[1] This lecture is a mid course review. You are looking at the topics we have covered up to now. We started with [2] binary theory, and I think you can appreciate how useful it is for writing specifications and proving properties of programs. You should know the [3] laws that you need all the time, and you should be a little familiar with the others. At least know how to find them fast. And you have to know how to [4] prove refinements, so you need to know some proof techniques. You probably don't need to spend any time on [5] number theory because I'm sure you know it already. You certainly don't need to spend time on [6] character theory because it's nothing. We learned about [7] bunch theory, but we haven't used it much yet. But we will use it a lot in the second half of the course, so you really do need to know it. We won't be using [8] sets very much, but we will use [9] strings and lists quite a lot. [10] Functions haven't been used much yet, but they are main players in the second half. You don't need all the details; right now you just need the basics about application and scope. And of course you need to know [11] quantifiers. You've already seen how important they are in the theory of programming. They're in the definition of implementability, and sequential composition, and variable declaration. [12]

It's too bad that a course has to start with the basics, instead of doing the interesting stuff right away. But it's the same for anything in life. If you want to learn to play a musical instrument, you can't start off playing your favorite tunes. You have to learn the basics, the techniques, the music notations, and practice a lot, before you get good at it. But I shouldn't be too negative about the basic stuff. This may be the first time you've ever seen a consistent notation and set of axioms for the foundation of mathematics. If you ever want to write a program to do mathematics, like Maple or Matlab or Mathematica, or even just understand how one works, you need to have a basis like the one I've given you.

The main point of the course is [13] program development, starting with specifications, and refining them to programs. That's what you have to know best. Including [14] time and space calculations.

Maybe the best way to prepare for a test, after reading the textbook, is to choose some problems from the textbook and try them. If you're not sure whether you're on the right track, ask the course instructor. That's what the instructor is there for. And that's what I'm going to do right now, choose a random exercise and do it.

[15] The exercise says to write a program that cubes using only addition, subtraction, and test for zero. I'm just going to use natural numbers, and [16] here's my specification. x prime equals n cubed. There must be lots of ways to refine this, and [17] here's the first one that comes to my mind. x gets n to start with, and then multiply by another n, and then once more. What I like about this is that I just have one new specification to refine, even though I used it twice. Now, wait a minute. The [18] middle one says to multiply x by n, but it doesn't say leave n unchanged. So the [19] last one might get a changed value for n, and that would be wrong. If I hadn't noticed that I would have found out anyway as soon as I tried to prove this refinement. There are two ways to fix it. One is to add n prime equals n to this specification, and the other is to make n a constant. Either way would work, and I'm [20] choosing to make n a constant. Now we have to [21] refine the new specification. We need x times n, but we don't have multiplication, so we have to add. That could be x plus x plus x and so on, n of them. Or it could be n plus n plus n and so on, x of them. If we count up to x, or up to n, we have a problem because we can only test for zero. So we'll have to count down. Counting x down isn't so good because we don't want x to end at 0. We want it to end with the answer. We want to accumulate a sum, like [22] this. x gets 0, and then x prime equals x plus something. We're adding n's. We want to add x of them, but that's x before we assigned it 0. So I'll introduce [23] variable y, and initialize it to x, and we'll count y down to 0. So now I can finish off this line by saying we have to add [24] y more n's. The proof is two substitutions. Replace x by 0 and then y by x and you've got the left side. Now we have a [25] new specification to refine. We can start with [26] this, which is why we introduced y.

If y equals 0, then we want x prime equals x plus 0, so that's ok. [27] Else add one more n, decrease y, and round again. We're done the program, but I still have the timing and the proof to do.

