[MUSIC PLAYING]

PROFESSOR: Well, Hal just told us how you build robust systems. The key idea was-- I'm sure that many of you don't really assimilate that yet-- but the key idea is that in order to make a system that's robust, it has to be insensitive to small changes, that is, a small change in the problem should lead to only a small change in the solution. There ought to be a continuity. The space of solutions ought to be continuous in this space of problems.

The way he was explaining how to do that was instead of solving a particular problem at every level of decomposition of the problem at the subproblems, where you solve the class of problems, which are a neighborhood of the particular problem that you're trying to solve. The way you do that is by producing a language at that level of detail in which the solutions to that class of problems is representable in that language. Therefore when you makes more changes to the problem you're trying to solve, you generally have to make only small local changes to the solution you've constructed, because at the level of detail you're working, there's a language where you can express the various solutions to alternate problems of the same type.

Well that's the beginning of a very important idea, the most important perhaps idea that makes computer science more powerful than most of the other kinds of engineering disciplines we know about. What we've seen so far is sort of how to use embedding of languages. And, of course, the power of embedding languages partly comes from procedures like this one that I showed you yesterday.

What you see here is the derivative program that we described yesterday. It's a procedure that takes a procedure as an argument and returns a procedure as a value. And using such things is very nice. You can make things like push combinators and all that sort of wonderful thing that you saw last time.

However, now I'm going to really muddy the waters. See this confuses the issue of what's the procedure and what is data, but not very badly. What we really want to do is confuse it very badly. And the best way to do that is to get involved with the manipulation of the algebraic expressions that the procedures themselves are expressed in.

So at this point, I want to talk about instead of things like on this slide, the derivative procedure

being a thing that manipulates a procedure-- this is a numerical method you see here. And what you're seeing is a representation of the numerical approximation to the derivative. That's what's here. In fact what I'd like to talk about is instead things that look like this. And what we have here are rules from a calculus book.

These are rules for finding the derivatives of the expressions that one might write in some algebraic language. It says things like a derivative of a constant is 0. The derivative of the valuable with respect to which you are taking the derivative is 1. The derivative of a constant times the function is the constant times the derivative of the function, and things like that. These are exact expressions. These are not numerical approximations. Can we make programs? And, in fact, it's very easy to make programs that manipulate these expressions.

Well let's see. Let's look at these rules in some detail. You all have seen these rules in your elementary calculus class at one time or another. And you know from calculus that it's easy to produce derivatives of arbitrary expressions. You also know from your elementary calculus that it's hard to produce integrals. Yet integrals and derivatives are opposites of each other. They're inverse operations. And they have the same rules. What is special about these rules that makes it possible for one to produce derivatives easily and integrals why it's so hard? Let's think about that very simply.

Look at these rules. Every one of these rules, when used in the direction for taking derivatives, which is in the direction of this arrow, the left side is matched against your expression, and the right side is the thing which is the derivative of that expression. The arrow is going that way. In each of these rules, the expressions on the right-hand side of the rule that are contained within derivatives are subexpressions, are proper subexpressions, of the expression on the left-hand side.

So here we see the derivative of the sum, with is the expression on the left-hand side is the sum of the derivatives of the pieces. So the rule of moving to the right are reduction rules. The problem becomes easier. I turn a big complicated problem it's lots of smaller problems and then combine the results, a perfect place for recursion to work.

If I'm going in the other direction like this, if I'm trying to produce integrals, well there are several problems you see here. First of all, if I try to integrate an expression like a sum, more than one rule matches. Here's one that matches. Here's one that matches. I don't know which one to take. And they may be different. I may get to explore different things. Also, the

expressions become larger in that direction. And when the expressions become larger, then there's no guarantee that any particular path I choose will terminate, because we will only terminate by accidental cancellation. So that's why integrals are complicated searches and hard to do.

Right now I don't want to do anything as hard as that. Let's work on derivatives for a while. Well, these roles are ones you know for the most part hopefully. So let's see if we can write a program which is these rules. And that should be very easy. Just write the program. See, because while I showed you is that it's a reduction rule, it's something appropriate for a recursion. And, of course, what we have for each of these rules is we have a case in some case analysis.

So I'm just going to write this program down. Now, of course, I'm going to be saying something you have to believe. Right? What you have to believe is I can represent these algebraic expressions, that I can grab their parts, that I can put them together. We've invented list structures so that you can do that. But you don't want to worry about that now. Right now I'm going to write the program that encapsulates these rules independent of the representation of the algebraic expressions.

