6.001 Structure and Interpretation of Computer Programs, Spring 2005
Transcript – 6A: Streams, Part 1

PROFESSOR: Well, last time Gerry really let the cat out of the bag. He introduced the idea of assignment. Assignment and state. And as we started to see, the implications of introducing assignment and state into the language are absolutely frightening. First of all, the substitution model of evaluation breaks down. And we have to use this much more complicated environment model and this very mechanistic thing with diagrams, even to say what statements in the programming language mean.

And that's not a mere technical point. See, it's not that we had this particular substitution model and, well, it doesn't quite work, so we have to do something else. It's that nothing like the substitution model can work. Because suddenly, a variable is not just something that stands for a value. A variable now has to somehow specify a place that holds a value. And the value that's in that place can change.

Or for instance, an expression like f of x might have a side effect in it. So if we say f of x and it has some value, and then later we say f of x again, we might get a different value depending on the order. So suddenly, we have to think not only about values but about time.

And then things like pairs are no longer just their CARs and their CDRs. A pair now is not quite its CAR and its CDR. It's rather its identity. So a pair has identity. It's an object. And two pairs that have the same CAR and CDR might be the same or different, because suddenly we have to worry about sharing.

So all of these things enter as soon as we introduce assignment. See, this is a really far cry from where we started with substitution. It's a technically harder way of looking at things because we have to think more mechanistically about our programming language. We can't just think about it as mathematics. It's philosophically harder, because suddenly there are all these funny issues about what does it mean that something changes or that two things are the same. And also, it's programming harder, because as Gerry showed last time, there are all these bugs having to do with bad sequencing and aliasing that just don't exist in a language where we don't worry about objects.

Well, how'd we get into this mess? Remember what we did, the reason we got into this is because we were looking to build modular systems. We wanted to build systems that fall apart into chunks that seem natural. So for instance, we want to take a random number generator and package up the state of that random number generator inside of it so that we can separate the idea of picking random numbers from the general Monte Carlo strategy of estimating something and separate that from the particular way that you work with random numbers in that formula developed by Cesaro for pi.

And similarly, when we go off and construct some models of things, if we go off and model a system that we see in the real world, we'd like our program to break into natural pieces, pieces that mirror the parts of the system that we see in the real world. So for example, if we look at a digital circuit, we say, gee, there's a circuit and it has a piece and it has another piece. And these different pieces sort of have identity. They have state. And the state sits on these wires. And we think of this piece as an object that's different from that as an object. And when we watch the system change, we think about a signal coming in here and changing a state that might be here and going here and interacting with a state that might be stored there, and so on and so on.

So what we'd like is we'd like to build in the computer systems that fall into pieces that mirror our view of reality, of the way that the actual systems we're modeling seem to fall into pieces. Well, maybe the reason that building systems like this seems to introduce such technical complications has nothing to do with computers.

See, maybe the real reason that we pay such a price to write programs that mirror our view of reality is that we have the wrong view of reality. See, maybe time is just an illusion, and nothing ever changes. See, for example, if I take this chalk, and we say, gee, this is an object and it has a state. At each moment it has a position and a velocity. And if we do something, that state can change.

But if you studied any relativity, for instance, you know that you don't think of the path of that chalk as something that goes on instant by instant. It's more insightful to think of that whole chalk's existence as a path in space-time. that's all splayed out. There aren't individual positions and velocities. There's just its unchanging existence in space-time.

Similarly, if we look at this electrical system, if we imagine this electrical system is implementing some sort of signal processing system, the signal processing engineer who put that thing together doesn't think of it as, well, at each instance there's a voltage coming in. And that translates into something. And that affects the state over here, which changes the state over here. Nobody putting together a signal processing system thinks about it like that.

Instead, you say there's this signal that's splayed out over time. And if this is acting as a filter, this whole thing transforms this whole thing for some sort of other output. You don't think of it as what's happening instant by instant as the state of these things. And somehow you think of this box as a whole thing, not as little pieces sending messages of state to each other at particular instants.

Well, today we're going to look at another way to decompose systems that's more like the signal processing engineer's view of the world than it is like thinking about objects that communicate sending messages. That's called stream processing. And we're going to start by showing how we can make our programs more uniform and see a lot more commonality if we throw out of these programs what you might say is an inordinate concern with worrying about time.

Let me start by comparing two procedures. The first one does this. We imagine that there's a tree. Say there's a tree of integers. It's a binary tree. So it looks like this. And there's integers in each of the nodes. And what we would like to compute is for each odd number sitting here, we'd like to find the square and then sum up all those squares.