As a C program, [28] it looks like this. And you see that I haven't used anything but adding, subtracting, and test for 0. The multiplications were just in the non program part, in specifications that got refined. I wouldn't trust a C compiler to do a good job of compiling function call, so I'll change the final call to function R into a go to, and then inline it, like [29] this. — Or, instead of a goto, we could use a [30] while. – And we could inline Q at both its calls, and then I guess we don't need function names anymore [31]. — [32]

Next I want to do the [33] timing. And let me move things over to make room for timing expressions. [34] There. We have to put t gets t plus 1 before the recursive call on the last line, if we're doing recursive timing. [35] There. And that's the only loop, so we don't need any other time increment. To find the timing specifications, I think the easiest place to start is the last refinement, because it just counts y down to 0, so it's [36] t prime equals t plus y. And I'll prove it. By cases, starting with [37] y equals 0 and ok. [38] Expand ok. With the context y equals 0, we can [39] change x prime equals x into x prime equals x plus y times n, and we can change t prime equals t into t prime equals t plus y, and we can [40] specialize and drop the two conjuncts we don't need. [41] The [42] other case looks like this. We should use the [43] substitution law 3 times, first replacing t with t plus 1, then replacing y with y minus 1, then replacing x with x plus n. — Then we [44] simplify, and drop y unequal to 0, which didn't help us anyway, and we get the left side. [45] Now I'm going to prove the middle refinement, starting with the right side. [46] Use the substitution law twice. First replace x with 0, and then replace y with x, and you get [47] this. And if you simplify, you get [48] this, and that tells us what the timing is for the middle refinement. [49] [50] Now for the top refinement, [51] starting with the right side. We have an assignment, and then two specifications, so we can't use the substitution law from right to left. We can just use it to replace x with n in the middle part. [52] — Now I think we have to use the definition of sequential composition. [53] We can get rid of x double prime by one point because we have x double prime equals n squared. We can get rid of t double prime by one point because we have t double prime equals t plus n. And we can get rid of y double prime because it's not there at all. [54] And that tells us the timing for the whole program. [55] It's n squared plus n. I couldn't have guessed. But I can calculate, and there it is. [56]

I think we can do better than n squared. Maybe we can get a linear time program if we express n cubed in terms of n minus 1 cubed. [57] n minus 1 cubed is not a problem because we find 0 cubed then 1 cubed then 2 cubed, and so on, each one from the previous one. The multiplications don't scare me because we can just add 3 of them. What scares me is the square. How do we get n squared? — Well, maybe the same way. We express n squared in terms of [58] n minus 1 squared. We'll need a [59] variable x to keep track of the cubes, variable y to keep track of the squares, and n will have to be a variable too this time. So [60] here's how we refine x prime equals n cubed. We strengthen it by conjoining y prime equals n squared. Now [61] we have to refine that. [62] When n equals 0 it's easy. This is looking a lot like Fibonacci. [63] If n isn't 0, we decrease it by 1, and then with a recursive call we find n cubed and n squared for the decreased value of n. Now we have to upgrade x and y to the original value of n using the formulas. First I'll upgrade y because I need n squared to get n cubed. So [64] y gets y, which is currently n minus 1 squared, plus n plus n minus 1. Oh, wait a minute. n was decreased, so it's not plus n plus n. Oh, it's worse than that, because [65] this call has forgotten the value of n. It says what x are y are after the call, but n could be anything. Well, we can fix that by saying [66] n prime equals n. The [67] first refinement is still good. The [68] then-part of this refinement is still good because the two assignments leave n unchanged. But the [69] else-part decreases n. So after the recursive call, which now promises to leave n unchanged, we need to [70] increase it back to its

original value, which fulfills the promise. Now the y upgrade is good, and [71] here's the x upgrade. x gets x, which is n minus 1 cubed, plus 3 n squared, minus 3 n, plus 1.

For timing, just put t gets [72] t plus 1 before the recursive call, and you can see that n goes down just as t goes up, so [73] it takes time n. The proof is very straightforward, so I won't do that right now.

