PROFESSOR: Well today we're going to learn about something quite amazing. We're going to understand what we mean by a program a little bit more profoundly than we have up till now. Up till now, we've been thinking of programs as describing machines.

So for example, looking at this still store, we see here is a program for factorial. And what it is, is a character string description, if you will, of the wiring diagram of a potentially infinite machine. And we can look at that a little bit and just see the idea. That this is a sort of compact notation which says, if n is 0, the result is one.

Well here comes n coming into this machine, and if it's 0, then I control this switch in such a way that the switch allows the output to be one. Otherwise, it's n times factorial of n minus one. Well, I'm computing factorial of n minus one and multiplying that by n, and, in the case that it's not 0, this switch makes the output come from there.

Of course, this is a machine with a potentially infinite number of parts, because factorial occurs within factorial, so we don't know how deep it has to be. But that's basically what our notation for programs really means to us at this point. It's a character string description, if you will, of a wiring diagram that could also be drawn some other way.

And, in fact, many people have proposed to me, programming languages look graphical like this. I'm not sure I believe there are many advantages. The major disadvantage, of course, is that it takes up more space on a page, and, therefore, it's harder to pack into a listing or to edit very well.

But in any case, there's something very remarkable that can happen in the competition world which is that you can have something called a universal machine. If we look at the second slide, what we see is a special machine called eval.

There is a machine called eval, and I'm going to show it to you today. It's very simple. What is remarkable is that it will fit on the blackboard. However, eval is a machine which takes as input a description of another machine.

It could take the wiring diagram of a factorial machine as input. Having done so, it becomes a simulator for the factorial machine such that, if you put a six in, out comes a 720. That's a very remarkable sort of machine. And the most amazing part of it is that it fits on a blackboard.

By contrast, one could imagine in the analog electronics world a very different machine, a machine which also was, in some sense, universal, where you gave a circuit diagram as one of the inputs, for example, of this little low-pass filter, one-pole low-pass filter. And you can imagine that you could, for example, scan this out-- the scan lines are the signal that's describing what this machine is to simulate-- then the analog of that which is made out of electrical circuits, should configure itself into a filter that has the frequency response specified by the circuit

diagram. That's a very hard machine to make, and, surely, there's no chance that I could put it on a blackboard.

So we're going to see an amazing thing today. We're going to see, on the blackboard, the universal machine. And we'll see that among other things, it's extremely simple.

Now, we're getting very close to the real spirit in the computer at this point. So I have to show a certain amount of reverence and respect, so I'm going to wear a suit jacket for the only time that you'll ever see me wear a suit jacket here. And I think I'm also going to put on an appropriate hat for the occasion. Now, this is a lecturer which I have to warn you-- let's see, normally, people under 40 and who don't have several children are advised to be careful. If they're really worried, they should leave. Because there's a certain amount of mysticism that will appear here which may be disturbing and cause trouble in your minds.

Well in any case, let's see, I wish to write for you the evaluator for Lisp. Now the evaluator isn't very complicated. It's very much like all the programs we've seen already. That's the amazing part of it. It's going to be-- and I'm going to write it right here-- it's a program called eval. And it's a procedure of two arguments in expression of an environment. And like every interesting procedure, it's a case analysis.

But before I start on this, I want to tell you some things. The program we're going to write on the blackboard is ugly, dirty, disgusting, not the way I would write this is a professional. It is written with concrete syntax, meaning you've got really to use lots of CARs and CDRs which is exactly what I told you not to do. That's on purpose in this case, because I want it to be small, compact, fit on the blackboard so you can get the whole thing. So I don't want to use long names like I normally use. I want to use CAR-CDR because it's short.

Now, that's a trade-off. I don't want you writing programs like this. This is purely for an effect. Now, you're going to have to work a little harder to read it, but I'm going to try to make it clear as I'm writing it. I'm also-- this is a pretty much complete interpreter, but there's going to be room for putting in more things-- I'm going to leave out definition and assignment, just because they are not essential, for a mathematical reason I'll show you later and also they take up more space.

But, in any case, what do we have to do? We have to do a dispatch which breaks the types of expressions up into particular classes. So that's what we're going to have here. Well, what expressions are there? Let's look at the kinds of expressions.

We can have things like the numeral three. What do I want that to do? I can make choices, but I think right now, I want it to be a three. That's what I want. So that's easy enough. That means I want, if the thing is a number, the expression, that I want the expression itself as the answer.

Now the next possibility is things that we represent as symbols. Examples of symbols are things like x, n, eval, number, x. What do I mean them to be? Those are things that stand for other things. Those are the variables of our language.

And so I want to be able to say, for example, that x, for example, transforms to it's value which might be three. Or I might ask something like car. I want to have as its value-- be something like some procedure, which I don't know what is inside there, perhaps a machine language code or something like that. So, well, that's easy enough. I'm going to push that off on someone else. If something is a symbol, if the expression is a symbol, then I want the answer to be the result, looking up the expression in the environment.

