[1] The main kind of programming that's done in the world, by far, is imperative programming, and that's what almost all of this course is about. But in this one section I want to show you another kind of programming, called functional programming. Think of it as what you get if you [2] take away assignment, take away sequential composition, and add functions. Imperative means commanding, and an imperative program commands a computer to change state. The assignment is the simplest program that commands a change of state, and that's gone. In functional programming, we don't talk about state at all. In functional programming, [3] a specification is a function. The parameters are the input, and the result of the function is the output. [4] A program is just an implemented specification, as always, but here that means it's a function. [5] We include application, composition, and selective union as implemented operations, but [6] not quantifiers. [7] Feeding inputs to a program means applying a function to some arguments.

[8] Here's an example. This specification is about a computation that takes a list of rationals as its input and produces their sum as its output. The only thing that stops this specification from being a program is the summation quantifier. [9] We have to find a way of expressing this sum that is implemented. I can't do that in one step, but [10] here's a step towards that goal. The function here is a generalization of the original specification. This function describes the problem of finding the sum from position n onwards. And it's applied to 0, so that's the sum of the whole list. But now we have to [11] re-express the problem of finding the sum from position n onwards. One thing I don't like about this function is the plus 1 in the domain. We need it because n could be equal to the length of the list. But it's ugly. One way to get rid of it is to [12] divide the domain into two parts. One part is 0 to but not including length of L, and then the other part is just length of L. Actually, that's not just to be pretty, it's because we have to treat those two cases separately. I can separate them by [13] using a selective union. Now we have two functions to re-express. [14] Here's the left one. If n is in 0 to length of L, then it can index an item, so the sum is [15] L n plus the sum from n plus 1 onwards. The [16] right side of the selective union just has a one element domain, so n is the length of the list, and the sum is [17] 0. There's [18] still one summation quantifier to deal with. The sum from n plus 1 onwards can be changed to a [19] sum from n onwards by making a function with n as parameter, and then applying it to argument n plus 1. That's the functional version of n gets n plus 1. And the point of doing that is the new function we created is actually [20] one we've already dealt with. That's the recursion. And we're done. [21] Well, we're done according to most functional programmers, but I want to do the timing. All we do is [22] change the result we want to compute into the [23] time it takes to compute it, which is the length of the list. So then [24] this should be the length of the list [25]. So now [26] this has to be changed to the time left in the computation, which is [27] the length minus n. And this equation is easy to prove. Now [28] this has to be changed to the length of the list minus n, and so do all the other function results in this group of equations [29]. They're still theorems because they were about the domains, not the results. Now [30] this result has to go in [31] this position, and now we have to change the next line appropriately. First, we have to decide what we're charging for, what costs time in this computation. One possible answer is [32] additions, and this next line is the one that does them. So let's say we charge time 1 for each addition. That makes it [33] 1 plus something, so the something obviously has to be length minus n minus 1. The [34] next line came from the right part of the selective union, so that makes it [35] length minus n. And by coincidence, the equation is already correct without changing the other side. The [36] last line came from up there, so we have to change it to [37] length minus n minus 1, and change its right side to length minus n, which is exactly right for the recursion. And that's the timing proof, counting [38] 1 for each addition. Another timing policy we could have used is recursive time. For that one, we don't charge anything here for the addition. Instead we charge [39] here for the recursion. So [40] here's the change.

[41] We've been writing equations, and that's what functional programmers do, because they don't know about nondeterministic functions, so they don't know how to deal with nondeterminism. But we do. And that's important in order to avoid overspecification. If we would be happy with a range of results, we should be able to say so in the specification. Here are the relevant definitions for functional programming. [42] A specification S is unsatisfiable for domain element x means that S applied to x produces the empty bunch of results. [43] It's satisfiable if it produces at least 1 result. It's [44] deterministic if it doesn't produce more than 1 result, and [45] nondeterministic if it produces more than 1 result. Just to make it look more like the imperative case, [46] satisfiable means exists y such that y is in S of x, and [47] implementable means for all x there exists y such that y is in S of x. Or [48] for all x, S of x is not null. Finally, we have to define [49] refinement in this functional world. In the imperative world, where a specification is a binary expression, refinement is implication, which is the ordering on binary values. So in the functional world, where a specification is a function, it's [50] the ordering on function spaces, which is just inclusion. If we have specification S, whose result is a bunch of possibilities, we can refine it to a function P whose result is a smaller bunch of possibilities. We resolve some nondeterminism. Refine means decrease the bunch of results, or increase the domain, or both. For imperative refinement, I like to turn the implication symbol backwards so I can write the problem first, and then say what refines it. For functional programming, that's [51] double colon.

Now let's [52] look at an example. We'll search for an item in a list. The item might be in the list several places, and we don't care which one of them we find, so that makes it nondeterministic. The [53] specification has to have the list L as one parameter, and the item we're looking for x as another parameter. And the result is any of those places n where L n equals x. I said it's nondeterministic, but actually that depends on the list and the item. There's even a possibility that x isn't in the list, and that makes this specification [54] unimplementable. Let's patch up that hole [55] . If x is anywhere in L, then the result is any position where it is. Else it's any natural number that doesn't index the list. I could have picked the length of the list, but I just wanted to make the example as nondeterministic as I could. Now we have to [56] refine. First we introduce a [57] new variable i, and the result of this function is the result of the search from position i onwards. We put i where the two 0s were. And then we apply the function to argument 0. If this were an imperative program, we would be saying i gets 0. As a functional program, it's parameter i gets argument 0. Now [58] we have to refine the result of this new function. And we can start with [59] if i equals the length of the list, then the length is the result. I wrote the refinement symbol the first time, but it was really an equality. This time it really is refinement, because we just chose one out of many possibilities for the result. [60] Else if x equals L i then i. Again that's refinement, but not equality, if there are other occurrences of x. [61] Else recursively call the function that looks from i onwards, but apply it to i plus 1. In an imperative program, we would say i gets i plus 1. And we're done except for timing. The [62] timing expression is also nondeterministic because the time might be anything from 0 right up to and including the length of the list, depending on how soon we find x. The result of this specification [63] needs refinement, and it has to be a function with the same domain as before and applied to 0. So the expression has to be 0 to the length minus i plus 1. And [64] we have to refine that the same way we refined the corresponding expression before. So [65] if i equals the length then 0, and, yes, 0 is in the bunch 0 to 1, which is the bunch on the left when i equals the length. [66] Else if x equals L i then 0 again for the remaining time. [67] Else we have the recursive call, so we have to charge 1 for that, and then it's applied to i plus 1. I want to [68] check that last case. [69] Apply the function, then [70] the ones cancel. Now [71] add one. And that's almost the same bunch as the left side but 0 is missing. So it's [72] included in the left side, and that's all we need.

Now I want to compare [73] functional programming with imperative programming. It's really [74] all the same steps, but in a different notation. In [75] functional programming we apply functions, using the application axiom. In [76] imperative programming we use the substitution law. But they're exactly the same. Just substitute. In spite of the similarity, there are two camps of programmers, each claiming that their way is somehow better than the other.