[MUSIC PLAYING]

PROFESSOR: Well, so far we've invented enough programming to do some very complicated things. And you surely learned a lot about programming at this point. You've learned almost all the most important tricks that usually don't get taught to people until they have had a lot of experience. For example, data directed programming is a major trick, and yesterday you also saw an interpreted language.

We did this all in a computer language, at this point, where there was no assignment statement. And presumably, for those of you who've seen your Basic or Pascal or whatever, that's usually considered the most important thing. Well today, we're going to do some thing horrible. We're going to add an assignment statement.

And since we can do all these wonderful things without it, why should we add it? An important thing to understand is that today we're going to, first of all, have a rule, which is going to always be obeyed, which is the only reason we ever add a feature to our language is because there is a good reason. And the good reason is going to boil down to the ability, you now get an ability to break a problem into pieces that are different sets of pieces then you could have broken it down without that, give you another means of decomposition.

However, let's just start. Let me quick begin by reviewing the kind of language that we have now. We've been writing what's called functional programs. And functional programs are a kind of encoding of mathematical truths. For example, when we look at the factorial procedure that you see on the slide here, it's basically two clauses. If n is one, the result is one, otherwise n times factorial n minus one. That's factorial of n. Well, that is factorial of n.

And written down in some other obscure notation that you might have learned in calculus classes, mathematical logic, what you see there is if n equals one, for the result of n factorial is one, otherwise, greater than one, n factorial is n times n minus one factorial.

True statements, that's the kind of language we've been using. And whenever we have true statements of that sort, there is a kind of, a way of understanding how they work which is that such processes can be involved by substitution.

And so we see on the second slide here, that the way we understand the execution implied by those statements in arranged in that order, is that you do successive substitutions of arguments for formal parameters in the body of a procedure.

This is basically a sequence of equalities. Factorial four is four times factorial three. That is four times three times factorial of two and so on. We're always preserving truth. Even though we're talking about true statements, there

might be more than one organization of these true statements to describe the computation of a particular function, the computation of the value of a particular function.

So, for example, looking at the next one here. Here is a way of looking at the sum of n and m. And we did this one by a recursive process. It's the increment of the sum of the decrement of n and m. And, of course, there is some piece of mathematical logic here that describes that. It's the increment of the sum of the decrement of n and m, just like that. So there's nothing particularly magic about that.

And, of course, if we can also look at an iterative process for the same, a program that evolves an iterative process, for the same function. These are two things that compute the same answer. And we have equivalent mathematical truths that are arranged there. And just the way you arrange those truths determine the particular process. In the way choose and arrange them determines the process that's evolved.

So we have the flexibility of talking about both the function to be computed, and the method by which it's computed. So it's not clear we need more. However, today I'm going to this awful thing. I'm going to introduce this assignment operation.

Now, what is this? Well, first of all, there is going to be another kind of kind of statement, if you will, in a programming language called Set! Things that do things like assignment, I'm going to put exclamation points after. We'll talk about what that means in a second. The exclamation point, again like question mark, is an arbitrary thing we attach to the symbol which is the name, has no significance to the system. The only significance is to me and you to alert you that this is an assignment of some sort.

But we're going to set a variable to a value. And what that's going to mean is that there is a time at which something happens. Here's a time. If I have time going this way, it's a time access. Time progresses by walking down the page. Then an assignment is the first thing we have that produces the difference between a before and an after.

All the other programs that we've written, that have no assignments in them, the order in which they were evaluated didn't matter. But assignment is special, it produces a moment in time. So there is a moment before the set occurs and after, such that after this moment in time, the variable has the value, value. Independent of what value it had before, set! changes the value of the variable.

Until this moment, we had nothing that changed. So, for example, one of the things we can think of is that the procedures we write for something like factorial are in fact pretty much identical to the function factorial. Factorial of four, if I write fact4, independent of what context it's in, and independent of how many times I write it, I always get the same answer. It's always 24. It's a unique map from the argument to the answer.

And all the programs we've written so far are like that. However, once I have assignment, that isn't true. So, for example, if I were to define count to be one. And then I'm going to define also a procedure, a simple procedure called demo, which takes argument x and does the following operations. It first sets x to x plus one. My gosh, this looks just like FORTRAN, right-- in a funny syntax. And then add to x count, Oh, I just made a mistake.

I want to say, set! count to one plus count. It's this thing defined here. And then plus x count. Then I can try this procedure. Let's run it. So, suppose I get a prompt and I say, demo three.

Well, what happens here? The first thing that happens is count is currently one. Currently, there is a time. We're talking about time. x gets three. At this moment, I say, oh yes, count is incremented, so count is two. two plus three is five. So the answer I get out is five. Then I say, demo of say, three again.

