[1] This lecture is about the control structures in popular programming languages. We already have sequential execution, and we'll talk a lot about concurrent execution later. We have if-then-else-fi for making a choice. The one kind of control structure we don't have yet is loops. The simplest kind of loop is usually called the while-loop. I'll use this syntax, though your language may have a slightly different syntax. b is a binary expression in unprimed variables, and P is any specification, not necessarily a program. For constructs other than loops, we just equate them to a binary expression in ordinary notations. But for loops, we don't. Instead, we [2] look at what specifications a loop refines. We take this refinement to be an alternative way [3] of writing this refinement. Its execution is exactly the same as the loop execution. If b is true, we execute the body of the loop P, and then we re-execute the loop. If b is false, we're done. And we already know how to prove a refinement like this.

[4] Here's an example. It's a loop for summing the items in a list. while n is not equal to the length of the list, add item n, increase n, and for recursive time we need t gets t plus 1 somewhere in the loop. The specification [5] we want to refine with it says the final sum is the sum so far plus the sum of the remaining items, and the final time is the time so far plus the distance to the end of the list. To prove this refinement, we [6] instead prove this refinement in the usual way. And that's all a programmer needs to know about while loops. There is another way to define the while-loop construct, and we'll meet it in chapter 6.

Some languages have a way to jump out of the middle of a loop. In C and Java it's called break. In other languages it's called [7] exit. This is supposed to be a loop with a conditional exit in the middle. Since it's a loop, we don't say what it's equal to. We say what [8] this refinement means, where L is an implementable specification. And [9] here's what it means. In execution, the [10] first thing that happens is A. The [11] next thing that happens is b is tested, and if it's true, that's the end of execution of the loop. If b is false, execution continues with [12] C, and then the [13] loop is re-executed. Both these refinements have the same execution, so proving the second one, which doesn't have any loop construct in it, is the way to prove the first one.

Sometimes you might want to exit from the middle of several nested loops at once. [14] Here we have a loop within a loop, and inside the inner loop, we have an exit. exit 2 when c means exit 2 levels of loop if condition c is true. We don't give a meaning to these loops, but we do give a meaning to [15] a refinement, where P is refined as this nested loop. The [16] meaning is a refinement of P that starts out with [17] A, because that's the first thing that happens. The [18] next thing that happens is we enter another loop. And for refinement, every loop must have a specification. So we have to invent one. And that is actually the hardest part of proving the correctness of a completed program. It's much easier to prove correctness while you are writing the program. Anyway, the [19] first thing in Q is B. And the [20] second thing tests condition c, and if it's true, then we exit both loops, and we're done with them. If c is false, [21] we continue with D, and then we're at the [22] end of the inner loop, which means we have to start it again. And you can see from the refinements at the bottom, that there's nothing more to refine. — — You might be wondering [23] what about E? Well, there's no way E can be executed. It's dead code. So it doesn't have any place in the refinements at the bottom. For E to be executed, the inner loop would have to have a 1 level exit. So [24] here's an example like that. To turn this into [25] refinements, it starts with [26] A, and then there's a [27] test of b, and if it's true we exit the outer loop and we're done, and if it's false, we execute [28] C, and then there's another loop, so we need a specification for it, let's say [29] Q. Within Q, the first thing is [30] D. Then we test [31] e, and if it's true, we exit 2 loops, and we're done. If not, we [32] execute F. And then we test [33] g. And if it's true, we exit 1 loop, so the next thing is [34] I, and then we come to the [35] end of the outer loop, which means we have to re-execute P. If g wasn't true, we do [36] H, and then we come to the [37] end of the inner loop, which means we re-execute Q. Now

we have 2 refinements to prove, and that's how we prove that this nested loop with 1 and 2 level exits refines its specification. All other looping constructs can be treated exactly the same way.

