[MUSIC PLAYING BY J.S. BACH]

PROFESSOR: The last time we began having a look at how languages are constructed. Remember the main point that an evaluator for, LISP, say, has two main elements. There is EVAL, and EVAL's job is to take in an expression and an environment and turn that into a procedure and some arguments and pass that off to APPLY. And APPLY takes the procedure in the arguments, turns that back into, in a general case, another expression to be evaluated in another environment and passes that off to EVAL, which passes it to APPLY, and there's this whole big circle where things go around and around and around until you get either to some very primitive data or to a primitive procedure.

See, what this cycle has to do with is unwinding the means of combination and the means of abstraction in the language. So for instance, you have a procedure in LISP-- a procedure is a general way of saying, I want to be able to evaluate this expression for any value of the arguments, and that's sort of what's going on here. That's what APPLY does. It says the general thing coming in with the arguments reduces to the expression that's the body, and then if that's a compound expression or another procedure application, the thing will go around and around the circle. Anyway, that's sort of the basic structure of gee, pretty much any interpreter.

The other thing that you saw is once you have the interpreter in your hands, you have all this power to start playing with the language. So you can make it dynamically scoped, or you can put in normal order evaluation, or you can add new forms to the language, whatever you like. Or more generally, there's this notion of metalinguistic abstraction, which says that part of your perspective as an engineer, as a software engineer, but as an engineer in general is that you can gain control of complexity by inventing new languages sometimes. See, one way to think about computer programming is that it only incidentally has to do with getting a computer to do something. Primarily what a computer program has to do with, it's a way of expressing ideas with communicating ideas. And sometimes when you want to communicate new kinds of ideas, you'd like to invent new modes of expressing that.

Well, today we're going to apply this framework to build a new language. See, once we have the basic idea of the interpreter, you can pretty much go build any language that you like. So for example, we can go off and build Pascal. And gee, we would worry about syntax and parsing and various kinds of compiler optimizations, and there are people who make honest livings doing that, but at the level of abstraction that we're talking, a Pascal interpreter would not look very different at all from what you saw Gerry do last time.

Instead of that, we'll spend today building a really different language, a language that encourages you to think about programming not in terms of procedures, but in a really different way. And the lecture today is going to be at two levels simultaneously. On the one hand, I'm going to show you what this language looks like, and on the other

hand, I'll show you how it's implemented. And we'll build an implementation in LISP and see how that works.

And you should be drawing lessons on two levels. The first is to realize just how different a language can be. So if you think that the jump from Fortran to LISP is a big deal, you haven't seen anything yet. And secondly, you'll see that even with such a very different language, which will turn out to not have procedures at all and not talk about functions at all, there will still be this basic cycle of eval and apply that's unwinds the means of combination and the means an abstraction. And then thirdly, as kind of a minor but elegant technical point, you'll see a nice use of streams to avoid backtracking.

OK, well, I said that this language is very different. To explain that, let's go back to the very first idea that we talked about in this course, and that was the idea of the distinction between the declarative knowledge of mathematics-- the definition of a square root as a mathematical truth-- and the idea that computer science talks about the how to knowledge-- contrast that definition of square root with a program to compute a square root. That's where we started off. Well, wouldn't it be great if you could somehow bridge this gap and make a programming language which sort of did things, but you talked about it in terms of truth, in declarative terms? So that would be a programming language in which you specify facts. You tell it what is. You say what is true. And then when you want an answer, somehow the language has built into it automatically general kinds of how to knowledge so it can just take your facts and it can evolve these methods on its on using the facts you gave it and maybe some general rules of logic.

So for instance, I might go up to this program and start telling it some things. So I might tell it that the son of Adam is Abel. And I might tell it that the son of Adam is Cain. And I might tell it that the son of Cain is Enoch. And I might tell it that the son of Enoch is Irad, and all through the rest of our chapter whatever of Genesis, which ends up ending in Adah, by the way, and this shows the genealogy of Adah from Cain.

Anyway, once you tell it these facts, you might ask it things. You might go up to your language and say, who's the son of Adam? And you can very easily imagine having a little general purpose search program which would be able to go through and in response to that say, oh yeah, there are two answers: the son of Adam is Abel and the son of Adam is Cain. Or you might say, based on the very same facts, who is Cain the son of? And then you can imagine generating another slightly different search program which would be able to go through and checked for who is Cain, and son of, and come up with Adam. Or you might say, what's the relationship between Cain and Enoch? And again, a minor variant on that search program. You could figure out that it said son of.

