[1] This lecture is the final review, and course summary. These are the topics we covered in the first half. We've been using [2] the mathematical background a lot, so you should be used to it by now. The [3] main point of the course is how to develop programs by writing specifications, and refining them, proving your refinements as you go, so that executing the programs will always satisfy the specifications. And we include any time and space limitations in our specifications and prove that they are met in exactly the same way we prove that the computations have the right results.

In the [4] last part of the first half we looked at how some common programming language features fit into the theory of programming. Variable declaration is existential quantification. Frame, which appears as module or object in some languages, temporarily removes variables from a scope. Array element assignment causes a small problem, and we saw that the solution is to rewrite it as an assignment to the whole array. And we saw how to reason about loop constructs.

[5] Here are the topics of the second half of the course. We [6] continued with some programming language features, such as time dependence and backtracking and random number generators and functions, that make a big and interesting difference to the way we write programs. Then we saw how [7] construction and induction axioms are used to define data structures and program structures, so we could use them in [8] theory design and implementation. Theory design sounds like something for logicians, but actually all programmers are theory designers, and we looked at stacks, queues, and trees as examples of theory design. You should know how to show that a theory is consistent or inconsistent, and what it means for a theory to be complete or incomplete. [9] Data transformation is a great way to reimplement a data structure, and you should know how to do it and why it's best to do it that way, rather than reprogramming the operations from scratch. [10] Then we looked at concurrency, both as something a programmer might use, and as something a compiler might introduce into a program. And finally [11], we saw how to formalize and reason about interaction. We looked at a kind of shared variables, called interactive variables, and at communication channels, for interaction between concurrent processes or between programs and people. It's a lot, but I want you to have a really good introduction to formal methods, and see how they apply across a broad range of programming styles and constructs.

The world doesn't stand still, and people continue to propose new patterns of execution and new notations for them. For example, there are several kinds of concurrent execution that are currently used. You can find some of them in the exercises of the textbook. I don't think it's important that you know them. I think it's much more important that when you are presented with a new execution pattern, you are able to describe it formally, and prove properties of it. And when you are presented with a formal description, you are able to understand the description and see what execution patterns are possible. For example, one of the exercises talks about [12] disjoint composition. I'll read it. Concurrent composition P parallel Q requires that P and Q have no variables in common, although each can make use of the initial values of the other's variables by making a private copy. An alternative, let's say disjoint composition, is to allow both P and Q to use all the variables with no restrictions, and then to choose disjoint sets of variables v and w and define disjoint composition like that. As you can see, it's [13] P and Q, as usual for concurrent composition, but this time P and Q aren't limited to work on just part of the variables, so P and Q could easily be false if P says the final value of a variable is one thing, and Q says it's another. So that's why we have [14] these equations. After P is executed, we forget about everything P says about the final values of variables except for v. And after Q is executed, we forget about everything Q says about the final values of variables except for w. Since v and w are disjoint sets of variables, there's no conflict.

The exercise asks [15] us to prove that this kind of concurrency is implementable. To prove that, I'm going to need a notation for substitution. [16] Actually, we already have one. The application law says that a function applied to an argument is equal to the body, but substitute the argument for the parameter. So function application is a formal notation for substitution. The question says that variables v and variables w are disjoint, with no variables in common. But it doesn't say that's all the variables. [17] So let me use x for the remaining variables, if there are any. Before I prove implementability, let me re-express disjoint composition. [18] I'll start with P dot v prime equals v. We get [19] rid of the sequential composition. There exist intermediate values v double prime, w double prime, and x double prime, and now we want P but replace the primed variables with double primed variables, conjoined with v prime equals v, but replace the unprimed variables v with double primed variables. Now we have exists v double prime, and we have v double prime equals something, so we can get rid of v double prime by [20] one point. What I want to do next is to [21] rename variables w double prime and x double prime. I want to rename them to w prime and x prime. Now it seems that we already have w prime and x prime in [22] here. But in here, w prime and x prime are local to this little function. They aren't the same variables as the ones that will replace w double prime and x double prime. The new w prime and x prime are not local to this little function. So they are fresh variables. And [23] here's the result of the renaming. Now the little function's parameters look the same as its arguments, even though they aren't the same variables. So when we apply the function to its arguments, we get [24] the same expression P. That's the end of that little calculation. And of course we can [25] do exactly the same for Q dot w prime equals w. Now I can re-express [26] disjoint composition, which is defined [27] this way, and now we can write it like [28] this. And that's a lot more convenient for proving implementability. [29] Here we go. We haven't been talking about time, so implementability is [30] for all values of unprimed variables, there exist values of primed variables such that disjoint composition is satisfied. Now we re-express disjoint composition [31]. We can get rid of the [32] first quantification over x prime because [33] x prime is local to both conjuncts in the body [34]. Now [35] exists v prime and w prime can be split into two separate quantifications [36]. And that's so that we can factor out [37] this conjunct in front of exists w prime [38]. And now we factor [39] this conjunct out of the exists v prime [40]. And I'll just [41] regroup the existential quantifications. Now we use a splitting law [42], and we have the [43] definitions of implementability of P and Q. So we've proved that the disjoint composition of P and Q is implementable if and only if both P and Q are.