You have a derivative of an expression with respect to a variable. This is a different thing than the derivative of the function. That's what we saw last time, that numerical approximation. It's something you can't open up a function. It's just the answers. The derivative of an expression is the way it's written. And therefore it's a syntactic phenomenon. And so a lot of what we're going to be doing today is worrying about syntax, syntax of expressions and things like that.

Well, there's a case analysis. Anytime we do anything complicated thereby a recursion, we presumably need a case analysis. It's the essential way to begin. And that's usually a conditional of some large kind. Well, what are their possibilities? the first rule that you saw is this something a constant? And what I'm asking is, is the expression a constant with respect to the variable given? If so, the result is 0, because the derivative represents the rate of change of something.

If, however, the expression that I'm taking the derivative of is the variable I'm varying, then this is the same variable, the expression var, then the rate of change of the expression with respect to the variable is 1. It's the same 1.

Well now there are a couple of other possibilities. It could, for example, be a sum. Well, I don't

know how I'm going to express sums yet. Actually I do. But I haven't told you yet. But is it a sum? I'm imagining that there's some way of telling. I'm doing a dispatch on the type of the expression here, absolutely essential in building languages. Languages are made out of different expressions. And soon we're going to see that in our more powerful methods of building languages on languages.

Is an expression a sum? If it's a sum, well, we know the rule for derivative of the sum is the sum of the derivatives of the parts. One of them is called the addend and the other is the augend. But I don't have enough space on the blackboard to such long names. So I'll call them A1 and A2.

I want to make a sum. Do you remember which is the sum for end or the menu end? Or was it the dividend and the divisor or something like that? Make sum of the derivative of the A1, I'll call it. It's the addend of the expression with respect to the variable, and the derivative of the A2 of the expression, because the two arguments, the addition with respect to the variable.

And another rule that we know is product rule, which is, if the expression is a product. By the way, it's a good idea when you're defining things, when you're defining predicates, to give them a name that ends in a question mark. This question mark doesn't mean anything. It's for us as an agreement. It's a conventional interface between humans so you can read my programs more easily. So I want you to, when you write programs, if you define a predicate procedure, that's something that rings true of false, it should have a name which ends in question mark. The list doesn't care. I care.

I want to make a sum. Because the derivative of a product is the sum of the first times the derivative of the second plus the second times the derivative of the first. Make a sum of two things, a product of, well, I'm going to say the M1 of the expression, and the derivative of the M2 of the expression with respect to the variable, and the product of the derivative of M1, the multiplier of the expression, with respect to the variable. It's the product of that and the multiplicand, M2, of the expression. Make that product. Make the sum. Close that case.

And, of course, I could add as many cases as I like here for a complete set of rules you might find in a calculus book. So this is what it takes to encapsulate those rules. And you see, you have to realize there's a lot of wishful thinking here. I haven't told you anything about how I'm going to make these representations. Now, once I've decided that this is my set of rules, I think it's time to play with the representation. Let's attack that/

Well, first of all, I'm going to play a pun. It's an important pun. It's a key to a sort of powerful idea. If I want to represent sums, and products, and differences, and quotients, and things like that, why not use the same language as I'm writing my program in? I write my program in algebraic expressions that look like the sum of the product on a and the product of x and x, and things like that. And the product of b and x and c, whatever, make that a sum of the product. Right now I don't want to have procedures with unknown numbers of arguments, a product of b and x and c.

This is list structure. And the reason why this is nice, is because any one of these objects has a property. I know where the car is. The car is the operator. And the operands are the successive cdrs the successive cars of the cdrs of the list that this is. It makes it very convenient. I have to parse it. It's been done for me. I'm using the embedding and Lisp to advantage.

So, for example, let's start using list structure to write down the representation that I'm implicitly assuming here. Well I have to define various things that are implied in this representation. Like I have to find out how to do a constant, how you do same variable. Let's do those first. That's pretty easy enough. Now I'm going to be introducing lots of primitives here, because these are the primitives that come with list structure.

OK, you define a constant. And what I mean by a constant, an expression that's constant with respect to a veritable, is that the expression is something simple. I can't take it into pieces, and yet it isn't that variable. I can't break it up, and yet it isn't that variable. That does not mean that there may be other expressions that are more complicated that are constants. It's just that I'm going to look at the primitive constants in this way.

So what this is, is it says that's it's the and. I can combine predicate expressions which return true or false with and. Something atomic, The expression is atomic, meaning it cannot be broken into parts. It doesn't have a car and a cdr. It's not a list. It adds a special test built into the system. And it's not identically equal to that variable. I'm representing my variable by things that are symbols which cannot be broken into pieces, things like x, and y, things like this. Whereas, of course, something like this can be broken up into pieces.