What do I get? Well, now count is two, it's not one anymore, because I have incremented it. But now I go through this process, three goes into x, count becomes one plus count, so that's three now. The sum of those two is six, so the answer is six.

And what we see is the same expression leads to two different answers, depending upon time. So demo is not a function, does not compute a mathematical function. In fact, you could also see why now, of course, this is the first place where the substitution model isn't going to work. This kills the substitution model dead.

You know, with quotation there were some little problems that a philosopher might notice with the substitutions, because you have to worry about what deductions you can make when you substitute into quotes, if you're allowed to do that at all. But here the substitution model is dead, can't do anything at all.

Because, supposing I wanted to use a substitution model to consider substituting for count? Well, my gosh, if I substitute for here and here, they're different ones. It's not the same count any more. I get the wrong answer. The substitution model is a static phenomenon that describes things that are true and not things that change. Here, we have truths that change.

OK, Well, before I give you any understanding of this, this is very bad. Now, we've lost our model of computation. Pretty soon, I'm going to have to build you a new model of computation. But ours plays with this, just now, in an informal sense. Of course, what you already see is that when I have something like assignment, the model that we're going to need is different from the model that we had before in that the variables, those symbols like count, or x are no longer going to refer to the values they have, but rather to some sort of place where the value restored.

We're going to have to think that way for a while. And it's going to be a very bad thing and cause a lot of trouble.

And so, as I said, the very fact that we're inventing this bad thing, means that there had better be a good reason for it, otherwise, just a waste of time and a lot of effort.

Let's just look at some of it just to play. Supposing we write down the functional version, functional meaning in the old style, of factorial by an iterative process. Factorial of n, we're going to iterate of m and i, which says if i is greater than n, then the result is m, otherwise, the result of iterating the product of i and m. So m is going to be the product that I'm accumulating. m is the product. And the count I'm going to increase by one. Plus, ITER, ELSE, COND, define.

I'm going to start this up. And these days, you should have no trouble reading something like this. What I have here is a product there being accumulated and a counter. I start them up both at one. I'm going to buzz the counter up, i goes to i plus one every time around. But that's only our putting a time on the process, each of this is just a set of truths, true rules. And m is going to get a new values of i and m, i times m each time around, and eventually i is going to be bigger than n, in which case, the answer's going to be m.

Now, I'm speaking to you, use time in this. That's just because I know how the computer works. But I didn't have to. This could be a purely mathematical description at this point, because substitution will work for this.

But let's set right down a similar sort of program, using the same algorithm, but with assignments. So this is called the functional version. I want to write down an imperative version. Factorial of n. I'm going to create my two variables. Let i initialize itself to one, and m be initialized to one, similar.

We'll create a loop which has COND greater than i, and if i is greater than n, we're done. And the result is m, the product I'm accumulating. Otherwise, I'm going to write down three things to do. I'm going to set! m to the product of i and m, set! i to the sum of i and one, and go around the loop again. Looks very familiar to you FORTRAN programmers. ELSE, COND, define, funny syntax though. Start the loop up, and that's the program.

Now, this program, how do we think about it? Well, let's just say what we're seeing here. There are two local variables, i and m, that have been initialized to one. Every time around the loop, I test to see if i is greater than n, which is the input argument, and if so, the result is the product being accumulated in m. However, if it's not the end of the loop, if I'm not done, then what I'm going to do is change the product to be the result of multiplying i times the current product.

Which is sort of what we were doing here. Except here I wasn't changing. I was making another copy, because the substitution model says, you copy the body of the procedure with the arguments substituted for the formal parameters. Here I'm not worried about copying, here I've changed the value of m. I also then change the value of i to i plus one, and go buzzing around.

Seems like essentially the same program, but there are some ways of making errors here that didn't exist until today. For example, if I were to do the horrible thing of not being careful in writing my program and interchange those two assignments, the program wouldn't compute the same function.

I get a timing error because there's a dependency that m depends upon having the last value of i. If I try to i first, then I've got the wrong value of i when I multiply by m. It's a bug that wasn't available until this moment, until we introduced something that had time in it.

So, as I said, first we need a new model of computation, and second, we have to be damn good reason for doing this kind of ugly thing. Are there any questions? Speak loudly, David.

AUDIENCE: I'm confused about, we've introduced set now, but we had let before and define before. I'm confused about the difference between the three. Wouldn't define work in the same situation as set if you introduced it a bit?

PROFESSOR: No, define is intended for setting something once the first time, for making it. You've never seen me write on a blackboard two defines in a row whose intention was to change the old value of some variable to a new one.

AUDIENCE: Is that by convention or--

PROFESSOR: No, it's intention. The answer is that, for example, internal to a procedure, two defines in a row are illegal, two defines in a row of the same variable. x can't be defined twice. Whether or not a system catches that error is a different question, but I legislate to you that define happens once on anything.