But even here in this very simple example, what you see is that a single fact, see, a single fact like the son of Adam is Cain can be used to answer different kinds of questions. You can say, who's the son of, or you can say who's the son of Adam, or you can say what's the relation between Adam and Cain? Those are different questions

being run by different traditional procedures all based on the same fact.

And that's going to be the essence of the power of this programming style, that one piece of declarative knowledge can be used as the basis for a lot of different kinds of how-to knowledge, as opposed to the kinds of procedures we're writing where you sort of tell it what input you're giving it and what answer you want. So for instance, our square root program can perfectly well answer the question, what's the square root of 144? But in principle, the mathematical definition of square root tells you other things. Like it could say, what is 17 the square root of? And that would be have to be answered by a different program. So the mathematical definition, or in general, the facts that you give it are somehow unbiased as to what the question is. Whereas the programs we tend to write specifically because they are how-to knowledge tend to be looking for a specific answer. So that's going to be one characteristic of what we're talking about.

We can go on. We can imagine that we've given our language some sort of facts. Now let's give it some rules of inference. We can say, for instance, if the-- make up some syntax here-- if the son of x is y-- I'll put question marks to indicate variables here-- if the son of x is y and the son of y is z, then the grandson of x is z. So I can imagine telling my machine that rule and then being able to say, for instance, who's the grandson of Adam? Or who is Irad the grandson of? Or deduce all grandson relationships you possibly can from this information. We can imagine somehow the language knowing how to do that automatically.

Let me give you maybe a little bit more concrete example. Here's a procedure that merges two sorted lists. So x and y are two, say, lists of numbers, lists of distinct numbers, if you like, that are in increasing order. And what merge does is take two such lists and combine them into a list where everything's in increasing order, and this is a pretty easy programs that you ought to be able to write. It says, if x is empty, the answer is y. If y is empty, the answer is x. Otherwise, you compare the first two elements. So you pick out the first thing in x and the first thing in y, and then depending on which of those first elements is less, you stick the lower one on to the result a recursively merging, either chopping the first one off x or chopping the first one off y. That's a standard kind of program.

Let's look at the logic. Let's forget about the program and look at the logic on which that procedure is based. See, there's some logic which says, gee, if the first one is less, then we get the answer by sticking something onto the result of recursively merging the rest. So let's try and be explicit about what that logic is that's making the program work.

So here's one piece. Here's the piece of the program which recursively chops down x if the first thing in x is smaller. And if you want to be very explicit about what the logic is there, what's really going on is a deduction, which says, if you know that some list, that we'll call cdr of x, and y merged to form z, and you know that a is less

than the first thing in y, then you know that if you put a onto the cdr of x, then that result and y merge to form a onto z. And what that is, that's the underlying piece of logic-- I haven't written it as a program, I wrote it a sort of deduction that's underneath this particular clause that says we can use the recursion there. And then similar, here's the other clause just to complete it. The other clause is based on this piece of logic, which is almost the same and I won't go through it, and then there's the n cases where we tested for null, and that's based on the idea that for any x, x and the empty list merge to form an x, or for any y, the empty list and y merge to form y.

OK, so there's a piece of procedure and the logic on which it's based. And notice a big difference. The procedure looked like this: it said there was a box-- and all the things we've been doing have the characteristic we have boxes and things going in and things going out-- there was this box called merge, and in came an x and y, and out came an answer. That's the character of the procedure that we wrote. These rules don't look like that. These rules talk about a relation. There's some sort of relation that in those slides I called mrege-to-form. So I said x and y merge to form z, and somehow this is a function. Right? The answer is a function of x and y, and here what I have is a relation between three things. And I'm not going to specify which is the input and which is the output. And the reason I want to say that is because in principle, we could use exactly those same logic rules to answer a lot of different questions.

So we can say, for instance-- imagine giving our machine those rules of logic. Not the program, the underlying rules of logic. Then it ought to be able to say-- we could ask it-- 1, 3, 7 and 2, 4, 8 merge to form what? And that's a question it ought to be able to answer. That's exactly the same question that our list procedure answered.

