[talking head] This lecture we study theory design and implementation. Programmers have two roles to play here. In one role, they are theory users. They use numbers, and lists, and lots of things that have been invented by other people, and formalized as theories, so programmers can prove their refinements. Programmers also use whatever features their programming language includes, and it's the responsibility of the language designers and implementers to define those features formally as theories.

The other role programmers play is designers and implementers of new theories. Every time you write a program, you invent new abstractions and new structures. In order to prove your refinements, you have to formalize what you invent, so you are designing and implementing new theories. And that's what I want to talk about this lecture.

I'm going to present stacks, queues, and trees as case studies. I suppose you already know what these data structures are and how they're used. That's the material of a data structures course, not this course. In this course, we need to be able to design the axioms that define the structures we invent. So I'll use stacks, queues, and trees as examples of theory design and implementation.

The first work on this was done about 1970, some of it at the University of Toronto where I was. Back then, programs were not considered to be mathematical expressions, but functions were. So push and pop and all the other operations on these structures were not formalized as programs that change the state of memory. Instead they were formalized as functions that have a data structure as input parameter, and another one as result. These are the data theories that we look at in this lecture. Now that we *do* consider programs to be mathematical expressions too, we can formalize the operations as programs, and that's the topic of the next lecture.

Let's start with the stack.

[1] Like any theory, stack theory introduces some new notations and some axioms about these new notations. [2] stack is the bunch of all stacks whose items are of some type, and I'll use a big X for the type. [3] empty is a particular stack. [4] push is a function that takes a stack and an item and gives back another stack. [5] pop takes a stack and produces a stack. [6] top takes a stack and produces an item. [7] And [8] here are some axioms that say formally most of what I just said informally. – Ok, [9] here's the empty stack. If we push something onto it, we get [10] another stack, let's call it s1. And if we push something onto that we get [11] another stack, let's say s2. Push again, and get [12] s3. And again and get [13] s4. Actually, this picture should be a tree, because from each stack, you can get many different stacks by pushing different things onto it. If we push again, how do we know we get a different stack from the ones we already have? Push just says we get a stack, it doesn't say a different stack, so [14] as far as these axioms are concerned, it could be empty. Of course, that's not what we want, so we need another [15] axiom to prevent that. It says the result of a push is never empty. Ok, but maybe it's [16] still a stack we had. To prevent a loop like that, we need [17] *another* axiom. It says: The only way 2 pushes can result in the same stack is if they both start with the same stack and push the same thing onto it. So that eliminates all possibility of looping. But maybe [18] there's a whole other line of stacks, with no beginning and no end. Maybe there are infinitely many other lines of stacks. The axioms allow it. empty is a stack, pushing always gives another stack, pushing never gives empty, and 2 pushes don't give the same stack. To eliminate other lines, we need another axiom. Do you know which one? — It's induction that says there are no other stacks except those you get from construction. So let's [19] rewrite the first 2 axioms as a single axiom, and then we can [20] see what induction is. It says of all bunches satisfying construction, stack is the smallest. An alternative way of writing it is [21] in predicate form. It says [22] if you prove P about empty, and [23] if assuming P of stack s allows you to prove [24] P of the result of pushing something onto s, then you have [25] proven P of all stacks. [26] So far, we don't have any axioms that say a stack is a last-in-first-out structure. We need to say [27]

if you push something onto a stack, and then pop the stack, you get back exactly the stack you had before the push. And [28] if you push something onto a stack, and then ask what is the top item, it's exactly the item you just pushed. That's all the stack axioms. Actually, it's more than all the axioms because some of them are restatements of others.

[29] Now I want to talk about implementation. The theory introduced [30] these new names. All we have to do to implement stacks is to [31] define each of them using constructs, or theories, that are already implemented. Let's say lists and functions are already implemented, and let's say the type of items is the integers. Then we can define stack as [32] all lists of integers. We can define empty as [33] the empty list. Push can [34] join the new item to the end of the list. Pop [35] says if the stack is empty then return empty, else return the list with its last item cut off. [36] Top says if the stack is empty return 0, else return the last item. So that's an implementation of stack theory using list theory and function theory. Well I claim it is, but we don't accept claims in this course without [37] proof. How would you prove that this implementation is correct, or perhaps I should say, prove that this is an implementation. — We need to [38] prove that the axioms of the theory are satisfied by the definitions of the implementation. That means [39] the axioms of the theory are implied by the definitions of the implementation. The axioms are the [40] specification of the data type, so we're proving that a specification is implied by, or refined by, or implemented by, the implementation, same as usual. So we assume the definitions of the implementation, and prove the axioms of the theory. That sounds wrong, because axioms don't need to be proven. What we're really proving is that the implementation satisfies the axioms. Here's [41] a proof of one of the axioms. I won't go through it in detail, but I just want to point out that [42] we have to use the definitions of the implementation, of course. And we also have to use [43] list theory and function theory, because those are the theories we used in the implementation. And notice right [44] here it says: if s join x equals the empty list. Now, if you join an item onto a list, it can't be empty, so that's false, and the if-then-else-fi reduces to its else-part. That means that the 0 in the then-part is never used. It could have been anything, and the proof would still work. The other axioms are proven similarly, so I won't show them right now.

