TOMER ULLMAN: And today, with your active help and participation, I hope to run a probabilistic programming tutorial in the time that we have left. And we're going to focus specifically on a language called Church, which is a probabilistic programming language that was developed in Josh Tenenbaum's Group, but it's now taken on a life of its own and has set up shop in other places.

Before I get started, I should say, I was, sort of, looking for a good image. I didn't like a blank page, so I was googling just Church tutorial. This is the first thing I found. It's an image for *Minecraft* about how to build a church in *Minecraft.* Does any of us-- people have heard of *Minecraft?*

AUDIENCE: Yeah.

TOMER ULLMAN: They've played with *Minecraft?* OK-- just in case you don't know, *Minecraft* is a sort of procedurally-generated world where you get some building blocks, literally, building blocks, that you can build stuff with. And you can build an infinite number of things, including a computer, and a church. And it's very cool. And I thought it's actually not that bad of an image for a tutorial about probabilistic programming language, which is also about, sort of, procedurally-generative things that use small building blocks to build up an entire world.

And I thought, OK, that's the first hit I got. What's the other hit? Well, it's just another church, and another church, and another church, another church, another church that you can build in *Minecraft* from many different angles and many different tutorials, so maybe, instead of this, you can just train some deep-learning algorithm to, I don't know, learn a billion churches and do that. That's not what we're after.

So probabilistic programming, Josh already talked a bunch about this, so I'll sort of be repeating him, or channeling him. It's about combining the best of both worlds in the, sort of, two states of AI right now, which is statistical modeling and logic. And in many models, you have this, sort of, dual question of representation and learning.

And it's really, sort of, a problem for cognitive science, going back to the days of before cognitive science, right? I mean, this is the sort of problem that a lot of people had when they tried to model the human mind. This goes back to Turing.

Sort of, when we want to build a system that is human-like in its intelligence, the two questions that we face are, what are the representations that it will have, and how is it going to learn them or how is it going to learn anything new? And you often have to, sort of-- it's a short blanket problem, right? If you try to cover your head, your feet are sort of not getting anything. If you try to cover your feet, your head's not getting anything.

Because oftentimes, you find that, if you stick to a particularly easy representation that's sort of easy to code or rather something, a kind of a presentation that's easy to learn, like say a vector of weights that you're just trying to shift your weights around, then, yes, that might be easy, relatively easy, but you're sort of stuck with the representation that you can learn are weights. Or Josh was making a big point about this, and it is a big point, that if you try to learn something like causal Bayes nets, then you're sort of limited by that representation.

That is your representation of these sort of circles and arrows that go into other circles. And that might get you very, very far. And you might even have very good learning algorithms for those particular models, for those particular representations, that are tailored for those representations. Like, in these causal circles and arrows, belief propagation might be a very good learning algorithm, but if you commit to that representation, then you are sort of stuck with that representation. And you might not be flexible enough to learn all the stuff that you want to.

And a very flexible representation, sort of, one of the more flexible ones that have come onto the scene in the past years is why don't we try to learn a program? I say it's come onto the scene in recent years. That's not exactly true. People have been interested in learning programs for many, many years, for many, many decades, but they sort of try to infer them from kind of a logical perspective, not really getting these probabilistic learning algorithms.

I'm sort of throwing words out there, but it'll make more sense as I go through it. And you already have some of Josh's stuff to carry you through.

But the point is, there's always these questions of learning and representation. For probabilistic programming languages, the representation is not circles and arrows, it's not

vectors of weights, it is programs. That's what you're trying to learn. That's what you're trying to figure out the world with. And then there's a question of how do you learn these programs, but we'll get to that.

OK, let's see, so we think it's a good representation for AI and cognition for all the reasons that Josh just talked about. And there's been a growing interest in these things for the past 10 years, witnessed both by the proliferation of many, many different types of programming languages-- sorry, probabilistic programming languages. I don't know whether to call them PPL, or people, or what exactly.

But probabilistic programming languages, there's been PyMC based on Python, there's Church, which you're going to play with right now, but also BLOG, WinBUGS, ProbLog, Venture, many others that I haven't mentioned here. So first of all, there's many of them. And also, DARPA has started taking interest and has given a large grant to advance this field. They think it might be big.

If you're in, sort of, probabilistic programming more generally than Church, you think it's interesting to follow, you want to learn more about it, you should go to this thing, probabilistic-programming.org wiki. It sort of keeps it up-to-date with many, many, many, different types of programming languages. You don't necessarily have to write this down right now. I will send you the slides later, but just, sort of, keep it in mind form, link to it in your head.

There's also this, sort of, nice summary from this DARPA-- so DARPA started sending this about a year ago. And someone already went to a summer school on probabilistic programming and, sort of, wrote the state of the field. It's six months ago. It's a bit outdated, but it also makes for an interesting read for those of you who want to follow that. OK, so that's about probabilistic programming, very, very, very generally.

What about Church very, very, very generally? So as I said, Church is one example of a probabilistic programming language. It was developed by several people at MIT who have since gone on to do other different things like continue to develop Church at Stanford. That's Professor Noah Goodman. Although, of course, he's doing many, many other things. There's also been Vikash Mansinghka, who has gone on to develop other probabilistic programming languages like Venture at MIT.

And one thing to say generally about probabilistic programming languages is that, usually they are based on an already existing language. So you take MATLAB and you try to make it

probabilistic. You take Python and you try to make it probabilistic. Julia has a probabilistic programming implementation.