Now the environment is a dictionary which maps the symbol names to their values. And that's all it is. How it's done? Well, we'll see that later. It's very easy. It's easy to make data structures that are tables of various sorts. But it's only a table, and this is the access routine for some table.

Well, the next thing, another kind of expression-- you have things that are described constants that are not numbers, like 'foo. Well, for my convenience, I want to syntactically transform that into a list structure which is, quote foo.

A quoted object, whatever it is, is going to be actually an abbreviation, which is not part of the evaluator but happens somewhere else, an abbreviation for an expression that looks like this. This way, I can test for the type of the expression as being a quotation by examining the car of the expression. So I'm not going to worry about that in the evaluator. It's happening somewhere earlier in the reader or something.

If the expression of the expression is quote, then what I want, I want quote foo to itself evaluate to foo. It's a constant. This is just a way of saying that this evaluates to itself. What is that? That's the second of the list. It's the second element of the list. The second element of the list is it's CADR. So I'm just going to write here, CADR.

What else do we have here? We have lambda expressions, for example, lambda of x plus x y. Well, I going have to have some representation for the procedure which is the value of an expression, of a lambda expression. The procedure here is not the expression lambda x. That's the description of it, the textual description.

However, what what I going to expect to see here is something which contains an environment as one of its parts if I'm implementing a lexical language. And so what I'd like to see is some type flags. I'm going to have to be able to distinguish procedures later, procedures which were produced by lambdas, from ones that may be primitive. And so I'm going to have some flag, which I'll just arbitrarily call closure, just for historical reasons.

Now, to say what parts of this are important. I'm going to need to know the bound variable list and the body. Well, that's the CDR of this, so it's going to be x and plus x y and some environment. Now this is not something that

users should ever see, this is purely a representation, internally, for a procedure object. It contains a bound variable list, a body, and an environment, and some type tag saying, I am a procedure.

I'm going to make one now. So if the CAR of the expression is quote lambda, then what I'm going to put here is-- I'm going to make a list of closure, the CDR of the procedure description was everything except the lambda, and the current environment.

This implements the rule for environments in the environment model. It has to do with construction of procedures from lambda expressions. The environment that was around at the time the evaluator encountered the lambda expression is the environment where the procedure resulting interprets it's free variables. So that's part of that. And so we have to capture that environment as part of the procedure object. And we'll see how that gets used later.

There are also conditional expressions of things like COND of say, p one, e one, p two, e two. Where this is a predicate, a predicate is a thing that is either true or false, and the expression to be evaluated if the predicate is true. A set of clauses, if you will, that's the name for such a thing. So I'm going put that somewhere else. We're going to worry about that in another piece of code.

So EQ-- if the CAR of the expression is COND, then I'm going to do nothing more than evaluate the COND, the CDR of the expression. That's all the clauses in the environment that I'm given.

Well, there's one more case, arbitrary thing like the sum of x and three, where this is an operator applied to operands, and there's nothing special about it. It's not one of the special cases, the special forms. These are the special forms.

And if I were writing here a professional program, again, I would somehow make this data directed. So there wouldn't be a sequence of conditionals here, there'd be a dispatch on some bits if I were trying to do this in a more professional way. So that, in fact, I can add to the thing without changing my program much. So, for example, they would run fast, but I'm not worried about that.

Here we're trying to look at this in its entirety. So it's else. Well, what do we do? In this case, I have to somehow do an addition. Well, I could find out what the plus is. I have to find out what the x and the three are. And then I have to apply the result of finding what the plus is to the result of finding out what the x and the three are. We'll have a name for that.

So I'm going to apply the result of evaluating the CAR of the expression-- the car of the expression is the operator-- in the environment given. So evaluating the operator gets me the procedure. Now I have to evaluate all

the operands to get the arguments. I'll call that EVLIST, the CDR of the operands, of the expression, with respect to the environment. EVLIST will come up later-- EVLIST, apply, COND pair, COND, lambda, define.

So that what you are seeing here now is pretty much all there is in the evaluator itself. It's the case dispatch on the type of the expression with the default being a general application or a combination.

Now there is lots of things we haven't defined yet. Let's just look at them and see what they are. We're going to have to do this later, evcond. We have to write apply. We're going to have to write EVLIST. We're going to write LOOKUP. I think that's everything, isn't there? Everything else is something which is simple, or primitive, or something like that.

And, of course, we could many more special forms here, but that would be a bad idea in general in a language. You make a language very complicated by putting a lot of things in there. The number of reserve words that should exist in a language should be no more than a person could remember on his fingers and toes. And I get very upset with languages which have hundreds of reserve words. But that's where the reserve words go.

Well, now let's get to the next part of this, the kernel, apply. What else is this doing? Well, apply's job is to take a procedure and apply it to its arguments after both have been evaluated to come up with a procedure and the arguments rather the operator symbols and the operand symbols, whatever they are-- symbolic expressions.

So we will define apply to be a procedure of two arguments, a procedure and arguments. And what does it do? It does nothing very complicated. It's got two cases. Either the procedure is primitive-- And I don't know exactly how that is done.

