[1] The first problem today is fast exponentiation. Given rational variables x and z, and natural variable y, write a program for z prime equals x to the power y. If exponentiation is one of the implemented operations, then this problem is trivial; just 1 assignment. So let's say it's not. It's certainly not one of the hardware operations of the computer, so if it is provided in a programming language, it's only because someone programmed it. And today that someone is us. The statement of the problem says fast, but it doesn't say how fast. We're going to write the fastest exponentiation ever written. But it's just to a natural power, not to a rational or real power. So it's not as complicated as exponentiation to a real-valued exponent. [2] We already have the specification. And since we don't have exponentiation, about the only possibility is to accumulate a product — x times x times x as many times as necessary. And if you're going to accumulate a product, you start [3] the accumulator at 1. And then the remaining problem [4] is – the final product, z prime, equals the product so far, z, times the remaining factors, x to the power y. And let's do this properly, with [5] proof. Use the [6] substitution law, and replace z with 1 in the following expression, and [7] 1 is the identity for multiplication, and that's why we start at 1. [8] Now we have to [9] refine z prime equals z times x to the power y. An easy way to do that is to [10] test if y equals 0, because if it is, x to the y is 1, so we just want [11] z prime equal z. And if it isn't, [12] we'll just make a note of that. The [13] proof of the first case goes like this. [14] Expand ok. Now get rid of the two conjuncts that aren't helping, so that's [15] specialization and makes an implication on the left margin, and throw in a times 1, which doesn't change anything. [16] Change the 1 into x to the power 0. Now use the [17] context y equals 0 to change the 0 into y, and throw away y equals 0. The other case, [18] the else-part, is so direct I won't write it. Now we [19] have to refine the else-part. y tells the number of factors of x still to be multiplied in, and y is greater than 0, so the easiest way is [20] to multiply in one more factor of x, [21] decrease y by 1, and then [22] finish the job. [23] Here's the proof. First, let's get all the antecedents on the same side [24]. Now we use the [25] substitution law twice. Now in the antecedent we have z prime equals z times x times x to the power y minus 1. So we can [26] simplify that. And now we [27] specialize, and we're done. Well, we're [28] done in the sense that there are no unsolved problems, and we can get it executed, and get the answer. But we're not done in the sense that it isn't yet the world's fastest exponentiation program. The time is basically the time to count y down to 0, so if we can do that faster, we win. And when y is even, there's a way to cut it in half. So the first improvement is to modify that last refinement, by [29] dividing into even and odd cases. Given that y is greater than 0, if it's even, then it's even and greater than 0. Else it's odd and greater than 0, but I don't need to say greater than 0, because 0 is even. Now we have 2 specifications to refine. First the even case. [30] We square x, and divide y by 2, and then finish the job. Well, we're not allowed to square, so we multiply x by itself. Here's the [31] proof. Start with [32] portation to get the antecedents together. Now use the [33] substitution law in the bracketed part, so it becomes z prime equals z times x squared to the power y by 2. And we can [34] simplify that to just x to the power y, and then [35] by specialization we're done. [36] What's interesting to me is that this refinement doesn't change z at all. It doesn't accumulate any more of the product towards the answer. But it speeds up the computation a lot by making a big decrease in y. In the [37] odd case, I can't think of anything better than what we had before, and it has the same proof as before. Well, again we're done in the sense that there are no unsolved problems, and the program is correct and works, but still we can do better. Look [38] at this even case. If y is even and greater than 0, then it's at least 2. So when we cut it in half, it's at least 1. Let's not throw that information away. Following y gets y by 2 [39] we know y is greater than 0. And we've already solved that one, so again it's done, and better. Look [40] at the odd case. If it's odd and we subtract 1, then it's even. So let's not throw that away. [41] Have we already – no, that's not one we've refined. All right, [42] we have to refine it. One way is to [43] test for 0, because if it

is 0, that's just ok, and if it isn't 0 that's a problem we've already solved. Again it's done, but there's still one more improvement we can make. That first [44] test for y equals 0 divides execution into 2 cases, but the two cases are nowhere near equiprobable. It's far more likely that y isn't 0 than that it is. To get the best execution time on average, it's best to divide into cases that have more equal probability. So I'm changing this refinement [45] to test for evenness. And the 2 new problems are already solved, so there's nothing more to do. I can't think of anything more. This is now the world's best exponentiation program. I'll do the timing part in a minute, but I want to make some comments about this example first.

[46] Here's a flow chart of the final program. The first time we had completed the program, it was very simple; just a loop. I should show you a series of flowcharts to go with the series of modifications we made, but I don't want to draw them. Maybe you could draw them. The first modification was to introduce an if which tests for evenness inside the loop, with different actions for each case. The next modification changed one of those actions into an inner loop. The next change was a dandy; it turned the loops inside out, making the outer loop into the inner loop and vice versa, with an exit from the middle of the loop. And the last modification, at the top of this flowchart, changed the loop entry into a choice of 2 entry points, one of them entering into the middle of the loop. Programmers modify programs all the time, sometimes to fix bugs, sometimes to add features, sometimes to make them run faster like we did. And it's well known that program modification is very error prone. Even if you are fixing bugs, you are likely to introduce more. And you would never make the modifications we did, turning loops inside out and branching into the middle of a loop. That's if you weren't using the theory. But using the theory, each modification was easy. That's because all the information you need in order to make modifications safely is in the specifications. Specifications are the ultimate comments. And the proof of the refinements is the guarantee of correctness.

