[1] The first topic this lecture is refinement, and refinement is the main idea in the course. Some people call this whole subject programming by refinement. The next best textbook for this course, after my own, of course, is named refinement calculus. So what is refinement? Specification P is refined by specification S means that whenever S is satisfied, so is P. Formally, [2] that's for all pre- and poststates, P is implied by S. If someone gives you a specification P and says please implement this, you can refine P, which means choosing an equal or stronger specification S, and implement S instead. That's because computer behavior that satisfies S also satisfies P, so the customer will be happy. For example, suppose the variables are x and y, of type integer. And the specification is [3] x prime is greater than x. The customer wants x increased. Yes, I know it's a stupid little example, but it gives the idea. One way to refine that specification is by x prime equals x plus 1 and y prime equals y. You had to decide by how much to increase x, and you decided to increase it by 1. And you had to decide what to do with y, and you decided to leave it alone. Refinement means making such decisions, reducing the nondeterminacy. The new specification is [4] equal to x gets x plus 1, and that is also refinement, because if P equals S, then P is implied by S. The [5] next example is refining x prime less than or equal to x by if x equals 0 then leave x alone, else decrease x, which we could [6] write as a disjunction if we want. And [7] one more example. Make x bigger than y in the end, and make them both bigger than x was to start. That can be refined by y gets x plus 1 followed by x gets y plus 1. For the proof of this refinement, you can [8] rewrite the last assignment, and then use the substitution law, to get [9] x prime equals x plus 2 and y prime equals x plus 1.

[talking head] We've been talking about specifications of computer behavior. Now I want to start talking about programs. A program tells a computer what to do. So a program specifies computer behavior. People often confuse programs with computer behavior. They talk about what a program *does*. But a program doesn't do anything. It just sits there. It's the computer that does something when it executes the program. People sometimes ask whether a program terminates; of course it does; every program is a finite number of lines of code. It's the computer behavior, when executing the program, that may not terminate.

A program is not behavior; it's a specification of behavior. Furthermore, a computer may not behave as specified by a program for a variety of reasons. For example, the compiler may have a bug. Or the hardware could malfunction. Or the execution might run out of memory. If any of those things happen, then obviously the program is not the same as the computer behavior.

A program is a specification of desired computer behavior. And a specification is a binary expression. So that's what a program is. Every program is a specification, but not every specification is a program.

[10] A program is an implemented specification. It's a specification for which an implementation has been provided, so that a computer can execute it. We need only a very few programming notations that are found in most popular programming languages, so I'm sure you know them, or something very like them, already. The first one is [11] ok, which you already know means do nothing. The [12] next is assignment, where the expression e uses only unprimed variables and implemented operators. I'm going to consider that the expressions of the basic theories and basic structures are implemented, so that's [13] binary expressions, numbers, characters, bunches, sets, strings, and lists are implemented. Lists are the most complicated of these, and something very like them, for example arrays, are implemented in all the languages I know. Now in C and Java and lots of languages, you can't just have an assignment as a whole program. You need to say import this, and public class that, or some such blather, but I'm going to ignore all that. Still, you can't assign to a variable if you haven't declared the variable. I'm using the word "program" for what *you* might call a statement in a program. But little programs are put together into bigger programs, including all the declarations and structure and so on. So I'm using the word

"program" at all levels, from the very smallest level like ok or an assignment, to the very biggest level. In mathematics, we might have an expression that's just the number 0, or we might have an expression that's a thousand lines long. We don't call it something else when its size changes. And I want to do that for programs too. The [14] next programming notation is the if-then-else-fi. If b is a binary expression in unprimed variables, whose operators are all implemented, and P and Q are both programs, then – if b then P else Q fi – is a program. Please notice that there is no if without an else. I have 2 good reasons for that. One reason is this is the same if then else fi we met in Binary Theory. It has three operands. It is not the 2-operand implication. The most important reason is that when you write an un-elsed if, there are still 2 cases to prove, but one of them, the else ok case, is very likely to be forgotten. So please, in this course, all ifs have elses. And [15] there's one more programming notation. If P and Q are programs, so is P dot Q. That's it. The whole programming language. As I promised at the start of the course, it took you 30 seconds to learn.

There's also a way of making new programming notation. [16] If you have a specification, and it's implementable, and you refine it by a program, then the specification becomes a program, because now it's implemented. And recursion is allowed means it can even be used in the program that refines it. For example, in one integer variable x, [17] here's an implementable specification. If x is nonnegative, make it 0. That's not a program, yet. But [18] here's a refinement for it. [19] if-then-else-fi is a programming notation, and [20] x equals 0 is an implemented expression, and [21] ok is program, and [22] x gets x minus 1 is program, and [23] sequential composition is program. That just leaves one specification. Saying recursion is allowed means we can [24] count it as program too, so the whole right side is program, and so this [25] specification is now program. That's because we have provided a way of executing it. To execute the specification on the left, just execute the program on the right. And when you encounter the specification again, you again execute the whole program on the right. So that's a loop. First test if x equals 0. If it does, then execute ok, which means there's nothing more to do. If x isn't equal to 0, then execute x gets x minus 1, and then start again. If x equals 0, we're done. If not, decrease x, and start again. And so on.

