NaturalLanguageProcessing-Lecture16

Instructor (Christopher Manning): Okay, hi, everyone. So today I'm going to go through in some detail some of the ideas that I started on last week of how you can actually go about building a compositional semantic representation to provide some kind of handle on how you can start with meanings of words and some tactic structure, and actually put together a meaning for a complex sentence. And so most of the work that's gonna be talked about here is how can we have enough mathematical machinery and essentially what we're gonna using is lambda calculus to glue things together, and first order star predicate logic is the output form, so that you can actually have a complex meaning being built compositionally out of the parts. And so that'll be today, and then for the last two lectures next week, on Monday we'll then talk about well, how can we learn about and represent the meanings of words, so that'll be the lexical semantics part. And then the last lecture, I'll talk about some applications like question answering and textural inference. [Inaudible] say a sentence about quizzes. So again, have precisely one more quiz and sort of shifting the schedule between having had the holiday on this Monday and lectures ending next Wednesday, our goal is to hand out the final quiz by the end of this week, Friday, and then for it to be due on the Wednesday midnight, the day of the last lecture. Okay. Okay, so first of all, for the first couple of slides, essentially I'm gonna be kept in a slightly more formal way with the kind of material that we talked about last time, and then go on and talk about how that we can get further than this. So this sort of bout looks familiar from last time. And so we have two people, "Kathy" and "Fong," we have a verb "respects" which we're representing with lambda calculus taking two arguments, and inference of the verb "run." And then here is our grammar, and so I'm stop writing – stopping writing definite clause grammars, so our grammars are just effectively being written before the colon and just like a contextually grammar before, and after the colon is writing how meanings are combined to represent the whole sentence.

And the form of what happens with the semantics in our grammar rules is always gonna be incredibly simple, so what we're doing here is we're saying our composition rule is, "Take the meaning of both halves and do function application." Do function application [inaudible] only one thing a verb phrase rewrites as a verb, and so you just pass the meaning up. So essentially, we're always going to be either just doing function application or passing the meaning up. And so the action then comes into well, how can we have lexical representations of meanings that'll allow the right kind of combination? Okay. So having this is then enough that we can do the same kind of example that I showed last time, so we have, "Kathy respects Fong." We write down the meanings of the words next to their base categories, we – and then build a verb phrase where we do function application lambda calculus just like this. So we've got the meaning of "respects," the meaning of "Fong." When we do function application we say, "Okay, "Fong" substitutes in for the value of y, and therefore the return value is lambda x respects x Fong." And then we do the same again at the top node. We say, "Okay, here's the meaning of the verb phrase." We apply it to the argument of the subject "Kathy," and so we get out, "Kathy respects Fong." And so note in particular we kind of do that swapping here, so that internally the verb is taking its object first. So the outside

argument here is representing the object argument, not the subject argument, and then it's kinda swapped around inside. So we do get out the right meaning of, "Kathy respects Fong," and not the other way around that Fong respects Kathy. Okay, so the general scenario that I've kinda used as my main motivating example at the moment is that essentially, what we're going to be doing is building a database interface. There are other things that you could imagine. You could imagine that these are meant to be commands that your pet robot understands or something like that, but as a kind of a concrete way to have a semantics that we can map onto, I'm going to assume that we've got databases. Now, I'm not actually going to really spend any time on talking precisely about the mapping onto SQL, I'm just gonna to sort of show random motivate – show bits of SQL that you could imagine mapping it onto and take that as motivational. So I'm really going to show the mapping for first order logic, but for those of you who've seen databases, I mean following the kind of data along tradition of databases, it should be fairly straightforward to see how that if we can represent things first order logic, we can then do a further tetonistic transformation into an SQL query.

Okay, so in particular what I'm going to assume essentially is that we've got database tables that represent our predicates of interest, so in that previous example we had a respectralation between people. And so when we have a respectralation in our predicate logic, I'm going to assume that we have a respect table, and the respect table is going to have two columns. It's going to have the IDs of people who respect other people, the respecter column, and the respected column. So that if we have Kathy and Fong as our two IDs for people in the database, we could then – if we take, "Kathy respects Fong," as kind of a new information statement, we can that as, "Okay, what we wanna do is insert into our respects table a pair of respecter/respected, where Kathy is the respecter and Fong is the respected." Alternatively, in PEP's slightly more interestingly we could also look at the goal of then answering the question, so we all want to be able to answer the question of, "Does Kathy respect Fong?" And so then we'll be wanting to do a query of the database and work out whether it holds a particular row. So the little – this point we use the teeta little SQL trick where we say, "Select "yes" from respects where respects dot respecter equals k, and respects dot respecter equals f." So we're wanting to ask "yes/no" questions like, "Does Kathy respect Fong?" And we kind of fake that up in SQL by saying, "So select "yes." And so then if this relation is in the database SQL will return "yes," and if it isn't in the database SQL will return "no rows returned," and we interpret "no rows returned" as just meaning "no." Okay. Okay, so that'll be our goal to be able to build that kind of interface. And one other note I should've said – I mean just like last – I mean in the examples I show there, and the semantics I make, I'm assuming that for every predicate in the semantic form, we exactly have a relation that in the database for this database table that exactly matches that relation. That kind of makes it convenient and easy. I mean in practice that may not be true, and the example that you could think of there is equivalent to the example from last time with chat AD, so that chat AD actually only has one database table for countries. It has all the information about countries, but what it has is predicates, which you could think of as equivalent to database views where it has a capital predicate, which is taking just then two columns out of the countries table of country name and capital name. And so I'm assuming in the same sort of way that I've got database views where I – that map precisely onto the predicates of interest. Okay, so

let me go through a little bit more formally the kind of representational base that we're gonna use for this inference. So the model I'm gonna use here is going to be using the higher order lambda calculus, so the higher order is meaning that you can have variables that aren't only individuals, but the variables themselves will be functions like last time. And in particular the version of things that I'm going to present is then assuming that we've got a type lambda calculus. So traditionally when you're presented a lot of things in either the lambda calculus world or the logic world, things are completely untyped. You just have variables x and y, and do things with them, but there's absolutely no reason that that has to be the case. And just like most people have decided that programming languages tend to work better when you have types on variables, you can also put types on variables into logical systems.