Well, that should be a familiar kind of thing. There's a recursive strategy for doing it. We look at each leaf, and either it's going to contribute the square of the number if it's odd or 0 if it's even. And then recursively, we can say at each tree, the sum of all of them is the sum coming from the right branch and the left branch, and recursively down through the nodes. And that's a familiar way of thinking about programming.

Let's actually look at that on the slide. We say to sum the odd squares in a tree, well, there's a test. Either it's a leaf node, and we're going to check to see if it's an integer, and then either it's odd, in which we take the square, or else it's 0. And then the sum of the whole thing is the sum coming from the left branch and the right branch.

OK, well, let me contrast that with a second problem. Suppose I give you an integer n, and then some function to compute of the first of each integer in 1 through n. And then I want to collect together in a list all those function values that satisfy some property. That's a general kind of thing. Let's say to be specific, let's imagine that for each integer, k, we're going to compute the k Fibonacci number. And then we'll see which of those are odd and assemble those into a list.

So here's a procedure that does that. Find the odd Fibonacci numbers among the first n. And here is a standard loop the way we've been writing it. This is a recursion. It's a loop on k, and says if k is bigger than n, it's the empty list. Otherwise we compute the k-th Fibonacci number, call that f. If it's odd, we CONS it on to the list starting with the next one. And otherwise, we just take the next one. And this is the standard way we've been writing iterative loops. And we start off calling that loop with 1.

OK, so there are two procedures. Those procedures look very different. They have very different structures. Yet from a certain point of view, those procedures are really doing very much the same thing. So if I was talking like a signal processing engineer, what I might say is that the first procedure enumerates the leaves of a tree. And then we can think of a signal coming out of that, which is all the leaves.

We'll filter them to see which ones are odd, put them through some kind of filter. We'll then put them through a kind of transducer. And for each one of those things, we'll take the square. And then we'll accumulate all of those. We'll accumulate them by sticking them together with addition starting from 0. That's the first program.

The second program, I can describe in a very, very similar way. I'll say, we'll enumerate the numbers on this interval, for the interval 1 through n. We'll, for each one, compute the Fibonacci number, put them through a transducer. We'll then take the result of that, and we'll filter it for oddness. And then we'll take those and put them into an accumulator. This time we'll build up a list, so we'll accumulate with CONS starting from the empty list.

So this way of looking at the program makes the two seem very, very similar. The problem is that that commonality is completely obscured when we look at the procedures we wrote. Let's go back and look at some odd squares again, and say things like, where's the enumerator? Where's the enumerator in this program? Well, it's not in one place. It's a little bit in this leaf-node test, which is going to stop. It's a little bit in the recursive structure of the thing itself.

Where's the accumulator? The accumulator isn't in one place either. It's partly in this 0 and partly in this plus. It's not there as a thing that we can look at. Similarly, if we look at odd Fibs, that's also, in some sense, an enumerator and an accumulator, but it looks very different. Because partly, the enumerator is here in this greater than sign in the test. And partly it's in this whole recursive structure in the loop, and the way that we call it. And then similarly, that's also mixed up in there with the accumulator, which is partly over there and partly over there.

So these very, very natural pieces, these very natural boxes here don't appear in our programs. Because they're kind of mixed up. The programs don't chop things up in the right way. Going back to this fundamental principle of computer science that in order to control something, you need the name of it, we don't really have control over thinking about things this way because we don't have our hands in them explicitly. We don't have a good language for talking about them.

Well, let's invent an appropriate language in which we can build these pieces. The key to the language is these guys, is what is these things I called signals? What are these things that are flying on the arrows between the boxes? Well, those things are going to be data structures called streams. That's going to be the key to inventing this language.

What's a stream? Well, a stream is, like anything else, a data abstraction. So I should tell you what its selectors and constructors are. For a stream, we're going to have one constructor that's called CONS-stream. CONS-stream is going to put two things together to form a thing called a stream. And then to extract things from the stream, we're going to have a selector called the head of the stream.

So if I have a stream, I can take its head or I can take its tail. And remember, I have to tell you George's contract here to tell you what the axioms are that relate these. And it's going to be for any x and y, if I form the CONS-stream and take the head, the head of CONS-stream of x and y is going to be x and the tail of CONS-stream of x and y is going to be y. So those are the constructor, two selectors for streams, and an axiom.

There's something fishy here. So you might notice that these are exactly the axioms for CONS, CAR, and CDR. If instead of writing CONS-stream I wrote CONS and I said head was the CAR and tail was the CDR, those are exactly the axioms for pairs. And in fact, there's another thing here. We're going to have a thing called the-empty-stream, which is like the-empty-list.

