[1] This lecture is called recursive program definition, but I think that's not the right name. Programs are just special cases of specifications, and what I have to say applies to specifications generally. So let's rename it [2] recursive specification definition. And [3] here is one, where x is an integer variable and t is time. It's really just a [4] recursive procedure or method definition as you might write in a programming language, except for some syntactic differences, and also the t gets t plus 1 that I put in for recursive time. [5] But it doesn't define zap very well because this equation has many [6] solutions for zap. Here are 6 of them. The [7] top solution is the weakest one; it says the least about the computation because it only talks about when x is nonnegative. The [8] bottom two are too strong to be talking about computation; they're unimplementable. [9] Here's the relationship among these solutions. A structure like this is called a lattice. Even if we don't know any more about zap, we at least know that it's [10] stronger than or equal to the weakest solution, so that means we can use it to refine at least this one specification. And we know [11] zap is refined by its constructor, so it's implemented and we can execute it. I said constructor because the zap equation is a fixed-point construction axiom. It says zap equals a function of zap. [12] Let me show how we can define zap by ordinary construction and induction. [13] Here are the construction axioms. And to help see why these are construction axioms, [14] here are the nat construction axioms for comparison. The [15] first one is a base case. For zap it says that time doesn't go backwards, which is true for all computation. The reverse implication in the zap axiom is playing the same role as the colon in the nat axiom. The [16] other construction axiom takes a step. For nat it's adding one; for zap it's one loop iteration. And in each case, we can [17] write the pair as a single axiom. And that's helpful for writing the [18] induction axiom. For nat, the induction axiom says that of all bunches satisfying the construction axiom, nat is the smallest. For zap, induction says that of all specifications satisfying the construction axiom, zap is the weakest. I'm not sure that we want to define zap as the weakest solution, but if we do, we need this induction axiom. [19] Or, another way to get the same result, is [20] fixed-point construction and induction, where we strengthen the implication to an equation, both in the construction axiom and in the antecedent of the induction axiom.

[21] There's a procedure for finding the fixed point, just like there was for recursive data definition. We can start with [22] zap sub 0 equals true, and then [23] zap sub 1 equals the constructor but with zap 0 replacing zap. This can be [24] simplified. x equals 0 implies x prime and y prime are both 0 and t prime equals t. To get [25] zap 2, use zap 1. And this simplifies to [26] if x is in 0 to 2, then x prime and y prime are both 0, and t prime equals t plus x. Maybe we should do a few more, or maybe we can now guess the general case. I'm guessing [27] zap n equals if x is in 0 to n, then x prime and y prime are 0 and t prime is t plus x. Then we [28] replace n with infinity, and we have a candidate, which we have to test by seeing if it satisfies the construction axiom. And if we have an induction axiom, we have to test it in that too. In this case, it does satisfy both of them. It is the weakest solution for zap.

This recursive construction procedure has some options. We [29] don't have to start at true. We can start with any specification. Different starting points may give different candidates for solutions. And [30] instead of substituting infinity for n, we could take the limit. Occasionally that gives a different candidate for solution. [31] My favorite starting place is [32] t prime greater than or equal to t. Using that for zap 0, we get [33] this for zap 1, — and [34] this for zap 2, — and [35] this for zap n, — and finally [36] this for zap infinity. It's a different solution of the zap construction axiom. It's not the weakest solution, so it doesn't satisfy zap induction. I think it's better than the weakest solution as a description of the zap computation.

I gave you the best way of proving programs with while loops and other loop constructs in them in a previous lecture. But I said there is another way of dealing with

them. And [37] here it is. We can define a while-loop by construction and induction. Here are the two construction axioms. The first one is a base case saying that while loops don't make time go backwards. The second one is the step, saying that a while loop implements its first unrolling. The time increase and placement depends on your choice of timing policy. We can write these two axioms as a [38] single axiom, like this. And that helps for writing the [39] induction axiom, which says that of all specifications solving the construction axiom, the while loop is the weakest. That's ordinary construction and induction. We could also write [40] fixed-point construction and induction by changing an implication into an equation. The fixed-point definition is equivalent to the ordinary definition. If you ever hear the term "least fixed-point semantics", this is what it means. It's the most popular way to define the meaning of loops, but I think the way I defined loops in the previous chapter is both simpler and more useful.