PROFESSOR: Now, the standard thing you do with a recursive data type in programming is you define recursive procedures on them, so let's look at how that works.

I'm going to define a function f on a recursively defined data type R, and the way I'm going to do it is I'm going to define f of b explicitly in terms of b and operations that are already understood for all of the base cases of b in R. And then I'm going to define f of a constructor applied to x in terms of x and f of x. And if I keep to that structure, that gives me a recursive definition of the function f on the recursively defined data set R.

Let's look at an example to make this recipe explicit and clearer. Let's think about a recursive function on a set of matched brackets. This is a somewhat interesting one. Let's define the depth of a string as follows, and the idea is it's how deeply nested are the successive pairs of left and right brackets.

Well, the depth of the empty string is 0. You got to start somewhere, and it's got no brackets, we'll call it depth 0.

Now, what about the depth of the constructor putting brackets around s and then following it by t? Well, putting brackets around s gives you a string that's 1 deeper than s is, and then you follow it by t, and it's as deep as t is. So the result is that the depth of the constructor is a string which is a number which is equal to 1 plus the depth of s and the depth of t, whichever is larger. The max of 1 plus depth of s and depth of t, and that's our recursive definition of depth.

Let's look at maybe another even more familiar example of recursive definition. Let's define the nth power of an integer or real number k. The zeroth power k is defined to be 1, and the n plus first power of k is defined to be k times the nth power of k, and this would be an executable definition of the exponentiation function in a lot of programming languages. And my point here is that this familiar definition, recursive definition on a nonnegative integer n, is in fact a structural induction using the fact that the nonnegative integers can be defined recursively as follows.

0 is a nonnegative integer, and if n is a nonnegative integer, then n plus 1 is a nonnegative integer. So to summarize, the recipe for a recursive function definition is you define f going from the recursive data type to values-- whatever kind of values you want to assign to these recursive data-- f of b is defined directly for the base case b, of base cases b, and f of the

constructor of x is defined using f of x and x.

Now, once you've gotten a function defined recursively, you can start proving things about it by structural induction or by induction on its own definition, its own recursive definition. So let's look at an example of that.

I want to prove the following property of the depth of strings in M, namely that if I look at the length of a string r plus 2-- so the vertical bars around r mean the number of brackets in the string r plus 2 is less than or equal to 2 to the power of depth plus 1-- twice the 2 to of the depth of the string. And I want to prove that this holds for all strings r of matched brackets, and I'm going to prove it by structural induction. And just as a walk-through, here's how the proof is going to go.

Let's suppose that r is the base case. Is it the case that this inequality holds for the empty string? Well, the length of r is 0, so length of r plus 2 is 0 plus 2, or 2, which is the same as 2 to the 0 plus 1, which is in fact equal to 2 to the depth of the empty string plus 1. So this inequality actually holds as an equality in the base case, and we're good there.

What we next need to show is that this inequality holds in the constructor case. So we're looking at an arbitrary string r that's built out of s and t, and r is left bracket, s right bracket, t, and I want to show that r satisfies this inequality.

Well, by induction hypothesis, I can assume that s and t satisfy the inequality. So I have that the length of s plus 2 is at most 2 to the depth of s plus 1 and the length of t plus 2 is at most 2 to the depth of t plus 1, and let's just walk through the proof.

You can slow this down and replay it if need be, so I'm just going to go through it quickly. The length of r plus 2-- r is, after all, brackets s, t, so it's simply the length of that string plus 2 by the definition of r. The length of brackets s, t is the length of t plus the length of s plus the 2 for the 2 brackets that we've added, and so we're plugging that into the previous term and getting that plus 2.

Then just rearrange the terms. It's the same as the size of s plus 2 plus the size of t plus 2. And I arranged it that way because by induction hypothesis, I know that the size of s plus 2 is less than or equal to 2 to the depth of s plus 1, and likewise for t.

Now I just play a nice trick to get these 2 exponents to look alike. I say that the depth of s is less than or equal to the max of the depth of s and the depth of t, and likewise for the depth of t. So in both of those terms here, I can replace the exponent or replace the depth of s by the max of depth s and t, and likewise here.

Now I've got the same term twice, so I can say that it's simply twice the max depth. And of course, that is equal to, by definition of the depth of r, twice 2 to the depth of r, which is of course 2 to the depth of r plus 1. And I have by more or less automatically plugging into the definitions and a structural induction, I've proved that this inequality holds for the recursively defined depth function, and we're done.