Now, indeed, in interactive debugging, we intend that you interacting with your computer will redefine things, and so there's a special exception made for interactive debugging. But define is intended to mean to set up something which will be forever that value after that point. It's as if all the defines were done at the beginning. In fact, the only legal place to put a define in Scheme, internal to a procedure, is just at the beginning of a lambda expression, the beginning of the body of a procedure.

Now, let of course does nothing like either of that. I mean, if you look at what's happening with a let, this happens again exactly once. It sets up a context where i and m are values one and one. That context exists throughout this scope, this region of the program. However, you don't think of that let as setting i again. It doesn't change it. i never changes because of the let. i gets created because of let.

In fact, the let is a very simple idea. Let does nothing more, Let a variable one to have value one; I'll write this down a little bit more neatly; Let's write, var one have value, the value of expression e1, and variable two, have this value of the expression e2, in an expression e3, is the same thing as a procedure of var one and var two, the

formal parameters, and e3 being the body, where var one is bound to the value of e1, and var two gets the value of e2.

So this is, in fact, a perfectly understandable thing from a substitution point of view. This is really the same expression written in two different ways. In fact, the way the actual system works is this gets translated into this before anything happens.

AUDIENCE: OK, I'm still unclear as then what makes the difference between a let and a define. They could--

PROFESSOR: A define is a syntactic sugar, whereby, essentially a bunch of variables get created by lets and then set up once. OK, time for the first break, I think. Thank you. [MUSIC PLAYING]

Well let's see. I now have to rebuild the model of computation, so you understand how some such mechanical mechanism could work that can do what we've just talked about. I just recently destroyed your substitution model. Unfortunately, this model is significantly more complicated than the substitution model.

It's called the environment model. And I'm going to have to introduce some terminology, which is very good terminology for you to know anyway. It's about names. And we're going to give names to the kinds of names things have and the way those names are used. So this is a meta-description, if you will. Anyway, there is a pile of an unfortunate terminology here, but we're going to need this to understand what's called the environment model. We're about to do a little bit of boring, dog-work here.

Let's look at the first transparency. And we see a description of a word called bound. And we're going to say that a variable, v, is bound in an expression, e, if the meaning of e is unchanged by the uniform replacement of a variable w, not occurring in e, for every occurrence of v in e. Now that's a long sentence, so, I think, I'm going to have to say a little bit about that before we even fool around at all here.

Bound variables we're talking about here. And you've seen lots of them. You may not know that you've seen lots of them. Well, I suppose in your logic you saw a logical variables like, for every x there exists a y such that p is true of x and y from your calculus class. This variable, x, and this variable, y, are bound, because the meaning of this expression does not depend upon the particular letters I used to describe x and y. If I were to change the w for x, then said for every w there exists a y such that p is true of w and y, it would be the same sentence. That's what it means.

Or another case of this that you've seen is integral say, from 0 to one of dx over one plus x square. Well that's something you see all the time. And this x is a bound variable. If I change that to a t, the expression is still the same thing. This is a 1/4 of the arctan of one or something like that. Yes, that's the arctan of one. So bound

variables are actually fairly common, for those of you who have played a bit with mathematics.

Well, let's go into the programming world. Instead of the quantifier being something like, for every, or there exists, or integral, a quantifier is a symbol that binds a variable. And we are going to use the quantifier lambda as being the essential thing that binds variables.

And so we have some nice examples here like that procedure of one argument y which does the following thing. It calls the procedure of one argument x, which multiplies x by y, and applies that to three. That procedure has the property there of two bound variables in it, x and y. This quantifier, lambda here, binds this y, and this quantifier, lambda, binds that x. Because, if I were to take an arbitrary symbol does not occur in this expression like w and replace all y's with w's in this expression, the expression is still the same, the same procedure.

And this is an important idea. The reason why we had such things like that is a kind of modularity. If two people are writing programs, and they work together, it shouldn't matter what names they use internal to their own little machines that they're building. And so, what I'm really telling you there, is that, for example, this is equivalent to that procedure of one argument y which uses that procedure of one argument d which multiplies z by y. Because nobody cares what I used in here. It's a nice example.

On the other hand, I have some variables that are not bound. For example, that procedure of one argument x which multiplies x by y. In this case, y is not bound. Supposing y had the value three, and z had the value four, then this procedure would be the thing that multiplies its argument by three. If I were to replace every instance of y with z, I would have a different procedure which multiplies every argument that's given by four.

And, in fact, we have a name for such a variable. Here, we say that a variable, v, is free in the expression, e, if the meaning of the expression, e, is changed by the uniform replacement of a variable, w, not occurring in e for every occurrence of v and e.