It's possible there's some type information just like we made closure for, here, being the description of the type of a compound thing-- probably so. But it is not essential how that works, and, in fact, it turns out, as you probably know or have deduced, that you don't need any primitives anyway. You can compute anything without them because some of the lambda that I've been playing with. But it's nice to have them.

So here we're going to do some magic which I'm not going to explain. Go to machine language, apply primop. Here's how it adds. Execute an add instruction. However, the interesting part of a language is the glue by which the predicates are glued together.

So let's look at that. Well, the other possibility is that this is a compound made up by executing a lambda expression, this is a compound procedure. Well, we'll check its type. If it is closure, if it's one of those, then I have to do an eval of the body. The way I do this, the way I deal with this at all, is the way I evaluate the application of a procedure to its arguments, is by evaluating the body of the procedure in the environment resulting from extending the environment of the procedure with the bindings of the formal parameters of the procedure to the

arguments that were passed to it. That was a long sentence.

Well that's easy enough. Now here's going to be a lot of CAR-CDRing. I have to get the body of the procedure. Where's the body of the procedure in here? Well here's the CAR, here's the CDR is the whole rest of this. So here's the CADR. And so I see, what I have here is the body is the second element of the second element of the procedure. So it's the CADR of the CADR or the CADADR.

It's the C-A-D-A-D-R, CADADR of the procedure. To evaluate the body in the result of binding that's making up more environment, well I need the formal parameters of the of the procedure, what is that? That's the CAR of the CDR. It's horrible isn't it? --of the procedure. Bind that to the arguments that were passed in the environment, which is passed also as part of the procedure. Well, that's the CAR of the CDR of the CDR of this, CADADR, of the procedure. Bind, eval, pair, COND, lamda, define--

Now, of course, if I were being really a neat character, and I was being very careful, I would actually put an extra case here for checking for certain errors like, did you try to apply one to an argument? You get a undefined procedure type. So I may as well do that anyway. --else, some sort of error, like that.

Now, of course, again, in some sort of more real system, written for professional reasons, this would be written with a case analysis done by some sort of dispatch. Over here, I would probably have other cases like, is this compiled code? It's very important.

I might have distinguished the kind of code that's produced by a directly evaluating a lambda in interpretation from code that was produced by somebody's compiler or something like that. And we'll talk about that later. Or is this a piece Fortran program I have to go off and execute. It's a perfectly possible thing, at this point, to do that.

In fact, in this concrete syntax evaluator I'm writing here, there's an assumption built in that this is Lisp, because I'm using CARs and CDRs. CAR means the operator, and CDR means the operand. In the text, there is an abstract syntax evaluator for which these could be-- these are given abstract names like operator, and operand, and all these other things are like that. And, in that case, you could reprogram it to be ALGOL with no problem.

Well, here we have added another couple of things that we haven't defined. I don't think I'll worry about these at all, however, this one will be interesting later. Let's just proceed through this and get it done. There's only two more blackboards so it can't be very long. It's carefully tailored to exactly fit.

Well, what do we have left? We have to define EVLIST, which is over here. And EVLIST is nothing more than a map down a bunch of operands producing arguments. But I'm going to write it out. And one of the reasons I'm going to write this out is for a mystical reason, which is I want to make this evaluator so simple that it can

understand itself. I'm going to really worry about that a little bit.

So let's write it out completely. See, I don't want to worry about whether or not the thing can pass functional arguments. The value evaluator is not going to use them. The evaluator is not going to produce functional values. So even if there were a different, alternative language that were very close to this, this evaluates a complex language like Scheme which does allow procedural arguments, procedural values, and procedural data.

But even if I were evaluating ALGOL, which doesn't allow procedural values, I could use this evaluator. And this evaluator is not making any assumptions about that. And, in fact, if this value were to be restricted to not being able to that, it wouldn't matter, because it doesn't use any of those clever things. So that's why I'm arranging this to be super simple. This is sort of the kernel of all possible language evaluators. How about that?

Evlist-- well, what is it? It's the procedure of two arguments, l and an environment, where l is a list such that if the list of arguments is the empty list, then the result is the empty list. Otherwise, I want to cons up the result of evaluating the CAR of the list of operands in the environment. So I want the first operand evaluated, and I'm going to make a list of the results by CONSing that onto the result of this EVLISTing as a CDR recursion, the CDR of the list relative to the same environment. Evlist, cons, else, COND, lambda, define--

And I have one more that I want to put on the blackboard. It's the essence of this whole thing. And there's some sort of next layer down. Conditionals-- conditionals are the only thing left that are sort of substantial. Then below that, we have to worry about things like lookup and bind, and we'll look at that in a second. But of the substantial stuff at this level of detail, next important thing is how you deal with conditionals.

Well, how do we have a conditional thing? It's a procedure of a set of clauses and an environment. And what does it do? It says, if I've no more clauses, well, I have to give this a value. It could be that it was an error. Supposing it run off the end of a conditional, it's pretty arbitrary. It's up to me as programmer to choose what I want to happen. It's convenient for me, right now, to write down that this has a value which is the empty list, doesn't matter. For error checking, some people might prefer something else.