And I think there are good reasons of both convenience and foundationally that once you're working with higher order logics, I – it can be very convenient to have type logics, and I'm not gonna go off on – about that for a long time, but we're gonna assume that all of our variables have types. So we're going to have Booleans, zeron 1, we're going to have individuals, who are people and things in our model. And then we're going to have functional types where we can talk about properties of individuals. So the basic property is something that takes an individual and says whether something is true or false about them, and then we have more interesting ones like binary relations, or ternary relations. So in terms of the kind of model we have in mind, this picture that I put on the blackboard is the kind of picture of our model. So we have an oval world, which is the outer circle, and inside the world we have various individuals, which I've denoted with x's, and some of them have names, and here's Kathy and here's Fong, and here's some other individuals in our world. And particular individuals can be objects; they don't have to be people. Okay, so then if we have properties of an individual, properties can be represented as subsets effectively. The subset where that the property is true, so this is the car property, and the things inside of the car, so A and B are cars, and the other six things – the other four things in my world aren't cars. Similarly, this thing here is in Palo Alto, and that thing there is in Palo Alto, so I've got this [inaudible] in Palo Alto property. Red things [inaudible], so those are my various properties I can have in my models as effectively my model of the world that I can do inference on – with in a model theoretic way. It's a little bit more complex when you then have predicates that have more than one argument because then I can't draw the same convenient circles in the same way. But you can think of a two-place predicate like "likes," or "respects" is like this. So it takes a first argument, which I've only partially represented it here, so you can read it. So the first argument should be able to be any individual, but I've only showed two of them. And what it then does is returns another function, so the function it returns is then a property. So the property is then saying whether the "likes Kathy" property holds of these – I'm sorry – you know what? I drew this the wrong way around for some – I was wanting – let me just quickly change this. This whole point was it was meant to take the objects first, and then take the people second, is what I just illustrated with my previous example. Let me just quickly change this. A, B, C, [inaudible]. Okay, sorry. Okay. This is one [inaudible]. So the first argument it takes is the object, so if you have the predicate, so if you have "likes," and its object is A, you're then what's returned is a property, a function that takes a single individual argument and returns a Boolean.

So this property here is the set of people that like object A. And so that set there is "Kathy" as I represented it here. And here's the property of individuals that like object B, and the set that for which that is true is "Kathy and Fong." Okay, and you can apply in that same general way if you want to have functions that take three arguments that you're building these higher order functions, or return lower order functions until they ground it in either individuals or Booleans. Okay. Okay. Something to note that I will use sometimes though tends to confuse people, so I won't use all of the time, once you have types in your – so the conventional way of writing untyped lambda calculus is things like this, lambda y, lambda x, respect xy where here shows that this is something that takes two arguments. Now once you have a typed lambda calculus, that all functions inherently have a type. So it's just like java that if you have a method that it has a particular type that is its signature, and you can know that a function has a particular type. So you kind of actually don't have to write all these lambda arguments, you can just write "respect" because it's just a fact about respect that respect is a function that takes two individual arguments and returns a Boolean, that that's just the nature of the function. And so because of that we can then write things in the shorter form, which I will just show. So this is referred to as the long form where you're showing lambda abstractions of all the arguments and then how they're being applied, but the short form is just to write the function without excessive lambdas.

And so I mentioned very briefly and I'll come back to in a minute, the idea of having quantifiers. So that when you have a quantifier like "every" – it's a relationship between two properties, so this is, "Every kid runs." And so you're evaluating the property of being a kid and the property of things that run, and what you're asking is whether for everything that this property is told, whether this property also holds. So this is writing out a generalized quantifier in the form of the long form. But actually, we can just write it out in the short form like this because "kid" is a property of functions of individuals to Booleans. "Run" is a function from individuals to Boolean and the generalized term in "every" is then a function that takes two properties and returns a Boolean as to whether it holds. And so we can write it out in the shorter form. And so though these kind of general rules for lambda calculus, which I'm not gonna spend a lot of detailed time on the fine points of, but the general things that are for lambda calculus that you can do lambda extractions, so you can always pull out lambdas if you want to like this if it's useful to. You can do function application, which gets called beta reduction, so you can learn from this that Alonzo Church was really fond of Greek letters and named absolutely everything with a different Greek letter just to keep things complex. In those eta reductions, so eta reduction just let's you do the reverse of lambda extraction, and then alpha reduction is used for renaming of variables, so you don't get chance collisions between variable names. Okay. And so the central idea is that we can effectively come up with good semantic types that correspond to syntactic categories, or at least common pieces of syntactic categories. We'll see that there are various complications in various places. So in general when you have nouns or verb phrases that they will be properties. So something like being a car is a property, so that there's a certain set of things that are cars. But also a verb phrase, so the "liking A" is also a property because then you're picking out the set of subjects for which liking A is true. The simplest case of noun phrases are individuals, so when we have things like the, "Kathy respects Fong,"