So that's why this variable over here, y, is a free variable. And so free variables in this expression-- And other examples of that is that procedure of one argument y, which is just what we had before, which uses that procedure of one argument x that multiplies x by y-- use that on three. This procedure has a free variable in it which is asterisk. See, because, if that has a normal meaning of multiplication, then if I were to replace uniformly all asterisks with pluses, then the meaning of this expression would change. That's what you mean by a free variable.

So, so far you've learned some logician words which describe the way names are used. Now, we have to do a little bit more playing around here, a little bit more. I want to tell you about the regions are over which variables are defined.

You see, we've been very informal about this up till now, and, of course, many of you have probably understood very clearly or most of you, that the x that's being declared here is defined only in here. This x is the defined only in here, and this y is defined only in here. We have a name for such an idea. It's called a scope.

And let me give you another piece of terminology. It's a long story. If x is a bound variable in e, then there is a lambda expression where it is bound. So the only way you can get a bound variable ultimately is by lambda expression. Then you may worry, does define quite an exception to this? And it turns out, we could always arrange things so you don't need any defines. And we'll see that in a while. It's a very magical thing. So define really can go away. The really, only thing that makes names is lambda . That's its job. And what's so amazing about a lot of things is you can compute with only lambda.

But, in any case, a lambda expression has a place where it declares a variable. We call it the formal parameter list or the bound variable list. We say that the lambda expression binds-- so it's a verb-- binds the variables declared in it's found variable list. In addition, those parts of the expression where the variable is defined, which was declared by some declaration, is called the scope of that variable. So these are scopes. This is the scope of y. And this is the scope of x-- that sort of thing.

OK, well, now we have enough terminology to begin to understand how to make a new model for computation, because the key thing going on here is that we destroyed the substitution model, and we now have to have a model that represents the names as referring to places. Because if we are going to change something, then we have a place where it's stored.

You see, if a name only refers to a value, and if I tried to change the name's meaning, well, that's not clear. There's nothing that is the place that that name referred to. How am I really saying it? There is nothing shared among all of the instances of that name. And what we really mean, by a name, is that we fan something out. We've given something a name, and you have it, and you have it, because I'm given you a reference to it, and I've given you a reference to it. And we'll see a lot about that.

So let me tell you about environments. I need the overhead projection machine, thank you. And so here is a bunch of environment structures. An environment is a way of doing substitutions virtually. It represents a place where something is stored which is the substitutions that you haven't done. It's a place where everything accumulates, where the names of the variables are associated with the values they have such that when you say, what dose this name mean, you look it up in an environment.

So an environment is a function, or a table, or something like that. But it's a structured sort of table. It's made out of things called frames. Frames are pieces of environment, and they are chained together, in some nice ways, by

what's called parent links or something like that.

So here, we have an environment structure consisting of three environments, basically, a, b, and c. d is also an environment, but it's the same one, they share. And that's the essence of assignment. If I change a variable, a value of a valuable that lives here, like that one, it should be visible from all places that you're looking at it from. Take this one, x. If I change the x to four, it's visible from other places. But I'm not going to worry about that right now. We're going to talk a lot about that in a little while.

What do we have here? Well, these are called frames. Here is a frame, here's a frame, and here's a frame. a is an environment which consists of the table which is frame two, followed by the table labeled frame one. And, in this environment, in say this environment, frame two, x and y are bound. They have values. Sorry, in frame one-- In frame two, z is bound, and x is bound, and y is bound, but the value of x that we see, looking from this point of view, is this x. It's x is seven, rather than this one which is three. We say that this x shadows this x. From environment three-- from frame three, from environment b, which refers to frame three, we have variables n and y bound and also x. This y shadow this one. So the value, looking from this point of view, of y is two. The value for looking from this point of view and m is one. And the value, looking from this point of view, of x is three.

So there we have a very simple environment structure made out of frames. These correspond to the applications of procedures. And we'll see that in a second. So now I have to make you some other nice little structure that we build.

Next slide, we see an object, which I'm going to draw procedures. This is a procedure. A procedure is made out of two parts. It's sort of like a cons. However, it's the two parts. The first part refers to some code, something that can be executed, a set of instructions, if you will. You can think of it that way. And the second part is the environment. The procedure is the whole thing. And we're going to have to use this to capture the values of the free variables that occur in the procedure. If a variable occurs in the procedure it's either bound in that procedure or free. If it's bound, then the value will somehow be easy to find. It will be in some easy environment to get at. If it's free, we're going to have to have something that goes with the procedure that says where we'll go look for its value. And the reasons why are not obvious yet, but will be soon.

