6.001 Structure and Interpretation of Computer Programs, Spring 2005
Transcript – 1A: Overview and Introduction to Lisp

[MUSIC PLAYING] PROFESSOR: I'd like to welcome you to this course on computer science. Actually, that's a terrible way to start. Computer science is a terrible name for this business. First of all, it's not a science. It might be engineering or it might be art, but we'll actually see that computer so-called science actually has a lot in common with magic, and we'll see that in this course.

So it's not a science. It's also not really very much about computers. And it's not about computers in the same sense that physics is not really about particle accelerators, and biology is not really about microscopes and petri dishes. And it's not about computers in the same sense that geometry is not really about using surveying instruments.

In fact, there's a lot of commonality between computer science and geometry. Geometry, first of all, is another subject with a lousy name. The name comes from Gaia, meaning the Earth, and metron, meaning to measure. Geometry originally meant measuring the Earth or surveying.

And the reason for that was that, thousands of years ago, the Egyptian priesthood developed the rudiments of geometry in order to figure out how to restore the boundaries of fields that were destroyed in the annual flooding of the Nile. And to the Egyptians who did that, geometry really was the use of surveying instruments.

Now, the reason that we think computer science is about computers is pretty much the same reason that the Egyptians thought geometry was about surveying instruments. And that is, when some field is just getting started and you don't really understand it very well, it's very easy to confuse the essence of what you're doing with the tools that you use. And indeed, on some absolute scale of things, we probably know less about the essence of computer science than the ancient Egyptians really knew about geometry.

Well, what do I mean by the essence of computer science? What do I mean by the essence of geometry? See, it's certainly true that these Egyptians went off and used surveying instruments, but when we look back on them after a couple of thousand years, we say, gee, what they were doing, the important stuff they were doing, was to begin to formalize notions about space and time, to start a way of talking about mathematical truths formally.

That led to the axiomatic method. That led to sort of all of modern mathematics, figuring out a way to talk precisely about so-called declarative knowledge, what is true.

Well, similarly, I think in the future people will look back and say, yes, those primitives in the 20th century were fiddling around with these gadgets called computers, but really what they were doing is starting to learn how to formalize intuitions about process, how to do things, starting to develop a way to talk precisely about how-to knowledge, as opposed to geometry that talks about what is true.

Let me give you an example of that. Let's take a look. Here is a piece of mathematics that says what a square root is. The square root of X is the number Y, such that Y squared is equal to X and Y is greater than 0. Now, that's a fine piece of mathematics, but just telling you what a square root is doesn't really say anything about how you might go out and find one.

So let's contrast that with a piece of imperative knowledge, how you might go out and find a square root. This, in fact, also comes from Egypt, not ancient, ancient Egypt. This is an algorithm due to Heron of Alexandria, called how to find a square root by successive averaging. And what it says is that, in order to find a square root, you make a guess, you improve that guess-- and the way you improve the guess is to average the guess and X over the guess, and we'll talk a little bit later about why that's a reasonable thing-- and you keep improving the guess until it's good enough.

That's a method. That's how to do something as opposed to declarative knowledge that says what you're looking for. That's a process. Well, what's a process in general? It's kind of hard to say. You can think of it as like a magical spirit that sort of lives in the computer and does something. And the thing that directs a process is a pattern of rules called a procedure.

So procedures are the spells, if you like, that control these magical spirits that are the processes. I guess you know everyone needs a magical language, and sorcerers, real sorcerers, use ancient Arcadian or Sumerian or Babylonian or whatever. We're going to conjure our spirits in a magical language called Lisp, which is a language designed for talking about, for casting the spells that are procedures to direct the processes.

Now, it's very easy to learn Lisp. In fact, in a few minutes, I'm going to teach you, essentially, all of Lisp. I'm going to teach you, essentially, all of the rules. And you shouldn't find that particularly surprising. That's sort of like saying it's very easy to learn the rules of

chess. And indeed, in a few minutes, you can tell somebody the rules of chess. But of course, that's very different from saying you understand the implications of those rules and how to use those rules to become a masterful chess player.

Well, Lisp is the same way. We're going to state the rules in a few minutes, and it'll be very easy to see. But what's really hard is going to be the implications of those rules, how you exploit those rules to be a master programmer. And the implications of those rules are going to take us the, well, the whole rest of the subject and, of course, way beyond.

OK, so in computer science, we're in the business of formalizing this sort of how-to imperative knowledge, how to do stuff. And the real issues of computer science are, of course, not telling people how to do square roots. Because if that was all it was, there wouldn't be no big deal. The real problems come when we try to build very, very large systems, computer programs that are thousands of pages long, so long that nobody can really hold them in their heads all at once.

And the only reason that that's possible is because there are techniques for controlling the complexity of these large systems. And these techniques that are controlling complexity are what this course is really about. And in some sense, that's really what computer science is about.