[38] Two dimensional search is a good example. If you are searching for an item in a 2 dimensional array, there are 2 nested loops. And when you find the item, you want to jump out of both loops. [39] Here's a specification and a picture. Array A is n by m, and we're looking for item x. If x is anywhere in the array, then the final values of variables i and j will tell us where. If x isn't anywhere in A, then the final value of i will be n and the final value of j will be m. Array A, dimensions n and m, and item x are all constants. i and j are variables. And we'll call that specification P. We [40] refine P by assigning 0 to i. And then we have a new problem. [41] Specification Q describes the search when we're part way through, when we've searched the first i rows and didn't find x. So Q says look from row i onwards. It's identical to P except change that first 0 to i. And initially, i is 0 so that means we still have to search the whole array. The [42] problem we need to refine now is: given that i is less than or equal to n, search from row i onwards. [43] We have to check if i equals n, because if it is, we searched everywhere, and there's no unsearched area left. So we know x isn't anywhere, and we want i prime to be n and j prime to be m. Well i is already n, so we just need the assignment j gets m and we're done. [44] If i isn't n, then there are still rows to be searched. Given i is less than n, search from row i onwards. [45] Now we need to refine that. So we [46] assign 0 to j, and the remaining problem is: given i is less than n and j is less than or equal to m, do R. [47] R describes the search when we've searched the first i rows and the first j columns within row i. It says: if x is in array A [48] in row i from column j to the end of the row, [49] or in row i plus 1 onwards, any column, then report where, and if it isn't, report that. So [50] now we're searching in the white area of the picture. We [51] check if j equals m. If it does, then we have finished row i. We increase i, and now we no longer know that i is less than n, just less than or equal to n, and we want to search from row i onwards, and that's a problem we've already solved. [52] If j is not equal to m, then we know that j is less than m. [53] Finally we know that both i and j are indexes in the array. [54] We refine this problem by checking if A i j is the item we're looking for. If it is, we're done. [55] If it isn't, we increase j by 1, so we still know i is less than n, but all we know about j is that it's less than or equal to m. And that's a problem we've solved before. We're completely finished.

For the [56] timing, we make the same refinements, except for 2 things. We change the specifications to talk about time, and we [57] put t gets t plus 1 somewhere. We could put it before each recursive call, but I've been a bit clever and put a single increment here, because here it's inside both loops at the same time. The [58] execution time is bounded above by n times m, which is the area of the array. It may be less than that, because execution finishes if the item is found. For the [59] other specifications, it's always the remaining search area, which is n minus i times m for the remaining rows, and [60] that minus j, when part of row i has been searched.

If I write [61] a simple identifier for each specification, the program looks like this. And that helps for [62] compiling it to an execution language. In C it looks like this. Remember that C is my object language, not my programming language. A friend of mine was writing a C textbook a few years ago, and I said – why spend time on such a bad programming language. And he said that I should think of C as the world's best assembly language. And so it is. Notice that we don't have a binary variable, usually called found, that we set to false to start with, and to true when the item is found, and test at each iteration of each loop.

The [63] for-loop is a kind of loop construct in some popular programming languages. The syntax I'm going to use looks like this. [64] the name is treated as a state constant, not a variable; you can't assign to it inside the loop body. [65] m and n are integer

expressions such that m is less than or equal to n, and it's their initial values that control the iteration. i starts at m and goes up to but not including n, [66] so the number of iterations is n minus m. The [67] body P doesn't have to be a program. It can be a specification that still has to be refined. [68] As with all other loop constructs, we don't have a direct meaning for the for-loop. We need a specification, [69] and since the for-loop is indexed, so is the specification. Specification F i describes the computation from index i to the end. So [70] F i is true if index i is in m to n, and to execute the iterations from i to the end, first execute one iteration P, then all the iterations from i plus 1 to the end. And [71] at the end, there is nothing more to do. [72] The for-loop refines F m, which describes its computation from the start to the end. We don't have to prove this refinement, or rather, we prove it by proving the other two refinements.

Let's do an [73] example. The specification x prime equals 2 to the power n is not in the right form for a for-loop. Define [74] F i as x prime equals x times 2 to the n minus i, which says that the final product is the product so far times the remaining factors. Now we can [75] refine the problem by initializing x to 1, and then F 0 does the job. We have to [76] prove this refinement, which is quick and easy. Just use the substitution law and replace x by 1, then simplify 1 times 2 to the n minus 0. Now [77] we refine F 0 with a for-loop. The starting index is always what F is applied to. The for-loop rule says that to prove this refinement we [78] have to prove these two things. So let's start with [79] the first one. And [80] here's the proof. Just substitution law, simplify, and specialize. [81] and now [82] the other thing we have to prove, and [83] there's very little to that proof, and we're done. It's a very small example, but it shows how the for-loop rule works.

For the [84] timing of a for-loop, suppose iteration i takes time G of i. Then the time for the whole loop is the sum of the times of each iteration. We can prove this exactly the same way we proved the previous example. Define [85] F i to say that the final time is the time so far plus the sum of the times of the remaining iterations, from i to the end. [86] This is the right kind of specification to be refined by a for-loop. Here's [87] what we have to prove. In the first one, filling in [88] F i and P and F i plus 1, over on the right side, you see that G i gets added to the time, and then G i plus 1 onward. Putting those together, we get that G i onward is added to the time, and that's the left side. The other thing we have to prove [89] is that if you leave t alone, then you add nothing more to t. So that's the timing. An important special case is [90] when each iteration takes the same length of time. Then the time for the loop is the number of iterations times the time for one iteration.