And the same variable of an expression with respect to a variable is, in fact, an atomic expression. I want to have an atomic expression, which is identical. I don't want to look inside this stuff anymore. These are primitive maybe. But it doesn't matter. I'm using things that are

given to me with a language. I'm not terribly interest in them

Now how do we deal with sums? Ah, something very interesting will happen. A sum is something which is not atomic and begins with the plus symbol. That's what it means. So here, I will define. An question is a sum if and it's not atomic and it's head, it's beginning, its car of the expression is the symbol plus.

Now you're about to see something you haven't seen before, this quotation. Why do I have that quotation there? Say your name,

AUDIENCE: Susanna.

PROFESSOR: Louder.

AUDIENCE: Susanna

PROFESSOR: Say your name.

AUDIENCE: Your name.

PROFESSOR: Louder.

AUDIENCE: Your name.

PROFESSOR: OK. What I'm showing you here is that the words of English are ambiguous. I was saying, say your name. I was also possibly saying say, your name. But that cannot be distinguished in speech. However, we do have a notation in writing, which is quotation for distinguishing these two possible meanings.

In particular, over here, in Lisp we have a notation for distinguishing these meetings. If I were to just write a plus here, a plus symbol, I would be asking, is the first element of the expression, is the operator position of the expression, the addition operator? I don't know. I would have to have written the addition operator there, which I can't write. However, this way I'm asking, is this the symbolic object plus, which normally stands for the addition operator? That's what I want. That's the question I want to ask. Now before I go any further, I want to point out the quotation is a very complex concept, and adding it to a language causes a great deal of troubles.

Consider the next slide. Here's a deduction which we should all agree with. We have, Alyssa is

smart and Alyssa is George's mother. This is an equality, is. From those two, we can deduce that George's mother is smart. Because we can always substitute equals for equals in expressions. Or can we?

Here's a case where we have "Chicago" has seven letters. The quotation means that I'm discussing the word Chicago, not what the word represents. Here I have that Chicago is the biggest city in Illinois. As a consequence of this, I would like to deduce that the biggest city in Illinois has seven letters. But that's manifestly false. Wow, it works.

OK, so once we have things like that, our language gets much more complicated. Because it's no longer true that things we tend to like to do with languages, like substituting equals for equals and getting right answers, are going to work without being very careful. We can't substitute into what's called referentially opaque contexts, of which a quotation is the prototypical type of referentially opaque context. If you know what that means, you can consult a philosopher. Presumably there is one in the room.

In any case, let's continue now, now that we at least have an operational understanding of a 2000-year-old issue that has to do with name, and mention, and all sorts of things like that. I have to define what I mean, how to make a sum of two things, an a1 and a2. And I'm going to do this very simply. It's a list of the symbol plus, and a1, and a2. And I can determine the first element. Define a1 to be cadr. I've just introduced another primitive. This is the car of the cdr of something.

You might want to know why car and cdr are names of these primitives, and why they've survived, even though they're much better ideas like left and right. We could have called them things like that. Well, first of all, the names come from the fact that in the great past, when Lisp was invented, I suppose in '58 or something, it was on a 704 or something like that, which had a machine. It was a machine that had an address register and a decrement register. And these were the contents of the address register and the decrement register. So it's an historical accident.

Now why have these names survived? It's because Lisp programmers like to talk to each other over the phone. And if you want to have a long sequence of cars and cdrs you might say, cdaddedr, which can be understood. But left of right or right of left is not so clear if you get good at it. So that's why we have these words. All of them up to four deep are defined typically in a Lisp system. A2 to be-- and, of course, you can see that if I looked at one of these

expressions like the sum of 3 and 5, what that is is a list containing the symbol plus, and a number 3, and a number 5. Then the car is the symbol plus. The car of the cdr. Well I take the cdr and then I take the car. And that's how I get to the 3. That's the first argument. And the car of the cdr of the cdr gets me to this one, the 5.

And similarly, of course, I can define what's going on with products. Let's do that very quickly. Is the expression a product? Yes if and if it's true, that's it's not atomic and it's EQ quote, the asterisk symbol, which is the operator for multiplication.

Make product of an M1 and an M2 to be list, quote, the asterisk operation and M1 and M2. and I define M1 to be cadr and M2 to be caddr. You get to be a good Lisp programmer because you start talking that way. I cdr down lists and console them up and so on.

