path through the "if" expression. Our substitution model says that the value of this expression is found by finding the values of the subexpressions, then applying the first to the others. Finding the value of * is easy, as is the value of the sum. By induction, we can assume that the value of the last subexpression will be the correct value of factorial.



As a consequence, applying the multiplication operator to the value of n+1 and n! will return (n+1)! and hence we can conclude that this will always produce the right answer!



Slide 3.5.12

Now, let's pull this together. The message to take away from this exercise is that induction provides a basis for understanding, analyzing and proving correctness of recursive procedure definitions.

Moreover, exactly this kind of thinking can be used when we design programs, not just when we analyze them. When given a new problem to solve, it is valuable to identify the base case and find a solution for it. Then, one turns to the issue of breaking the problem down into a simpler version of the same problem, assuming that the code will solve that version, and using that to construct the inductive step: how to solve the full version of the problem.

We hope that you will use this approach as you build your own procedures.