So here's a procedure object. It's a composite object consisting of a piece of code and a environment structure. Now I will tell you the new rules, the complete new rules, for evaluation. The first rule is-- there's only two of them. These correspond to the substitution model rules. And the first one has to do with how do you apply a procedure to its arguments? And a procedural object is applied to a set of arguments by constructing a new frame. That frame will contain the mapping of the former parameters to the actual parameters of the arguments that were supplied in the call.

As you know, when we make up a call to a procedure like lambda x times x y, and we call that with the argument three, then we're going to need some mapping of x to three. It's the same thing as later substituting, if you will, the three for the x in the old model. So I'm going to build a frame which contains x equals three as the information in that frame.

Now, the body of the procedure will then have to be evaluated which is this. I will be evaluated in an environment which is constructed by adjoining the new frame that we just made to the environment which was part of the procedure that we applied.

So I'm going to make a little example of that here. Supposing I have some environment. Here's a frame which represents it. And some procedure-- which I'm going to draw with circles here because it's easier than little triangles-- Sorry, those are rhombuses, rhomboidal little pieces of fruit jelly or something.

So here's a procedure which takes this environment. And the procedure has a piece of code, which is a lambda expression, which binds x and y and then executes an expression, e. And this is the procedure. We'll call it p. I wish to apply that procedure to three and four. So I want to do p of three and four.

What I'm going to do, of course, is make a new frame. I build a frame which contains x equals three, and y equals four. I'm going to connect that frame to this frame over here. And then this environment, with I will call b, is the environment in which I will evaluate the body of e. Now, e may contain references to x and y and other things. x and y will have values right here. Other things will have their values here.

How do we get this frame? That we do by the construction of procedures which is the other rule. And I think that's the next slide. Rule two, when a lambda expression is evaluated, relative to a particular environment-- See, the way I get a procedure is by evaluating the lambda expression.

Here's a lambda expression. By evaluating it, I get a procedure which I can apply to three. Now this lambda expression is evaluated in an environment where y is defined. And I want the body of this which contains a free version of y. y is free in here, it's bound over the whole thing, but it's free over here. I want that y to be this one. I evaluate this body of this procedure in the environment where y was created. That's this kind of thing, because that was done by application.

Now, if I ever want to look up the value of y, I have to know where it is. Therefore, this procedural was created, the creation of the procedure which is the result of evaluating that lambda expression had better capture a pointer or remember the frame in which y was bound. So that's what this rule is telling us.

So, for example, if I happen to be evaluating a lambda expression, lambda expression in e, lambda of say, x and

y, let's call it g in e, evaluating that. Well, all that means is I now construct a procedure object. e is some environment. e is something which has a pointer to it. I construct a procedure object that points up to that environment, where the code of that is a lambda expression or whatever that translates into. And this is the procedure.

So this produces for me-- this object here, this environment pointer, captures the place where this lambda expression was evaluated, where the definition was used, where the definition was used to make a procedure, to make the procedure. So it picks up the environment from the place where that procedure was defined, stores it in the procedure itself, and then when the procedure is used, the environment where it was defined is extended with the new frame.

So this gives us a locus for putting where a variable has a value. And, for example, if there are lots of guys pointing in at that environment, then they share that place. And we'll see more of that shortly.

Well, now you have a new model for understanding the execution of programs. I suppose I'll take questions now, and then we'll go on and use that for something.

AUDIENCE: Is it right to say then, the environment is that linked chain of frames--

PROFESSOR: That's right.

AUDIENCE: starting with-- working all the way back?

PROFESSOR: Yes, the environment is a sequence of frames linked together. And the way I like to think about it, it's the pointer to the first one, because once you've got that you've got them all. Anybody else?

AUDIENCE: Is it possible to evaluate a procedure or to define a procedure in two different environments such that it will behave differently, and have pointers to both--

PROFESSOR: Oh, yes. The same procedure is not going to have two different environments. The same code, the same lambda expression can be evaluated in two environments producing two different procedures. Each procedure--

AUDIENCE: Their definition has the same name. Their operation--

PROFESSOR: The definition is written the same, with the same characters. I can evaluate that set of characters, whatever, that list structure that defines, that is the textual representation. I can evaluate that in two different environments producing two different procedures. Each of those procedures has its own local sets of variables, and we'll see that right now. Anybody else? OK, thank you. Let's take a break. [MUSIC PLAYING]

Well, now I've done this terrible thing to you. I've introduced a very complicated thing, assignment, which destroys most of the interesting mathematical properties of our programs. Why should I have done this? What possible good could this do? Clearly not a nice thing, so I better have a good excuse. Well, let's do a little bit of playing, first of all, with some very interesting programs that have assignment. Understand something special about them that makes them somewhat valuable.