But the interesting things are the following ones. If I've got an else clause-- You see, if I have a list of clauses, then each clause is a list. And so the predicate part is the CAAR of the clauses. It's the CAR, which is the first part of the first clause in the list of clauses. If it's an else, then it means I want my result of the conditional to be the result of evaluating the matching expression. So I eval the CADR. So this is the first clause, the second element of it, CADAR-- CADAR of a CAR-- of the clauses, with respect to the environment.

Now the next possibility is more interesting. If it's false, if the first predicate in the predicate list is not an else, and it's not false, if it's not the word else, and if it's not a false thing-- Let's write down what it is if it's a false thing. If the

result of evaluating the first predicate, the clauses-- respect the environment, if that evaluation yields false, then it means, I want to look at the next clause. So I want to discard the first one. So we just go around loop, evcond, the CDR of the clauses relative to that environment. And otherwise, I had a true clause, in which case, what I want is to evaluate the CADAR of the clauses relative to that environment.

Boy, it's almost done. It's quite close to done. I think we're going to finish this part off. So just buzzing through this evaluator, but so far you're seeing almost everything. Let's look at the next transparency here.

Here is bind. Bind is for making more table. And what we are going to do here is make a-- we're going to make a no-frame for an environment structure. The environment structure is going to be represented as a list of frames. So given an existing environment structure, I'm going to make a new environment structure by consing a new frame onto the existing environment structure, where the new frame consists of the result of pairing up the variables, which are the bound variables of the procedure I'm applying, to the values which are the arguments that were passed that procedure.

This is just making a list, adding a new element to our list of frames, which is an environment structure, to make a new environment. Where pair-up is very simple. Pair-up is nothing more than if I have a list of variables and a list of values, well, if I run out of variables and if I run out of values, everything's OK. Otherwise, I've given too many arguments.

If I've not run out of variables, but I've run out of values, that I have too few arguments. And in the general case, where I don't have any errors, and I'm not done, then I really am just adding a new pair of the first variable with the first argument, the first value, onto a list resulting from pairing-up the rest of the variables with the rest of the values.

Lookup is of course equally simple. If I have to look up a symbol in an environment, well, if the environment is empty, then I've got an unbound variable. Otherwise, what I'm going to do is use a special pair list lookup procedure, which we'll have very shortly, of the symbol in the first frame of the environment. Since I know the environment is not empty, it must have a first frame.

So I lookup the symbol in the first frame. That becomes the value cell here. And then, if the value cell is empty, if there is no such value cell, then I have to continue and look at the rest of the frames. It means there was nothing found there.

So that's a property of ASSQ is it returns emptiness if it doesn't find something. but if it did find something, then I'm going to use the CDR of the value cell here, which is the thing that was the pair consisting of the variable and the value. So the CDR of it is the value part.

Finally, ASSQ is something you've probably seen already. ASSQ takes a symbol and a list of pairs, and if the list is empty, it's empty. If the symbol is the first thing in the list-- That's an error. That should be CAAR, C-A-A-R. Everybody note that. Right there, OK?

And in any case, if the symbol is the CAAR of the A list, then I want the first, the first pair, in the A list. So, in other words, if this is the key matching the right entry, otherwise, I want to look up that symbol in the rest. Sorry for producing a bug, bugs appear.

Well, in any case, you're pretty much seeing the whole thing now. It's a very beautiful thing, even though it's written in an ugly style, being the kernel of every language. I suggest that we just-- let's look at it for a while.

[MUSIC PLAYING]

Are there any questions? Alright, I suppose it's time to take a small break then. [MUSIC PLAYING]

OK, now we're just going to do a little bit of practice understanding what it is we've just shown you. What we're going to do is go through, in detail, an evaluation by informally substituting through the interpreter. And since we have no assignments or definitions in this interpreter, we have no possible side effects, and so the we can do substitution with impunity and not worry about results.

So the particular problem I'd like to look at is it an interesting one. It's the evaluation of quote, open, open, open, lambda of x, lambda of y plus x y, lambda, lambda, applied to three, applied to four, in some global environment which I'll call e0.

So what we have here is a procedure of one argument x, which produces as its value a procedure of one argument y, which adds x to y. We are applying the procedure of one argument x to three. So x should become three. And the result of that should be procedure of one argument y, which will then apply to 4. And there is a very simple case, they will then add those results.

And now in order to do that, I want to make a very simple environment model. And at this point, you should already have in your mind the environments that this produces. But we're going to start out with a global environment, which I'll call e0, which is that. And it's going to have in it things, definitions for plus, and times, and-- using Greek letters, isn't that interesting, for the objects-- and minus, and quotient, and CAR, and CDR, and CONS, and EQ, and everything else you might imagine in a global environment. It's got something there for each of those things, something the machine is born with, that's e0.