Church in particular is based on Scheme, which is the derivative of LISP, which is itself sort of an attempt to capture lambda calculus, which is not a programming language, it is an approach to trying to think about all possible functions developed by Alonzo Church. And that's why Church is called Church. It has nothing to do with the actual buildings.

So the point about Scheme which is very nice is that it's very compositional. And anything that you write can then be passed off into the other functions as the data. You'll see some examples of that. Church has several inference engines that you can try to run. We'll get into that. The backbone of it is Metropolis-Hastings-type sampling over possible programs, but it has other types of programming, including explicit enumeration.

If your space is small enough, it can just look at all the possible ways to run a program. It has rejection query. Again, we'll get to this. Don't worry about, like, what is he talking about. Yeah, so it has a whole bunch of-- you know, particle filtering is one attempt at that.

But the point is there are-- each probabilistic programming language has its own set of inference engine. Some of them try to go the Metropolis-Hastings route. Some of them try to say, well, it's a probabilistic programming language, but it's really limited to causal Bayes nets, so the inference engines are going to be stuff that's good for causal Bayes nets.

But all of them sort of share this dream of, it's easier to write the forward model than the inference. And it's really annoying. Those of you who have ever tried to write an inference engine or to write inference over any sort of model, it's really annoying to write that.

And it usually sort of only works for the one thing that you've built. And one of the selling points of probabilistic programming languages, one of the reasons that DARPA took an interest, beyond the fact that they can try to capture the human mind, and flexible AI, and all that, is they have this sort of promise, this pitch that, why don't you just write down the forward model, how you think the world works, and we'll, kind of, take care of inference for you.

And in many cases, it turns out to be a lot easier to write the forward model than to try to write the inference engine for it. In fact, you can very quickly get to something that's even, like, five or six lines of code long, that would be intractable, would be very hard to write down the analytic expression for, would be very hard to think about what would be the inference engine

for, but it's really just easy to write.

I mean, all you have is a set of assumptions. And you're trying to figure out how they work together. Again, we'll see some examples of that. But my point was all probabilistic programming languages are about writing the forward model and then, sort of, trying to do the inference for you.

Another point about Church in particular, it is under construction, so you'll notice this when you write it now. It will break. It will freeze. It will do all sorts of annoying things, so it is under construction. It's not exactly something that you would then go and work with like MATLAB.

Let me put some caveats on that caveat, which is these two asterisks right here. First of all, despite being a, sort of, a toy language, it's already been used in several serious scientific papers, including a paper in *Science,* because it is very easy to make certain points about cognition or about computational cognition in Church that is very hard to do in certain other languages. In particular, things that require recursion, or inference over inference, where you write down sort of the way that you think about an agent, then you put that into another agent, that can be very hard to write in certain languages. Church can kind of do that more easily.

Let's see, I had another caveat, which is-- what was that? Oh, another caveat is that, despite it being under construction, you sort of think, well, why should I worry about this thing? Why should I even bother hacking with it? Is because, you'll notice there's probmods.org. And there are just a ton, a ton of examples. There's a semester worth of examples of all sorts of things from both cognition, and AI, and interesting statistical models that are very easy to understand in Church.

And for me at least, it was very much a process of demystification that something like this can help with. You learn about something like the Chinese restaurant process, the Dirichlet process, nonparametrics, and it's kind of hard to read the textbook description of it. It's hard to wrap your head around.

And then you go and you write three lines of code, or five lines of code. And you think, oh, that wasn't so bad, right? And it's sort of easy to write a bunch of these things in Church, so it's a useful tool for demystification. It's a useful tool to get a handle on certain models in cognition and statistics, so those are the two asterisks.

Be warned, but also, you know, do play around with it. Let's see, the founding paper, for those

of you who are interested, you can look at this link later on. It was by Goodman, Mansinghka, Dan Roy, Bonawitz, and Tenenbaum.

And for those of you who, by the way, have already read about Church a bit, you think that this tutorial is a bit-- maybe it was-- I should say, we'll start off very, very easy, OK? We'll do things like addition. We'll do things like flipping coins, OK?

If you think that this is-- maybe you've already read through probmods, you've already done a few chapters of that, by all means, use this time to continue to think about probabilistic programming, for example, either by talking to me, and I'll find something for you, or by going to forestdb.org. Again, I'll give you that link for those of you who want it. It has a whole repository of different probabilistic programming models that you can play with, think about, see how you would change them, and things like that. Also after this tutorial, if you're still interested, you can go to that link.

Oh and one last thing. There's sort of a-- you can't see that right there. One last thing that I should say about Church, it's based on Scheme. But a lot of the people that have sort of been doing a lot of work on it have become more in love with JavaScript. In fact, the thing that you're going to be working on is sort of a JavaScript implementation of Church under the hood.

And they've started to implement something called WebPPL, so Web Probabilistic Programming Language. It's a language that's specifically a derivative of JavaScript. For those of you who like JavaScript, you can play with that.

And if you go to WebPPL.org, if you search for WebPPL, again, I can leave you the link for that. It's sort of here, but you can't see it. There are, again, a lot of nice examples there of different programming language-- programs that you can write in JavaScript. OK, that was a very long-winded introduction, caveats, and setting up different things.

The objectives for this tutorial is, first of all, to become familiar with the Church syntax, it can be a little wonky, if you don't know it, at first, to run forward a few models to give you an example of just, before inference, an example of, here's my forward model, here's how I describe the world, now let's try sampling from it. Let's sample, sample again, sample again, sample again, see what distributions we get.

