[1] In this lecture I continue with the formal treatment of features that are common to many programming languages. And the first one this lecture is the value expression. It's a way of using ifs and loops and all the state changing features and flow of control in order to express some value. It may not look like this in your favorite programming language, but I want to show it in its simplest form. P is a program, or maybe a specification that still hasn't been refined, and e is an expression. If P is a program and e uses only implemented operators, we can consider this to be an implemented expression for use in programs. Its meaning is roughly [2] execute P and then evaluate e. But [3] don't change the state. Here's [4] an example of its use. P starts by declaring two local variables, and initializing them. Then there's a loop. Then the result is the value of one of the variables. By the way, it computes the base of the natural logarithms, which we usually call e. [5] Here's its axiom. After executing P, the value expression is equal to e. [6] But, don't double-prime the value expression, and don't substitute in it. [7] Letting P be the assignment x gets x plus 1, and letting e be x, the value axiom says this. Now we can use the substitution law, [8] but the substitution just replaces the final x. Writing x gets x plus 1 value x is exactly the same as writing x plus 1. [9] So if you [10] use it, say in an assignment, it does not say that x is incremented. --

[11] How do you execute a value expression without changing the state? [12] Here's the answer. The implementation has to introduce a fresh local variable for each nonlocal variable that gets assigned. Then all changes that would have been made to the state are instead made to local variables that disappear at the end. It's an easy implementation. But [13] the popular programming languages fail to introduce local variables, so evaluating an expression can cause the state to change. Some people even consider it a good feature. It's called [14] side effects. The reason it's a terrible feature is that you have to throw away all of mathematics if there are side effects. [15] You can't even be sure that x equals x, [16] because the first evaluation may change the state so the second evaluation gives a different answer. [17] You cannot do any arithmetic, because [18] the laws don't hold. If the programming language has expression with side effects, and you want to reason about the program, then [19] you have turn the side effects into main effects. The assignment x gets a value expression, which says variable y doesn't change, has to be replaced by P followed by x gets e, because P might change y. It's a bit more complicated than that in [20] this example. Is [21] this supposed to be the value of y before or after P is executed? Addition is symmetric, so we could have written the operands of addition the other way round. I'll suppose it's the value of y before P is executed, so we have to [22] declare a local variable to remember the value y had initially before P is executed, then [23] use the local variable in place of y for the result. My main message here is just that side effects cause problems, and you have to be careful. It's better to never have side effects.

In the timing, I have tended to neglect the time for expression evaluation, which is reasonable for simple expressions. But if expressions can have loops in them [24], then it's not reasonable, so with value expressions you have to start accounting for expression evaluation time.

Next [25] I want to talk about a feature of all languages that I know, namely the function, or it might be called a method. Here's an example in C, and it's probably not too different in your favorite language. This function computes the binary exponential, though not very efficiently. A C function starts with [26] an assertion about the result, namely its type, int. Then there's the [27] name of the function. Then [28] the parameters, then [29] the curly brackets delimit the scope of the local declarations. And then there's a [30] value expression, except that C uses the word return. Everything I said about side effects applies here. In the notations of this course, it looks like [31] this. It doesn't look very different. The real difference is that in C, and other programming languages, the function is a package deal. If you want one feature, you get them all. If I don't want [32] to name my function, I

shouldn't have to. It's still a function, and can still be applied to an argument. And there are lots of things I want to name besides functions, so [33] we need a general naming notation, not just one for functions. Similarly if I [34] don't want to make an assertion about the result, I shouldn't have to, and [35] there are lots of places I might want to make assertions, not just about the result of a function, and there are lots of things to assert besides the type of the result. In C, functions are the only place you can make local declarations, except for the very start of the program. You can't just put a declaration anywhere you want it. But you should be able to. I have explained each of these features by itself, and the formalization is not too hard to understand. But the C function is far too complicated to have an understandable, and usable, formalization, except by decoupling the features and reasoning about them separately. Other languages have even more complicated beasts. Their designers didn't care about mathematical reasoning, and proof, and that makes life difficult for people who do care.

