[talking head] We've now finished all the math we need for this course. Now it's time to apply it to programming. But before we can write a program, we have to think a lot about what the program is for. Maybe we have to talk to a lot of people and find out what their needs are. And they might not agree with each other. Even one person might have conflicting goals, and you can't satisfy all of them. Sorting all this out is called Requirements Engineering. It's a great subject, and if you get the chance, you should take a course on it. The end product of Requirements Engineering is a specification, and that's our starting point.

Well, what's a specification? We can specify anything – cars, food, anything. And we do all the time. Suppose I want you to go out and buy me – a table, let's say. So I tell you - it should be wood, and it should be brown, and the height must be between 70 and 80 cm.

What I'm telling you is a binary expression. The variables are the color, the height, whatever characteristics I'm interested in. Color equals brown and 70cm less than or equal to height less than 80cm, and so on. Whenever you see a table, it instantiates the binary expression. It provides values for the variables, so you can evaluate the expression, and if you get true, you buy it, and if you get false, you keep on looking. And that's why we had to review Binary Theory.

Ok, what are we specifying in this course? You might think it's programs. And we *could* specify programs. I could say I want it to be in Java, and I want it to be no more than 20 pages long, and I don't want that stupid format that puts the left brackets way over on the right and the right brackets on the left – I want the brackets lined up nicely or whatever. But that's not the kind of specification you get from Requirements Engineering. We're not specifying what the program looks like. We're specifying what the screen looks like during execution; what the outputs are for each input; what kind of performance we want. In other words, we're specifying the computer behavior that we want – how the computer behaves when executing the program.

[1] A computer has a memory, and [2] the memory has some contents called the state, or state of memory, so the memory is also called the state space. For example, the state space might be [3] this, which is a memory that can hold an integer, followed by an integer in the range 0 to 20, followed by a character, followed by a rational number. And [4] here's an example of a state of that memory. At the beginning of a computation, the state is called the [5] prestate, and at the end it's called the poststate. Or we could say [6] initial and final state. And we use the Greek [7] letter sigma for the prestate, and sigma prime for the poststate. The little example memory has [8] 4 addresses, so in the initial state, [9] the memory contents are sigma 0, sigma 1, sigma 2, and sigma 3. And in the final state they are sigma prime 0, and so on. Referring to memory contents by address is very [10] low level, so instead we talk about [11] state variables, which just means giving names to each item of the state. Like maybe i, n, c, and x. And we just use those [12] same names for the initial values of the state variables, and primed versions for the final values. So a state variable, like i, which is really a memory location, is 2 mathematical variables, i and i prime. And these are the variables that our specifications talk about. They are the nonlocal variables of our specifications. We could specify [13] lots of other things about computations, and we will, later. We will talk about execution time, and space requirements, and interactions and communications. By the way, termination means the execution time is finite, and nontermination means the execution time is infinite. So termination and nontermination are timing issues, and we'll get to them later. But we're going to start out very simply, just talking about the prestate and the poststate, not about timing, so not about termination. [14]

[15] For now, a low level specification of computer behavior is a binary expression in variables sigma and sigma prime. [16] We provide a prestate, sigma, as input. The computer makes a computation that satisfies a specification by creating a satisfactory poststate as output. That means [17] the prestate and poststate must make the specification

true. Now [18] the same again, but high-level. A specification is a binary expression in unprimed and primed state variables. We provide values for the unprimed variables as input. The computer provides values for the primed variables as output so that all values together make the specification true.

[19] Here's some terminology about specifications, all about how many satisfactory poststates there are. [20] Specification S is unsatisfiable for prestate sigma means the number of satisfactory final states is 0. And it's [21] satisfiable if there's at least 1 satisfactory final state. The same specification could be unsatisfiable for one initial state, and satisfiable for another initial state. [22] A specification is deterministic for some initial state if there's at most 1 satisfactory final state. And it's [23] nondeterministic if there's more than one satisfactory final state. [24] An equivalent way to define satisfiable for a prestate is – there exists a satisfactory poststate. The most important definition [25] on this page is implementability. A specification is implementable means for all prestates there exists a poststate to satisfy the specification. The prestates are input, so the computer has no control over that. We could provide any prestate. And the computer has to provide a poststate as output to satisfy the specification. So for each input, there must be at least one satisfactory output. If not, there's no way a computer can provide one, so the specification is unimplementable.

