[1] We've been talking about Binary Theory, and we've pretty well covered it, but there are a couple of advanced tricks I'd like to tell you about. The first one is called monotonicity and antimonotonicity. Some people call it [2] covariance and contravariance. In high school I learned to say [3] varies directly as and varies inversely as. [4] Or we could say nondecreasing and nonincreasing, [5] or even sorted, and sorted backwards. If we were talking about numbers instead of binary values, [6] here's what it means. Function f is monotonic means if x is less than or equal to y then f of x is less than or equal to f of y. And antimonotonic means turn one of the less than or equal to signs around. [7] Here's the picture. Monotonic means increasing or staying the same, but never decreasing. Antimonotonic means decreasing or staying the same, but never increasing. Fine for numbers, [8] but what about binary values? [9] Less than or equal to is the ordering on numbers, and the corresponding order for binary values is implication. [10] The word implies doesn't sound very much like an ordering, so we could say [11] x is falser than or equal to y. But the word "falser", if it's even a word, comes from an application area, and that's not the word that's used. The word that's used is [12] "stronger". There are lots of examples of the number ordering; [13] here are two examples. And we say one operand is smaller and the other is larger. For the binary values, there are very few examples, and we say one operand is stronger and the other is weaker. Now don't try to take any meaning at all from the words stronger and weaker. They don't make any sense. I would prefer to call the binary values bottom and top, and to say that bottom is lower than top. But the words "stronger" and "weaker" are standard, so we'll use them. [14] We've seen monotonic for numbers, so for binary values we just replace less than or equal to with stronger than or equal to. [15] For numbers, we say that as x gets larger, f x gets larger – or stays the same. So for binary values we say as x gets weaker, f of x gets weaker – or stays the same. Or I could say it the other way round – as x gets smaller or stronger, f of x gets smaller or stronger. The point is, that as x goes one way, f of x goes the same way. [16] And then antimonotonic is just the other way around. As x goes one way, f of x goes the other way. — [17] The negation operator is antimonotonic. If *a* gets stronger, not *a* gets weaker, and vice versa. [18] Conjunction is monotonic in both its operands. If one of the operands gets stronger, the whole conjunction gets stronger, or at least stays the same. [19] Same for disjunction. [20] Implication is antimonotonic in the antecedent, and monotonic in the consequent. [21] And the same for reverse implication, but the operands are reversed. [22] If-then-else-fi is monotonic in both the then-part and the else-part. The if-part is neither monotonic nor antimonotonic. What's all this good for? [23] It can save a lot of steps in a proof. Here's an example. [24] The first hint is the law of generalization, which says *a* implies *a* or b. Let me say *a* is stronger than or equal to *a* or b. We're going to replace the disjunction *a* or b with *a*. So we're replacing something with something stronger. I really mean stronger or equal. But let me just say stronger to save some words. So the [25] negation is getting weaker, because negation is antimonotonic. So the [26] conjunction is getting weaker. So the [27] whole expression is getting stronger. So that tells us that [28] we need an implication sign out front and we need it that way round. The first line is weaker than or equal to the second line. And the second line is the [29] law of noncontradiction, so it's equal to true. We want to prove the top line, so we want to show that it's equal to true. What we've shown is that the top line is weaker than or equal to true. But there's nothing weaker than true. True is the weakest. So the top line must be equal to true. In a proof, if you're driving towards true on the bottom line, the signs over on the left side don't all have to be equal signs. They can be any mixture of equals and is-implied-by signs. This *is* a proof of the top line.

The other advanced trick of Binary Theory that I want to tell you about is called [30] context. Here's a context rule. Now imagine that *a* is a long binary expression, and so is b. And on some line of a proof, somewhere in the line, we have the conjunction of *a* and b.

And in this [31] proof step we're changing *a* into some other expression, say c. The context rule says we can treat b as an axiom temporarily to help change *a* into c. Why? Well, b might be true, or it might be false. [32] If it happens to be true, then we made the right assumption. So that's all right. [33] But maybe b is false and we made the wrong assumption. But if b is false, then it doesn't matter what *a* was or what c is, both lines are false anyway, so they're still equal. So whenever you're working on a conjunct, [34] you can assume the other conjunct. [35] Either way round. [36] Here's the same example we did before. It has a conjunction in it. Let's change the [37] right conjunct, and so we get to assume the left conjunct. That means that in the right conjunct [38] we can replace *a* with true. That's what the context rule says. And the rest is easy because [39] true or anything is true, and the [40] negation of true is false, and [41] anything and false is false, and finally [42] not false is true. This proof is longer than the one we did before, but it was an easy proof. [43] These context rules aren't the only ones. [44] There are lots more. They all work for exactly the same sort of reason as the conjunction rules. You can find them all in the textbook. And that's all I want to say about Binary Theory.

[45] The next theory we need is Number Theory, but we don't need to spend much time on it because you've been using it since primary school. The main use of number theory is to reason about quantities of things. [46] Number expressions are pretty standard. But we need one more number expression that's not standard, namely [47] infinity. The reason we need that is that we'll be calculating execution times of programs. And for those executions that don't terminate, the execution time is infinity. I'm writing [48] multiplication with a little x the way mathematicians have done for centuries, and the way it appears on all calculators, not with the star that programmers use. The reason that programming languages use a star is because long ago card punching machines didn't have any multiplication symbol but they did have a star. Please don't leave out the multiplication symbol, because then it looks like a two character identifier, or maybe a function applied to an argument. I don't mind if [49] division is a slash or a horizontal line with operands above and below. [50] Exponentiation looks the way it does in all mathematics books. [51] An up-arrow means maximum. x up-arrow y is the maximum of x and y. [52] Down-arrow means minimum. There's also an [53] if-then-else-fi where the if-part is binary but the then-part and else-part are both numbers. [54] And there are binary expressions that have number subexpressions. The laws are listed at the back of the book. They should be pretty familiar, except for infinity. Have a look at them.

[55] The last of the 3 basic theories is character theory. There's nothing much to it. A character is written between double-quotes. So on the top line there's a [56] capital A character, then a [57] small *a* character, then a [58] blank space character. The [59] left double-quote character has to be written twice, and likewise the right double-quote character. We might have [60] successor and predecessor operators, and an [61] if-the-else-fi. And [62] ways to make binary expressions with character subexpressions. And that's it for the basic theories. We have binary data, number data, and character data. Now we need ways of making data structures. That's next lecture.