[MUSIC PLAYING]

PROFESSOR: Well, let's see. What we did so far was a lot of fun, was it useful for anything? I suppose the answer is going to be yes. If these metacircular interpreters are a valuable thing to play with. Well, there have been times I spend 50% of my time, over a year, trying various design alternatives by experimenting with them with metacircular interpreters-- metacircular interpreters like the sort you just saw. Metacircular is because they are defined in terms of themselves in such a way that the language they interpret contains itself.

Such interpreters are a convenient medium for exploring language issues. If you want to try adding a new feature, it's sort of a snap, it's easy, you just do it and see what happens. You play with that language for a while you say, gee, I'm didn't like that, you throw it away. Or you might want to see what the difference is if you'd make a slight difference in the binding strategy, or some more complicated things that might occur.

In fact, these metacircular interpreters are an excellent medium for people exchanging ideas about language design, because they're pretty easy to understand, and they're short, and compact, and simple. If I have some idea that I want somebody to criticize like say, Dan Friedman at Indiana, I'd write a little metacircular interpreter and send him some network mail with this interpreter in it. He could whip it up on his machine and play with it and say, that's no good. And then send it back to me and say, well, why don't you try this one, it's a little better.

So I want to show you some of that technology. See, because, really, it's the essential, simple technology for getting started in designing your own languages for particular purposes. Let's start by adding a very simple feature to a Lisp. Now, one thing I want to tell you about is features, before I start.

There are many languages that have made a mess of themselves by adding huge numbers of features. Computer scientists have a joke about bugs that transform it to features all the time. But I like to think of it is that many systems suffer from what's called creeping featurism.

Which is that George has a pet feature he'd like in the system, so he adds it. And then Harry says, gee, this system is no longer what exactly I like, so I'm going to add my favorite feature. And then Jim adds his favorite feature. And, after a while, the thing has a manual 500 pages long that no one can understand.

And sometimes it's the same person who writes all of these features and produces this terribly complicated thing. In some cases, like editors, it's sort of reasonable to have lots of features, because there are a lot of things you want to be able to do and many of them arbitrary. But in computer languages, I think it's a disaster to have too much stuff in them.

The other alternative you get into is something called feeping creaturism, which is where you have a box which has a display, a fancy display, and a mouse, and there is all sorts of complexity associated with all this fancy IO. And your computer language becomes a dismal, little, tiny thing that barely works because of all the swapping, and disk twitching, and so on, caused by your Windows system.

And every time you go near the computer, the mouse process wakes up and says, gee do you have something for me to do, and then it goes back to sleep. And if you accidentally push mouse with you elbow, a big puff of smoke comes out of your computer and things like that. So two ways to disastrously destroy a system by adding features.

But try right now to add a little, simple feature. This actually is a good one, and in fact, real Lisps have it. As you've seen, there are procedures like plus and times that take any number of arguments. So we can write things like the sum of the product of a and x and x, and the product of b and x and c.

As you can see here, addition takes three arguments or two arguments, multiplication takes two arguments or three arguments, taking numbers of arguments all of which are to be treated in the same way. This is a valuable thing, indefinite numbers of arguments.

Yet the particular Lisp system that I showed you is one where the numbers of arguments is fixed, because I had to match the arguments against the formal parameters in the binder, where there's a pairup. Well, I'd like to be able to define new procedures like this that can have any number of arguments. Well there's several parts to this problem. The first part is coming up with the syntactic specification, some way of notating the additional arguments, of which you don't know how many there are. And then there's the other thing, which is once we've notated it, how are we going to interpret that notation so as to do the right thing, whatever the right thing is?

So let's consider an example of a sort of thing we might want to be able to do. So an example might be, that I might want to be able to define a procedure which is a procedure of one required argument x and a bunch of arguments, I don't know how many there are, called y. So x is required, and there are many y's, many argument-- y will be the list of them.

Now, with such a thing, we might be able to say something like, map-- I'm going to do something to every one-- of that procedure of one argument u, which multiplies x by u, and we'll apply that to y. I've used a dot here to indicate that the thing after this is a list of all the rest of the arguments. I'm making a syntactic specification.