Now, now that we have essentially a complete program for finding derivatives, you can add more rules if you like. What kind of behavior do we get out of it? I'll have to clear that x. Well, supposing I define foo here to be the sum of the product of ax square and bx plus c. That's the same thing we see here as the algebraic expression written in the more conventional notation over there.

Well, the derivative of foo with respect to x, which we can see over here, is this horrible, horrendous mess. I would like it to be 2ax plus b. But it's not. It's equivalent to it. What is it? I have here, what do I have? I have the derivative of the product of x and x. Over here is, of course, the sum of x times 1 and 1 times x.

Now, well, it's the first times the derivative of the second plus the second times the derivative of the first. It's right. That's 2x of course. a times 2x is 2ax plus 0X square doesn't count plus B over here plus a bunch of 0's. Well the answer is right. But I give people take off points on an exam for that, sadly enough. Let's worry about that in the next segment. Are there any questions? Yes?

AUDIENCE: If you had left the quote when you put the plus, then would that be referring to the procedure plus and could you do a comparison between that procedure and some other procedure if you wanted to?

PROFESSOR: Yes. Good question. If I had left this quotation off at this point, if I had left that quotation off at that point, then I would be referring here to the procedure which is the thing that plus is defined to be.

And indeed, I could compare some procedures with each other for identity. Now what that means is not clear right now. I don't like to think about it. Because I don't know exactly what it would need to compare procedures. There are reasons why that may make no sense at all. However, the symbols, we understand. And so that's why I put that quote in. I want to talk about the symbol that's apparent on the page. Any other questions? OK. Thank you. Let's take a break.

[MUSIC PLAYING]

PROFESSOR: Well, let's see. We've just developed a fairly plausible program for computing the derivatives of algebraic expressions. It's an incomplete program, if you would like to add more rules. And perhaps you might extend it to deal with uses of addition with any number of arguments and multiplication with any of the number of arguments. And that's all rather easy.

However, there was a little fly in that ointment. We go back to this slide. We see that the expressions that we get are rather bad. This is a rather bad expression. How do we get such an expression? Why do we have that expression? Let's look at this expression in some detail. Let's find out where all the pieces come from. As we see here, we have a sum-- just what I showed you at the end of the last time-- of X times 1 plus 1 time X. That is a derivative of this product. The produce of a times that, where a does not depend upon x, and therefore is constant with respect to x, is this sum, which goes from here all the way through here and through here. Because it is the first thing times the derivative of the second plus the derivative of the first times the second as the program we wrote on the blackboard indicated we should do.

And, of course, the product of bx over here manifests itself as B times 1 plus 0 times X because we see that B does not depend upon X. And so the derivative of B is this 0, and the derivative of X with respect itself is the 1. And, of course, the derivative of the sums over here turn into these two sums of the derivatives of the parts.

So what we're seeing here is exactly the thing I was trying to tell you about with Fibonacci numbers a while ago, that the form of the process is expanded from the local rules that you see in the procedure, that the procedure represents a set of local rules for the expansion of this process. And here, the process left behind some stuff, which is the answer. And it was constructed by the walk it takes of the tree structure, which is the expression. So every part in the answer we see here derives from some part of the problem.

Now, we can look at, for example, the derivative of foo, which is ax square plus bx plus c, with respect to other things, like here, for example, we can see that the derivative of foo with respect to a. And it's very similar. It's, in fact, the identical algebraic expression, except for the fact that theses 0's and 1's are in different places. Because the only degree of freedom we have in this tree walk is what's constant with respect to the variable we're taking the derivative with respect to and was the same variable.

In other words, if we go back to this blackboard and we look, we have no choice what to do when we take the derivative of the sum or a product. The only interesting place here is, is the expression the variable, or is the expression a constant with respect to that variable for very, very small expressions? In which case we get various 1's and 0's, which if we go back to this slide, we can see that the 0's that appear here, for example, this 1 over here in derivative of foo with respect to A, which gets us an X square, because that 1 gets the multiply of X and X into the answer, that 1 is 0. Over here, we're not taking the derivative of foo with respect to c. But the shapes of these expressions are the same. See all those shapes. They're the same.

Well is there anything wrong with our rules? No. They're the right rules. We've been through this one before. One of the things you're going to begin to discover is that there aren't too many good ideas. When we were looking at rational numbers yesterday, the problem was that we got 6/8 rather then 3/4. The answer was unsimplified. The problem, of course, is very similar. There are things I'd like to be identical by simplification that don't become identical. And yet the rules for doing addition a multiplication of rational numbers were correct.