So why am I introducing this terminology? Why don't I just keep talking about pairs and lists? Well, we'll see. For now, if you like, why don't you just pretend that streams really are just a terminology for lists. And we'll see in a little while why we want to keep this extra abstraction layer and not just call them lists.

OK, now that we have streams, we can start constructing the pieces of the language to operate on streams. And there are a whole bunch of very useful things that we could start making. For instance, we'll make our map box to take a stream, s, and a procedure, and to generate a new stream which has as its elements the procedure applied to all the successive elements of s. In fact, we've seen this before. This is the procedure map that we did with lists. And you see it's exactly map, except we're testing for empty-stream.

Oh, I forgot to mention that. Empty-stream is like the null test. So if it's empty, we generate the empty stream. Otherwise, we form a new stream whose first element is the procedure applied to the head of the stream, and whose rest is gotten by mapping along with the procedure down the tail of the stream. So that looks exactly like the map procedure we looked at before.

Here's another useful thing. Filter, this is our filter box. We're going to have a predicate and a stream. We're going to make a new stream that consists of all the elements of the original one that satisfy the predicate. That's case analysis. When there's nothing in the stream, we return the empty stream. We test the predicate on the head of the stream. And if it's true, we add the head of the stream onto the result of filtering the tail of the stream. And otherwise, if that predicate was false, we just filter the tail of the stream. Right, so there's filter.

Let me run through a couple more rather quickly. They're all in the book and you can look at them. Let me just flash through. Here's accumulate. Accumulate takes a way of combining things and an initial value in a stream and sticks them all together. If the stream's empty, it's just the initial value. Otherwise, we combine the head of the stream with the result of accumulating the tail of the stream starting from the initial value. So that's what I'd use to add up everything in the stream. I'd accumulate with plus.

How would I enumerate the leaves of a tree? Well, if the tree is just a leaf itself, I make something which only has that node in it. Otherwise, I append together the stuff of enumerating the left branch and the right branch. And then append here is like the ordinary append on lists. You can look at that. That's analogous to the ordinary procedure for appending two lists. How would I enumerate an interval? This will take two integers, low and high, and generate a stream of the integers going from low to high. And we can make a whole bunch of pieces.

So that's a little language of talking about streams. Once we have streams, we can build things for manipulating them. Again, we're making a language. And now we can start expressing things in this language. Here's our original procedure for summing the odd squares in a tree.

And you'll notice it looks exactly now like the block diagram, like the signal processing block diagram. So to sum the odd squares in a tree, we enumerate the leaves of the tree. We filter that for oddness. We map that for squareness. And we accumulate the result of that using addition, starting from 0. So we can see the pieces that we wanted.

Similarly, the Fibonacci one, how do we get the odd Fibs? Well, we enumerate the interval from 1 to n, we map along that, computing the Fibonacci of each one. We filter the result of those for oddness. And we accumulate all of that stuff using CONS starting from the empty-list.

OK, what's the advantage of this? Well, for one thing, we now have pieces that we can start mixing and matching. So for instance, if I wanted to change this, if I wanted to compute the squares of the integers and then filter them, all I need to do is pick up a standard piece like this in that square and put it in. Or if we wanted to do this whole Fibonacci computation on the leaves of a tree rather than a sequence, all I need to do is replace this enumerator with that one.

See, the advantage of this stream processing is that we're establishing-- this is one of the big themes of the course-- we're establishing conventional interfaces that allow us to glue things together. Things like map and filter are a standard set of components that we can start using for pasting together programs in all sorts of ways. It allows us to see the commonality of programs.

I just ought to mention, I've only showed you two procedures. But let me emphasize that this way of putting things together with maps, filters, and accumulators is very, very general. It's the generate and test paradigm for programs. And as an example of that, Richard Waters, who was at MIT when he was a graduate student, as part of his thesis research went and analyzed a large chunk of the IBM scientific subroutine library, and discovered that about 60% of the programs in it could be expressed exactly in terms using no more than what we've put here-- map, filter, and accumulate. All right, let's take a break. Questions?

AUDIENCE: It seems like the essence of this whole thing is just that you have a very uniform, simple data structure to work with, the stream.

PROFESSOR: Right. The essence is that you, again, it's this sense of conventional interfaces. So you can start putting a lot of things together. And the stream is as you say, the uniform data structure that supports that. This is very much like APL, by the way. APL is very much the same idea, except in APL, instead of this stream, you have arrays and vectors. And a lot of the power of APL is exactly the same reason of the power of this. OK, thank you. Let's take a break.

All right. We've been looking at ways of organizing computations using streams. What I want to do now is just show you two somewhat more complicated examples of that. Let's start by thinking about the following kind of utility procedure that will come in useful. Suppose I've got a stream. And the elements of this stream are themselves streams. So the first thing might be 1, 2, 3.

