[1] Recursive data definition refers to a way of defining infinite bunches by means of axioms called construction and induction. The simplest and best known example is the bunch of natural numbers, nat, so we start there. The elements of nat [2] can be constructed by starting with 0 and repeatedly adding 1. So the [3] first construction axiom says that 0 is in nat. And the [4] other construction axiom says that if you have an element of nat, and you add 1 to it, you again have an element of nat. With this pair of axioms we can prove that all the natural numbers are in nat, like [5] this. Starting with true, we use the [6] first axiom to see that 0 is there. Then we [7] add 1 to each side. We can simplify 0 plus 1 to 1 by an axiom of arithmetic, and [8] nat plus 1 is included in nat by the second construction axiom. So now we have 1 in nat. Again we [9] add 1 to each side, and [10] again we use an axiom of arithmetic to simplify the left side, and the second construction axiom to find that 2 is in nat. [11] and so on we can find out that [12] all the natural numbers are in nat. So [13] is nat equal to 0, 1, 2, and so on? Forget for a minute that the name nat sounds like it's the natural numbers, and just think of nat as unknown. We're wondering what nat might be. We already proved that 0 and 1 and 2 are in it, and we see the pattern, so we can prove that 3, 4, 5, and so on are in it, but [14] maybe other numbers are in there too. Maybe all the integers. 0 is an integer, and for each integer, adding 1 makes another integer. So according to the construction axioms, nat could be the integers. [15] Or the rationals. [16] Or the reals. Or [17] 0, a half, 1, 1 and a half, 2, 2 and a half, and so on. 0 is there, and for each number there, the number 1 greater is also there. The construction axioms by themselves don't define nat. We need another axiom to say that there's nothing in nat other than 0, 1, 2, and so on. That's [18] the induction axiom. It says: if you have a bunch B such that 0 is in B and if you add 1 to an element of B you get an element of B, then nat is included in B. We've just seen several examples of such bunches, and there are infinitely many more. The construction axioms say that nat is one of them. And the induction axiom says that nat is the smallest such bunch. That's how it says that there aren't any elements other than 0, 1, 2, and so on, in nat. So now it's ok again to think of nat as the natural numbers.

Let me rewrite [19] the 2 construction axioms as 1 axiom, and then I rewrite the induction axiom to match it. These 2 axioms are equivalent to the 3 above. And I want to show you one more equivalent pair of axioms [20] using predicates instead of bunches, because I'm sure you'll recognize the induction axiom this way. It says if predicate P is true of 0, and assuming it is true of n you can prove it's true of n plus 1, then you have proved P is true of all elements of nat. It's saying there aren't any other elements in nat by saying that you just have to prove P of these elements and you've proved it of all elements. And construction is the same but with the main implication pointing the other way. In the textbook there's a proof that the bunch and predicate versions are equivalent. I want to show you a [21] whole lot of equivalent versions of induction. The [22] first one is the usual one. [23] This one is prettier, I think, because it looks more balanced. It says if you can go from any natural to the next, then you can go from 0 to any natural. And [24] this one is interesting because it doesn't mention 0. It's called course of values induction. In the antecedent, in order to prove P of n, you get to assume P of all previous values, not just one previous value. And the [25] last one is really interesting. It says, reading the right side first, if there's a natural with property P, then, moving to the left side, there's a natural n such that, all previous naturals don't have property P, and n does have property P. It says, if there's a natural number with some property, then there's a first natural with that property. Induction is equivalent to the concept of first. Did you know that?

Now I'd like to mention how the word induction is used in philosophy, because logic came from philosophy, and a lot of the philosophical terminology is still with us. Far too much, in my opinion. [26] In philosophy, induction means guessing the general case from some special cases. And they [27] contrast that with deduction, which means proving, using the rules of logic. In [28] mathematics, induction is an axiom, or sometimes it's presented as

a proof rule. And you use it in proofs, just like any other axiom. So mathematical induction is not the same as philosophical induction. In fact, it's part of philosophical deduction. Then there's [29] engineering induction, which means: if it works for n equals 1, 2, and 3, then that's good enough for me. I think engineers are being slandered, here. And there's [30] military induction, and [31] tax deduction, but [32] let's get back to recursive data definition, and the next example is int.

The easiest definition of int is just [33] nat and minus nat. But we could define int using [34] construction and induction. The constructors here are 0, adding 1, and subtracting 1. The induction says that of all bunches satisfying construction, int is the smallest. And there's [35] an equivalent predicate version. It says proving P of 0, and assuming P of i and proving it of i plus 1, and assuming P of i and proving it of i minus 1, is the same as proving P of all integers.

The [36] next example I'm calling pow, and it's the powers of 2. The easiest definition is [37] 2 to the nat, or we could define it as [38] those p in nat such that there is an m in nat such that p equals 2 to the m. But the point of this lecture is that we [39] could define it using construction and induction. The constructors are 1 and doubling. And there's the induction axiom, or if you prefer the predicate version, [40] here it is. I guess the point here is that induction is not just for the natural numbers. It's for anything that can be defined constructively. That includes the rational numbers, but it doesn't include the reals. You can make a list of the rational numbers, but you cannot make a list of the real numbers.