Now what does it mean to do this evaluation? Well, we go through the set of special forms. First of all, this is not a number. This is not a symbol. Gee, it's not a quoted expression. This is a quoted expression, but that's not what I

interested in. The question is, whether or not the thing which is quoted is quoted expression? I'm evaluating an expression. This just says it's this particular expression. This is not a quoted expression. It's not a thing that begins with lambda. It's not a thing that begins with COND. Therefore, it's an application of its of an operated operands. It's a combination.

The combination thus has this as the operator and this is the operands. Well, that means that what I'm going to do is transform this into apply of eval, of quote, open, open lambda of x, lambda of y-- I'm evaluating the operator-- plus x y, in the environment, also e0, with the operands that I'm going to apply this to, the arguments being the result of EVLIST, the list containing four, fin e0.

I'm using this funny notation here for e0 because this should be that environment. I haven't a name for it, because I have no environment to name it in. So this is just a representation of what would be a quoted expression, if you will. The data structure, which is the environment, goes there.

Well, that's what we're seeing here. Well in order to do this, I have to do this, and I have to do that. Well this one's easy, so why don't we do that one first. This turns into apply of eval-- just copying something now. Most of the substitution rule is copying. So I'm going to not say the words when I copy, because it's faster. And then the EVLIST is going to turn into a cons, of eval, of four, in e0-- because it was not an empty list-- onto the result of EVLISTing, on the empty list, in e0.

And I'm going to start leaving out steps soon, because it's going to get boring. But this is basically the same thing as apply, of eval-- I'm going to keep doing this-- the lambda of x, the lambda of y, plus xy, 3, close, e0. I'm a pretty good machine.

Well, eval of four, that's meets the question, is it a number. So that's cons, cons of 4. And EVLIST of the empty list is the empty list, so that's this. And that's very simple to understand, because that means the list containing four itself. So this is nothing more than apply of eval, quote, open, open, lambda of x, lambda of y, plus x y, three applied to, e0, applied to the list four-- bang. So that's that step.

Now let's look at the next, more interesting thing. What do I do to evaluate that? Evaluating this means I have to evaluate-- Well, it's not. It's nothing but an application. It's not one of the special things. If the application of this operator, which we see here-- here's the operator-- applied to this operands, that combination. But we know how to do that, because that's the last case of the conditional. So substituting in for this evaluation, it's apply of eval of the operator in the EVLIST of the operands.

Well, it's apply, of apply, of eval, of quote, open, lambda of x, lambda of y, plus x y, lambda, lambda, in environment e0. I'm going to short circuit the evaluation of the operands , because they're the same as they were

before. I got a list containing three, apply that, and apply that to four.

Well let's see. Eval of a lambda expression produces a procedure object. So this is apply, of apply, of the procedure object closure, which contains the body of the procedure, x, which is lambda-- which binds x [UNINTELLIGIBLE] the internals of the body, it returns the procedure of one argument y, which adds x to y. Environment e0 is now captured in it, because this was evaluated with respect to e0. e0 is part now of the closure object. Apply that to open, three, close, apply, to open, 4, close, apply.

So going from this step to this step meant that I made up a procedure object which captured in it e0 as part of the procedure object. Now, we're going to pass those to apply. We have to apply this procedure to that set of arguments. Well, but that procedure is not primitive. It's, in fact, a thing which has got the tag closure, and, therefore, what we have to do is do a bind.

We have to bind. A new environment is made at this point, which has as its parent environment the one over here, e0, that environment. And we'll call this one, e1. Now what's bound in there? x is bound to three. So I have x equal three. That's what's in there. And we'll call that e1. So what this transforms into is an eval of the body of this, which is this, the body of that procedure, in the environment that you just saw.

So that's an apply, of eval, quote, open, lambda of y, plus x y-- the body-- in e1. And apply the result of that to four, open, close, 4-- list of arguments. Well, that's sensible enough because evaluating a lambda, I know what to do.

That means I apply, the procedure which is closure, binds one argument y, adds x to y, with e1 captured in it. And you should really see this. I somehow manufactured a closure. I should've put this here. There was one over here too. Well, there's one here now. I've captured e1, and this is the procedure of one argument y, whatever this is. That's what that is there, that closure.

I'm going to apply that to four. Well, that's easy enough. That means I have to make a new environment by copying this pointer, which was the pointer of the procedure, which binds y equal 4 with that environment. And here's my new environment, which I'll call e2. And, of course, this application then is evaluate the body in e2.

So this is eval, the body, which is plus x y, in the environment e2. But this is an application, so this is the apply, of eval, plus in e2, an EVLIST, quote, open, x y, in e2. Well, but let's see. That is apply, the object which is a result of that and plus. So here we are in e2, plus is not here, it's not here, oh, yes, but's here as some primitive operator. So it's the primitive operator for addition. Apply that to the result of evaluating x and y in e2. But we can see that x is three and y is four. So that's a three and four, here. And that magically produces for me a seven.

I wanted to go through this so you would see, essentially, one important ingredient, which is what's being passed

around, and who owns what, and what his job is. So what do we have here? We have eval, and we have apply, the two main players. And there is a big loop the goes around like this. Which is eval produces a procedure and arguments for apply.