Instead I want to [74] try another solution to this problem using a for-loop, just to get some practice with the for-loop rule. And I'll try using the invariant form. It doesn't always work, but let's see what happens. The solution should be something like [75] this. x starts at 0 because it will accumulate a sum. Then we need an invariant *A* so we can make the form of specification that is refined by a for-loop. And the body of the loop has to be x gets x plus something. We have to define the invariant *A*; we have to figure out what to add to x. And we have to prove the first and last refinement. But we don't have to prove the middle refinement. It's given to us by the for-loop rule. Well it's not hard to see what the invariant is. [76] A k is x equals k cubed. In the [77] top refinement, x gets 0 makes *A* 0 true, because 0 equals 0 cubed. And *A* prime of n is x prime equals n cubed, which is what we want. So the first refinement is satisfied. Now we have to work on the last refinement. [78] I left out k is in 0 to n, but I'll put it back if I find I need it. *A* of k is [79] x equals k cubed, and *A* prime of k plus 1 is x prime equals k plus 1 cubed. Expanding that, we get [80] k cubed plus 3 k squared plus 3 k plus 1. And since the antecedent says that k cubed is x, that's refined by the assignment [81] x gets x plus 3 k squared plus 3 k plus 1. So now we know what the question mark is. Again, 3 times doesn't bother me, but k squared does. So here's what I'm going to do. I'm going to give myself a present. I'm going to strengthen the invariant by saying [82] y equals just the thing we want to add to x. And we [83] have *A* of k, so we can refine by [84] x gets x plus y. Since we've changed the invariant, we have to look at [85] the first refinement again. We have to make *A* 0 true. When k is 0, y has to be [86] 1. And we have to [87] recalculate the last refinement. Expand *A k* and *A* prime of k plus 1. [88] We already expanded k plus 1 cubed, and with the antecedent as context that's x prime equals x plus y. As for y prime, it's equal to 3 k squared, plus 6 k plus 3 more k, so that's 9 k, plus 3 plus 3 plus 1 [89]. Using the context again, that's [90] y plus 6 k plus 6. This can be [91] refined as program that just adds. So I'll [92] put it where it goes, and we can call it done. But I don't like the inelegance of adding 6 k's, so I think I'll [93] strengthen the invariant again, saying z equals 6 k plus 6. And in the [94] bottom refinement, we have *A* k, so I can just say [95] y gets y plus z. I think that's much more elegant. In the [96] top refinement, we have to make *A* 0 true, so I have to add z gets, let's see, when k is 0, z is [97] 6. And we have to redo the [98] bottom refinement. We expand *A k* and *A* prime of k plus 1, and get [99] this. Using the antecedent as context, we can simplify, and dropping the antecedent we get [100] this. Which we refine like [101] this. I fix the bottom refinement [102], and I like this better. For execution, we don't need the specification. The program is just [103] this.

I couldn't write this program without the theory. Why would you ever initialize a variable to 6, and keep adding 6's? I feel like the theory wrote the program. I just decided to use a for-loop, and the theory dictated all the rest. So it's a great example of how the theory doesn't just tell you if you've made a correct or incorrect step. It can show you steps you wouldn't think of. But we shouldn't leave out the specifications, because I don't think anyone could guess that the one line program at the bottom is cubing.

[104] Of course it isn't necessary to use a for-loop. It's never necessary. [105] Here's a program without a for-loop, and including recursive time. Specification [106] Q is clearly related to the invariant we just had. As a C program, it looks like [107] this. If we're given specification Q, it's not hard to prove the refinements. And we get the specification while we're programming. But if you're just given the C program, even if you know it's supposed to cube, it's really hard to think up the intermediate specification and then verify. That's why I like verifying as we program, not afterward.

I have done this problem 4 different ways now, so you should get the idea that there's not just one right answer. But I guess there are a lot more wrong answers than right ones.

From these lectures, you might get the impression that I'm really fast at programming and proving. But I've got a secret: while I'm recording, I've got a pause button, so I can think as long as it takes me, and you don't see how long it takes me. And of course, I have to prepare each of the visuals, and that takes time too. But you have a pause button too, so don't just let the lecture go by in a blur. Pause to think, as long as you need, at each proof step.