Now, what this depends upon, the reason why this is sort of a reasonable thing to do, is because this happens to be a syntax that's used in the Lisp reader for representing conses. We've never introduced that before. You may have seen when playing with the system that if you cons two things together, you get the first, space, dot, the second, space-- the first, space, dot, space, the second with parentheses around the whole thing.

So that, for example, this x dot y corresponds to a pair, which has got an x in it and a y in it. The other notations that you've seen so far are things like a procedure of arguments x and y and z which do things, and that looks like-- Just looking at the bound variable list, it looks like this, x, y, z, and the empty thing.

If I have a list of arguments I wish to match this against, supposing, I have a list of arguments one, two, three, I want to match these against. So I might have here a list of three things, one, two, three. And I want to match x, y, z against one, two, three.

Well, it's clear that the one matches the x, because I can just sort of follow the structure, and the two matches the y, and the three matches the z. But now, supposing I were to compare this x dot y-- this is x dot y-- supposing I compare that with a list of three arguments, one, two, three. Let's look at that again. One, two, three--

Well, I can walk along here and say, oh yes, x matches the one, the y matches the list, which is two and three. So the notation I'm choosing here is one that's very natural for Lisp system. But I'm going to choose this as a notation for representing a bunch of arguments.

Now, there's an alternative possibility. If I don't want to take one special out, or two special ones out or something like that, if I don't want to do that, if I want to talk about just the list of all the arguments like in addition, well then the argument list I'm going to choose to be that procedure of all the arguments x which does something with x. And which, for example, if I take the procedure, which takes all the arguments x and returned the list of them, that's list. That's the procedure list.

How does this work? Well, indeed what I had as the bound variable list in this case, whatever it is, is being matched against a list of arguments. This symbol now is all of the arguments. And so this is the choice I'm making for a particular syntactic specification, for the description of procedures which take indefinite numbers of arguments.

There are two cases of it, this one and this one. When you make syntactic specifications, it's important that it's unambiguous, that neither of these can be confused with a representation we already have, this one. I can always tell whether I have a fixed number of explicitly named arguments made by these formal parameters, or a fixed number of named formal parameters followed by a thing which picks up all the rest of them, or a list of all the arguments which will be matched against this particular formal parameter called x, because these are syntactically distinguishable.

Many languages make terrible errors in that form where whole segments of interpretation are cut off, because there are syntactic ambiguities in the language. They are the traditional problems with ALGOL like languages

having to do with the nesting of ifs in the predicate part.

In any case, now, so I've told you about the syntax, now, what are we going to do about the semantics of this? How do we interpret it? Well this is just super easy. I'm going to modify the metacircular interpreter to do it. And that's a one liner. There it is. I'm changing the way you pair things up. Here's the procedure that pairs the variables, the formal parameters, with the arguments that were passed from the last description of the metacircular interpreter.

And here's some things that are the same as they were before. In other words, if the list of variables is empty, then if the list of values is empty, then I have an empty list. Otherwise, I have too many arguments, that is, if I have empty variables but not empty values. If I have empty values, but the variables are not empty, I have too few arguments.

The variables are a symbol-- interesting case-- then, what I should do is say, oh yes, this is the special case that I have a symbolic tail. I have here a thing just like we looked over here. This is a tail which is a symbol, y. It's not a nil. It's not the empty list. Here's a symbolic tail that is just the very beginning of the tail. There is nothing else.

In that case, I wish to match that variable with all the values and add that to the pairing that I'm making. Otherwise, I go through the normal arrangement of making up the whole pairing. I suppose that's very simple. And that's all there is to it. And now I'll answer some questions. The first one-- Are there any questions? Yes?

AUDIENCE: Could you explain that third form?

PROFESSOR: This one? Well, maybe we should look at the thing as a piece of list structure. This is a procedure which contains a lambda. I'm just looking at the list structure which represents this. Here's x. These are our symbols. And then the body is nothing but x. If I were looking for the bound variable list part of this procedure, I would go looking at the CADR, and I'd find a symbol. So the, naturally, which is this pairup thing I just showed you, is going to be matching a symbolic object against a list of arguments that were passed. And it will bind that symbol to the list of arguments.