Now some things eval could do by itself. Those are little self things here. They're not interesting. Also eval evaluates all of the arguments, one after another. That's not very interesting. Apply can apply some procedures like plus, not very interesting. However, if apply can't apply a procedure like plus, it produces an expression and environment for eval. The procedural arguments wrap up essentially the state of a computation and, certainly, the expression of environment.

And so what we're actually going to do next is not the complete state, because it doesn't say who wants the answers. But what we're going to do-- it's always got something like an expression of environment or procedure and arguments as the main loop that we're going around.

There are minor little sub loops like eval through EVLIST, or eval through evcond, or apply through a primitive apply. But they're not the essential things. So that's what I wanted you to see. Are there any questions? Yes.

AUDIENCE: I'm trying to understand how x got down to three instead of four. At the early part of the--

PROFESSOR: Here. You want to know how x got down to three?

AUDIENCE: Because x is the outer procedure, and x and y are the inner procedure.

PROFESSOR: Fine. Well, I was very careful and mechanical. First of all, I should write those procedures again for you, pretty printed. First order of business, because you're probably not reading them well.

So I have here that procedure of-- was it x over there-- which is-- value of that procedure of y, which adds x to y, lambda, lambda, applied that to three, takes the result of that, and applied that to four. Is that not what I wrote?

Now, you should immediately see that here is an application-- let me get a white piece of chalk-- here is an application, a combination. That combination has this as the operator and this as the operand. The three is going in for the x here. The result of this is a procedure of one argument y, which gets applied to four. So you just weren't reading the expression right.

The way you see that over here is that here I have the actual procedure object, x. It's getting applied to three, the list containing three. What I'm left over with is something which gets applied to four. Are there any other questions? Time for our next small break then. Thank you. [MUSIC PLAYING]

Let's see, at this point, you should be getting the feeling, what's this nonsense this Sussman character is feeding me? There's an awful lot of strange nonsense here. After all, he purported to explain to me Lisp, and he wrote me a Lisp program on the blackboard.

The Lisp program was intended to be interpreted for Lisp, but you need a Lisp interpreter in order to understand that program. How could that program have told me anything there is to be known about Lisp? How is that not completely vacuous? It's a very strange thing. Does it tell me anything at all?

Well, you see, the whole thing is sort of like these Escher's hands that we see on this slide. Yes, eval and apply each sort of draw each other and construct the real thing, which can sit out and draw itself. Escher was a very brilliant man, he just didn't know the names of these spirits.

Well, I'm going to do now, is I'm going to try to convince you that both this mean something, and, as a aside, I'm going to show you why you don't need definitions. Just turns out that that sort of falls out, why definitions are not essential in a mathematical sense for doing all the things we need to do for computing.

Well, let's see here. Consider the following small program, what does it mean? This is a program for computing exponentials.

The exponential of x to the nth power is if-- and is zero, then the result is one. Otherwise, I want the product of x and the result of exponentiating x to the n minus one power. I think I got it right.

Now this is a recursive definition. It's a definition of the exponentiation procedure in terms of itself. And, as it has been mentioned before, your high school geometry teacher probably gave you a hard time about things like that. Was that justified? Why does this self referential definition make any sense?

Well, first of all, I'm going to convince you that your high school geometry teacher was I telling you nonsense. Consider the following set of definitions here. x plus y equals three, and x minus y equal one. Well, gee, this tells you x in terms of y, and this one tells you y in terms of x, presumably. And yet this happens to have a unique solution in x and y.

However, I could also write two x plus two y is six. These two equations have an infinite number solutions. And I could write you, for example, x minus y equal 2, and these two equations have no solutions.

Well, I have here three sets of simultaneous linear equations, this set, this set, and this set. But they have different numbers of solutions. The number of solutions is not in the form of the equations. They all three sets have the same form. The number of solutions is in the content.

I can't tell by looking at the form of a definition whether it makes sense, only by its detailed content. What are the coefficients, for example, in the case of linear equations? So I shouldn't expect to be able to tell looking at something like this, from some simple things like, oh yes, EXPT is the solution of this recursion equation. Expt is the procedure which if substituted in here, gives me EXPT back.

I can't tell, looking at this form, whether or not there's a single, unique solution for EXPT, an infinite number of solutions, or no solutions. It's got to be how it counts and things like that, the details. And it's harder in programming than linear algebra. There aren't too many theorems about it in programming.

Well, I want to rewrite these equations a little bit, these over here. Because what we're investigating is equations like this. But I want to play a little with equations like this that we understand, just so we get some insight into this kind of question. We could rewrite our equations here, say these two, the ones that are interesting, as x equals three minus y, and y equals x minus one. What do we call this transformation? This is a linear transformation, t.

Then what we're getting here is an equation x y equals t of x y. What am I looking for? I'm looking for a fixed point of t. The solution is a fixed point of t.

So the methods we should have for looking for solutions to equations, if I can do it by fixed points, might be applicable. If I have a means of finding a solution to an equations by fixed points-- just, might not work-- but it might be applicable to investigating solutions of equations like this.

