[talking head] So far, we have a very small set of programming notations. We have ok, for doing nothing. We have assignment, for changing the value of a variable. We have if-then-else-fi, for making a choice. And we have dependent composition, for sequencing. In a theoretical sense, that's all you need; you can program anything with just those notations. But in a practical sense, it would be convenient to have a few more features in our programming language. I said a *few* more. Currently popular languages, like Java, have so many features that it's hard or impossible to remember them all and understand them all formally. Programming becomes like shopping. You go hunting for the right feature that solves your problem. Maybe you don't find exactly the right feature, so you settle for one that seems near enough. I guess it's a kind of programming. But the only reason you can program that way is that someone else implemented all those features, and that was the real programming.

In the next few lectures, I want to show you how to reason about some of the most common and useful features of programming languages. The first one is local variable declaration. This feature is so useful that every decent programming language includes it.

[1] I'll use this syntax, which may be different from what you are used to, but the syntax is not the real issue here. This [2] declares local state variable x with type T and scope P. The type is just the bunch of values you can assign to the variable. And the scope is the part of the program where we have this new variable. Except that P doesn't have to be a program. It can be a specification that isn't refined yet. And that's important for this feature and every language feature you might define, because you have to be able to use a feature, and prove that you have used it correctly, even before you refine any new specifications. Variable declaration is really just [3] existential quantification, with 2 mathematical variables x and x prime. Let's look at an [4] example. Suppose the nonlocal variables are y and z. var x colon int. Then x gets 2, and y gets z plus z. [5] First we get rid of the programming notations. var is existential quantification over x and x prime. And the 2 assignments say the final value of x is 2, the final value of y is 2 plus z, and z is unchanged. We can get rid of exists x because x just doesn't occur in the body. And we can get rid of exists x prime by a one point law [6] leaving y prime equals 2 plus z, and z prime equals z. x and x prime are eliminated, and that's reasonable because any expression talks about its nonlocal variables.

[7] Here's an example in which the local variable is used but not initialized. Removing the programming notations [8] we get exists x and x prime in int such that x is unchanged, y prime equals x, and z is unchanged. We get rid of the quantifications by [9] one point, and we're left with z prime equals z. x isn't mentioned because it was local, and y isn't mentioned because its final value is unknown. This kind of declaration gives us a local variable whose initial value is arbitrary, and presumably it's the garbage left in that storage location from its previous use. But we can still use it. For example, [10] var x and then y gets x minus x works out to be [11] y prime equals 0 and z is unchanged. [12] In some languages, a local variable is automatically initialized to a special value called the undefined value. For that kind of declaration, we just need to [13] invent a new value, which we might as well call undefined, and define var like this. What makes this value undefined is not that we call it undefined, but that we don't give any axioms about it, so we can't prove anything about it. So if you want to prove something about the result of a computation, it better not use undefined.

Some languages have an [14] initializing declaration like this, where you get to choose the initial value. That's defined like [15] this. The initializing value is the only domain element for x.

Any kind of declaration increases the number of variables. Locally, we have all the nonlocal variables we already had, plus the new local ones. [16] Sometimes we need to make a local reduction in the number of variables. That's what the frame notation is for. If

the state variables are w, x, y, and z, then frame w, x reduces the state variables to just w and x. [17] You can still refer to y and z, but locally they are state constants. There's no y prime, and no z prime. The frame notation means [18] the same as its body, and all the variables not listed in the frame are unchanged. When I defined assignment, I used the informal 3 dots to say “and all other variables are unchanged”. If I had defined frame first, [19] I could have defined assignment formally like this. Likewise [20] ok could be defined formally with an empty frame.

The problem [21] of summing a list could be expressed like this. It says we want s to be assigned the sum, and that means we want all other variables to be unchanged. But we need an index variable to count through the items in the list. So we first [22] narrow the frame to s, which makes sure nothing but s will be changed. Then we [23] declare a local variable n to index through the list. And the problem is now [24] s prime equals the sum, which allows n to change. And we [25] need to refine that, but we've done it before, so we won't repeat it here.