Now, that may seem like a very strange thing to say. Because after all, a lot of people besides computer scientists deal with controlling complexity. A large airliner is an extremely complex system, and the aeronautical engineers who design that are dealing with immense complexity. But there's a difference between that kind of complexity and what we deal with in computer science.

And that is that computer science, in some sense, isn't real. You see, when an engineer is designing a physical system, that's made out of real parts. The engineers who worry about that have to address problems of tolerance and approximation and noise in the system. So for example, as an electrical engineer, I can go off and easily build a one-stage amplifier or a two-stage amplifier, and I can imagine cascading a lot of them to build a million-stage amplifier. But it's ridiculous to build such a thing, because long before the millionth stage, the thermal noise in those components way at the beginning is going to get amplified and make the whole thing meaningless.

Computer science deals with idealized components. We know as much as we want about these little program and data pieces that we're fitting things together. We don't have to worry about tolerance. And that means that, in building a large program, there's not all that much difference between what I can build and what I can imagine, because the parts are these abstract entities that I know as much as I want.

I know about them as precisely as I'd like. So as opposed to other kinds of engineering, where the constraints on what you can build are the constraints of physical systems, the constraints of physics and noise and approximation, the constraints imposed in building large software systems are the limitations of our own minds.

So in that sense, computer science is like an abstract form of engineering. It's the kind of engineering where you ignore the constraints that are imposed by reality. Well, what are some of these techniques? They're not special to computer science. First technique, which is used in all of engineering, is a kind of abstraction called black-box abstraction. Take something and build a box about it.

Let's see, for example, if we looked at that square root method, I might want to take that and build a box. That sort of says, to find the square root of X. And that might be a whole complicated set of rules. And that might end up being a kind of thing where I can put in, say, 36 and say, what's the square root of 36? And out comes 6.

And the important thing is that I'd like to design that so that if George comes along and would like to compute, say, the square root of A plus the square root of B, he can take this thing and use it as a module without having to look inside and build something that looks like this, like an A and a B and a square root box and another square root box and then something that adds that would put out the answer.

And you can see, just from the fact that I want to do that, is from George's point of view, the internals of what's in here should not be important. So for instance, it shouldn't matter that, when I wrote this, I said I want to find the square root of X. I could have said the square root of Y, or the square root of A, or anything at all. That's the fundamental notion of putting something in a box using black-box abstraction to suppress detail.

And the reason for that is you want to go off and build bigger boxes. Now, there's another reason for doing black-box abstraction other than you want to suppress detail for building bigger boxes. Sometimes you want to say that your way of doing something, your how-to

method, is an instance of a more general thing, and you'd like your language to be able to express that generality.

Let me show you another example sticking with square roots. Let's go back and take another look at that slide with the square root algorithm on it. Remember what that says. That says, in order to do something, I make a guess, and I improve that guess, and I sort of keep improving that guess. So there's the general strategy of, I'm looking for something, and the way I find it is that I keep improving it. Now, that's a particular case of another kind of strategy for finding a fixed point of something.

So you have a fixed point of a function. A fixed point of a function is something, is a value. A fixed point of a function F is a value Y, such that F of Y equals Y. And the way I might do that is start with a guess. And then if I want something that doesn't change when I keep applying F, is I'll keep applying F over and over until that result doesn't change very much. So there's a general strategy.

And then, for example, to compute the square root of X, I can try and find a fixed point of the function which takes Y to the average of X/Y. And the idea that is that if I really had Y equal to the square root of X, then Y and X/Y would be the same value. They'd both be the square root of X, because X over the square root of X is the square root of X.

And so the average if Y were equal to the square of X, then the average wouldn't change. So the square root of X is a fixed point of that particular function. Now, what I'd like to have, I'd like to express the general strategy for finding fixed points. So what I might imagine doing, is to find, is to be able to use my language to define a box that says "fixed point," just like I could make a box that says "square root." And I'd like to be able to express this in my language.

So I'd like to express not only the imperative how-to knowledge of a particular thing like square root, but I'd like to be able to express the imperative knowledge of how to do a general thing like how to find fixed point. And in fact, let's go back and look at that slide again.

See, not only is this a piece of imperative knowledge, how to find a fixed point, but over here on the bottom, there's another piece of imperative knowledge which says, one way to compute square root is to apply this general fixed point method. So I'd like to also be able to express that imperative knowledge. What would that look like?

That would say, this fixed point box is such that if I input to it the function that takes Y to the average of Y and X/Y, then what should come out of that fixed point box is a method for finding square roots. So in these boxes we're building, we're not only building boxes that you input numbers and output numbers, we're going to be building in boxes that, in effect, compute methods like finding square root.