But what I want you to feel is that this is an equation. It's an expression with several instances of various names which puts a constraint on the name, saying what that name could have as its value, rather than some sort of mechanical process of substitution right now. This is an equation which I'm going to try to solve.

Well, let's play around and solve it. First of all, I want to write down the function which corresponds to t. First I want to write down the function which corresponds to t whose fixed point is the answer to this question. Well, let's consider the following procedure f. I claim it computes that function. f is that procedure of one argument g, which is that procedure of two arguments x and n. Which have the property that if n is zero, then the result is one, otherwise, the result is the product of x and g, applied to x, and minus n1. g, times, else, COND, lambda, lambda--

Here f is a procedure, which if I had a solution to that equation, if I had a good exponentiation procedure, and I applied f to that procedure, then the result would be a good exponentiation procedure. Because, what does it do? Well, all it is is exposing g were a good exponentiation procedure, well then this would produce, as its value, a procedure to arguments x and n, such that if n were 0, the result would be one, which is certainly true of exponentiation. Otherwise, it will be the result of multiplying x by the exponentiation procedure given to me with x and n minus one as arguments. So if this computed the correct exponentiation for n minus one, then this would be

the correct exponentiation for exponent n, so this would have been the right exponentiation procedure.

So what I really want to say here is E-X-P-T is a fixed point of f. Now our problem is there might be more than one fixed point. There might be no fixed points. I have to go hunting for the fixed points. Got to solve this equation.

Well there are various ways to hunt for fixed points. Of course, the one we played with at the beginning of this term worked for cosine. Go into radians mode on your calculator and push cosine, and just keep doing it, and you get to some number which is about 0.73 or 0.74. I can't remember which. By iterating a function, whose fixed point I'm searching for, it is sometimes the case that that function will converge in producing the fixed point.

I think we luck out in this case, so let's look for it. Let's look at this slide. Consider the following sequence of procedures. e0 over here is the procedure which does nothing at all. It's the procedure which produces an error for any arguments you give it. It's basically useless. Well, however, I can make an approximation. Let's consider it the worst possible approximation to exponentiation, because it does nothing.

Well, supposing I substituted e0 for g by calling f, as you see over here on e0. So you see over here, have e0 there. Then gee, what's e1? e1 is a procedure which exponentiate things to the 0th power, with no trouble. It gets the right answer, anything to the zero is one, and it makes an error on anything else.

Well, now what if I take e1 and I substitute if for g by calling f on e1? Oh gosh, I have here a procedure of two arguments. Now remember e1 was appropriate for taking exponentiations of 0, for raising to the 0 exponent. So here, is n is 0, the result is one, so this guy is good for that too. However, I can use something for raising to the 0th power to multiply it by x to raise something to the first power. So e2 is good for both power 0 and one.

And e3 is constructed from e2 in the same way. And e3, of course, by the same argument is good for powers 0, one, and two. And so I will assert for you, without proof, because the proof is horribly difficult. And that's the sort of thing that people called denotational semanticists do. This great idea was invented by Scott and Strachey. They're very famous mathematician types who invented the interpretation for these programs that we have that I'm talking to you about right now. And they proved, by topology that there is such a fixed point in the cases that we want.

But the assertion is E-X-P-T is limit as n goes to infinity of em. and And that we've constructed this by the following way. --is Well, it's f of, f of, f of, f of, f of-- f applied to anything at all. It didn't matter what that was, because, in fact, this always produces an error. Applied to this-- That's by infinite nesting of f's.

So now my problem is to make some infinite things. We need some infinite things. How am I going to nest up an f an infinite number of times? I'd better construct this. Well, I don't know. How would I make an infinite loop at all?

Let's take a very simple infinite loop, the simplest infinite loop imaginable. If I were to take that procedure of one

argument x which applies x to x and apply that to the procedure of one argument x which applies x to x, then this is an infinite loop.

The reason why this is an infinite loop is as follows. The way I understand this is I substitute the argument for the formal parameter in the body. But if I do that, I take for each of these x's, I substitute one of these, making a copy of the original expression I just started with, the simplest infinite loop.

Now I want to tell you about a particular operator which is constructed by a perturbation from this infinite loop. I'll call it y. This is called Curry's Paradoxical Combinator of y after a fellow by the name of Curry, who was a logician of the 1930s also. And if I have a procedure of one argument f, what's it going to have in it? It's going to have a kind of infinite loop in it, which is that procedure of one argument x which applies f to x of x, applied to that procedure of one argument x, which applies f to f of x.

Now what's this do? Suppose we apply y to F. Well, that's easy enough. That's this capital F over here. Well, the easiest thing to say there is, I substitute F for here. So that's going to give me, basically-- because then I'm going to substitute this for x in here.

Let me actually do it in steps, so you can see it completely. I'm going to be very careful. This is open, open, lambda of x , capital F, x, x, applied to itself, F of x of x. Substituting this for this in here, this is F applied to-- what is it-- substituting this in here, open, open, lambda of x, F, of x and x, applied to lambda of x, F of x of x, F, lambda, pair, F.