In that tiny example, we can go from a specification to an implementation in 1 refinement. But in a larger program we can't. So we need some [26] refinement laws. Refinement by steps says – if you have a specification A, and you refine it by an if-then-else-fi as we just did, except that the then-part, C, and the else-part, D, are still not programs, you can [27] refine C and D separately, because the [28] solutions for C and D can be used in the solution for A. All I'm really saying is that if-then-else-fi is monotonic in its then-part and else-part. And you already know that. Similarly, [29] if you refine A by a sequential composition B dot C, you can then refine B and refine C, because A will be refined by them too. In other words, sequential composition is monotonic in both its operands. And easiest of all, [30] if you refine A by B, and then refine B by C, then A is refined by C. In other words, implication is transitive. So that's stepwise refinement.

Another way to break a problem up is [31] refinement by parts. Suppose we have 2 specifications A and E. And we refine each of them by an if-then-else-fi with the same binary condition. Then [32] their conjunction is also refined by that if-then-else-fi using the conjunctions of the then-parts and else-parts. And there's a [33] similar law for sequential composition. If 2 specifications A and D are refined by sequential compositions, then their conjunction is refined by the sequential composition of the conjunctions. And [34] if you refine A by B and C by D, then A and C is refined by B and D. That's just the law of conflation from the back of the book. Refinement by parts means that if you refine 2 specifications the same way, then their conjunction is refined that same way too. And the last one is [35] refinement by cases, and it's just an ordinary binary law too. It says that if you're trying to prove that P is refined by if b then Q else R fi, you can prove it by proving P is implied by b and Q, and also that P is implied by not b and R. That's all the laws we need for programming. From now on we just apply the laws we have.

[36] The first problem I want to apply them to is list summation. It's an easy problem that you could solve with your eyes closed. I think it's best to show how the theory works on an easy problem, and work up to harder problems later. We're given a list of numbers L. It's not a state variable, not a variable in the programmer's sense, because we're not changing L. There's no L prime. And we're given a number variable s. This is a state variable, and we use it to store the answer. The specification is s prime equals the sum of list L. We're going to be adding up numbers. So we [37] initialize s to 0. And we also need to keep track of how many list items have been added, so we [38] give ourselves a natural variable n and initialize it to 0. Now the next bit is the key to programming by refinement. We have to write a specification that describes what's left to be done. What's left? Well, everything, because we haven't really done anything yet. But try to forget that, and write what's left to be done at any time during the computation. [39] We want s in the end, that's s prime, to be the sum so far, that's s, plus the sum of the remaining items, that's the sum from item n to the end of the list. To prove this refinement, use the substitution law from right to left. [40] First substitute 0 for n. Then [41] substitute 0 for s. So the right side is s prime equals 0 plus the sum from 0 to the end, and that's the sum of the whole list, so the right side is the same as the left side. Now [42] we can say we have solved the problem s prime equals sum of L, because we have refined it. But we haven't refined it by a program. Part of it is program, but there's a specification that's not program yet. [43] It still needs to be refined. Now the problem doesn't say anything about s and n being 0. They might be, or they might not be. One possibility is [44] that n equals the length of the list. If it is, [45] then the new problem is the same problem, but with the added knowledge that n equals the length of the list. Otherwise [46] the new problem is the same as the old problem, but with the added knowledge that n is not equal to the length of the list. And that refinement is just [47] a binary law called case creation from the back of the book, so that's an easy proof. In fact, in a nice programming environment, we would just have to write if n equals the length of the list, and it would just fill in the then-part and else-part for us. [48] So that solves the problem, but raises 2 new problems to be solved. Each of the new problems is weaker than the old problem, because it has an added antecedent, and weaker means easier to solve. Choosing [49] the first of the new problems, if n is equal to the length of the list, then from n to the end is an empty segment of the list, so its sum is 0, and we just want s prime equals s. And that's [50] ok. Putting it another way, if n equals the length of the list, execution is done. But we have [51] one more problem to solve. If n is not equal to the length of the list, then there's at least one more item to be added. So we [52] add it to s, and then we [53] increase n because it counts how many items have been added, and then the remaining problem is to [54] add all the rest of the items. Every time we refine, we have to prove the refinement, and the proof of this one is to use the substitution law, going from right to left, so let me write down the right end, [55] there, now we have to replace n by n plus 1, so let's [56] do that, and then we have to replace s by s plus L n, so let's [57] do that. [58] This part can be simplified. It's L n plus the sum of the items from n plus 1 onwards, so that's just the sum of the items [59] from n onwards. And now the right side implies the left side. [60] In solving that last problem, we raised another problem, but not a new one. We've already refined that one. We don't have any unsolved problems, so we're done refining. The fact that we have already refined that last specification means we've created a loop.