And my take is their inputs functions, like Y goes to the average of Y and X/Y. The reason we want to do that, the reason this is a procedure, will end up being a procedure, as we'll see, whose value is another procedure, the reason we want to do that is because procedures are going to be our ways of talking about imperative knowledge. And the way to make that very powerful is to be able to talk about other kinds of knowledge.

So here is a procedure that, in effect, talks about another procedure, a general strategy that itself talks about general strategies. Well, our first topic in this course-- there'll be three major topics-- will be black-box abstraction. Let's look at that in a little bit more detail. What we're going to do is we will start out talking about how Lisp is built up out of primitive objects. What does the language supply with us? And we'll see that there are primitive procedures and primitive data.

Then we're going to see, how do you take those primitives and combine them to make more complicated things, means of combination? And what we'll see is that there are ways of putting things together, putting primitive procedures together to make more complicated procedures. And we'll see how to put primitive data together to make compound data.

Then we'll say, well, having made those compounds things, how do you abstract them? How do you put those black boxes around them so you can use them as components in more complex things? And we'll see that's done by defining procedures and a technique for dealing with compound data called data abstraction.

And then, what's maybe the most important thing, is going from just the rules to how does an expert work? How do you express common patterns of doing things, like saying, well,

there's a general method of fixed point and square root is a particular case of that? And we're going to use-- I've already hinted at it-- something called higher-order procedures, namely procedures whose inputs and outputs are themselves procedures. And then we'll also see something very interesting. We'll see, as we go further and further on and become more abstract, there'll be very-- well, the line between what we consider to be data and what we consider to be procedures is going to blur at an incredible rate.

Well, that's our first subject, black-box abstraction. Let's look at the second topic. I can introduce it like this. See, suppose I want to express the idea-- remember, we're talking about ideas-- suppose I want to express the idea that I can take something and multiply it by the sum of two other things. So for example, I might say, if I had 1 and 3 and multiply that by 2, I get 8. But I'm talking about the general idea of what's called linear combination, that you can add two things and multiply them by something else.

It's very easy when I think about it for numbers, but suppose I also want to use that same idea to think about, I could add two vectors, a1 and a2, and then scale them by some factor x and get another vector. Or I might say, I want to think about a1 and a2 as being polynomials, and I might want to add those two polynomials and then multiply them by 2 to get a more complicated one.

Or a1 and a2 might be electrical signals, and I might want to think about summing those two electrical signals and then putting the whole thing through an amplifier, multiplying it by some factor of 2 or something. The idea is I want to think about the general notion of that.

Now, if our language is going to be good language for expressing those kind of general ideas, if I really, really can do that, I'd like to be able to say I'm going to multiply by x the sum of a1 and a2, and I'd like that to express the general idea of all different kinds of things that a1 and a2 could be. Now, if you think about that, there's a problem, because after all, the actual primitive operations that go on in the machine are obviously going to be different if I'm adding two numbers than if I'm adding two polynomials, or if I'm adding the representation of two electrical signals or wave forms.

Somewhere, there has to be the knowledge of the kinds of various things that you can add and the ways of adding them. Now, to construct such a system, the question is, where do I put that knowledge? How do I think about the different kinds of choices I have? And if tomorrow George comes up with a new kind of object that might be added and multiplied, how do I add George's new object to the system without screwing up everything that was already there?

Well, that's going to be the second big topic, the way of controlling that kind of complexity. And the way you do that is by establishing conventional interfaces, agreed upon ways of plugging things together. Just like in electrical engineering, people have standard impedances for connectors, and then you know if you build something with one of those standard impedances, you can plug it together with something else.

So that's going to be our second large topic, conventional interfaces. What we're going to see is, first, we're going to talk about the problem of generic operations, which is the one I alluded to, things like "plus" that have to work with all different kinds of data. So we talk about generic operations. Then we're going to talk about really large-scale structures. How do you put together very large programs that model the kinds of complex systems in the real world that you'd like to model?

And what we're going to see is that there are two very important metaphors for putting together such systems. One is called object-oriented programming, where you sort of think of your system as a kind of society full of little things that interact by sending information between them. And then the second one is operations on aggregates, called streams, where you think of a large system put together kind of like a signal processing engineer puts together a large electrical system.

That's going to be our second topic. Now, the third thing we're going to come to, the third basic technique for controlling complexity, is making new languages. Because sometimes, when you're sort of overwhelmed by the complexity of a design, the way that you control that complexity is to pick a new design language. And the purpose of the new design language will be to highlight different aspects of the system. It will suppress some kinds of details and emphasize other kinds of details.

This is going to be the most magical part of the course. We're going to start out by actually looking at the technology for building new computer languages. The first thing we're going to do is actually build in Lisp. We're going to express in Lisp the process of interpreting Lisp itself. And that's going to be a very sort of self-circular thing. There's a little mystical symbol that has to do with that. The process of interpreting Lisp is sort of a giant wheel of two processes, apply and eval, which sort of constantly reduce expressions to each other.