[41] Here are the nat construction and induction axioms again, and [42] here are 2 very similar axioms, called fixed-point construction and induction. If you compare [43] the two construction axioms, you see that the only difference is a colon in ordinary construction has become an equal sign in fixed-point construction. So fixed-point construction is stronger. And [44] in the induction axioms, the same colon becomes equals. But it's in an antecedent, so fixed-point induction is weaker than ordinary induction. This new pair of axioms, one stronger and one weaker, is exactly equivalent to the old pair. From ordinary construction and induction you can prove fixed-point construction and induction, and vice versa. Some people prefer the fixed-point version. The reason it's called fixed-point comes from function theory. [45] x is called a fixed-point of function f if [46] x equals f of x. Applying f to x gives back x. [47] And that looks like fixed-point construction, where nat is equal to some function of nat. The [48] fixed-point induction axiom says that of all the fixed-points of the constructor function, nat is the smallest, or least, so it's the least fixed-point. One place you might have seen this before is a [49] grammar. Maybe the grammars you've seen used colon colon equals instead of equals, and a vertical line instead of a comma, and maybe nothing instead of a semicolon for joining. If it was equating something to some function of itself, then it was a fixed-point construction axiom. For grammars, the induction axiom is usually just said in words like: there are no other ways of forming expressions. But it could be [50] written formally, like this. And you have to write it formally if you want to prove anything about all expressions. It's called structural induction.

[51] When we define something by construction and induction, sometimes it's not easy to see what's been defined. Recursive data construction is a procedure for finding out what's been defined. It usually works, but not always. [52] A fixed-point construction axiom equates some name we're defining to some expression involving that name. I wrote the fixed-point version here, but it doesn't matter which version I write because, as I said fixed-point and ordinary construction and induction are equivalent. The [53] first step in this procedure is to write a sequence of definitions, for name sub 0, name sub 1, name sub 2, and so on. Name sub 0 is null, because to start with we don't know anything about what's in the bunch being defined. Name sub 1 uses the constructor but putting name sub 0 in place of the name, and so on. We know exactly what name sub 0 is, it's null. So we know exactly what name sub 1 is. And so on. When we've got several of these definitions, maybe we can see a

pattern, so the [54] second step is to guess the general case. That's a philosophical induction. The [55] next step is to substitute infinity for n. Now we have a nonrecursive definition, and that might be the answer. I say might be, because this procedure doesn't always work. So [56] now we have to test this candidate and see if it really is a fixed-point. If it passes that test, then [57] we test to see if it's the least fixed-point. Here's how it works on the [58] pow example that we saw earlier. pow equals 1 comma 2 times pow. Suppose we have no idea what pow is, and we want to find out. The [59] first step is to construct. So we start with [60] pow 0 equals null. Now [61] pow 1 equals, and we use the construction axiom, but put pow 0 in place of pow. So that's [62] 1 comma 2 times null, and 2 times null is [63] null, and that's just [64] 1. [65] Pow 2 is 1 comma 2 times pow 1, which is [66] 1 comma 2 times 1, which is [67] 1, 2. [68] Pow 3 equals that, which is [69] 1 comma 2 times 1 and 2, which is [70] 1, 2, and 4. Maybe we're ready for the [71] guess. I am. I guess [72] pow n equals 2 to the powers 0 to n. It doesn't really matter if this guess is the correct generalization of what we've seen. We have to test it anyway to see if it fits the axioms. But first, we [73] substitute infinity for n, like [74] this, which is [75] 2 to the power nat. Now we [76] test. That means we [77] put our candidate in place of pow in the axiom, and then try to prove it. First [78] rewrite 1 and 2 as powers of 2. Now use the law of exponents to change [79] the product into a sum of exponents. And [80] exponentiation distributes over bunch union, so factor out the 2. And this [81] is implied by the fact that nat equals 0 comma nat plus 1, which is [82] the fixed-point construction axiom for nat. [83] Now we have to [84] test if our candidate is the least fixed-point. That means putting 2 to the nat in place of pow in the fixed-point induction axiom, and trying to prove it. I'll do it like [85] this, 2 to the nat colon B is implied by B equals 1 comma 2 times B, and try to fill in the lines in between. My first step is to rewrite the top line as [86] a universal quantification. 2 to the nat is in B means that for every n in nat, 2 to the n is in B. And now I can prove this universal quantification over nat by [87] nat induction. Well, you knew we were going to have to use it somewhere. I leave you to read [88] the rest of the steps in the textbook. There might be a more direct proof using the bunch form of induction instead of the predicate form. If you find it, please let me know. [89]

So that's recursive data construction. I'll just mention a couple of variations on the procedure. [90] We started with name 0 equal null. But actually, you can start with name 0 equal anything you want. I find null usually works best because we're trying to find the smallest fixed point, so it's usually good to start with the smallest bunch. But sometimes that doesn't work, and some other starting point does. The other variation [91] is, after you have name sub n, instead of putting infinity in place of n, take the limit of name sub n. Mostly that's the same as name sub infinity, but on rare occasions it isn't. I mention it, but I seldom do it because it's more work and it's not guaranteed to succeed any more than name sub infinity is.