But the exact same rules should also be able to answer a question like this: 1, 3, 7 and what merged to form 1, 2, 3, 4, 7, 8? The same rules of logic can answer this, although the procedure we wrote can't answer that question. Or we might be able to say what and what else merge to form-- what and what else merge to form 1, 2, 3, 4, 7, 8? And the thing should be able to go through, if it really can apply that logic, and deduce all, whatever is, 2 to the sixth answers to that question. It could be 1 and the rest, or it could be 1, 2 and the rest. Or it could be 1 and 3 and 7 and the rest. There's a whole bunch of answers. And in principle, the logic should be enough to deduce that.

So there are going to be two big differences in the kind of program we're going to look at and not only list, but essentially all the programming you've probably done so far in pretty much any language you can think of. The first is, we're not going to be computing functions. We're not going to be talking about things that take input and output. We're going to be talking about relations. And that means in principle, these relations don't have directionality. So the knowledge that you specify to answer this question, that same knowledge should also allow you to answer these other questions and conversely.

And the second issue is that since we're talking about relations, these relations don't necessarily have one answer. So that third question down there doesn't have a particular answer, it has a whole bunch of answers.

Well, that's where we're going. This style of programming, by the way, is called logic programming, for kind of obvious reasons. And people who do logic programming say that-- they have this little phrase-- they say the point of logic programming is that you use logic to express what is true, you use logic to check whether something is true, and you use logic to find out what is true. The best known logic programming language, as you probably know, is called Prolog. The language that we're going to implement this morning is something we call the query language, and it essentially has the essence of prologue. It can do about the same stuff, although it's a lot slower because we're going to implement it in LISP rather than building a particular compiler. We're going to interpret it on top of the LISP interpreter. But other than that, it can do about the same stuff as prolog. It has about the same power and about the same limitations.

All right, let's break for question.

STUDENT: Yes, could you please repeat what the three things you use logic programming to find? In other words, to find what is true, learn what is true-- what is the?

PROFESSOR: Right. Sort of a logic programmer's little catechism. You use logic to express what is true, like these rules. You use logic to check whether something is true, and that's the kind of question I didn't answer here. I might say-- another question I could put down here is to say, is it true that 1, 3, 7 and 2, 4, 8 merge to form 1, 2, 6, 10 And that same logic should be enough to say no. So I use logic to check what is true, and then you also use logic to find out what's true.

All right. Let's break.

[MUSIC PLAYING BY J.S. BACH]

[MUSIC ENDS]

[MUSIC PLAYING BY J.S. BACH]

PROFESSOR: OK, let's go ahead and take a look at this query language and operation. The first thing you might notice, when I put up that little biblical database, is that it's nice to be able to ask this language questions in relation to some collection of facts. So let's start off and make a little collection of facts. This is a tiny fragment of personnel records for a Boston high tech company, and here's a piece of the personnel records of Ben Bitdiddle. And Ben Bitdiddle is the computer wizard in this company, he's the underpaid computer wizard in this company. His supervisor is all Oliver Warbucks, and here's his address.

So the format is we're giving this information: job, salary, supervisor, address. And there are some other conventions. Computer here means that Ben works in the computer division, and his position in the computer division is wizard.

Here's somebody else. Alyssa, Alyssa P. Hacker is a computer programmer, and she works for Ben, and she lives in Cambridge. And there's another programmer who works for Ben who's Lem E. Tweakit. And there's a programmer trainee, who is Louis Reasoner, and he works for Alyssa. And the big wheel of the company doesn't work for anybody, right? That's Oliver Warbucks.

Anyway, what we're going to do is ask questions about that little world. And that'll be a sample world that we're going to do logic in. Let me just write up here, for probably the last time, what I said is the very most important thing you should get out of this course, and that is, when somebody tells you about a language, you say, fine--what are the primitives, what are the means of combination, how do you put the primitives together, and then how do you abstract them, how do you abstract the compound pieces so you can use them as pieces to make something more complicated? And we've said this a whole bunch of times already, but it's worth saying again.

Let's start. The primitives. Well, there's really only one primitive, and the primitive in this language is called a query. A primitive query. Let's look at some primitive queries. Job x. Who is a computer programmer? Or find every fact in the database that matches job of the x is computer programmer. And you see a little syntax here. Things without question marks are meant to be literal, question mark x means that's a variable, and this thing will match, for example, the fact that Alyssa P. Hacker is a computer programmer, or x is Alyssa P. Hacker. Or more generally, I could have something with two variables in it. I could say, the job of x is computer something, and that'll match computer wizard. So there's something here: type will match wizard, or type will match programmer, or x might match various certain things. So there are, in our little example, only three facts in that database that match that query.