Then we're going to see all sorts of other magical things. Here's another magical symbol. This is sort of the Y operator, which is, in some sense, the expression of infinity inside our procedural language. We'll take a look at that. In any case, this section of the course is

called Metalinguistic Abstraction, abstracting by talking about how you construct new languages.

As I said, we're going to start out by looking at the process of interpretation. We're going to look at this apply-eval loop, and build Lisp. Then, just to show you that this is very general, we're going to use exactly the same technology to build a very different kind of language, a so-called logic programming language, where you don't really talk about procedures at all that have inputs and outputs. What you do is talk about relations between things.

And then finally, we're going to talk about how you implement these things very concretely on the very simplest kind of machines. We'll see something like this. This is a picture of a chip, which is the Lisp interpreter that we will be talking about then in hardware. Well, there's an outline of the course, three big topics. Black-box abstraction, conventional interfaces, metalinguistic abstraction. Now, let's take a break now and then we'll get started.

[MUSIC PLAYING]

Let's actually start in learning Lisp now. Actually, we'll start out by learning something much more important, maybe the very most important thing in this course, which is not Lisp, in particular, of course, but rather a general framework for thinking about languages that I already alluded to. When somebody tells you they're going to show you a language, what you should say is, what I'd like you to tell me is what are the primitive elements? What does the language come with?

Then, what are the ways you put those together? What are the means of combination? What are the things that allow you to take these primitive elements and build bigger things out of them? What are the ways of putting things together?

And then, what are the means of abstraction? How do we take those complicated things and draw those boxes around them? How do we name them so that we can now use them as if they were primitive elements in making still more complex things? And so on, and so on, and so on. So when someone says to you, gee, I have a great new computer language, you don't say, how many characters does it take to invert a matrix? It's irrelevant.

What you say is, if the language did not come with matrices built in or with something else built in, how could I then build that thing? What are the means of combination which would allow me to do that? And then, what are the means of abstraction which allow me then to use those as elements in making more complicated things yet?

Well, we're going to see that Lisp has some primitive data and some primitive procedures. In fact, let's really start. And here's a piece of primitive data in Lisp, number 3. Actually, if I'm being very pedantic, that's not the number 3. That's some symbol that represents Plato's concept of the number 3. And here's another. Here's some more primitive data in Lisp, 17.4. Or actually, some representation of 17.4.

And here's another one, 5. Here's another primitive object that's built in Lisp, addition. Actually, to use the same kind of pedantic-- this is a name for the primitive method of adding things. Just like this is a name for Plato's number 3, this is a name for Plato's concept of how you add things. So those are some primitive elements. I can put them together. I can say, gee, what's the sum of 3 and 17.4 and 5?

And the way I do that is to say, let's apply the sum operator to these three numbers. And I should get, what? 8, 17. 25.4. So I should be able to ask Lisp what the value of this is, and it will return 25.4. Let's introduce some names. This thing that I typed is called a combination. And a combination consists, in general, of applying an operator-- so this is an operator-- to some operands. These are the operands.

And of course, I can make more complex things. The reason I can get complexity out of this is because the operands themselves, in general, can be combinations. So for instance, I could say, what is the sum of 3 and the product of 5 and 6 and 8 and 2? And I should get-- let's see-- 30, 40, 43. So Lisp should tell me that that's 43.

Forming combinations is the basic needs of combination that we'll be looking at. And then, well, you see some syntax here. Lisp uses what's called prefix notation, which means that the operator is written to the left of the operands. It's just a convention. And notice, it's fully parenthesized. And the parentheses make it completely unambiguous. So by looking at this, I can see that there's the operator, and there are 1, 2, 3, 4 operands.

And I can see that the second operand here is itself some combination that has one operator and two operands. Parentheses in Lisp are a little bit, or are very unlike parentheses in conventional mathematics. In mathematics, we sort of use them to mean

grouping, and it sort of doesn't hurt if sometimes you leave out parentheses if people understand that that's a group. And in general, it doesn't hurt if you put in extra parentheses, because that maybe makes the grouping more distinct.

Lisp is not like that. In Lisp, you cannot leave out parentheses, and you cannot put in extra parentheses, because putting in parentheses always means, exactly and precisely, this is a combination which has meaning, applying operators to operands. And if I left this out, if I left those parentheses out, it would mean something else.

In fact, the way to think about this, is really what I'm doing when I write something like this is writing a tree. So this combination is a tree that has a plus and then a 3 and then a something else and an 8 and a 2. And then this something else here is itself a little subtree that has a star and a 5 and a 6.

And the way to think of that is, really, what's going on are we're writing these trees, and parentheses are just a way to write this two-dimensional structure as a linear character string. Because at least when Lisp first started and people had teletypes or punch cards or whatever, this was more convenient. Maybe if Lisp started today, the syntax of Lisp would look like that.