So I've got a stream. And each element of the stream is itself a stream. And what I'd like to do is build a stream that collects together all of the elements, pulls all of the elements out of these sub-streams and strings them all together in one thing. So just to show you the use of this language, how easy it is, call that flatten. And I can define to flatten this stream of streams. Well, what is that? That's just an accumulation. I want to accumulate using append, by successively appending. So I accumulate using append streams, starting with the-empty-stream down that stream of streams.

OK, so there's an example of how you can start using these higher order things to do some interesting operations. In fact, there's another useful thing that I want to do. I want to define a procedure called flat-map, flat map of some function and a stream. And what this is going to do is f will be a stream of elements. f is going to be a function that for each element in the stream produces another stream.

And what I want to do is take all of the elements and all of those streams and combine them together. So that's just going to be the flatten of map f down s. Each time I apply f to an element of s, I get a stream. If I map it all the way down, I get a stream of streams, and I'll flatten that.

Well, I want to use that to show you a new way to do a familiar kind of problem. The problem's going to be like a lot of problems you've seen, although maybe not this particular one. I'm going to give you an integer, n. And the problem is going to be find all pairs and integers i and j, between 0 and i, with j less than i, up to n, such that i plus j is prime.

So for example, if n equals 6, let's make a little table here, i and j and i plus j. So for, say, i equals 2 and j equals 1, I'd get 3. And for i equals 3, I could have j equals 2, and that would be 5. And 4 and 1 would be 5 and so on, up until i goes to 6. And what I'd like to return is to produce a stream of all the triples like this, let's say i, j, and i plus j. So for each n, I want to generate this stream.

OK, well, that's easy. Let's build it up. We start like this. We're going to say for each i, we're going to generate a stream. For each i in the interval 1 through n, we're going to generate a stream. What's that stream going to be? We're going to start by generating all the pairs. So for each i, we're going to generate, for each j in the interval 1 to i minus 1, we'll generate the pair, or the list with two elements i and j.

So we map along the interval, generating the pairs. And for each i, that generates a stream of pairs. And we flatmap it. Now we have all the pairs i and j, such that i is less than j. So that builds that.

Now we're got to test them. Well, we take that thing we just built, the flatmap, and we filter it to see whether the i-- see, we had an i and a j. i was the first thing in the list, j was the second thing in the list. So we have a predicate which says in that list of two elements is the sum of the CAR and the CDR prime. And we filter that collection of pairs we just built. So those are the pairs we want.

Now we go ahead and we take the result of that filter and we map along it, generating the list i and j and i plus j. And that's our procedure prime-sum-pairs. And then just to flash it up, here's the whole procedure. A map, a filter, a flatmap. There's the whole thing, even though this isn't particularly readable. It's just expanding that flatmap.

So there's an example which illustrates the general point that nested loops in this procedure start looking like compositions of flatmaps of flatmaps of flatmaps of maps and things. So not only can we enumerate individual things, but by using flatmaps, we can do what would correspond to nested loops in most other languages.

Of course, it's pretty awful to keep writing these flatmaps of flatmaps of flatmaps. Prime-sum-pairs you saw looked fairly complicated, even though the individual pieces were easy. So what you can do, if you like, is introduced some syntactic sugar that's called collect. And collect is just an abbreviation for that nest of flatmaps and filters arranged in that particular way. Here's prime-sum-pairs again, written using collect. It says to find all those pairs, I'm going to collect together a result, which is the list i, j, and i plus j, that's going to be generated as i runs through the interval from 1 to n and as j runs through the interval from 1 to i minus 1, such that i plus j is prime.

So I'm not going to say what collect does in general. You can look at that by looking at it in the book. But pretty much, you can see that the pieces of this are the pieces of that original procedure I wrote. And this collect is just some syntactic sugar for automatically generating that nest of flatmaps and flatmaps.

OK, well, let me do one more example that shows you the same kind of thing. Here's a very famous problem that's used to illustrate a lot of so-called backtracking computer algorithms. This is the eight queens problem. This is a chess board. And the eight queens problem says, find a way to put down eight queens on a chess board so that no two are attacking each other.

And here's a particular solution to the eight queens problem. So I have to make sure to put down queens so that no two are in the same row or the same column or sit along the same diagonal. Now, there's sort of a standard way of doing that. Well, first we need to do is below the surface, at George's level. We have to find some way to represent a board, and represent positions. And we'll not worry about that.