In this case, if I'm looking for it, the match will be against this in the bound variable list position. Now, if what this does is it gets a list of arguments and returns it, that's list. That's what the procedure is. Oh well, thank you. Let's take a break.

[MUSIC PLAYING]

PROFESSOR: Well let's see. Now, I'm going to tell you about a rather more substantial variation, one that's a famous variation that many early Lisps had. It's called dynamic binding of variables. And we'll investigate a little bit

about that right now. I'm going to first introduce this by showing you the sort of thing that would make someone want this idea.

I'm not going to tell what it is yet, I'm going to show you why you might want it. Suppose, for example, we looked at the sum procedure again for summing up a bunch of things. To be that procedure, of a term, lower bound, method of computing the next index, and upper bound, such that, if a is greater than b then the result is 0, otherwise, it's the sum, of the term, procedure, applied to a and the result of adding up, terms, with the next a being the a, the next procedure passed along, and the upper bound being passed along. Blink, blink, blink--

Now, when I use this sum procedure, I can use it, for example, like this. We can define the sum of the powers to be, for example, sum of a bunch of powers x to the n, to be that procedure of a, b, and n-- lower bound, the upper bound, and n-- which is sum, of lambda of x, the procedure of one argument x, which exponentiates x to the n, with the a, the incrementer, and b, being passed along.

So we're adding up x to n, given an x. x takes on values from a to b, incrementing by one. I can also write the-- That's right. Product, excuse me. The product of a bunch of powers. It's a strange name. I'm going to leave it there. Weird-- I write up what I have. I'm sure that's right. And if I want the product of a bunch of powers-- That was 12 brain cells, that double-take.

I can for example use the procedure which is like sum, which is for making products, but it's similar to that, that you've seen before. There's a procedure of three arguments again. Which is the product of terms that are constructed, or factors in this case, constructed from exponentiating x to the n, where I start with a, I increment, and I go to b.

Now, there's some sort of thing here that should disturb you immediately. These look the same. Why am I writing this code so many times? Here I am, in the same boat I've been in before. Wouldn't it be nice to make an abstraction here? What's an example of a good abstraction to make? Well, I see some codes that's identical. Here's one, and here's another.

And so maybe I should be able to pull that out. I should be able to say, oh yes, the sum of the powers could be written in terms of something called the nth power procedure. Imagine somebody wanted to write a slightly different procedure that looks like this. The sum powers to be a procedure of a, b, and n, as the result of summing up the nth power. We're going to give a name to that idea, for starting at a, going by one, and ending at b.

And similarly, I might want to write the product powers this way, abstracting out this idea. I might want this. Product powers, to be a procedure of a, b, and n, which is the product of the nth power operation on a with the incrementation and b being my arguments for the analogous-thing product.

And I'd like to be able to define, I'd like to be able to define nth power-- I'll put it over here. I'll put it at the top. --to be, in fact, my procedure of one argument x which is the result of exponentiating x to the n. But I have a problem. My environment model, that is my means of interpretation for the language that we've defined so far, does not give me a meaning for this n. Because, as you know, this n is free in this procedure.

The environment model tells us that the meaning of a free variable is determined in the environment in which this procedure is defined. In a way I have written it, assuming these things are defined on the blackboard as is, this is defined in the global environment, where there is no end. Therefore, n is unbound variable.

But it's perfectly clear, to most of us, that we would like it to be this n and this n. On the other hand, it would be nice. Certainly we've got to be careful here of keeping this to be this, and this one over here, wherever it is to be this one.

Well, the desire to make this work has led to a very famous bug. I'll tell you about the famous bug. Look at this slide. This is an idea called dynamic binding. Where, instead of the free variable being interpreted in the environment of definition of a procedure, the free variable is interpreted as having its value in the environment of the caller of the procedure.

So what you have is a system where you search up the chain of callers of a particular procedure, and, of course, in this case, since nth power is called from inside product whatever it is-- I had to write our own sum which is the analogous procedure-- and product is presumably called from product powers, as you see over here, then since product powers bind with variable n , then nth powers n would be derived through that chain.

