[talking head] Formal Methods of Software Engineering means the use of mathematics as an aid to writing programs. Before we can start applying mathematics to programming, we have to introduce the relevant mathematics. Now don't worry about this. I'm just going to introduce the math that we really are going to use, and nothing more. And you may have already learned most of this math in other courses. If you did, this is mostly review. But there will be a few things you didn't already know. And I want to make sure we're all using the same notations.

If you look at the last couple of pages of the textbook, you'll see all the notations used in the course, and I bet you already know most of them.

To start with, this lecture talks about Binary Theory, which is also called boolean algebra, or maybe basic logic, or some people say propositional calculus.

[1] The expressions of Binary Theory are called binary expressions. Some binary expressions are called theorems, and others are called antitheorems. [2] Binary expressions are used to represent anything that comes in two kinds. The theorems represent one kind, and the antitheorems represent the other kind. For example, [3] if you think that statements about the world come in two kinds, namely the true statements and the false statements, then you can use binary expressions to represent statements. You can use the theorems to represent the true statements, and the antitheorems to represent the false statements. I don't say that statements about the world really do come in just those two kinds, and it's no business of Binary Theory to say which statements are true and which are false. But if someone supplies us with statements that are labeled true and false, we can use Binary Theory to represent them. [4] Another application is digital circuits because the output of a digital circuit is either high voltage or low voltage. We can use the theorems to represent the circuits with high voltage output, and the antitheorems to represent the circuits with low voltage output. [5] Another application is human behavior, as viewed by the legal system. We can use the theorems to represent innocent behavior, and the antitheorems to represent guilty behavior. [6] Here are the two simplest binary expressions. If we're using them to represent statements, we might call the first one true and the other one false. If we're using them to represent circuits, we might call the first one power and the other one ground. And similarly we could choose words from other application areas. Or, to be independent of all application areas, we can call the first one top and the other one bottom. I tend to call them true and false. [7] The next notation is the NOT operator, or negation operator. It has one operand. Actually, there are four operators that have one operand, but two of them are degenerate, which means they don't make use of their operand, and one is just the identity operator. [8] Next are the operators that have two operands. Actually, there are sixteen operators that have two operands, but six of them are degenerate, leaving only ten good operators. And we're using only six of them. The [9] first one is pronounced x and y. It's a conjunction, and its operands are called conjuncts. The [10] next one is pronounced x or y. It's a disjunction, and its operands are called disjuncts. The [11] next one is pronounced x implies y. Or we could say x is stronger than or equal to y. It's an implication, and its left operand is called the antecedent, and its right operand is called the consequent. This [12] one is pronounced x is implied by y. Or, x is weaker than or equal to y. It's a reverse implication. Its left operand is the consequent, and its right operand is the antecedent. This [13] one is pronounced x equals y. It's an equation. It's not an assignment. Nothing is changing value here. It's an expression whose result is true or false. Its operands don't have any special names; we just say left side and right side. [14] The last one says x is unequal to y. [15] There are a lot of operators that have three operands, 256 of them, but we just want one of them. It's called conditional composition, or if-then-else-fi, and the operands are called the if-part, the then-part, and the else-part.

[16] There's a precedence table to say what the order of evaluation is; you can find it on the last page of the textbook. And if you want some other order of evaluation, that's what parentheses, round brackets, are for. [17] On that same table it says that some operators are

associative. For example, conjunction is associative. That means it doesn't matter which way we parenthesize, the result is the same. So we don't bother to write the parentheses. And we can write a longer chain of conjunctions without parentheses because all ways of parenthesizing give the same result. Same for disjunction. Even though equals and unequals are also associative, we don't use their associativity to save parentheses. That's because [18] they are continuing operators. That means if I write *x* equals *y* equals *z* I mean *x* equals *y* and *y* equals *z*. That's standard mathematical practice, but it's not the practice in most programming languages. Likewise the implications are continuing, so if I write x implies y implies z I mean x implies y and y implies z. And we can make longer chains of equations, longer chains of implications, and even a mixture of the two, and it means a conjunction of single equations and implications. You notice I didn't need parentheses to write a conjunction of equations, but I did need them to write a conjunction of implications. That's because, on the precedence table, EQUALS comes before AND, but IMPLIES comes after AND.