Get a sense for the point that I'm going to make a few times, which is once you write your

forward model, that is a representation of a distribution-- and I'll come back to this point, but just, sort of, keep that in mind. You write down a program. And you run it forward. And you get a sample. You run it again and you get a different sample. You run it in the limit, you get some distribution.

Some other constructs like memoization-- after we do all of this, we'll try to get at sampling, and the query operator, and really, conditioning and inference. So we said we'll try to run a few models forward. Once we do that, we'll try to get the hang of inference.

So you'll try to write down a forward model about things like a coin, or goal inference, or things like that. And you'll try to actually infer something, like what is the weight of the coin, from some data, like some coin flips, some very simple stuff. OK, and we'll go through some examples, like, as I said, coin flipping, maybe causal networks, maybe intuitive physics and intuitive psychology. I do hope to get to intuitive psychology. We'll see if we get to that.

So some prerequisites and set up, that's what I asked you to do at the beginning. If you happen to have a local implementation, you can open that now. If you didn't, just go to probmods.org/play-space.html and open that up. And we're going to play a game of Noisy Tomer Says.

So now you should also-- open this, open a browser, go to that, or open your local implementation. Also open up the file that I sent you of-- it should have have been called, like, student copy, something like that. It contains a bunch of things that we're basically going to just sort of copy, paste into the browser. Now, the nice thing about this browser is, it is sort of a working implementation of Church. You just paste in the code. You hit run. It runs, OK?

So you guys should all more or less have a screen like this. I'll take this out so I don't sit on it right now. Does everyone have more or less something like this, some sort of browser that you can type things into and press run? Over there?

OK, we'll start off with some very, very simple stuff that you should already have in the syntax of the Church tutorial, so just try either pasting in or typing in things like this thing. So the first thing you'll notice is that, over here, it's what's called-- sorry, let me adjust this screen so it's not actually-- so that you can see it. Zone C over here, you should be looking-- I've sort of done over here, plus 2 2, and the result is 4.

So the first thing to see, some of you may be familiar with this, who's somebody with Polish

notation, where you just go plus 2 2? Instead of going 2 plus-- who is not familiar with Polish notation? OK, good, thank you. Polish notation just means that, instead of writing 2 plus 2, you write plus 2 2, so you write that the thing that operates, the function, outside, and you write all the arguments for the function like that.

In fact, most of the time, you do this. When you write down functions for code, you usually write the function then the things that it operates on. But here, it's going to work for everything. And it can be a bit confusing at first when you do things like plus 2 2.

The second thing is that you put brackets on anything that you want to evaluate, OK? So, for example, here is an expression. The expression is plus 2 2. And you want to evaluate that expression. So for example, I wanted to evaluate the expression-- I think I put some, like, cursor for-- so you can see what I'm doing with my thing.

OK, if you want to do something like, you know, times 2 2, that would be the same thing. And I would go to run. And that would be, of course, 4 again.

It let's you do some other examples from here. Like there's a bunch of simple logic, like you might do display. Display is just a way to run it, to-- sorry, to display the result over here. You can do a bunch of logic things, like equal.

So again, the operator is outside. And you would do equal question mark 2 2, and then evaluate that expression. And you can do bigger than equals, all these different things.

AUDIENCE: the question mark?

TOMER ULLMAN: The question mark is just-- I've just named it that way. It doesn't actually have any sense. I could have just called it equal-- sorry, no, sorry.

There is no particular meaning to the question mark. It's just that this thing, this operator, is called equal question mark. That's the name for it. And it's just-- it is the equals operator. That's how you check if two things are equal to one another. In languages like Python, you would do, you know, equals equals, like that. This is how you do it here, OK?

Let's see, a few other simple syntax things. So you might say, for example, the statement for defining variables is, shockingly enough, define, so you would do define x 3. And now, the next time that I do x, then hopefully-- and I run that-- then it'll show 3.

There are a few other basic syntax things, like lists, that might be important, like, you know, define x to be a list of 1 2 3. And if you run that, then you'll get 1 2 3. Again, we're starting out very, very slow, but we'll hopefully get soon to more things like Gaussian processes.

Some simple things like if-then statements-- OK, I'm just copying and pasting off of this document that you should all have, so that's why I'm, sort of, running through it. But the point is that you would do-- the syntax for doing an if-then conditional statement is like this. You write down if, and then you write down the condition that either evaluates to true or to false. So it's if this condition, do the first thing. If it's false, do the second thing.

In this particular case, I have defined a variable called socrates. I've defined it as drunk. And then I run the condition equal socrates drunk, if that's true, then return the answer true. Or, you know, I could have written return the answer, Socrates is a drunk.

If it's false, return the answer false. Did everyone more or less get the conditional? It just says, if condition, return the first thing otherwise, the thing on the second line.

Another important thing before we start getting at more things like recursion and forward sampling is the notion of, how would I define a function? So, so far we've defined variables, right? I could have defined something like define x 2, right? And then that would have just been that.

But I want to define, probably, functions, so I might define something like define-- and now I have two options. There are two ways of defining functions in Church. One of them is to do the following. You define square. And then you say, well, square is, itself, a procedure. It is a lambda. And I'll explain this as I go along, just watch me, sort of, type it. It takes in a particular argument, say, x. And then what it does to, is it multiplies x by x.