Similarly, this n, the nth power in n in this case, would come through nth power here being called from inside sum. You can see it being called from inside sum here. It's called term here. But sum was called from inside of sum powers, which bound n. Therefore, there would be an n available for that n to get it's value from.

What we have below this white line plus over here, is what's called a dynamic binding view of the world. If that works, that's a dynamic binding view. Now, let's take a look, for example, at just what it takes to implement that. That's real easy.

In fact, the very first Lisps that had any interpretations of the free variables at all, had dynamic binding interpretations for the free variables. APL has dynamic binding interpretation for the free variables, not lexical or static binding. So, of course, the change is in eval. And it's really in two places.

First of all, one thing we see, is that things become a little simpler. If I don't have to have the environment be the environment of definition for procedure, the procedure need not capture the environment at the time it's defined.

And so if we look here at this slide, we see that the clause for a lambda expression, which is the way a procedure is defined, does not make up a thing which has a type closure and a attached environment structure. It's just the expression itself. And we'll decompose that some other way somewhere else.

The other thing we see is the applicator must be able to get the environment of the caller. The caller of a procedure is right here. If the expression we're evaluating is anpplication or a combination, then we're going to call a procedure which is the value of the operator. The environment of the caller is the environment we have right here, available now.

So all I have to do is pass that environment to the applicator, to apply. And if we look at that here, the only change we have to make is that fellow takes that environment and uses that environment for the purpose of extending that environment when abiding the formal parameters of the procedure to the arguments that were passed, not an environment that was captured in the procedure.

The reason why the first Lisps were implemented this way, is the sort of the obvious, accidental implementation. And, of course, as usual, people got used to it and liked it. And there were some people said, this is the way to do it. Unfortunately that causes some serious problems. The most important, serious problem in using dynamic binding is there's a modularity crisis that's involved it.

If two people are working together on some big system, then an important thing to want is that the names used by each one don't interfere with the names of the other. It's important that when I invent some segment of code that no one can make my code stop working by using my names that I use internal to my code, internal to his code.

However, dynamic binding violates that particular modularity constraint in a clear way. Consider, for example, what happens over here. Suppose it was the case that I decided to change the word next. Supposing somebody is writing sum, and somebody else is going to use sum.

The writer of sum has a choice of what names he may use. Let's say, I'm that writer. Well, by gosh, just happens I didn't want to call this next. I called it n. So all places where you see next, I called it n. Whoops. I changed nothing about the specifications of this program, but this program stops working. Not only that, unfortunately, this one does too.

Why do these programs stop working? Well, it's sort of clear. Instead of chasing out the value of the n that occurs in nth power over here or over here, through the environment of definition, where this one is always linked to this one, if it was through the environment of definition, because here is the definition. This lambda expression was executed in the environment where that n was defined. If instead of doing that, I have to chase through the call chain, then look what horrible thing happens.

Well, this was called from inside sum as term, term a. I'm looking for a value of n. Instead of getting this one, I get that one. So by changing the insides of this program, this program stops working. So I no longer have a quantifier, as I described before. The lambda symbol is supposed to be a quantifier. A thing which has the property that the names that are bound by it are unimportant, that I can uniformly substitute any names for these throughout this thing, so long as they don't occur in here, the new names, and the meaning of this expression should remain unchanged.

I've just changed the meaning of the expression by changing the one of the names. So lambda is no longer a well defined idea. It's a very serious problem. So for that reason, I and my buddies have given up this particular kind of abstraction, which I would like to have, in favor of a modularity principle.

But this is the kind of experiment you can do if you want to play with these interpreters. You can try them out this way, that way, and the other way. You see what makes a nicer language. So that's a very important thing to be able to do. Now, I would like to give you a feeling for I think the right thing to do is here.

How are you going to I get this kind of power in a lexical system? And the answer is, of course, what I really want is a something that makes up for me an exponentiator for a particular n. Given an n, it will make me an exponentiator. Oh, but that's easy too. In other words, I can write my program this way. I'm going to define a thing called PGEN, which is a procedure of n which produces for me an exponentiator. --x to the n.