[19] The equation and implication operators come in two sizes. The big operators mean exactly the same thing as the little operators. All the same laws apply. The only difference between a little operator and a big operator is the precedence, so you can choose the size that saves you some parentheses. Too many parentheses cluttering up an expression can make it confusing. On the last line is an example of a mixture of big operators being used in a continuing way.

[20] Binary expression are evaluated according to these truth tables. The operands are on the top line, and the result is inside the table. [21] For negation, it says that NOT applied to true gives false, and not applied to false gives true. Actually, it says more than that. The symbol for true is representing any theorem, and the symbol for false is representing any antitheorem. So they are more than truth tables; they are theorem tables. It says that if you apply NOT to a theorem you get an antitheorem, and if you apply NOT to an antitheorem you get a theorem. [22] This row shows why conjunction is called AND. You see that the result is true just when both the left conjunct AND the right conjunct are true. [23] And on this row you see why disjunction is called OR. The result is true when either the left disjunct OR the right disjunct is true. It's false only when they're both false. If I could remember to call them top and bottom, which would be better because it's independent of application, then I should say that conjunction gives the minimum of its two operands, and disjunction gives the maximum of its two operands. Do you see that? [24] Implication is the ordering operator. It means "lower than or equal to". The first entry says top is lower than or equal to top, because they're equal. The next entry says that top isn't lower than or equal to bottom. Bottom *is* lower than or equal to top. And last, bottom is lower than or equal to itself. If I'm saying true and false, I could call this operator falser than or equal to. [25] On the next line we have reverse implication, which means higher than or equal to. Or if you prefer, truer than or equal to. [26] The equal operator is pretty obvious. [27] And so is the unequal operator; its result is true just when its operands are unequal. Some people like to call this operator exclusive or because its result is true when either the left operand or the right operand is true, but not both of them. [28] The if-then-else-fi is easy to remember because if the if-part is true, then the result is the same as the then-part. If the if-part is false, then the result is the same as the else-part. The if at the beginning and the fi at the end are like opening and closing brackets for this operator.

[29] The purpose of variables in an expression is to substitute something else in their place. Substituting something for a variable is called instantiation. When you substitute, you have to be a little careful about the precedence of operators. For example, [30] if we have the expression x and y, we can replace x by anything, so let's replace it by false. And we can replace y by anything, so let's replace it by false or true. So the result is supposed to be the conjunction of false on the left, with false or true on the right. But because of the precedence table, we have to put parentheses around false or true. We started with a conjunction, and

after we replace the operands, we should still have a conjunction. If we didn't add the parentheses, it would end up as a disjunction. [31] The other rule of substitution is that it has to be systematic. If the same variable occurs more than once, you have to replace it with the same thing at each occurrence. [32] Different variables can be replaced by different things, but they could also be replaced by the same thing if we want. So from x and y, we can get false and false. [33] Or, we can get true and false if we want.

Well, that's all there is to binary expressions. So now we have to apply binary theory to some application area. An application always supplies its own new binary expressions. [34] For example, it might supply these new binary expressions. The grass is green, the sky is green, there is life elsewhere in the universe, and intelligent messages are coming from space. I don't know what this application is about, but anyway, it gives us these new binary expressions. When we get to number theory, it will give us [35] these new binary expressions. And infinitely many others. The application also has to tell us which of the new expressions are theorems, and which are antitheorems. When you're classifying binary expressions as theorems and antitheorems, it's really important to be [36] consistent. That means you don't want to classify some binary expression as both a theorem and an antitheorem. You might like to be [37] complete also, which means you've classified every binary expression one way or the other. But you don't have to be complete. You do have to be consistent.