example, Kathy and Fong we both represent as individuals. That kind of very quickly stops working when you want to deal with more complex cases like, “Every car is red,” and so I’ll come back to how people deal with that in a moment. What about if you want to sort of be building a meaning of “red car?” Well, the prop – it seems fine to say that red is a property at the end of the day, the set of things that are red things, but we have this kind of idea that we want to be able to build up the meanings of words by doing function application. And so if we simply have the meaning of car is a property and the meaning of red is a property, well, we kind of can’t combine them by function application any more. And at this point there’s actually a choice as to how you go about designing your system. One way you could go about designing your system is to say, “Yep, G’s function application isn’t going to work,” so what instead we’re going to do is say, “Every time we write a syntax rule down, we’re then going to write special rules as to how to generate the meaning according to that syntax rule.” So for example I could say that car’s a property and red’s a property. And if I have the syntax rule “noun phrase goes to adjective noun,” what I say is the way you translate that is if the adjective meaning is the property p, and the noun meaning is the property q, you translate that as lambda z, p of z, nq of z. So I’m kind of building up in my syntax rule a translation rule. And that’s referred to in logic as using a syncategoremic translation where you’re introducing extra stuff in your syntax rule.

The general trend that people have used when doing computational semantics has been to really avoid doing that because you get a lot more simplicity and power if you can just have automatic translation by function application. But the cost of that is you then have to encode the extra complexity down in the lexicon, and so that’s what’s then shown on this slide. So if we just want to be able to make a meaning for “red car” by function application, then what we have to do is say that one word is a function that takes the other one as an argument. And so we say it’s the adjective that does that. So we say that well, red takes the meaning of car as an argument and returns something. Well, what does that mean that red should be? Well, the meaning of car as a property, it’s something from individuals to Boolean, and when we have a red car, that’s also a property, so this set here is the set of red cars, and that’s also a property. And so therefore, the meaning of red should be something that takes a property and returns a property, and so its type is what you see here. It takes a property individual to Boolean, and returns another property. Okay, and once you start going down that route, you just kind of keep on heading down that route and you get into higher and higher order functions. So if you want to have a adverb like “very” in “very happy,” you say, “Well, what does “very happy” do?” It takes an adjective like “happy” and it returns something that has a meaning that’s kind of like an adjective, right? “Very happy” is sort of like an adjective meaning apart from it’s describing a more extreme adjective meaning. Okay, so if that means the type that we’re going to give to adverbs that modify adjectives at least is it takes as its argument the adjective’s type, and it returns something that is of an adjective type. So it’s intebol to intebol, to intebol to intebol, which is just about incomprehensible, but in practice, these things are fairly straightforward. If you kind of just think of them in terms of higher-level notions. So you – once you think that these are properties, and adjective is something that maps from one property to another, and this is something that maps from adjective meaning to another, that then things sort of fall into place in sort of straightforward

enough way. Okay, so here is my grammar fragment that I'm going to start off doing some examples for the grammar fragment, and I'll add to it a bit more so I can answer questions. So this is a very straightforward PCFG grammar – I should say CFG grammar – I mean the only way it's different from the kind of ones that we've seen in the Pentree Bank, is if you look at the noun phrase structure, I'm actually kind of producing a binary noun phrase structure rather than flat noun phrase structure. If you want to build up semantics of noun phrases, it really helps to have a binary noun phrase structure, different reason than making parsing going fast. So a noun phrase goes to a determiner and then an N-Bar. An N-Bar can have any number adjectival PP modifiers and then goes to a noun.

But apart from that, note that the semantics is in all cases trivial, so the semantics of our rules is simply function application that you have something be the head, and it's going to be the head [inaudible], and if there are other things, they just become arguments of it that I'm denoting there. And that means that all the interesting stuff then happens in the lexicon. And so this shows some of the lexicon, and I'll go through some of this with examples, but just go through a couple of things at the start. So we have "Kathy," "Fong," and "Palo Alto," and so they're things that are individuals. I guess when I did this I didn't actually put "Palo Alto" in as an individual, but we could have somewhere here "Palo Alto" also represented as an individual. Okay, then you have things like "car" where "car" is a property, so it's something from individuals to Booleans just like here. For "red," red is a little bit more complex since what I've represented here as a property is actually then "red prime." So "red prime" is the property of things that are red. So the red here is then this red that's of the intebol to intebol tied, where what it does is it takes a property "p," so this is the higher order part of our lambda calculus. And inside its lexical inferri, it does the meaning combination. So it says, "Okay, if you're gonna make up something like "red car," I take as an argument the property and then I build as my lexical inferri to return the property in which p is true of it, and the red property is true of it. And so we kind of do the clever bits of meaning combination down in the lexical entries. Okay. And then maybe more straightforwardly down at the bottom we then have the verbs, which are much like before. So "run" is taking individual and returning a Boolean. "Respects" is taking two individuals and then saying whether it's true or not, similarly for "likes." So "likes" is just like here, and an example like "runs" is effectively another property here, so there's the things that run, and so maybe this runs or that car doesn't run. Okay.