Given that I have that, then I can capture the abstraction I wanted even better, because now it's encapsulated in a way where I can't be destroyed by a change of names. I can define some powers to be a procedure again of a, b, and n which is the sum of the term function generated by using this generator, PGEN, n, with a, incrementer, and b. And I can define the product of powers to be a procedure of a, b, and n which is the product PGEN, n, with a, increment, and b.

Now, of course, this is a very simple example where this object that I'm trying to abstract over is small. But it could be a 100 lines of code. And so, the purpose of this is, of course, to make it simple. I'd give a name to it, it's just that here it's a parameterized name. It's a name that depends upon, explicitly, the lexically apparent value of n. So you can think of this as a long name. And here, I've solved my problem by naming the term generation procedures within an n in them.

Are there any questions? Oh, yes, David.

AUDIENCE: Is the only solution to the problem you raise to create another procedure? In other words, can this only work in languages that are capable of defining objects as procedures?

PROFESSOR: Oh, I see. My solution to making this abstraction, when I didn't want include the procedure inside the body, depends upon my ability to return a procedure or export one. And that's right. If I don't have that, then I just don't have this ability to make an abstraction in a way where I don't have possibilities of symbol conflicts that were unanticipated. That's right. I consider being able to return the procedural value and, therefore, to sort of have first class procedures, in general, as being essential to doing very good modular programming.

Now, indeed there are many other ways to skin this cat. What you can do is take for each of the bad things that you have to worry about, you can make a special feature that covers that thing. You can make a package system. You can make a module system as in Ada, et cetera. And all of those work, or they cover little regions of it. The thing is that returning procedures as values cover all of those problems. And so it's the simplest mechanism that gives you the best modularity, gives you all of the known modularity mechanisms.

Well, I suppose it's time for the next break, thank you.

[MUSIC PLAYING]

PROFESSOR: Well, yesterday when you learned about streams, Hal worried to you about the order of evaluation and delayed arguments to procedures. The way we played with streams yesterday, it was the responsibility of the caller and the callee to both agree that an argument was delayed, and the callee must force the argument if it needs the answer. So there had to be a lot of hand shaking between the designer of a procedure and user of it over delayedness.

That turns out, of course, to be a fairly bad thing, it works all right with streams. But as a general thing, what you want is an idea to have a locus, a decision, a design decision in general, to have a place where it's made, explicitly, and notated in a clear way. And so it's not a very good idea to have to have an agreement, between the person who writes a procedure and the person who calls it, about such details as, maybe, the arguments of evaluation, the order of evaluation.

Although, that's not so bad. I mean, we have other such agreements like, the input's a number. But it would be nice if only one of these guys could take responsibility, completely. Now this is not a new idea.

ALGOL 60 had two different ways of calling a procedure. The arguments could be passed by name or by value. And what that meant was that a name argument was delayed. That when you passed an argument by name, that its value would only be obtained if you accessed that argument.

So what I'd like to do now is show you, first of all, a little bit about, again, we're going to make a modification to a language. In this case, we're going to add a feature. We're going to add the feature of, by name parameters, if you will, or delayed parameters. Because, in fact, the default in our Lisp system is by the value of a pointer. A

pointer is copied, but the data structure it points at is not. But I'd like to, in fact, show you is how you add name arguments as well.

Now again, why would we need such a thing? Well supposing we wanted to invent certain kinds of what otherwise would be special forms, reserve words? But I'd rather not take up reserve words. I want procedures that can do things like if.

If is special, or cond, or whatever it is. It's the same thing. It's special in that it determines whether or not to evaluate the consequent or the alternative based on the value of the predicate part of an expression. So taking the value of one thing determines whether or not to do something else.

Whereas all the procedures like plus, the ones that we can define right now, evaluate all of their arguments before application. So, for example, supposing I wish to be able to define something like the reverse of if in terms of if. Call it unless. We've a predicate, a consequent, and an alternative.