So the point is, you say, well, here, x is a particular thing. It is an object. What is it? It is just 2. Here, square is a thing. What sort of thing is it? It is this thing. Ah, what is this thing? This thing is a procedure that-- this is the only thing that you need to know about functions.

Lambda is the thing that actually defines functions, OK? It is a procedure that takes in some number of arguments, in this case, just one argument. You could have called it anything. I just called it x. You could have called it argument1. You could have called it socrates. You could have called it fubar.

But the point is, it takes in this argument. And then what does it do with it is the next thing? So

you say, lambda, number of arguments that you take in. And then what do you do with it? In this case, you just do times x x. So this is a function called square, very basic stuff. It takes in an argument and it multiplies it by itself, so it is the square of x, x times x, very simple.

There's another way of doing that if you don't want to type out lambdas, if you don't want to start doing lambda this, lambda that, it's sort of annoying. Let me just give you one more example. Like, if I wanted something with two arguments, I could have done-- you know, I could have called it something like my-proc lambda x y. And now, what it does is, it multiplies xy.

OK, this is an example of a thing. What sort of thing is it? It is a procedure. I know it's a procedure because it starts with lambda.

It takes in two arguments. Here they're called x and y. What does that do with it? It multiplies x times y. Really, this is just multiplication. So after I define this procedure, I could then do, like, my-proc-- sorry, I should have explained that. Then you do my-proc, say, 2 8, or something like that.

AUDIENCE: [INAUDIBLE]

TOMER ULLMAN: Yeah, sorry-- that's a very good question. And it would bring back 16. Sorry, once I define my thing, this is an operator now. This is an operator that can be applied to arguments. And you apply it by doing that parentheses that we just saw.

If I just tried, by the way, like, without applying it, if I just tried something like this, what you would get back is, it would say, this is a function, because it just says, what is this thing? You try to evaluate it.

You're not evaluating on anything, so it just returns, what is this thing? It's a function. It's a function that expects x y and then multiplies them. If you actually want to apply it on something, you would need to provide with some input arguments.

So I said, let's try to define square as a lambda of x. That does-- it takes in an x and multiplies x by x. There's one more way to define a function, which, it sort of gets rid of this lambda type thing. It's exactly equivalent to the thing I just showed you, it just takes a bit less writing, which is to say, define-- I just misspelled square, didn't I? Yes.

Define square x-- like that-- times x x. Now what this is saying, so this just goes straight to

saying, like, before I would say, define this thing 2. OK, and then I said, define this thing, the square, as this procedure.

Here you can say, I want to directly define a procedure. I'm not going to bother with this lambda stuff. I want to directly define a function. I want to directly define a procedure.

Can I do that? Yes, you could if you wanted to. You would just directly put these brackets right there. You would say define. And if the next thing is some brackets, then it says, OK, I'm going to define a procedure where the name of the procedure is square.

And it takes in one argument, which is x. And what it does to it is times x x. And if you do it that way, then under the hood, what Scheme does is actually writes it out like this. It puts in the lambda where it expects, but again, this is not terribly important stuff.

And those of you are, sort of, tuning out, and saying, well, fine. And you just wanted to learn about-- a bit more about how probabilistic programming works, don't worry. We'll get to some examples in about 10 minutes. Here's another very useful thing that you might want to do in many of your things.

This is called the map. And the way map works is, you map a function to a bunch of arguments. So you would say-- map is just a high-level function which takes in a particular procedure. Then it applies it to each one of these things individually, OK?

So square, in this case, as we said, it is a thing that takes in one argument. So this is now going to take square and apply it to 1. So then I'm going to take square and apply it to 2, take square and apply it to 3.

And the result of this is just going to be a list of squares, 1 4, 9, 16, 25, simple enough? Yes. But map is very useful. You should probably know about it.

OK, some simple things like, recursion, OK, so suppose I wanted to apply square to the list from 1 to 100, and suppose I didn't have the range 1 to 100. Most languages in Scheme actually does have something called range, which gives you all the numbers from 1 to 100.

Suppose I didn't. Suppose I want to construct all the numbers 1 to 100. I don't want to actually write them down-- 1, 2, 3, 4, 5, 6, all the way up to 100. I can write down something that does that. And it uses a little bit of recursion.

And the way it does it is this. This is just to get you used to recursion, because we'll be seeing it a little bit later. And this says, OK, I'm going to define something called range, which takes in an argument-- you should now be used to it, this is the same thing that we defined over here. We're going to call something a procedure. And we're going to call-- we're going to define a procedure.

It's called range. It takes in an argument, n, one argument. What does it do? Well, it depends. It does a conditional. A conditional, it depends, let's see, is n equal to 0? If it's 0, just give me back an empty list. Does everyone sort of see that, if equal n 0, give me back a list.

What if it's not 0? What if I did range 10? Oh, well, in that case, append-- another thing that you might want to know, so it's just combine these two things-- append what with what? Append range again with n minus 1 and with n.

The point here is to say, OK, how do I get the numbers 1 to 100? I just, sort of, say range-- I want the range 1 to 100, so I say, 100-- am I at 0 yet? No, so take 100 and append it with range 99. What does range 99 do? Well, is 99 0? No, so give me back 99 plus what? Plus range 98. Is 98 0? No, keep going, so it's basically recursing-- range is a recursive function that calls itself until it hits 0, very simple recursion.

And now you can do this to write out all the numbers from 1 to 100. And then you, if you were so inclined, you could do math square to that. OK, and we run that. And it gives me all the numbers from-- the squares of the numbers from 1 to 100.