Start with a very simple program which I'm going to call make-counter. I'm going to define make-counter to be a procedure of one argument n which returns as its value a procedure of no arguments-- a procedure that produces a procedure-- which sets n to the increment of n and returns that value of n.

Now we're going to investigate the behavior of this. It's a sort of interesting thing. In order to investigate the behavior, I have to make an environment model, because we can't understand this any other way. So let's just do that.

We start out with some sort of-- let's say there is a global environment that the machine is born with. Global we'll call it. And it's going to have in it a bunch of initial things. We all know what it's got. It's got things in it like say, plus, and times, and quotient, and difference, and CAR, and et cetera, lots of things. I don't know what they are, some various squiggles that are the things the machine is born with. And by doing the definition here, what I plan to do--

Well, what am I doing? I'm doing this relative to the global environment. So here's my environment pointer. In order to do that I have to evaluate this lambda expression. That means I make a procedure object.

So I'm going to make a procedure object here. And the procedure object has, as the place it's defined, the global environment. The procedure object contains some code that represents a procedure of one argument n which returns a procedure of no arguments which does something. And the define is a way of changing this environment, so that I now add to it a make-counter, a special rule for the special thing defined. But what that is, is it gives me that pointer to that procedure. So now the global environment contains make-counter as well.

Now, we're going to do some operations. I'm going to use this to make some counters. We'll see what a counter is. So let's define c1 to be a counter beginning at 0. Well, we know how to do this now, according to the model. I have to evaluate the expression make-counter in the global environment, make-counter of 0.

Well, I look up make-counter and see that it's a procedure. I'm going to have to apply that procedure. The way I apply the procedure is by constructing a frame. So I construct a frame which has a value for n in it which is 0, and the parent environment is the one which is the environment of definition of make-counter. So I've made an environment by applying make-counter to 0.

Now, I have to evaluate the body of make-counter, which is this lambda expression, in that environment. Well evaluating this body, this body is a lambda expression. Evaluate a lambda expression means make a procedure object. So I'm going to make a procedure object.

And that procedure object has the environment it was defined in being that, where n was defined to be 0. And it has some code, which is the procedure of no arguments which does something, that sets something, and returns n. And this thing is going to be the object, which in the global environment, will have the name c1. So we construct a name here, c1, and say that equals that.

Now, but also make another counter, c2 to be make-counter say, starting with 10. Then I do essentially the same thing. I apply the make-counter procedure, which I got from here, to make another frame with n being 10. That frame has the global environment as its parent. I then construct a procedure which has that as it's frame of definition. The code of it is the procedure of no arguments which does something. And it does a set, and so on. And n comes out. And c2 is this.

Well, you're already beginning to see something fairly interesting. There are two n's here. They are not one n. Each time I called make-counter, I made another instance of n. These are distinct and separate from each other. Now, let's do some execution, use those counters. I'm going to use those counters.

Well, what happens if I say, c1 at this point? Well, I go over here, and I say, oh yes, c1 is a procedure. I'm going to call this procedure on no arguments, but it has no parameters. That's right. What's its body? Well, I have to look over here, because I didn't write it down. It said, set n to one plus n and return n, increment n.

Well, the n it sees is this one. So I increment that n. That becomes one, and I return the value one. Supposing I then called c2. Well, what do I do? I say c2 is this procedure which does the same thing, but here's the n. It becomes 11. And so I have an 11 which is the value. I then can say, let's try c1 again. c1 is this, that's two, so the answer is two. And c2 gives me a 12 by the same method, by walking down here looking at that and saying, here's the n, I'm incrementing.

So what I have are computational objects. There are two counters, each with its own independent local state. Let's talk about this a little. This is a strange thing. What's an object? It's not at all obvious what an object is. We like to think about objects, because it's economical to think that way. It's an intellectual economy. I am an object. You are an object. We are not the same object.

I can divide the world into two parts, me and you, and there's other things as well, such that most of the things I might want to discuss about my workings do not involve you, and most of the things I want to discuss about your workings don't involve me. I have a blood pressure, a temperature, a respiration rate, a certain amount of sugar in

my blood, and numerous, thousands, of state variables-- millions actually, or I don't know how many-- huge numbers of state variables in the physical sense which represent the state of me as a particle, and you have gazillions of them as well.

And most of mine are uncoupled to most of yours. So we can compute the properties of me without worrying too much about the properties of you. If we had to work about both of us together, than the number of states that we have to consider is the product of the number of states you have and the number of states I have. But this way it's almost a sum.

Now, indeed there are forces that couple us. I'm talking to you and your state changes. I'm looking at you and my state changes. Some of my state variables, a very few of them, therefore, are coupled to yours. If you were to suddenly yell very loud, my blood pressure would go up.