Okay, so these are the ones in large print, when I get on to the more complex sentences, the print gets even smaller. And I think to some extent they'll sort of work through some of the details of this, you have to kind of look more carefully and see that it all makes sense more than I can do in lecture, but nevertheless I want to try and point out some of our high level points of what's going on. So what we wanna do is make a meaning for "red car Palo Alto." And I mean first of all, to think about it conceptually, well, what should the meaning of "red car in Palo Alto" be? The meaning of "red car in Palo Alto," well, that seem like that should still be a property. It should still be something that picks out a set of individuals that satisfy the property "red car in Palo Alto." I mean in particular, in terms of my model that I have right here, what I'm wanting it to do is pick out precisely this set here as this is the "red car in Palo Alto" set of individuals. So the

end result of the semantics that we make should be that we're getting out a property. So how do we actually get that all out? This is the bit that I've talked about already. So "car" is a property from individuals to Booleans, so "red" has this more complex meaning where it takes a property and returns a bigger property. So we can combine in here "car" as p, and what we get back is lambda x, car x and red prime x. So that's already created a property, which corresponds to the things that are cars and red, so that's that sort of bigger property up here. Okay, and then we wanna do combining in Palo Alto. And so in – at that kind of scary lexical entry I've written next to it, which I didn't stop to dwell on in the time. But while in principle you can kind of reason through in first principles what kind of types you have to put on words to be able to get things to work and then essentially to build those types. So if you think about "in Palo Alto" – "in Palo Alto" it should itself be a property, right. That that seems like it's something that can be true or false of individuals there. At least at a certain point it time, they're either in Palo Alto or they're not in Palo Alto. So here I have this property of things that are inside Palo Alto. Okay, so that means that when we build this pp up here of in Palo Alto, its meaning should be a property. Well – should complexify that once more. The idea of "in Palo Alto" should be a property, but just like an adjective we're then going to want to have the pp modifying the meaning of "red car" just like an adjective, so I'll explain that in a moment. Okay, so we have – we wanna build up a property meaning, but what we actually have is an individual "Palo Alto," and we have the word "in." So that suggests that we want to have something that takes an individual and is returning something. So the prepositions meaning starts off by taking an individual y.

And so well, what's it gonna return? It's gonna return for "in Palo Alto" something that's then an adjective meaning, something that will take a property and build a more complex property where it satisfies the previous property and "in Palo Alto." And so that means at the very end of the day, that what you get for some – a word like "in" is it's taking an individual, which is the object of the preposition, and then it's returning one of these adjective meanings. You can have "in" as having that kind of type. Okay, so it takes "Palo Alto" as an argument, so "Palo Alto" becomes y and substitutes into "in" to give this meaning, okay. And so now this is something that works just like an adjective, that it'll take a property as an argument and return a more complex property. And so here it takes as it's argument things that are "red cars," and so all of this becomes lambda p, and so you substitute "in," so then you get lambda x, car of x, and red prime of x, and in prime Palo Alto x, which is then the meaning of the whole thing. Does that sort of make sense? I know there's a lot of notation stuff, but the basic idea is we can take these things of various functional types, we can write them so that they take arguments of the appropriate functional type, and then they return an argument of the appropriate functional type. And so in this sort of way if we model all noun phrase modifiers as properties, that as things that map from properties of properties, we can then have successive noun phrase modifiers kind of combine together and the resulting meaning would be still a property. So by extending this we can have sort of the large leather green couch in my bedroom, which my uncle sold me, and we can kind of have each of those modifiers be something that maps from a property to another property, and we can combine it together and make up the meaning of the whole. It's worth noting finally, the way I've wrote the grammar – if I go back a few slides – but when you're at this N-Bar

level – at the N-Bar level you can just take modifiers. So you could take adjectives on the left, or you could take prepositional phrases on the right, and if I have other things like relative clauses, you could take them on the right as well, so you can just take any number of modifiers. And so that actually means that the grammar has a syntactic ambiguity where you can take the modifiers in either order. So here I built "red car" first, and here I built "car in Palo Alto" first. And interestingly, that's then what people call a spurious ambiguity, whether things that are licensed by the syntax that don't actually make any difference to the meaning that's generated for the sentence. So regardless of which one I combine with first, I get the same meaning of lambda x, car x, in Palo Alto x, and red x. Okay.

And precisely the reason why I get an ambiguity that's spurious there is that the meanings of the different modifiers I'm putting on are actually just turning into extra conjoined terms. So things that are just turn into extra-conjoined terms are referred to as intersective modifiers in linguistics, and so that's exactly what I was doing here. I was having the set of cars, I was intersecting it with the set of red things, and then I'm intersecting that with the set of in Palo Alto things, and I'm coming down to this intersection and that gives me intersective semantics. So if you're in that world – and so what's of the things that we think of as modifiers have intersective semantics? So "green leather couch." You have the set of couches, you have the set of green things, you have the set of leather things, you can intersect them, and you have the set of green leather couches. And providing we're in that world of intersective semantics, it's very easy to see how to do this with database queries because effectively you can have a table for each one of these properties, and then you can just join the tables and find out the individuals, the database IDs for – which are in all three of these tables. So that's kind of the easy case and the case that we can actually practically work with most of the time. And sort of important to note though, that there actually are – and this is something that's talked about quite a bit in linguistics – there are nonintersective modifiers where you get a meaning from the whole thing, which you can't kind of represent as an intersection of terms. So I mean the most famous example is of words like "alleged." So if you say that someone is an alleged murderer, that doesn't mean that they satisfy the property of being a murderer and that they satisfy the property of being alleged. It means that they satisfy the property of being an alleged murderer where alleged murderers somehow a property meaning that you can put together out of the meanings of the parts. And so somehow if you wanna be able to deal with alleged murderer, you wanna be able to deal with that, but it's kind of not so straightforward as to how to do that because effectively you want to be sort of working out a sort of a new functional form out of an old functional form. I mean here's the other example of this that I had here, which isn't quite so convincing, but I think illustrates the same point. So this example here is "overpriced house in Palo Alto." And so although it's not quite as dramatic as "alleged," I mean I think it's reasonable to say that overpriced isn't a term that has absolute reference.