So the way we might solve this problem is do the thing we did last time, which always works. If something worked last time it ought to work again. It's changed representation. Perhaps in the representation we could put in a simplification step that produces a simplified representation. This may not always work, of course. I'm not trying to say that it always works. But it's one of the pieces of artillery we have in our war against complexity.

You see, because we solved our problem very carefully. What we've done, is we've divided the world in several parts. There are derivatives rules and general rules for algebra of some sort at this level of detail. and i have an abstraction barrier. And i have the representation of the algebraic expressions, list structure.

And in this barrier, I have the interface procedures. I have constant, and things like same-var. I have things like sum, make-sum. I have A1, A2. I have products and things like that, all the

other things I might need for various kinds of algebraic expressions.

Making this barrier allows me to arbitrarily change the representation without changing the rules that are written in terms of that representation. So if I can make the problem go away by changing representation, the composition of the problem into these two parts has helped me a great deal.

So let's take a very simple case of this. What was one of the problems? Let's go back to this transparency again. And we see here, oh yes, there's horrible things like here is the sum of an expression and 0. Well that's no reason to think of it as anything other than the expression itself. Why should the summation operation have made up this edition? It can be smarter than that. Or here, for example, is a multiplication of something by 1. It's another thing like that. Or here is a product of something with 0, which is certainly 0. So we won't have to make this construction.

So why don't we just do that? We need to change the way the representation works, almost here. Make-sum to be. Well, now it's not something so simple. I'm not going to make a list containing the symbol plus and things unless I need to.

Well, what are the possibilities? I have some sort of cases here. If I have numbers, if anyone is a number-- and here's another primitive I've just introduced, it's possible to tell whether something's number-- and if number A2, meaning they're not symbolic expressions, then why not do the addition now? The result is just a plus of A1 and A2.

I'm not asking if these represent numbers. Of course all of these symbols represent numbers. I'm talking about whether the one I've got is the number 3 right now. And, for example, supposing A1 is a number, and it's equal to 0, well then the answer is just A2. There is no reason to make anything up. And if A2 is a number, and equal A20, then the result is A1.

And only if I can't figure out something better to do with this situation, well, I can start a list. Otherwise I want the representation to be the list containing the quoted symbol plus, and A1, and A2.

And, of course, a very similar thing can be done for products. And I think I'll avoid boring you with them. I was going to write it on the blackboard. I don't think it's necessary. You know what to do. It's very simple.

But now, let's just see the kind of results we get out of changing our program in this way. Well, here's the derivatives after having just changed the constructors for expressions. The same foo, aX square plus bX plus c, and what I get is nothing more than the derivative of that is 2aX plus B.

Well, it's not completely simplified. I would like to collect common terms and sums. Well, that's more work. And, of course, programs to do this sort of thing are huge and complicated. Algebraic simplification, it's a very complicated mess. There's a very famous program you may have heard of called Maxima developed at MIT in the past, which is 5,000 pages of Lisp code, mostly the algebraic simplification operations.

There we see the derivative of foo. In fact, X is at something I wouldn't take off more than 1 point for on an elementary calculus class. And the derivative of foo with respect to a, well it's gone down to X times X, which isn't so bad. And the derivative of foo with respect to b is just X itself. And the derivative of foo with respect to c comes out 1. So I'm pretty pleased with this.

What you've seen is, of course, a little bit contrived, carefully organized example to show you how we can manipulate algebraic expressions, how we do that abstractly in terms of abstract syntax rather than concrete syntax and how we can use the abstraction to control what goes on in building these expressions.

But the real story isn't just such a simple thing as that. The real story is, in fact, that I'm manipulating these expressions. And the expressions are the same expressions-- going back to the slide-- as the ones that are Lisp expressions. There's a pun here. I've chosen my representation to be the same as the representation in my language of similar things. By doing so, I've invoked a necessity. I created the necessity to have things like quotation because of the fact that my language is capable of writing expressions that talk about expressions of the language. I need to have something that says, this is an expression I'm talking about rather than this expression is talking about something, and I want to talk about that.

So quotation stops and says, I'm talking about this expression itself. Now, given that power, if I can manipulate expressions of the language, I can begin to build even much more powerful layers upon layers of languages. Because I can write languages that not only are embedded in Lisp or whatever language you start with, but languages that are completely different, that are just, if we say, interpreted in Lisp or something like that.

We'll get to understand those words more in the future. But right now I just want to leave you

with the fact that we've hit a line which gives us tremendous power. And this point we've bought a sledgehammer. We have to be careful to what flies when we apply it. Thank you.

[MUSIC PLAYING]