But let's assume that there's a predicate called safe. And what safe is going to do is going to say given that I have a bunch of queens down on the chess board, is it OK to put a queen in this particular spot? So safe is going to take a row and a column. That's going to be a place where I'm going to try and put down the next queen, and the rest of positions.

And what safe will say is given that I already have queens down in these positions, is it safe to put another queen down in that row and that column? And let's not worry about that. That's George's problem. and it's not hard to write. You just have to check whether this thing contains any things on that row or that column or in that diagonal.

Now, how would you organize the program given that? And there's sort of a traditional way to organize it called backtracking. And it says, well, let's think about all the ways of putting the first queen down in the first column. There are eight ways. Well, let's say try the first column. Try column 1, row 1. These branches are going to represent the possibilities at each level.

So I'll try and put a queen down in the first column. And now given that it's in the first column, I'll try and put the next queen down in the first column. I'll try and put the first queen, the one in the first column, down in the first row. I'm sorry. And then given that, we'll put the next queen down in the first row. And that's no good.

So I'll back up to here. And I'll say, oh, can I put the first queen down in the second row? Well, that's no good. Oh, can I put it down in the third row? Well, that's good. Well, now can I put the next queen down in the first column? Well, I can't visualize this chess board anymore, but I think that's right. And I try the next one.

And at each place, I go as far down this tree as I can. And I back up. If I get down to here and find no possibilities below there, I back all the way up to here, and now start again generating this sub-tree. And I sort of walk around. And finally, if I ever manage to get all the way down, I've found a solution.

So that's a typical sort of paradigm that's used a lot in AI programming. It's called backtracking search. And it's really unnecessary. You saw me get confused when I was visualizing this thing. And you see the complication. This is a complicated thing to say.

Why is it complicated? Its because somehow this program is too inordinately concerned with time. It's too much-- I try this one, and I try this one, and I go back to the last possibility. And that's a complicated thing. If I stop worrying about time so much, then there's a much simpler way to describe this. It says, let's imagine that I have in my hands the tree down to k minus 1 levels.

See, suppose I had in my hands all possible ways to put down queens in the first k columns. Suppose I just had that. Let's not worry about how we get it. Well, then, how do I extend that? How do I find all possible ways to put down queens in the next column? It's really easy. For each of these positions I have, I think about putting down a queen in each row to make the next thing. And then for each one I put down, I filter those by the ones that are safe.

So instead of thinking about this tree as generated step by step, suppose I had it all there. And to extend it from level k minus 1 to level k, I just need to extend each thing in all possible ways and only keep the ones that are safe. And that will give me the tree to level k. And that's a recursive strategy for solving the eight queens problem.

All right, well, let's look at it. To solve the eight queens problem on a board of some specified size, we write a sub-procedure called fill-columns. Fill-columns is going to put down queens up through column k. And here's the pattern of the recursion. I'm going to call fill-columns with the size eventually.

So fill-columns says how to put down queens safely in the first k columns of this chess board with a size number of rows in it. If k is equal to 0, well, then I don't have to put anything down. So my solution is just an empty chess board. Otherwise, I'm going to do some stuff. And I'm going to use collect.

And here's the collect. I find all ways to put down queens in the first k minus 1 columns. And this was just what I set for. Imagine I have this tree down to k minus 1 levels. And then I find all ways of trying a row, that's just each of the possible rows. They're size rows, so that's enumerate interval.

And now what I do is I collect together the new row I'm going to try and column k with the rest of the queens. I adjoin a position. This is George's problem. An adjoined position is like safe. It's a thing that takes a row and a column and the rest of the positions and makes a new position collection.

So I adjoin a position of a new row and a new column to the rest of the queens, where the rest of the queens runs through all possible ways of solving the problem in k minus 1 columns. And the new row runs through all possible rows such that it was safe to put one there. And that's the whole program. There's the whole procedure.

Not only that, that doesn't just solve the eight queens problem, it gives you all solutions to the eight queens problem. When you're done, you have a stream. And the elements of that stream are all possible ways of solving that problem. Why is that simpler? Well, we threw away the whole idea that this is some process that happens in time with state. And we just said it's a whole collection of stuff. And that's why it's simpler.

We've changed our view. Remember, that's where we started today. We've changed our view of what it is we're trying to model. we stop modeling things that evolve in time and have steps and have state. And instead, we're trying to model this global thing like the whole flight of the chalk, rather than its state at each instant. Any questions?

AUDIENCE: It looks to me like backtracking would be searching for the first solution it can find, whereas this recursive search would be looking for all solutions. And it seems that if you have a large enough area to search, that the second is going to become impossible.