So you can have one semantics for an overpriced house and it's in Palo Alto, and that's a kind of a global scale of what's an overpriced house, so using these semantics, well, every house in Palo Alto qualifies as an overpriced house by global standards. But if you make the modification the other way, you then have the set of houses that are in Palo

Alto, and then you're saying overpriced out of houses that are in Palo Alto, and so that seems to have a different semantics of houses that are overpriced even relative to the standards of Palo Alto. Okay, so you can get tricky things with nonintersective adjectives, but I'm not gonna deal with that further today, and I'm just gonna stick with the intersective adjectives. Okay. So all's good until now, but unfortunately natural languages just get a bit more complex that what we've done so far. Okay. So I don't know if this ever troubled any of you guys when you did a baby logic class, but when you think about how people translate English into a baby logic class, something fundamentally weird happens, right? So if you wanna translate "Kathy runs," you translate it into "runs Kathy.' That's easy. But if someone then tells you to then translate into predicate logic "no kid runs," well then, you kind of have to know to do this really funny thing where you write down "not there exist x, kid x, and run x." So run – here the noun phrase subject was an argument of the verb, just like it seems to be in the syntax of the sentence, "Kathy runs." Kathy is an argument of running. Syntactically this example doesn't seem any different. "No kid" is now the subject of "runs." Runs, a predicate, has taken a subject, that's his argument, but the translation is kinda completely different. We found this really funny thing where we've wrapped some weird logical structure around the outside of the whole thing, and "runs" is buried down here somewhere. So that looks kind of awkward, and so the question is how can we produce those kind of translations in our system. Well, here's another example of how things go wrong. So we have, "Nothing is better than a life of peace and prosperity. A cold egg-salad sandwich is better than nothing," and if you don't look too closely it seems that you should be able to deduce from that that a cold egg-salad sandwich is better than a life of peace and prosperity.

Why is that? Well, it seems like "is better than" should be a transitive relationship, right. That if you have that whatever – a Blackberry Pearl is better than a Motorola Razor, and an iPhone is better than a Blackberry Pearl, you should be able to conclude that an iPhone is better than a Motorola Razor, right. "Is better than" is a transitive relation. And so that's what I'm attempting to do here, I'm trying to do inference using the fact that "is better than" is the transitive relation. But with sentences like this, it just goes totally wrong, and well, why does it go wrong? Well, the reason is again, "nothing" is a quantifier. It's just like this "no kid runs," here when you have "no kid," that as soon as you have things that are quantifiers, it kind of just doesn't work if you're treating them as arguments to relation, and you seem to do something special and different. Okay, so somehow we need to be able to model in our system how we represent the meaning of these quantifiers and deal with them. And so well, how can we do that? Well, we've done pretty well with things that are properties. We have "red car" and "Palo Alto," and things like that. So we could build everything about noun phrases as complex properties, except when they have a determiner out the front of them like "the," or "every," or "no," and things like that. Once we see that, it seems like we have to do something very special. So what are we gonna do from the meaning of "the red car in Palo Alto," or "every red car in Palo Alto?" Well, first of all, mention what we gonna do with "the." What – for "the" what the easiest thing to do is say, well, we want to then map down to an individual. Just like we had Kathy and Fong as individuals. "The red car in Palo Alto," that should be an individual. It should be this guy right here, that that's the meaning of the "the red car in Palo Alto." An so a possible semantics for "the," and this is actually precisely the

semantics for “the” that was proposed by Bertram Russell, is to say that “the,” the meaning of “the” can be represented as the iota function. So iota is the semantics of “the,” which was back in the lexicon if you look back ten slides. The meaning of “the” is that this is a partial function. So if there’s precisely one thing in the model for which p of x is true, then the function returns that thing. So if I ask for “the red car in Palo Alto,” the meaning – what’s returned is the individual b. If there are multiple things for which a certain property is true, so for example if I just say, “The car,” then this is only a partial function, and the return value is undefined. So if the function doesn’t apply, it’s got no return value. So that’s our meaning of “the.” It will return an individual if one is uniquely specified, and if there – one isn’t uniquely specified, you don’t get any value in return.

Now, in – now I mean that’s kind of true. I mean in practice though, it’s not quite that simple. So when I – I mean basically we use “the” when we’re talking about things that we can identify. So we talk about “the president of Stanford University” because there’s only one of them, and so it can be uniquely identified, but we don’t say “the department chair of Stanford University” because that kind of – it sort of feels wrong because it doesn’t identify an individual, and that’s Bertram Russell’s core inside. I mean in practice it’s a little bit more complex than that because in practice people will interpret the meaning of “the” as relative to some situationally defined context, so if you say something like, “Pass me the phone,” you’re not making a claim there’s only one telephone in the world. What you’re making a claim is in this situationally defined context of the room that we’re sitting in, there’s one phone, and that’s the one that you’re meant to pass me. Okay, so that gives us a semantics for “the” nevertheless, so I mean now we could actually say a sentence like – work out a semantics for a sentence like, “Kim likes the red car in Palo Alto,” because the meaning of “the red car in Palo Alto” would just be this individual b, and then we can look across and say, okay, “likes b” returns this function, and if we evaluate it with respect to “Kim,” we get “true” as the answer, and so we can evaluate that kind of sentence. Okay. And so we can then translate that kind of sentence into our SQL database form, so “red car in Palo Alto,” properties where we make a query of a database, and we return the set of objects that satisfy it, just like in a regular database query. So this’ll be select cars dot odge – for properties I’m just assuming that it’s a one-column table where the field that’s called odge. Select cars dot odge from cars locations “red,” so I’ve joined my tables where cars dot odge equals location dot odge, and locations dot place equals Palo Alto, and cars dot odge equals red dot odge. Okay, so then if I want to do “the red car in Palo Alto,” what I’m wanting to say is the return set from that database query is a single row, and I can actually write that in my SQL by being a little bit clever and writing a having clause, and I can say, “Select all of the same stuff, having count equals one.” And so this having clause will precisely implement my Bertram Russell semantics of “the.” If there’s a single thing that satisfies it, I will get returned that one row of the database, and otherwise I don’t get a return. Okay. Well, that’s good up to there. If I could just say, “For the red car in Palo Alto,” well, it turns into this individual, and we know how to evaluate individuals with respect to something like “likes,” but what if I want to say, “Every red car in Palo Alto?” Well, then it seems like we have to do this fundamentally – change things around where with words like “every,” these determiners, they have this quantificational force where they take other things as their arguments, and so that’s the thing that we’re going to represent.