So far we've talked just about very basic stuff. This is no different from Scheme. You are all experts in Scheme notation and things like that. Let's move on to something a little bit more interesting that Church can do, which is, for example, take random sequences, and it can take random-- how should I put this?

Kind of like plus is a basic thing in certain programming languages, it's a primitive, right? It's written into the language what plus means, what times means. You don't have to define that.

The way most languages work is that they have this sort of long list of things that they need to evaluate. And they start evaluating them. And they're, sort of, OK, did I hit an expression I know, like a number or not? And it's, sort of, no, you didn't hit it yet. OK, fine, keep evaluating, keep evaluating, keep evaluating until you get some sort of primitive. And a primitive procedure could be something like plus or a number.

In Church, there are primitive procedures which are random primitive procedures. They are procedures that, when you hit them, what you do is, you just return a value, a sampled value, from this expression, from this probability distribution. So the most basic random primitive, the most basic distribution that you can do in Church is something called flip. And if you just write down flip in Church, what you'll get, if you run it like that, is it tells you, well, it's a function.

And it depends on certain arguments. And it tells you many, many things about it, but that's not what we want. We want to evaluate it, so put some parentheses around it. And we'll run it.

And it will give us back false. OK, let's try that again. So let's run that again. It will gave us back true, OK, interesting. And if we run that again, you know, we get false. We run it again, and we get maybe true, maybe false.

You could do repeat 1,000 times flip. OK, repeat is another important thing that you would need to know. It just says repeat as many times as you want to repeat some sort of function. In this case, the function is flip. OK, so repeat flip 1,000 times.

I hope you guys are trying this while I'm saying this. Are people trying this more or less? OK, cool. So repeat 1,000 times flip. And what you'll get back is this long list of true, false, true, false, false, true, false, true. And it's independent from one another, because it's an exchangeable random sequence.

And if you want to see what this looks like, well, you could just do something like hist. And you would run that. And you would get, you know, more or less 50-50. Not exactly 50-50, because I only ran it 1,000 times. If I had run this in the limit, what I would get is 50-50 on true-false.

Now, what's nice about this is that this sort of gets at this thing that I was talking about earlier, where there's dual representation for any sort of probability distribution. You could either write the probability distribution in math. You could sort of say, well, the probability of true is 0.5. And the probability of false is 0.5.

Now I've defined a distribution in math. And now you can say, well, what's conditioned on this, what can you do, and things like that. Or what you can do is, you can write a program such that, when you run it, it will sample one of these values. And in the limit, it samples it's such that it approximated the thing that we just defined in math.

And you might say, well, why not just define it in math? Because oftentimes, it gets very, very,

hairy very, very fast. And in fact, any sort of probability distribution that's well-defined and well-behaved, you can write as a program. A program which, if you run it many times, its sampling profile, the thing it will give you back if you sample it many, many different times, will give you back that probability distribution.

Or you could equivalently say that, what it means for a probability distribution to be a probability distribution is to be some sort of program, to be some sort of procedure that gives you back a sample. And in the limit, you get some sort of thing that we're going to call the probability distribution. Actually, that's the way we define the probability distribution.

And again, this gets in-- so one way to think about Church programs is that any Church program that you write-- if you just write plus 2 2, you'll get back 4. That's, in a way, a deterministic program, right? The probability of getting back 4 on this execution equals 1, but there are many other things that you could write and you could get back interesting things for them.

And the point is to write something like a generative model that describes some sort of thing about the world. And when you run it forward, you get to a certain sample, but if you run in many, many different times, it gives you the probability distribution that this model describes. And now, if you-- and again, I'm getting slightly ahead of myself.

If you change that model, if you, for example, condition on something, you'll get a different model. You'll get a different program. And you're trying to find the program such that its output will match the data.

OK, but let's back up a little bit. And we're still in flip land. So we have here something which is flip. That's very, very basic. Flip can also be--

AUDIENCE: [INAUDIBLE]

TOMER ULLMAN: OK. Flip can also be a biased coin. So for example, if I do-- I define something like, you know, define-- let's do this slightly differently. Let's call this lambda something. And what it does is flip 0.9.

So if you run this forward, what you'll get now is that flip can actually take in some arguments. If you don't give it any arguments, it'll just do flip 50-50. If you give it some arguments, it'll do flip a biased coin, where the coin is biased towards 0.9. And you can see that, after I repeated that 1,000 times, I get, you know, it's approximately 90% heads, or true, and about 10% tails.

AUDIENCE: Why did you make the lambda in there?

TOMER ULLMAN: Ah, perfect, I'm glad somebody has asked that question. So if I were just to do the following-- suppose that I were just do repeat flip 0.9 like that, think about what would happen. What would happen is, I would first evaluate flip 0.9.

OK, that would give me back a value, either true or false. And then this would say, repeat that 1,000 times. You would get, like, 1,000 trues, or 1,000 falses, or whatever it was that was first. In fact, it's going to fail, because repeat expects a function.

But the point is, the reason that this is going to fail is because it wants a particular function. This is not a function, this is a value. You evaluate this first. It gives you a value like true or false. And then you repeat that value 1,000 times.

That's not what you want. What you want is a procedure. A procedure, or a distribution, or something like that, some sort of function that, when you run it, you get a biased sample, so what would that look like? That would look like this.