Let's see, just to show you some syntax, the same query, this query doesn't match the job of x, doesn't match Lewis Reasoner, the reason for that is when I write something here, what I mean is that this is going to be a list of two symbols, of which the first is the word computer, and the second can be anything. And Lewis's job description here has three symbols, so it doesn't match.

And just to show you a little bit of syntax, the more general thing I might want to type is a thing with a dot here, and this is just standard this notation for saying, this is a list, of which the first element is the word computers, and THE REST, is something that I'll call type. So this one would match. Lewis's job is computer programmer trainee, and type here would be the cdr of this list. It would be the list programmer trainee. And that kind of dot processing is done automatically by the LISP reader.

Well, let's actually try this. The idea is I'm going to type in queries in this language, and answers will come out. Let's look at this. I can go up and say, who works in the computer division? Job of x is computer dot y. Doesn't matter what I call the dummy variables. It says the answers to that, and it's found four answers. Or I can go off and say, tell me about everybody's supervisor.

So I'll put in the query, the primitive query, the supervisor of x is y. There are all the supervisor relationships I know. Or I could go type in, who lives in Cambridge? So I can say, the address of x is Cambridge dot anything. And only one person lives in Cambridge.

OK, so those are primitive queries. And you see what happens to basic interaction with the system is you type in a query, and it types out all possible answers. Or another way to say that: it finds out all the possible values of those variables x and y or t or whatever I've called them, and it types out all ways of taking that query and instantiating it-- remember that from the rule system lecture-- instantiates the query with all possible values for those variables and then types out all of them.

And there are a lot of ways you can arrange a logic language. Prolog, for instance, does something slightly different. Rather than typing back your query, prolog would type out, x equals this and y equals that, or x sequels this and y equals that. And that's a very surface level thing, you can decide what you like. OK.

All right. So the primitives in this language? Only one, right? Primitive query.

OK. Means of combination. Let's look at some compound queries in this language. Here's one. This one says, tell me all the people who work in the computer division. Tell me all the people who work in the computer division together with their supervisors. The way I write that is the query is and. And the job of the x is computer something or other. And job of x is computer dot y. And the supervisor of x is z.

Tell me all the people in the computer division-- that's this-- together with their supervisors. And notice in this query I have three variables-- x, y, and z. And this x is supposed to be the same as that x. So x works in the computer division, and the supervisor of x is z.

Let's try another one. So one means of combination is and.

Who are all the people who make more than $30,000? And the salary of some person p is some amount a. And when I go and look at a, a is greater than $30,000. And LISP value here is a little piece of interface that interfaces the query language to the underlying LISP. And what the LISP value allows you to do is call any LISP predicate inside a query. So here I'm using the LISP predicate greater than, so I say LISP value. This I say and. So all the people whose salary is greater than $30,000.

Or here's a more complicated one. Tell me all the people who work in the computer division who do not have a supervisor who works in the computer division. and x works in the computer division. The job of x is computer dot y. And it's not the case that both x has a supervisor z and the job of z is computer something or other. All right, so again, this x has got to be that x, and this z is going to be that z.

And then you see another means a combination, not. All right, well, let's look at that. It works the same way.

I can go up to the machine and say and the job of the x is computer dot y. And the supervisor of x is z. And I typed that in like a query. And what it types back, what you see are the queries I typed in instantiated by all possible answers. And then you see there are a lot of answers.

All right. So the means of combination in this language-- and this is why it's called a logic language-- are logical operations. Means of combinations are things like AND and NOT and there's one I didn't show you, which is OR. And then I showed you LISP value, which is not logic, of course, but is a little special hack to interface that to LISP so you can get more power. Those are the means of combination.

OK, the means of abstraction. What we'd like to do-- let's go back for second and look at that last slide. We might like to take very complicated thing, the idea that someone works in a division but does not have a supervisor in the division. And as before, name that. Well, if someone works in a division and does not have a supervisor who works in that division, that means that person is a big shot. So let's make a rule that somebody x is a big shot in some department if x works in the department and it's not the case that x has a supervisor who works in the department.