[26] Let's look at some examples. Suppose the state space is integer variables x and y. This example says: make the final value of x be 1 bigger than its initial value, and make the final value of y be the same as its initial value. It says: increase x by 1, and leave y unchanged. So whatever x and y we provide, the computer can make final values to satisfy this specification. [27] It's implementable, and it's deterministic for all initial states, because there's only 1 final state that satisfies the specification. If you're worried about overflow, don't. By integer variable, I meant integer variable, not 32-bit 2's complement. Of course overflow is a problem on real computers, but it's one problem I'm leaving out of this course. That's because we'll have enough problems without it. [28] Here's another specification. It says: make the final value of x be bigger than the initial value. It says: increase x, and I don't care what happens to y. It says that last part by not saying anything about y prime. [29] It's implementable, and it's nondeterministic because there are many choices for x prime and for y prime. One way to satisfy this specification is to add 1 to x and leave y unchanged. And there are many more ways to satisfy it. This specification is weaker than the first one. So it's easier to implement. The [30] easiest specification to implement is true. All behavior satisfies it, so it doesn't matter what the computer does. It's [31] implementable and nondeterministic. And [32] the other extreme is false, which is not satisfied by anything. So it's [33] unimplementable. Here's [34] a specification that says that initially x is nonnegative, and finally y is 0. If we supply, say, 3 as input for x, the computer has to set y to 0, and the specification is satisfied. If we supply minus 3 as input for x then the specification is false no matter what the final values of x and y are. There's no way for the computer to satisfy the specification so it's [35] unimplementable. But maybe we never want to have a negative input. In that case, [36] here's the specification we should write. If we supply 3 as input to x, then just like before, the computer has to set y to 0. We don't care what would happen if x were negative, because we're not going to give x a negative value. And that's what this specification says. It says: if x is negative, then anything is fine. [37] This specification is implementable and nondeterministic.

There are 2 specification notations that are really useful. One is [38] ok, which means do nothing, leave everything alone. Using the low level state, that's [39] sigma prime equals sigma. Using the high level state variables, that's [40] x prime equals x and y prime equals y and so on, whatever the variables may be. The other useful notation is [41] assignment, x is assigned e, or x gets e, or set x to e, where x is a state variable and e is any expression. For example, [42] x gets x plus 1. Notice that the assignment symbol is colon

equals, not just equals. Now I know that in some programming languages it's just an equal sign, and I know why, too. [43] The Fortran language was designed long ago, in 1955, by John Backus, and he used an equal sign. And he told me that he regretted using the equal sign for assignment. Writing x equals x plus 1 is just stupid, because x isn't equal to x plus 1 unless x is infinity. So in 1958, when John was one of the designers of Algol, he and the others decided to use colon equals for assignment. And ever since then all well designed programming languages have used it, like Pascal and Ada and Modula and Turing and Eiffel, and so on. Poorly designed languages have copied the Fortran mistake. In this class, never use just an equal sign for assignment. [44] [45] At the low level, it means sigma prime is the same as sigma except at address x the value is e. [46] At the high level it means x prime equals e and y prime equals y and so on, whatever the other variables are. For example, if the variables are just x and y, then x gets x plus 1 means [47] x prime equals x plus 1 and y prime equals y. Assignment means the variable being assigned has the right final value, and all other variables are unchanged. Don't forget that last part.

One final example, with no new notations in it, is [48] this one. I guess the only point here is that you can mix all the notations together any way you want. It says that if x and y are initially equal, then the final value of x is the sum of the initial value of x and 1, and the final value of y is unchanged. And if x and y are initially unequal, then their final values should add up to 3. That's nondeterministic because there's some choice of final values.

[49] If you have 2 specifications S and R, dependent composition is a way of composing them into 1 specification, written S dot R. It describes sequential execution: first behave according to specification S, and then according to specification R. Its definition is [50] there exists x double prime, y double prime, and so on, whatever the state variables happen to be. Those double prime variables stand for the intermediate values of the variables in between execution of S and R. So the computation is described by S, except that we replace its primed variables with double primed variables because the final values according to S are really intermediate values, and then by R but replace its unprimed variables with double primed variables because the intermediate values are the initial values for R. I think an [51] example will help, and let's just have one integer state variable x. So first, either leave x alone or increase it by 1. And then again, either leave it alone or increase it by 1. Using the definition of dependent composition, [52] there exists x double prime, and then we write the first specification, but everywhere there was a prime, we put a double prime. And then, we change the dot to and. And then we write the second specification, but everywhere the was no prime we put a double prime. [53] Now we distribute and over or. That means we take [54] this and this, or'd with [55] this and this, or'd with [56] this and this, or'd with [57] this and this. So we get [58] a disjunction of 4 things. And one of the laws from the back of the book says [59] we can distribute the exists to each of the disjuncts. Now the [60] hint says one-point law. If you look that up you find that it comes in 2 versions, the existential version and the universal version, and we want the existential version. Which says: [61] if you have exists and a variable, and then you have an [62] equation with that variable, and it's [63] conjoined to something, then you can [64] get rid of the quantification, and get rid of the equation, and just write the rest down, but substituting for the variable. [65] So the first disjunct says [66] x prime equals x. The second one says [67] x prime equals x plus 1, the next one says x prime equals x plus 1 again, and there's no need to write it again, and the [68] last disjunct says x prime equals x plus 1 plus 1, which is x plus 2. So that's the resulting specification. It says either leave x alone, or increase it by 1, or increase it by 2. Notice that the specification still just talks about the initial and final values, not about the intermediate values. [69] Now let's look at a [70] picture of it. The [71] leftmost column says that either x stays the same or is increased by 1. That's being composed – see the dot – with [72] the same thing again. The definition of independent

composition says [73] there's an intermediate state. And the [74] final result shows that from 0 you can get to 0, 1, or 2. Actually, to get from 0 to 1 there are 2 routes, but the final answer doesn't say anything about how many routes there are.