Well, let's look at what that actually looks like on the computer. Here I have a Lisp interaction set up. There's a editor. And on the top, I'm going to type some values and ask Lisp what they are. So for instance, I can say to Lisp, what's the value of that symbol? That's 3. And I ask Lisp to evaluate it. And there you see Lisp has returned on the bottom, and said, oh yeah, that's 3.

Or I can say, what's the sum of 3 and 4 and 8? What's that combination? And ask Lisp to evaluate it. That's 15. Or I can type in something more complicated. I can say, what's the sum of the product of 3 and the sum of 7 and 19.5? And you'll notice here that Lisp has something built in that helps me keep track of all these parentheses. Watch as I type the next closed parentheses, which is going to close the combination starting with the star. The opening one will flash.

Here, I'll rub those out and do it again. Type close, and you see that closes the plus. Close again, that closes the star. Now I'm back to the sum, and maybe I'm going to add that all to 4. That closes the plus. Now I have a complete combination, and I can ask Lisp for the value of that.

That kind of paren balancing is something that's built into a lot of Lisp systems to help you keep track, because it is kind of hard just by hand doing all these parentheses. There's another kind of convention for keeping track of parentheses. Let me write another complicated combination. Let's take the sum of the product of 3 and 5 and add that to something.

And now what I'm going to do is I'm going to indent so that the operands are written vertically. Which the sum of that and the product of 47 and-- let's say the product of 47 with a difference of 20 and 6.8. That means subtract 6.8 from 20. And then you see the parentheses close. Close the minus. Close the star.

And now let's get another operator. You see the Lisp editor here is indenting to the right position automatically to help me keep track. I'll do that again. I'll close that last parentheses again. You see it balances the plus. Now I can say, what's the value of that?

So those two things, indenting to the right level, which is called pretty printing, and flashing parentheses, are two things that a lot of Lisp systems have built in to help you keep track. And you should learn how to use them. Well, those are the primitives. There's a means of combination. Now let's go up to the means of abstraction.

I'd like to be able to take the idea that I do some combination like this, and abstract it and give it a simple name, so I can use that as an element. And I do that in Lisp with "define." So I can say, for example, define A to be the product of 5 and 5. And now I could say, for example, to Lisp, what is the product of A and A? And this should be 25, and this should be 625.

And then, crucial thing, I can now use A-- here I've used it in a combination-- but I could use that in other more complicated things that I name in turn. So I could say, define B to be the sum of, we'll say, A and the product of 5 and A. And then close the plus.

Let's take a look at that on the computer and see how that looks. So I'll just type what I wrote on the board. I could say, define A to be the product of 5 and 5. And I'll tell that to Lisp. And notice what Lisp responded there with was an A in the bottom. In general, when you type in a definition in Lisp, it responds with the symbol being defined.

Now I could say to Lisp, what is the product of A and A? And it says that's 625. I can define B to be the sum of A and the product of 5 and A. Close a paren closes the star. Close the plus. Close the "define." Lisp says, OK, B, there on the bottom. And now I can say to Lisp, what's the value of B?

And I can say something more complicated, like what's the sum of A and the quotient of B and 5? That slash is divide, another primitive operator. I've divided B by 5, added it to A. Lisp says, OK, that's 55.

So there's what it looks like. There's the basic means of defining something. It's the simplest kind of naming, but it's not really very powerful. See, what I'd really like to name-- remember, we're talking about general methods-- I'd like to name, oh, the general idea that, for example, I could multiply 5 by 5, or 6 by 6, or 1,001 by 1,001, 1,001.7 by 1,001.7. I'd like to be able to name the general idea of multiplying something by itself.

Well, you know what that is. That's called squaring. And the way I can do that in Lisp is I can say, define to square something x, multiply x by itself. And then having done that, I could say to Lisp, for example, what's the square of 10? And Lisp will say 100.

So now let's actually look at that a little more closely. Right, there's the definition of square. To square something, multiply it by itself. You see this x here. That x is kind of a pronoun, which is the something that I'm going to square. And what I do with it is I multiply x, I multiply it by itself.

OK. So there's the notation for defining a procedure. Actually, this is a little bit confusing, because this is sort of how I might use square. And I say square root of x or square root of 10, but it's not making it very clear that I'm actually naming something. So let me write this definition in another way that makes it a little bit more clear that I'm naming something. I'll say, "define" square to be lambda of x times xx.

Here, I'm naming something square, just like over here, I'm naming something A. The thing that I'm naming square-- here, the thing I named A was the value of this combination. Here, the thing that I'm naming square is this thing that begins with lambda, and lambda is Lisp's way of saying make a procedure.