It would be-- or I could do something like this. Define my-coin weight-- OK, something like this. And what it does is this. Now what I've defined is, I've defined a procedure that takes in a particular weight. And what it does is that it gives you back a flip on that weight.

AUDIENCE: [INAUDIBLE]

TOMER ULLMAN: Yes, although you might, again, run into some problems, but we can get to that, because-- well, OK. So let's see--

AUDIENCE: How would define it as a lambda calculus?

TOMER ULLMAN: OK, so how you would define it with the lambda calculus is, you would say my-coin lambda weight this thing. OK, now we're saying, what sort of thing is coin? Coin is a procedure. How do we know it's a procedure? Because we have this lambda right here.

How many arguments does it expect? One, it's called weight. What does it do? It flips a coin. It gives you back that sample.

AUDIENCE: Can I do--

TOMER ULLMAN: The equivalent way of doing that is by writing this thing without any lambdas. You would just write define my-coin-- notice the brackets there, right? Before we didn't have brackets around that-- define my-coin weight flip weight, like that. And now you're sort of saying, like, this is a procedure. You should know it's a procedure, because it's the first thing that you're hitting after define because of the parentheses. What sort of procedure is it? It's called my-coin. It takes in weight. Again, these are equivalent. And to answer Nori's question about how would I just do that without having to define things, I would say something like, hist repeat 1,000. Now, what do I want to repeat? I want to repeat some sort of procedure that samples things. So it's-- I'll call it lambda. It's an empty lambda. It doesn't take in any arguments. It's just the procedure. And what it does is, it flips a coin 0.9. And if I run that, I'll get that. OK, yes, no? OK, good.

OK, so let's see, there are many other primitives that we could get to. There is uniform-draw. You can look at this online, but there's-- the basic primitives are things like multinomial, uniform, random integer, beta, Dirichlet, there's also the Chinese restaurant process.

So let's see, we can build in our own little distribution. OK, let's try doing that. So here I've defined something which, under the hood, it's actually-- it's an interesting distribution. You all probably know it. But the way I'm going to define it is, I'm going to call it times it counts until heads.

This is a procedure that's going to flip a coin. And if it comes up-- it's going to flip a coin with a particular weight. If it comes up true, if it comes up heads, then it's just going to stop. It's going to give you back 0.

If it doesn't stop, if it comes back tails, it's going to tell you that. It's going to write down somewhere, like, 1. And it's going to keep going. It's going to recurse somehow, call itself, and then keep going.

So this is for you, this is an exercise for you. You have it under the files, under 3.4, build your own distribution. I've left this open. Why don't you take two minutes. We're trying to build a procedure that gives me the amount of times that I need to flip a coin before I get back heads, OK?

If I take a particular coin-- I guess I don't want to have one handy-- but I flip a coin. And I just-- you know, I flip it. If it comes back heads, I write down 0 and I'm done. If it comes back tails, I'm going to keep flipping it, so I flip it again. And you know, I might flip it 10 times until I get heads, so the point is that this procedure will, in that case, return 10. That would be one

particular sample.

Now, of course, if I take the coin again and I flip it again, sometimes I get 10 times until heads, sometimes once, sometimes 5, sometimes 20, so I'm going to get a particular distribution on the number of times I need until I hit heads. And the thing that we're trying to implement right now is just a procedure that, what it does is, it implements this counting thing that I just said by literally flipping a coin-- well, I don't know if literally, but under the hood, flipping a coin.

If the coin comes back heads, because this thing evaluates to true, give back 0. If it doesn't, give back plus 1 plus what? So fill in those dots-- it shouldn't be a long expression-- such that you'll get what I was just talking about.

So, guys, let me tell you what I was going for. An int plus 1 countsTillHeads coinweight. OK, and now if you do something like countsTillHeads, I don't know, 0.1 or something like that, and you run it. And it gets saved-- so let's read through this for a second.

What happens is, you defined a procedure. It's called countsTillHeads. It takes in a coin weight. It flips a coin. If it comes back head, it gives you back 0. If it didn't come back heads, then you just do plus 1. And then you just call that thing again.

You do countTillHeads coinweight again and again. If it comes back 0, then this time, you'll have plus 1 plus 0 if it came back heads in here. But if it didn't, then this will be plus 1 plus something.

In effect, what we've defined here-- those of you that have defined it, and if not, just look at this-- what you've defined here is sort of a procedure that might give us back infinity in some way, except it's becoming extremely unlikely to do so with each particular flip of the coin. Now, I run it once with 0.1. I get 15. I can run it again and I'll get, you know, 8. That just means that, on that run, I flipped it eight times before I got heads.

And again, I can do this many, many different times. Like, I can do hist repeat 1,000 and then this thing, some empty procedure that does that. And what you gets is this, which, in case it doesn't look familiar-- sorry, it's just the way these things usually look. This is sort of flipping the x- and y-axis.

But the point is, how many times did I have to flip it to get, you know-- how many times did it happen? Did I flip it three times, or one, or two, three times? That's about 24%. And it sort of

goes down, and down, and down, because it becomes much, much, much more unlikely that I'll flip it 40 times until I get heads. It could be that I'll keep flipping it to infinity, but it's not going to happen.

This, in case you didn't know, falls off geometrically. It's the geometric distribution. That's a very fundamental, simple distribution. And one way to write it is to say, what's the probability of k? The probability of k is-- let's say, we have a coin which has the-- it's probability of coming up heads is p.