There are 2 ways to look at what we've got here. One way is that it's a collection of theorems. I *said* the proofs as we did the refinements, although I didn't write out the proofs nicely. If we had an automated prover, we could just give it to the prover, and let it prove the refinements. Or, better yet, it would be proving them as we write them, and tell us when we make a mistake. The other way to look at it is the way a compiler sees it. To a prover, programs are just a funny way of writing ordinary binary expressions. To a compiler, the program parts are clear enough, but what are those other, non program things? Well, to a compiler, they're just funny identifiers. [61] Here's what a compiler sees. I've just shortened the identifiers to single letters so we won't be distracted by prover information. To a compiler, a refinement is a little procedure, or method, but without any parameters or result or local scope, or anything, so it's simpler than a procedure or method. One thing a compiler would do with this [62] is called inlining, or macro expansion. Replace C by what it's refined by, and replace D by what it's refined by. It can do that because that's exactly what the law of refinement by steps says. You can always replace any specification by what it's refined by. Even B can be replaced here, and some optimizing compilers would do that. It's called unrolling the loop. You can't get rid of B that way, but each unrolling makes execution a tiny bit faster. The most direct translation into the C programming language would be [63] this, and Java would be similar. Method A sets s and n to 0 and then calls B. Actually, I guess we should put B before A so A can call it. Anyway, B tests if n equals the length of the list, which in C is the size of the list divided by the size of an item. The ok just becomes a semicolon. In the else-part, s is increased by a list item, and n is incremented, and then B is called. Some compilers do a miserably poor job of compiling calls. They save registers and other things on a stack, and they push a return address, and maybe they modify some other registers before they branch. But here, and most of the time, that's unnecessary. So [64] here's a translation that avoids all that. s and n are assigned 0. Then we have label B, and the call to B has just become go to B. A good compiler would compile both of these translations the same way, with a simple branch back making a loop. If C or Java is not your programming language, you can translate to whatever programming language you want.

Let's try another example. [65] Binary exponentiation. Given natural variables x and y, that means variables whose values are natural numbers, the problem is – assign to y the value of 2 to the power x. [66] How are we going to refine that? There are many solutions. We could start by [67] testing if x is 0. The [68] then-part is x equals 0 implies y prime equals 2 to the x, and the [69] else-part has the antecedent x not equal to 0, or, since x is natural, that's the same as x greater than 0. So, 2 new problems. We could choose either one first, it doesn't matter which. Taking the [70] first one, if x is equal to 0, then 2 to the x is 1, and it's [71] easy to assign y the value 1. This refinement is correct, but what about [72] this one? The specification says to make y prime equal to 1, but it doesn't say what to do with x. So that means we can do whatever we like with x. The sensible thing to do is leave x alone. And that's what the first assignment y gets 1 does. It's stupid to assign x the value 3, but it's allowed, so I'm going to do it just to mess with you a bit. The other problem [73] was when x is greater than 0. And [74] here's my solution. First, given that x is greater than 0, which we are given, make y be 2 to the x minus 1. Now that's only half of what y should be, so then double it. Now y has the right value. I should do a proper proof, here, but maybe you can see that this is right. If we first make y be half what it should be, and then double it, it will be right. That's 2 new problems, so picking the first one, is given x greater than 0, make y be 2 to the x minus 1. And I'll do that by [75] first decreasing x by 1, and then making y be 2 to the x. That's only 1 new problem, because we've already refined y prime equals 2 to the x. We still have the problem [76] of doubling y, which of course is [77] easy, but the specification doesn't say what should happen to x. So again, [78] I'm going to make a totally superfluous assignment just because I'm allowed to. Obviously I am not writing the best possible program here. The one problem left [79] is decreasing x, which is [80] easy, but I'm [81] sticking y gets 7 on the end. No more problems to solve, so we're done. We have to prove all the refinements, but the proofs are so easy I won't bother right now because I want to make a different point. [82] Here's what a compiler sees. If we take the [83] top line, and then we macro expand, or inline, or stepwise refine, or use monotonicity, or whatever you want to call it, we replace B we get [84] this. Replace C and get [85] this. Replace D and get [86] this. Replace E [87]. And replace F [88]. In the C language I guess we have to declare [89] x and y. And then we can write A as [90] a function quite directly. So we could [91] start x at 5, say, then call A, and then print y. And we're hoping that 32 will be printed. Do you think it will be? Want to bet with me? Suppose someone showed you this C program, and asked what it computes, could you guess? I couldn't. And if they told me it computes 2 to the x, I'd be pretty doubtful. It's really hard to trace the execution of this program. And those assignments of 3 and 5 and 7 look like they could make the computation wrong. The whole point of this example is that you cannot understand it by looking at its execution. I know it works, but not from looking at the C code, and not from executing it. I know it works from [92] proving the refinements. And the proofs are all easy.