We've implemented stacks, so [45] now we can declare variables of type stack, and we can [46] assign to those variables. *a* gets empty, and [47] b gets push *a* 2, and so on.

There's a very important question that comes up whenever you introduce some new axioms. The question is: [48] are the axioms consistent? Are they consistent with each other, and are they consistent with the axioms we already had? If the axioms are inconsistent, the way to prove it is to prove false. If the axioms are consistent, there's only one way to prove it, and that is [49] to implement them. We implemented stack theory, and that proves it's consistent. Actually, implementation proves that if the theories used in the implementation are consistent, then the theory we implemented is consistent also. So it's just relative consistency. Ultimately, consistency comes from the fact that the most basic theories are implemented in hardware, and we can run them.

Logicians have another word for implementation. They call it building a model. They use the word model the opposite way from everyone else in the world. According to physicists and architects and computer scientists and all the dictionaries, a model is more abstract, less detailed, than the thing it models. And a theory is a kind of model - a mathematical model. Stack theory tells us what properties we want of a stack, but it doesn't say how we get those properties. An implementation is more detailed, and says exactly how we get those properties. Only logicians use the word model to mean the more concrete, more detailed, implementation. Their usual language of implementation is set theory. They say they are building a set model to prove consistency. Again, that's just relative consistency, but they are very sure that set theory is consistent because they've been staring at it for a century now. There were some inconsistencies at first, but they fixed them, and they haven't

noticed any new inconsistencies for a long time. I prefer the assurance of a working computer over the assurance that logicians don't see any inconsistency.

The other question about axioms is [50] completeness. This question isn't nearly so important as consistency. Stack theory is [51] incomplete, and here are a couple of binary expressions that are neither theorems nor antitheorems according to the theory. Stack theory doesn't tell us what pop of empty and top of empty are. Our particular implementation makes these two binary expressions theorems, and that's a good example of an implementation being more detailed than the theory it implements. To prove that a theory is incomplete, you have to [52] implement it twice, so that in one implementation, some expression is a theorem, and in the other, it's an antitheorem. Then you know that in the theory, it was neither. We could make top of empty be 1, or 2, or anything, and the proof of implementation would still work.

[53] Here's a point that has been made before, but it's worth making again. A theory, or specification, acts as a firewall between the people who use the theory and the people who implement the theory. This becomes more important as the software becomes larger. The users can use only the properties provided by the theory, and not any extra properties that the implementation may have. And the implementer has to provide all the properties of the theory. That way, the user doesn't have to know about the implementation, and the implementer doesn't have to know about the uses of the theory. So they are each free to make changes on their own side of the firewall without affecting anything on the other side. [54]

[55] Here again are the axioms of data-stack theory. [56] This axiom says that popping a stack results in a stack. A consequence is [57] that popping empty results in a stack. So the implementation was obliged to provide an answer for pop of empty, even though it didn't matter which stack was the result. We already have an axiom [58] that tells the result of popping a nonempty stack, which is a stack that's the result of a push. I think it might be better *not* to have an axiom that says pop of empty is a stack. So let's [59] get rid of this axiom. Now an implementer can provide an error message as the result of pop of empty, and that's much more useful. Similarly [60] this axiom has the consequence that top of empty is a value of type X. We already have [61] an axiom that says what the top of a nonempty stack is, and that's all we need for top. Let's get [62] rid of this axiom, so an implementer isn't obliged to give a value of type X for top of empty, and instead can make an error message as the result. We don't need [63] this one because it's just a repetition of these [64] axioms using predicates, so I'll [65] cross it off. But maybe we don't need this [66] induction axiom either. We need it if we want to prove something about all stacks. But maybe we just want to prove that a program using a stack is correct, and we don't need to prove anything about all stacks. So [67] cross it off. Now let's see, are there any more axioms I can get rid of? You know, we don't really need [68] empty. The way stacks are used is to work on top of the stack you're given, and when you're done, leave the stack the way you found it. If you really need empty, you could make the equivalent effect by first pushing some special value onto the stack, and then the test for empty is just a test to see if that special value is the current top of stack. So we don't need [69] these, except that we do need to know that there's at least one stack, so we can declare variables of type stack. We don't even need [70] this one, that says pushing always makes different stacks. [71] All we really need is just these 4 axioms. We [72] need to be able to declare variables of type stack, we [73] always need to be able to push and keep pushing, when we [74] pop we need to get back the stack we had before, and we want [75] top to be the last item pushed.