Then we say the probability of k is p to the k minus 1 times 1 minus p, yes? It's I flip the coin 1 minus p times to the k. The point is, you can define the geometric distribution by sort of saying, what's the probability of any particular number?

Or you can define the procedure for it, OK? Instead of writing down what should be the probability of any particular sequence, you can just write down the procedure that it describes. This is the procedure. The procedure doesn't explicitly tell you what the distribution is, it just samples it.

You've built a procedure for flipping a coin. And if you do it many, many, many different times, what you'll get is the geometric distribution. This is will approach the geometric distribution.

I can probably also do density, and then it'll show you it like that. So that's what I was talking about before with, like, trying to wrap your head around something like the equivalence between a probability distribution that you can write down in math or as an analytical expression and writing down the equivalent procedure for generating that probability distribution.

Let's move on to something a little bit more interesting like Gaussian sampling. If you're not with us, you can look at it in 3.5, Gaussian Samples. What I've done here is, basically, I'm defining a particular center.

Let's walk through this for a second. I'm defining a two-dimensional Gaussian. What it does is, it takes a particular center. A center is just an x-y point. And it does, you know, Gaussian around the first one.

I'm trying to define a two-dimensional Gaussian. The way I do it is, I take a point around-- a one-dimensional Gaussian around this point. And I take a one-dimensional Gaussian around the second point. And then I just draw it.

So in this particular case, I'm going to define my Gaussian center as 3, 2. OK, I'm going to take it x equals 3, y equals 2. And I want to sample a Gaussian around 3, 2. So I'm going to sample of Gaussian around 3 and a Gaussian around 2. And I'm going to give you that back.

And if I repeat this 1,000 times, then-- and I scatter it, I'll end up with a plot that looks a bit like this. And you can see on the x-axis, this is 3. And this is 2. And it's basically a Gaussian with sampling points from around this thing, another forward procedure that I can sample.

OK, is everyone more or less on board with this? Let's take two seconds to read this again. A basic procedure in Church is Gaussian. What I do is I basically-- I try to call Gaussian on some number.

Gaussian takes in two arguments. Gaussian takes in a mean and a variance. In particular, I'm going to take a Gaussian. And its mean is going to be the first argument of center. Its variance it's going to be 1.

I'm going to take a Gaussian sampled from the second argument, the y, and a variance of 1. And then I'm going to just give you back to that point. So this is a procedure that takes in a center point. And each time you sample it, it will give you a sample from around the mean 3, 2.

And if I run that-- so now I've defined a particular center. You know, I've defined it 3, 2. I could have done many other different things. And I repeat that 100 times. I've basically drawn a sample from something around 3, 2.

This can quickly get more interesting if you do something like a mixture of Gaussians. So a Gaussian mixture model is usually just saying, OK, I have some particular space. And I'm trying to figure out how many Gaussians are in this scene, so let's write down the forward model for that thing.

What's the forward model for a mixture model? The forward model saying, I'm going to draw out some number of Gaussians. I don't know how many. And I don't necessarily know what their center point is, right? And from each one of these, I'm going to draw some number of samples.

Does everyone understand, more or less, that description that I just gave? We're going to write it out now. But the point is, the generative model in your head for a mixture of Gaussians should be, there are some number of Gaussians. I don't know what it is. Each one of them is

centered on some point. I don't know what it is.

Let's say I know the variance just for simplicity, but I could obviously put a prior on that. And then I just sample from that. And I'll get some distribution. And then you could use-- we'll later on see, once you write down that forward model, it's pretty simple to then just invert it and say, OK, I see some number of points. How many Gaussians are there actually?

But let's write down the forward model. So I have already done this ahead of time. And I'll do it here. So what I've done here, minus the typo, thanks, is to say something like, I want a sample of Gaussian center where I don't know where it is, but I'm going to say that it's in this two-dimensional space between 0 and 10, a box that's 10 wide and 10 tall.

So for each new Gaussian, I don't know where its center is, but I'm assuming it's somewhere in this box that we're looking at. And the way I do that is, I say, OK, I define some sort of procedure. Each time you evaluate this procedure, what it's going to give you back is a pair, where the first thing in the pair is a uniform between 0 and 10, the second thing in the pair is a uniform between 0 and 10. If all you were to do are to sample Gaussian center, you would get back some number uniformly-distributed in the 10 box, where the first one is, let's say, x, and the second one is y.

And the next thing I do is, let's say I want to define some number of Gaussians and I don't know how many there are. Let's say, for example, that I want to put some sort of ignorance prior on Gaussians between-- there might be one, there might be two, there might be 10. Let's say I stop it at 10 or something like that.

So in this case, I just say, sample the number of Gaussians from something like random integer 10, since this goes to 0, and you don't want 0, I'm just adding the number 1 here. But what I also could have done, and I think I was going to do this is an exercise, but since we want to get to physics, and psychology, and some more interesting stuff, what I could have done here is define number of Gaussians-- suppose I wanted to put a prior on there being potentially an infinite number of Gaussian, what would I do?

AUDIENCE: Dirichlet.

TOMER ULLMAN: A Dirichlet, right? Or what else can I do that we've already learned? We could do the geometric, right? We just defined the geometric a second ago. The geometric gives us a probability on numbers basically going from 0 to infinity.

And it dies off very quickly, so this gives us sort of a natural prior of some sort to say, I think that there are some number of Gaussians here. I don't know what it is. I'm pretty sure it dies off. Like, I don't think 100 is as equally likely as 10. I don't think 10 is as equally likely as 1.

