[1] If you're going to skip a lecture, this one is probably the one to skip. Or, you could listen to it but not worry about it. Because the things I have to say now are the fine points about functions. I've already said the important stuff. If you are still listening, functions can be classified as partial or total. A partial function is one that, for some domain elements, produces no result. And a total function is one that always produces at least one result. [2] Another way functions can be classified is as deterministic or nondeterministic. A deterministic function always produces at most one result, and a nondeterministic function produces more than one result for some domain elements. And [3] here is an example function that is both total and nondeterministic. For each n in nat, it produces 2 results, n and n+1. So if we [4] apply it to 3, say, we get both 3 and 4. Well, you could say that's just one result, namely the bunch consisting of 3 and 4. But that's what a nondeterministic function means. And a partial function means the result is null for some domain elements. That's just terminology.

[5] A lot of things distribute over bunch union, including function application. If you apply a bunch of functions to an argument, you get a bunch of results. [6] If you apply a function to a bunch of arguments, you get a bunch of results. Almost anything you do with functions and bunches works. But there's one way you can go wrong. [7] Here's a function that doubles its argument. So [8] double of 2 is 4. And [9] double of 2 and 3 is 4 and 6. So far so good. But when you apply double to a bunch of arguments, [10] you can't just substitute the bunch of arguments for the parameter. You apply a function to an element of its domain using the application law. But you can't use the application law for a nonelementary argument. You have to use distribution.

So what do you do if you really want a function that applies to bunches? For example, a function that tells whether a bunch is small or large? The answer is: use sets. Set brackets turn a bunch into an element. That's what they're for. For example, [11] here's a function, called tiny, that introduces a set valued variable, and returns a binary that tells whether its size is less than 3. If we apply it [12] to the null set, we get true. If we apply it to [13] this set, we get false. But if [14] we apply it to the null bunch, we get null because any function applied to null is null. If you apply tiny to the null bunch of sets, you get the null bunch of results.

Now on [15] to the next fine point about functions. Here's the function inclusion axiom. It tells us when function f is included in function g. Remember, a function is not an elementary bunch, and it's just an English problem that I have to use a singular noun for it. f is included in g when the domain of g is included in the domain of f, and for all elements in the smaller domain, the result of f is included in the result of g. That first part about the domains might look backwards, but I'll explain it in a minute. First let's [16] look at an example. This expression is used by mathematicians to say something about the domain and result of function f. A arrow B is [17] an abbreviated way of writing a constant function, whose result doesn't use the function's variable. [18] It's a function whose domain is A and whose result is B. Using [19] the definition of function inclusion, f is included in A arrow B when A is included in the domain of f, and for any element of A, the result of f is included in B. So we could say that [20] A arrow B is all those functions whose domain is at least A and whose result is at most B. At least A and at most B. [21] A arrow B is a space of functions that's antimonotonic in A, and monotonic in B. The bigger A is, the smaller the space. The bigger B is, the bigger the space.

[22] Now we can prove that the successor function is in nat to nat. [23] First use the function inclusion axiom. [24] The domain of successor is nat, and successor of n is n plus 1. [25] The first conjunct is true because colon is reflexive, and the second conjunct is true by the definition of nat. [26]

[27] And in exactly the same way that we proved successor is in nat to nat, we can prove that even is in int to bin, and max is in xrat to xrat to xrat.

Ok, so [28] here's a theorem about function inclusion. If A is in B, and f maps things in B to things in C, and C is in D, then f maps things in A, which are in B, to things in C, which are in D. It says you can decrease the domain, and increase the range. It's just saying again that arrow is antimonotonic on its left and monotonic on its right. [29] So it says that successor maps things in 0 to 10 – to – integers.

[30] Here's a function whose variable is [31] f and whose domain [32] is the space of functions that map things in 0 to 10 – to – integers. It's a function that applies to functions. Logicians call it a higher order function. We need the domain of any argument to be at least 0 to 10 because [33] it applies its argument to everything in 0 to 10. It doesn't matter if the domain of the argument is bigger than that; the other domain elements just won't be used. And we need the result of any argument to be in the integers [34] because it tests the result for evenness. It doesn't matter if the result of an argument is never negative, as long as it isn't a fraction or a character or something else that can't be tested for evenness. So we can [35] apply this higher order function to successor. Well, there's another fine point in the textbook about this, but I'm just explaining why arrow is antimonotonic on its left and monotonic on its right. The result is [36] this, which says the first 10 successors are even, [37] which is false.

So that was function inclusion, and [38] this is function equality. It says two functions are equal if their domains are equal and their results are equal. Not surprising.