[75] Here's another example [76], this time using 2 integer variables x and y. x gets 3 and then y gets x plus y. x gets 3 means [77] x prime equals 3 and y is unchanged. y gets x plus y means y prime equals x plus y and x is unchanged. The dependent composition means [78] there exist intermediate values x double prime and y double prime, of type int, which I often don't bother to write but it really is there, such that, now change final values to intermediate values, so x double prime equals 3 and y double prime equals y, and now change initial values to intermediate values, so x prime equals x double prime and y prime equals x double prime plus y double prime. Now we can use one point twice because we have exists with 2 variables, and each one has an equation. So leave out the exists, leave out the equations, and write the [79] rest but replace the double prime variables with what they're equal to. And so x ends up 3, and y ends up 3 plus the initial value of y. And that's exactly what you knew the assignments said.

[80] Here are some laws about specifications. [81] The first one says that ok is both a left and right identity for dependent composition. If you do nothing, and then do P, it's the same as doing P. Also, if you do P, and then do nothing, that's just doing P. The [82] next one says that dependent composition is associative. So we won't bother to write the parentheses. The [83] next few laws are just laws of binary theory, but we can look at them again as specification laws. The first two are laws that programmers know. The first one says that if the then-part and else-part are the same, you might as well not have the if at all. And if you negate the if-part, you switch the then-part and else-part. Then the case analysis laws which give two ways to rewrite an if. [84] Now there's a distributive law. Let me read it to you. If you do P or Q, and then you do R or S, that's the same as doing P and then R, or P and then S, or Q and then R, or Q and then S. And then a few more laws. They're all in the back of the book. But I want to pay special attention to the [85] last one, the substitution law. It says, if you have an assignment followed by any specification, that's the same as that specification, but with a substitution. Replace x with e in P.

[86] Here are some examples of it. First, [87] x gets y plus 1, and then y prime greater than x prime. So replace x by y plus 1 in y prime greater than x prime. But there is no x in y prime greater than x prime. There's x prime, but no x. So substitution does nothing, and the [88] result is still y prime greater than x prime. That means the computer chooses any final values for x and y such that y is greater than x. It could choose 3 for x and 5 for y. If it's going to do that, then there was no point at all to the assignment x gets y plus 1. The [89] next example says increase x by 1, and then make the final values of both y and x be bigger than the initial value of x. Now be careful. The initial value of x means initial for the second part, so that's after x is increased. Anyway, replace both occurrences of x with x plus 1, and the [90] result is that the final values of y and x must be bigger than the initial value of x plus 1. In the [91] next example, the only point is that when you replace x by y plus 1, you have to [92] add parentheses so the precedence doesn't go wrong. And the point [93] of the next example is that you replace only the nonlocal x, not the local x. So you get [94] 1 is greater than or equal to 1, and that can be simplified, so [95] the final result is that y has to be even in the end. In the [96] next example, we're replacing x with y, and that x is nonlocal, so we do have to replace it, but we can't put a nonlocal y in a place where it would look local. So first we have to rename the local variable. Let's call it [97] k. And now [98] we can make the substitution. The [99] next example says x gets 1, and then ok. The point here is that you don't say there's no x to be replaced. You have to realize that ok stands for [100] x prime equals x and y prime equals y, assuming the variables are x and y in these examples. So there is an x to replace, and the result [101] is x prime equals 1 and y prime equals y. The [102] next example is really the same thing again. In the second assignment, the occurrence

of x is not apparent. But if we [103] rewrite it, we see the x, and then we [104] replace it, and get x prime equals 1 and y prime equals 2, which is obviously the effect of those two assignments.

There are still 2 more examples, so let's [105] clear some space for them. In the [106] next example, we have 3 assignments in a row. We just did the first 2, but that's not the smart way to do this one. The smart way is to rewrite the last assignment like [107] this, and then use the substitution law for y gets 2 followed by the last part. So that's [108] x prime equals x plus 2 and y prime equals 2. And then we use the substitution law again to get [109] x prime equals 1 plus 2 and y prime equals 2. In a long string of assignments, it's best to work from right to left, because that way you can keep using the substitution law. If you go in the other direction, you get the same answer, but it's more work because you have to keep using existential quantifications. In the [110] last example, there's only 1 assignment, so we may as well use the substitution law where we can, replacing x by 1 in the middle part. So we get [111] x prime greater than 1, and we still have the last part. And there's nothing we can do but use the definition of dependent composition [112]. Now we have exists x double prime and y double prime. The y double prime isn't actually used, so we can get rid of that. [113] Good. Now to use the one point law, we need to see x double prime equal something. So that's [114] easy to arrange. Now we replace x double prime with x prime minus 1 in the first conjunct [115], and then simplify [116] to get x prime greater than 2.

All of these examples were small and easy to see what's happening, but the same techniques work for large programs where it's not obvious what is being computed. And best of all, it's possible to write a program to do these kinds of calculations automatically. That's where we stop this lecture.