So I could have said, define number of Gaussians, just draw from geometric. And then I would have gotten some number, potentially infinite. You've just defined an infinite Gaussian mixture model. And then I draw some number of centers by basically repeating this procedure.

I sample the Gaussians. And then I scatter the points. Let's see, and then you can look at the points. And this is a fun game to play. It's basically recapturing a bit of what Josh said before, which is to say, how many Gaussians do you think are in this image?

And you can sort of play that with yourself to get a sense of it. You know, you've defined some procedure. You don't know how many Gaussians you actually created. You don't know exactly where they are, but you can run it forward.

And you can look at it and say, well, here I think it's pretty obvious. I think there's sort of a Gaussian here, maybe a Gaussian here. So I guess the number here is 2, but here it's a bit less obvious. And again, you can play with this.

So those of you who've written this down, and assuming you've done either a Dirichlet or a geometric distribution what you've basically done is written down the forward model for an infinite Gaussian mixture model. And you did it in, more or less, five lines of code. Yeah?

AUDIENCE: What is the fold there?

TOMER ULLMAN: Where do you see fold here?

AUDIENCE: Visualize scatter fold append

TOMER ULLMAN: Ah, yes, so fold is another high-level procedure. It's not terribly important for the purposes of this tutorial, but what it does is, it basically takes in a function. It takes in a list of stuff. And it basically applies it to the first argument. Then it takes it and applies it to whatever the result was plus the next item--

AUDIENCE: Plus?

TOMER ULLMAN: --in the list. Well, not exactly plus--

AUDIENCE: In addition?

TOMER ULLMAN: --but, yes, in addition, so you can have a fold which has, for example, two arguments. And what it does is it multiplies. So then you would take a list. And you would basically do-- or rather, what is sum. what some is basically is a fold of plus over a list, because it takes the first number, sums it up with the second one, takes that result, sums it up with a third one--

AUDIENCE: [INAUDIBLE]

TOMER ULLMAN: Fold needs three arguments. Fold needs a particular-- well, it needs the function that you're going to apply. It needs a starting point to start from. And it needs a lot that it's going to work on, again, not terribly important for--

AUDIENCE: So why do this?

TOMER ULLMAN: So in this particular case, what I'm trying to do in the background is, I'm going to get a lot of Gaussians. I don't know how many. I'm going to get basically a list of lists. It could be one. It could be three. It could be 10.

Each one of them is going to define some number of points. And I just want to scatter them. But scatter works by taking in one list, so it's basically just a way of collapsing. Say I have three, or 10, I don't know how many. I'm trying to collapse some number of lists into a single list.

We've defined some number of Gaussians. This is a London Blitz example. Josh was talking about this a little bit. Those of you who want to, sort of, jump back in again, you can go to 3.5.2 in the student document. You can copy and whatever is under that and paste it. And let's talk about that example for a second. What this thing is doing is, it's sort of Josh's example-- do you remember his example of, we have some sort of grid.

And we're trying to say, is there a suspicious cluster somewhere, a disease cluster? We have some dots. And we're trying to figure out is there something going on here? You know, there's sort of a faulty, I don't know, whatever, asbestos or something like that. And I want to figure that out.

So what you're going to get is sort of a 2D map. You're going to get some dots from that map. And you're trying to figure out-- your hypothesis is either this is sort of randomly-distributed, it's a uniform, or there's some sort of center here. So how do we write down the forward model for

something like that?

We would write down either-- the particular example, I'm doing here is another example that Tom Griffiths did, which is, during the Blitz, during the London bombing-- this is actually a very old example of finding patterns. Some of the British, the people of London, were convinced that there were spies in London that were telling the Germans where to bomb during the Blitz.

And the way that they reasoned this is, they looked at the pattern of bombings. And they said, there's no way that this is random. They just looked at, like, dots on a map. And to them, it looked a bit like Gaussians, or things like that. They were working from, sort of, few examples.

When you look at, there's, sort of, these nice web-- "nice," I don't know if it's nice-- but there's these websites that show you the entire Blitz from when it started to when it ended. And it's basically a random distribution. If you run statistical tests on it, it's no different from a random distribution.

How would you run such a test on it? What you would do, for example, is you would write a forward model that says it's either random, uniform, or it's not. Now, tell me which one is more likely. And that's what people have, kind of, done. That's a nice data set to play around with.

The way that we've written it over here is to say, look, we have two options. Either it's a uniform bombing or it's some targeted bombing. The uniform bombing is basically going to give us just some point between 0-- between this box of 0 to 10, just this thing that we were talking about before.

It's going to sample uniformly from this box. The targeted bombing is going to sample some Gaussians, just like we defined before. You don't know how many. You don't know where the center is. And it's going to then sample from those Gaussians. And it's going to give you back some sort of scatter.

And you're basically going to say, OK, I don't know if it's random, uniform, or if there's some targeted bombing going on here, so I'm going to place, basically, some inference. I'm going to flip a coin. If it comes up heads, I'm going to do uniform bombing. If it comes up tails, I'm going to do targeted bombing.

And then you could look at something like this. And you can say, well, I don't know. That's kind of odd. I mean, it doesn't exactly look like a uniform bombing. There's all this missing empty

space over here, right? It doesn't exactly look like one particular target.

And again, you can sort of play with this. And we'll get into the inference about how to invert this thing. But just as a forward model, you can play with this, run it forward, and try to see if you can guess.