PROFESSOR: OK, the answer to that question is the whole rest of this lecture. It's exactly the right question. And without trying to anticipate the lecture too much, you should start being suspicious at this point, and exactly those kinds of suspicions. It's wonderful, but isn't it so terribly inefficient? That's where we're going. So I won't answer now, but I'll answer later. OK, let's take a break.

Well, by now you should be starting to get suspicious. See, I've showed your this simple, elegant way of putting programs together, very unlike these other traditional programs that sum the odd squares or compute the odd Fibonacci numbers. Very unlike these programs that mix up the enumerator and the filter and the accumulator. And by mixing it up, we don't have all of these wonderful conceptual advantages of these streams pieces, these wonderful mix and match components for putting together lots and lots of programs.

On the other hand, most of the programs you've seen look like these ugly ones. Why's that? Can it possibly be that computer scientists are so obtuse that they don't notice that if you'd merely did this thing, then you can get this great programming elegance? There's got to be a catch. And it's actually pretty easy to see what the catch is.

Let's think about the following problem. Suppose I tell you to find the second prime between 10,000 and 1 million, or if your computer's larger, say between 10,000 and 100 billion, or something. And you say, oh, that's easy. I can do that with a stream. All I do is I enumerate the interval from 10,000 to 1 million. So I get all those integers from 10,000 to 1 million. I filter them for prime-ness, so test all of them and see if they're prime. And I take the second element. That's the head of the tail.

Well, that's clearly pretty ridiculous. We'd not even have room in the machine to store the integers in the first place, much less to test them. And then I only want the second one.

See, the power of this traditional programming style is exactly its weakness, that we're mixing up the enumerating and the testing and the accumulating. So we don't do it all. So the very thing that makes it conceptually ugly is the very thing that makes it efficient. It's this mixing up.

So it seems that all I've done this morning so far is just confuse you. I showed you this wonderful way that programming might work, except that it doesn't. Well, here's where the wonderful thing happens. It turns out in this game that we really can have our cake and eat it too. And what I mean by that is that we really can write stream programs exactly like the ones I wrote and arrange things so that when the machine actually runs, it's as efficient as running this traditional programming style that mixes up the generation and the test.

Well, that sounds pretty magic. The key to this is that streams are not lists. We'll see this carefully in a second, but for now, let's take a look at that slide again. The image you should have here of this signal processing system is that what's going to happen is there's this box that has the integers sitting in it. And there's this filter that's connected to it and it's tugging on them. And then there's someone who's tugging on this stuff saying what comes out of the filter.

And the image you should have is that someone says, well, what's the first prime, and tugs on this filter. And the filter tugs on the integers. And you look only at that much, and then say, oh, I really wanted the second one. What's the second prime? And that no computation gets done except when you tug on these things.

Let me try that again. This is a little device. This is a little stream machine invented by Eric Grimson who's been teaching this course at MIT. And the image is here's a stream of stuff, like a whole bunch of the integers. And here's some processing elements. And if, say, it's filter of filter of map, or something.

And if I really tried to implement that with streams as lists, what I'd say is, well, I've got this list of things, and now I do the first filter. So do all this processing. And I take this and I process and I process and I process and I process. And now I'm got this new stream. Now I take that result in my hand someplace. And I put that through the second one. And I process the whole thing. And there's this new stream. And then I take the result and I put it all the way through this one the same way.

That's what would happen to these stream programs if streams were just lists. But in fact, streams aren't lists, they're streams. And the image you should have is something a little bit more like this. I've got these gadgets connected up by this data that's flowing out of them. And here's my original source of the streams. It might be starting to generate the integers.

And now, what happens if I want a result? I tug on the end here. And this element says, gee, I need some more data. So this one comes here and tugs on that one. And it says, gee, I need some more data. And this one tugs on this thing, which might be a filter, and says, gee, I need some more data. And only as much of this thing at the end here gets generated as I tugged. And only as much of this stuff goes through the processing units as I'm pulling on the end I need. That's the image you should have of the difference between implementing what we're actually going to do and if streams were lists.

Well, how do we make this thing? I hope you have the image. The trick is how to make it. We want to arrange for a stream to be a data structure that computes itself incrementally, an on-demand data structure. And the basic idea is, again, one of the very basic ideas that we're seeing throughout the whole course. And that is that there's not a firm distinction between programs and data.

So what a stream is going to be is simultaneously this data structure that you think of, like the stream of the leaves of this tree. But at the same time, it's going to be a very clever procedure that has the method of computing in it. Well, let me try this. It's going to turn out that we don't need any more mechanism. We already have everything we need simply from the fact that we know how to handle procedures as first-class objects.

Well, let's go back to the key. The key is, remember, we had these operations. CONS-stream and head and tail. When I started, I said you can think about this as CONS and think about this as CAR and think about that as CDR, but it's not. Now, let's look at what they really are.