Let's look at one more familiar example. I want to give [? the ?] recursive definition of the positive powers of 2. So the base case is the 2 is a positive power of 2, and the constructor is just one constructor I'm going to use-- that if x and y are positive powers of 2, then their product is a positive power of 2. So let's look at some examples.

I can start with 2 and then the only thing I can do as a constructor is multiply 2 by 2 to get 4. Once I got 4, I can do 4 times 2 to get 8, and I can do 4 times 4 to get 16, and I can do 4 times 8 to get 32, and all of these are positive powers of 2.

Now let's define the log to the base 2 of a positive power of 2 recursively. Well, the log to the base 2 of 2 is 1. I'd have to define log to the base 2 in the base case, and that's easy to do.

What about in the constructor case? Well, the log to the base 2 of x, y is equal to the log to the base 2 of x plus the log to the base 2 of y for all the x, y's that are positive powers of 2, and so I have defined a log of the constructor x, y in terms of the function log of x and the function log applied to y. It conforms to the standard definition of a recursive function on a recursively defined data type, [INAUDIBLE] positive powers of 2.

Now, this looks OK. Well, let's just check it out. So log of 4 is log of 2 times 2, which is by the definition of the log of 2 plus the log of 2, which is 1 plus 1, which is equal to 2, and guess what? That's correct.

The log of 8-- well, 8 is 2 times 4, so by the recursive definition, that's the log of 2 plus the log of 4, which we previously figured out log of 4 was 2, so we get 3 and the answer comes out right. Now, remember, you're not supposed to in this reasoning use the properties that you know that log to the base 2 has because we're defining this function which we're calling log to the base 2 and implicitly claiming that it's right.

But in order to prove that it's right, we need to be using just see the structural definition of log to the base 2 to prove its properties. So that was what I was illustrating with this reasoning, that just plugging in the constructor case of log of x, y is log of x plus log of y, I can get these numbers out. So the point of this is just to make the following definition look reasonable.

I'm going to define a new function which I'm going to call the log e function, and it's another function on the positive powers of 2, and here's the definition of the log e function. The log e of 2 is going to be 1, just as the log is, but the log e of x, y is going to be x plus the log e of y for all x, y and positive powers of 2. Well, let's try that definition.

Log e of 4 is log e of 2 times 2, which according to the recursive definition is 2 plus the log e of 2, which is 1-- namely, it's 3.

Log e of 8-- well, 8 is 2 times 4, so the log e of 8 is 2 plus the log e of 4. We just figured out that the log e of 4 was 3, so it's 2 plus 3 is 5. Log e of 8 is 5.

And finally, log e of 16-- well, 16 is 8 times 2, so the log e of 8 times 2 is 8 plus the log e of 2, which we know is 1. It's 9. So we've just figured out the log e of 16 is 9, but now comes the problem.

16, of course, is not only 8 times 2, but it's 2 times 8, and so the log e of 2 times 8 is 2 plus the log e of 8. Log e of 8 we previously figured out was 5, so the log e of 16 is 7, and now I have an inconsistency. I have used this recursive definition of log e to conclude that the log e of 16 is both 9 and 7, and we got a problem. It's not a good definition of a function.

The problem is a simple one called ambiguity. There's more than one way to construct the elements of PP2, of positive powers of 2, from the constructor x times y. 16 was 8 times 2, but it's also 2 times 8, and of course, it's also 4 times 4. And depending on which constructor you use to construct 16, you're going to get out different values assigned to the log e function.

So when you have an ambiguously defined recursive data structure-- for example F18 is very ambiguous-- then defining recursive functions on that definition is not going to work well, and you have to very carefully prove that a recursive definition actually works in a single value. So for example, log to the base 2 does work, but that would require proof it doesn't follow on general principles [INAUDIBLE] you define a recursive function on an ambiguous data type.

On the other hand, the reason why we chose that somewhat unexpected single constructor for the balanced strings M, the balanced bracket strings, was that it turns out to be unambiguous. And so that definition of depth is a good definition, as is any definition based on the recursive definition of the set M.

So the general problem we have to watch out for--

[AUDIO OUT]

Constructor created the datum e. If there's more than one way to construct e, then you're not quite sure which case to use to define the function f, and that's why this issue of whether or not the data structure's ambiguous is critical to getting good definitions of recursive functions.