So what we're gonna take as the type of a generalized determiner, a word like "every" and "no," is to say, "This is something that takes one property." So for "every car" it'll take the property of being a car. For "every red car in Palo Alto" it'll take the property of "red car in Palo Alto," and then what it's going to return is something that maps from properties to truth-values.

Okay, and so I can have semantics for generalized determiners in terms of these what I'll call generalized quantifiers that take a single argument. So "some kid" will take two – "some kid runs," you take the two properties as arguments, you can join them together and then the "some" predicate says is there at least one individual for which this property is true? And the "every" predicate says is it the case that this is true for all individuals? Okay. Just what's expressed there. Okay, so now I'm kind of just about in business in the following sense. Okay, here's kind of how I want to do "every student likes the red car." Okay, so I have "car," I build up the bigger property of "red car," and then I combine it with the determiner "the red car," so this is this iota operator here. And when we evaluate the iota operator of "the red car," its value is just an individual. It's the individual b in our model over here. Okay, so because its value as an individual I can apply the predicate "likes" to it, and then I get something that is a property of "liking the red car." In particular the property that I get out is precisely this property here of "liking the red car." Okay. And so then what I want to say is that, "Every student likes the red car." And so well, I'd really be in business at this point if I could do the following trick. Up until now I've treated the verb as the head of the sentence and say, "Do function application." That's how I handled "Kathy likes Fong." But "likes" took two arguments, "Kathy" and "Fong," and it works, but it seems like that's not gonna work for me for "every student." But what if I just changed my mind and said, "Well, let's take "every student" as the head in the semantics, and have it take "likes the red car" as an argument." How would things work out then? Well, "every" is something that takes two properties and then returns a truth-value, so it combines with one property here to give us "every student." And well, look, the meaning of the verb phrase is a property. We've made precisely this property that I put a box around on the board, so "every student" could take that as an argument, and then we'd precisely get the "every" predicate evaluating those two properties. And so it would be saying, "Is this property true of every individual?" And at least for the partial representation I've shown over here, it is true of every individual, and so I'd get x my answer "yes." Okay, that looks really cool apart from the one problem that I played this mucky trick whereas before the verb phrase was the head of the sentence, I now made the subject noun phrase the head of the sentence, and that might then get you worried as to well, kind of what happens after that. Like what if I put a quantifier in object position? What if I said, "Every student likes every car?" Well then, I sort of am in worse trouble then because then if the meaning of "every car" is something that takes a property and returns a truth-value, it sort of seems like I can't combine that with "likes" at all. So I need to sort of build some more technology to be able to deal with quantifiers. So the person who sort of initiated this enterprise was Richard Montague. So Richard Montague was a philosopher in the late 60s and early 1970s. He actually committed suicide, which is a long story I will not digress into for – too much detail at the moment. But based – the central idea of his work was to notice the fact that modern logic had developed all of these rather nice tools that had completely abandoned the goal of actually trying to

provide semantic representations for human languages, whereas the original goal of logic as I mentioned before, was actually to help assess human arguments. The ancient Greeks actually believed that logic was a tool, so that when you listened to George Bush say something on television you could evaluate whether what he was saying was sensible or not.

So Richard Montague had the idea that maybe now that there are all of these good formal logical tools, maybe there actually could be applied vector natural language if we have the right technology for doing it. And so he introduced these kind of techniques of using higher order logics and lambda calculus. And so the idea that Montague came up with to solve the problem of how to incorporate quantifiers into things, was to say, "Well, it just doesn't work to say that noun phrases refer to individuals because we wanna be able to say things like, "Every student likes every car," or, "Every student likes some car," or something like that. And these quantifiers we have to represent differently. So in his model all noun phrases got turned into these higher order things. So any noun phrase meaning got turned into something that took a property and then returned a Boolean. And so the meaning of "Kathy" is then no longer an individual "Kathy." The meaning of it is this thing that takes a property and returns that property evaluated with respect to Kathy. Slightly confusing, but sort of makes sense, but once you have this, if you want to have the semantics of "Kathy runs," or you've got the property of running things, and you stick that in as p, and you're evaluating whether Kathy is true of it. So you've fundamentally changed around one of the functions and arguments. A lot of modern work has sort of seen that as rather ugly and unnecessary, and so has instead gone in this direction that essentially follows Montague, that essentially says you want to allow flexible shifting between types. So Kathy is just an individual, but if you want to for semantic combination, you can certainly turn it into something that takes properties and evaluates whether it's true of that property. We had properties like "man" and "car," and well, it proved handy for "the" to have this iota operator, which could map from a property to an individual that satisfies it as a partial function, which is what those dots are meant to show. And so noun phrases are commonly these days represented by allowing these kind of flexible type shifting operations. Okay. Now this point time is getting short, and I'm gonna skip a little bit, but earlier on we talked about how there are quantifiers scope ambiguities, and I will assure you if you read the handout that goes with this class really carefully, it explains this beautiful way in which you can model quantifier scope ambiguities in terms of doing different amounts of type shifting, but I'm not gonna go through it right now. But believe me, here are the two different representations for "some kid broke every toy." And if you look very carefully as I flip them back and forth, you see in one that the "some" quantifier is on the outside, and in the other the "every" quantifier is on the outside.