Well, CONS-stream of x and y is going to be an abbreviation for the following thing. CONS form a pair, ordinary CONS, of x to a thing called delay of y. And before I explain that, let me go and write the rest. The head of a stream is going to be just the CAR. And the tail of a stream is going to be a thing called force the CDR of the stream.

Now let me explain this. Delay is going to be a special magic thing. What delay does is take an expression and produce a promise to compute that expression when you ask for it. It doesn't do any computation here. It just gives you a rain check. It produces a promise. And CONS-stream says I'm going to put together in a pair x and a promise to compute y.

Now, if I want the head, that's just the CAR that I put in the pair. And the key is that the tail is going to be-- force calls in that promise. Tail says, well, take that promise and now call in that promise. And then we compute that thing. That's how this is going to work. That's what CONS-stream, head, and tail really are.

Now, let's see how this works. And we'll go through this fairly carefully. We're going to see how this works in this example of computing the second prime between 10,000 and a million. OK, so we start off and we have this expression. The second prime-- the head of the tail of the result of filtering for primality the integers between 10,000 and 1 million.

Now, what is that? What that is, that interval between 10,000 and 1 million, well, if you trace through enumerate interval, there builds a CONS-stream. And the CONS-stream is the CONS of 10,000 to a promise to compute the integers between 10,001 and 1 million.

So that's what this expression is. Here I'm using the substitution model. And we can use the substitution model because we don't have side effects and state. So I have CONS of 10,000 to a promise to compute the rest of the integers. So only one integer, so far, got enumerated.

Well, I'm going to filter that thing for primality. Again, you go back and look at the filter code. What the filter will first do is test the head. So in this case, the filter will test 10,000 and say, oh, 10,000's not prime. Therefore, what I have to do recursively is filter the tail. And what's the tail of it, well, that's the tail of this pair with a promise in it.

Tail now comes in and says, well, I'm going to force that. I'm going to force that promise, which means now I'm going to compute the integers between 10,001 and 1 million. OK, so this filter now is looking at that. That enumerate itself, well, now we're back in the original enumerate situation. The enumerate is the CONS of the first thing, 10,001, onto a promise to compute the rest.

So now the primality filter is going to go look at 10,001. It's going to decide if it likes that or not. It turns out 10,001 isn't prime. So it'll force it again and again and again. And finally, I think the first prime it hits is 10,009. And at that point, it'll stop. And that will be the first prime, and then eventually, it'll need the second prime. So at that point, it will go again.

So you see what happens is that no more gets generated than you actually need. That enumerator is not going to generate any more integers than the filter asks it for as it's pulling in things to check for primality. And the filter is not going to generate any more stuff than you ask it for, which is the head of the tail. You see, what's happened is we've put that mixing of generation and test into what actually happens in the computer, even though that's not apparently what's happening from looking at our programs.

OK, well, that seemed easy. All of this mechanism got put into this magic delay. So you're saying, gee, that must be where the magic is. But see there's no magic there either. You know what delay is. Delay on some expression is just an abbreviation for-- well, what's a promise to compute an expression? Lambda of nil, procedure of no arguments, which is that expression. That's what a procedure is. It says I'm going to compute an expression.

What's force? How do I take up a promise? Well, force of some procedure, a promise, is just run it. Done. So there's no magic there at all.

Well, what have we done? We said the old style, traditional style of programming is more efficient. And the stream thing is more perspicuous. And we managed to make the stream procedures run like the other procedures by using delay. And the thing that delay did for us was to de-couple the apparent order of events in our programs from the actual order of events that happened in the machine. That's really what delay is doing.

That's exactly the whole point. We've given up the idea that our procedures, as they run, or as we look at them, mirror some clear notion of time. And by giving that up, we give delay the freedom to arrange the order of events in the computation the way it likes. That's the whole idea. We de-couple the apparent order of events in our programs from the actual order of events in the computer.

OK, well there's one more detail. It's just a technical detail, but it's actually an important one. As you run through these recursive programs unwinding, you'll see a lot of things that look like tail of the tail of the tail. That's the kind of thing that would happen as I go CONSing down a stream all the way. And if each time I'm doing that, each time to compute a tail, I evaluate a procedure which then has to go re-compute its tail, and re-compute its tail and recompute its tail each time, you can see that's very inefficient compared to just having a list where the elements are all there, and I don't have to re-compute each tail every time I get the next tail.

So there's one little hack to slightly change what delay is, and make it a thing which is-- I'll write it this way. The actual implementation, delay is an abbreviation for this thing, memo-proc of a procedure. Memo-proc is a special thing that transforms a procedure. What it does is it takes a procedure of no arguments and it transforms it into a procedure that'll only have to do its computation once.

