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 [38] redundant. You can remove them from the program. But in case there's something wrong, they [39] add robustness to the program. They're a useful way to check if things are all right. [40] But they're not free. They cost execution time. So use them only where the error checking is worth the cost.

The [41] next sort of assertion is ensure b. It means something like [42] make b be true without doing anything. Formally, that's [43] if b then fine, same as before, but if not, then make it be true without doing anything. Well that's just [44] b prime and ok. I'm going to show you how to use this construct, but first we should note that it's [45] 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 [46] by itself, it can be part of something larger that *is* implementable.

[47] Nondeterministic choice can be obtained by disjunction. Do P or Q. We resolve nondeterminism by refinement to a program. Another possibility is to [48] 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 [49] 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 [50] x gets 0 or x gets 1, which is [51] 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 [52] ensure x equals 1, which is x prime equals 1 and all variables unchanged. The sequential composition is [53] an existential quantification, like this. And we can get rid of it by using [54] one point for x double prime, replacing it with x prime, and [55] one-point for y double prime, replacing it with y prime. And [56] this is what we get. And we can simplify this to [57] x prime equals 1 and y prime equals y, which is the same as [58] x gets 1. We start 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 [59] 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.

[60] 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 [61] 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 [62] 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