Okay, let me just for a couple of minutes sort of show a couple of more examples of how you can then start to do questions like this. Okay, so we want to have – we want to be able to do "yes/no" questions, so we wanna check on a database something like "is Kathy running? What things are true of the world?" And perhaps more interestingly, we might want to do content questions like, "Who likes Kathy?" What is the right semantics for questions is something people still write papers about in philosophy and linguistics. The

model I've got here is that a question is just going to itself be a property. So a question is I say a property like "red" and "in Palo Alto," that the meaning of the question is just that property, and that's going to be interpreted as what is the set of things for which that property is true, and that's the answer to the question. Okay. So is how I then extend my grammar to deal with questions. And there's one kind of tricky bit here, which is I use this idea of "get threading" for questions. The complex thing about questions is that they – in English – is that they have these long distance dependency. So if you have something like, "Who did Bill report that John said Sue likes?" The "who" at the front is actually the object of the liking, and somehow you have to connect those two positions together. And there are various ways that it could be done, but a common way in some formal grammars, and the method that I use here, is this idea of "gap threading" where you can introduce a variable as an empty node, and then kind of shift it up through the syntax until you discharge it. So here I will have an empty introduced with a meaning of a variable z, and then I will allow rules to pass up that z until I have my whole sentence, and then I will discharge the z as being an argument for that predicate. I will show an example. And then here are my syntax and semantics for question words. So the question – for a question word like "what" it kind of doesn't have any meaning in this model, it just takes a property and returns that property. So if we have something like "what is running," we have the property of "is running," and what will take it as an argument and just return it does nothing to it. But some of the other "w-h" words do more interesting things, so for "who is running," it takes the same property as an argument, but it makes a more complex property, which says the argument property has to be true, and the thing has to be human. And if I asked, "How many birds are running?" Then I'm taking a property of being a bird, and then I'm going to be taking a property of "is running," and then what I'm going to be returning is the count of the individuals for which that combined property is true.

Okay. So if I take that and put it all together, I can actually now make quite interesting queries and translate them into something that can be evaluated over a database. And again, I'll probably go through this a little quickly, but you can peer closely at the examples and see that it really does work. So here's an easy case, which I'll use to illustrate this gap threading idea. So, "What does Kathy like?" So the idea is we actually have empties in our grammar now, so there's a missing object of like, which is introduced with this variable z, an individual meaning, so this is the – has a meaning of kind of "like z," but we don't actually know who z is yet. So here's "like z" as the meaning, then we get "Kathy likes z," which has the meaning of "like z, Kathy." And then what we can do – we kinda pass up a gap as the splash categories, so this is a verb phrase, which had supplied inside it an empty noun phrase whose meaning was z. And so now here we have a sentence, which has inside it an empty noun phrase whose meaning was z. So at any point we want to we can then discharge our gap thread at empty and say, "Okay, this sentence's meaning is actually "lambda z, likes z, Kathy." So that we've turned back into a property the thing that wasn't actually present, that was missing. So this is kind of like in predicate calculus if you do natural deduction where you assume something and then you later discharge the thing that you assumed. Okay, so now I have a property of "lambda z, likes z, Kathy," and so for, "What does Kathy like," "what" introduces no new semantics of it's own, and so the meaning of the whole thing is just

“lambda z, likes z, Kathy,” i.e. here’s a property and the set of rules that are returned from the database for that property is the answer to the question. So I get select liked from likes, were likes dot liker equals Kathy. Cool. Okay, but we can go on from that and do much more complex examples. So, “Who does Kathy like,” works essentially the same, apart from “who” has this more complex semantic form, so that the result is that you’ve got this property of “Kathy likes the thing and the thing is human.” So now we have select like from likes humans where likes dot likes equals Kathy, and humans dot odge equals likes dot likes, so we’re restricting it to being a human thing. Okay, but we can put in more complex question words like, “Which cars did Kathy like?” So now “which cars” is saying so “which” is taking the property of a car, and it’s taking the property of “Kathy like,” and asking if both of them be true at the top, which gives us this database translation.

And at that point we can just keep on using all the stuff that I’ve tried to present today, and it really does all fit together and work right, so I can ask, “Which cars did every student like?” And so we have the same empty z. We can make a meaning for “every student likes.” So now we have the set of – this property is the set of z such that every student likes z. And then we can combine that with something like “which car,” and make a much more complex meaning. And we can keep on going through examples like this. “How many red cars in Palo Alto does Kathy like?” Or, “Did Kathy see the red car in Palo Alto?” So at this point I’m starting to be in the space that the trees that you have to build are too large to fit in my narrow margin, so you’ll just have to believe me that the proofs work out okay, but they do – or the derivations work out okay. So this is, “How many cars in Palo Alto does Kathy like?” So we kinda make up the “does Kathy like” part here. And then for the, “How many red cars in Palo Alto,” we make this property of red cars in Palo Alto just like before, and then how many takes first this property, and then it takes the “likes Kathy” property, and so it’s returning the count of how many things satisfy this big composite property, and then we could translate into our piece of SQL where we’re then doing a count operation on the things that are satisfied. Now, [inaudible]. So I thought then just for the last few minutes I’d mention just a little bit about some work that’s been happening recently that could actually connect these things together. I mean I think it’s fair to say that for – until about 2005 I guess, 2004, 2005 – that there was really kind of a complete disconnect between work that was using statistical probabilistic machine learning methods and OP, and logically oriented deeper meaning representation work on NOP, that that was something that there’d been a lot of work on in the 70s and 80s, and there was still an occasional little bits of work going on in the 1990s, but effectively the two didn’t have anything to do with each other. That by and large people were doing probabilistic machine learning stuff, were doing more surface work, so they were doing things like building part of speech [inaudible] recognizes and statistical parsers, all the things that you guys have been doing, but really were doing nothing significant in the way of doing full sentence meaning representations. They were doing some meaning tasks, but low-level meaning tasks like doing name density recognition, or extracting particular semantic relations.

