PROFESSOR: Recursive data play a key role in programming, so let's take a mathematical look at what goes on. So the basic idea of recursive data is roughly that you're going to define a class of objects in terms of the simpler versions of the same object.

With a little more precision, the idea is that you [? can ?] build up a recursive data type by starting with some stuff that you understand that's not recursive and you give me some base objects that I can begin with and declare that they belong to this recursive datum, and then you give me some rules called constructor rules which enable me to build new objects of the recursive type by applying these constructors to objects that I've already built up.

There's nothing circular about it because I'm always building up new stuff from stuff I already have. Let's look at an example. I'm going to define a set E that's a subset of the integers, and I'm going to give you a recursive definition of E. The base case is that I'm going to tell you that 0 is an E and I'm going to give you two constructors.

The first one says that if you have an n that's an E, you can add 2 to it and get a new element in E, providing that n is not negative. The second constructor is that if you have an n that's an E, you can negate it. You can take minus n, providing that n is positive, and those are the two constructor rules.

Well, let's look at what goes on here. What is this telling us? Well, let's just use the first constructor rule and use it repeatedly. I can start off with 0. That's the base case, and then I can apply the constructor to add 2 to it.

Then I can apply the constructor again to add 2 to 0 plus 2, and then I can apply the third time to get add 2 to 0 plus 2 plus 2. And it's clear what I'm getting is 0, 2, 4, 6, and so on, and I'm going to get all of the non-negative even numbers in that way.

Now, I can apply to these, the positive numbers, I can apply the negation constructor. So I can get minus 2, minus 4, minus 6, and it becomes apparent then that I can get all of the even numbers.

So we just figured out that E contains the even numbers. Is there anything else in E? And the answer is no, and the reason is that an implicit part of the understanding of a definition like this is that the only way that things can get into E is by being a base case or by being constructed from previously constructed elements by applying the constructor rules. In other words, there's an implicit clause here that says that's all.

That implicit clause is called the extremal clause. And it's taken for granted and rarely mentioned explicitly as part of a recursive definition, but it's always to be understood. So what we can conclude from this is that E is exactly the even integers because there's nothing else there except those ones that were built up in the way indicated.

So let's look at a slightly more interesting example now. I want to define the set of strings that consists only of left and right parentheses such that the left and right parentheses match up. Well, writing parentheses on the slide turns out to be confusing with parentheses that are actually used to delimit things, so I'm going to replace parentheses by brackets in blue-- a right bracket and a left bracket.

This notation here, by the way, stands for the set of finite strings of right and left brackets. It's a general notation. If you have some collection of objects which you think of as letters and you write an asterisk as a superscript, that means the finite strings of those letters. So these are the finite strings of right and left brackets, and I want to give a recursive definition of a set M which I plan will be precisely those strings where the left and right brackets match up appropriately.

The way to think about matching up is take let's say a standard arithmetic expression involving plus and times and so on, and make sure that it's fully parenthesized. So whenever you add two things, there's parentheses around that, and whenever you multiply two things, there's parentheses around that, meaning brackets.

Then if you erased everything but the brackets, what you'd be left with would be a set of matched brackets. Actually, it would be a set of matched brackets or you could have several of them next to each other. Those would still could be considered matched, so that's the way our definition is going to behave.

Let's give it. So the base case is about the simplest it could be. I'm going to start with the empty string. An empty string is this thing that acts like a zero under putting strings next to each other, or the concatenation operation. If you concatenate the empty string with any string, it doesn't change the string. And by definition, then, the empty string is a string with no characters, has length zero, and it acts as an identity under putting strings next to each other.

There's going to be one constructor in M that's slightly ingenious. There's other maybe simpler or more natural ways to make up constructors that would define M, but this one is particularly nice because I can get away with just one, and it has some nice properties that we'll explore later.

So here's the rule-- if I've built up two strings s and t of matched brackets that are in M, then I can build a new one by putting matched brackets around s and concatenating it with t-- that is, if s and t are strings of brackets in M, then if I start with a left bracket followed by the brackets in s followed by a right bracket followed by the brackets in t, that new string is yet another element that I've built up in M.

Let's practice this to see how it works. So there's the constructor again. Well, how do I get started? To start, all I have is the base case. s and t have to both be the empty string because that's the only thing available to apply the constructor to. So if I do that, basically the s and the t disappear in this constructor expression and all I'm left with is a matching left and right bracket.