We just saw an example where a function had a function-valued parameter. So a function g can be applied to a function f. [39] If function f is not in the domain of function g, then g f is not g applied to f, but g composed with f. And the result of the composition is a new function. Looking [40] first at the result, applying g f to an argument x means applying f, and then applying g to the result. It looks like an associative law, doesn't it? So the [41] domain of g f is those things you can apply f to, that result in things you can apply g to. For example, [42] even is a function that applies to numbers, not to functions. So successor is not in the domain of even. So even successor is a composition. Its domain is the naturals. [43] And when you apply even successor to 3, you [44] first apply successor and get [45] 4, then you apply even and get [46] true.

Operators and functions are about the same thing. So operators can be composed the same way as functions can. And they can be mixed, too. [47] Here we compose minus and successor. Since minus doesn't apply to successor, we apply successor to 3, and then apply minus to the result. [48] Here we compose not with even. Not even applied to 3 gives true.

[49] Next we have something that really doesn't matter at all. I just think it's cute. It's called Polish notation, and you may have seen it in a compiler course. [50] Suppose x and y are number variables, suppose f and g are functions of 1 number variable, and suppose h is a function of 2 number variables. Polish notation means writing compositions and applications without parentheses, like [51] this. The rules of precedence say you bracket to the left, like [52] this. The rules of composition say that because h doesn't apply to f, we instead [53] apply f to x and h to the result. And because h composed with f of x doesn't apply to g, instead we [54] apply g to y and h f x to the result. Dropping [55] one superfluous pair of brackets, we get h applied to 2 arguments, namely f of x and g of y. Even though we didn't write any brackets to start with, it all sorts itself out and comes out right.

The [56] next thing is a bit more important. If m is an index of list L, then this function can be [57] applied to m, and what you get is [58] L m. So the [59] function applied to m is the same as the list indexed with m. The function and the list are playing the same role here. [60] Function composition and list composition work exactly the same way. So you can even compose them with each other. For example, [61] successor composed with a list of arguments gives a [62] list of results. Do you remember the [63] list modification operator from list theory? This one is a list whose item 1 is 21, and otherwise it's the same as 10 11 12. Another way to look at it is [64] that 1 arrow 21 is a function that

maps 1 to 21, and [65] the list 10 11 12 is a function also, and between them there's a selective union operator. The result of that selective union is a function that's like the list 10 21 12. — [66] And, we can even apply a [67] quantifier to a list. This is the sum of the list. It's the same as if we apply the quantifier to the [68] function that's like the list.

And the final topic in this collection of topics is [69] limits. If we have a function whose domain includes nat and whose range is in rat, then [70] f0, f1, f2, and so on, is a sequence of rationals. [71] Limit is an operator that applies to such a function. People sometimes say the limit of the sequence, rather that the limit of the function. And [72] this is its axiom. It says that the limit is at least as large as the maximum of all the minimums for each starting point in the sequence. And symmetrically, the limit is at most the minimum of all the maximums. Well, I'm not going to try to explain that. Here [73] LIM is applied to a function, but like with the other quantifiers, I didn't write the scope brackets. The variable is n, and its domain is nat but I didn't write the domain either. The body of the function is 1 over n plus 1, and as n becomes large, the result approaches 0. And indeed [74] 0 is the limit. This [75] next example, minus 1 to the n, is one where traditional mathematics says there is no limit, and, the axiom doesn't tell us what the limit is. But it does say [76] that it lies between minus 1 and 1. And [77] always, the minimum is less than or equal to the limit, which is less than or equal to the maximum. Traditional mathematics texts use the words liminf and limsup instead of minimum and maximum.

This is a computing course, not a math course, so it's important that everything is formally defined. Because formal definition is required for implementation on a computer. And I have defined everything formally with only one exception. I didn't define the real numbers formally yet. But I didn't use them yet, either. [78] Here's the definition. The reals are the limits of all these functions. Actually, that's the extended reals. To get the reals, just chop off infinity and minus infinity.

[79] Let p be a predicate on the natural numbers. Then [80] p0, p1, p2, and so on, is a sequence of binary values. And [81] here's the axiom for the limit of a predicate, or sequence of binary values. It's just like the previous axiom, if you remember that disjunction is the same as max, conjunction is the same as min, and implication is the same as less than or equal to. I think the easiest way to understand it is to look at the two implications separately. The [82] first one says that LIM p is true, for some particular values of the nonlocal variables, if there exists a point in the sequence, such that ever after that point, the predicate is true. If the sequence becomes true and stays true, then the limit is true. And the other implication can be rewritten like [83] this. It says, if the sequence becomes false and stays false, then the limit is false. Here's [84] my example. The limit of 1 over n plus 1 was 0. But the limit of the binary sequence 1 over n plus 1 equals 0 – is false, because it's false all along.

If you stuck with this lecture to the end, I bet you're feeling overwhelmed by all these details. But remember, I said they're not really so important. Anyway, that's the end of all the math that I need for the whole course. Starting next lecture, we look at programming and computation, and I think the math will make more sense when we're applying it to programming.