Oh, but what is this? This thing over here that I just computed, is this thing over here. But I just wrapped another F around it. So by applying y to F, I make an infinite series of F's. If I just let this run forever, I'll just keep making more and more F's outside. I ran an infinite loop which is useless, but it doesn't matter that the inside is useless.

So y of F is F applied to y of F. So y is a magical thing which, when applied to some function, produces the object which is the fixed point of that function, if it exists, and if this all works. Because, indeed, if I take y of F and put it into F, I get y of F out.

Now I want you to think this in terms of the eval-apply interpreter for a bit. I wrote down a whole bunch of recursion equations out there. They're simultaneous in the same way these are simultaneous equations. Exponentiation was not a simultaneous equation. It was only one variable I was looking for a meaning for.

But what Lisp is is the fixed point of the process which says, if I knew what Lisp was and substituted it in for eval, and apply, and so on, on the right hand sides of all those recursion equations, then if it was a real good Lisp, is a real one, then the left hand side would also be Lisp. So I made sense of that definition. Now whether or not there's an answer isn't so obvious. I can't attack that.

Now these arguments that I'm giving you now are quite dangerous. Let's look over here. These are limit arguments. We're talking about limits, and it's really calculus, or topology, or something like that, a kind of analysis. Now here's an argument that you all believe. And I want to make sure you realize that I could be bullshitting you.

What is this? u is the sum of 1/2, 1/4, and 1/8, and so on, the sum of a geometric series. And, of course, I could play a game here. u minus one is 1/2, plus 1/4, plus 1/8, and so on. What I could do here-- oops. There is a parentheses error here. But I can put here two times u minus one is one plus 1/2, plus 1/4, plus 1/8. Can I fix that? Yes, well. But that gives me back two times u minus one is u, therefore, we conclude that u is two. And this actually is true. There's no problem like that. But supposing I did something different.

Supposing I start up with something which manifestly has no sum. v is one, plus two, plus four, plus 8, plus dot, dot, dot. Well, v minus one is surely two, plus four, plus eight, plus dot, dot, dot. v minus one over two, gee, that looks like v again. From that I should be able to conclude that-- that's also wrong, apparently. v equals minus one. That should be a minus one. And that's certainly a false conclusion.

So when you play with limits, arguments that may work in one case they may not work in some other case. You have to be very careful. The arguments have to be well formed. And I don't know, in general, what the story is about arguments like this. We can read a pile of topology and find out.

But, surely, at least you understand now, why it might be some meaning to the things we've been writing on the blackboard. And you understand what that might mean. So, I suppose, it's almost about time for you to merit being made a member of the grand recursive order of lambda calculus hackers. This is the badge. Because you now understand, for example, what it says at the very top, y F equals F y F. Thank you. Are there any questions? Yes, Lev.

AUDIENCE: With this, it seems that then there's no need to define, as you imply, to just remember a value, to apply it later. Defines were kind of a side-effect it seemed in the language. [INTERPOSING] are order dependent. Does this eliminate the side-effect from the [INTERPOSING]

PROFESSOR: The answer is, this is not the way these things were implemented. Define, indeed is implemented as an operation that actually modifies an environment structure, changes the frame that the define is executed in. And there are many reasons for that, but a lot of this has to do with making an interactive system. What this is saying is that if you've made a system, and you know you're not going to do any debugging or anything like that, and you know everything there is all at once, and you want to say, what is the meaning of a final set of equations? This gives you a meaning for it. But in order to make an interactive system, where you can change the meaning of

one thing without changing everything else, incrementally, you can't do that by implementing it this way. Yes.

AUDIENCE: Another question on your danger slide. It seemed that the two examples that you gave had to do with convergence and non-convergence? And that may or may not have something to do with function theory in a way which would lead you to think of it in terms of linear systems, or non-linear systems. How does this convergence relate to being able to see a priori what properties of that might be violated?

PROFESSOR: I don't know. The answer is, I don't know under what circumstances. I don't know how to translate that into less than an hour of talk more. What are the conditions under which, for which we know that these things converge? And v, all that was telling you that arguments that are based on convergence are flaky if you don't know the convergence beforehand. You can make wrong arguments. You can make deductions, as if you know the answer, and not be stopped somewhere by some obvious contradiction.

AUDIENCE: So can we say then that if F is a convergent mathematical expression, then the recursion property can be--

PROFESSOR: Well, I think there's a technical kind of F, there is a technical description of those F's that have the property that when you iteratively apply them like this, you converge. Things that are monotonic, and continuous, and I forgot what else. There is a whole bunch of little conditions like that which have this property. Now the real problem is deducing from looking at the F, its definition here, whether not it has those properties, and that's very hard. The properties are easy. You can write them down.

You can look in a book by Joe Stoy. It's a great book-- Stoy. It's called, *The Scott-Strachey Method of Denotational Semantics*, and it's by Joe Stoy, MIT Press. And he works out all this in great detail, enough to horrify you. But it really is readable.

OK, well, thank you. Time for the bigger break, I suppose.