Now on to the timing. The rule of recursive time is that every loop must have a time increase of at least 1 time unit. Looking at the flow chart, we see that we can get away with a single increase inside the inner loop, just after x is squared and y is cut in half, because that's also inside the outer loop. In the program, that's [47] here. But we have to change the specifications to talk about time. And if it's not clear what they should be, we can try running the program with the time variable in it, and printing out some timings. [48] Here's what I got. Once again it's clearly logarithmic. Just [49] look at where time increases – at 1, 2, 4, 8, and 16. This time it's the floor of the log. But there's [50] one point that doesn't fit the pattern. The log of 1 is 0. The log of 0 isn't well defined. So [51] here's my timing specification. If y equals 0 then t prime equals t else t prime equals t plus floor log y fi. That's recursive time exactly, but it's easier to prove [52] that time is bounded above by log y, without the floor function. I'll leave the proofs to you because I want to move on to our next great example

[53] the fibonacci numbers. Here's one way to define them. Fibonacci of 0 and 1 are given, and after that, each fibonacci number is the sum of the previous 2. [54] Here's another way to define it. 0 maps to 0, 1 maps to 1, and then n maps to fib of n minus 2 plus fib of n minus 1. Or [55] you can use an if-then-else-fi if you prefer. Each of these definitions is very close to being a program in some programming language. Maybe some minor syntactic differences. But the execution time is exponential, and [56] we can do much better, even without functions. Since a fibonacci number is defined as the sum of the two previous numbers, the idea is to keep track of a pair of adjacent fibonacci numbers. [57] So if the problem is x prime equals fib of n, we refine it by [58] saying we'll do more than that. We'll also make y prime be fib of n plus 1. And let me just call that new problem of finding a pair [59] P so I don't have to write so much. [60] Here's one way to refine P. If n equals 0 then x gets 0 and y gets 1. Else decrease n by 1. Now P will assign x and y the pair of fibonacci numbers for the decreased n, because that's what specification P says it will do. After P, [61]

here's the situation. The f's are fibonacci numbers, and we have a pair or them as the values of x and y. But we want the next pair, so we need to move x and y along one pair. x should be what y is, and y should be the sum of what x and y are. [62] [63] To refine that last specification, we could start with x gets y, but then we've lost the old value of x and we still need it. We could start with y gets x plus y, but then we've lost the old value of y and we still need it. It's just like the problem of swapping the values of 2 variables. We need a temporary variable. And we could declare a new variable for that purpose. But the specification says we care what x and y become, but we don't care what happens to variable n. So we can use it. [64] If I were not using any theory, I would not do that, because I would think that it's likely to cause a problem, and because anyone else looking at the program later will be confused by this assignment to n. But with the theory, you just have to prove the refinements, and you know it's right.

[65] The timing is linear in n. I put t gets t plus 1 just before the recursive call, and the final specification, which is just 3 assignments, doesn't take any time, at least in the recursive measure. Linear time is a lot better than exponential time, but [66] we can do even better than that. [67] Here are a couple of equations that relate a pair of fibonacci numbers to a pair halfway back. One of the exercises at the back of the book is to prove these and other equations about fibonacci numbers by induction. But we'll just use them to write a log time fibonacci program. The program [68] starts out the same as before. If n equals 0 then x gets 0 and y gets 1. Otherwise, it splits into even and odd cases. If n is even then we know it's even and it's not 0. If not, we know it's odd and it's not 0, but we don't have to say that because 0 isn't odd. Now I'm going to refine the [69] second new problem. At the start of this refinement, [70] we know n is odd, so it's equal to 2 k plus 1, for some k. The purpose of the first assignment is to decrease n [71] down to k, roughly half what it was. Then P says that it delivers [72] the fibonacci numbers at this n and n plus 1 as the values of x and y. That's x prime and y prime just before the dot, and it's [73] x and y just after the dot. At the end we want [74] x prime and y prime to be fibonacci of the original n and n plus 1, which is 2 k plus 1 and 2 k plus 2. The job of the last specification is to bridge that gap by using the formulas at the top. x prime, that's fib of 2 k plus 1, equals fib k squared, that's x squared, plus fib k plus 1 squared, that's y squared. And similarly for y prime. — [75] [76] In the even case, we know n is even and greater than 0, so it's [77] equal to 2 k plus 2 for some k at the start of the refinement. So we decrease n down to k. So calling P makes [78] x equal fib k and y equal fib k plus 1. At the end we want [79] x prime and y prime to be fibonacci of the original n and n plus 1, which is 2 k plus 2 and 2 k plus 3. In the last specification, the first conjunct uses the second formula at the top. x prime, that's fib 2 k plus 2, equals 2 x y plus y squared. But we don't have a formula for fib of 2 k plus 3. We get y prime by adding fib of 2 k plus 1 and fib of 2 k plus 2. — [80] In the [81] final 2 refinements, I have again used n as a temporary variable. — To prove the time [82], you just change the specifications to talk about time, and you stick t gets t plus 1 before the recursive calls, same as usual. But one thing that's a bit strange is that the last 2 refinements are refining the same specification. But remember, this isn't the whole specification. The whole specification is the conjunction of the result part and the timing part. And after you put the parts together, the last 2 specifications are different.

It's easy [83] to translate these 5 refinements to any programming language you like, or even one you don't like. Here it is in C. So that's fibonacci in log time. Which is an amazing improvement over the exponential program we had to start with.

The point of this lecture wasn't how to program the fibonacci numbers, or how to write a fast exponentiation program. I was just using them as examples of how to program using a theory of programming, so your programs are proven, and modifications can be made safely.