And what I mean by that is, you give it a procedure. The result of memo-proc will be a new procedure, which the first time you call it, will run the original procedure, remember what result it got, and then from ever on after, when you call it, it just won't have to do the computation. It will have cached that result someplace.

And here's an implementation of memo-proc. Once you have the idea, it's easy to implement. Memo-proc is this little thing that has two little flags in there. It says, have I already been run? And initially it says, no, I haven't already been run. And what was the result I got the last time I was run?

So memo-proc takes a procedure called proc, and it returns a new procedure of no arguments. Proc is supposed to be a procedure of no arguments. And it says, oh, if I'm not already run, then I'm going to do a sequence of things. I'm going to compute proc, I'm going to save that. I'm going to stash that in the variable result. I'm going to make a note to myself that I've already been run, and then I'll return the result.

So that's if you compute it if it's not already run. If you call it and it's already been run, it just returns the result. So that's a little clever hack called memoization. And in this case, it short circuits having to re-compute the tail of the tail of the tail of the tail of the tail. So there isn't even that kind of inefficiency. And in fact, the streams will run with pretty much the same efficiency as the other programs precisely.

And remember, again, the whole idea of this is that we've used the fact that there's no really good dividing line between procedures and data. We've written data structures that, in fact, are sort of like procedures. And what that's allowed us to do is take an example of a common control structure, in this place iteration. And we've built a data structure which, since itself is a procedure, kind of has this iteration control structure in it. And that's really what streams are. OK, questions?

AUDIENCE: Your description of tail-tail-tail, if I understand it correctly, force is actually execution of a procedure, if it's done without this memo-proc thing. And you implied that memo-proc gets around that problem. Doesn't it only get around it if tail-tail-tail is always executing exactly the same--

PROFESSOR: Oh, that's-- sure.

AUDIENCE: I guess I missed that point.

PROFESSOR: Oh, sure. I mean the point is-- yeah. I mean I have to do a computation to get the answer. But the point is, once I've found the tail of the stream, to get the tail of the tail, I shouldn't have had to re-compute the first tail. See, and if I didn't use memo-proc, that re-computation would have been done.

AUDIENCE: I understand now.

AUDIENCE: In one of your examples, you mentioned that we were able to use the substitution model because there are no side effects. What if we had a single processing unit-- if we had a side effect, if we had a state? Could we still practically build the stream model?

PROFESSOR: Maybe. That's a hard question. I'm going to talk a little bit later about the places where substitution and side effects don't really mix very well. But in general, I think the answer is unless you're very careful, any amount of side effect is going to mess up everything.

AUDIENCE: Sorry, I didn't quite understand the memo-proc operation. When do you execute the lambda? In other words, when memo-proc is executed, just this lambda expression is being generated. But it's not clear to me when it's executed.

PROFESSOR: Right. What memo-proc does-- remember, the thing that's going into memo-proc, the thing proc, is a procedure of no arguments. And someday, you're going to call it.

Memo-proc translates that procedure into another procedure of no arguments, which someday you're going to call. That's that lambda.

So here, where I initially built as my tail of the stream, say, this procedure of no arguments, which someday I'll call. Instead, I'm going to have the tail of the stream be memo-proc of it, which someday I'll call. So that lambda of nil, that gets called when you call the memo-proc, when you call the result of that memo-proc, which would be ordinarily when you would have called the original thing that you set it.

AUDIENCE: OK, the reason I ask is I had a feeling that when you call memo-proc, you just return this lambda.

PROFESSOR: That's right. When you call memo-proc, you return the lambda. You never evaluate the expression at all, until the first time that you would have evaluated it.

AUDIENCE: Do I understand it right that you actually have to build the list up, but the elements of the list don't get evaluated? The expressions don't get evaluated? But at each stage, you actually are building a list.

PROFESSOR: That's-- I really should have said this. That's a really good point. No, it's not quite right. Because what happens is this. Let me draw this as pairs. Suppose I'm going to make a big stream, like enumerate interval, 1 through 1 billion. What that is, is a pair with a 1 and a promise. That's exactly what it is. Nothing got built up.

When I go and force this, and say, what happens? Well, this thing is now also recursively a CONS. So that this promise now is the next thing, which is a 2 and a promise to do more. And so on and so on and so on.

So nothing gets built up until you walk down the stream. Because what's sitting here is not the list, but a promise to generate the list. And by promise, technically I mean procedure. So it doesn't get built up. Yeah, I should have said that before this point. OK. Thank you. Let's take a break.