Now what I would like to sort of be able to do is say-- oh, I'll do it in terms of cond. Cond, if not the predicate, then take the consequent, otherwise, take the alternative. Now, what I'd like this to mean, is supposing I do something like this. I'd like this unless say if equals one, 0, then the answer is two, otherwise, the quotient of one and 0.

What I'd like that to mean is the result of substituting equal one, 0, and two, and the quotient of one, 0 for p, c, and a. I'd like that to mean, and this is funny, I'd like it to transform into or mean cond not equal one, 0, then the result is two, otherwise I want it to be the quotient one and 0.

Now, you know that if I were to type this into Lisp, I'd get a two. There's no problem with that. However, if I were to type this into Lisp, because all the arguments are evaluated before I start, then I'm going to get an error out of this. So that if the substitutions work at all, of course, I would get the right answer. But here's a case where the substitutions don't work. I don't get the wrong answer. I get no answer. I get an error.

Now, however, I'd like to be able to make my definition so that this kind of thing works. What I want to do is say something special about c and a. I want them to be delayed automatically. I don't want them to be evaluated at the time I call.

So I'm going to make a declaration, and then I'm going to see how to implement such a declaration. But again, I want you to say to yourself, oh, this is an interesting kluge he's adding in here. The piles of kluges make a big complicated mess. And is this going to foul up something else that might occur. First of all, is it syntactically unambiguous? Well, it will be syntactically unambiguous with what we've seen so far.

But what I'm going to do may, in fact, cause trouble. It may be that the thing I had will conflict with type

declarations I might want to add in the future for giving some system, some compiler or something, the ability to optimize given the types are known. Or it might conflict with other types of declarations I might want to make about the formal parameters. So I'm not making a general mechanism here where I can add declarations. And I would like to be able to do that. But I don't want to talk about that right now.

So here I'm going to do, I'm going to build a kluge. So we're going to define unless of a predicate-- and I'm going to call these by name-- the consequent, and name the alternative. Huh, huh-- I got caught in the corner. If not p then the result is c, else-- that's what I'd like. Where I can explicitly declare certain of the parameters to be delayed, to be computed later.

Now, this is actually a very complicated modification to an interpreter rather than a simple one. The ones you saw before, dynamic binding or adding indefinite argument procedures, is relatively simple. But this one changes a basic strategy. The problem here is that our interpreter, as written, evaluates a combination by evaluating the procedure, the operator producing the procedure, and evaluating the operands producing the arguments, and then doing apply of the procedure to the arguments.

However, here, I don't want to evaluate the operands to produce the arguments until after I examined the procedure to see what the procedure's declarations look like. So let's look at that. Here we have a changed evaluator. I'm starting with the simple lexical evaluator, not dynamic, but we're going to have to do something sort of similar in some ways. Because of the fact that, if I delay a procedure-- I'm sorry-- delay an argument to a procedure, I'm going to have to attach and environment to it.

Remember how Hal implemented delay. Hal implemented delay as being a procedure of no arguments which does some expression. That's what delay of the expression is. --of that expression. This turned into something like this.

Now, however, if I evaluate a lambda expression, I have to capture the environment. The reason why is because there are variables in there who's meaning I wish to derive from the context where this was written.

So that's why a lambda does the job. It's the right thing. And such that the forcing of a delayed expression was same thing as calling that with no arguments. It's just the opposite of this. Producing an environment of the call which is, in fact, the environment where this was defined with an extra frame in it that's empty. I don't care about that.

Well, if we go back to this slide, since it's the case, if we look at this for a second, everything is the same as it was before except the case of applications or combinations. And combinations are going to do two things. One, is I have to evaluate the procedure-- forget the procedure-- by evaluating the operator. That's what you see right

here. I have to make sure that that's current, that is not a delayed object, and evaluate that to the point where it's forced now. And then I have to somehow apply that to the operands. But I have to keep the environment, pass that environmental along. So some of those operands I may have to delay. I may have to attach that environment to those operands.

This is a rather complicated thing happening here. Looking at that in apply. Apply, well it has a primitive procedure thing just like before. But the compound one is a little more interesting. I have to evaluate the body, just as before, in an environment which is the result of binding some formal parameters to arguments in the environment. That's true.