[38] Finding out if a binary expression is a theorem or antitheorem is called proving. And there are six rules for how to do that. The first rule is trivial. It's the axiom rule. Long ago, mathematicians thought that some things in mathematics were obviously true, and those things were called axioms. For example, it was obvious to them that parallel lines never meet. So that was an axiom. Later on, mathematicians discovered that there are interesting non-Euclidean geometries in which parallel lines *do* meet. Every single thing they thought was obviously true turned out to be just a choice. If you say it's true, you get one kind of mathematics. If you say it's false, that's another kind of mathematics. In my opinion, there's nothing true or false in mathematics. An application area may have true and false statements, and so you have to design your mathematics to apply to that application area. For example, we include [39] this expression in our mathematics to [40] represent the truth, in some applications, that when you put quantities together, the total quantity does not depend on the order in which you put them together. There are other applications where that's not true. But we're designing our mathematics for those applications where it is true, so [41] we make it an axiom. And according to the axiom rule, [42] it's a theorem. And we'll see that that makes it [43] equivalent to top. However, just like everyone else in the world, [44] I'll probably say that x plus y equals y plus x is true, even though that's not quite right; we just choose to use it to represent a truth in an application area. But I'm getting [45] way ahead of myself here; let's get back to binary theory.

[46] The only axiom of binary theory is top, and the only antiaxiom is bottom. For our silly example application area, [47] we have an axiom that says the grass is green, and an antiaxiom that says the sky is green. Now [48] here's another axiom. I have no idea if there's life elsewhere in the universe, so I won't make that an axiom or an antiaxiom. I'll just leave it unclassified until we find out if it's true or false. And I also have no idea if intelligent messages are coming from space, so I'll leave that unclassified too. But if intelligent messages are coming from space, then there must be life elsewhere, so we might want this implication to be an axiom.

[49] The second proof rule is the evaluation rule. It says: If all the subexpressions are classified, then use the theorem tables. If you know what the operands are, then the theorem tables tell you what the whole expression is.

[50] Next comes the completion rule. It says: if an expression contains unclassified subexpressions, and all ways of classifying them place it in the same class, then it is in that class. — If you don't know what the operands are, you might still be able to tell what the

whole expression is. For example, [51] there is life elsewhere in the universe or true. Now if there's life elsewhere, then that's true or true, which is true. If there isn't life elsewhere, then that's false or true, which is still true. So either way, it's true. So the completion rule says it's a theorem. [52] Here's another example. Either there's life elsewhere, or there isn't. If we put true in both places, we get true. If we put false in both places, we get true. So it's true. [53] And one more example. There's life elsewhere and there isn't. If we put true in both places, we get false. If we put false in both places, we get false. So it's false.

[54] Next we have the consistency rule. It says: if a classified binary expression contains binary subexpressions, and only one way of classifying them is consistent, then they are classified that way. This time, we know what the whole expression is, and we're wondering what the parts might be. [55] Here's an example. We're given that x is a theorem, and also that x implies y is a theorem. And the question is: what's y? Here's [56] how we figure it. Suppose y is an antitheorem. Now we have x is a theorem, and we're supposing y is an antitheorem. So we can use a theorem table to find out that x implies y is an antitheorem. But we already know it's a theorem. So that's inconsistent. So y has to be a theorem. Here's [57] another example. We're given that not x is a theorem. What's x? [58] If x is a theorem, then the theorem tables say that not x is an antitheorem. But we already know it's a theorem. So x has to be an antitheorem. This example is important because it means that we [59] never need to talk about antiaxioms or antitheorems. If we want to say something is an antitheorem, we just say its negation is a theorem. We also don't need to have both true and false; for false we could say not true. The logic tradition here seems a bit odd to me. Traditionally, logicians use both the words true and false, but they don't use both axiom and antiaxiom, and they don't use both theorem and antitheorem. Oh well.

The next [60] proof rule is the transparency rule. A binary expression does not gain, lose, or change classification when a classified subexpression is replaced by another expression in the same class. For example, [61] here is a binary expression. I don't care whether it is a theorem, or an antitheorem, or unclassified. It has subexpression true, which is a theorem. So if we [62] replace true with y or true, which is also a theorem, we don't change the whole expression; if it was a theorem, it still is; if it was an antitheorem, it still is; if it was unclassified, it still is. -- The subexpression x and x might be a theorem or an antitheorem or unclassified. Whatever it is, [63] x is the same, so if we replace it with x, we don't change the whole expression. It stays whatever it was.