Let's look at that more closely on the slide. The way I read that definition is to say, I define square to be make a procedure-- that's what the lambda is-- make a procedure with an argument named x. And what it does is return the results of multiplying x by itself. Now, in general, we're going to be using this top form of defining, just because it's a little bit more convenient. But don't lose sight of the fact that it's really this.

In fact, as far as the Lisp interpreter's concerned, there's no difference between typing this to it and typing this to it. And there's a word for that, sort of syntactic sugar. What syntactic sugar means, it's having somewhat more convenient surface forms for typing something.

So this is just really syntactic sugar for this underlying Greek thing with the lambda. And the reason you should remember that is don't forget that, when I write something like this, I'm really naming something. I'm naming something square, and the something that I'm naming square is a procedure that's getting constructed.

Well, let's look at that on the computer, too. So I'll come and I'll say, define square of x to be times xx. Now I'll tell Lisp that. It says "square." See, I've named something "square." Now, having done that, I can ask Lisp for, what's the square of 1,001? Or in general, I could say, what's the square of the sum of 5 and 7? The square of 12's 144.

Or I can use square itself as an element in some combination. I can say, what's the sum of the square of 3 and the square of 4? 9 and 16 is 25. Or I can use square as an element in some much more complicated thing. I can say, what's the square of, the sqare of, the square of 1,001?

And there's the square of the square of the square of 1,001. Or I can say to Lisp, what is square itself? What's the value of that? And Lisp returns some conventional way of telling me that that's a procedure. It says, "compound procedure square." Remember, the value of square is this procedure, and the thing with the stars and the brackets are just Lisp's conventional way of describing that.

Let's look at two more examples of defining. Here are two more procedures. I can define the average of x and y to be the sum of x and y divided by 2. Or having had average and mean square, having had average and square, I can use that to talk about the mean square of something, which is the average of the square of x and the square of y.

So for example, having done that, I could say, what's the mean square of 2 and 3? And I should get the average of 4 and 9, which is 6.5. The key thing here is that, having defined square, I can use it as if it were primitive. So if we look here on the slide, if I look at mean square, the person defining mean square doesn't have to know, at this point, whether square was something built into the language or whether it was a procedure that was defined.

And that's a key thing in Lisp, that you do not make arbitrary distinctions between things that happen to be primitive in the language and things that happen to be built in. A person using that shouldn't even have to know. So the things you construct get used with all the power and flexibility as if they were primitives. In fact, you can drive that home by looking on the computer one more time.

We talked about plus. And in fact, if I come here on the computer screen and say, what is the value of plus? Notice what Lisp types out. On the bottom there, it typed out, "compound procedure plus." Because, in this system, it turns out that the addition operator is itself a compound procedure. And if I didn't just type that in, you'd never know that, and it wouldn't make any difference anyway. We don't care. It's below the level of the abstraction that we're dealing with.

So the key thing is you cannot tell, should not be able to tell, in general, the difference between things that are built in and things that are compound. Why is that? Because the things that are compound have an abstraction wrapper wrapped around them. We've seen almost all the elements of Lisp now. There's only one more we have to look at, and that is how to make a case analysis.

Let me show you what I mean. We might want to think about the mathematical definition of the absolute value functions. I might say the absolute value of x is the function which has the property that it's negative of x. For x less than 0, it's 0 for x equal to 0. And it's x for x greater than 0. And Lisp has a way of making case analyses.

Let me define for you absolute value. Say define the absolute value of x is conditional. This means case analysis, COND. If x is less than 0, the answer is negate x. What I've written here is a clause. This whole thing is a conditional clause, and it has two parts. This part here is a predicate or a condition.

That's a condition. And the condition is expressed by something called a predicate, and a predicate in Lisp is some sort of thing that returns either true or false. And you see Lisp has a primitive procedure, less-than, that tests whether something is true or false.

And the other part of a clause is an action or a thing to do, in the case where that's true. And here, what I'm doing is negating x. The negation operator, the minus sign in Lisp is a little bit funny. If there's two or more arguments, if there's two arguments it subtracts the second one from the first, and we saw that. And if there's one argument, it negates it. So this corresponds to that.

And then there's another COND clause. It says, in the case where x is equal to 0, the answer is 0. And in the case where x is greater than 0, the answer is x. Close that clause. Close the COND. Close the definition. And there's the definition of absolute value. And you see it's the case analysis that looks very much like the case analysis you use in mathematics.

There's a somewhat different way of writing a restricted case analysis. Often, you have a case analysis where you only have one case, where you test something, and then depending on whether it's true or false, you do something. And here's another definition of absolute value which looks almost the same, which says, if x is less than 0, the result is negate x. Otherwise, the answer is x. And we'll be using "if" a lot.