But just in the last couple of years, there’s actually then started to be some nice threads of work where people are actually starting to return to how can you learn and use much

deeper semantics while still fitting into using probabilistic and machine learning ideas, and I thought that I'd just mention a couple of things of this sort. So one question is, "Well, could we possibly learn representations of this sort?" And that's something that a couple of groups have been working on recently. So Luke Zettlemoyer and Michael Collins at MIT have been doing a couple of papers on that, and then also Ray Mooney and students at Texas have been doing this sort of work. And I can't really go through the details here, but this is in one very short slide the outline of the Zettlemoyer and Collins approach. So that they assume that they have some stuff at the beginning. They assume that they have an initial lexicon. The initial lexicon is mainly knowing about some function word, so for things like prepositions you have a meaning of what does this preposition mean? They also though assume that they know a few specific – the main specific words, so that they might assume that they know the meaning of "Texas" for example. They then start off with category templates, and so category templates is effectively possible semantic forms words could have. And so that's kind of like I was saying that there are these dominant patters that we have properties and two-argument things and things like that. They have a set of category templates. So they start off with a little bit of stuff, not nothing, but primarily then beyond that, what they assume that they have is pairs of sentences and meaning representations. That may seem like a fair bit, and in some sense it is a fair bit, but I mean a prominent belief of how human language acquisition works is that humans don't actually acquire stuff about sentences until they understand what they mean, right. That basically people are working out the meaning of things from situational inference, so that after the first ten days when mommy is holding the orange juice and saying, "Do you want juice?" That you start to understand what that means, and then you map between the words and semantics. So you have a set of examples like this, and so the goal of the – the goal of their papers is that you want to do combined learning of both lexicon and syntactic structure. So although you have a kind of a bootstrap for the lexicon, most words you don't know what they mean, and you have no information whatsoever of syntactic structure and semantic combination rules. So this isn't like the Pentree Bank where you're given all of the trees, you've been given no trees whatsoever, and no basis for which to put together compositional semantics. And so what they do is that it's built as a kind of an introvert bootstrapping process where you're postulating possible lexical forms, so that's where you're using these category templates, and you're trying to use them in a parser, and the parser is a maxent parsing model. And effectively you're then learning lexical representations for words that seem to work in terms of the fact that they're able to combine together with words that are in your initial lexicon, and work to be producing the final semantics for sentences that you'd like to see, and effectively through each iteration through the bootstrap, what you hope to do is figure out a successful semantic representation for some more words, and then start building up your lexicon so it grows over time. But it's – it is a kind of a cool joint learning task because I mean at the start there's absolutely no parsing model whatsoever, so that they're simultaneously trying to postulate possible lexical entries, and then optimize the parameters of a maxent parser to try to parse things in a particular way as it goes along.

There's then also been work on reasoning with such representations. Now in particular domains, one possibility is just to go with logical reasoning. And I mean I think that can

actually be quite successful in certain domains of business rules, laws, and things like that that you can do logical reasoning. But I think in general it is the case that just like all kind of probabilistic argument in general goes, that knowledge of the world is in general so incomplete and uncertain that we just can't get very far with strict probabilistic inference. In particular a lot of the time things only go through assuming various plausible assumptions. That all the time we're assuming sort of plausible things will happen in the world, and that allows our inferences to go through where they're not actually kind of the climates that'll happen. Most of the time we just assume that a really big earthquake won't happen in the next ten seconds, and so therefore it's reasonable to make assumptions about what the world will be like in ten seconds time. So that then leads us into the work on doing probabilistic models and knowledge representation reasoning. And so that's also been an area that's been actively pursued. I mean it's the case that all the current work on doing probabilistic inference still assumes restricted – something more restricted than first order logic. So first order logic is in general very difficult to deal with because you have this open domain of individuals that perform [inaudible] is in the first order logic, is that you have an infinite space of individuals and you can quantify arbitrarily over them, whereas all of the existing models are probabilistic knowledge representation and inference in some way uses somewhat more restricted model where you can still do certain kinds of quantification that is not quite open [inaudible] like in first order logic. And so one thread of that is definitely Collins work here on probabilistic relational models. Another thread of work that's been going on recently is Pedro Domingo and colleagues have been doing Markov logic, and they differ in various ways. Part of how they differ is the Daphne's model was a basing net directed graphical model whereas this is then an undirected Markov network star model.

Actually just the last week – I know someone who's been doing undergrad honors with me has been trying to apply some of the Markov logic ideas to doing mappings of sentences into Markov logic and then doing inference on that with at least reasonable success on range of sentences, it's not that it's a full coverage system. So I think that this is an area where it's now possible and it's starting to be a lot of exciting ideas where you can kind of start to combine machine learning statistical methods with using these richer representations and then you have to do richer semantic inference. Okay. And that's the end for today.

[End of Audio]

Duration: 75 minutes