The [26] next language feature I want to talk about is the array. An array is a collection of state variables. You can assign a value to an element of an array with a notation like this. Or maybe your favorite language uses [27] square brackets. I'm using round brackets for precedence, and square brackets for lists, so [28] let's just get rid of the brackets. Array element assignment A i gets e [29] means that [30] A afterward at element i has the value e, and [31] all the other elements of the array are unchanged, and [32] all other variables are unchanged. I should be using the frame notation to say this. [33] The trouble with array element assignment is that the substitution law doesn't work. I'll show you. [34] Look at this example. There are 3 assignments, and then a binary expression that compares A i with A2. If you look back, you see that i has value 2. So we're testing whether A2 equals A2, and we should get true. [35] Looks like we should be able to apply the substitution law here. We [36] can't because it's an array element assignment, but [37] let's see what happens if we do. This [38] time we can use it, and we get [39] this. And [40] here's another wrong use, and we get [41] 4 equals 3, which is [42] false, and that's the [43] wrong answer. [44] Here's another example. The first assignment sets A2 to 2. So the second assignment is really A2 gets 3. So the binary expression at the end is really 3 equals 2, which is false. If we [45] use the substitution law, there's no A of A of 2 to replace, so it [46] stays A2 equals 2, And now there is an A2 to replace, so [47] it becomes 2 equals 2, which is [48] true, and that's the [49] wrong answer. So you can't use the [50] substitution law on array element assignment. But here's what you *can* do. You can rewrite array element assignment like [51] this. It says the final value of array A is a list that maps index i to the value e, and otherwise it's the same as it was, and all other variables are the same as they were. And that's the same as [52] an assignment to A. It says A gets a new value, which is pretty much the same as the old value, except that index i has value e. And now it's an assignment with a single identifier on the left, and that's the kind for which the substitution law works. So let's try those examples again. The [53] first one was this. This time, we rewrite the array element assignments like [54] this. (A2 gets 3) becomes (A gets (2 maps to 3) otherwise A). And (A i gets 4) becomes (A gets (i maps to 4) otherwise A). Now we [55] can use the substitution law. We have to replace A. Both occurrences. [56] Here's what we get. On the left side of the equal sign we have i arrow 4 otherwise A, and it's indexed by i, so we could simplify it to 4. Or, we could leave it, and [57] apply the substitution law again, replacing 3 occurrences of i with 2, and we get [58] this. Now the 2 sides of the equal sign are the same, so I [59] simplify it to true. The [60] last substitution doesn't change anything, and we get true, which is the [61] right answer. Now [62] for the [63] other example. We rewrite the array element assignments [64]. Now we can use the [65] substitution law. There's no simplification possible here. So we use the substitution law again, replacing both As [66]. Now look at [67] this part. It simplifies to [68] 2. Now the left side starts off with 2 mapping to 3, and it's

indexed by 2, so it simplifies to [69] 3. And 3 equals 2 is [70] false. And that's the [71] right answer. So for [72] arrays, the only thing to [73] remember is: change all array element assignments into array assignments before you do anything else. If it's a 2 dimensional array, [74] change it also, and so on for more dimensions, but you'll have to look up that stuff at the end of chapter 2.

[75] The other standard data structure found in most popular languages is the record. At least that's what it's called in Pascal, Ada, Modula, and Turing. In Oberon, Eiffel, and Java it's a special case of class. In C it's called a structure, and the keyword is struct. It's like an array in that it has elements, which are called fields, but they can be of different types, and they are not indexed by a number but by an identifier, which is called a tag or field name. [76] Here's what we've already got. For the tag I'm using texts, or character strings, name and age. This is a function space in which name maps to a text, and age maps to a natural number. And [77] here's a variable declaration creating variable p of type person. And [78] here's an assignment to p. This assigns an entire record to p, and it doesn't cause any problems. But all these languages with records or structures always have a way of assigning to a single field, like [79] this. This assigns 18 to the age field of record p. And that's the problem. When an assignment has anything other than a single identifier on the left, the substitution law doesn't work. The solution to the problem is exactly the same here as it was for arrays. Just change the assignment to [80] this. The new value of p maps age to 18, and all other fields stay as they are.

The array and record are the usual basic data structures for most languages. More complicated structures can be built with pointers, and I'll talk about them later in the chapter on recursive definition.