But again, the thing to remember is that this form of absolute value that you're looking at here, and then this one over here that I wrote on the board, are essentially the same. And "if" and COND are-- well, whichever way you like it. You can think of COND as syntactic sugar for "if," or you can think of "if" as syntactic sugar for COND, and it doesn't make any difference. The person implementing a Lisp system will pick one and implement the other in terms of that. And it doesn't matter which one you pick.

Why don't we break now, and then take some questions. How come sometimes when I write define, I put an open paren here and say, define open paren something or other, and sometimes when I write this, I don't put an open paren? The answer is, this particular form of "define," where you say define some expression, is this very special thing for defining procedures. But again, what it really means is I'm defining this symbol, square, to be that.

So the way you should think about it is what "define" does is you write "define," and the second thing you write is the symbol here-- no open paren-- the symbol you're defining and

what you're defining it to be. That's like here and like here. That's sort of the basic way you use "define." And then, there's this special syntactic trick which allows you to define procedures that look like this. So the difference is, it's whether or not you're defining a procedure.

[MUSIC PLAYING]

Well, believe it or not, you actually now know enough Lisp to write essentially any numerical procedure that you'd write in a language like FORTRAN or Basic or whatever, or, essentially, any other language. And you're probably saying, that's not believable, because you know that these languages have things like "for statements," and "do until while" or something.

But we don't really need any of that. In fact, we're not going to use any of that in this course. Let me show you. Again, looking back at square root, let's go back to this square root algorithm of Heron of Alexandria. Remember what that said. It said, to find an approximation to the square root of X, you make a guess, you improve that guess by averaging the guess and X over the guess. You keep improving that until the guess is good enough. I already alluded to the idea. The idea is that, if the initial guess that you took was actually equal to the square root of X, then G here would be equal to X/G.

So if you hit the square root, averaging them wouldn't change it. If the G that you picked was larger than the square root of X, then X/G will be smaller than the square root of X, so that when you average G and X/G, you get something in between. So if you pick a G that's too small, your answer will be too large. If you pick a G that's too large, if your G is larger than the square root of X and X/G will be smaller than the square root of X.

So averaging always gives you something in between. And then, it's not quite trivial, but it's possible to show that, in fact, if G misses the square root of X by a little bit, the average of G and X/G will actually keep getting closer to the square root of X. So if you keep doing this enough, you'll eventually get as close as you want.

And then there's another fact, that you can always start out this process by using 1 as an initial guess. And it'll always converge to the square root of X. So that's this method of successive averaging due to Heron of Alexandria. Let's write it in Lisp.

Well, the central idea is, what does it mean to try a guess for the square root of X? Let's write that. So we'll say, define to try a guess for the square root of X, what do we do? We'll say, if the guess is good enough to be a guess for the square root of X, then, as an answer, we'll take the guess. Otherwise, we will try the improved guess. We'll improve that guess for the square root of X, and we'll try that as a guess for the square root of X. Close the "try." Close the "if." Close the "define." So that's how we try a guess.

And then, the next part of the process said, in order to compute square roots, we'll say, define to compute the square root of X, we will try 1 as a guess for the square root of X. Well, we have to define a couple more things. We have to say, how is a guess good enough? And how do we improve a guess? So let's look at that.

The algorithm to improve a guess for the square root of X, we average-- that was the algorithm-- we average the guess with the quotient of dividing X by the guess. That's how we improve a guess. And to tell whether a guess is good enough, well, we have to decide something. This is supposed to be a guess for the square root of X, so one possible thing you can do is say, when you take that guess and square it, do you get something very close to X? So one way to say that is to say, I square the guess, subtract X from that, and see if the absolute value of that whole thing is less than some small number, which depends on my purposes.

So there's a complete procedure for how to compute the square root of X. Let's look at the structure of that a little bit. I have the whole thing. I have the notion of how to compute a square root. That's some kind of module. That's some kind of black box. It's defined in terms of how to try a guess for the square root of X.

"Try" is defined in terms of, well, telling whether something is good enough and telling how to improve something. So good enough. "Try" is defined in terms of "good enough" and "improve." And let's see what else I fill in. Well, I'll go down this tree. "Good enough" was defined in terms of absolute value, and square. And improve was defined in terms of something called averaging and then some other primitive operator.

Square root's defined in terms of "try." "Try" is defined in terms of "good enough" and "improve," but also "try" itself. So "try" is also defined in terms of how to try itself. Well, that may give you some problems. Your high school geometry teacher probably told you that it's naughty to try and define things in terms of themselves, because it doesn't make sense. But that's false.

Sometimes it makes perfect sense to define things in terms of themselves. And this is the case. And we can look at that. We could write down what this means, and say, suppose I asked Lisp what the square root of 2 is. What's the square root of 2 mean? Well, that means I try 1 as a guess for the square root of 2.