[91] Here's another example. Add 1 to each item in a list. The main use of for-loops is to do something to every item in a list. It doesn't matter what that something is. In this example, it's just adding 1, but even if it were something complicated, that wouldn't make the for-loop reasoning any more complicated. [92] Here's the specification. It says, without changing the length of the list, make each item have the desired value. Now we need an indexed specification to describe what's left to be done at any iteration. [93] This says, without changing the length of the list, leave the items alone that we've already done, and make the rest of the items have the right value. [94] The problem we are given is F 0, and we can refine F 0 with a for-loop. What we have to prove [95] is these two refinements. Proving is always work, but these proofs are not hard work, so I'll leave the example here. By the way, if you get F i wrong, you find out when you do the proofs, and you even find out what's wrong so you can correct it. I'm likely to get the part that says the segment from i to the end is increased, but I'm likely to forget the part that says leave the segment from 0 to i alone. And then when I try to prove the F i refinement, I'll discover the error. So it's self correcting.

[96] We have already seen how to prove things about for-loops. This isn't a new rule here; it's just a special case of the rule we already have. It's the special case when F i is an implication, and the antecedent has unprimed variables in it, and the consequent is exactly

the same as the antecedent except that the variables have primes. A i is an assertion, which means it's a binary expression about a single state. This particular way of using an assertion, in which an assertion about a prestate implies the very same assertion about a poststate, is called an invariant. The bad thing about this special case is that most of the time specifications just aren't this form. The example we just did, which was to do something to every item in a list, isn't this form, and that accounts for most of the for-loops. The good thing about this special case is there's nothing to prove, because [97] the two things we should be proving are theorems no matter what A is. So if the specification has this invariant form, there's nothing to prove. Now we need an [98] example, and I'm going to use exponentiation again. The [99] invariant, A of i, is x equals 2 to the power i. And it's the [100] first refinement that puts the problem in the right form. Set x to 1, and then, given x equals 2 to the power 0, make x prime be 2 to the power n. [101] Then we refine that with a for-loop. No thought there. Just fill in the 0 to n, and the body is always the same. And we have to [102] refine that, and we do so by doubling x. The first and last of these 3 refinements have to be proven, but I'm not going to do that right now. The middle refinement doesn't have to be proven. It's a gift.

[103] I want to do a more interesting example. Given a list L of integers, possibly including negatives, write a program to find the minimum sum of any nonempty segment. [104] Here's an example list, and it looks to me like [105] this segment is the one whose sum is minimum. Actually, we're not being asked to find the segment whose sum is minimum. Just the minimum sum, so that's minus 10. [106] The specification is s prime equals the minimum, as i and j vary, of the sum of L from i to j. I'm going to use a for-loop to solve it; that's why it's in this part of this lecture. It's not always possible to use a for-loop. And even when it is, a for-loop is not always the best solution. And I'm going to try to use the invariant form of the for-loop rule. So I have to find an [107] invariant. Picking an arbitrary index k, I define A k to say that s is the minimum sum of all nonempty segments that end before or at k. So it's the same as the main specification, but just replace length of L with k. There aren't any nonempty segments ending before or at index 0, and the minimum of an empty bunch of segments is infinity, so we can [108] refine by setting s to infinity, which satisfies A 0, and we want A prime of the length of L. Now the specification is the right kind for a for-loop. The proof of this refinement is very easy – just one substitution and some simplification. Now we get the [109] gift. We don't have to prove this one. But we do have to [110] refine the body of the loop. We have to go from k to k plus 1. s is already the minimum of all nonempty segments so far, so we just need to look at the new segments. Those are the nonempty segments ending at k plus 1. [111] Each one of them is a one-item extension of a segment ending at k, except for the segment that's just item k. In the example, the minimum sum of all the new segments that are extensions is 7 more than the minimum sum of all the nonempty segments ending at k. That would be easy if we just knew the minimum sum of the nonempty segments ending at k. So let's just add that to the [112] invariant. The new bit says c is the minimum sum of the nonempty segments ending at k. Now we have to go back and revise the first refinement [113] by assigning c the value infinity, because there aren't any nonempty segment ending at 0. Now it's easy to [114] extend the segments from k to k plus 1. The minimum sum of the extensions is c plus L k. We take the minimum of that and the one new segment that isn't an extension to get the minimum sum of all nonempty segments ending at k plus 1. And we can simplify that by [115] by factoring out L k. And finally, [116] the new value of s is the minimum of its old value and c. And we're done. Even though there are something like n squared over 2 segments, where n is the length of the list, we have a linear time program to find the minimum sum. That's pretty nice.