But now I've got a matching left and right bracket, so I can use that to apply the constructor to, so I could let s be the matching brackets and t still be the empty string. Now when I plug into the constructor, the t still disappears, but I find brackets within brackets, and that's another string that I've built up in M.

Now, being methodical, I could let s be empty and t be the brackets. And if I do that, then the s goes away and the t becomes the matched pair of brackets and I wind up with a matched pair next to a matched pair.

Then, of course, I could let both of them be the matched brackets, and then I get a nested pair next to a matched pair. And now that I've got also going back to the very beginning the next most complicated string that I had was the nested pair of brackets, I could let s be that and t be empty, and then I would get brackets nested to depth three, and so on. That's the idea.

Now, it may or may not be clear that you get exactly the strings of matched brackets in this way. That's taken up further in the notes and in some problems, but we're just trying to understand how this definition works and take it for granted that, in fact, it's right.

Let's use that definition to prove some things about M, but I want to prove the things based on the definition of M not assuming that it works as intended. So I'm going to claim based on the definition that it's impossible to find a string in M that starts with a right bracket. Now, of course, since we're assuming M is the right definition of matched brackets, it's clear that a string that starts with a right bracket already has nothing to match-- no left bracket matching it-- so it shouldn't be in there, but let's just make sure that the definition behaves in the way that we intend or it might be wrong.

So how do I prove that no string in M starts with a right bracket? Well, let's look at the definition. The base case doesn't have any brackets at all, so it certainly doesn't start with a right bracket. And looking at the constructor rule, all the strings that you can construct start with a left bracket. And so we're really appealing to the implicit that's all clause, the extremal clause that says that since the only way to get things in M is by applying the constructor, you're not going to be able to get anything that starts with a right bracket.

One more example of a recursively-defined data type that's interesting, and we'll be doing some lovely class problems with, is the class that I call F18 functions. These are the functions from a first term of calculus, like as you study in 18.01, functions of a single real variable, and here's a recursive definition that I think covers all of the functions that are considered in 18.01.

I'm going to start off with the identity function and any constant function and the function sine of x, and declare that those are the base cases. Those are the functions in F18.

Then here are the constructor rules. If I have two functions that are in F18, I can add and multiply them or take 2 to the f where f is in there, and those will all also be functions in F18. So I can start building up a bunch of interesting stuff like polynomials and exponentials. In addition, if I have a function that's in F18, I can take it's inverse-- at least, insofar as the inverse is defined in the function-- and I can also compose two functions that are in F18 to get another one.

Let's look at how this definition works. I claim that, in fact, the function minus x is in F18. How do I build up minus x from the rules? Well, minus 1 is a constant function, so I have that. And x is just the identity function, and I can multiply two functions that I have. So if I multiply minus 1 times x, guess what? I got minus x, so I've just figured out that that function is in F18.

What about the square root of x? Well, if I multiply the identity by itself, I get the function x squared. And then if I take its inverse, that's square root of x.

Well, I gave you sine x, but not cosine x or any other trig functions. Why not? Well, I want them all, but I can get them by the rules already. So how do you get cosine x? Well, cosine x is just sine of x plus pi.

Well, why is that in there? Pi is a constant, x is the identity. So the sum is a function that's in F18. And then if I compose that function with sine, I get sine of x plus pi, which is cosine x. So cosine x is there.

Now, this was actually pointed out to me by students, this simple way of getting cosine x. The original way that I thought, and I was using that square root operation where I was going to use the identity that cosine squared plus sine squared is equal to 1. So if I take 1 minus sine squared and then take the square root, that's another way to get cosine x, the point being that there's a lot of ways to derive the same function as being in F18 built up from the operations applied to other functions.

What about log of x? Let's just close with that. How do I get log of x? Well, log of x is the inverse of e of the x. How do I get e to the x? Well, e to the x is what you get by taking 2 to the log to the base 2 of e, which is e, and then raising that to the power x.

So if I take log e, which is a constant log to the base 2 of e, which is a constant, I multiply it by x the identity function and I take 2 to that power, I'm composing, in other words, x log x with the constructor 2 to the F, then I wind up with the function e to the x. And when I take its inverse, I get log of x, as claimed.