So this is our means of abstraction. This is a rule. And a rule has three parts. The thing that says it's a rule. And then there's the conclusion of the rule. And then there's the body of the rule. And you can read this as a piece of logic which says, if you know that the body of the rule is true, then you can conclude that the conclusion is true. Or in order to deduce that x is a big shot in some department, it's enough to verify that. So that's what rules look like.

Let's go back and look at that merge example that I did before the break. Let's look at how that would look in terms of rules. I'm going to take the logic I put up and just change it into a bunch of rules in this format.

We have a rule. Remember, there was this thing merge-to-form. There is a rule that says, the empty list and y merge to form y. This is the rule conclusion. And notice this particular rule has no body. And in this language, a rule with no body is something that is always true. You can always assume that's true.

And there was another piece of logic that said anything in the empty list merged to form the anything. That's this. A rule y and the empty list merge to form y. Those corresponded to the two end cases in our merge procedure,

but now we're talking about logic, not about procedures.

Then we had another rule, which said if you know how shorter things merge, you can put them together. So this says, if you have a list x and y and z, and if you want to deduce that a dot x-- this means constant a onto x, or a list whose first thing is a and whose rest is x-- so if you want to deduce that a dot x and b dot y merge to form b dot c-- that would say you merge these two lists a x and b y and you're going to get something that starts with b-- you can deduce that if you know that it's the case both that a dot x and y merge to form z and a is larger than b.

So when I merge them, b will come first in the list. That's a little translation of the logic rule that I wrote in pseudo-English before.

And then just for completeness, here's the other case. a dot x and b dot y merge to form a dot z if x and b dot y merged to form z and b is larger than a. So that's a little program that I've typed in in this language, and now let's look at it run.

So I typed in the merge rules before, and I could use this like a procedure. I could say merge to form 1 and 3 and 2 and 7. So here I'm using it like the LISP procedure. Now it's going to think about that for a while and apply these rules. So it found an answer. Now it's going to see if there are any other answers but it doesn't know a priori there's only one answer. So it's sitting here checking all possibilities, and it says, no more. Done.

So there I've used those rules like a procedure. Or remember the whole point is that I can ask different kinds of questions. I could say merge to form, let's see, how about 2 and a. Some list of two elements which I know starts with 2, and the other thing I don't know, and x and some other list merge to form a 1, 2, 3 and 4. So now it's going to think about that. It's got to find-- so it found one possibility. It said a could be 3, and x could be the list 1, 4. And now, again, it's got to check because it doesn't a priori know that there aren't any other possibilities going on.

Or like I said, I could say something like merge to form, like, what and what else merge to form 1, 2, 3, 4, 5? Now it's going to think about that. And there are a lot of answers that it might get. And what you see is here you're really paying the price of slowness. And kind of for three reasons. One is that this language is doubly interpreted. Whereas in a real implementation, you would go compile this down to primitive operations. The other reason is that this particular algorithm for merges is doubly recursive. So it's going to take a very long time. And eventually, this is going to go through and find-- find what? Two to the fifth possible answers.

And you see they come out in some fairly arbitrary order, depending on which order it's going to be trying these rules. In fact, what we're going to do when they edit the videotape is speed all this up. Don't you like taking out these weights? And don't you wish you could do that in your demos?

Anyway, it's still grinding there. Anyway, there are 32 possibilities-- we won't wait for it to print out all of them.

OK, so the needs of abstraction in this language are rules. So we take some bunch of things that are put together with logic and we name them. And you can think of that as naming a particular pattern of logic. Or you can think of that as saying, if you want to deduce some conclusion, you can apply those rules of logic. And those are three elements of this language.

Let's break now, and then we'll talk about how it's actually implemented.

STUDENT: Does using LISP value primitive or whatever interfere with your means to go both directions on a query?

PROFESSOR: OK, that's a-- the question is, does using LISP value interfere with the ability to go both directions on the query? We haven't really talked about the implementation yet, but the answer is, yes, it can. In general, as we'll see at the end-- although I really won't to go into details-- it's fairly complicated, especially when you use either not or LISP value-- or actually, if you use anything besides only and, it becomes very complicated to say when these things will work. They won't work quite in all situations. I'll talk about that at the end of the second half today.

But the answer to your question is, yes, by dragging in a lot more power from LISP value, you lose some of the principal power of logic programming. That's a trade-off that you have to make.

OK, let's take a break.