If you're a mathematician, you want the strongest theory possible. You want as many axioms as possible so you can prove as much as possible about stacks. If you're a software engineer, you want the fewest axioms possible. You want just the axioms you need to prove your programs correct. You want the weakest theory possible, because that makes it easiest

to implement.

[76] Now on to data-queue theory. [77] We need an empty queue, which is a queue. We can get away without an empty stack, but we have to have an empty queue. And we need a [78] join operation, that takes a queue and an item and gives back a queue. Or maybe we want this [79] slightly weaker axiom. It's almost the same, but it doesn't say anything about domains. Maybe we want to say that [80] join doesn't produce empty, and [81] two joins produce the same queue if and only if they start with the same queue and join the same thing on. Or maybe not. I'm not sure. We need a [82] leave operation that takes a queue and gives back a shorter queue, but this one is too strong. We can weaken it to [83] not talk about domains, but we need to weaken it more so it doesn't say anything about leave of empty. Maybe [84] this is what we want. We also need [85] front to tell us what the front item is, but again [86] weaker, and even [87] weaker. If we want to prove something about all queues, we need [88] induction. If not, we should leave that one out. What's left to say is that queues are first in first out. That takes 4 [89] more axioms. This [90] one says if something joins an empty queue, and then something leaves the queue, the result is the empty queue. This [91] one says if something joins a nonempty queue, and then something leaves the queue, the result is the same as if something leaves first and then something joins. In other words, if the queue is nonempty, joining and leaving commute. This [92] one says if something joins an empty queue, then it's the front. And the [93] last one says if something joins a nonempty queue, then the front doesn't change. And that's it for queue theory.

The last theory in this lecture is [94] tree theory. And I'll start with a strong version. We might start with [95] an empty tree, emptree. Stupid name. Never mind. And then [96] a way to build bigger trees out of littler ones. Graft takes a tree, which will be used as the left subtree, and a value of type X, which will be the new root, and another tree, which will be the right subtree, and gives back a tree. emptree and graft are the constructors, so we're building binary trees with information at the internal nodes. And the leaves are all empty trees. And here's [97] induction. The [98] next one says that grafting does not produce an empty tree. And the [99] next one says that two grafts result in the same tree if and only if they start with the same trees and item. These axioms should be looking pretty familiar to you now because they're so similar to the axioms for other data structures. And [100] finally we have 3 axioms that undo what graft does. left takes a tree and returns its left subtree. root takes a tree and returns its root value. And right takes a tree and returns its right subtree. And that's all the axioms for binary trees. And it's more [101] axioms than a software engineer needs or wants. All that's needed is [102] that there is at least one tree, so we can declare variables of type tree. And graft produces new trees from old ones, and left, root, and right do their job. That's enough so that if you put data in a tree, you'll find it when you come back there later. That's all the user wants, and more would just make the implementer's job harder. [103]

I would like to say a few words about [104] tree implementation. We have to define each of the new names using theories that are already implemented, and I'll assume once again that lists and functions are implemented. [105] emptree is an empty list, and [106] graft makes a list with 3 items. Item 0 is the left subtree, item 1 is the root, and item 2 is the right subtree. In this implementation, here [107] is an example tree. We're dealing with lists that could be any length, so how do we reserve memory for a tree value? The standard technique for storing values that could be small or large or any size is to reserve some particular size space, and whenever the value fits, put it there, and whenever it doesn't fit, put an address, or pointer, there and put the value somewhere else. For our implementation, leave space that can hold an empty list, in case it's the empty tree, or 2 addresses and an integer, in case it's a nonempty tree. I'm assuming the items are integers. So [108] here's what our example tree looks like. It looks like a tree. The pointers are there, not because our implementation of tree says anything about pointers, and certainly not because the axioms

say anything about pointers, but because it's a standard way to store things whose size is unpredictable.

Something very strange has happened in the design of programming languages and the programming taught in introductory courses. Pointers are to data what go tos are to program. Pointers are data addresses, and go tos are code addresses. We don't need go tos if the programming language allows recursive program definitions, and implements them well, which means it implements most recursive calls as go tos. We don't need pointers if the programming language allows recursive data definitions, and implements them well, which means as pointers. And some programming languages *do* allow recursive data definitions, and do implement them well. But most of the popular imperative languages allow recursive program definition, with varying qualities of implementation, and disallow recursive data definition. The introductory programming courses say don't use go tos, and they have a whole chapter on how to use pointers. To me, that's a strange inconsistency in language design and programming practice.

Here's [109] another way we might implement trees, just to show that there's more than one way to do it. The [110] empty tree is the number 0, and [111] graft makes a record, or structure, or object with 3 fields or attributes. In this implementation, our example tree looks like [112] this. When formatted nicely, like this, it looks like a table of contents, which is also a great example of a tree.