However, and it may not be always appropriate to think about the world as being made out of independent states and independent particles. Lots of the bugs that occur in things like quantum mechanics, or the bugs in our minds that occur when we think about things like quantum mechanics, are due the fact that we are trying to think about things being broken up into independent pieces, when in fact there's more coupling than we see on the surface, or that we want to believe in, because we want to compute efficiently and effectively. We've been trained to think that way.

Well, let's see. How would we know if we had objects at all? How can we tell if we have objects? Consider some possible optical illusions. This could be done. These pieces of chalk are not appropriately identical, but supposing you couldn't tell the difference of them by looking at them. Well, there's a possibility that this all a game I'm playing with mirrors. It's really the same piece of chalk, but you're seeing two of them. How would you know if you're seeing one or two? Well, there's only one way I know. You grab one of them and change it and see if the other one changed. And it didn't, so there's two of them.

And, on the other hand, there is some other screwy properties of things like that. Like, how do we know if something changed? We have to look at it before and after the change. The change is an assignment, it's a moment in time. But that means we have to know it was the same one that we're looking at. So some very strange, and unusual, and obscure, and-- I don't understand the problems associated with assignment, and change, and objects. These could get very, very bad.

For example, here I am, I am a particular person, a particular object. Now, I can take out my knife, and cut my fingernail. A piece of my fingernail has fallen off onto the table. I believe I am the same person I was a second ago, but I'm not physically the same in the slightest.

I have changed. Why am I the same? What is the identity of me? I don't know. Except for the fact that I have some sort of identity. And so, I think by introducing assignment and objects, we have opened ourselves up to all the horrible questions of philosophy that have been plaguing philosophers for some thousands of years about this sort of thing. It's why mathematics is a lot cleaner.

Let's look at the best things I know to say about actions and identity. We say that an action, a, had an effect on an object, x, or equivalently, that x was changed by a, if some property, p, which was true of x before a, became false of x after a. Let's test. It still means I have to have the x before and after. Or, the other way of saying this is, we say that two objects x and y are the same for any action which has an effect on x has the same effect on y.

However, objects are very useful, as I said, for intellectual economy. One of the things that's incredibly useful about them, is that the world is, we like to think about, made out of independent objects with independent local state. We like to think that way, although it isn't completely true.

When we want to make very complicated programs that deal with such a world, if we want those programs to be understandable by us and also to be changeable, so that if we change the world we change the program only a little bit, then we want there to be connections, isomorphism, between the objects in the world and the objects in our mental model. The modularity of the world can give us the modularity in our programming. So we invent things called object-oriented programming and things like that to provide us with that power.

But it's even easier. Let's play a little game. I want to play a little game, show you an even easier example of where modularity can be enhanced by using an assignment statement, judiciously. One thing I want to enforce and impress on you, is don't use assignment statements the way you use it in FORTRAN or Basic or something or Pascal, to do the things you don't have to do with it. It's not the right way to think for most things. Sometimes it's essential, or maybe it's essential. We'll see more about that too.

OK, let me show you a fun game here. There was mathematician by the name of Cesaro-- or Cesaro, Cesaro I suppose it is-- who figured out a clever way of computing pi. It turns out that if I take to random numbers, two integers at random, and compute the greatest common divisor, their greatest common divisor is either one or it's not one. If it's one, then they have no common divisors. If their greatest common divisor is one-- the probability that two random numbers, two numbers chosen at random, has as greatest common divisor one is related to pi.

In fact-- yes, it's very strange-- of course there are other ways of computing pi, like dropping pins on flags, and things like that, and sort of the same kind of thing. So the probability of that the GCD of number one and number two, two random numbers chosen, is 6 over pi squared. I'm not going to try to prove that. It's actually not too hard and sort of fun.

How would we estimate such probability? Well, the way we do that, the way we estimate probabilities, is by doing lots of experiments, and then computing the ratios of the ones that come out one way to the total number of experiments we do. It's called Monte Carlo, and it's useful in other contexts for doing things like integrals where you have lots and lots of variables-- the space which is limiting the dimensions you are doing you integral in.

But going back to here, Let's look at this slide, We can use Cesaro's method for estimating pi with n trials by taking the square root of six over a Monte Carlo, a Monte Carlo experiment with n trials, using Cesaro's experiment, where Cesaro's experiment is the test of whether the GCD of two random numbers-- And you can see that I've already got some assignments in here, just by what I wrote. The fact that this word rand, in parentheses, therefore, that procedure call, yields a different value than this one, at least that's what I'm assuming by writing this this way, indicates that this is not a function, that there's internal state in it which is changing. If the GCD of those two random numbers is equal to one, that's the experiment.