[44] Part b of the exercise asks us to describe how this kind of concurrency can be executed. [45] Here's one way. Make a copy of all variables. Execute P using the original set of variables and concurrently execute Q using the copies. Then copy back from the copy w to the original w. Then throw away the copies. There may be variables other than v and w; if so, their final values are arbitrary, and this implementation makes them be what P says they should be. Well, that's an informal description of the execution, and we can't prove it's correct until we formalize it. [46] Making a copy of the variables means declaring new variables that are initialized to the values of the old variables. Then we execute P and concurrently we execute Q but substitute the new variables in place of the old. Then we copy back w. Throwing away the copies happens because it's the end of their scope. Now that we've formalized the implementation, we can prove it correct by proving this implication. It's not hard, and I won't bother.

I don't think disjoint composition is important, but I don't know what new programming constructs will come along in the future, and I think it is important to be able to reason about whatever new constructs may come our way, and to be able to prove programming steps that use the new constructs. This was just an example of how you formalize a new construct and prove its implementability.

[talking head] The way we learn to program, certainly the way I learned to program, and I bet the same for you, is to learn how programs are executed. And if you don't take this course, or one like it, then that's your only understanding of programs. With that understanding, the only way to check whether a program is correct is to test it – by executing it with a variety of inputs to see if the outputs are right. All programs should be tested, but there are two limitations, or problems, with testing.

One problem with testing is: how do you know if the outputs are right? Maybe you wrote the program to tell you answers you didn't already know, so testing doesn't tell you if it's right. In that case, you should test to see at least if the answers are reasonable.

The other problem with testing is: you cannot try all inputs. There are too many. Maybe an infinite number. Even if all the test cases you *do* try give reasonable answers, there might be errors lurking in cases you didn't test.

From this course, you now have an understanding of programs that's completely different from execution. When you prove that a program refines a specification, you are considering all inputs at once, even if there are infinitely many of them. And you are proving that the outputs have the properties stated in the specification, even if you didn't know what the value of the output should be. That's far more than can ever be accomplished by testing.

But it's also more work than trying some inputs and looking at the outputs. So that raises the question: when is the extra assurance of correctness worth the extra work?

If the program you are writing is easy enough that you can probably get it right without any theory, and it doesn't really matter if there are some errors in it, then maybe the extra assurance isn't worth the trouble. If you are writing a pacemaker controller for a heart, or the software that controls a subway system, or an air traffic control program, or nuclear power plant software, or any other programs that people's lives will depend on, then the extra assurance is *definitely* worth the trouble, and you would be negligent, and you could be sued for negligence, if you did not use the theory.

To prove that a program refines a specification after the program is finished is very difficult. It's much easier to do the proof while you're writing the program. The information you need, to make one step in programming, is exactly the same information you need to prove that step is correct. The extra work is mainly to write down that information formally. It's also the same information that will be needed later for program modification, so writing it explicitly at each step will save effort later. And if you try to prove a step, and you find that it's incorrect, you save all the effort of building the rest of your program on a wrong step. And after you become practiced and skillful at using the theory, you find that it helps in the program design; it suggests programming steps. In the end, it may not be any extra effort at all.

In this course we looked only at small programs. But the theory is not limited to small programs; it's independent of scale; it's applicable to any size of software. In a large software project, the first design decision might be to divide the task into several pieces that will fit together in some way. You can write this decision as a refinement, specifying exactly what the parts are and how they fit together. And then the refinement can be proven. Using the theory in the early stages of a large project is enormously beneficial, because if an early step is wrong, it's enormously costly to correct it later.

As a programmer, you might not be allowed to use formal programming methods. Your manager might not know how to use them, and it may not be company policy. But that will change, especially when people like you become managers.

What's needed right now are good tools to support the use of formal methods. Ideally, an automated prover watches what you do, suggesting refinements, and proving each refinement for you. As long as your program is correct, it keeps quiet. But it complains whenever there is a mistake, and says exactly what's wrong. We have syntax checkers that

say: syntax error on line 23. And type checkers that say: type violation on line 23. This is just the next thing in that series. It's a logic checker. It says: bug on line 23, and tells you what the bug is. And the only way to do that is to use the theory of programming – to write specifications, and prove refinements. At present there are a few tools in use that provide some assistance, but they're far from ideal. There are still plenty of opportunities for tool builders.

The main thing that I hope you got out of this course is a new understanding of programs. It's been a pleasure. Bye bye.