[1] We introduced a time variable for the purpose of calculating execution time. The time variable has played no role in the computation itself, so far. Now I want to suggest that it can play a role in the computation. Computers have clocks, and the clock tells the current value of the time variable. We can have an assignment like [2] this one, where t is the time according to the computer's clock. It causes [3] no problem at all in the theory of programming. Variable deadline has been assigned a future time, and we can make a [4] test like this, maybe inside a loop, that says: if the current time according to the clock is before the deadline, do one thing, and if not, do something else. So the time variable can affect the course of the computation. Still it's [5] no problem for the theory, and for the calculation of execution time. What would be a problem is [6] an assignment to t like this one. It's unimplementable. We could say it resets the clock, but we can't say it resets time. If we're going to make assignments like this, then t no longer has any relationship to time. The only assignments to t that are allowed are those that represent the passage of time, according to some policy, like real time or recursive time.

In some programming languages, there are delay statements, maybe like [7] this, which means do nothing but wait until time w. The formal definition is [8] t gets max of t and w. In other words, increase the time variable to w, unless t is already bigger than w, in which case just leave it alone. We can't make time go backwards. And leave all other variables unchanged. So it's an assignment, but it's not a program, just a specification, because the only assignment we're allowed to make to t in a program is one that follows a timing policy. So we have to refine this specification. The refinement is called a [9] busy-wait loop. And [10] here it is, assuming time is integer-valued and using recursive time. We can [11] prove this refinement by cases. [12] Here's the first case. I rewrite ok as [13] t gets t because I want to use the other conjunct as context to say that [14] t is the max of t and w, and [15] that implies our specification. [16] We [17] other case, well, we have to replace wait until [18] with its definition. In the left conjunct, t less than w is the same as [19] t plus 1 less than or equal to w. And in the right conjunct, we should rewrite the final assignment and then use the substitution law. I'll just take a [20] shortcut. With the left conjunct as context, the assignment is [21] t gets w, and I've rewritten that context back in its earlier form to make it more obvious that the assignment is the same as [22] t gets max of t and w. And that [23] implies the specification. So now we have implemented a wait construct.

[24] We have seen the use of a space variable to calculate space requirements. Computers do keep track of the space usage, and if they make this information available in a form that can be accessed from within a program, then we can [25] write programs in which the course of the computation depends on space usage. It's just like the time. You can read it, but [26] not set it arbitrarily. [27] The assignments to the space variable must account for space usage. And it has to be [28] real space usage, if I can use that term by analogy to real time usage, meaning it has to depend on the compiler and operating system of the computer that the program will run on. And the same was true for time, by the way. If you want to use the computer's clock, then you'll have to use the real time measure. I don't have any good examples of space dependent programs.

We move on now to [29] assertions. In some programming languages you can write [30] assert b, where b is a binary expression, to mean something like [31] I believe b is true, but I'm not absolutely certain, so I put in this check. If it comes at the beginning of a procedure or method, it might be called [32] a precondition. If it comes at the end, or refers to the state at the end, it might be called a [33] postcondition. If it comes at the beginning or end of a loop, it might be called an [34] invariant. Whatever it's called, [35] here's its formal definition. If b is true, fine, there's nothing to be done; continue execution with whatever follows this assert. If not, something has gone wrong, so print an error message, and suspend execution. Don't go on and execute anything that follows this assert. That's what wait until infinity says. — If the program is correct, then all asserts are [36] redundant. You

can remove them from the program. But in case there's something wrong, they [37] add robustness to the program. They're a useful way to check if things are all right. [38] But they're not free. They cost execution time. So use them only where the error checking is worth the cost.

The [39] next sort of assertion is ensure b. It means something like [40] make b be true without doing anything. Formally, that's [41] if b then fine, same as before, but if not, then make it be true without doing anything. Well that's just [42] b prime and ok. I'm going to show you how to use this construct, but first we should note that it's [43] unimplementable. You can't just make anything be true that you might like to be true without doing anything. But even though it's unimplementable [44] by itself, it can be part of something larger that *is* implementable.

[45] Nondeterministic choice can be obtained by disjunction. Do P or Q. We resolve nondeterminism by refinement to a program. Another possibility is to [46] make disjunction a programming notation, so we don't have to refine it. Maybe it would look more like a programming notation if I use a [47] bold word **or**. Making it a programming notation just means that the programming language implementer has to resolve the nondeterminism, maybe by always choosing the first one, or maybe by choosing randomly. So we can write [48] x gets 0 or x gets 1, which is [49] x prime equals 0 and all other variables, well let's just have one other variable y, is unchanged, or x prime equals 1 and y is unchanged. So x could be 0 or 1 in the end. But I'm going to follow that with [50] ensure x equals 1, which is x prime equals 1 and all variables unchanged. The dependent composition is [51] an existential quantification, like this. And we can get rid of it by using [52] one point for x double prime, replacing it with x prime, and [53] one-point for y double prime, replacing it with y prime. And [54] this is what we get. And we can simplify this to [55] x prime equals 1 and y prime equals y, which is the same as [56] x gets 1. We started off with a choice between x gets 0 and x gets 1, but it turns out later that it wasn't a free choice. We had to choose x gets 1. The implementation of these constructs is called [57] backtracking. You make a choice, and if it turns out later that it was the wrong choice, you back up and choose the other option. This was a simple example, with one choice and one ensure immediately after. But you could have many of each, and they could be ordered and nested in complicated ways. You execute it like this. Whenever there's a choice, choose one. Whenever you hit an ensure and it turns out the condition is false, back up to the last previous choice that still has untried options, and choose a different option. There are two different ways such an execution might end. One way is that you finally make it past all ensures right to the end of the program. That's success. The other way is that you come to an ensure with a false condition, and you back up right to the beginning of the program without finding any choices with untried options. That's failure. It means the execution did not satisfy the program. But you have to expect that, sometimes, if you use an unimplementable programming construct.

[58] Here's a nice little example. Given natural n, find natural s satisfying this condition, which means that s is the square root of n, rounded down. The square root of 16 is 4, and the square root of 17 up to 24 is 4 point something, so we take it to be 4. How can we calculate square root? With these constructs, it's easy. First, a [59] nondeterministic assignment, using the notation suggested in one of the exercises at the back of the book. That doesn't mean s gets the bunch 0 to n plus 1. It means s gets one of the numbers in the bunch 0 to n plus 1. Any one of them. It's s gets 0, or s gets 1, or s gets 2, and so on. And then [60] ensure exactly the thing we want to be true. The execution will choose one of the numbers in the bunch, maybe 0, and test the condition, and if it's false it makes another choice, maybe s gets 1, and test again, and so on. In the end, s will be the square root, but it's a really inefficient program.

As you see, the combination of nondeterministic assignment and ensure makes some programs very easy to write, but very inefficient to execute. There are programming languages with these kinds of features. The best known is probably Prolog. And there are some that have backtracking in the limited context of pattern matching, such as ML and Haskell. So I have shown you how to reason about such programs. But in this course, if there's a programming exercise, don't use this feature. I want much better programs than you get with this feature.

Ok, that's it for this lecture.