On to the last proof rule. [64] The instance rule. If a binary expression is classified as a theorem or an antitheorem, then all its instances have that same classification. For example: [65] in the textbook somewhere there's the axiom x equals x. So by the axiom rule, [66] it's a theorem. And so [67] if I substitute any expression for both x's, it's still a theorem. We could evaluate this expression to find out that it's a theorem, but the instance rule says it's a theorem just because it's an instance of a theorem. In [68] this example, we cannot evaluate the two sides, but still it is an instance of x equals x, so it's a theorem.

That's all the rules. In this course we can use all six rules, [69] and that's called classical logic. In a logic course, it's interesting to see what you can do if you don't allow the completion rule; that's called constructive logic. And if you don't allow the consistency rule and the completion rule, it's called evaluation logic. But we can use all six rules.

[70] Now I want to talk a little about format. It helps a lot if you leave more space around operators with less precedence. Since conjunction comes ahead of disjunction on the precedence table, this expression should be written with the spacing shown. You may think that's a silly little point, but [71] look what happens if you write the same expression with a different spacing. It's misleading, and causes errors. [72] When an expression is too long to fit on one line, it should be broken at a sensible place, which usually means at the main connective. Programmers seem to appreciate the importance of good format better than mathematicians because programmers' formulas, namely programs, are much longer than mathematicians' formulas. Now about the parentheses, or brackets. There's a popular

convention in programming [73] that puts the opening bracket of a loop over on the right side, and the closing bracket over on the left. Do you know why? I'm old enough to remember where that convention comes from. It comes from the days when programs were punched onto cards, one line per card. Now you might want to reorder the lines in the body of the loop, and you do that by reordering the cards. Or you might want to delete the first line by throwing away one card, or add a new first line, or delete the last line, or add one more line to the end. So you don't want the opening bracket on the first line of the body of the loop because after editing it might be in the middle of the body somewhere, or get thrown away. And the same for the closing bracket. But we don't use punched cards anymore. When we edit, we can select just what we want to delete, and it doesn't have to be a whole line. And we can insert text where we want it and everything else moves over. So there's no purpose to that convention any more. Still, all those C and Java programmers are following it without even knowing why. [74] Let's get rid of that, and put our left brackets on the left, and our right brackets on the right. [75] And if we don't have brackets, format the same way but without the brackets.

[76] Here's a continuing equation, nicely formatted. It really means [77] that expression 0 equals expression 1, and expression 1 equals expression 2, and expression 2 equals expression 3. [78] A continuing equation is very useful as a proof that the first expression equals the last one. To help the reader understand the proof, [79] we put hints over on the right side of the page. Hint 0 is supposed to say why expression 0 equals expression 1. Usually it's just the name of some law. And hint 1 says why expression 1 equals expression 2, and so on. The hints go between the two expressions that we're trying to bridge. [80] Let's look at an example. Actually, this example is a law that's in the back of the book. It's called portation, and it's a really useful law. You should probably memorize this one. But suppose we don't know it's a law. If we want to prove this equation, we can do so as follows. [81] I started with the left side, but I could just as well start with the right side and work down to the left side. I usually start with the more complicated side, but here they're about equally complicated. The first hint says material implication. [82] That's a law from the back of the book that equates an implication and a disjunction. A law is just a theorem that's worth giving a name and worth remembering. [83] We're replacing *a* with *a* and b. Both occurrences of *a* get replaced. The second time, I had to add parentheses. And we're replacing b with c. So that's how the first line of the proof becomes the second line. [84] And the same for the other lines.

[85] Now here's another way to write the proof. I put the whole equation that I'm trying to prove on the top line, and show that it's equal to true on the bottom line. One reason I like this way is that proof is just the same as simplifying. Proof means simplify to true. Another reason is that equations aren't the only thing we prove. We could be proving a conjunction, or a disjunction, or anything. We can always prove by simplifying to true.

That's enough for this lecture. I'll finish up Chapter 1 next time.