So here I have an experimental method for estimating the value of pi. Where, I can easily divide this problem into two parts. One is the specific Monte Carlo experiment of Cesaro, which you just saw, and the other is the general technique of doing Monte Carlo experiments. And that's what this is.

If I want to do Monte Carlo experiments with n trials, a certain number of trials, and a particular experiment, the way I do that is I make a little iterative procedure which has variable the number of trials remaining and the number trials that have been passed, that I've gotten true. And if the number remaining is 0, then the answer is the number past divided by this whole number of trials, was the estimate of the probability. And if it's not, if I have more trials to do, then let's do one.

We do an experiment. We call the procedure which is experiment on no arguments. We do the experiment and then, if that turned out to be true, we go around the loop decrementing the number of experiments we have to do by one and incrementing the number that were passed.

And if the experiment was false, we just go around the loop decrementing the number of experiments remaining and keeping the number passed the same. We start this up iterating over the total number of trials with 0 experiments past. A very elegant little program. And I don't have to just do this with Cesaro's experiment, it could be lots of Monte Carlo experiments I might do.

Of course, this depends upon the existence of some sort of random number generator. And random number generators generally look something like this. There is a random number generator-- is in fact a procedure which is going to do something just like the counter. It's going to update an x to the result of applying some function to x, where this function is some screwy kind of function that you might find out in Knuth's books on the details of programming. He does these wonderful books that are full of the details of programming, because I can't

remember how to make a random number generator, but I can look it up there, and I can find out.

And then, eventually, I return the value of x which is the state variable internal to the random number generator. That state variable is initialized somehow, and has a value. And this procedure is defined in the context where that variable is bound. So this is a hidden piece of local state that you see here. And this procedure is defined in that context.

Now, that's a very simple thing to do. And it's very nice. Supposing, I didn't want to use assignments. Supposing, I wanted to write this program without assignments. What problems would I have? Well, let's see. I'd like to use the overhead machine here, thank you.

First of all, let's look at the whole thing. It's a big story. Unfortunately, which tells you there is something wrong. It's at least that big, and it's monolithic. You don't have to understand or look at the text there right now to see that it's monolithic. It isn't a thing which is Cesaro's experiment. It's not pulled out from the Monte Carlo process. It's not separated.

Let's look why. Remember, the constraint here is that every procedure return the same value for the same arguments. Every procedure represents a function. That's a different kind of constraint. Because when I have assignments, I can change some internal state variable.

So let's see how that causes things to go wrong. Well, start at the beginning. The estimate of pi looks sort of the same. What I'm doing is I take the square root of six over the random GCD test applied to n, whereas that's what this is.

But here, we are beginning to see something funny. The random GCD test of a certain number of trials is just like we had before, an iteration on the number of trials remaining, the number of trials that have been passed, and another variable x.

What's that x? That x is the state of the random number generator. And it is now going to be used here. The same random update function that I have over here is the one I would have used in a random number generator if I were building it the other way, the one I get out of Knuth's books. x is going to get transformed into x1, I need two random numbers. And x1 is going to get transformed into x2, I have two random numbers. I then have to do exactly what I did before. I take the GCD of x1 x2. If that's one, then I go around the loop with x2 being the next value of x.

You see what's happened here is that the state of the random number generator is no longer confined to the insides of the random number generator. It has leaked out. It has leaked out into my procedure that does the

Monte Carlo experiment. But what's worse than that, is it's also, because it was contained inside my experiment itself, Cesaro, it leaked out of that too. Because Cesaro called twice, has to have a different value each time, if I going to have a legitimate experimental test. So Cesaro can't be a function either, unless I pass it the seed of the random number generator that is going to go wandering around.

So unfortunately, the seed of random number generator has leaked out into Cesaro, from the random number generator, that's leaked into the Monte Carlo experiment. And, unfortunately, my Monte Carlo experiment here is no longer general. The Monte Carlo experiment here knows how many random numbers I need to do the experiment.

That's sort of horrible. I lost an ability to decompose a problem into pieces, because I wasn't willing to accept the little loop of information, the feedback process, that happens inside the random number generator before that was made by having an assignment to a state variable that was confined to the random number generator. So the fact that the random number generator is an object, with an internal state variable, it's affected by nothing, but it'll give you something, and it will apply it's force to you, that was what we're missing now.

OK, well I think we've seen enough reason for doing this, and it all sort of looks very wonderful. Wouldn't it be nice if assignment was a good thing and maybe it's worth it, but I'm not sure. As Mr. Gilbert and Sullivan said, things are seldom what they seem, skim milk masquerades as cream.

Are there any questions? Are there any philosophers here? Anybody want to argue about objects? You're just floored, right? And you haven't done your homework yet. You haven't come up with a good question. Oh, well. Sure, thank you. Let's take the long break now.