The environment is the one that comes from the procedure now. It's a lexical language, statically bound. However, one thing I have to do is strip off the declarations to get the names of the variables. That's what this guy does, vnames. And the other thing I have to do is process these declarations, deciding which of these operands-- that's the operands now, as opposed to the arguments-- which of these operands to evaluate, and which of them are to be encapsulated in delays of some sort.

The other thing you see here is that we got a primitive, a primitive like plus, had better get at the real operands. So here is a place where we're going to have to force them. And we're going to look at what evlist is going to have to do a bunch of forces.

So we have two different kinds of evlist now. We have evlist and gevlist. Gevlist is going to wrap delays around some things and force others, evaluate others. And this guy's going to do some forcing of things. Just looking at this a little bit, this is a game you must play for yourself, you know. It's not something that you're going to see all possible variations on an evaluator talking to me. What you have to do is do this for yourself. And after you feel this, you play this a bit, you get to see all the possible design decisions and what they might mean, and how they interact with each other. So what languages might have in them. And what are some of the consistent sets that make a legitimate language. Whereas what things are complicated kluges that are just piles of junk.

So evlist of course, over here, just as I said, is a list of operands which are going to be undelayed after evaluation. So these are going to be forced, whatever that's going to mean. And gevlist, which is the next thing-- Thank you.

What we see here, well there's a couple of possibilities. Either it's a normal, ordinary thing, a symbol sitting there like the predicate in the unless, and that's what we have here. In which case, this is intended to be evaluated in applicative order. And it's, essentially, just what we had before. It's mapping eval down the list. In other words, I evaluate the first expression and continue gevlisting the CDR of the expression in the environment.

However, it's possible that this is a name parameter. If it's a name parameter, I want to put a delay in which

combines that expression, which I'm calling by name, with the environment that's available at this time and passing that as the parameter. And this is part of the mapping process that you see here.

The only other interesting place in this interpreter is cond. People tend to write this thing, and then they leave this one out. There's a place where you have to force. Conditionals have to know whether or not the answer is true or false. It's like a primitive. When you do a conditional, you have to force.

Now, I'm not going to look at any more of this in any detail. It isn't very exciting. And what's left is how you make delays. Well, delays are data structures which contain an expression, an environment, and a type on them.

And it says they're a thunk. That comes from ALGOL language, and it's claimed to be the sound of something being pushed on a stack. I don't know. I was not an ALGOLician or an ALGOLite or whatever, so I don't know. But that's what was claimed.

And undelay is something which will recursively undelay thunks until the thunk becomes something which isn't a thunk. This is the way you implement a call by name like thing in ALGOL. And that's about all there is.

Are there any questions?

AUDIENCE: Gerry?

PROFESSOR: Yes, Vesko?

AUDIENCE: I noticed you avoided calling by name in the primitive procedures, I was wondering what cause you have on that? You never need that?

PROFESSOR: Vesko is asking if it's ever reasonable to call a primitive procedure by name? The answer is, yes. There's one particular case where it's reasonable, actually two. Construction of a data structure like cons where making an array if you have arrays with any number of elements. It's unnecessary to evaluate those arguments. All you need is promises to evaluate those arguments if you look at them. If I cons together two things, then I could cons together the promises just as easily as I can cons together the things. And it's not even when I CAR CDR them that I have to look at them. That just gets out the promises and passes them to somebody.

That's why the lambda calculus definition, the Alonzo Church definition of CAR, CDR, and cons makes sense. It's because no work is done in CAR, CDR, and cons, it's just shuffling data, it's just routing, if you will.

However, the things that do have to look at data are things like plus. Because they have a look at the bits that the numbers are made out of, unless they're lambda calculus numbers which are funny. They have to look at the bits to be able to crunch them together to do the add.

So, in fact, data constructors, data selectors, and, in fact, things that side-effect data objects don't need to do any forcing in the laziest possible interpreters. On the other hand predicates on data structures have to. Is this a pair? Or is it a symbol? Well, you better find out. You got to look at it then. Any other questions? Oh, well, I suppose it's time for a break. Thank you. [MUSIC PLAYING] and