[1] Functions. Here's what a function looks like. A function introduces a new local variable, or you could call it a parameter if you want. And that's really the one and only purpose of a function. The angle brackets show the scope of that local variable, so we could call them scope brackets. D is the domain of the function, or you could say the domain of the variable, or the type of the variable. Variables are for substitution, and the domain says what expressions can be substituted for the variable. The body is any expression whatever. The only thing different about it is that it can make use of the new local variable. All the normal rules of proof apply within the body, plus [2] one extra rule, namely v colon D, which says the variable is in the domain. [3] Here's an example function, and a picture of it. It introduces local variable n with domain nat, and the body or result is n plus 1. In the picture, the domain is the left column, and the result is the right column, and the arrows are saying that each domain element is mapped to its successor. This is called the successor function. Now it doesn't really matter what name you choose for the variable. [4] If you rename the variable to some fresh name, it's an equivalent function. [5] There's a domain operator that gives the domain of a function, and a size operator, which gives the size of a function, which really means the size of the domain. So [6] if we apply domain to the successor function, we get nat, and if we apply size we get infinity. A function can be [7] applied to any element of its domain. We say f applied to x or [8] just f of x. x is called the argument or operand. But please [9] don't write f x without a space between them because that looks like a single two-letter name. Some people like to put [10] parentheses around the argument, and you can if you like. You could [11] put them around the function if you like. But they're not necessary unless [12] the precedence requires them.

A function is really just an operator, like [13] minus, or [14] negation, and we don't put parentheses around the operand there, although we could. When we [15] apply the successor function to 3 we [16] substitute 3 for n in the body, [17] and get 4. A function of [18] 2 variables is really just a function of 1 variable whose body is a function of 1 variable. You can [19] see that average here is a function by the scope brackets and raised dot, and its [20] variable is x, and its [21] domain is xrat. And its [22] body is a function of 1 variable. [23] So we can apply average to an argument, [24] say 3, and since the result is a function, we can [25] apply it to an argument, say 5, and [26] find out that the average of 3 and 5 is 4. Now a warning. [27] You might be used to writing function application like this, with parentheses and commas. In this course, that's a bunch of arguments, and [28] you get a bunch of results, and that's probably not what you meant.

[29] If a function has a binary result, we call it a predicate. This function tells whether its argument is even or not. And if a [30] function has a predicate result, we call it a relation. So a relation is a function of two variables that has a binary result. The divides function here tells whether its first argument divides into its second argument with no remainder. That makes the [31] even function equal to the divides function applied to the single argument 2.

[32] There's an operator called selective union that takes two functions and produces a new function. The [33] domain of the new function is the union of the domains of the two operands. The [34] result of the new function, when applied to argument x, is the following. If x is in the domain of f, then the result is what f would give. If not, the result is what g would give. So f otherwise g is mostly like f, except when f doesn't apply, and then it's like g.

[35] Next I want to give you some abbreviations you can use for functions. [36] The first one is for functions of 2 or more variables that all have the same domain. [37] Just group the variables and write the domain once. That's like programming languages where you can declare several variables of the same type together. For the next abbreviation, I'll use the [38] successor function as an example. The abbreviation is [39] to leave out the domain. We do that only if everyone knows what the domain is, maybe because we say all

domains today are nat. So then we don't need to write them. Or, sometimes the domain doesn't matter for the point being made, and we can leave it out. In [40] the next example, we have a very small domain, just the number 2, but that's irrelevant for the example. What's relevant is that the body doesn't use the variable. That's called a constant function. And then there's no need to [41] write a variable. And you don't need scope brackets to show the scope of a variable if you don't write any variable. And the dot somehow turns into an arrow. So 2 arrow 3 is a little function with domain 2 and result 3. It maps 2 to 3. And finally, [42] we could abbreviate so much that we leave out both the variables and the domains. In this example, that would be [43] just x plus 3. Sometimes people say an expression is a function of its variables. But you might not be able to see all the variables, and you don't know what order they come in, so you can't apply an expression like this.

[44] The final topic this lecture is scope and substitution. A function introduces a local variable, which the body of the function can refer to. But the body can also refer to other variables, which are nonlocal. Some people use the word [45] global instead of nonlocal. Some people call them [46] bound and free variables. Some people call them [47] hidden and visible variables. Some people call them [48] private and public variables. That's a lot of terminology. I'll just stick with local and nonlocal. [49] A variable is local to an expression if the variable is introduced, using the function notation, within the expression. Otherwise it's nonlocal. [50] In this example, I've just shown the variables, and left out the other parts of the expression. [51] There is one local variable introduction, introducing local variable x, and that local variable occurs once. [52] There are two nonlocal variables. One of them is also named x, but it's not the same variable as the local x. It occurs once. The other nonlocal variable is y and it occurs twice. [53] Now [54] here's an example in which x is introduced locally, twice. The [55] outer function's x is used twice. That outer function is being applied to 3, so 3 should replace the outer function's x at both its occurrences. Like [56] this. But it doesn't replace the inner function's x. [57] And one last example. [58] Again there's a function within a function. The outer function [59] introduces variable y, and it occurs 3 times. This outer function is being applied to argument [60] x, which is a nonlocal variable. So this x has to replace all 3 occurrences of y. The problem is [61] this y. The nonlocal x has to replace it, but putting nonlocal x here will make it look like the local x of the inner function. We won't be able to tell them apart, so we must not do that. Before we can apply the outer function to its argument, [62] we have to rename the inner x to something that won't cause confusion. [63] Now we can apply the outer function, and replace all occurrences of y with x.

I'll have a little more to say about functions later, but that's enough for now.