In some languages there's a [36] procedure, which is simpler than a function, but still a bit of a package deal. The main thing is to be able to parameterize a program. Not just a program, but any specification, even one that has not been refined to a program. For example, [37] procedure P here has an integer parameter, and the body says that the final values of variables *a* and b should be on the two sides of the parameter. Of course we have to refine this specification, but even before we refine it, we can already use the procedure. We can say that [38] P of 3 puts *a* and b on the two sides of 3. And [39] P of *a* plus 1 puts the final values of *a* and b on the two sides of the initial value of *a* plus 1. [40] Here's a refinement. I have chosen to make *a* 1 less, and b 1 greater than the parameter value. But this refinement is completely irrelevant to the use of the procedure. Wherever the procedure is used, it's the specification we use. That way the users don't have to know the implementation, and the implementer doesn't have to know the uses.

[41] Here's an equation that says that a procedure with an argument is the same as having a local variable that's initialized to the argument value. And in some programming languages, a parameter is just a local variable that gets initialized when the procedure is called. But look at the side condition. They're equal only if you don't make any assignment to the parameter within the procedure body. As soon as you do assign to the parameter in the procedure body, it's not being a parameter any more. It's just being a local variable, which is part of the implementation of the procedure. So that's another mistake in language design made by people who don't know any theory of programming.

[42] The kind of parameter we have been talking about is called a value parameter, because it stands for some value that will be supplied as argument. There's another kind of parameter called a variable parameter. It used to be called a reference parameter, or a var parameter. This kind of parameter stands for a variable that will be supplied as argument. I'll use [43] the keyword new to mean it's a variable parameter. This is a procedure with variable parameter x of type int. A variable parameter should be assigned within the body of the procedure, because that's the point of it. This procedure assigns 3 to nonlocal variable *a*, then 4 to nonlocal variable b, then 5 to variable parameter x. The argument [44] can't be an expression like 5 or *a* plus 1. It has to be just a variable, like *a*. And this means [45] *a* gets 3 and then b gets 4, and then *a* gets 5, which is [46] *a* prime equals 5 and b prime equals 4. A problem with variable parameters is that the procedure body has to be a program, with no other specification notations. If we [47] write this procedure body as *a* prime equals 3 and b prime equals 4 and x prime equals 5, we can't get the right result by substituting *a* for x, or even *a* prime for x prime. It just doesn't work for arbitrary specifications. Worse than that, even if you stick to programs, you can't even reason about the procedure body. The assignments appear to be to 3 separate variables, so [48] I should be able to reorder them. But that gives a different answer. So variable parameters carry a [49] warning label. Use them only for programs, don't manipulate the procedure body. Substitute arguments for

parameters before doing anything else. So that means you have to apply programming theory separately for each call. You can't reason about the procedure by itself. You have to know how it's called. And that, in my opinion, defeats the whole purpose of having a procedure.

[50] Here's a standard sort of picture people draw to represent the state of a computation. Variable i has value 2, and r is a variable parameter that stands for variable i. p is a pointer variable that points to array element 1 at the moment. As you can see, a storage location may have no name, it may have one name, it may have more than one name. When a storage location has more than one name, the names are called aliases. And aliases are a problem for reasoning. We've seen that the substitution law doesn't work if you have array elements that can be individually assigned, and we just saw that variable parameters are a problem, and so are pointer variables.

There are really two mappings here. [51] One is a mapping from names to storage locations, and the other is a mapping from locations to values. A single assignment can change both mapping at once. If we change the value of variable i, we not only change the right hand mapping, we also move the name A of i to a different location. We can certainly invent a theory that can cope with aliases, but it would be a more complicated and less usable theory. So here's an alternative. Let's simplify the picture, by [52] eliminating the storage cells. Obviously we're not going to eliminate them from the implementation of a programming language, but our understanding of programs shouldn't be so tied to the implementation. We already saw that we solve the array element assignment problem by talking about the value of the array as a whole. We can solve the pointer problem by using an array index instead. If a variable is used only to index a single array, it can be implemented as an address for efficiency. As for the variable parameter, it can be replaced by having functions that can return data structures as values, rather than just single numbers. I haven't shown that on the picture. But the point is that we can simplify the picture by raising our level of understanding from the implementation, which reads and writes memory, to the theory, which applies laws and proves correctness. If having a good, usable theory were a design criterion for programming languages, they would be a lot simpler and easier to use.