Now I look. I say, gee, is 1 a good enough guess for the square root of 2? And that depends on the test that "good enough" does. And in this case, "good enough" will say, no, 1 is not a good enough guess for the square root of 2. So that will reduce to saying, I have to try an improved-- improve 1 as a guess for the square root of 2, and try that as a guess for the square root of 2. Improving 1 as a guess for the square root of 2 means I average 1 and 2 divided by 1. So this is going to be average. This piece here will be the average of 1 and the quotient of 2 by 1. That's this piece here.

And this is 1.5. So this square root of 2 reduces to trying 1 for the square root of 2, which reduces to trying 1.5 as a guess for the square root of 2. So that makes sense. Let's look at the rest of the process. If I try 1.5, that reduces. 1.5 turns out to be not good enough as a guess for the square root of 2. So that reduces to trying the average of 1.5 and 2 divided by 1.5 as a guess for the square root of 2.

That average turns out to be 1.333. So this whole thing reduces to trying 1.333 as a guess for the square root of 2. And then so on. That reduces to another called a "good enough," 1.4 something or other. And then it keeps going until the process finally stops with something that "good enough" thinks is good enough, which, in this case, is 1.4142 something or other.

So the process makes perfect sense. This, by the way, is called a recursive definition. And the ability to make recursive definitions is a source of incredible power. And as you can already see I've hinted at, it's the thing that effectively allows you to do these infinite computations that go on until something is true, without having any other constricts other than the ability to call a procedure.

Well, let's see, there's one more thing. Let me show you a variant of this definition of square root here on the slide. Here's sort of the same thing. What I've done here is packaged the definitions of "improve" and "good enough" and "try" inside "square root." So, in effect, what I've done is I've built a square root box. So I've built a box that's the square root procedure that someone can use. They might put in 36 and get out 6. And then, packaged inside this box are the definitions of "try" and "good enough" and "improve."

So they're hidden inside this box. And the reason for doing that is that, if someone's using this square root, if George is using this square root, George probably doesn't care very much that, when I implemented square root, I had things inside there called "try" and "good enough" and "improve." And in fact, Harry might have a cube root procedure that has "try" and "good enough" and "improve." And in order to not get the whole system confused, it'd be good for Harry to package his internal procedures inside his cube root procedure.

Well, this is called block structure, this particular way of packaging internals inside of a definition. And let's go back and look at the slide again. The way to read this kind of procedure is to say, to define "square root," well, inside that definition, I'll have the definition of an "improve" and the definition of "good enough" and the definition of "try." And then, subject to those definitions, the way I do square root is to try 1.

And notice here, I don't have to say 1 as a guess for the square root of X, because since it's all inside the square root, it sort of has this X known.

Let me summarize. We started out with the idea that what we're going to be doing is expressing imperative knowledge. And in fact, here's a slide that summarizes the way we looked at Lisp. We started out by looking at some primitive elements in addition and multiplication, some predicates for testing whether something is less-than or something's equal.

And in fact, we saw really sneakily in the system we're actually using, these aren't actually primitives, but it doesn't matter. What matters is we're going to use them as if they're primitives. We're not going to look inside. We also have some primitive data and some numbers. We saw some means of composition, means of combination, the basic one being composing functions and building combinations with operators and operands.

And there were some other things, like COND and "if" and "define." But the main thing about "define," in particular, was that it was the means of abstraction. It was the way that we name things. You can also see from this slide not only where we've been, but holes we have to fill in. At some point, we'll have to talk about how you combine primitive data to get compound data, and how you abstract data so you can use large globs of data as if they were primitive. So that's where we're going.

But before we do that, for the next couple of lectures we're going to be talking about, first of all, how it is that you make a link between these procedures we write and the processes

that happen in the machine. And then, how it is that you start using the power of Lisp to talk not only about these individual little computations, but about general conventional methods of doing things.

OK, are there any questions?

AUDIENCE: Yes. If we defined A using parentheses instead of as we did, what would be the difference?

PROFESSOR: If I wrote this, if I wrote that, what I would be doing is defining a procedure named A. In this case, a procedure of no arguments, which, when I ran it, would give me back 5 times 5.

AUDIENCE: Right. I mean, you come up with the same thing, except for you really got a different--

PROFESSOR: Right. And the difference would be, in the old one-- Let me be a little bit clearer here. Let's call this A, like here. And pretend here, just for contrast, I wrote, define D to be the product of 5 and 5. And the difference between those, let's think about interactions with the Lisp interpreter. I could type in A and Lisp would return 25. I could type in D, if I just typed in D, Lisp would return compound procedure D, because that's what it is. It's a procedure.

I could run D. I could say, what's the value of running D? Here is a combination with no operands. I see there are no operands. I didn't put any after D. And it would say, oh, that's 25. Or I could say, just for completeness, if I typed in, what's the value of running A? I get an error. The error would be the same one as over there. It'd be the error would say, sorry, 25, which is the value of A, is not an operator that I can apply to something.