[1] For the past several lectures, you have been learning the best programming theory there is. It applies more generally than other programming theories, and it is simpler to use. A specification is just a binary expression, and refinement is just implication. In spite of being the best, it is not the best known, and not the most widely used program theory. That honor belongs to the original program theory from 1969. It is due to my friend and colleague Sir Tony Hoare, and is often called Hoare Logic. Tony saw that the theory of this course is better, and switched to it, but other people just continue doing and teaching what they learned to do, so in this lecture I want to introduce you to the terminology of the old theory. And I'll use this binary search program for that purpose. In the old theory, you put [2] assertions at strategic places in the program. An assertion is a binary expression that talks about a single state. And you put set brackets around it. It isn't a set. The reason for the set brackets is that long ago, in the Pascal programming language, set brackets were used for comments. The idea is that the assertion is true whenever execution passes through the place in the program where the assertion is. [3] An assertion that appears at the beginning of a program is called a precondition. [4] An assertion that appears at the end of a program is called a postcondition. [5] An assertion that appears at the start of a loop and at the end of the loop body is called an invariant. [6] A variant is an integer expression that is positive at the start of a loop body, [7] and at the end of the loop body it is decreased but not below zero. And [8] there are other assertions that are needed for a proof, but they don't have special names.

One problem with the old theory is that assertions talk about a single state, making it awkward to relate a prestate to a poststate. Another problem is that the assertions are situated in the program, making it awkward to start with a specification, and refine it to get a program. So the old theory hasn't been used much for program development; it is used mostly for verification after a program has been written. And it's not very good for that, either.

I'm not going to tell you the proof rules for this old theory because you already know a better theory, so there's no point. Instead, I'm going to give these old terms new definitions within the modern theory, independent of location in a program. [9] I'll start with precondition. If P and S are specifications, the exact precondition for specification P to be refined by specification S is this formula, which is exactly the same as the refinement formula, except refinement says, for all prestates sigma and poststates sigma prime, P is implied by S, and this formula just quantifies over poststates sigma prime. Now P and S can talk about both sigma and sigma prime. This formula makes sigma prime local, and sigma is nonlocal, so this formula just talks about sigma, the prestate.

Let's [10] look at an example with one integer variable x. We want to find the exact precondition for x prime greater than 5 to be refined by x gets x plus 1. First of all, x prime greater than 5 **is not** refined by x gets x plus 1. Adding 1 to x doesn't necessarily make it bigger than 5. The formula says [11] for all x prime x prime is greater than 5 if x gets x plus 1. Since there's only 1 variable, the assignment [12] is just x prime equals x plus 1. Now we use [13] a one-point law to get rid of the quantifier, and then [14] simplify, and we find that the exact precondition for x prime greater than 5 to be refined by x gets x plus 1 is - x greater than 4. The exact precondition tells us, under what initial condition, does increasing x by 1 make it bigger than 5. And the answer is, of course, if x starts out bigger than 4. This is just a tiny example, but the same calculation works for large examples too. If you want to know under what condition some program works, you don't have to guess. You calculate. And you find out exactly when it works and when it doesn't. [15] x prime greater than 5 is *not* refined by x gets x plus 1, so there's 2 things you can do about it. One is to change the right side, and find something it is refined by. The other is to weaken the left side by [16] adding the exact precondition as antecedent. Promise a little less. Don't promise to make x bigger than 5. Promise that if x starts out bigger than 4, then it will become bigger than 5.

[17] There's a similar formula for the exact postcondition, where sigma is local and sigma prime is nonlocal. So it's talking about the poststate. As an [18] example, find the exact postcondition for x greater than 4 to be refined by x gets x plus 1. The specification x greater than 4 isn't even implementable because there's no way a computer can make its input be greater than 4. But let's do it anyway and see what we get. [19] Here's what the formula says. And again, in one variable, the assignment [20] is just an equation. To use one point, we would need x equals something in the antecedent. So [21] that's easily arranged. And now [22] one point says it's x prime minus 1 greater than 4, which is [23] x prime greater than 5. So that tells us that, although adding 1 to x doesn't make the input be bigger than 4, if the result turns out to be bigger than 5, then we know the input was bigger than 4. And although [24] this isn't a refinement, if we just [25] weaken the problem with the antecedent we just calculated, we get a refinement. On the left, we can read that as saying if the output is greater than 5 then the input was greater than 4. Or we can use [26] the contrapositive law to turn the implication around. a implies b is the same as not b implies not a. So we can rewrite the specification to say – if x starts out less than or equal to 4, then it will end up less than or equal to 5. And that *is* refined by x gets x plus 1.

[27] A sufficient precondition is any precondition that implies the exact precondition. A necessary precondition is any precondition that is implied by the exact precondition. So the exact precondition is the necessary and sufficient precondition. And similarly for postconditions. [28] These laws say what you already know. You can use a sufficient precondition or sufficient postcondition to weaken a specification and turn a non refinement into a refinement.

[29] Next, I'll define invariant. Let S be a specification. Let I be an assertion with all variables unprimed. Let I prime be the same as I but with primes on all variables. Then I is an invariant for S if I implies I prime is refined by S. Or we can use portation and write it [30] this way. This says that executing S in a state where I is true produces a state where I is again true.

[31] Here is an example. In variables x and y, prove that y equals x squared is an invariant for those two assignments. So we [32] start with the formula. Well, I'm working inside the quantifiers, so I don't need to write them. Now [33] put in the invariant and specification for this question. And [34] replace the last assignment by its equivalent binary expression so we can use the [35] substitution law. Now we don't have any programming notations, so we [36] use the usual binary and arithmetic laws to get true. You can check that out if you want to.

And finally, [37] a variant is just a time bound, using the recursive measure, but with a clock that runs backwards. The clock starts at the final execution time, and then runs down to zero.

We won't be using the old theory. But we will use invariants as I've just defined them when we look at for-loops, and we will see assertions as a programming language notation in the next chapter of the textbook.